Article Recommendation System

Based on Keyword using Map-Reduce

May, 2015

Submitted in partially fulfillment of the requirement for the Degree of

Bachelor of Technology

in

Information Technology

Under the Supervision of

Mrs. Sanjana Singh

by

Nitish Bhargava (111435)

to

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

and INFORMATION TECHNOLOGY,

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY,

WAKNAGHAT.

CERTIFICATE

This is to certify that the work titled "**Article Recommendation System based on keyword using Map Reduce"** submitted by **Nitish Bhargava** in the partial fulfillment for the award of degree of Bachelor of Technology in Information Technology from Jaypee University of Information Technology, Waknaghat has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor: ` Name of Supervisor : Mrs. Sanjana Singh

Designation : Assistant Professor

Date :

ACKNOWLEDGEMENT

I would like to express my gratitude to all those who gave us the possibility to complete this project. I want to thank the Department of CSE & IT in JUIT for giving us the permission to commence this project in the first instance, to do the necessary research work.

I am deeply indebted to my project guide Mrs. Sanjana Singh, whose help, stimulating suggestions and encouragement helped me in all the time of research on this project. I feel motivated and encouraged every time I get his encouragement. For his coherent guidance throughout the tenure of the project, I feel fortunate to be taught by him, who gave me his unwavering support.

I am also grateful to **Mr.Amit Singh(CSE Project lab**) for his practical help and guidance.

Nitish Bhargava

Contents

List of Figures

Abstract

Map Reduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model.

A Map Reduce program is composed of a Map() procedure that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a Reduce() procedure that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "Map Reduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

The model is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the Map Reduce framework is not the same as in their original forms. The key contributions of the Map Reduce framework are not the actual map and reduce functions, but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine once. As such, a single-threaded implementation of Map Reduce (such as MongoDB) will usually not be faster than a traditional (non-Map Reduce) implementation, any gains are usually only seen with multi-threaded implementations. Only when the optimized distributed shuffle operation (which reduces network communication cost) and fault tolerance features of the Map Reduce framework come into play, is the use of this model beneficial.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of Map Reduce runs on a large cluster of commodity machines and is highly scalable: a typical Map Reduce computation processes many terabytes of data on thousands of machines. Programmers and the system easy to use: hundreds of Map Reduce programs have been implemented and upwards of one thousand Map Reduce jobs are executed on Google's clusters every day.

Map Reduce libraries have been written in many programming languages, with different levels of optimization. A popular open-source implementation is Apache Hadoop. The name Map Reduce originally referred to the proprietary Google technology, but has since been generalized.

Hadoop is a Java software framework that supports data-intensive distributed applications and is developed under open source license. It enables applications to work with thousands of nodes and petabytes of data. The two major pieces of Hadoop are HDFS: Hadoop's own file system. This is designed to scale to petabytes of storage and runs on top of the file systems of the underlying operating systems.

Chapter 1: Introduction

Map Reduce 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. Map Reduce 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.

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, altering, 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 Map Reduce in 2004 to processing 20 PB a day with Map Reduce in 2008 . 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 revealed2 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.

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. 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.

- 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.

1.1 Computing in the Cloud

For better or for worse, it is often difficult to untangled Map Reduce 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, 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 Map Reduce 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 Map Reduce. 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 Map Reduce.

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 or 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 , that is available open source. Increased competition will benefit cloud users, but what direct relevance does this have for Map Reduce? The connection is quite simple: processing large amounts of data with Map Reduce 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 Map Reduce much more accessible.

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 Map Reduce. To be fair, all of these ideas have been discussed in the computer science literature for some time (some for decades), and Map Reduce 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.

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 Holzle 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 Map Reduce programming model are designed around clusters of low-end commodity servers.

Capital costs in acquiring servers are, 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. 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.

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 happened." | Leslie Valiant.

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 concerned 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. 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). Nevertheless, parallel processing became an important issue at the forefront of everyone's mind it represented the only way forward.

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

To be fair, Map Reduce 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]. In the model, an arbitrary number of processors, sharing an unboundedly large memory, operate synchronously on a shared input to produce some output.

CHAPTER 2 : LITERATURE SURVEY

2.1. Summary of Papers

Title of Paper	Map Reduce: Simplified Data Processing on Large Clusters
Authors	Jeffrey Dean and Sanjay Ghemawat
Year of Publication	2004
Publishing Details	IEEE Transactions on Knowledge and Data Engineering,Vol.24 No.1
Summary	The Map Reduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as Map Reduce computations. For example, Map Reduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of Map Reduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

Title of Paper	Data-Intensive Text Processing with MapReduce
Authors	Jimmy Lin and Chris Dyer
Year of Publication	2008
Publishing Details	IEEE Computer Society
Summary	Used for many years in the business community,data mining and predictive analytics are finding new roles in areas outside business. Also referred to as *knowledge discovery* or *sense making* tools these analytical processes can help analysts, managers, and operational personnel identify actionable patterns and trends in data. Briefly, data mining is "[a]n information extraction activity whose goal is to discover hidden facts contained in the databases of many sites. In other words, data mining involves the systematic analysis of data using automated methods in an effort to identify meaningful or otherwise interesting patterns, trends, or relationships in the data.Crime and criminal behavior, including the most aberrant or heinous crimes, frequently can be categorized and modeled— a characteristic used successfully in the apprehension of serial killers and child predators, as well as drug dealers,robbers, and thieves. So it's no surprise that data mining and predictive analytics are rapidly gaining acceptance and use in the applied public safety, security,and intelligence .

Title of Paper	Next Step for Learning Analytics
Authors	Jinan Fiaidhi
Year of Publication	2014
Publishing Details	IEEE Computer Society
Summary	The paper presents description about Text analytics applies a variety of natural language processing analysis techniques along with linguistics, statistical, and data-mining techniques to extract concepts and patterns that can be applied to categorize and classify textual documents. It also attempts to transform the unstructured information into data that can be used with more traditional learning analytics techniques. Finally, it helps identify meaning and relationships in large volumes of information. However, there is no single method appropriate for all text analysis tasks. Learning analytics approaches must take several different perspectives and accommodate different data sources.The ideal vision for learning analytics is to integrate analytics for both structured and unstructured data (mainly of textual nature of a comprehensive learning analytics architecture.

Title of Paper	Predictive Analytics
Authors	Ravi Kalakota
Year of Publication	2014
Web link	http://practicalanalytics.wordpress.com/predictive-analytics-101/
Summary	This article presents the various techniques which are changing the world business and turning them into valuable and actionable information like summation, predictive, descriptive and prescriptive where descriptive is of our interest where data mining comes into action.It's a new world with new rules especially around man +machine interactions. "how companies find customers " to "how customers find companies today" is evolving. Serving customers with "with few/isolated channels, screens,devices".How demographic segmentation was enough to complex behavious segmentation to drive1:1personalization. The end goal of predictive analytics = [Better outcomes, smarter decisions, actionable insights, relevant information]. How you execute this varies by industry and information supply chain *(Raw Data -> Aggregated Data -> Contextual Intelligence -> Analytical Insights (reporting vs. prediction) -> Decisions (Human or Automated Downstream Actions)).*

2.2. Integrated Summary of Literature Studied

Literature survey of the research papers helped us to understand the need for finding the Effective Pattern with use in identifying fraudulent activity and the various techniques which can help make an accurate and effective system for analyzing inoxes for fraudulent activity. Inbox data is huge for a operating company and need to be structured for the extraction of useful pattern. Also there are techniques which aim to solve the problems described through other data mining techniques with varying results. Studying various papers revealed the existence of various approaches to solve the above stated problem by using term weight approach .But there are certain issues which needs to be resolved .

