LECTURE NOTES

ON

BIG DATA AND BUSINESS ANALYTICS

2020 – 2021

VII Semester

(IARE-R16)

Dr. M Madhu Bala, Professor

Computer Science and Engineering

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad - 500 043

UNIT I

INTRODUCTION TO BIG DATA

Topics

- ✓ **Introduction to Big data**
- ✓ **Characteristics of Data**
- ✓ **Evolution of Big Data**
- ✓ **Definition of Big Data**
- ✓ **Challenges with Big Data**
- ✓ **Traditional Business Intelligence (BI) versus Big Data**
- ✓ **Big data analytics**
- ✓ **Classification of Analytics**
- ✓ **Importance and challenges facing big data**
- ✓ **Terminologies Used in Big Data Environments**
- ✓ **The Big Data Technology Landscape.**

1.1 INTRODUCTION TO BIG DATA

Characteristics of Data:

Before going to know in detail about the characteristics of data. Let us know the term BIG DATA, BIG means bigger in size and DATA means raw data.

Irrespective of the size of the enterprise (big or small), data continues to be a precious and irreplaceable asset. Data is present internal to the enterprise and also exists outside the four walls and firewalls of the enterprise. Data is present in homogeneous sources as well as in heterogeneous sources. The need of the hour is to understand, manage, process, and take the data for analysis to draw valuable insights.

Data -- Information

Information -- Insights

Broadly classify the difference between the Data and Information

	Data	**Information**
Meaning:	Data is raw, unorganized facts that need to be processed. Data can be something simple and seemingly random and useless until it is organized.	When data is processed, organized, structured or presented in a given context so as to make it useful, it is called Information.
Example:	Each student's test score is one piece of data	The class' average score or the school's average score is the information that can be concluded from the given data.
Definition:	Latin 'datum' meaning "that which is given". Data was the plural form of datum singular.	Information is interpreted data.

Characteristics of Data:

Let us start with the characteristics of data, As depicted in below figure, data has three key characteristics:

Composition: The composition of data deals with the structure of data, that is, the sources or at different granularity, the types, and the nature of data as to whether it is static or real-time streaming

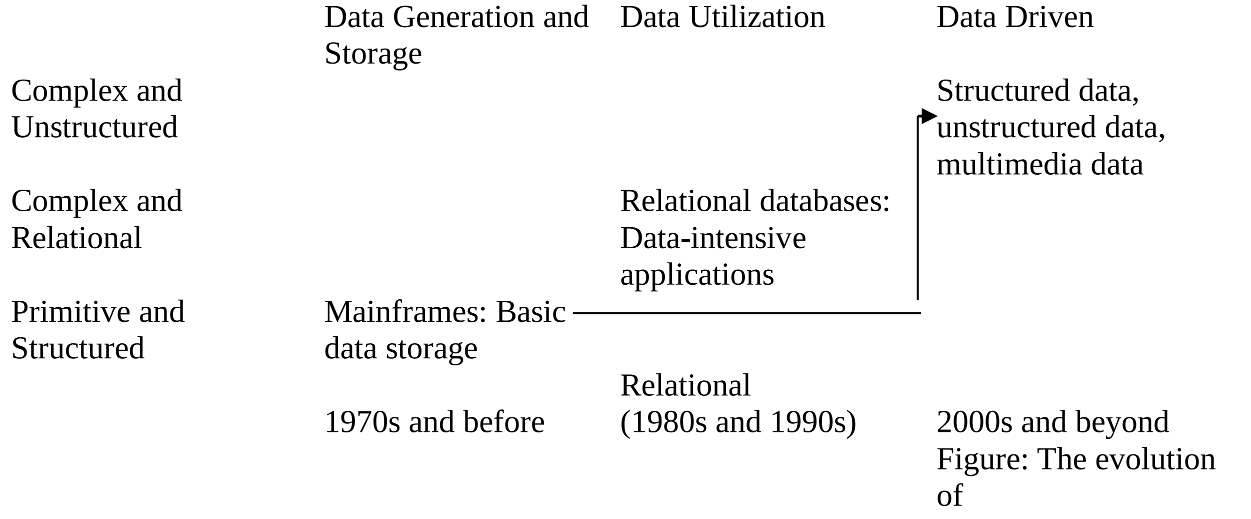
Condition: The condition of data deals with the state of data, that is, "Can one use this data as is for analysis? or "Does it require cleansing for further enhancement and enrichment: **Context**: The context of data deals with Where has this data been generated? Why was this data generated? "How sensitive is this data?" What are the events associated with this data? and so on.

Small data (data as it existed prior to the big data revolution) is about certainty. It is about fairly known data sources; it is about no major changes to the composition or context of data.

Most often we have answers to queries like why this data was generated, where and when it was generated exactly how we would like to use it, what questions will this data be able to answer, and so on. Big data is about complexity in terms of multiple and unknown datasets, in terms of exploding volume, in terms of the speed at which the data is being generated and the speed at which the data is being generated and the speed at which it needs to be processed, and in terms of the variety of data(internal or external, behavioral or social) that is being generated.

Evolution of Big Data:

1970c and before was the era of mainframes. The data was essentially primitive and structured. Relational databases evolved in 1980s and 1990s. The era was of data intensive applications. The World Wide Web www) and the Internet of Things (loT) have led to an onslaught of structured, unstructured, and multimedia data.



	Data Generation and Storage	Data Utilization	Data Driven
Complex and Unstructured			Structured data, unstructured data, multimedia data
Complex and Relational		Relational databases: Data-intensive applications	
Primitive and Structured	Mainframes: Basic data storage		
	1970s and before	Relational (1980s and 1990s)	2000s and beyond Figure: The evolution of

Figure: Evolution of Big Data

Definition of Big Data:

If we were to ask you the simple question: "Define Big Data", what would your answer be? Well, we will give you a few responses that we have heard over time:

1. Anything beyond the human and technical infrastructure needed to support storage, processıng, and analysis.

2. Today's BIG may be tomorrow's NORMAL.

3. Terabytes or petabytes or zettabytes of data.

4. I think it is about 3 Vs.

Well, all of these responses are correct. But it is not one of these; in fact, big data is all of the above and more.

Big data is high-volume, high-velocity, and high-variety information assets that demand cost effective, innovative forms of information processing for enhanced insight and decision making.

--Gartner.

The 3Vs concept was proposed by the Gartner analyst Doug Laney in a 2001 Meta Group research
publication, titled, 3D Data Management: Controlling Data Volume, Variety and Velocity.

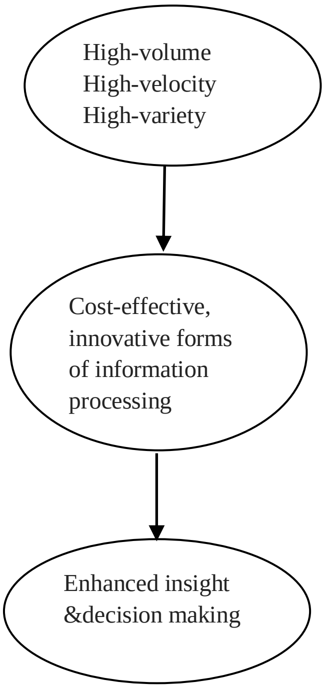


Figure: Definition of big data Gartner.

Part I of the definition "big data is high-volume, high-velocity, and high-variety information assets" talks about voluminous data (humongous data) that may have great varíety (a good mix of structured, semi-structured, and unstructured data) and will require a good speed/pace for storage, preparation, processing and analysis.

Part II of the definition "cost effective, innovative forms of information processing talks about embracing new techniques and technologies to capture (ingest), store, process, persist, integrate, and visualize the high-volume, high-velocity, and high-variety data.

Part III of the definition "enhanced insight and decision making" talks about deriving deeper, richer, and meaningful insights and then using these insights to make faster and better decisions to gain business value and thus a competitive edge.

Data → Information → Actionable Intelligence→Better decisions → Enhanced business value.

Challenges with Big Data:

1. Data today is growing at an exponential rate. Most of the data that we have today has been generated in the last 2-3 years. This high tide of data will continue to rise incessantly. The key questions here are: " Will all this data be useful for analysis?", "Do we work with al this data or a subset of it?", "How will we separate the knowledge from the noise?", etc.

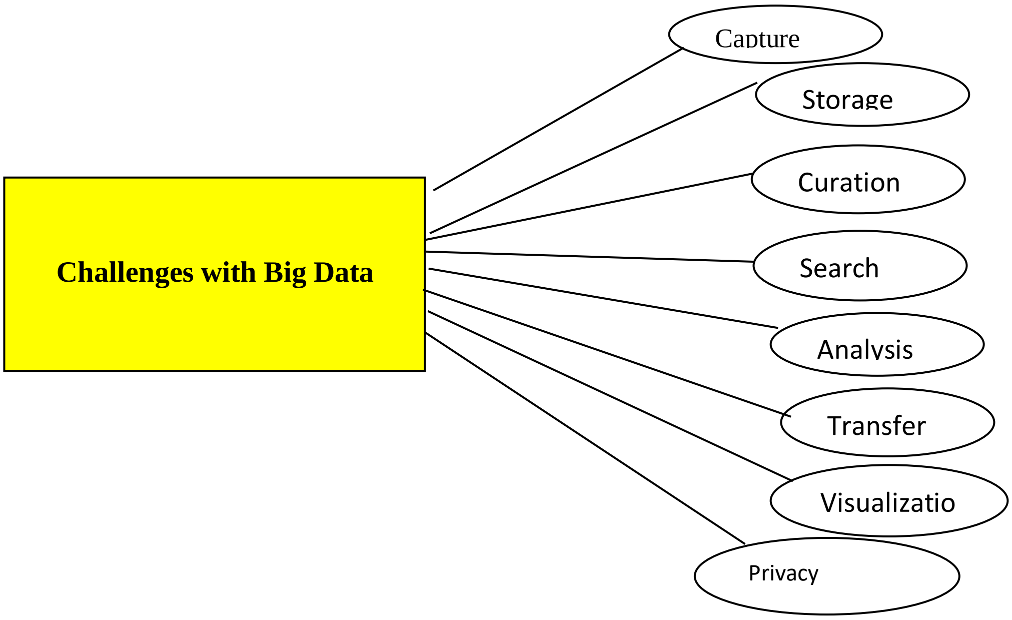
2. Cloud computing and virtualization are here to stay. Cloud computing is the answer to managing infrastructure for big data as far as cost-efficiency, elasticity, and easy upgrading/downgrading is concerned. This further complicates the decision to host big data solutions outside the enterprise.

3. The other challenge is to decide on the period of retention of big data. Just how long should one retain this data? A tricky question indeed as some data is useful for making long-term decisions, whereas in few cases, the data may quickly become irrelevant and obsolete just a few hours after having being generated.

4. There is a dearth of skilled professionals who possess a high level of proficiency in data sciences that is vital in implementing big data solutions.

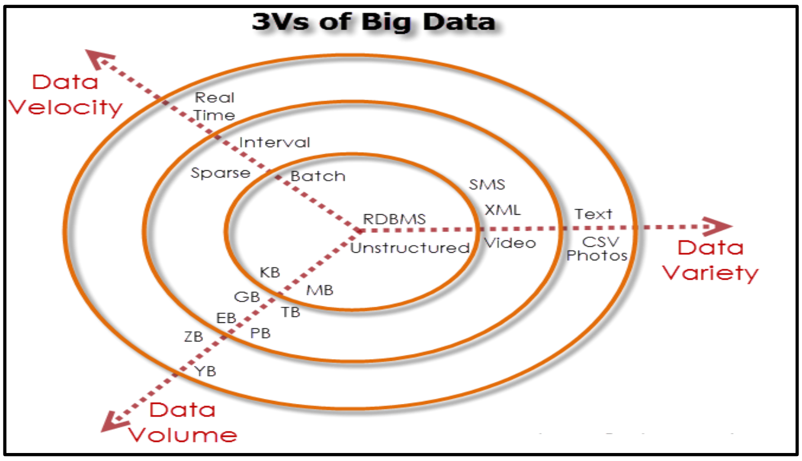
5. Then, of course, there are other challenges with respect to capture, storage, preparation, search, analysis, transfer, security, and visualization of big data. Big data refers to datasets whose size is typically beyond the storage capacity of traditional database software tools. There is no explicit definition of how big the dataset should be for it to be considered big data. Here we are to deal with data that is just too big, moves way to fast, and does not fit the structures of typical database systems. The data changes are highly dynamic and therefore there is a need to ingest this as quickly as possible.

6. Data visualization is becoming popular as a separate discipline. We are short by quite a number, as far as business visualization experts are concerned.

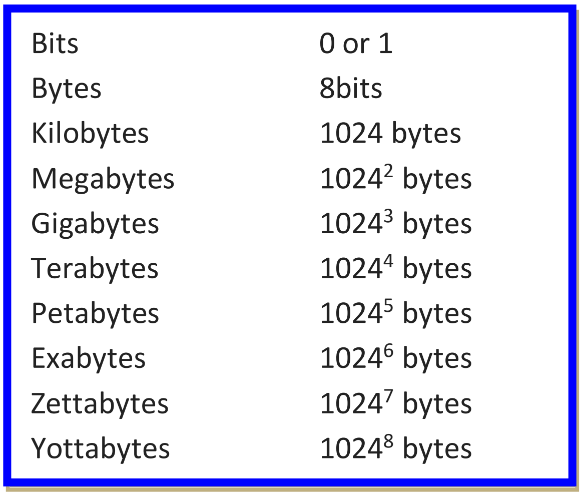


What is Big Data?

Big data is data that is big in volume, velocity, and variety



Volume: It grow from bits to bytes to petabytes and exabytes.



Bits	0 or 1
Bytes	8bits
Kilobytes	1024 bytes
Megabytes	1024^2 bytes
Gigabytes	1024^3 bytes
Terabytes	1024^4 bytes
Petabytes	1024^5 bytes
Exabytes	1024^6 bytes
Zettabytes	1024^7 bytes
Yottabytes	1024^8 bytes

Figure: Growth of data

Where does this data get generated?

There are a multitude of sources for big data. An XLS, a DOC, a PDF, etc. is unstructured data; a video on YouTube, a chat conversation on Internet Messenger, customer feedback form on an online retail website is unstructured data; a CCTV coverage, weather forecast report is unstructured data too.

1. Typical internal data sources: Data present within an organization's firewall. It is as follows: Data storage: File systems, SQL (RDBMSs- Oracle, MS SQL Server, DB2, MySQL, PostgreSQL, etc.), NoSQL (MongoDB, Cassandra, etc.), and so on.

Archives: Archives of scanned documents, paper archives, customer correspondence records, patients health records, students admission records, students assessment records, and so on

2. External data sources: Data residing outside an organization's firewall. It is as follows:

Public Web: Wikipedia, weather, regulatory, compliance, census, etc.

3. Both (internal + external data sources)

Sensor data:

Car sensors, smart electric meters, office buildings, air conditioning units, refrigera tors, and so on.

Machine log data: Event logs, application logs, Business process logs, audit logs, clickstream data, etc.

Social media:

Twitter, blogs, Facebook, Linkedln, YouTube, Instagram, etc.

Business apps:

ERP, CRM, HR, Google Docs, and so on.

Media: Audio, Video, Image, Podcast, etc.

Docs: Comma separated value (CSV), Word Documents, PDF, XLS, PPT, and so on.

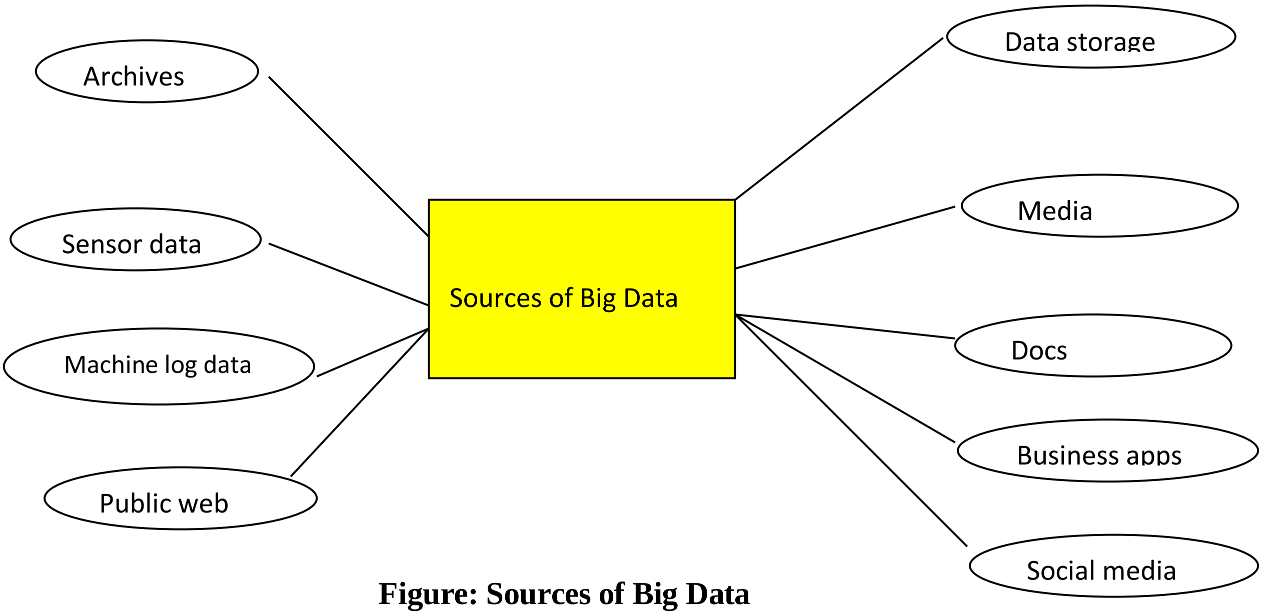


Figure: Sources of Big Data

Velocity:

We have moved from the days of batch processing to real time processing.

Batch → **periodic** → **Near real time** → **Real time processing**

Variety:

Variety deals with a Wide range of data types and sources of data. We will study this under three categories:

Structured data, semi-structured data and unstructured data.

1. Structured data: From traditional transaction processing systems and RDBMS, etc.

2. Semi-structured data: For example Hyper Text Markup Language (HTML), eXensible Markup Language (XML).

3. Unstructured data: For example unstructured text documents, audios, videos, emails, photos, PDFs, social media, etc.

Evolution of Big Data, Definition of Big Data, Challenges with Big Data, Traditional Business Intelligence (BI) versus Big Data.

Big data analytics: Classification of Analytics, Importance and challenges facing big data, Terminologies Used in Big Data Environments, The Big Data Technology Landscape.

Big Data is becoming one of the most talked about technology trends nowadays. The real challenge with the big organization is to get maximum out of the data already available and predict what kind of data to collect in the future. How to take the existing data and make it meaningful that it provides us accurate insight in the past data is one of the key discussion points in many of the executive meetings in organizations.

With the explosion of the data the challenge has gone to the next level and now a Big Data is becoming the reality in many organizations. The goal of every organization and expert is same to get maximum out of the data, the route and the starting point are different for each organization and expert. As organizations are evaluating and architecting big data solution they are also learning the ways and opportunities which are related to Big Data.

There is not a single solution to big data as well there is not a single vendor which can claim to know all about Big Data. Big Data is too big a concept and there are many players – different architectures, different vendors and different technology.

The three Vs of Big data are Velocity, Volume and Variety.

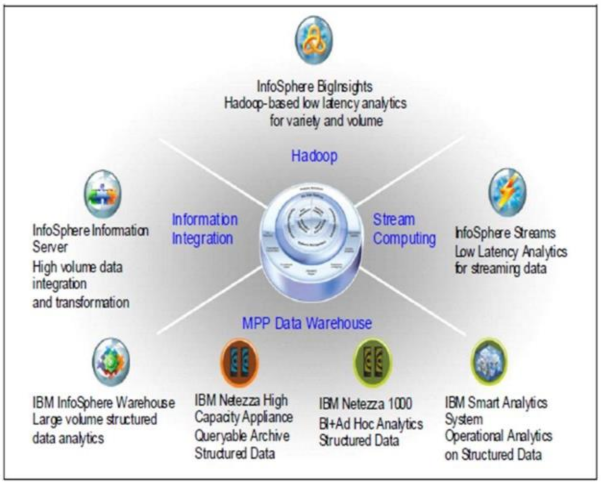


Figure 1.1: Big Data Sphere

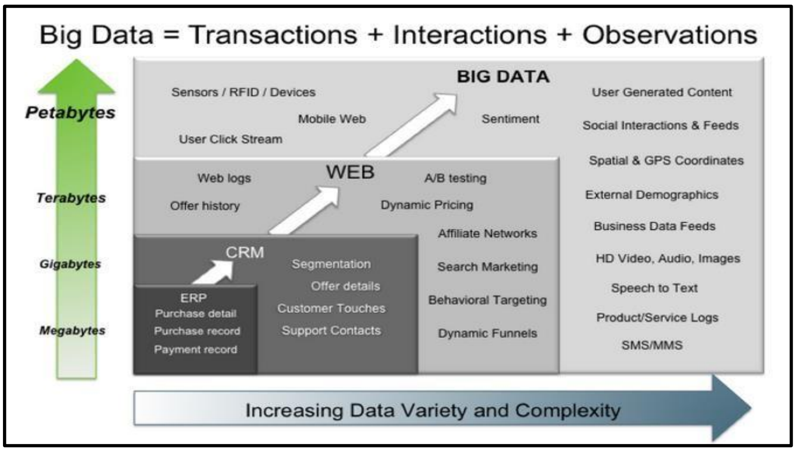


Figure 1.2: Big Data – Transactions, Interactions, Observations

1.2 BIG DATA CHARACTERISTICS

1. The three Vs of Big data are Velocity, Volume and Variety

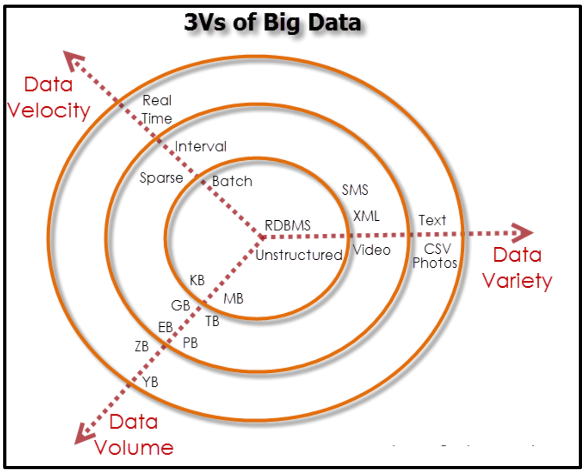


Figure : Characteristics of Big

VOLUME

The exponential growth in the data storage as the data is now more than text data. The data can be found in the format of videos, music’s and large images on our social media channels. It is very common to have Terabytes and Petabytes of the storage system for enterprises. As the database grows the applications and architecture built to support the data needs to be re-evaluated quite often.

Sometimes the same data is re-evaluated with multiple angles and even though the original data is the same the new intelligence creates explosion of the data. The big volume indeed represents Big Data.

VELOCITY

The data growth and social media explosion have changed how we look at the data. There was a time when we used to believe that data of yesterday is recent. The matter of the fact newspapers is still following that logic. However, news channels and radios have changed how fast we receive the news.

Today, people reply on social media to update them with the latest happening. On social media sometimes a few seconds old messages (a tweet, status updates etc.) is not something interests users.

They often discard old messages and pay attention to recent updates. The data movement is now almost real time and the update window has reduced to fractions of the seconds. This high velocity data represent Big Data.

VARIETY

Data can be stored in multiple format. For example database, excel, csv, access or for the matter of the fact, it can be stored in a simple text file. Sometimes the data is not even in the traditional format as we assume, it may be in the form of video, SMS, pdf or something we might have not thought about it. It is the need of the organization to arrange it and make it meaningful.

It will be easy to do so if we have data in the same format, however it is not the case most of the time. The real world have data in many different formats and that is the challenge we need to overcome with the Big Data. This variety of the data represent Big Data.

1.3 Big Data Technologies

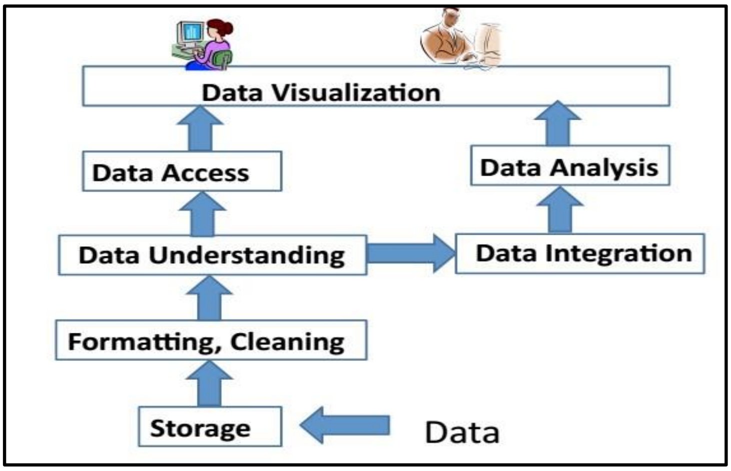


Figure 1.5 : Big Data Layout

1. APACHE HADOOP

Apache Hadoop is one of the main supportive elements in Big Data technologies. It simplifies the processing of large amount of structured or unstructured data in a cheap manner. Hadoop is an open source project from apache that is continuously improving over the years. "Hadoop is basically a set of software libraries and frameworks to manage and process big amount of data from a single server to thousands of machines.

It provides an efficient and powerful error detection mechanism based on application layer rather than relying upon hardware."

In December 2012 apache releases Hadoop 1.0.0, more information and installation guide can be found at Apache Hadoop Documentation. Hadoop is not a single project but includes a number of other technologies in it.

2.MAPREDUCE

MapReduce was introduced by google to create large amount of web search indexes.It is basically a framework to write applications that processes a large amount of structured or unstructured data over the web. MapReduce takes the query and breaks it into parts to run it on multiple nodes. By distributed query processing it makes it easy to maintain large amount of data by dividing the data into several different machines.Hadoop MapReduce is a software framework for easily writing applications to manage large amount of data sets with a highly fault tolerant manner. More tutorials and getting started guide can be found at Apache Documentation.

3.HDFS (Hadoop distributed file system)

HDFS is a java based file system that is used to store structured or unstructured data over large clusters of distributed servers. The data stored in HDFS has no restriction or rule to be applied, the data can be either fully unstructured of purely structured.In HDFS the work to make data senseful is done by developer's code only. Hadoop distributed file system provides a highly fault tolerant atmosphere with a deployment on low cost hardware machines. HDFS is now a part of Apache Hadoop project, more information and installation guide can be found at Apache HDFS documentation.

4. HIVE

Hive was originally developed by Facebook, now it is made open source for some time. Hive works something like a bridge in between sql and Hadoop, it is basically used to make Sql queries on Hadoop clusters. Apache Hive is basically a data warehouse that provides ad-hoc queries, data summarization and analysis of huge data sets stored in Hadoop compatible file systems.

Hive provides a SQL like called HiveQL query-based implementation of huge amount of data stored in Hadoop clusters. In January 2013 apache releases Hive 0.10.0, more information and installation guide can be found at Apache Hive Documentation.

5. PIG

Pig was introduced by yahoo and later on it was made fully open source. It also provides a bridge to query data over Hadoop clusters but unlike hive, it implements a script implementation to make Hadoop data access able by developers and businesspersons. Apache pig provides a high-level programming platform for developers to process and analyses Big Data using user defined functions and programming efforts. In January 2013 Apache released Pig 0.10.1 which is defined for use with Hadoop 0.10.1 or later releases. More information and installation guide can be found at Apache Pig Getting Started Documentation.

1.4 TRADITIONAL VS BIG DATA BUSINESS APPROACH

1. Schema less and Column oriented Databases (NoSql)

We are using table and row based relational databases over the years, these databases are just fine with online transactions and quick updates. When unstructured and large amount of data comes into the picture we needs some databases without having a hard code schema attachment. There are a number of databases to fit into this category, these databases can store unstructured, semi structured or even fully structured data.

Apart from other benefits the finest thing with schema less databases is that it makes data migration very easy. MongoDB is a very popular and widely used NoSQL database these days.NoSQL and schema less databases are used when the primary concern is to store a huge amount of data and not to maintain relationship between elements. "NoSQL (not only Sql) is a type of databases that does not primarily rely upon schema based structure and does not use Sql for data processing."

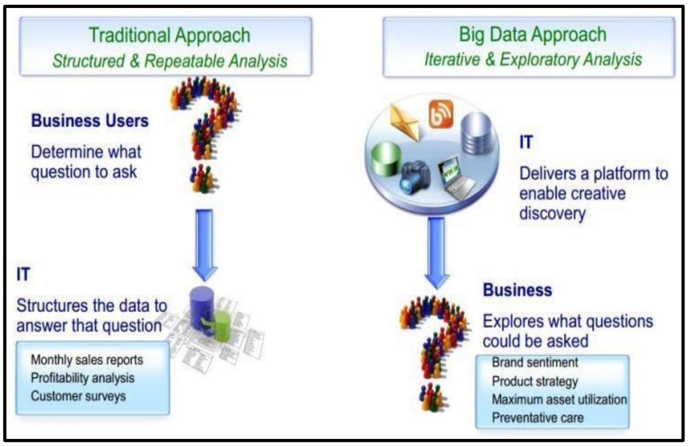

Traditional Approach
Structured & Repeatable Analysis
Business Users
Determine what question to ask
IT
Structures the data to answer that question
Monthly sales reports
Profitability analysis
Customer surveys
Big Data Approach
Iterative & Exploratory Analysis
IT
Delivers a platform to enable creative discovery
Business
Explores what questions could be asked
Brand sentiment
Product strategy
Maximum asset utilization
Preventative care

Figure 1. : Big Data

The traditional approach work on the structured data that has a basic layout and the structure provided.

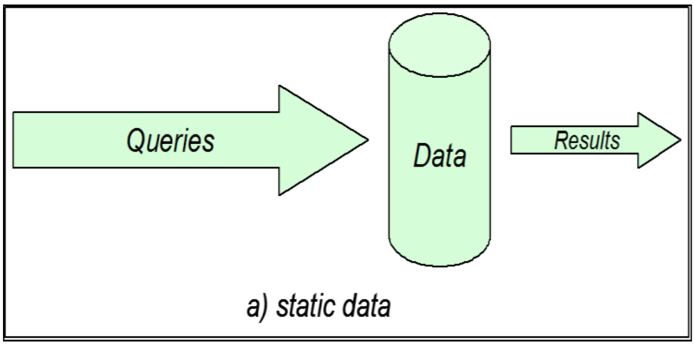


Figure 1.7: Static Data

The structured approach designs the database as per the requirements in tuples and columns. Working on the live coming data, which can be an input from the ever changing scenario cannot be dealt in the traditional approach. The Big data approach is iterative.

