Introduction to Big Data

Dr. Osama Ghoneim

2022-2023

Chapter 1

Introduction to Big Data

1.1 Introduction

The term ”Big Data” has become one of the most famous aspects in this era, but there is no specific definition of Big Data. Usually, a lot of data science experts would like to define the procedure of Extract, Transform, and Load for massive quantity of data as the concept of Big Data [3]. The common elaboration of the Big Data is based on the most common three features of datasets: velocity, variety , and volume (or 3Vs). However, it does not hold all of the Big Data features precisely. To present a complete connotation of Big Data, an investigation of the Big Data term from a historical sight will be presented to see how Big Data aspect has been developing from yesterdays sense to todays meaning [2, 4]. Historically, the term Big Data is ill-defined and quite vague. It is not a precise term and does not give exact sense rather than the notion of its size. The idiom Big is very elastic. The question how Big is Big and how Small is small is related to the time, space and a circumstance [5]. From an advanced perspective, thesize of Big Data is evolving in a very short time [1]. By using the modern capacity of Internet traffic as a measuring scale, the volumes of Big Data will be in between Zettabyte (ZB or 1021or 270) and Terabyte (TB or 1012 or 240) range. Based on historical data traffic growth rate, Cisco pretended that human has came in the ZB era in 2015 [2]. Looking to the distinct averagesizes of data files given in Table 1.1 we will understand significance of the data volumes impact.

The major purpose of this chapter is to give a historical overview of Big Data aspects and to discuss that Big Data is not just 3Vs, but rather 11Vs [6]. These additional features of Big Data

TABLE 1.1: Sizes of Distinct Sizes of Data Files [2].

Media	Average Size of Data	Observed (2014)
Movie	100 - 120 GB	60 frames per second (Full High Definition, MPEG-4 format, 2 hours)
Song	3.5 - 8.5 MB	average 1.9 MB/per minute(MP3) 256 Kbps rate (3 mins)
eBook	1 - 5 MB	200-350 pages
Web Page	1.6 - 2 MB	average 100 objects

indicate the actual inspiration behind Big Data Analytics (BDA) [1]. These extended attributeswill help data scientists to give answers for some basic questions about the real meaning of BDA: what problems can be addressed using Big Data , and what are the problems should be considered to clarify the BDA concept. These aspects are discussed in this chapter through analysis of historical evolution.

the remaining of this chapter will be arranged as the following:

1. Historical Overview of Big Data.
2. Big Data Definition.
3. Sources of Big Data.

4. Where Big Data will be used?
5. Literature Survey for Big Data Mining.
6. Challenging Issues for Big Data Mining.
7. Problem Statement.
8. Research Objectives.
9. Thesis Organization.

1.2 Historical Overview of Big Data

To capture the core of Big Data, we present the history and origin of BDA and then provide an accurate definition of BDA.

1.2.1 Big Data Origin

Many studies have been conducting on developments and historical reviews in BDA area. A short historical overview of Big Data starting from 1944 introduced by Gil Press [7] based upon Riders search [8]. This work provides the evolution of Big Data starting from 1944 to 2012. The study indicated that the border among the expansion of data and Big Data became unclear. In [7]a comprehensive study covering all BDA and Data Science events until the end of 2013 has been introduced.

The origin of the Big Data established by Frank Ohlhorst [9] in 1880 when the US was doing its10th census, in the 19th century, The greatest problem faced by the US was the statistics aspects,these aspects were, how to document and survey and 50 million citizens of North-American. On the other hand, Winshuttle [2] consider the foundation of Big Data in the 19th century. They presented the Big Data as the datasets which are so complicated and beyond conventional management and process capability. In addition, Winshuttles review presents an estimation for the growth of data upto 2020.

Bernard Marr presents the longest historical survey [1]. This survey concentrates on the historic bases of the Big Data, which are several techniques for people to collect, store, analyze data. Marr showed that Erik [10] as the pioneer of the Big Data. Erik presented an article in the Magazine of Harper and this article reprinted also in the Washington post-1989 presenting afirst description of the Big Data concept [1].

A great discussion a bout Big data origin has been presented by Steve Lohr [11]. In this discussion he was trying to identify when the Big Data expression had been used for the first time.It was difficult to hunt the source of Big Data because the expression of the Big Data is so generic[12] . Instead, the aim was the initial usage of the Big Data term that recommends its current understanding that is, not only the massive quantity of data, but also distinct forms of data managed in modern ways. Cox and Ellsworth presented the origin of Big Data from another pointof view. They introduced a relatively precise significance to the current view of Big Data, which they defined datasets are too huge to be manged in the capacities of local disk, basic memory, and even remote disk. This problem is considered as Big Data [13].

A great visualizing historical survey was conducted [2]. This review dedicated to the time-line of the implementation of BDA. This historic discussion is basically specified by actions and eventsrelated to Big Data pushed by several IT association like Google, Facebook, Yahoo, Youtube, Twitter, and Apple. Figure 1.1 shows the history of Big Data and Big Data Analytics tools.

FIGURE 1.1: A Brief History of Big Data [1].

1.3 Big Data Definition

1.3.1 Gartner- 3Vs Definition

Starting from 1997, a lot of characteristics have been combined to Big Data. The most common three characteristics, which they have been popularly adopted and cited: Gartners interpretation of 3Vs was the first one. The origin of this concept created in February 2001 [14]. According tothis interpretation, data is increasing with three basic features, namely:

1. **Volume**, means that the data generated in cumulative and stream volume.
2. **Velocity**, denotes to the rate of generating data.
3. **Variety**, represents the diversity of inconsistent and incompatible data structures.

This 3Vs definition has been popularly considered as the common characteristics of Big Data.

1.3.2 IBM- 4Vs Definition

Another characteristic of Big Data Veracity which has been added to 3Vs definition by Gartner,and results with IBM new 4Vs definition of Big Data. It defines each V as following [15, 16]:

1. **Volume** refers to the data scale.
2. **Velocity** stands for analysing data streams.
3. **Variety** implies the distinct format of data.
4. **Veracity** infers uncertainty of data.

The new V veracity characteristic was added because of the need to the quality issues of the big data sources requested by customers began facing with the initiatives of Big Data [17].

1.3.3 Microsoft- 6Vs Definition

To maximize the business value of Big Data, Microsoft expanded the 3Vs definition of Big Datato 6 Vs by adding three more Vs, which are Veracity, Variability, and Visibility [18]:

- **Veracity** concentrates on credibility of data sources.
- **Variability** denotes to the complication of datasets.
- **Visibility** confirms that there is a need for a complete picture of data to take better decisions.

1.3.4 More Vs for Big Data

Since 2001, Laney [19] introduced the 3Vs definition of Big Data. Data scientists added more characteristics or Vs to Big Data. The number Vs for Big Data becomes eleven [19].In 2013 Yuri Demchenko presents 5 Vs Big Data definition. Demchenko [20] added the value dimension to the 4Vs definition presented by IBM (see Figure 1.2). As well as, more additional Vs like Venue, Variability, Vocabulary, and Vageness have been added to the Big Data definition in order to clearify and understand Big Data concept [20].

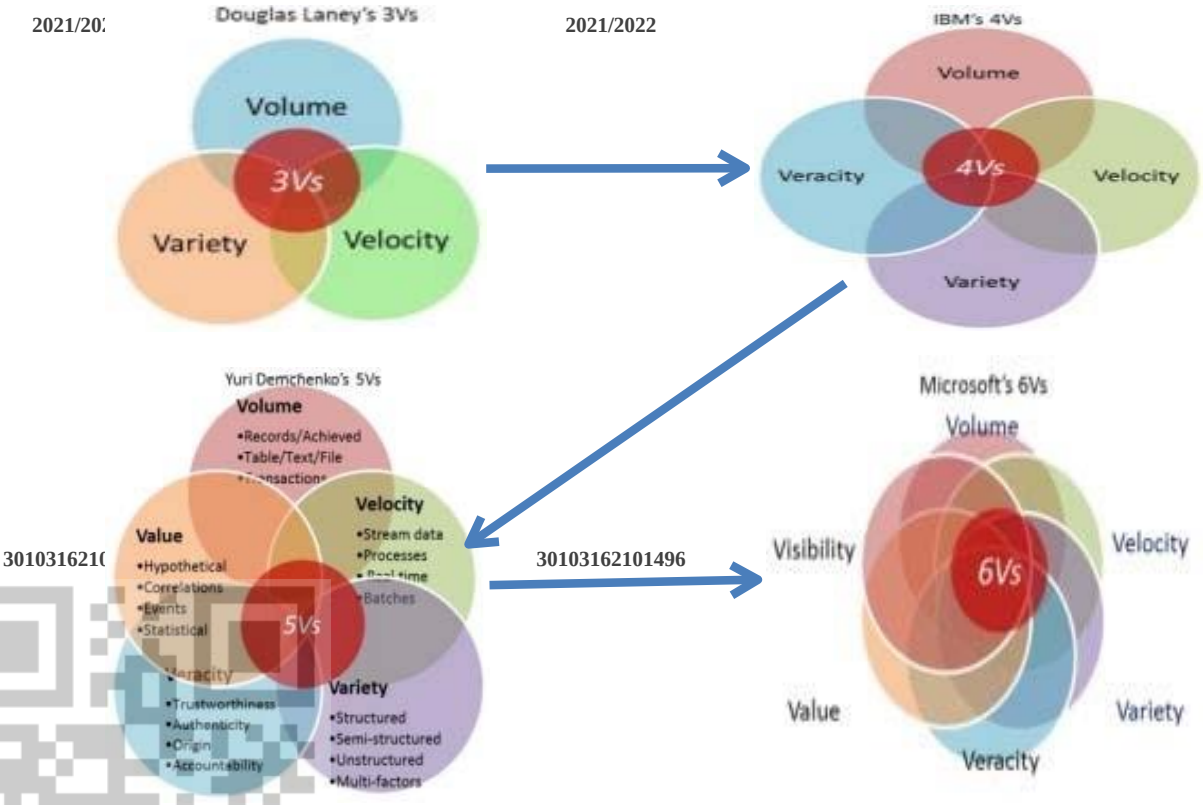


FIGURE 1.2: 3Vs, 4Vs, 5Vs and 6Vs Definitions of Big Data

1.4 Sources of Big Data

Now, Big Data can be found in many scenarios like RFID generated data, web logs, sensor networks, satellite and geo-spatial data, Internet text and, social data from social networks, Internet search indexing, call detail records, Internet text and documents, atmospheric science, astronomy, medical records, genomics, biogeochemical, biological, and other complex and/or military surveillance, interdisciplinary scientific research, video archives, photography archives, and large-scale e-commerce [21].

Firstly, Considering Wal-Mart [22] manages 1 million transactions every hour, which is collected in huge databases, and predicted to include more than 2.5 petabytes of data. The internet is considered as the greatest source for Big Data as the volume of data generated in one minute on the internet in 2017 is so huge (Figure 1.3). A newly study anticipated that every 60 seconds, Google is receiving more than 3500000 queries, more than 150 million e-mail messages will be sent, more than 4 million videos will be watched in YouTube and 72 hours of video will be uploaded to it, users of Facebook will share more than 2 million pieces of content, and users of Twitter are generating more than 452,000 tweets [22].

A very important source of big data is IoT [23]. Internet of Things (IoT) is a concept and a model that considers the spread presence in the environment or a variety or objects /things that via wired and wireless links and single addressing schemes are capable to cooperate and interact with each other or with other things in order to produce modern services/applications [24, 25]. The world where the real, Digital and the virtual are converging to create smart environments that make energy, transport, cities and many other areas more intelligent. The aim of the IoT is to allow things to be linked at any place, anytime, with anyone and anything through any path/network and any service. Objects in IoT make themselves identifiable and they become more intelligence by making or allowing context related decisions about the reality that they can interconnect information about themselves and they can contact information that has been collected by other things, or they can be components, objects, or complex services [25].

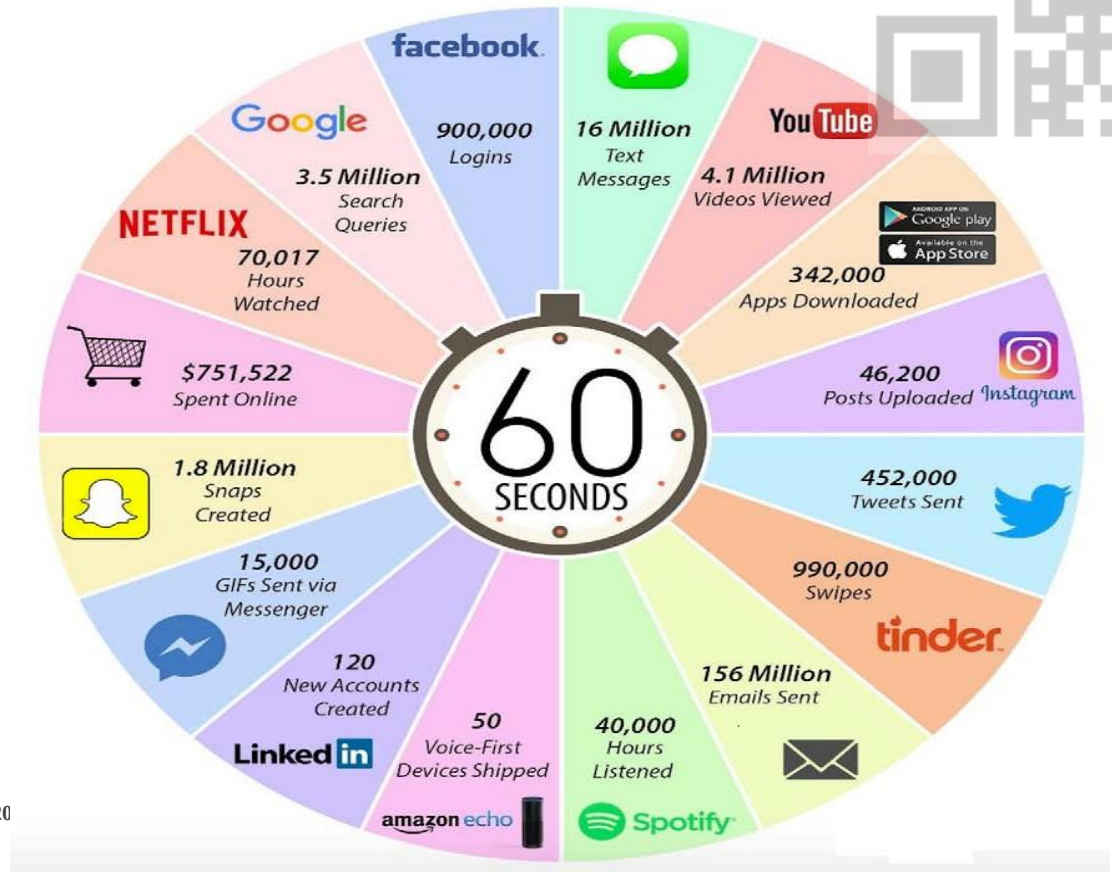


FIGURE 1.3: One Minute in Internet 2017

1.5 Big Data Applications

As stated by The McKinsey Global Institute (MGI), financial value of Big Data can be generated across many sectors like: Health care, Public sector administration, Manufacturing, Social personal/professional data [23]. Some of Big Data use Cases are :

- Energy sector.
- Sentiment analysis.
- Log analytsis.
- Smart city and smart home.

- Environment and weather prediction.
- Risk modeling and management.

1.6 Literature Survey for Big Data Mining

Data Mining refers to the procedure of exploring valuable knowledge like patterns, associations, anomalies, changes, and critical components from huge volumes of data which has been accumulated in data warehouses, databases, or any other data repository [26]. Also, data mining is classified as a very powerful tool to discover hidden relationships in a large quantities of data. Now, the modern aspect of Big Data is used to recognize the datasets, which are of a vast size and have higher complexity [27]. So the conventional data mining platforms or methodologies cannot manage, store, and analyze these huge quantities of data . Big data contains composite collections of both unstructured and structured data types. Now, much of the data science efforts are chiefly interested in handling unstructured types of data. Mining of Big Data refers to the ability to extract useful information from these massive datasets, this was very difficult before due to its variety, volume, velocity, and veracity.

Knowledge extraction is very useful and the knowledge mining is the representation of various types of patterns in meaningful way. One of the most important tasks for Data Mining is to analyze data from different perspectives and summarize it into beneficial information that can be utilized as good solutions in business and trying to predict the future direction. Information Mining helps the associations to take better knowledge-driven decisions. Data Mining is knownas Knowledge Discovery in Databases (KDD) [28]. Data Mining make use of many computa- tional methods from information retrieval, statistics, , machine learning and pattern recognition. Data Mining tasks can be categorized into summarization, classification, clustering, and associ- ation analysis [28].

With tremendous technological advances, which led to an increase in data storage capabilities, processing methods and data availability was the basic reason for the growth and emergence of Big Data. Big Data technology attempting to present good business serves and helping in

decision making via handling, managing, and processing huge datasets (these datasets may be private, general, and enterprise specific). We can define the process of Mining Big Data as the activity of dealing with big data sets to explore useful information. Big data can be found in various fields such as: natural disaster, atmospheric science, astronomy, social networking sites, life sciences, medical science, government data, web logs, mobile phones, and sensor networks [29].

TABLE 1.2: List of Related Papers on Data Mining on Big Data

Mining Algorithm	Goal	Data Source	References
Clustering	• Network performance enhancement • Inhabitant action prediction • Provisioning of the needed services • in smart home and smart city • Relationships in a social network	• wireless sensor • X10 lamp and home appliances • Raw location tracking data • Vacuum sensor GPS and sensor for agriculture • RFID, smart phone, PDA, and so on	[31] [32] [33] [34] [34] [35] [36] [37] [38] [39] [40]
Classification	• Device recognition • Traffic event detection • Parking management • Inhabitant action prediction • Physiology signal analysis	• Smart phone, and vehicle sensor • Wireless ECG sensor, video camera, microphone, • Wearable kinematic sensor, etc...	[41] [42] [43] [44] [45] [46] [47] [48] [49] [50]
Frequent Pattern	• RFID tag management • Spatial collocation pattern • Analysis Purchase • Behavior analysis • Inhabitant action prediction	• RFID. • GPS. • sensor.	[51] [52] [53] [54] [55] [56] [57]
Hybrid	• Inhabitant action prediction	• RFID • sensor	[58] [56] [59] [60] [61]

The main objectives of Big Data analytic are: developing efficient techniques that can predict the future trends precisely and to get information about the important relationship among the data features to be used in scientific aims. Big Data has a lot of applications in different fields suchas Technology, Business, Health, IoT, Smart cities etc. These applications will allow people tohave better services, better customer experiences, and also to prevent and detect illness much easier than before [30]. The fast progress in the Internet and mobile technologies plays a greatrole in the data generation, growth and storage. As the amount of generated data is increasing

exponentially, advanced analysis techniques for massive datasets is necessary to discover the valuable information that best matches people interests. A summary of data mining techniques applied to Big Data is given in Table 1.2.

1.7 Big Data Mining Challenging

Data scientist faces three major challenges to mine Big Data which are:

- Design of efficient Mining algorithms.
- Mining platform.
- Privacy.

Basically, the Big Data is stored at different places and also the data volumes may get increasedas the data keeps on increasing continuously. So, to collect all the data stored in different placesis that much expensive. Suppose, if we use these typical data mining methods (those methods which are used for mining the small scale data in our personal computer systems) for mining Big Data, and then it would become an obstacle for it.

Because the typical methods are required data to be loaded into main memory, though we have super large main memory. To maintain the privacy is one of the main aims of data mining algorithms. Presently, to mine information from big data, parallel computing based algorithms such as MapReduce are used. In such algorithms, large datasets are divided into a number of subsets and then, mining algorithms are applied to those subsets. Finally,summation algorithms are applied to the results of mining algorithms, to meet the goal of big data mining. In this whole procedure, Chapter 2 explains different tools and platforms for big analysis.

1.8 Problem Statement

Today, the rapid growth in the size of generated data is so huge and complex that traditional data processing application tools and platforms are inadequate to deal with it. Therefore, the

Big Data require suitable analysis mechanisms for data processing and analysis in an efficient manner. Consequently, developing and designing new scalable data mining techniques is very important and necessary mission for researchers and scientists in the last years. Scaling is the ability of the system to adapt increased demands in terms of data processing. To support Big Data processing, different platforms incorporate scaling in different forms. From a broader scene, the Big Data platforms can be classified into the following two classes of scaling:

- **Horizontal Scaling** includes dividing the workload across a number of servers which maybe even commodity machines. It is also known as scale out, where multiple independent machines are added together in order to enhance the processing capability. Typically, various instances of the operating system are working on separated machines.
- **Vertical Scaling** depends on installing more memory, more processors, and faster hardware, typically, within a single server. Vertical Scaling is also known as scale up and it usually involves a single instance of an operating system. Our research makes use of both horizontal and vertical scaling to develop and implement data mining techniques using new big data platforms such as Apache Hadoop, Apache Spark, and H2O.

Chapter 2

Scalability & Big Data Analytics Tools

2.1 Scalability

At this time, and with astonishingly huge volumes of data, the task of mining big dataset has become one of the most time-consuming tasks that are difficult to implement using conventional data mining tools or even by using a single machine. This led to the urgent need for scaling methods. These scaling methods handling the continuously increasing the volume of workload by expanding the computation devices. The definition of Scalability is therefore defined as the ability of the system or system to cope with increasing workloads, or its ability to expand in order to expand to handle this growing growth. Any system is scalable if it has the capability to increase its total output while increasing the load when adding some capabilities such as hardware or improving it [21]. In general, it is difficult to define the scalability as a featureof any system, but in any particular case, it is necessary to define specific requirements for scalability in these trends which are important. Scalability is one of the most important issues in many things such as routers, databases, networking, and electronics systems. A system that improves its performance by adding some hardware, in proportion to the added capacity, is said to be a scalable device. In order to scale any system, there are two ways: horizontal and vertical scaling [62].

2.1.1 Horizontal Scalability

Horizontal scaling is the addition or deletion of many computers nodes to the system. With the continued decline in the prices of computing devices with increased efficiency in its perfor-

mance, new computer applications in many fields such as biotechnology workloads and seismic analysis have been adopted to use low-cost but high-performance commodity systems at the same time.

The specialists have now relied on assembling hundreds of small computers and integrating them into one system to obtain superior computing power. Such progress necessarily led to the need for software types that allow effective control and good handling of many computers connected together as well as hardware components like shared data memory with superior input/output execution. Size of scalability to any system determined by the highest number of processors that can be involved in this system [63].

2.1.2 Vertical Scalability

Vertical scaling is the addition or deleting many of the resources of a single computer such as memory or CPUs. This type of scaling allows us to use virtualization technology better becauseit provides many resources that can be shared to run more than one application at a time.

In order to build scalable systems for mining large-scale data, new tools were needed to analyze this enormous volume of data. Therefore, we have devoted the remaining part of this chapter to talk about some of the new tools used in the analysis of big data, which allows the constructionof mining systems can be scalable.

2.2 Hadoop

Hadoop is one of the open source frameworks. It had written in Java and it had his debut in October 2003. Hadoop has given a very important advantage in allowing the handling and processing of massive datasets through the use of clusters of computers using simple software models. Hadoop is designed to extend the processing and storage of huge data using thousands of computers rather than using a single node.

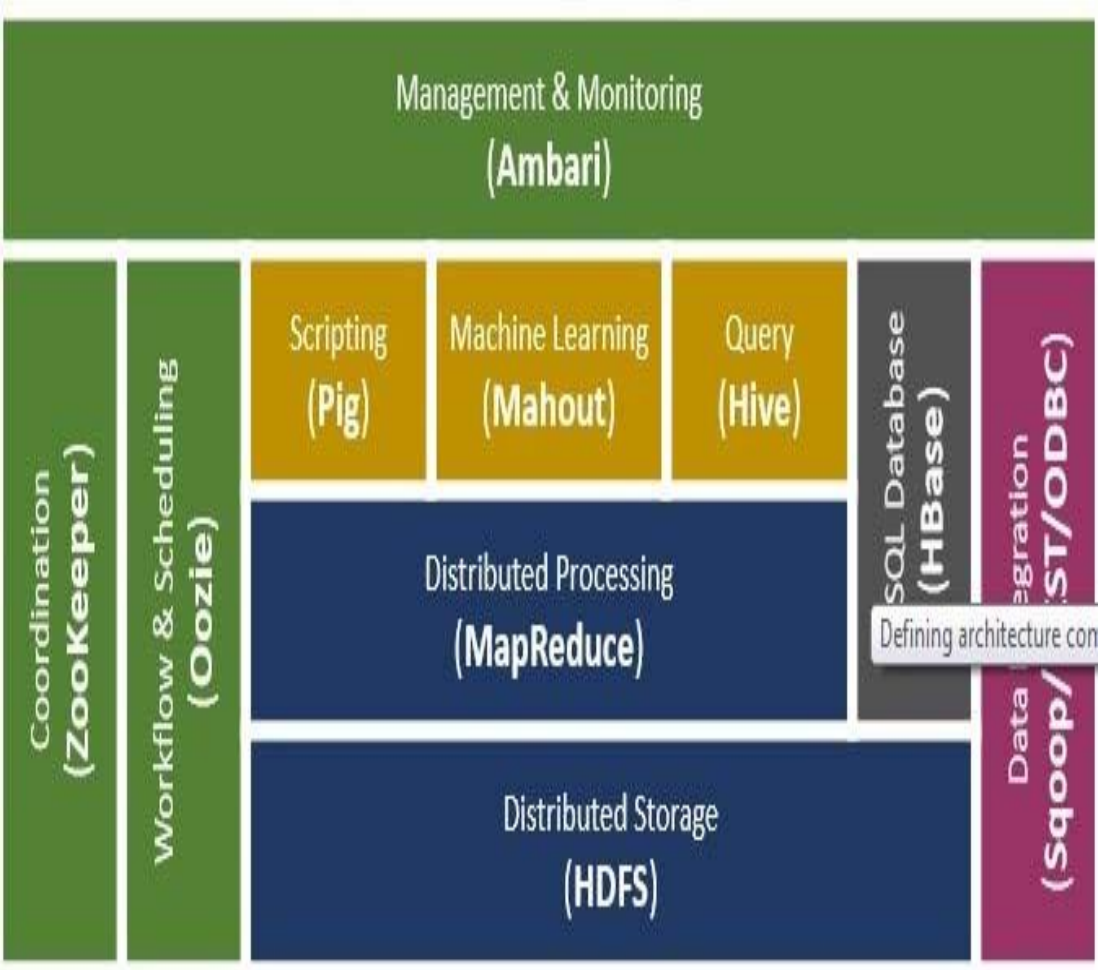
The essence of the Hadoop consists of two parts: the first is the part of the data storage, which is known as Hadoop Distributed File System(HDFS). The second one, which is responsible for the

process of data processing through MapReduce programming paradigm. The nature of the work is based on dividing the large data files into small blocks distributed on the machines in the cluster to be handled in a parallel way by writing programming codes in MapReduce.

2.2.1 Hadoop Architecture

The base of Hadoop paradigm consists of the following four modules which are shown in Figure2.1.

FIGURE 2.1: Architecture of Hadoop



1. **Hadoop Common**: Containing utilities and libraries in Java which are required to start Hadoop as it is necessary for the operation of other units such as Hive, HBase, etc.
2. **Hadoop Distributed File System (HDFS)**: Is considered to be the secret to the success of the Hathoub system, as it represents the class responsible for data storage. It first dividesthe large data files and then distributes them to the different machines within the system,

allowing high speed to access and collect data. It produces many copies of the data files and keeps them on a number of nodes in the Hadoop cluster until the process of processing finished. It contains 3 important components:

- **NameNode**.
- **DataNode**.
- **Secondary NameNode**.

HDFS works using the Master and Slave model, in which NameNode plays the master role, which follows up the process of storing and distributing data, and DataNode plays the role of slave and is based on summarizing and assembling the data from the different nodes within the Hadoop cluster (see Figure 2.2).

3. **Hadoop YARN**: It is a model for managing the resources and job scheduling in Hadoop cluster.

4. **Hadoop MapReduce**: A Java-based programming model has been created by Google to handle the data stored in the HDFS. MapReduce disaggregates the task of processing big data into smaller tasks. MapReduce aggregates data and the results from the vari- ous machines in the Hadoop cluster to end the analysis process in a parallel model. MapReduce model is primarily based on the YARN that supports the parallel processingin the analysis of big data. By using MapReduce, It is easy to write applications that can be implemented using a large number of machines, taking into account failure manage- ment and fault. The basic principle of the work in MapReduce is that Map Job sends a query to process the data files stored in the HDFS in the Hadoop Cluster, but as well as the Reduce job collects all the results in one output and store it in the output file. Thefunction of the Map in the Hadoop system takes the inputs and divide them into indepen-dent parts and the outputs of this stage will be inputs for the next stage. After this, the Hadoop system will collect these results in one output file and all inputs and outputs fromdifferent tasks are stored in the HDFS (Figure 2.3).

