Data-Intensive Text Processing with MapReduce

Jimmy Lin and Chris Dyer

Draft of January 27, 2013

This is the post-production manuscript of a book in the Morgan & Claypool Synthesis Lectures on Human Language Technologies (with errata, additions, elaborations, etc.).

Please cite as: Jimmy Lin and Chris Dyer. Data-Intensive Text Processing with MapReduce. Morgan & Claypool Publishers, 2010.

Refer to http://mapreduce.cc/ for more details.

Contents

Contents ii

1 Introduction 1 1.1 Computing in the Clouds . . . . . . . . . . . . . . . . . . . . . 6 1.2 Big Ideas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9 1.3 Why Is This Different? . . . . . . . . . . . . . . . . . . . . . . . 14 1.4 What This Book Is Not . . . . . . . . . . . . . . . . . . . . . . 17

2 MapReduce Basics 19 2.1 Functional Programming Roots . . . . . . . . . . . . . . . . . . 21 2.2 Mappers and Reducers . . . . . . . . . . . . . . . . . . . . . . . 23 2.3 The Execution Framework . . . . . . . . . . . . . . . . . . . . . 27 2.4 Partitioners and Combiners . . . . . . . . . . . . . . . . . . . . 29 2.5 The Distributed File System . . . . . . . . . . . . . . . . . . . . 32 2.6 Hadoop Cluster Architecture . . . . . . . . . . . . . . . . . . . 36 2.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

3 Basic MapReduce Algorithm Design 39 3.1 Local Aggregation . . . . . . . . . . . . . . . . . . . . . . . . . 41 3.2 Pairs and Stripes . . . . . . . . . . . . . . . . . . . . . . . . . . 49 3.3 Computing Relative Frequencies . . . . . . . . . . . . . . . . . 56 3.4 Secondary Sorting . . . . . . . . . . . . . . . . . . . . . . . . . 60 3.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

4 Inverted Indexing for Text Retrieval 65 4.1 Web Crawling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66 4.2 Inverted Indexes . . . . . . . . . . . . . . . . . . . . . . . . . . 68 4.3 Inverted Indexing: Baseline Implementation . . . . . . . . . . . 70 4.4 Inverted Indexing: Revised Implementation . . . . . . . . . . . 72 4.5 Index Compression . . . . . . . . . . . . . . . . . . . . . . . . . 74 4.6 What About Retrieval? . . . . . . . . . . . . . . . . . . . . . . 81 4.7 Summary and Additional Readings . . . . . . . . . . . . . . . . 84

5 Graph Algorithms 87 5.1 Graph Representations . . . . . . . . . . . . . . . . . . . . . . . 89

ii

5.2 Parallel Breadth-First Search . . . . . . . . . . . . . . . . . . . 90 5.3 PageRank . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97 5.4 Issues with Graph Processing . . . . . . . . . . . . . . . . . . . 103 5.5 Summary and Additional Readings . . . . . . . . . . . . . . . . 105

6 Processing Relational Data 107 6.1 Relational Joins . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

7 EM Algorithms for Text Processing 113 7.1 Expectation Maximization . . . . . . . . . . . . . . . . . . . . . 116 7.2 Hidden Markov Models . . . . . . . . . . . . . . . . . . . . . . 123 7.3 EM in MapReduce . . . . . . . . . . . . . . . . . . . . . . . . . 135 7.4 Case Study: Word Alignment for Statistical Machine Translation 138 7.5 EM-Like Algorithms . . . . . . . . . . . . . . . . . . . . . . . . 147 7.6 Summary and Additional Readings . . . . . . . . . . . . . . . . 150

8 Closing Remarks 153 8.1 Limitations of MapReduce . . . . . . . . . . . . . . . . . . . . . 153 8.2 Alternative Computing Paradigms . . . . . . . . . . . . . . . . 155 8.3 MapReduce and Beyond . . . . . . . . . . . . . . . . . . . . . . 157

Bibliography 159

Chapter 1

Introduction

MapReduce [45] is a programming model for expressing distributed computa tions on massive amounts of data and an execution framework for large-scale data processing on clusters of commodity servers. It was originally developed by Google and built on well-known principles in parallel and distributed pro cessing dating back several decades. MapReduce has since enjoyed widespread adoption via an open-source implementation called Hadoop, whose develop ment was led by Yahoo (now an Apache project). Today, a vibrant software ecosystem has sprung up around Hadoop, with significant activity in both in dustry and academia.

This book is about scalable approaches to processing large amounts of text with MapReduce. Given this focus, it makes sense to start with the most basic question: Why? There are many answers to this question, but we focus on two. First, “big data” is a fact of the world, and therefore an issue that real-world systems must grapple with. Second, across a wide range of text processing applications, more data translates into more effective algorithms, and thus it makes sense to take advantage of the plentiful amounts of data that surround us.

Modern information societies are defined by vast repositories of data, both public and private. Therefore, any practical application must be able to scale up to datasets of interest. For many, this means scaling up to the web, or at least a non-trivial fraction thereof. Any organization built around gath ering, analyzing, monitoring, filtering, searching, or organizing web content must tackle large-data problems: “web-scale” processing is practically synony mous with data-intensive processing. This observation applies not only to well established internet companies, but also countless startups and niche players as well. Just think, how many companies do you know that start their pitch with “we’re going to harvest information on the web and. . . ”?

Another strong area of growth is the analysis of user behavior data. Any operator of a moderately successful website can record user activity and in a matter of weeks (or sooner) be drowning in a torrent of log data. In fact, logging user behavior generates so much data that many organizations simply

1

can’t cope with the volume, and either turn the functionality off or throw away data after some time. This represents lost opportunities, as there is a broadly held belief that great value lies in insights derived from mining such data. Knowing what users look at, what they click on, how much time they spend on a web page, etc. leads to better business decisions and competitive advantages. Broadly, this is known as business intelligence, which encompasses a wide range of technologies including data warehousing, data mining, and analytics.

How much data are we talking about? A few examples: Google grew from processing 100 TB of data a day with MapReduce in 2004 [45] to processing 20 PB a day with MapReduce in 2008 [46]. In April 2009, a blog post1 was written about eBay’s two enormous data warehouses: one with 2 petabytes of user data, and the other with 6.5 petabytes of user data spanning 170 trillion records and growing by 150 billion new records per day. Shortly thereafter, Facebook revealed2similarly impressive numbers, boasting of 2.5 petabytes of user data, growing at about 15 terabytes per day. Petabyte datasets are rapidly becoming the norm, and the trends are clear: our ability to store data is fast overwhelming our ability to process what we store. More distressing, increases in capacity are outpacing improvements in bandwidth such that our ability to even *read* back what we store is deteriorating [91]. Disk capacities have grown from tens of megabytes in the mid-1980s to about a couple of terabytes today (several orders of magnitude). On the other hand, latency and bandwidth have improved relatively little: in the case of latency, perhaps 2*×* improvement during the last quarter century, and in the case of bandwidth, perhaps 50*×*. Given the tendency for individuals and organizations to continuously fill up whatever capacity is available, large-data problems are growing increasingly severe.

Moving beyond the commercial sphere, many have recognized the impor tance of data management in many scientific disciplines, where petabyte-scale datasets are also becoming increasingly common [21]. For example:

*•* The high-energy physics community was already describing experiences with petabyte-scale databases back in 2005 [20]. Today, the Large Hadron Collider (LHC) near Geneva is the world’s largest particle accelerator, designed to probe the mysteries of the universe, including the fundamen tal nature of matter, by recreating conditions shortly following the Big Bang. When it becomes fully operational, the LHC will produce roughly 15 petabytes of data a year.3

*•* Astronomers have long recognized the importance of a “digital observa tory” that would support the data needs of researchers across the globe— the Sloan Digital Sky Survey [145] is perhaps the most well known of these projects. Looking into the future, the Large Synoptic Survey Telescope (LSST) is a wide-field instrument that is capable of observing the entire

1http://www.dbms2.com/2009/04/30/ebays-two-enormous-data-warehouses/ 2http://www.dbms2.com/2009/05/11/facebook-hadoop-and-hive/

3http://public.web.cern.ch/public/en/LHC/Computing-en.html

sky every few days. When the telescope comes online around 2015 in Chile, its 3.2 gigapixel primary camera will produce approximately half a petabyte of archive images every month [19].

*•* The advent of next-generation DNA sequencing technology has created a deluge of sequence data that needs to be stored, organized, and delivered to scientists for further study. Given the fundamental tenant in modern genetics that genotypes explain phenotypes, the impact of this technol ogy is nothing less than transformative [103]. The European Bioinfor matics Institute (EBI), which hosts a central repository of sequence data called EMBL-bank, has increased storage capacity from 2.5 petabytes in 2008 to 5 petabytes in 2009 [142]. Scientists are predicting that, in the not-so-distant future, sequencing an individual’s genome will be no more complex than getting a blood test today—ushering a new era of person alized medicine, where interventions can be specifically targeted for an individual.

Increasingly, scientific breakthroughs will be powered by advanced comput ing capabilities that help researchers manipulate, explore, and mine massive datasets [72]—this has been hailed as the emerging “fourth paradigm” of sci ence [73] (complementing theory, experiments, and simulations). In other areas of academia, particularly computer science, systems and algorithms incapable of scaling to massive real-world datasets run the danger of being dismissed as “toy systems” with limited utility. Large data is a fact of today’s world and data-intensive processing is fast becoming a necessity, not merely a luxury or curiosity.

Although large data comes in a variety of forms, this book is primarily con cerned with processing large amounts of text, but touches on other types of data as well (e.g., relational and graph data). The problems and solutions we dis cuss mostly fall into the disciplinary boundaries of natural language processing (NLP) and information retrieval (IR). Recent work in these fields is dominated by a data-driven, empirical approach, typically involving algorithms that at tempt to capture statistical regularities in data for the purposes of some task or application. There are three components to this approach: data, represen tations of the data, and some method for capturing regularities in the data. Data are called *corpora* (singular, corpus) by NLP researchers and *collections* by those from the IR community. Aspects of the representations of the data are called *features*, which may be “superficial” and easy to extract, such as the words and sequences of words themselves, or “deep” and more difficult to ex tract, such as the grammatical relationship between words. Finally, algorithms or models are applied to capture regularities in the data in terms of the ex tracted features for some application. One common application, classification, is to sort text into categories. Examples include: Is this email spam or not spam? Is this word part of an address or a location? The first task is easy to understand, while the second task is an instance of what NLP researchers call named-entity detection [138], which is useful for local search and pinpointing

locations on maps. Another common application is to rank texts according to some criteria—search is a good example, which involves ranking documents by relevance to the user’s query. Another example is to automatically situ ate texts along a scale of “happiness”, a task known as sentiment analysis or opinion mining [118], which has been applied to everything from understand ing political discourse in the blogosphere to predicting the movement of stock prices.

There is a growing body of evidence, at least in text processing, that of the three components discussed above (data, features, algorithms), data probably matters the most. Superficial word-level features coupled with simple models in most cases trump sophisticated models over deeper features and less data. But why can’t we have our cake and eat it too? Why not both sophisticated models *and* deep features applied to lots of data? Because inference over sophisticated models and extraction of deep features are often computationally intensive, they don’t scale well.

Consider a simple task such as determining the correct usage of easily con fusable words such as “than” and “then” in English. One can view this as a supervised machine learning problem: we can train a classifier to disambiguate between the options, and then apply the classifier to new instances of the prob lem (say, as part of a grammar checker). Training data is fairly easy to come by—we can just gather a large corpus of texts and assume that most writers make correct choices (the training data may be noisy, since people make mis takes, but no matter). In 2001, Banko and Brill [14] published what has become a classic paper in natural language processing exploring the effects of training data size on classification accuracy, using this task as the specific example. They explored several classification algorithms (the exact ones aren’t impor tant, as we shall see), and not surprisingly, found that more data led to better accuracy. Across many different algorithms, the increase in accuracy was ap proximately linear in the log of the size of the training data. Furthermore, with increasing amounts of training data, the accuracy of different algorithms con verged, such that pronounced differences in effectiveness observed on smaller datasets basically disappeared at scale. This led to a somewhat controversial conclusion (at least at the time): machine learning algorithms really don’t mat ter, all that matters is the amount of data you have. This resulted in an even more controversial recommendation, delivered somewhat tongue-in-cheek: we should just give up working on algorithms and simply spend our time gather ing data (while waiting for computers to become faster so we can process the data).

As another example, consider the problem of answering short, fact-based questions such as “Who shot Abraham Lincoln?” Instead of returning a list of documents that the user would then have to sort through, a question answer ing (QA) system would directly return the answer: John Wilkes Booth. This problem gained interest in the late 1990s, when natural language processing researchers approached the challenge with sophisticated linguistic processing techniques such as syntactic and semantic analysis. Around 2001, researchers discovered a far simpler approach to answering such questions based on pattern

matching [27, 53, 92]. Suppose you wanted the answer to the above question. As it turns out, you can simply search for the phrase “shot Abraham Lincoln” on the web and look for what appears to its left. Or better yet, look through multiple instances of this phrase and tally up the words that appear to the left. This simple strategy works surprisingly well, and has become known as the *redundancy-based approach* to question answering. It capitalizes on the insight that in a very large text collection (i.e., the web), answers to commonly-asked questions will be stated in obvious ways, such that pattern-matching techniques suffice to extract answers accurately.

Yet another example concerns smoothing in web-scale language models [25]. A language model is a probability distribution that characterizes the likelihood of observing a particular sequence of words, estimated from a large corpus of texts. They are useful in a variety of applications, such as speech recognition (to determine what the speaker is more likely to have said) and machine trans lation (to determine which of possible translations is the most fluent, as we will discuss in Section 7.4). Since there are infinitely many possible strings, and probabilities must be assigned to all of them, language modeling is a more chal lenging task than simply keeping track of which strings were seen how many times: some number of likely strings will never be encountered, even with lots and lots of training data! Most modern language models make the Markov assumption: in a *n*-gram language model, the conditional probability of a word is given by the *n −* 1 previous words. Thus, by the chain rule, the proba bility of a sequence of words can be decomposed into the product of *n*-gram probabilities. Nevertheless, an enormous number of parameters must still be estimated from a training corpus: potentially *Vn* parameters, where *V* is the number of words in the vocabulary. Even if we treat every word on the web as the training corpus from which to estimate the *n*-gram probabilities, most *n*-grams—in any language, even English—will never have been seen. To cope with this sparseness, researchers have developed a number of smoothing tech niques [35, 102, 79], which all share the basic idea of moving probability mass from observed to unseen events in a principled manner. Smoothing approaches vary in effectiveness, both in terms of intrinsic and application-specific metrics. In 2007, Brants et al. [25] described language models trained on up to two tril lion words.4 Their experiments compared a state-of-the-art approach known as Kneser-Ney smoothing [35] with another technique the authors affectionately referred to as “stupid backoff”.5 Not surprisingly, stupid backoff didn’t work as well as Kneser-Ney smoothing on smaller corpora. However, it was simpler and could be trained on *more* data, which ultimately yielded better language models. That is, a simpler technique on more data beat a more sophisticated technique on less data.

Recently, three Google researchers summarized this data-driven philosophy

4As an aside, it is interesting to observe the evolving definition of *large* over the years. Banko and Brill’s paper in 2001 was titled *Scaling to Very Very Large Corpora for Natural Language Disambiguation*, and dealt with a corpus containing a billion words. 5As in, so stupid it couldn’t possibly work.

in an essay titled *The Unreasonable Effectiveness of Data* [65].6 Why is this so? It boils down to the fact that language *in the wild*, just like human behavior in general, is messy. Unlike, say, the interaction of subatomic particles, human *use* of language is not constrained by succinct, universal “laws of grammar”. There are of course rules that govern the formation of words and sentences—for example, that verbs appear before objects in English, and that subjects and verbs must agree in number in many languages—but real-world language is affected by a multitude of other factors as well: people invent new words and phrases all the time, authors occasionally make mistakes, groups of individuals write within a shared context, etc. The Argentine writer Jorge Luis Borges wrote a famous allegorical one-paragraph story about a fictional society in which the art of cartography had gotten so advanced that their maps were as big as the lands they were describing.7 The world, he would say, is the best description of itself. In the same way, the more observations we gather about language use, the more accurate a description we have of language itself. This, in turn, translates into more effective algorithms and systems.

So, in summary, why large data? In some ways, the first answer is similar to the reason people climb mountains: because they’re there. But the second answer is even more compelling. Data represent the rising tide that lifts all boats—more data lead to better algorithms and systems for solving real-world problems. Now that we’ve addressed the *why*, let’s tackle the *how*. Let’s start with the obvious observation: data-intensive processing is beyond the capability of any individual machine and requires clusters—which means that large-data problems are fundamentally about organizing computations on dozens, hun dreds, or even thousands of machines. This is exactly what MapReduce does, and the rest of this book is about the *how*.

1.1 Computing in the Clouds

For better or for worse, it is often difficult to untangle MapReduce and large data processing from the broader discourse on cloud computing. True, there is substantial promise in this new paradigm of computing, but unwarranted hype by the media and popular sources threatens its credibility in the long run. In some ways, cloud computing is simply brilliant marketing. Before clouds, there were grids,8 and before grids, there were vector supercomputers, each having

6This title was inspired by a classic article titled *The Unreasonable Effectiveness of Mathematics in the Natural Sciences* [155]. This is somewhat ironic in that the original article lauded the beauty and elegance of mathematical models in capturing natural phenomena, which is the exact opposite of the data-driven approach.

7*On Exactitude in Science* [23]. A similar exchange appears in Chapter XI of *Sylvie and Bruno Concluded* by Lewis Carroll (1893).

8What *is* the difference between cloud computing and grid computing? Although both tackle the fundamental problem of how best to bring computational resources to bear on large and difficult problems, they start with different assumptions. Whereas clouds are assumed to be relatively homogeneous servers that reside in a datacenter or are distributed across a relatively small number of datacenters controlled by a single organization, grids are assumed to be a less tightly-coupled federation of heterogeneous resources under the control of distinct

claimed to be the best thing since sliced bread.

So what exactly is cloud computing? This is one of those questions where ten experts will give eleven different answers; in fact, countless papers have been written simply to attempt to define the term (e.g., [9, 31, 149], just to name a few examples). Here we offer up our own thoughts and attempt to explain how cloud computing relates to MapReduce and data-intensive processing.

At the most superficial level, everything that used to be called web applica tions has been rebranded to become “cloud applications”, which includes what we have previously called “Web 2.0” sites. In fact, anything running inside a browser that gathers and stores user-generated content now qualifies as an example of cloud computing. This includes social-networking services such as Facebook, video-sharing sites such as YouTube, web-based email services such as Gmail, and applications such as Google Docs. In this context, the cloud simply refers to the servers that power these sites, and user data is said to reside “in the cloud”. The accumulation of vast quantities of user data creates large-data problems, many of which are suitable for MapReduce. To give two concrete examples: a social-networking site analyzes connections in the enor mous globe-spanning graph of friendships to recommend new connections. An online email service analyzes messages and user behavior to optimize ad selec tion and placement. These are all large-data problems that have been tackled with MapReduce.9

Another important facet of cloud computing is what’s more precisely known as utility computing [129, 31]. As the name implies, the idea behind utility com puting is to treat computing resource as a metered service, like electricity or natural gas. The idea harkens back to the days of time-sharing machines, and in truth isn’t very different from this antiquated form of computing. Under this model, a “cloud user” can dynamically provision any amount of comput ing resources from a “cloud provider” on demand and only pay for what is consumed. In practical terms, the user is paying for access to virtual machine instances that run a standard operating system such as Linux. Virtualization technology (e.g., [15]) is used by the cloud provider to allocate available physi cal resources and enforce isolation between multiple users that may be sharing the same hardware. Once one or more virtual machine instances have been provisioned, the user has full control over the resources and can use them for arbitrary computation. Virtual machines that are no longer needed are de stroyed, thereby freeing up physical resources that can be redirected to other users. Resource consumption is measured in some equivalent of machine-hours and users are charged in increments thereof.

Both users and providers benefit in the utility computing model. Users are

but cooperative organizations. As a result, grid computing tends to deal with tasks that are coarser-grained, and must deal with the practicalities of a federated environment, e.g., verifying credentials across multiple administrative domains. Grid computing has adopted a middleware-based approach for tackling many of these challenges.