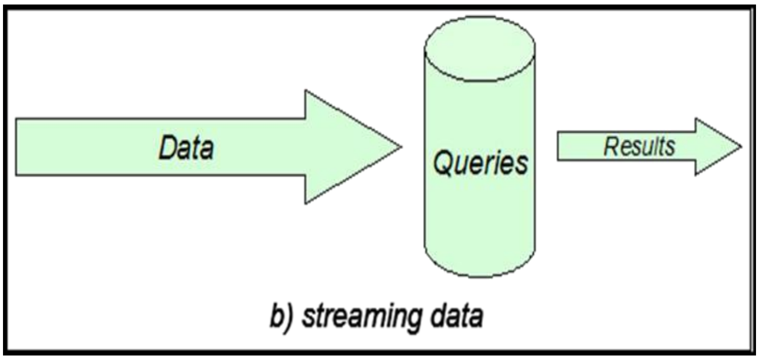
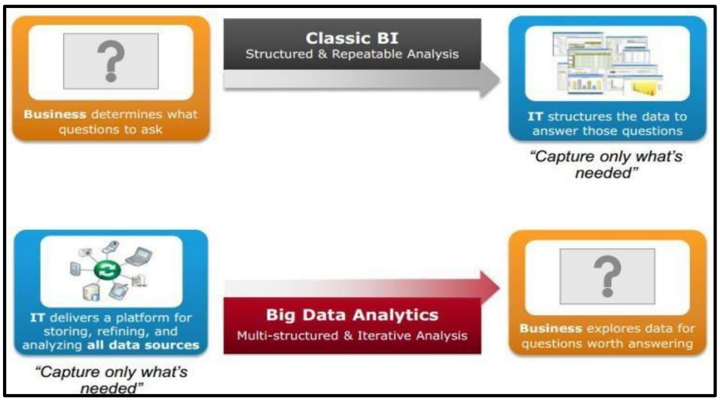


Figure 1.8: Streaming Data

The Big data analytics work on the unstructured data, where no specific pattern of the data is defined. The data is not organized in rows and columns. The live flow of data is captured and the analysis is done on it.

Efficiency increases when the data to be analyzed is large.


Business determines what questions to ask
Classic BI
Structured & Repeatable Analysis
IT structures the data to answer those questions
"Capture only what's needed"
IT delivers a platform for storing, refining, and analyzing all data sources
"Capture only what's needed"
Big Data Analytics
Multi-structured & Iterative Analysis
Business explores data for questions worth answering

1.9 Big Data Architecture

UNIT II

INTRODUCTION TO HADOOP

Topics:

- ✓ **Introducing Hadoop**
- ✓ **Why Hadoop**
- ✓ **Why not RDBMS**
- ✓ **RDBMS vs Hadoop**
- ✓ **Distributed Computing Challenges**
- ✓ **History of Hadoop**
- ✓ **Clickstream data analysis with Hadoop –Pig**
- ✓ **Demonstration on usecase**
- ✓ **Hadoop Distributors**
- ✓ **Processing Data with Hadoop**

Interacting with Hadoop Ecosystem

2.1 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 powerful distributed platform to store and manage BigData.
4. It is licensed under an Apache V2license.
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.

2.2 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. AffordableandCostEffective–Wedonotneedanyspecialhardwareforrunning 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.

2.3 CORE HADOOP COMPONENTS

There are two major components of the Hadoop framework and both of them does two of the important task 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 resources and they have processed them locally. Once the commodity server has processed the data they send 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 store do not he r servers (usually3) 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 name space 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 editlog. In case if the name node crashes, then the namespace image 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 also checks for any failed tasks and reschedules the failed tasks on another DataNode. Job tracker can be run on the NameNode or a separate node.

7. Task Tracker: Task tracker runs on the DataNodes. Task trackers responsibility is torunthemaporreducetasksassignedbytheNameNodeandtoreportthestatus 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

2.4 RDBMS

Why can’t we used at a bases with lots of disk storage- 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 traditional 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.

MapReduce is a good fit for problems that need to analyze the whole dataset, in a batch fashion, particularly for adhoc 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 a sa 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 anon- local operation, and one of the central assumptions that MapReduce makes is that itis 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 host names are specified in full each time, even though the same client may appear many times), and this is one reason that log files 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 sizeof the data or the cluster that they are operating on, so they can be used unchanged for a small data set 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.

Overtime,however,thedifferencesbetweenrelationaldatabasesandMapReduce systems 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.

2.5 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, butit 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 in2002, and a working crawler and search system quickly emerged.

However,theyrealizedthattheirarchitecturewouldn'tscaletothebillionsofpagesonthe 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 particular, 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. 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.

NDFSandtheMapReduceimplementationinNutchwereapplicablebeyondtherealm 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.

InJanuary2008, 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- binationofAmazon'spay-by-the-hourmodel(whichallowedtheNYTtoaccessalarge 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 terabyte in 68 seconds. As the first edition of this book was going to press (May2009), it was announced that a team at Yahoo! Used Hadoop to sort one tera byte 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 query as 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.

TheinputtoourmapphaseistherawNCDCdata.Wechooseatextinputformatthat 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 preparation phase, setting up the data in such a way that the reducer function can do its work on it: finding the 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 mapworks, consider the following sample lines of input data (some unused columns have been dropped to fit the page, indicated by ellipses):

```
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+9999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+9999999999...)
(212,   0043011990999991950051518004...9999999N9-00111+9999999999...)
```

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

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

The keys are the line off sets with in 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])

Eachyearappearswithalistofallitsairtemperaturereadings.Allthereducefunction has to do no w 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.

Thewholedataflowisillustratedin2.2.AtthebottomofthediagramisaUnix pipeline, which mimics the whole MapReduce flow, and which we will see again later in the chapter when we look at Hadoop Streaming.

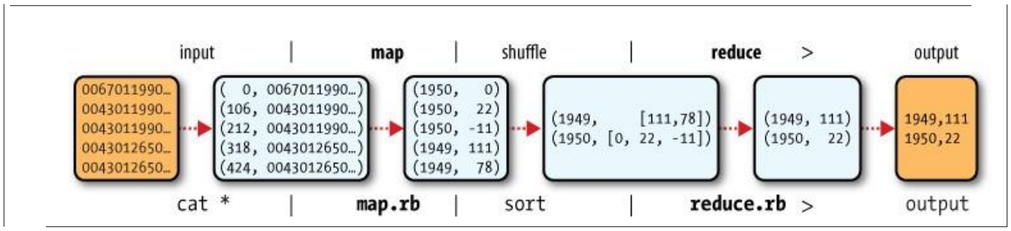


Figure 2-1. MapReduce logical data flow

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 inputkey,inputvalue,outputkey,andoutputvaluetypesofthemapfunction.Forthe 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 Javatypes, Hadoop provides its own set of basic types that are optimized for network serialization. These are found in the org.apache.hadoop.iopackage.

Here we use LongWritable, which corresponds to a JavaLong, Text (like JavaString), and IntWritable (like JavaInteger).

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 out put types, this time for the 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.

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 beasinglefile,adirectory(inwhichcase,theinputformsallthefilesinthatdirectory), or a file pattern. As the name suggests, addInputPath() can be called more than once to use input from multiplepaths.

The output path (of which there is only one) is specified by the static setOutputPath()method on FileOutputFormat. It specifies a directory where the output files from the reducer functions are written. The directory should n'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 annoying to 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 forthemapandthereducefunctions,whichareoftenthesame,astheyareinourcase. 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 inputformat, which we have not explicitly sets using the default TextInputFormat.

After setting the classes that define the map and reduce functions, we are ready to run the job. The wait ForCompletion() 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 flushout 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 Hadooprunsusingthelocalfilesystemwithalocaljobrunner.Theninstallandcompile the examples using the instructions on the book'swebsite.

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-r-00000
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 JavaMapReduceAPI used in the previous section was first released in Hadoop

0.20.0. This newAPI, sometimes referred to as "ContextObjects,"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 newAPIisnotcompleteinthe1.x(formerly0.20) 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 in variably 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.

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. For example, the MapperandReducer 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 inorg.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, butin 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 processed. In the old API this is possible form appears 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.InterruptedException. 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 needsto.

In the new APIthereduce()method passes values as a java.lang.Iterable, rather than a java.lang.Iterator (as the oldAPIdoes). This change make site as over the values using Java's for-each loop construct: for (VALUEIN value : values) { ...}

2.7 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

Adataflowlanguageandexecutionenvironmentforexploringverylargedatasets. Pig runs on HDFS and MapReduceclusters.

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.

2.8 PHYSICALARCHITECTURE

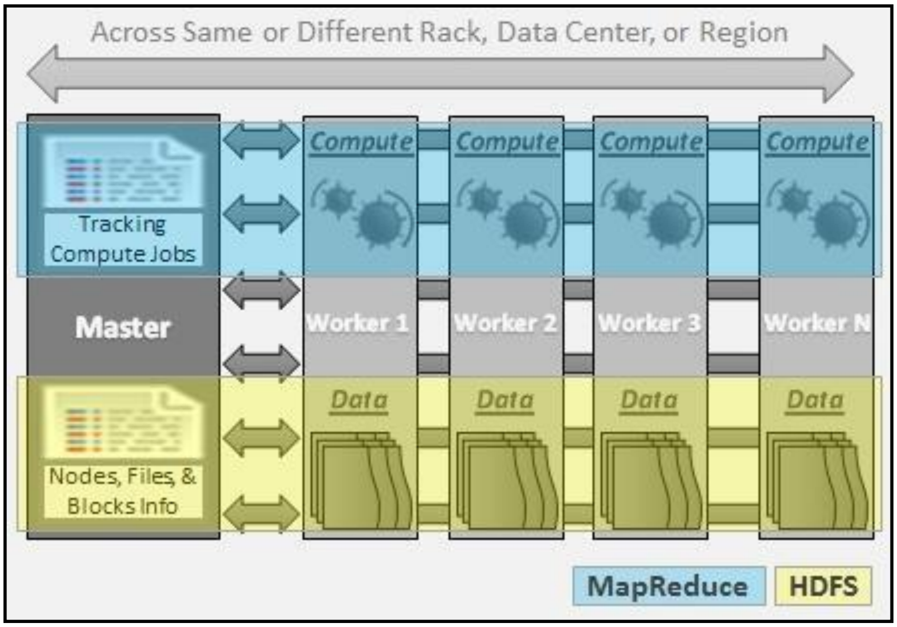


Figure 2.2: Physical Architecture

Hadoop Cluster - Architecture, Core Components and Work-flow

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

HADOOP CLUSTER

i. Hadoopclusterisaspecialtypeofcomputationalclusterdesignedforstoringand analyzing vast amount of unstructured data in a distributed computing environment

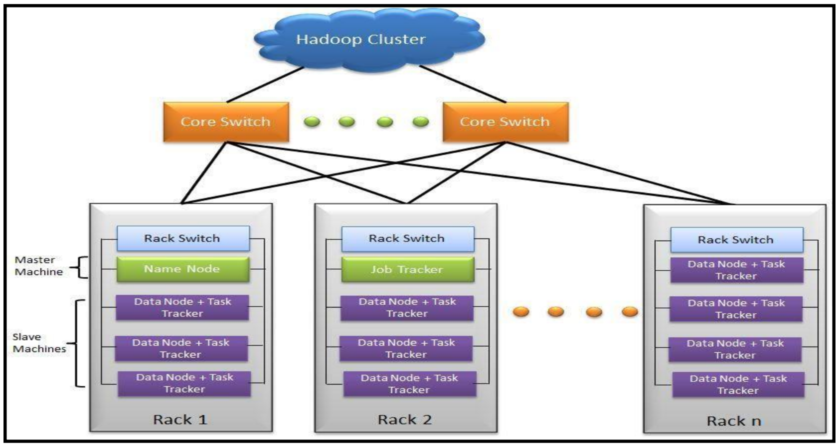


Figure 2.3: Hadoop Cluster

i. These clusters run on lowcost commodity computers.

ii. 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.

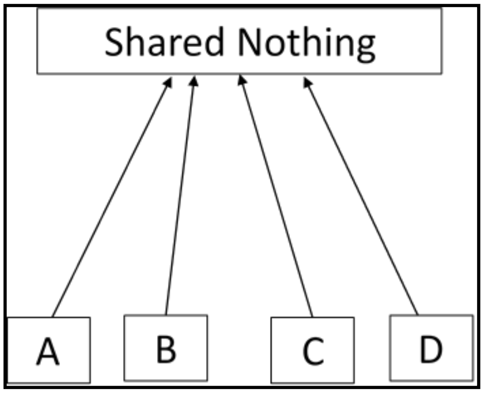


Figure 2.4: Shared Nothing

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

- AsmallHadoopclusterincludesasinglemasternodeandmultipleworkerorslave node. As discussed earlier, the entire cluster contains twolayers.
- One of the layer of MapReduce Layer and another is of HDFSLayer.
- Each of these layer have its own relevant component.
- The master node consists of a JobTracker,TaskTracker, NameNodeandDataNode.
- A slave or worker node consists of a Data Node and TaskTracker.

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

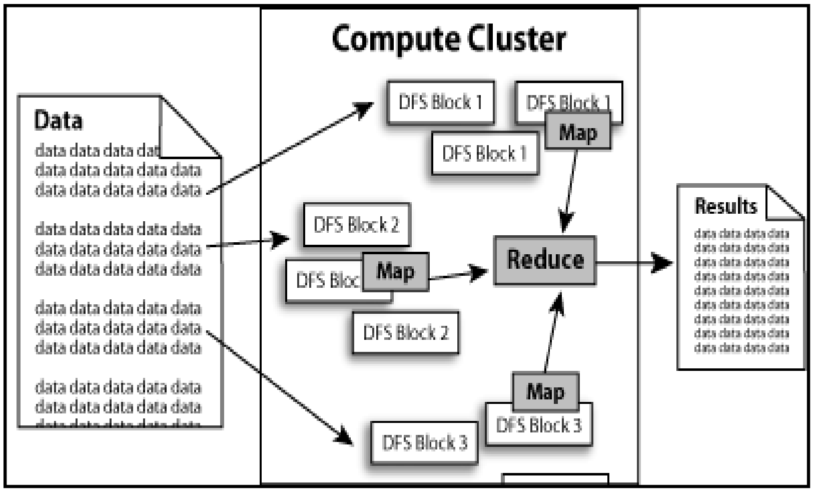


Figure 2.4: NameNode Cluster

HADOOP CLUSTER ARCHITECTURE:

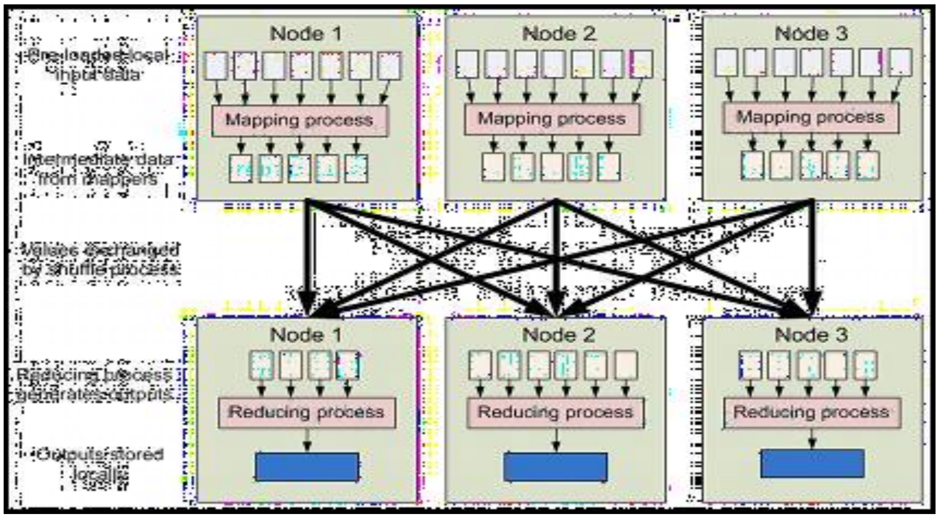


Figure 2.5: Hadoop Cluster Architecture Hadoop Cluster would consists of

- 110 differentracks
- Each rack would have around 40 slavemachine
- At the top of each rack there is a rackswitch
- Each slave machine(rack server in a rack) has cables coming out it from both theends
- Cables are connected to rack switch at the top which means that top rack switch will have around 80ports
- There are global 8 core switches
- The rack switch has uplinks connected to core switches and hence connecting all other racks with uniform bandwidth, forming the Cluster

- Inthecluster, you have few machines to act as Namenode and as JobTracker. They are referred as Masters. These masters have different configuration favoring more DRAM and CPU and less localstorage.

Hadoop cluster has 3 components:

1. Client
2. Master
3. Slave

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

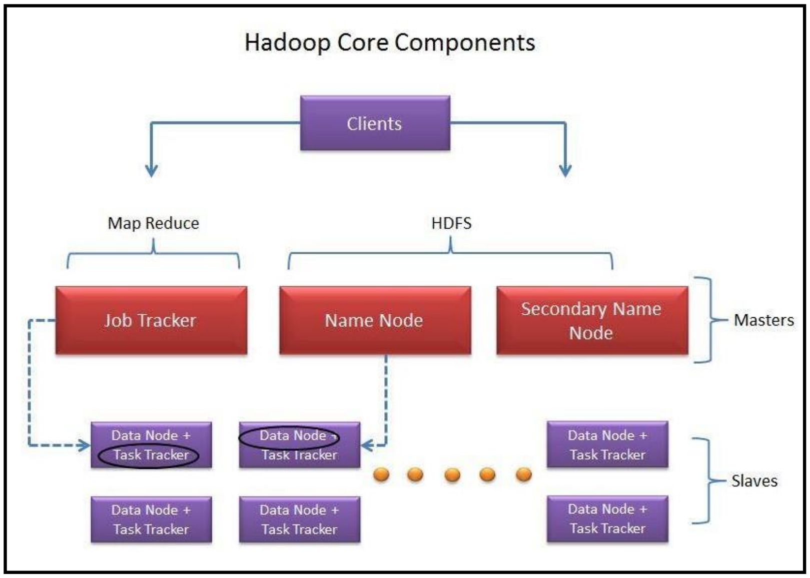


Figure 2.6: Hadoop Core Component

Client:

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 see the response after job completion.

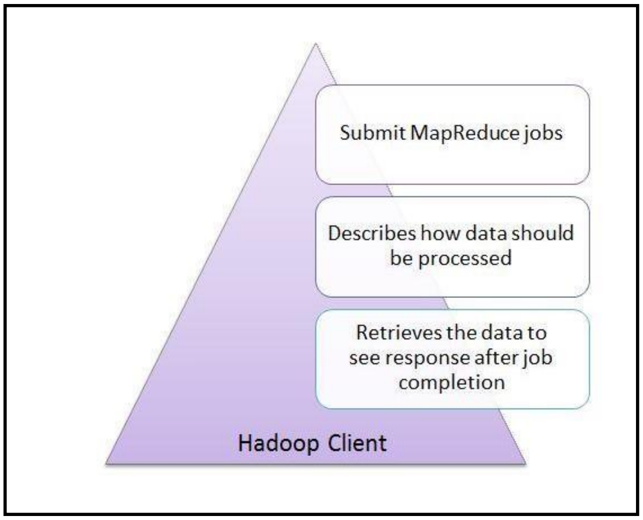

Submit MapReduce jobs
Describes how data should be processed
Retrieves the data to see response after job completion
Hadoop Client

Figure 2.6: Hadoop Client

Masters:

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

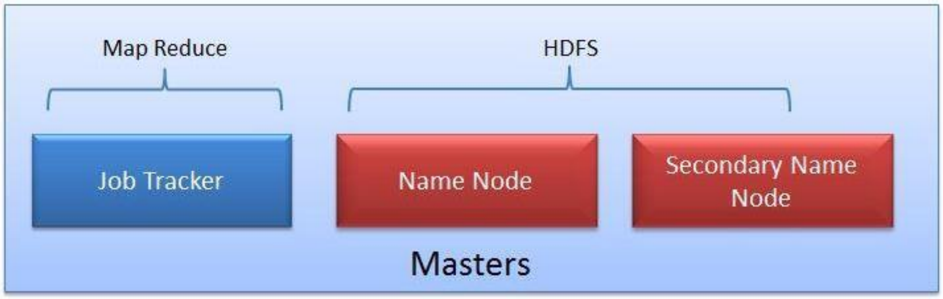


Figure 2.7: MapReduce - HDFS

NameNode:

NameNode does NOT store the files but only the file's metadata. In later section we will see it is actually the Data Node which stores the files.

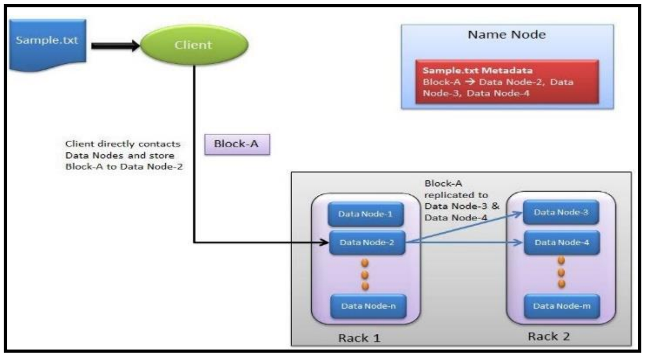

Sample.txt
Client
Name Node
Sample.txt Metadata
Block-A → Data Node-2, Data Node-3, Data Node-4
Client directly contacts Data Nodes and store Block-A to Data Node-2
Block-A
Block-A replicated to Data Node-3 & Data Node-4
Data Node-1
Data Node-2
Data Node-n
Data Node-3
Data Node-4
Data Node-m
Rack 1
Rack 2

Figure 2.8: NameNode

- NameNode oversees the health of DataNode and coordinates access to the data stored in DataNode.

- Name node keeps track of all the file system related information such asto
- Which section of file is saved in which part of thecluster
- Last access time for the files
- User permissions like which user have access to the file

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)

Secondary NameNode

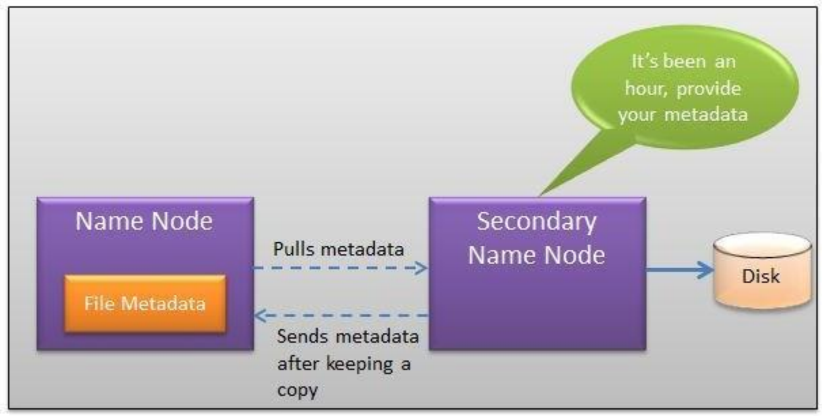

It's been an hour, provide your metadata
Name Node
File Metadata
Pulls metadata
Sends metadata after keeping a copy
Secondary Name Node
Disk

Figure 2.9: Secondary NameNode

- The job of Secondary Node is to contact NameNode in a periodic manner after certain time interval (by default 1hour).
- NameNode which keeps all filesystem metadata in RAM has no capability to process that metadata 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 metadata information out ofNameNode.
- It shuffle and merge this information into clean file folder and sent to back again to NameNode, while keeping a copy foritself.
- Hence Secondary Node is not the backup rather it does job of housekeeping.
- In case of NameNode failure, saved metadata can rebuild it easily.

Slaves:

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

- Store the data
- Process the computation
- Each slave runs both a DataNode and Task Tracker daemon which communicates to their masters.
- The Task Tracker daemon is a slave to the Job Tracker and the DataNode daemon a slave to the NameNode

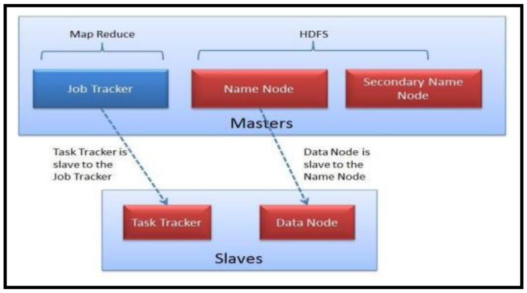


Figure 2.10: Slaves

Hadoop- Typical Workflow in HDFS: Take the example of input file as Sample.txt.

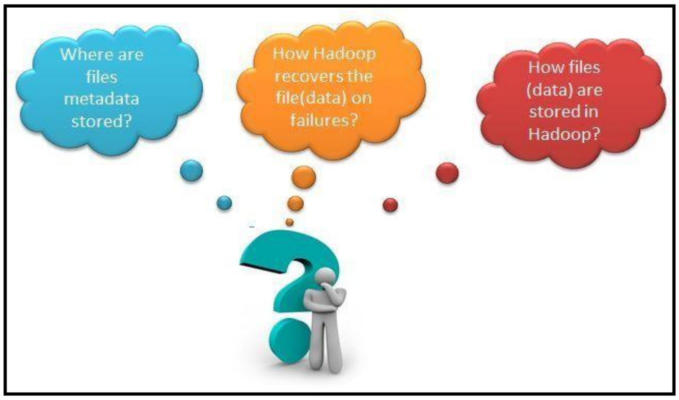

Where are files metadata stored?
How Hadoop recovers the file(data) on failures?
How files (data) are stored in Hadoop?

Figure 2.11: HDFS Workflow

1. ow TestingHadoop.txt gets loaded into the HadoopCluster?

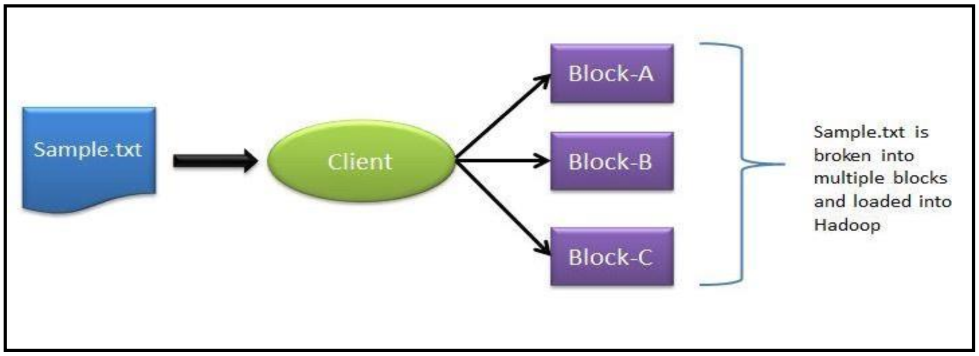

Sample.txt
Client
Block-A
Block-B
Block-C
Sample.txt is broken into multiple blocks and loaded into Hadoop

Figure 2.12: Loading file in Hadoop Cluster

- Client machine does this step and loads the Sample.txt intocluster.
- It breaks the sample.txt into smaller chunks which are known as "Blocks" in Hadoopcontext.
- Clientputtheseblocksondifferentmachines(datanodes)throughoutthecluster.

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

- Now NameNode comes intopicture.
- The NameNode used its Rack Awareness intelligence to decide on which DataNodetoprovide.
- 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 andDataNode-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 of machine 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 to other Datanodes.
- When one block gets written in all 3 DataNode then only cycle repeats for next block.

5. Who does the block replication?

- In Hadoop Gen1 there is only one Name Node where in Gen2 there is active passive model in Name Node 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 can be configured with "dfs.replication" parameter of hdfs-site.xml file.
- Keep note that Client directly writes the block to the DataNode without any intervention of NameNode in this process.

Hadoop limitations

- Network File system is the oldest and the most commonly used distributed file system and was designed for the general class of applications, Hadoop only specific kind of applications can make use of it.
- 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 fast access, but it should be known that the benefits that come with Hadoop come at some cost.
- Hadoop is designed for applications that require random reads; So if a file has four parts the file would like to read all the parts one-by-one going from 1 to 4 till the end. Random seek is where you want to go to a specific location in the file; this is something that isn't possible with Hadoop. Hence, Hadoop is designed for non- real-time batch processing of data.
- 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 is n't available through Hadoop.
- 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, youhavetoknowthatinupdate0.19appendingwillbe supported for those files that aren't closed. But for those files that have been closed, updating is n't possible.
- 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.

UNIT-III

THE HADOOP DISTRIBUTED FILESYSTEM

Topics

- ✓ The Design of HDFS
- ✓ HDFS Concepts
- ✓ Basic Filesystem Operations
- ✓ Hadoop Filesystems
- ✓ The Java Interface
- ✓ Reading Data from a Hadoop URL
- ✓ Reading Data Using the Filesystem API
- ✓ Writing Data
- ✓ Data Flow- Anatomy of a File Read
- ✓ Anatomy of a File Write
- ✓ Limitations.

When a data set out grows the storage capacity of a single physical machine, it becomes necessary to partition it a cross a number of separate machines. File systems 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 with out suffering data loss.

Hadoop comes with a distributed filesystem called HDFS, which stands for Hadoop Distributed Filesystem. (You may some times see references to “DFS”—informally or in older documentation or configurations—which is the same thing.) HDFS is Hadoop’s flag ship file system and is the focus of this chapter, but Hadoop actually has a general- purpose file system abstraction, so we’ll see along the way how Hadoop integrates with other storage systems(such as the local file system and AmazonS3).

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. Let’s examine this statement in more detail:

VERY LARGEFILES

“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) 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 areas where HDFS is not a goodfit today:

LOW-LATENCY DATA ACCESS

Applications that require low-latency access to data, in the tens of milliseconds range, will notwork 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 meta data 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 300MB of memory. While storing millions of files is feasible, billions is beyond the capability of current hardware.

MULTIPLE WRITERS, ARBITRARY FILE MODIFICATIONS

FilesinHDFSmaybewrittentobyasinglewriter.Writesarealwaysmadeatthe 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 relativelyinefficient.)

HDFS CONCEPTS

BLOCKS

A disk has a blocksize, which is the minimum amount of data that it can read or write. File systems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk blocksize. 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—64MB by default. Like in a file system 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 a block abstraction for a distributed filesystem brings several benefits. The first 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 best or edon 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 file on 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 in formation does not need to be stored with the blocks, so another system can handle meta data 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 thecluster.

Like its diskfilesystem 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 theformoftwofiles:thenamespaceimageandtheeditlog.Thenamenodealsoknows 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.

Aclientaccessesthefilesystemonbehalfoftheuserbycommunicatingwiththename- 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 configuration choice is to write to local disk as well as a remote NFSmount.

It is also possible to run a secondary namenode, which despiteits 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 newprimary.

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 filesrooted 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 them etadata for the name space, 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 namespace managed by other namenodes. Block pool storage is not partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple blockpools.

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

Thecombinationofreplicatingnamenodemetadataonmultiplefilesystems,andusing the secondary namenode to create checkpoints protects against data loss, but does not providehigh-availabilityofthefilesystem.Thenamenodeisstillasinglepointoffail- ure (SPOF), since if it did fail, all clients—including MapReduce jobs—would beun- 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 broughtonline.

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 interruption. 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 NFSfiler, 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 in a namenode's memory, and not ondisk.

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 even to best and by 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 operational 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 controller arranges an orderly transition for both namenodes to switch roles.

In the case of anungraceful 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 arrange offen cing 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 graphically known as STONITH, or "shoot the other node in the head", which uses a specialized power distribution unit to forcibly power down the hostmachine.

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 host name which is mapped to a pair of name node 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 directories. You can type **Hadoop fs -help** to get detailed help on everycommand.

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 sub commands—in this case, we are running - copyFromLocal. Thelocalfile quangle.txt is copied to the file/user/tom/quangle.txtontheHDFS instance running on localhost. Infact, we could have omitted the scheme and host of the URI and picked up the default, hdfs://localhost, as specified incore-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:

```
% hadoopfs-mkdir books
% hadoopfs -ls .
Found 2 items
drwxr-xr-x-tomsupergroup          0          2009-04-02          22:41/user/tom/books-rw-r--r--
1tomsupergroup2009-04-02 22:29/user/tom/quangle.txt
```

The information returned is very similar to the Unix command ls -l, with a few min or differences. The first column shows the filemode.

The second column is the replication factor of the file (something a traditional Unix filesystem does not have).