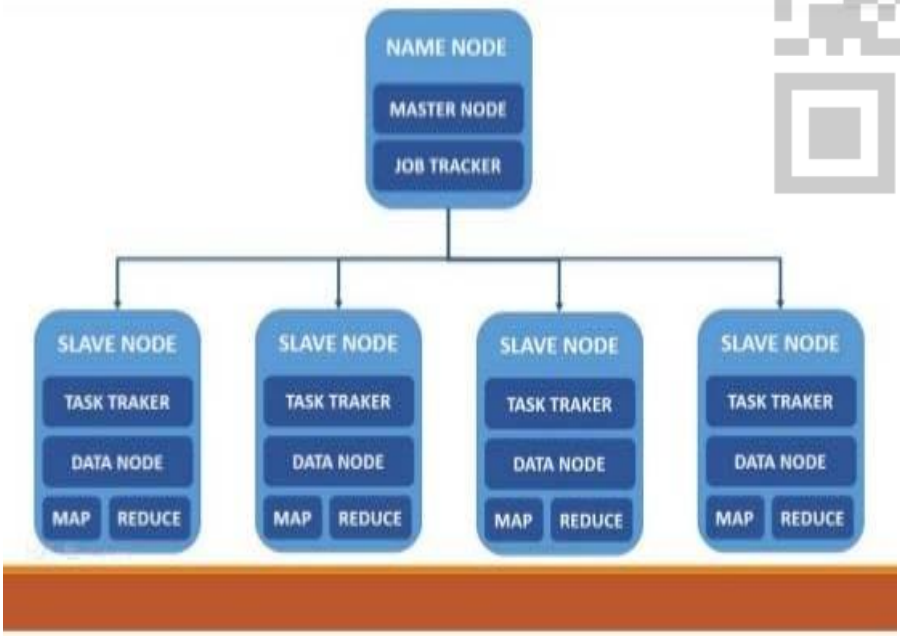


FIGURE 2.2: Master & Slave in Architecture of Hadoop

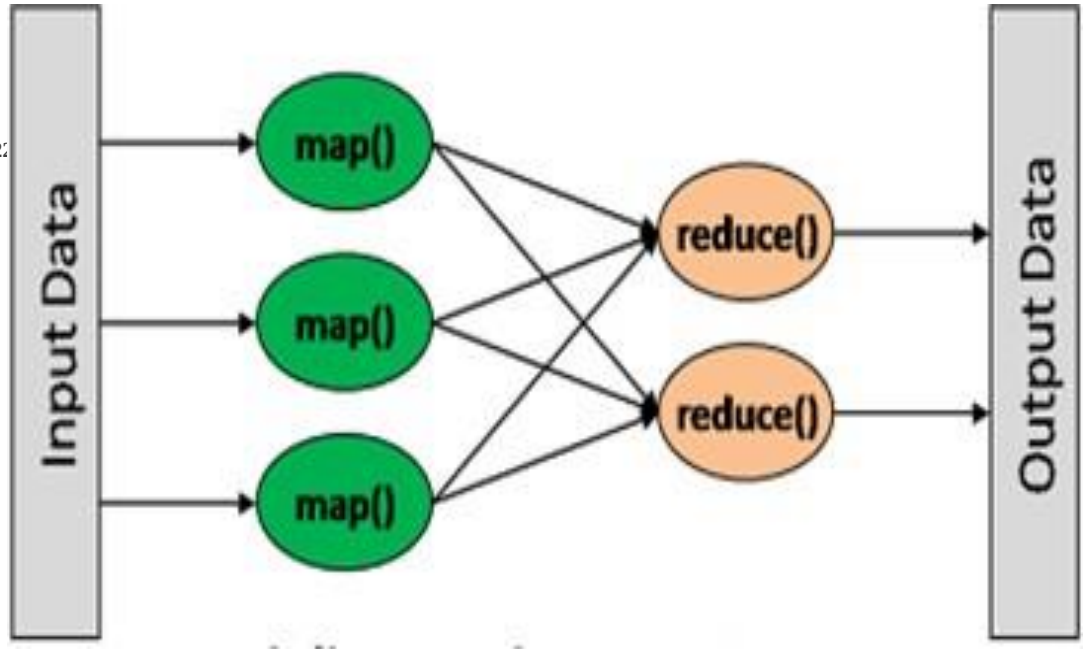


FIGURE 2.3: MapReduce Model

2.2.2 Hadoop Data Access Modules

2.2.2.1 Pig

Pig is designed by Yahoo to make the process of analyzing large data easier and more efficient.Its main mission is to provide a high level of data flow in the system of Hadoop and it is

characterized by ease of use and scalability. The most important feature of Pig is its open structure,making it easy to handle large data in parallel.

2.2.2.2 Hive

Hive was developed by Facebook and is a data warehouse that has been placed on the top of the Hadoop, providing a simple language such as SQL called HiveQL, used to analyze, query and summarize big data. The most important feature is that it makes the process of querying big data faster because of its use of the indexing process.

2.2.3 Hadoop Data Integration Modules

2.2.3.1 Sqoop

The main task of Scoop is to transfer data from its external sources and place it in the designated places in the Hadoop system, such as HDFS also used to transfer data from inside the Hadoop to store in external sources of data. Scoop is a data transfer that works in a parallel way, makingit more effective in data analysis and faster copying of data.

2.2.3.2 Flume

The main use of the Flume is to collect and aggregate huge data from its main sources and send it to any proper layer in the Hadoop system which is HDFS. The task of data flow is accomplished by Flume 3 basic structures, which are channels, sinks, and sources.The process of data flow by Flume is defined as a factor, as are the Bits of the data transmitted by the Flume is known as events.

2.2.4 Hadoop Data Storage Modules

2.2.4.1 HBase

The HBase component is a column-oriented database that stores data in HDFS. HBase Supports random reading and batch computing by using MapReduce. We can make huge tables containing millions of rows and columns and stored in the Hadoop system using HBaseNoSQL. The bestuse is when you need to read or write randomly to access large data [**?**].

2.2.5 Hadoop Monitoring & Management Modules

2.2.5.1 Oozie

The Oozie component is a process scheduler that uses a directed graph to create workflow maps. Oozie stores the current tasks, their cases and their changes with the workflow map to control the implementation of the running tasks in the Hadoop system. Workflow maps extracted by Oozie depend on time and data dependencies.

2.2.5.2 Zookeeper

Zookeeper is the ultimate coordinator and provides fast, simple, reliable and ordered services in the Hadoop system. Its primary task is synchronization service and distributed configuration service to provide a list of distributed system naming.

2.2.6 More Modules of Hadoop Ecosystem

2.2.6.1 MAHOUT

Mahout is one of the most important components of Hadoop where it provides implementation for the majority of machine learning techniques. This component gave the opportunity for a good analysis of large data through data mining methods.

2.2.6.2 Apache Kafka

Apache Kafka is durable, scalable, and fast distributed public-subscribe message designed by LinkedIn.

2.3 Apache Spark

Apache Spark is one of the latest and fastest data analysis tools based on cluster computing technology. It is designed and developed based on Hadoop MapReduce system where the MapReduce framework has been developed to become more effective for use in more types of computations such as processing data streams and interactive queries. The main advantage of Spark is that it uses the method of in-memory computation, which makes it faster in processing the application. Spark has been developed to cover a wide range of workloads in iterative algorithms, interactive queries, and batch computation.Its basic structure was the resilient distributed dataset (RDD) which enabled it to deal with the analysis of data in parallel, in a cluster of a large number of computers and in fault tolerance way.

2.3.1 Apache Spark Evolution

In 2009 at UC Berkeley's AMPLab, Matei Zaharia presented Spark as one of the sub- projects of the Hadoop platform. In 2010 it became open source under the BSD license. In 2013 it was donated to the foundation of Apache software and then in 2014 Spark became one of themost important Apache projects used in big data analytics.

2.3.2 Apache Spark Features

Apache Spark has many important features:

- **Speed**: The Apache Spark is very fast in carrying out tasks in the Hadoop system, whichis 100 times faster when used in-memory and up to 10 times when using disk storage. The speed of Spark is due to the fact that reading and writing data from and to memory has decreased. spark stores current operations in memory.

- **Supports the use of many programming languages**: Spark is characterized by the pres-ence of built-in APIs in many programming languages such as Scala, Java, Python, or R. This has made it easier for users to write their applications in any of the available programming languages. Spark's new releases are supported by more than 80 high-level interactive searches.

- **Advanced Analysis**: Spark supports the use of Streaming data, SQL queries, Graph al-gorithms, and Machine learning (ML), as it also supports the use MapReduce model.

2.3.3 Spark Modules

The Figure2.4 depicts the distinct modules of the Spark which are given as the following.

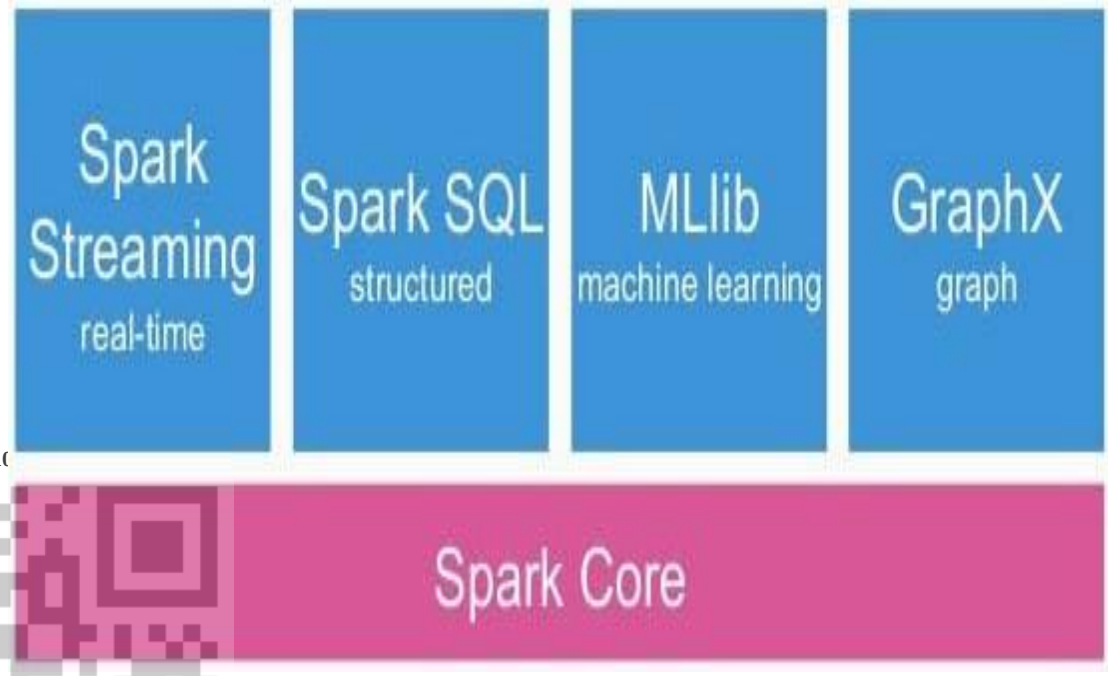


FIGURE 2.4: Spark Modules

- **Spark Core**: Spark Core is the main component of Apache Spark execution engine and all other Spark components have been built on it. It provides the in-memory computation model and it contains the referencing of datasets in the external storage system.
- **Spark SQL**: Spark SQL is an Apache module built on Spark Core. It provides a type ofdata known as SchemaRDD. It also provides support for analyzing structured and semi-structured data through the query process.
- **Spark Streaming**: Spark Streaming is the component that has increased the speed of SparkCor fast scheduling capabilities to enable analysis of data streams. It receives data streams and analyzes them in real time.
- **Machine Learning Library (MLlib)**: MLlib is a framework for distributed machine learning build on Spark Core and contains an implementation for the majority of the ma- chine learning techniques.it takes the place Mahuot in the Hadoop system and is charac-terized by the super-speed as it relies on in-memory computation.

- **GraphX**: GraphX is a paradigm for distributed graph-processing build on Spark Core. It provides an excellent programming interface that allows the use of Pregel abstraction APIto write applications of the of the graph computation executed in an ideal time.

2.4 H2O

H2O is one of the big data analytics frameworks open source and is characterized as distributed, in-memory, scalable, and fast machine learning. H2O allows users to develop predictive and machine learning models on big data. In 2014 that the mathematical core of H2O was built and developed by his Amo Candel as apart of Fortune's 2014 "Big Data All Stars".H2o can be run from Scala, Python, R, and Java. In addition, users can make use H2O to be integrated with Spark streaming and Spark paradigm through the Sparkling Water project. This method is based on breaking the main job into many small tasks that are executed in parallel. The use of this method has resulted in a significant improvement in the distribution of increasing workloads in

big data analysis tasks.The next diagram shows the basic components of H2O. Asshown in Figure 2.5 it is divided into two bottom and top parts separated by the part of the cloud network.The bottom part contains the parts that work with the core to perform different tasks. The top part provides a lot of APIs for users to write their applications easily.

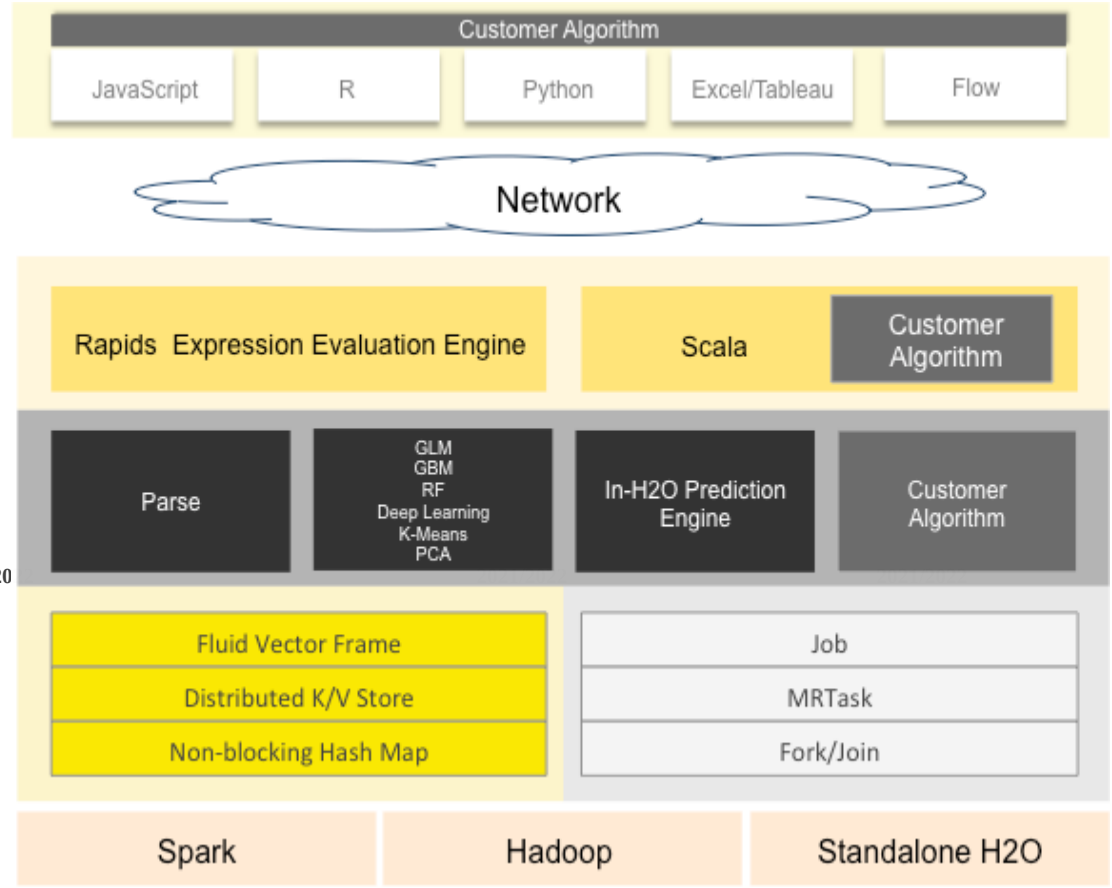


FIGURE 2.5: H2O Components

2.5 Flink

Initially, Flink was developed under the name Stratosphere at the Technical University of Berlin. Flink is now one of the strongest, scalable, in-memory top-level projects after passing thestage of the Apache in January 2015. As in Spark, Flink possesses the ability to stream and batch processing by allowing the implementation of Lamda architectural applications. Users can run Flank completely independently of the Hadoop system or build it on top of MapReduce by

integrating it with YARN or HDFS. Flink's processing system performs both of transformations(like map and reduce) and functions (like iterate, group, and join) on parallel datasets. Flink's team developed its own machine learning library Flink-ML in April 2015.

2.6 Clustering

Most of the distributed paradigms, like Spark and Hadoop, make use of the concept of clusters. Where a cluster is defined as a collection of connected nodes which execute a task together. In the case of Spark and Hadoop, the cluster is a group of computing devices (computers) which distributes the workload. These modern frameworks offer an easy way to create clusters and then execute tasks on these clusters. Often, It is not convenient to keep physical nodes.Third- party providers have developed the need for easy contact with nodes. One of the most famous and widely used such association is Amazon and its service Amazon Elastic Compute Cloud (EC2). EC2 lets its users rent computers by the hour and multiple computers may be spawned and removed trivially.

2.7 Fault Tolerance

The possibility of fall down increases when the running time of a problem grows. When dealing with a huge cluster, things are limited to break eventually. When things break, it leads to errors, or worse, faulty results. It is vital then to select a system which has a suspiciously considered fault policy. Both Spark and MapReduce provide fault tolerance on multiple levels with precautions such as restarting of nodes and exiting the whole system. Exiting the whole system might sound undesirable but is actually more helpful then silent errors which in the worst case corrupt the result.

Chapter 3

Introduction to Hadoop

3.1 Hadoop

1. Hadoop is an open source framework that supports the processing of large data sets in a distributed computing environment.
2. Hadoop consists of MapReduce, the Hadoop distributed file system (HDFS) and a number of related projects such as Apache Hive, HBase and Zookeeper. MapReduce and Hadoop distributed file system (HDFS) are the main component of Hadoop.
3. Apache Hadoop is an open-source, free and Java based software framework offers a powerfuldistributed platform to store and manage Big Data.
4. It is licensed under an Apache V2 license.
5. It runs applications on large clusters of commodity hardware and it processes thousands of terabytes of data on thousands of the nodes. Hadoop is inspired from Google's MapReduce and Google File System (GFS) papers.
6. The major advantage of Hadoop framework is that it provides reliability and high availability.

3.2 Use of Hadoop

There are many advantages of using Hadoop:

1. Robust and Scalable – We can add new nodes as needed as well modify them.
2. Affordable and Cost Effective – We do not need any special hardware for running Hadoop. We can just use commodity server.
3. Adaptive and Flexible – Hadoop is built keeping in mind that it will handle structured and unstructured data.
4. Highly Available and Fault Tolerant – When a node fails, the Hadoop framework automatically fails over to another node.

3.3 Core Hadoop Components

There are two major components of the Hadoop framework and both of them does two of the importanttask for it.

1. Hadoop MapReduce is the method to split a larger data problem into smaller chunk and distribute it to many different commodity servers. Each server have their own set of resourcesand they have processed them locally. Once the commodity server has processed the data theysend it back collectively to main server. This is effectively a process where we process large data effectively and efficiently

2. Hadoop Distributed File System (HDFS) is a virtual file system. There is a big difference between any other file system and Hadoop. When we move a file on HDFS, it is automatically split into many small pieces. These small chunks of the file are replicated and stored on otherservers (usually 3) for the fault tolerance or high availability.
3. Namenode: Namenode is the heart of the Hadoop system. The NameNode manages the file system namespace. It stores the metadata information of the data blocks. This metadata is stored permanently on to local disk in the form of namespace image and edit log file. The NameNode also knows the location of the data blocks on the data node. However the NameNode does not store this information persistently. The NameNode creates the block to DataNode mapping when it is restarted. If the NameNode crashes, then the entire Hadoop system goes down. Read more about Namenode
4. Secondary Namenode: The responsibility of secondary name node is to periodically copy and merge the namespace image and edit log. In case if the name node crashes, then the namespaceimage stored in secondary NameNode can be used to restart the NameNode.
5. DataNode: It stores the blocks of data and retrieves them. The DataNodes also reports the blocks information to the NameNode periodically.
6. Job Tracker: Job Tracker responsibility is to schedule the client's jobs. Job tracker creates map and reduce tasks and schedules them to run on the DataNodes (task trackers). Job Tracker alsochecks for any failed tasks and reschedules the failed tasks on another DataNode. Job trackercan be run on the NameNode or a separate node.
7. Task Tracker: Task tracker runs on the DataNodes. Task trackers responsibility is to run the map or reduce tasks assigned by the NameNode and to report the status of the tasks to the NameNode.

Besides above two core components Hadoop project also contains following modules as well.

1. Hadoop Common: Common utilities for the other Hadoop modules
2. Hadoop Yarn: A framework for job scheduling and cluster resource management

3.4 RDBMS

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

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

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

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

Another difference between MapReduce and an RDBMS is the amount of structure in the datasets that they operate on. Structured data is data that is organized into entities that have a defined format, such as XML documents or database tables that conform to a particular predefined schema. This is the realm of the RDBMS. Semi-structured data, on the other hand, is looser, and though there may be a schema, it is often ignored, so it may be used only as a guide to the structure of the data: for example, a spreadsheet, in which the structure is the grid of cells, although the cells themselves may hold any form of data. Unstructured data does not have any particular internal structure: for example, plain text or image data. MapReduce works well on unstructured or semi- structured data, since it is designed to interpret the data at processing time. In other words, the input keys and values for MapReduce are not an intrinsic property of the data, but they are chosen by the person analyzing the data.

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

A web server log is a good example of a set of records that is not normalized (for ex- ample, the client hostnames are specified in full each time, even though the same client may appear many times), and this is one reason that logfiles of all kinds are particularly well-suited to analysis with MapReduce.

MapReduce is a linearly scalable programming model. The programmer writes two functions—a map function and a reduce function—each of which defines a mapping from one set of key-value pairs to another. These functions are oblivious to the size of the data or the cluster that they are operating on, so they can be used unchanged for a small dataset and for a massive one. More important, if you double the size of the input data, a job will run twice as slow. But if you also double the size of the cluster, a job will run as fast as the original one. This is not generally true of SQL queries.

Over time, however, the differences between relational databases and MapReduce sys- tems are likely to blur—both as relational databases start incorporating some of the ideas from MapReduce (such as Aster Data's and Greenplum's databases) and, from the other direction, as higher-level query languages built on MapReduce (such as Pig and Hive) make MapReduce systems more approachable to traditional database programmers.

3.5 A BRIEF HISTORY OF HADOOP

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

Building a web search engine from scratch was an ambitious goal, for not only is the software required to crawl and index websites complex to write, but it is also a challenge to run without a dedicated operations team, since there are so many moving parts. It's expensive, too: Mike Cafarella and Doug Cutting estimated a system supporting a 1-billion-page index would cost around half a million dollars in hardware, with a monthly running cost of $30,000.Nevertheless, they believed it was a worthy goal, as it would open up and ultimately democratize search engine algorithms.

Nutch was started in 2002, and a working crawler and search system quickly emerged. However, they realized that their architecture wouldn't scale to the billions of pages on the Web. Help was at hand with the publication of a paper in 2003 that described the architecture of Google's distributed filesystem, called GFS, which was being used in production at Google.[11] GFS, or something like it, would solve their storage needs for the very large files generated as a part of the web crawl and indexing process. In par- ticular, GFS would free up time being spent on administrative tasks such as managing storage nodes. In 2004, they set about writing an open source implementation, the Nutch Distributed Filesystem (NDFS).

In 2004, Google published the paper that introduced MapReduce to the world.[12] Early in 2005, the Nutch developers had a working MapReduce implementation in Nutch, and by the middle of that year all the major Nutch algorithms had been ported to run using MapReduce and NDFS.

NDFS and the MapReduce implementation in Nutch were applicable beyond the realm of search, and in February 2006 they moved out of Nutch to form an independent subproject of Lucene called Hadoop. At around the same time, Doug Cutting joined Yahoo!, which provided a dedicated team and the resources to turn Hadoop into a system that ran at web scale (see sidebar). This was demonstrated in February 2008 when Yahoo! announced that its production search index was being generated by a 10,000-core Hadoop cluster.[13]

In January 2008, Hadoop was made its own top-level project at Apache, confirming its success and its diverse, active community. By this time, Hadoop was being used by many other companies besides Yahoo!, such as Last.fm, Facebook, and the New York Times.

In one well-publicized feat, the New York Times used Amazon's EC2 compute cloud to crunch through four terabytes of scanned archives from the paper converting them to PDFs for the Web.[14] The processing took less than 24 hours to run using 100 ma- chines, and the project probably wouldn't have been embarked on without the com- bination of Amazon's pay-by-the-hour model (which allowed the NYT to access a large number of machines for a short period) and

Hadoop's easy-to-use parallel programming model.

In April 2008, Hadoop broke a world record to become the fastest system to sort a terabyte of data. Running on a 910-node cluster, Hadoop sorted one terabyte in 209 seconds (just under 3½ minutes), beating the previous year's winner of 297 seconds. In November of the same year, Google reported that its MapReduce implementation sorted one ter- abyte in 68 seconds.[15] As the first edition of this book was going to press (May 2009), it was announced that a team at Yahoo! used Hadoop to sort one terabyte in 62 seconds.

2.6 ANALYZING THE DATA WITH HADOOP

To take advantage of the parallel processing that Hadoop provides, we need to express our queryas a MapReduce job. After some local, small-scale testing, we will be able to run it on a cluster of machines.

MAP AND REDUCE

MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the map function and the reduce function.

2021/2022 2021/2022 2021/2022

The input to our map phase is the raw NCDC data. We choose a text input format that gives us each line in the dataset as a text value. The key is the offset of the beginning of the line from the beginning of the file, but as we have no need for this, we ignore it.

Our map function is simple. We pull out the year and the air temperature, since these are the only fields we are interested in. In this case, the map function is just a data preparationphase, setting up the data in such a way that the reducer function can do its work on it: findingthe maximum temperature for each year. The map function is also a good place to drop bad records: here we filter out temperatures that are missing, suspect, or erroneous.

To visualize the way the map works, consider the following sample lines of input data (some unused columns have been dropped to fit the page, indicated by ellipses):

30103162101496 30103162101496 30103162101496

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

These lines are presented to the map function as the key-value pairs:

(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)

(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)

(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)

(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)

(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)

The keys are the line offsets within the file, which we ignore in our map function. The map function merely extracts the year and the air temperature (indicated in bold text), and emits them as its output (the temperature values have been interpreted as integers):

(1950, 0)

(1950, 22)

(1950, −11)

(1949, 111)

(1949, 78)

The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing sorts and groups the key-value pairs by key. So, continuing the example, our reduce function sees the following input:

(1949, [111, 78])

(1950, [0, 22, −11])

Each year appears with a list of all itsair temperature readings. All the reduce function has to do now is iterate through the list and pick up the maximum reading:

(1949, 111)

(1950, 22)

This is the final output: the maximum global temperature recorded in each year.

The whole data flow is illustrated in 2.2. At the bottom of the diagram is a Unix pipeline, which mimics the whole MapReduce flow, and which we will see again later in the chapter when we look at Hadoop Streaming.

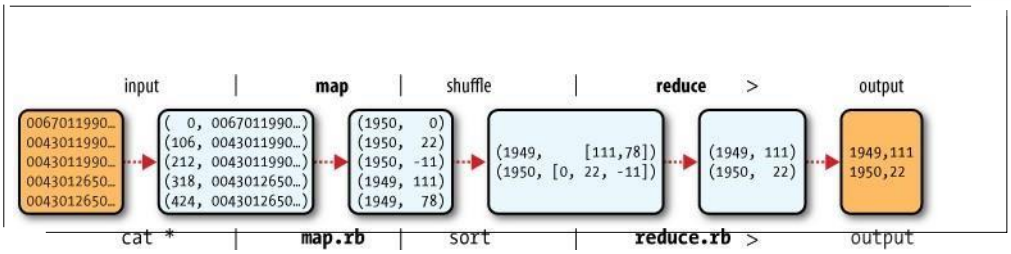


Figure 2-1. MapReduce logical data flow

JAVA MAPREDUCE

Having run through how the MapReduce program works, the next step is to express it in code. We need three things: a map function, a reduce function, and some code to run the job. The map function is represented by the Mapper class, which declares an abstract map() method.

The Mapper class is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function. For the present example, the input key is a long integer offset, the input value is a line of text, the output key is a year,and the output value is an air temperature (an integer). Rather than use built-in Java types, Hadoop provides its own set of basic types that are opti- mized for network serialization. These are found in the org.apache.hadoop.io package. Here we use LongWritable, which corresponds to a Java Long, Text (like Java String), and IntWritable (like Java Integer).

The map() method is passed a key and a value. We convert the Text value containing the line of input into a Java String, then use its substring() method to extract the columns we are interested in.

The map() method also provides an instance of Context to write the output to. In this case, we write the year as a Text object (since we are just using it as a key), and the temperature is wrapped in an IntWritable. We write an output record only if the tem- perature is present and the quality code indicates the temperature reading is OK.

Again, four formal type parameters are used to specify the input and output types, this time forthe reduce function. The input types of the reduce function must match the output types of the map function: Text and IntWritable. And in this case, the output types of the reduce function are Text and IntWritable, for a year and its maximum temperature, which we find by iterating through the temperatures and comparing each with a record of the highest found so far.

A Job object forms the specification of the job. It gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster). Rather than explicitly specify the name of the JAR file, we can pass a class in the Job's setJarByClass() method, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.

Having constructed a Job object, we specify the input and output paths. An input path is specified by calling the static addInputPath() method on FileInputFormat, and it can be a single file, a directory (in which case, the input forms all the files in that directory), or a file pattern. As the name suggests, addInputPath() can be called more than once to use input from multiple paths.

The output path (of which there is only one) is specified by the static setOutput Path() method on FileOutputFormat. It specifies a directory where the output files from the reducer functions are written. The directory shouldn't exist before running the job, as Hadoop will complain and not run the job. This precaution is to prevent data loss(it can be very annoyingto accidentally overwrite the output of a long job with another).

Next, we specify the map and reduce types to use via the setMapperClass() and setReducerClass() methods.

The setOutputKeyClass() and setOutputValueClass() methods control the output types for the map and the reduce functions, which are often the same, as they are in our case. If they are different, then the map output types can be set using the methods setMapOutputKeyClass() and setMapOutputValueClass().

The input types are controlled via the input format, which we have not explicitly set since we are using the default TextInputFormat.

After setting the classes that define the map and reduce functions, we are ready to run the job. The waitForCompletion() method on Job submits the job and waits for it to finish. The method's boolean argument is a verbose flag, so in this case the job writes information about its progress to the console.

The return value of the waitForCompletion() method is a boolean indicating success (true)or failure (false), which we translate into the program's exit code of 0 or 1.