9The first example is Facebook, a well-known user of Hadoop, in exactly the manner as described [68]. The second is, of course, Google, which uses MapReduce to continuously improve existing algorithms and to devise new algorithms for ad selection and placement.

freed from upfront capital investments necessary to build datacenters and sub stantial reoccurring costs in maintaining them. They also gain the important property of elasticity—as demand for computing resources grow, for example, from an unpredicted spike in customers, more resources can be seamlessly al located from the cloud without an interruption in service. As demand falls, provisioned resources can be released. Prior to the advent of utility computing, coping with unexpected spikes in demand was fraught with challenges: under provision and run the risk of service interruptions, or over-provision and tie up precious capital in idle machines that are depreciating.

From the utility provider point of view, this business also makes sense be cause large datacenters benefit from economies of scale and can be run more efficiently than smaller datacenters. In the same way that insurance works by aggregating risk and redistributing it, utility providers aggregate the computing demands for a large number of users. Although demand may fluctuate signif icantly for each user, overall trends in aggregate demand should be smooth and predictable, which allows the cloud provider to adjust capacity over time with less risk of either offering too much (resulting in inefficient use of capi tal) or too little (resulting in unsatisfied customers). In the world of utility computing, Amazon Web Services currently leads the way and remains the dominant player, but a number of other cloud providers populate a market that is becoming increasingly crowded. Most systems are based on proprietary infrastructure, but there is at least one, Eucalyptus [111], that is available open source. Increased competition will benefit cloud users, but what direct relevance does this have for MapReduce? The connection is quite simple: pro cessing large amounts of data with MapReduce requires access to clusters with sufficient capacity. However, not everyone with large-data problems can afford to purchase and maintain clusters. This is where utility computing comes in: clusters of sufficient size can be provisioned only when the need arises, and users pay only as much as is required to solve their problems. This lowers the barrier to entry for data-intensive processing and makes MapReduce much more accessible.

A generalization of the utility computing concept is “everything as a ser vice”, which is itself a new take on the age-old idea of outsourcing. A cloud provider offering customers access to virtual machine instances is said to be offering infrastructure as a service, or IaaS for short. However, this may be too low level for many users. Enter platform as a service (PaaS), which is a rebrand ing of what used to be called hosted services in the “pre-cloud” era. Platform is used generically to refer to any set of well-defined services on top of which users can build applications, deploy content, etc. This class of services is best exemplified by Google App Engine, which provides the backend datastore and API for anyone to build highly-scalable web applications. Google maintains the infrastructure, freeing the user from having to backup, upgrade, patch, or otherwise maintain basic services such as the storage layer or the programming environment. At an even higher level, cloud providers can offer software as a service (SaaS), as exemplified by Salesforce, a leader in customer relationship management (CRM) software. Other examples include outsourcing an entire

organization’s email to a third party, which is commonplace today. What does this proliferation of services have to do with MapReduce? No doubt that “everything as a service” is driven by desires for greater business efficiencies, but scale and elasticity play important roles as well. The cloud allows seamless expansion of operations without the need for careful planning and supports scales that may otherwise be difficult or cost-prohibitive for an organization to achieve. Cloud services, just like MapReduce, represents the search for an appropriate level of abstraction and beneficial divisions of labor. IaaS is an abstraction over raw physical hardware—an organization might lack the capital, expertise, or interest in running datacenters, and therefore pays a cloud provider to do so on its behalf. The argument applies similarly to PaaS and SaaS. In the same vein, the MapReduce programming model is a powerful abstraction that separates the *what* from the *how* of data-intensive processing.

1.2 Big Ideas

Tackling large-data problems requires a distinct approach that sometimes runs counter to traditional models of computing. In this section, we discuss a num ber of “big ideas” behind MapReduce. To be fair, all of these ideas have been discussed in the computer science literature for some time (some for decades), and MapReduce is certainly not the first to adopt these ideas. Nevertheless, the engineers at Google deserve tremendous credit for pulling these various threads together and demonstrating the power of these ideas on a scale previ ously unheard of.

Scale “out”, not “up”. For data-intensive workloads, a large number of commodity low-end servers (i.e., the scaling “out” approach) is preferred over a small number of high-end servers (i.e., the scaling “up” approach). The lat ter approach of purchasing symmetric multi-processing (SMP) machines with a large number of processor sockets (dozens, even hundreds) and a large amount of shared memory (hundreds or even thousands of gigabytes) is not cost effec tive, since the costs of such machines do not scale linearly (i.e., a machine with twice as many processors is often significantly more than twice as expensive). On the other hand, the low-end server market overlaps with the high-volume desktop computing market, which has the effect of keeping prices low due to competition, interchangeable components, and economies of scale.

Barroso and H¨olzle’s recent treatise of what they dubbed “warehouse-scale computers” [18] contains a thoughtful analysis of the two approaches. The Transaction Processing Council (TPC) is a neutral, non-profit organization whose mission is to establish objective database benchmarks. Benchmark data submitted to that organization are probably the closest one can get to a fair “apples-to-apples” comparison of cost and performance for specific, well-defined relational processing applications. Based on TPC-C benchmark results from late 2007, a low-end server platform is about four times more cost efficient than a high-end shared memory platform from the same vendor. Excluding

storage costs, the price/performance advantage of the low-end server increases to about a factor of twelve.

What if we take into account the fact that communication between nodes in a high-end SMP machine is orders of magnitude faster than communica tion between nodes in a commodity network-based cluster? Since workloads today are beyond the capability of any *single* machine (no matter how power ful), the comparison is more accurately between a smaller cluster of high-end machines and a larger cluster of low-end machines (network communication is unavoidable in both cases). Barroso and H¨olzle model these two approaches under workloads that demand more or less communication, and conclude that a cluster of low-end servers approaches the performance of the equivalent clus ter of high-end servers—the small performance gap is insufficient to justify the price premium of the high-end servers. For data-intensive applications, the conclusion appears to be clear: scaling “out” is superior to scaling “up”, and therefore most existing implementations of the MapReduce programming model are designed around clusters of low-end commodity servers.

Capital costs in acquiring servers is, of course, only one component of the total cost of delivering computing capacity. Operational costs are dominated by the cost of electricity to power the servers as well as other aspects of datacenter operations that are functionally related to power: power distribution, cooling, etc. [67, 18]. As a result, energy efficiency has become a key issue in building warehouse-scale computers for large-data processing. Therefore, it is important to factor in operational costs when deploying a scale-out solution based on large numbers of commodity servers.

Datacenter efficiency is typically factored into three separate components that can be independently measured and optimized [18]. The first component measures how much of a building’s incoming power is actually delivered to computing equipment, and correspondingly, how much is lost to the building’s mechanical systems (e.g., cooling, air handling) and electrical infrastructure (e.g., power distribution inefficiencies). The second component measures how much of a server’s incoming power is lost to the power supply, cooling fans, etc. The third component captures how much of the power delivered to computing components (processor, RAM, disk, etc.) is actually used to perform useful computations.

Of the three components of datacenter efficiency, the first two are relatively straightforward to objectively quantify. Adoption of industry best-practices can help datacenter operators achieve state-of-the-art efficiency. The third component, however, is much more difficult to measure. One important issue that has been identified is the non-linearity between load and power draw. That is, a server at 10% utilization may draw slightly more than half as much power as a server at 100% utilization (which means that a lightly-loaded server is much less efficient than a heavily-loaded server). A survey of five thousand Google servers over a six-month period shows that servers operate most of the time at between 10% and 50% utilization [17], which is an energy-inefficient operating region. As a result, Barroso and H¨olzle have advocated for research and development in energy-proportional machines, where energy consumption

would be proportional to load, such that an idle processor would (ideally) con sume no power, but yet retain the ability to power up (nearly) instantaneously in response to demand.

Although we have provided a brief overview here, datacenter efficiency is a topic that is beyond the scope of this book. For more details, consult Barroso and H¨olzle [18] and Hamilton [67], who provide detailed cost models for typical modern datacenters. However, even factoring in operational costs, evidence suggests that scaling out remains more attractive than scaling up.

Assume failures are common. At warehouse scale, failures are not only inevitable, but commonplace. A simple calculation suffices to demonstrate: let us suppose that a cluster is built from reliable machines with a mean time between failures (MTBF) of 1000 days (about three years). Even with these reliable servers, a 10,000-server cluster would still experience roughly 10 failures a day. For the sake of argument, let us suppose that a MTBF of 10,000 days (about thirty years) were achievable at realistic costs (which is unlikely). Even then, a 10,000-server cluster would still experience one failure daily. This means that any large-scale service that is distributed across a large cluster (either a user-facing application or a computing platform like MapReduce) must cope with hardware failures as an intrinsic aspect of its operation [66]. That is, a server may fail at any time, without notice. For example, in large clusters disk failures are common [123] and RAM experiences more errors than one might expect [135]. Datacenters suffer from both planned outages (e.g., system maintenance and hardware upgrades) and unexpected outages (e.g., power failure, connectivity loss, etc.).

A well-designed, fault-tolerant service must cope with failures up to a point without impacting the quality of service—failures should not result in inconsis tencies or indeterminism from the user perspective. As servers go down, other cluster nodes should seamlessly step in to handle the load, and overall perfor mance should gracefully degrade as server failures pile up. Just as important, a broken server that has been repaired should be able to seamlessly rejoin the service without manual reconfiguration by the administrator. Mature imple mentations of the MapReduce programming model are able to robustly cope with failures through a number of mechanisms such as automatic task restarts on different cluster nodes.

Move processing to the data. In traditional high-performance computing (HPC) applications (e.g., for climate or nuclear simulations), it is commonplace for a supercomputer to have “processing nodes” and “storage nodes” linked to gether by a high-capacity interconnect. Many data-intensive workloads are not very processor-demanding, which means that the separation of compute and storage creates a bottleneck in the network. As an alternative to moving data around, it is more efficient to move the processing around. That is, MapReduce assumes an architecture where processors and storage (disk) are co-located. In such a setup, we can take advantage of data locality by running code on the

processor directly attached to the block of data we need. The distributed file system is responsible for managing the data over which MapReduce operates.

Process data sequentially and avoid random access. Data-intensive processing by definition means that the relevant datasets are too large to fit in memory and must be held on disk. Seek times for random disk access are fun damentally limited by the mechanical nature of the devices: read heads can only move so fast and platters can only spin so rapidly. As a result, it is desirable to avoid random data access, and instead organize computations so that data is processed sequentially. A simple scenario10 poignantly illustrates the large performance gap between sequential operations and random seeks: assume a 1 terabyte database containing 1010 100-byte records. Given reasonable assump tions about disk latency and throughput, a back-of-the-envelop calculation will show that updating 1% of the records (by accessing and then mutating each record) will take about a month on a single machine. On the other hand, if one simply reads the entire database and rewrites all the records (mutating those that need updating), the process would finish in under a work day on a single machine. Sequential data access is, literally, orders of magnitude faster than random data access.11

The development of solid-state drives is unlikely to change this balance for at least two reasons. First, the cost differential between traditional magnetic disks and solid-state disks remains substantial: large-data will for the most part remain on mechanical drives, at least in the near future. Second, although solid state disks have substantially faster seek times, order-of-magnitude differences in performance between sequential and random access still remain.

MapReduce is primarily designed for batch processing over large datasets. To the extent possible, all computations are organized into long streaming operations that take advantage of the aggregate bandwidth of many disks in a cluster. Many aspects of MapReduce’s design explicitly trade latency for throughput.

Hide system-level details from the application developer. According to many guides on the practice of software engineering written by experienced industry professionals, one of the key reasons why writing code is difficult is be cause the programmer must simultaneously keep track of many details in short term memory—ranging from the mundane (e.g., variable names) to the sophis ticated (e.g., a corner case of an algorithm that requires special treatment). This imposes a high cognitive load and requires intense concentration, which leads to a number of recommendations about a programmer’s environment (e.g., quiet office, comfortable furniture, large monitors, etc.). The challenges in writing distributed software are greatly compounded—the programmer must manage details across several threads, processes, or machines. Of course, the

10Adapted from a post by Ted Dunning on the Hadoop mailing list.

11For more detail, Jacobs [76] provides real-world benchmarks in his discussion of large data problems.

biggest headache in distributed programming is that code runs concurrently in unpredictable orders, accessing data in unpredictable patterns. This gives rise to race conditions, deadlocks, and other well-known problems. Programmers are taught to use low-level devices such as mutexes and to apply high-level “de sign patterns” such as producer–consumer queues to tackle these challenges, but the truth remains: concurrent programs are notoriously difficult to reason about and even harder to debug.

MapReduce addresses the challenges of distributed programming by provid ing an abstraction that isolates the developer from system-level details (e.g., locking of data structures, data starvation issues in the processing pipeline, etc.). The programming model specifies simple and well-defined interfaces be tween a small number of components, and therefore is easy for the programmer to reason about. MapReduce maintains a separation of *what* computations are to be performed and *how* those computations are actually carried out on a cluster of machines. The first is under the control of the programmer, while the second is exclusively the responsibility of the execution framework or “run time”. The advantage is that the execution framework only needs to be de signed once and verified for correctness—thereafter, as long as the developer expresses computations in the programming model, code is guaranteed to be have as expected. The upshot is that the developer is freed from having to worry about system-level details (e.g., no more debugging race conditions and addressing lock contention) and can instead focus on algorithm or application design.

Seamless scalability. For data-intensive processing, it goes without saying that scalable algorithms are highly desirable. As an aspiration, let us sketch the behavior of an ideal algorithm. We can define scalability along at least two dimensions.12 First, in terms of data: given twice the amount of data, the same algorithm should take at most twice as long to run, all else being equal. Second, in terms of resources: given a cluster twice the size, the same algorithm should take no more than half as long to run. Furthermore, an ideal algorithm would maintain these desirable scaling characteristics across a wide range of settings: on data ranging from gigabytes to petabytes, on clusters consisting of a few to a few thousand machines. Finally, the ideal algorithm would exhibit these desired behaviors without requiring any modifications whatsoever, not even tuning of parameters.

Other than for embarrassingly parallel problems, algorithms with the char acteristics sketched above are, of course, unobtainable. One of the fundamental assertions in Fred Brook’s classic *The Mythical Man-Month* [28] is that adding programmers to a project behind schedule will only make it fall further be hind. This is because complex tasks cannot be chopped into smaller pieces and allocated in a linear fashion, and is often illustrated with a cute quote: “nine women cannot have a baby in one month”. Although Brook’s obser

12See also DeWitt and Gray [50] for slightly different definitions in terms of *speedup* and *scaleup*.

vations are primarily about software engineers and the software development process, the same is also true of algorithms: increasing the degree of paralleliza tion also increases communication costs. The algorithm designer is faced with diminishing returns, and beyond a certain point, greater efficiencies gained by parallelization are entirely offset by increased communication requirements.

Nevertheless, these fundamental limitations shouldn’t prevent us from at least striving for the unobtainable. The truth is that most current algorithms are far from the ideal. In the domain of text processing, for example, most algorithms today assume that data fits in memory on a single machine. For the most part, this is a fair assumption. But what happens when the amount of data doubles in the near future, and then doubles again shortly thereafter? Simply buying more memory is not a viable solution, as the amount of data is growing faster than the price of memory is falling. Furthermore, the price of a machine does not scale linearly with the amount of available memory beyond a certain point (once again, the scaling “up” vs. scaling “out” argument). Quite simply, algorithms that require holding intermediate data in memory on a single machine will simply break on sufficiently-large datasets—moving from a single machine to a cluster architecture requires fundamentally different algorithms (and reimplementations).

Perhaps the most exciting aspect of MapReduce is that it represents a small step toward algorithms that behave in the ideal manner discussed above. Re call that the programming model maintains a clear separation between *what* computations need to occur with *how* those computations are actually orches trated on a cluster. As a result, a MapReduce algorithm remains fixed, and it is the responsibility of the execution framework to execute the algorithm. Amaz ingly, the MapReduce programming model is simple enough that it is actually possible, in many circumstances, to *approach* the ideal scaling characteristics discussed above. We introduce the idea of the “tradeable machine hour”, as a play on Brook’s classic title. If running an algorithm on a particular dataset takes 100 machine hours, then we should be able to finish in an hour on a cluster of 100 machines, or use a cluster of 10 machines to complete the same task in ten hours.13 With MapReduce, this isn’t so far from the truth, at least for some applications.

1.3 Why Is This Different?

“Due to the rapidly decreasing cost of processing, memory, and communication, it has appeared inevitable for at least two decades that parallel machines will eventually displace sequential ones in computationally intensive domains. This, however, has not hap pened.” — Leslie Valiant [148]14

13Note that this idea meshes well with utility computing, where a 100-machine cluster running for one hour would cost the same as a 10-machine cluster running for ten hours. 14Guess when this was written? You may be surprised.

For several decades, computer scientists have predicted that the dawn of the age of parallel computing was “right around the corner” and that sequential processing would soon fade into obsolescence (consider, for example, the above quote). Yet, until very recently, they have been wrong. The relentless progress of Moore’s Law for several decades has ensured that most of the world’s prob lems could be solved by single-processor machines, save the needs of a few (scientists simulating molecular interactions or nuclear reactions, for example). Couple that with the inherent challenges of concurrency, and the result has been that parallel processing and distributed systems have largely been con fined to a small segment of the market and esoteric upper-level electives in the computer science curriculum.

However, all of that changed around the middle of the first decade of this century. The manner in which the semiconductor industry had been exploiting Moore’s Law simply ran out of opportunities for improvement: faster clocks, deeper pipelines, superscalar architectures, and other tricks of the trade reached a point of diminishing returns that did not justify continued investment. This marked the beginning of an entirely new strategy and the dawn of the multi core era [115]. Unfortunately, this radical shift in hardware architecture was not matched at that time by corresponding advances in how software could be easily designed for these new processors (but not for lack of trying [104]). Nevertheless, parallel processing became an important issue at the forefront of everyone’s mind—it represented the only way forward.

At around the same time, we witnessed the growth of large-data problems. In the late 1990s and even during the beginning of the first decade of this century, relatively few organizations had data-intensive processing needs that required large clusters: a handful of internet companies and perhaps a few dozen large corporations. But then, everything changed. Through a combination of many different factors (falling prices of disks, rise of user-generated web con tent, etc.), large-data problems began popping up everywhere. Data-intensive processing needs became widespread, which drove innovations in distributed computing such as MapReduce—first by Google, and then by Yahoo and the open source community. This in turn created more demand: when organiza tions learned about the availability of effective data analysis tools for large datasets, they began instrumenting various business processes to gather even more data—driven by the belief that more data leads to deeper insights and greater competitive advantages. Today, not only are large-data problems ubiq uitous, but technological solutions for addressing them are widely accessible. Anyone can download the open source Hadoop implementation of MapReduce, pay a modest fee to rent a cluster from a utility cloud provider, and be hap pily processing terabytes upon terabytes of data within the week. Finally, the computer scientists are right—the age of parallel computing has begun, both in terms of multiple cores in a chip and multiple machines in a cluster (each of which often has multiple cores).

Why is MapReduce important? In practical terms, it provides a very ef fective tool for tackling large-data problems. But beyond that, MapReduce is important in how it has changed the way we organize computations at a

massive scale. MapReduce represents the first *widely-adopted* step away from the von Neumann model that has served as the foundation of computer sci ence over the last half plus century. Valiant called this a *bridging model* [148], a conceptual bridge between the physical implementation of a machine and the software that is to be executed on that machine. Until recently, the von Neumann model has served us well: Hardware designers focused on efficient im plementations of the von Neumann model and didn’t have to think much about the actual software that would run on the machines. Similarly, the software industry developed software targeted at the model without worrying about the hardware details. The result was extraordinary growth: chip designers churned out successive generations of increasingly powerful processors, and software en gineers were able to develop applications in high-level languages that exploited those processors.

Today, however, the von Neumann model isn’t sufficient anymore: we can’t treat a multi-core processor or a large cluster as an agglomeration of many von Neumann machine instances communicating over some interconnect. Such a view places too much burden on the software developer to effectively take advantage of available computational resources—it simply is the wrong level of abstraction. MapReduce can be viewed as the first breakthrough in the quest for new abstractions that allow us to organize computations, not over individual machines, but over entire clusters. As Barroso puts it, the datacenter *is* the computer [18, 119].