The entry in this column is empty for directories in 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 column is 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 ordirectory

HADOOP FILESYSTEMS

Hadoop has an abstract notion of filesystem, of which HDFS is just one implementation. The Java abstract class org.apache.hadoop.fs.FileSystem 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:

```
% hadoopfs -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 JavaAPI. 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 underlying filesystem (FTP clients for FTP, S3tools 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 aproxy (or proxies),which accesses HDFS on the client’s behalf using the usual DistributedFileSystem API.

In the firstcase, directory listings are served by the namenode’s embedded webserver (which runs on port 50070) formatted in XML or JSON, while file data is streamed from datanodes by their web servers (running on port50075).

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

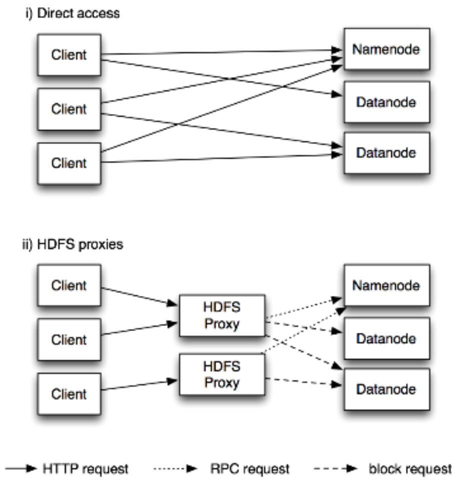


Figure 3-1. Accessing HDFS over HTTP directly, and via a bank of HDFS proxies

The second way of accessing HDFS over HTTP relies on one or more standalone proxy servers.(The proxies are state less 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 datacenters.

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- lease0.23, there is a new proxy called HttpFS that has read and write capabilities, and which exposes the same HTTP interfaceas WebHDFS, soclients can access either rusing web hdfs URIs.

The HTTP REST API that Web HDFS 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.

FUSE

FilesysteminUserspace(FUSE)allowsfilesystemsthatareimplementedinuserspace 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-DFSisimplementedinCusinglibhdfsastheinterfacetoHDFS.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 hdfsURLscheme. This is achieved by calling the setURLStreamHandlerFactory method on URL

1. Fromrelease0.21.0, there is a new file system interface called File Context with better handling of multiple filesystems (so a single File Context can resolve multiple file system schemes, for example) and a cleaner, more consistent interface.

2. withaninstanceofFsUrlStreamHandlerFactory.Thismethodcanonlybecalledonce perJVM, 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.

Program for displaying files from Hadoop file systems 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{
static {
URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
}
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);
}
}
```

READING DATA USING THE FILESYSTEM API

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

A file in a Hadoop filesystem is represented by a Hadoop Path object (and not a java.io.Fileobject, 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 filesystemwewanttouse—HDFSinthiscase.Thereareseveralstaticfactorymethods for getting a FileSysteminstance:

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

A Configuration object encapsulates a clientorserver's configuration, which is set using configuration files read from the classpath, such as conf/core-site.xml.Thefirstmethod 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 file system 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.

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);
}
}
}
```

FSDataInputStream

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 position in 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:

FSDataInputStreamalsoimplementsthePositionedReadableinterfaceforreadingparts ofafileatagivenoffset:

```
public interface PositionedReadable{
public int read(long position, byte[] buffer, int offset, int length) throwsIOException;
public void readFully(long position, byte[] buffer, int offset, int length) throwsIOException;
public void readFully(long position, byte[] buffer) throws IOException;
}
```

All of these methods preserve the current offset in the file and are thread-safe, so they provideaconvenientwaytoaccessanotherpartofthefile—metadataperhaps—while reading the main body of the file. In fact, they are just implemented usingtheSeekable interface using the followingpattern:

```
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 FileSystem 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 FSDataOutputStreamcreate(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 FSDataOutputStreamappend(Path f) throws IOException
```

Theappendoperationallowsasinglewritertomodifyanalreadywrittenfilebyopening 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 filesystemsdon’t.

TocopyalocalfiletoaHadoopfilesystem.Weillustratepro- 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();
FileSystemfs= 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:

```
% hadoopFileCopyWithProgress 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.

The Java Interface

The Hadoop's FileSystem class: the API for interacting with one of Hadoop's filesystems.

The focus on HDFS implementation, DistributedFileSystem, in general to write 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);
}
```

To make Java recognize Hadoop's hdfs URL scheme

By calling the setURLStreamHandlerFactory method on URL with an instance of FsUrlStreamHandlerFactory.

This method can only be called once per JVM, so it is typically executed in a static block.

IOUtils class that comes with Hadoop for closing the stream 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.

Example: Displaying files from a Hadoop filesystem on standard output using a URLStreamHandler

```
public class URLCat {
 static {
 URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
 }
 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);
          }
 }
}
```

A sample run:

```
% hadoop URLCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

Reading Data Using the Filesystem API

To set a **URLStreamHandlerFactory** for your application.

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).

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 - 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 a file:

```
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.

Example: 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);
}
}
}
```

The program runs as follows:

```
% hadoop FileSystemCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

FSDataInputStream

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, absolute position in the file.

Example: 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);
}
}
}
```

The result of running it on a small file:

```
% hadoop FileSystemDoubleCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
```

On account of his Beaver Hat.

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.

DATA FLOW

ANATOMY OF A FILE READ

TogetanideaofhowdataflowsbetweentheclientinteractingwithHDFS,thenamenodeandthedatanodes,whichshowsthemainsequenceofevents when reading afile.

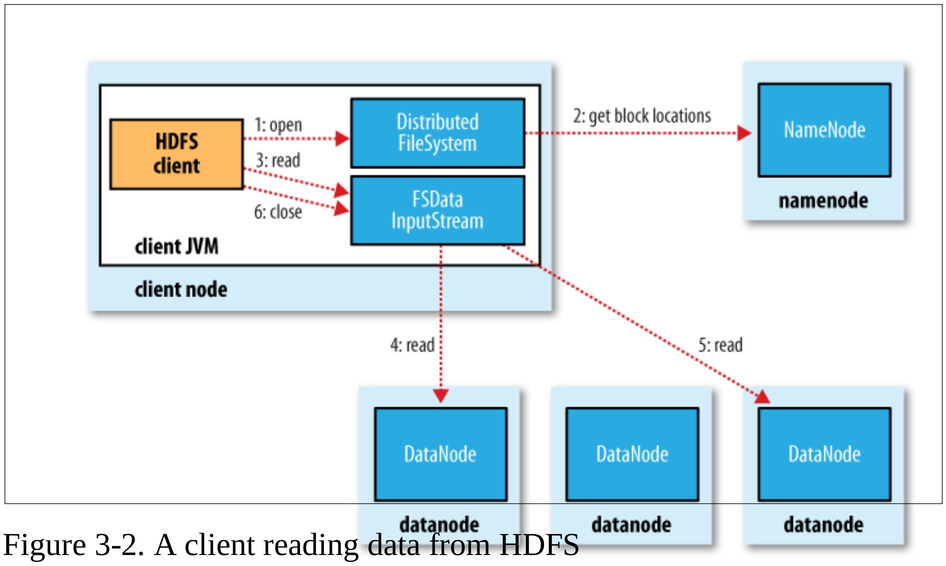


Figure 3-2. A client reading data from HDFS

ANATOMY OF A FILE WRITE

Nextwe’lllookathowfilesarewrittentoHDFS.Althoughquitedetailed,itisinstructive to understand the data flow since it clarifies HDFS’s coherencymodel.

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

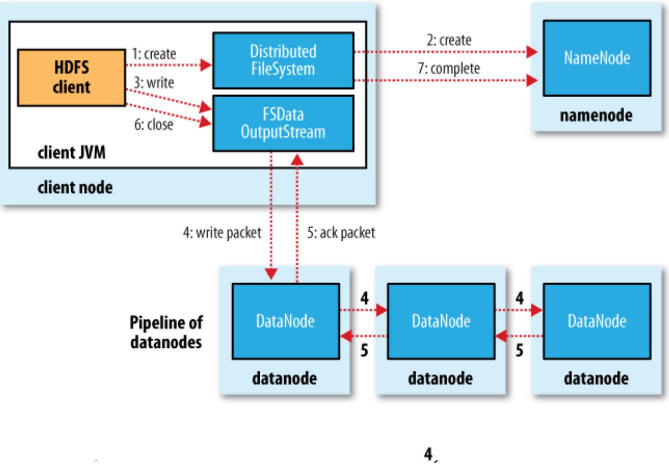
The client creates the file by calling create() on Distributed Filesystem (step 1 in Distributed FilesystemmakesanRPCcalltothenamenodetocreateanew fileinthefilesystem’snamespace,withnoblocksassociatedwithit(step2).Thename-nodeperformsvariouscheckstomakesurethefiledoesn’talreadyexist,andthatthe clienthastherightpermissionstocreatethefile.Ifthesecheckspass,thename node makesarecordofthenewfile;otherwise,filecreationfailsandtheclientisthrownan IOException. The Distributed Filesystem returns an FS Data Output Stream for theclient to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutput Stream, which handles communication with the datanodes and namenode.

Astheclientwritesdata(step3),DFSOutputStreamsplitsitintopackets,whichitwrites to an internal queue, called the data queue. The data queue is consumed by the Data Streamer, whose responsibility it is to ask the namenode 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 DataStreamer 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 (step4).

DFSOutputStream 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 queueonlywhenithasbeenacknowledgedbyallthedatanodesinthepipeline(step5).

Ifadatanodefailswhiledataisbeingwrittentoit,thenthefollowingactionsaretaken, whicharetransparenttotheclientwritingthedata.Firstthepipelineisclosed,andany packets in the ack queue are added to the front of the data queue so that datanodes thataredownstreamfromthefailednodewillnotmissanypackets.Thecurrentblock on the good datanodes is given a new identity, which is communicated to the name-node,sothatthepartialblockonthefaileddatanodewillbedeletedifthefailed.

Figure 3-4. A clier



datanode recove d from the pipeline and the remainderofthebl eline.The namenodenotices rtherreplica to be created on another node. Sul

It'spossible,butu eingwritten.

Aslongasdfs.repl itewillsucceed, and the block will be a synch star get replication factor is reached (dfs.repl

When the client has finished writing data, it calls close() on the stream(step6).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 Stream erasking for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

LIMITATIONS

Thereareafewlimitationstobeawareofwith HARfiles.Creatinganarchivecreates 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 recreate 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 dail your 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 FileInputFormat” discusses another approach to this problem.

Finally, if you are hitting name node 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

UNIT IV

UNDERSTANDING MAPREDUCE FUNDAMENTALS

Topics:

- ✓ **Map Reduce Framework: Exploring the features of Map Reduce**
- ✓ **Working of Map Reduce**
- ✓ **Exploring Map and Reduce Functions**
- ✓ **Techniques to optimize Map Reduce jobs**
- ✓ **Uses of Map Reduce**
- ✓ **Controlling MapReduce Execution with InputFormat**
- ✓ **Reading Data with custom RecordReader**
- ✓ **Reader, Writer, Combiner, Partitioners**
- ✓ **Map Reduce Phases**
- ✓ **Developing simple MapReduce Application**

MapReduce Features

This chapter looks at some of the more advanced features of MapReduce, including counters and sorting and joining datasets.

Counters

There are often things you would like to know about the data you are analyzing but that are peripheral to the analysis you are performing. For example, if you were counting invalid records and discovered that the proportion of invalid records in the whole dataset was very high, you might be prompted to check why so many records were being marked as invalid—perhaps there is a bug in the part of the program that detects invalid records? Or if the data were of poor quality and genuinely did have very many invalid records, after discovering this, you might decide to increase the size of the dataset so that the number of good records was large enough for meaningful analysis.

Counters are a useful channel for gathering statistics about the job: for quality control or for application-level statistics. They are also useful for problem diagnosis. If you are tempted to put a log message into your map or reduce task, it is often better to see whether you can use a counter instead to record that a particular condition occurred. In addition to counter values being much easier to retrieve than log output for large distributed jobs, you get a record of the number of times that condition occurred, which is more work to obtain from a set of logfiles.

Sorting

The ability to sort data is at the heart of MapReduce. Even if your application isn't concerned with sorting per se, it may be able to use the sorting stage that MapReduce provides to organize its data.

Joins

MapReduce can perform joins between large datasets but writing the code to do joins from scratch is fairly involved. Rather than writing MapReduce programs, you might consider using a higher-level framework such as Pig, Hive, or Cascading, in which join operations are a core part of the implementation.

Let's briefly consider the problem we are trying to solve. We have two datasets—for example, the weather stations database and the weather records—and we want to reconcile the two. Let's say we want to see each station's history, with the station's metadata inlined in each output row. This is illustrated in Figure 8-2.

How we implement the join depends on how large the datasets are and how they are partitioned. If one dataset is large (the weather records) but the other one is small enough to be distributed to each node in the cluster (as the station metadata is), the join can be effected by a MapReduce job that brings the records for each station together (a partial sort on station ID, for example). The mapper or reducer uses the smaller dataset to look up the station metadata for a station ID, so it can be written out with each record. See Side Data Distribution for a discussion of this approach, where we focus on the mechanics of distributing the data to tasktrackers.

If the join is performed by the mapper, it is called a *map-side join*, whereas if it is performed by the reducer it is called a *reduce-side join*.

If both datasets are too large for either to be copied to each node in the cluster, we can still join them using MapReduce with a map-side or reduce-side join, depending on how the data is structured. One common example of this case is a user database and a log of some user activity (such as access logs). For a popular service, it is not feasible to distribute the user database (or the logs) to all the MapReduce nodes.

MapReduce Working

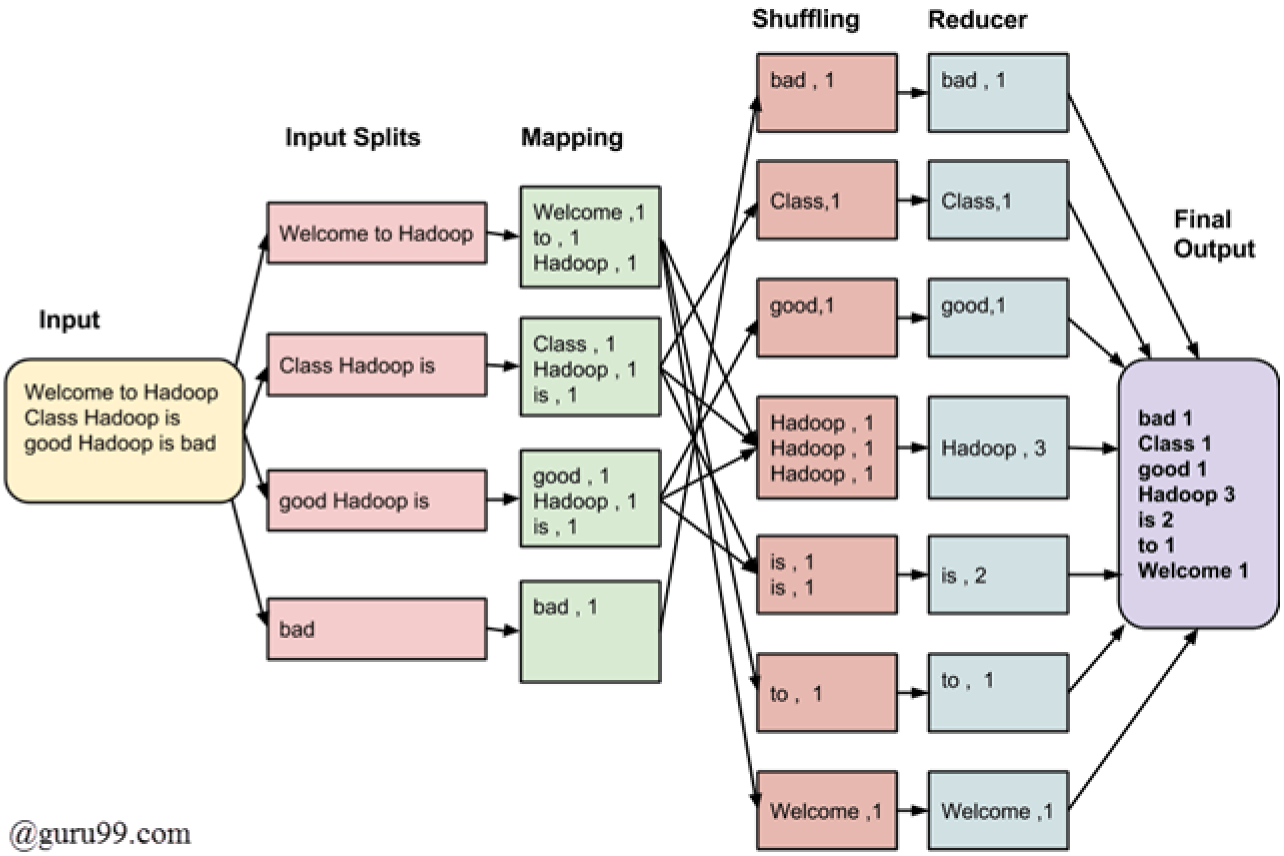
The whole process goes through four phases of execution namely, splitting, mapping, shuffling, and reducing.

Consider you have following input data for your Map Reduce Program

Welcome to Hadoop Class

Hadoop is good

Hadoop is bad



MapReduce Architecture

The final output of the MapReduce task is

bad	1
Class	1
good	1
Hadoop	3
is	2
to	1
Welcome	1

The data goes through the following phases

Input Splits:

An input to a MapReduce job is divided into fixed-size pieces called input splits Input split is a chunk of the input that is consumed by a single map

Mapping

This is the very first phase in the execution of map-reduce program. In this phase data in each split is passed to a mapping function to produce output values. In our example, a job of mapping phase is to count a number of occurrences of each word from input splits (more details about input-split is given below) and prepare a list in the form of <word, frequency>

Shuffling

This phase consumes the output of Mapping phase. Its task is to consolidate the relevant records from Mapping phase output. In our example, the same words are clubed together along with their respective frequency.

Reducing

In this phase, output values from the Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset. In our example, this phase aggregates the values from Shuffling phase i.e., calculates total occurrences of each word.

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. Traditional model 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 files simultaneously.

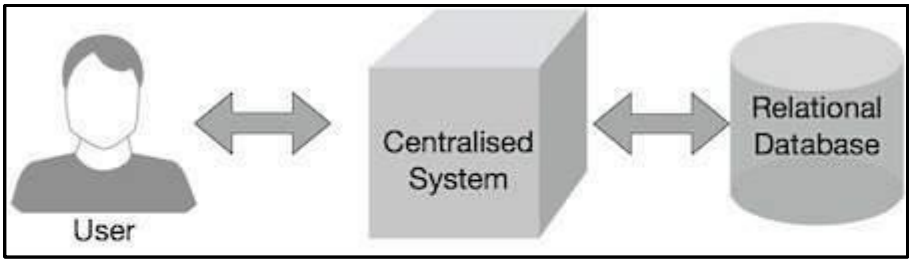


Figure 4.1: MapReduce

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.

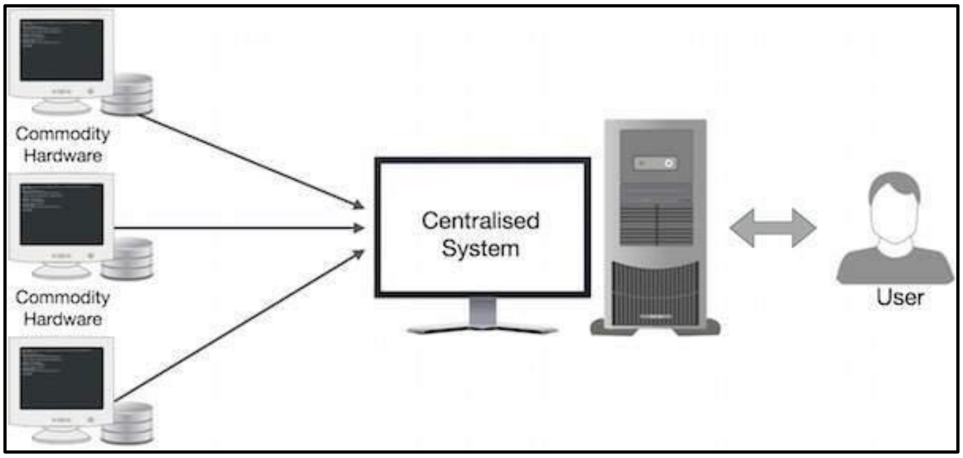


Figure 4.2: Physical structure

1. 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 produced from 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. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task.
- ✓ The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way. The manner of combination of values is determined by the code written by the user for the Reduce function.

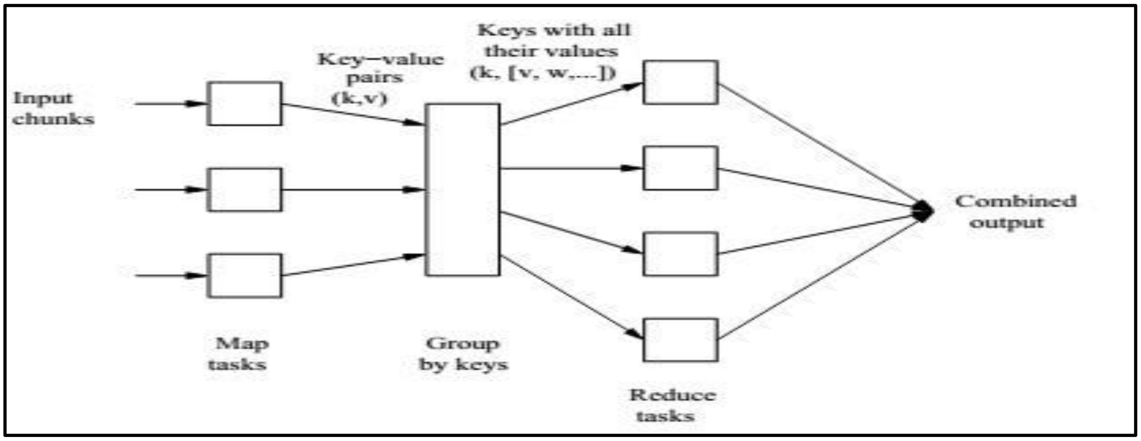


Figure 4.3: Schematic MapReduce Computation

A. The Map Task

1. We view input files for a Map task as consisting of elements, which can be any type: a tuple or a document, for example.
2. A chunk is a collection of elements, and no element is stored across two chunks.
3. 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.
4. Insisting on this form for inputs and outputs is motivated by the desire to allow composition of several MapReduce processes.
5. The Map function takes an input element as its argument and produces zero or more key-value pairs.
6. The types of keys and values are each arbitrary. vii. Further, keys are not "keys" in the usual sense; they do not have to be unique.
7. Rather a Map task can produce several key-value pairs with the same key, even from the same element.

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, the input file is a repository of documents, and each document is an element. The Map function for this example uses keys that are of type String (the words) and values that are integers. The Map task reads a 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:

$(w_1, 1), (w_2, 1), . . . , (w_n, 1)$

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 the values 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 merges the 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 key k 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 of values 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 are merged into a single file.

vi. Reducers may be partitioned among a smaller number of Reduce tasks is by hashing the keys and associating 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 and the 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 of w 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 sum of 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 elements are broken down into tuples (key-value pairs).

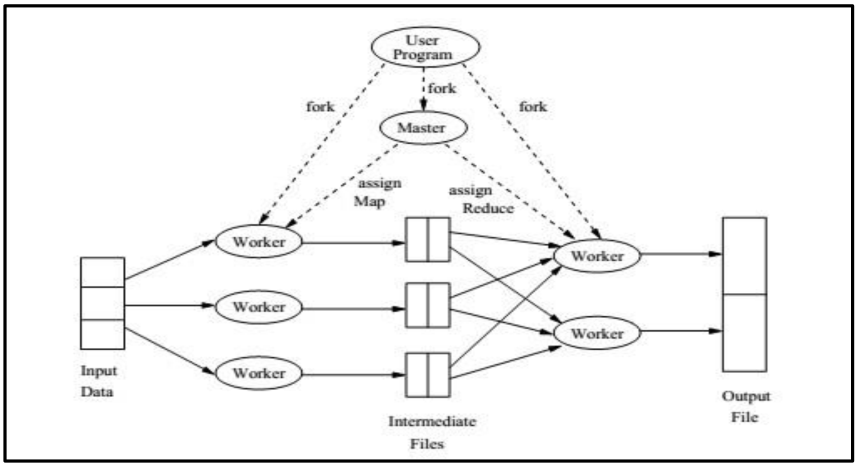


Figure 4.4: Overview of the execution of a MapReduce program

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.

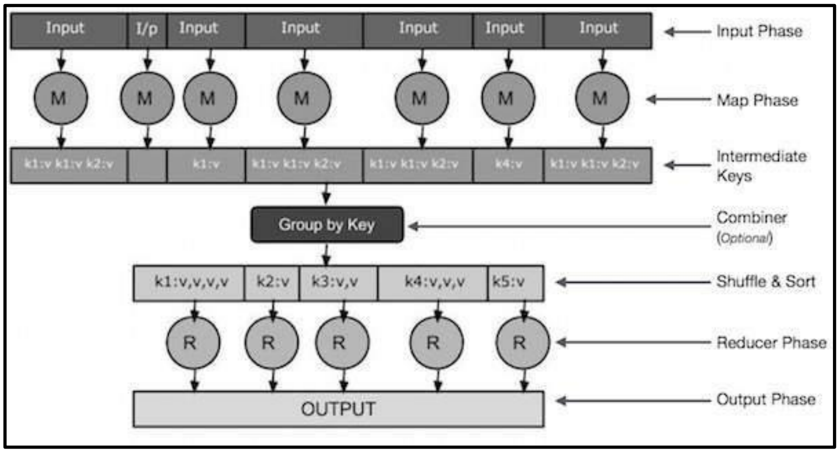


Figure 4.5: Reduce job

- **Input Phase** − Here we have a Record Reader that translates each record in an input file and sends the 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 each one 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 MapReduce algorithm; 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 their values 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-value pairs from the Reducer function and writes them onto a file using a record writer.

iv. The MapReduce phase

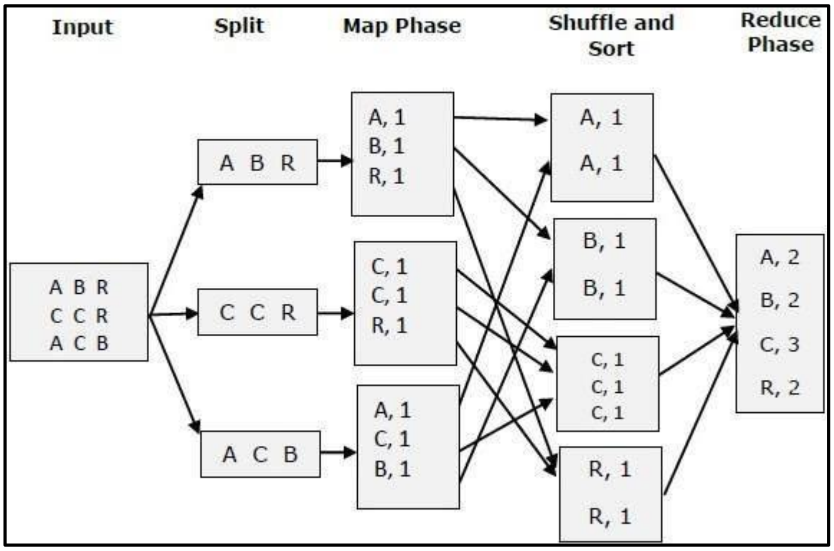


Figure 46 The MapReduce Phase

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.

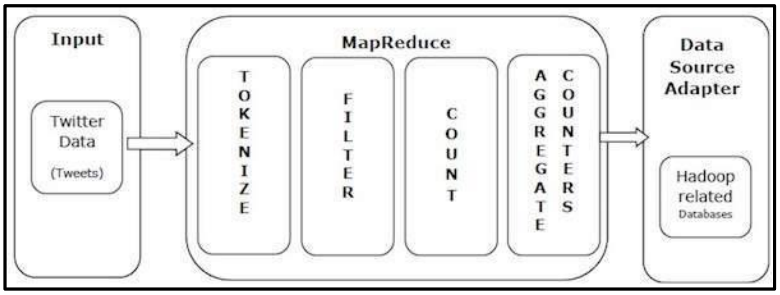


Figure4.7: Example

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

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

Count − Generates a token counter per word.

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.

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.

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.

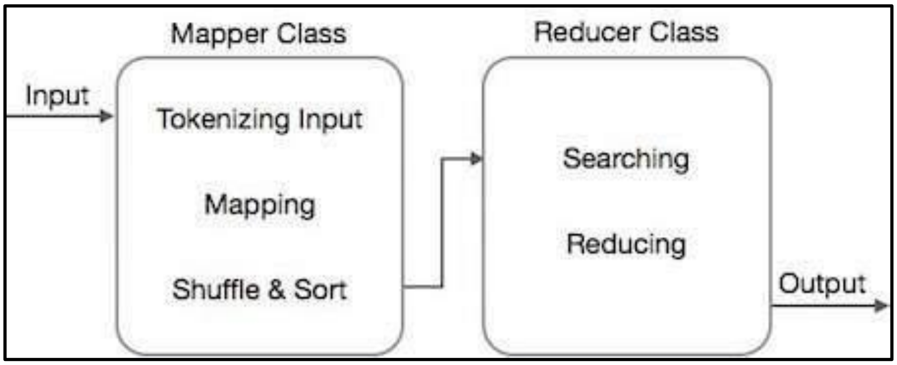


Figure 4.8: The MapReduce Class

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 the Master, 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.

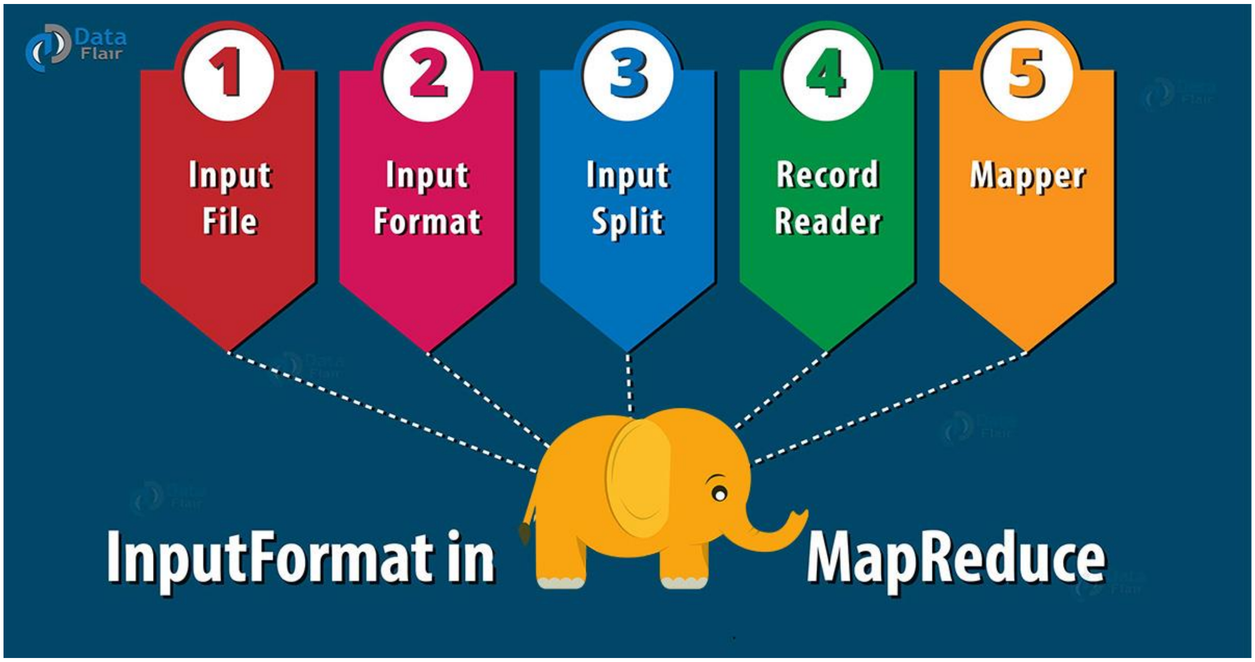
v. The Master sets the status of each of these Map tasks to idle and will schedule them on a Worker when one becomes available.