A TEST RUN

After writing a MapReduce job, it's normal to try it out on a small dataset to flush out any immediate problems with the code.

First install Hadoop in standalone mode— there are instructions for how to do this in Appendix A. This is the mode in which Hadoop runs using the local filesystem with a local job runner. Then install and compile the examples using the instructions on the book's website.

When the hadoop command is invoked with a classname as the first argument, it launches a JVM to run the class. It is more convenient to use hadoop than straight java since the former adds the Hadoop libraries (and their dependencies) to the class- path and picks up the Hadoop configuration, too. To add the application classes to the classpath, we've defined an environment variable called HADOOP_CLASSPATH, which the hadoop script picks up.

The last section of the output, titled "Counters," shows the statistics that Hadoop generates for each job it runs. These are very useful for checking whether the amount of data processed is what you expected. For example, we can follow the number of records that went through the system: five map inputs produced five map outputs, then five reduce inputs in two groups produced two reduce outputs.

The output was written to the output directory, which contains one output file per reducer. The job had a single reducer, so we find a single file, named part-r-00000:

```
% cat output/part-r00000
```



```
1949 111

1950 22
```

This result is the same as when we went through it by hand earlier. We interpret this as saying that the maximum temperature recorded in 1949 was 11.1°C, and in 1950 it was 2.2°C.

THE OLD AND THE NEW JAVA MAPREDUCE APIS

The Java MapReduce API used in the previous section was first released in Hadoop

0.20.0. This new API, sometimes referred to as "Context Objects," was designed to make the API easier to evolve in the future. It is type-incompatible with the old, how- ever, so applications need to be rewritten to take advantage of it.

The new API is not complete in the 1.x (formerly 0.20) release series, so the old API is recommended for these releases, despite having been marked as deprecated in the early 0.20 releases. (Understandably, this recommendation caused a lot of confusion so the deprecation warning was removed from later releases in that series.)

Previous editions of this book were based on 0.20 releases, and used the old API throughout (although the new API was covered, the code invariably used the old API). In this edition the new API is used as the primary API, except where mentioned. How- ever, should you wish to use the old API, you can, since the code for all the examples in this book is available for the old API on the book's website.[1]

There are several notable differences between the two APIs:

- The new API favors abstract classes over interfaces, since these are easier to evolve. For example, you can add a method (with a default implementation) to an abstract class without breaking old implementations of the class[2]. For example, the Mapper and Reducer interfaces in the old API are abstract classes in the new API.
- The new API is in the org.apache.hadoop.mapreduce package (and subpackages). The old API can still be found in org.apache.hadoop.mapred.
- The new API makes extensive use of context objects that allow the user code to communicate with the MapReduce system. The new Context, for example, essen- tially unifies the role of the JobConf, the OutputCollector, and the Reporter from the old API.
- In both APIs, key-value record pairs are pushed to the mapper and reducer, but in addition, the new API allows both mappers and reducers to control the execution flow by overriding the run() method. For example, records can be processed in batches, or the execution can be terminated before all the records have been pro- cessed. In the old API this is possiblefor mappers by writing a MapRunnable, but no equivalent exists for reducers.
- Configuration has been unified. The old API has a special JobConf object for job configuration, which is an extension of Hadoop's vanilla Configuration object (used for configuring daemons. In the new API, this distinction is dropped, so job configuration is done through a Configuration.
- Job control is performed through the Job class in the new API, rather than the old JobClient, which no longer exists in the new API.
- Output files are named slightly differently: in the old API both map and reduce outputs are named part-nnnnn, while in the new API map outputs are named part- m-nnnnn, and reduce outputs are named part-r-nnnnn (where nnnnn is an integer designating the part number, starting from zero).
- User-overridable methods in the new API are declared to throw java.lang.Inter ruptedException. What this means is that you can write your code to be reponsive to interupts so that the framework can gracefully cancel long-running operations if it needs to[3].
- In the new API the reduce() method passes values as a java.lang.Iterable, rather than a java.lang.Iterator (as the old API does). This change makes it easier to iterate over the values using Java's for-each loop construct: for (VALUEIN value : values) { ... }

3.7 Hadoop Ecosystem

Although Hadoop is best known for MapReduce and its distributed filesystem (HDFS, renamed from NDFS), the term is also used for a family of related projects that fall under the umbrella of infrastructure for distributed computing and large-scale data processing.

All of the core projects covered in this book are hosted by the Apache Software Foundation, which provides support for a community of open source software projects, including the original HTTP Server from which it gets its name. As the Hadoop eco- system grows, more projects are appearing, not necessarily hosted at Apache, which provide complementary services to Hadoop, or build on the core to add higher-level abstractions.

The Hadoop projects that are covered in this book are described briefly here:

Common

A set of components and interfaces for distributed filesystems and general I/O (serialization,Java RPC, persistent data structures).

Avro

A serialization system for efficient, cross-language RPC, and persistent data storage.

MapReduce

A distributed data processing model and execution environment that runs on large clusters of commodity machines.

HDFS

A distributed filesystem that runs on large clusters of commodity machines.

Pig

A data flow language and execution environment for exploring very large datasets. Pig runs on HDFS and MapReduce clusters.

Hive

A distributed data warehouse. Hive manages data stored in HDFS and provides a query language based on SQL (and which is translated by the runtime engine to MapReduce jobs) for querying the data.

HBase

A distributed, column-oriented database. HBase uses HDFS for its underlying storage, and supports both batch-style computations using MapReduce and point queries (random reads).

ZooKeeper

A distributed, highly available coordination service. ZooKeeper provides primitives such as distributed locks that can be used for building distributed applications.

Sqoop

A tool for efficiently moving data between relational databases and HDFS.

3.8 PHYSICAL ARCHITECTURE

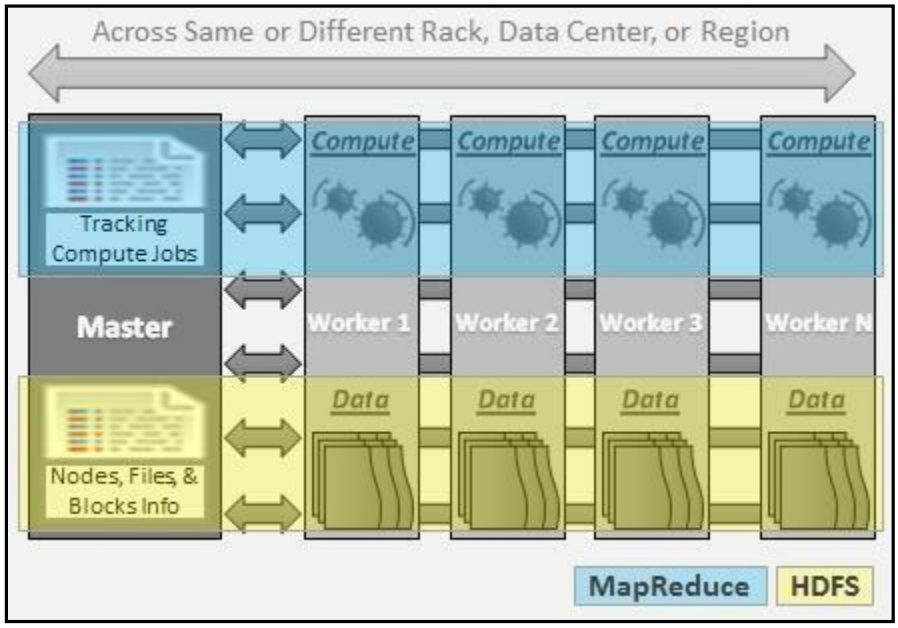


Figure 2.2: Physical Architecture

Hadoop Cluster - Architecture, Core Components and Work-flow

1. The architecture of Hadoop Cluster
2. Core Components of Hadoop Cluster
3. Work-flow of How File is Stored in Hadoop

A. Hadoop Cluster

i. Hadoop cluster is a special type of computational cluster designed for storing and analyzing vast amount of unstructured data in a distributed computing environment

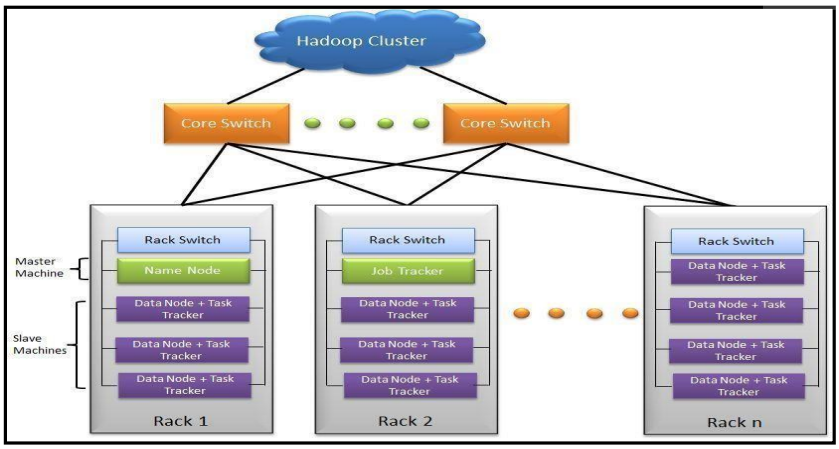


Figure 2.3: Hadoop Cluster

ii. These clusters run on low cost commodity computers.

iii. Hadoop clusters are often referred to as "shared nothing" systems because the only thing that is shared between nodes is the network that connects them.

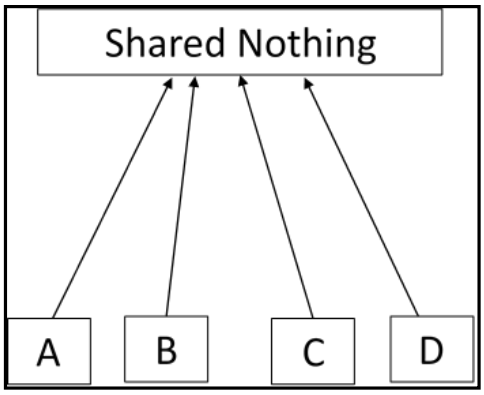


Figure 2.4: Shared Nothing

iv. Large Hadoop Clusters are arranged in several racks. Network traffic between different nodes in the same rack is much more desirable than network traffic acrossthe racks.

A Real Time Example: Yahoo's Hadoop cluster. They have more than 10,000 machines running Hadoop and nearly 1 petabyte of user data.

Figure 2.4: Yahoo Hadoop Cluster

v. A small Hadoop cluster includes a single master node and multiple worker or slavenode. As discussed earlier, the entire cluster contains two layers.

vi. One of the layer of MapReduce Layer and another is of HDFS Layer.

vii. Each of these layer have its own relevant component.

viii. The master node consists of a JobTracker, TaskTracker, NameNode and DataNode.

ix. A slave or worker node consists of a DataNode and TaskTracker.

It is also possible that slave node or worker node is only data or compute node. Thematter of the fact that is the key feature of the Hadoop.

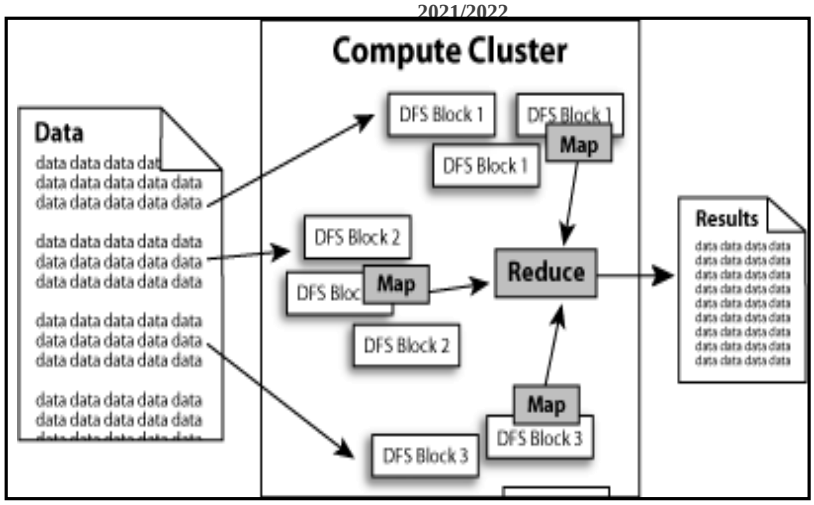


Figure 2.4: NameNode Cluster

B. HADOOP CLUSTER ARCHITECTURE:

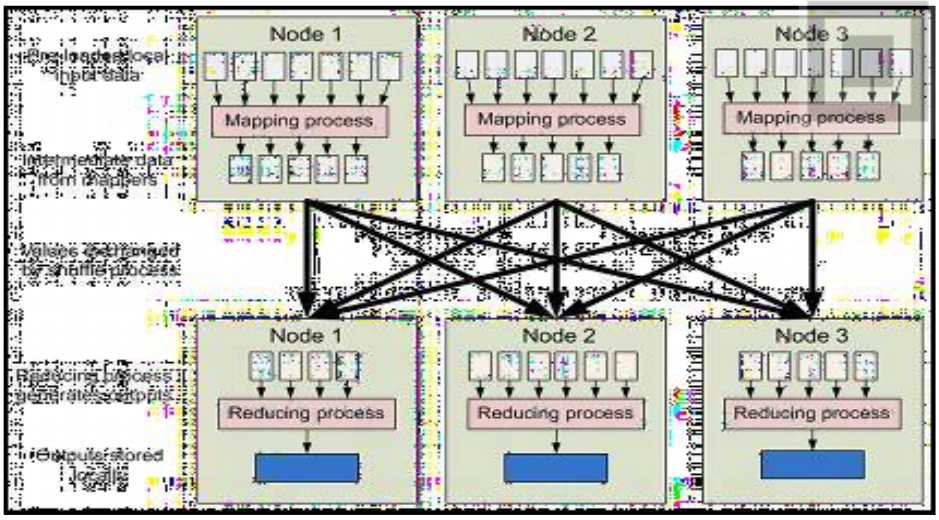


Figure 2.5: Hadoop Cluster Architecture Hadoop Cluster would consists of

- 110 different racks
- Each rack would have around 40 slave machine
- At the top of each rack there is a rack switch

- Each slave machine(rack server in a rack) has cables coming out it from both the ends
- Cables are connected to rack switch at the top which means that top rack switchwill have around 80 ports
- There are global 8 core switches
- The rack switch has uplinks connected to core switches and henceconnecting allother racks with uniform bandwidth, forming the Cluster
- In the cluster, you have few machines to act as Name node and as JobTracker. They are referred as Masters. These masters have different configuration favoring more DRAM and CPU and less local storage.

Hadoop cluster has 3 components:

1. Client
2. Master
3. Slave

The role of each components are shown in the below image.

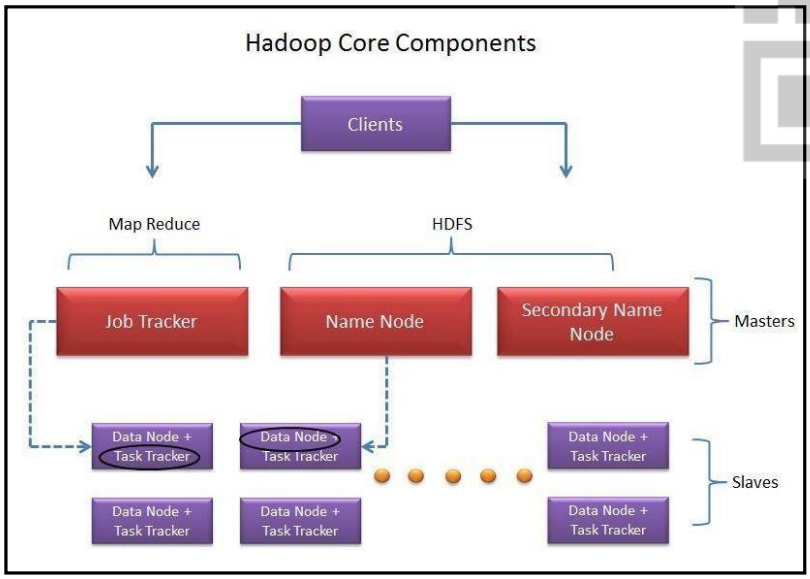


Figure 2.6: Hadoop Core Component

1. Client:

i. It is neither master nor slave, rather play a role of loading the data into cluster, submit MapReduce jobs describing how the data should be processed and then retrieve the data to seethe response after job completion.

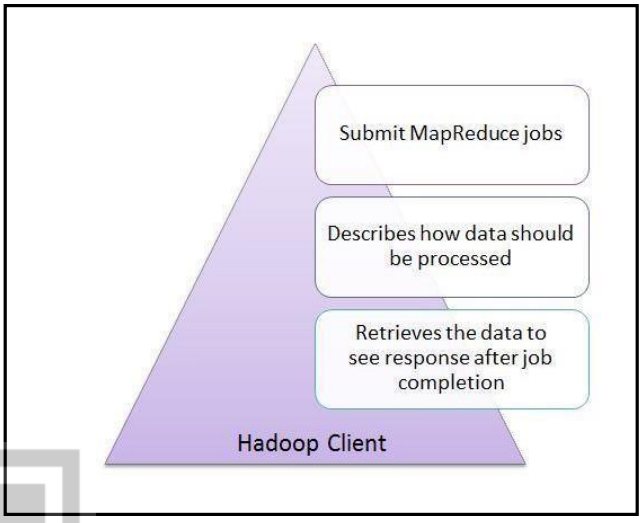


Figure 2.6: Hadoop Client

2. Masters:

The Masters consists of 3 components NameNode, Secondary Node name and JobTracker.

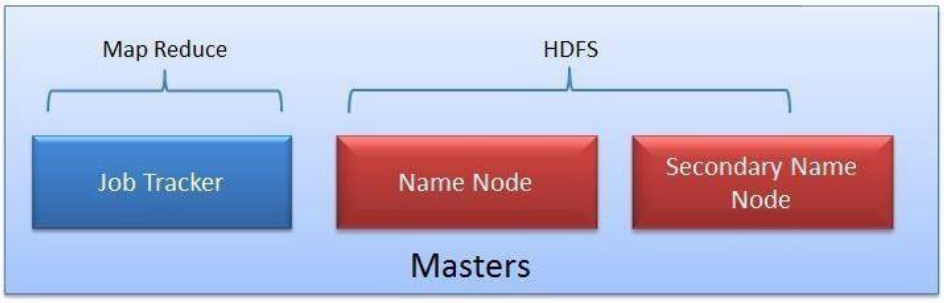


Figure 2.7: MapReduce - HDFS

i. NameNode:

- NameNode does NOT store the files but only the file's metadata. In later section we will see itis actually the DataNode which stores the files.

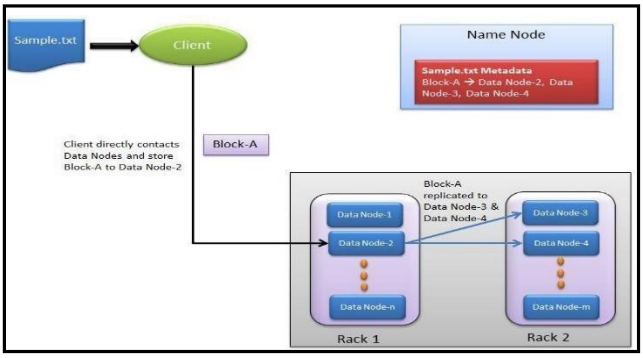


Figure 2.8: NameNode

- NameNode oversees the health of DataNode and coordinates access to the data stored inDataNode.
- Name node keeps track of all the file system related information such as to
  - ✓ Which section of file is saved in which part of the cluster
  - ✓ Last access time for the files
  - ✓ User permissions like which user have access to the file

ii. JobTracker:

JobTracker coordinates the parallel processing of data using MapReduce.

To know more about JobTracker, please read the article All You Want to Know about MapReduce (The Heart of Hadoop)

iii. Secondary Name Node:

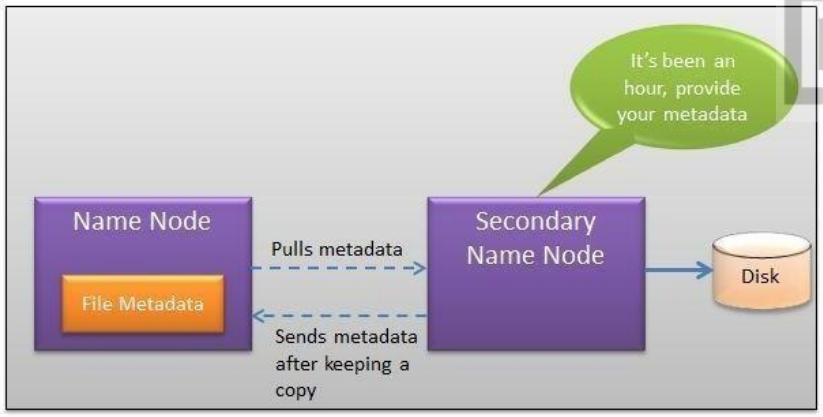


Figure 2.9: Secondary NameNode

- The job of Secondary Node is to contact NameNode in a periodic manner after certain timeinterval (by default 1 hour).
- NameNode which keeps all filesystem metadata in RAM has no capability to process thatmetadata on to disk.
- If NameNode crashes, you lose everything in RAM itself and you don't have any backup of filesystem.

- What secondary node does is it contacts NameNode in an hour and pulls copy of metadatainformation out of NameNode.
- It shuffle and merge this information into clean file folder and sent to back again to NameNode,while keeping a copy for itself.
- Hence Secondary Node is not the backup rather it does job of housekeeping.
- In case of NameNode failure, saved metadata can rebuild it easily.

3. Slaves:

i. Slave nodes are the majority of machines in Hadoop Cluster and are responsible to

- Store the data
- Process the computation

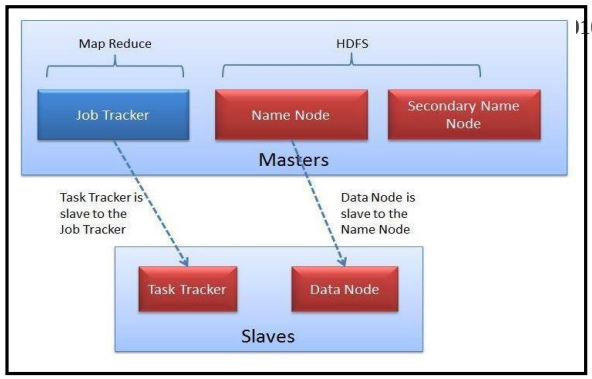


Figure 2.10: Slaves

ii. Each slave runs both a DataNode and Task Tracker daemon which communicates to theirmasters.

iii. The Task Tracker daemon is a slave to the Job Tracker and the DataNode daemon a slave tothe NameNode

II. Hadoop- Typical Workflow in HDFS:

Take the example of input file as Sample.txt.

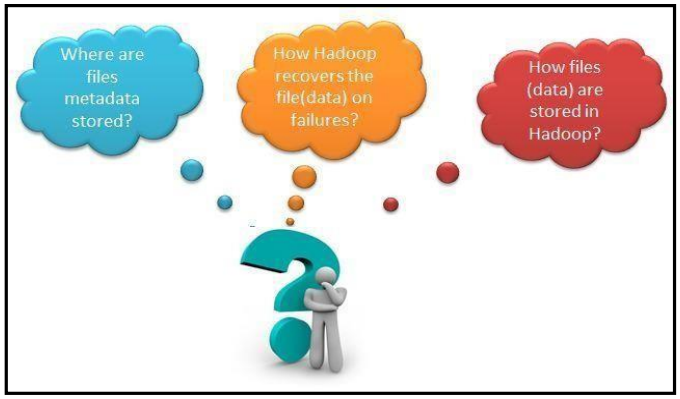


Figure 2.11: HDFS Workflow

1. How TestingHadoop.txt gets loaded into the Hadoop Cluster?

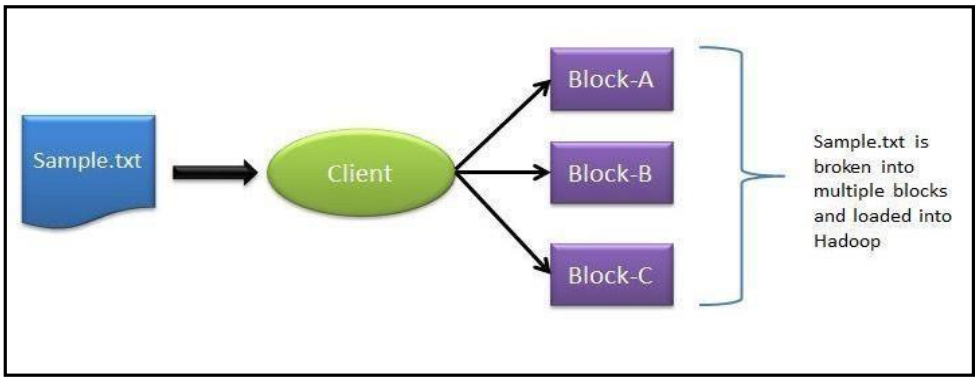


 Figure 2.12: Loading file in Hadoop Cluster

- Client machine does this step and loads the Sample.txt into cluster.
- It breaks the sample.txt into smaller chunks which are known as "Blocks" in Hadoop context.
- Client put these blocks on different machines (data nodes) throughout the cluster.

2. Next, how does the Client knows that to which data nodes load the blocks?

- Now NameNode comes into picture.
- The NameNode used its Rack Awareness intelligence to decide on which DataNode to provide.

- For each of the data block (in this case Block-A, Block-B and Block-C), Client contacts NameNode and in response NameNode sends an ordered list of 3 DataNodes.

3. How does the Client knows that to which data nodes load the blocks?

- For example in response to Block-A request, Node Name may send DataNode-2, DataNode-3 and DataNode-4.
- ✓ Block-B DataNodes list DataNode-1, DataNode-3, DataNode-4 and for Block C data node list DataNode-1, DataNode-2, DataNode-3. Hence
- ❖ Block A gets stored in DataNode-2, DataNode-3, DataNode-4
- ❖ Block B gets stored in DataNode-1, DataNode-3, DataNode-4
- ❖ Block C gets stored in DataNode-1, DataNode-2, DataNode-3
- ✓ Every block is replicated to more than 1 data nodes to ensure the data recovery on the time ofmachine failures. That's why NameNode send 3 DataNodes list for each individual block

4. Who does the block replication?

- Client write the data block directly to one DataNode. DataNodes then replicate the block toother Data nodes.
- When one block gets written in all 3 DataNode then only cycle repeats for next block.

5. Who does the block replication?

- In Hadoop Gen 1 there is only one NameNode wherein Gen2 there is active passive model inNameNode where one more node "Passive Node" comes in picture.
- The default setting for Hadoop is to have 3 copies of each block in the cluster. This setting canbe configured with "dfs.replication" parameter of hdfs-site.xml file.
- Keep note that Client directly writes the block to the DataNode without any intervention ofNameNode in this process.

3.9 Hadoop limitations

i. Network File system is the oldest and the most commonly used distributed file system and wasdesigned for the general class of applications, Hadoop only specific kind of applications can make use of it.

ii. It is known that Hadoop has been created to address the limitations of the distributed file system, where it can store the large amount of data, offers failure protection and provides fastaccess, but it should be known that the benefits that come with Hadoop come at some cost.

iii. Hadoop is designed for applications that require random reads; so if a file has four parts the filewould like to read all the parts one-by-one going from 1 to 4 till the end. Random seek is whereyou want to go to a specific location in the file; this issomething that isn't possible with Hadoop. Hence, Hadoop is designed for non- real-time batch processing of data.

iv. Hadoop is designed for streaming reads caching of data isn't provided. Caching of data is provided which means that when you want to read data another time, it can be read very fast from the cache. This caching isn't possible because you get faster access to the data directly by doing the sequential read; hence caching isn't available through Hadoop.

v. It will write the data and then it will read the data several times. It will not be updating the data that it has written; hence updating data written to closed files is not available. However, you have to know that in update 0.19 appending will be supported for those files that aren't closed But for those files that have been closed, updating isn't possible.

vi. In case of Hadoop we aren't talking about one computer; in this scenario we usually have a large number of computers and hardware failures are unavoidable; sometime one computer will fail and sometimes the entire rack can fail too. Hadoop gives excellent protection against hardware failure; however the performance will go down proportionate to the number of computers that are down. In the big picture, it doesn't really matter and it is not generally noticeable since if you have 100 computers and in them if 3 fail then 97 are still working. So the proportionate loss of performance isn't that noticeable. However, the way Hadoop works there is the loss in performance. Now this loss of performance through hardware failures is something that is managed through replication strategy.

Chapter 4

HADOOP Distributed Filesystem

When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. Filesystems that manage the storage across a network of machines are called distributed filesystems. Since they are network-based, all the complications of network programming kick in, thus making distributed filesystems more complex than regular disk filesystems. For example, one of the biggest challenges is making the filesystem tolerate node failure without suffering data loss.

Hadoop comes with a distributed filesystem called HDFS, which stands for Hadoop Distributed Filesystem. (You may sometimes see references to "DFS"—informally or in older documentation or configurations—which is the same thing.) HDFS is Hadoop's flagship filesystem and is the focus of this chapter, but Hadoop actually has a general- purpose filesystem abstraction, so we'll see along the way how Hadoop integrates with other storage systems (such as the local filesystem and Amazon S3).

THE DESIGN OF HDFS

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.[1] Let's examine this statement in more detail:

VERY LARGE FILES

"Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.

STREAMING DATA ACCESS

HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

COMMODITY HARDWARE

Hadoop doesn't require expensive, highly reliable hardware to run on. It's designed to run on clusters of commodity hardware (commonly available hardware available from multiple vendors[3]) for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

It is also worth examining the applications for which using HDFS does not work so well. While this may change in the future, these are areas where HDFS is not a good fit today:

LOW-LATENCY DATA ACCESS

Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency. HBase is currently a better choice for low-latency access.