To be fair, MapReduce is certainly not the first model of parallel com putation that has been proposed. The most prevalent model in theoretical computer science, which dates back several decades, is the PRAM [77, 60].15 In the model, an arbitrary number of processors, sharing an unboundedly large memory, operate synchronously on a shared input to produce some output. Other models include LogP [43] and BSP [148]. For reasons that are beyond the scope of this book, none of these previous models have enjoyed the success that MapReduce has in terms of adoption and in terms of impact on the daily lives of millions of users.16

MapReduce is the most successful abstraction over large-scale computa tional resources we have seen to date. However, as anyone who has taken an introductory computer science course knows, abstractions manage complexity by hiding details and presenting well-defined behaviors to users of those ab stractions. They, inevitably, are imperfect—making certain tasks easier but others more difficult, and sometimes, impossible (in the case where the detail suppressed by the abstraction is exactly what the user cares about). This cri tique applies to MapReduce: it makes certain large-data problems easier, but suffers from limitations as well. This means that MapReduce is not the final

15More than a theoretical model, the PRAM has been recently prototyped in hard ware [153].

16Nevertheless, it is important to understand the relationship between MapReduce and existing models so that we can bring to bear accumulated knowledge about parallel algo rithms; for example, Karloff et al. [82] demonstrated that a large class of PRAM algorithms can be efficiently simulated via MapReduce.

word, but rather the first in a new class of programming models that will allow us to more effectively organize computations at a massive scale. So if MapReduce is only the beginning, what’s next beyond MapReduce? We’re getting ahead of ourselves, as we can’t meaningfully answer this question before thoroughly understanding what MapReduce can and cannot do well. This is exactly the purpose of this book: let us now begin our exploration.

1.4 What This Book Is Not

Actually, not quite yet. . . A final word before we get started. This book is about MapReduce algorithm design, particularly for text processing (and related) applications. Although our presentation most closely follows the Hadoop open source implementation of MapReduce, this book is explicitly *not* about Hadoop programming. We don’t for example, discuss APIs, command-line invocations for running jobs, etc. For those aspects, we refer the reader to Tom White’s excellent book, “Hadoop: The Definitive Guide”, published by O’Reilly [154].

Chapter 2

MapReduce Basics

The only feasible approach to tackling large-data problems today is to divide and conquer, a fundamental concept in computer science that is introduced very early in typical undergraduate curricula. The basic idea is to partition a large problem into smaller sub-problems. To the extent that the sub-problems are independent [5], they can be tackled in parallel by different workers— threads in a processor core, cores in a multi-core processor, multiple processors in a machine, or many machines in a cluster. Intermediate results from each individual worker are then combined to yield the final output.1

The general principles behind divide-and-conquer algorithms are broadly applicable to a wide range of problems in many different application domains. However, the details of their implementations are varied and complex. For example, the following are just some of the issues that need to be addressed:

*•* How do we break up a large problem into smaller tasks? More specifi cally, how do we decompose the problem so that the smaller tasks can be executed in parallel?

*•* How do we assign tasks to workers distributed across a potentially large number of machines (while keeping in mind that some workers are better suited to running some tasks than others, e.g., due to available resources, locality constraints, etc.)?

*•* How do we ensure that the workers get the data they need? *•* How do we coordinate synchronization among the different workers?

*•* How do we share partial results from one worker that is needed by an other?

*•* How do we accomplish all of the above in the face of software errors and hardware faults?

1We note that promising technologies such as quantum or biological computing could potentially induce a paradigm shift, but they are far from being sufficiently mature to solve real world problems.

19

In traditional parallel or distributed programming environments, the devel oper needs to explicitly address many (and sometimes, all) of the above issues. In shared memory programming, the developer needs to explicitly coordinate access to shared data structures through synchronization primitives such as mutexes, to explicitly handle process synchronization through devices such as barriers, and to remain ever vigilant for common problems such as deadlocks and race conditions. Language extensions, like OpenMP for shared memory parallelism,2 or libraries implementing the Message Passing Interface (MPI) for cluster-level parallelism,3 provide logical abstractions that hide details of operating system synchronization and communications primitives. However, even with these extensions, developers are still burdened to keep track of how resources are made available to workers. Additionally, these frameworks are mostly designed to tackle processor-intensive problems and have only rudimen tary support for dealing with very large amounts of input data. When using existing parallel computing approaches for large-data computation, the pro grammer must devote a significant amount of attention to low-level system details, which detracts from higher-level problem solving.

One of the most significant advantages of MapReduce is that it provides an abstraction that hides many system-level details from the programmer. There fore, a developer can focus on what computations need to be performed, as opposed to how those computations are actually carried out or how to get the data to the processes that depend on them. Like OpenMP and MPI, MapReduce provides a means to distribute computation without burdening the programmer with the details of distributed computing (but at a different level of granularity). However, organizing and coordinating large amounts of computation is only part of the challenge. Large-data processing by definition requires bringing data and code together for computation to occur—no small feat for datasets that are terabytes and perhaps petabytes in size! MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a scalable, ro bust, and efficient manner. As we mentioned in Chapter 1, instead of moving large amounts of data around, it is far more efficient, if possible, to move the code to the data. This is operationally realized by spreading data across the local disks of nodes in a cluster and running processes on nodes that hold the data. The complex task of managing storage in such a processing envi ronment is typically handled by a distributed file system that sits underneath MapReduce.

This chapter introduces the MapReduce programming model and the un derlying distributed file system. We start in Section 2.1 with an overview of functional programming, from which MapReduce draws its inspiration. Sec tion 2.2 introduces the basic programming model, focusing on mappers and reducers. Section 2.3 discusses the role of the execution framework in actually running MapReduce programs (called jobs). Section 2.4 fills in additional de

2http://www.openmp.org/

3http://www.mcs.anl.gov/mpi/

tails by introducing partitioners and combiners, which provide greater control over data flow. MapReduce would not be practical without a tightly-integrated distributed file system that manages the data being processed; Section 2.5 cov ers this in detail. Tying everything together, a complete cluster architecture is described in Section 2.6 before the chapter ends with a summary.

2.1 Functional Programming Roots

MapReduce has its roots in functional programming, which is exemplified in languages such as Lisp and ML.4 A key feature of functional languages is the concept of higher-order functions, or functions that can accept other functions as arguments. Two common built-in higher order functions are *map* and *fold*, illustrated in Figure 2.1. Given a list, *map* takes as an argument a function *f* (that takes a single argument) and applies it to all elements in a list (the top part of the diagram). Given a list, *fold* takes as arguments a function *g* (that takes two arguments) and an initial value: *g* is first applied to the initial value and the first item in the list, the result of which is stored in an intermediate variable. This intermediate variable and the next item in the list serve as the arguments to a second application of *g*, the results of which are stored in the intermediate variable. This process repeats until all items in the list have been consumed; *fold* then returns the final value of the intermediate variable. Typically, *map* and *fold* are used in combination. For example, to compute the sum of squares of a list of integers, one could map a function that squares its argument (i.e., *λx.x*2) over the input list, and then fold the resulting list with the addition function (more precisely, *λxλy.x* + *y*) using an initial value of zero.

We can view *map* as a concise way to represent the transformation of a dataset (as defined by the function *f*). In the same vein, we can view *fold* as an aggregation operation, as defined by the function *g*. One immediate observation is that the application of *f* to each item in a list (or more generally, to elements in a large dataset) can be parallelized in a straightforward manner, since each functional application happens in isolation. In a cluster, these operations can be distributed across many different machines. The *fold* operation, on the other hand, has more restrictions on data locality—elements in the list must be “brought together” before the function *g* can be applied. However, many real-world applications do not require *g* to be applied to *all* elements of the list. To the extent that elements in the list can be divided into groups, the fold aggregations can also proceed in parallel. Furthermore, for operations that are commutative and associative, significant efficiencies can be gained in the *fold* operation through local aggregation and appropriate reordering.

In a nutshell, we have described MapReduce. The map phase in MapReduce roughly corresponds to the *map* operation in functional programming, whereas the reduce phase in MapReduce roughly corresponds to the *fold* operation in

4However, there are important characteristics of MapReduce that make it non-functional in nature—this will become apparent later.

*f f f f f*

*g g g g g*

Figure 2.1: Illustration of *map* and *fold*, two higher-order functions commonly used together in functional programming: *map* takes a function *f* and applies it to every element in a list, while *fold* iteratively applies a function *g* to aggregate results.

Figure 2.1: IllustraPon of map and fold, two higher-order funcPons commonly used together in funcPonal programming: map takes a funcPon f and applies it to every element in a list, while fold iteraPvely applies a funcPon g to aggregate results.

functional programming. As we will discuss in detail shortly, the MapReduce execution framework coordinates the map and reduce phases of processing over large amounts of data on large clusters of commodity machines.

Viewed from a slightly different angle, MapReduce codifies a generic “recipe” for processing large datasets that consists of two stages. In the first stage, a user-specified computation is applied over all input records in a dataset. These operations occur in parallel and yield intermediate output that is then aggre gated by another user-specified computation. The programmer defines these two types of computations, and the execution framework coordinates the ac tual processing (very loosely, MapReduce provides a functional abstraction). Although such a two-stage processing structure may appear to be very restric tive, many interesting algorithms can be expressed quite concisely—especially if one decomposes complex algorithms into a sequence of MapReduce jobs. Subsequent chapters in this book focus on how a number of algorithms can be implemented in MapReduce.

To be precise, MapReduce can refer to three distinct but related concepts. First, MapReduce is a programming model, which is the sense discussed above. Second, MapReduce can refer to the execution framework (i.e., the “runtime”) that coordinates the execution of programs written in this particular style. Fi nally, MapReduce can refer to the software implementation of the programming model and the execution framework: for example, Google’s proprietary imple mentation vs. the open-source Hadoop implementation in Java. And in fact, there are many implementations of MapReduce, e.g., targeted specifically for multi-core processors [127], for GPGPUs [71], for the CELL architecture [126], etc. There are some differences between the MapReduce programming model

implemented in Hadoop and Google’s proprietary implementation, which we will explicitly discuss throughout the book. However, we take a rather Hadoop centric view of MapReduce, since Hadoop remains the most mature and acces sible implementation to date, and therefore the one most developers are likely to use.

2.2 Mappers and Reducers

Key-value pairs form the basic data structure in MapReduce. Keys and values may be primitives such as integers, floating point values, strings, and raw bytes, or they may be arbitrarily complex structures (lists, tuples, associative arrays, etc.). Programmers typically need to define their own custom data types, although a number of libraries such as Protocol Buffers,5 Thrift,6 and Avro7 simplify the task.

Part of the design of MapReduce algorithms involves imposing the key value structure on arbitrary datasets. For a collection of web pages, keys may be URLs and values may be the actual HTML content. For a graph, keys may represent node ids and values may contain the adjacency lists of those nodes (see Chapter 5 for more details). In some algorithms, input keys are not particularly meaningful and are simply ignored during processing, while in other cases input keys are used to uniquely identify a datum (such as a record id). In Chapter 3, we discuss the role of complex keys and values in the design of various algorithms.

In MapReduce, the programmer defines a mapper and a reducer with the following signatures:

map: (*k*1*, v*1) *→* [(*k*2*, v*2)]

reduce: (*k*2*,* [*v*2]) *→* [(*k*3*, v*3)]

The convention [*. . .*] is used throughout this book to denote a list. The input to a MapReduce job starts as data stored on the underlying distributed file system (see Section 2.5). The mapper is applied to every input key-value pair (split across an arbitrary number of files) to generate an arbitrary number of intermediate key-value pairs. The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs.8Implicit between the map and reduce phases is a distributed “group by” operation on intermediate keys. Intermediate data arrive at each reducer in order, sorted by the key. However, no ordering relationship is guaranteed for keys across different reducers. Output key-value pairs from each reducer are written per sistently back onto the distributed file system (whereas intermediate key-value pairs are transient and not preserved). The output ends up in *r* files on the

5 http://code.google.com/p/protobuf/

6 http://incubator.apache.org/thrift/

7 http://hadoop.apache.org/avro/

8This characterization, while conceptually accurate, is a slight simplification. See Sec tion 2.6 for more details.

distributed file system, where *r* is the number of reducers. For the most part, there is no need to consolidate reducer output, since the *r* files often serve as input to yet another MapReduce job. Figure 2.2 illustrates this two-stage processing structure.

A simple word count algorithm in MapReduce is shown in Algorithm 2.1. This algorithm counts the number of occurrences of every word in a text collec tion, which may be the first step in, for example, building a unigram language model (i.e., probability distribution over words in a collection). Input key values pairs take the form of (docid, doc) pairs stored on the distributed file system, where the former is a unique identifier for the document, and the lat ter is the text of the document itself. The mapper takes an input key-value pair, tokenizes the document, and emits an intermediate key-value pair for ev ery word: the word itself serves as the key, and the integer one serves as the value (denoting that we’ve seen the word once). The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Therefore, in our word count algorithm, we simply need to sum up all counts (ones) associated with each word. The reducer does exactly this, and emits final key-value pairs with the word as the key, and the count as the value. Final output is written to the distributed file system, one file per reducer. Words within each file will be sorted by alphabetical order, and each file will contain roughly the same number of words. The partitioner, which we discuss later in Section 2.4, controls the assignment of words to re ducers. The output can be examined by the programmer or used as input to another MapReduce program.

There are some differences between the Hadoop implementation of Map Reduce and Google’s implementation.9In Hadoop, the reducer is presented with a key and an iterator over all values associated with the particular key. The values are arbitrarily ordered. Google’s implementation allows the pro grammer to specify a secondary sort key for ordering the values (if desired)—in which case values associated with each key would be presented to the devel oper’s reduce code in sorted order. Later in Section 3.4 we discuss how to overcome this limitation in Hadoop to perform secondary sorting. Another dif ference: in Google’s implementation the programmer is not allowed to change the key in the reducer. That is, the reducer output key must be exactly the same as the reducer input key. In Hadoop, there is no such restriction, and the reducer can emit an arbitrary number of output key-value pairs (with different keys).

To provide a bit more implementation detail: pseudo-code provided in this book roughly mirrors how MapReduce programs are written in Hadoop. Map pers and reducers are objects that implement the Map and Reduce methods, respectively. In Hadoop, a mapper object is initialized for each map task (as sociated with a particular sequence of key-value pairs called an input split) and the Map method is called on each key-value pair by the execution frame work. In configuring a MapReduce job, the programmer provides a hint on the

9Personal communication, Jeff Dean.

A α B β C γ D δ E ε F ζ

mapper mapper mapper mapper

a 1 b 2 c 3 c 6 a 5 c 2 b 7 c 8

Shuffle and Sort: aggregate values by keys

a 1 5 b 2 7 c 2 9 8

reducer reducer reducer

X 5 Y 7 Z 9

Figure 2.2: Simplified view of MapReduce. Mappers are applied to all input Figure 2.2: Simplified view of MapReduce. Mappers are applied to all input key-value pairs,

key-value pairs, which generate an arbitrary number of intermediate key-value which generate an arbitrary number of intermediate key-value pairs. Reducers are applied to

pairs. Reducers are applied to all values associated with the same key. Between all values associated with the same key. Between the map and reduce phases lies a barrier

the map and reduce phases lies a barrier that involves a large distributed sort that involves a large distributed sort and group by.

and group by.

Algorithm 2.1 Word count

The mapper emits an intermediate key-value pair for each word in a document. The reducer sums up all counts for each word.

1: class Mapper

2: method Map(docid *a,* doc *d*)

3: for all term *t ∈* doc *d* do

4: Emit(term *t,* count 1)

1: class Reducer

2: method Reduce(term *t,* counts [*c*1*, c*2*, . . .*])

3: *sum ←* 0

4: for all count *c ∈* counts [*c*1*, c*2*, . . .*] do

5: *sum ← sum* + *c*

6: Emit(term *t,* count *sum*)

number of map tasks to run, but the execution framework (see next section) makes the final determination based on the physical layout of the data (more details in Section 2.5 and Section 2.6). The situation is similar for the reduce phase: a reducer object is initialized for each reduce task, and the Reduce method is called once per intermediate key. In contrast with the number of map tasks, the programmer can precisely specify the number of reduce tasks. We will return to discuss the details of Hadoop job execution in Section 2.6, which is dependent on an understanding of the distributed file system (covered in Section 2.5). To reiterate: although the presentation of algorithms in this book closely mirrors the way they would be implemented in Hadoop, our fo cus is on algorithm design and conceptual understanding—not actual Hadoop programming. For that, we would recommend Tom White’s book [154].

What are the restrictions on mappers and reducers? Mappers and reduc ers can express arbitrary computations over their inputs. However, one must generally be careful about use of external resources since multiple mappers or reducers may be contending for those resources. For example, it may be unwise for a mapper to query an external SQL database, since that would introduce a scalability bottleneck on the number of map tasks that could be run in parallel (since they might all be simultaneously querying the database).10 In general, mappers can emit an arbitrary number of intermediate key-value pairs, and they need not be of the same type as the input key-value pairs. Similarly, reducers can emit an arbitrary number of final key-value pairs, and they can differ in type from the intermediate key-value pairs. Although not permitted in functional programming, mappers and reducers can have side effects. This is a powerful and useful feature: for example, preserving state across multiple inputs is central to the design of many MapReduce algorithms (see Chapter 3). Such algorithms can be understood as having side effects that only change state that is *internal* to the mapper or reducer. While the correctness of such algorithms may be more difficult to guarantee (since the function’s behavior depends not only on the current input but on previous inputs), most potential synchronization problems are avoided since internal state is private only to in dividual mappers and reducers. In other cases (see Section 4.4 and Section 7.5), it may be useful for mappers or reducers to have *external* side effects, such as writing files to the distributed file system. Since many mappers and reducers are run in parallel, and the distributed file system is a shared global resource, special care must be taken to ensure that such operations avoid synchroniza tion conflicts. One strategy is to write a temporary file that is renamed upon successful completion of the mapper or reducer [45].

In addition to the “canonical” MapReduce processing flow, other variations are also possible. MapReduce programs can contain no reducers, in which case mapper output is directly written to disk (one file per mapper). For embar rassingly parallel problems, e.g., parse a large text collection or independently analyze a large number of images, this would be a common pattern. The converse—a MapReduce program with no mappers—is not possible, although

10Unless, of course, the database itself is highly scalable.

in some cases it is useful for the mapper to implement the identity function and simply pass input key-value pairs to the reducers. This has the effect of sorting and regrouping the input for reduce-side processing. Similarly, in some cases it is useful for the reducer to implement the identity function, in which case the program simply sorts and groups mapper output. Finally, running identity mappers and reducers has the effect of regrouping and resorting the input data (which is sometimes useful).

Although in the most common case, input to a MapReduce job comes from data stored on the distributed file system and output is written back to the distributed file system, any other system that satisfies the proper abstractions can serve as a data source or sink. With Google’s MapReduce implementation, BigTable [34], a sparse, distributed, persistent multidimensional sorted map, is frequently used as a source of input and as a store of MapReduce output. HBase is an open-source BigTable clone and has similar capabilities. Also, Hadoop has been integrated with existing MPP (massively parallel processing) relational databases, which allows a programmer to write MapReduce jobs over database rows and dump output into a new database table. Finally, in some cases MapReduce jobs may not consume any input at all (e.g., computing *π*) or may only consume a small amount of data (e.g., input parameters to many instances of processor-intensive simulations running in parallel).

2.3 The Execution Framework

One of the most important idea behind MapReduce is separating the *what* of distributed processing from the *how*. A MapReduce program, referred to as a job, consists of code for mappers and reducers (as well as combiners and partitioners to be discussed in the next section) packaged together with config uration parameters (such as where the input lies and where the output should be stored). The developer submits the job to the submission node of a cluster (in Hadoop, this is called the jobtracker) and execution framework (sometimes called the “runtime”) takes care of everything else: it transparently handles all other aspects of distributed code execution, on clusters ranging from a single node to a few thousand nodes. Specific responsibilities include:

Scheduling. Each MapReduce job is divided into smaller units called tasks (see Section 2.6 for more details). For example, a map task may be responsible for processing a certain block of input key-value pairs (called an input split in Hadoop); similarly, a reduce task may handle a portion of the intermediate key space. It is not uncommon for MapReduce jobs to have thousands of individual tasks that need to be assigned to nodes in the cluster. In large jobs, the total number of tasks may exceed the number of tasks that can be run on the cluster concurrently, making it necessary for the scheduler to maintain some sort of a task queue and to track the progress of running tasks so that waiting tasks can be assigned to nodes as they become available. Another aspect of scheduling involves coordination among tasks belonging to different jobs (e.g.,

from different users). How can a large, shared resource support several users simultaneously in a predictable, transparent, policy-driven fashion? There has been some recent work along these lines in the context of Hadoop [131, 160].