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 be rescheduled on another reduce worker later.

Controlling MapReduce Execution with InputFormat

Hadoop InputFormat checks the Input-Specification of the job. InputFormat split the Input file into InputSplit and assign to individual Mapper.
Will learn
what is InputFormat in Hadoop **MapReduce**,
different methods to get the data to the mapper and
different types of InputFormat in Hadoop like FileInputFormat in Hadoop, TextInputFormat, KeyValueTextInputFormat, etc.



How the input files are split up and read in Hadoop is defined by the InputFormat.

An Hadoop InputFormat is the first component in Map-Reduce, it is responsible for creating the input splits and dividing them into records.
Initially, the data for a MapReduce task is stored in input files, and input files typically reside in **HDFS**. Although these files format is arbitrary, line-based log files and binary format can be used. Using InputFormat we define how these input files are split and read.

The InputFormat class is one of the fundamental classes in the Hadoop MapReduce framework which provides the following functionality:

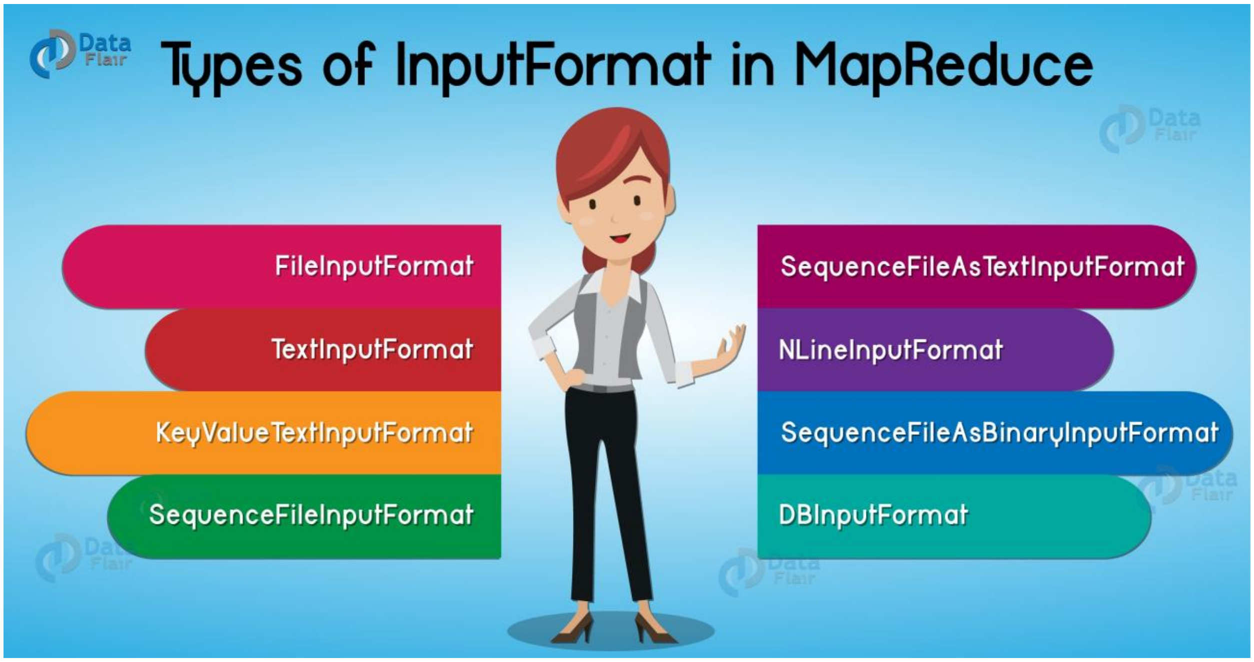
- The files or other objects that should be used for input is selected by the InputFormat.
- InputFormat defines the Data splits, which defines both the size of individual **Map tasks** and its potential execution server.
- InputFormat defines the **RecordReader**, which is responsible for reading actual records from the input files.

3. How we get the data to mapper?

We have 2 methods to get the data to **mapper** in MapReduce: getsplits() and createRecordReader() as shown below:

```
public abstract class InputFormat<K, V>
{
public abstract List<InputSplit> getSplits(JobContext context)
throws IOException, InterruptedException;
public abstract RecordReader<K, V>
createRecordReader(InputSplit split,
TaskAttemptContext context) throws IOException,
InterruptedException;
}
```

4. Types of InputFormat in MapReduce



FileInputFormat in Hadoop

It is the base class for all file-based InputFormats. Hadoop FileInputFormat specifies input directory where data files are located. When we start a Hadoop job, FileInputFormat is provided with a path containing files to read. FileInputFormat will read all files and divides these files into one or more InputSplits.

TextInputFormat

It is the default InputFormat of MapReduce. TextInputFormat treats each line of each input file as a separate record and performs no parsing. This is useful for unformatted data or line-based records like log files.

- **Key –** It is the byte offset of the beginning of the line within the file (not whole file just one split), so it will be unique if combined with the file name.
- **Value –** It is the contents of the line, excluding line terminators.

KeyValueTextInputFormat

It is similar to TextInputFormat as it also treats each line of input as a separate record. While TextInputFormat treats entire line as the value, but the KeyValueTextInputFormat breaks the line itself into key and value by a tab character (‘/t’). Here Key is everything up to the tab character while the value is the remaining part of the line after tab character.

SequenceFileInputFormat

Hadoop **SequenceFileInputForma**t is an InputFormat which reads sequence files. Sequence files are binary files that stores sequences of binary **key-value pairs**. Sequence files block-compress and provide direct serialization and deserialization of several arbitrary data types (not just text). Here Key & Value both are user-defined.

SequenceFileAsTextInputFormat

Hadoop **SequenceFileAsTextInputFormat** is another form of SequenceFileInputFormat which converts the sequence file key values to Text objects. By calling **‘tostring()’** conversion is performed on the keys and values. This InputFormat makes sequence files suitable input for streaming.

SequenceFileAsBinaryInputFormat

Hadoop **SequenceFileAsBinaryInputFormat** is a SequenceFileInputFormat using which we can extract the sequence file’s keys and values as an opaque binary object.

NLineInputFormat

Hadoop **NLineInputFormat** is another form of TextInputFormat where the keys are byte offset of the line and values are contents of the line. Each mapper receives a variable number of lines of input with TextInputFormat and KeyValueTextInputFormat and the number depends on the size of the split and the length of the lines. And if we want our mapper to receive a fixed number of lines of input, then we use NLineInputFormat.

N is the number of lines of input that each mapper receives. By default (N=1), each mapper receives exactly one line of input. If N=2, then each split contains two lines. One mapper will receive the first two Key-Value pairs and another mapper will receive the second two key-value pairs.

DBInputFormat

Hadoop **DBInputFormat** is an InputFormat that reads data from a relational database, using JDBC. As it doesn't have portioning capabilities, so we need to careful not to swamp the database from which we are reading too many mappers. So it is best for loading relatively small datasets, perhaps for joining with large datasets from HDFS using MultipleInputs. Here Key is LongWritables while Value is DBWritables.

UNIT V

INTRODUCTION TO PIG AND HIVE

Topics:

- ✓ **Introducing Pig: Pig architecture**
- ✓ **Benefits**
- ✓ **Installing Pig**
- ✓ **Properties of Pig**
- ✓ **Running Pig**
- ✓ **Getting started with Pig Latin**
- ✓ **Working with operators in Pig**
- ✓ **Working with functions in Pig.**
- ✓ **Introducing Hive: Getting started with Hive**
- ✓ **Hive Services**
- ✓ **Data types in Hive**
- ✓ **Built-in functions in Hive**
- ✓ **Hive DDL.**

What is Apache Pig?

Apache Pig is an abstraction over MapReduce. It is a tool/platform which is used to analyze larger sets of data representing them as data flows. Pig is generally used with **Hadoop**; we can perform all the data manipulation operations in Hadoop using Apache Pig.

To write data analysis programs, Pig provides a high-level language known as **Pig Latin**. This language provides various operators using which programmers can develop their own functions for reading, writing, and processing data.

To analyze data using **Apache Pig**, programmers need to write scripts using Pig Latin language. All these scripts are internally converted to Map and Reduce tasks. Apache Pig has a component known as **Pig Engine** that accepts the Pig Latin scripts as input and converts those scripts into MapReduce jobs.

Why Do We Need Apache Pig?

Programmers who are not so good at Java normally used to struggle working with Hadoop, especially while performing any MapReduce tasks. Apache Pig is a boon for all such programmers.

- Using **Pig Latin**, programmers can perform MapReduce tasks easily without having to type complex codes in Java.
- Apache Pig uses **multi-query approach**, thereby reducing the length of codes. For example, an operation that would require you to type 200 lines of code (LoC) in Java can be easily done by

typing as less as just 10 LoC in Apache Pig. Ultimately Apache Pig reduces the development time by almost 16 times.

- Pig Latin is **SQL-like language** and it is easy to learn Apache Pig when you are familiar with SQL.
- Apache Pig provides many built-in operators to support data operations like joins, filters, ordering, etc. In addition, it also provides nested data types like tuples, bags, and maps that are missing from MapReduce.
- Features of Pig
- Apache Pig comes with the following features −
- **Rich set of operators** − It provides many operators to perform operations like join, sort, filer, etc.
- **Ease of programming** − Pig Latin is similar to SQL and it is easy to write a Pig script if you are good at SQL.
- **Optimization opportunities** − The tasks in Apache Pig optimize their execution automatically, so the programmers need to focus only on semantics of the language.
- **Extensibility** − Using the existing operators, users can develop their own functions to read, process, and write data.
- **UDF's** − Pig provides the facility to create **User-defined Functions** in other programming languages such as Java and invoke or embed them in Pig Scripts.
- **Handles all kinds of data** − Apache Pig analyzes all kinds of data, both structured as well as unstructured. It stores the results in HDFS.

Apache Pig Vs MapReduce

Listed below are the major differences between Apache Pig and MapReduce.

Apache Pig	**MapReduce**
Apache Pig is a data flow language.	MapReduce is a data processing paradigm.
It is a high level language.	MapReduce is low level and rigid.
Performing a Join operation in Apache Pig is pretty simple.	It is quite difficult in MapReduce to perform a Join operation between datasets.
Any novice programmer with a basic knowledge of SQL can work conveniently with Apache Pig.	Exposure to Java is must to work with MapReduce.

Apache Pig uses multi-query approach, thereby reducing the length of the codes to a great extent.	MapReduce will require almost 20 times more the number of lines to perform the same task.
There is no need for compilation. On execution, every Apache Pig operator is converted internally into a MapReduce job.	MapReduce jobs have a long compilation process.

Apache Pig Vs SQL

Listed below are the major differences between Apache Pig and SQL.

Pig	**SQL**
Pig Latin is a **procedural** language.	SQL is a **declarative** language.
In Apache Pig, **schema** is optional. We can store data without designing a schema (values are stored as $01, $02 etc.)	Schema is mandatory in SQL.
The data model in Apache Pig is **nested relational**.	The data model used in SQL **is flat relational**.
Apache Pig provides limited opportunity for **Query optimization**.	There is more opportunity for query optimization in SQL.

In addition to above differences, Apache Pig Latin −

- Allows splits in the pipeline.
- Allows developers to store data anywhere in the pipeline.
- Declares execution plans.
- Provides operators to perform ETL (Extract, Transform, and Load) functions.

Apache Pig Vs Hive

Both Apache Pig and Hive are used to create MapReduce jobs. And in some cases, Hive operates on HDFS in a similar way Apache Pig does. In the following table, we have listed a few significant points that set Apache Pig apart from Hive.

Apache Pig	**Hive**
Apache Pig uses a language called **Pig Latin**. It was originally created at **Yahoo**.	Hive uses a language called **HiveQL**. It was originally created at **Facebook**.
Pig Latin is a data flow language.	HiveQL is a query processing language.
Pig Latin is a procedural language and it fits in pipeline paradigm.	HiveQL is a declarative language.
Apache Pig can handle structured, unstructured, and semi-structured data.	Hive is mostly for structured data.

Applications of Apache Pig

Apache Pig is generally used by data scientists for performing tasks involving ad-hoc processing and quick prototyping. Apache Pig is used −

- To process huge data sources such as web logs.
- To perform data processing for search platforms.
- To process time sensitive data loads.

Apache Pig – History

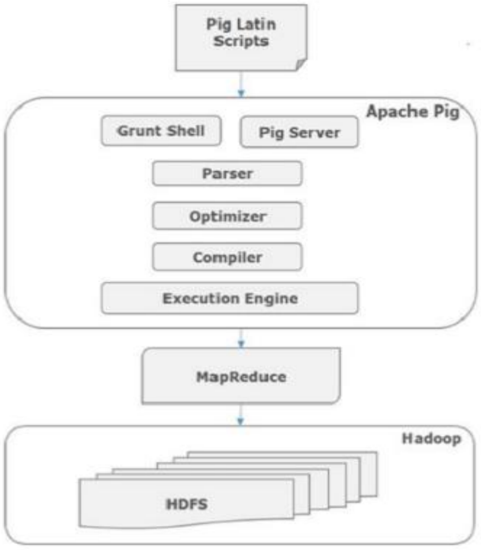
In **2006**, Apache Pig was developed as a research project at Yahoo, especially to create and execute MapReduce jobs on every dataset. In **2007**, Apache Pig was open sourced via Apache incubator. In **2008**, the first release of Apache Pig came out. In **2010**, Apache Pig graduated as an Apache top-level project.

Apache Pig - Architecture

The language used to analyze data in Hadoop using Pig is known as **Pig Latin**. It is a highlevel data processing language which provides a rich set of data types and operators to perform various operations on the data.

To perform a particular task Programmers using Pig, programmers need to write a Pig script using the Pig Latin language, and execute them using any of the execution mechanisms (Grunt Shell, UDFs, Embedded). After execution, these scripts will go through a series of transformations applied by the Pig Framework, to produce the desired output.

Internally, Apache Pig converts these scripts into a series of MapReduce jobs, and thus, it makes the programmer’s job easy. The architecture of Apache Pig is shown below.



Apache Pig Components

As shown in the figure, there are various components in the Apache Pig framework. Let us take a look at the major components.

Parser

Initially the Pig Scripts are handled by the Parser. It checks the syntax of the script, does type checking, and other miscellaneous checks. The output of the parser will be a DAG (directed acyclic graph), which represents the Pig Latin statements and logical operators.

In the DAG, the logical operators of the script are represented as the nodes and the data flows are represented as edges.

Optimizer

The logical plan (DAG) is passed to the logical optimizer, which carries out the logical optimizations such as projection and pushdown.

Compiler

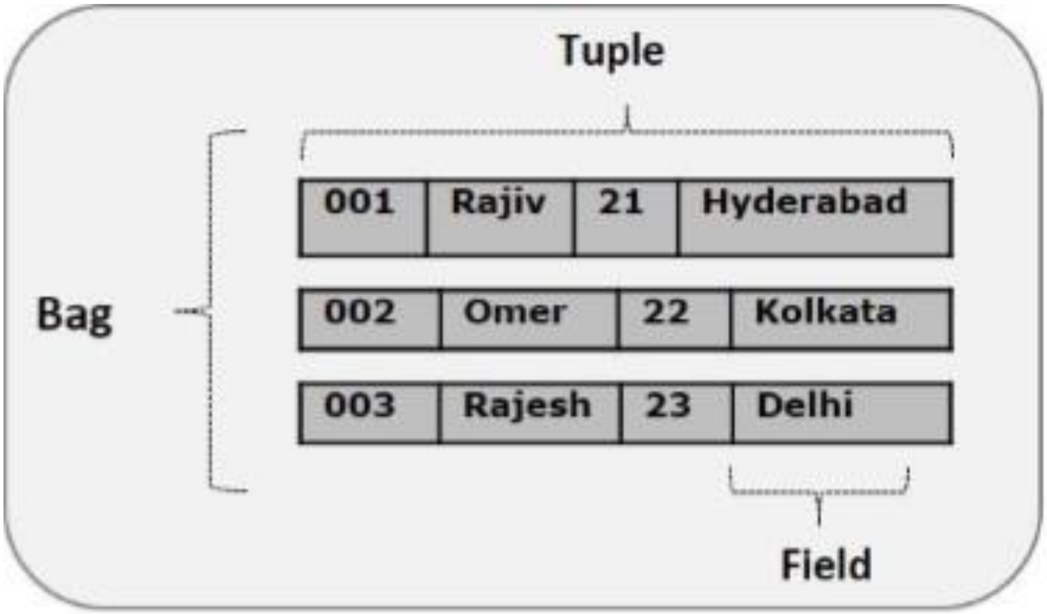
The compiler compiles the optimized logical plan into a series of MapReduce jobs.

Execution engine

Finally the MapReduce jobs are submitted to Hadoop in a sorted order. Finally, these MapReduce jobs are executed on Hadoop producing the desired results.

Pig Latin Data Model

The data model of Pig Latin is fully nested and it allows complex non-atomic datatypes such as **map** and **tuple**. Given below is the diagrammatical representation of Pig Latin's data model.



001	Rajiv	21	Hyderabad
002	Omer	22	Kolkata
003	Rajesh	23	Delhi

Atom

Any single value in Pig Latin, irrespective of their data, type is known as an **Atom**. It is stored as string and can be used as string and number. int, long, float, double, chararray, and bytearray are the atomic values of Pig. A piece of data or a simple atomic value is known as a **field**.

Example − 'raja' or '30'

Tuple

A record that is formed by an ordered set of fields is known as a tuple, the fields can be of any type. A tuple is similar to a row in a table of RDBMS.

Example − (Raja, 30)

Bag

A bag is an unordered set of tuples. In other words, a collection of tuples (non-unique) is known as a bag. Each tuple can have any number of fields (flexible schema). A bag is represented by '{}'. It is similar to a table in RDBMS, but unlike a table in RDBMS, it is not necessary that every tuple contain the same number of fields or that the fields in the same position (column) have the same type.

Example − {(Raja, 30), (Mohammad, 45)}

A bag can be a field in a relation; in that context, it is known as **inner bag**.

Example − {Raja, 30, **{9848022338, raja@gmail.com,}**}

Map

A map (or data map) is a set of key-value pairs. The **key** needs to be of type chararray and should be unique. The **value** might be of any type. It is represented by '[]'

Example − [name#Raja, age#30]

Relation

A relation is a bag of tuples. The relations in Pig Latin are unordered (there is no guarantee that tuples are processed in any particular order).

Apache Pig - Installation

This chapter explains the how to download, install, and set up **Apache Pig** in your system.

Prerequisites

It is essential that you have Hadoop and Java installed on your system before you go for Apache Pig. Therefore, prior to installing Apache Pig, install Hadoop and Java by following the steps given in the following link −

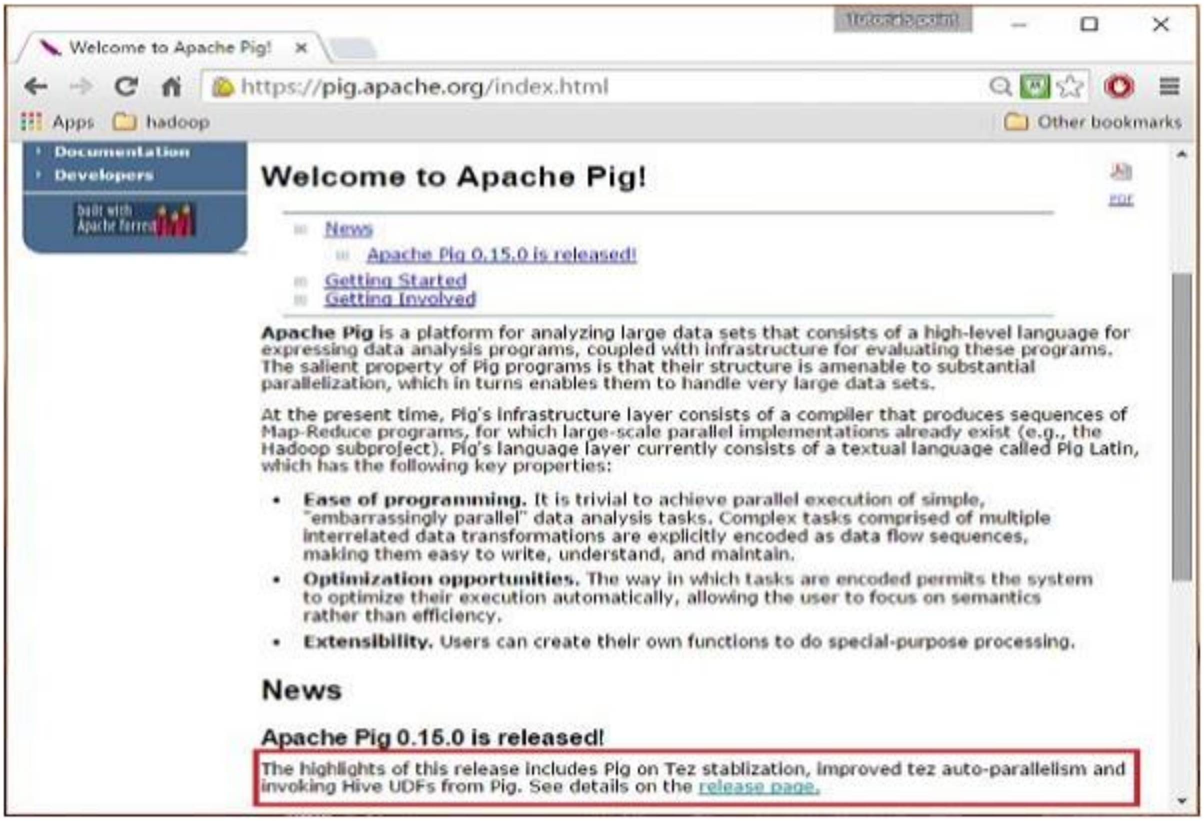
http://www.tutorialspoint.com/hadoop/hadoop_enviornment_setup.htm

Download Apache Pig

First of all, download the latest version of Apache Pig from the following website − https://pig.apache.org/

Step 1

Open the homepage of Apache Pig website. Under the section **News,** click on the link **release page** as shown in the following snapshot.



Welcome to Apache Pig!

- News
  - Apache Pig 0.15.0 is released!
- Getting Started
- Getting Involved

Apache Pig is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. The salient property of Pig programs is that their structure is amenable to substantial parallelization, which in turns enables them to handle very large data sets.

At the present time, Pig's infrastructure layer consists of a compiler that produces sequences of Map-Reduce programs, for which large-scale parallel implementations already exist (e.g., the Hadoop subproject). Pig's language layer currently consists of a textual language called Pig Latin, which has the following key properties:

- **Ease of programming.** It is trivial to achieve parallel execution of simple, "embarrassingly parallel" data analysis tasks. Complex tasks comprised of multiple interrelated data transformations are explicitly encoded as data flow sequences, making them easy to write, understand, and maintain.
- **Optimization opportunities.** The way in which tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.
- **Extensibility.** Users can create their own functions to do special-purpose processing.

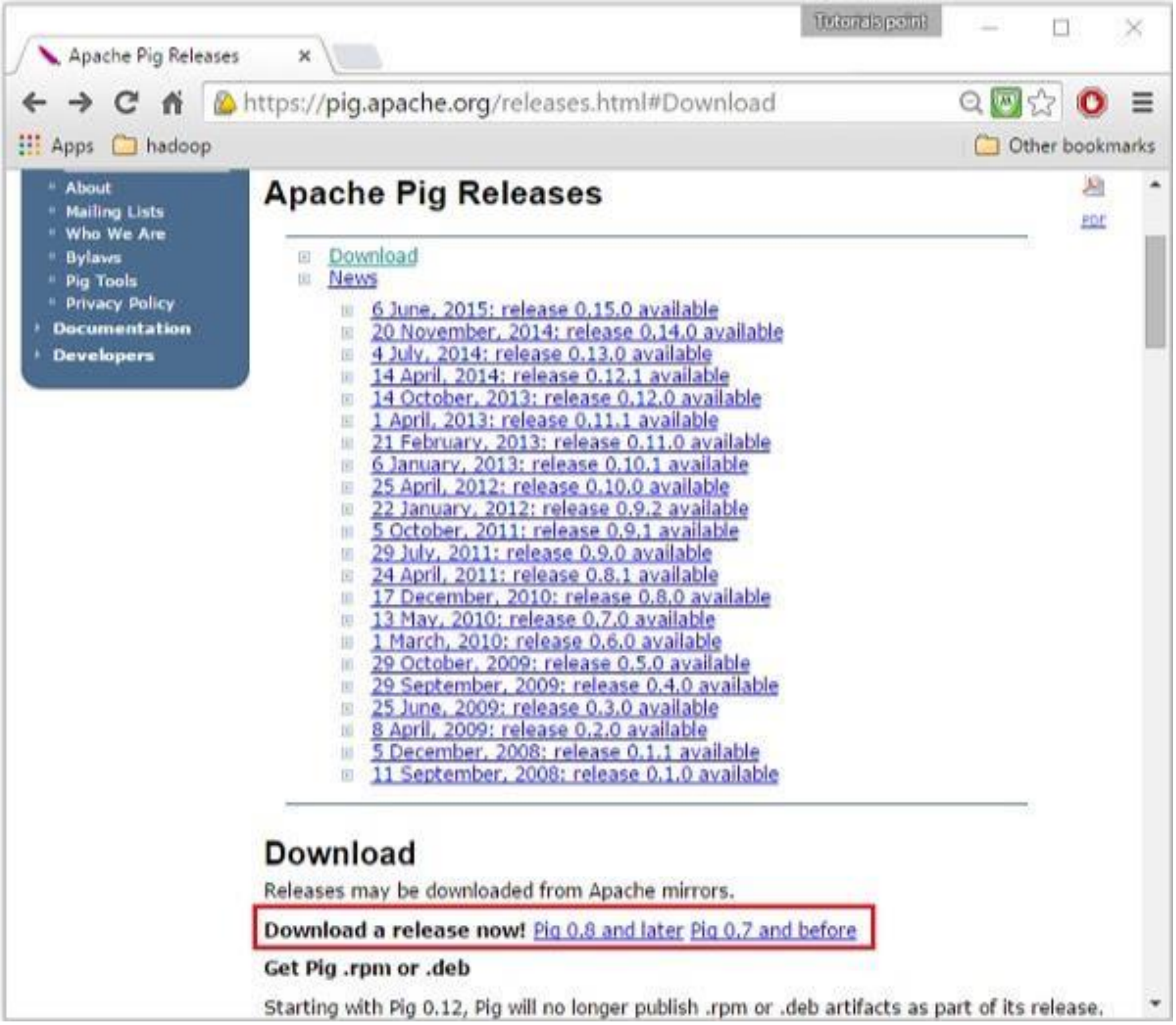
News

Apache Pig 0.15.0 is released!

The highlights of this release includes Pig on Tez stablization, improved tez auto-parallelism and invoking Hive UDFs from Pig. See details on the release page.

Step 2

On clicking the specified link, you will be redirected to the **Apache Pig Releases** page. On this page, under the **Download** section, you will have two links, namely, **Pig 0.8 and later** and **Pig 0.7 and before**. Click on the link **Pig 0.8 and later**, then you will be redirected to the page having a set of mirrors.



Apache Pig Releases

- Download
- News
  - 6 June, 2015: release 0.15.0 available
  - 20 November, 2014: release 0.14.0 available
  - 4 July, 2014: release 0.13.0 available
  - 14 April, 2014: release 0.12.1 available
  - 14 October, 2013: release 0.12.0 available
  - 1 April, 2013: release 0.11.1 available
  - 21 February, 2013: release 0.11.0 available
  - 6 January, 2013: release 0.10.1 available
  - 25 April, 2012: release 0.10.0 available
  - 22 January, 2012: release 0.9.2 available
  - 5 October, 2011: release 0.9.1 available
  - 29 July, 2011: release 0.9.0 available
  - 24 April, 2011: release 0.8.1 available
  - 17 December, 2010: release 0.8.0 available
  - 13 May, 2010: release 0.7.0 available
  - 1 March, 2010: release 0.6.0 available
  - 29 October, 2009: release 0.5.0 available
  - 29 September, 2009: release 0.4.0 available
  - 25 June, 2009: release 0.3.0 available
  - 8 April, 2009: release 0.2.0 available
  - 5 December, 2008: release 0.1.1 available
  - 11 September, 2008: release 0.1.0 available

Download

Releases may be downloaded from Apache mirrors.

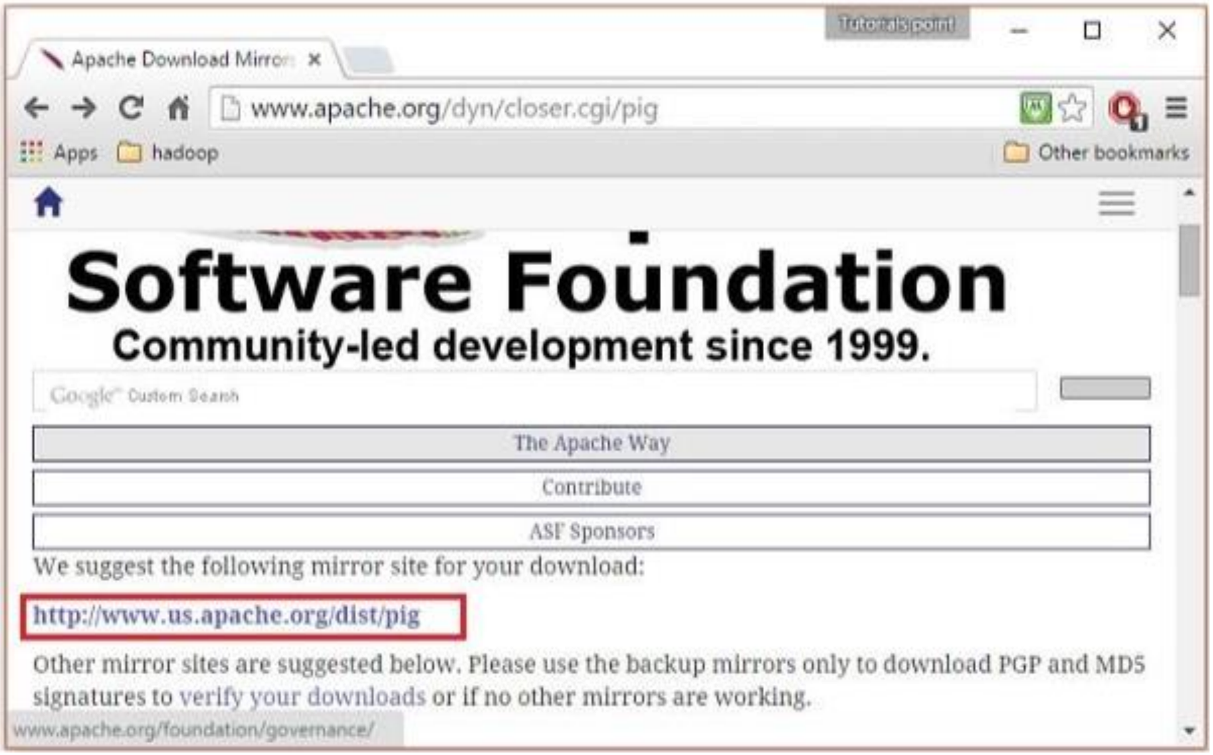
Download a release now! Pig 0.8 and later Pig 0.7 and before

Get Pig .rpm or .deb

Starting with Pig 0.12, Pig will no longer publish .rpm or .deb artifacts as part of its release.

