GUDLAVALLERU ENGINEERING COLLEGE

(An Autonomous Institute with Permanent Affiliation to JNTUK, Kakinada)

Seshadri Rao Knowledge Village, Gudlavalleru– 521 356.

Department of Computer Science and Engineering

HANDOUT

on

BIG DATA

Vision

To be a center of excellence in Computer Science and Engineering education and training to meet the challenging needs of the industry and society.

Mission

- To impart quality education through well-designed curriculum in tune with the growing software needs of the industry.
- To serve our students by inculcating in them problem solving, leadership, teamwork skills and the value of commitment to quality, ethical behavior & respect for others.
- To foster industry-academia relationship for mutual benefit and growth

Program Educational Objectives

PEO1: Identify, analyze, formulate and solve Computer Science and Engineering problems both independently and in a team environment by using the appropriate modern tools.

PEO2: Manage software projects with significant technical, legal, ethical, social, environmental and economic considerations

PEO3: Demonstrate commitment and progress in lifelong learning, professional development, leadership and Communicate effectively with professional clients and the public.

HANDOUT ON BIGDATA

Class & Sem. :	IV B.Tech–II Semester	Year :	2019-20
Branch :	CSE	Credits :	3

== ==

1. Brief History and Scope of the Subject

Hadoop is an open -source software framework for distributed storage and large -scale processing of data -sets on clusters of commodity hardware.

In 2004 Google publishes Google File System (GFS) and MapReduce framework papers.2005 Doug Cutting and Nutch team implemented Google's frameworks in Nutch 2006 Yahoo hires Doug Cutting to work on Hadoop with dedicated team

2008 Hadoop became Apache Top Level Project. The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), and a processing part called MapReduce. Hadoop splits files into large blocks and distributes them across no des in a cluster. To process data, Hadoop transfers packaged code for nodes to process in parallel based on the data that needs to be processed.

The base Apache Hadoop framework is compo sed of the following modules:

- Hadoop Common– contains libraries and utilities needed by other Hadoop modules;
- Hadoop Distributed File System (HDFS)– a distributed file -system that stores data o n commodity machines, providing very high aggregate bandwidth across the cluster.
- Hadoop YARN–a resource -management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications;
- Hadoop MapReduce – an implementation of the MapReduce programming model for large scale data processing.

The term Hadoop has come to refer not just to the base modules above, but also to the ecosystem, or collection of additional software packages that can be installed on top of or alongside Hadoop, such as Apache Pig, Apache Hive,
Apache HBase, Apache Phoenix, Apache Spark, Apache Zookeeper, Cloudera Impala, Apache Flume, Apache Sqoop, Apache Oozie, Apache Storm. The Hadoop framework itself is mostly written in the Java programming language, with some native code in C and command line utilities written as shell scripts.

Big data is the term for a collection of data sets so large and complex that it becomes difficult to store and process using on-hand database management tools or traditional data processing applications.

Technologies Supported by Big Data

- Column-oriented databases, Schema-less databases, or NoSQL databases, MapReduce, this is a programming paradigm. Hadoop open source platform for handling Big Data.

- Hive is a "SQL-like" bridge that allows conventional BI applications to run queries against a Hadoop cluster. PIG is another bridge that tries to bring Hadoop closer to the realities of developers and business users, similar to Hive. Unlike Hive, however, PIG consists of a "Perl-like" language that allows for query execution over data stored on a Hadoop cluster, instead of a "SQL -like" language.

Storage Technologies

- Big Data in the cloud

- Big Data and cloud computing go hand-in-hand. Cloud computing enable s companies of all sizes to get more value from their data than ever before, by enabling blazing -fast analytics at a fraction of previous costs. This, in turn drives companies to acquire and store even more data, creating more need for processing power and driving a virtuous circle.

Pre-Requisites

Students should have Basic knowledge of JAVA, Python, Linux and SQL

2. Course Objectives:

- To familiarize the fundamental concepts of cloud for laying a strong foundation of Apache Hadoop (Big data framework).
- To gain knowledge of HDF file system, MapReduce frameworks and relevant tools.

3. Course Outcomes:

Student will be able to

CO1: Describe the fundamentals of Big cloud and data architectures

CO2: Use HDFS file structure and MapReduce frameworks to solve complex problems

CO3: Know how to analyze data using Unix tools and Hadoop

CO4: Understand how to develop environment for analyzing Bigdata

CO5: Understand how to use mapper and reducer functions

CO6: Access the database in a Hadoop environment using Hive

4. Program Outcomes

Graduates of the Computer Science and Engineering Program will have ability to Engineering graduate will be able to

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and engg. specialization to the solution of complex engineering problems.

2. **Problem analysis**: Identify, formulate, research literature, and analyze engineering problems to arrive at substantiated conclusions using first principles of mathematics, natural, and engineering sciences.

3. **Design/ development of solutions**: Design solutions for complex engineering problems and design system components, processes to meet the specifications with consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems**: Use research-based knowledge including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development

8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in teams, and in multidisciplinary settings.

10. **Communication**: Communicate effectively with the engineering community and with society at large. Be able to comprehend and write effective reports documentation. Make effective presentations, and give and receive clear instructions.

11. **Project management and finance**: Demonstrate knowledge and understanding of engineering and management principles and apply these to one's own work, as a member and leader in a team. Manage projects in multidisciplinary environments.

12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

6. Mapping of Course Outcomes with Program Outcomes:

	1	2	3	4	5	6	7	8	9	10	11	12
CO1	3		2									
CO2	2	2	3	3								
CO3			1	2	2							
CO4			3	2	3							
CO5	2	2	3	2	2							
CO6	1		2	2	3							

3- High Level Mapping **2-** Medium Level Mapping **1-**Low Level Mapping

7. Prescribed Text Books

1. Tom White, Hadoop, “The Definitive Guide”, 3rd. Edition, O’Reilly Publications, 2012.
2. DrikdeRoos, Chris Eaton, George Lapis, Paul Zikopoulos, Tom Deutsch, “Understanding Big Data Analytics for Enterprise Class Hadoop and Streaming Data”,1st Edition, TMH,2012.

8. Reference Text Books

1. Frank J.Ohlhorst,“Big Data Analystics:Turning Big Data Into Big Money”,2Nd Edition, TMH, 2012.

9. URLs and Other E -Learning Resources

a. Hadoop: http:/ / hadoop.apache.org/

b. https://drive.google.com/drive/folders/1CSiyqbRvT65XZ309CJiVxJBrrZHfgcqC

Digital Learning Materials

http://www.dataversity.net/category/data-topics/big-data

10 Lecture Schedule / Lesson Plan

Topics	**Theory**
What is Bigdata, Why Bigdata is Important	2
Meet Hadoop – Data	2
Data Storage and Analysis	1
Comparison with other systems	1
Grid Computing	1
A brief history of Hadoop	1
Apache Hadoop and the Hadoop Eco System	1
	9
UNIT –II: MapReduce	
Analyzing data with UNIX tools	1
Analyzing data with Hadoop	1
Java MapReduce classes (new API)	2
Data flow	2
Combiner functions	2
Running a distributed MapReduce Job	1
	9
HDFS concepts	1
Command line interface to HDFS	1
Hadoop File systems	1
Interfaces, Java Interface to Hadoop	1
Anatomy of a file read	2
Anatomy of a file write	2
Replica placement and Coherency Model	1
Parallel copying with distcp	1
Keeping an HDFS cluster balanced	1
	9
Setting up the development environment	1
Managing the configuration	2
Writing a unit test with MRUnit	2
Running a job in local job runner	2

Running on a cluster	1
Launching a job	1
The MapReduce WebUI	1
	8
Classic MapReduce	2
Job submission, Job Initialization	2
Task Assignment, Task execution	1
Process and status updates	1
Job Completion	1
Shuffle and sort on Map and reducer side	1
Configuration tuning	1
MapReduce types	1
Input formats	1
Output formats	1
	10
Hive	1
The Hive Shell, Hive services	1
Hive clients	1
The meta store	1
Comparison with traditional databases	1
Hive QL	2
Tables	1
Querying data	1
User defined functions	1
	10
Total No.of Periods:	**62**

1.1 What is Big Data

to Big Data

Definition:

- Big Data is often described as extremely large data sets that have grown beyond the ability to manage and analyze them with traditional data processing tools. The data set has grown so large that it is difficult to manage and even harder to garner value out of it.

- The primary difficulties are the acquisition, storage, searching, sharing, analytics, and visualization of data. Not only the size of the data set but also difficult to process the data

The data come from everywhere: Sensors used to gather climate information, posts to social media sites, digital pictures and videos posted online, transaction records of online purchases, and cell phone GPS signals etc. All of these data have intrinsic value that can be extracted using analytics, algorithms, and other technique

Characteristics of Big Data

Three characteristics define Big Data: ***volume, variety,*** **and** ***velocity***

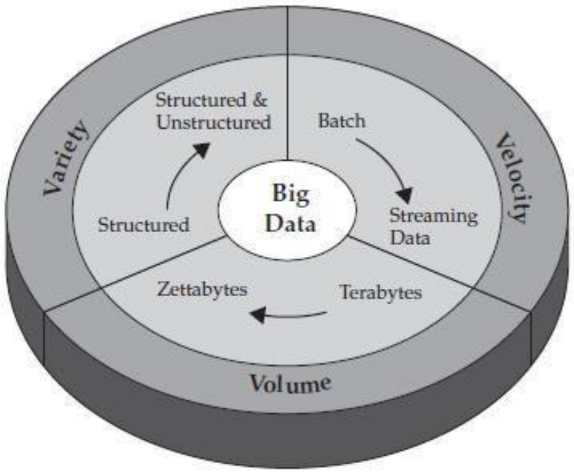


Fig1: Big Data Characteristics

Volume: The amount of data

The size of available data has been growing at an increasing rate. The volume of data is growing. Experts predict that the volume of data in the world will grow to 35 Zetta bytes in 2020.

- Twitter alone generates more than 7 terabytes (TB) of data every day. Facebook 10TB
- That same phenomenon affects every business – their data is growing at the same exponential rate too.
- This applies to companies and to individuals. A text file is a few kilo bytes, a sound file is a few meg bytes while a full-length movie is a few giga bytes. More sources of data are added on continuous basis.
- For companies, in the old days, all data was generated internally by employees. Currently, the data is generated by employees, partners and customers.
- For a group of companies, the data is also generated by machines. For example, Hundreds millions of smart phones send a variety of information to the network infrastructure.
- We store everything: Environmental data, financial data, medical data, surveillance data. Petabyte datasets are common these days and Exabyte is not far away.

Velocity: How fast it is generated

- Data is increasingly accelerating the velocity at which it is created and at which it is integrated We have moved from batch to a real-time business.
- Initially, companies analyzed data using a batch process. One takes a chunk of data, submits a job to the server and waits for delivery of the result. That scheme works when the incoming data rate is slower than the batch -processing rate and when the result is useful despite the delay.
- With the new sources of data such as social and mobile applications, the batch process breaks down. The data is now streaming into the server in real time, in a continuous fashion and the result is only useful if the delay is very short.

Variety: Represents all kinds of data

Data can be classified under several categories: structured data, semi structured data and unstructured data.

Structured data are normally found in traditional databases (SQL or others) where data are organized into tables based on defined business rules. Structured data usually prove to be the easiest type of data to work with, simply because the data are defined and indexed, making access and filtering easier.

Unstructured data, are not organized into tables and cannot be natively used by applications or interpreted by a database. A good example of unstructured data would be a collection of binary image files.

Semi structured data fall between unstructured and structured data. Semi structured data do not have a formal structure like a database with tables and relationships. However, unlike unstructured data, semi structured data have tags or other markers to separate the elements and provide a hierarchy of records and fields, which define the data.

- Big data extend beyond structured data to include unstructured data off all varieties: text, audio, video, click streams, log files and more.
- The growth in data sources has fueled the growth in data types. In fact, 80% of the world's data is unstructured and only 20% structured data Yet most traditional methods apply analytics only to structured information.

1.2 Why Big Data is Important

- Big Data solutions are ideal for analyzing not only raw structured data, but semi structured and unstructured data from a wide variety of sources.
- Big Data solutions are ideal when all, or most, of the data needs to be analyzed versus a sample of the data; or a sampling of data isn't nearly as effective as a larger set of data from which to derive analysis.
- Big Data solutions are ideal for iterative and exploratory analysis when business measures on data are not predetermined.

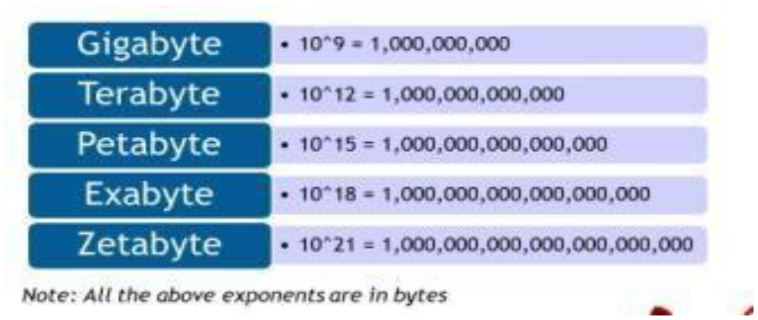
- Big Data is well suited for solving information challenges that don't natively fit within a traditional relational database approach for handling the problem at hand.
- Big Data has already proved its importance and value in several areas. Organizations such as the National Oceanic and Atmospheric Administration (NOAA), the National Aeronautics and Space Administration (NASA), several pharmaceutical companies, and numerous energy companies have amassed huge amounts of data and now leverage Big Data technologies on a daily basis to extract value from them.

  NOAA uses Big Data approaches to aid in climate, ecosystem, weather, and commercial research,
- NASA uses Big Data for aeronautical and another research.
- Pharmaceutical companies and energy companies have leveraged Big Data for more tangible results. such as drug testing and geophysical analysis.
- The New York Times has used Big Data tools for text analysis and Web mining
- Walt Disney Company uses them to correlate and understand customer behavior in all of its stores, theme parks.
- Companies such as Facebook, Amazon, and Google rely on Big Data analytics a part of their primary marketing schemes as well as a means of servicing their customers better.
- This accomplished by storing each customer's searches and purchases and other piece of information av aimable, and then applying algorithms to that information to compare one customer's information with all other customers information.
- Big Data plays another role in today's businesses: Large organizations increasingly face the need to maintain massive amounts of structured and unstructured data — from transaction information in data warehouses to employee tweets, from supplier records to regulatory filings—to comply with government regulations.

1.3 Meet Hadoop - data

Data:

Every day zeta bytes or peta bytes of data is generated by People and machines.



Gigabyte	• 10^9 = 1,000,000,000
Terabyte	• 10^12 = 1,000,000,000,000
Petabyte	• 10^15 = 1,000,000,000,000,000
Exabyte	• 10^18 = 1,000,000,000,000,000,000
Zetabyte	• 10^21 = 1,000,000,000,000,000,000,000

Note: All the above exponents are in bytes

Fig2: sizes of data

If the amount of data is more than hundreds of terabytes then such a data is called as **big data.**

Data generated by People:

1.Through individual interactions -

- Phone calls- emails- documents

2.Through social media

-twitter-Facebook-what sup etc.

3. Data generated by Machines:

-RFID readers-Sensor networks -Vehicle GPS traces-Machine logs

1.4 Data Storage and Analysis

Problem:

- Struggling with storage and analysis of the data.
- Even though the storage capacities of hard drives have increased massively over the years, access speeds (the rate at which data can be read from drives)
- Take long time to read all data on a single drive —and writing is even slower.
- The obvious way to reduce the time is to read from multiple disks at once. **Ex:** if we had 100 drives, each holding one hundredth of the data. Working in parallel, we could read the data in under two minutes.
- Even though read and write data in parallel to or from multiple disks , there are some more problems.

First Problem: Hardware failure

As soon as you start using many pieces of hardware, the chance that one will fail is fairly high. A common way of avoiding data loss is through

replication: redundant copies of the data are kept by the system so that in the event of failure, there is another copy available. This is how RAID (redundant array of inexpensive disks) works.

Second problem: most analysis tasks need to be able to combine the data in some way; i.e. data read from one disk may need to be combined with the data from any of the other 99 disks.

Solution for above problems:

Building distributed systems —for data storage, data analysis, and coordination.

Hadoop provides: a reliable shared storage and analysis system. The storage is provided by **HDFS** and analysis by **MapReduce.**

1.HDFS - Hadoop Distributed File System.

It avoids data loss is through **replication.** Minimum of three replicas for the data.

2.MapReduce - Programming model. It abstracts the problem from disk reads and writes, transforming it into a computation over sets of keys and values

1.5 Comparison with other systems

- The approach taken by MapReduce may seem like a brute **-force approach** on the entire dataset —or at least a good portion of it —is processed for each query.
- MapReduce is a **batch** query processor, and the ability to run an adhoc query against the whole dataset and get the results in a reasonable time is transformative.
- It changes the way you think about data, and unlocks data that was previously archived on tape or disk.
- Why can't we use databases with lots of disks to do large-scale batch analysis? Why is MapReduce needed? MapReduce can be seen as a complement to an RDBMS. The differences between the two systems are shown in Table

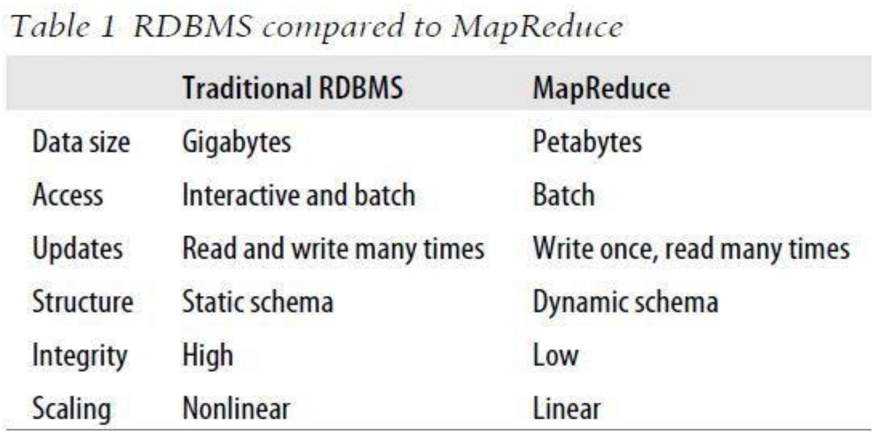


Table 1 RDBMS compared to MapReduce

	Traditional RDBMS	MapReduce
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Structure	Static schema	Dynamic schema
Integrity	High	Low
Scaling	Nonlinear	Linear

Fig3: Comparison between RDBMS & MapReduce

- MapReduce is a good fit for problems that need to analyze the whole dataset, in a batch fashion, particularly for adhoc analysis. 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. Relational database is good for datasets that are continually updated.
- Another difference is the amount of structure in the datasets that they operate on RDBMS 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. Map Reduce operate on Semi - structured and Unstructured data. In Se mi -structured data there may be a schema, it is often ignored, so it may be used only as a guide to the structure of the data

Ex: 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

Ex: 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.

Relational data is normalized to retain its integrity(assurance of accuracy) and remove redundancy.

Normalization poses problems for MapReduce, since it makes reading a record a nonlocal operation, and one of the central assumptions that MapReduce makes is t h at it is possible to perform (high -speed) streaming reads and writes.

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

- ➢ MapReduce is a linearly scalable programming model. The programmer writes two functions —a map function and a reduce function —each of which defines a mapping from one set of key -value pairs to another.
- ➢ These functions are unmind to the size of the data or the cluster that they are operating on, so they can be used unchanged for a small dataset and for a massive one.
- ➢ 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.

1.6 Grid Computing

Grid computing

- ➢ The H PC and Grid computing doing large scale data processing using APIs as Message Passing Interface (MPI)The approach of HPC is to distribute the work across a cluster of machines Which access shared files system Hosted by a Storage Area Network (SAN) Works well for compute intensive jobs
- ➢ It faces problem when nodes need to access larger data volumes i.e. hundreds of giga bytes. Reason is the network bandwidth is the bottleneck and computer nodes become idle. (At this point Hadoop starts shines)
- ➢ MapReduce tries to collocate the data with the compute node, so data access is fast since it is local. This feature, known as ***data locality****, is at the heart of MapReduce and* is the reason n for its good performance.
- ➢ Network bandwidth is more precious resource in the data center environment (easy to saturate network links by copying data around)
- ➢ Hadoop models its network topology by consuming bandwidth as less as possible. It does not prevent high –CPU analysis in Hadoop.

1) MPI gives great control to the programmer, but requires that explicitly handle the mechanics of the

-- data flow

-- exposed via low -level C routines

-- constructs, such as sockets

-- the higher -level algorithm for the analysis.

MapReduce operates only at the higher level: the programmer thinks in terms of functions of key and value pairs, and the data flow is implicit.

2) Coordinating the processes in a large -scale distributed computation is a challenge. The hardest aspect is gracefully handling partial failure—you don't know if a remote process has failed or not.

MapReduce spares the programmer from having to think about failure, since the implementation detects failed map or reduce tasks and reschedules replacements on m machines that are healthy.

MapReduce is able to do this since it is a ***shared -nothing*** architecture, meaning that tasks have no dependence on one other.

(This is a slight oversimplification, since the output from mappers is fed to the reducers, but this is under the control of the MapReduce system; it needs to take more care rerunning a failed reducer than rerunning a failed map, it has to make sure it can retrieve the

necessary map outputs, and if not, regenerate them by running the relevant maps again.)

So from the programmer's point of view, the order in which the tasks run doesn't matter.

By contrast, MPI programs have to explicitly manage their own check pointing and recovery, which gives more control to the programmer, but makes them more difficult to write.

- MapReduce is a restrictive programming model, and in a sense, it is: limited to key and value types that are related in specified ways, and mappers and reducers run with very limited coordination between one another.
- MapReduce was invented by engineers at Google It was inspired by older ideas from the functional programming, distributed computing, and database communities.

- ➢ Many applications in many industries use MR . It is pleasantly surprising to see the range of algorithms that can be expressed in MapReduce, from image analysis, to graph -based problems, to machine learning algorithms.
- ➢ It can't solve every problem, but it is a general data-processing tool.

Volunteer Computing

- ➢ Volunteer computing is one in which volunteers donate CPU time from their idle computers to analyze data.
- ➢ Volunteer computing projects work by breaking the problem they are trying to solve in to chunks called work unit.
- ➢ Work units are sent to computers around the world to be analyzed.

Ex: SETI (the Search for Extra -Terrestrial Intelligence) runs a project SETI@home in which volunteers donate CPU time from their idle computers to analyze radio telescope data for signs of intelligent life outside earth. In SETI@home work unit is about 0.35 MB of radio telescope data, and takes hours or days to analyze on a typical home computer.

When the analysis is completed, the results are sent back to the server, and the client gets another work unit.

- As a precaution to combat cheating, each work unit is sent to three different machines and needs at least two results to agree to be accepted.
- SETI@home may be superficially similar to MapReduce (breaking a problem into independent pieces to be worked on in parallel).
- The difference is SETI@home problem is very CPU -intensive, which makes it suitable for running on hundreds of thousands of computers across the world.
- The time to transfer the work unit is very small by the time to run the computation on it. Volunteers are donating CPU cycles, not bandwidth.
- MapReduce is designed to run jobs that last minutes or hours on trusted, dedicated hardware running in a single data center with very high aggregate bandwidth interconnects.
- By contrast, SETI@home perform computation on untrusted machines on the Internet with highly variable connection speeds and no data locality.

1.7 A brief history of Hadoop

Apache Hadoop is an open -source software framework for storage and large -scale processing of data -sets on clusters of commodity hardware.

it gives companies the capability to gather, store and analyze huge sets of data.

Some of the characteristics:

- Open source
- Distributed processing
- Distributed storage
- Scalable
- Reliable
- Fault -tolerant
- Economical
- Flexible

Originally built as a Infrastructure for the "Nutch" project. Based on Google's map reduce and Google File System. Created by **Doug Cutting** in 2005 at Yahoo Named after his son's toy **yellow elephant** Written in Java.