Since the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode. As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory. While storing millions of files is feasible, billions is be- yond the capability of current hardware.

MULTIPLE WRITERS, ARBITRARY FILE MODIFICATIONS

Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers, or for modifications at arbitrary offsets in the file. (These might be supported in the future, but they are likely to be relatively inefficient.)

HDFS CONCEPTS

BLOCKS

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. Filesystem blocks are typically a few kilobytes in size, while disk blocks are normally 512 bytes. This is generally transparent to the filesystem user who is simply reading or writing a file—of whatever length. However, there are tools to perform filesystem maintenance, such as df and fsck, that operate on the filesystem block level.

HDFS, too, has the concept of a block, but it is a much larger unit—64 MB by default. Like ina filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage. When unqualified, the term "block" in this book refers to a block in HDFS.

Having ablockabstractionforadistributedfilesystembringsseveralbenefits. Thefirst benefit is the most obvious: a file can be larger than any single disk in the network. There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster. In fact, it would be possible, if unusual, to store a single fileon an HDFS cluster whose blocks filled all the disks in the cluster.

Second, making the unit of abstraction a block rather than a file simplifies the storage subsystem. Simplicity is something to strive for all in all systems, but is especially important for a distributed system in which the failure modes are so varied. The storage subsystem deals with blocks, simplifying storage management (since blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns (blocks are just a chunk of data to be stored—file metadata such as permissions information does not need to be stored with the blocks, so another system can handle metadata separately).

Furthermore, blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three). If a block becomes unavailable, a copy can be read from another location in a way that is trans- parent to the client. A block that is no longer available due to corruption or machine failure can be replicated from its alternative locations to other live machines to bring the replication factor back to the normal level. Similarly, some applications may choose to set a high replication factor for the blocks in a popular file to spread the read load on the cluster.

Like its disk filesystem cousin, HDFS's fsck command understands blocks. For example, running:

% hadoop fsck / -files -blocks will list the blocks that make up each file in the filesystem.

NAMENODES AND DATANODES

An HDFS cluster has two types of node operating in a master-worker pattern: a name- node (the master) and a number of datanodes (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

A client accesses the filesystem on behalf of the user by communicating with the name- node and datanodes. The client presents a POSIX-like filesystem interface, so the user code does not need to know about the namenode and datanode to function.

Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, it is important to make the namenode resilient to failure, and Hadoop provides two mechanisms for this.

The first way is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configu- ration choice is to write to local disk as well as a remote NFS mount.

It is also possible to run a secondary namenode, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged name- space image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode's metadata files that are on NFS to the secondary and run it as the new primary.

HDFS FEDERATION

The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling. HDFS Federation, introduced in the 0.23 release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one namenode might manage all the files rooted under /user, say, and a second namenode might handle files under /share.

Under federation, each namenode manages a namespace volume, which is made up of the metadata for the namespace, and a block pool containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namen- odes. Block pool storage is not partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.

To access a federated HDFS cluster, clients use client-side mount tables to map file paths to namenodes. This is managed in configuration using the ViewFileSystem, and viewfs:// URIs.

HDFS HIGH-AVAILABILITY

The combination of replicating namenode metadata on multiple filesystems, and using the secondary namenode to create checkpoints protects against data loss, but does not provide high-availability of the filesystem. The namenode is still a single point of fail- ure (SPOF), since if it did fail, all clients—including MapReduce jobs—would be un- able to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event the whole Hadoop system would effectively be out of service until a new namenode could be brought online.

To recover from a failed namenode in this situation, an administrator starts a new primary namenode with one of the filesystem metadata replicas, and configures da- tanodes and clients to use this new namenode. The new namenode is not able to serve requests until it has i) loaded its namespace image into memory, ii) replayed its edit log, and iii) received enough block reports from the datanodes to leave safe mode. On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.

The long recovery time is a problem for routine maintenance too. In fact, since unex- pected failure of the namenode is so rare, the case for planned downtime is actually more important in practice.

The 0.23 release series of Hadoop remedies this situation by adding support for HDFS high-availability (HA). In this implementation there is a pair of namenodes in an active- standby configuration. In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant inter- ruption. A few architectural changes are needed to allow this to happen:

The namenodes must use highly-available shared storage to share the edit log. (In the initial implementation of HA this will require an NFS filer, but in future releases more options will be provided, such as a BookKeeper-based system built on Zoo- Keeper.) When a standby namenode comes up it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entries as they are written by the active namenode.

Datanodes must send block reports to both namenodes since the block mappings are stored ina namenode’s memory, and not on disk.

Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

If the active namenode fails, then the standby can take over very quickly (in a few tens of seconds) since it has the latest state available in memory: both the latest edit log entries, and an up-to-date block mapping. The actual observed failover time will be longer in practice (around a minute or so), since the system needs to be conservative in deciding that the active namenode has failed.

In the unlikely event of the standby being down when the active fails, the administrator can still start the standby from cold. This is no worse than the non-HA case, and from an operational point of view it's an improvement, since the process is a standard op- erational procedure built into Hadoop.

FAILOVER AND FENCING

The transition from the active namenode to the standby is managed by a new entity in the system called the failover controller. Failover controllers are pluggable, but the first implementation uses ZooKeeper to ensure that only one namenode is active. Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures (using a simple heartbeating mechanism) and trigger a failover should a namenode fail.

Failover may also be initiated manually by an adminstrator, in the case of routine maintenance, for example. This is known as a graceful failover, since the failover con- troller arranges an orderly transition for both namenodes to switch roles.

In the case of an ungraceful failover, however, it is impossible to be sure that the failed namenode has stopped running. For example, a slow network or a network partition can trigger a failover transition, even though the previously active namenode is still running, and thinks it is still the active namenode. The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as fencing. The system employs a range of fencing mechanisms, including killing the namenode's process, revoking its access to the shared storage directory (typically by using a vendor-specific NFS com- mand), and disabling its network port via a remote management command. As a last resort, the previously active namenode can be fenced with a technique rather graphi- cally known as STONITH, or "shoot the other node in the head", which uses a specialized power distribution unit to forcibly power down the host machine.

Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname which is mapped to a pair of namenode addresses (in the configuration file), and the client library tries each namenode address until the operation succeeds.

BASIC FILESYSTEM OPERATIONS

The filesystem is ready to be used, and we can do all of the usual filesystem operations such as reading files, creating directories, moving files, deleting data, and listing direc- tories. You can type hadoop fs -help to get detailed help on every command.

Start by copying a file from the local filesystem to HDFS:

% hadoop fs -copyFromLocal input/docs/quangle.txt hdfs://localhost/user/tom/quangle.txt

This command invokes Hadoop's filesystem shell command fs, which supports a number of subcommands—in this case, we are running -copyFromLocal. The local file quangle.txtis copiedtothefile/user/tom/quangle.txtonthe HDFSinstancerunning on localhost. In fact, we could have omitted the scheme and host of the URI and picked up the default, hdfs://localhost, as specified in core-site.xml:

% hadoop fs -copyFromLocal input/docs/quangle.txt /user/tom/quangle.txt

We could also have used a relative path and copied the file to our home directory in HDFS, which in this case is /user/tom:

% hadoop fs -copyFromLocal input/docs/quangle.txt quangle.txt

Let's copy the file back to the local filesystem and check whether it's the same:

% hadoop fs -copyToLocal quangle.txt quangle.copy.txt

% md5 input/docs/quangle.txt quangle.copy.txt

MD5 (input/docs/quangle.txt) = a16f231da6b05e2ba7a339320e7dacd9 MD5 (quangle.copy.txt) = a16f231da6b05e2ba7a339320e7dacd9

The MD5 digests are the same, showing that the file survived its trip to HDFS and is back intact.

Finally, let's look at an HDFS file listing. We create a directory first just to see how it is displayed in the listing:

% hadoop fs -mkdir books

% hadoop fs -ls .

Found 2 items

drwxr-xr-x- tom supergroup 0 2009-04-02 22:41 /user/tom/books-rw-r--r--1 tom supergroup 2009-04-02 22:29 /user/tom/quangle.txt

The information returned is very similar to the Unix command ls -l, with a few minor differences. The first column shows the file mode. The second column is the replication factor of the file (something a traditional Unix filesystem does not have). Remember we set the default replication factor in the site-wide configuration to be 1, which is why we see the same value here. The entry in this column is empty for directories since the concept of replication does not apply to them—directories are treated as metadata and stored by the namenode, not the datanodes. The third and fourth columns show the file owner and group. The fifth columnis the size of the file in bytes, or zero for directories. The sixth and seventh columns are the last modified date and time. Finally, the eighth column is the absolute name of the file or directory

HADOOP FILESYSTEMS

Hadoop has an abstract notion of filesystem, of which HDFS is just one implementation. The Java abstract class org.apache. hadoop .fs. File System represents a filesystem in Hadoop, and there are several concrete implementations

Hadoop provides many interfaces to its filesystems, and it generally uses the URI scheme to pick the correct filesystem instance to communicate with. For example, the filesystem shell that we met in the previous section operates with all Hadoop filesys- tems. To list the files in the root directory of the local filesystem, type:

```
% hadoop fs -ls file:///
```

Although it is possible (and sometimes very convenient) to run MapReduce programs that access any of these filesystems, when you are processing large volumes of data, you should choose a distributed filesystem that has the data locality optimization, notably HDFS.

INTERFACES

Hadoop is written in Java, and all Hadoop filesystem interactions are mediated through the Java API. The filesystem shell, for example, is a Java application that uses the Java FileSystem class to provide filesystem operations. The other filesystem interfaces are discussed briefly in this section. These interfaces are most commonly used with HDFS, since the other filesystems in Hadoop typically have existing tools to access the under- lying filesystem (FTP clients for FTP, S3 tools for S3, etc.), but many of them will work with any Hadoop filesystem.

HTTP

There are two ways of accessing HDFS over HTTP: directly, where the HDFS daemons serve HTTP requests to clients; and via a proxy (or proxies), which accesses HDFS on the client's behalf using the usual Distributed File System API.

In the first case, directory listings are served by the namenode's embedded web server (which runs on port 50070) formatted in XML or JSON, while file data is streamed from datanodes by their web servers (running on port 50075).

The original direct HTTP interface (HFTP and HSFTP) was read-only, while the new WebHDFS implementation supports all filesystem operations, including Kerberos authentication. Web HDFS must be enabled by setting dfs. web hdfs. enabled to true, for you to be able to use webhdfs URIs.

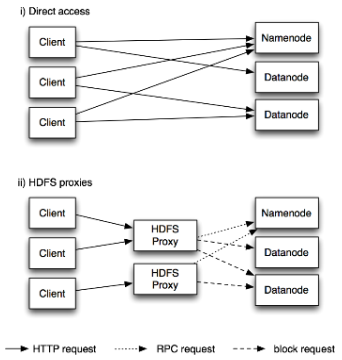


Figure 3-1. Accessing HDFS over HTTP directly, and via a bank of HDFS proxies

The second way of accessing HDFS over HTTP relies on one or more standalone proxy servers. (The proxies are stateless so they can run behind a standard load balancer.) All traffic to the cluster passes through the proxy. This allows for stricter firewall and bandwidth limiting policies to be put in place. It's common to use a proxy for transfers between Hadoop clusters located in different data centers.

The original HDFS proxy (in src/contrib/hdfsproxy) was read-only, and could be ac- cessed by clients using the HSFTP FileSystem implementation (hsftp URIs). From re- lease 0.23, there is a new proxy called HttpFS that has read and write capabilities, and which exposes the same HTTP interface as WebHDFS, so clients can access either using webhdfs URIs.

The HTTP REST API that WebHDFS exposes is formally defined in a specification, so it is likely that over time clients in languages other than Java will be written that use it directly.

C

Hadoop provides a C library called libhdfs that mirrors the Java FileSystem interface (it was written as a C library for accessing HDFS, but despite its name it can be used to access any Hadoop filesystem). It works using the Java Native Interface (JNI) to call a Java filesystem client.

The C API is very similar to the Java one, but it typically lags the Java one, so newer features may not be supported. You can find the generated documentation for the C API in the libhdfs/docs/api directory of the Hadoop distribution.

Hadoop comes with prebuilt libhdfs binaries for 32-bit Linux, but for other platforms, you will need to build them yourself using the instructions at http://wiki.apache.org/hadoop/LibHDFS.

FUSE

Filesystem in Userspace (FUSE) allows filesystems that are implemented in user space to be integrated as a Unix filesystem. Hadoop's Fuse-DFS contrib module allows any Hadoop filesystem (but typically HDFS) to be mounted as a standard filesystem. You can then use Unix utilities (such as ls and cat) to interact with the filesystem, as well as POSIX libraries to access the filesystem from any programming language.

Fuse-DFS is implemented in C using libhdfs as the interface to HDFS. Documentation for compiling and running Fuse-DFS is located in the src/contrib/fuse-dfs directory of the Hadoop distribution.

THE JAVAINTERFACE

In this section, we dig into the Hadoop's FileSystem class: the API for interacting with one of Hadoop's filesystems.[5] While we focus mainly on the HDFS implementation, DistributedFileSystem, in general you should strive to write your code against the FileSystem abstract class, to retain portability across filesystems. This is very useful when testing your program, for example, since you can rapidly run tests using data stored on the local filesystem.

READING DATA FROM A HADOOP URL

One of the simplest ways to read a file from a Hadoop filesystem is by using a java.net.URL object to open a stream to read the data from. The general idiom is:


```
InputStream in = null; try {
in = new URL("hdfs://host/path").openStream();
// process in
} finally { IOUtils.closeStream(in);
}
```

There's a little bit more work required to make Java recognize Hadoop's hdfs URL scheme. This is achieved by calling the setURLStreamHandlerFactory method on URL

1. From release 0.21.0, there is a new filesystem interface called FileContext with better handling of multiple filesystems (so a single FileContext can resolve multiple filesystem schemes, for example) and a cleaner, more consistent interface.

2. with an instance of Fs Url Stream Handler Factory. This method can only be called once per JVM, so it is typically executed in a static block. This limitation means that if some other part of your program—perhaps a third-party component outside your control— sets a URL Stream Handler Factory, you won't be able to use this approach for reading data from Hadoop. The next section discusses an alternative.

Program for displaying files from Hadoop filesystems on standard output, like the Unix cat command.

Example 3-1. Displaying files from a Hadoop filesystem on standard output using a URLStreamHandler

```
public class URLCat {

3.
static {
URL.setURLStreamHandlerFactory(new    FsUrlStreamHandlerFactory());
}

4
public static void main(String[] args) throws Exception { InputStream in = null;
try {
in = new URL(args[0]).openStream(); IOUtils.copyBytes(in, System.out, 4096, false);
} finally { IOUtils.closeStream(in);
}
}
```


We make use of the handy IOUtils class that comes with Hadoop for closing the stream in the finally clause, and also for copying bytes between the input stream and the output stream (System.out in this case). The last two arguments to the copyBytes method are the buffer size used for copying and whether to close the streams when the copy is complete. We close the input stream ourselves, and System.out doesn't need to be closed.

READING DATA USING THE FILESYSTEM API

As the previous section explained, sometimes it is impossible to set a URLStreamHand lerFactory for your application. In this case, you will need to use the FileSystem API to open an input stream for a file.

A file in a Hadoop filesystem is represented by a Hadoop Path object (and not a java.io.File object, since its semantics are too closely tied to the local filesystem). You can think of a Path as a Hadoop filesystem URI, such as hdfs://localhost/user/tom/ quangle.txt.

FileSystem is a general filesystem API, so the first step is to retrieve an instance for the filesystem we want to use—HDFS in this case. There are several static factory methods for getting a FileSystem instance:

```
public static FileSystem get(Configuration conf) throws IOException

public static FileSystem get(URI uri, Configuration conf) throws IOException

public static FileSystem get(URI uri, Configuration conf, String user) throws IOException
```

A Configuration object encapsulates a client or server's configuration, which is set using configuration files read from the classpath, such as conf/core-site.xml. The first method returns the default filesystem (as specified in the file conf/core-site.xml, or the default local filesystem if not specified there). The second uses the given URI's scheme and authority to determine the filesystem to use, falling back to the default filesystem if no scheme is specified in the given URI. The third retrieves the filesystem as the given user.

In some cases, you may want to retrieve a local filesystem instance, in which case you can use the convenience method, getLocal():

```
public static LocalFileSystem getLocal(Configuration conf) throws IOException
```

With a FileSystem instance in hand, we invoke an open() method to get the input stream for afile:

```
public FSDataInputStream open(Path f) throws IOException

public abstract FSDataInputStream open(Path f, int bufferSize) throws IOException
```

The first method uses a default buffer size of 4 K.

Putting this together, we can rewrite Example 3-1 as shown in Example 3-2.

Example 3-2. Displaying files from a Hadoop filesystem on standard output by using the FileSystem directly

```
public class FileSystemCat {

public static void main(String[] args) throws Exception { String uri = args[0];

Configuration conf = new Configuration();

FileSystem fs = FileSystem.get(URI.create(uri), conf); InputStream in = null;
```

```
try {
in = fs.open(new Path(uri)); IOUtils.copyBytes(in, System.out, 4096, false);
} finally { IOUtils.closeStream(in);
}
}
}
```

FS Data Input Stream

The open() method on FileSystem actually returns a FSDataInputStream rather than a standard java.io class. This class is a specialization of java.io.DataInputStream with support for random access, so you can read from any part of the stream:

```
package org.apache.hadoop.fs;
public class FSDataInputStream extends DataInputStream implements Seekable, PositionedReadable {
// implementation elided
}
```


The Seekable interface permits seeking to a position in the file and a query method for the current offset from the start of the file (getPos()):

```
public interface Seekable {
void seek(long pos) throws IOException; long getPos() throws IOException;
}
```


Calling seek() with a position that is greater than the length of the file will result in an IOException. Unlike the skip() method of java.io.InputStream that positions the stream at a point later than the current position, seek() can move to an arbitrary, ab- solute positionin the file.

Example 3-3 is a simple extension of Example 3-2 that writes a file to standard out twice: after writing it once, it seeks to the start of the file and streams through it once again.

Example 3-3. Displaying files from a Hadoop filesystem on standard output twice, by using seek

```
public class FileSystemDoubleCat {

public static void main(String[] args) throws Exception { String uri = args[0];

Configuration conf = new Configuration();

FileSystem fs = FileSystem.get(URI.create(uri), conf); FSDataInputStream in = null;

try {

in = fs.open(new Path(uri)); IOUtils.copyBytes(in, System.out, 4096, false); in.seek(0); // go back to the start of the file IOUtils.copyBytes(in, System.out, 4096, false);

} finally { IOUtils.closeStream(in);

}    }        }
```

Here's the result of running it on a small file:

FSDataInputStream also implements the PositionedReadable interface for reading parts of a file at a given offset:

```
public interface PositionedReadable {

public int read(long position, byte[] buffer, int offset, int length) throws IOException;

public void readFully(long position, byte[] buffer, int offset, int length) throws IOException;

public void readFully(long position, byte[] buffer) throws IOException;

}
```


The read() method reads up to length bytes from the given position in the file into the buffer at the given offset in the buffer. The return value is the number of bytes actually read: callers should check this value as it may be less than length. The readFully() methods will read length bytes into the buffer (or buffer.length bytes for the version that just takes a byte array buffer), unless the end of the file is reached, in which case an EOFException is thrown.

All of these methods preserve the current offset in the file and are thread-safe, so they provide a convenient way to access another part of the file—metadata perhaps—while reading the main body of the file. In fact, they are just implemented using the Seekable interface using the following pattern:

```
long oldPos = getPos(); try {

seek(position);

// read data

} finally { seek(oldPos);

}
```

Finally, bear in mind that calling seek() is a relatively expensive operation and should be used sparingly. You should structure your application access patterns to rely on streaming data, (by using MapReduce, for example) rather than performing a large number of seeks.

WRITING DATA

The File System class has a number of methods for creating a file. The simplest is the method that takes a Path object for the file to be created and returns an output stream to write to:

```
public FS Data Output Stream create(Path f) throws IOException
```

There are overloaded versions of this method that allow you to specify whether to forcibly overwrite existing files, the replication factor of the file, the buffer size to use when writing the file, the block size for the file, and file permissions.

There's also an overloaded method for passing a callback interface, Progressable, so your application can be notified of the progress of the data being written to the datanodes:

```
package org.apache.hadoop.util;

public interface Progressable { public void progress();

}
```

As an alternative to creating a new file, you can append to an existing file using the append() method (there are also some other overloaded versions):

```
public FSDataOutputStream append(Path f) throws IOException
```

The append operation allows a single writer to modify an already written file by opening it and writing data from the final offset in the file. With this API, applications that produce unbounded files, such as logfiles, can write to an existing file after a restart, for example. The append operation is optional and not implemented by all Hadoop filesystems. For example, HDFS supports append, but S3 filesystems don't.

To copy a local file to a Hadoop filesystem. We illustrate pro- gress by printing a period every time the progress() method is called by Hadoop, which is after each 64 K packet of data is written to the datanode pipeline. (Note that this particular behavior is not specified by the API, so it is subject to change in later versions of Hadoop. The API merely allows you to infer that "something is happening.")

Example 3-4. Copying a local file to a Hadoop filesystem

```
public class FileCopyWithProgress {

public static void main(String[] args) throws Exception { String localSrc = args[0];

String dst = args[1];

InputStream in = new BufferedInputStream(new FileInputStream(localSrc));
Configuration conf = new Configuration();

FileSystem fs = FileSystem.get(URI.create(dst), conf); OutputStream out =
fs.create(new Path(dst), new Progressable() {

public void progress() { System.out.print(".");

}
```



```
});

IOUtils.copyBytes(in, out, 4096, true);

}

}
```

Typical usage:

```
% hadoop FileCopyWithProgress input/docs/1400-8.txt hdfs://localhost/user/tom/ 1400-8.txt
```

Currently, none of the other Hadoop filesystems call progress() during writes. Progress is important in MapReduce applications, as you will see in later chapters.

FS Data Output Stream

The create() method on FileSystem returns an FSDataOutputStream, which, like

FSDataInputStream, has a method for querying the current position in the file:

```
package org.apache.hadoop.fs;
```

```
public class FSDataOutputStream extends DataOutputStream implements Syncable {

public long getPos() throws IOException {

// implementation elided

}

// implementation elided

}
```

However, unlike FS Data Input Stream, FS Data Output Stream does not permit seeking. This is because HDFS allows only sequential writes to an open file or appends to an already written file. In other words, there is no support for writing to anywhere other than the end of the file, so there is no value in being able to seek while writing.

DATA FLOW

ANATOMY OF A FILE READ

To get an idea of how data flows between the client interacting with HDFS, the name- node and the datanodes, which shows the main sequence of events when reading a file.

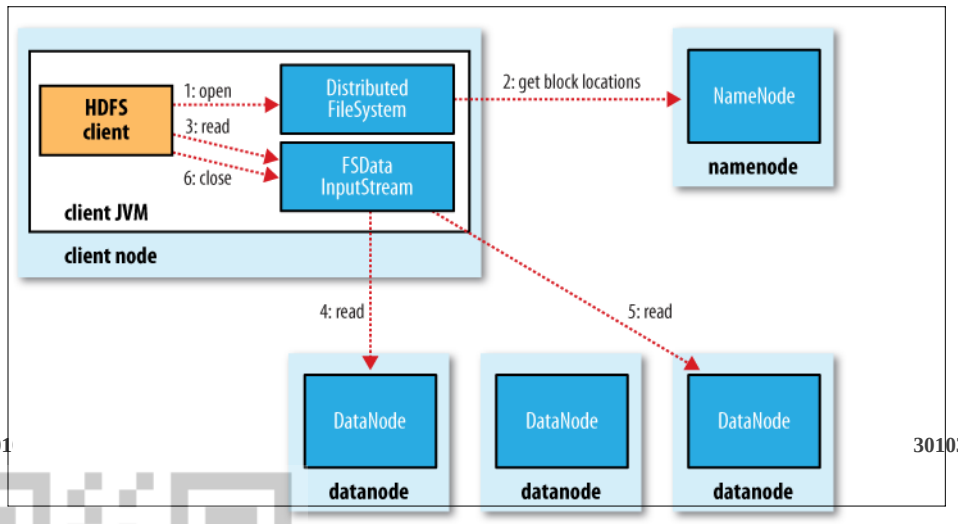


Figure 3-2. A client reading data from HDFS

The client opens the file it wishes to read by calling open() on the File System object, which for HDFS is an instance of Distributed File System Distributed File System calls the

namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file (step 2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client (according to the top- ology of the cluster's network; see "Network Topology and Hadoop"). If the client is itself a datanode (in the case of a MapReduce task, for instance), then it will read from the local datanode, if it hosts a copy of the block.

The DistributedFileSystem returns an FSDataInputStream (an input stream that sup- ports file seeks) to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O.

The client then calls read() on the stream (step 3). DFSInputStream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls read() repeatedly on the stream (step 4). When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream.

Blocks are read in order with the DFSInputStream opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it callsclose() on the FSDataInputStream (step 6).

During reading, if the DFSInputStream encounters an error while communicating with a datanode, then it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The DFSInputStream also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, it is reported to the namenode before the DFSInput Stream attempts to read a replica of the block from another datanode.

One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients, since the data traffic is spread across all the datanodes in the cluster. The namenode meanwhile merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bot- tleneck as the number of clients grew.

ANATOMY OF A FILE WRITE

Next we'll look at how files are written to HDFS. Although quite detailed, it is instructive to understand the data flow since it clarifies HDFS's coherency model.

The case we're going to consider is the case of creating a new file, writing data to it, then closing the file.

The client creates the file by calling create() on Distributed Filesystem (step 1 in Distributed Filesystem makes an RPC call to the name node to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The name- node performsvarious checks to make sure the file doesn't already exist, and that the client has the right permissions to create the file. If these checks pass, the name node makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException. The Distributed Filesystem returns an FS Data Output Stream for the client

to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutput Stream, which handles communication with the datanodes and namenode.

As the client writes data (step 3), DFS Output Stream splits it into packets, which it writes to an internal queue, called the data queue. The data queue is consumed by the Data Streamer, whose responsibility it is to ask the name node to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline—we'll assume the replication level is three, so there are three nodes in the pipeline. The Data Streamer streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).

DFS Output Stream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the ack queue. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline (step 5).

If a datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. First the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. The current block on the good datanodes is given a new identity, which is communicated to the name- node, so that the partial block on the failed datanode will be deleted if the failed.

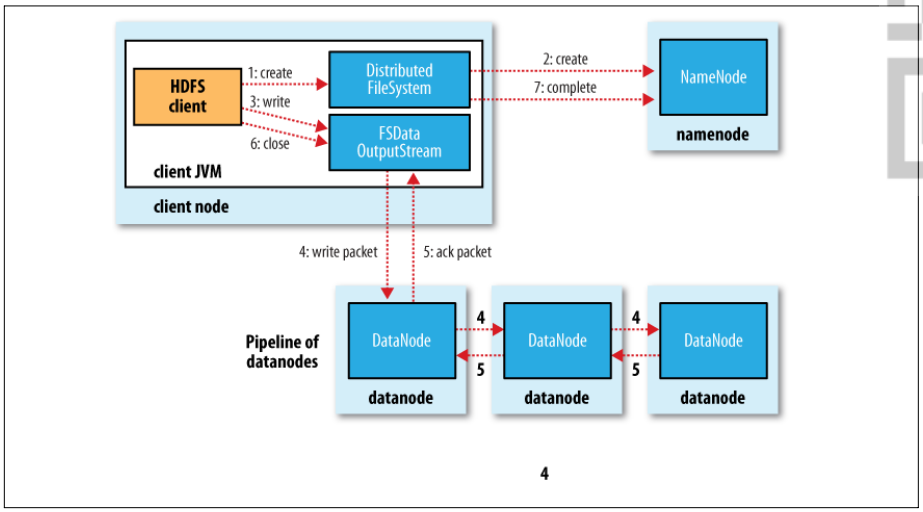


Figure 3-4. A client writing data to HDFS

datanode recovers later on. The failed datanode is removed from the pipeline and the remainder of the block's data is written to the two good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal.

It's possible, but unlikely, that multiple datanodes fail while a block is being written. As long as dfs.replication .min replicas (default one) are written, the write will succeed, and the blockwill be asynchronously replicated across the cluster until its target rep- lication factor is reached (dfs.replication, which defaults to three).

When the client has finished writing data, it calls close() on the stream (step 6). This action flushes all the remaining packets to the datanode pipeline and waits for ac- knowledgments before contacting the namenode to signal that the file is complete (step 7). The namenode already knows which blocks the file is made up of (via Data Streamer asking for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

LIMITATIONS

30103162101496 30103162101496 30103162101496

There are a few limitations to be aware of with HAR files. Creating an archive creates a copy of the original files, so you need as much disk space as the files you are archiving to create the archive (although you can delete the originals once you have created the archive). There is currently no support for archive compression, although the files that go into the archive can be compressed (HAR files are like tar files in this respect).

Archives are immutable once they have been created. To add or remove files, you must re-create the archive. In practice, this is not a problem for files that don't change after being written, since they can be archived in batches on a regular basis, such as daily or weekly.

As noted earlier, HAR files can be used as input to MapReduce. However, there is no archive-aware InputFormat that can pack multiple files into a single MapReduce split, so processing lots of small files, even in a HAR file, can still be inefficient**. "Small files and Combine File Input Format"** discusses another approach to this problem.

Finally, if you are hitting namenode memory limits even after taking steps to minimize the number of small files in the system, then consider using HDFS Federation to scale the namespace

Chapter 5

Understanding MAPREDUCE Fundamentals

MapReduce

1. Traditional Enterprise Systems normally have a centralized server to store and process data.
2. The following illustration depicts a schematic view of a traditional enterprise system. Traditionalmodel is certainly not suitable to process huge volumes of scalable data and cannot be accommodated by standard database servers.
3. Moreover, the centralized system creates too much of a bottleneck while processing multiple filessimultaneously.

