Chapter 1

Introduction

This is a post-production manuscript of: Jimmy Lin and Chris Dyer. Data-Intensive Text Processing with MapReduce. Morgan & Claypool Publishers, 2010. This version was compiled on December 25, 2017.

MapReduce [45] is a programming model for expressing distributed computations 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 processing dating back several decades. MapReduce has since enjoyed widespread adoption via an open-source implementation called Hadoop, whose development was led by Yahoo (now an Apache project). Today, a vibrant software ecosystem has sprung up around Hadoop, with significant activity in both industry 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 gathering, analyzing, monitoring, filtering, searching, or organizing web content must tackle large-data problems: "web-scale" processing is practically synonymous 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 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 post[1] 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 revealed[2] similarly 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 importance 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 fundamental 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]

[1] http://www.dbms2.com/2009/04/30/ebays-two-enormous-data-warehouses/

[2] http://www.dbms2.com/2009/05/11/facebook-hadoop-and-hive/

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

- Astronomers have long recognized the importance of a "digital observatory" 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 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 technology is nothing less than transformative [103]. The European Bioinformatics 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 personalized medicine, where interventions can be specifically targeted for an individual.

Increasingly, scientific breakthroughs will be powered by advanced computing capabilities that help researchers manipulate, explore, and mine massive datasets [72]—this has been hailed as the emerging "fourth paradigm" of science [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 concerned 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 discuss 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 attempt to capture statistical regularities in data for the purposes of some task or application. There are three components to this approach: data, representations 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 extract, such as the grammatical relationship between words. Finally, algorithms or models are applied to capture regularities in the data in terms of the extracted 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 situate texts along a scale of "happiness", a task known as sentiment analysis or opinion mining [118], which has been applied to everything from understanding 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 confusable 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 problem (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 mistakes, 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 important, as we shall see), and not surprisingly, found that more data led to better accuracy. Across many different algorithms, the increase in accuracy was approximately linear in the log of the size of the training data. Furthermore, with increasing amounts of training data, the accuracy of different algorithms converged, 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 matter, 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 gathering 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 answering (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 translation (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 challenging 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 probability 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 V^n 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 techniques [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 trillion 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

[4] As 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.

[5] As in, so stupid it couldn't possibly work.

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 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, hundreds, 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

[6] This 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).

were grids,[8] and before grids, there were vector supercomputers, each having 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 applications 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 enormous globe-spanning graph of friendships to recommend new connections. An online email service analyzes messages and user behavior to optimize ad selection 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 computing 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 computing 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 physical 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

[8] What *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 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.

[9] The 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.

arbitrary computation. Virtual machines that are no longer needed are destroyed, 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 freed from upfront capital investments necessary to build datacenters and substantial 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 allocated 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 because 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 significantly 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 capital) 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: processing 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 service”, 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 rebranding 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 number 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 previously 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 latter 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 effective, 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 Hölzle'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 communication between nodes in a commodity network-based cluster? Since workloads today are beyond the capability of any *single* machine (no matter how powerful), 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 Hölzle 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 cluster 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 Hölzle 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) consume 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 Hölzle [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 inconsistencies or indeterminism from the user perspective. As servers go down, other cluster nodes should seamlessly step in to handle the load, and overall performance 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 implementations 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 together 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 fundamentally 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 scenario[10] poignantly illustrates the large performance gap between sequential operations and random seeks: assume a 1 terabyte database containing 10^{10} 100-byte records. Given reasonable assumptions 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 because the programmer must simultaneously keep track of many details in short term memory—ranging from the mundane (e.g., variable names) to the sophisticated (e.g., a corner case of an algorithm that requires special treatment).

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

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

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 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 "design 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 providing 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 between 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 "runtime". The advantage is that the execution framework only needs to be designed once and verified for correctness—thereafter, as long as the developer expresses computations in the programming model, code is guaranteed to behave 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 characteristics sketched above are, of course, unobtainable. One of the fundamental

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

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 behind. 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 observations are primarily about software engineers and the software development process, the same is also true of algorithms: increasing the degree of parallelization 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. Recall that the programming model maintains a clear separation between *what* computations need to occur with *how* those computations are actually orchestrated on a cluster. As a result, a MapReduce algorithm remains fixed, and it is the responsibility of the execution framework to execute the algorithm. Amazingly, 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

[13] Note 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.

> that parallel machines will eventually displace sequential ones in computationally intensive domains. This, however, has not happened." — Leslie Valiant [148][14]

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 problems 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 confined 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 content, 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 organizations 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 ubiquitous, 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 happily processing terabytes upon terabytes of data within the week. Finally, the

[14] Guess when this was written? You may be surprised.

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 effective 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 science 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 implementations 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 engineers 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 computation 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 computational resources we have seen to date. However, as anyone who has taken an introductory computer science course knows, abstractions manage complexity

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

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

by hiding details and presenting well-defined behaviors to users of those abstractions. 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 critique applies to MapReduce: it makes certain large-data problems easier, but suffers from limitations as well. This means that MapReduce is not the final 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].