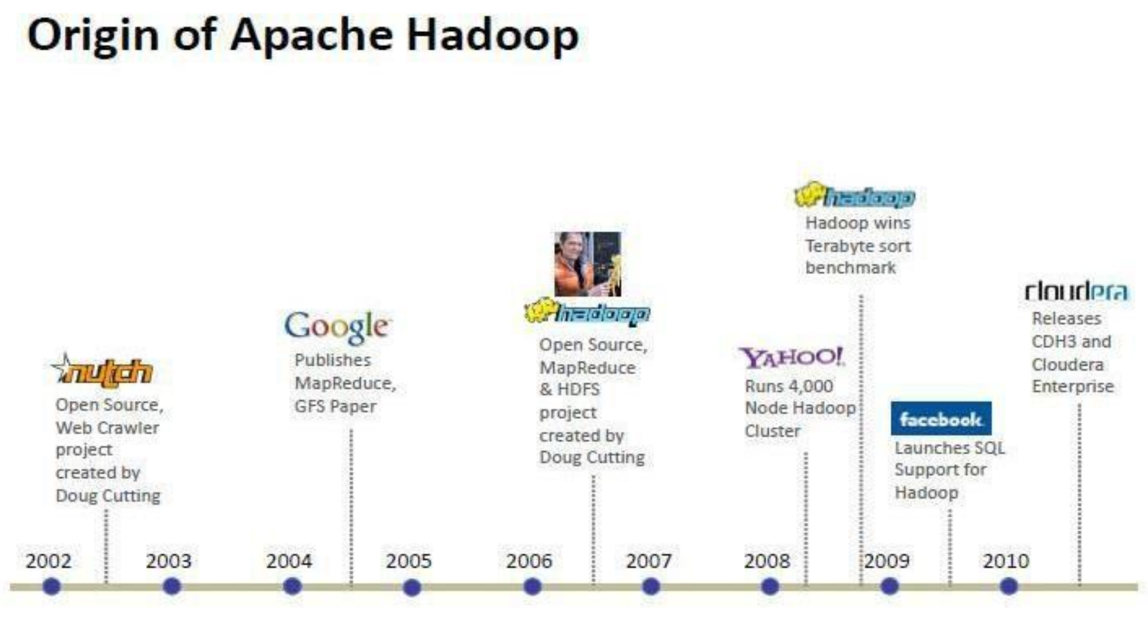

Origin of Apache Hadoop
nutch
Open Source, Web Crawler project created by Doug Cutting
Google
Publishes MapReduce, GFS Paper
hadoop
Open Source, MapReduce & HDFS project created by Doug Cutting
YAHOO!
Runs 4,000 Node Hadoop Cluster
hadoop
Hadoop wins Terabyte sort benchmark
facebook
Launches SQL Support for Hadoop
cloudera
Releases CDH3 and Cloudera Enterprise
2002
2003
2004
2005
2006
2007
2008
2009
2010

Fig 4: History of Hadoop

2002 – Nutch an open source web search engine started. This architecture wouldn't scale to the billions of pages on the Web.

2003 – Google published a paper that describes the architecture of Google's distributed filesystem, called GFS, which was being used in production at Google, would solve their storage needs for the very large files generated as a part of the web crawl and indexing process.

2004–writing an open source implementation, the Nutch Distributed Filesystem (NDFS). In the same year Google published the paper that introduced MapReduce to the world.

Early 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.

2006 - Nutch to form an independent sub project 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.

2008-Yahoo! Announced that its productionsearch index was being generated by a 10,000 -core Hadoop cluster. January 2008, Hadoop was made its own top -level project at Apache.

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.

November 2008 - Google reported that its MapReduce implementation sorted one terabyte in 68 seconds.

May 2009 - A team at Yahoo! used Hadoop to sort one terabyte in 62 seconds. Hadoop's is a general -purpose storage and analysis platform for big data.

Hadoop used by many companies Last.fm, Facebook and New York Times etc., Hadoop distributions from the large, established enterprise vendors EMC, IBM, Microsoft, and Oracle. Specialist Hadoop companies are Cloudera, Hortonworks and MapReduce

Apache Hadoop and the Hadoop Eco System

The term ecosystem is also used for a family of related projects that fall under the umbrella of infrastructure for distributed computing and largescale data processing.

All of the core projects 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.

Hadoop ecosystem 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

1.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.

2.MapReduce

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

3.HDFS

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

4.Pig

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

5.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.

6.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).

7.ZooKeeper

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

8.Sqoop

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

UNIT -I

Assignment -Cum -Tutorial Questions

SECTION -A

Objective Questions

1. The amount of data generated by machines will be greater than generated by people through [ ]

   i) Machine logs, RFID readers   iii) Sensor networks

   ii) Vehicle GPS traces   iv) Retails transactions

   A) i and ii   B) ii and iii   C) iii and iv   D) All

2. Which of the following is distributed data warehouse [ ]

   A) Hive   B) Pig   C) HBasse   D) ZooKepper

3. HDFS is [ ]

   A) Hardware Distributed File System   C) Adobe Distributed File System

   B) Hardware Distributed Filter System   D) Adobe Distributed Filter System

4. Map Reduce Provides__________Model [ ]

   A) Storage   B) Application   C) Programming   D) None.

5. Hadoop provides a reliable shared storage and analysis system [True/ False]

6. Map Reduce is a_______________ [ ]

   A) Batch query Processing   C) Multilevel query Processing

   B) Sequential query Processing   D) Interactive query Processing

7. The difference between Map Reduce and RDBMS is______________________.

8. Map Reduce Works well on [ ]

   A) Unstructured data   C) Structured Data

   B) Semi-Structured data   D) Both A & B

9. Big Data is well suited for solving information challenges that don't natively fit with in a traditional relational database approach for handling the problem at hand. [True / False]

10. What does commodity Hardware in Hadoop Would mean [ ]

A) Very cheap Hardware C) Industry Standard Hardware

B) Discard Hardware D) Low Specifications Industry Grade Hardware

11. The Type of data Hadoop can deal with is [ ]

A) Structured B) Semi-Structured C) Unstructured D) None

12. What is are true about HDFS [ ]

A) HDFS filesystem can be mounted on a local client's Filesystem using NFS.

B) HDFS filesystem can never be mounted on a local client's Filesystem.

C) You can edit an existing record in HDFS file which is already mounted using NFS.

D) You cannot append to a HDFS file which is mounted using NFS.

13. Data locality feature in Hadoop means__________________ [ ]

A) Collect Data Within the computed node

B) Collect data in data node

C) Collect data from main memory

D) None

14. BI(Business Intelligence) is a broad Category of Analytics__________________ Tools that help companies make sense of their structured and unstructured data for the purpose of making better business decisions. [ ]

A) Data Mining B) Dash Boards C) Reporting D) All

15. Which of the following are not Big Data Problems? [ ]

A) Parsing 5MB XML file every 5 Minutes

B) Processing IPL Tweet Sentiments

C) Processing online bank transactions

D) Both A & C

16. Which of the following are examples of Real Time Big Data Processing?

    A) Complex Event Processing (CEP) platforms.

    B) Stock market data analysis.

    C) Bank Fraud Transactions Detection   D) Both A & C.

17. What does "Velocity" in Big Data meant [ ]

    A)Speed of input data generation

    B) Speed of individual machine processors C)Speed only storing data

    D)Speed of storing and processing data

18. The term Big Data first originated from [ ]

    A) Stock Markets Domain   C) Genomics and Astronomy Domain

    B) Banking and finance Domain D) Social Media Domain

19. Which of the following Batch Processing instances is NOT an Example of Big Data Batch Processing [ ]

    A) Processing 10 GB sales data every 6 hours

    B) Processing flights Sensor Data.

    C) Web Crawling App.

    D) Trending topic analysis of tweets for last 15 minutes.

20. Which of the following are the core components of Hadoop? [ ]

    A) HDFS   B) Map Reduce   C) HBase   D) Both A & B

21.Match the Following.

I)Volume	[ ]	a) different data formats
II)Velocity	[ ]	b)rate at which data grows
III)Variety	[ ]	c)uncertainty of available data
IV)Veracity	[ ]	d)amount of data

22.Match the Following.

I) Semi structured Data	[ ]	a) images
II) Structured Data	[ ]	b) Bigdatacse@gmail.com
III) Unstructured Data	[ ]	c) Log Files

SECTION -B

SUBJ ECTIVE QUESTIONS

1. Discuss the importance of Big Data?
2. Examine the characteristics of Big Data?
3. List the companies who use the Hadoop tool to solve the Real world problems?
4. Distinguish Structured data, Semi-Structured and Unstructured data.
5. Explain the Brief history of Hadoop.
6. Illustrate volunteer computing Grid computing with map Reduce programming.
7. Discuss Hadoop Eco System, the projects supported by Hadoop.
8. Elaborate the importance of Hadoop and discuss the its Framework.
9. Justify how Bigdata analytics helps to increase the business revenue with example?
10. Compare and contrast Hadoop with Traditional RDBMS?
11. Describe the main components of a Hadoop.
12. Identify the problems involved in data storage and analysis of Bigdata?

BIG DATA
UNIT-2

Objective:

To familiarize with the Map Reduce of Big data

Syllabus:

Analyzing data with UNIX tools, Analyzing data with hadoop, Java Map Reduce classes (new API), Data flow, Combiner functions, Running a distributed Map Reduce Job

Learning Outcomes:

At the end of the unit, students will be able to:

1. Analyzing Map Reduce with Unix ,hadoop,java tools
2. Explain the data flow.
3. Develop the Map Reduce using the java.
4. Develop the Map Reduce in distributed Environment.

Learning Material

2.1 Analyzing the data with UNIX tools

- Without using Hadoop, as this information will provide a performance baseline, as well as a useful means to check our results.
- The classic tool for processing line-oriented data is *awk*, is a small script to calculate the maximum temperature for each year.

```
#!/usr/bin/env bash
for year in all/*
do
echo -ne `basename $year .gz`"\t"
gunzip -c $year | \
awk '{ temp = substr($0, 88, 5) + 0;
q = substr($0, 93, 1);
if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp }
```

```
END { print max }'
Done
```

- The script loops through the compressed year files, first printing the year, and then processing each file using *awk*.
- The *awk* script extracts two fields from the data: the air temperature and the quality code.
- The air temperature value is turned into an integer by adding 0.
- Next, a test is applied to see if the temperature is valid (the value 9999 signifies a missing value in the NCDC dataset) and if the quality code indicates that the reading is not suspect or erroneous.
- If the reading is OK, the value is compared with the maximum value seen so far, which is updated if a new maximum is found.
- The END block is executed after all the lines in the file have been processed, and it prints the maximum value.

Run the program

```
% ./max_temperature.sh
1901 317
1902 244
1903 289
1904 256
1905 283
...
```

- The temperature values in the source file are scaled by a factor of 10, so this works out as a maximum temperature of 31.7°C for 1901
- The complete run for the century took 42 minutes in one run on a single EC2 High-CPU Extra Large Instance.
- To speed up the processing, we need to run parts of the program in parallel
- There are a few problems with this,

- **First, dividing the work into equal-size pieces isn't always easy**, the file size for different years varies widely, so some processes will finish much earlier than others. The whole run is dominated by the longest file.
  A better approach, although one that requires more work, is to split the input into fixed-size chunks and assign each chunk to a process.
- **Second, combining the results from independent processes** may need further processing, the result for each year is independent of other years and may be combined by concatenating all the results, and sorting by year.
  If using the fixed-size chunk approach, the combination is more delicate. For this example, data for a particular year will typically be split into several chunks, each processed independently.
- **Third, you are still limited by the processing capacity of a single machine**. If the best time you can achieve is 20 minutes with the number of processors you have, then that's it. You can't make it go faster.
  Also, some datasets grow beyond the capacity of a single machine. When we start using multiple machines, a whole host of other factors come into play, mainly falling in the category of coordination and reliability.

2.2 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:

1. the map phase
2. The reduce phase.

- ➢ Each phase has key-value pairs as input and output,
- ➢ The input to our map phase is the raw NCDC data, a text input format that gives us each line in the dataset as a text value
- ➢ The key is the offset of the beginning of the line from the beginning of the file

- **Map function**: pull out the year and the air temperature, 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.
- **Reduce function** finding the maximum temperature for each year.

Input data

Sample lines of input data

```
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, 006701199099999**1950**051507004...9999999N9+**0000**1+99999999999...)
(106, 004301199099999**1950**051512004...9999999N9+**0022**1+99999999999...)
(212, 004301199099999**1950**051518004...9999999N9-**0011**1+99999999999...)
(318, 004301265099999**1949**032412004...0500001N9+**0111**1+99999999999...)
(424, 004301265099999**1949**032418004...0500001N9+**0078**1+99999999999...)

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

(1950, 0)
(1950, 22)
(1950, −11)
(1949, 111)
(1949, 78)

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

(1949, [111, 78])

(1950, [0, 22, −11])

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

(1949, 111)

(1950, 22)

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

The whole data flow is illustrated in Figure

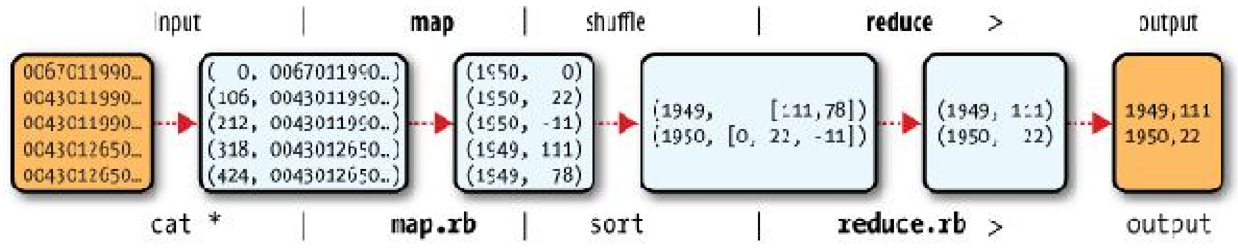


Figure: MapReduce logical data flow

- ➢ Java MapReduce 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.

Mapper for maximum temperature example

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class MaxTemperatureMapper
extends Mapper<LongWritable, Text, Text, IntWritable> {
private static final int MISSING = 9999;
@Override
public void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
String line = value.toString();
String year = line.substring(15, 19);
int airTemperature;
if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
airTemperature = Integer.parseInt(line.substring(88, 92));
} else {
airTemperature = Integer.parseInt(line.substring(87, 92));
}
String quality = line.substring(92, 93);
if (airTemperature != MISSING && quality.matches("[01459]")) {
context.write(new Text(year), new IntWritable(airTemperature));
}
}
}
```

Reducer for maximum temperature example

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
public class MaxTemperatureReducer
extends Reducer<Text, IntWritable, Text, IntWritable> {
@Override
public void reduce(Text key, Iterable<IntWritable> values,
Context context)
throws IOException, InterruptedException {
int maxValue = Integer.MIN_VALUE;
for (IntWritable value : values) {
maxValue = Math.max(maxValue, value.get());
}
context.write(key, new IntWritable(maxValue));
}
}
```

- The third piece of code runs the MapReduce job

Application to find the maximum temperature in the weather dataset

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class MaxTemperature {
public static void main(String[] args) throws Exception {
if (args.length != 2) {
System.err.println("Usage: MaxTemperature <input path> <output path>");
System.exit(-1);
}
Job job = new Job();
job.setJarByClass(MaxTemperature.class);
job.setJobName("Max temperature");
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.setMapperClass(MaxTemperatureMapper.class);
job.setReducerClass(MaxTemperatureReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

A test run

After writing a MapReduce job

```
% export HADOOP_CLASSPATH=hadoop-examples.jar
% hadoop MaxTemperature input/ncdc/sample.txt output
11/09/15 21:35:14 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobT
racker, sessionId=
11/09/15 21:35:14 WARN util.NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
11/09/15 21:35:14 WARN mapreduce.JobSubmitter: Use GenericOptionsParser for parsing t
he arguments. Applications should implement Tool for the same.
11/09/15 21:35:14 INFO input.FileInputFormat: Total input paths to process : 1
11/09/15 21:35:14 WARN snappy.LoadSnappy: Snappy native library not loaded
11/09/15 21:35:14 INFO mapreduce.JobSubmitter: number of splits:1
11/09/15 21:35:15 INFO mapreduce.Job: Running job: job_local_0001
11/09/15 21:35:15 INFO mapred.LocalJobRunner: Waiting for map tasks
11/09/15 21:35:15 INFO mapred.LocalJobRunner: Starting task: attempt_local_0001_m_000
000_0
11/09/15 21:35:15 INFO mapred.Task: Using ResourceCalculatorPlugin : null
11/09/15 21:35:15 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584)
11/09/15 21:35:15 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
11/09/15 21:35:15 INFO mapred.MapTask: soft limit at 83886080
11/09/15 21:35:15 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
11/09/15 21:35:15 INFO mapred.MapTask: kvstart = 26214396; length = 6553600
11/09/15 21:35:15 INFO mapred.LocalJobRunner:
11/09/15 21:35:15 INFO mapred.MapTask: Starting flush of map output
11/09/15 21:35:15 INFO mapred.MapTask: Spilling map output
11/09/15 21:35:15 INFO mapred.MapTask: bufstart = 0; bufend = 45; bufvoid = 104857600
11/09/15 21:35:15 INFO mapred.MapTask: kvstart = 26214396(104857584); kvend =
2621438
0(104857520); length = 17/6553600
11/09/15 21:35:15 INFO mapred.MapTask: Finished spill 0
11/09/15 21:35:15 INFO mapred.Task: Task:attempt_local_0001_m_000000_0 is done. And i
s in the process of commiting
11/09/15 21:35:15 INFO mapred.LocalJobRunner: map
11/09/15 21:35:15 INFO mapred.Task: Task 'attempt_local_0001_m_000000_0' done.
```

```
11/09/15 21:35:15 INFO mapred.LocalJobRunner: Finishing task: attempt_local_0001_m_00
0000_0
11/09/15 21:35:15 INFO mapred.LocalJobRunner: Map task executor complete.
11/09/15 21:35:15 INFO mapred.Task: Using ResourceCalculatorPlugin : null
11/09/15 21:35:15 INFO mapred.Merger: Merging 1 sorted segments
11/09/15 21:35:15 INFO mapred.Merger: Down to the last merge-pass, with 1 segments le
ft of total size: 50 bytes
11/09/15 21:35:15 INFO mapred.LocalJobRunner:
11/09/15 21:35:15 WARN conf.Configuration: mapred.skip.on is deprecated. Instead, use
mapreduce.job.skiprecords
11/09/15 21:35:15 INFO mapred.Task: Task:attempt_local_0001_r_000000_0 is done. And i
s in the process of commiting
11/09/15 21:35:15 INFO mapred.LocalJobRunner:
11/09/15 21:35:15 INFO mapred.Task: Task attempt_local_0001_r_000000_0 is allowed to
commit now
11/09/15 21:35:15 INFO output.FileOutputCommitter: Saved output of task 'attempt_loca
l_0001_r_000000_0' to file:/Users/tom/workspace/hadoop-book/output
11/09/15 21:35:15 INFO mapred.LocalJobRunner: reduce > reduce
11/09/15 21:35:15 INFO mapred.Task: Task 'attempt_local_0001_r_000000_0' done.
11/09/15 21:35:16 INFO mapreduce.Job: map 100% reduce 100%
11/09/15 21:35:16 INFO mapreduce.Job: Job job_local_0001 completed successfully
11/09/15 21:35:16 INFO mapreduce.Job: Counters: 24
File System Counters
FILE: Number of bytes read=255967
FILE: Number of bytes written=397273
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
Map-Reduce Framework
Map input records=5
Map output records=5
Map output bytes=45
Map output materialized bytes=61
```

```
Input split bytes=124
Combine input records=0
Combine output records=0
Reduce input groups=2
Reduce shuffle bytes=0
Reduce input records=5
Reduce output records=2
Spilled Records=10
Shuffled Maps =0
Failed Shuffles=0
Merged Map outputs=0
GC time elapsed (ms)=10
Total committed heap usage (bytes)=379723776
File Input Format Counters
Bytes Read=529
File Output Format Counters
Bytes Written=29
```

2.3 Java MapReduce classes (new API)

The Java MapReduce API used first released in Hadoop 0.20.0. This new API, sometimes referred to as "Context Objects," was designed to make the API easier to evolve in the future.

It is type-incompatible with the old, the new API is not complete in the 1.x (formerly 0.20) release series, so the old API is recommended for these releases, despite having been marked as deprecated in the early 0.20 releases.

The 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.

For example, the Mapper and Reducer interfaces in the old API are abstract classes in the new API.

- The new API is in the org.apache.hadoop.mapreduce package (and subpackages). The old API can still be found in org.apache.hadoop.mapred.
- The new API makes extensive use of context objects that allow the user code to communicate with the MapReduce system. The new Context, for example, essentially unifies the role of the JobConf, the OutputCollector, and the Reporter from the old API.
- In both APIs, key-value record pairs are pushed to the mapper and reducer, but in addition, the new API allows both mappers and reducers to control the execution flow by overriding the run() method. For example, records can be processed in batches, or the execution can be terminated before all the records have been processed. In the old API this is possible for mappers by writing a MapRunnable, but no equivalent exists for reducers.
- Configuration has been unified. The old API has a special JobConf object for job configuration, which is an extension of Hadoop's vanilla Configuration object (used for configuring daemons; see "The Configuration API" on page 146). 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 *partm- 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 responsive to interupts so that the framework can gracefully cancel long-running operations if it needs to3.

- In the new API the reduce() method passes values as a java.lang.Iterable, rather than a java.lang.Iterator (as the old API does). This change makes it easier to iterate over the values using Java's for-each loop construct:for (VALUEIN value : values) { ... }

2.4 Data Flow

- A MapReduce *job* is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information.
- Hadoop runs the job by dividing it into *tasks*, of which there are two types: *map tasks* and *reduce tasks*.
- There are two types of nodes that control the job execution process: a *jobtracker* and a number of *tasktrackers*.
- The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers.
- Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker.
- Hadoop divides the input to a MapReduce job into fixed-size pieces called *inputsplits*, or just *splits*.
- Hadoop creates one map task for each split, which runs the userdefined map function for each *record* in the split.
- Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if we are processing the splits in parallel, the processing is better load-balanced if the splits are small.
- If splits are too small, then the overhead of managing the splits and of map task creation begins to dominate the total job execution time.

- For most jobs, a good split size tends to be the size of an HDFS block, 64 MB by default, although this can be changed for the cluster (for all newly created files), or specified when each file is created.

2.5 Combiner Functions

- Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks.
- Hadoop allows the user to specify a *combiner function* to be run on the map output—the combiner function's output forms the input to the reduce function.
- Since the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record.
- Calling the combiner function zero, one, or many times should produce the same output from the reducer.
- The maximum temperature example, readings for the year 1950 were processed by two maps (because they were in different splits). Imagine the first map produced the output:

(1950, 0)

(1950, 20)

(1950, 10)

And the second produced:

(1950, 25)

(1950, 15)

- The reduce function would be called with a list of all the values:

(1950, [0, 20, 10, 25, 15])

with output:

(1950, 25)

- Use a combiner function that, just like the reduce function, finds the maximum temperature for each map output. The reduce would then be called with:

(1950, [20, 25])

and the reduce would produce the same output as before. More succinctly, we may express the function calls on the temperature values in this case as follows:

max(0, 20, 10, 25, 15) = *max*(*max*(0, 20, 10), *max*(25, 15)) = *max*(20, 25) = 25

- Not all functions possess this property.4 For example, if we were calculating mean temperatures, then we couldn't use the mean as our combiner function, since:

mean(0, 20, 10, 25, 15) = 14

but: *mean*(*mean*(0, 20, 10), *mean*(25, 15)) = *mean*(10, 20) = 15

- The combiner function doesn't replace the reduce function.
- The reduce function is still needed to process records with the same key from different maps.
- it can help cut down the amount of data shuffled between the maps and the reduces,and for this reason alone it is always worth considering whether you can use a combiner function in your MapReduce job.

Specifying a combiner function

In the Java MapReduce program, the combiner function is defined using the Reducer class, and for this application, it is the same implementation as the reducer function in MaxTemperatureReducer.

The only change we need to make is to set the combiner class on the Job

Application to find the maximum temperature, using a combiner function for efficiency