Speculative execution is an optimization that is implemented by both Hadoop and Google’s MapReduce implementation (called “backup tasks” [45]). Due to the barrier between the map and reduce tasks, the map phase of a job is only as fast as the slowest map task. Similarly, the completion time of a job is bounded by the running time of the slowest reduce task. As a result, the speed of a MapReduce job is sensitive to what are known as *stragglers*, or tasks that take an usually long time to complete. One cause of stragglers is flaky hardware: for example, a machine that is suffering from recoverable errors may become significantly slower. With speculative execution, an identical copy of the same task is executed on a different machine, and the framework simply uses the result of the first task attempt to finish. Zaharia et al. [161] presented different execution strategies in a recent paper, and Google has reported that speculative execution can improve job running times by 44% [45]. Although in Hadoop both map and reduce tasks can be speculatively executed, the com mon wisdom is that the technique is more helpful for map tasks than reduce tasks, since each copy of the reduce task needs to pull data over the network. Note, however, that speculative execution cannot adequately address another common cause of stragglers: skew in the distribution of values associated with intermediate keys (leading to reduce stragglers). In text processing we often observe Zipfian distributions, which means that the task or tasks responsible for processing the most frequent few elements will run much longer than the typical task. Better local aggregation, discussed in the next chapter, is one possible solution to this problem.

Data/code co-location. The phrase *data distribution* is misleading, since one of the key ideas behind MapReduce is to move the code, not the data. However, the more general point remains—in order for computation to occur, we need to somehow feed data to the code. In MapReduce, this issue is in explicably intertwined with scheduling and relies heavily on the design of the underlying distributed file system.11 To achieve data locality, the scheduler starts tasks on the node that holds a particular block of data (i.e., on its local drive) needed by the task. This has the effect of moving code to the data. If this is not possible (e.g., a node is already running too many tasks), new tasks will be started elsewhere, and the necessary data will be streamed over the network. An important optimization here is to prefer nodes that are on the same rack in the datacenter as the node holding the relevant data block, since inter-rack bandwidth is significantly less than intra-rack bandwidth.

Synchronization. In general, synchronization refers to the mechanisms by which multiple concurrently running processes “join up”, for example, to share

11In the canonical case, that is. Recall that MapReduce may receive its input from other sources.

intermediate results or otherwise exchange state information. In MapReduce, synchronization is accomplished by a barrier between the map and reduce phases of processing. Intermediate key-value pairs must be grouped by key, which is accomplished by a large distributed sort involving all the nodes that executed map tasks and all the nodes that will execute reduce tasks. This necessarily involves copying intermediate data over the network, and therefore the process is commonly known as “shuffle and sort”. A MapReduce job with *m* mappers and *r* reducers involves up to *m × r* distinct copy operations, since each mapper may have intermediate output going to every reducer.

Note that the reduce computation cannot start until all the mappers have finished emitting key-value pairs and all intermediate key-value pairs have been shuffled and sorted, since the execution framework cannot otherwise guarantee that all values associated with the same key have been gathered. This is an important departure from functional programming: in a *fold* operation, the ag gregation function *g* is a function of the intermediate value and the next item in the list—which means that values can be lazily generated and aggregation can begin as soon as values are available. In contrast, the reducer in Map Reduce receives *all* values associated with the same key at once. However, it is possible to start copying intermediate key-value pairs over the network to the nodes running the reducers as soon as each mapper finishes—this is a common optimization and implemented in Hadoop.

Error and fault handling. The MapReduce execution framework must ac complish all the tasks above in an environment where errors and faults are the norm, not the exception. Since MapReduce was explicitly designed around low-end commodity servers, the runtime must be especially resilient. In large clusters, disk failures are common [123] and RAM experiences more errors than one might expect [135]. Datacenters suffer from both planned outages (e.g., system maintenance and hardware upgrades) and unexpected outages (e.g., power failure, connectivity loss, etc.).

And that’s just hardware. No software is bug free—exceptions must be ap propriately trapped, logged, and recovered from. Large-data problems have a penchant for uncovering obscure corner cases in code that is otherwise thought to be bug-free. Furthermore, any sufficiently large dataset will contain cor rupted data or records that are mangled beyond a programmer’s imagination— resulting in errors that one would never think to check for or trap. The Map Reduce execution framework must thrive in this hostile environment.

2.4 Partitioners and Combiners

We have thus far presented a simplified view of MapReduce. There are two additional elements that complete the programming model: partitioners and combiners.

Partitioners are responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. In other words, the par-

titioner specifies the task to which an intermediate key-value pair must be copied. Within each reducer, keys are processed in sorted order (which is how the “group by” is implemented). The simplest partitioner involves computing the hash value of the key and then taking the mod of that value with the number of reducers. This assigns approximately the same number of keys to each reducer (dependent on the quality of the hash function). Note, however, that the partitioner only considers the key and ignores the value—therefore, a roughly-even partitioning of the key space may nevertheless yield large dif ferences in the number of key-values pairs sent to each reducer (since different keys may have different numbers of associated values). This imbalance in the amount of data associated with each key is relatively common in many text processing applications due to the Zipfian distribution of word occurrences.

Combiners are an optimization in MapReduce that allow for local aggrega tion before the shuffle and sort phase. We can motivate the need for combiners by considering the word count algorithm in Algorithm 2.1, which emits a key value pair for each word in the collection. Furthermore, all these key-value pairs need to be copied across the network, and so the amount of intermediate data will be larger than the input collection itself. This is clearly inefficient. One solution is to perform local aggregation on the output of each mapper, i.e., to compute a local count for a word over all the documents processed by the mapper. With this modification (assuming the maximum amount of local aggregation possible), the number of intermediate key-value pairs will be at most the number of unique words in the collection times the number of map pers (and typically far smaller because each mapper may not encounter every word).

The combiner in MapReduce supports such an optimization. One can think of combiners as “mini-reducers” that take place on the output of the mappers, prior to the shuffle and sort phase. Each combiner operates in isolation and therefore does not have access to intermediate output from other mappers. The combiner is provided keys and values associated with each key (the same types as the mapper output keys and values). Critically, one cannot assume that a combiner will have the opportunity to process *all* values associated with the same key. The combiner can emit any number of key-value pairs, but the keys and values must be of the same type as the mapper output (same as the reducer input).12 In cases where an operation is both associative and commutative (e.g., addition or multiplication), reducers can directly serve as combiners. In general, however, reducers and combiners are not interchangeable.

In many cases, proper use of combiners can spell the difference between an impractical algorithm and an efficient algorithm. This topic will be discussed

12A note on the implementation of combiners in Hadoop: by default, the execution frame work reserves the right to use combiners at its discretion. In reality, this means that a combiner may be invoked zero, one, or multiple times. In addition, combiners in Hadoop may actually be invoked in the reduce phase, i.e., after key-value pairs have been copied over to the reducer, but before the user reducer code runs. As a result, combiners must be carefully written so that they can be executed in these different environments. Section 3.1 discusses this in more detail.

A α B β C γ D δ E ε F ζ

mapper mapper mapper mapper

a 1 b 2 c 3 c 6 a 5 c 2 b 7 c 8

combiner combiner combiner combiner

a 1 b 2 c 9 a 5 c 2 b 7 c 8

partitioner partitioner partitioner partitioner

Shuffle and Sort: aggregate values by keys

a 1 5 b 2 7 c 2 9 8

reducer reducer reducer

X 5 Y 7 Z 9

Figure 2.4: Complete view of MapReduce, illustraPng combiners and parPPoners in addiPon Figure 2.3: Complete view of MapReduce, illustrating combiners and parti

to mappers and reducers. Combiners can be viewed as “mini-reducers” in the map phase. tioners in addition to mappers and reducers. Combiners can be viewed as

ParPPoners determine which reducer is responsible for a parPcular key. “mini-reducers” in the map phase. Partitioners determine which reducer is responsible for a particular key.

in Section 3.1, which focuses on various techniques for local aggregation. It suffices to say for now that a combiner can significantly reduce the amount of data that needs to be copied over the network, resulting in much faster algorithms.

The complete MapReduce model is shown in Figure 2.3. Output of the mappers are processed by the combiners, which perform local aggregation to cut down on the number of intermediate key-value pairs. The partitioner deter mines which reducer will be responsible for processing a particular key, and the execution framework uses this information to copy the data to the right loca tion during the shuffle and sort phase.13 Therefore, a complete MapReduce job consists of code for the mapper, reducer, combiner, and partitioner, along with job configuration parameters. The execution framework handles everything else.

13In Hadoop, partitioners are actually executed before combiners, so while Figure 2.3 is conceptually accurate, it doesn’t precisely describe the Hadoop implementation.

2.5 The Distributed File System

So far, we have mostly focused on the *processing* aspect of data-intensive pro cessing, but it is important to recognize that without data, there is nothing to compute on. In high-performance computing (HPC) and many traditional cluster architectures, storage is viewed as a distinct and separate component from computation. Implementations vary widely, but network-attached storage (NAS) and storage area networks (SAN) are common; supercomputers often have dedicated subsystems for handling storage (separate nodes, and often even separate networks). Regardless of the details, the processing cycle remains the same at a high level: the compute nodes fetch input from storage, load the data into memory, process the data, and then write back the results (with perhaps intermediate checkpointing for long-running processes).

As dataset sizes increase, more compute capacity is required for process ing. But as compute capacity grows, the link between the compute nodes and the storage becomes a bottleneck. At that point, one could invest in higher performance but more expensive networks (e.g., 10 gigabit Ethernet) or special purpose interconnects such as InfiniBand (even more expensive). In most cases, this is not a cost-effective solution, as the price of networking equipment in creases non-linearly with performance (e.g., a switch with ten times the capac ity is usually more than ten times more expensive). Alternatively, one could abandon the separation of computation and storage as distinct components in a cluster. The distributed file system (DFS) that underlies MapReduce adopts exactly this approach. The Google File System (GFS) [57] supports Google’s proprietary implementation of MapReduce; in the open-source world, HDFS (Hadoop Distributed File System) is an open-source implementation of GFS that supports Hadoop. Although MapReduce doesn’t necessarily require the distributed file system, it is difficult to realize many of the advantages of the programming model without a storage substrate that behaves much like the DFS.14

Of course, distributed file systems are not new [74, 32, 7, 147, 133]. The MapReduce distributed file system builds on previous work but is specifically adapted to large-data processing workloads, and therefore departs from previ ous architectures in certain respects (see discussion by Ghemawat et al. [57] in the original GFS paper.). The main idea is to divide user data into blocks and replicate those blocks across the local disks of nodes in the cluster. Block ing data, of course, is not a new idea, but DFS blocks are significantly larger than block sizes in typical single-machine file systems (64 MB by default). The distributed file system adopts a master–slave architecture in which the mas ter maintains the file namespace (metadata, directory structure, file to block mapping, location of blocks, and access permissions) and the slaves manage the actual data blocks. In GFS, the master is called the GFS master, and the slaves are called GFS chunkservers. In Hadoop, the same roles are filled by the

14However, there is evidence that existing POSIX-based distributed cluster file systems (e.g., GPFS or PVFS) can serve as a replacement for HDFS, when properly tuned or modified for MapReduce workloads [146, 6]. This, however, remains an experimental use case.

**Application**

(file name, block id)

**HDFS namenode** /foo/bar

HDFS Client

(block id, block location)

File namespace

instructions to datanode

block 3df2

(block id, byte range) block data

datanode state

**HDFS datanode**

Linux file system

…

**HDFS datanode** Linux file system …

Figure 2.4: The architecture of HDFS. The namenode (master) is responsible for maintaining the file namespace and directing clients to datanodes (slaves) that actually hold data blocks containing user data.

Figure 2.5: The architecture of HDFS. The namenode (master) is responsible for maintaining the le namespace and direcPng clients to datanodes (slaves) that actually hold data blocks namenode and datanodes, respectively.15 This book adopts the Hadoop termi

containing user data.

nology, although for most basic file operations GFS and HDFS work much the same way. The architecture of HDFS is shown in Figure 2.4, redrawn from a similar diagram describing GFS [57].

In HDFS, an application client wishing to read a file (or a portion thereof) must first contact the namenode to determine where the actual data is stored. In response to the client request, the namenode returns the relevant block id and the location where the block is held (i.e., which datanode). The client then contacts the datanode to retrieve the data. Blocks are themselves stored on standard single-machine file systems, so HDFS lies on top of the standard OS stack (e.g., Linux). An important feature of the design is that data is never moved through the namenode. Instead, all data transfer occurs directly between clients and datanodes; communications with the namenode only in volves transfer of metadata.

By default, HDFS stores three separate copies of each data block to en sure both reliability, availability, and performance. In large clusters, the three replicas are spread across different physical racks, so HDFS is resilient towards two common failure scenarios: individual datanode crashes and failures in net working equipment that bring an entire rack offline. Replicating blocks across physical machines also increases opportunities to co-locate data and processing in the scheduling of MapReduce jobs, since multiple copies yield more oppor tunities to exploit locality. The namenode is in periodic communication with the datanodes to ensure proper replication of all the blocks: if there aren’t

15To be precise, namenode and datanode may refer to physical machines in a cluster, or they may refer to daemons running on those machines providing the relevant services.

enough replicas (e.g., due to disk or machine failures or to connectivity losses due to networking equipment failures), the namenode directs the creation of additional copies;16 if there are too many replicas (e.g., a repaired node rejoins the cluster), extra copies are discarded.

To create a new file and write data to HDFS, the application client first contacts the namenode, which updates the file namespace after checking per missions and making sure the file doesn’t already exist. The namenode allocates a new block on a suitable datanode, and the application is directed to stream data directly to it. From the initial datanode, data is further propagated to additional replicas. In the most recent release of Hadoop as of this writing (release 0.20.2), files are immutable—they cannot be modified after creation. There are current plans to officially support file appends in the near future, which is a feature already present in GFS.

In summary, the HDFS namenode has the following responsibilities:

*•* Namespace management. The namenode is responsible for maintaining the file namespace, which includes metadata, directory structure, file to block mapping, location of blocks, and access permissions. These data are held in memory for fast access and all mutations are persistently logged.

*•* Coordinating file operations. The namenode directs application clients to datanodes for read operations, and allocates blocks on suitable datanodes for write operations. All data transfers occur directly between clients and datanodes. When a file is deleted, HDFS does not immediately reclaim the available physical storage; rather, blocks are lazily garbage collected.

*•* Maintaining overall health of the file system. The namenode is in pe riodic contact with the datanodes via heartbeat messages to ensure the integrity of the system. If the namenode observes that a data block is under-replicated (fewer copies are stored on datanodes than the desired replication factor), it will direct the creation of new replicas. Finally, the namenode is also responsible for rebalancing the file system.17 During the course of normal operations, certain datanodes may end up holding more blocks than others; rebalancing involves moving blocks from datanodes with more blocks to datanodes with fewer blocks. This leads to better load balancing and more even disk utilization.

Since GFS and HDFS were specifically designed to support Google’s propri etary and the open-source implementation of MapReduce, respectively, they were designed with a number of assumptions about the operational environ ment, which in turn influenced the design of the systems. Understanding these choices is critical to designing effective MapReduce algorithms:

16Note that the namenode coordinates the replication process, but data transfer occurs directly from datanode to datanode.

17In Hadoop, this is a manually-invoked process.

*•* The file system stores a relatively modest number of large files. The definition of “modest” varies by the size of the deployment, but in HDFS multi-gigabyte files are common (and even encouraged). There are several reasons why lots of small files are to be avoided. Since the namenode must hold all file metadata in memory, this presents an upper bound on both the number of files and blocks that can be supported.18 Large multi-block files represent a more efficient use of namenode memory than many single-block files (each of which consumes less space than a single block size). In addition, mappers in a MapReduce job use individual files as a basic unit for splitting input data. At present, there is no default mechanism in Hadoop that allows a mapper to process multiple files. As a result, mapping over many small files will yield as many map tasks as there are files. This results in two potential problems: first, the startup costs of mappers may become significant compared to the time spent actually processing input key-value pairs; second, this may result in an excessive amount of across-the-network copy operations during the “shuffle and sort” phase (recall that a MapReduce job with *m* mappers and *r* reducers involves up to *m × r* distinct copy operations).

*•* Workloads are batch oriented, dominated by long streaming reads and large sequential writes. As a result, high sustained bandwidth is more important than low latency. This exactly describes the nature of Map Reduce jobs, which are batch operations on large amounts of data. Due to the common-case workload, both HDFS and GFS do not implement any form of data caching.19

*•* Applications are aware of the characteristics of the distributed file system. Neither HDFS nor GFS present a general POSIX-compliant API, but rather support only a subset of possible file operations. This simplifies the design of the distributed file system, and in essence pushes part of the data management onto the end application. One rationale for this decision is that each application knows best how to handle data specific to that application, for example, in terms of resolving inconsistent states and optimizing the layout of data structures.

*•* The file system is deployed in an environment of cooperative users. There is no discussion of security in the original GFS paper, but HDFS explic itly assumes a datacenter environment where only authorized users have access. File permissions in HDFS are only meant to prevent unintended operations and can be easily circumvented.20

*•* The system is built from unreliable but inexpensive commodity com ponents. As a result, failures are the norm rather than the exception.

18According to Dhruba Borthakur in a post to the Hadoop mailing list on 6/8/2008, each block in HDFS occupies about 150 bytes of memory on the namenode. 19However, since the distributed file system is built on top of a standard operating system such as Linux, there is still OS-level caching.

20However, there are existing plans to integrate Kerberos into Hadoop/HDFS.

HDFS is designed around a number of self-monitoring and self-healing mechanisms to robustly cope with common failure modes.

Finally, some discussion is necessary to understand the single-master design of HDFS and GFS. It has been demonstrated that in large-scale distributed sys tems, simultaneously providing consistency, availability, and partition tolerance is impossible—this is Brewer’s so-called CAP Theorem [58]. Since partitioning is unavoidable in large-data systems, the real tradeoff is between consistency and availability. A single-master design trades availability for consistency and significantly simplifies implementation. If the master (HDFS namenode or GFS master) goes down, the entire file system becomes unavailable, which trivially guarantees that the file system will never be in an inconsistent state. An al ternative design might involve multiple masters that jointly manage the file namespace—such an architecture would increase availability (if one goes down, another can step in) at the cost of consistency, not to mention requiring a more complex implementation (cf. [4, 105]).

The single-master design of GFS and HDFS is a well-known weakness, since if the master goes offline, the entire file system and all MapReduce jobs run ning on top of it will grind to a halt. This weakness is mitigated in part by the lightweight nature of file system operations. Recall that no data is ever moved through the namenode and that all communication between clients and datanodes involve only metadata. Because of this, the namenode rarely is the bottleneck, and for the most part avoids load-induced crashes. In practice, this single point of failure is not as severe a limitation as it may appear—with diligent monitoring of the namenode, mean time between failure measured in months are not uncommon for production deployments. Furthermore, the Hadoop community is well-aware of this problem and has developed several reasonable workarounds—for example, a warm standby namenode that can be quickly switched over when the primary namenode fails. The open source en vironment and the fact that many organizations already depend on Hadoop for production systems virtually guarantees that more effective solutions will be developed over time.

2.6 Hadoop Cluster Architecture

Putting everything together, the architecture of a complete Hadoop cluster is shown in Figure 2.5. The HDFS namenode runs the namenode daemon. The job submission node runs the jobtracker, which is the single point of contact for a client wishing to execute a MapReduce job. The jobtracker monitors the progress of running MapReduce jobs and is responsible for coordinating the execution of the mappers and reducers. Typically, these services run on two separate machines, although in smaller clusters they are often co-located. The bulk of a Hadoop cluster consists of slave nodes (only three of which are shown in the figure) that run both a tasktracker, which is responsible for actually running user code, and a datanode daemon, for serving HDFS data.