Also Research papers have been published on Stop word elimination and Porterstemming and how they help in reduction of the search space by eliminating stop words which are general English words. Porters Stemming helps in suffix removal by converting a word into its root and improves the performance of an IR system will be improved if term groups are conflated into a single term. This may be done by removal of the various suffixes -ED, -ING, -ION, -IONS to leave the single term. In addition, the suffix stripping process will reduce the total number of terms in the IR system, and hence reduce the size and complexity of the data in the system, which is always advantageous.

We know that multiple data mining methods have been developed for finding useful patterns in contents like PDF files, text files. Current paper addresses the problem of making text mining results more effective to humanities scholars, journalists, intelligence analysts, and other researchers. To use effective and bring to up to date discovered patterns is still an open research task, especially in the domain of text mining. Text mining is the finding of very interesting knowledge (or features) in the text documents. It is a very difficult to find exact knowledge (or features) in text documents to help users what they actually want. A d-pattern mining technique is discovered. It evaluates specificities of patterns and then evaluates term-

weights according to the distribution of terms in the discovered patterns. It solves Misinterpretation Problem.

The study of the associated literature about the above mentioned topics aided in developing an integrated approach as to how to develop an accurate system for mining effective pattern and present it to the user with the most appropriate answer where the user can decide the further action.

Chapter 3

Map Reduce

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, 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. 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 specifically, 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 another?

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

In traditional parallel or distributed programming environments, the developer 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, or libraries implementing the Message Passing Interface (MPI) for cluster-level parallelism, 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 rudimentary support for dealing with very large amounts of input data. When using existing parallel computing approaches for large-data computation, the programmer 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 Map Reduce is that it provides an abstraction that hides many system-level details from the programmer. Therefore, 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, Map Reduce 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! Map Reduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a scalable, robust, and efficient manner. 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 environment is typically handled by a distributed file system that sits underneath Map Reduce.

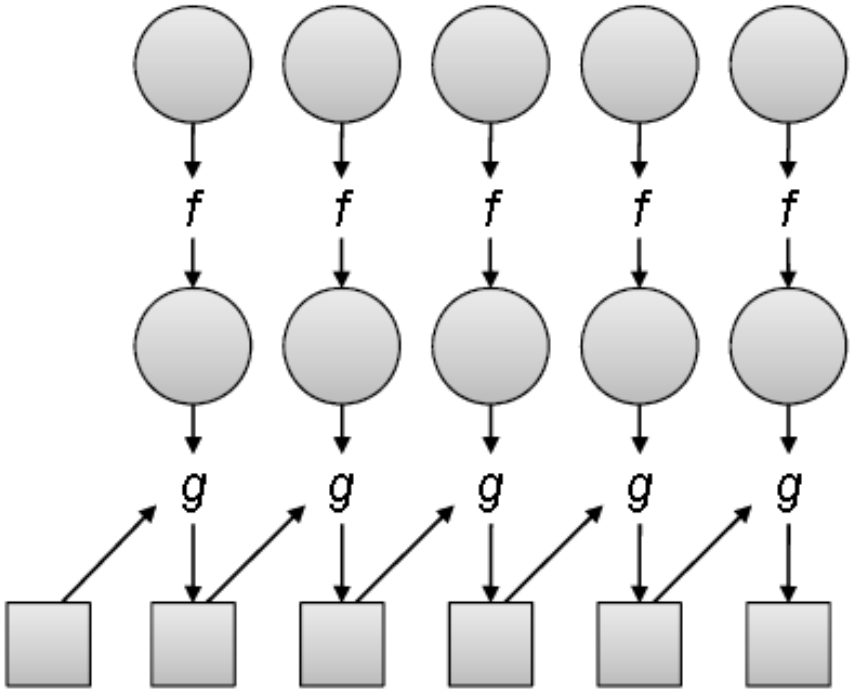


Illustration of map and fold, two higher-order functions commonly used together in functional programming: map takes a function f and applies it to every element in a list, while fold iteratively applies a function g to aggregate results.

- **Picture from refrence Book[1].**

3.1 Functional Programming Roots

Map Reduce has its roots in functional programming, which is exemplified in languages such as Lisp and ML. 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.

In a nutshell, we have described Map Reduce. The map phase in Map Reduce roughly corresponds to the map operation in functional programming, whereas the reduce phase in Map Reduce roughly corresponds to the fold operation in functional programming. As we will discuss in detail shortly, the Map Reduce 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, Map Reduce 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 aggregated by another user-specified computation. The programmer defines these two types of computations, and the execution framework coordinates the actual processing (very loosely, Map Reduce provides a functional

abstraction). Although such a two-stage processing structure may appear to be very restrictive, many interesting algorithms can be expressed quite concisely especially if one decomposes complex algorithms into a sequence of Map Reduce jobs. Subsequent chapters in this book focus on how a number of algorithms can be implemented in Map Reduce.

3.2 Mapper and Reducer

Key-value pairs form the basic data structure in Map Reduce. 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 Map Reduce 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 Map Reduce, the programmer defines a mapper and a reducer with the following signatures:

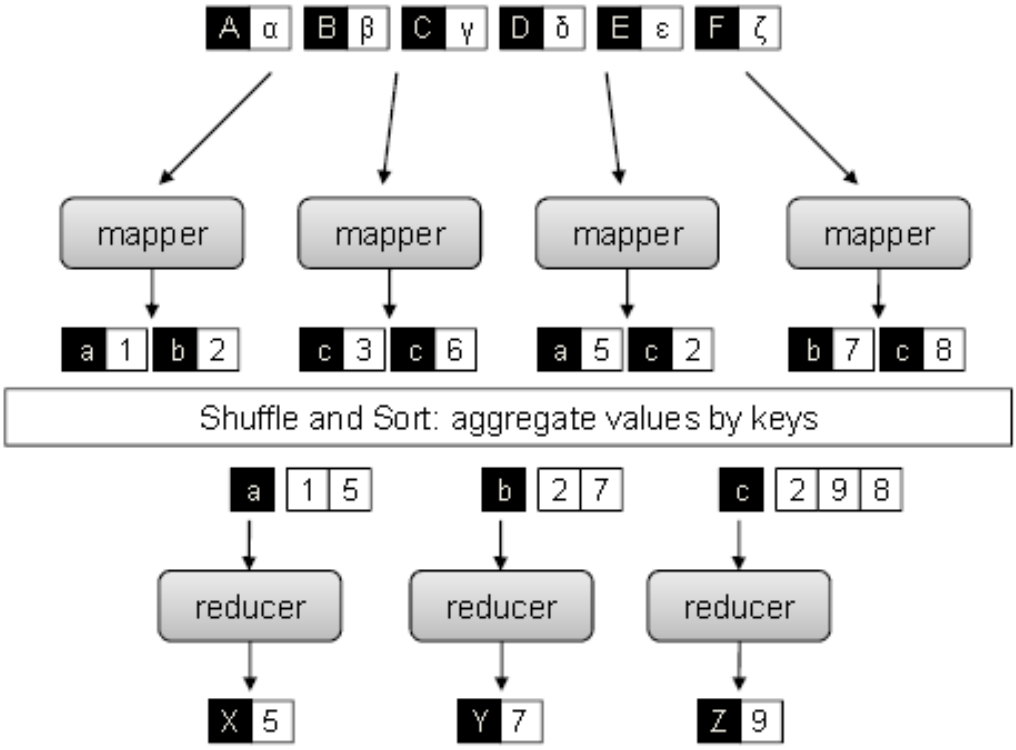
```
map      (k1, v1)          → list(k2, v2)
reduce   (k2, list(v2))    → list(v2)
```

- **Picture from refrence paper[1].**

The input to a Map Reduce job starts as data stored on the underlying distributed file system . 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. Implicit 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 persistently 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 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 Map Reduce job.