```
public class MaxTemperatureWithCombiner {
public static void main(String[] args) throws Exception {
if (args.length != 2) {
System.err.println("Usage: MaxTemperatureWithCombiner <input path> " +
"<output path>");
System.exit(-1);
}
Job job = new Job();
job.setJarByClass(MaxTemperatureWithCombiner.class);
job.setJobName("Max temperature");
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.setMapperClass(MaxTemperatureMapper.class);
job.setCombinerClass(MaxTemperatureReducer.class);
job.setReducerClass(MaxTemperatureReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
System.exit(job.waitForCompletion(true) ? 0 : 1);
}       }
```

2.6 Running a Distributed MapReduce Job

The same program will run, without alteration, on a full dataset.

This is the point of MapReduce: it scales to the size of your data and the size of your hardware. Here's one data point: on a 10-node EC2 cluster running High-CPU Extra Large Instances, the program took six minutes to run.

Practical aspects of developing a MapReduce application in Hadoop.

Writing a program in MapReduce has a certain flow to it. You start by writing your map and reduce functions, ideally with unit tests to make sure they do what you expect.

Then you write a driver program to run a job, which can run from your IDE using a small subset of the data to check that it is working.

If it fails, then you can use your IDE's debugger to find the source of the problem. With this information, you can expand your unit tests to cover this case and improve your mapper or reducer as appropriate to handle such input correctly.

UNIT-II
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. Mapper implementations are passed the JobConf for the job via the ______method. [ ]
   A) JobConfigure.Configure C)JobConfigurable.configureable
   B) JobConfigurable.Configure D) None
2. Input to the _____ is the sorted output of the mappers. [ ]
   A) Reducer B) Mapper C)Shuffle D) All
3. The output of the ___ is not sorted in the Map Reduce frame work for Hadoop
   A) Mapper B) CasCader C) Scalding D) None [ ]
4. Which of the following phase occur simultaneously? [ ]
   A) Shuffle and Sort C) Shuffle and Map
   B) Reduce and Sort D) All
5. ____ is a programming model designed for processing large volumes of the data in parallel by dividing the work into a set of independent tasks.
   A) Hive B) Map Reduce C) Pig D) Lucene [ ]
6. The daemons associated with the Map Reduce phase are___________ and task_trackers. [ ]
   A) Job-Tracker B) Map-Tracker C) Reduce-Tracker D) All
7. The Job Tracker pushes work out to available____________ nodes in the cluster , striving to keep the work as close to the data as possible.[ ]
   A) Data Nodes B) Task Tracker C) Action Nodes D) All
8. Input Format class calls the_____ function and computes splits for each file and then sends them to the job tracker. [ ]
   A) Puts B) Gets C) GetSplits D) All
9. On a Task Tracker the map Task pass the split to the create RecordReader() method on InputFormat to obtain a_____ for that split. [ ]

A) InputReader B) RecordReader C) OutputReader D) None

10. The default InputFormat is ____ which treats each value of input a new value and the associated key is byte offset. [ ]

A) TextFormat B) TextInputFormat C) InputFormat D) All

11. ____Controls the partitioning of thekeys of the intermediate map_outputs.

A) Collector B) Partitioner C) InputFormat D) None [ ]

12. Output of the mapper is first written on the local disk for sorting and ______Process.

13. Point out the correct statement [ ]

A) Data locality means movement of algorithm to the data instead of data algorithm.

B) When the processing is done on the data algorithm is moved across the Action Nodes rather than data to the algorithm.

C) Moving Computation is expensive than Moving Data.

D) None.

14. Point out the wrong statement [ ]

A) The map function in Hadoop MapReduce have the following general form map(K1,V1)->list(K2,V2)

B) The reduce function in Hadoop MapReduce have the following general form:reduce(K2,list(V2))->list(K3,V3)

C) MapReduce has a complex model of data processing: inputs and outputs for the map and reduce functions are key-value pairs.

D) None.

15. The right number of reduces seems to be [ ]

A) 0.90 B) 0.80 C) 0.36 D)0.95

16. Mapper and Reducer implementations can use the _______ to report progress or just indicate that they are alive. [ ]

A) Partitioner B) OutputCollector C) Reporter D) All

17. ______ is a generalization of the facility provided by the MapReduce frame work to collect data output by the Mapper or the Reducer. [ ]

A) Partitioner B) OutputCollector C) Reporter D) All

18. _____ is the primary interface for a user to describe a MapReduce job to the Hadoop frame work for execution . [ ]

    A) Map Parmeters B) JobConf C) MemoryConf D) None

19. The Hadoop MapReduce Frame work spawns one map task for each ____ generated by the InputFormat for the job. [ ]

    A) OutputSplit B) InputSplit C) inputSplitStream D) All

20. The right level of parallelism for maps seems to be around ____________ maps per-node. [ ]

    A) 1-10 B) 10-10 C) 100-15 D) 150-200

SECTION-B

SUBJECTIVE QUESTIONS

1. Write about analyzing data with unix tools?
2. Explain about analyzing data with Hadoop?
3. Outline about the Java Map Reduce?
4. Explain MapReduce Logical Data Flow?
5. Discuss about Job Tracker?
6. Write about Task Tracker?
7. What is a combiner function? Explain
8. Discuss about running a Distributed Map Reduce Job?
9. List out the problems in analyzing data with unix tools?
10. Illustrate the Map Reduce Works.
11. Explain how the data can be analyzed by using Hadoop
12. Write a program for Map Reduce using JAVA.
13. Draw the Map Reduce Data Flow with a Single Reduce Task.
14. Draw the Map Reduce Data Flow with a Multiple Reduce Tasks.
15. Draw the Map Reduce Data Flow with a No Reduce Tasks.
16. Design a Application to find the maximum temperature using a combiner function for efficiency.

17. Write the steps to Map function for maximum temperature in Ruby.
18. Write the steps to Reduce function for maximum temperature in Ruby.
19. Write the steps to Map function for maximum temperature in Python.
20. Write the steps to Reduce function for maximum temperature in Python.

UNIT-III

Hadoop Distributed File System

Objective: To familiarize with the fundamental concepts of Hadoop Distributed File system.

Syllabus:

Hadoop Distributed File System

HDFS concepts, Command line interface to HDFS, Hadoop File systems, Interfaces, Java Interface to

Hadoop, Anatomy of a file read, Anatomy of a file write, Replica placement and Coherency Model,

Parallel copying with distcp, Keeping an HDFS cluster balanced.

Learning Outcomes:

At the end of the unit, students will be able to:

1. Understand the fundamental concepts of HDFS
2. Describe Hadoop interfaces, read, write and replica placement of Hadoop.

Learning Material

Introduction

- **Distributed Filesystem:** Filesystems that manage the storage across a network of machines are called *distributed filesystems.*
- **Challenge :** making the filesystem tolerate node failure without suffering data loss.

- Hadoop comes with a distributed filesystem called **HDFS**, which stands for *Hadoop Distributed Filesystem*

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.**

Very large files

“Very large” in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size.

There are Hadoop clusters running today that store petabytes of data.

Streaming data access

- HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern.

- A dataset is typically generated or copied from source, then various analyses are performed on that dataset over time.

- Each analysis will involve a large proportion, the time to read the whole dataset is more important than the latency in reading the first record.

Commodity hardware

- Hadoop doesn’t require eJxpensive, highly reliable hardware to run on. It’s designed to run on clusters of commodity hardware (commonly available hardware available from multiple vendors).

- The chance of node failure across the cluster is high.

- HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

HDFS does not work well for some areas

Low-latency data access

Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS because HDFS is optimized for delivering a high throughput of data

Lots of small files

The namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode each file, directory, and block takes about 150 bytes.

Ex: if you had one million files, each taking one block, you would need at least 300 MB of memory.

file modifications

Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers, or for modifications at arbitrary offsets in the file.

3.1 HDFS Concepts

-Blocks

-Namenodes and DataNodes -HDFS fedaration

-HDFS High-Availability.

Blocks

- A disk has a block size, which is the minimum amount of data that it can read or write.
- Disk blocks are 512 bytes. FileSystem block size is 64MB. HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks.
- Files in HDFS are broken into block-sized chunks, which are stored as independent units.
- Map tasks in Map Reduce normally operate on one block at a time.

Distributed file system having block abstraction

Benefits are :

1) A file can be larger than any single disk in the network.

   It doesn't requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster.

   It would be possible, to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.

2) Making the unit of abstraction a block rather than a file simplifies the storage subsystem, storage management(easy to calculate how many can be stored in a single disk), and eliminate metadata (permissions need not store in block)

3) Blocks with replication providing fault tolerance and availability. Each block is replicated to a small number of physically separate machines(three).

4) If the block is not available, copy of the block is read from another location.

5) If the block is corrupted due to machine failure can be replicated to other live machines.

6) Command to list the bolcks is **$hadoop fsck / -files –blocks**.

Namenodes and DataNodes

- An HDFS cluster has two types of nodes operating in a master-worker pattern:
  1) **N*amenode* (the master)**
  2) **A number of *datanodes (workers).***

Namenode tasks

- Manages the filesystem namespace.

- Maintains the filesystem tree and the metadata for all the files and directories in the tree.

- This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log.

- The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

- A client accesses the filesystem on behalf of the user by communicating with the namenode and datanodes.

- The user code does not need to know about the namenode and datanode to function.

Datanode tasks

- Datanodes are the workhorses of the filesystem.

- DNs store and retrieve blocks when they are told to (by clients or the namenode)

- Periodically report to the namenode with lists of blocks that they are storing.

Handling of Namenode failure

- Without the namenode, the filesystem cannot be used. If the machine running the namenode is failed, all the files on the filesystem would be lost.
- It is important to make the namenode resilient to failure, Hadoop provides two mechanisms for this.

1. Back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured that the namenode writes its persistent state to multiple filesystems i.e write to local disk as well as a remote NFS. These writes are synchronous and atomic.

2. Run a secondary namenode, it doesnot act as a namenode. 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 runs on a separate physical machine, because it requires plenty of CPU and as much memory as the namenode to perform the merge.

- It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing.

- The state of the secondary namenode lags that of the primary, In the event of total failure of the primary, data loss is almost certain.

- It copy the namenode's metadata files that are on NFS to the secondary and run it as the new primary.

HDFS Federation

- The namenode keeps a reference to every file and block in the filesystem in memory, 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.

Ex : one namenode might manage all the files rooted under */user, and a second* namenode might handle files under */share*

- Each namenode manages a *namespace volume, which is made up of* the metadata for the namespace, and a *block pool containing all the blocks for the files* in the namespace.

https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/Federation.html

- Namespace volumes are independent of each other, namenodes do not communicate with one another, and the failure of one namenode does not affect the availability of the namespaces managed by other namenodes.

HDFS High-Availability

- The combination of replicating namenode metadata on multiple filesystems, and using the secondary nJamenode to create checkpoints protects against data loss, but does not provide high-availability of the filesystem.

- The namenode is still a *single point of failure* (SPOF), since if it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping.

- To recover from a failed administrator starts a new primary namenode the filesystem namenode with one of and configures metadata replicas, datanodes 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.

- 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 stand by 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

    - Datanodes must send block reports to both namenodes since the block mappings are stored in a namenode's memory, and not on disk.

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

- If the active namenode fails, then the standby can take over very quickly (in a few tens of seconds) .

- it has the latest state available in memory: both the latest edit log entries, and an up-to-date block mapping.

3.2 The Command-Line Interface

- There are many other interfaces to HDFS, but the command line is one of the simplest and the most familiar.

- There are two properties that we set in the pseudo-distributed configuration

1. The first is **fs.default.name**, set to *hdfs://localhost/, which is* used to **set a default filesystem** for HadoopThe HDFS daemons will use this property to determine the host and port for the HDFS namenode.

   We'll be running it on **localhost**, on the default HDFS port, **8020**.

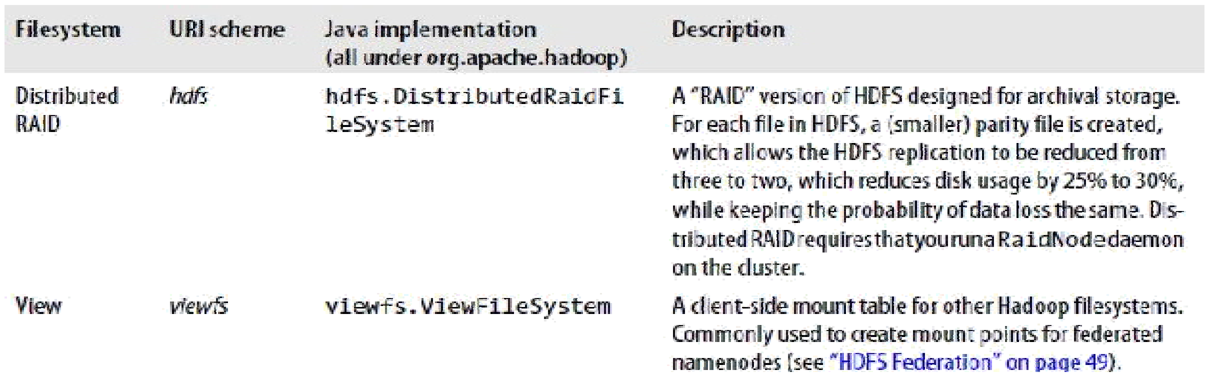
2. Set the second property, **dfs.replication**, to 1 so that HDFS doesn't blocks by the default factor of three.

replicat

Basic Filesystem Operations

Filesystem operations such as reading files, creating directories, moving files, deleting data, and listing directories.

Type hadoop fs -help to get detailed help on every command.



Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
Distributed RAID	*hdfs*	`hdfs.DistributedRaidFileSystem`	A "RAID" version of HDFS designed for archival storage. For each file in HDFS, a (smaller) parity file is created, which allows the HDFS replication to be reduced from three to two, which reduces disk usage by 25% to 30%, while keeping the probability of data loss the same. Distributed RAID requires that you run a `RaidNode` daemon on the cluster.
View	*viewfs*	`viewfs.ViewFileSystem`	A client-side mount table for other Hadoop filesystems. Commonly used to create mount points for federated namenodes (see "HDFS Federation" on page 49).

1. copying a file from the local filesystem to HDFS:

   Ex: %

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

2 create a directory then see how it is displayed in the listing:

   %**hadoop fs -mkdir books**

   %**hadoop fs -ls .**

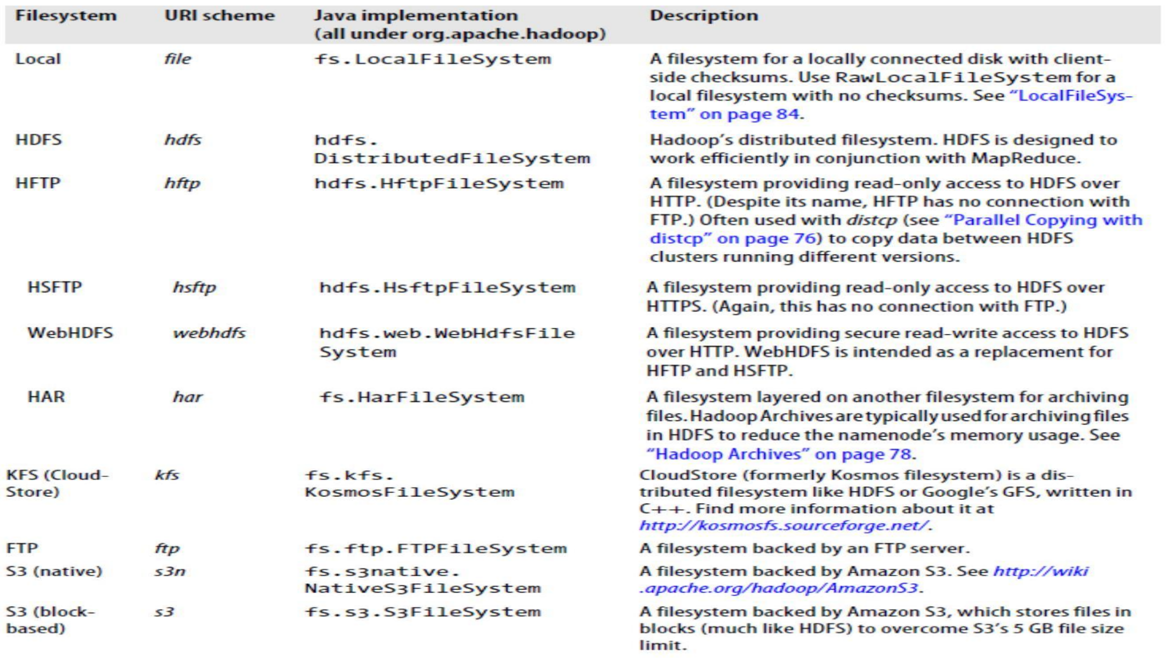
   Found 2 items

   drwxr-xr-x - tom supergroup 0 2009-04-02 22:41 /user/tom/books

   -rw-r--r-- 1 tom supergroup 118 2009-04-02 22:29 /user/tom/quangle.txt

3.3 Hadoop Filesystems

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



Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
Local	*file*	`fs.LocalFileSystem`	A filesystem for a locally connected disk with client-side checksums. Use `RawLocalFileSystem` for a local filesystem with no checksums. See "LocalFileSystem" on page 84.
HDFS	*hdfs*	`hdfs.DistributedFileSystem`	Hadoop's distributed filesystem. HDFS is designed to work efficiently in conjunction with MapReduce.
HFTP	*hftp*	`hdfs.HftpFileSystem`	A filesystem providing read-only access to HDFS over HTTP. (Despite its name, HFTP has no connection with FTP.) Often used with *distcp* (see "Parallel Copying with distcp" on page 76) to copy data between HDFS clusters running different versions.
HSFTP	*hsftp*	`hdfs.HsftpFileSystem`	A filesystem providing read-only access to HDFS over HTTPS. (Again, this has no connection with FTP.)
WebHDFS	*webhdfs*	`hdfs.web.WebHdfsFileSystem`	A filesystem providing secure read-write access to HDFS over HTTP. WebHDFS is intended as a replacement for HFTP and HSFTP.
HAR	*har*	`fs.HarFileSystem`	A filesystem layered on another filesystem for archiving files. Hadoop Archives are typically used for archiving files in HDFS to reduce the namenode's memory usage. See "Hadoop Archives" on page 78.
KFS (Cloud-Store)	*kfs*	`fs.kfs.KosmosFileSystem`	CloudStore (formerly Kosmos filesystem) is a distributed filesystem like HDFS or Google's GFS, written in C++. Find more information about it at *http://kosmosfs.sourceforge.net/*.
FTP	*ftp*	`fs.ftp.FTPFileSystem`	A filesystem backed by an FTP server.
S3 (native)	*s3n*	`fs.s3native.NativeS3FileSystem`	A filesystem backed by Amazon S3. See *http://wiki.apache.org/hadoop/AmazonS3*.
S3 (block-based)	*s3*	`fs.s3.S3FileSystem`	A filesystem backed by Amazon S3, which stores files in blocks (much like HDFS) to overcome S3's 5 GB file size limit.

Hadoop provides many interfaces to its filesystems, and it generally uses the URI(uniform resource identifier) scheme to pick the correct filesystem instance to communicate with.

3.4 Interfaces

- Hadoop is written in Java, and all Hadoop filesystem interactions are mediated through the Java API.

- The filesystem shell, for example, is a Java application that uses the Java FileSystem class to provide filesystem operations.

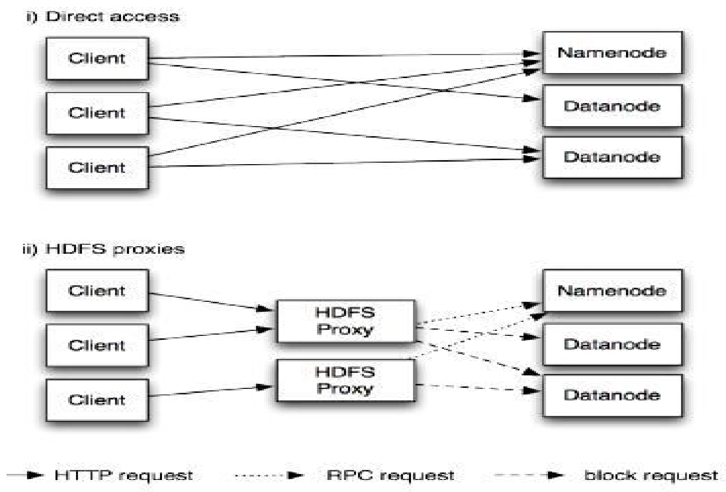
The other filesystem interfaces are FTP clients for FTP, S3 tools for S3, etc.), but many of them will work with any Hadoop filesystem.

HTTP

There are two ways of accessing HDFS over HTTP:

1. directly, where the HDFS daemons serve HTTP requests to clients;

2. via a proxy (or proxies), which accesses HDFS on the client's behalf using the usual DistributedFileSystem API



C

- Hadoop provides a C library called ***libhdfs*** *that mirrors the Java FileSystem interface*
- It works using the *Java Native Interface (JNI) to call* a Java filesystem client.
- The C API is very similar to the Java one, but it typically lags the Java one, so newer features may not be supported.
- Documentation for the C API in the **libhdfs/docs/api** directory of the Hadoop distribution.
- Hadoop comes with prebuilt libhdfs binaries for 32-bit Linux, but for other platforms, build them using the instructions at **http://wiki.apache.org/hadoop/LibHDFS**.

FUSE

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

- Documentation for compiling and running Fuse-DFS is located in the **src/contrib/fuse-dfs** directory of the Hadoop distribution

3.5 The Java Interface TO Hadoop

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

- Write the code against the FileSystem abstract class, to retain portability across filesystems.

Reading Data from a Hadoop URL

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.

https://kannandreams.wordpress.com/2013/11/14/what-is-uri-and-difference-between-uriurl-and-urn/

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

//    process in }
finally          {
IOUtils.closeStre
am(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.

Ex: 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.closeStre
am(in);
}}}
```

- IOUtils class that comes with Hadoop for closing the stream in the finally clause,

- 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.

Syntax :

copyBytes(InputStream in, OutputStream out, int buffSize, boolean close)

Copies from one stream to another.

- close the input stream ourselves, and System.out doesn't need to be closed.

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

- Sometimes it is impossible to set a URLStreamHandlerFactory for the application. In this case, 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).

Ex: Path as a Hadoop filesystem URI is,

h*dfs://localhost/user/tom/quangle.txt.*

- FileSystem is a general filesystem API, so the first step is **to retrieve an instance for the filesystem** we want to use—HDFS in this case.

- There are several static factory methods for getting a FileSystem instance:

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

- A Configuration object encapsulates a client or server's configuration, which is set using configuration files read from the classpath ***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.

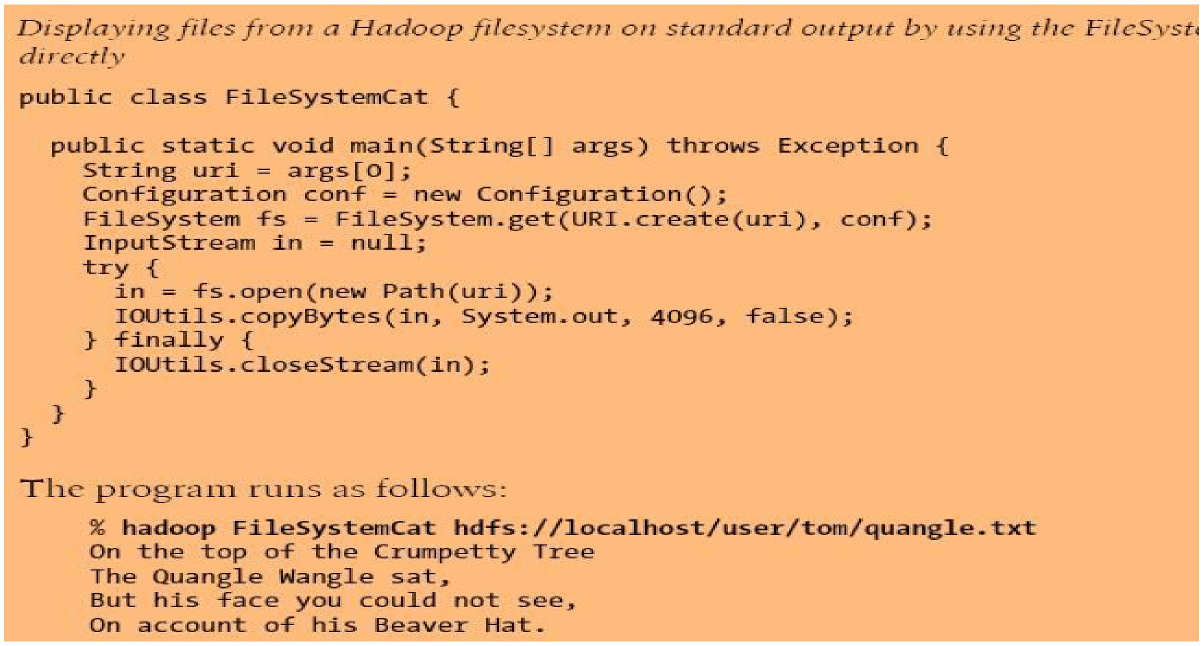
- In some cases, to retrieve a local filesystem instance, use the convenience method, getLocal(): **public static LocalFileSystem getLocal(Configuration conf) throws IOException**

- With a FileSystem instance 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.