**namenode**

**namenode daemon**

**job submission node jobtracker**

**tasktracker**

**datanode daemon** Linux file system …

**slave node**

**tasktracker**

**datanode daemon** Linux file system …

**slave node**

**tasktracker**

**datanode daemon** Linux file system …

**slave node**

Figure 2.5: Architecture of a complete Hadoop cluster, which consists of three separate components: the HDFS master (called the namenode), the job sub Figure 2.6: Architecture of a complete Hadoop cluster, which consists of three separate mission node (called the jobtracker), and many slave nodes (three shown here). components: the HDFS master (called the namenode), the job submission node (called the

Each of the slave nodes runs a tasktracker for executing map and reduce tasks jobtracker), and many slave nodes (three shown here). Each of the slave nodes runs a

and a datanode daemon for serving HDFS data.

tasktracker for execuPng map and reduce tasks and a datanode daemon for serving HDFS data.

A Hadoop MapReduce job is divided up into a number of map tasks and reduce tasks. Tasktrackers periodically send heartbeat messages to the job tracker that also doubles as a vehicle for task allocation. If a tasktracker is available to run tasks (in Hadoop parlance, has empty task slots), the return acknowledgment of the tasktracker heartbeat contains task allocation informa tion. The number of reduce tasks is equal to the number of reducers specified by the programmer. The number of map tasks, on the other hand, depends on many factors: the number of mappers specified by the programmer serves as a hint to the execution framework, but the actual number of tasks depends on both the number of input files and the number of HDFS data blocks occu pied by those files. Each map task is assigned a sequence of input key-value pairs, called an input split in Hadoop. Input splits are computed automatically and the execution framework strives to align them to HDFS block boundaries so that each map task is associated with a single data block. In scheduling map tasks, the jobtracker tries to take advantage of data locality—if possi ble, map tasks are scheduled on the slave node that holds the input split, so that the mapper will be processing local data. The alignment of input splits with HDFS block boundaries simplifies task scheduling. If it is not possible to run a map task on local data, it becomes necessary to stream input key-value pairs across the network. Since large clusters are organized into racks, with far greater intra-rack bandwidth than inter-rack bandwidth, the execution frame work strives to at least place map tasks on a rack which has a copy of the data block.

Although conceptually in MapReduce one can think of the mapper being

applied to all input key-value pairs and the reducer being applied to all values associated with the same key, actual job execution is a bit more complex. In Hadoop, mappers are Java objects with a Map method (among others). A mapper object is instantiated for every map task by the tasktracker. The life cycle of this object begins with instantiation, where a hook is provided in the API to run programmer-specified code. This means that mappers can read in “side data”, providing an opportunity to load state, static data sources, dic tionaries, etc. After initialization, the Map method is called (by the execution framework) on all key-value pairs in the input split. Since these method calls occur in the context of the same Java object, it is possible to preserve state across multiple input key-value pairs within the same map task—this is an im portant property to exploit in the design of MapReduce algorithms, as we will see in the next chapter. After all key-value pairs in the input split have been processed, the mapper object provides an opportunity to run programmer specified termination code. This, too, will be important in the design of Map Reduce algorithms.

The actual execution of reducers is similar to that of the mappers. Each reducer object is instantiated for every reduce task. The Hadoop API provides hooks for programmer-specified initialization and termination code. After ini tialization, for each intermediate key in the partition (defined by the parti tioner), the execution framework repeatedly calls the Reduce method with an intermediate key and an iterator over all values associated with that key. The programming model also guarantees that intermediate keys will be presented to the Reduce method in sorted order. Since this occurs in the context of a single object, it is possible to preserve state across multiple intermediate keys (and associated values) within a single reduce task. Once again, this property is critical in the design of MapReduce algorithms and will be discussed in the next chapter.

2.7 Summary

This chapter provides a basic overview of the MapReduce programming model, starting with its roots in functional programming and continuing with a descrip tion of mappers, reducers, partitioners, and combiners. Significant attention is also given to the underlying distributed file system, which is a tightly-integrated component of the MapReduce environment. Given this basic understanding, we now turn our attention to the design of MapReduce algorithms.

Chapter 3

Basic MapReduce Algorithm Design

A large part of the power of MapReduce comes from its simplicity: in addition to preparing the input data, the programmer needs only to implement the map per, the reducer, and optionally, the combiner and the partitioner. All other aspects of execution are handled transparently by the execution framework— on clusters ranging from a single node to a few thousand nodes, over datasets ranging from gigabytes to petabytes. However, this also means that any con ceivable algorithm that a programmer wishes to develop must be expressed in terms of a small number of rigidly-defined components that must fit to gether in very specific ways. It may not appear obvious how a multitude of algorithms can be recast into this programming model. The purpose of this chapter is to provide, primarily through examples, a guide to MapReduce al gorithm design. These examples illustrate what can be thought of as “design patterns” for MapReduce, which instantiate arrangements of components and specific techniques designed to handle frequently-encountered situations across a variety of problem domains. Two of these design patterns are used in the scalable inverted indexing algorithm we’ll present later in Chapter 4; concepts presented here will show up again in Chapter 5 (graph processing) and Chap ter 7 (expectation-maximization algorithms).

Synchronization is perhaps the most tricky aspect of designing MapReduce algorithms (or for that matter, parallel and distributed algorithms in general). Other than embarrassingly-parallel problems, processes running on separate nodes in a cluster must, at some point in time, come together—for example, to distribute partial results from nodes that produced them to the nodes that will consume them. Within a single MapReduce job, there is only one opportu nity for cluster-wide synchronization—during the shuffle and sort stage where intermediate key-value pairs are copied from the mappers to the reducers and grouped by key. Beyond that, mappers and reducers run in isolation without any mechanisms for direct communication. Furthermore, the programmer has little control over many aspects of execution, for example:

39

*• Where* a mapper or reducer runs (i.e., on which node in the cluster). *• When* a mapper or reducer begins or finishes.

*• Which* input key-value pairs are processed by a specific mapper. *• Which* intermediate key-value pairs are processed by a specific reducer.

Nevertheless, the programmer does have a number of techniques for controlling execution and managing the flow of data in MapReduce. In summary, they are:

1. The ability to construct complex data structures as keys and values to store and communicate partial results.

2. The ability to execute user-specified initialization code at the beginning of a map or reduce task, and the ability to execute user-specified termination code at the end of a map or reduce task.

3. The ability to preserve state in both mappers and reducers across multiple input or intermediate keys.

4. The ability to control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys.

5. The ability to control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer.

It is important to realize that many algorithms cannot be easily expressed as a single MapReduce job. One must often decompose complex algorithms into a sequence of jobs, which requires orchestrating data so that the output of one job becomes the input to the next. Many algorithms are iterative in nature, re quiring repeated execution until some convergence criteria—graph algorithms in Chapter 5 and expectation-maximization algorithms in Chapter 7 behave in exactly this way. Often, the convergence check itself cannot be easily ex pressed in MapReduce. The standard solution is an external (non-MapReduce) program that serves as a “driver” to coordinate MapReduce iterations.

This chapter explains how various techniques to control code execution and data flow can be applied to design algorithms in MapReduce. The focus is both on scalability—ensuring that there are no inherent bottlenecks as algorithms are applied to increasingly larger datasets—and efficiency—ensuring that algo rithms do not needlessly consume resources and thereby reducing the cost of parallelization. The gold standard, of course, is linear scalability: an algorithm running on twice the amount of data should take only twice as long. Similarly, an algorithm running on twice the number of nodes should only take half as long.

The chapter is organized as follows:

*•* Section 3.1 introduces the important concept of local aggregation in Map Reduce and strategies for designing efficient algorithms that minimize the amount of partial results that need to be copied across the network. The proper use of combiners is discussed in detail, as well as the “in-mapper combining” design pattern.

*•* Section 3.2 uses the example of building word co-occurrence matrices on large text corpora to illustrate two common design patterns, which we dub “pairs” and “stripes”. These two approaches are useful in a large class of problems that require keeping track of joint events across a large number of observations.

*•* Section 3.3 shows how co-occurrence counts can be converted into relative frequencies using a pattern known as “order inversion”. The sequencing of computations in the reducer can be recast as a sorting problem, where pieces of intermediate data are sorted into exactly the order that is re quired to carry out a series of computations. Often, a reducer needs to compute an aggregate statistic on a set of elements before individual el ements can be processed. Normally, this would require two passes over the data, but with the “order inversion” design pattern, the aggregate statistic can be computed in the reducer before the individual elements are encountered. This may seem counter-intuitive: how can we compute an aggregate statistic on a set of elements before encountering elements of that set? As it turns out, clever sorting of special key-value pairs enables exactly this.

*•* Section 3.4 provides a general solution to secondary sorting, which is the problem of sorting values associated with a key in the reduce phase. We call this technique “value-to-key conversion”.

3.1 Local Aggregation

In the context of data-intensive distributed processing, the single most impor tant aspect of synchronization is the exchange of intermediate results, from the processes that produced them to the processes that will ultimately consume them. In a cluster environment, with the exception of embarrassingly-parallel problems, this necessarily involves transferring data over the network. Further more, in Hadoop, intermediate results are written to local disk before being sent over the network. Since network and disk latencies are relatively expen sive compared to other operations, reductions in the amount of intermediate data translate into increases in algorithmic efficiency. In MapReduce, local aggregation of intermediate results is one of the keys to efficient algorithms. Through use of the combiner and by taking advantage of the ability to preserve state across multiple inputs, it is often possible to substantially reduce both the number and size of key-value pairs that need to be shuffled from the mappers to the reducers.

Algorithm 3.1 Word count (repeated from Algorithm 2.1) The mapper emits an intermediate key-value pair for each word in a document. The reducer sums up all counts for each word.

1: class Mapper

2: method Map(docid *a,* doc *d*)

3: for all term *t ∈* doc *d* do

4: Emit(term *t,* count 1)

1: class Reducer

2: method Reduce(term *t,* counts [*c*1*, c*2*, . . .*])

3: *sum ←* 0

4: for all count *c ∈* counts [*c*1*, c*2*, . . .*] do

5: *sum ← sum* + *c*

6: Emit(term *t,* count *sum*)

Combiners and In-Mapper Combining

We illustrate various techniques for local aggregation using the simple word count example presented in Section 2.2. For convenience, Algorithm 3.1 repeats the pseudo-code of the basic algorithm, which is quite simple: the mapper emits an intermediate key-value pair for each term observed, with the term itself as the key and a value of one; reducers sum up the partial counts to arrive at the final count.

The first technique for local aggregation is the combiner, already discussed in Section 2.4. Combiners provide a general mechanism within the Map Reduce framework to reduce the amount of intermediate data generated by the mappers—recall that they can be understood as “mini-reducers” that pro cess the output of mappers. In this example, the combiners aggregate term counts across the documents processed by each map task. This results in a re duction in the number of intermediate key-value pairs that need to be shuffled across the network—from the order of *total* number of terms in the collection to the order of the number of *unique* terms in the collection.1

An improvement on the basic algorithm is shown in Algorithm 3.2 (the mapper is modified but the reducer remains the same as in Algorithm 3.1 and therefore is not repeated). An associative array (i.e., Map in Java) is introduced inside the mapper to tally up term counts within a single document: instead of emitting a key-value pair for each term in the document, this version emits a

1More precisely, if the combiners take advantage of all opportunities for local aggregation, the algorithm would generate at most *m × V* intermediate key-value pairs, where *m* is the number of mappers and *V* is the vocabulary size (number of unique terms in the collection), since every term could have been observed in every mapper. However, there are two additional factors to consider. Due to the Zipfian nature of term distributions, most terms will not be observed by most mappers (for example, terms that occur only once will by definition only be observed by one mapper). On the other hand, combiners in Hadoop are treated as *optional* optimizations, so there is no guarantee that the execution framework will take advantage of all opportunities for partial aggregation.

Algorithm 3.2 Word count mapper using associative arrays 1: class Mapper

2: method Map(docid *a,* doc *d*)

3: *H ←* new AssociativeArray

4: for all term *t ∈* doc *d* do

5: *H{t} ← H{t}* + 1 *.* Tally counts for entire document 6: for all term *t ∈ H* do

7: Emit(term *t,* count *H{t}*)

Algorithm 3.3 Word count mapper using the“in-mapper combining” 1: class Mapper

2: method Initialize

3: *H ←* new AssociativeArray

4: method Map(docid *a,* doc *d*)

5: for all term *t ∈* doc *d* do

6: *H{t} ← H{t}* + 1 *.* Tally counts *across* documents 7: method Close

8: for all term *t ∈ H* do

9: Emit(term *t,* count *H{t}*)

key-value pair for each *unique* term in the document. Given that some words appear frequently within a document (for example, a document about dogs is likely to have many occurrences of the word “dog”), this can yield substantial savings in the number of intermediate key-value pairs emitted, especially for long documents.

This basic idea can be taken one step further, as illustrated in the variant of the word count algorithm in Algorithm 3.3 (once again, only the mapper is modified). The workings of this algorithm critically depends on the details of how map and reduce tasks in Hadoop are executed, discussed in Section 2.6. Recall, a (Java) mapper object is created for each map task, which is responsible for processing a block of input key-value pairs. Prior to processing any input key-value pairs, the mapper’s Initialize method is called, which is an API hook for user-specified code. In this case, we initialize an associative array for holding term counts. Since it is possible to preserve state across multiple calls of the Map method (for each input key-value pair), we can continue to accumulate partial term counts in the associative array *across* multiple documents, and emit key-value pairs only when the mapper has processed all documents. That is, emission of intermediate data is deferred until the Close method in the pseudo-code. Recall that this API hook provides an opportunity to execute user-specified code *after* the Map method has been applied to all input key value pairs of the input data split to which the map task was assigned.

With this technique, we are in essence incorporating combiner functionality directly inside the mapper. There is no need to run a separate combiner,

since all opportunities for local aggregation are already exploited.2 This is a sufficiently common design pattern in MapReduce that it’s worth giving it a name, “in-mapper combining”, so that we can refer to the pattern more conveniently throughout the book. We’ll see later on how this pattern can be applied to a variety of problems. There are two main advantages to using this design pattern:

First, it provides control over when local aggregation occurs and how it exactly takes place. In contrast, the semantics of the combiner is underspecified in MapReduce. For example, Hadoop makes no guarantees on how many times the combiner is applied, or that it is even applied at all. The combiner is provided as a semantics-preserving optimization to the execution framework, which has the *option* of using it, perhaps multiple times, or not at all (or even in the reduce phase). In some cases (although not in this particular example), such indeterminism is unacceptable, which is exactly why programmers often choose to perform their own local aggregation in the mappers.

Second, in-mapper combining will typically be more efficient than using actual combiners. One reason for this is the additional overhead associated with actually materializing the key-value pairs. Combiners reduce the amount of intermediate data that is shuffled across the network, but don’t actually reduce the number of key-value pairs that are emitted by the mappers in the first place. With the algorithm in Algorithm 3.2, intermediate key-value pairs are still generated on a per-document basis, only to be “compacted” by the combiners. This process involves unnecessary object creation and destruction (garbage collection takes time), and furthermore, object serialization and dese rialization (when intermediate key-value pairs fill the in-memory buffer holding map outputs and need to be temporarily spilled to disk). In contrast, with in mapper combining, the mappers will generate only those key-value pairs that need to be shuffled across the network to the reducers.

There are, however, drawbacks to the in-mapper combining pattern. First, it breaks the functional programming underpinnings of MapReduce, since state is being preserved across multiple input key-value pairs. Ultimately, this isn’t a big deal, since pragmatic concerns for efficiency often trump theoretical “pu rity”, but there are practical consequences as well. Preserving state across multiple input instances means that algorithmic behavior may depend on the order in which input key-value pairs are encountered. This creates the poten tial for ordering-dependent bugs, which are difficult to debug on large datasets in the general case (although the correctness of in-mapper combining for word count is easy to demonstrate). Second, there is a fundamental scalability bot tleneck associated with the in-mapper combining pattern. It critically depends on having sufficient memory to store intermediate results until the mapper has completely processed all key-value pairs in an input split. In the word count example, the memory footprint is bound by the vocabulary size, since it is

2Leaving aside the minor complication that in Hadoop, combiners can be run in the reduce phase also (when merging intermediate key-value pairs from different map tasks). However, in practice it makes almost no difference either way.

theoretically possible that a mapper encounters every term in the collection. Heap’s Law, a well-known result in information retrieval, accurately models the growth of vocabulary size as a function of the collection size—the somewhat surprising fact is that the vocabulary size never stops growing.3 Therefore, the algorithm in Algorithm 3.3 will scale only up to a point, beyond which the associative array holding the partial term counts will no longer fit in memory.4

One common solution to limiting memory usage when using the in-mapper combining technique is to “block” input key-value pairs and “flush” in-memory data structures periodically. The idea is simple: instead of emitting intermedi ate data only after *every* key-value pair has been processed, emit partial results after processing every *n* key-value pairs. This is straightforwardly implemented with a counter variable that keeps track of the number of input key-value pairs that have been processed. As an alternative, the mapper could keep track of its own memory footprint and flush intermediate key-value pairs once memory usage has crossed a certain threshold. In both approaches, either the block size or the memory usage threshold needs to be determined empirically: with too large a value, the mapper may run out of memory, but with too small a value, opportunities for local aggregation may be lost. Furthermore, in Hadoop physical memory is split between multiple tasks that may be running on a node concurrently; these tasks are all competing for finite resources, but since the tasks are not aware of each other, it is difficult to coordinate resource consump tion effectively. In practice, however, one often encounters diminishing returns in performance gains with increasing buffer sizes, such that it is not worth the effort to search for an *optimal* buffer size (personal communication, Jeff Dean).

In MapReduce algorithms, the extent to which efficiency can be increased through local aggregation depends on the size of the intermediate key space, the distribution of keys themselves, and the number of key-value pairs that are emitted by each individual map task. Opportunities for aggregation, after all, come from having multiple values associated with the same key (whether one uses combiners or employs the in-mapper combining pattern). In the word count example, local aggregation is effective because many words are encoun tered multiple times within a map task. Local aggregation is also an effective technique for dealing with reduce stragglers (see Section 2.3) that result from a highly-skewed (e.g., Zipfian) distribution of values associated with interme diate keys. In our word count example, we do not filter frequently-occurring words: therefore, without local aggregation, the reducer that’s responsible for computing the count of ‘the’ will have a lot more work to do than the typical reducer, and therefore will likely be a straggler. With local aggregation (either

3In more detail, Heap’s Law relates the vocabulary size *V* to the collection size as follows: *V* = *kTb*, where *T* is the number of tokens in the collection. Typical values of the parameters *k* and *b* are: 30 *≤ k ≤* 100 and *b ∼* 0*.*5 ([101], p. 81).

4A few more details: note what matters is that the partial term counts encountered within particular *input split* fits into memory. However, as collection sizes increase, one will often want to increase the input split size to limit the growth of the number of map tasks (in order to reduce the number of distinct copy operations necessary to shuffle intermediate data over the network).

Algorithm 3.4 Compute the mean of values associated with the same key 1: class Mapper

2: method Map(string *t,* integer *r*)

3: Emit(string *t,* integer *r*)

1: class Reducer

2: method Reduce(string *t,* integers [*r*1*, r*2*, . . .*])

3: *sum ←* 0

4: *cnt ←* 0

5: for all integer *r ∈* integers [*r*1*, r*2*, . . .*] do

6: *sum ← sum* + *r*

7: *cnt ← cnt* + 1

8: *ravg ← sum/cnt*

9: Emit(string *t,* integer *ravg*)

combiners or in-mapper combining), we substantially reduce the number of values associated with frequently-occurring terms, which alleviates the reduce straggler problem.

Algorithmic Correctness with Local Aggregation

Although use of combiners can yield dramatic reductions in algorithm running time, care must be taken in applying them. Since combiners in Hadoop are viewed as optional optimizations, the correctness of the algorithm cannot de pend on computations performed by the combiner or depend on them even being run at all. In any MapReduce program, the reducer input key-value type must match the mapper output key-value type: this implies that the combiner input *and* output key-value types must match the mapper output key-value type (which is the same as the reducer input key-value type). In cases where the reduce computation is both commutative and associative, the reducer can also be used (unmodified) as the combiner (as is the case with the word count example). In the general case, however, combiners and reducers are not inter changeable.