A simple word count algorithm in Map Reduce is shown:



- **Picture from refrence Book[1].**

Pseudo Code For it.

```
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 [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

- **Picture from refrence Book[1].**

3.3 The Execution Framework

One of the most important idea behind Map Reduce is separating the what of distributed processing from the how. A Map Reduce 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 configuration 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 Map Reduce job is divided into smaller units called tasks. 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 Map Reduce 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.

Data/Code Co-location: The phrase data distribution is misleading, since one of the key ideas behind Map Reduce 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 inexplicably intertwined with scheduling and relies heavily on the design of the underlying distributed file system. 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 intermediate results or otherwise exchange state information. In Map Reduce, 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 x r distinct copy operations, since each mapper may have intermediate output going to every reducer.

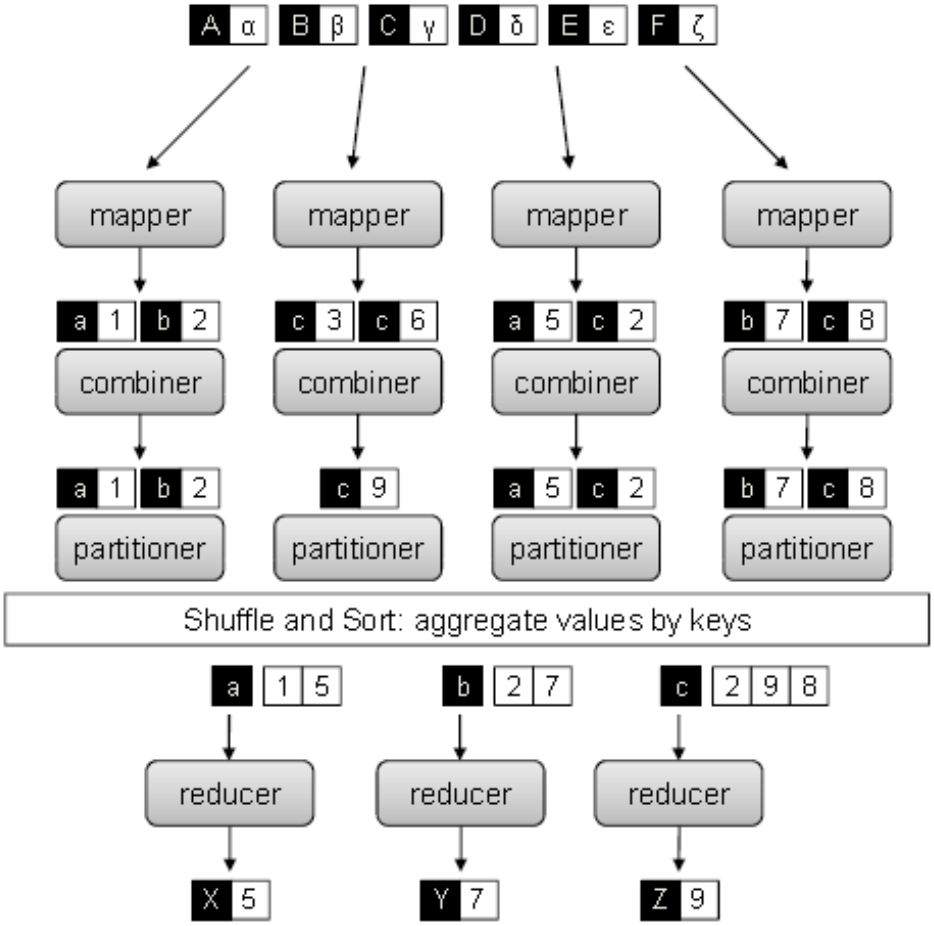
Error and Fault Handling: The Map Reduce execution framework must accomplish all the tasks above in an environment where errors and faults are the norm, not the exception. Since Map Reduce was explicitly designed around low-end commodity servers, the runtime must be especially resilient. In large clusters,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

3.4 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 partitioner 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 differences 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 Map Reduce that allow for local aggregation before the shuffle and sort phase. We can motivate the need for combiners by considering the word count algorithm, 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 mappers (and typically far smaller because each mapper may not encounter every word).

The combiner in Map Reduce 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). 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.



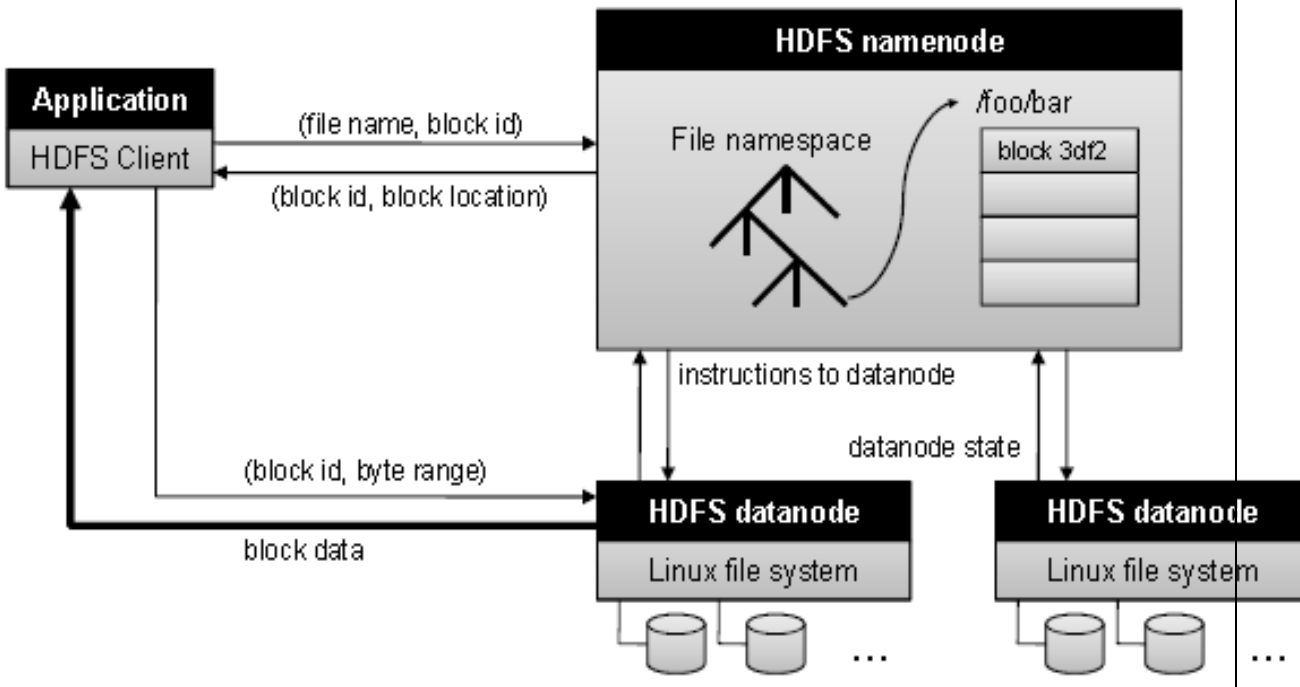
- **Picture from refrence Book[1].**

3.5 Hadoop Distributed File System (HDFS)

As dataset sizes increase, more compute capacity is required for processing. 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 Infini Band (even more expensive). In most cases, this is not a cost-effective solution, as the price of networking equipment increases non-linearly with performance (e.g., a switch with ten times the capacity 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 Map Reduce adopts exactly this approach. The Google File System (GFS) [57] supports Google's proprietary implementation of Map Reduce; in the open-source world, HDFS (Hadoop Distributed File System) is an open-source implementation of GFS that supports Hadoop. Although Map Reduce 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. The Map-Reduce distributed file system builds on previous work but is specifically adapted to large-data processing workloads, and therefore departs from previous architectures in certain respects. The main idea is to divide user data into blocks and replicate those blocks across the local disks of nodes in the cluster. Blocking 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 master 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 called by the namenode and datanodes, respectively. I adopts the Hadoop terminology, although for most basic file operations GFS and HDFS work much the same way.

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 involves transfer of metadata

The architecture of HDFS is shown in Figure.



- **Picture from refrence Book[1].**

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 periodic 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. 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.

- **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. 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.

- **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.

Chapter 4

Map Reduce Algorithm Design

A large part of the power of Map Reduce comes from its simplicity: in addition to preparing the input data, the programmer needs only to implement the mapper, 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 conceivable algorithm that a programmer wishes to develop must be expressed in terms of a small number of rigidly-defined components that must put together 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 Map Reduce algorithm design. These examples illustrate what can be thought of as "design patterns" for Map Reduce, which instantiate arrangements of components and specific techniques designed to handle frequently-encountered situations across a variety of problem domains.

Synchronization is perhaps the most tricky aspect of designing Map Reduce 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 Map-Reduce job, there is only one opportunity 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, mapper and reducers run in isolation without any mechanisms for direct communication. Furthermore, the programmer has little control over many aspects of execution, for example:

- 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.

The programmer does have a number of techniques for controlling execution and managing the flow of data in Map Reduce.

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.

4.1 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the Map Reduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The Map Reduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

4.2 Types and Example

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