Displaying files from a Hadoop filesystem on standard output by using the FileSyst 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

}

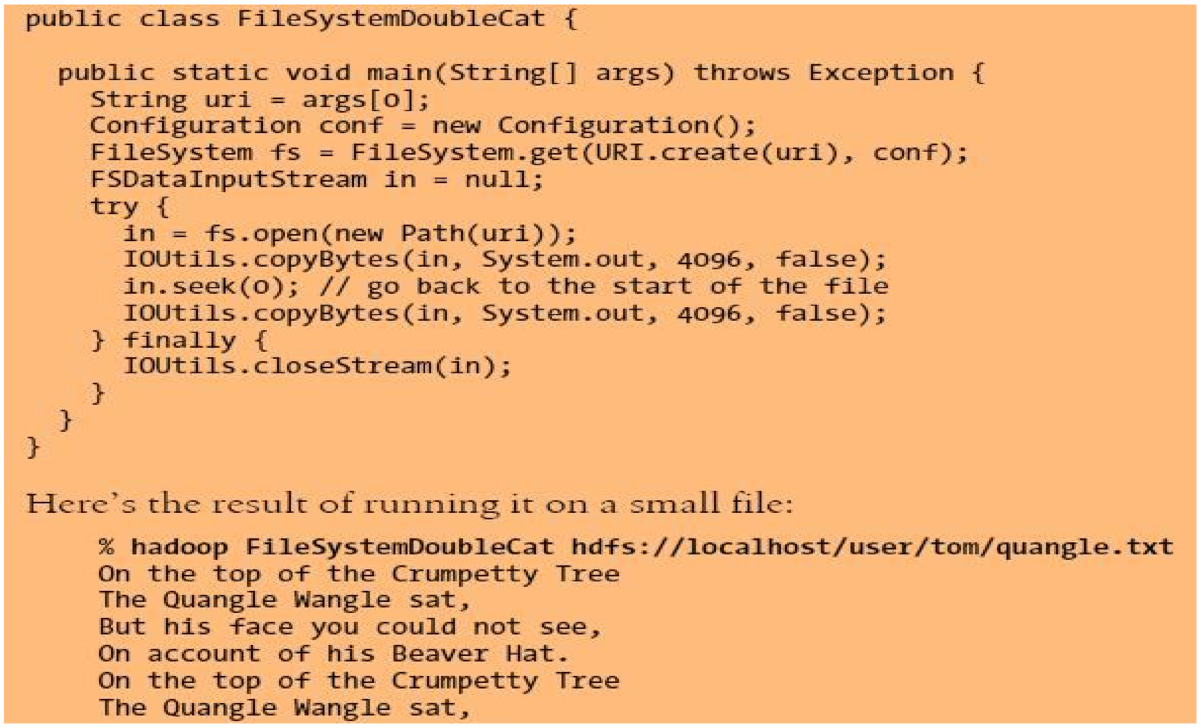
- The Seekable interface permits seeking to a position in the file and a query method for the current offset from the start of the file (getPos()):

```
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

A simple extension of previous example 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.

Ex: 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:

```
% 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,
```

- 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
- The readFully() methods will read length bytes into the buffer unless the end of the file is reached, in which case an EOFException is thrown.

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 FSDataOutputStream create(Path f) throws IOException
```

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

- **create()** methods create any parent directories of the file to be written that don't already exist.

- **exists()** method check for the existence of the parent directory.

- **Progressable()** is an overloaded method for passing a callback interface, 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();

}
```

- **append()** method is an alternative to creating a new file. It allows a single writer to modify an already written file by opening it.

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

Below Example shows how to copy a local file to a Hadoop filesystem. We illustrate progress 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.

```
public class FileCopyWithProgress {

public static void main(String[] args) throws Exception {
String localSrc = args[0];
String dst = args[1];

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

FileSystem fs = FileSystem.get(URI.create(dst), conf);
OutputStream out = fs.create(new Path(dst), new Progressable() {
public void progress() {
System.out.print(".");
}
});
IOUtils.copyBytes(in, out, 4096, true); }}
```

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

FSDataOutputStream

The create() method on FileSystem returns an FSDataOutputStream, has a method for querying the current position in the file:

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

```
public class FSDataOutputStream extends DataOutputStream
implements Syncable { public long getPos() throws
IOException {

//    implementation elided
}
//    implementation elided
}
```

FSDataInputStream, FSDataOutputStream permit seeking by using getPos() method, but HDFS does not support for writing to anywhere other than the end of the file, so there is no value in being able to seek while writing.

Directories

FileSystem provides a method to create a directory:

```
public boolean mkdirs(Path f) throws IOException
```

- This method creates all of the necessary parent directories if they don't already exist,just like the java.io.File's mkdirs() method.

- It returns true if the directory (and all parent directories) was (were) successfully created.

- No need to explicitly create a directory, since writing a file, by calling create(), will automatically create any parent directories.

Querying the Filesystem

File metadata: FileStatus

- An important feature of any filesystem is the ability to navigate its directory structure and retrieve information about the files and directories that it stores.
- FileStatus class encapsulates filesystem metadata for files and directories, including file length, block size, replication, modification time, ownership, and permission information.

getFileStatus():

This method is used for getting a FileStatus object for a single file or directory.

Demonstrating file status information

```
public class ShowFileStatusTest {
private MiniDFSCluster cluster; // use an in-process HDFS cluster
for testing private
FileSystem fs;
@Before
public void setUp() throws IOException {
Configuration conf = new Configuration();
if (System.getProperty("test.build.data") == null) {
System.setProperty("test.build.data", "/tmp");
}
cluster = new MiniDFSCluster(conf, 1, true, null);

fs = cluster.getFileSystem();
```

```
OutputStream out = fs.create(new Path("/dir/file"));
out.write("content".getBytes("UTF-8"));
out.close();
}
@After
public void tearDown() throws IOException {
if (fs != null) { fs.close(); }
if (cluster != null) { cluster.shutdown(); }
}
@Test(expected = FileNotFoundException.class)

public void throwsFileNotFoundForNonExistentFile()
throws IOException { fs.getFileStatus(new Path("no-
such-file")); }

@Test
public void fileStatusForFile() throws IOException {

Path file = new Path("/dir/file");
FileStatus stat = fs.getFileStatus(file);
assertThat(stat.getPath().toUri().getPath(),
is("/dir/file"));        assertThat(stat.isDir(),
is(false)); assertThat(stat.getLen(), is(7L));
assertThat(stat.getModificationTime(),
is(lessThanOrEqualTo(System.currentTim
eMillis())));
```

```
assertThat(stat.getReplication(), is((short)
1)); assertThat(stat.getBlockSize(), is(64 *
1024                    *                    1024L));
assertThat(stat.getOwner(),       is("tom"));
assertThat(stat.getGroup(),
is("supergroup"));
assertThat(stat.getPermission().toString()
, is("rw-r--r--")); }

@Test

public    void    fileStatusForDirectory()
throws  IOException  {  Path  dir  =  new
Path("/dir");

FileStatus   stat   =   fs.getFileStatus(dir);
assertThat(stat.getPath().toUri().getPath(),
is("/dir")); assertThat(stat.isDir(), is(true));
assertThat(stat.getLen(),                 is(0L));
assertThat(stat.getModificationTime(),
is(lessThanOrEqualTo(System.currentTim
eMillis())));
assertThat(stat.getReplication(),  is((short)
0)); assertThat(stat.getBlockSize(), is(0L));
assertThat(stat.getOwner(),        is("tom"));
assertThat(stat.getGroup(),
is("supergroup"));
assertThat(stat.getPermission().toString(),
is("rwxr-xr-x")); }

}
```

If no file or directory exists, a FileNotFoundException is thrown, to check the existence of a file or directory, then use **exists()** method.

public boolean exists(Path f) throws IOException

Listing files

FileSystem's listStatus() methods used for finding information on a single file or directory.

public FileStatus[] listStatus(Path f) throws IOException

public FileStatus[] listStatus(Path f, PathFilter filter) throws IOException public FileStatus[] listStatus(Path[] files) throws IOException

public FileStatus[] listStatus(Path[] files, PathFilter filter) throws IOException

If the argument is a file, it returns an array of FileStatus objects of length 1.

If the argument is a directory, it returns zero or more FileStatus objects representing the files and directories contained in the directory.

PathFilter used to restrict the files and directories to match. This is useful for building up lists of input files to process from distinct parts of the filesystem tree. i.e which allows programmatic control over matching.

Showing the file statuses for a collection of paths in a Hadoop filesystem

```
public class ListStatus {

public static void main(String[] args) throws Exception {

String uri = args[0];

Configuration conf = new
Configuration(); FileSystem fs =
FileSystem.get(URI.create(uri), conf);
Path[] paths = new Path[args.length];
for (int i = 0; i < paths.length; i++) {

paths[i] = new Path(args[i]);
}
FileStatus[] status = fs.listStatus(paths);
Path[] listedPaths = FileUtil.stat2Paths(status);
for (Path p : listedPaths) {
System.out.println(p);
}}}
```

This program is used to find the union of directory listings for a collection of paths:

```
% hadoop ListStatus hdfs://localhost/
hdfs://localhost/user/tom
hdfs://localhost/user
hdfs://localhost/user/tom/books
hdfs://localhost/user/tom/quangle.txt
```

File patterns

- To process sets of files in a single operation.Ex:MapReduce job for log processing might analyze a month's worth of files contained in a number of directories.

- use wildcard characters to match multiple files with a single expression, rather than to enumerate each file and directory to specify the input. This operation is known as globbing.

- Hadoop provides two FileSystem method for processing globs:

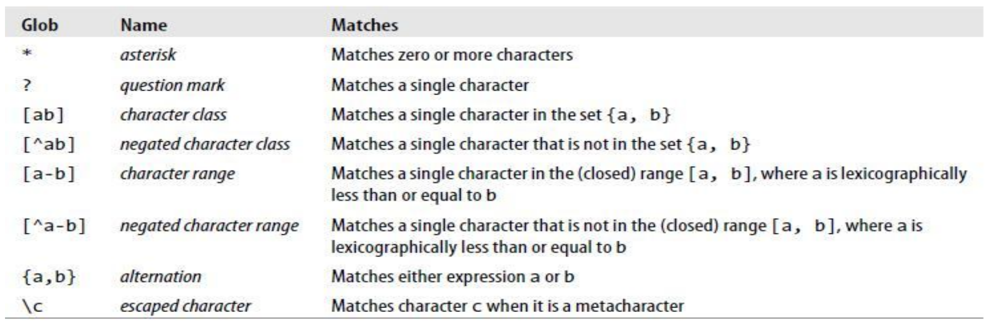
  public FileStatus[] globStatus(Path pathPattern) throws IOException
  public FileStatus[] globStatus(Path pathPattern, PathFilter filter) throws IOException

- The globStatus() method returns an array of FileStatus objects whose paths matchthe supplied pattern, sorted by path.

- An optional PathFilter can be specified to restrict the matches further.

Hadoop supports the same set of glob characters as Unix bash

Glob characters and their meanings



Glob	Name	Matches
`*`	*asterisk*	Matches zero or more characters
`?`	*question mark*	Matches a single character
`[ab]`	*character class*	Matches a single character in the set `{a, b}`
`[^ab]`	*negated character class*	Matches a single character that is not in the set `{a, b}`
`[a-b]`	*character range*	Matches a single character in the (closed) range `[a, b]`, where a is lexicographically less than or equal to b
`[^a-b]`	*negated character range*	Matches a single character that is not in the (closed) range `[a, b]`, where a is lexicographically less than or equal to b
`{a,b}`	*alternation*	Matches either expression a or b
`\c`	*escaped character*	Matches character c when it is a metacharacter

Ex: logfiles are stored in a directory structure organized hierarchically by date here the last day of 2007 would go in a directory named /2007/12/31.

Suppose that the full file listing is:

```
/2007/12/30
/2007/12/31
/2008/01/01
/2008/01/02
```

Glob Expansion for the above files

```
/* /2007 /2008
/*/* /2007/12 /2008/01
/*/12/* /2007/12/30 /2007/12/31

/200? /2007 /2008
/200[78] /2007 /2008
/200[7-8] /2007 /2008
/200[^01234569] /2007 /2008
/*/*/{31,01} /2007/12/31 /2008/01/01
/*/*/3{0,1} /2007/12/30 /2007/12/31
/*/{12/31,01/01} /2007/12/31 /2008/01/01
```

Deleting Data

delete() method on FileSystem used to permanently remove files or directories:

public boolean delete(Path f, boolean recursive) throws IOException

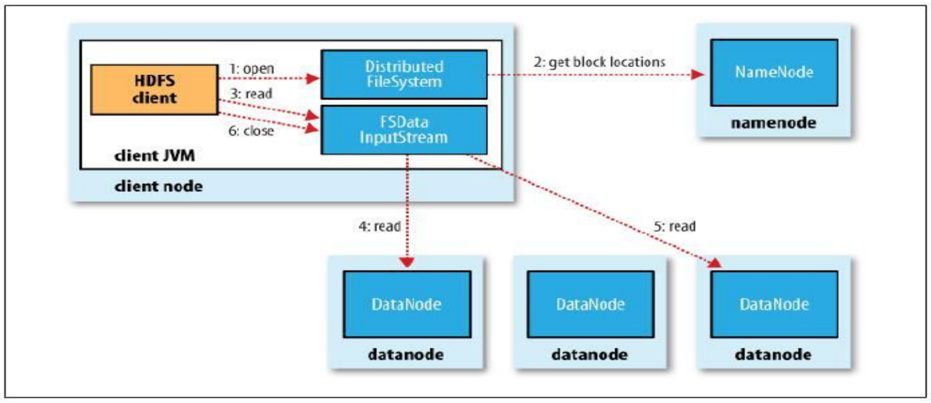
- If f is a file or an empty directory, then the value of recursive is ignored

- if recursive is true a nonempty directory is deleted, along with its contents. otherwise an IOException is thrown

Data Flow

3.6 Anatomy of a File Read

When reading a file how data flows between the client interacting with HDFS, the namenode and the datanodes, consider Figure which shows the main sequence of events



A client reading data from HDFS

Step 1 : The client opens the file to be read by calling open() on the FileSystem object, which for HDFS is an instance of DistributedFileSystem .

Step 2 : DistributedFileSystem calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file

- For each block, the namenode returns the addresses of the datanodes that have a copy of that block.
- The datanodes are sorted according to their proximity to the client.
- If the client is itself a datanode then it will read from the datanode, if it hosts a copy of the block.

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

Step 3 : The client then calls read() on the stream.

- DFSInputStream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file.

Step 4 : Data is streamed from the datanode back to the client, which calls read() repeatedly on the stream.

Step 5 : When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block.

- This is transparent to the client, which from its point of view is just reading a continuous stream.
- Blocks are read in order with the DFSInputStream opening new connections to datanodes as the client reads through the stream.
- It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed.

Step 6 : When the client has finished reading, it calls close() on the FSDataInputStream.

- During reading, if the DFSInputStream encounters an error while communicating with a datanode, then it will try the next closest one for that block .

- The DFSInputStream also verifies checksums for the data transferred to it from the datanode.

- If a corrupted block is found, it is reported to the namenode before the DFSInput Stream attempts to read a replica of the block from another datanode.

- Important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block.

- This design allows HDFS to scale to a large number of concurrent clients, since the data traffic is spread across all the datanodes in the cluster.

- The namenode has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottleneck as the number of clients grew.

Network Topology and Hadoop

What does it mean for two nodes in a local network to be “close” to each other?

In the context of high-volume data processing, the limiting factor is the rate at which we can transfer data between nodes—bandwidth is a scarce commodity.

The idea is to use the bandwidth between two nodes as a measure of distance.

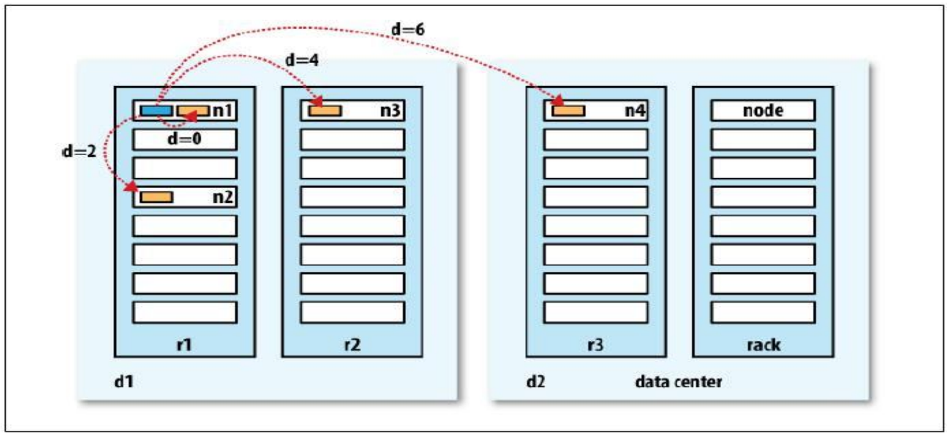
Rather than measuring bandwidth between nodes, which can be difficult to do in practice Hadoop takes a simple approach in which the network is represented as a tree and the distance between two nodes is the sum of their distances to their closest common ancestor bandwidth available for each of the following scenarios becomes progressively less:

- Processes on the same node
- Different nodes on the same rack
- Nodes on different racks in the same data center
- Nodes in different data centers7

For example, imagine a node n1 on rack r1 in data center d1. This can be represented as /d1/r1/n1.

Using this notation, here are the distances for the four scenarios:

- distance(/d1/r1/n1, /d1/r1/n1) = 0 (processes on the same node)
- distance(/d1/r1/n1, /d1/r1/n2) = 2 (different nodes on the same rack)
- distance(/d1/r1/n1, /d1/r2/n3) = 4 (nodes on different racks in the same data center)
- distance(/d1/r1/n1, /d2/r3/n4) = 6 (nodes in different data centers)



Network distance in Hadoop

3.7 Anatomy of a File Write

How files are written to HDFS. To understand the data flow since it clarifies HDFS's coherency model, creating a new file, writing data to it.

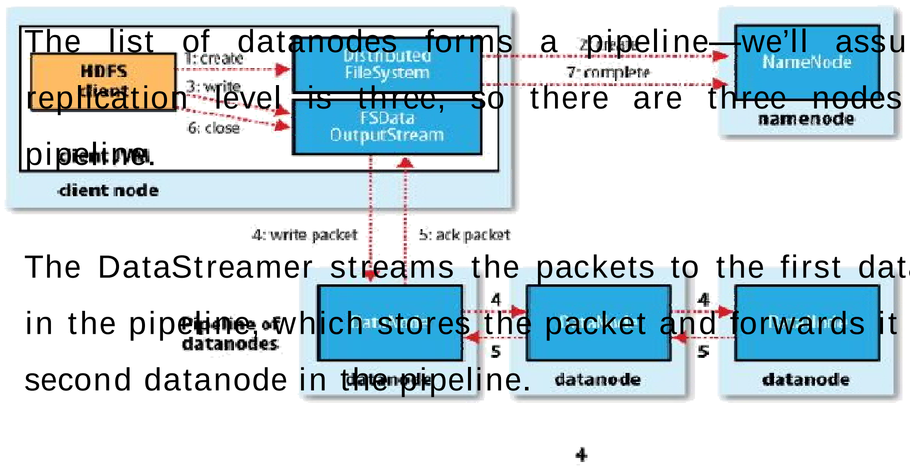
Step 1 : The client creates the file by calling create() on DistributedFileSystem.

Step 2 : DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it.

- The namenode performs various checks to make sure the file doesn't already exist, and that the client has the right permissions to create the file.
- If these checks pass, the namenode makes a record of the new file;
- otherwise, file creation fails and the client is thrown an IOException.
- The DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutput Stream, which handles communication with the datanodes and namenode.

Step 3 : Client writes data DFSOutputStream splits it into packets, which it writes to an internal queue, called the **data queue**.

- The data queue is consumed by the Data Streamer, whose responsibility it is to ask the 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.

Step 4 : The second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline.

- DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the **ack queue**.

Step 5 *:* A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline .

client writing data to HDFS

If a datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data.

- First the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets.
- The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on.

- The failed datanode is removed from the pipeline and the remainder of the block's data is written to the two good datanodes in the pipeline.

- The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal.

Step 6 : When the client has finished writing data, it calls close() on the stream.

3.8 Replica Placement

This strategy tells how does the namenode choose which datanodes to store replicas on? By taking into consider read bandwidth and write bandwidth.

- placing all replicas on a single node incurs the lowest write bandwidth penalty the replication pipeline runs on a single node, but this offers no real redundancy. (if the node fails, the data for that block is lost). Also, the read bandwidth is high for off-rack reads.

- placing replicas in different data centers may maximize redundancy, but at the cost of bandwidth.

- There are a variety of placement strategies.

Hadoop's default strategy :

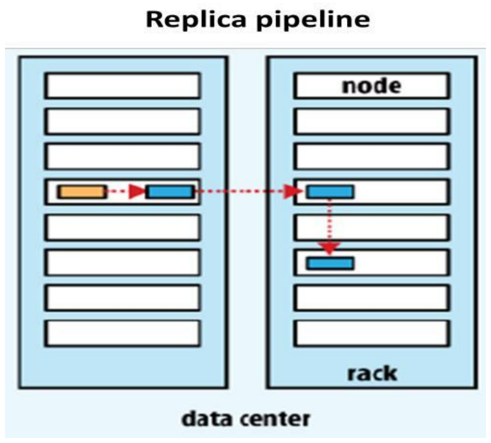
- place the **first replica** on the same node as the client (node is chosen at random, although the system tries not to pick nodes that are too full or too busy).

- The **second replica** is placed on a different rack from the first (*off-rack), chosen at random.*

- *The* **third replica** *is placed on* the same rack as the second, but on a different node chosen at random.

- **Further replicas** are placed on random nodes on the cluster, although the system tries to avoid placing too many replicas on the same rack.

Once the replica locations have been chosen, a pipeline is built, taking network topology into account



3.9 Coherency Model

- Coherency model for a filesystem describes the data visibility of reads and writes for a file.

- After creating a file, it is visible in the filesystem namespace, as expected:

```
Path p = new Path("p");
fs.create(p);

assertThat(fs.exists(p), is(true));
```

- Any content written to the file is not guaranteed to be visible, even if the stream is flushed. So the file appears to have a length of zero:

```
Path p = new Path("p");
OutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();
assertThat(fs.getFileStatus(p).getLen(), is(0L));
```

- Once more than a block's worth of data has been written, the first block will be visible to new readers.

- This is true of subsequent blocks, too: it is always the current block being written that is not visible to other readers.

- HDFS provides a method for forcing all buffers to be synchronized to the datanodes via the sync() method on FSDataOutputStream.

- After a successful return from sync(), HDFS guarantees that the data written up to that point in the file is persisted and visible to all new readers:

```
Path p = new Path("p");

FSDataOutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();
out.sync();
assertThat(fs.getFileStatus(p).getLen(),is(((long)
"content".length())));
```

Ex: Using the standard Java API to write a local file, we are guaranteed to see the content after flushing the stream and synchronizing:

```
FileOutputStream out = new
FileOutputStream(localFile);
out.write("content".getBytes("UTF-8"));

out.flush(); // flush to operating system
out.getFD().sync(); // sync to disk
assertThat(localFile.length(), is(((long)
"content".length())));
```

Closing a file in HDFS performs an implicit sync(), too:

```
        Path p = new Path("p");
        OutputStream out = fs.create(p);
        out.write("content".getBytes("UTF-8"));
        out.close();
assertThat(fs.getFileStatus(p).getLen(), is(((long) "content".length())));
```

Consequences for application design

This coherency model has implications for the way you design applications.

With no calls to sync(), you should be prepared to lose up to a block of data in the event of client or system failure, this is unacceptable for many applications.

So call sync() at suitable points, such as after writing a certain number of records or number of bytes.

This is overhead, so there is a trade-off between data robustness and throughput.

Acceptable trade-off is application-dependent, and suitable values can be selected after measuring your application’s performance with different sync() frequencies.

3.10 Parallel Copying with distcp

- The distcp copying large amounts of data to and from Hadoop filesystems in parallel.

- The canonical use case for distcp is for transferring data between two HDFS clusters. % **hadoop distcp hdfs://namenode1/foo hdfs://namenode2/bar**

- This will copy the /foo directory (and its contents) from the first cluster to the /bar directory on the second cluster

- If /bar doesn't exist, it will be created first .

- By default, distcp will skip files that already exist in the destination, but they can be overwritten by supplying the **-overwrite** option. You can also update only files that have changed using the **- update** option.

% **hadoop distcp -update hdfs://namenode1/foo hdfs://namenode2/bar/foo**

- The options to control the behavior of *distcp, are preserve file* attributes, ignore failures, and limit the number of files or total data copied.

- *distcp is implemented as a MapReduce job where the work of copying is done by the* maps that run in parallel across the cluster. There are no reducers.

- Each file is copied by a single map, and *distcp tries to give each map approximately the same amount of* data, by bucketing files into roughly equal allocations.

No. of Maps

- The number of maps is decided as follows.

- Each map copies at least 256 MB (unless the total size of the input is less, in which case one map handles it all).

  Ex: 1 GB of files will be given four map tasks.

- When the data size is very large, it becomes necessary to limit the number of maps in order to limit bandwidth and cluster utilization.

- By default, the maximum number of maps is 20 per (tasktracker) cluster node

- -m argument to distcp used to specify number of maps. **Ex** : -m 1000 would allocate 1,000 maps.

- Use distcp between two HDFS clusters that are running different versions, the copy will fail if you use the hdfs protocol, since the RPC systems are incompatible.

- Use the read-only HTTP-based HFTP filesystem to read from the source. The job must run on the destination cluster so that the HDFS RPC versions are compatible.

Ex: using HFTP:

```
% hadoop distcp hftp://namenode1:50070/foo hdfs://namenode2/bar
```

- Need to specify the namenode's web port in the source URI. This is determined by the dfs.http.address property, which defaults to 50070.

- Using the newer webhdfs protocol (which replaces hftp) it is possible to use HTTP for both the source and destination clusters without hitting any wire incompatibility problems.