Consider a simple example: we have a large dataset where input keys are strings and input values are integers, and we wish to compute the mean of all integers associated with the same key (rounded to the nearest integer). A real-world example might be a large user log from a popular website, where keys represent user ids and values represent some measure of activity such as elapsed time for a particular session—the task would correspond to computing the mean session length on a per-user basis, which would be useful for under standing user demographics. Algorithm 3.4 shows the pseudo-code of a simple algorithm for accomplishing this task that does not involve combiners. We use an identity mapper, which simply passes all input key-value pairs to the reducers (appropriately grouped and sorted). The reducer keeps track of the running sum and the number of integers encountered. This information is used

to compute the mean once all values are processed. The mean is then emitted as the output value in the reducer (with the input string as the key). This algorithm will indeed work, but suffers from the same drawbacks as the basic word count algorithm in Algorithm 3.1: it requires shuffling all key-value pairs from mappers to reducers across the network, which is highly inefficient. Unlike in the word count example, the reducer cannot be used as a combiner in this case. Consider what would happen if we did: the combiner would compute the mean of an arbitrary subset of values associated with the same key, and the reducer would compute the mean of those values. As a concrete example, we know that:

Mean(1*,* 2*,* 3*,* 4*,* 5) *6*= Mean(Mean(1*,* 2)*,* Mean(3*,* 4*,* 5)) (3.1)

In general, the mean of means of arbitrary subsets of a set of numbers is not the same as the mean of the set of numbers. Therefore, this approach would not produce the correct result.5

So how might we properly take advantage of combiners? An attempt is shown in Algorithm 3.5. The mapper remains the same, but we have added a combiner that partially aggregates results by computing the numeric compo nents necessary to arrive at the mean. The combiner receives each string and the associated list of integer values, from which it computes the sum of those values and the number of integers encountered (i.e., the count). The sum and count are packaged into a pair, and emitted as the output of the combiner, with the same string as the key. In the reducer, pairs of partial sums and counts can be aggregated to arrive at the mean. Up until now, all keys and values in our algorithms have been primitives (string, integers, etc.). However, there are no prohibitions in MapReduce for more complex types,6 and, in fact, this represents a key technique in MapReduce algorithm design that we introduced at the beginning of this chapter. We will frequently encounter complex keys and values throughput the rest of this book.

Unfortunately, this algorithm will not work. Recall that combiners must have the same input and output key-value type, which also must be the same as the mapper output type and the reducer input type. This is clearly not the case. To understand why this restriction is necessary in the programming model, remember that combiners are optimizations that cannot change the correctness of the algorithm. So let us remove the combiner and see what happens: the output value type of the mapper is integer, so the reducer expects to receive a list of integers as values. But the reducer actually expects a list of pairs! The correctness of the algorithm is contingent on the combiner running on the output of the mappers, and more specifically, that the combiner is run exactly once. Recall from our previous discussion that Hadoop makes no

5There is, however, one special case in which using reducers as combiners *would* produce the correct result: if each combiner computed the mean of equal-size subsets of the values. However, since such fine-grained control over the combiners is impossible in MapReduce, such a scenario is highly unlikely.

6In Hadoop, either custom types or types defined using a library such as Protocol Buffers, Thrift, or Avro.

Algorithm 3.5 Compute the mean of values associated with the same key Note that this algorithm is incorrect. The mismatch between combiner input and output key-value types violates the MapReduce programming model. 1: class Mapper

2: method Map(string *t,* integer *r*)

3: Emit(string *t,* integer *r*)

1: class Combiner

2: method Combine(string *t,* integers [*r*1*, r*2*, . . .*])

3: *sum ←* 0

4: *cnt ←* 0

5: for all integer *r ∈* integers [*r*1*, r*2*, . . .*] do

6: *sum ← sum* + *r*

7: *cnt ← cnt* + 1

8: Emit(string *t,* pair (*sum, cnt*)) *.* Separate sum and count

1: class Reducer

2: method Reduce(string *t,* pairs [(*s*1*, c*1)*,*(*s*2*, c*2)*. . .*]) 3: *sum ←* 0

4: *cnt ←* 0

5: for all pair (*s, c*) *∈* pairs [(*s*1*, c*1)*,*(*s*2*, c*2)*. . .*] do

6: *sum ← sum* + *s*

7: *cnt ← cnt* + *c*

8: *ravg ← sum/cnt*

9: Emit(string *t,* integer *ravg*)

guarantees on how many times combiners are called; it could be zero, one, or multiple times. This violates the MapReduce programming model. Another stab at the solution is shown in Algorithm 3.6, and this time, the algorithm is correct. In the mapper we emit as the value a pair consisting of the integer and one—this corresponds to a partial count over one instance. The combiner separately aggregates the partial sums and the partial counts (as before), and emits pairs with updated sums and counts. The reducer is similar to the combiner, except that the mean is computed at the end. In essence, this algorithm transforms a non-associative operation (mean of num bers) into an associative operation (element-wise sum of a pair of numbers, with an additional division at the very end).

Let us verify the correctness of this algorithm by repeating the previous exercise: What would happen if no combiners were run? With no combiners, the mappers would send pairs (as values) directly to the reducers. There would be as many intermediate pairs as there were input key-value pairs, and each of those would consist of an integer and one. The reducer would still arrive at the correct sum and count, and hence the mean would be correct. Now add in the combiners: the algorithm would remain correct, no matter how many times they run, since the combiners merely aggregate partial sums and counts

Algorithm 3.6 Compute the mean of values associated with the same key This algorithm correctly takes advantage of combiners.

1: class Mapper

2: method Map(string *t,* integer *r*)

3: Emit(string *t,* pair (*r,* 1))

1: class Combiner

2: method Combine(string *t,* pairs [(*s*1*, c*1)*,*(*s*2*, c*2)*. . .*]) 3: *sum ←* 0

4: *cnt ←* 0

5: for all pair (*s, c*) *∈* pairs [(*s*1*, c*1)*,*(*s*2*, c*2)*. . .*] do

6: *sum ← sum* + *s*

7: *cnt ← cnt* + *c*

8: Emit(string *t,* pair (*sum, cnt*))

1: class Reducer

2: method Reduce(string *t,* pairs [(*s*1*, c*1)*,*(*s*2*, c*2)*. . .*]) 3: *sum ←* 0

4: *cnt ←* 0

5: for all pair (*s, c*) *∈* pairs [(*s*1*, c*1)*,*(*s*2*, c*2)*. . .*] do

6: *sum ← sum* + *s*

7: *cnt ← cnt* + *c*

8: *ravg ← sum/cnt*

9: Emit(string *t,* integer *ravg*)

to pass along to the reducers. Note that although the output key-value type of the combiner must be the same as the input key-value type of the reducer, the reducer can emit final key-value pairs of a different type.

Finally, in Algorithm 3.7, we present an even more efficient algorithm that exploits the in-mapper combining pattern. Inside the mapper, the partial sums and counts associated with each string are held in memory across input key value pairs. Intermediate key-value pairs are emitted only after the entire input split has been processed; similar to before, the value is a pair consisting of the sum and count. The reducer is exactly the same as in Algorithm 3.6. Moving partial aggregation from the combiner directly into the mapper is subjected to all the tradeoffs and caveats discussed earlier this section, but in this case the memory footprint of the data structures for holding intermediate data is likely to be modest, making this variant algorithm an attractive option.

3.2 Pairs and Stripes

One common approach for synchronization in MapReduce is to construct com plex keys and values in such a way that data necessary for a computation are naturally brought together by the execution framework. We first touched on this technique in the previous section, in the context of “packaging” partial

Algorithm 3.7 Compute the mean of values associated with the same key This mapper illustrates the in-mapper combining design pattern. The reducer is the same as in Algorithm 3.6

1: class Mapper

2: method Initialize

3: *S ←* new AssociativeArray

4: *C ←* new AssociativeArray

5: method Map(string *t,* integer *r*)

6: *S{t} ← S{t}* + *r*

7: *C{t} ← C{t}* + 1

8: method Close

9: for all term *t ∈ S* do

10: Emit(term *t,* pair (*S{t}, C{t}*))

sums and counts in a complex value (i.e., pair) that is passed from mapper to combiner to reducer. Building on previously published work [54, 94], this section introduces two common design patterns we have dubbed “pairs” and “stripes” that exemplify this strategy.

As a running example, we focus on the problem of building word co occurrence matrices from large corpora, a common task in corpus linguistics and statistical natural language processing. Formally, the co-occurrence ma trix of a corpus is a square *n × n* matrix where *n* is the number of unique words in the corpus (i.e., the vocabulary size). A cell *mij* contains the number of times word *wi* co-occurs with word *wj* within a specific context—a natural unit such as a sentence, paragraph, or a document, or a certain window of *m* words (where *m* is an application-dependent parameter). Note that the upper and lower triangles of the matrix are identical since co-occurrence is a symmet ric relation, though in the general case relations between words need not be symmetric. For example, a co-occurrence matrix *M* where *mij* is the count of how many times word *i* was immediately succeeded by word *j* would usually not be symmetric.

This task is quite common in text processing and provides the starting point to many other algorithms, e.g., for computing statistics such as pointwise mutual information [38], for unsupervised sense clustering [136], and more gen erally, a large body of work in lexical semantics based on distributional profiles of words, dating back to Firth [55] and Harris [69] in the 1950s and 1960s. The task also has applications in information retrieval (e.g., automatic thesaurus construction [137] and stemming [157]), and other related fields such as text mining. More importantly, this problem represents a specific instance of the task of estimating distributions of discrete joint events from a large number of observations, a very common task in statistical natural language processing for which there are nice MapReduce solutions. Indeed, concepts presented here are also used in Chapter 7 when we discuss expectation-maximization algorithms. Beyond text processing, problems in many application domains share sim-

ilar characteristics. For example, a large retailer might analyze point-of-sale transaction records to identify correlated product purchases (e.g., customers who buy *this* tend to also buy *that*), which would assist in inventory manage ment and product placement on store shelves. Similarly, an intelligence analyst might wish to identify associations between re-occurring financial transactions that are otherwise unrelated, which might provide a clue in thwarting terrorist activity. The algorithms discussed in this section could be adapted to tackle these related problems.

It is obvious that the space requirement for the word co-occurrence problem is *O*(*n*2), where *n* is the size of the vocabulary, which for real-world English corpora can be hundreds of thousands of words, or even billions of words in web scale collections.7 The computation of the word co-occurrence matrix is quite simple if the entire matrix fits into memory—however, in the case where the matrix is too big to fit in memory, a na¨ıve implementation on a single machine can be very slow as memory is paged to disk. Although compression techniques can increase the size of corpora for which word co-occurrence matrices can be constructed on a single machine, it is clear that there are inherent scalability limitations. We describe two MapReduce algorithms for this task that can scale to large corpora.

Pseudo-code for the first algorithm, dubbed the “pairs” approach, is shown in Algorithm 3.8. As usual, document ids and the corresponding contents make up the input key-value pairs. The mapper processes each input document and emits intermediate key-value pairs with each co-occurring word pair as the key and the integer one (i.e., the count) as the value. This is straightforwardly accomplished by two nested loops: the outer loop iterates over all words (the left element in the pair), and the inner loop iterates over all neighbors of the first word (the right element in the pair). The neighbors of a word can either be defined in terms of a sliding window or some other contextual unit such as a sentence. The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Thus, in this case the reducer simply sums up all the values associated with the same co-occurring word pair to arrive at the absolute count of the joint event in the corpus, which is then emitted as the final key-value pair. Each pair corresponds to a cell in the word co-occurrence matrix. This algorithm illustrates the use of complex keys in order to coordinate distributed computations.

An alternative approach, dubbed the “stripes” approach, is presented in Algorithm 3.9. Like the pairs approach, co-occurring word pairs are generated by two nested loops. However, the major difference is that instead of emitting intermediate key-value pairs for each co-occurring word pair, co-occurrence information is first stored in an associative array, denoted *H*. The mapper emits key-value pairs with words as keys and corresponding associative arrays

7The size of the vocabulary depends on the definition of a “word” and techniques (if any) for corpus pre-processing. One common strategy is to replace all rare words (below a certain frequency) with a “special” token such as *<*UNK*>* (which stands for “unknown”) to model out-of-vocabulary words. Another technique involves replacing numeric digits with #, such that 1.32 and 1.19 both map to the same token (#.##).

Algorithm 3.8 Compute word co-occurrence (“pairs” approach) 1: class Mapper

2: method Map(docid *a,* doc *d*)

3: for all term *w ∈* doc *d* do

4: for all term *u ∈* Neighbors(*w*) do

5: Emit(pair (*w, u*)*,* count 1) *.* Emit count for each co-occurrence

1: class Reducer

2: method Reduce(pair *p,* counts [*c*1*, c*2*, . . .*])

3: *s ←* 0

4: for all count *c ∈* counts [*c*1*, c*2*, . . .*] do

5: *s ← s* + *c .* Sum co-occurrence counts 6: Emit(pair *p,* count *s*)

Algorithm 3.9 Compute word co-occurrence (“stripes” approach) 1: class Mapper

2: method Map(docid *a,* doc *d*)

3: for all term *w ∈* doc *d* do

4: *H ←* new AssociativeArray

5: for all term *u ∈* Neighbors(*w*) do

6: *H{u} ← H{u}* + 1 *.* Tally words co-occurring with *w* 7: Emit(Term *w,* Stripe *H*)

1: class Reducer

2: method Reduce(term *w,* stripes [*H*1*, H*2*, H*3*, . . .*])

3: *Hf ←* new AssociativeArray

4: for all stripe *H ∈* stripes [*H*1*, H*2*, H*3*, . . .*] do

5: Sum(*Hf , H*) *.* Element-wise sum 6: Emit(term *w,*stripe *Hf* )

as values, where each associative array encodes the co-occurrence counts of the neighbors of a particular word (i.e., its context). The MapReduce execution framework guarantees that all associative arrays with the same key will be brought together in the reduce phase of processing. The reducer performs an element-wise sum of all associative arrays with the same key, accumulating counts that correspond to the same cell in the co-occurrence matrix. The final associative array is emitted with the same word as the key. In contrast to the pairs approach, each final key-value pair encodes a row in the co-occurrence matrix.

It is immediately obvious that the pairs algorithm generates an immense number of key-value pairs compared to the stripes approach. The stripes rep resentation is much more compact, since with pairs the left element is repeated for every co-occurring word pair. The stripes approach also generates fewer and shorter intermediate keys, and therefore the execution framework has less sorting to perform. However, values in the stripes approach are more complex, and come with more serialization and deserialization overhead than with the pairs approach.

Both algorithms can benefit from the use of combiners, since the respec tive operations in their reducers (addition and element-wise sum of associative arrays) are both commutative and associative. However, combiners with the stripes approach have more opportunities to perform local aggregation because the key space is the vocabulary—associative arrays can be merged whenever a word is encountered multiple times by a mapper. In contrast, the key space in the pairs approach is the cross of the vocabulary with itself, which is far larger—counts can be aggregated only when the same co-occurring word pair is observed multiple times by an individual mapper (which is less likely than observing multiple occurrences of a word, as in the stripes case).

For both algorithms, the in-mapper combining optimization discussed in the previous section can also be applied; the modification is sufficiently straightfor ward that we leave the implementation as an exercise for the reader. However, the above caveats remain: there will be far fewer opportunities for partial aggre gation in the pairs approach due to the sparsity of the intermediate key space. The sparsity of the key space also limits the effectiveness of in-memory combin ing, since the mapper may run out of memory to store partial counts before all documents are processed, necessitating some mechanism to periodically emit key-value pairs (which further limits opportunities to perform partial aggre gation). Similarly, for the stripes approach, memory management will also be more complex than in the simple word count example. For common terms, the associative array may grow to be quite large, necessitating some mechanism to periodically flush in-memory structures.

It is important to consider potential scalability bottlenecks of either al gorithm. The stripes approach makes the assumption that, at any point in time, each associative array is small enough to fit into memory—otherwise, memory paging will significantly impact performance. The size of the associa tive array is bounded by the vocabulary size, which is itself unbounded with respect to corpus size (recall the previous discussion of Heap’s Law). There-

fore, as the sizes of corpora increase, this will become an increasingly pressing issue—perhaps not for gigabyte-sized corpora, but certainly for terabyte-sized and petabyte-sized corpora that will be commonplace tomorrow. The pairs approach, on the other hand, does not suffer from this limitation, since it does not need to hold intermediate data in memory.

Given this discussion, which approach is faster? Here, we present previously published results [94] that empirically answered this question. We have imple mented both algorithms in Hadoop and applied them to a corpus of 2.27 million documents from the Associated Press Worldstream (APW) totaling 5.7 GB.8 Prior to working with Hadoop, the corpus was first preprocessed as follows: All XML markup was removed, followed by tokenization and stopword removal using standard tools from the Lucene search engine. All tokens were then re placed with unique integers for a more efficient encoding. Figure 3.1 compares the running time of the pairs and stripes approach on different fractions of the corpus, with a co-occurrence window size of two. These experiments were performed on a Hadoop cluster with 19 slave nodes, each with two single-core processors and two disks.

Results demonstrate that the stripes approach is much faster than the pairs approach: 666 seconds (*∼*11 minutes) compared to 3758 seconds (*∼*62 minutes) for the entire corpus (improvement by a factor of 5.7). The mappers in the pairs approach generated 2.6 billion intermediate key-value pairs totaling 31.2 GB. After the combiners, this was reduced to 1.1 billion key-value pairs, which quantifies the amount of intermediate data transferred across the network. In the end, the reducers emitted a total of 142 million final key-value pairs (the number of non-zero cells in the co-occurrence matrix). On the other hand, the mappers in the stripes approach generated 653 million intermediate key-value pairs totaling 48.1 GB. After the combiners, only 28.8 million key-value pairs remained. The reducers emitted a total of 1.69 million final key-value pairs (the number of rows in the co-occurrence matrix). As expected, the stripes approach provided more opportunities for combiners to aggregate intermediate results, thus greatly reducing network traffic in the shuffle and sort phase. Figure 3.1 also shows that both algorithms exhibit highly desirable scaling characteristics—linear in the amount of input data. This is confirmed by a linear regression applied to the running time data, which yields an *R*2 value close to one.

An additional series of experiments explored the scalability of the stripes approach along another dimension: the size of the cluster. These experiments were made possible by Amazon’s EC2 service, which allows users to rapidly provision clusters of varying sizes for limited durations (for more information, refer back to our discussion of utility computing in Section 1.1). Virtualized computational units in EC2 are called instances, and the user is charged only for the instance-hours consumed. Figure 3.2 (left) shows the running time of the stripes algorithm (on the same corpus, with same setup as before), on varying

8This was a subset of the English Gigaword corpus (version 3) distributed by the Lin guistic Data Consortium (LDC catalog number LDC2007T07).

) s

d

n

o

c

e

s

(

e

m

i

t

g

n

i

n

n

u

R

4000 3500 3000 2500 2000 1500 1000

500 0

"stripes" approach "pairs" approach

R2 = 0.999 R2 = 0.992

0 20 40 60 80 100

Percentage of the APW corpus

Figure 3.1: Running time of the “pairs” and “stripes” algorithms for computing word co-occurrence matrices on different fractions of the APW corpus. These experiments were performed on a Hadoop cluster with 19 slaves, each with two single-core processors and two disks.

cluster sizes, from 20 slave “small” instances all the way up to 80 slave “small” instances (along the *x* -axis). Running times are shown with solid squares. Figure 3.2 (right) recasts the same results to illustrate scaling characteristics. The circles plot the relative size and speedup of the EC2 experiments, with respect to the 20-instance cluster. These results show highly desirable linear scaling characteristics (i.e., doubling the cluster size makes the job twice as fast). This is confirmed by a linear regression with an *R*2 value close to one.

Viewed abstractly, the pairs and stripes algorithms represent two different approaches to counting co-occurring events from a large number of observa tions. This general description captures the gist of many algorithms in fields as diverse as text processing, data mining, and bioinformatics. For this rea son, these two design patterns are broadly useful and frequently observed in a variety of applications.