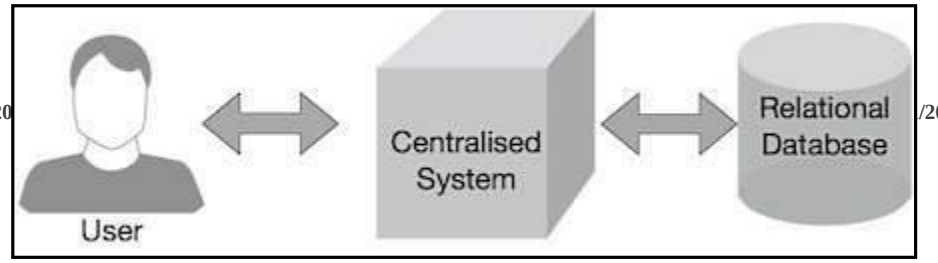


Figure 4.1: MapReduce

4. Google solved this bottleneck issue using an algorithm called MapReduce. MapReduce divides a task into small parts and assigns them to many computers.
5. Later, the results are collected at one place and integrated to form the result dataset.

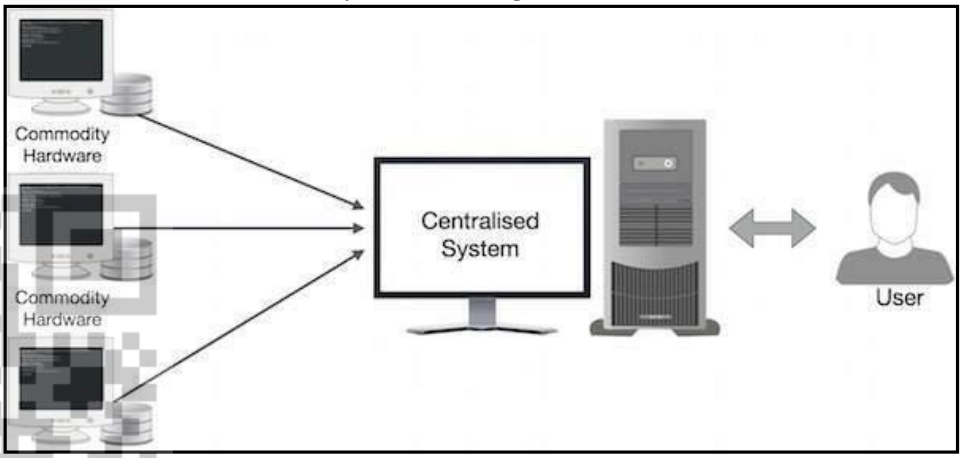


Figure 4.2: Physical structure

6. A MapReduce computation executes as follows:

- Some number of Map tasks each are given one or more chunks from a distributed file system. These Map tasks turn the chunk into a sequence of key-value pairs. The way key-value pairs are producedfrom the input data is determined by the code written by the user for the Map function.
- The key-value pairs from each Map task are collected by a master controller and sorted by key. Thekeys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at thesame Reduce task.
- The Reduce tasks work on one key at a time, and combine all the values associated with that key insome way. The manner of combination of values is determined by the code written by the user for the Reduce function.

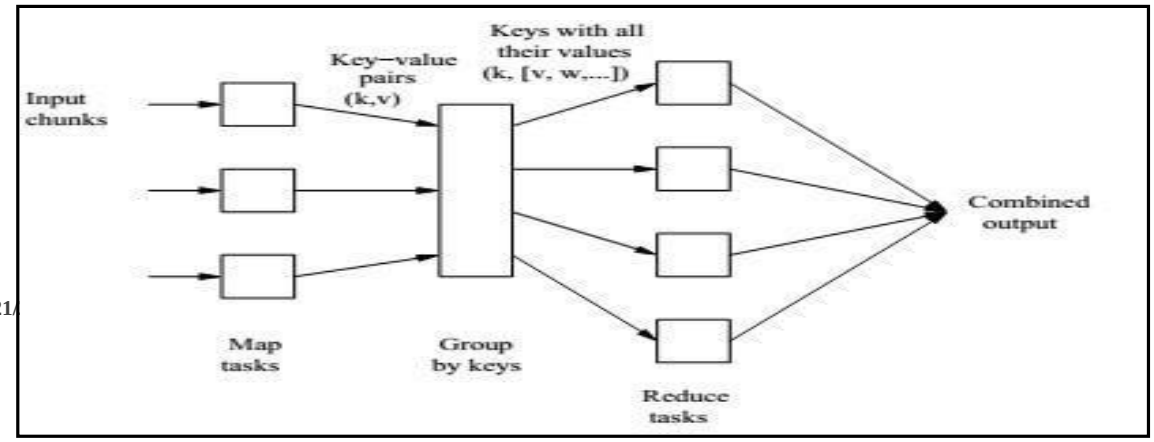


Figure 4.3: Schematic MapReduce Computation

A. The Map Task

i. We view input files for a Map task as consisting of elements, which can be any type: a tuple or adocument, for example.

ii. A chunk is a collection of elements, and no element is stored across two chunks.

iii. Technically, all inputs to Map tasks and outputs from Reduce tasks are of the key-value-pair form,but normally the keys of input elements are not relevant and we shall tend to ignore them.

iv. Insisting on this form for inputs and outputs is motivated by the desire to allow composition ofseveral MapReduce processes.

v. The Map function takes an input element as its argument and produces zero or more key-valuepairs.

vi. The types of keys and values are each arbitrary. vii. Further, keys are not “keys” in the usualsense; they do not have to be unique.

vii. Rather a Map task can produce several key-value pairs with the same key, even from the sameelement.

Example 1: A MapReduce computation with what has become the standard example application: counting the number of occurrences for each word in a collection of documents. In this example, theinput file is a repository of documents, and each document is an element. The Map function for thisexample uses keys that are of type String (the words) and values that are integers. The Map task readsa document and breaks it into its sequence of words w_1, w_2, . . . , w_n. It then emits a sequence of key-value pairs where the value is always 1. That is, the output of the Map task for this document is the sequence of key-value pairs:

$(w_1, 1), (w_2, 1), \ldots, (w_n, 1)$

A single Map task will typically process many documents – all the documents in one or more chunks. Thus, its output will be more than the sequence for the one document suggested above. If a word *w* appears *m* times among all the documents assigned to that process, then there will be m key-value pairs (w, 1) among its output. An option, is to combine these m pairs into a single pair (w, m), but we can only do that because, the Reduce tasks apply an associative and commutative operation, addition, to the values.

B. Grouping by Key

i. As the Map tasks have all completed successfully, the key-value pairs are grouped by key, and thevalues associated with each key are formed into a list of values.

ii. The grouping is performed by the system, regardless of what the Map and Reduce tasks do.

iii. The master controller process knows how many Reduce tasks there will be, say r such tasks.

iv. The user typically tells the MapReduce system what r should be.

v. Then the master controller picks a hash function that applies to keys and produces a bucket number from 0 to r − 1.

vi. Each key that is output by a Map task is hashed and its key-value pair is put in one of r local files. Each file is destined for one of the Reduce tasks.1.

vii. To perform the grouping by key and distribution to the Reduce tasks, the master controller mergesthe files from each Map task that are destined for a particular Reduce task and feeds the merged file to that process as a sequence of key-list-of-value pairs.

viii. That is, for each key k, the input to the Reduce task that handles key k is a pair of the form (k, [v1, v2, . . . , vn]), where (k, v1), (k, v2), . . . , (k, vn) are all the key-value pairs with keyk coming from all the Map tasks.

C. The Reduce Task

i. The Reduce function's argument is a pair consisting of a key and its list of associated values.

ii. The output of the Reduce function is a sequence of zero or more key-value pairs.

iii. These key-value pairs can be of a type different from those sent from Map tasks to Reduce tasks,but often they are the same type.

iv. We shall refer to the application of the Reduce function to a single key and its associated list ofvalues as a reducer. A Reduce task receives one or more keys and their associated value lists.

v. That is, a Reduce task executes one or more reducers. The outputs from all the Reduce tasks aremerged into a single file.

vi. Reducers may be partitioned among a smaller number of Reduce tasks is by hashing the keys andassociating each

vii. Reduce task with one of the buckets of the hash function.

The Reduce function simply adds up all the values. The output of a reducer consists of the word andthe sum. Thus, the output of all the Reduce tasks is a sequence of (w, m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences ofw among all those documents.

D. Combiners

i. A Reduce function is associative and commutative. That is, the values to be combined can be combined in any order, with the same result.

ii. The addition performed in Example 1 is an example of an associative and commutative operation. It doesn't matter how we group a list of numbers $v_1, v_2, \ldots, v_n$; the sum will be the same.

iii. When the Reduce function is associative and commutative, we can push some of what the reducers do to the Map tasks

iv. These key-value pairs would thus be replaced by one pair with key w and value equal to the sumof all the 1's in all those pairs.

v. That is, the pairs with key w generated by a single Map task would be replaced by a pair (w, m),where m is the number of times that w appears among the documents handled by this Map task.

E. Details of MapReduce task

The MapReduce algorithm contains two important tasks, namely Map and Reduce.

i. The Map task takes a set of data and converts it into another set of data, where individual elementsare broken down into tuples (key-value pairs).

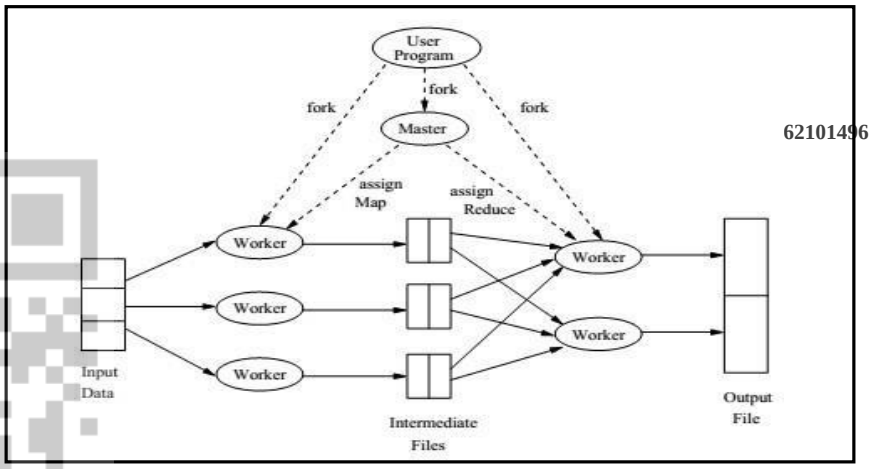


Figure 4.4: Overview of the execution of a MapReduce program

ii. The Reduce task takes the output from the Map as an input and combines those data tuples (key-value pairs) into a smaller set of tuples.

iii. The reduce task is always performed after the map job.

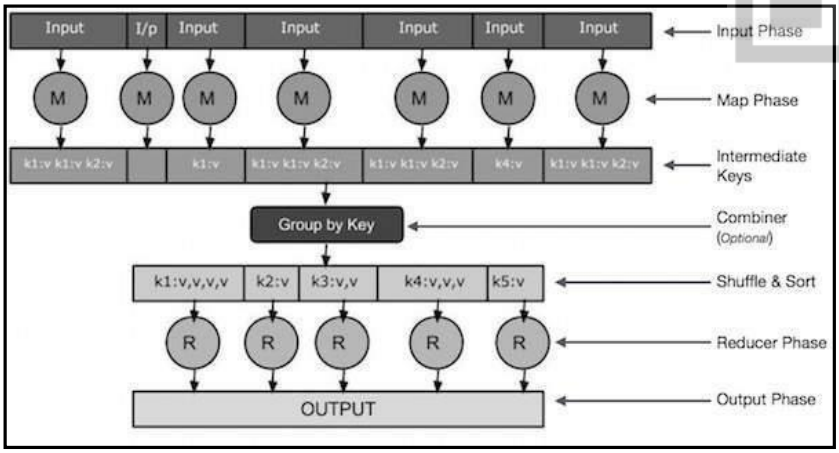


Figure 4.5: Reduce job

- **Input Phase** − Here we have a Record Reader that translates each record in an input file and sendsthe parsed data to the mapper in the form of key-value pairs.
- **Map** − Map is a user-defined function, which takes a series of key-value pairs and processes eachone of them to generate zero or more key-value pairs.

- **Intermediate Keys** − they key-value pairs generated by the mapper are known as intermediate keys.
- **Combiner** − A combiner is a type of local Reducer that groups similar data from the map phase into identifiable sets. It takes the intermediate keys from the mapper as input and applies a user-defined code to aggregate the values in a small scope of one mapper. It is not a part of the main MapReducealgorithm; it is optional.
- **Shuffle and Sort** − The Reducer task starts with the Shuffle and Sort step. It downloads the grouped key-value pairs onto the local machine, where the Reducer is running. The individual key-value pairs are sorted by key into a larger data list. The data list groups the equivalent keys together so that theirvalues can be iterated easily in the Reducer task.
- **Reducer** − The Reducer takes the grouped key-value paired data as input and runs a Reducer function on each one of them. Here, the data can be aggregated, filtered, and combined in a number of ways, and it requires a wide range of processing. Once the execution is over, it gives zero or more key- value pairs to the final step.

- **Output Phase** − In the output phase, we have an output formatter that translates the final key-valuepairs from the Reducer function and writes them onto a file using a record writer.

iv. The MapReduce phase

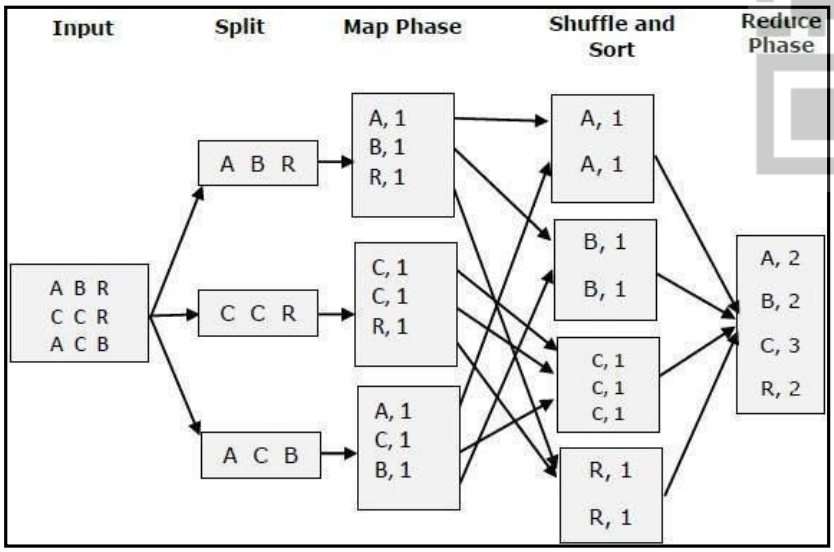


Figure 46 The MapReduce Phase

F. MapReduce-Example

Twitter receives around 500 million tweets per day, which is nearly 3000 tweets per second. The following illustration shows how Tweeter manages its tweets with the help of MapReduce.

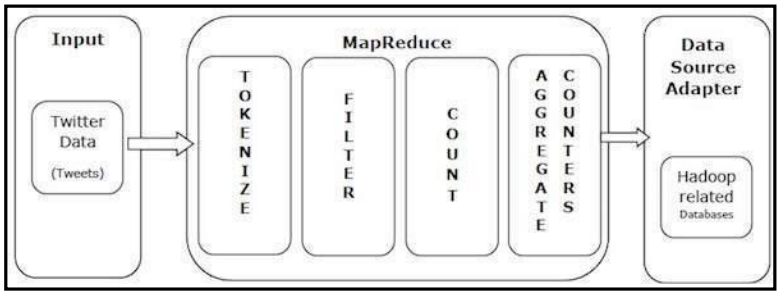


Figure4.7: Example

i. **Tokenize** – Tokenizes the tweets into maps of tokens and writes them as key-value pairs.

ii. **Filter** – Filters unwanted words from the maps of tokens and writes the filtered maps as key-value pairs.

iii. **Count** – Generates a token counter per word.

iv. **Aggregate Counters** – Prepares an aggregate of similar counter values into small manageable units.

G. MapReduce – Algorithm

The MapReduce algorithm contains two important tasks, namely Map and Reduce.

i. The map task is done by means of Mapper Class

Mapper class takes the input, tokenizes it, maps and sorts it. The output of Mapper class is used as input by Reducer class, which in turn searches matching pairs and reduces them.

ii. The reduce task is done by means of Reducer Class.

MapReduce implements various mathematical algorithms to divide a task into small parts and assign them to multiple systems. In technical terms, MapReduce algorithm helps in sending the Map& Reduce tasks to appropriate servers in a cluster.

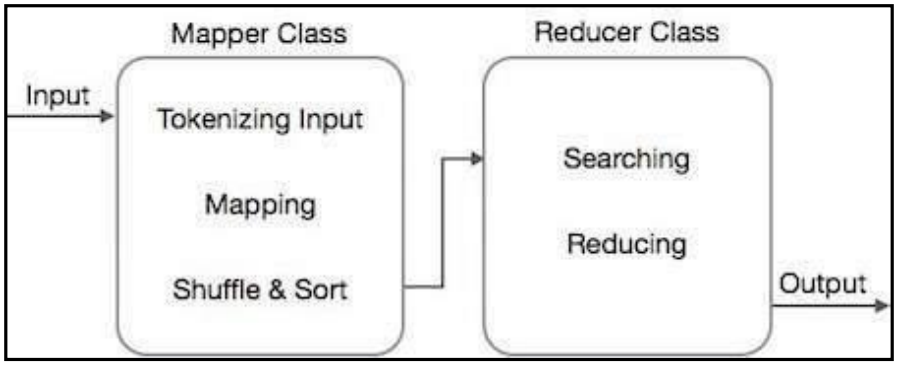


Figure 4.8: The MapReduce Class

H. Coping With Node Failures

i. The worst thing that can happen is that the compute node at which the Master is executing fails. In this case, the entire MapReduce job must be restarted.

ii. But only this one node can bring the entire process down; other failures will be managed by the Master, and the MapReduce job will complete eventually.

iii. Suppose the compute node at which a Map worker resides fails. This failure will be detected by theMaster, because it periodically pings the Worker processes.

iv. All the Map tasks that were assigned to this Worker will have to be redone, even if they had completed. The reason for redoing completed Map asks is that their output destined for the Reduce tasks resides at that compute node, and is now unavailable to the Reduce tasks.

v. The Master sets the status of each of these Map tasks to idle and will schedule them on a Worker when one becomes available.

vi. The Master must also inform each Reduce task that the location of its input from that Map task has changed. Dealing with a failure at the node of a Reduce worker is simpler.

vii. The Master simply sets the status of its currently executing Reduce tasks to idle. These will berescheduled on another reduce worker later

Chapter 6

Introduction to MONGODB AND MAPREDUCE Programming

MongoDB is a cross-platform, document-oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database

Database is a physical container for collections. Each database gets its own set of files on the filesystem. A single MongoDB server typically has multiple databases.

Collection

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

The following table shows the relationship of RDBMS terminology with MongoDB.

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents

Primary Key	Primary Key (Default key _id provided by MongoDB itself)
Database Server and Client	
mysqld/Oracle	mongod
mysql/sqlplus	mongo

Sample Document

Following example shows the document structure of a blog site, which is simply a comma separated key value pair.

```
{
   _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
   description: 'MongoDB is no sql database',
   by: 'tutorials point',
   url: 'http://www.tutorialspoint.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100,
   comments: [
      {
         user:'user1',
         message: 'My first comment',
         dateCreated: new Date(2011,1,20,2,15),
         like: 0
```

```
    },
    {
      user:'user2',
      message: 'My second comments',
      dateCreated: new Date(2011,1,25,7,45),
      like: 5
    }
  ]
}
```

_id is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide _id while inserting the document. If you don't provide then MongoDB provides a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental VALUE.

Any relational database has a typical schema design that shows number of tables and the relationship between these tables. While in MongoDB, there is no concept of relationship.

Advantages of MongoDB over RDBMS

- **Schema less** − MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- Structure of a single object is clear.
- No complex joins.
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.

- Tuning.
- **Ease of scale-out** − MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.
- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

Why Use MongoDB?

- **Document Oriented Storage** − Data is stored in the form of JSON style documents.
- Index on any attribute
- Replication and high availability
- Auto-Sharding
- Rich queries
- Fast in-place updates
- Professional support by MongoDB

Where to Use MongoDB?

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

MongoDB supports many datatypes. Some of them are −

- **String** − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** − This type is used to store a boolean (true/ false) value.
- **Double** − This type is used to store floating point values.
- **Min/ Max keys** − This type is used to compare a value against the lowest and highestBSON elements.
- **Arrays** − This type is used to store arrays or list or multiple values into one key.

- **Timestamp** − ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object** − This datatype is used for embedded documents.
- **Null** − This type is used to store a Null value.
- **Symbol** − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.

- **Date** − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** − This datatype is used to store the document's ID.
- **Binary data** − This datatype is used to store binary data.
- **Code** − This datatype is used to store JavaScript code into the document.
- **Regular expression** − This datatype is used to store regular expression.

The find() Method

To query data from MongoDB collection, you need to use MongoDB's **find()** method.Syntax

The basic syntax of **find()** method is as follows −

```
>db.COLLECTION_NAME.find()
```

find() method will display all the documents in a non-structured way.

Example

Assume we have created a collection named mycol as −

```
> use sampleDB switched
to db sampleDB
> db.createCollection("mycol")
{ "ok" : 1 }
>
```

And inserted 3 documents in it using the insert() method as shown below −

```
> db.mycol.insert([
        {
                title: "MongoDB Overview",
                description: "MongoDB is no SQL database",
                by: "tutorials point",
                url:  "http://www.tutorialspoint.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 100
        },
        {
                title: "NoSQL Database",
                description: "NoSQL database doesn't have tables",
                by: "tutorials point",
                url:  "http://www.tutorialspoint.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 20,
                comments: [
```

```
                {
                        user:"user1",
                        message: "My first comment",
                        dateCreated: new Date(2013,11,10,2,35),
                        like: 0
                }
        ]
    }
])
```

Following method retrieves all the documents in the collection −

```
> db.mycol.find()
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534c"), "title" : "MongoDB Overview", "description"
: "MongoDB is no SQL database", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com","tags" : [
"mongodb", "database", "NoSQL" ], "likes" : 100 }
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534d"), "title" : "NoSQL Database", "description" : "NoSQL
database doesn't have tables", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com", "tags" : [
"mongodb", "database", "NoSQL" ], "likes" : 20, "comments" : [ { "user" : "user1", "message" : "My first
comment", "dateCreated" : ISODate("2013-12-09T21:05:00Z"), "like" : 0 } ] }
>
```

The pretty() Method

To display the results in a formatted way, you can use pretty() method.

Syntax

>db.COLLECTION_NAME.find().pretty()Example

Following example retrieves all the documents from the collection named mycol and arrangesthem in an easy-to-read format.

```
> db.mycol.find().pretty()
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
```

```
}
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534d"),
        "title" : "NoSQL Database",
        "description" : "NoSQL database doesn't have tables","by"
        : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 20,
        "comments" : [
                {
                        "user" : "user1",
                        "message" : "My first comment",
                        "dateCreated" : ISODate("2013-12-09T21:05:00Z"),
                        "like" : 0
                }
        ]
}
```


The findOne() method

Apart from the find() method, there is **findOne()** method, that returns only one document.Syntax

>db.COLLECTIONNAME.findOne()

Example

Following example retrieves the document with title MongoDB Overview.

```
> db.mycol.findOne({title: "MongoDB Overview"})
{
        "_id" : ObjectId("5dd6542170fb13eec3963bf0"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
}
```

RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:{$eg;<value>}}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50
Values in an array	{<key>:{$in:[<value1>, <value2>,……<valueN>]}}	db.mycol.find({"name":{$in:["Raj", "Ram", "Raghu"]}}).pretty()	Where name matches any of thevalue in :["Raj", "Ram", "Raghu"]

2021/2022 30103162101496 2021/2022 30103162101496

Values not in an array	{<key>:{$nin:<value>}}	db.mycol.find({"name":{$nin:["Ramu", "Raghav"]}}).pretty()	Where name values is not in the array :["Ramu", "Raghav"] or, doesn’texist at all

AND in MongoDB

Syntax

To query documents based on the AND condition, you need to use $and keyword. Following isthe basic syntax of AND –

>db.mycol.find({ $and: [{<key1>:<value1>}, { <key2>:<value2>}] })Example

Following example will show all the tutorials written by 'tutorials point' and whose title is 'MongoDB Overview'.


```
> db.mycol.find({$and:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
}
```


For the above given example, equivalent where clause will be **' where by = 'tutorials point' AND title = 'MongoDB Overview' '**. You can pass any number of key, value pairs in find clause.

OR in MongoDB

Syntax

To query documents based on the OR condition, you need to use **$or** keyword. Following is thebasic syntax of **OR** −

```
>db.mycol.find(
   {
      $or: [
         {key1: value1}, {key2:value2}
      ]
   }
).pretty()
```

Example

Following example will show all the tutorials written by 'tutorials point' or whose title is 'MongoDB Overview'.

```
>db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com", "tags":
   ["mongodb", "database", "NoSQL"],"likes":
   "100"
}
>
```


Using AND and OR Together

Example

The following example will show the documents that have likes greater than 10 and whose title is either 'MongoDB Overview' or by is 'tutorials point'. Equivalent SQL where clause is **'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')'**

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},
   {"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

NOR in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword. Following isthe basic syntax of **NOT** −

```
>db.COLLECTION_NAME.find(
        {
                $not: [
                        {key1: value1}, {key2:value2}
                ]
        }
)
```

Example

Assume we have inserted 3 documents in the collection **empDetails** as shown below −

```
db.empDetails.insertMany(
        [
                {
                        First_Name: "Radhika",
                        Last_Name: "Sharma",
                        Age: "26",
                        e_mail: "radhika_sharma.123@gmail.com",
                        phone: "9000012345"
                },
                {
                        First_Name: "Rachel",
                        Last_Name: "Christopher",
                        Age: "27",
                        e_mail: "Rachel_Christopher.123@gmail.com",
                        phone: "9000054321"
                },
                {
                        First_Name: "Fathima",
                        Last_Name: "Sheik",
                        Age: "24",
                        e_mail: "Fathima_Sheik.123@gmail.com",
                        phone: "9000054321"
                }
        ]
)
```

Following example will retrieve the document(s) whose first name is not "Radhika" and last nameis not "Christopher"

```
> db.empDetails.find(
        {
                $nor:[
                        40
                        {"First_Name": "Radhika"},
                        {"Last_Name": "Christopher"}
                ]
        }
).pretty()
{
        "_id" : ObjectId("5dd631f270fb13eec3963bef"),
        "First_Name" : "Fathima",
        "Last_Name" : "Sheik",
        "Age" : "24",
        "e_mail" : "Fathima_Sheik.123@gmail.com",
        "phone" : "9000054321"
```

NOT in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword following is thebasic syntax of **NOT** −

```
>db.COLLECTION_NAME.find(
        {
                $NOT: [
                        {key1: value1}, {key2:value2}
                ]
        }
).pretty()
```

Example

Following example will retrieve the document(s) whose age is not greater than 25

```
> db.empDetails.find( { "Age": { $not: { $gt: "25" } } } )
{
        "_id" : ObjectId("5dd6636870fb13eec3963bf7"),
        "First_Name" : "Fathima",
        "Last_Name" : "Sheik",
        "Age" : "24",
        "e_mail" : "Fathima_Sheik.123@gmail.com",
        "phone" : "9000054321"
}
```

MapReduce:

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.

ich 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.

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. 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. MAPPERS AND REDUCERS Key-value pairs form the basic data structure in MapReduce. Keys and values may be primitives such as integers, floating point values, strings, and raw bytes, or they may be arbitrarily complex structures (lists, tuples, associative arrays, etc.). Programmers typically need to define their own custom data types, although a number of libraries such as Protocol Buffers,5 Thrift,6 and Avro7 simplify the task. Part of the design of MapReduce algorithms involves imposing the key-value structure on arbitrary datasets. For a collection of web pages, keys may be URLs and values may be the actual HTML content. For a graph, keys may represent node ids and values may contain the adjacency lists of those nodes (see Chapter 5 for more details). In some algorithms, input keys are not particularly

meaningful and are simply ignored during processing, while in other cases input keys are used to uniquely identify a datum (such as a record id). In Chapter 3, we discuss the role of complex keys and values in the design of various algorithms. In MapReduce, the programmer defines a mapper and a reducer with the following signatures: map: (k1, v1) → [(k2, v2)] reduce: (k2, [v2]) → [(k3, v3)] The convention [. . .] is used throughout this book to denote a list. The input to a MapReduce job starts as data stored on the underlying distributed file system (see Section 2.5). The mapper is applied to every input key-value pair (split across an arbitrary number of files) to generate an arbitrary number of intermediate key-value pairs. The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs.8 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 MapReduce job. Figure 2.2 illustrates this two-stage processing structure. A simple word count algorithm in MapReduce is shown in Figure 2.3. This algorithm counts the number of occurrences of every word in a text collection, which may be the first step in, for example, building a unigram language model (i.e., probability

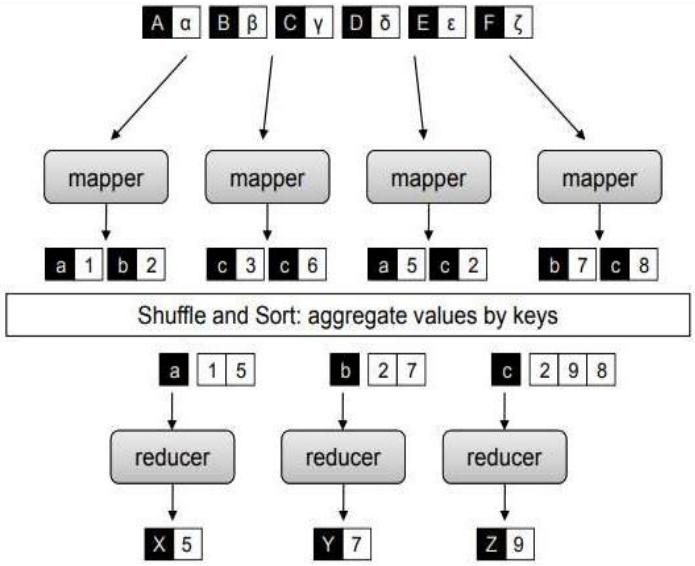


Figure 2.2: Simplified view of MapReduce. Mappers are applied to all input key-value pairs, which generate an arbitrary number of intermediate key-value pairs. Reducers are applied to all values associated with the same key. Between the map and reduce phases lies a barrier that involves a large distributed sort and group by.

distribution over words in a collection). Input key-values pairs take the form of (docid, doc) pairsstored on the distributed file system, where the former is a unique identifier for the document, andthe latter is the text of the document itself. The mapper takes an input key-value pair, tokenizes the document, and emits an intermediate key-value pair for every word: the word itself serves as the key, and the integer one serves as the value (denoting that we've seen the word once). The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Therefore, in our word count algorithm, we simply need to sum up all counts (ones) associated with each word. The reducer does exactly this, and emits final keyvalue pairs with the word as the key, and the count as the value. Final output is written to the distributed file system, one file per reducer. Words within each file will be sorted by alphabeticalorder, and each file will contain roughly the same number of words. The partitioner, which we discuss later in Section 2.4, controls the assignment of words to reducers. The output can be examined by the programmer or used as input to another MapReduce program.