```
%hadoop distcp webhdfs://namenode1:50070/foo
 webhdfs://namenode2:50070/bar
```

- Another variant is to use an HDFS HTTP proxy as the distcp source or destination, which has the advantage of being able to set firewall and bandwidth controls.

3.11 Keeping an HDFS Cluster Balanced

- When copying data into HDFS, it is important to consider cluster balance.

- HDFS works best when the file blocks are evenly spread across the cluster, so you want to ensure that distcp doesn't disrupt this.

- Ex: 1,000 GB data specifying -m1 a single map do the copy not using the cluster resources efficiently— the first replica of each block would reside on the node running the map (until the disk filled up).

- The second and third replicas would be spread across the cluster, but this one node would be unbalanced.

- By having more maps than nodes in the cluster, this problem is avoided— it’s best to start by running distcp with the default of 20 maps per node. If it is not possible use use the balancer tool for load distribution across the cluster .

UNIT-III
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. HDFS is designed for_____________ [ ]
   A) Storing very large files C) Commodity Hardware
   B) Streaming data access D) All
2. The default HDFS port is _____________.
3. Distcp command used for copy large blocks of data across the cluster [True/False]
4. HDFS is _____________Architecture.
5. Data node is __________daemon. [ ]
   A) Storage B) Computing C) Server D) None
6. ________________ model for a file system describe the data visibility of reads and writes for a file. [ ]
   A) Map Reduce B) Coherency C) HDFS D) Pig
7. Use____tool for load distribution across the cluster. [ ]
   A. Loader B) Distributer C) Balancer D) None
8. On a fully configured cluster, "running Hadoop" means running___ daemons on the different servers in the network. [ ]
   A) NameNode, DataNode C) JobTracker,TaskTracker
   B) Secondary NameNode D) All
9. What mode that a Hadoop can run? [ ]
   A) Standalone C) Fully Distributed Mode
   B) Pseudo-Distributed mode D) All
10. For reading/Writing data to/from HDFS.Clients first connect to[ ]
    A) Name Node C) Secondary Name Node
    B) Data Node D) none
11. The main goal of HDFS high availability is___________. [ ]
    A) Faster creation of the replicas of primary namenode.
    B) To reduce the cycle time required to bring back a new primary namenode after existing primary fails.

C) Prevent data loss due to failure of primary namenode.

D) Prevent the primary namenode form becoming single point of failure.

12. A Negative aspect to the importance of the Name Node___. [ ]

A) Single point of failure C) No failure

B) Double point of failure D) None.

13. What is the way of accessing HDFS over HTTP [ ]

A) Direct B) via proxy C) Both A & B D) None

14. If we use Cloudera distributation of hadoop which is the default directory fo HDFS [ ]

A) /home/cloud era C) /cloudera

B) /user/cloudera D) None

15. The information mapping data blocks with their corresponding files is stored in [ ]

A) Data Node C) Job Tracker

B) Name Node D) Task Tracker

16. What happen if number of reducer is 0 in Hadoop? [ ]

A) Map-only job take place

B)) Reduce-only job take place.

C) Reducer output will be the final output D) None

17. The HDFS command to create the copy of a file from a local system is which of the following? [ ]

A) copyFromLocal C) copyfromlocal

B) CopyFromLocal D) copylocal

18. In order to read any file in HDFS, instance of [ ]

A) fileSystem B) datastream C) outstream D) inputstream

19. is method to copy byte from input stream to any other stream in Hadoop.

A) Iutils B) Utils C) IOUtils D) All [ ]

20. The daemons associated with the Map Reduce phase are______ and Task-Trackers. [ ]

A) Job Tracker C) Reduce Trackers

B) Map Tracker D) All

SECTION-B

SUBJECTIVE QUESTIONS

1. Distinguish distributed file system and HDFS? In what areas HDFS does not work well.
2. Outline the architecture of HDFS.
3. Write the benefits of Distributed File System having block abstraction.
4. List some concrete File System implementations.
5. What methods are required for querying the current position in the file?
6. Write the methods for creating directory an display its status.
7. What is the default replica placement strategy? Explain.
8. Explain about parallel copying with distcp
9. List out glob characters supported by Hadoop.
10. How to handle failure of Name Node? Explain
11. Defend how to achieve High-Availability in HDFS.
12. Identify various Hadoop daemons and explain their roles in a Hadoop Cluster.
13. Why interface is required for HDFS? Explain different types of interfaces to HDFS.
14. Develop the code for reading data from a Hadoop URL.
15. Develop the code for reading data using the File System API.
16. Examine anatomy of file read with a neat diagram.
17. Sketch and explain Anatomy of file write.
18. How to keep balance of HDFS cluster?

Unit 4

Developing a MapReduce application

Objective: To familiarize with the Map Reduce development Environment

Syllabus: Analyzing data with unix tools, Analyzing data with hadoop, Java MapReduce classes(new API), Data flow, Combiner functions, Running a distributed MapReduce Job

Learning Outcomes:

At the end of the unit, students will be able to:

1. Develop Map reduce configuration files
2. Explain the managing configuration .
3. Develop the test cases for Map Reduce
4. Develop the web Interface for Map Reduce.

Learning Material

4.1 Setting up the development environment

The first step is to create a project so you can build mapreduce programs and run them in local (standalone) mode from the command line or within IDE.

The Maven POM show the dependencies needed for building and testing mapreduce programs.

Apache Maven is a build automation tool that can be used for java projects.

4.2 Managing Configuration

The entire apache Hadoop ecosystem is written in java, Maven is a great tool for managing projects that build on the top of the Hadoop APIs.

1. The first step is to download the version of Hadoop that you plan to use and unpack it on your development machine
2. IDE, create a new project and add all the JAR files from the top level of the unpacked distribution and from the *lib* directory to the classpath.
3. Run on a local “pseudo distributed” cluster.

Hadoop configuration files containing the connection settings for each cluster you run and specify which one you are using.

conf directory contains three configuration files: hadoop-local.xml, hadoop-localhost.xml, and hadoop-cluster.xml

The *hadoop-local.xml* file contains the default Hadoop configuration for the default filesystem and the jobtracker:

```
<?xml version="1.0"?>
<configuration>
<property>
<name>fs.default.name</name>
<value>file:///</value>
</property>
<property>
<name>mapred.job.tracker</name>
<value>local</value>
 </property>
</configuration>
```

The settings in *hadoop-localhost.xml* point to a namenode and a jobtracker both running on localhost:

```
<?xml version="1.0"?>
<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://localhost/</value>
</property>
<property>
<name>mapred.job.tracker</name>
<value>localhost:8021</value>
</property>
</configuration>
```

Finally, *hadoop-cluster.xml* contains details of the cluster's namenode and jobtracker addresses

```
<?xml version="1.0"?>
 <configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://namenode/</value>
</property>
<property>
<name>mapred.job.tracker</name>
<value>jobtracker:8021</value>
</property>
</configuration>
```

You can add other configuration properties to these files as needed. For example, if you wanted to set your Hadoop username for a particular cluster, directory listing on the HDFS server running in pseudo distributed mode on local host:

```
% hadoop fs -conf conf/hadoop-localhost.xml -ls .
Found 2 items
drwxr-xr-x - tom supergroup 0 2009-04-08 10:32 /user/tom/input
drwxr-xr-x - tom supergroup 0 2009-04-08 13:09 /user/tom/output
```

Installing apache Hadoop

Prerequisites Hadoop is written in java, need to install version 6 or later. Hadoop runs on unix and on windows. Linux is the only supported production platform, but other flavors of unix(including Mac OS X) can be used to run hadoop for development. Windows is only supported as a development platform, and additionally requires Cygwin to run.

Hadoop can be run in one of the three modes:

1. Standalone(or local) mode

There are no daemons running and everything runs in a single JVM. Standalone mode is suitable for running mapreduce programs during development, since it is easy to test and debug them.

2. **Pseudo distributed mode**

The Hadoop daemons run on the local machine, thus simulating a cluster on a small scale.

3. Fully distributed mode

The Hadoop daemons run on a cluster of machines.

GenericOptionsParser, Tool, and ToolRunner

- Hadoop comes with a few helper classes for making it easier to run jobs from the command line.
- GenericOptionsParser is a class that interprets common Hadoop command-line options and sets them on a Configuration object for your application.
- You don't usually use GenericOptionsParser directly, as it's more convenient to implement the Tool interface and run your application with the ToolRunner, which uses GenericOptionsParser internally:

```
public interface Tool extends Configurable
{
int run(String [] args) throws Exception;
}
```

Example: Tool implementation for printing the properties in a Configuration

```
public class ConfigurationPrinter extends Configured implements Tool {
static {
Configuration.addDefaultResource("hdfs-default.xml");
Configuration.addDefaultResource("hdfs-site.xml");
Configuration.addDefaultResource("mapred-default.xml");
Configuration.addDefaultResource("mapred-site.xml");
}
@Override
public int run(String[] args) throws Exception {
Configuration conf = getConf();
for (Entry<String, String> entry: conf) {
System.out.printf("%s=%s\n", entry.getKey(), entry.getValue());
}
return 0;
}
public static void main(String[] args) throws Exception {
int exitCode = ToolRunner.run(new ConfigurationPrinter(), args);
System.exit(exitCode);
 }
}
```

- Make ConfigurationPrinter a subclass of Configured, which is an implementation of the Configurable interface. All implementations of Tool need to implement Configurable
- The run () method obtains the Configuration using Configurable's getConf() method and then iterates over it, printing each property to standard output.
- ConfigurationPrinter's main() method does not invoke its own run() method directly.
- Instead, we call ToolRunner's static run() method, which takes care of creating a Configuration object for the Tool, before calling its run() method.
- ToolRunner also uses a GenericOptionsParser to pick up any standard options specified on the command line and set them on the Configuration instance .

GenericOptionParser also allows you to set individual properties. For example

```
% hadoop ConfigurationPrinter -D color=yellow | grep color
color=yellow
```

The -D option is used to set the configuration property with key color to the value yellow.

4.3 Writing a unit test with MRUnit

- The map and reduce functions in MapReduce are easy to test in isolation
- For known inputs, they produce known outputs.
- Since outputs are written to a Context (or an OutputCollector in the old API), rather than simply being returned from the method call, the Context needs to be replaced with a mock so that its outputs can be verified.
- All of the tests described here can be run from within an IDE.
- Below example shows how to test mapper
- Here it passes a weather record as input to the mapper, then checks the output is the year and temperature reading
- The input key is ignored by the mapper, so we can pass in anything, including null as we do here.
- To create a mock Context, we call Mockito's mock() method (a static import), passing the class of the type we want to mock.
- Then we invoke the mapper's map() method, which executes the code being tested.
- Finally, we verify that the mock object was called with the correct method and arguments, using Mockito's verify() method
- Here we verify that Context's write() method was called with a Text object representing the year (1950) and an IntWritable representing the temperature (−1.1°C).

3. Writing a Unit Test with MRUnit

The map and reduce functions in MapReduce are easy to test in isolation, which is a consequence of their functional style. MRUnit is a testing library that makes it easy to pass known inputs to a mapper or a reducer and check that the outputs are as expected. MRUnit is used in conjunction with a standard test execution framework, such as JUnit, so you can run the tests for MapReduce jobs in your normal development environment. For example, all of the tests described here can be run from within an IDE by following the instructions in Setting Up the Development Environment.

Mapper

The test for the mapper is shown in Example 6-5.

Example 6-5. Unit test for MaxTemperatureMapper

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.junit.*;
public class MaxTemperatureMapperTest{
@Test
public void processesValidRecord() throws IOException, InterruptedException
{
Text value = new Text("0043011990999991950051518004+68750+023550FM-
12+0382" +
// Year ^^^^
"99999V0203201N00261220001CN9999999N9-00111+99999999999");
// Temperature ^^^^^
new MapDriver<LongWritable, Text, Text, IntWritable>()
.withMapper(new MaxTemperatureMapper())
.withInput(new LongWritable(0), value)
.withOutput(new Text("1950"), new IntWritable(-11))
.runTest();
}
}
```

The idea of the test is very simple: pass a weather record as input to the mapper, and check that the output is the year and temperature reading. Since we are testing the mapper, we use MRUnit'sMapDriver, which we configure with the mapper under test (MaxTemperatureMapper), the input key and value, and the expected output key (a Text object representing the year, 1950) and expected output value (an IntWritable representing the temperature, -1.1°C), before finally calling the runTest() method to execute the test. If the expected output values are not emitted by the mapper, MRUnit will fail the test. Notice that the input key could be set to any value because our mapper ignores it. Proceeding in a test-driven fashion, we create a Mapper implementation that passes the test (see Example 6-6). Because we will be evolving the classes in this chapter, each is put in a different package indicating its version for ease of exposition. For example, v1.MaxTemperatureMapper is version 1 of MaxTemperatureMapper. In reality, of course, you would evolve classes without repackaging them. *Example 6-6. First version of a Mapper that passes*

MaxTemperatureMapperTest

```
public class MaxTemperatureMapper
extends Mapper<LongWritable, Text, Text, IntWritable> {
@Override
public void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
String line = value.toString();
String year = line.substring(15, 19);
intairTemperature = Integer.parseInt(line.substring(87, 92));
context.write(new Text(year), new IntWritable(airTemperature));
}
}
```

This is a very simple implementation that pulls the year and temperature fields from theline and writes them to the Context.

Reducer

The reducer has to find the maximum value for a given key. Here's a simple test for this feature, which uses a ReduceDriver:

```
@Test
public void returnsMaximumIntegerInValues() throws IOException,
InterruptedException {
new ReduceDriver<Text, IntWritable, Text, IntWritable>()
.withReducer(new MaxTemperatureReducer())
.withInput(new Text("1950"),
Arrays.asList(new IntWritable(10), new IntWritable(5)))
.withOutput(new Text("1950"), new IntWritable(10))
.runTest();
}
```

We construct a list of some IntWritable values and then verify hatMaxTemperatureReducer picks the largest. The code in Example 6-9 is for animplementation of MaxTemperatureReducer that passes the test.

Example 6-9. Reducer for the maximum temperature example

```
public class MaxTemperatureReducer
extends Reducer<Text, IntWritable, Text, IntWritable> {
@Override
public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
intmaxValue = Integer.MIN_VALUE;
for (IntWritable value : values) {
```

```
maxValue = Math.max(maxValue, value.get());
}
context.write(key, new IntWritable(maxValue));
}
}
```

Running Locally on Test Data

Now that we have the mapper and reducer working on controlled inputs, the next step is to write a job driver and run it on some test data on a development machine.

Running a Job in a Local Job Runner

Using the Tool interface introduced earlier in the chapter, it’s easy to write a driver to run our MapReduce job for finding the maximum temperature by year (see MaxTemperatureDriver in Example 6-10). *Example 6-10. Application to find the maximum temperature*

```
public class MaxTemperatureDriver extends Configured implements Tool {
@Override
public intrun(String[] args) throws Exception {
if (args.length != 2) {
System.err.printf("Usage: %s [generic options] <input><output>\n",
getClass().getSimpleName());
ToolRunner.printGenericCommandUsage(System.err);
return -1;
}
 Job job = new Job(getConf(), "Max temperature");
job.setJarByClass(getClass());
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.setMapperClass(MaxTemperatureMapper.class);
job.setCombinerClass(MaxTemperatureReducer.class);
job.setReducerClass(MaxTemperatureReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
return job.waitForCompletion(true) ? 0 : 1;
}
public static void main(String[] args) throws Exception {
intexitCode = ToolRunner.run(new MaxTemperatureDriver(), args);
System.exit(exitCode);
}
}
```

MaxTemperatureDriver implements the Tool interface, so we get the benefit of being able to set the options that GenericOptionsParser supports. The run() method constructs a Job object based on the tool's configuration, which it uses to launch a job. Among the possible job configuration parameters, we set the input and output file paths; the mapper, reducer, and combiner classes; and the output types (the input types are determined by the input format, which defaults to TextInputFormat and has LongWritable keys and Text values). It's also a good idea to set a name for the job (Max temperature) so that you can pick it out in the job list during execution and after it has completed. By default, the name is the name of the JAR file, which normally is not particularly descriptive. Now we can run this application against some local files. Hadoop comes with a local job runner, a cut-down version of the MapReduce execution engine for running MapReduce jobs in a single JVM. It's designed for testing and is very convenient for use in an IDE, since you can run it in a debugger to step through the code in your mapper and reducer. The local job runner is used if mapreduce.framework.name is set to local, which is the default.[49] From the command line, we can run the driver by typing:

```
% mvn compile
% export HADOOP_CLASSPATH=target/classes/
% hadoop v2.MaxTemperatureDriver -confconf/hadoop-local.xml \
input/ncdc/micro output
```

Equivalently, we could use the -fs and -jt options provided by GenericOptionsParser:

```
% hadoop v2.MaxTemperatureDriver -fs file:/// -jt local input/ncdc/micro
output
```

This command executes MaxTemperatureDriver using input from the local *input/ncdc/micro* directory, producing output in the local *output* directory. Note that although we've set -fs so we use the local filesystem (file:///), the local job runner will actually work fine against any filesystem, including HDFS (and it can be handy to do this if you have a few files that are on HDFS). We can examine the output on the local filesystem:

```
% cat output/part-r-00000
1949 111
1950 22
```

Testing the Driver

Apart from the flexible configuration options offered by making your application implement Tool, you also make it more testable because it allows you to inject an arbitrary Configuration. You can take advantage of this to write a test that uses a local job runner to run a job against known input data, which checks that the output is as expected.

There are two approaches to doing this. The first is to use the local job runner and run the job against a test file on the local filesystem. The code in Example

6-11 gives an idea of how to do this.

Example 6-11. A test for MaxTemperatureDriver that uses a local, in-process job runner

```
@Test
public void test() throws Exception {
Configuration conf = new Configuration();
conf.set("fs.defaultFS", "file:///");
conf.set("mapreduce.framework.name", "local");
conf.setInt("mapreduce.task.io.sort.mb", 1);
Path input = new Path("input/ncdc/micro");
Path output = new Path("output");
FileSystemfs = FileSystem.getLocal(conf);
fs.delete(output, true); // delete old output
MaxTemperatureDriver driver = new MaxTemperatureDriver();
driver.setConf(conf);
intexitCode = driver.run(new String[] {
input.toString(), output.toString() });
assertThat(exitCode, is(0));
checkOutput(conf, output);
}
```

The test explicitly sets fs.defaultFS and mapreduce.framework.name so it uses the local filesystem and the local job runner. It then runs the MaxTemperatureDriver via its Tool interface against a small amount of known data. At the end of the test, the checkOutput() method is called to compare the actual output with the expected output, line by line.
The **second**way of testing the driver is to run it using a "mini-" cluster. Hadoop has a set of testing classes, called MiniDFSCluster, MiniMRCluster, and MiniYARNCluster, that provide a programmatic way of creating in-process clusters. Unlike the local job runner, these allow testing against the full HDFS,

MapReduce, and YARN machinery. Bear in mind, too, that node managers in a mini-cluster launch separate JVMs to run tasks in,which can make debugging more difficult.

Mapper

The test for the mapper is shown

```
 import static org.mockito.Mockito.*;
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.junit.*;
public class MaxTemperatureMapperTest {
@Test
public void processesValidRecord() throws IOException, InterruptedException {
MaxTemperatureMapper mapper = new MaxTemperatureMapper();
Text value = new Text("0043011990999991950051518004+68750+023550FM-
12+0382" +
// Year ^^^^
"99999V0203201N00261220001CN9999999N9-00111+99999999999");
// Temperature ^^^^^
MaxTemperatureMapper.Context context =
mock(MaxTemperatureMapper.Context.class);
mapper.map(null, value, context);
verify(context).write(new Text("1950"), new IntWritable(-11));
}
}
```

We create a Mapper implementation that passes the test

```
public class MaxTemperatureMapper
extends Mapper<LongWritable, Text, Text, IntWritable> {
@Override
public void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
```

```
String line = value.toString();
String year = line.substring(15, 19);
int airTemperature = Integer.parseInt(line.substring(87, 92));
context.write(new Text(year), new IntWritable(airTemperature));
}
 }
```

This is a very simple implementation, which pulls the year and temperature fields from the line and writes them to the Context. Then add a test for missing values, which in the raw data are represented by a temperature of +9999:

```
@Test
public void ignoresMissingTemperatureRecord() throws IOException,
InterruptedException { MaxTemperatureMapper mapper = new
MaxTemperatureMapper();
Text value = new Text("0043011990999991950051518004+68750+023550FM-
12+0382" +
// Year ^^^^
"99999V0203201N00261220001CN9999999N9+99991+99999999999");
// Temperature ^^^^^
MaxTemperatureMapper.Context context =
mock(MaxTemperatureMapper.Context.class);
mapper.map(null, value, context);
verify(context, never()).write(any(Text.class), any(IntWritable.class));
 }
```

The existing test fails with a NumberFormatException, as parseInt() cannot parse integers with a leading plus sign, so we fix up the implementation (version 2) to handle missing values:

```
 @Override
public void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
String line = value.toString();
String year = line.substring(15, 19);
String temp = line.substring(87, 92);
if (!missing(temp)) {
int airTemperature = Integer.parseInt(temp);
context.write(new Text(year), new IntWritable(airTemperature));
}
}
private boolean missing(String temp) {
 return temp.equals("+9999");
}
```

With the test for the mapper passing, we move on to writing the reducer.

Reducer

The reducer has to find the maximum value for a given key. Here's a simple test for this feature:

```
@Test
public void returnsMaximumIntegerInValues() throws IOException,
InterruptedException {
 MaxTemperatureReducer reducer = new MaxTemperatureReducer();
 Text key = new Text("1950");
List<IntWritable> values = Arrays.asList(
 new IntWritable(10), new IntWritable(5));
MaxTemperatureReducer.Context context =
mock(MaxTemperatureReducer.Context.class);
reducer.reduce(key, values, context);
verify(context).write(key, new IntWritable(10)); }
```

construct a list of some IntWritable values and then verify that MaxTemperatureReducer picks the largest.

Ex:Reducer for maximum temperature example

```
 public class MaxTemperatureReducer
extends Reducer<Text, IntWritable, Text, IntWritable> {
@Override
public void reduce(Text key, Iterable<IntWritable> values,
Context context)
throws IOException, InterruptedException {
int maxValue = Integer.MIN_VALUE;
for (IntWritable value : values) {
maxValue = Math.max(maxValue, value.get()); }
context.write(key, new IntWritable(maxValue));
}
}
```

Mapper and reducer working on controlled inputs, the next step is to write a job driver and run it on some test data on a development machine.

4.4 Running a job in local job runner

Write a driver to run our MapReduce job for finding the maximum temperature by year.

Ex: Application to find the maximum temperature

```
public class MaxTemperatureDriver extends Configured implements Tool {
@Override
public int run(String[] args) throws Exception {
if (args.length != 2) {
System.err.printf("Usage: %s [generic options] <input> <output>\n",
getClass().getSimpleName());
ToolRunner.printGenericCommandUsage(System.err);
return -1;
}
Job job = new Job(getConf(), "Max temperature");
job.setJarByClass(getClass());
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.setMapperClass(MaxTemperatureMapper.class);
job.setCombinerClass(MaxTemperatureReducer.class);
job.setReducerClass(MaxTemperatureReducer.class);
job.setOutputKeyClass(Text.class); job.setOutputValueClass(IntWritable.class);
return job.waitForCompletion(true) ? 0 : 1; }
public static void main(String[] args) throws Exception {
int exitCode = ToolRunner.run(new MaxTemperatureDriver(), args);
System.exit(exitCode);
}
}
```

- MaxTemperatureDriver implements the Tool interface, to set the options that GenericOptionsParser supports.
- The run() method constructs Job object based on the tool's configuration, which it uses to launch a job.
- Run this application against some local files. Hadoop comes with a local job runner, running Map-Reduce jobs in a single JVM. It's designed for testing and is very convenient for use in an IDE
- The local job runner is only designed for simple testing of MapReduce programs it differs from the full MapReduce implementation.
- The biggest difference is that it can't run more than one reducer.

The local job runner is enabled by a configuration setting. Normally, mapred.job.tracker is a host:port pair to specify the address of the jobtracker, but when it has the special value of local, the job is run in-process without accessing an external jobtracker.

From the command line, we can run the driver by typing:

```
% hadoop v2.MaxTemperatureDriver -conf conf/hadoop-local.xml \
input/ncdc/micro output
```

Equivalently, we could use the -fs and -jt options provided by GenericOptionsParser:

```
% hadoop v2.MaxTemperatureDriver -fs file:/// -jt local input/ncdc/micro
output
```

This command executes MaxTemperatureDriver using input from the local *input/ncdc/micro* directory, producing output in the local *output* directory. To work local job runner against any filesystem set –fs so we use the local filesystem (file:///)

When we run the program, it fails and prints the following exception: **java.lang.NumberFormatException: For input string: "+0000"**

Fixing the mapper

This exception shows that the map method still can't parse positive temperatures. Run the test in a local debuggerEarlier, we made it handle the special case of missing temperature, +9999, but not the general case of any positive temperature and parser class to encapsulate the parsing logic.