```
map     (k1,v1)          → list(k2,v2)
reduce  (k2,list(v2))    → list(v2)
```

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values. Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

Here are a few simple examples of interesting programs that can be easily expressed as Map Reduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs {URL; 1}. The reduce function adds together all values for the same URL and emits a {URL; total count} pair.

Reverse Web-Link Graph: The map function outputs {target; source} pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: {target; list(source)}

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of {word; frequency} pairs. The map function emits a {hostname; term vector} pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final {hostname; term vector} pair.

Inverted Index: The map function parses each document, and emits a sequence of {word; document ID} pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a {word; list(document ID)} pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a {key; record} pair. The reduce function emits all pairs unchanged.

Chapter 5

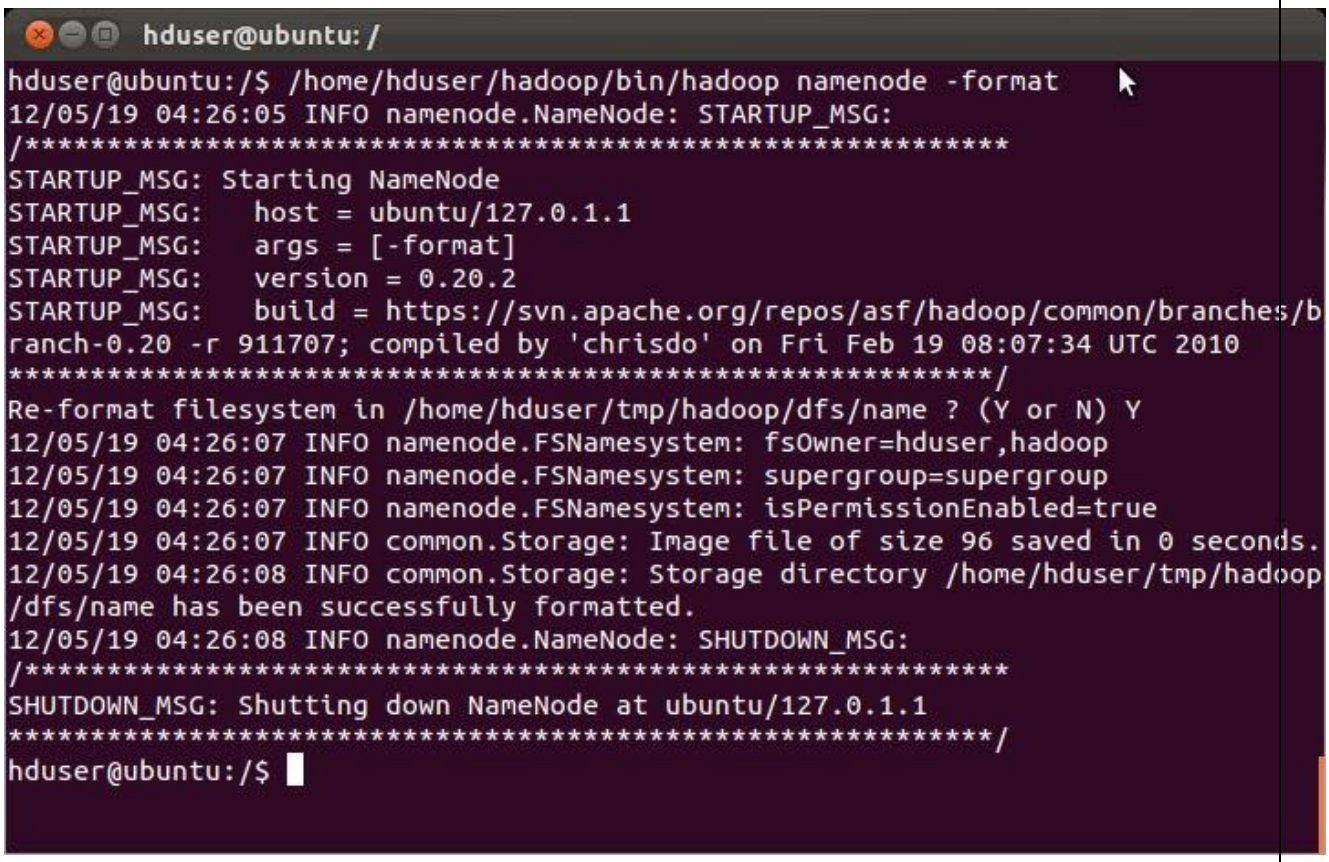
Implementation

Many different implementations of the Map Reduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.
Like I am implementing my project on a Single node Hadoop System.

1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory on my machine.

2) Data Replication is used. A set of data is replicated once, twice or more times to see the working of the datanode with rest to the namenode.

3) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available space within a cluster.



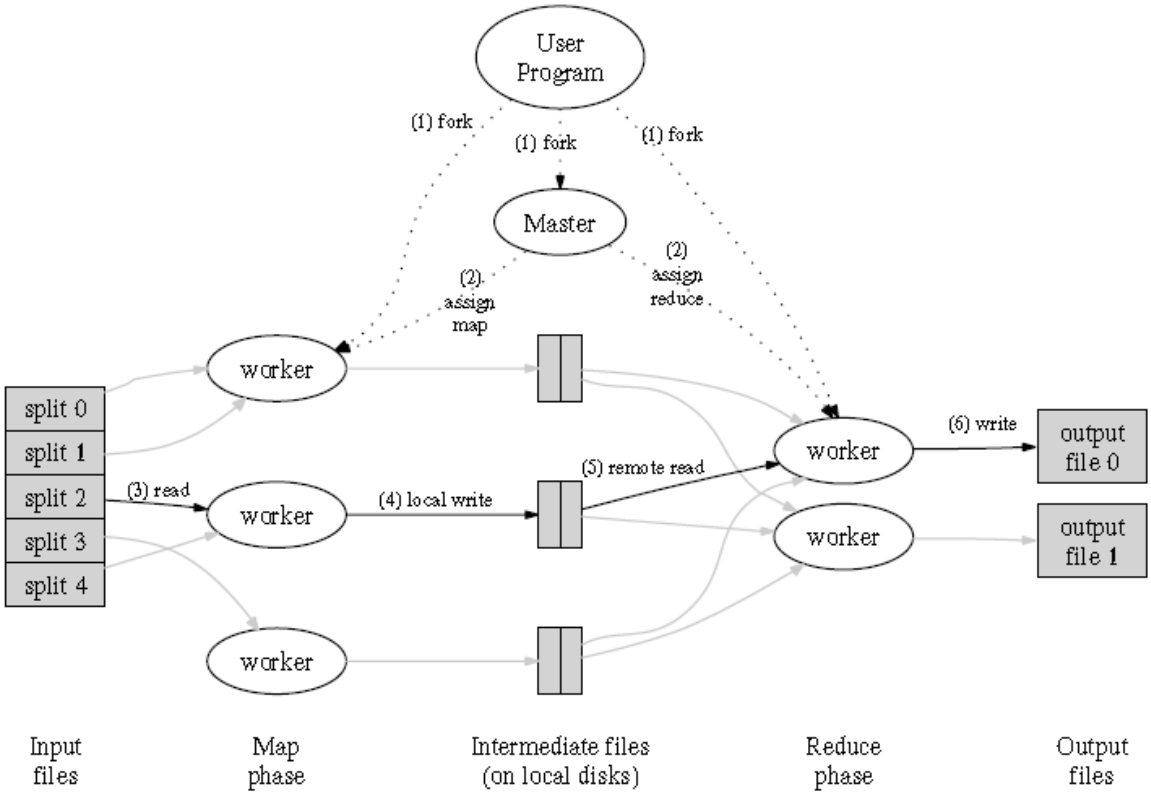
***Picture of formatting of namenode**

In the above picture , it is shown the formatting of the namenode.

It is the basic step in the execution of the program of the hadoop cluster. The formatting of the namenode helps in erasing all the previous data hold in the memory before and make the memory available for further execution

5.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data into a set of M *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R). The number of partitions (R) and the partitioning function are specified by the user. Figure shows the overall flow of a Map Reduce operation in our implementation. When the user program calls the Map Reduce function, the following sequence of actions occurs (the numbered labels in Figure correspond to the numbers in the list below):



- **Picture from refrence paper[1].**

1) The MapReduce library in the user program First splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2) One of the copies of the program is special : the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3) A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4) Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5) When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

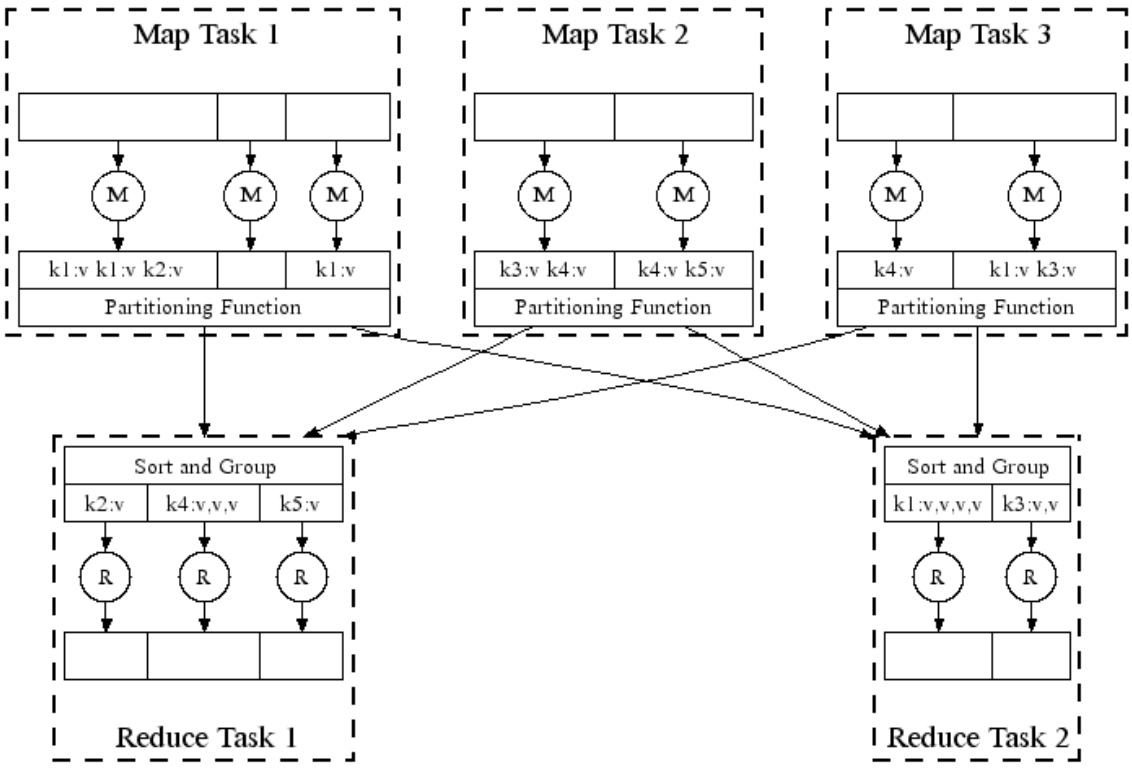
6) The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.

7) When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the Map Reduce call in the user program returns back to the user code.

5.2 Master Data Structure

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks). The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

Parallel Execution



- **Picture from refrence Book[1].**

5.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

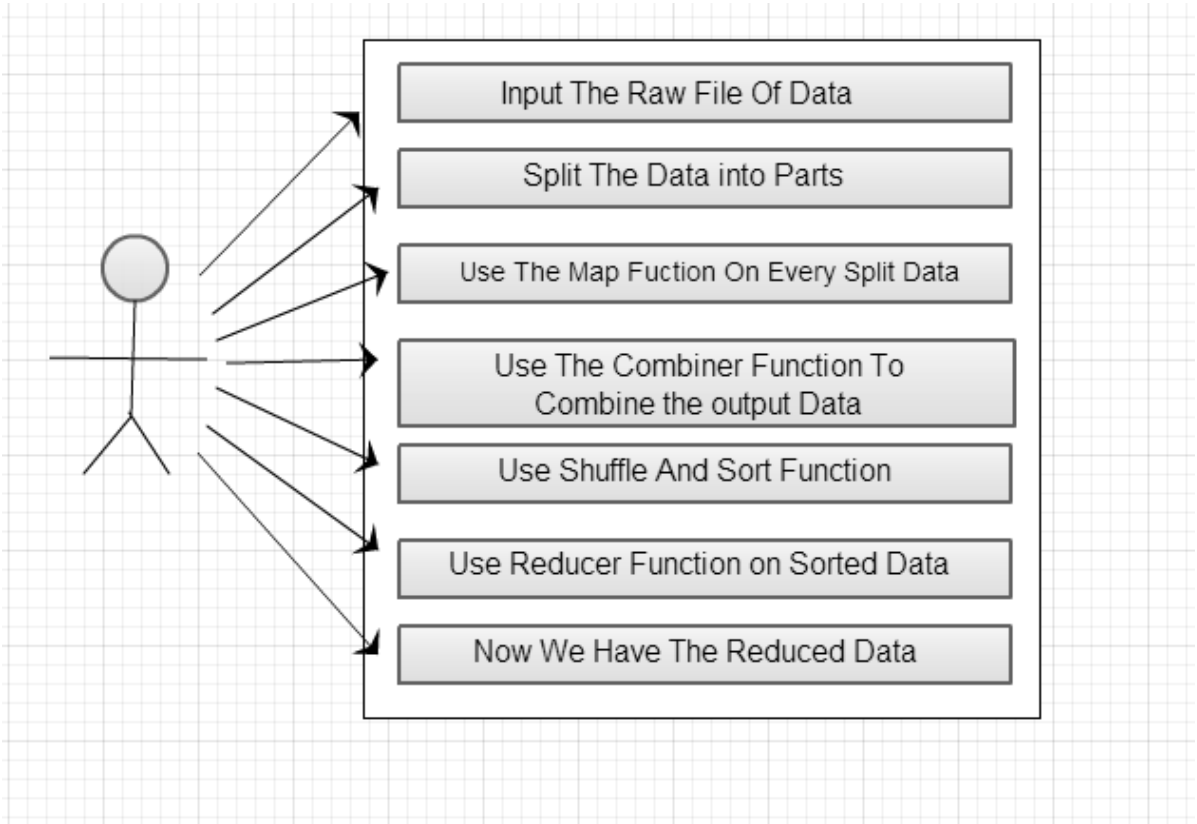
Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling. Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system. When a map task is executed first by worker A and then later executed by worker B (because A failed), all workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker A will read the data from worker B. Map Reduce is resilient to large-scale worker failures. For example, during one Map Reduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The Map Reduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the Map Reduce operation.

Master Failure

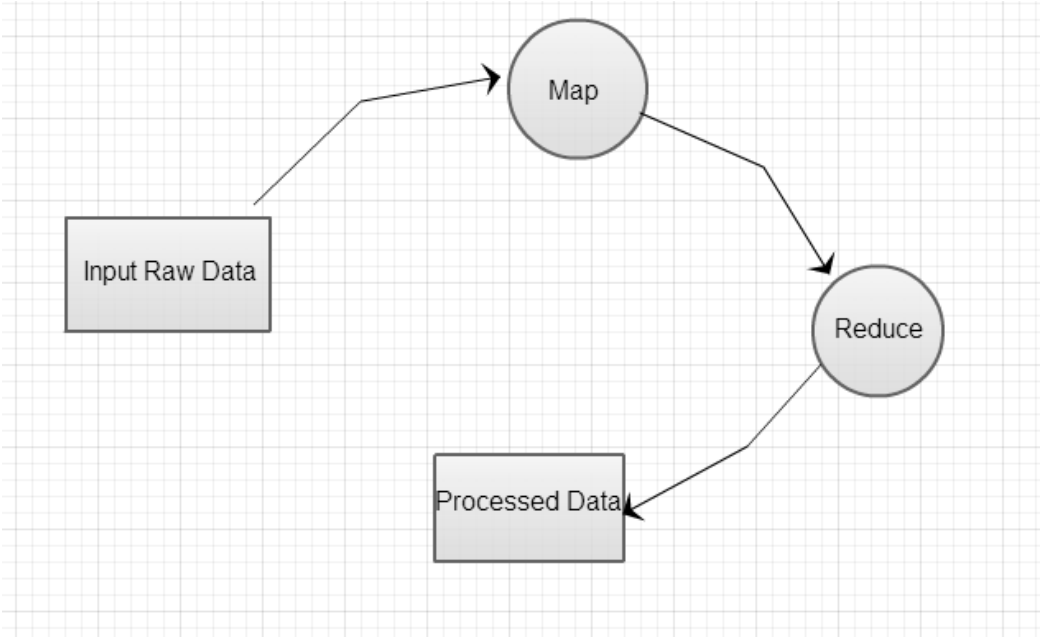
It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

5.4 Use Case Diagram

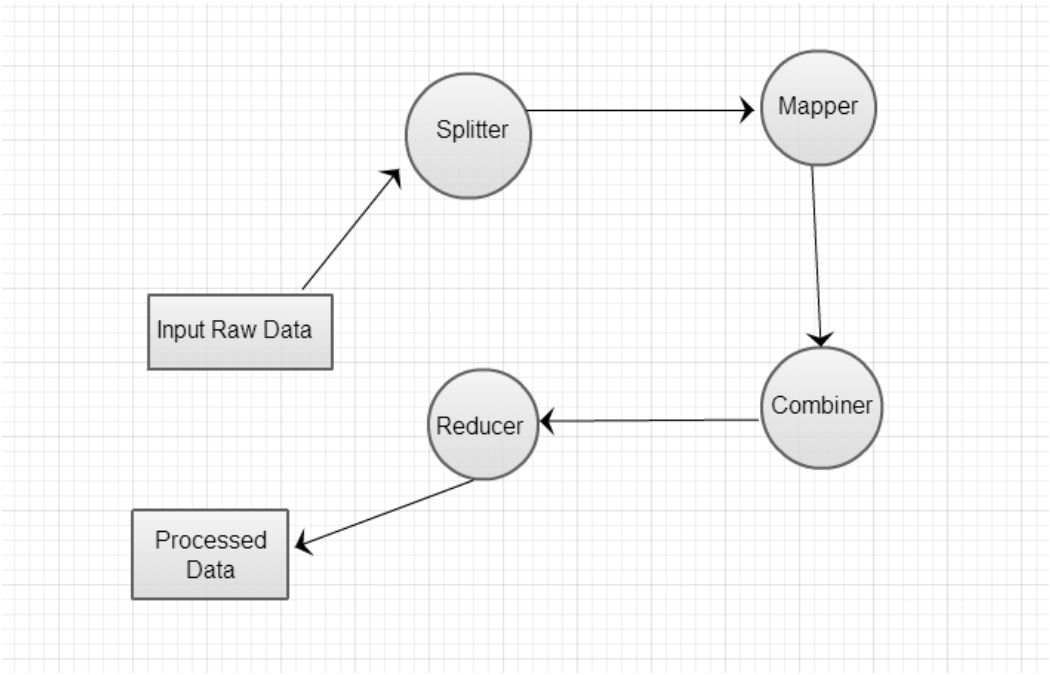


5.5 Data Flow Diagram

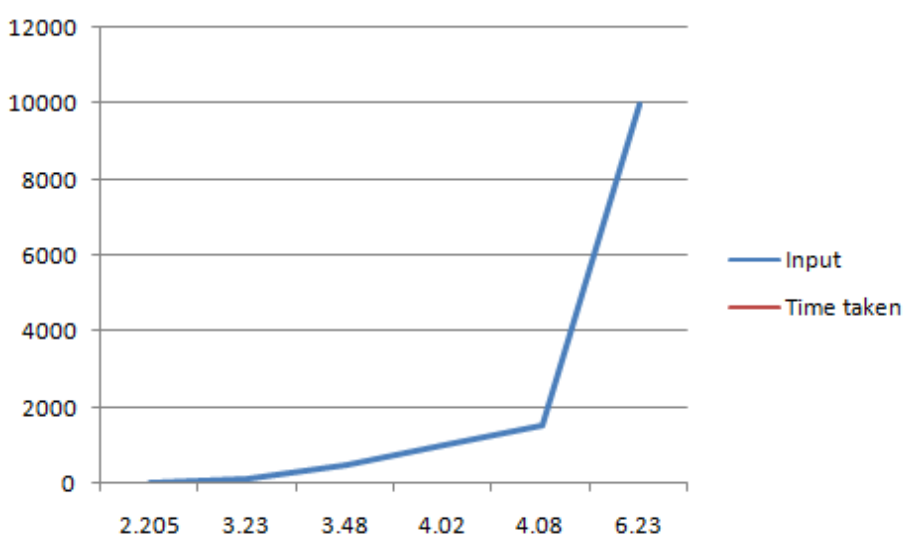
Level 1



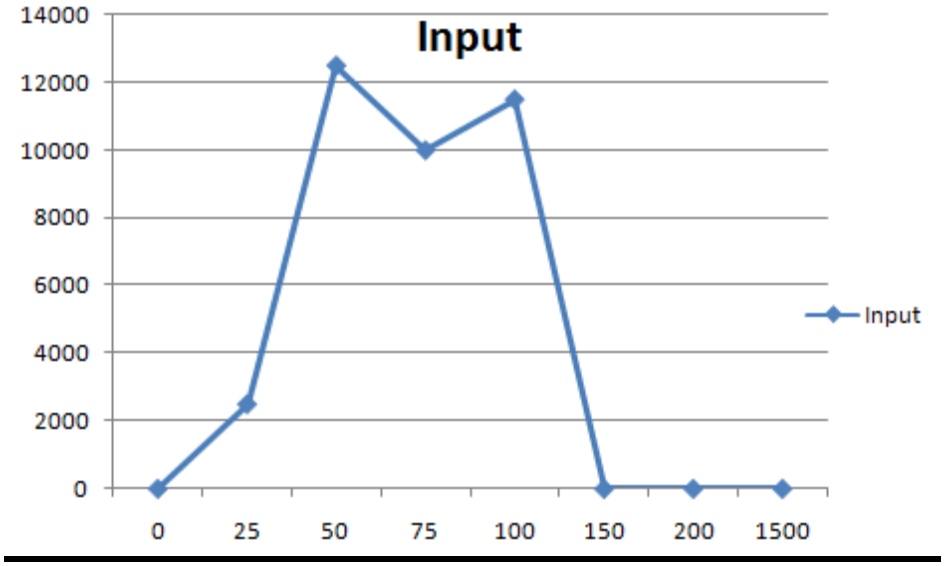
Level 2



5.6 Analysis



The above analysis shows that for a given size of data file the amount of time a map reduce function take to do the overall computing of the data.



Data transfer rate of the input file

Chapter 6: Conclusion