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

To provide a bit more implementation detail: pseudo-code provided in this book roughly mirrors how MapReduce programs are written in Hadoop. Mappers and reducers are objects that implement the Map and Reduce methods, respectively. In Hadoop, a mapper object is initialized for each map task (associated with a particular sequence of key-value pairs called an input split) and the Map method is called on each key-value pair by the execution framework. In configuring a MapReduce job, the programmer provides a hint on the number of map tasks to run, but the execution framework (see next section) makes the final determination based on the physical layoutof the data (more details in Section 2.5 and Section 2.6). The situation is similar for the reduce phase: a reducer object is initialized for each reduce task, and the Reduce method is called once per intermediate key. In contrast with the number of map tasks, the programmer can precisely specify the number of reduce tasks. We will return to discuss the details of Hadoop job execution in Section 2.6, which is dependent on an understanding of the distributed file system (covered in Section 2.5). To reiterate: although the presentation of algorithms in this book closely mirrors theway they would be implemented in Hadoop, our focus is on algorithm design and conceptual

understanding—not actual Hadoop programming. For that, we would recommend Tom White's book [154]. What are the restrictions on mappers and reducers? Mappers and reducers can express arbitrary computations over their inputs. However, one must generally be careful about use of external resources since multiple mappers or reducers may be contending for those resources. For example, it may be unwise for a mapper to query an external SQL database, since that would introduce a scalability bottleneck on the number of map tasks that could be run in parallel (since they might all be simultaneously querying the database).10 In general, mappers can emit an arbitrary number of intermediate key-value pairs, and they need not be of the same type as the input key-value pairs. Similarly, reducers can emit an arbitrary number of final key-value pairs, and they can differ in type from the intermediate key-value pairs. Although not permitted in functional programming, mappers and reducers can have side effects. This is a powerful and useful feature: for example, preserving state across multiple inputs is central to the design of many MapReduce algorithms (see Chapter 3). Such algorithms can be understood as having side effects that only change state that is internal to the mapper or reducer. While the correctness of such algorithms may be more difficult to guarantee (since the function's behavior depends not only on the current input but on previous inputs), most potential synchronization problems are avoided since internal state is private only to individual mappers and reducers. In other cases (see Section
4.4 and Section 6.5), it may be useful for mappers or reducers to have external side effects, such as writing files to the distributed file system. Since many mappers and reducers are run in parallel, and the distributed file system is a shared global resource, special care must be taken to ensure that such operations avoid synchronization conflicts. One strategy is to write a temporary file that is renamed upon successful completion of the mapper or reducer .

In addition to the "canonical" MapReduce processing flow, other variations are also possible. MapReduce programs can contain no reducers, in which case mapper output is directly written to disk (one file per mapper). For embarrassingly parallel problems, e.g., parse a large text collection or independently analyze a large number of images, this would be a common pattern. The converse—a MapReduce program with no mappers—is not possible, although in some cases it is useful for the mapper to implement the identity function and simply pass input key-value pairs to the reducers. This has the effect of sorting and regrouping the input for reduce-side processing. Similarly, in some cases it is useful for the reducer to implement the identity function, in which case the program simply sorts and groups mapper output. Finally, running identity mappers and reducers has the effect of regrouping and resorting the input data (which is sometimes useful).

Although in the most common case, input to a MapReduce job comes from data stored on the distributed file system and output is written back to the distributed file system, any other system that satisfies the proper abstractions can serve as a data source or sink. With Google's MapReduce implementation, BigTable [34], a sparse, distributed, persistent multidimensional sorted map, is frequently used as a source of input and as a store of MapReduce output. HBase is an open-source BigTable clone and has similar capabilities. Also, Hadoop has been integrated with existing MPP (massively parallel processing) relational databases, which allows a programmer to write MapReduce jobs over database rows and dump output into a new database table. Finally, in some

cases MapReduce jobs may not consume any input at all (e.g., computing π) or may only consumea small amount of data (e.g., input parameters to many instances of processorintensive simulations running in parallel).

PARTITIONERS AND COMBINERS

We have thus far presented a simplified view of MapReduce. There are two additional elements that complete the programming model: partitioners and combiners. Partitioners are responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. In other words, the 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 MapReduce that allow for local aggregation before the shuffle and sort phase. We can motivate the need for combiners by considering the word count algorithm in Figure 2.3, 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).

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

In many cases, proper use of combiners can spell the difference between an impractical algorithmand an efficient algorithm. This topic will be discussed in Section 3.1, which focuses on various techniques for local aggregation. It suffices to say for now that a combiner can significantly reducethe amount of data that needs to be copied over the network, resulting in much faster algorithms. The complete MapReduce model is shown in Figure 2.4. Output of the mappers are processed bythe combiners, which perform local aggregation to cut down on the number of intermediate key- value pairs. The partitioner determines which reducer will be responsible for processing a particular key, and the execution framework uses this information to copy the data to the right location during the shuffle and sort phase.13 Therefore, a complete MapReduce job consists of code for the mapper, reducer, combiner, and partitioner, along with job configuration parameters. The execution framework handles everything else.

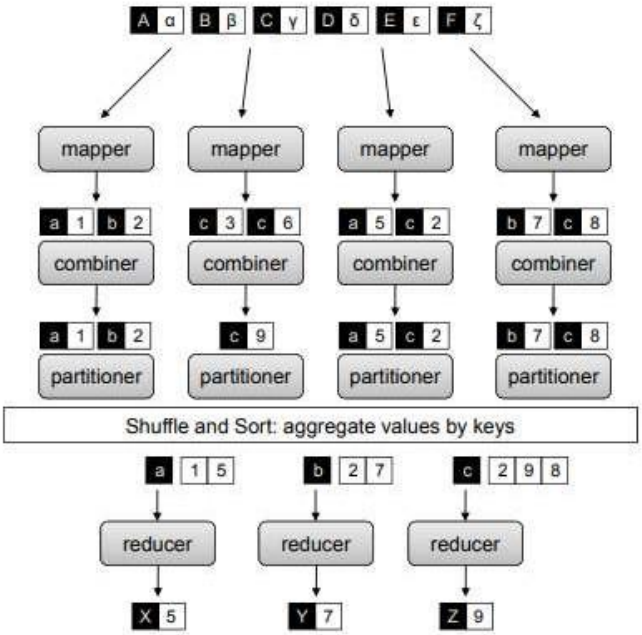


Figure 2.4: Complete view of MapReduce, illustrating combiners and partitioners in addition to mappers and reducers. Combiners can be viewed as "mini-reducers" in the map phase. Partitioners determine which reducer is responsible for a particular key.

SECONDARY SORTING

MapReduce sorts intermediate key-value pairs by the keys during the shuffle and sort phase, whichis very convenient if computations inside the reducer rely on sort order (e.g., the order inversion design pattern described in the previous section).

However, what if in addition to sorting by key, we also need to sort by value? Google's MapReduce implementation provides built-in functionality for (optional) secondary sorting, which guarantees that values arrive in sorted order.Hadoop, unfortunately, does not have this capability built in.

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

(t1, m1, r80521)

(t1, m2, r14209)

(t1, m3, r76042) ...

(t2, m1, r21823)

(t2, m2, r66508)

(t2, m3, r98347)

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

m1 → (t1, r80521)

This would bring all readings from the same sensor together in the reducer. However, since MapReduce makes no guarantees about the ordering of values associated with the same key, the sensor readings will not likely be in temporal order. The most obvious solution is to buffer all thereadings in memory and then sort by timestamp before additional processing. However, it shouldbe apparent by now that any in-memory buffering of data introduces a potential scalability bottleneck. What if we are working with a high frequency sensor or sensor readings over a long period of time? What if the sensor readings themselves are large complex objects? This approach may not scale in these cases—the reducer would run out of memory trying to buffer all values associated with the same key.

This is a common problem, since in many applications we wish to first group together data one way (e.g., by sensor id), and then sort within the groupings another way (e.g., by time). Fortunately,there is a general purpose solution, which we call the "value-to-key conversion" design pattern. The basic idea is to move part of the value into the intermediate key to form a composite key, andlet the MapReduce execution framework handle the sorting. In the above example, instead of emitting the sensor id as the key, we would emit the sensor id and the timestamp as a composite key: (m1, t1) → (r80521)

The sensor reading itself now occupies the value. We must define the intermediate key sort orderto first sort by the sensor id (the left element in the pair) and then by the timestamp (the right element in the pair). We must also implement a custom partitioner so that all pairs associated withthe same sensor are shuffled to the same reducer. Properly orchestrated, the key-value pairs will be presented to the reducer in the correct sorted order: (m1, t1) → [(r80521)] (m1, t2) → [(r21823)](m1, t3) → [(r146925)] . . .

However, note that sensor readings are now split across multiple keys. The reducer will need to preserve state and keep track of when readings associated with the current sensor end and the nextsensor begin.9 The basic tradeoff between the two approaches discussed above (buffer and inmemory sort vs. value-to-key conversion) is where sorting is performed. One can explicitly implement secondary sorting in the reducer, which is likely to be faster but suffers from a scalability bottleneck.10 With value-to-key conversion, sorting is offloaded to the MapReduce execution framework. Note that this approach can be arbitrarily extended to tertiary, quaternary, etc. sorting. This pattern results in many more keys for the framework to sort, but distributed sorting is a task that the MapReduce runtime excels at since it lies at the heart of the programmingmodel.

INDEX COMPRESSION

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

Let us consider the canonical case where each posting consists of a document id and the term frequency. A na¨ıve implementation might represent the first as a 32-bit integer9 and the second as a 16-bit integer. Thus, a postings list might be encoded as follows: [(5, 2),(7, 3),(12, 1),(49, 1),(51, 2), . . .]

where each posting is represented by a pair in parentheses. Note that all brackets, parentheses, and commas are only included to enhance readability; in reality the postings would be represented as a long stream of integers. This na¨ıve implementation would require six bytes per posting. Using this scheme, the entire inverted index would be about as large as the collection itself. Fortunately, we can do significantly better. The first trick is to encode differences between document ids as opposed to the document ids themselves. Since the postings are sorted by document ids, the differences (called d-gaps) must be positive integers greater than zero. The above postings list, represented with d-gaps, would be: [(5, 2),(2, 3),(5, 1),(37, 1),(2, 2)

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

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

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

POSTINGS COMPRESSION

Having completed our slight detour into integer compression techniques, we can now return to the scalable inverted indexing algorithm shown in Figure 4.4 and discuss how postings lists can be properly compressed. As we can see from the previous section, there is a wide range of choices that represent different tradeoffs between compression ratio and decoding speed. Actual performance also depends on characteristics of the collection, which, among other factors, determine the distribution of d-gaps. B¨uttcher et al. [30] recently compared the performance of various compression techniques on coding document ids. In terms of the amount of compression that can be obtained (measured in bits per docid), Golomb and Rice codes performed the best, followed by γ codes, Simple-9, varInt, and group varInt (the least space efficient). In terms of rawdecoding speed, the order was almost the reverse: group varInt was the fastest, followed by varInt.14 Simple-9 was substantially slower, and the bit-aligned codes were even slower than that. Within the bit-aligned codes, Rice codes were the fastest, followed by γ, with Golomb codes beingthe slowest (about ten times slower than group varInt).

Let us discuss what modifications are necessary to our inverted indexing algorithm if we were to adopt Golomb compression for d-gaps and represent term frequencies with γ codes. Note that thisrepresents a space-efficient encoding, at the cost of slower decoding compared to alternatives. Whether or not this is actually a worthwhile tradeoff in practice is not important here: use of Golomb codes serves a pedagogical purpose, to illustrate how one might set compression parameters.

Coding term frequencies with γ codes is easy since they are parameterless. Compressing d-gaps with Golomb codes, however, is a bit tricky, since two parameters are required: the size of the document collection and the number of postings for a particular postings list (i.e., the document frequency, or df). The first is easy to obtain and can be passed into the reducer as a constant. The df of a term, however, is not known until all the postings have been processed—and unfortunately,

the parameter must be known before any posting is coded. At first glance, this seems like a chicken-and-egg problem. A two-pass solution that involves first buffering the postings (in memory) would suffer from the memory bottleneck we've been trying to avoid in the first place.

To get around this problem, we need to somehow inform the reducer of a term's df before any of its postings arrive. This can be solved with the order inversion design pattern introduced in Section 3.3 to compute relative frequencies. The solution is to have the mapper emit special keys of the form ht, ∗i to communicate partial document frequencies. That is, inside the mapper, in addition to emitting intermediate key-value pairs of the following form:

(tuple ht, docidi,tf f)

we also emit special intermediate key-value pairs like this:

(tuple ht, ∗i, df e)

to keep track of document frequencies associated with each term. In practice, we can accomplishthis by applying the in-mapper combining design pattern (see Section 3.1). The mapper holds an in-memory associative array that keeps track of how many documents a term has been observed in (i.e., the local document frequency of the term for the subset of documents processed by the mapper). Once the mapper has processed all input records, special keys of the form ht, ∗i are emitted with the partial df as the value.

To ensure that these special keys arrive first, we define the sort order of the tuple so that the special symbol ∗ precedes all documents (part of the order inversion design pattern). Thus, for each term, the reducer will first encounter the ht, ∗i key, associated with a list of values representing partial df values originating from each mapper. Summing all these partial contributions will yield the term's df, which can then be used to set the Golomb compression parameter b. This allows the postings to be incrementally compressed as they are encountered in the reducer—memory bottlenecks are eliminated since we do not need to buffer postings in memory.

Once again, the order inversion design pattern comes to the rescue. Recall that the pattern is usefulwhen a reducer needs to access the result of a computation (e.g., an aggregate statistic) before it encounters the data necessary to produce that computation. For computing relative frequencies, that bit of information was the marginal. In this case, it's the document frequency.

PARALLEL BREADTH-FIRST SEARCH

One of the most common and well-studied problems in graph theory is the single-source shortest path problem, where the task is to find shortest paths from a source node to all other nodes in the graph (or alternatively, edges can be associated with costs or weights, in which case the task is to compute lowest-cost or lowest-weight paths). Such problems are a staple in undergraduate

algorithm courses, where students are taught the solution using Dijkstra's algorithm. However, this famous algorithm assumes sequential processing—how would we solve this problem in parallel, and more specifically, with MapReduce?

Dijkstra(G, w, s)
2: d[s] ← 0
3: for all vertex v ∈ V do
4: d[v] ← ∞
5: Q ← {V }
6: while Q 6= Ø do
7: u ← ExtractMin(Q)
8: for all vertex v ∈ u.AdjacencyList do
9: if d[v] > d[u] + w(u, v) then
10: d[v] ← d[u] + w(u, v)

Figure 5.2: Pseudo-code for Dijkstra's algorithm, which is based on maintaining a global priority queue of nodes with priorities equal to their distances from the source node. At each iteration, the algorithm expands the node with the shortest distance and updates distances to all reachable nodes.As a refresher and also to serve as a point of comparison, Dijkstra's algorithm is shown in Figure 5.2, adapted from Cormen, Leiserson, and Rivest's classic algorithms textbook [41] (often simply known as CLR). The input to the algorithm is a directed, connected graph G = (V, E) represented with adjacency lists, w containing edge distances such that w(u, v) ≥ 0, and the source node s. The algorithm begins by first setting distances to all vertices d[v], v ∈ V to ∞, except for the source node, whose distance to itself is zero. The algorithm maintains Q, a global priority queue of vertices with priorities equal to their distance values d

Dijkstra's algorithm operates by iteratively selecting the node with the lowest current distance from the priority queue (initially, this is the source node). At each iteration, the algorithm "expands" that node by traversing the adjacency list of the selected node to see if any of those nodes can be reached with a path of a shorter distance. The algorithm terminates when the priority queue Q is empty, or equivalently, when all nodes have been considered. Note that the algorithm as presented in Figure 5.2 only computes the shortest distances. The actual paths can be recovered by storing "backpointers" for every node indicating a fragment of the shortest path.

A sample trace of the algorithm running on a simple graph is shown in Figure 5.3 (example also adapted from CLR). We start out in (a) with n1 having a distance of zero (since it's the source) and all other nodes having a distance of ∞. In the first iteration (a), n1 is selected as the node to expand (indicated by the thicker border). After the expansion, we see in (b) that n2 and n3 can be reached at a distance of 10 and 5, respectively. Also, we see in (b) that n3 is the next node selected for expansion. Nodes we have already considered for expansion are shown in black. Expanding n3, we see in (c) that the distance to n2 has decreased because we've found a shorter path. The nodes that will be expanded next, in order, are n5, n2, and n4. The algorithm terminates with the end state shown in (f), where we've discovered the shortest distance to all nodes.

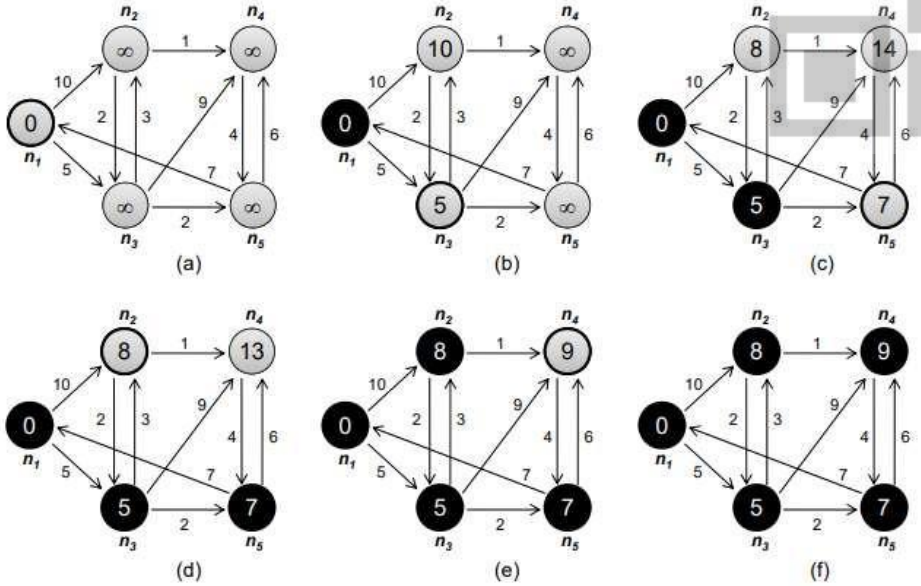


Figure 5.3: Example of Dijkstra's algorithm applied to a simple graph with five nodes, with n_1 as the source and edge distances as indicated. Parts (a)–(e) show the running of the algorithm at each iteration, with the current distance inside the node. Nodes with thicker borders are those being expanded; nodes that have already been expanded are shown in black.

The key to Dijkstra's algorithm is the priority queue that maintains a globallysorted list of nodes by current distance. This is not possible in MapReduce, as the programming model does not provide a mechanism for exchanging global data. Instead, we adopt a brute force approach known as parallel breadth-first search. First, as a simplification let us assume that all edges have unit distance (modeling, for example, hyperlinks on the web). This makes the algorithm easier to understand, but we'll relax this restriction later.

The intuition behind the algorithm is this: the distance of all nodes connected directly to the sourcenode is one; the distance of all nodes directly connected to those is two; and so on. Imagine waterrippling away from a rock dropped into a pond— that's a good image of how parallel breadth-first search works. However, what if there are multiple paths to the same node? Suppose we wish to compute the shortest distance to node n. The shortest path must go through one of the nodes in Mthat contains an outgoing edge to n: we need to examine all m ∈ M to find ms, the node with the shortest distance. The shortest distance to n is the distance to ms plus one.

Pseudo-code for the implementation of the parallel breadth-first search algorithm is provided in Figure 5.4. As with Dijkstra's algorithm, we assume a connected, directed graph represented as adjacency lists. Distance to each node is directly stored alongside the adjacency list of that node, and initialized to ∞ for all nodes except for the source node. In the pseudo-code, we use n to denotethe node id (an integer) and N to denote the node's corresponding data structure (adjacency list and current distance). The algorithm works by mapping over all nodes and emitting a key-value pair for each neighbor on the node's adjacency list. The key contains the node id of the neighbor,and the value is the current distance to the node plus one. This says: if we can reach node n with adistance d, then we must be able to reach all the nodes that are connected to n with distance d + 1.

After shuffle and sort, reducers will receive keys corresponding to the destination node ids and distances corresponding to all paths leading to that node. The reducer will select the shortest of these distances and then update the distance in the node data structure.

h iteration corresponds to a MapReduce job. The first time we run the algorithm, we "discover" all nodes that are connected to the source. The second iteration, we discover all nodes connected to those, and so on. Each iteration of the algorithm expands the "search frontier" by one hop, and, eventually, all nodes will be discovered with their shortest distances (assuming a fully-connected graph). Before we discuss termination of the algorithm, there is one more detail required to make the parallel breadth-first search algorithm work. We need to "pass along" the graph structure from one iteration to the next. This is accomplished by emitting the node data structure itself, with the node id as a key (Figure 5.4, line 4 in the mapper). In the reducer, we must distinguish the node data structure from distance values (Figure 5.4, lines 5–6 in the reducer), and update the minimum distance in the node data structure before emitting it as the final value. The final output is now ready to serve as input to the next iteration.

So how many iterations are necessary to compute the shortest distance to all nodes? The answer is the diameter of the graph, or the greatest distance between any pair of nodes. This number is surprisingly small for many real-world problems: the saying "six degrees of separation" suggests that everyone on the planet is connected to everyone else by at most six steps (the people a person knows are one step away, people that they know are two steps away, etc.). If this is indeed true, then parallel breadthfirst search on the global social network would take at most six MapReduce iterations.

class Mapper
2: method Map(nid n, node N)
3: d ← N.Distance
4: Emit(nid n, N) . Pass along graph structure
5: for all nodeid m ∈ N.AdjacencyList do
6: Emit(nid m, d + 1) .

Emit distances to reachable nodes
1: class Reducer
2: method Reduce(nid m, [d1, d2, . . .])
3: dmin ← ∞
4: M ← ∅
5: for all d ∈ counts [d1, d2, . . .] do
6: if IsNode(d) then
7: M ← d . Recover graph structure
8: else if d < dmin then . Look for shorter distance
9: dmin ← d
10: M.Distance ← dmin . Update shortest distance
11: Emit(nid m, node M)

Figure 5.4: Pseudo-code for parallel breath-first search in MapReduce: the mappers emit distances to reachable nodes, while the reducers select the minimum of those distances for each destination node. Each iteration (one MapReduce job) of the algorithm expands the "search frontier" by one hop.

For more serious academic studies of "small world" phenomena in networks, we refer the reader to a number of publications [61, 62, 152, 2]. In practical terms, we iterate the algorithm until there are no more node distances that are ∞. Since the graph is connected, all nodes are reachable, and since all edge distances are one, all discovered nodes are guaranteed to have the shortest distances (i.e., there is not a shorter path that goes through a node that hasn't been discovered).

The actual checking of the termination condition must occur outside of MapReduce. Typically, execution of an iterative MapReduce algorithm requires a nonMapReduce "driver" program, which submits a MapReduce job to iterate the algorithm, checks to see if a termination condition has been met, and if not, repeats. Hadoop provides a lightweight API for constructs called "counters", which, as the name suggests, can be used for counting events that occur during execution, e.g., number of corrupt records, number of times a certain condition is met, or anything that the programmer desires. Counters can be defined to count the number of nodes that have distances of ∞: at the end of the job, the driver program can access the final counter value and check to see if another iteration is necessary.

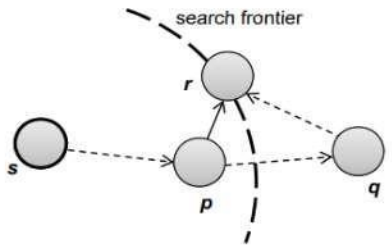


Figure 5.5: In the single source shortest path problem with arbitrary edge distances, the shortest path from source s to node r may go outside the current search frontier, in which case we will not find the shortest distance to r until the search frontier expands to cover q.

Finally, as with Dijkstra's algorithm in the form presented earlier, the parallel breadth-first search algorithm only finds the shortest distances, not the actual shortest paths. However, the path can be straightforwardly recovered. Storing "backpointers" at each node, as with Dijkstra's algorithm, will work, but may not be efficient since the graph needs to be traversed again to reconstruct the path segments. A simpler approach is to emit paths along with distances in the mapper, so that each node will have its shortest path easily accessible at all times. The additional space requirements for shuffling these data from mappers to reducers are relatively modest, since for the most part paths (i.e., sequence of node ids) are relatively short.

Up until now, we have been assuming that all edges are unit distance. Let us relax that restriction and see what changes are required in the parallel breadth-first search algorithm. The adjacency lists, which were previously lists of node ids, must now encode the edge distances as well. In line

6 of the mapper code in Figure 5.4, instead of emitting d + 1 as the value, we must now emit d + w where w is the edge distance. No other changes to the algorithm are required, but the termination behavior is very different. To illustrate, consider the graph fragment in Figure 5.5, where s is the source node, and in this iteration, we just "discovered" node r for the very first time. Assume for the sake of argument that we've already discovered the shortest distance to node p, and that the shortest distance to r so far goes through p. This, however, does not guarantee that we've discovered the shortest distance to r, since there may exist a path going through q that we haven'tencountered yet (because it lies outside the search frontier).6 However, as the search frontier expands, we'll eventually cover q and all other nodes along the path from p to q to r—which means that with sufficient iterations, we will discover the shortest distance to r. But how do we know thatwe've found the shortest distance to p? Well, if the shortest path to p lies within the search frontier, we would have already discovered it. And if it doesn't, the above argument applies. Similarly, wecan repeat the same argument for all nodes on the path from s to p. The conclusion is that, with sufficient iterations, we'll eventually discover all the shortest distances.

So exactly how many iterations does "eventually" mean? In the worst case, we might need as many iterations as there are nodes in the graph minus one. In fact, it is not difficult to construct graphs that will elicit this worse-case behavior: Figure 5.6 provides an example, with n1 as the source. The parallel breadth-first search algorithm would not discover that the shortest path from n1 to n6goes through n3, n4, and n5 until the fifth iteration. Three more iterations are necessary to cover the rest of the graph. Fortunately, for most real-world graphs, such extreme cases are rare, and the number of iterations necessary to discover all shortest distances is quite close to the diameter of the graph, as in the unit edge distance case.

In practical terms, how do we know when to stop iterating in the case of arbitrary edge distances?The algorithm can terminate when shortest distances at every node no longer change. Once again,we can use counters to keep track of such events. Every time we encounter a shorter distance in the reducer, we increment a counter. At the end of each MapReduce iteration, the driver program reads the counter value and determines if another iteration is necessary.

Compared to Dijkstra's algorithm on a single processor, parallel breadth-first search in MapReduce can be characterized as a brute force approach that "wastes" a lot of time performingcomputations whose results are discarded. At each iteration, the algorithm attempts to recompute distances to all nodes, but in reality only useful work is done along the search frontier: inside the search frontier, the algorithm is simply repeating previous computations.7 Outside the search frontier, the algorithm hasn't discovered any paths to nodes there yet, so no meaningful work is done. Dijkstra's algorithm, on the other hand, is far more efficient. Every time a node is explored, we're guaranteed to have already found the shortest path to it. However, this is made possible by maintaining a global data structure (a priority queue) that holds nodes sorted by distance—this is not possible in MapReduce because the programming model does not provide support for global data that is mutable and accessible by the mappers and reducers. These inefficiencies represent thecost of parallelization.

The parallel breadth-first search algorithm is instructive in that it represents the prototypical structure of a large class of graph algorithms in MapReduce. They share in the following characteristics:

The graph structure is represented with adjacency lists, which is part of some larger node data structure that may contain additional information (variables to store intermediate output, features of the nodes). In many cases, features are attached to edges as well (e.g., edge weights).

The graph structure is represented with adjacency lists, which is part of some larger node data structure that may contain additional information (variables to store intermediate output, features of the nodes). In many cases, features are attached to edges as well (e.g., edge weights).

In addition to computations, the graph itself is also passed from the mapper to the reducer.In the reducer, the data structure corresponding to each node is updated and written back to disk.

Graph algorithms in MapReduce are generally iterative, where the output of the previous iteration serves as input to the next iteration. The process is controlled by a non-MapReduce driver program that checks for termination.

For parallel breadth-first search, the mapper computation is the current distance plus edgedistance (emitting distances to neighbors), while the reducer computation is the Min function (selecting the shortest path). As we will see in the next section, the MapReduce algorithm for PageRank works in much the same way

Chapter 7

Introduction to HIVE AND PIG

The term 'Big Data' is used for collections of large datasets that include huge volume, high velocity, and a variety of data that is increasing day by day. Using traditional data management systems, it is difficult to process Big Data. Therefore, the Apache Software Foundation introduced a framework called Hadoop to solve Big Data management and processing challenges.