Example : A class for parsing weather records in NCDC format

```
 public class NcdcRecordParser {
private static final int MISSING_TEMPERATURE = 9999;
private String year;
 private int airTemperature;
private String quality;
public void parse(String record) {
year = record.substring(15, 19);
String airTemperatureString;
// Remove leading plus sign as parseInt doesn't like them
if (record.charAt(87) == '+') { airTemperatureString = record.substring(88, 92); }
else {
airTemperatureString = record.substring(87, 92); Big Data 10 IV-II SEMESTER
}
airTemperature = Integer.parseInt(airTemperatureString);
quality = record.substring(92, 93); }
public void parse(Text record) {
parse(record.toString()); }
public boolean isValidTemperature() {
```

```
return airTemperature != MISSING_TEMPERATURE &&
quality.matches("[01459]"); }
public String getYear() {
return year; } public int getAirTemperature() {
return airTemperature;
}
}
```

- parse() method, which parses the fields of interest from a line of input, checks whether a valid temperature was found using the isValidTemperature() query method, and if it was, retrieves the year and the temperature using the getter methods on the parser.
- Check the quality status field as well as missing temperatures in isValidTemperature() to filter out poor temperature readings.
- Another benefit of creating a parser class is that it makes it easy to write related mappers for similar jobs without duplicating code. It also gives us the opportunity to write unit tests directly against the parser, for more targeted testing.

Example A Mapper that uses a utility class to parse records

```
public class MaxTemperatureMapper
extends Mapper<LongWritable, Text, Text, IntWritable> {
private NcdcRecordParser parser = new NcdcRecordParser(); @Override
public void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
parser.parse(value);
if (parser.isValidTemperature()) {
context.write(new Text(parser.getYear()),
new IntWritable(parser.getAirTemperature()));
}
}
 }
```

With these changes, the test passes

4.5 Running on a cluster Now that we are happy with the program running on a small test dataset, we are ready to try it on the full dataset on a Hadoop cluster.

Packaging

- ➢ The local job runner uses a single JVM to run a job, so all the classes that the job needs are on its classpath.
- ➢ In distributed environment job classes must be packaged into a job JAR file to send to the cluster.
- ➢ Hadoop will find the job JAR automatically by searching for the JAR on the drivers classpath that contains the class set in the setJarByClass() method. To set explicit JAR file by it path use setJar() method.

```
<jar destfile="hadoop-examples.jar" basedir="${classes.dir}"/>
```

If you have a single job per JAR, then you can specify the main class to run in the JAR file's manifest.

If the main class is not in the manifest, then it must be specified on the command line (as you will see shortly).

Also, any dependent JAR files should be packaged in a *lib* subdirectory in the JAR file.

Launching a Job

- ➢ To launch the job, we need to run the driver, specifying the cluster that we want to run the job on with the -conf option (we could equally have used the -fs and -jt options):

```
% hadoop jar hadoop-examples.jar v3.MaxTemperatureDriver -conf
  conf/hadoop-cluster.xml \
input/ncdc/all max-temp
```

The waitForCompletion() method on Job launches the job and polls for progress, writing a line summarizing the map and reduce's progress whenever either changes. Here's the output (some lines have been removed for clarity):

```
09/04/11 08:15:52 INFO mapred.FileInputFormat: Total input paths to
process : 101
09/04/11 08:15:53 INFO mapred.JobClient: Running job:
job_200904110811_0002
09/04/11 08:15:54 INFO mapred.JobClient: map 0% reduce 0%
```

```
09/04/11 08:16:06 INFO mapred.JobClient: map 28% reduce 0%
09/04/11 08:16:07 INFO mapred.JobClient: map 30% reduce 0%
....
09/04/11 08:21:36 INFO mapred.JobClient: map 100% reduce 100%
09/04/11 08:21:38 INFO mapred.JobClient: Job complete:
job_200904110811_0002 09/04/11 08:21:38 INFO mapred.JobClient:
Counters: 19 09/04/11 08:21:38 INFO mapred.JobClient: Job Counters
09/04/11 08:21:38 INFO mapred.JobClient: Launched reduce tasks=32
09/04/11 08:21:38 INFO mapred.JobClient: Rack-local map tasks=82
09/04/11 08:21:38 INFO mapred.JobClient: Launched map tasks=127
09/04/11 08:21:38 INFO mapred.JobClient: Data-local map tasks=45
09/04/11 08:21:38 INFO mapred.JobClient: FileSystemCounters 09/04/11
08:21:38 INFO mapred.JobClient: FILE_BYTES_READ=12667214 09/04/11
08:21:38 INFO mapred.JobClient: HDFS_BYTES_READ=33485841275
09/04/11 08:21:38 INFO mapred.JobClient: FILE_BYTES_WRITTEN=989397
09/04/11 08:21:38 INFO mapred.JobClient: HDFS_BYTES_WRITTEN=904
09/04/11 08:21:38 INFO mapred.JobClient: Map-Reduce Framework
09/04/11 08:21:38 INFO mapred.JobClient: Reduce input groups=100
09/04/11 08:21:38 INFO mapred.JobClient: Combine output records=4489
09/04/11 08:21:38 INFO mapred.JobClient: Map input records=1209901509
09/04/11 08:21:38 INFO mapred.JobClient: Reduce shuffle bytes=19140
09/04/11 08:21:38 INFO mapred.JobClient: Reduce output records=100
09/04/11 08:21:38 INFO mapred.JobClient: Spilled Records=9481 09/04/11
08:21:38 INFO mapred.JobClient: Map output bytes=10282306995 09/04/11
08:21:38 INFO mapred.JobClient: Map input bytes=274600205558 09/04/11
08:21:38 INFO mapred.JobClient: Combine input records=1142482941
09/04/11 08:21:38 INFO mapred.JobClient: Map output records=1142478555
09/04/11 08:21:38 INFO mapred.JobClient: Reduce input records=103
```

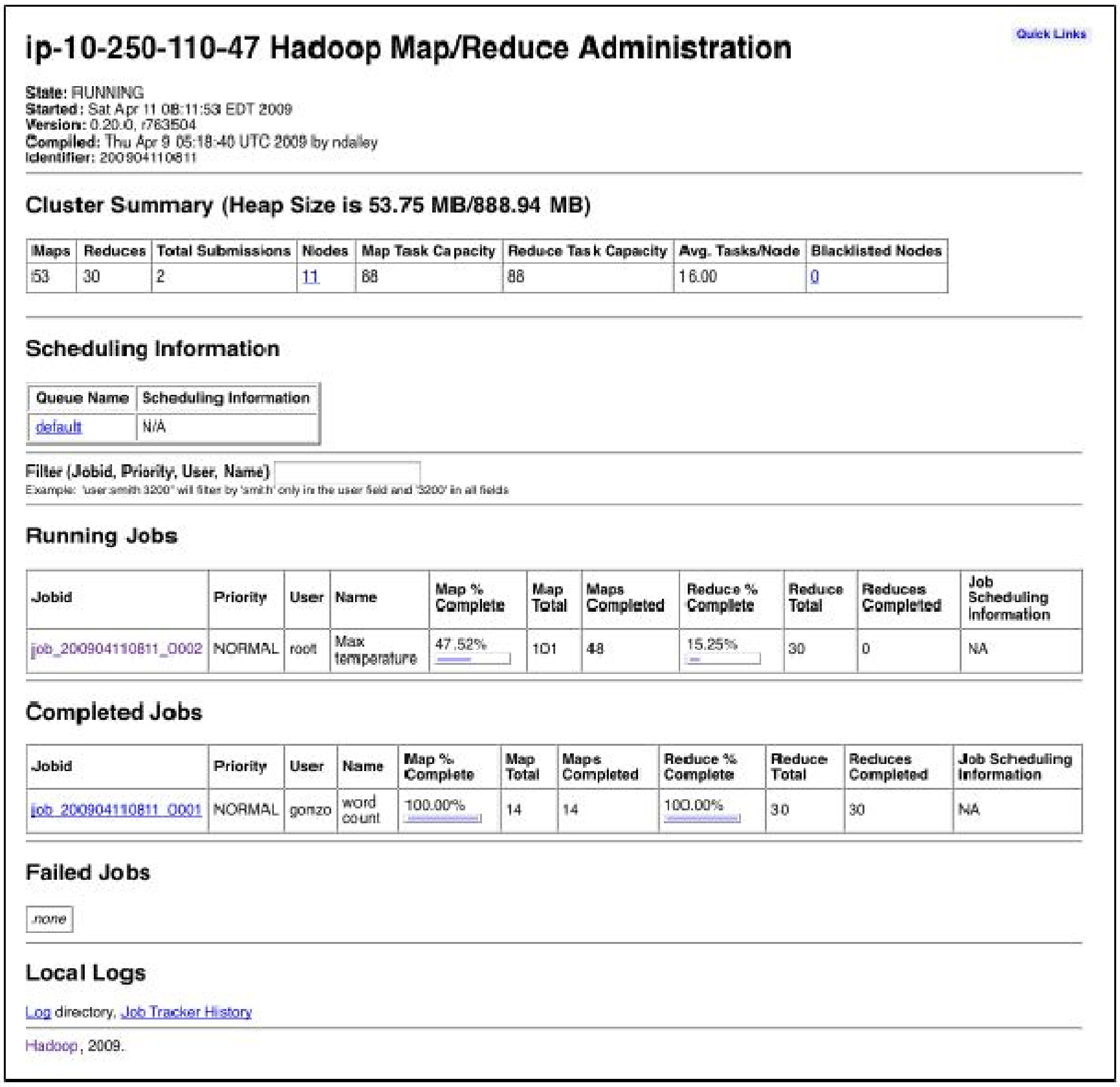
- The output includes more useful information.
- Before the job starts, its ID is printed: this is needed whenever you want to refer to the job, in logfiles for example, or when interrogating it via the hadoop job command.
- When the job is complete, its statistics (known as counters) are printed out. These are very useful for confirming that the job did what you expected. For example, for this job we can see that around 275 GB of input data was analyzed ("Map input bytes"), read from around 34 GB of compressed files on HDFS ("HDFS_BYTES_READ").
- The input was broken into 101 gzipped files of reasonable size, so there was no problem with not being able to split them.

The MapReduce Web UI

Hadoop comes with a web UI for viewing information about your jobs. It is useful for following a job's progress while it is running, as well as finding job statistics and logs after the job has completed. You can find the UI at *http://jobtracker-host:50030/*.

The jobtracker page

A screenshot of the home page is shown in Figure 5-1. The first section of the page gives details of the Hadoop installation, such as the version number and when it was compiled, and the current state of the jobtracker (in this case, running), and when it was started.



Quick Links

ip-10-250-110-47 Hadoop Map/Reduce Administration

State: RUNNING
Started: Sat Apr 11 08:11:53 EDT 2009
Version: 0.20.0, r763504
Compiled: Thu Apr 9 05:18:40 UTC 2009 by ndaley
Identifier: 200904110811

Cluster Summary (Heap Size is 53.75 MB/888.94 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
53	30	2	11	88	88	16.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0002	NORMAL	root	Max temperature	47.52%	101	48	15.25%	30	0	NA

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0001	NORMAL	gonzo	word count	100.00%	14	14	100.00%	30	30	NA

Failed Jobs

none

Local Logs

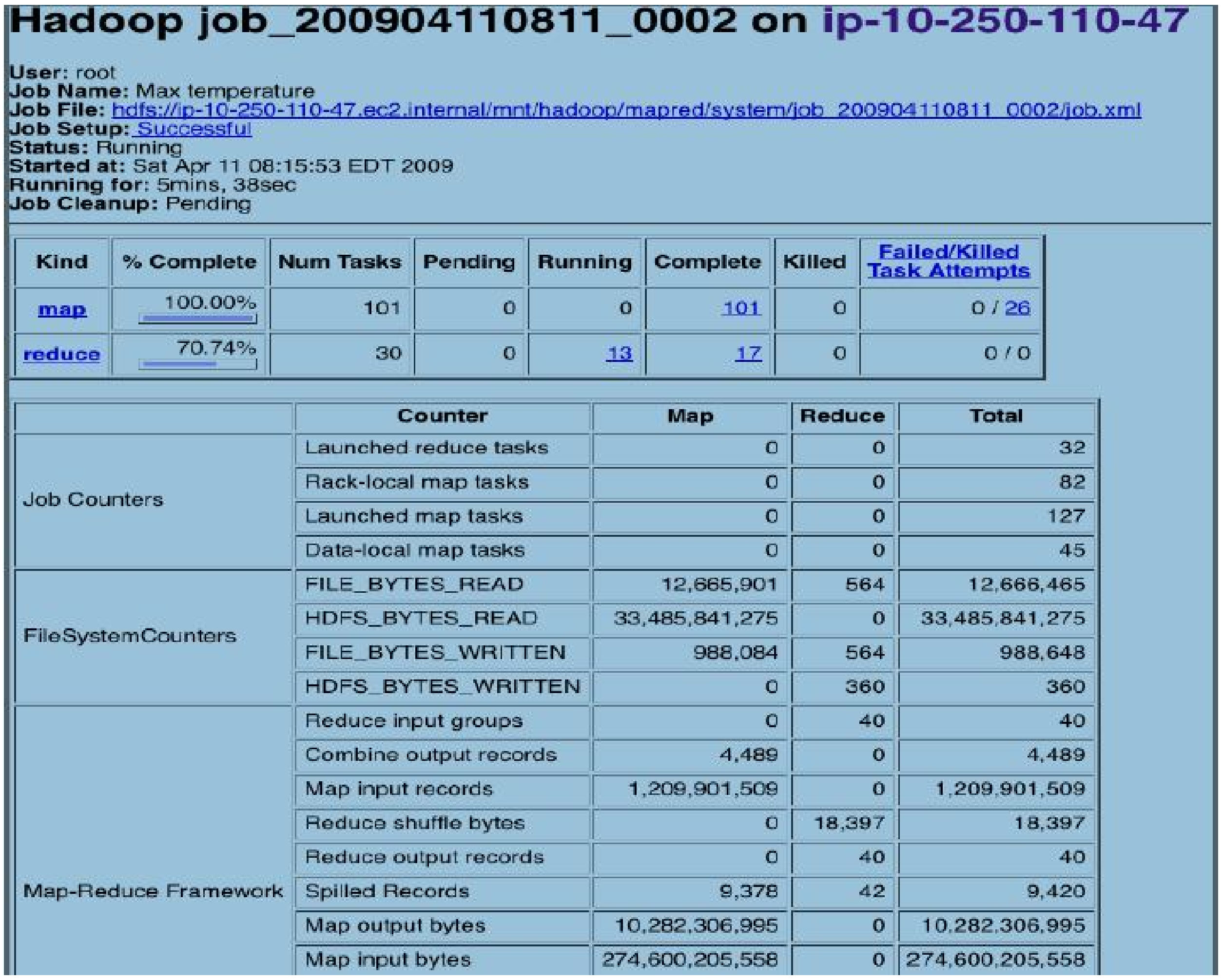
Log directory, Job Tracker History

Hadoop, 2009.

Next is the summary of the cluster, which has measures of cluster capacity and utilization. This show

- The number of maps and reduces currently running on the cluster.
- The total number of job submissions.
- The number of tasktracker nodes currently available.
- The cluster's capacity in terms of the number of map and reduce slots available across the cluster.
- The number of available slots per node, on average.
- The number of tasktrackers that have been blacklisted by the job tracker.

Below the summary, there is a section about the job scheduler that is running (here the default). You can click through to see job queues. Further down, we see sections for running, (successfully) completed, and failed jobs. Each of these sections has a table of jobs, with a row per job that shows the job's ID, owner, name (as set in the Job constructor or setJobName() method, both of which internally set the mapred.job.name property) and progress information. Finally, at the foot of the page, there are links to the jobtracker's logs, and the jobtracker's history: information on all the jobs that the jobtracker has run. The main view displays only 100 jobs (configurable via the mapred.jobtracker.completeuserjobs.maximum property), before consigning them to the history page.

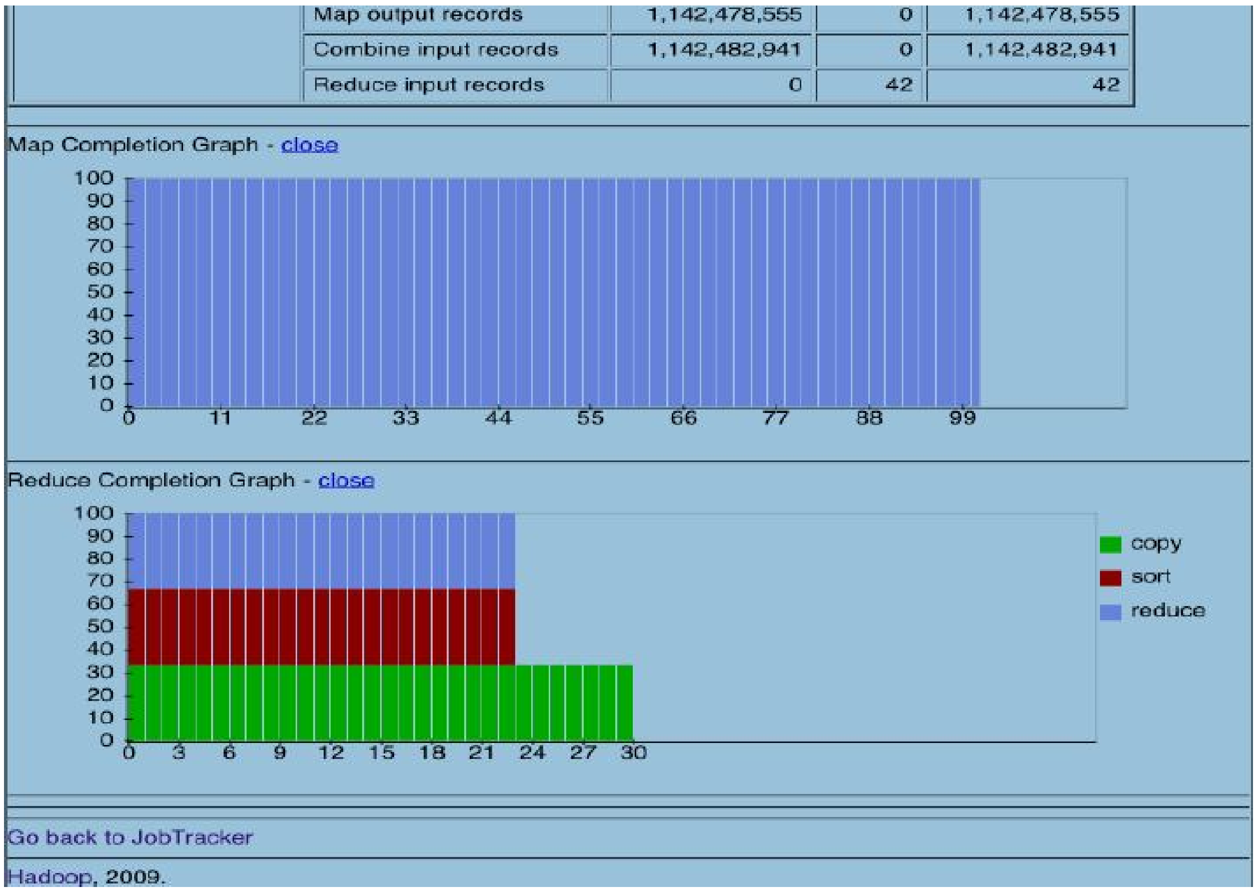


Hadoop job_200904110811_0002 on ip-10-250-110-47

User: root
Job Name: Max temperature
Job File: hdfs://ip-10-250-110-47.ec2.internal/mnt/hadoop/mapred/system/job_200904110811_0002/job.xml
Job Setup: Successful
Status: Running
Started at: Sat Apr 11 08:15:53 EDT 2009
Running for: 5mins, 38sec
Job Cleanup: Pending

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	101	0	0	101	0	0 / 26
reduce	70.74%	30	0	13	17	0	0 / 0

	Counter	Map	Reduce	Total
Job Counters	Launched reduce tasks	0	0	32
	Rack-local map tasks	0	0	82
	Launched map tasks	0	0	127
	Data-local map tasks	0	0	45
FileSystemCounters	FILE_BYTES_READ	12,665,901	564	12,666,465
	HDFS_BYTES_READ	33,485,841,275	0	33,485,841,275
	FILE_BYTES_WRITTEN	988,084	564	988,648
	HDFS_BYTES_WRITTEN	0	360	360
Map-Reduce Framework	Reduce input groups	0	40	40
	Combine output records	4,489	0	4,489
	Map input records	1,209,901,509	0	1,209,901,509
	Reduce shuffle bytes	0	18,397	18,397
	Reduce output records	0	40	40
	Spilled Records	9,378	42	9,420
	Map output bytes	10,282,306,995	0	10,282,306,995
	Map input bytes	274,600,205,558	0	274,600,205,558



Map output records	1,142,478,555	0	1,142,478,555
Combine input records	1,142,482,941	0	1,142,482,941
Reduce input records	0	42	42

Map Completion Graph - close

Reduce Completion Graph - close

Go back to JobTracker

Hadoop, 2009.

UNIT-IV
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. Which of the following is the default partitioner for Map Reduce [ ]

   A) Merge Partitioner C) Hash Partitioner

   B) Hashed Partitioner D) None

2. Which of the following partitions the key space/ [ ]

   A) Partitioner B) Compactor C) Collector D) All

3. ______ is a generalization of the facility provided by the Map Reduce frame work to collect data output by the Mapper or the Reducer. [ ]

   A) OutputCompactor C) InputCollector

   B) OutputCollector D) All

4. ______ is the primary interface for a user to describe a Map Reduce job to the Hadoop frame work for execution. [ ]

   A) Jobconfig B) Jobconf C) Jobconfiguration D) All

5. The __________________ executes the Mapper / Reducer task as a child process in a separate JVM. [ ]

   A) JobTracker B) TaskTracker C) TaskScheduler D) None

6. Maximum virtual memory of the launched child-task is specified using

   A) Mapv B) mapred C) mapvim D) All [ ]

7. Which of the following parameter is the threshold for the accounting and serialization butters? [ ]

   A) Io.sort.spill.percent C) io.sort.mb

   B) Io.sort.record.percent D) None

8. ____________ is percentage of memory relative to the maximum heap size in which map output may be retained during the reduce. [ ]

   A) Mapred.job.shuffle.merge.percent

   B) Mapred.job.reduce.input.buffer.percen

   C) Mapred.inmem.merge.threshold

   D) Io.sort.factor

9. ________ specifies the number of segments on disk to be merged at the same time. [ ]
   A) Mapred.job.shuffle.merge.percent
   B) Mapred.job.reduce.input.buffer.percen
   C) Mapred.inmem.merge.threshold.
   D) Io.sort.factor.
10. Map output larger that __ percent of the memory allocated to copying map outputs. [ ]
    A) 10 B) 15 C) 25 D) 35
11. Jobs can enable task JVM to be reused by specifying the job configuration. [ ]
    A) Mapred.job.recycle.jvm.num.tasks
    B) Mapissue.job.reuse.jvm.num.tasks.
    C) Mapred.job.reuse.jvm.num.tasks.
    D) All
12. During the execution of a streaming job, the names of the ________ parameters are transformed. [ ]
    A) Vmap B) mapvim C) mapreduce D) mapred
13. The standard output(stdout) and error (stderr) streams of the task are read by the Task Tracker and logged to
    A) ${HADOOP_LOG_DIR}/user
    B) ${HADOOP_LOG_DIR}/userlogs
    C) ${HADOOP_LOG_DIR}/logs
    D) None
14. _____ is the primary interface by which user-job interacts with the Job Tracker. [ ]
    A) Jobconf B) JobClient C) JobServer D) All
15. The ________ can also be used to distribute both jars and native libraries for use in the map and/or reduce tasks. [ ]
    A) DistributeLog C) DistributedJars
    B) Distributed Cache D) None

16. ___ is used to filter log files from the output directory listing. [ ]
    A) Outputlog B) OutputLogFilter C) DistributedLog D) DisttibutedJar
17. Which of the following class provides access to configuration parameters? [ ]
    A) Config B) configuration C) outputConfig D) None
18. ___________ gives site-specific configuration for a given hadoop installation. [ ]
    A) Core-default.xml C) coredefault.xml
    B) Core-site.xm; D) None
19. ___ method clears all keys from the configuration [ ]
    A) Clear B) addResource C) getClass D) None
20. ____ is useful for iterating the properties when all deprecated properties for currently set properties need to be present. [ ]
    A) addResource C) addDefaultResource
    B) setDeprecatedProperties D) None

SECTION-B

SUBJECTIVE QUESTIONS

1. How is the configuration of the development environment managed in Hadoop?
2. Write about the Managing configuration.
3. What you mean by MRUnit? Explain detail.
4. Explain the local job runner?
5. Write about the running on a cluster?
6. Explain about MapReduce WebUI?
7. Write the simple configuration file using XML
8. Explain about GenericOptionParser and Toolrunner options?
9. Write a program for a unit test MaxTemeratureMapper?
10. Write a test case for Reducer.
11. Write a program for local job runner?
12. Illustrate about packaging a job

13. Explain about launching a job?
14. Design the resource manage page?

UNIT-V

MapReduce Working

Objective:

To familiarize with the working of Map Reduce in Hadoop.