Step 3

Choose and click any one of these mirrors as shown below.



Software Foundation

Community-led development since 1999.

The Apache Way

Contribute

ASF Sponsors

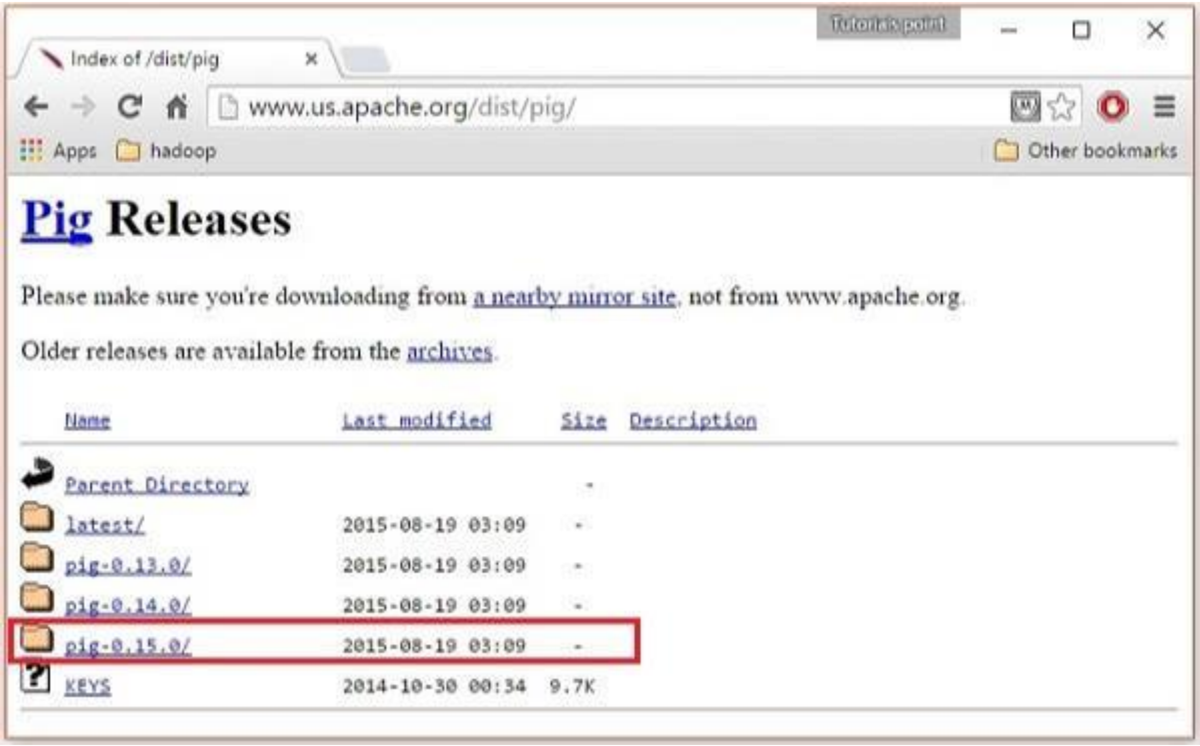
We suggest the following mirror site for your download:

http://www.us.apache.org/dist/pig

Other mirror sites are suggested below. Please use the backup mirrors only to download PGP and MD5 signatures to verify your downloads or if no other mirrors are working.

Step 4

These mirrors will take you to the **Pig Releases** page. This page contains various versions of Apache Pig. Click the latest version among them.



Pig Releases

Please make sure you're downloading from a nearby mirror site, not from www.apache.org.

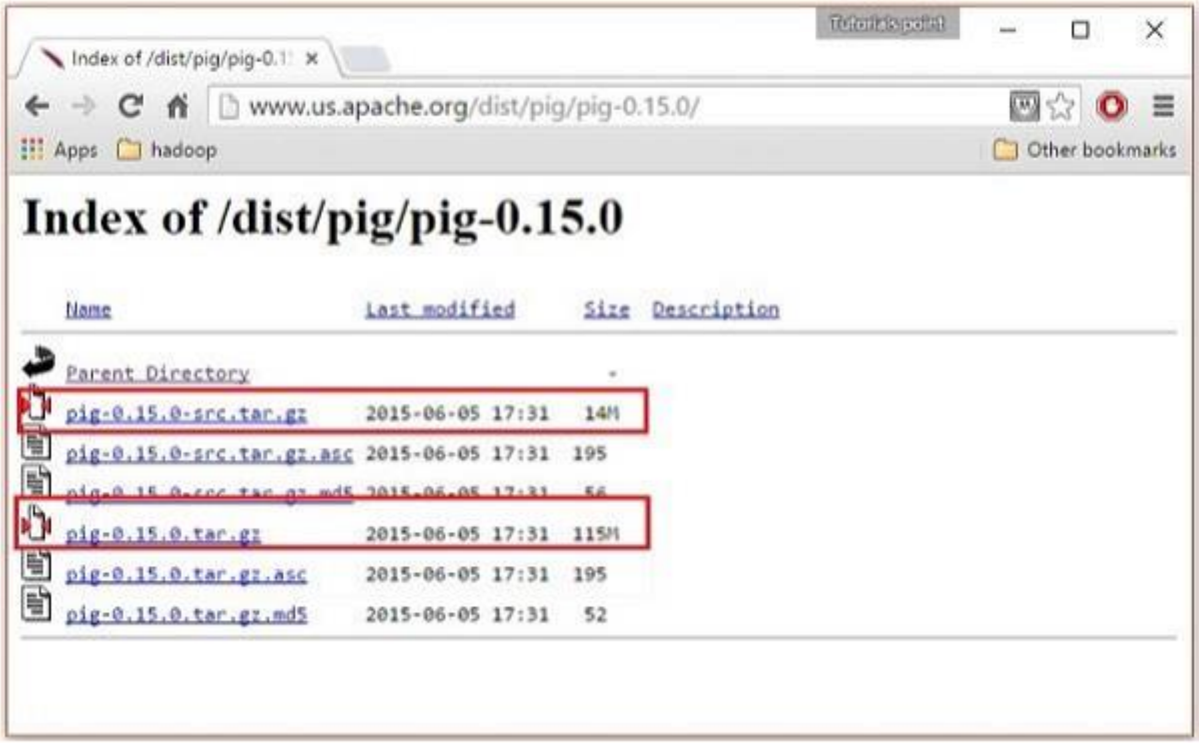
Older releases are available from the archives.

```
Name                 Last modified      Size  Description

Parent Directory                          -
latest/              2015-08-19 03:09    -
pig-0.13.0/          2015-08-19 03:09    -
pig-0.14.0/          2015-08-19 03:09    -
pig-0.15.0/          2015-08-19 03:09    -
KEYS                 2014-10-30 00:34  9.7K
```

Step 5

Within these folders, you will have the source and binary files of Apache Pig in various distributions. Download the tar files of the source and binary files of Apache Pig 0.15, **pig0.15.0-src.tar.gz** and **pig-0.15.0.tar.gz.**


Tutorials point
Index of /dist/pig/pig-0.1
www.us.apache.org/dist/pig/pig-0.15.0/
Apps
hadoop
Other bookmarks

Index of /dist/pig/pig-0.15.0

```
Name                        Last modified      Size  Description

Parent Directory                                 -
pig-0.15.0-src.tar.gz       2015-06-05 17:31   14M
pig-0.15.0-src.tar.gz.asc   2015-06-05 17:31   195
pig-0.15.0-src.tar.gz.md5   2015-06-05 17:31    56
pig-0.15.0.tar.gz           2015-06-05 17:31  115M
pig-0.15.0.tar.gz.asc       2015-06-05 17:31   195
pig-0.15.0.tar.gz.md5       2015-06-05 17:31    52
```

Install Apache Pig

After downloading the Apache Pig software, install it in your Linux environment by following the steps given below.

Step 1

Create a directory with the name Pig in the same directory where the installation directories of **Hadoop, Java,** and other software were installed. (In our tutorial, we have created the Pig directory in the user named Hadoop).

```
$ mkdirPig
```

Step 2

Extract the downloaded tar files as shown below.

```
$ cd Downloads/
$ tar zxvf pig-0.15.0-src.tar.gz
$ tar zxvf pig-0.15.0.tar.gz
```

Step 3

Move the content of **pig-0.15.0-src.tar.gz** file to the **Pig** directory created earlier as shown below.

```
$ mv pig-0.15.0-src.tar.gz/* /home/Hadoop/Pig/
```

Configure Apache Pig

After installing Apache Pig, we have to configure it. To configure, we need to edit two files − **bashrc and pig.properties**.

.bashrc file

In the **.bashrc** file, set the following variables −

- **PIG_HOME** folder to the Apache Pig’s installation folder,
- **PATH** environment variable to the bin folder, and
- **PIG_CLASSPATH** environment variable to the etc (configuration) folder of your Hadoop installations (the directory that contains the core-site.xml, hdfs-site.xml and mapred-site.xml files).

```
export PIG_HOME = /home/Hadoop/Pig
export PATH  = $PATH:/home/Hadoop/pig/bin
export PIG_CLASSPATH = $HADOOP_HOME/conf
```

pig.properties file

In the **conf** folder of Pig, we have a file named **pig.properties**. In the pig.properties file, you can set various parameters as given below.

```
pig -h properties
```

Verifying the Installation

Verify the installation of Apache Pig by typing the version command. If the installation is successful, you will get the version of Apache Pig as shown below.

```
$ pig –version

Apache Pig version 0.15.0 (r1682971)
compiled Jun 01 2015, 11:44:35
```

Running Pig

You can run Pig (execute Pig Latin statements and Pig commands) using various modes.

	Local Mode	**Tez Local Mode**	**Spark Local**	**Mapreduce Mode**	**Tez Mode**	**Spark Mode**

			Mode			
Interactive Mode	yes	experimental	yes	yes		
Batch Mode	yes	experimental	yes	yes		

Execution Modes

Pig has six execution modes or exectypes:

- **Local Mode** - To run Pig in local mode, you need access to a single machine; all files are installed and run using your local host and file system. Specify local mode using the -x flag (pig -x local).
- **Tez Local Mode** - To run Pig in tez local mode. It is similar to local mode, except internally Pig will invoke tez runtime engine. Specify Tez local mode using the -x flag (pig -x tez_local).

Note: Tez local mode is experimental. There are some queries which just error out on bigger data in local mode.

- **Spark Local Mode** - To run Pig in spark local mode. It is similar to local mode, except internally Pig will invoke spark runtime engine. Specify Spark local mode using the -x flag (pig -x spark_local).

Note: Spark local mode is experimental. There are some queries which just error out on bigger data in local mode.

- **Mapreduce Mode** - To run Pig in mapreduce mode, you need access to a Hadoop cluster and HDFS installation. Mapreduce mode is the default mode; you can, *but don't need to*, specify it using the -x flag (pig OR pig -x mapreduce).
- **Tez Mode** - To run Pig in Tez mode, you need access to a Hadoop cluster and HDFS installation. Specify Tez mode using the -x flag (-x tez).
- **Spark Mode** - To run Pig in Spark mode, you need access to a Spark, Yarn or Mesos cluster and HDFS installation. Specify Spark mode using the -x flag (-x spark). In Spark execution mode, it is necessary to set env::SPARK_MASTER to an appropriate value (local - local mode, yarn-client - yarn-client mode, mesos://host:port - spark on mesos or spark://host:port - spark cluster. For more information refer to spark documentation on Master URLs, *yarn-cluster mode is currently not supported*). Pig scripts run on Spark can take advantage of the dynamic allocation feature. The feature can be enabled by simply enabling *spark.dynamicAllocation.enabled*. Refer to spark configuration for additional configuration details. In general all properties in the pig script prefixed with *spark.* are copied

to the Spark Application Configuration. Please note that Yarn auxillary service need to be enabled on Spark for this to work. See Spark documentation for additional details.

You can run Pig in either mode using the "pig" command (the bin/pig Perl script) or the "java" command (java -cp pig.jar ...).

Examples

This example shows how to run Pig in local and mapreduce mode using the pig command.

```
/* local mode */
$ pig -x local ...

/* Tez local mode */
$ pig -x tez_local ...

/* Spark local mode */
$ pig -x spark_local ...

/* mapreduce mode */
$ pig ...
or
$ pig -x mapreduce ...

/* Tez mode */
$ pig -x tez ...

/* Spark mode */
$ pig -x spark ...
```

Interactive Mode

You can run Pig in interactive mode using the Grunt shell. Invoke the Grunt shell using the "pig" command (as shown below) and then enter your Pig Latin statements and Pig commands interactively at the command line.

Example

These Pig Latin statements extract all user IDs from the /etc/passwd file. First, copy the /etc/passwd file to your local working directory. Next, invoke the Grunt shell by typing the "pig" command (in local or hadoop mode). Then, enter the Pig Latin statements interactively at the grunt prompt (be sure to include the semicolon after each statement). The DUMP operator will display the results to your terminal screen.

```
grunt> A = load 'passwd' using PigStorage(':');
grunt> B = foreach A generate $0 as id;
grunt> dump B;
```

Local Mode

```
$ pig -x local
... - Connecting to ...
grunt>
```

Tez Local Mode

```
$ pig -x tez_local
... - Connecting to ...
grunt>
```

Spark Local Mode

```
$ pig -x spark_local
... - Connecting to ...
grunt>
```

Mapreduce Mode

```
$ pig -x mapreduce
... - Connecting to ...
grunt>

or

$ pig
... - Connecting to ...
grunt>
```

Tez Mode

```
$ pig -x tez
... - Connecting to ...
grunt>
```

Spark Mode

```
$ pig -x spark
... - Connecting to ...
grunt>
```

Batch Mode

You can run Pig in batch mode using Pig scripts and the "pig" command (in local or hadoop mode).

Example

The Pig Latin statements in the Pig script (id.pig) extract all user IDs from the /etc/passwd file. First, copy the /etc/passwd file to your local working directory. Next, run the Pig script from the command line (using local or mapreduce mode). The STORE operator will write the results to a file (id.out).

```
/* id.pig */

A = load 'passwd' using PigStorage(':');  -- load the passwd file
B = foreach A generate $0 as id;  -- extract the user IDs
store B into 'id.out';  -- write the results to a file name id.out
```

Local Mode

```
$ pig -x local id.pig
```

Tez Local Mode

```
$ pig -x tez_localid.pig
```

Spark Local Mode

```
$ pig -x spark_localid.pig
```

Mapreduce Mode

```
$ pig id.pig
or
$ pig -x mapreduceid.pig
```

Tez Mode

```
$ pig -x tezid.pig
```

Spark Mode

```
$ pig -x spark id.pig
```

Pig Scripts

Use Pig scripts to place Pig Latin statements and Pig commands in a single file. While not required, it is good practice to identify the file using the *.pig extension.

You can run Pig scripts from the command line and from the Grunt shell (see the run and exec commands).

Pig scripts allow you to pass values to parameters using parameter substitution.

Comments in Scripts

You can include comments in Pig scripts:

- For multi-line comments use /* …. */
- For single-line comments use --

```
/* myscript.pig
My script is simple.
It includes three Pig Latin statements.
*/


A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float); -- loading data
B = FOREACH A GENERATE name;  -- transforming data
DUMP B;  -- retrieving results
```

Scripts and Distributed File Systems

Pig supports running scripts (and Jar files) that are stored in HDFS, Amazon S3, and other distributed file systems. The script's full location URI is required (see REGISTER for information about Jar files). For example, to run a Pig script on HDFS, do the following:

```
$ pig hdfs://nn.mydomain.com:9020/myscripts/script.pig
```

Pig Latin Statements

Pig Latin statements are the basic constructs you use to process data using Pig. A Pig Latin statement is an operator that takes a relation as input and produces another relation as output. (This definition applies to all Pig Latin operators except LOAD and STORE which read data from and write data to the file system.) Pig Latin statements may include expressions and schemas. Pig Latin statements can span multiple lines and must end with a semi-colon (;). By default, Pig Latin statements are processed using multi-query execution.

Pig Latin statements are generally organized as follows:

- A LOAD statement to read data from the file system.
- A series of "transformation" statements to process the data.
- A DUMP statement to view results or a STORE statement to save the results.

Note that a DUMP or STORE statement is required to generate output.

- In this example Pig will validate, but not execute, the LOAD and FOREACH statements.

```
A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float);
B = FOREACH A GENERATE name;
```

- In this example, Pig will validate and then execute the LOAD, FOREACH, and DUMP statements.

```
A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float);
B = FOREACH A GENERATE name;
DUMP B;
(John)
(Mary)
(Bill)
(Joe)
```

Loading Data

Use the LOAD operator and the load/store functions to read data into Pig (PigStorage is the default load function).

Working with Data

Pig allows you to transform data in many ways. As a starting point, become familiar with these operators:

- Use the FILTER operator to work with tuples or rows of data. Use the FOREACH operator to work with columns of data.
- Use the GROUP operator to group data in a single relation. Use the COGROUP, inner JOIN, and outer JOIN operators to group or join data in two or more relations.
- Use the UNION operator to merge the contents of two or more relations. Use the SPLIT operator to partition the contents of a relation into multiple relations.

Storing Intermediate Results

Pig stores the intermediate data generated between MapReduce jobs in a temporary location on HDFS. This location must already exist on HDFS prior to use. This location can be configured using the pig.temp.dir property. The property's default value is "/tmp" which is the same as the hardcoded location in Pig 0.7.0 and earlier versions.

Storing Final Results

Use the STORE operator and the load/store functions to write results to the file system (PigStorage is the default store function).

Note: During the testing/debugging phase of your implementation, you can use DUMP to display results to your terminal screen. However, in a production environment you always want to use the STORE operator to save your results (see Store vs. Dump).

Debugging Pig Latin

Pig Latin provides operators that can help you debug your Pig Latin statements:

- Use the DUMP operator to display results to your terminal screen.
- Use the DESCRIBE operator to review the schema of a relation.
- Use the EXPLAIN operator to view the logical, physical, or map reduce execution plans to compute a relation.
- Use the ILLUSTRATE operator to view the step-by-step execution of a series of statements.

Shortcuts for Debugging Operators

Pig provides shortcuts for the frequently used debugging operators (DUMP, DESCRIBE, EXPLAIN, ILLUSTRATE). These shortcuts can be used in Grunt shell or within pig scripts. Following are the shortcuts supported by pig

- \d alias - shourtcut for DUMP operator. If alias is ignored last defined alias will be used.
- \de alias - shourtcut for DESCRIBE operator. If alias is ignored last defined alias will be used.
- \e alias - shourtcut for EXPLAIN operator. If alias is ignored last defined alias will be used.
- \i alias - shourtcut for ILLUSTRATE operator. If alias is ignored last defined alias will be used.
- \q - To quit grunt shell

Pig Properties

Pig supports a number of Java properties that you can use to customize Pig behavior. You can retrieve a list of the properties using the help properties command. All of these properties are optional; none are required.

To specify Pig properties use one of these mechanisms:

- The pig.properties file (add the directory that contains the pig.properties file to the classpath)
- The -D and a Pig property in PIG_OPTS environment variable (export PIG_OPTS=-Dpig.tmpfilecompression=true)
- The -P command line option and a properties file (pig -P mypig.properties)
- The set command (set pig.exec.nocombiner true)

Note: The properties file uses standard Java property file format.

The following precedence order is supported: pig.properties< -D Pig property < -P properties file < set command. This means that if the same property is provided using the –D command line option as well as the –P command line option (properties file), the value of the property in the properties file will take precedence.

To specify Hadoop properties you can use the same mechanisms:

- Hadoop configuration files (include pig-cluster-hadoop-site.xml)
- The -D and a Hadoop property in PIG_OPTS environment variable (export PIG_OPTS=–Dmapreduce.task.profile=true)
- The -P command line option and a property file (pig -P property_file)
- The set command (set mapred.map.tasks.speculative.execution false)

The same precedence holds: Hadoop configuration files < -D Hadoop property < -P properties_file< set command.

Hadoop properties are not interpreted by Pig but are passed directly to Hadoop. Any Hadoop property can be passed this way.

All properties that Pig collects, including Hadoop properties, are available to any UDF via the UDFContext object. To get access to the properties, you can call the getJobConf method.

Apache Pig - Diagnostic Operators

The **load** statement will simply load the data into the specified relation in Apache Pig. To verify the execution of the **Load** statement, you have to use the **Diagnostic Operators**. Pig Latin provides four different types of diagnostic operators −

- Dump operator
- Describe operator
- Explanation operator
- Illustration operator

In this chapter, we will discuss the Dump operators of Pig Latin.

Dump Operator

The **Dump** operator is used to run the Pig Latin statements and display the results on the screen. It is generally used for debugging Purpose.

Syntax

Given below is the syntax of the **Dump** operator.

```
grunt> Dump Relation_Name
```

Example

Assume we have a file **student_data.txt** in HDFS with the following content.

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

And we have read it into a relation **student** using the LOAD operator as shown below.

```
grunt> student = LOAD 'hdfs://localhost:9000/pig_data/student_data.txt'
   USING PigStorage(',')
as(id:int,firstname:chararray,lastname:chararray,phone:chararray,
city:chararray);
```

Now, let us print the contents of the relation using the **Dump operator** as shown below.

```
grunt> Dump student
```

Once you execute the above **Pig Latin** statement, it will start a MapReduce job to read data from HDFS.

Apache Pig - Describe Operator

The **describe** operator is used to view the schema of a relation.

Syntax

The syntax of the **describe** operator is as follows −

```
grunt> Describe Relation_name
```

Apache Pig - Explain Operator

The **explain** operator is used to display the logical, physical, and MapReduce execution plans of a relation.

Syntax

Given below is the syntax of the **explain** operator.

```
grunt> explain Relation_name;
```

Apache Pig - Illustrate Operator

The **illustrate** operator gives you the step-by-step execution of a sequence of statements.

Syntax

Given below is the syntax of the **illustrate** operator.

```
grunt> illustrate Relation_name;
```

Apache Pig - Group Operator

The **GROUP** operator is used to group the data in one or more relations. It collects the data having the same key.

Syntax

Given below is the syntax of the **group** operator.

```
grunt>Group_data = GROUP Relation_name BY age;
```

Apache Pig - Cogroup Operator

The **COGROUP** operator works more or less in the same way as the GROUP operator. The only difference between the two operators is that the **group** operator is normally used with one relation, while the **cogroup** operator is used in statements involving two or more relations.

Grouping Two Relations using Cogroup

Assume that we have two files namely **student_details.txt** and **employee_details.txt** in the HDFS directory **/pig_data/**

Apache Pig - Join Operator

The **JOIN** operator is used to combine records from two or more relations. While performing a join operation, we declare one (or a group of) tuple(s) from each relation, as keys. When these keys match, the two particular tuples are matched, else the records are dropped. Joins can be of the following types −

- Self-join
- Inner-join
- Outer-join − left join, right join, and full join

This chapter explains with examples how to use the join operator in Pig Latin. Assume that we have two files namely **customers.txt** and **orders.txt** in the **/pig_data/** directory of HDFS

Apache Pig - Cross Operator

The **CROSS** operator computes the cross-product of two or more relations. This chapter explains with example how to use the cross operator in Pig Latin.

Syntax

Given below is the syntax of the **CROSS** operator.

grunt> Relation3_name = CROSS Relation1_name, Relation2_name;

Apache Pig - Union Operator

The **UNION** operator of Pig Latin is used to merge the content of two relations. To perform UNION operation on two relations, their columns and domains must be identical.

Syntax

Given below is the syntax of the **UNION** operator.

grunt> Relation_name3 = UNION Relation_name1, Relation_name2;

Apache Pig - Split Operator

The **SPLIT** operator is used to split a relation into two or more relations.

Syntax

Given below is the syntax of the **SPLIT** operator.

grunt> SPLIT Relation1_name INTO Relation2_name IF (condition1), Relation2_name (condition2),

Apache Pig - Filter Operator

The **FILTER** operator is used to select the required tuples from a relation based on a condition.

Syntax

Given below is the syntax of the **FILTER** operator.

grunt> Relation2_name = FILTER Relation1_name BY (condition);

Apache Pig - Distinct Operator

The **DISTINCT** operator is used to remove redundant (duplicate) tuples from a relation.

Syntax

Given below is the syntax of the **DISTINCT** operator.

grunt> Relation_name2 = DISTINCT Relatin_name1;

Apache Pig - Foreach Operator

The **FOREACH** operator is used to generate specified data transformations based on the column data.

Syntax

Given below is the syntax of **FOREACH** operator.

grunt> Relation_name2 = FOREACH Relatin_name1 GENERATE (required data);

Working with functions in Pig

Apache Pig provides various built-in functions namely **eval, load, store, math, string, bag** and **tuple** functions.

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 tuples of 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.

The **Load** and **Store** functions in Apache Pig are used to determine how the data goes ad comes out of Pig. These functions are used with the load and store operators. Given below is the list of load and store functions available in Pig.

S.N.	Function & Description
1	PigStorage() To load and store structured files.
2	TextLoader()

	To load unstructured data into Pig.
3	BinStorage() To load and store data into Pig using machine readable format.
4	Handling Compression In Pig Latin, we can load and store compressed data.

Given below is the list of Bag and Tuple functions.

S.N.	Function & Description
1	TOBAG() To convert two or more expressions into a bag.
2	TOP() To get the top **N** tuples of a relation.
3	TOTUPLE() To convert one or more expressions into a tuple.
4	TOMAP() To convert the key-value pairs into a Map.

We have the following String functions in Apache Pig.

S.N.	Functions & Description
1	ENDSWITH(string, testAgainst) To verify whether a given string ends with a particular substring.
2	STARTSWITH(string, substring) Accepts two string parameters and verifies whether the first string starts with the second.
3	SUBSTRING(string, startIndex, stopIndex) Returns a substring from a given string.

4	EqualsIgnoreCase(string1, string2) To compare two stings ignoring the case.
5	INDEXOF(string, ‘character’, startIndex) Returns the first occurrence of a character in a string, searching forward from a start index.
6	LAST_INDEX_OF(expression) Returns the index of the last occurrence of a character in a string, searching backward from a start index.
7	LCFIRST(expression) Converts the first character in a string to lower case.
8	UCFIRST(expression) Returns a string with the first character converted to upper case.
9	UPPER(expression) UPPER(expression) Returns a string converted to upper case.
10	LOWER(expression) Converts all characters in a string to lower case.
11	REPLACE(string, ‘oldChar’, ‘newChar’); To replace existing characters in a string with new characters.
12	STRSPLIT(string, regex, limit) To split a string around matches of a given regular expression.
13	STRSPLITTOBAG(string, regex, limit) Similar to the **STRSPLIT()** function, it splits the string by given delimiter and returns the result in a bag.
14	TRIM(expression) Returns a copy of a string with leading and trailing whitespaces removed.

15	LTRIM(expression) Returns a copy of a string with leading whitespaces removed.
16	RTRIM(expression) Returns a copy of a string with trailing whitespaces removed.

Apache Pig provides the following Date and Time functions −

S.N.	Functions & Description
1	ToDate(milliseconds) This function returns a date-time object according to the given parameters. The other alternative for this function are ToDate(iosstring), ToDate(userstring, format), ToDate(userstring, format, timezone)
2	CurrentTime() returns the date-time object of the current time.
3	GetDay(datetime) Returns the day of a month from the date-time object.
4	GetHour(datetime) Returns the hour of a day from the date-time object.
5	GetMilliSecond(datetime) Returns the millisecond of a second from the date-time object.
6	GetMinute(datetime) Returns the minute of an hour from the date-time object.
7	GetMonth(datetime) Returns the month of a year from the date-time object.
8	GetSecond(datetime) Returns the second of a minute from the date-time object.

9	GetWeek(datetime) Returns the week of a year from the date-time object.
10	GetWeekYear(datetime) Returns the week year from the date-time object.
11	GetYear(datetime) Returns the year from the date-time object.
12	AddDuration(datetime, duration) Returns the result of a date-time object along with the duration object.
13	SubtractDuration(datetime, duration) Subtracts the Duration object from the Date-Time object and returns the result.
14	DaysBetween(datetime1, datetime2) Returns the number of days between the two date-time objects.
15	HoursBetween(datetime1, datetime2) Returns the number of hours between two date-time objects.
16	MilliSecondsBetween(datetime1, datetime2) Returns the number of milliseconds between two date-time objects.
17	MinutesBetween(datetime1, datetime2) Returns the number of minutes between two date-time objects.
18	MonthsBetween(datetime1, datetime2) Returns the number of months between two date-time objects.
19	SecondsBetween(datetime1, datetime2) Returns the number of seconds between two date-time objects.
20	WeeksBetween(datetime1, datetime2)

	Returns the number of weeks between two date-time objects.
21	YearsBetween(datetime1, datetime2) Returns the number of years between two date-time objects.

We have the following Math functions in Apache Pig −

S.N.	Functions & Description
1	ABS(expression) To get the absolute value of an expression.
2	ACOS(expression) To get the arc cosine of an expression.
3	ASIN(expression) To get the arc sine of an expression.
4	ATAN(expression) This function is used to get the arc tangent of an expression.
5	CBRT(expression) This function is used to get the cube root of an expression.
6	CEIL(expression) This function is used to get the value of an expression rounded up to the nearest integer.
7	COS(expression) This function is used to get the trigonometric cosine of an expression.
8	COSH(expression) This function is used to get the hyperbolic cosine of an expression.
9	EXP(expression) This function is used to get the Euler’s number e raised to the power of x.

10	FLOOR(expression) To get the value of an expression rounded down to the nearest integer.
11	LOG(expression) To get the natural logarithm (base e) of an expression.
12	LOG10(expression) To get the base 10 logarithm of an expression.
13	RANDOM( ) To get a pseudo random number (type double) greater than or equal to 0.0 and less than 1.0.
14	ROUND(expression) To get the value of an expression rounded to an integer (if the result type is float) or rounded to a long (if the result type is double).
15	SIN(expression) To get the sine of an expression.
16	SINH(expression) To get the hyperbolic sine of an expression.
17	SQRT(expression) To get the positive square root of an expression.
18	TAN(expression) To get the trigonometric tangent of an angle.
19	TANH(expression) To get the hyperbolic tangent of an expression.