Hadoop

Hadoop is an open-source framework to store and process Big Data in a distributed environment. It contains two modules, one is MapReduce and another is Hadoop Distributed File System (HDFS).

- **MapReduce:** It is a parallel programming model for processing large amounts of structured, semi-structured, and unstructured data on large clusters of commodity hardware.
- **HDFS:** Hadoop Distributed File System is a part of Hadoop framework, used to store and process the datasets. It provides a fault-tolerant file system to run on commodity hardware.

The Hadoop ecosystem contains different sub-projects (tools) such as Sqoop, Pig, and Hive that are used to help Hadoop modules.

- **Sqoop:** It is used to import and export data to and from between HDFS and RDBMS.
- **Pig:** It is a procedural language platform used to develop a script for MapReduce operations.
- **Hive:** It is a platform used to develop SQL type scripts to do MapReduce operations.

Note: There are various ways to execute MapReduce operations:

- The traditional approach using Java MapReduce program for structured, semi-structured, and unstructured data.
- The scripting approach for MapReduce to process structured and semi structured data using Pig.
- The Hive Query Language (HiveQL or HQL) for MapReduce to process structured data using Hive.

What is Hive

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

Hive is not

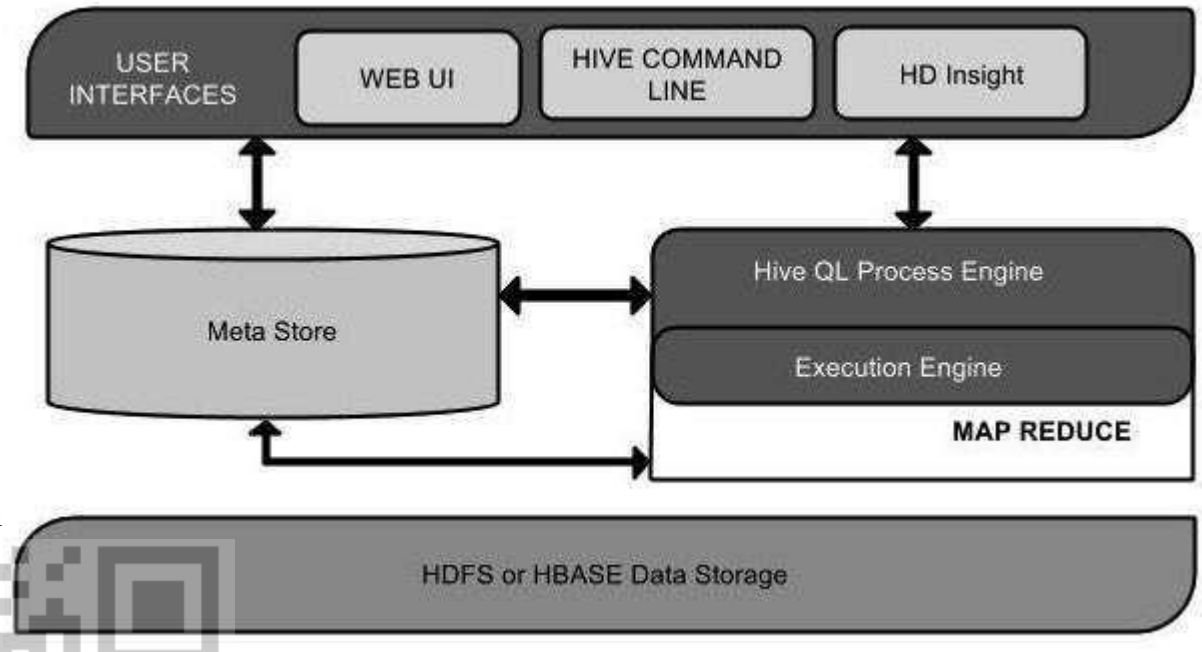
- A relational database
- A design for OnLine Transaction Processing (OLTP)
- A language for real-time queries and row-level updates

Features of Hive

- It stores schema in a database and processed data into HDFS.
- It is designed for OLAP.
- It provides SQL type language for querying called HiveQL or HQL.
- It is familiar, fast, scalable, and extensible.

Architecture of Hive

The following component diagram depicts the architecture of Hive:

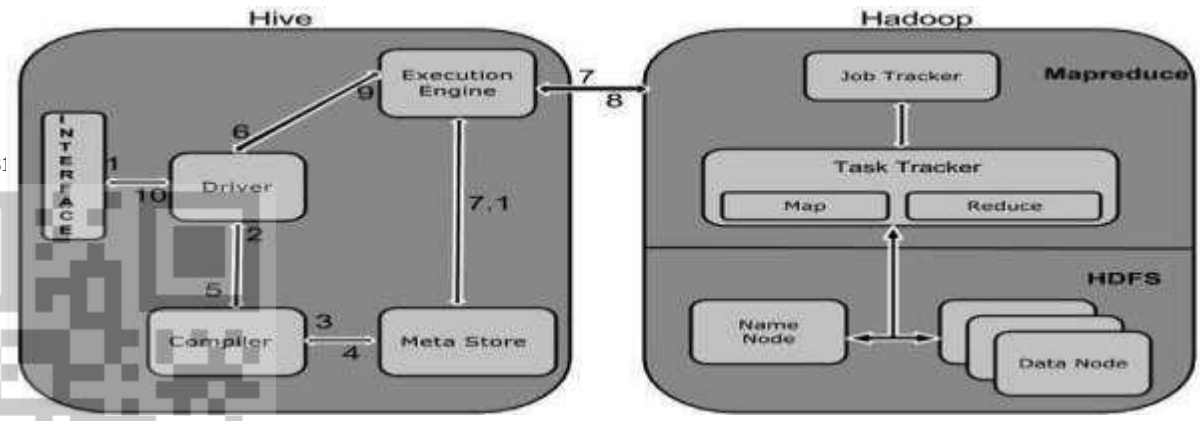


This component diagram contains different units. The following table describes each unit:

Unit Name	Operation
User Interface	Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server).
Meta Store	Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.
HiveQL Process Engine	HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approachfor MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it.
Execution Engine	The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce.
HDFS or HBASE	Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Working of Hive

The following diagram depicts the workflow between Hive and Hadoop.



The following table defines how Hive interacts with Hadoop framework:

Step No.	Operation
1	**Execute Query** The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute.
2	**Get Plan** The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query.
3	**Get Metadata** The compiler sends metadata request to Metastore (any database).
4 2021/2022 2021/2022 2021/2022	**Send Metadata** Metastore sends metadata as a response to the compiler.
5	**Send Plan** The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete.
6	**Execute Plan** The driver sends the execute plan to the execution engine.
30103162101496 30103162101496 30103162101496	
7	**Execute Job** Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, whichis in Data node. Here, the query executes MapReduce job.
7.1	**Metadata Ops**

	Meanwhile in execution, the execution engine can execute metadata operations with Metastore.
8	**Fetch Result** The execution engine receives the results from Data nodes.
9	**Send Results** The execution engine sends those resultant values to the driver.
10	**Send Results** The driver sends the results to Hive Interfaces.

File Formats in Hive

- File Format specifies how records are encoded in files
- Record Format implies how a stream of bytes for a given record are encoded
- The default file format is **TEXTFILE** – each record is a line in the file
- Hive uses different control characters as delimeters in textfiles
  - ^A (octal 001) , ^B(octal 002), ^C(octal 003), \n
- The term field is used when overriding the default delimiter
  - **FIELDS TERMINATED BY '\001'**
- Supports text files – csv, tsv
- TextFile can contain JSON or XML documents.

ommonly used File Formats –

1. **TextFile format**
   - Suitable for sharing data with other tools
   - Can be viewed/edited manually
2. **SequenceFile**
   - Flat files that stores binary key ,value pair
   - SequenceFile offers a Reader ,Writer, and Sorter classes for reading ,writing, and sortingrespectively
   - Supports – Uncompressed, Record compressed (only value is compressed) and Block compressed (both key,value compressed) formats
3. **RCFile**
   - RCFile stores columns of a table in a record columnar way
4. **ORC**
5. **AVRO**

Hive Commands

Hive supports Data definition Language(DDL), Data Manipulation Language(DML) and Userdefined functions.

Hive DDL Commands

create database

drop database

create table

drop table

alter table

create index

create view

Hive DML Commands

Select

Where

Group By

Order By

Load Data

Join:

- Inner Join
- Left Outer Join
- Right Outer Join
- Full Outer Join

Hive DDL Commands

Create Database

Statement

A database in Hive is a namespace or a collection of tables.

1. hive> CREATE SCHEMA userdb;
2. hive> SHOW DATABASES;

Drop database

1. ive> DROP DATABASE IF EXISTS userdb;Creating

Hive Tables

Create a table called Sonoo with two columns, the first being an integer and the other a string.

1. hive> CREATE TABLE Sonoo(foo INT, bar STRING);

Create a table called HIVE_TABLE with two columns and a partition column called ds. The partition column is a virtual column. It is not part of the data itself but is derived from the partitionthat a particular dataset is loaded into.By default, tables are assumed to be of text input format andthe delimiters are assumed to be ^A(ctrl-a).

1. hive> CREATE TABLE HIVE_TABLE (foo INT, bar STRING) PARTITIONED BY (ds STRING);

Browse the table

1. hive> Show tables;

Altering and Dropping Tables

1. hive> ALTER TABLE Sonoo RENAME TO Kafka;
2. hive> ALTER TABLE Kafka ADD COLUMNS (col INT);
3. hive> ALTER TABLE HIVE_TABLE ADD COLUMNS (col1 INT COMMENT 'a comment');
4. hive> ALTER TABLE HIVE_TABLE REPLACE COLUMNS (col2 INT, weight STRING, bazINT COMMENT 'baz replaces new_col1');

Hive DML Commands

To understand the Hive DML commands, let's see the employee and employee_department table first.

Employee			Employee Department	
EMP ID	Emp Name	Address	Emp ID	Department
1	Rose	US	1	IT
2	Fred	US	2	IT
3	Jess	In	3	Eng
4	Frey	Th	4	Admin

LOAD DATA

1. hive> LOAD DATA LOCAL INPATH './usr/Desktop/kv1.txt' OVERWRITE INTO TABLE Employee;

SELECTS and FILTERS

1. hive> SELECT E.EMP_ID FROM Employee E WHERE E.Address='US';GROUP BY

1. hive> hive> SELECT E.EMP_ID FROM Employee E GROUP BY E.Addresss;Adding

a Partition

We can add partitions to a table by altering the table. Let us assume we have a tablecalled **employee** with fields such as Id, Name, Salary, Designation, Dept, and yoj.

Syntax:

```
ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION partition_spec[LOCATION
'location1'] partition_spec [LOCATION 'location2'] ...;

partition_spec:
: (p_column = p_col_value, p_column = p_col_value, ...)
```

2021/2022 2021/2022 2021/2022

The following query is used to add a partition to the employee table.

```
hive> ALTER TABLE employee
> ADD PARTITION (year=’2012’)
> location '/2012/part2012';
```

Renaming a Partition

The syntax of this command is as follows.

```
ALTER TABLE table_name PARTITION partition_spec RENAME TO PARTITION
partition_spec;
```

30103162101496 30103162101496 30103162101496

The following query is used to rename a partition:

```
hive> ALTER TABLE employee PARTITION (year=’1203’)
  > RENAME TO PARTITION (Yoj=’1203’);
```

Dropping a Partition

The following syntax is used to drop a partition:

```
ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec, PARTITION
partition_spec,...;
```

The following query is used to drop a partition:

```
hive> ALTER TABLE employee DROP [IF EXISTS]
  > PARTITION (year='1203');
```

Hive Query Language

The Hive Query Language (HiveQL) is a query language for Hive to process and analyze structured data in a Metastore. This chapter explains how to use the SELECT statement with WHERE clause.

SELECT statement is used to retrieve the data from a table. WHERE clause works similar to a condition. It filters the data using the condition and gives you a finite result. The built-in operators and functions generate an expression, which fulfils the condition.

Syntax

Given below is the syntax of the SELECT query:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY col_list]]
[LIMIT number];
```

Example

Let us take an example for SELECT…WHERE clause. Assume we have the employee table as given below, with fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details who earn a salary of more than Rs 30000.

```
+------+--------------+-------------+-------------------+--------+
| ID   | Name         | Salary      | Designation       | Dept   |
+------+--------------+-------------+-------------------+--------+
|1201  | Gopal        | 45000       | Technical manager | TP     |
|1202  | Manisha      | 45000       | Proofreader       | PR     |
|1203  | Masthanvali  | 40000       | Technical writer  | TP     |
|1204  | Krian        | 40000       | Hr Admin          | HR     |
|1205  | Kranthi      | 30000       | Op Admin          | Admin  |
+------+--------------+-------------+-------------------+--------+
```

The following query retrieves the employee details using the above scenario:

```
hive> SELECT * FROM employee WHERE salary>30000;
```

On successful execution of the query, you get to see the following response:

```
+------+--------------+-------------+-------------------+--------+
| ID   | Name         | Salary      | Designation       | Dept   |
+------+--------------+-------------+-------------------+--------+
|1201  | Gopal        | 45000       | Technical manager | TP     |
|1202  | Manisha      | 45000       | Proofreader       | PR     |
|1203  | Masthanvali  | 40000       | Technical writer  | TP     |
|1204  | Krian        | 40000       | Hr Admin          | HR     |
+------+--------------+-------------+-------------------+--------+
```

JDBC Program

The JDBC program to apply where clause for the given example is as follows.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet; import
java.sql.Statement; import
java.sql.DriverManager;

public class HiveQLWhere {
   private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";public

   static void main(String[] args) throws SQLException {

      // Register driver and create driver instance
      Class.forName(driverName);

      // get connection
      Connection con  =  DriverManager.getConnection("jdbc:hive://localhost:10000/userdb",  "",
"");

      // create statement
      Statement stmt = con.createStatement();

      // execute statement
      Resultset res = stmt.executeQuery("SELECT * FROM employee WHERE salary>30000;");

      System.out.println("Result:");
      System.out.println(" ID \t Name \t Salary \t Designation \t Dept ");

      while (res.next()) {
         System.out.println(res.getInt(1) + " " + res.getString(2) + " " + res.getDouble(3) + " " +
res.getString(4) + " " + res.getString(5));
      }
      con.close();
   }
}
```

Save the program in a file named HiveQLWhere.java. Use the following commands to compileand execute this program.

```
$ javac HiveQLWhere.java
$ java HiveQLWhere
```

Output:

```
ID      Name          Salary      Designation          Dept
1201     Gopal         45000       Technical manager      TP
1202     Manisha        45000       Proofreader          PR
1203     Masthanvali     40000       Technical writer       TP
1204     Krian          40000       Hr Admin             HR
```

The ORDER BY clause is used to retrieve the details based on one column and sort the result setby ascending or descending order.

Syntax

Given below is the syntax of the ORDER BY clause:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[ORDER BY col_list]] [LIMIT
number];
```

Example

Let us take an example for SELECT...ORDER BY clause. Assume employee table as given below, with the fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details in order by using Department name.

```
+------+--------------+-------------+--------------------+--------+
| ID   | Name         | Salary      | Designation        | Dept   |
+------+--------------+-------------+--------------------+--------+
|1201  | Gopal        | 45000       | Technical manager  | TP     |
|1202  | Manisha      | 45000       | Proofreader        | PR     |
|1203  | Masthanvali  | 40000       | Technical writer   | TP     |
|1204  | Krian        | 40000       | Hr Admin           | HR     |
|1205  | Kranthi      | 30000       | Op Admin           | Admin  |
+------+--------------+-------------+--------------------+--------+
```

The following query retrieves the employee details using the above scenario:

```
hive> SELECT Id, Name, Dept FROM employee ORDER BY DEPT;
```

On successful execution of the query, you get to see the following response:

```
+------+---------------+-------------+--------------------+--------+
| ID   | Name          | Salary      | Designation        | Dept   |
+------+---------------+-------------+--------------------+--------+
|1205  | Kranthi       | 30000       | Op Admin           | Admin  |
|1204  | Krian         | 40000       | Hr Admin           | HR     |
|1202  | Manisha       | 45000       | Proofreader        | PR     |
|1201  | Gopal         | 45000       | Technical manager  | TP     |
|1203  | Masthanvali   | 40000       | Technical writer   | TP     |
+------+---------------+-------------+--------------------+--------+
```

JDBC Program

Here is the JDBC program to apply Order By clause for the given example.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet; import
java.sql.Statement; import
java.sql.DriverManager;

public class HiveQLOrderBy {
  private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";public

  static void main(String[] args) throws SQLException {

    // Register driver and create driver instance
    Class.forName(driverName);

    // get connection
    Connection con =  DriverManager.getConnection("jdbc:hive://localhost:10000/userdb",  "",
"");

    // create statement
    Statement stmt = con.createStatement();

    // execute statement
    Resultset res = stmt.executeQuery("SELECT * FROM employee ORDER BY DEPT;");
    System.out.println(" ID \t Name \t Salary \t Designation \t Dept ");

    while (res.next()) {
      System.out.println(res.getInt(1) + " " + res.getString(2) + " " + res.getDouble(3) + " " +
res.getString(4) + " " + res.getString(5));
    }

    con.close();
  }
}
```

Save the program in a file named HiveQLOrderBy.java. Use the following commands to compileand execute this program.

```
$ javac HiveQLOrderBy.java
$ java HiveQLOrderBy
```

Output:

```
ID      Name          Salary      Designation          Dept
1205    Kranthi        30000       Op Admin             Admin
1204    Krian          40000      Hr Admin             HR
1202    Manisha         45000       Proofreader          PR
1201    Gopal          45000      Technical manager    TP
1203    Masthanvali     40000        Technical writer       TP
1204    Krian          40000       Hr Admin             HR
```

The GROUP BY clause is used to group all the records in a result set using a particular collectioncolumn. It is used to query a group of records.

Syntax

The syntax of GROUP BY clause is as follows:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[ORDER BY col_list]]
[LIMIT number];
```

Example

Let us take an example of SELECT…GROUP BY clause. Assume employee table as given below, with Id, Name, Salary, Designation, and Dept fields. Generate a query to retrieve the number of employees in each department.

```
+------+--------------+-------------+--------------------+--------+
| ID   | Name         | Salary      | Designation        | Dept   |
+------+--------------+-------------+--------------------+--------+
|1201  | Gopal        | 45000       | Technical manager  | TP     |
|1202  | Manisha      | 45000       | Proofreader        | PR     |
|1203  | Masthanvali  | 40000       | Technical writer   | TP     |
|1204  | Krian        | 45000       | Proofreader        | PR     |
|1205  | Kranthi      | 30000       | Op Admin           | Admin  |
+------+--------------+-------------+--------------------+--------+
```

The following query retrieves the employee details using the above scenario.

```
hive> SELECT Dept,count(*) FROM employee GROUP BY DEPT;
```

On successful execution of the query, you get to see the following response:

```
+-------+-----------------+
| Dept | Count(*)      |
+-------+-----------------+
|Admin |    1          |
|PR    |   2           |
|TP    |   3           |
+-------+-----------------+
```

JDBC Program

Given below is the JDBC program to apply the Group By clause for the given example.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet; import
java.sql.Statement; import
java.sql.DriverManager;

public class HiveQLGroupBy {
  private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";public

  static void main(String[] args) throws SQLException {
2021/2022                                    2021/2022                                    2021/2022
    // Register driver and create driver instance
    Class.forName(driverName);

    // get connection
    Connection con = DriverManager.
    getConnection("jdbc:hive://localhost:10000/userdb", "", "");

    // create statement
    Statement stmt = con.createStatement();

    // execute statement
    Resultset res = stmt.executeQuery("SELECT Dept,count(*) " + "FROM employee GROUPBY
DEPT; ");
    System.out.println(" Dept \t count(*)");
30103162101496                               30103162101496                               30103162101496
    while (res.next()) {
      System.out.println(res.getString(1) + " " + res.getInt(2));
    }
    con.close();
  }
}
```

Save the program in a file named HiveQLGroupBy.java. Use the following commands to compileand execute this program.

```
$ javac HiveQLGroupBy.java
$ java HiveQLGroupBy
```

Output:

```
Dept     Count(*)
Admin       1
PR          2
TP          3
```

JOIN is a clause that is used for combining specific fields from two tables by using valuescommon to each one. It is used to combine records from two or more tables in the database.

Syntax

```
join_table:

  table_reference JOIN table_factor [join_condition]
  | table_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table_reference
  join_condition
  | table_reference LEFT SEMI JOIN table_reference join_condition
  | table_reference CROSS JOIN table_reference [join_condition]
```


Example

We will use the following two tables in this chapter. Consider the following table namedCUSTOMERS..

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```


Consider another table ORDERS as follows:

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 | 3000   |
| 100 | 2009-10-08 00:00:00 |           3 | 1500   |
| 101 | 2009-11-20 00:00:00 |           2 | 1560   |
| 103 | 2008-05-20 00:00:00 |           4 | 2060   |
+-----+---------------------+-------------+--------+
```

There are different types of joins given as follows:

- JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

JOIN

JOIN clause is used to combine and retrieve the records from multiple tables. JOIN is same as OUTER JOIN in SQL. A JOIN condition is to be raised using the primary keys and foreign keysof the tables.

The following query executes JOIN on the CUSTOMER and ORDER tables, and retrieves therecords:

```
hive> SELECT c.ID, c.NAME, c.AGE, o.AMOUNTFROM
CUSTOMERS c JOIN ORDERS o
ON (c.ID = o.CUSTOMER_ID);
```

On successful execution of the query, you get to see the following response:

```
+----+----------+-----+---------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+---------+
| 3  | kaushik  | 23  | 3000   |
| 3  | kaushik  | 23  | 1500   |
| 2  | Khilan   | 25  | 1560   |
| 4  | Chaitali | 25  | 2060   |
+----+----------+-----+---------+
```


LEFT OUTER JOIN

The HiveQL LEFT OUTER JOIN returns all the rows from the left table, even if there are no matches in the right table. This means, if the ON clause matches 0 (zero) records in the right table, the JOIN still returns a row in the result, but with NULL in each column from the right table.

A LEFT JOIN returns all the values from the left table, plus the matched values from the right table, or NULL in case of no matching JOIN predicate.

The following query demonstrates LEFT OUTER JOIN between CUSTOMER and ORDERtables:


```
hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATEFROM
CUSTOMERS c
LEFT OUTER JOIN ORDERS oON
(c.ID = o.CUSTOMER_ID);
```

On successful execution of the query, you get to see the following response:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
| 1  | Ramesh   | NULL   | NULL                |
| 2  | Khilan   | 1560   | 2009-11-20 00:00:00 |
| 3  | kaushik  | 3000   | 2009-10-08 00:00:00 |
| 3  | kaushik  | 1500   | 2009-10-08 00:00:00 |
| 4  | Chaitali | 2060   | 2008-05-20 00:00:00 |
| 5  | Hardik   | NULL   | NULL                |
| 6  | Komal    | NULL   | NULL                |
| 7  | Muffy    | NULL   | NULL                |
+----+----------+--------+---------------------+
```

RIGHT OUTER JOIN

The HiveQL RIGHT OUTER JOIN returns all the rows from the right table, even if there are nomatches in the left table. If the ON clause matches 0 (zero) records in the left table, the JOIN stillreturns a row in the result, but with NULL in each column from the left table.

A RIGHT JOIN returns all the values from the right table, plus the matched values from the lefttable, or NULL in case of no matching join predicate.

The following query demonstrates RIGHT OUTER JOIN between the CUSTOMER and ORDERtables.

notranslate"> hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS cRIGHT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER_ID);

On successful execution of the query, you get to see the following response:

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
| 3    | kaushik  | 3000   | 2009-10-08 00:00:00 |
| 3    | kaushik  | 1500   | 2009-10-08 00:00:00 |
| 2    | Khilan   | 1560   | 2009-11-20 00:00:00 |
| 4    | Chaitali | 2060   | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

FULL OUTER JOIN

The HiveQL FULL OUTER JOIN combines the records of both the left and the right outer tablesthat fulfil the JOIN condition. The joined table contains either all the records from both the tables,or fills in NULL values for missing matches on either side.

The following query demonstrates FULL OUTER JOIN between CUSTOMER and ORDERtables:

```
hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATEFROM
CUSTOMERS c
FULL OUTER JOIN ORDERS oON
(c.ID = o.CUSTOMER_ID);
```

On successful execution of the query, you get to see the following response:

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

Bucketing #

•Bucketing concept is based on (hashing function on the bucketed column) mod (by total numberof buckets). The hash_function depends on the type of the bucketing column.

•Records with the same bucketed column will always be stored in the same bucket.

•We use CLUSTERED BY clause to divide the table into buckets.

•Physically, each bucket is just a file in the table directory, and Bucket numbering is 1-based.

•Bucketing can be done along with Partitioning on Hive tables and even without partitioning.

•Bucketed tables will create almost equally distributed data file parts, unless there is skew in data.

•Bucketing is enabled by setting hive.enforce.bucketing=

true;Advantages

•Bucketed tables offer efficient sampling than by non-bucketed tables. With sampling, we can tryout queries on a fraction of data for testing and debugging purpose when the original data sets arevery huge.

•As the data files are equal sized parts, map-side joins will be faster on bucketed tables than non-bucketed tables.

•Bucketing concept also provides the flexibility to keep the records in each bucket to be sorted byone or more columns. This makes map-side joins even more efficient, since the join of each bucketbecomes an efficient merge-sort.

Bucketing Vs Partitioning

•Partitioning helps in elimination of data, if used in WHERE clause, where as bucketing helps in organizing data in each partition into multiple files, so that the same set of data is always written in same bucket.

•Bucketing helps a lot in joining of columns.

•Hive Bucket is nothing but another technique of decomposing data or decreasing the data into more manageable parts or equal parts.

Sampling

•TABLESAMPLE() gives more disordered and random records from a table as compared to LIMIT. •We can sample using the rand() function, which returns a random number.

SELECT * from users TABLESAMPLE(BUCKET 3 OUT OF 10 ON rand()) s;SELECT * from users

TABLESAMPLE(BUCKET 3 OUT OF 10 ON rand()) s;

•Here rand() refers to any random column. •The denominator in the bucket clause represents the number of buckets into which data will be hashed. •The numerator is the bucket number selected.

2021/2022 **2021/2022** **2021/2022**

SELECT * from users TABLESAMPLE(BUCKET 2 OUT OF 4 ON name) s;

•If the columns specified in the TABLESAMPLE clause match the columns in the CLUSTERED BY clause, TABLESAMPLE queries only scan the required hash partitions of the table.

SELECT * FROM buck_users TABLESAMPLE(BUCKET 1 OUT OF 2 ON id) s LIMIT 1;

Joins and Types #

Reduce-Side Join

•If datasets are large, reduce side join takes

30103162101496 **30103162101496** **30103162101496**

place.Map-Side Join

•In case one of the dataset is small, map side join takes place. •In map side join, a local job runs tocreate hash-table from content of HDFS file and sends it to every node.

SET hive.auto.convert.join =true;

Bucket Map Join

•The data must be bucketed on the keys used in the ON clause and the number of buckets for onetable must be a multiple of the number of buckets for the other table.

•When these conditions are met, Hive can join individual buckets between tables in the map phase, because it does not have to fetch the entire content of one table to match against each bucket in the other table.

•set hive.optimize.bucketmapjoin =true;

•SET hive.auto.convert.join =true;

SMBM Join

•Sort-Merge-Bucket (SMB) joins can be converted to SMB map joins as well.

•SMB joins are used wherever the tables are sorted and bucketed.

•The join boils down to just merging the already sorted tables, allowing this operation to be fasterthan an ordinary map-join.

•set hive.enforce.sortmergebucketmapjoin =false;

•set hive.auto.convert.sortmerge.join =true;

•set hive.optimize.bucketmapjoin = true;

•set hive.optimize.bucketmapjoin.sortedmerge =

true;LEFT SEMI JOIN

•A left semi-join returns records from the lefthand table if records are found in the righthand tablethat satisfy the ON predicates.

•It’s a special, optimized case of the more general inner join.

•Most SQL dialects support an IN … EXISTS construct to do the same thing.

•SELECT and WHERE clauses can’t reference columns from the righthand table.

•Right semi-joins are not supported in Hive.

•The reason semi-joins are more efficient than the more general inner join is as follows:

•For a given record in the lefthand table, Hive can stop looking for matching records in therighthand table as soon as any match is found.

•At that point, the selected columns from the lefthand table record can be projected

•A file format is a way in which information is stored or encoded in a computer file.

•In Hive it refers to how records are stored inside the file.

•InputFormat reads key-value pairs from files.

•As we are dealing with structured data, each record has to be its own structure.

•How records are encoded in a file defines a file format.

•These file formats mainly vary between data encoding, compression rate, usage of space and diskI/O.

•Hive does not verify whether the data that you are loading matches the schema for the table ornot.

•However, it verifies if the file format matches the table definition or not.

SerDe in Hive #

•The SerDe interface allows you to instruct Hive as to how a record should be processed.

•A SerDe is a combination of a Serializer and a Deserializer (hence, Ser-De).

•The Deserializer interface takes a string or binary representation of a record, and translates it intoa Java object that Hive can manipulate.

•The Serializer, however, will take a Java object that Hive has been working with, and turn it into something that Hive can write to HDFS or another supported system.

•Commonly, Deserializers are used at query time to execute SELECT statements, and Serializers are used when writing data, such as through an INSERT-SELECT statement.

CSVSerDe

•Use ROW FORMAT SERDE ‘org.apache.hadoop.hive.serde2.OpenCSVSerde’

•Define following in SERDEPROPERTIES("

separatorChar " = < value_of_separator

, " quoteChar " = < value_of_quote_character ,

" escapeChar “ = < value_of_escape_character

)

JSONSerDe

•Include hive-hcatalog-core-0.14.0.jar
•Use ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe '

RegexSerDe

•It is used in case of pattern matching.