Syllabus:

MapReduce Working

Classic MapReduce, Job submission, Job Initialization, Task Assignment, Task execution, Progress and status

updates, Job completion, Shuffle and sort on Map and Reduce side, Configuration tuning, Map Reduce types,

Input formats, Output formats.

Learning Outcomes:

At the end of the unit, students will be able to:

1. Write more advanced Map Reduce programs.
2. Describe data types supported by MapReduce and Input and Output formats.

Learning Material

Introduction

- Run a MapReduce job with a single method call: submit() on a Job object which will submit the job and call waitForCompletion(), wait for it to finish.
- The steps Hadoop takes to run a job. We saw in previous chapter that the way Hadoop executes a MapReduce program depends on a couple of configuration settings.
- In releases of Hadoop up to and including the 0.20 release series, mapred.job.tracker determines the means of execution.

1) If this configuration property is set to local, the default, then the local job runner is used. This runner runs the whole job in a single JVM.

2)It's designed for testing and for running MapReduce programs on small datasets.

3) If mapred.job.tracker is set to a colon-separated host and port pair, then the property is interpreted as a jobtracker address, and the runner submits the job to the jobtracker at that address.

5.1 Classic MapReduce (MapReduce 1)

A job run in classic MapReduce is illustrated in Figure .

At the highest level, there are four independent entities:

1. The client, which submits the MapReduce job.

2. The jobtracker, which coordinates the job run. The jobtracker is a Java application whose main class is JobTracker.

3. The tasktrackers, which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is TaskTracker.

4. The distributed filesystem (normally HDFS), which is used for sharing job files between the other entities.

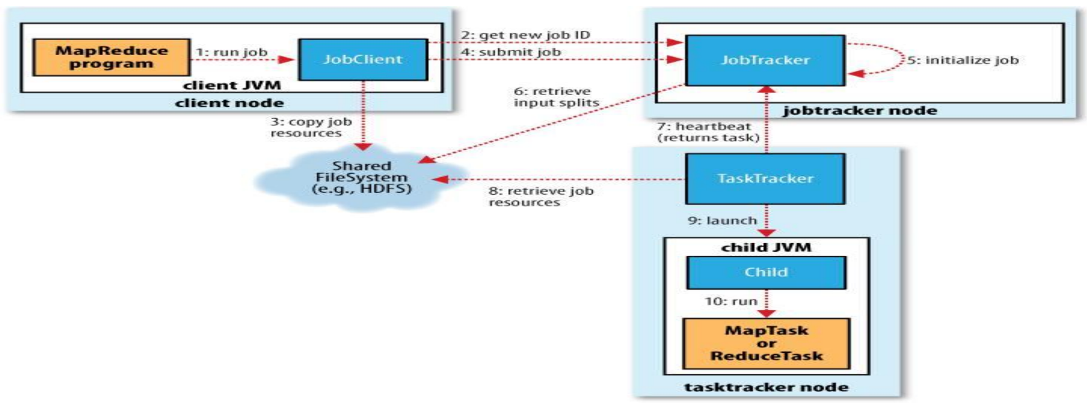


Fig : How Hadoop runs a MapReduce job using the classic framework

There are six detailed levels in workflows. They are:

1. Job Submission
2. Job Initialization
3. Task Assignment
4. Task Execution
5. Task Progress and status updates
6. Task Completion

5.2 Job Submission

The submit() method on Job creates an internal JobSummitter instance and calls submitJobInternal() on it (step 1 in Figure).

After submitted the job, waitForCompletion() polls the job's progress once a second and reports the progress to the console.

When the job is complete, if it was successful, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented by JobSummitter does the following:

- Asks the jobtracker for a new job ID (by calling getNewJobId() on obTracker) (step 2).

- Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.

- Computes the input splits for the job. If the splits cannot be computed, because the input paths don't exist, for example, then the job is not submitted and an error is thrown to the MapReduce program.

- Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the jobtracker's filesystem in a directory named after the job ID.

- The job JAR is copied with a high replication factor (controlled by the mapred.submit.replication property, which defaults to 10) so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job (step 3).

- Tells the jobtracker that the job is ready for execution (by calling submitJob() on JobTracker) (step 4).

5.3 Job Initialization

- When the JobTracker receives a call to its submitJob() method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it

Initialization involves

Bookkeeping information to keep track of the tasks' status and progress (step 5).

creating an object to represent the job being run, which encapsulates its tasks, and

1. To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the client from the shared filesystem (step 6).

2. It then creates one map task for each split.

3. The number of reduce tasks to create is determined by the mapred.reduce.tasks property in the Job, which is set by the setNumReduceTasks() method, and the scheduler simply creates this number of reduce tasks to be run.

4. Tasks are given IDs at this point. In addition to the map and reduce tasks, two further tasks are created: a job setup task and a job cleanup task.

5. These are run by tasktrackers and are used to run code to setup the job before any map tasks run, and to clean up after all the reduce tasks are complete.

6. The OutputCommitter that is configured for the job determines the code to be run, andby default this is a FileOutputCommitter.

7. For the job setup task it will create the final output directory for the job and the temporary working space for the task output, and for the job cleanup task it will delete the temporary working space for the task output.

5.4 Task Assignment

- Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker.

- Heartbeats tell the jobtracker that a tasktracker is alive, but they also double as a channel for messages.

- As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the jobtracker will allocate it a task, which it communicates to the tasktracker using the heartbeat return value (step 7).

- Before it can choose a task for the tasktracker, the jobtracker must choose a job to select the task from. There are various scheduling algorithms but the default one simply maintains a priority list of jobs.

- Having chosen a job, the jobtracker now chooses a task for the job. Tasktrackers have a fixed number of slots for map tasks and for reduce tasks:

  Ex: a tasktracker may be able to run two map tasks and two reduce tasks simultaneously. (The precise number depends on the number of cores and the amount of memory on the tasktracker)

- The default scheduler fills empty map task slots before reduce task slots, so if the tasktracker has at least on empty map task slot, the jobtracker will select a map task; otherwise, it will select a reduce task.

- To choose a reduce task, the jobtracker simply takes the next in its list of yet-to-be-run reduce tasks, since there are no data locality considerations.

- For a map task, however, it takes account of the tasktracker's network location and picks a task whose input split is as close as possible to the tasktracker.

- In the optimal case, the task is *data-local, that* is, running on the same node that the split resides on.

- Alternatively, the task may be *rack-local: on the same rack, but not the same node, as the split. Some tasks are neither* data-local nor rack-local and retrieve their data from a different rack from the one they are running on.

- You can tell the proportion of each type of task by looking at a job's counters.

5.5 Task Execution

- Now that the tasktracker has been assigned a task, the next step is for it to run the task.

- First, it localizes the job JAR by copying it from the shared filesystem to the tasktracker's filesystem.

  It also copies any files needed from the distributed cache by the application to the local disk.

- Second, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory.

- Third, it creates an instance of TaskRunner to run the task. TaskRunner launches a new Java Virtual Machine (step 9) to run each task in (step 10), so that any bugs in the user-defined map and reduce functions don't affect the tasktracker (by causing it to crash or hang, for example).

- It is, however, possible to reuse the JVM between tasks.

- The child process communicates with its parent through the *umbilical interface. This* way it informs the parent of the task's progress every few seconds until the task is complete.

- Each task can perform setup and cleanup actions, which are run in the same JVM as the task itself, and are determined by the OutputCommitter for the job

- The cleanup action is used to commit the task, which in the case of file-based jobs means that its output is written to the final location for that task.

- The commit protocol ensures that when speculative execution is enabled, only one of the duplicate tasks is committed and the other is aborted.

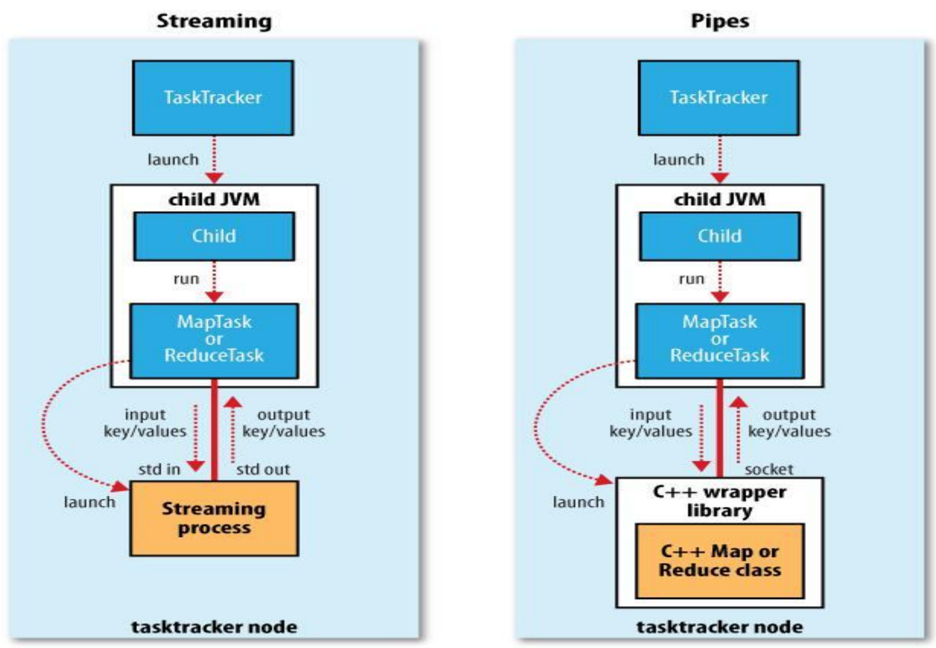


Fig : The relationship of the Streaming and Pipes executable to the tasktracker and its child

- Both Streaming and Pipes run special map and reduce tasks for the purpose of launching the user-supplied executable and communicating with it (Fig).

- In the case of Streaming, the Streaming task communicates with the process (which may be written in any language) using standard input and output streams.

- The Pipes task, on the other hand, listens on a socket and passes the C++ process a port number in its environment, so that on startup, the C++ process can establish a persistent socket connection back to the parent Java Pipes task.

- In both cases, during execution of the task, the Java process passes input key-value pairs to the external process, which runs it through the user-defined map or reduce function and passes the output key-value pairs back to the Java process.

- From the tasktracker's point of view, it is as if the tasktracker child process ran the map or reduce code itself.

5.6 Progress and Status Updates

- MapReduce jobs are long-running batch jobs, taking anything from minutes to hours to run.
- Progress reporting is important for the user to get feedback on how the job is progressing.
- The following operations constitute **progress**:
  - Reading an input record (in a mapper and reducer)
  - Writing and output record (in a mapper and reducer)
  - Setting the status description on a reporter (by using Reporter's setStatus() method)
  - Incrementing a counter (using Reporter's incrCounter() method)
  - Calling Reporter's Progress() method
- A job and each of its tasks have a **status**, which includes
  - State of the job or task (e.g., running, successfully completed, failed)
  - The progress of maps and reduces
  - The values of the job's counters and
  - A status message or description (which may be set by user code).
- These statuses change over the course of the job, they get communicated back to the client regarding progress by displaying

  - The proportion of the task completed.
  - For map tasks, the proportion of the input that has been processed.
  - For reduce tasks, the proportion of the reduce input processed.
    - It does this by dividing the total progress into three parts, corresponding to the three phases of the shuffle.
    - Display counters that count various events as the task runs such as number of map output records written.

- If a task reports progress, it sets a flag to indicate that the status change should be sent to the tasktracker.
- The flag is checked in a separate thread every three seconds, and if set it notifies the tasktracker of the current task status.
- Meanwhile, the tasktracker is sending heartbeats to the jobtracker every five seconds (this is a minimum, as the heartbeat interval is actually dependent on the size of the cluster: for larger clusters,the interval is longer)
- The status of all the tasks being run by the tasktracker is sent in the call.
- Counters are sent less frequently than every five seconds, because they can be relatively high-bandwidth.
- The jobtracker combines these updates to produce a global view of the status of all the jobs being run and their constituent tasks.
- Finally, the Job receives the latest status by polling the jobtracker every second.

- Clients can also use Job’s getStatus() method to obtain a JobStatus instance, which contains all of the status information for the job.

5.7 Job Completion

- When the jobtracker receives a notification that the last task for a job is complete it changes the status for the job to “successful.”

- Then, when the Job polls for status, it learns that the job has completed successfully, it prints a message to tell the user and then returns from the waitForCompletion() method.

- The jobtracker also sends an HTTP job notification.

- Last, the jobtracker cleans up its working state for the job and instructs tasktrackers to do the same (so intermediate output is deleted, for example).

5.8 Shuffle and Sort

- MapReduce makes the guarantee that the input to every reducer is sorted by key.

- The process by which the system performs the sort—and transfers the map outputs to the reducers as inputs—is known as the shuffle.

- The shuffle is an area of the codebase where refinements and improvements are continually being made.

Shuffle sort on Map Side

- When the map function starts producing output, it is not simply written to disk. The process is more involved, and takes advantage of buffering writes in memory and doing some presorting for efficiency reasons.

Shuffle and sort in MapReduce

- The buffer is 100 MB by default, change the size by using io.sort.mb property.
- When the contents of the buffer reaches a certain threshold size a background thread will start to spill the contents to disk.
- Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete.
- Spills are written in round-robin fashion to the directories specified by the mapred.local.dir property.
- Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers to send.
- Within each partition, the background thread performs an in-memory sort by key, and if there is a combiner function, it is run on the output of the sort.
- Running the combiner function makes more compact map output, so less data to write to local disk and to transfer to the reducer.
- Each time the memory buffer reaches the spill threshold, a new spill file is created
- Before the task is finished, the spill files are merged into a single partitioned and sorted output file.
- The configuration property io.sort.factor controls the maximum number of streams to merge at once; the default is 10.

- If there are at least three spill files then the combiner is run again before the output file is written.

- Compress the map output as it is written to disk, makes it faster to write to disk, saves disk space, and reduces the amount of data to transfer to the reducer.

- By default, the output is not compressed, but it is easy to enable by setting mapred.compress.map.output to true.

- The output file's partitions are made available to the reducers over HTTP.

- The maximum number of worker threads used to serve the file partitions is controlled by the tasktracker.http.threads property. The default of 40 may need increasing for large clusters running large jobs.

Shuffle and sort on Reduce Side

The map output file is sitting on the local disk of the machine that ran the map task. The reduce task needs the map output for its particular partition from several map tasks across the cluster.

There are three phases for reducer 1) copy phase 2) sort phase 3) reduce phase.

1)Copy phase:

- The map tasks may finish at different times, so the reduce task starts copying their outputs as soon as each completes.

- The reduce task has a small number of copier threads so that it can fetch map outputs in parallel.

- The default is five threads, but this number can be changed by setting the mapred.reduce.parallel.copies property.

- The map outputs are copied to reduce task JVM’s memory otherwise, they are copied to disk.

  When the in-memory buffer reaches a threshold size or reaches a threshold number of map outputs it is merged and spilled to disk.

- Any map outputs that were compressed have to be decompressed in memory in order to perform a merge on them.

- When all the map outputs have been copied, the reduce task moves into the sort phase.

2)Sort phase:

- In this phase merge the map outputs, maintaining their sort ordering.

- This is done in rounds. For example, if there were 50 map outputs, and the merge factor was 10, then there would be 5 rounds. Each round would merge 10 files into one, so at the end there would be five intermediate files.

- These five files into a single sorted file, the merge saves a trip to disk by directly feeding the reduce function. This final merge can come from a mixture of in-memory and on-disk segments.

3)Reduce phase:

- During the reduce phase, the reduce function is invoked for each key in the sorted output.

- The output of this phase is written directly to the output filesystem, typically HDFS.

- In the case of HDFS, since the tasktracker node is also running a datanode, the first block replica will be written to the local disk.

5.9 Configuration Tuning

- Configuration tuning is to tune the shuffle to improve MapReduce performance.
- The general principle is to give the shuffle as much memory as possible.
- There is a trade-off, in that you need to make sure that your map and reduce functions get enough memory to operate.
- Write map and reduce functions to use as little memory as possible, should not use an unbounded amount of memory.
- The amount of memory given to the JVMs in which the map and reduce tasks run is set by the mapred.child.java.opts property.
- To make this as large as possible for the amount of memory on your task nodes.

On the map side

- The best performance can be obtained by avoiding multiple spills to disk; one is optimal.
- If you can estimate the size of your map outputs, then you can set the io.sort.* properties appropriately to minimize the number of spills.
- There is a MapReduce counter that counts the total number of records that were spilled to disk over the course of a job, which can be useful for tuning.
- The counter includes both map and reduces side spills.

On the reduce side

- The best performance is obtained when the intermediate data can reside entirely in memory.
- By default, this does not happen, since for the general case all the memory is reserved for the reduce function.

- If your reduce function has mapred.inmem.merge.threshold to 0 and a performance boost.

light memory requirements, then setting mapred.job.reduce.input.buffer.percent to 1.0 may bring

- Hadoop uses a buffer size of 4 KB by default, which is low, so you should increase this across the cluster.

5.10 MapReduce Types

The map and reduce functions in Hadoop MapReduce have the following **general form:**

map: (K1, V1) → list(K2, V2)

reduce: (K2, list(V2)) → list(K3, V3)

- The map input key and value types (K1 and V1) are different from the map output types (K2 and V2).

- The reduce input must have the same types as the map output, although the reduce output types may be different again (K3 and V3).

The **Java API** mirrors this **general form:**

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

public class Context extends MapContext<KEYIN, VALUEIN,
        KEYOUT, VALUEOUT> { / / ...

}

protected void map(KEYIN key, VALUEIN value, Context
        context) throws IOException, InterruptedException {

        / / ...
}

}

public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

public class Context extends ReducerContext<KEYIN,
        VALUEIN, KEYOUT, VALUEOUT> { / / ...

}

protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context
context) throws

     IOException, InterruptedException {

        / / ...
}
}
```

- The context objects are used for emitting key-value pairs, so they are parameterized by the output types, so that the signature of the write() method is:

public void write(KEYOUT key, VALUEOUT value) throws IOException, InterruptedException

- Mapper and Reducer are separate classes the type parameters have different scopes, and the actual type argument of KEYIN (say) in the Mapper may be different to the type of the type parameter of the same name (KEYIN) in the Reducer.

  Ex: in the maximum temparature example from earlier chapters, KEYIN is replaced by LongWritable for the Mapper, and by Text for the Reducer.

- The map output types and the reduce input types must match

If combiner function is used, then it is the same form as the reduce function (and is an implementation of Reducer), except its output types are the intermediate key and value types (K2 and V2), so they can feed the reduce function:

map: (K1, V1) → list(K2, V2)

combine: (K2, list(V2)) → list(K2, V2)

reduce: (K2, list(V2)) → list(K3, V3)

- combine and reduce functions are the same, in which case, K3 is the same as K2, and V3 is the same as V2.

- The partition function operates on the intermediate key and value types (K2 and V2), and returns the partition index.

- The partition is determined by the key (the value is ignored):

partition: (K2, V2) → integer

In Java:

```
public abstract class Partitioner<KEY, VALUE> {
public abstract int getPartition(KEY key, VALUE value, int
numPartitions);

}
```

5.11 Input Formats

Hadoop can process many different types of data formats, from flat text files to databases.

1) Input Splits and Records:

- An input split is a chunk of the input that is processed by a single map. Each map processes a single split.

- Each split is divided into records, and the map processes each record—a key-value pair—in turn.

public abstract class InputSplit {

public abstract long getLength() throws IOException, InterruptedException; public abstract String[] getLocations() throws IOException, InterruptedException;
}

FileInputFormat:

- FileInputFormat is the **base class** for all implementations of InputFormat that use files as their data source.

- It provides two things: a place to define which files are included as the input to a job, and an implementation for generating splits for the input files.

FileInputFormat input paths:

- The input to a job is specified as a collection of paths, which offers great flexibility in constraining the input to a job.

- FileInputFormat offers four static convenience methods for setting a Job's input paths:

public static void addInputPath(Job job, Path path)

public static void addInputPaths(Job job, String commaSeparatedPaths)

public static void setInputPaths(Job job, Path... inputPaths)

public static void setInputPaths(Job job, String commaSeparatedPaths)

- The addInputPath() and addInputPaths() methods add a path or paths to the list of inputs.

- The setInputPaths() methods set the entire list of paths in one go

- To exclude certain files from the input, you can set a filter using the setInputPathFilter() method

public static void setInputPathFilter(Job job, Class<? extends PathFilter> filter)

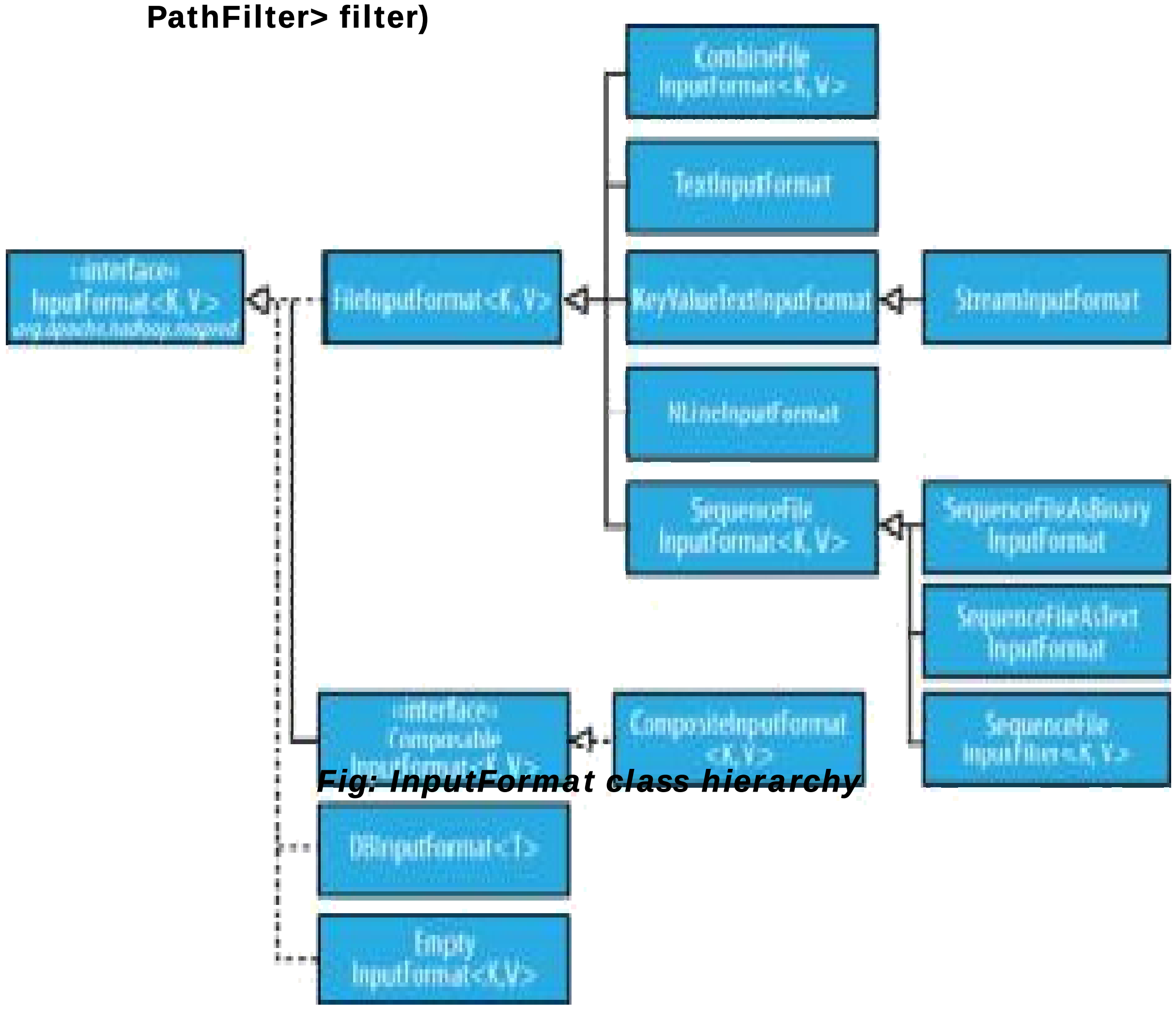
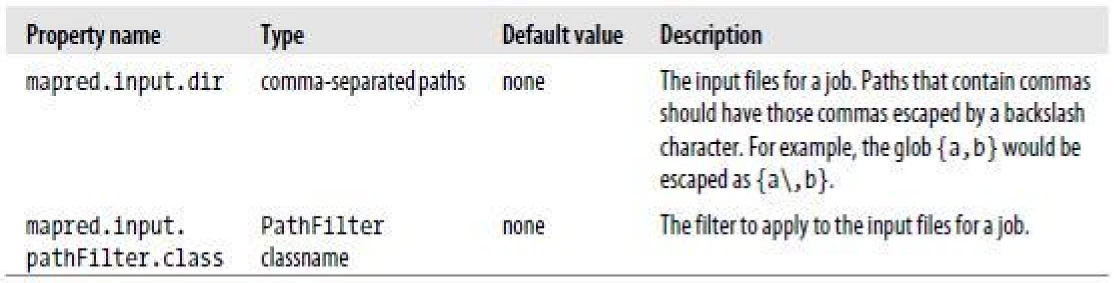

«interface» InputFormat<K, V>
org.apache.hadoop.mapred
FileInputFormat<K,V>
CombineFile InputFormat<K,V>
TextInputFormat
KeyValueTextInputFormat
StreamInputFormat
NLineInputFormat
SequenceFile InputFormat<K,V>
SequenceFileAsBinary InputFormat
SequenceFileAsText InputFormat
SequenceFile InputFilter<K,V>
«interface» Composable InputFormat<K,V>
CompositeInputFormat <K,V>
DBInputFormat<T>
Empty InputFormat<K,V>

Fig: InputFormat class hierarchy

Paths and filters can be set through configuration properties

Table: ***Input path and filter properties***

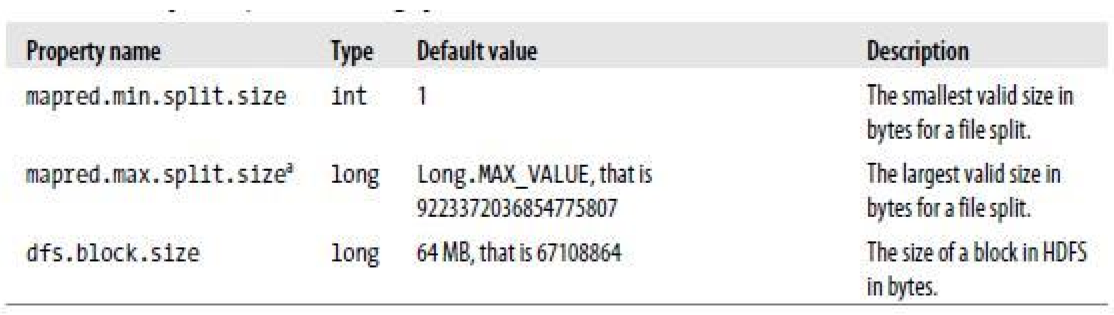


Property name	Type	Default value	Description
`mapred.input.dir`	comma-separated paths	none	The input files for a job. Paths that contain commas should have those commas escaped by a backslash character. For example, the glob `{a,b}` would be escaped as `{a\,b}`.
`mapred.input.pathFilter.class`	`PathFilter` classname	none	The filter to apply to the input files for a job.

FileInputFormat input splits:

- FileInputFormat splits only large files. Here “large” means larger than an HDFS block. The split size is normally the size of an HDFS block.

Table: ***Properties for controlling split size***



Property name	Type	Default value	Description
`mapred.min.split.size`	`int`	1	The smallest valid size in bytes for a file split.
`mapred.max.split.size`[a]	`long`	`Long.MAX_VALUE`, that is 9223372036854775807	The largest valid size in bytes for a file split.
`dfs.block.size`	`long`	64 MB, that is 67108864	The size of a block in HDFS in bytes.

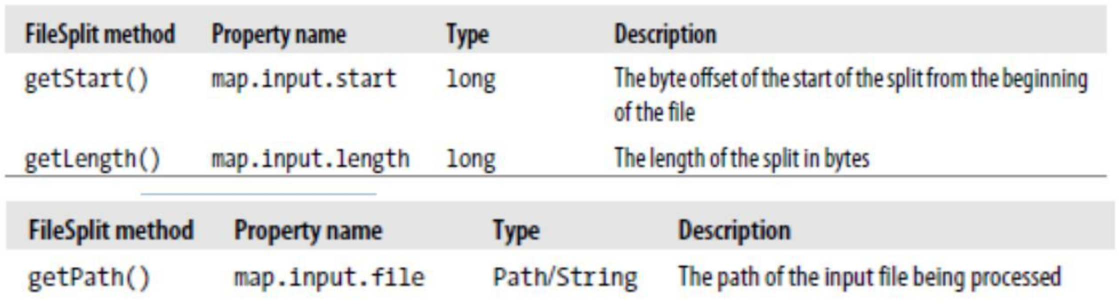
Preventing splitting:

- There are a couple of ways to ensure that an existing file is not split.
  First way is to increase the minimum split size to be larger than the largest file in your system. **Second way** is to subclass the concrete subclass of FileInputFormat that you want to use, to override the isSplitable() method to return false.

File information in the mapper:

- A mapper processing a file input split can find information about the split by calling the getInputSplit() method on the Mapper's Context object.

Table: *File split properties*



FileSplit method	Property name	Type	Description
getStart()	map.input.start	long	The byte offset of the start of the split from the beginning of the file
getLength()	map.input.length	long	The length of the split in bytes
getPath()	map.input.file	Path/String	The path of the input file being processed

Processing a whole file as a record:

- A related requirement that sometimes crops up is for mappers to have access to the full contents of a file. The listing for WholeFileInputFormat shows a way of doing this.

  Ex : An InputFormat for reading a whole file as a record