To conclude, it is worth noting that the pairs and stripes approaches rep resent endpoints along a continuum of possibilities. The pairs approach indi vidually records *each* co-occurring event, while the stripes approach records *all* co-occurring events with respect a conditioning event. A middle ground might be to record a subset of the co-occurring events with respect to a condition ing event. We might divide up the entire vocabulary into *b* buckets (e.g., via hashing), so that words co-occurring with *wi* would be divided into *b* smaller

5000

4000

)

s

d

n

3000

o

c

e

s

(

e

m

i

t

2000

g

n

i

n

n

u

R

1000

0

10 20 30 40 50 60 70 80 90 Size of EC2 cluster (number of slave instances)

4x

R2 = 0.997

p

u

de

e

p

s

e

v

it

a

le

R

3x

2x

1x

1x 2x 3x 4x Relative size of EC2 cluster

Figure 3.2: Running time of the stripes algorithm on the APW corpus with Hadoop clusters of different sizes from EC2 (left). Scaling characteristics (rel ative speedup) in terms of increasing Hadoop cluster size (right).

“sub-stripes”, associated with ten separate keys, (*wi,* 1)*,*(*wi,* 2)*. . .*(*wi, b*). This would be a reasonable solution to the memory limitations of the stripes ap proach, since each of the sub-stripes would be smaller. In the case of *b* = *|V |*, where *|V |* is the vocabulary size, this is equivalent to the pairs approach. In the case of *b* = 1, this is equivalent to the standard stripes approach.

3.3 Computing Relative Frequencies

Let us build on the pairs and stripes algorithms presented in the previous section and continue with our running example of constructing the word co occurrence matrix *M* for a large corpus. Recall that in this large square *n × n*

matrix, where *n* = *|V |* (the vocabulary size), cell *mij* contains the number of times word *wi* co-occurs with word *wj* within a specific context. The drawback of absolute counts is that it doesn’t take into account the fact that some words appear more frequently than others. Word *wi* may co-occur frequently with *wj* simply because one of the words is very common. A simple remedy is to convert absolute counts into relative frequencies, *f*(*wj |wi*). That is, what proportion of the time does *wj* appear in the context of *wi*? This can be computed using the following equation:

*f*(*wj |wi*) = *N*(*wi, wj* )

P

*w0 N*(*wi, w0*)(3.2)

Here, *N*(*·, ·*) indicates the number of times a particular co-occurring word pair is observed in the corpus. We need the count of the joint event (word co occurrence), divided by what is known as the marginal (the sum of the counts of the conditioning variable co-occurring with anything else).

Computing relative frequencies with the stripes approach is straightforward. In the reducer, counts of all words that co-occur with the conditioning variable (*wi*in the above example) are available in the associative array. Therefore, it

suffices to sum all those counts to arrive at the marginal (i.e., P*w0 N*(*wi, w0*)), and then divide all the joint counts by the marginal to arrive at the relative frequency for all words. This implementation requires minimal modification to the original stripes algorithm in Algorithm 3.9, and illustrates the use of com plex data structures to coordinate distributed computations in MapReduce. Through appropriate structuring of keys and values, one can use the Map Reduce execution framework to bring together all the pieces of data required to perform a computation. Note that, as with before, this algorithm also as sumes that each associative array fits into memory.

How might one compute relative frequencies with the pairs approach? In the pairs approach, the reducer receives (*wi, wj* ) as the key and the count as the value. From this alone it is not possible to compute *f*(*wj |wi*) since we do not have the marginal. Fortunately, as in the mapper, the reducer can preserve state across multiple keys. Inside the reducer, we can buffer in memory all the words that co-occur with *wi* and their counts, in essence building the associative array in the stripes approach. To make this work, we must define the sort order of the pair so that keys are first sorted by the left word, and then by the right word. Given this ordering, we can easily detect if all pairs associated with the word we are conditioning on (*wi*) have been encountered. At that point we can go back through the in-memory buffer, compute the relative frequencies, and then emit those results in the final key-value pairs.

There is one more modification necessary to make this algorithm work. We must ensure that all pairs with the same left word are sent to the same reducer. This, unfortunately, does not happen automatically: recall that the default partitioner is based on the hash value of the intermediate key, modulo the number of reducers. For a complex key, the raw byte representation is used to compute the hash value. As a result, there is no guarantee that, for

example, (dog, aardvark) and (dog, zebra) are assigned to the same reducer. To produce the desired behavior, we must define a custom partitioner that only pays attention to the left word. That is, the partitioner should partition based on the hash of the left word only.

This algorithm will indeed work, but it suffers from the same drawback as the stripes approach: as the size of the corpus grows, so does that vocabulary size, and at some point there will not be sufficient memory to store all co occurring words and their counts for the word we are conditioning on. For computing the co-occurrence matrix, the advantage of the pairs approach is that it doesn’t suffer from any memory bottlenecks. Is there a way to modify the basic pairs approach so that this advantage is retained?

As it turns out, such an algorithm is indeed possible, although it requires the coordination of several mechanisms in MapReduce. The insight lies in properly sequencing data presented to the reducer. If it were possible to some how compute (or otherwise obtain access to) the marginal in the reducer before processing the joint counts, the reducer could simply divide the joint counts by the marginal to compute the relative frequencies. The notion of “before” and “after” can be captured in the ordering of key-value pairs, which can be explicitly controlled by the programmer. That is, the programmer can define the sort order of keys so that data needed earlier is presented to the reducer be fore data that is needed later. However, we still need to compute the marginal counts. Recall that in the basic pairs algorithm, each mapper emits a key-value pair with the co-occurring word pair as the key. To compute relative frequen cies, we modify the mapper so that it additionally emits a “special” key of the form (*wi, ∗*), with a value of one, that represents the contribution of the word pair to the marginal. Through use of combiners, these partial marginal counts will be aggregated before being sent to the reducers. Alternatively, the in-mapper combining pattern can be used to even more efficiently aggregate marginal counts.

In the reducer, we must make sure that the special key-value pairs rep resenting the partial marginal contributions are processed before the normal key-value pairs representing the joint counts. This is accomplished by defining the sort order of the keys so that pairs with the special symbol of the form (*wi, ∗*) are ordered before any other key-value pairs where the left word is *wi*. In addition, as with before we must also properly define the partitioner to pay attention to only the left word in each pair. With the data properly sequenced, the reducer can directly compute the relative frequencies.

A concrete example is shown in Figure 3.3, which lists the sequence of key value pairs that a reducer might encounter. First, the reducer is presented with the special key (dog*, ∗*) and a number of values, each of which represents a par tial marginal contribution from the map phase (assume here either combiners or in-mapper combining, so the values represent partially aggregated counts). The

reducer accumulates these counts to arrive at the marginal, P*w0 N*(dog*, w0*). The reducer holds on to this value as it processes subsequent keys. After (dog*, ∗*), the reducer will encounter a series of keys representing joint counts; let’s say the first of these is the key (dog*,* aardvark). Associated with this key

key values

(dog*, ∗*) [6327, 8514, *. . .*] compute marginal:

P

*w0 N*(dog*, w0*) = 42908

(dog*,* aardvark) [2,1] *f*(aardvark*|*dog) = 3*/*42908 (dog*,* aardwolf) [1] *f*(aardwolf*|*dog) = 1*/*42908 . . .

(dog*,* zebra) [2,1,1,1] *f*(zebra*|*dog) = 5*/*42908 (doge*, ∗*) [682, *. . .*] compute marginal:

P

*w0 N*(doge*, w0*) = 1267

. . .

Figure 3.3: Example of the sequence of key-value pairs presented to the reducer in the pairs algorithm for computing relative frequencies. This illustrates the application of the order inversion design pattern.

will be a list of values representing partial joint counts from the map phase (two separate values in this case). Summing these counts will yield the final joint count, i.e., the number of times dog and aardvark co-occur in the entire collection. At this point, since the reducer already knows the marginal, sim ple arithmetic suffices to compute the relative frequency. All subsequent joint counts are processed in exactly the same manner. When the reducer encoun ters the next special key-value pair (doge*, ∗*), the reducer resets its internal state and starts to accumulate the marginal all over again. Observe that the memory requirement for this algorithm is minimal, since only the marginal (an integer) needs to be stored. No buffering of individual co-occurring word counts is necessary, and therefore we have eliminated the scalability bottleneck of the previous algorithm.

This design pattern, which we call “order inversion”, occurs surprisingly often and across applications in many domains. It is so named because through proper coordination, we can access the result of a computation in the reducer (for example, an aggregate statistic) before processing the data needed for that computation. The key insight is to convert the sequencing of computations into a sorting problem. In most cases, an algorithm requires data in some fixed order: by controlling how keys are sorted and how the key space is partitioned, we can present data to the reducer in the order necessary to perform the proper computations. This greatly cuts down on the amount of partial results that the reducer needs to hold in memory.

To summarize, the specific application of the order inversion design pattern for computing relative frequencies requires the following:

*•* Emitting a special key-value pair for each co-occurring word pair in the mapper to capture its contribution to the marginal.

*•* Controlling the sort order of the intermediate key so that the key-value pairs representing the marginal contributions are processed by the reducer before any of the pairs representing the joint word co-occurrence counts.

*•* Defining a custom partitioner to ensure that all pairs with the same left word are shuffled to the same reducer.

*•* Preserving state across multiple keys in the reducer to first compute the marginal based on the special key-value pairs and then dividing the joint counts by the marginals to arrive at the relative frequencies.

As we will see in Chapter 4, this design pattern is also used in inverted index construction to properly set compression parameters for postings lists.

3.4 Secondary Sorting

MapReduce sorts intermediate key-value pairs by the keys during the shuffle and sort phase, which is very convenient if computations inside the reducer rely on sort order (e.g., the order inversion design pattern described in the previous section). However, what if in addition to sorting by key, we also need to sort by value? Google’s MapReduce implementation provides built-in functionality for (optional) secondary sorting, which guarantees that values arrive in sorted order. Hadoop, unfortunately, does not have this capability built in.

Consider the example of sensor data from a scientific experiment: there are *m* sensors each taking readings on continuous basis, where *m* is potentially a large number. A dump of the sensor data might look something like the following, where *rx* after each timestamp represents the actual sensor readings (unimportant for this discussion, but may be a series of values, one or more complex records, or even raw bytes of images).

(*t*1*, m*1*, r*80521)

(*t*1*, m*2*, r*14209)

(*t*1*, m*3*, r*76042)

*...*

(*t*2*, m*1*, r*21823)

(*t*2*, m*2*, r*66508)

(*t*2*, m*3*, r*98347)

Suppose we wish to reconstruct the activity at each individual sensor over time. A MapReduce program to accomplish this might map over the raw data and emit the sensor id as the intermediate key, with the rest of each record as the value:

*m*1 *→* (*t*1*, r*80521)

This would bring all readings from the same sensor together in the reducer. However, since MapReduce makes no guarantees about the ordering of values associated with the same key, the sensor readings will not likely be in temporal order. The most obvious solution is to buffer all the readings in memory and then sort by timestamp before additional processing. However, it should be apparent by now that any in-memory buffering of data introduces a potential

scalability bottleneck. What if we are working with a high frequency sensor or sensor readings over a long period of time? What if the sensor readings themselves are large complex objects? This approach may not scale in these cases—the reducer would run out of memory trying to buffer all values associ ated with the same key.

This is a common problem, since in many applications we wish to first group together data one way (e.g., by sensor id), and then sort within the groupings another way (e.g., by time). Fortunately, there is a general purpose solution, which we call the “value-to-key conversion” design pattern. The basic idea is to move part of the value into the intermediate key to form a composite key, and let the MapReduce execution framework handle the sorting. In the above example, instead of emitting the sensor id as the key, we would emit the sensor id and the timestamp as a composite key:

(*m*1*, t*1) *→* (*r*80521)

The sensor reading itself now occupies the value. We must define the inter mediate key sort order to first sort by the sensor id (the left element in the pair) and then by the timestamp (the right element in the pair). We must also implement a custom partitioner so that all pairs associated with the same sensor are shuffled to the same reducer.

Properly orchestrated, the key-value pairs will be presented to the reducer in the correct sorted order:

(*m*1*, t*1) *→* [(*r*80521)]

(*m*1*, t*2) *→* [(*r*21823)]

(*m*1*, t*3) *→* [(*r*146925)]

*. . .*

However, note that sensor readings are now split across multiple keys. The reducer will need to preserve state and keep track of when readings associated with the current sensor end and the next sensor begin.9

The basic tradeoff between the two approaches discussed above (buffer and in-memory sort vs. value-to-key conversion) is where sorting is performed. One can explicitly implement secondary sorting in the reducer, which is likely to be faster but suffers from a scalability bottleneck.10 With value-to-key conversion, sorting is offloaded to the MapReduce execution framework. Note that this approach can be arbitrarily extended to tertiary, quaternary, etc. sorting. This pattern results in many more keys for the framework to sort, but distributed sorting is a task that the MapReduce runtime excels at since it lies at the heart of the programming model.

9Alternatively, Hadoop provides API hooks to define “groups” of intermediate keys that should be processed together in the reducer.

10Note that, in principle, this need not be an in-memory sort. It is entirely possible to implement a disk-based sort within the reducer, although one would be duplicating function ality that is already present in the MapReduce execution framework. It makes more sense to take advantage of functionality that is already present with value-to-key conversion.

3.5 Summary

This chapter provides a guide on the design of MapReduce algorithms. In particular, we present a number of “design patterns” that capture effective solutions to common problems. In summary, they are:

*•* “In-mapper combining”, where the functionality of the combiner is moved into the mapper. Instead of emitting intermediate output for every in put key-value pair, the mapper aggregates partial results across multiple input records and only emits intermediate key-value pairs after some amount of local aggregation is performed.

*•* The related patterns “pairs” and “stripes” for keeping track of joint events from a large number of observations. In the pairs approach, we keep track of each joint event separately, whereas in the stripes approach we keep track of all events that co-occur with the same event. Although the stripes approach is significantly more efficient, it requires memory on the order of the size of the event space, which presents a scalability bottleneck.

*•* “Order inversion”, where the main idea is to convert the sequencing of computations into a sorting problem. Through careful orchestration, we can send the reducer the result of a computation (e.g., an aggregate statis tic) before it encounters the data necessary to produce that computation.

*•* “Value-to-key conversion”, which provides a scalable solution for sec ondary sorting. By moving part of the value into the key, we can exploit the MapReduce execution framework itself for sorting.

Ultimately, controlling synchronization in the MapReduce programming model boils down to effective use of the following techniques:

1. Constructing complex keys and values that bring together data necessary for a computation. This is used in all of the above design patterns.

2. Executing user-specified initialization and termination code in either the mapper or reducer. For example, in-mapper combining depends on emis sion of intermediate key-value pairs in the map task termination code.

3. Preserving state across multiple inputs in the mapper and reducer. This is used in in-mapper combining, order inversion, and value-to-key con version.

4. Controlling the sort order of intermediate keys. This is used in order inversion and value-to-key conversion.

5. Controlling the partitioning of the intermediate key space. This is used in order inversion and value-to-key conversion.

This concludes our overview of MapReduce algorithm design. It should be clear by now that although the programming model forces one to express algorithms in terms of a small set of rigidly-defined components, there are many tools at one’s disposal to shape the flow of computation. In the next few chapters, we will focus on specific classes of MapReduce algorithms: for inverted indexing in Chapter 4, for graph processing in Chapter 5, and for expectation-maximization in Chapter 7.

Chapter 4

Inverted Indexing for Text Retrieval

Web search is the quintessential large-data problem. Given an information need expressed as a short query consisting of a few terms, the system’s task is to retrieve relevant web objects (web pages, PDF documents, PowerPoint slides, etc.) and present them to the user. How large is the web? It is difficult to compute exactly, but even a conservative estimate would place the size at several tens of billions of pages, totaling hundreds of terabytes (considering text alone). In real-world applications, users demand results quickly from a search engine—query latencies longer than a few hundred milliseconds will try a user’s patience. Fulfilling these requirements is quite an engineering feat, considering the amounts of data involved!

Nearly all retrieval engines for full-text search today rely on a data structure called an inverted index, which given a term provides access to the list of documents that contain the term. In information retrieval parlance, objects to be retrieved are generically called “documents” even though in actuality they may be web pages, PDFs, or even fragments of code. Given a user query, the retrieval engine uses the inverted index to score documents that contain the query terms with respect to some ranking model, taking into account features such as term matches, term proximity, attributes of the terms in the document (e.g., bold, appears in title, etc.), as well as the hyperlink structure of the documents (e.g., PageRank [117], which we’ll discuss in Chapter 5, or related metrics such as HITS [84] and SALSA [88]).

The web search problem decomposes into three components: gathering web content (crawling), construction of the inverted index (indexing) and rank ing documents given a query (retrieval). Crawling and indexing share similar characteristics and requirements, but these are very different from retrieval. Gathering web content and building inverted indexes are for the most part offline problems. Both need to be scalable and efficient, but they do not need to operate in real time. Indexing is usually a batch process that runs peri odically: the frequency of refreshes and updates is usually dependent on the

65

design of the crawler. Some sites (e.g., news organizations) update their con tent quite frequently and need to be visited often; other sites (e.g., government regulations) are relatively static. However, even for rapidly changing sites, it is usually tolerable to have a delay of a few minutes until content is searchable. Furthermore, since the amount of content that changes rapidly is relatively small, running smaller-scale index updates at greater frequencies is usually an adequate solution.1 Retrieval, on the other hand, is an online problem that demands sub-second response time. Individual users expect low query laten cies, but query throughput is equally important since a retrieval engine must usually serve many users concurrently. Furthermore, query loads are highly variable, depending on the time of day, and can exhibit “spikey” behavior due to special circumstances (e.g., a breaking news event triggers a large number of searches on the same topic). On the other hand, resource consumption for the indexing problem is more predictable.

A comprehensive treatment of web search is beyond the scope of this chap ter, and even this entire book. Explicitly recognizing this, we mostly focus on the problem of inverted indexing, the task most amenable to solutions in Map Reduce. This chapter begins by first providing an overview of web crawling (Section 4.1) and introducing the basic structure of an inverted index (Sec tion 4.2). A baseline inverted indexing algorithm in MapReduce is presented in Section 4.3. We point out a scalability bottleneck in that algorithm, which leads to a revised version presented in Section 4.4. Index compression is dis cussed in Section 4.5, which fills in missing details on building compact index structures. Since MapReduce is primarily designed for batch-oriented process ing, it does not provide an adequate solution for the retrieval problem, an issue we discuss in Section 4.6. The chapter concludes with a summary and pointers to additional readings.

4.1 Web Crawling

Before building inverted indexes, we must first acquire the document collection over which these indexes are to be built. In academia and for research purposes, this can be relatively straightforward. Standard collections for information retrieval research are widely available for a variety of genres ranging from blogs to newswire text. For researchers who wish to explore web-scale retrieval, there is the ClueWeb09 collection that contains one billion web pages in ten languages (totaling 25 terabytes) crawled by Carnegie Mellon University in early 2009.2 Obtaining access to these standard collections is usually as simple as signing an appropriate data license from the distributor of the collection, paying a reasonable fee, and arranging for receipt of the data.3

1Leaving aside the problem of searching live data streams such a tweets, which requires different techniques and algorithms.

2 http://boston.lti.cs.cmu.edu/Data/clueweb09/

3As an interesting side note, in the 1990s, research collections were distributed via postal mail on CD-ROMs, and later, on DVDs. Electronic distribution became common earlier this decade for collections below a certain size. However, many collections today are so large that

For real-world web search, however, one cannot simply assume that the collection is already available. Acquiring web content requires crawling, which is the process of traversing the web by repeatedly following hyperlinks and storing downloaded pages for subsequent processing. Conceptually, the process is quite simple to understand: we start by populating a queue with a “seed” list of pages. The crawler downloads pages in the queue, extracts links from those pages to add to the queue, stores the pages for further processing, and repeats. In fact, rudimentary web crawlers can be written in a few hundred lines of code.

However, effective and efficient web crawling is far more complex. The following lists a number of issues that real-world crawlers must contend with:

*•* A web crawler must practice good “etiquette” and not overload web servers. For example, it is common practice to wait a fixed amount of time before repeated requests to the same server. In order to respect these constraints while maintaining good throughput, a crawler typically keeps many execution threads running in parallel and maintains many TCP connections (perhaps hundreds) open at the same time.

*•* Since a crawler has finite bandwidth and resources, it must prioritize the order in which unvisited pages are downloaded. Such decisions must be made online and in an adversarial environment, in the sense that spammers actively create “link farms” and “spider traps” full of spam pages to trick a crawler into overrepresenting content from a particular site.