•Use ROW FORMAT SERDE

'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'

•In SERDEPROPERTIES, define input pattern and output fields.

For Example

•input.regex = '(.)//(.)@(.*)'
•output.format.string' = ' 1 s 2 s 3 s';

USE PARTITIONING AND BUCKETING

•Partitioning a table stores data in sub-directories categorized by table location, which allows Hiveto exclude unnecessary data from queries without reading all the data every time a new query is made.

•Hive does support Dynamic Partitioning (DP) where column values are only known at EXECUTION TIME. To enable Dynamic Partitioning :

SET hive.exec.dynamic.partition =true;

•Another situation we want to protect against dynamic partition insert is that the user may accidentally specify all partitions to be dynamic partitions without specifying one static partition,while the original intention is to just overwrite the sub-partitions of one root partition.

SET hive.exec.dynamic.partition.mode =strict; To

enable bucketing:

SET hive.enforce.bucketing =true;

Optimizations in Hive #

•Use Denormalisation , Filtering and Projection as early as possible to reduce data before join.

•Join is a costly affair and requires extra map-reduce phase to accomplish query job. With De-normalisation, the data is present in the same table so there is no need for any joins, hence the selects are very fast.

•As join requires data to be shuffled across nodes, use filtering and projection as early as possibleto reduce data before join.

TUNE CONFIGURATIONS

•To increase number of mapper, reduce split size :

SET mapred.max.split.size =1000000; (~1 MB)

•Compress map/reduce output

SET mapred.compress.map.output =true;

SET mapred.output.compress =true;

•Parallel execution

•Applies to MapReduce jobs that can run in parallel, for example jobs processing different sourcetables before a join.

SET hive.exec.parallel =true;

USE ORCFILE

•Hive supports ORCfile , a new table storage format that sports fantastic speed improvementsthrough techniques like predicate push-down, compression and more.

•Using ORCFile for every HIVE table is extremely beneficial to get fast response times for your HIVE queries.

USE TEZ

•With Hadoop2 and Tez , the cost of job submission and scheduling is minimized.

•Also Tez does not restrict the job to be only Map followed by Reduce; this implies that all thequery execution can be done in a single job without having to cross job boundaries.

•Let's look at an example. Consider a click-stream event table:

```
CREATE TABLE clicks (
timestamp date,
sessionID string,
url string,
source_ip string
)
STORED as ORC
tblproperties (“ orc.compress ” = “SNAPPY”);
```

•Each record represents a click event, and we would like to find the latest URL for each sessionID

• One might consider the following approach:

```
SELECT clicks.sessionID, clicks.url FROM clicks inner join (select sessionID, max(timestamp) as max_ts from clicks group by sessionID) latest ON clicks.sessionID = latest.sessionID and clicks.timestamp = latest.max_ts;
```

•In the above query, we build a sub-query to collect the timestamp of the latest event in eachsession, and then use an inner join to filter out the rest.

•While the query is a reasonable solution —from a functional point of view— it turns out there’sa better way to re-write this query as follows:

```
SELECT ranked_clicks.sessionID , ranked_clicks.url FROM (SELECT sessionID , url, RANK() over (partition by sessionID,order by timestamp desc ) as rank FROM clicks) ranked_clicks WHERE ranked_clicks.rank =1;
```

•Here, we use Hive’s OLAP functionality (OVER and RANK) to achieve the same thing, but without a Join.

•Clearly, removing an unnecessary join will almost always result in better performance, and whenusing big data this is more important than ever.

MAKING MULTIPLE PASS OVER SAME DATA

•Hive has a special syntax for producing multiple aggregations from a single pass through a sourceof data, rather than rescanning it for each aggregation.

•This change can save considerable processing time for large input data sets.

•For example, each of the following two queries creates a table from the same source table, history:

```
INSERT OVERWRITE TABLE sales
SELECT * FROM history WHERE action=‘purchased’;
INSERT OVERWRITE TABLE credits
SELECT * FROM history WHERE action=‘returned’;
```

Optimizations in Hive

•This syntax is correct, but inefficient.

•The following rewrite achieves the same thing, but using a single pass through the source historytable:

FROM history

INSERT OVERWRITE sales SELECT * WHERE action='purchased' INSERT

OVERWRITE credits SELECT * WHERE action='returned';

What is Apache Pig

Apache Pig is a high-level data flow platform for executing MapReduce programs of Hadoop. Thelanguage used for Pig is Pig Latin.

The Pig scripts get internally converted to Map Reduce jobs and get executed on data stored in HDFS. Apart from that, Pig can also execute its job in Apache Tez or Apache Spark.

Pig can handle any type of data, i.e., structured, semi-structured or unstructured and stores the corresponding results into Hadoop Data File System. Every task which can be achieved using PIGcan also be achieved using java used in MapReduce.

Features of Apache Pig

Let's see the various uses of Pig technology.

1) Ease of programming

Writing complex java programs for map reduce is quite tough for non-programmers. Pig makes this process easy. In the Pig, the queries are converted to MapReduce internally.

2) Optimization opportunities

It is how tasks are encoded permits the system to optimize their execution automatically, allowingthe user to focus on semantics rather than efficiency.

3) Extensibility

A user-defined function is written in which the user can write their logic to execute over the dataset.

4) Flexible

It can easily handle structured as well as unstructured data.

5) In-built operators

It contains various type of operators such as sort, filter and joins.Differences

between Apache MapReduce and PIG

Advantages of Apache Pig

- Less code - The Pig consumes less line of code to perform any operation.
- Reusability - The Pig code is flexible enough to reuse again.
- Nested data types - The Pig provides a useful concept of nested data types like tuple, bag,and map.

Pig Latin

The Pig Latin is a data flow language used by Apache Pig to analyze the data in Hadoop. It is atextual language that abstracts the programming from the Java MapReduce idiom into a notation.

Pig Latin Statements

The Pig Latin statements are used to process the data. It is an operator that accepts a relation as aninput and generates another relation as an output.

- It can span multiple lines.
- Each statement must end with a semi-colon.
- It may include expression and schemas.
- By default, these statements are processed using multi-query

executionPig Latin Conventions

Convention	Description
( ) 2021/2022	The parenthesis can enclose one or more items. It can also be used to indicate the tuple data type.Example - (10, xyz, (3,6,9)) 2021/2022 2021/2022
[ ]	The straight brackets can enclose one or more items. It can also be used to indicate the map data type.Example - [INNER | OUTER]
{ }	The curly brackets enclose two or more items. It can also be used to indicate the bag data typeExample - { block | nested_block }
...	The horizontal ellipsis points indicate that you can repeat a portion of the code.Example - cat path [path ...]

Latin Data Types

Simple Data Types

Type	Description
int	It defines the signed 32-bit integer. Example - 2
long	It defines the signed 64-bit integer. Example - 2L or 2l
float	It defines 32-bit floating number. pointExample - 2.5F or 2.5f or 2.5e2f or 2.5.E2F
double	It defines 64-bit floating number. pointExample - 2.5 or 2.5 or 2.5e2f or 2.5.E2F
chararray	It defines character array in Unicode UTF-8 format. Example - javatpoint
bytearray	It defines the byte array.
boolean	It defines the boolean values. typeExample - true/false
datetime	It defines the values in order. datetimeExample - 1970-01-01T00:00:00.000+00:00
biginteger	It defines Java BigInteger values. Example - 5000000000000
bigdecimal	It defines Java BigDecimal values. Example - 52.232344535345

Pig Data Types

Apache Pig supports many data types. A list of Apache Pig Data Types with description and examples are given below.

Type	Description	Example
Int	Signed 32 bit integer	2
Long	Signed 64 bit integer	15L or 15l
Float	32 bit floating point	2.5f or 2.5F
Double	32 bit floating point	1.5 or 1.5e2 or 1.5E2
charArray	Character array	hello javatpoint
byteArray	BLOB(Byte array)	
tuple	Ordered set of fields	(12,43)
bag	Collection f tuples	{(12,43),(54,28)}
map	collection of tuples	[open#apache]

Apache Pig Execution Modes

You can run Apache Pig in two modes, namely, **Local Mode** and **HDFS mode**.

Local Mode

In this mode, all the files are installed and run from your local host and local file system. There is no need of Hadoop or HDFS. This mode is generally used for testing purpose.MapReduce

Mode

MapReduce mode is where we load or process the data that exists in the Hadoop File System (HDFS) using Apache Pig. In this mode, whenever we execute the Pig Latin statements to processthe data, a MapReduce job is invoked in the back-end to perform a particular operation on the data that exists in the HDFS.

Apache Pig Execution Mechanisms

Apache Pig scripts can be executed in three ways, namely, interactive mode, batch mode, and embedded mode.

- **Interactive Mode** (Grunt shell) – You can run Apache Pig in interactive mode using the Grunt shell. In this shell, you can enter the Pig Latin statements and get the output (usingDump operator).

- **Batch Mode** (Script) − You can run Apache Pig in Batch mode by writing the Pig Latin script in a single file with **.pig** extension.
- **Embedded Mode** (UDF) − Apache Pig provides the provision of defining our own functions (**U**ser **D**efined **F**unctions) in programming languages such as Java, and using them in our script.
- Given below in the table are some frequently used Pig Commands.

Command	Function
load	Reads data from the system
Store	Writes data to file system
foreach	Applies expressions to each record and outputs one or more records
filter	Applies predicate and removes records that do not return true
Group/cogroup	Collects records with the same key from one or more inputs
join	Joins two or more inputs based on a key

order	Sorts records based on a key
distinct	Removes duplicate records
union	Merges data sets
split	Splits data into two or more sets based on filter conditions
stream	Sends all records through a user-provided binary
dump	Writes output to stdout
limit	Limits the number of records

Complex Types

Type	Description
tuple	It defines ordered set of fields. anExample - (15,12)
bag	It defines a collection of tuples. Example - {(15,12), (12,15)}
map	It defines set of key-value pairs. aExample - [open#apache]

Pig Latin – Relational Operations

The following table describes the relational operators of Pig Latin.

Operator	Description
Loading and Storing	
LOAD	To Load the data from the file system (local/HDFS) into a relation.
STORE	To save a relation to the file system (local/HDFS).
Filtering	

FILTER	To remove unwanted rows from a relation.
DISTINCT	To remove duplicate rows from a relation.
FOREACH, GENERATE	To generate data transformations based on columns of data.
STREAM	To transform a relation using an external program.
Grouping and Joining	
JOIN	To join two or more relations.
COGROUP	To group the data in two or more relations.
GROUP	To group the data in a single relation.
CROSS	To create the cross product of two or more relations.
Sorting	
ORDER	To arrange a relation in a sorted order based on one or more fields (ascending or descending).
LIMIT	To get a limited number of tuples from a relation.
Combining and Splitting	
UNION	To combine two or more relations into a single relation.
SPLIT	To split a single relation into two or more relations.

Diagnostic Operators	
DUMP	To print the contents of a relation on the console.
DESCRIBE	To describe the schema of a relation.
EXPLAIN	To view the logical, physical, or MapReduce execution plans to computea relation.
ILLUSTRATE	To view the step-by-step execution of a series of statements.

Eval Functions

Given below is the list of **eval** functions provided by Apache Pig.

S.N.	Function & Description
1	AVG() To compute the average of the numerical values within a bag.
2	BagToString() To concatenate the elements of a bag into a string. While concatenating, we can place a delimiter between these values (optional).
3	CONCAT() To concatenate two or more expressions of same type.
4	COUNT() To get the number of elements in a bag, while counting the number of tuples in a bag.
5	COUNT_STAR()

	It is similar to the **COUNT()** function. It is used to get the number of elements in a bag.
6	DIFF() To compare two bags (fields) in a tuple.
7	IsEmpty() To check if a bag or map is empty.
8	MAX() To calculate the highest value for a column (numeric values or chararrays) in a single-column bag.
9	MIN() To get the minimum (lowest) value (numeric or chararray) for a certain column in a single-column bag.
10	PluckTuple() Using the Pig Latin **PluckTuple()** function, we can define a string Prefix and filter the columns in a relation that begin with the given prefix.
11	SIZE() To compute the number of elements based on any Pig data type.
12	SUBTRACT() To subtract two bags. It takes two bags as inputs and returns a bag which contains the tuplesof the first bag that are not in the second bag.
13	SUM() To get the total of the numeric values of a column in a single-column bag.
14	TOKENIZE() To split a string (which contains a group of words) in a single tuple and return a bag which contains the output of the split operation.

Apache Pig provides extensive support for **U**ser **D**efined **F**unctions (UDF's). Using these UDF's, we can define our own functions and use them. The UDF support is provided in six programming languages, namely, Java, Jython, Python, JavaScript, Ruby and Groovy.

For writing UDF's, complete support is provided in Java and limited support is provided in all the remaining languages. Using Java, you can write UDF's involving all parts of the processing like data load/store, column transformation, and aggregation. Since Apache Pig has been written in Java, the UDF's written using Java language work efficiently compared to other languages.

In Apache Pig, we also have a Java repository for UDF's named **Piggybank**. Using Piggybank, we can access Java UDF's written by other users, and contribute our own UDF's.

Types of UDF's in Java

While writing UDF's using Java, we can create and use the following three types of functions −

- **Filter Functions** − The filter functions are used as conditions in filter statements. These functions accept a Pig value as input and return a Boolean value.
- **Eval Functions** − The Eval functions are used in FOREACH-GENERATE statements. These functions accept a Pig value as input and return a Pig result.
- **Algebraic Functions** − The Algebraic functions act on inner bags in a FOREACHGENERATE statement. These functions are used to perform full MapReduce operations on an inner bag.

Writing UDF's using Java

To write a UDF using Java, we have to integrate the jar file **Pig-0.15.0.jar**. In this section, we discuss how to write a sample UDF using Eclipse. Before proceeding further, make sure you haveinstalled Eclipse and Maven in your system.

Follow the steps given below to write a UDF function −

- Open Eclipse and create a new project (say **myproject**).
- Convert the newly created project into a Maven project.
- Copy the following content in the pom.xml. This file contains the Maven dependencies forApache Pig and Hadoop-core jar files.

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0http://maven.apache
.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>Pig_Udf</groupId>
  <artifactId>Pig_Udf</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <build>
    <sourceDirectory>src</sourceDirectory>
```

```
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.3</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>

    <dependency>
      <groupId>org.apache.pig</groupId>
      <artifactId>pig</artifactId>
      <version>0.15.0</version>
    </dependency>

    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-core</artifactId>
      <version>0.20.2</version>
    </dependency>
```



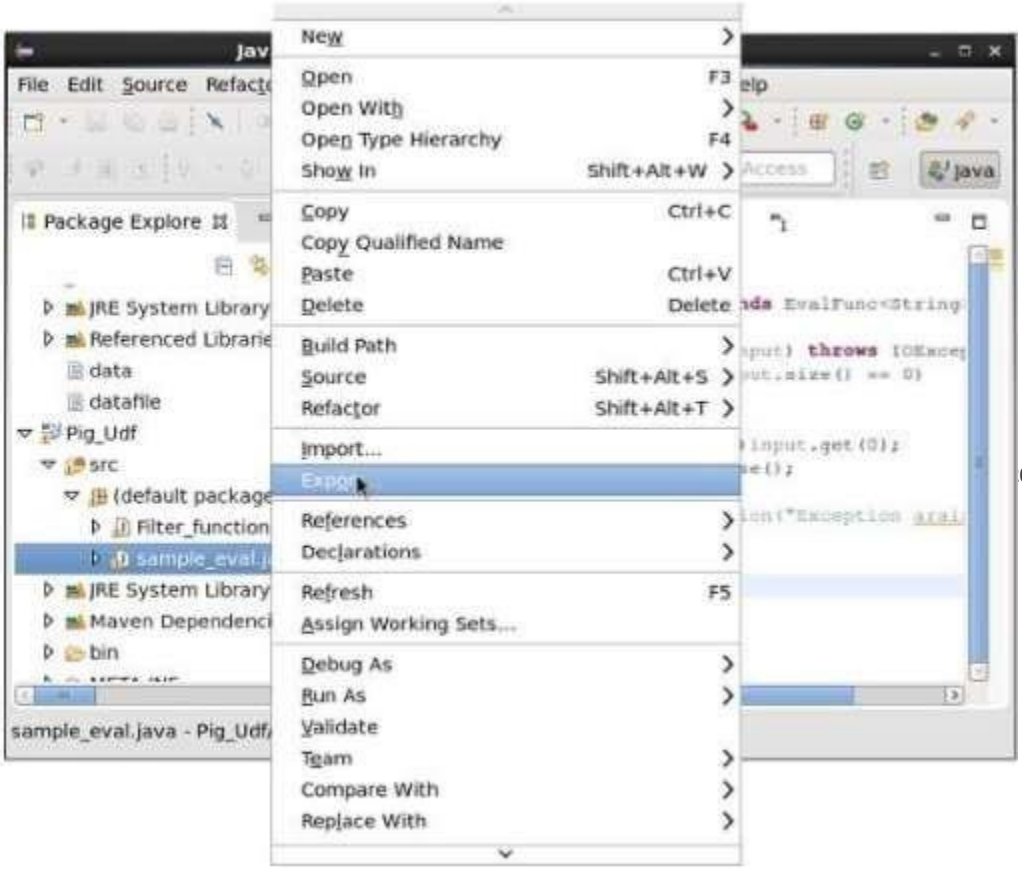
```
  </dependencies>

</project>
```

- Save the file and refresh it. In the **Maven Dependencies** section, you can find the downloaded jar files.
- Create a new class file with name **Sample_Eval** and copy the following content in it.

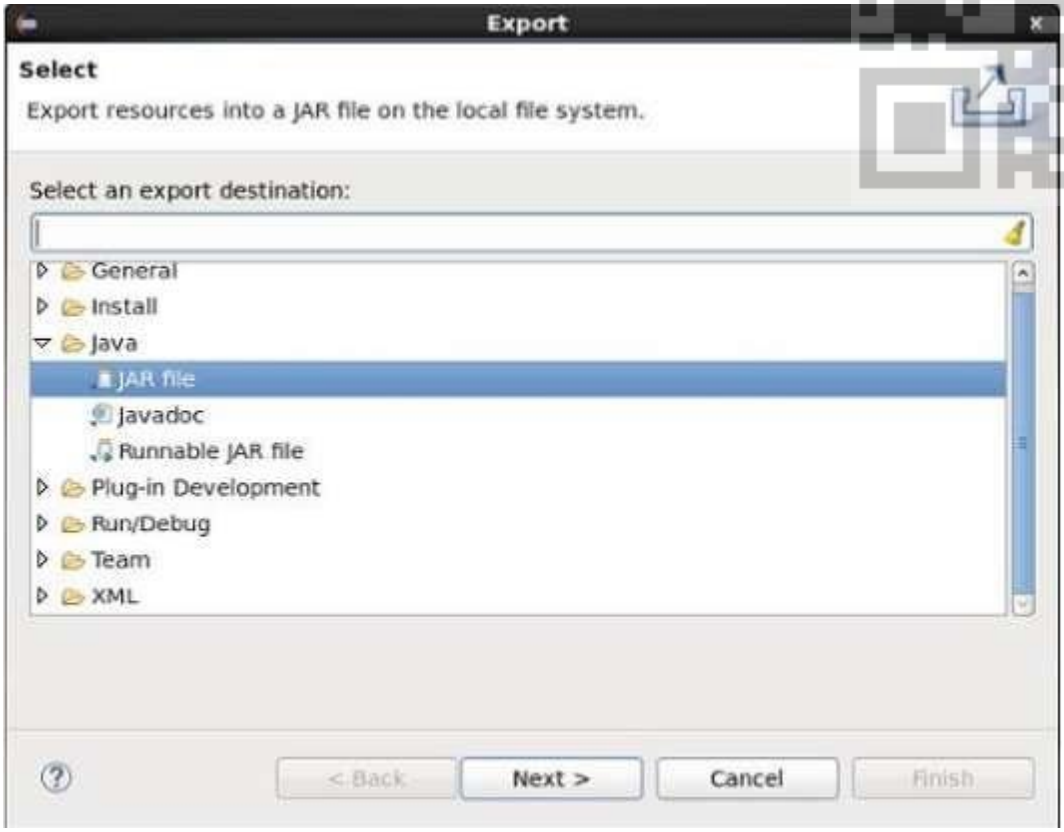
```
import java.io.IOException; import
org.apache.pig.EvalFunc; import
org.apache.pig.data.Tuple;

import java.io.IOException; import
org.apache.pig.EvalFunc; import
org.apache.pig.data.Tuple;

public class Sample_Eval extends EvalFunc<String>{

   public String exec(Tuple input) throws IOException {
      if (input == null || input.size() == 0)
      return null;
      String str = (String)input.get(0);
      return str.toUpperCase();
   }
}
```

While writing UDF's, it is mandatory to inherit the EvalFunc class and provide implementation to **exec()** function. Within this function, the code required for the UDF is written. In the above example, we have return the code to convert the contents of the given column to uppercase.

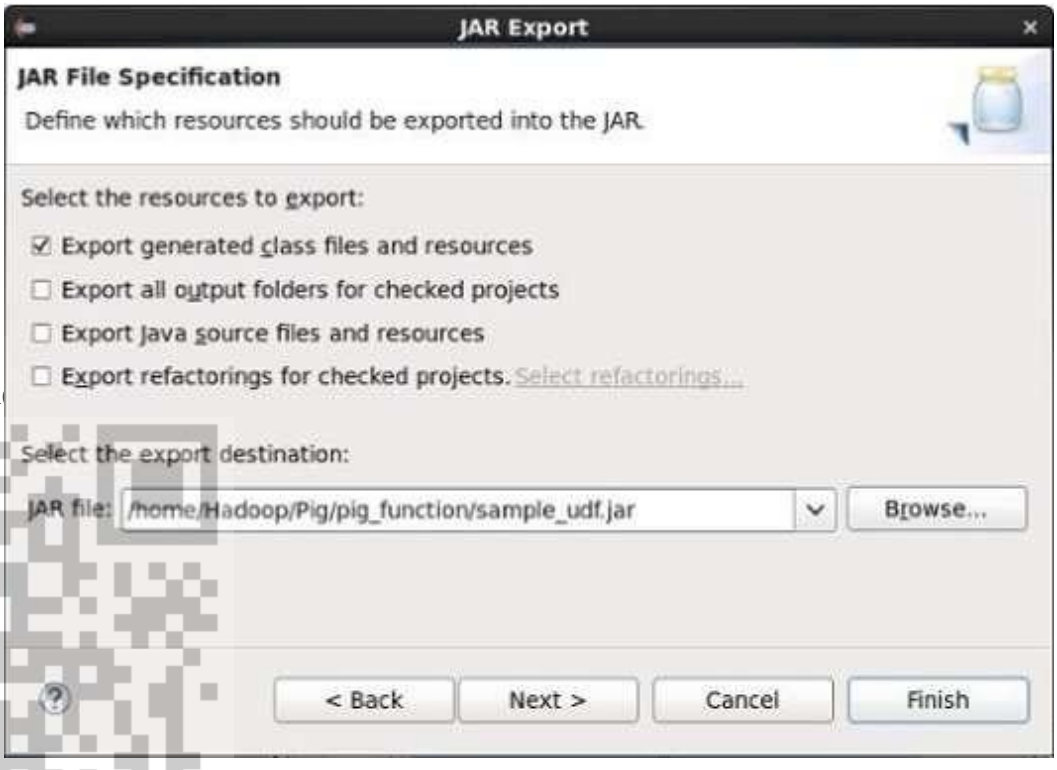
- After compiling the class without errors, right-click on the Sample_Eval.java file. It gives you a menu. Select **export** as shown in the following screenshot.



- On clicking **export**, you will get the following window. Click on **JAR file**.



- Proceed further by clicking **Next>** button. You will get another window where you needto enter the path in the local file system, where you need to store the jar file.



- Finally click the **Finish** button. In the specified folder, a Jar file **sample_udf.jar** is created. This jar file contains the UDF written in Java.

Using the UDF

After writing the UDF and generating the Jar file, follow the steps given below –Step 1:

Registering the Jar file

After writing UDF (in Java) we have to register the Jar file that contain the UDF using the Register operator. By registering the Jar file, users can intimate the location of the UDF to Apache Pig.

Syntax

Given below is the syntax of the Register operator.

```
REGISTER path;
```

Example

As an example let us register the sample_udf.jar created earlier in this chapter.

Start Apache Pig in local mode and register the jar file sample_udf.jar as shown below.

```
$cd PIG_HOME/bin
$./pig –x local
```



```
REGISTER '/$PIG_HOME/sample_udf.jar'
```

Note – assume the Jar file in the path – /$PIG_HOME/sample_udf.jarStep 2:

Defining Alias

After registering the UDF we can define an alias to it using the **Define** operator.

Syntax

Given below is the syntax of the Define operator.

```
DEFINE alias {function | [`command` [input] [output] [ship] [cache] [stderr] ] };
```

Example

Define the alias for sample_eval as shown below.

```
DEFINE sample_eval sample_eval();
```


Step 3: Using the UDF

After defining the alias you can use the UDF same as the built-in functions. Suppose there is afile named emp_data in the HDFS **/Pig_Data/** directory with the following content.

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuwaneshwar
006,Maggy,22,Chennai
```

007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuwaneshwar
012,Kelly,22,Chennai

And assume we have loaded this file into Pig as shown below.

```
grunt> emp_data = LOAD 'hdfs://localhost:9000/pig_data/emp1.txt' USING PigStorage(',')as
   (id:int, name:chararray, age:int, city:chararray);
```

Let us now convert the names of the employees in to upper case using the UDF **sample_eval**.

```
grunt> Upper_case = FOREACH emp_data GENERATE sample_eval(name);
```

Verify the contents of the relation **Upper_case** as shown below.

grunt> Dump Upper_case;

(ROBIN)
(BOB)
(MAYA)
(SARA)
(DAVID)
(MAGGY)
(ROBERT)
(SYAM)
(MARY)
(SARAN)
(STACY)
(KELLY)

Parameter substitution in Pig

Earlier I have discussed about writing reusable scripts using Apache Hive, now we see how toachieve same functionality using Pig Latin.

Pig Latin has an option called param, using this we can write dynamic scripts .

Assume ,we have a file called numbers with below data.12

23

34

12

56

34

57

12

```
Numbers = load '/data/numbers' as (number:int);
specificNumber = filter numbers by number==12;
Dump specificNumber;
```

Usually we write above code in a file .let us assume we have written it in a file called numbers.pigAnd we write code from file using

```
Pig –f /path/to/numbers.pig
```

Later if we want to see only numbers equals to 34, then we change second line to

```
specificNumber = filter numbers by number==34;
```

and we re-run the code using same command.
But Its not a good practice to touch the code in production ,so we can make this script dynamic byusing –param option of Piglatin.
Whatever values we want to decide at the time of running we make them dynamic .now we wantto decide number to be filtered at the time running job,we can write second line like below.

```
specificNumber = filter numbers by number==$dynanumber
```

and we run code like below.


```
Pig –param dynanumber=12 –f numbers.pig
```

Assume we even want to take path at the time of running script, now we write code like below

```
Numbers = load '$path' as (number:int);
specificNumber = filter numbers by number=='$ dynanumber';
```

```
Dump specificNumber;
```

And run like below

```
Pig –param path=/data/path –param dynanumber =34 –f numbers.pig
```

If you feel this code is missing readability, we can specify all these dynamic values in a file likebelow ##Dyna.params (file name)

```
Path = /data/numbers

dynanumber = 34
```

Then you can run script with param-file option like below.

```
Pig –param-file dyna.params –f numbers.pig
```

Pig Latin provides four different types of diagnostic operators –

- Dump operator
- Describe operator
- Explanation operator
- Illustration operator

Word Count Example Using Pig Script:

30103162101496 **30103162101496** **30103162101496**

```
lines = LOAD '/user/hadoop/HDFS_File.txt' AS (line:chararray);
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) as word;grouped = GROUP words BY word;
wordcount = FOREACH grouped GENERATE group, COUNT(words);DUMP wordcount;
```

The above pig script, first splits each line into words using the **TOKENIZE** operator. Thetokenize function creates a bag of words. Using the **FLATTEN** function, the bag is

converted into a tuple. In the third statement, the words are grouped together so that the count can be computed which is done in fourth statement.

Pig at Yahoo

Pig was initially developed by Yahoo! for its data scientists who were using Hadoop. It was incepted to focus mainly on analysis of large datasets rather than on writing mapper and reduce functions. This allowed users to focus on what they want to do rather than bothering with how its done. On top of this with Pig language you have the facility to write commands in other languages like Java, Python etc. Big applications that can be built on Pig Latin can be custom built for different companies to serve different tasks related to data management. Pig systemizes all the branches of data and relates it in a manner that when the time comes, filtering and searching data is checked efficiently and quickly.

Pig Versus Hive

Pig Vs Hive

Here are some basic difference between Hive and Pig which gives an idea of which to usedepending on the type of data and purpose.

Pig	Hive
Used by Programmers and Researchers	Used by Analysts
Used for Programming	Used for Reporting
Procedural data-flow language	Declarative SQLish language
Works on the Client side of a Cluster	Works on the Server side of a Cluster
For Semi-Structured Data	For Structured Data

Why Go for Hive When Pig is There?

The tabular column below gives a comprehensive comparison between the two. The Hive can beused in places where partitions are necessary and when it is essential to define and create cross- language services for numerous languages.

Features	Hive	Pig
Language	SQL-like	PigLatin
Schemas/Types	Yes (explicit)	Yes (implicit)
Partitions	**Yes**	**No**
Server	Optional (Thrift)	No
User Defined Functions (UDF)	Yes (Java)	Yes (Java)
Custom Serializer/Deserializer	Yes	Yes
DFS Direct Access	Yes (implicit)	Yes (explicit)
Join/Order/Sort	Yes	Yes
Shell	Yes	Yes
Streaming	Yes	Yes
Web Interface	Yes	No
JDBC/ODBC	Yes (limited)	No