Hive - Introduction

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

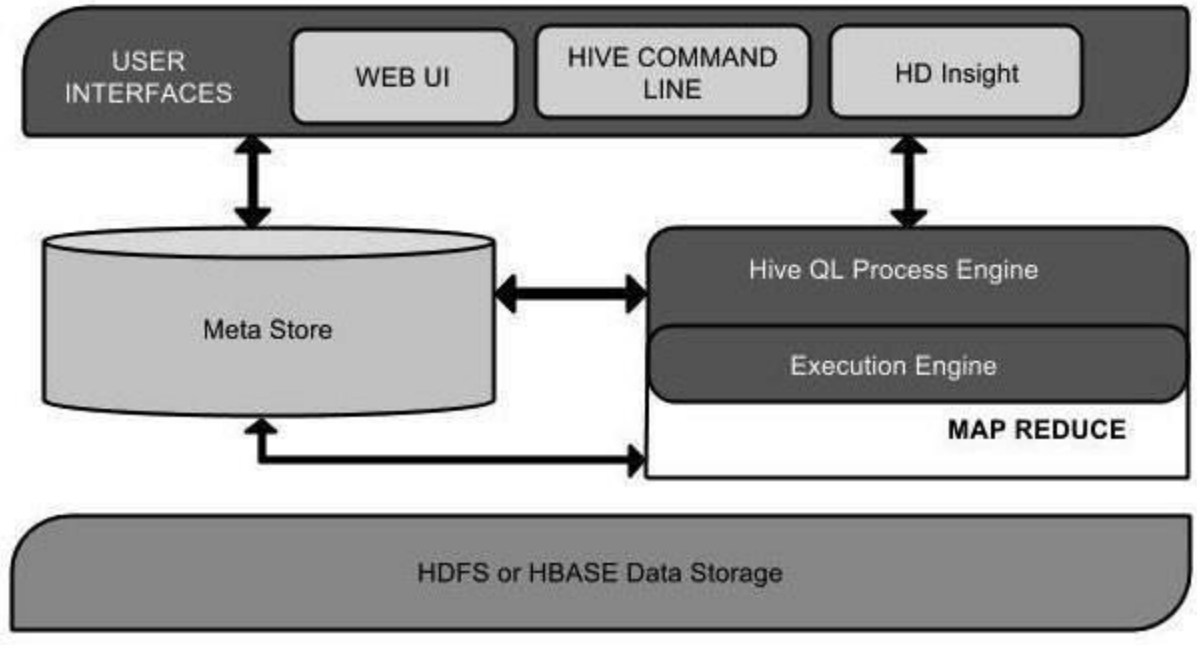
Features of Hive

- ✓ 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:



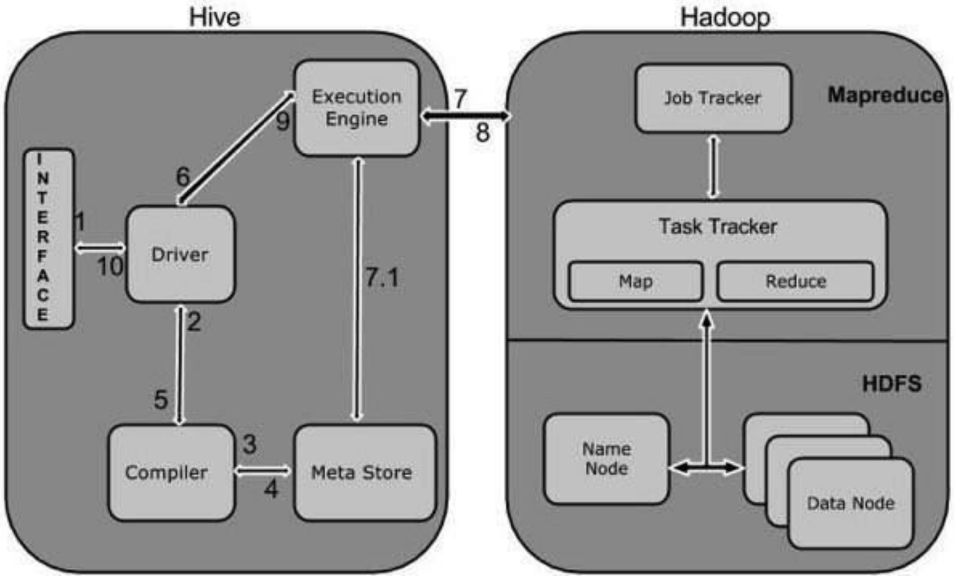
This component diagram contains different units. The following table describes each unit:

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 approach for 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	**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.
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, which is 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.

Installation and Configuration

You can install a stable release of Hive by downloading a tarball, or you can download the source code and build Hive from that.

Running HiveServer2 and Beeline

Requirements

Java 1.7

Note: Hive versions 1.2 onward require Java 1.7 or newer. Hive versions 0.14 to 1.1 work with Java 1.6 as well. Users are strongly advised to start moving to Java 1.8 (see HIVE-8607).

Hadoop 2.x (preferred), 1.x (not supported by Hive 2.0.0 onward). Hive versions up to 0.13 also supported Hadoop 0.20.x, 0.23.x.

Hive is commonly used in production Linux and Windows environment. Mac is a commonly used development environment. The instructions in this document are applicable to Linux and Mac. Using it on Windows would require slightly different steps.

Installing Hive from a Stable Release

Start by downloading the most recent stable release of Hive from one of the Apache download mirrors (see Hive Releases).

Next you need to unpack the tarball. This will result in the creation of a subdirectory named hive-x.y.z (where x.y.z is the release number):

```
  $ tar -xzvf hive-x.y.z.tar.gz
```

Set the environment variable HIVE_HOME to point to the installation directory:

```
  $ cd hive-x.y.z
  $ export HIVE_HOME={{pwd}}
```

Finally, add $HIVE_HOME/bin to your PATH:

```
  $ export PATH=$HIVE_HOME/bin:$PATH
```

Building Hive from Source

The Hive GIT repository for the most recent Hive code is located here: git clone https://git-wip-us.apache.org/repos/asf/hive.git (the master branch).

All release versions are in branches named "branch-0.#" or "branch-1.#" or the upcoming "branch-2.#", with the exception of release 0.8.1 which is in "branch-0.8-r2". Any branches with other names are feature branches for works-in-progress. See Understanding Hive Branches for details.

As of 0.13, Hive is built using Apache Maven.

Compile Hive on master

To build the current Hive code from the master branch:

```
  $ git clone https://git-wip-us.apache.org/repos/asf/hive.git
  $ cd hive
  $ mvn clean package -Pdist [-DskipTests -Dmaven.javadoc.skip=true]
  $ cd packaging/target/apache-hive-{version}-SNAPSHOT-bin/apache-hive-{version}-SNAPSHOT-bin
  $ ls
  LICENSE
  NOTICE
  README.txt
  RELEASE_NOTES.txt
  bin/ (all the shell scripts)
  lib/ (required jar files)
  conf/ (configuration files)
  examples/ (sample input and query files)
```

```
hcatalog / (hcatalog installation)
  scripts / (upgrade scripts for hive-metastore)
```

Here, {version} refers to the current Hive version.

If building Hive source using Maven (mvn), we will refer to the directory "/packaging/target/apache-hive-{version}-SNAPSHOT-bin/apache-hive-{version}-SNAPSHOT-bin" as <install-dir> for the rest of the page.

Compile Hive on branch-1

In branch-1, Hive supports both Hadoop 1.x and 2.x. You will need to specify which version of Hadoop to build against via a Maven profile. To build against Hadoop 1.x use the profile hadoop-1; for Hadoop 2.x use hadoop-2. For example to build against Hadoop 1.x, the above mvn command becomes:

```
  $ mvn clean package -Phadoop-1,dist
```

Compile Hive Prior to 0.13 on Hadoop 0.20

Prior to Hive 0.13, Hive was built using Apache Ant. To build an older version of Hive on Hadoop 0.20:

```
  $ svn co http://svn.apache.org/repos/asf/hive/branches/branch-{version} hive
  $ cd hive
  $ ant clean package
  $ cd build/dist
  # ls
  LICENSE
  NOTICE
  README.txt
  RELEASE_NOTES.txt
  bin/ (all the shell scripts)
  lib/ (required jar files)
  conf/ (configuration files)
  examples/ (sample input and query files)
hcatalog / (hcatalog installation)
  scripts / (upgrade scripts for hive-metastore)
```

If using Ant, we will refer to the directory "build/dist" as <install-dir>.

Compile Hive Prior to 0.13 on Hadoop 0.23

To build Hive in Ant against Hadoop 0.23, 2.0.0, or other version, build with the appropriate flag; some examples below:

```
  $ ant clean package -Dhadoop.version=0.23.3 -Dhadoop-0.23.version=0.23.3 -Dhadoop.mr.rev=23
  $ ant clean package -Dhadoop.version=2.0.0-alpha -Dhadoop-0.23.version=2.0.0-alpha -Dhadoop.mr.rev=23
```

Running Hive

Hive uses Hadoop, so:

- you must have Hadoop in your path OR
- export HADOOP_HOME=<hadoop-install-dir>

In addition, you must use below HDFS commands to create /tmp and /user/hive/warehouse (aka hive.metastore.warehouse.dir) and set them chmodg+w before you can create a table in Hive.

```
  $ $HADOOP_HOME/bin/hadoop fs -mkdir       /tmp
  $ $HADOOP_HOME/bin/hadoop fs -mkdir       /user/hive/warehouse
  $ $HADOOP_HOME/bin/hadoop fs -chmodg+w   /tmp
  $ $HADOOP_HOME/bin/hadoop fs -chmodg+w   /user/hive/warehouse
```

You may find it useful, though it's not necessary, to set HIVE_HOME:

```
  $ export HIVE_HOME=<hive-install-dir>
```

Running Hive CLI

To use the Hive command line interface (CLI) from the shell:

```
  $ $HIVE_HOME/bin/hive
```

Running HiveServer2 and Beeline

Starting from Hive 2.1, we need to run the schematool command below as an initialization step. For example, we can use "derby" as db type.

```
  $ $HIVE_HOME/bin/schematool -dbType<db type> -initSchema
```

HiveServer2 (introduced in Hive 0.11) has its own CLI called Beeline. HiveCLI is now deprecated in favor of Beeline, as it lacks the multi-user, security, and other capabilities of HiveServer2. To run HiveServer2 and Beeline from shell:

```
  $ $HIVE_HOME/bin/hiveserver2
  $ $HIVE_HOME/bin/beeline -u jdbc:hive2://$HS2_HOST:$HS2_PORT
```

Beeline is started with the JDBC URL of the HiveServer2, which depends on the address and port where HiveServer2 was started. By default, it will be (localhost:10000), so the address will look like jdbc:hive2://localhost:10000.

Or to start Beeline and HiveServer2 in the same process for testing purpose, for a similar user experience to HiveCLI:

```
  $ $HIVE_HOME/bin/beeline -u jdbc:hive2://
```

Running HCatalog

To run the HCatalog server from the shell in Hive release 0.11.0 and later:

```
  $ $HIVE_HOME/hcatalog/sbin/hcat_server.sh
```

To use the HCatalog command line interface (CLI) in Hive release 0.11.0 and later:

```
  $ $HIVE_HOME/hcatalog/bin/hcat
```

For more information, see HCatalog Installation from Tarball and HCatalog CLI in the HCatalog manual.

Running WebHCat (Templeton)

To run the WebHCat server from the shell in Hive release 0.11.0 and later:

```
  $ $HIVE_HOME/hcatalog/sbin/webhcat_server.sh
```

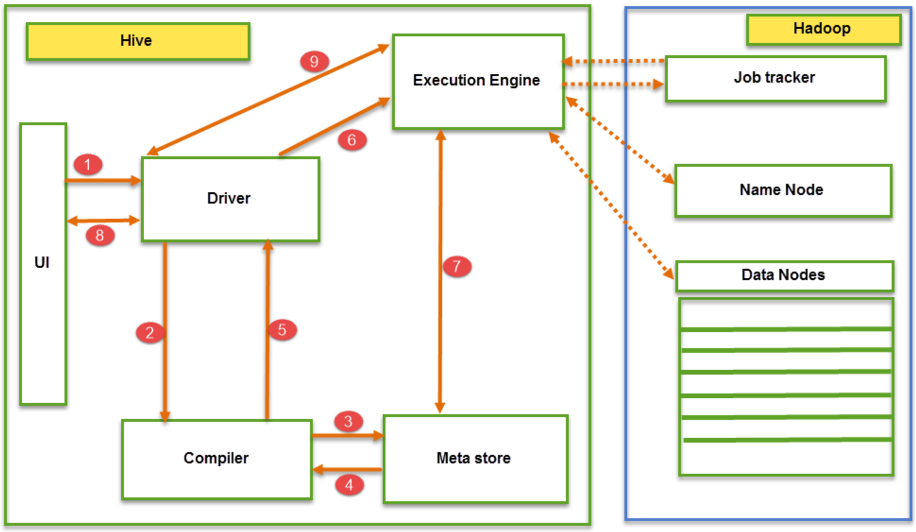
For more information, see WebHCat Installation in the WebHCat manual.

Hive services

Hive services such as Meta store, File system, and Job Client in turn communicates with Hive storage and performs the following actions

- Metadata information of tables created in Hive is stored in Hive "Meta storage database".
- Query results and data loaded in the tables are going to be stored in Hadoop cluster on HDFS.

Job execution flow:



From the above screenshot we can understand the Job execution flow in Hive with Hadoop

The data flow in Hive behaves in the following pattern;

1. Executing Query from the UI(User Interface)
2. The driver is interacting with Compiler for getting the plan. (Here plan refers to query execution) process and its related metadata information gathering
3. The compiler creates the plan for a job to be executed. Compiler communicating with Meta store for getting metadata request
4. Meta store sends metadata information back to compiler
5. Compiler communicating with Driver with the proposed plan to execute the query
6. Driver Sending execution plans to Execution engine
7. Execution Engine (EE) acts as a bridge between Hive and Hadoop to process the query. For DFS operations.

- EE should first contacts Name Node and then to Data nodes to get the values stored in tables.
- EE is going to fetch desired records from Data Nodes. The actual data of tables resides in data node only. While from Name Node it only fetches the metadata information for the query.
- It collects actual data from data nodes related to mentioned query
- Execution Engine (EE) communicates bi-directionally with Meta store present in Hive to perform DDL (Data Definition Language) operations. Here DDL operations like CREATE, DROP and ALTERING tables and databases are done. Meta store will store information about

database name, table names and column names only. It will fetch data related to query mentioned.

- Execution Engine (EE) in turn communicates with Hadoop daemons such as Name node, Data nodes, and job tracker to execute the query on top of Hadoop file system

8. Fetching results from driver

9. Sending results to Execution engine. Once the results fetched from data nodes to the EE, it will send results back to driver and to UI (front end)

Hive Continuously in contact with Hadoop file system and its daemons via Execution engine. The dotted arrow in the Job flow diagram shows the Execution engine communication with Hadoop daemons.

Hive - Data Types

This chapter takes you through the different data types in Hive, which are involved in the table creation. All the data types in Hive are classified into four types, given as follows:

- Column Types
- Literals
- Null Values
- Complex Types

Column Types

Column type are used as column data types of Hive. They are as follows:

Integral Types

Integer type data can be specified using integral data types, INT. When the data range exceeds the range of INT, you need to use BIGINT and if the data range is smaller than the INT, you use SMALLINT. TINYINT is smaller than SMALLINT.

The following table depicts various INT data types:

Type	**Postfix**	**Example**
TINYINT	Y	10Y
SMALLINT	S	10S
INT	-	10
BIGINT	L	10L

String Types

String type data types can be specified using single quotes (' ') or double quotes (" "). It contains two data types: VARCHAR and CHAR. Hive follows C-types escape characters.

The following table depicts various CHAR data types:

Data Type	**Length**
VARCHAR	1 to 65355
CHAR	255

Time stamp

It supports traditional UNIX timestamp with optional nanosecond precision. It supports java.sql.Timestamp format “YYYY-MM-DD HH:MM:SS.fffffffff” and format “yyyy-mm-dd hh:mm:ss.ffffffffff”.

Dates

DATE values are described in year/month/day format in the form {{YYYY-MM-DD}}.

Decimals

The DECIMAL type in Hive is as same as Big Decimal format of Java. It is used for representing immutable arbitrary precision. The syntax and example is as follows:

```
DECIMAL(precision, scale)
decimal(10,0)
```

Union Types

Union is a collection of heterogeneous data types. You can create an instance using **create union**. The syntax and example is as follows:

```
UNIONTYPE<int,double, array<string>,struct<a:int,b:string>>

{0:1}
{1:2.0}
{2:["three","four"]}
{3:{"a":5,"b":"five"}}
{2:["six","seven"]}
{3:{"a":8,"b":"eight"}}
```

```
{0:9}
{1:10.0}
```

Literals

The following literals are used in Hive:

Floating Point Types

Floating point types are nothing but numbers with decimal points. Generally, this type of data is composed of DOUBLE data type.

Decimal Type

Decimal type data is nothing but floating point value with higher range than DOUBLE data type. The range of decimal type is approximately -10^{-308} to 10^{308}.

Null Value

Missing values are represented by the special value NULL.

Complex Types

The Hive complex data types are as follows:

Arrays

Arrays in Hive are used the same way they are used in Java.

```
Syntax: ARRAY<data_type>
```

Maps

Maps in Hive are similar to Java Maps.

```
Syntax: MAP<primitive_type,data_type>
```

Structs

Structs in Hive is similar to using complex data with comment.

```
Syntax: STRUCT<col_name:data_type[COMMENT col_comment],...>
```

Hive - Built-in Functions

This chapter explains the built-in functions available in Hive. The functions look quite similar to SQL functions, except for their usage.

Built-In Functions

Hive supports the following built-in functions:

Return Type	Signature	Description

BIGINT	round(double a)	It returns the rounded BIGINT value of the double.
BIGINT	floor(double a)	It returns the maximum BIGINT value that is equal or less than the double.
BIGINT	ceil(double a)	It returns the minimum BIGINT value that is equal or greater than the double.
double	rand(), rand(int seed)	It returns a random number that changes from row to row.
string	concat(string A, string B,...)	It returns the string resulting from concatenating B after A.
string	substr(string A, int start)	It returns the substring of A starting from start position till the end of string A.
string	substr(string A, int start, int length)	It returns the substring of A starting from start position with the given length.
string	upper(string A)	It returns the string resulting from converting all characters of A to upper case.
string	ucase(string A)	Same as above.
string	lower(string A)	It returns the string resulting from converting all characters of B to lower case.
string	lcase(string A)	Same as above.
string	trim(string A)	It returns the string resulting from trimming spaces from both ends of A.
string	ltrim(string A)	It returns the string resulting from trimming spaces from the beginning (left hand side) of A.
string	rtrim(string A)	rtrim(string A) It returns the string resulting from trimming spaces from the end (right hand side) of A.
string	regexp_replace(string A, string B, string C)	It returns the string resulting from replacing all substrings in B that match the Java regular expression syntax with C.
int	size(Map<K.V>)	It returns the number of elements in the map type.
int	size(Array<T>)	It returns the number of elements in the array type.
value of <type>	cast(<expr> as <type>)	It converts the results of the expression expr to <type> e.g. cast('1' as BIGINT) converts the string '1' to it integral representation. A NULL is returned if the conversion does not succeed.

string	from_unixtime(int unixtime)	convert the number of seconds from Unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the format of "1970-01-01 00:00:00"
string	to_date(string timestamp)	It returns the date part of a timestamp string: to_date("1970-01-01 00:00:00") = "1970-01-01"
int	year(string date)	It returns the year part of a date or a timestamp string: year("1970-01-01 00:00:00") = 1970, year("1970-01-01") = 1970
int	month(string date)	It returns the month part of a date or a timestamp string: month("1970-11-01 00:00:00") = 11, month("1970-11-01") = 11
int	day(string date)	It returns the day part of a date or a timestamp string: day("1970-11-01 00:00:00") = 1, day("1970-11-01") = 1
string	get_json_object(string json_string, string path)	It extracts json object from a json string based on json path specified, and returns json string of the extracted json object. It returns NULL if the input json string is invalid.

Example

The following queries demonstrate some built-in functions:

round() function

```
hive> SELECT round(2.6)from temp;
```

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

3.0

floor() function

```
hive> SELECT floor(2.6)from temp;
```

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

2.0

ceil() function

```
hive> SELECT ceil(2.6)from temp;
```

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

3.0

Aggregate Functions

Hive supports the following built-in **aggregate functions**. The usage of these functions is as same as the SQL aggregate functions.

Return Type	Signature	Description
BIGINT	count(*), count(expr),	count(*) - Returns the total number of retrieved rows.
DOUBLE	sum(col), sum(DISTINCT col)	It returns the sum of the elements in the group or the sum of the distinct values of the column in the group.
DOUBLE	avg(col), avg(DISTINCT col)	It returns the average of the elements in the group or the average of the distinct values of the column in the group.
DOUBLE	min(col)	It returns the minimum value of the column in the group.
DOUBLE	max(col)	It returns the maximum value of the column in the group.

Hive DDL Commands –

Types of DDL Hive Commands

BY DATAFLAIR TEAM · UPDATED · MARCH 4, 2020

Want to run Hive queries for creating, modifying, dropping, altering tables and databases?

In this article, we are going to learn Hive DDL commands. The article describes the Hive Data Definition Language(DDL) commands for performing various operations like creating a table/database in Hive, dropping a table/database in Hive, altering a table/database in Hive, etc. There are many DDL commands. This article will cover each DDL command individually, along with their syntax and examples.

For running Hive DDL commands, you must have Hive installed on your system.

Introduction to Hive DDL commands

Hive DDL commands are the statements used for defining and changing the structure of a table or database in Hive. It is used to build or modify the tables and other objects in the database.

The several types of Hive DDL commands are:

1. CREATE
2. SHOW
3. DESCRIBE
4. USE

5. DROP

6. ALTER

7. TRUNCATE

Table-1 Hive DDL commands

DDL Command	**Use With**
CREATE	Database, Table
SHOW	Databases, Tables, Table Properties, Partitions, Functions, Index
DESCRIBE	Database, Table, view
USE	Database
DROP	Database, Table
ALTER	Database, Table
TRUNCATE	Table

Before moving forward, note that the Hive commands are **case-insensitive**.

CREATE DATABASE is the same as create database.

So now, let us go through each of the commands deeply. Let's start with the DDL commands on Databases in Hive.

DDL Commands On Databases in Hive

1. CREATE DATABASE in Hive

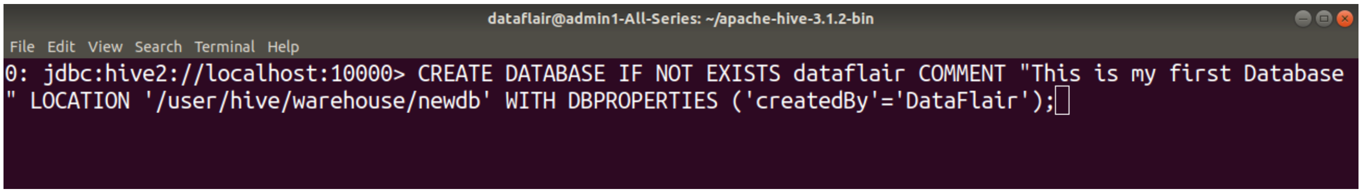
The **CREATE DATABASE** statement is used to create a database in the Hive. The DATABASE and SCHEMA are interchangeable. We can use either DATABASE or SCHEMA.

Syntax:

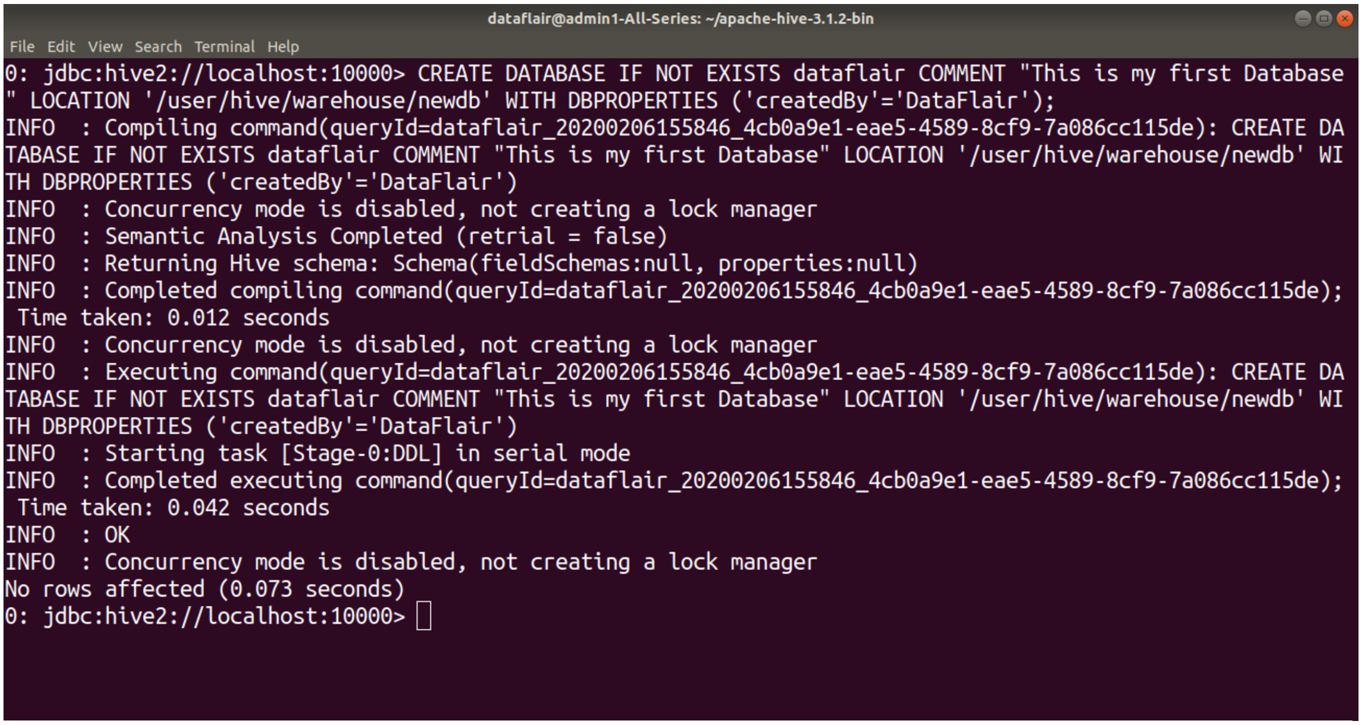
1. **CREATE**(DATABASE|SCHEMA)[IF NOT EXISTS]database_name
2. [COMMENT database_comment]
3. [LOCATION hdfs_path]
4. [WITH **DBPROPERTIES**(property_name=property_value, ...)];

DDL CREATE DATABASE Example:

Here in this example, we are creating a database 'dataflair'.


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> CREATE DATABASE IF NOT EXISTS dataflair COMMENT "This is my first Database
" LOCATION '/user/hive/warehouse/newdb' WITH DBPROPERTIES ('createdBy'='DataFlair');
```


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> CREATE DATABASE IF NOT EXISTS dataflair COMMENT "This is my first Database
" LOCATION '/user/hive/warehouse/newdb' WITH DBPROPERTIES ('createdBy'='DataFlair');
INFO  : Compiling command(queryId=dataflair_20200206155846_4cb0a9e1-eae5-4589-8cf9-7a086cc115de): CREATE DA
TABASE IF NOT EXISTS dataflair COMMENT "This is my first Database" LOCATION '/user/hive/warehouse/newdb' WI
TH DBPROPERTIES ('createdBy'='DataFlair')
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206155846_4cb0a9e1-eae5-4589-8cf9-7a086cc115de);
 Time taken: 0.012 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206155846_4cb0a9e1-eae5-4589-8cf9-7a086cc115de): CREATE DA
TABASE IF NOT EXISTS dataflair COMMENT "This is my first Database" LOCATION '/user/hive/warehouse/newdb' WI
TH DBPROPERTIES ('createdBy'='DataFlair')
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206155846_4cb0a9e1-eae5-4589-8cf9-7a086cc115de);
 Time taken: 0.042 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
No rows affected (0.073 seconds)
0: jdbc:hive2://localhost:10000>
```

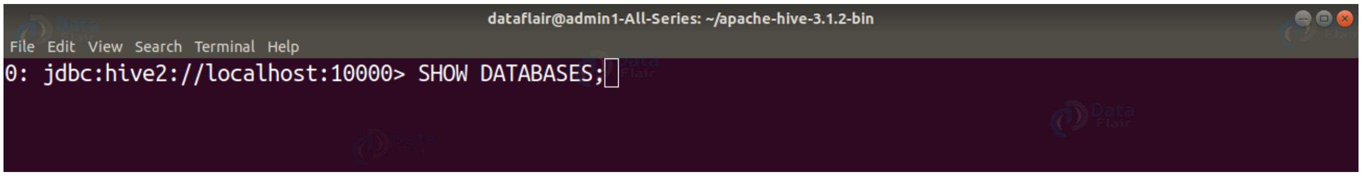
2. SHOW DATABASE in Hive

The **SHOW DATABASES** statement lists all the databases present in the Hive.

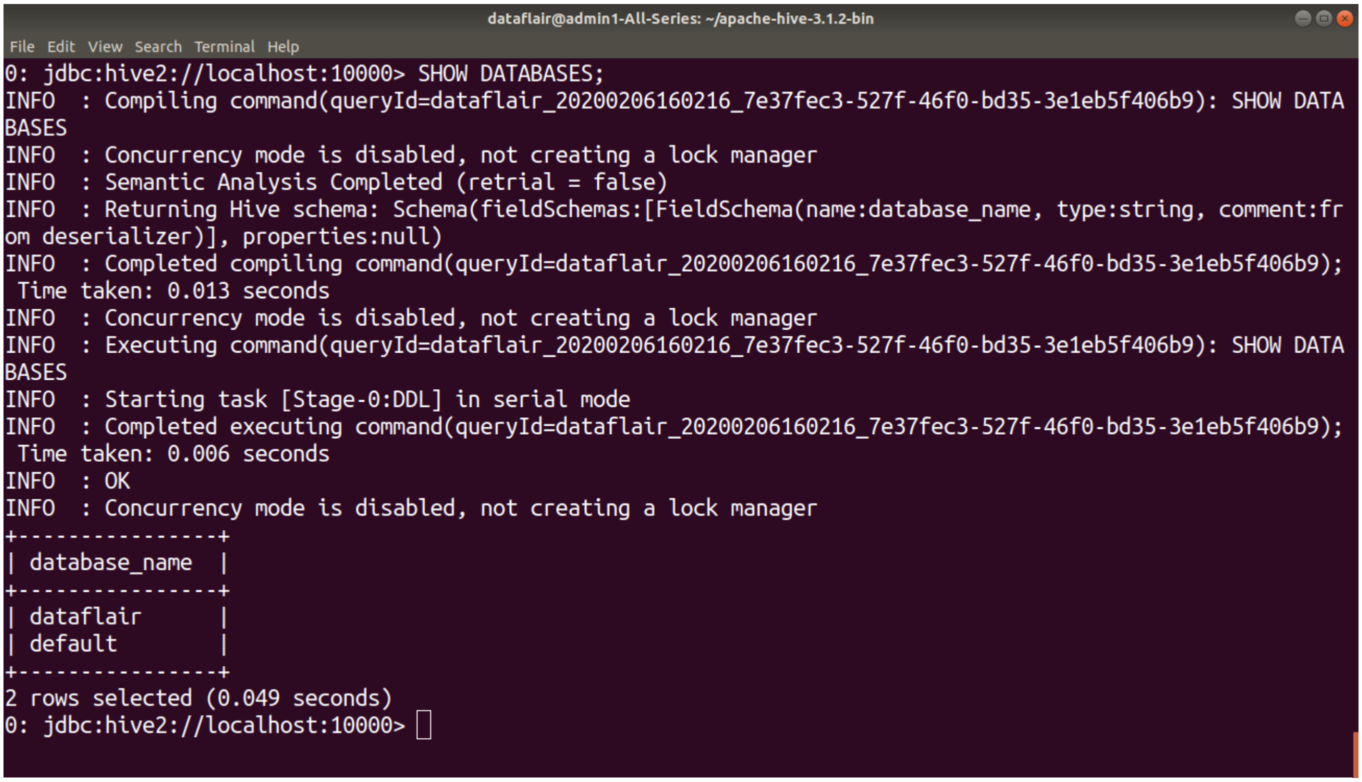
Syntax:

SHOW(DATABASES|SCHEMAS);

DDL SHOW DATABASES Example:


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> SHOW DATABASES;
```


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> SHOW DATABASES;
INFO  : Compiling command(queryId=dataflair_20200206160216_7e37fec3-527f-46f0-bd35-3e1eb5f406b9): SHOW DATA
BASES
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:database_name, type:string, comment:fr
om deserializer)], properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206160216_7e37fec3-527f-46f0-bd35-3e1eb5f406b9);
 Time taken: 0.013 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206160216_7e37fec3-527f-46f0-bd35-3e1eb5f406b9): SHOW DATA
BASES
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206160216_7e37fec3-527f-46f0-bd35-3e1eb5f406b9);
 Time taken: 0.006 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
+----------------+
| database_name  |
+----------------+
| dataflair      |
| default        |
+----------------+
2 rows selected (0.049 seconds)
0: jdbc:hive2://localhost:10000>
```

3. DESCRIBE DATABASE in Hive

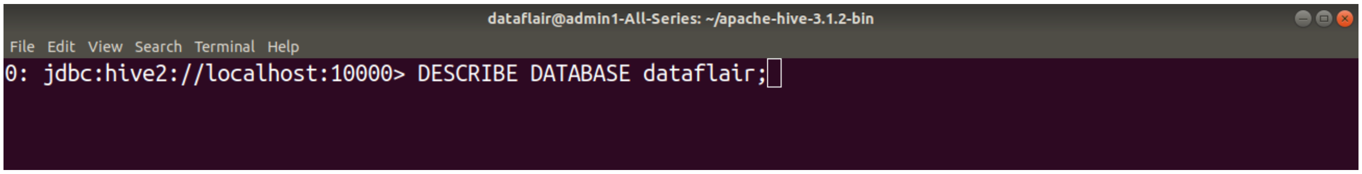
The **DESCRIBE DATABASE** statement in Hive shows the name of Database in Hive, its comment (if set), and its location on the file system.