The need to process enormous quantities of data has never been greater. Not only are terabyte- and petabyte-scale datasets rapidly becoming commonplace, but there is consensus that great value lies buried in them, waiting to be unlocked by the right computational tools. In the commercial sphere, business intelligence, driven by the ability to gather data from a dizzying array of sources, promises to help organizations better understand their customers and the marketplace, hopefully leading to better business decisions and competitive advantages. For engineers building information processing tools and applications, larger datasets lead to more effective algorithms for a wide range of tasks, from machine translation to spam detection. In the natural and physical sciences, the ability to analyze massive amounts of data may provide the key to unlocking the secrets of the cosmos or the mysteries of life. In the preceding chapters, we have shown how Map Reduce can be exploited to solve a variety of problems related to text processing at scales that would have been unthinkable a few years ago. However, no tool, no matter how powerful or flexible can be perfectly adapted to every task, so it is only fair to discuss the limitations of the Map Reduce programming model and survey alternatives.

Hadoop Map Reduce is a large scale, open source software framework dedicated to scalable, distributed, data-intensive computing

• The framework breaks up large data into smaller parallelizable chunks and handles scheduling

▫ Maps each piece to an intermediate value

▫ Reduces intermediate values to a solution

▫ User-specified partition and combiner options

• Fault tolerant, reliable, and supports thousands of nodes and petabytes of data

• If you can rewrite algorithms into Maps and Reduces, and your problem can be broken up into small pieces solvable in parallel, then Hadoop's Map Reduce is the way to go for a distributed problem solving approach to large datasets

• Tried and tested in production

• Many implementation options

6.1 Closing Remarks

The need to process enormous quantities of data has never been greater. Not only are terabyte- and petabyte-scale datasets rapidly becoming commonplace, but there is consensus that great value lies buried in them, waiting to be unlocked by the right computational tools. In the commercial sphere, business intelligence driven by the ability to gather data from a dizzying array of sources promises to help organizations better understand their customers and the marketplace, hopefully leading to better business decisions and competitive advantages. For engineers building information processing tools and applications, larger datasets lead to more effective algorithms for a wide range of tasks, from machine translation to spam detection. In the natural and physical sciences, the ability to analyze massive amounts of data may provide the key to unlocking the secrets of the cosmos or the mysteries of life.

In the preceding chapters, we have shown how MapReduce can be exploited to solve a variety of problems related to text processing at scales that would have been unthinkable a few years ago. However, no tool no matter how powerful or exible can be perfectly adapted to every task, so it is only fair to discuss the limitations of the MapReduce programming model and survey alternatives. Section 7.1 covers online learning algorithms and Monte Carlo simulations, which are examples of algorithms that require maintaining global state. As we have seen, this is difficult to accomplish in MapReduce.

6.2 Limitation of Map Reduce

As we have seen throughout, solutions to many interesting problems in text processing do not require global synchronization. As a result, they can be expressed naturally in Map Reduce, since map and reduce tasks run independently and in isolation. However, there are many examples of algorithms that depend crucially on the existence of shared global state during processing, making them difficult to implement in Map Reduce (since the single opportunity for global synchronization in Map Reduce is the barrier between the map and reduce phases of processing).

The first example is online learning. The concept of learning as the setting of parameters in a statistical model. Both EM and the gradient-based learning algorithms we described are instances of what are known as batch learning algorithms. This simply means that the full \batch" of training data is processed before any updates to the model parameters are made. On one hand, this is quite reasonable: updates are not made until the full evidence of the training data has been weighed against the model. An earlier update would seem, in some sense, to be hasty. However, it is generally the case that more frequent updates can lead to more rapid convergence of the model (in terms of number of training instances processed), even if those updates are made by considering less data [24]. Thinking in terms of gradient optimization online learning algorithms can be understood as computing an approximation of the true gradient, using only a few training instances. Although only an approximation, the gradient computed from a small subset of training instances is often quite reasonable, and the aggregate behavior of multiple updates tends to even out errors that are made. In the limit, updates can be made after every training instance. Unfortunately, implementing online learning algorithms in MapReduce is problematic.