*•* Most real-world web crawlers are distributed systems that run on clusters of machines, often geographically distributed. To avoid downloading a page multiple times and to ensure data consistency, the crawler as a whole needs mechanisms for coordination and load-balancing. It also needs to be robust with respect to machine failures, network outages, and errors of various types.

*•* Web content changes, but with different frequency depending on both the site and the nature of the content. A web crawler needs to learn these update patterns to ensure that content is reasonably current. Getting the right recrawl frequency is tricky: too frequent means wasted resources, but not frequent enough leads to stale content.

*•* The web is full of duplicate content. Examples include multiple copies of a popular conference paper, mirrors of frequently-accessed sites such as Wikipedia, and newswire content that is often duplicated. The problem is compounded by the fact that most repetitious pages are not exact duplicates but near duplicates (that is, basically the same page but with different ads, navigation bars, etc.) It is desirable during the crawling process to identify near duplicates and select the best exemplar to index.

the only practical method of distribution is shipping hard drives via postal mail.

*•* The web is multilingual. There is no guarantee that pages in one language only link to pages in the same language. For example, a professor in Asia may maintain her website in the local language, but contain links to publications in English. Furthermore, many pages contain a mix of text in different languages. Since document processing techniques (e.g., tokenization, stemming) differ by language, it is important to identify the (dominant) language on a page.

The above discussion is not meant to be an exhaustive enumeration of issues, but rather to give the reader an appreciation of the complexities involved in this intuitively simple task. For more information, see a recent survey on web crawling [113]. Section 4.7 provides pointers to additional readings.

4.2 Inverted Indexes

In its basic form, an inverted index consists of postings lists, one associated with each term that appears in the collection.4 The structure of an inverted index is illustrated in Figure 4.1. A postings list is comprised of individual postings, each of which consists of a document id and a *payload*—information about occurrences of the term in the document. The simplest payload is. . . nothing! For simple boolean retrieval, no additional information is needed in the posting other than the document id; the existence of the posting itself indicates that presence of the term in the document. The most common payload, however, is term frequency (*tf* ), or the number of times the term occurs in the document. More complex payloads include positions of every occurrence of the term in the document (to support phrase queries and document scoring based on term proximity), properties of the term (such as if it occurred in the page title or not, to support document ranking based on notions of importance), or even the results of additional linguistic processing (for example, indicating that the term is part of a place name, to support address searches). In the web context, anchor text information (text associated with hyperlinks from other pages to the page in question) is useful in enriching the representation of document content (e.g., [107]); this information is often stored in the index as well.

In the example shown in Figure 4.1, we see that:

*term*1 occurs in *{d*1*, d*5*, d*6*, d*11*, . . .}*,

*term*2 occurs in *{d*11*, d*23*, d*59*, d*84*, . . .}*, and

*term*3 occurs in *{d*1*, d*4*, d*11*, d*19*, . . .}*.

In an actual implementation, we assume that documents can be identified by a unique integer ranging from 1 to *n*, where *n* is the total number of doc uments.5 Generally, postings are sorted by document id, although other sort

4In information retrieval parlance, *term* is preferred over *word* since documents are processed (e.g., tokenization and stemming) into basic units that are often not words in the linguistic sense.

5It is preferable to start numbering the documents at one since it is not possible to code zero with many common compression schemes used in information retrieval; see Section 4.5.

**terms postings**

*term1*

*…*

*d1 p d5 p d6 p d11 p*

*term2  term3*

*d11 p d23 p d59 p d84 p … d1 p d4 p d11 p d19 p …*

*… …*

Figure 4.1: Simple illustration of an inverted index. Each term is associated

with a list of postings. Each posting is comprised of a document id and a

payload, denoted by *p* in this case. An inverted index provides quick access to

documents ids that contain a term.

orders are possible as well. The document ids have no inherent semantic mean

ing, although assignment of numeric ids to documents need not be arbitrary.

For example, pages from the same domain may be consecutively numbered. Or,

Figure 4.1: Simple illustraPon of an inverted index. Each term is associated with a list of

alternatively, pages that are higher in quality (based, for example, on Page

Rank values) might be assigned smaller numeric values so that they appear

posPngs. Each posPng is comprised of a document id and a payload, denoted by *p* in this toward the front of a postings list. Either way, an auxiliary data structure is

case. An inverted index provides quick access to documents ids that contain a term.

necessary to maintain the mapping from integer document ids to some other

more meaningful handle, such as a URL.

Given a query, retrieval involves fetching postings lists associated with query

terms and traversing the postings to compute the result set. In the simplest

case, boolean retrieval involves set operations (union for boolean OR and in

tersection for boolean AND) on postings lists, which can be accomplished very

efficiently since the postings are sorted by document id. In the general case,

however, query–document scores must be computed. Partial document scores

are stored in structures called *accumulators*. At the end (i.e., once all postings

have been processed), the top *k* documents are then extracted to yield a ranked

list of results for the user. Of course, there are many optimization strategies

for query evaluation (both approximate and exact) that reduce the number of

postings a retrieval engine must examine.

The size of an inverted index varies, depending on the payload stored in each

posting. If only term frequency is stored, a well-optimized inverted index can

be a tenth of the size of the original document collection. An inverted index

that stores positional information would easily be several times larger than

one that does not. Generally, it is possible to hold the entire vocabulary (i.e.,

dictionary of all the terms) in memory, especially with techniques such as front

coding [156]. However, with the exception of well-resourced, commercial web

search engines,6 postings lists are usually too large to store in memory and must

be held on disk, usually in compressed form (more details in Section 4.5). Query

6Google keeps indexes in memory.

Algorithm 4.1 Baseline inverted indexing algorithm

Mappers emit postings keyed by terms, the execution framework groups post ings by term, and the reducers write postings lists to disk.

1: class Mapper

2: procedure Map(docid *n,* doc *d*)

3: *H ←* new AssociativeArray

4: for all term *t ∈* doc *d* do

5: *H{t} ← H{t}* + 1

6: for all term *t ∈ H* do

7: Emit(term *t,* posting *hn, H{t}i*)

1: class Reducer

2: procedure Reduce(term *t,* postings [*hn*1*, f*1*i,hn*2*, f*2*i. . .*]) 3: *P ←* new List

4: for all posting *ha, fi ∈* postings [*hn*1*, f*1*i,hn*2*, f*2*i. . .*] do 5: Append(*P,ha, fi*)

6: Sort(*P*)

7: Emit(term *t,* postings *P*)

evaluation, therefore, necessarily involves random disk access and “decoding” of the postings. One important aspect of the retrieval problem is to organize disk operations such that random seeks are minimized.

Once again, this brief discussion glosses over many complexities and does a huge injustice to the tremendous amount of research in information retrieval. However, our goal is to provide the reader with an overview of the important issues; Section 4.7 provides references to additional readings.

4.3 Inverted Indexing: Baseline Implementation

MapReduce was designed from the very beginning to produce the various data structures involved in web search, including inverted indexes and the web graph. We begin with the basic inverted indexing algorithm shown in Al gorithm 4.1.

Input to the mapper consists of document ids (keys) paired with the actual content (values). Individual documents are processed in parallel by the map pers. First, each document is analyzed and broken down into its component terms. The processing pipeline differs depending on the application and type of document, but for web pages typically involves stripping out HTML tags and other elements such as JavaScript code, tokenizing, case folding, removing stop words (common words such as ‘the’, ‘a’, ‘of’, etc.), and stemming (removing affixes from words so that ‘dogs’ becomes ‘dog’). Once the document has been analyzed, term frequencies are computed by iterating over all the terms and keeping track of counts. Lines 4 and 5 in the pseudo-code reflect the process of computing term frequencies, but hides the details of document processing.

After this histogram has been built, the mapper then iterates over all terms. For each term, a pair consisting of the document id and the term frequency is created. Each pair, denoted by *hn, H{t}i* in the pseudo-code, represents an individual posting. The mapper then emits an intermediate key-value pair with the term as the key and the posting as the value, in line 7 of the mapper pseudo-code. Although as presented here only the term frequency is stored in the posting, this algorithm can be easily augmented to store additional infor mation (e.g., term positions) in the payload.

In the shuffle and sort phase, the MapReduce runtime essentially performs a large, distributed group by of the postings by term. Without any additional effort by the programmer, the execution framework brings together all the postings that belong in the same postings list. This tremendously simplifies the task of the reducer, which simply needs to gather together all the postings and write them to disk. The reducer begins by initializing an empty list and then appends all postings associated with the same key (term) to the list. The postings are then sorted by document id, and the entire postings list is emitted as a value, with the term as the key. Typically, the postings list is first compressed, but we leave this aside for now (see Section 4.4 for more details). The final key-value pairs are written to disk and comprise the inverted index. Since each reducer writes its output in a separate file in the distributed file system, our final index will be split across *r* files, where *r* is the number of reducers. There is no need to further consolidate these files. Separately, we must also build an index to the postings lists themselves for the retrieval engine: this is typically in the form of mappings from term to (file, byte offset) pairs, so that given a term, the retrieval engine can fetch its postings list by opening the appropriate file and seeking to the correct byte offset position in that file.

Execution of the complete algorithm is illustrated in Figure 4.2 with a toy example consisting of three documents, three mappers, and two reducers. Intermediate key-value pairs (from the mappers) and the final key-value pairs comprising the inverted index (from the reducers) are shown in the boxes with dotted lines. Postings are shown as pairs of boxes, with the document id on the left and the term frequency on the right.

The MapReduce programming model provides a very concise expression of the inverted indexing algorithm. Its implementation is similarly concise: the basic algorithm can be implemented in as few as a couple dozen lines of code in Hadoop (with minimal document processing). Such an implementation can be completed as a week-long programming assignment in a course for advanced undergraduates or first-year graduate students [83, 93]. In a non-MapReduce indexer, a significant fraction of the code is devoted to grouping postings by term, given constraints imposed by memory and disk (e.g., memory capacity is limited, disk seeks are slow, etc.). In MapReduce, the programmer does not need to worry about any of these issues—most of the heavy lifting is performed by the execution framework.

doc 1

one fish, two fish mapper

doc 2

red fish, blue fish mapper

doc 3

one red bird

mapper

*d1* fish 2 *d1* one 1 *d1* two 1

*d2* blue 1 fish *d2* 2 *d2* red 1

*d3* bird 1 one *d3* 1 *d3* red 1

Shuffle and Sort: aggregate values by keys

reducer

*d1* fish 2 *d2* 2

reducer

*d3* bird 1

*d1* one 1 *d1* two 1

*d3* 1

*d2* blue 1

*d2* red 1 *d3* 1

Figure 4.3: Simple illustraPon of the baseline inverted indexing algorithm in MapReduce with

Figure 4.2: Simple illustration of the baseline inverted indexing algorithm in three mappers and two reducers. PosPngs are shown as pairs of boxes (*docid*, *&*).

MapReduce with three mappers and two reducers. Postings are shown as pairs of boxes (*docid*, *tf* ).

4.4 Inverted Indexing: Revised Implementation

The inverted indexing algorithm presented in the previous section serves as a reasonable baseline. However, there is a significant scalability bottleneck: the algorithm assumes that there is sufficient memory to hold all postings asso ciated with the same term. Since the basic MapReduce execution framework makes no guarantees about the ordering of values associated with the same key, the reducer first buffers all postings (line 5 of the reducer pseudo-code in Algorithm 4.1) and then performs an in-memory sort before writing the post ings to disk.7 For efficient retrieval, postings need to be sorted by document id. However, as collections become larger, postings lists grow longer, and at some point in time, reducers will run out of memory.

There is a simple solution to this problem. Since the execution framework guarantees that keys arrive at each reducer in sorted order, one way to overcome

7See similar discussion in Section 3.4: in principle, this need not be an in-memory sort. It is entirely possible to implement a disk-based sort within the reducer.

the scalability bottleneck is to let the MapReduce runtime do the sorting for us. Instead of emitting key-value pairs of the following type:

(term *t,* posting *hdocid, fi*)

We emit intermediate key-value pairs of the type instead:

(tuple *ht, docidi,*tf *f*)

In other words, the key is a tuple containing the term and the document id, while the value is the term frequency. This is exactly the value-to-key con version design pattern introduced in Section 3.4. With this modification, the programming model ensures that the postings arrive in the correct order. This, combined with the fact that reducers can hold state across multiple keys, al lows postings lists to be created with minimal memory usage. As a detail, remember that we must define a custom partitioner to ensure that all tuples with the same term are shuffled to the same reducer.

The revised MapReduce inverted indexing algorithm is shown in Algo rithm 4.2. The mapper remains unchanged for the most part, other than differences in the intermediate key-value pairs. The Reduce method is called for each key (i.e., *ht, ni*), and by design, there will only be one value associated with each key. For each key-value pair, a posting can be directly added to the postings list. Since the postings are guaranteed to arrive in sorted order by document id, they can be incrementally coded in compressed form—thus ensuring a small memory footprint. Finally, when all postings associated with the same term have been processed (i.e., *t 6*= *tprev*), the entire postings list is emitted. The final postings list must be written out in the Close method. As with the baseline algorithm, payloads can be easily changed: by simply re placing the intermediate value *f* (term frequency) with whatever else is desired (e.g., term positional information).

There is one more detail we must address when building inverted indexes. Since almost all retrieval models take into account document length when com puting query–document scores, this information must also be extracted. Al though it is straightforward to express this computation as another MapReduce job, this task can actually be folded into the inverted indexing process. When processing the terms in each document, the document length is known, and can be written out as “side data” directly to HDFS. We can take advantage of the ability for a mapper to hold state across the processing of multiple documents in the following manner: an in-memory associative array is created to store doc ument lengths, which is populated as each document is processed.8 When the mapper finishes processing input records, document lengths are written out to HDFS (i.e., in the Close method). This approach is essentially a variant of the in-mapper combining pattern. Document length data ends up in *m* different files, where *m* is the number of mappers; these files are then consolidated into a more compact representation. Alternatively, document length information can

8In general, there is no worry about insufficient memory to hold these data.

Algorithm 4.2 Scalable inverted indexing

By applying the value-to-key conversion design pattern, the execution frame work is exploited to sort postings so that they arrive sorted by document id in the reducer.

1: class Mapper

2: method Map(docid *n,* doc *d*)

3: *H ←* new AssociativeArray

4: for all term *t ∈* doc *d* do

5: *H{t} ← H{t}* + 1

6: for all term *t ∈ H* do

7: Emit(tuple *ht, ni,*tf *H{t}*)

1: class Reducer

2: method Initialize

3: *tprev ← ∅*

4: *P ←* new PostingsList

5: method Reduce(tuple *ht, ni,*tf [*f*])

6: if *t 6*= *tprev ∧ tprev 6*= *∅* then

7: Emit(term *tprev,* postings *P*)

8: *P.*Reset()

9: *P.*Add(*hn, fi*)

10: *tprev ← t*

11: method Close

12: Emit(term *t,* postings *P*)

be emitted in special key-value pairs by the mapper. One must then write a custom partitioner so that these special key-value pairs are shuffled to a single reducer, which will be responsible for writing out the length data separate from the postings lists.

4.5 Index Compression

We return to the question of how postings are actually compressed and stored on disk. This chapter devotes a substantial amount of space to this topic be cause index compression is one of the main differences between a “toy” indexer and one that works on real-world collections. Otherwise, MapReduce inverted indexing algorithms are pretty straightforward.

Let us consider the canonical case where each posting consists of a document id and the term frequency. A na¨ıve implementation might represent the first as a 32-bit integer9 and the second as a 16-bit integer. Thus, a postings list might be encoded as follows:

9However, note that 232 *−* 1 is “only” 4,294,967,295, which is much less than even the most conservative estimate of the size of the web.

[(5*,* 2)*,*(7*,* 3)*,*(12*,* 1)*,*(49*,* 1)*,*(51*,* 2)*, . . .*]

where each posting is represented by a pair in parentheses. Note that all brackets, parentheses, and commas are only included to enhance readability; in reality the postings would be represented as a long stream of integers. This na¨ıve implementation would require six bytes per posting. Using this scheme, the entire inverted index would be about as large as the collection itself. For tunately, we can do significantly better.

The first trick is to encode *differences* between document ids as opposed to the document ids themselves. Since the postings are sorted by document ids, the differences (called *d*-gaps) must be positive integers greater than zero. The above postings list, represented with *d*-gaps, would be:

[(5*,* 2)*,*(2*,* 3)*,*(5*,* 1)*,*(37*,* 1)*,*(2*,* 2)*, . . .*]

Of course, we must actually encode the first document id. We haven’t lost any information, since the original document ids can be easily reconstructed from the *d*-gaps. However, it’s not obvious that we’ve reduced the space require ments either, since the largest possible *d*-gap is one less than the number of documents in the collection.

This is where the second trick comes in, which is to represent the *d*-gaps in a way such that it takes less space for smaller numbers. Similarly, we want to apply the same techniques to compress the term frequencies, since for the most part they are also small values. But to understand how this is done, we need to take a slight detour into compression techniques, particularly for coding integers.

Compression, in general, can be characterized as either *lossless* or *lossy*: it’s fairly obvious that loseless compression is required in this context. To start, it is important to understand that all compression techniques represent a time– space tradeoff. That is, we reduce the amount of space on disk necessary to store data, but at the cost of extra processor cycles that must be spent coding and decoding data. Therefore, it is possible that compression reduces size but also slows processing. However, if the two factors are properly balanced (i.e., decoding speed can keep up with disk bandwidth), we can achieve the best of both worlds: smaller *and* faster.

Byte-Aligned and Word-Aligned Codes

In most programming languages, an integer is encoded in four bytes and holds a value between 0 and 232 *−* 1, inclusive. We limit our discussion to *unsigned* integers, since *d*-gaps are always positive (and greater than zero). This means that 1 and 4,294,967,295 both occupy four bytes. Obviously, encoding *d*-gaps this way doesn’t yield any reductions in size.

A simple approach to compression is to only use as many bytes as is neces sary to represent the integer. This is known as variable-length integer coding (varInt for short) and accomplished by using the high order bit of every byte

as the *continuation bit*, which is set to one in the last byte and zero else where. As a result, we have 7 bits per byte for coding the value, which means that 0 *≤ n <* 27can be expressed with 1 byte, 27 *≤ n <* 214 with 2 bytes, 214 *≤ n <* 221 with 3, and 221 *≤ n <* 228 with 4 bytes. This scheme can be extended to code arbitrarily-large integers (i.e., beyond 4 bytes). As a concrete example, the two numbers:

127, 128

would be coded as such:

1 1111111, 0 0000001 1 0000000

The above code contains two code words, the first consisting of 1 byte, and the second consisting of 2 bytes. Of course, the comma and the spaces are there only for readability. Variable-length integers are byte-aligned because the code words always fall along byte boundaries. As a result, there is never any ambiguity about where one code word ends and the next begins. However, the downside of varInt coding is that decoding involves lots of bit operations (masks, shifts). Furthermore, the continuation bit sometimes results in fre quent branch mispredicts (depending on the actual distribution of *d*-gaps), which slows down processing.

A variant of the varInt scheme was described by Jeff Dean in a keynote talk at the WSDM 2009 conference.10 The insight is to code groups of four integers at a time. Each group begins with a prefix byte, divided into four 2-bit values that specify the byte length of each of the following integers. For example, the following prefix byte:

00,00,01,10

indicates that the following four integers are one byte, one byte, two bytes, and three bytes, respectively. Therefore, each group of four integers would consume anywhere between 5 and 17 bytes. A simple lookup table based on the prefix byte directs the decoder on how to process subsequent bytes to recover the coded integers. The advantage of this group varInt coding scheme is that values can be decoded with fewer branch mispredicts and bitwise operations. Experiments reported by Dean suggest that decoding integers with this scheme is more than twice as fast as the basic varInt scheme.

In most architectures, accessing entire machine words is more efficient than fetching all its bytes separately. Therefore, it makes sense to store postings in increments of 16-bit, 32-bit, or 64-bit machine words. Anh and Moffat [8] presented several word-aligned coding methods, one of which is called Simple 9, based on 32-bit words. In this coding scheme, four bits in each 32-bit word are reserved as a *selector*. The remaining 28 bits are used to code actual integer values. Now, there are a variety of ways these 28 bits can be divided

10 http://research.google.com/people/jeff/WSDM09-keynote.pdf