The **EXTENDED** can be used to get the database properties.

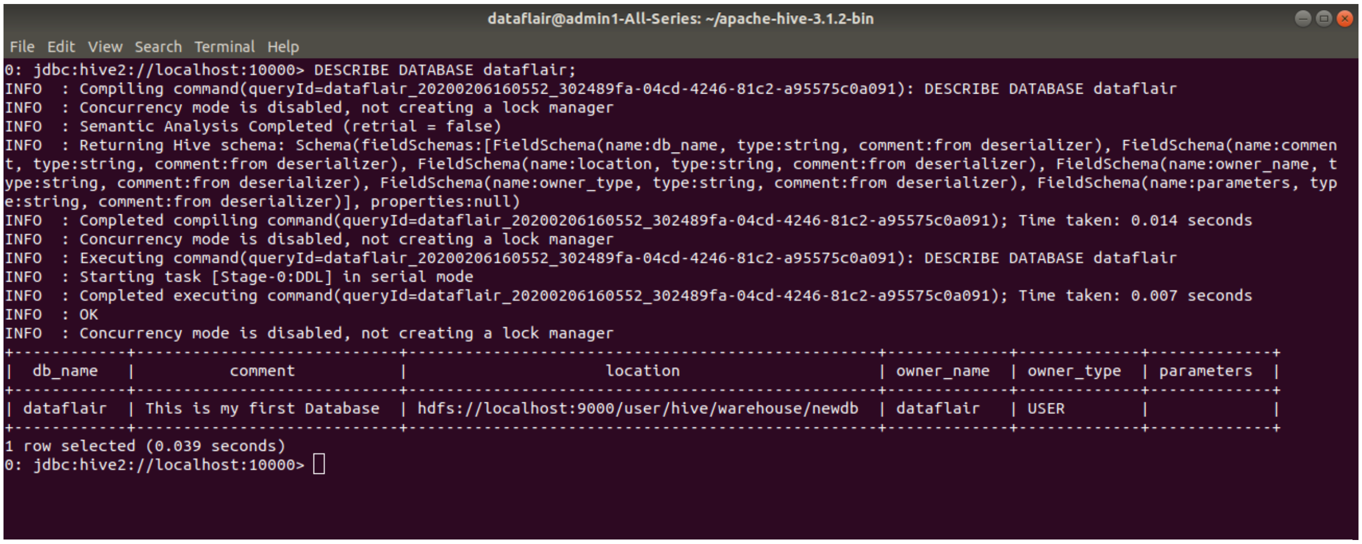
Syntax:

1. DESCRIBE DATABASE/SCHEMA [EXTENDED]db_name;

DDL DESCRIBE DATABASE Example:


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> DESCRIBE DATABASE dataflair;
```



dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> DESCRIBE DATABASE dataflair;
INFO  : Compiling command(queryId=dataflair_20200206160552_302489fa-04cd-4246-81c2-a95575c0a091): DESCRIBE DATABASE dataflair
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:db_name, type:string, comment:from deserializer), FieldSchema(name:commen
t, type:string, comment:from deserializer), FieldSchema(name:location, type:string, comment:from deserializer), FieldSchema(name:owner_name, t
ype:string, comment:from deserializer), FieldSchema(name:owner_type, type:string, comment:from deserializer), FieldSchema(name:parameters, typ
e:string, comment:from deserializer)], properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206160552_302489fa-04cd-4246-81c2-a95575c0a091); Time taken: 0.014 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206160552_302489fa-04cd-4246-81c2-a95575c0a091): DESCRIBE DATABASE dataflair
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206160552_302489fa-04cd-4246-81c2-a95575c0a091); Time taken: 0.007 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
+------------+----------------------------+--------------------------------------------------+-------------+-------------+-------------+
|  db_name   |          comment           |                     location                     | owner_name  | owner_type  | parameters  |
+------------+----------------------------+--------------------------------------------------+-------------+-------------+-------------+
| dataflair  | This is my first Database  | hdfs://localhost:9000/user/hive/warehouse/newdb  | dataflair   | USER        |             |
+------------+----------------------------+--------------------------------------------------+-------------+-------------+-------------+
1 row selected (0.039 seconds)
0: jdbc:hive2://localhost:10000> 
```

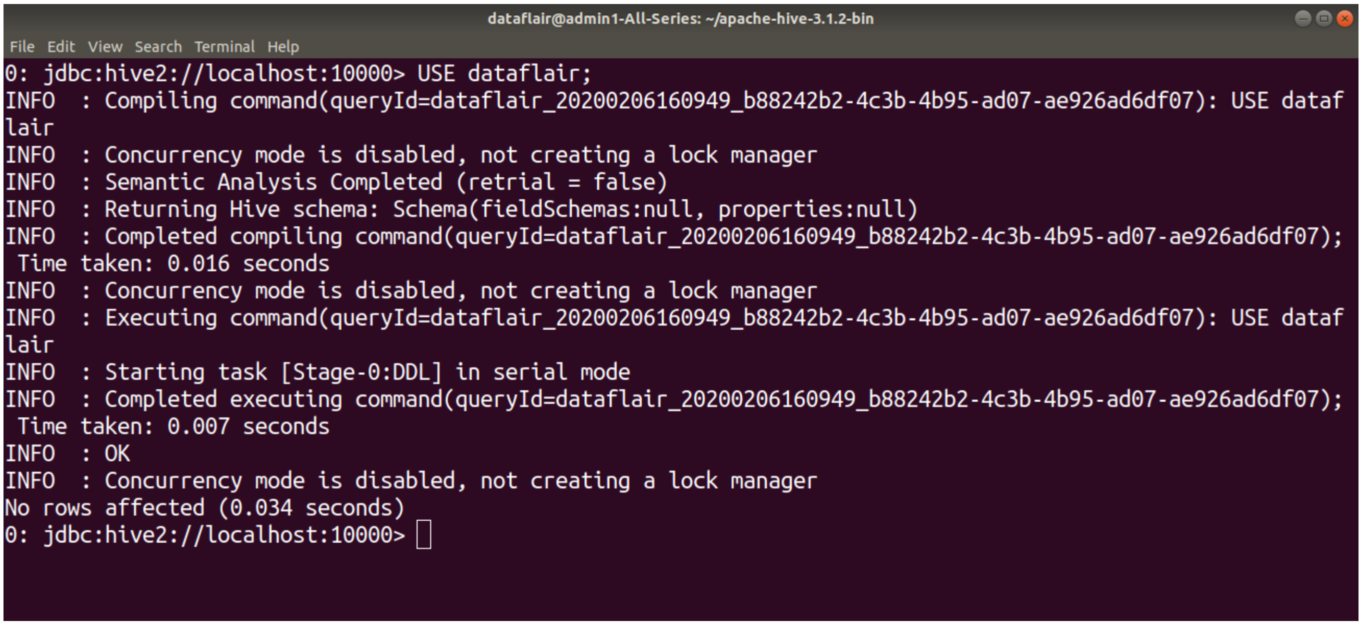
4. USE DATABASE in Hive

The **USE** statement in Hive is used to select the specific database for a session on which all subsequent HiveQL statements would be executed.

Syntax:

1. USE database_name;

DDL USE DATABASE Example:



dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> USE dataflair;
INFO  : Compiling command(queryId=dataflair_20200206160949_b88242b2-4c3b-4b95-ad07-ae926ad6df07): USE dataf
lair
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206160949_b88242b2-4c3b-4b95-ad07-ae926ad6df07);
 Time taken: 0.016 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206160949_b88242b2-4c3b-4b95-ad07-ae926ad6df07): USE dataf
lair
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206160949_b88242b2-4c3b-4b95-ad07-ae926ad6df07);
 Time taken: 0.007 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
No rows affected (0.034 seconds)
0: jdbc:hive2://localhost:10000> 
```

5. DROP DATABASE in Hive

The **DROP DATABASE** statement in Hive is used to Drop (delete) the database.

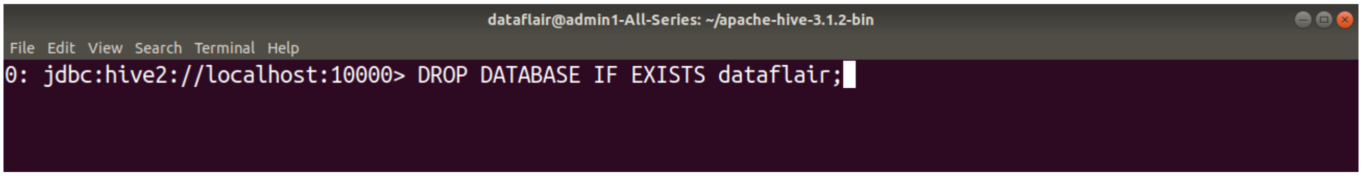
The default behavior is RESTRICT which means that the database is dropped only when it is empty. To drop the database with tables, we can use CASCADE.

Syntax:

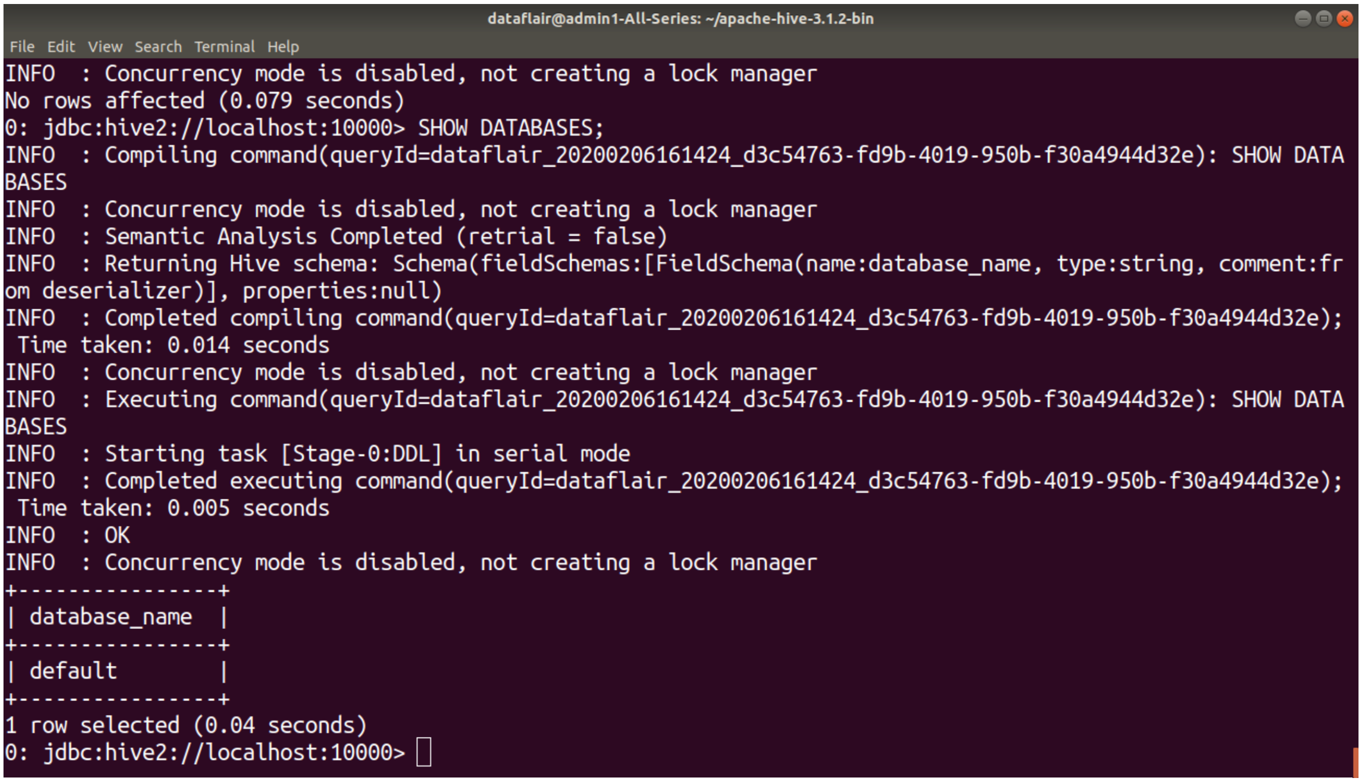
1. **DROP**(DATABASE|SCHEMA)[IF EXISTS]database_name[RESTRICT|CASCADE];

DDL DROP DATABASE Example:

Here in this example, we are dropping a database 'dataflair' using the DROP statement.


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> DROP DATABASE IF EXISTS dataflair;
```


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
INFO  : Concurrency mode is disabled, not creating a lock manager
No rows affected (0.079 seconds)
0: jdbc:hive2://localhost:10000> SHOW DATABASES;
INFO  : Compiling command(queryId=dataflair_20200206161424_d3c54763-fd9b-4019-950b-f30a4944d32e): SHOW DATA
BASES
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:database_name, type:string, comment:fr
om deserializer)], properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206161424_d3c54763-fd9b-4019-950b-f30a4944d32e);
 Time taken: 0.014 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206161424_d3c54763-fd9b-4019-950b-f30a4944d32e): SHOW DATA
BASES
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206161424_d3c54763-fd9b-4019-950b-f30a4944d32e);
 Time taken: 0.005 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
+----------------+
| database_name  |
+----------------+
| default        |
+----------------+
1 row selected (0.04 seconds)
0: jdbc:hive2://localhost:10000>
```

6. ALTER DATABASE in Hive

The **ALTER DATABASE** statement in Hive is used to change the metadata associated with the database in Hive.

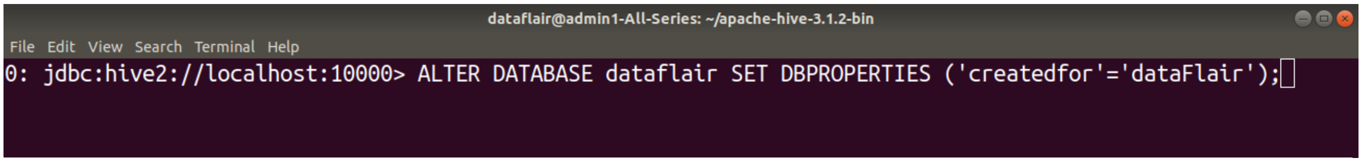
Syntax for changing Database Properties:

ALTER(DATABASE|SCHEMA)database_name

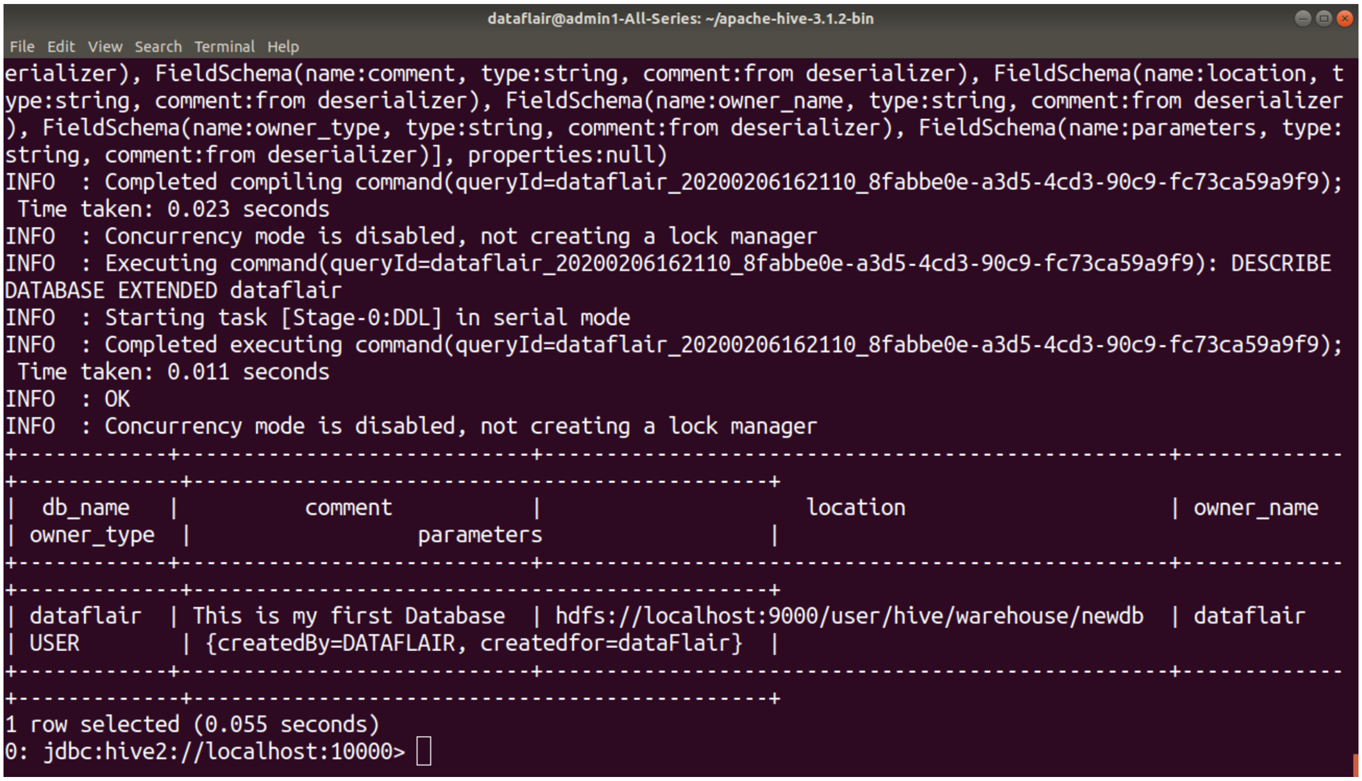
SET **DBPROPERTIES**(property_name=property_value, ...);

DDL ALTER DATABASE properties Example:

In this example, we are setting the database properties of the ‘dataflair’ database after its creation by using the ALTER command.


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> ALTER DATABASE dataflair SET DBPROPERTIES ('createdfor'='dataFlair');
```


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

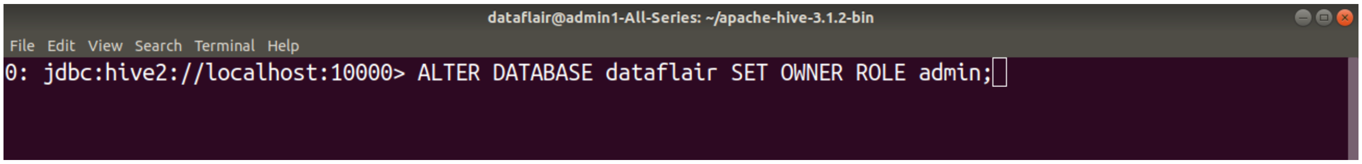
```
erializer), FieldSchema(name:comment, type:string, comment:from deserializer), FieldSchema(name:location, t
ype:string, comment:from deserializer), FieldSchema(name:owner_name, type:string, comment:from deserializer
), FieldSchema(name:owner_type, type:string, comment:from deserializer), FieldSchema(name:parameters, type:
string, comment:from deserializer)], properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206162110_8fabbe0e-a3d5-4cd3-90c9-fc73ca59a9f9);
 Time taken: 0.023 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206162110_8fabbe0e-a3d5-4cd3-90c9-fc73ca59a9f9): DESCRIBE
DATABASE EXTENDED dataflair
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206162110_8fabbe0e-a3d5-4cd3-90c9-fc73ca59a9f9);
 Time taken: 0.011 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
+------------+----------------------------+--------------------------------------------------+------------
+------------+---------------------------------------------+
|  db_name   |          comment           |                     location                     | owner_name
| owner_type |                 parameters                  |
+------------+----------------------------+--------------------------------------------------+------------
+------------+---------------------------------------------+
| dataflair  | This is my first Database  | hdfs://localhost:9000/user/hive/warehouse/newdb  | dataflair
| USER       | {createdBy=DATAFLAIR, createdfor=dataFlair}  |
+------------+----------------------------+--------------------------------------------------+------------
+------------+---------------------------------------------+
1 row selected (0.055 seconds)
0: jdbc:hive2://localhost:10000>
```

Syntax for changing Database owner:

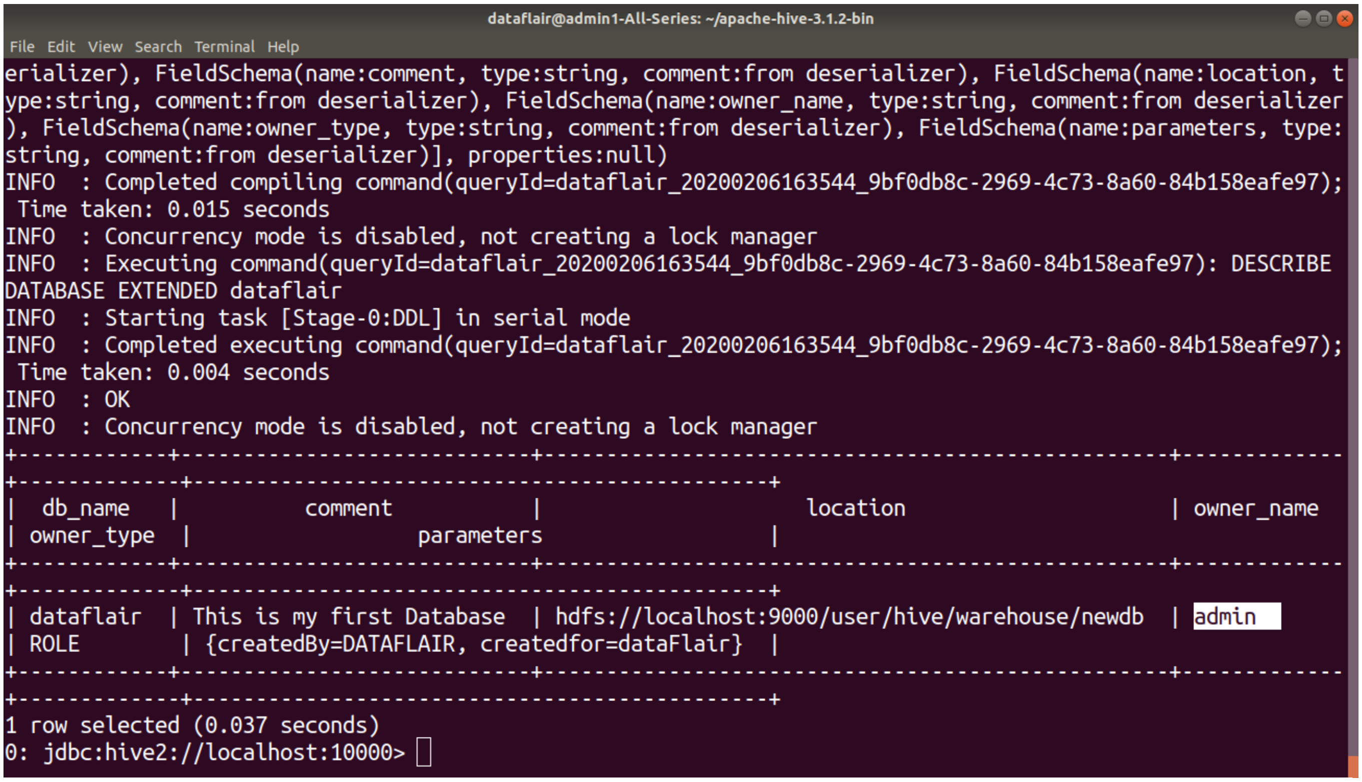
1. **ALTER**(DATABASE|SCHEMA)database_name SET OWNER [USER|ROLE]user_or_role;

DDL ALTER DATABASE owner Example:

In this example, we are changing the owner role of the ‘dataflair’ database using the ALTER statement.


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> ALTER DATABASE dataflair SET OWNER ROLE admin;
```


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
erializer), FieldSchema(name:comment, type:string, comment:from deserializer), FieldSchema(name:location, t
ype:string, comment:from deserializer), FieldSchema(name:owner_name, type:string, comment:from deserializer
), FieldSchema(name:owner_type, type:string, comment:from deserializer), FieldSchema(name:parameters, type:
string, comment:from deserializer)], properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206163544_9bf0db8c-2969-4c73-8a60-84b158eafe97);
 Time taken: 0.015 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206163544_9bf0db8c-2969-4c73-8a60-84b158eafe97): DESCRIBE
DATABASE EXTENDED dataflair
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206163544_9bf0db8c-2969-4c73-8a60-84b158eafe97);
 Time taken: 0.004 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
+------------+----------------------------+--------------------------------------------------+-------------
+-------------+----------------------------------------------+
|  db_name   |          comment           |                     location                     | owner_name
| owner_type  |                parameters                    |
+------------+----------------------------+--------------------------------------------------+-------------
+-------------+----------------------------------------------+
| dataflair  | This is my first Database  | hdfs://localhost:9000/user/hive/warehouse/newdb  | admin      
| ROLE        | {createdBy=DATAFLAIR, createdfor=dataFlair}  |
+------------+----------------------------+--------------------------------------------------+-------------
+-------------+----------------------------------------------+
1 row selected (0.037 seconds)
0: jdbc:hive2://localhost:10000> 
```

Syntax for changing Database Location:

ALTER(DATABASE|SCHEMA)database_name SET LOCATION hdfs_path;

Note: The ALTER DATABASE … SET LOCATION statement does not move the database current directory contents to the newly specified location. This statement does not change the locations associated with any tables or partitions under the specified database. Instead, it changes the default parent-directory, where new tables will be added for this database.

No other metadata associated with the database can be changed.

DDL Commands on Tables in Hive

CREATE TABLE

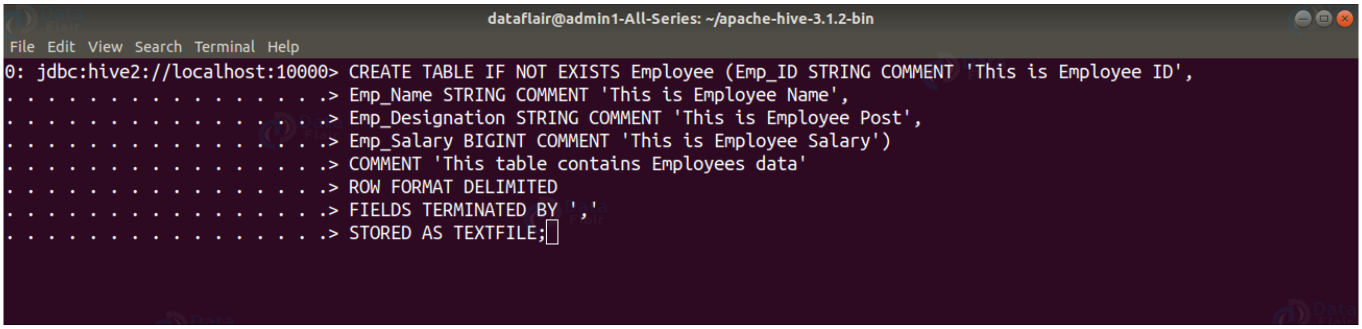
The **CREATE TABLE** statement in Hive is used to create a table with the given name. If a table or view already exists with the same name, then the error is thrown. We can use **IF NOT EXISTS** to skip the error.