The model parameters in a learning algorithm can be viewed as shared global state, which must be updated as the model is evaluated against training data. All processes performing the evaluation (presumably the mappers) must have access to this state. In a batch learner, where updates occur in one or more reducers (or, alternatively, in the driver code), synchronization of this resource is enforced by the Map Reduce framework. However, with online learning, these updates must occur after processing smaller numbers of instances. This means that the framework must be altered to support faster processing of smaller datasets, which goes against the design choices of most existing Map Reduce implementations. Since Map Reduce was specifically optimized for batch operations over large amounts of data, such a style of computation would likely result in inefficient use of resources. In Hadoop, for example, map and reduce tasks have considerable startup costs. This is acceptable because in most circumstances, this cost is amortized over the processing of many key-value pairs.

However, for small datasets, these high startup costs become intolerable. An alternative is to abandon shared global state and run independent instances of the training algorithm in parallel (on different portions of the data). A final solution is then arrived at by merging individual results. Experiments, however, show that the merged solution is inferior to the output of running the training algorithm on the entire dataset [52]. A related difficulty occurs when running what are called Monte Carlo simulations, which are used to perform inference in probabilistic models where evaluating or representing the model exactly is impossible. The basic idea is quite simple: samples are drawn from the random variables in the model to simulate its behavior, and then simple frequency statistics are computed over the samples. This sort of inference is particularly useful when dealing with so-called nonparametric models, which are models whose structure is not specified in advance, but is rather inferred from training data. For an illustration, imagine learning a hidden Markov model, but inferring the number of states, rather than having them specified. Being able to parallelize Monte Carlo simulations would be tremendously valuable, particularly for unsupervised learning applications where they have been found to be far more effective than EM-based learning (which requires specifying the model).

Chapter 7

References, IEEE Format

Book

[1] Jimmy Lin and Chris Dyer, Data-Intensive Text Processing with Map Reduce.

Research Paper

[2] Jeffrey Dean and Sanjay Ghemawat, "Map Reduce: Simplified Data Processing on Large Clusters", *Google, Inc.*

Web References

[3] http://hadoop.apache.org/

[4] http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

[5] http://en.wikipedia.org/wiki/MapReduce

[6] www.cse.buffalo.edu/~okennedy/courses/.../2.1-MapReduce.pptx

[7] www.cs.colorado.edu/~kena/classes/5448/s11/presentations/hadoop.pdf

[8] lintool.github.io/MapReduceAlgorithms/

Software

[9] Creately, for diagrams.

Appendix A

Technology And Tools Used :

1. Hadoop :

Apache Hadoop is an open-source software framework written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures (of individual machines, or racks of machines) are commonplace and thus should be automatically handled in software by the framework.

The core of Apache Hadoop consists of a storage part (Hadoop Distributed File System (HDFS)) and a processing part (MapReduce). Hadoop splits files into large blocks and distributes them amongst the nodes in the cluster. To process the data, Hadoop MapReduce transfers packaged code for nodes to process in parallel, based on the data each node needs to process. This approach takes advantage of data locality[3]—nodes manipulating the data that they have on hand—to allow the data to be processed faster and more efficiently than it would be in a more conventional supercomputer architecture that relies on a parallel file system where computation and data are connected via high-speed networking.[4]

The base Apache Hadoop framework is composed of the following modules:

- *Hadoop Common* – contains libraries and utilities needed by other Hadoop modules;
- *Hadoop Distributed File System (HDFS)* – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- *Hadoop YARN* – a resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications;[5][6] and

- *Hadoop MapReduce* – a programming model for large scale data processing.

The term "Hadoop" has come to refer not just to the base modules above, but also to the "ecosystem", or collection of additional software packages that can be installed on top of or alongside Hadoop, such as Apache Pig, Apache Hive, Apache HBase, Apache Spark, and others.

Apache Hadoop's MapReduce and HDFS components were inspired by Google papers on their MapReduce and Google File System.

The Hadoop framework itself is mostly written in the Java programming language, with some native code in C and command line utilities written as Shell script. For end-users, though MapReduce Java code is common, any programming language can be used with "Hadoop Streaming" to implement the "map" and "reduce" parts of the user's program.[11] Other related projects expose other higher-level user interfaces.

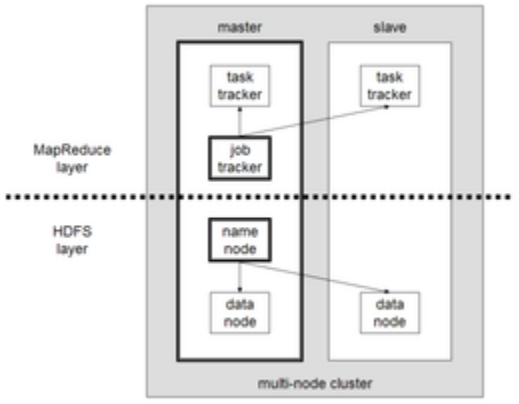
Prominent corporate users of Hadoop include Facebook and Yahoo. It can be deployed in traditional on-site datacenters but has also been implemented in public cloud spaces such as Microsoft Azure, Amazon Web Services, Google App Engine and IBM Bluemix.

Apache Hadoop is a registered trademark of the Apache Software Foundation.

Hadoop consists of the *Hadoop Common* package, which provides filesystem and OS level abstractions, a MapReduce engine (either MapReduce/MR1 or YARN/MR2)[16] and the Hadoop Distributed File System (HDFS). The Hadoop Common package contains the necessary Java ARchive (JAR) files and scripts needed to start Hadoop. The package also provides source code, documentation, and a contribution section that includes projects from the Hadoop Community.

For effective scheduling of work, every Hadoop-compatible file system should provide location awareness: the name of the rack (more precisely, of the network switch) where a worker node is. Hadoop applications can use this information to run work on the node where

the data is, and, failing that, on the same rack/switch, reducing backbone traffic. HDFS uses this method when replicating data to try to keep different copies of the data on different racks. The goal is to reduce the impact of a rack power outage or switch failure, so that even if these events occur, the data may still be readable.



A multi-node Hadoop cluster

A small Hadoop cluster includes a single master and multiple worker nodes. The master node consists of a JobTracker, TaskTracker, NameNode and DataNode. A slave or *worker node* acts as both a DataNode and TaskTracker, though it is possible to have data-only worker nodes and compute-only worker nodes. These are normally used only in nonstandard applications.

Hadoop requires Java Runtime Environment (JRE) 1.6 or higher. The standard startup and shutdown scripts require that Secure Shell (ssh) be set up between nodes in the cluster.

In a larger cluster, the HDFS is managed through a dedicated NameNode server to host the file system index, and a secondary NameNode that can generate snapshots of the namenode's memory structures, thus preventing file-system corruption and reducing loss of data. Similarly, a standalone JobTracker server can manage job scheduling. In clusters where the Hadoop MapReduce engine is deployed against an alternate file system, the NameNode,

secondary NameNode, and DataNode architecture of HDFS are replaced by the file-system-specific equivalents.

The **Hadoop distributed file system** (**HDFS**) is a distributed, scalable, and portable file-system written in Java for the Hadoop framework. A Hadoop cluster has nominally a single namenode plus a cluster of datanodes, although redundancy options are available for the namenode due to its criticality. Each datanode serves up blocks of data over the network using a block protocol specific to HDFS. The file system uses TCP/IP sockets for communication. Clients use remote procedure call (RPC) to communicate between each other.

HDFS stores large files (typically in the range of gigabytes to terabytes[20]) across multiple machines. It achieves reliability by replicating the data across multiple hosts, and hence theoretically does not require RAID storage on hosts (but to increase I/O performance some RAID configurations are still useful). With the default replication value, 3, data is stored on three nodes: two on the same rack, and one on a different rack. Data nodes can talk to each other to rebalance data, to move copies around, and to keep the replication of data high. HDFS is not fully POSIX-compliant, because the requirements for a POSIX file-system differ from the target goals for a Hadoop application. The trade-off of not having a fully POSIX-compliant file-system is increased performance for data throughput and support for non-POSIX operations such as Append.[21]