Syntax:

CREATE TABLE [IF NOT EXISTS][db_name.]table_name[(col_namedata_type[COMMENT col_comment], ... [COMMENT col_comment])][COMMENT table_comment][ROW FORMAT row_format][STORED AS file_format][LOCATION hdfs_path];

DDL CREATE TABLE Example:

In this table, we are creating a table 'Employee' in the 'dataflair' database.



dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> CREATE TABLE IF NOT EXISTS Employee (Emp_ID STRING COMMENT 'This is Employee ID',
. . . . . . . . . . . . . . . .> Emp_Name STRING COMMENT 'This is Employee Name',
. . . . . . . . . . . . . . . .> Emp_Designation STRING COMMENT 'This is Employee Post',
. . . . . . . . . . . . . . . .> Emp_Salary BIGINT COMMENT 'This is Employee Salary')
. . . . . . . . . . . . . . . .> COMMENT 'This table contains Employees data'
. . . . . . . . . . . . . . . .> ROW FORMAT DELIMITED
. . . . . . . . . . . . . . . .> FIELDS TERMINATED BY ','
. . . . . . . . . . . . . . . .> STORED AS TEXTFILE;
```

ROW FORMAT DELIMITED means we are telling the Hive that when it finds a new line character, that means a new record.

FIELDS TERMINATED BY ',' tells Hive what delimiter we are using in our files to separate each column.

STORED AS TEXTFILE is to tell Hive what type of file to expect.

Don't know about different Data Types supported by hive? Read **Hive Data Types** article.

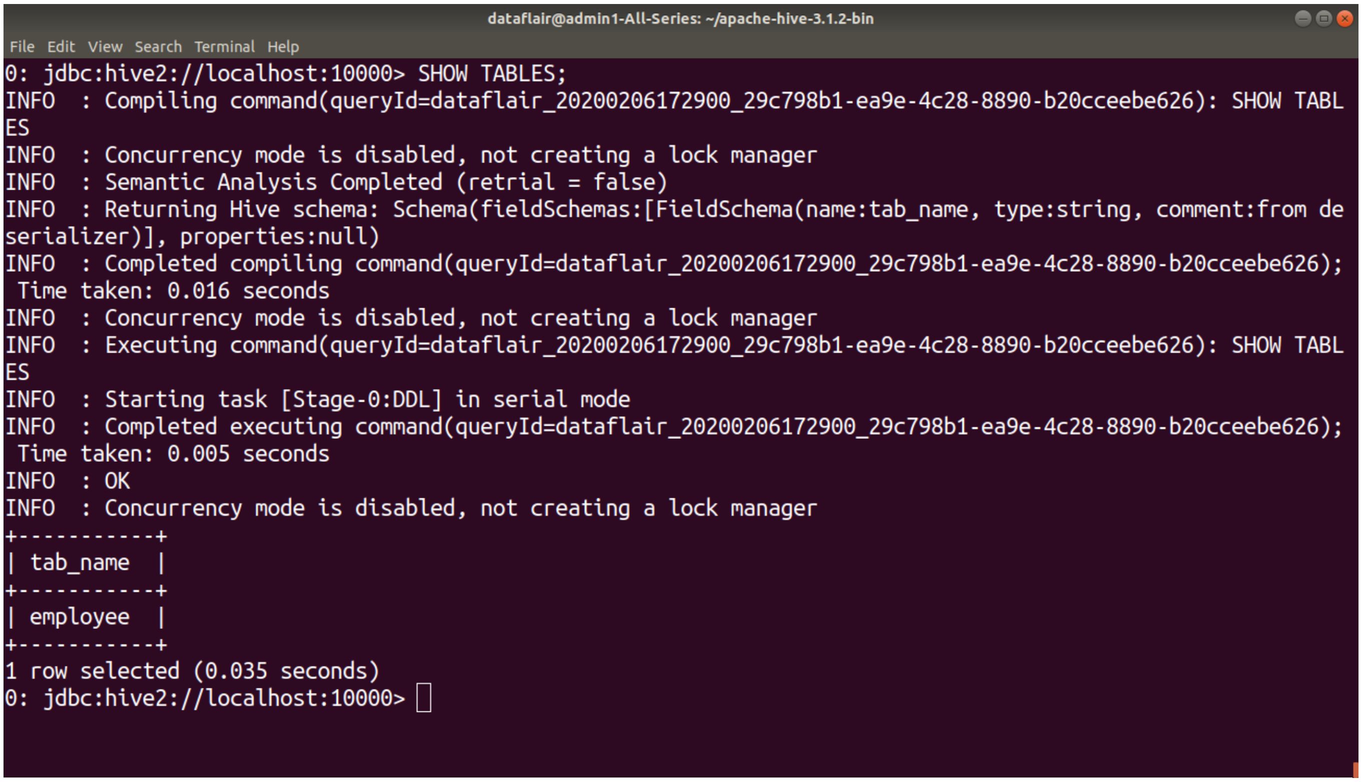
2. SHOW TABLES in Hive

The **SHOW TABLES** statement in Hive lists all the base tables and **views** in the current database.

Syntax:

1. SHOW TABLES [IN database_name];

DDL SHOW TABLES Example:


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> SHOW TABLES;
INFO  : Compiling command(queryId=dataflair_20200206172900_29c798b1-ea9e-4c28-8890-b20cceebe626): SHOW TABL
ES
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:tab_name, type:string, comment:from de
serializer)], properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206172900_29c798b1-ea9e-4c28-8890-b20cceebe626);
 Time taken: 0.016 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206172900_29c798b1-ea9e-4c28-8890-b20cceebe626): SHOW TABL
ES
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206172900_29c798b1-ea9e-4c28-8890-b20cceebe626);
 Time taken: 0.005 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
+-----------+
| tab_name  |
+-----------+
| employee  |
+-----------+
1 row selected (0.035 seconds)
0: jdbc:hive2://localhost:10000>
```

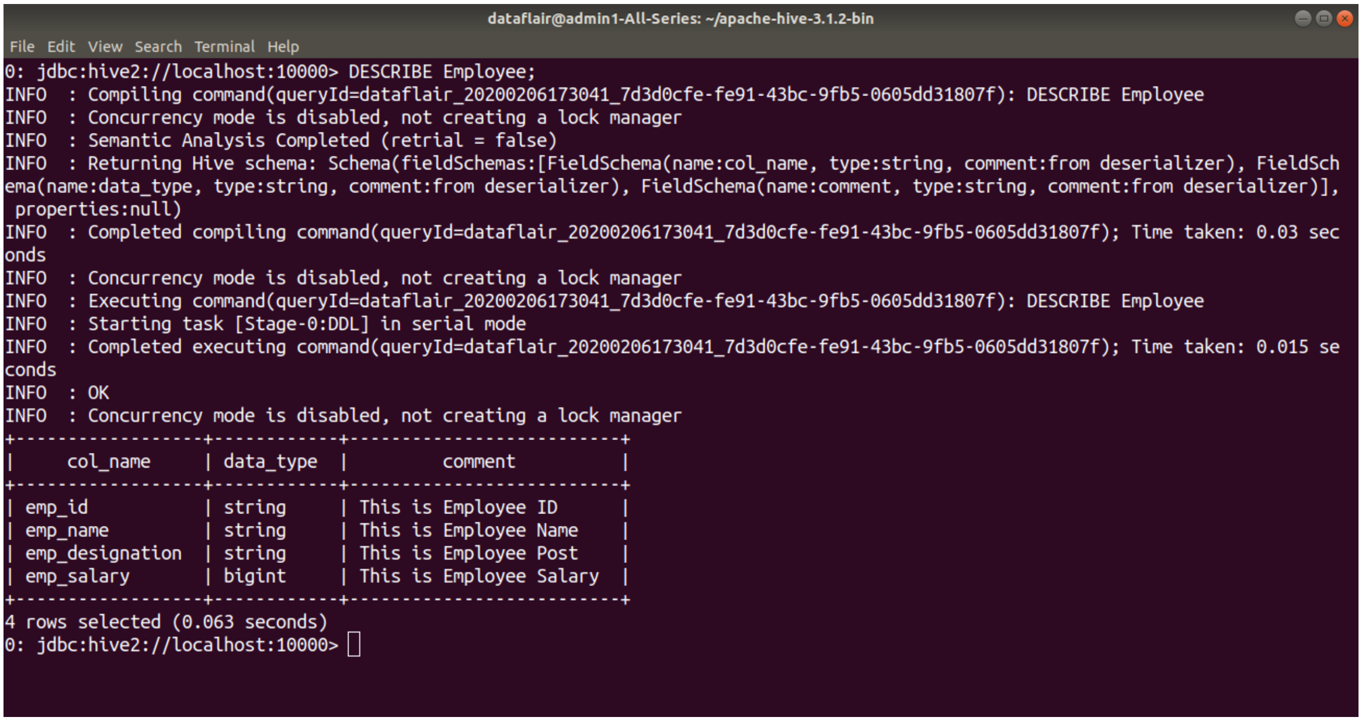
3. DESCRIBE TABLE in Hive

The **DESCRIBE** statement in Hive shows the lists of columns for the specified table.

Syntax:

1. DESCRIBE [EXTENDED|FORMATTED][db_name.]table_name[.**col_name**([.field_name])];

DDL DESCRIBE TABLE Example:


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> DESCRIBE Employee;
INFO  : Compiling command(queryId=dataflair_20200206173041_7d3d0cfe-fe91-43bc-9fb5-0605dd31807f): DESCRIBE Employee
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:col_name, type:string, comment:from deserializer), FieldSch
ema(name:data_type, type:string, comment:from deserializer), FieldSchema(name:comment, type:string, comment:from deserializer)],
 properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206173041_7d3d0cfe-fe91-43bc-9fb5-0605dd31807f); Time taken: 0.03 sec
onds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206173041_7d3d0cfe-fe91-43bc-9fb5-0605dd31807f): DESCRIBE Employee
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206173041_7d3d0cfe-fe91-43bc-9fb5-0605dd31807f); Time taken: 0.015 se
conds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
+------------------+------------+-------------------------+
|     col_name     | data_type  |         comment         |
+------------------+------------+-------------------------+
| emp_id           | string     | This is Employee ID     |
| emp_name         | string     | This is Employee Name   |
| emp_designation  | string     | This is Employee Post   |
| emp_salary       | bigint     | This is Employee Salary |
+------------------+------------+-------------------------+
4 rows selected (0.063 seconds)
0: jdbc:hive2://localhost:10000> 
```

DROP TABLE in Hive

The **DROP TABLE** statement in Hive deletes the data for a particular table and remove all metadata associated with it from Hive metastore.

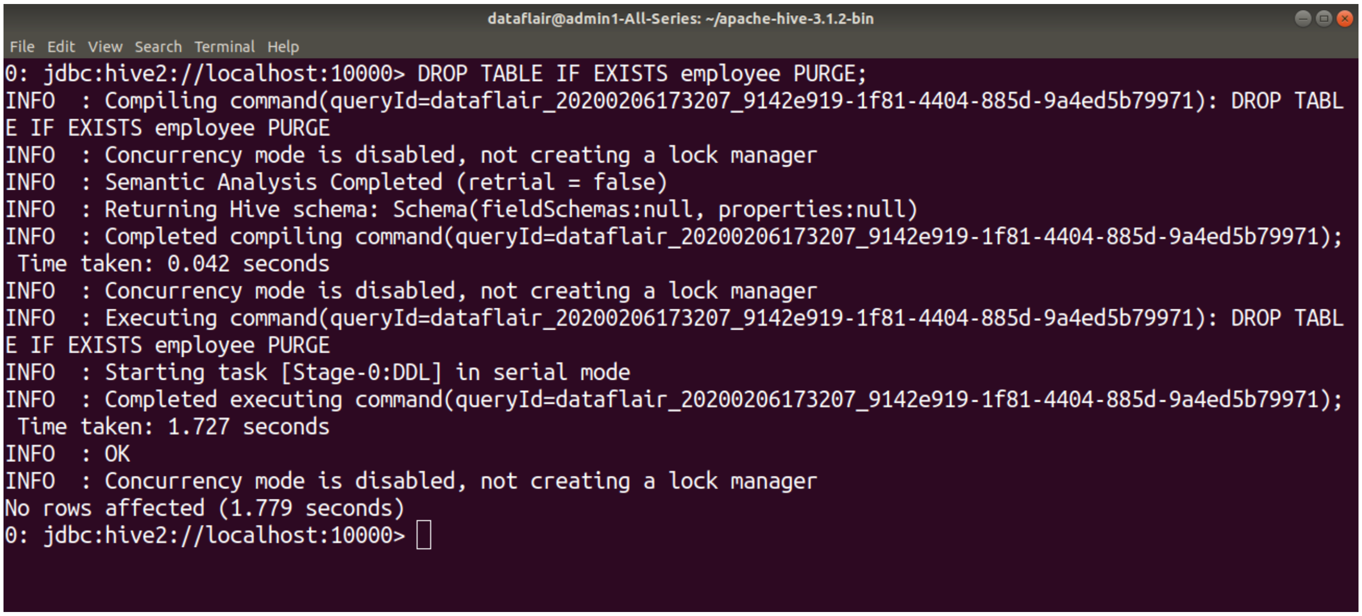
If **PURGE** is not specified then the data is actually moved to the .Trash/current directory. If **PURGE** is specified, then data is lost completely.

Syntax:

DROP TABLE [IF EXISTS]table_name[PURGE];

DDL DROP TABLE Example:

In the below example, we are deleting the ‘employee’ table.



dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> DROP TABLE IF EXISTS employee PURGE;
INFO  : Compiling command(queryId=dataflair_20200206173207_9142e919-1f81-4404-885d-9a4ed5b79971): DROP TABL
E IF EXISTS employee PURGE
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206173207_9142e919-1f81-4404-885d-9a4ed5b79971);
 Time taken: 0.042 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206173207_9142e919-1f81-4404-885d-9a4ed5b79971): DROP TABL
E IF EXISTS employee PURGE
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206173207_9142e919-1f81-4404-885d-9a4ed5b79971);
 Time taken: 1.727 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
No rows affected (1.779 seconds)
0: jdbc:hive2://localhost:10000>
```

ALTER TABLE in Hive

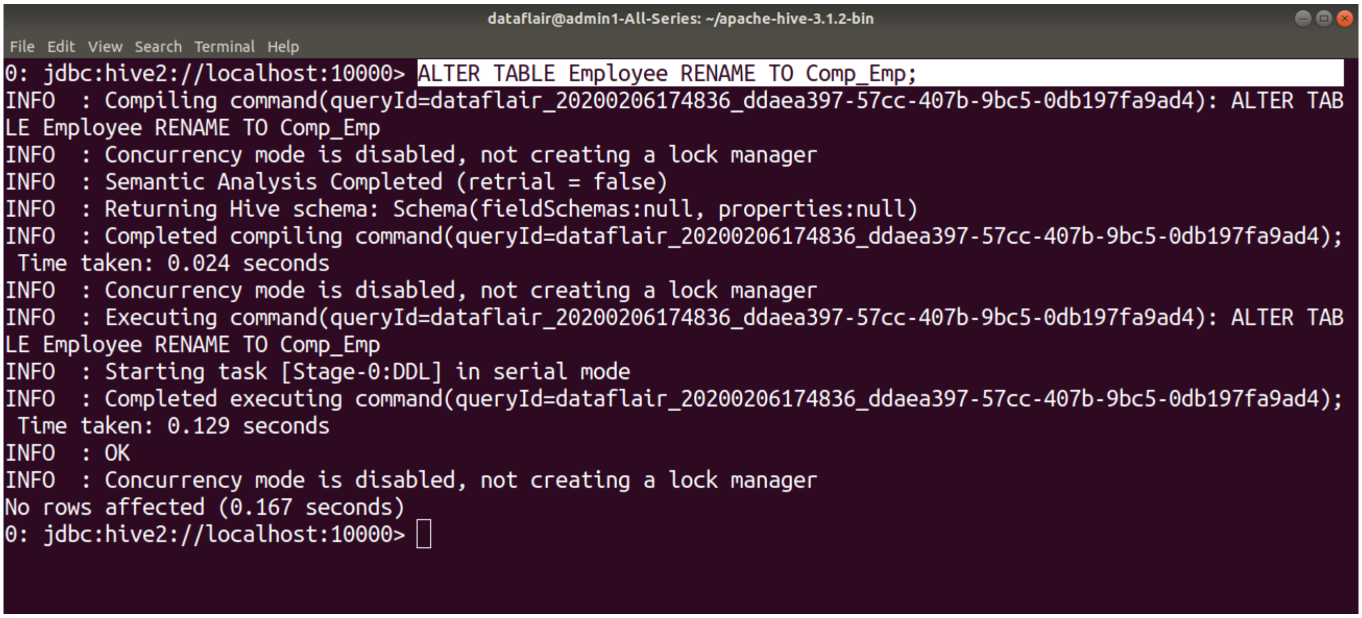
The **ALTER TABLE** statement in Hive enables you to change the structure of an existing table. Using the ALTER TABLE statement we can rename the table, add columns to the table, change the table properties, etc.

Syntax to Rename a table:

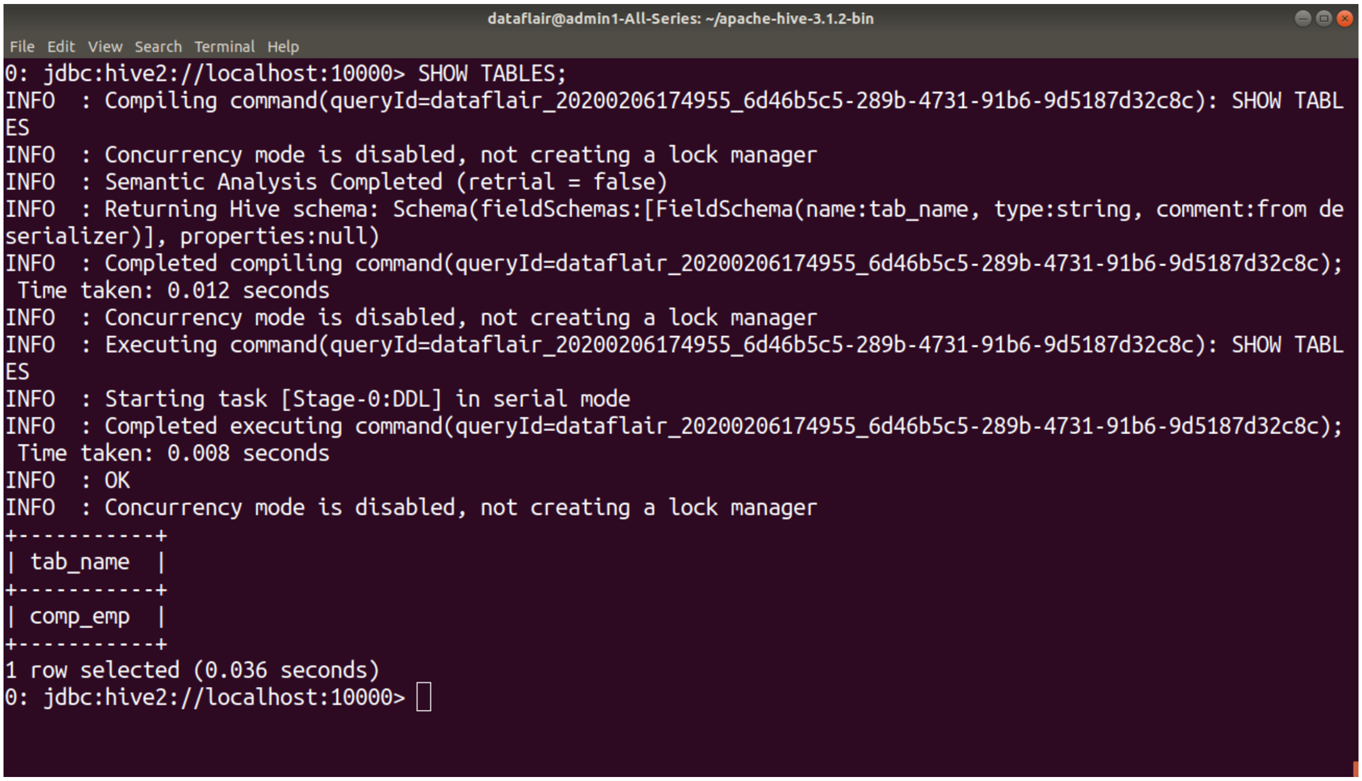
ALTER TABLE table_name RENAME TO new_table_name;

DDL ALTER TABLE name Example:

In this example, we are trying to rename the 'Employee' table to 'Com_Emp' using the ALTER statement.


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> ALTER TABLE Employee RENAME TO Comp_Emp;
INFO  : Compiling command(queryId=dataflair_20200206174836_ddaea397-57cc-407b-9bc5-0db197fa9ad4): ALTER TAB
LE Employee RENAME TO Comp_Emp
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206174836_ddaea397-57cc-407b-9bc5-0db197fa9ad4);
 Time taken: 0.024 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206174836_ddaea397-57cc-407b-9bc5-0db197fa9ad4): ALTER TAB
LE Employee RENAME TO Comp_Emp
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206174836_ddaea397-57cc-407b-9bc5-0db197fa9ad4);
 Time taken: 0.129 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
No rows affected (0.167 seconds)
0: jdbc:hive2://localhost:10000> 
```


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

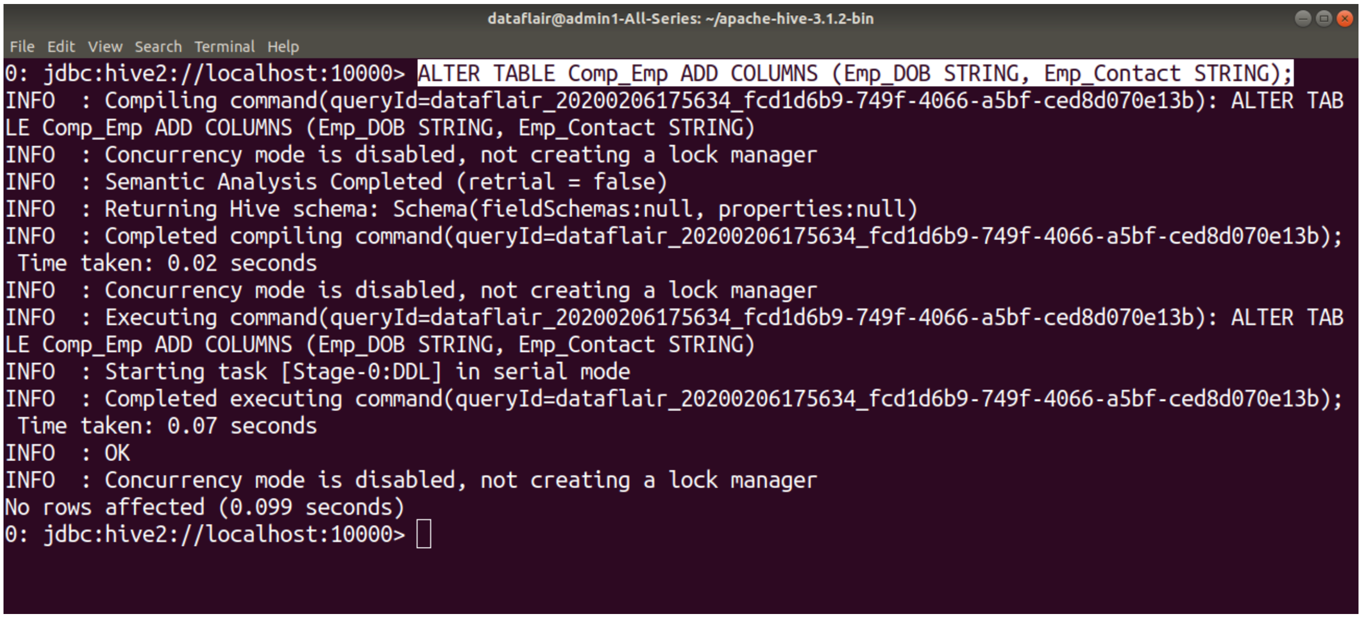
```
0: jdbc:hive2://localhost:10000> SHOW TABLES;
INFO  : Compiling command(queryId=dataflair_20200206174955_6d46b5c5-289b-4731-91b6-9d5187d32c8c): SHOW TABL
ES
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:tab_name, type:string, comment:from de
serializer)], properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206174955_6d46b5c5-289b-4731-91b6-9d5187d32c8c);
 Time taken: 0.012 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206174955_6d46b5c5-289b-4731-91b6-9d5187d32c8c): SHOW TABL
ES
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206174955_6d46b5c5-289b-4731-91b6-9d5187d32c8c);
 Time taken: 0.008 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
+-----------+
| tab_name  |
+-----------+
| comp_emp  |
+-----------+
1 row selected (0.036 seconds)
0: jdbc:hive2://localhost:10000> 
```

Syntax to Add columns to a table:

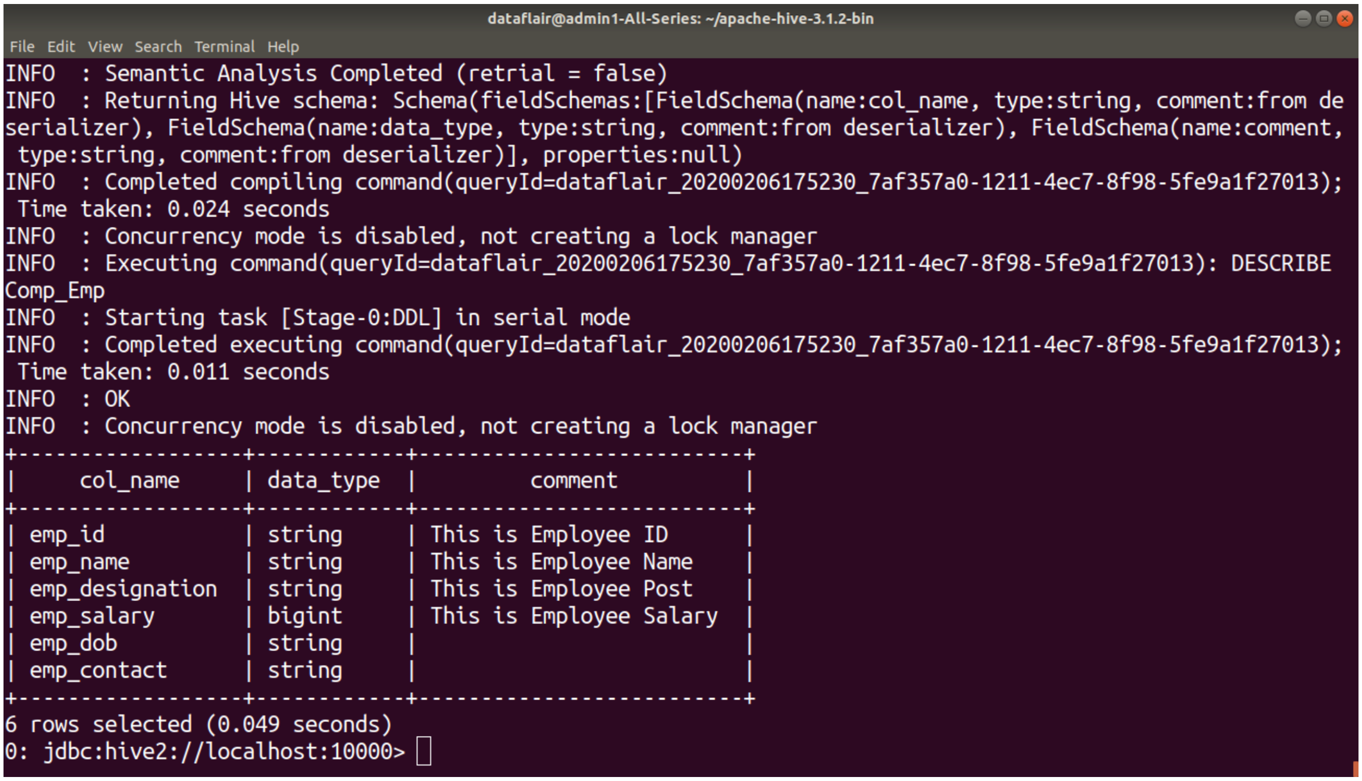
ALTER TABLE table_name ADD **COLUMNS**(column1, column2) ;

DDL ALTER TABLE columns Example:

In this example, we are adding two columns ‘Emp_DOB’ and ‘Emp_Contact’ in the ‘Comp_Emp’ table using the ALTER command.


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> ALTER TABLE Comp_Emp ADD COLUMNS (Emp_DOB STRING, Emp_Contact STRING);
INFO  : Compiling command(queryId=dataflair_20200206175634_fcd1d6b9-749f-4066-a5bf-ced8d070e13b): ALTER TAB
LE Comp_Emp ADD COLUMNS (Emp_DOB STRING, Emp_Contact STRING)
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206175634_fcd1d6b9-749f-4066-a5bf-ced8d070e13b);
 Time taken: 0.02 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206175634_fcd1d6b9-749f-4066-a5bf-ced8d070e13b): ALTER TAB
LE Comp_Emp ADD COLUMNS (Emp_DOB STRING, Emp_Contact STRING)
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206175634_fcd1d6b9-749f-4066-a5bf-ced8d070e13b);
 Time taken: 0.07 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
No rows affected (0.099 seconds)
0: jdbc:hive2://localhost:10000> 
```


dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

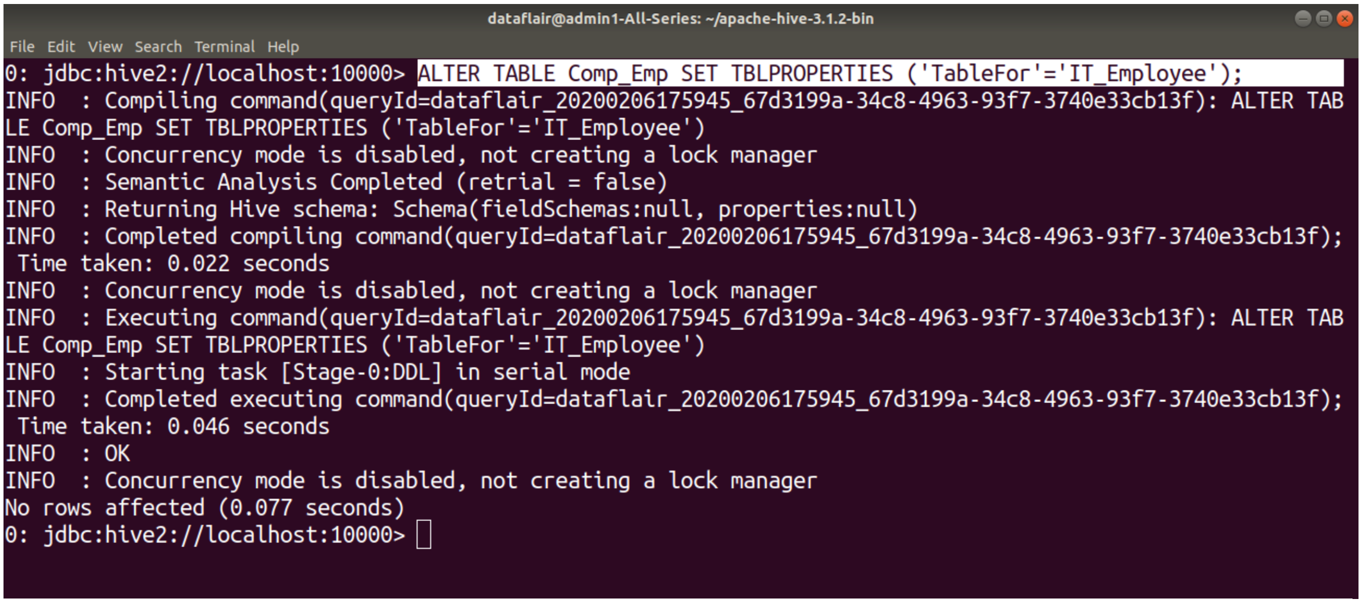
```
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:col_name, type:string, comment:from de
serializer), FieldSchema(name:data_type, type:string, comment:from deserializer), FieldSchema(name:comment,
 type:string, comment:from deserializer)], properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206175230_7af357a0-1211-4ec7-8f98-5fe9a1f27013);
 Time taken: 0.024 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206175230_7af357a0-1211-4ec7-8f98-5fe9a1f27013): DESCRIBE
Comp_Emp
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206175230_7af357a0-1211-4ec7-8f98-5fe9a1f27013);
 Time taken: 0.011 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
+------------------+-----------+-------------------------+
|     col_name     | data_type |         comment         |
+------------------+-----------+-------------------------+
| emp_id           | string    | This is Employee ID     |
| emp_name         | string    | This is Employee Name   |
| emp_designation  | string    | This is Employee Post   |
| emp_salary       | bigint    | This is Employee Salary |
| emp_dob          | string    |                         |
| emp_contact      | string    |                         |
+------------------+-----------+-------------------------+
6 rows selected (0.049 seconds)
0: jdbc:hive2://localhost:10000> 
```

Syntax to set table properties:

ALTER TABLE table_name SET **TBLPROPERTIES**(‘property_key’=’property_new_value’);

DDL ALTER TABLE properties Example:

In this example, we are setting the table properties after table creation by using ALTER command.



dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> ALTER TABLE Comp_Emp SET TBLPROPERTIES ('TableFor'='IT_Employee');
INFO  : Compiling command(queryId=dataflair_20200206175945_67d3199a-34c8-4963-93f7-3740e33cb13f): ALTER TAB
LE Comp_Emp SET TBLPROPERTIES ('TableFor'='IT_Employee')
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=dataflair_20200206175945_67d3199a-34c8-4963-93f7-3740e33cb13f);
 Time taken: 0.022 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=dataflair_20200206175945_67d3199a-34c8-4963-93f7-3740e33cb13f): ALTER TAB
LE Comp_Emp SET TBLPROPERTIES ('TableFor'='IT_Employee')
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=dataflair_20200206175945_67d3199a-34c8-4963-93f7-3740e33cb13f);
 Time taken: 0.046 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
No rows affected (0.077 seconds)
0: jdbc:hive2://localhost:10000>
```

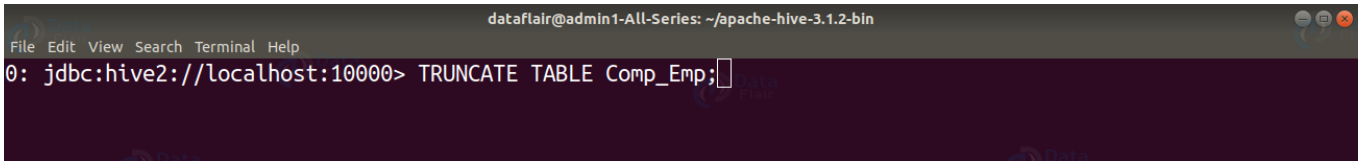
TRUNCATE TABLE

TRUNCATE TABLE statement in Hive removes all the rows from the table or partition.

Syntax:

TRUNCATE TABLE table_name;

DDL TRUNCATE TABLE Example:



dataflair@admin1-All-Series: ~/apache-hive-3.1.2-bin
File Edit View Search Terminal Help

```
0: jdbc:hive2://localhost:10000> TRUNCATE TABLE Comp_Emp;
```