HDFS added the high-availability capabilities, as announced for release 2.0 in May 2012,[22] letting the main metadata server (the NameNode) fail over manually to a backup. The project has also started developing automatic fail-over.

The HDFS file system includes a so-called *secondary namenode*, a misleading name that some might incorrectly interpret as a backup namenode for when the primary namenode goes offline. In fact, the secondary namenode regularly connects with the primary namenode and builds snapshots of the primary namenode's directory information, which the system then saves to local or remote directories. These checkpointed images can be used to restart a failed

primary namenode without having to replay the entire journal of file-system actions, then to edit the log to create an up-to-date directory structure. Because the namenode is the single point for storage and management of metadata, it can become a bottleneck for supporting a huge number of files, especially a large number of small files. HDFS Federation, a new addition, aims to tackle this problem to a certain extent by allowing multiple namespaces served by separate namenodes.

An advantage of using HDFS is data awareness between the job tracker and task tracker. The job tracker schedules map or reduce jobs to task trackers with an awareness of the data location. For example: if node A contains data (x,y,z) and node B contains data (a,b,c), the job tracker schedules node B to perform map or reduce tasks on (a,b,c) and node A would be scheduled to perform map or reduce tasks on (x,y,z). This reduces the amount of traffic that goes over the network and prevents unnecessary data transfer. When Hadoop is used with other file systems, this advantage is not always available. This can have a significant impact on job-completion times, which has been demonstrated when running data-intensive jobs.

For starters, let's take a quick look at some of those terms and what they mean.

- **Open-source software.** Open source software differs from commercial software due to the broad and open network of developers that create and manage the programs. Traditionally, it's free to download, use and contribute to, though more and more commercial versions of Hadoop are becoming available.
- **Framework.** In this case, it means everything you need to develop and run your software applications is provided – programs, tool sets, connections, etc.
- **Distributed.** Data is divided and stored across multiple computers, and computations can be run in parallel across multiple connected machines.
- **Massive storage.** The Hadoop framework can store huge amounts of data by breaking the data into blocks and storing it on clusters of lower-cost commodity hardware.

How did Hadoop get here?

As the World Wide Web grew at a dizzying pace in the late 1900s and early 2000s, search engines and indexes were created to help people find relevant information amid all of that text-based content. During the early years, search results were returned by humans. It's true! But as the number of web pages grew from dozens to millions, automation was required. Web crawlers were created, many as university-led research projects, and search engine startups took off (Yahoo, AltaVista, etc.).

One such project was Nutch – an open-source web search engine – and the brainchild of Doug Cutting and Mike Cafarella. Their goal was to invent a way to return web search results faster by distributing data and calculations across different computers so multiple tasks could be accomplished simultaneously. Also during this time, another search engine project called Google was in progress. It was based on the same concept – storing and processing data in a distributed, automated way so that more relevant web search results could be returned faster.

In 2006, Cutting joined Yahoo and took with him the Nutch project as well as ideas based on Google's early work with automating distributed data storage and processing. The Nutch project was divided. The web crawler portion remained as Nutch. The distributed computing and processing portion became Hadoop (named after Cutting's son's toy elephant). In 2008, Yahoo released Hadoop as an open-source project, and, today Hadoop's framework and family of technologies are managed and maintained by the non-profit Apache Software Foundation (ASF), a global community of software developers and contributors.