```
public class WholeFileInputFormat extends
FileInputFormat<NullWritable, BytesWritable> { @Override
protected boolean isSplitable(JobContext
context, Path file) { return false;
}
 }
```

- WholeFileRecordReader is responsible for taking a FileSplit and converting it into a single record, with a null key and a value containing the bytes of the file.

2) Text Input

- Hadoop can process unstructured text. It provide different InputFormat to process text.

TextInputFormat:

- TextInputFormat is the default InputFormat. Each record is a line of input.

- The key, a LongWritable, is the byte offset within the file of the beginning of the line.

- The value is the contents of the line, excluding any line terminators (newline, carriage return), and is packaged as a Text object.

- A file containing the following text:

  On the top of the Crumpetty Tree

  The Quangle Wangle sat,

  But his face you could not see,

  On account of his Beaver Hat.

is divided into one split of four records. The records are interpreted as the following key-value pairs:

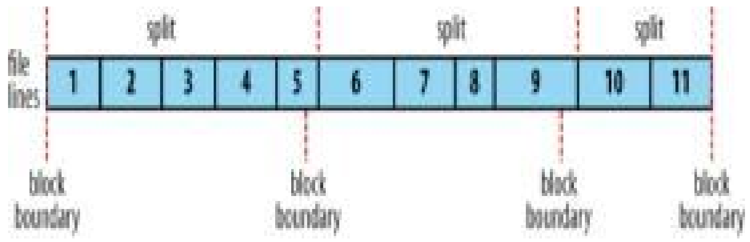
(0, On the top of the Crumpetty Tree)

(33, The Quangle Wangle sat,)

(57, But his face you could not see,)

(89, On account of his Beaver Hat.)

Fig: *Logical records and HDFS blocks for TextInputFormat*



KeyValueTextInputFormat:

- TextInputFormat's keys, being simply the offset within the file, are not normally very useful.

- It is common for each line in a file to be a key-value pair, separated by a delimiter such as a tab character by default.

- Specify the separator via the mapreduce.input.keyvaluelinerecordreader.key.value.separator property.

- Consider the following input file, where → represents a (horizontal) tab character:

  line1→On the top of the Crumpetty Tree

  line2→The Quangle Wangle sat,

  line3→But his face you could not see,

  line4→On account of his Beaver Hat.

- Like in the TextInputFormat case, the input is in a single split comprising four records, although this time the keys are the Text sequences before the tab in each line:
  (line1, On the top of the Crumpetty Tree) (line2, The Quangle Wangle sat,)
  (line3, But his face you could not see,) (line4, On account of his Beaver Hat.)

NLineInputFormat:

- N refers to the number of lines of input that each mapper receives.

- With N set to one, each mapper receives exactly one line of input. mapreduce.input.lineinputformat.linespermap property controls the value of N.

  Ex: N is two, then each split contains two lines. One mapper will receive the first two key-value pairs:

  (0, On the top of the Crumpetty Tree) (33, The Quangle Wangle sat,)

  And another mapper will receive the second two key-value pairs:

  (57, But his face you could not see,) (89, On account of his Beaver Hat.)

3)Binary Input:

- Hadoop MapReduce is not just restricted to processing textual data—it has support for binary formats, too.

- **SequenceFileInputFormat:** Hadoop's sequence file format stores sequences of binary key-value pairs.

- **SequenceFileAsTextInputFormat**: SequenceFileAsTextInputFormat is a variant of

SequenceFileInputFormat that converts the sequence file's keys and values to Text objects.

- **SequenceFileAsBinaryInputFormat**: SequenceFileAsBinaryInputFormat is a variant of

  SequenceFileInputFormat that retrieves the sequence file's keys and values as opaque binary objects.

4) Multiple Inputs:

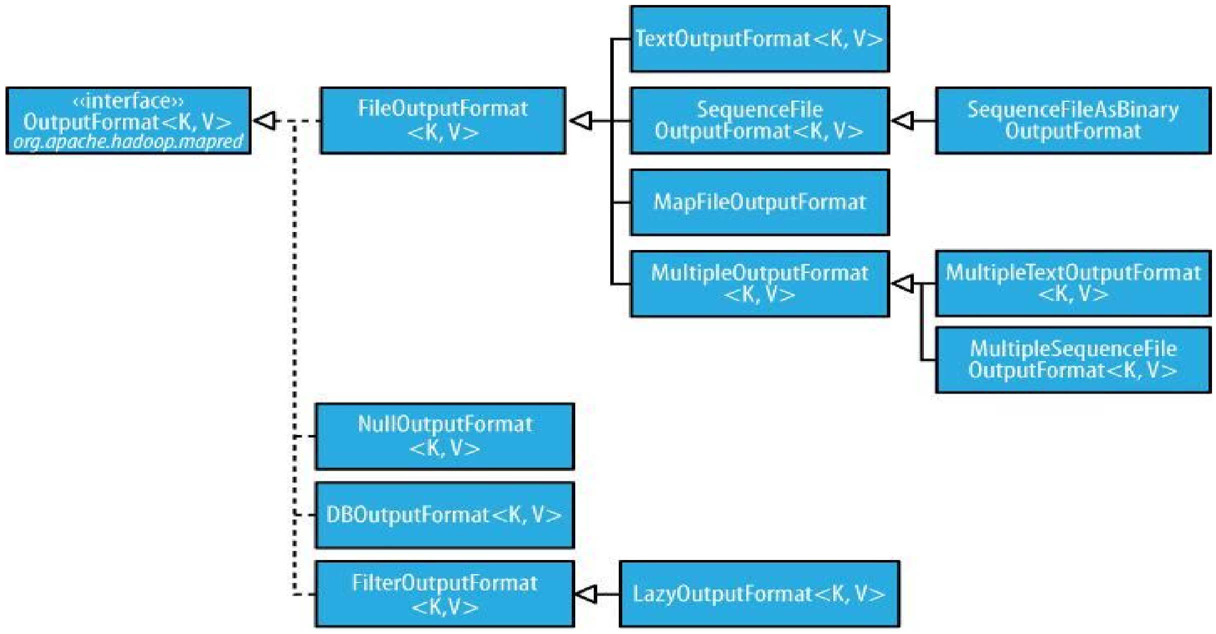
- Although the input to a MapReduce job may consist of multiple input files, all of the input is interpreted by a single InputFormat and a single Mapper.

- The MultipleInputs class has an overloaded version of addInputPath() that doesn't take a mapper: **public static void addInputPath(Job job, Path path, Class<? extends InputFormat> inputFormatClass**)

5.12 Output Formats

- Hadoop has output data formats that correspond to the input formats

Figure: OutputFormat class hierarchy



1) Text Output:

- The default output format, TextOutputFormat, writes records as lines of text.

- Its keys and values may be of any type, since TextOutputFormat turns them to strings by calling toString() on them.

- Each key-value pair is separated by a tab character, that may be changed using the mapreduce.output.textoutputformat.separator property.

2) Binary Output

- **SequenceFileOutputFormat:** As the name indicates, SequenceFileOutputFormat writes sequence files for its output. Compression is controlled via the static methods on SequenceFileOutputFormat.

- **SequenceFileAsBinaryOutputFormat**: SequenceFileAsBinaryOutputFormat is the counterpart to SequenceFileAsBinaryInput Format, and it writes keys and values in raw binary format into a SequenceFile container.

- **MapFileOutputFormat**: MapFileOutputFormat writes MapFiles as output. The keys in a MapFile must be added in order, so you need to ensure that your reducers emit keys in sorted order.

3) Multiple Outputs:

- FileOutputFormat and its subclasses generate a set of files in the output directory.

- There is one file per reducer, and files are named by the partition number: part-r-00000, partr-00001, etc.

- MapReduce comes with the MultipleOutputs class to do this.

  Zero reducers: There are no partitions, as the application needs to run only map tasks.

  One reducer: It can be convenient to run small jobs to combine the output of previous jobs into a single file when the amount of data is small enough to be processed.

MultipleOutputs:

- MultipleOutputs allows you to write data to files whose names are derived from the output keys and values.

- This allows each reducer (or mapper in a map-only job) to create **more than** a single file.

- File names are of the form **name-m-nnnnn** for map outputs and **name-r-nnnnn** for reduce outputs

- **name** is an arbitrary name that is set by the program, and **nnnnn** is an integer designating the part number, starting from zero.

- The part number ensures that outputs written from different partitions

Lazy Output:

- FileOutputFormat subclasses will create output (part-r-nnnnn) files, even if they are empty.

- Some applications prefer that empty files not be created, which is where LazyOutputFormat helps.
- It is a wrapper output format that ensures that the output file is Output Formats created only when the first record is emitted for a given partition.

Database Output:

- The output formats for writing to relational databases and to HBase.

UNIT-V
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. Number of mappers is decided by the___________. [ ]
   A) Mapper specified by the programmer C) Input Splits
   B) Available Mapper slots D) Input Format.
2. Map Reduce job can be written in________. [ ]
   A) Java C) Pyton
   B) Ruby D) Any Language which can read from input stream
3. YARN also called as_______________. [ ]
   A)MapReduce1 B) MapReduce2 C) MapReduce3 D) None
4. Expansion of YARN___ Yet Another Resource Navigator [T/F]
5. Classic MapReduce frame work also called as_____ [ ]
   A)MapReduce1 B) MapReduce2 C) MapReduce3 D) None
6. Input to every reducer is sorted by_______ [ ]
   A) Value B) key C) Both key-value pair D) key or value
7. The process of that performs sort and transfers the map outputs to the reducers as input is ___________. [ ]
   A) sort B) copy C) shuffle D) transfer
8. The default buffer size_______. [ ]
   A) 100MB B)64MB C)512MB D)128MB
9. Processing a whole file as a record by using isSplitable method that return false. [T/F]
10. Number of copier threads can be changed by setting_____property.
11. The amount of memory given to the JVM in which the map and reduce tasks run is set by the __________ property. [ ]
   A) mapred.child.java.opts C) mapred.child.java.mem
   B) mapred.child.java.jvm D) None
12. Spills are written in __________ fashion. [ ]
   A) Sequence B) Round-Robin C) FCFS D) None

13. The default input format is [ ]

A) binary input format B) file input format

C) Text input format D) None

14. ____ is the base class for all implementations of inputFormat tha use files as their data source. [ ]

A) BinaryInputFormat C) TextInputFormat

B) FileInputFormat D) None

15. Which static convenience method used for setting a job's input paths.

A) addInputPaths() C) setInputPaths() [ ]

B) addInputPath() D) All

16. Default key value separator in keyValueTextInputFormat is___ [ ]

A) Tab B) White Space C) New line Character D) None

17. InNLineInputFormat N Refers to the [ ]

A) number of lined of output that each mapper returns.

B) number of lined of input that each mapper returns.

C) number of lined of output that each Reducer returns.

D) number of lined of input that each Reducer returns.

18. MultipleInputs class has an overload version of___________ that doesn't take a mapper. [ ]

A) setInputPath() B) getStart() C) addInputPath() D) None

19. Default output format__________ [ ]

A) BinaryoutputFormat C) TextInputFormat

B) BinaryOutputFormat D) LazyoutputFormat

20. ______ format is used for writing relational databases and HBase[ ]

A) DatabaseInput C) HBaseInput

B) DatabaseOuput D) HBaseOutput

SECTION-B

SUBJECTIVE QUESTIONS

1. Define shuffle and sort? Why it is required?
2. Write the general form of map and reduce functions and also writ the JAVA API mirrors this general form.
3. Explain Map side Tuning properties in configuration tuning.
4. What constitutes progress in MapReduce? Explain
5. Illustrate reduce side tasks in shuffle and sort
6. What is input split? How to represent input splits? Which methods are used to get location and length of input splits.
7. Explain how to control split size with example.
8. Elaborate classic frame work to run a Mapreduce job in Hadoop
9. Draw and explain inputformat class hierarch
10. Differentiate TextInputFormat and KeyValueTextInputFormat
11. List the Reduce-side tuning properties in configuration tuning.
12. How status updates are propagated through the MapReduce 1 system
13. Draw and explain outputformat class hierarchy
14. List the Binary output formats supported by hadoop.
15. Examine the relationship of streaming and pipes executable to the tasktracker and its child.

HIVE

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 Jeff Hammerbacher at facebook, later the apache software foundation took it up and developed it further as an open source under the apache hive.

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

5.1 Hive Shell

The shell is the primary way that we will interact with Hive, by issuing commands in *HiveQL*. HiveQL is Hive's query language, a dialect of SQL. It is heavily influenced by MySQL, so if you are familiar with MySQL you should feel at home using Hive.

When starting Hive for the first time, we can check that it is working by listing its tables: there should be none. The command must be terminated with a semicolon to tell Hive to execute it: Ex: hive> **SHOW TABLES;**

```
OK
Time taken: 10.425 seconds
```

Features of shell

It is possible to run the hive shell in non-interactive mode. The –f option runs the commands in the specified file, ex: *script.q*,

```
% hive -f script.q
```

For short scripts, you can use the -e option to specify the commands inline, in which case the final semicolon is not required:

```
% hive -e 'SELECT * FROM dummy'

Hive history file=/tmp/tom/hive_job_log_tom_201005042112_1906486281.txt

OK

X

Time taken: 4.734 seconds
```

In both interactive and non-interactive mode, Hive will print information to standard error—such as the time taken to run a query, to surpress these messages using the –s option it shows only the result of the query.

% **hive -S -e 'SELECT * FROM dummy'**

X

Other useful Hive shell features include the ability to run commands on the host operating system by using a! Prefix to the command and the ability to access Hadoop filesystems using the dfs command

5.2 Hive Services

The Hive shell is only one of several services that you can run using the hive command.

You can specify the service to run using the --service option. Type hive –service help to get a list of available service names; the most useful are described below.

1. cli

The command line interface to Hive (the shell). This is the default service.

2. hiveserver

Runs Hive as a server exposing a Thrift service, enabling access from a range of clients written in different languages. Applications using the Thrift, JDBC, and ODBC connectors need to run a Hive server to communicate with Hive. Set the HIVE_PORT environment variable to specify the port the server will listen on (defaults to 10,000).

3. hwi

The Hive Web Interface.

4. Jar

   The Hive equivalent to hadoop jar, a convenient way to run Java applications that includes both Hadoop and Hive classes on the classpath.

5. metastore

Using this service, it is possible to run the metastore as a standalone (remote) process.

5.3 Hive Clients

Hive as a server (hive --service hiveserver), then there are a number of different mechanisms for connecting to it from applications.

The relationship between Hive clients and Hive services is illustrated in Figure

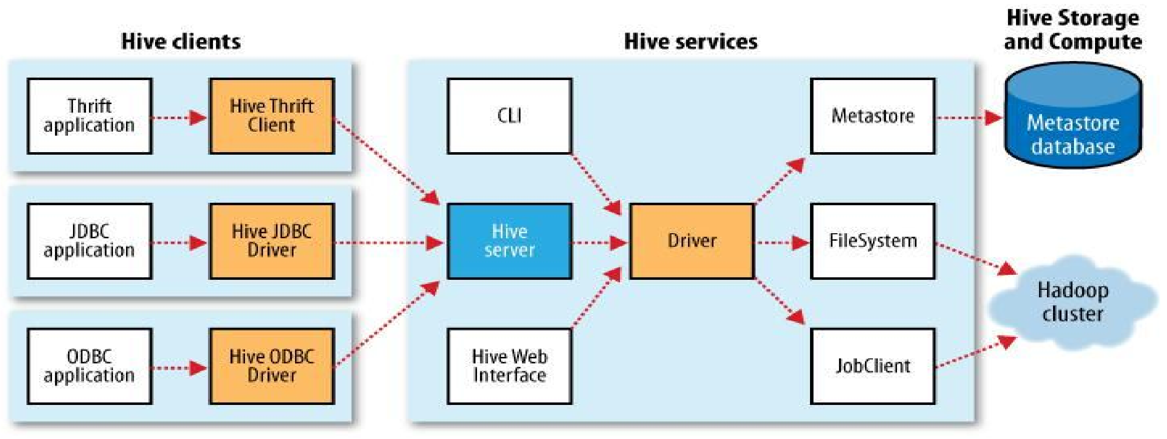


Fig: Hive architecture

- ***Thrift Client***

The Hive Thrift Client makes it easy to run Hive commands from a wide range of programming languages. Thrift bindings for Hive are available for C++, Java, PHP, Python, and Ruby.

- ***JDBC Driver***

Hive provides a Type 4 (pure Java) JDBC driver, defined in the class org.apache.hadoop.hive.jdbc.HiveDriver. When configured with a JDBC URI of the form jdbc:hive://*host*:*port*/*dbname*, a Java application will connect to a Hive server running in a separate process at the given host and port

- ***ODBC Driver***

The Hive ODBC Driver allows applications that support the ODBC protocol to connect to Hive.

5.4 The metastore

- The *metastore* is the central repository of Hive metadata.

- The metastore is divided into two pieces: a service and the backing store for the data. There are 3 different metastore configurations

1. Embedded metastore

- By default, the metastore service runs in the same JVM as the Hive service and contains an embedded Derby database instance backed by the local disk. This is called the *embedded metastore* configuration

- a simple way to get started with Hive

- only one embedded Derby database can access the database files on disk at any one time, which means you can only have one Hive session

- if we open another session it attempts to open a connection to the metastore i.e., trying to start a second session gives the error

Error: Failed to start database 'metastore_db'

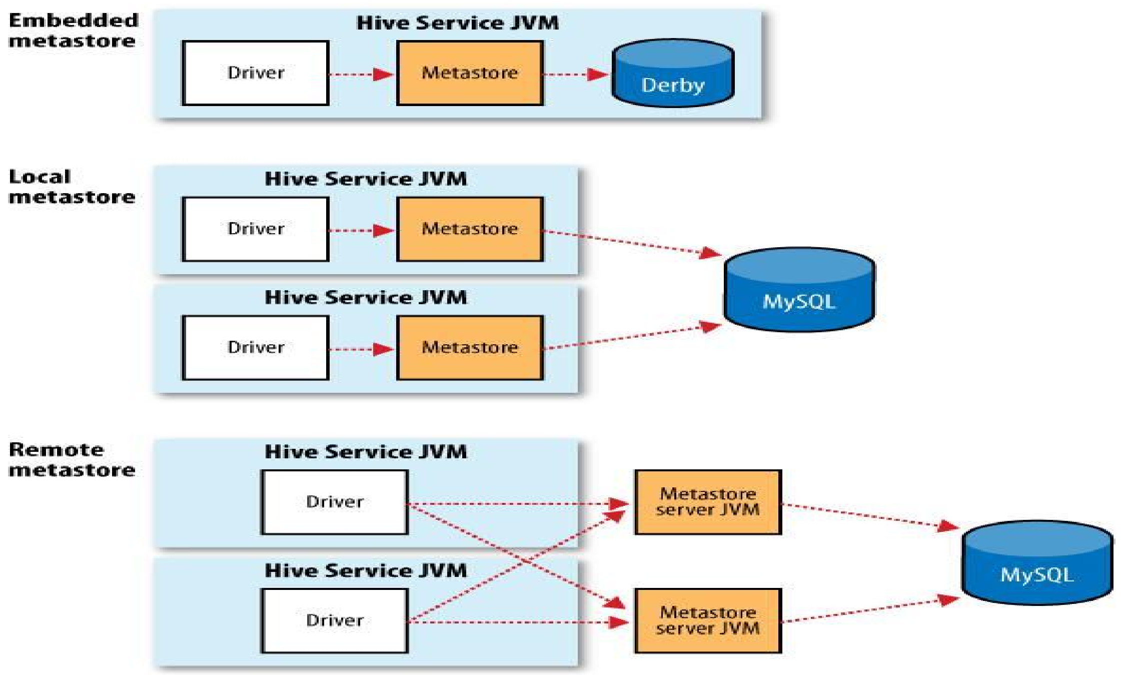


Figure : Metastore configurations

2.local metastore

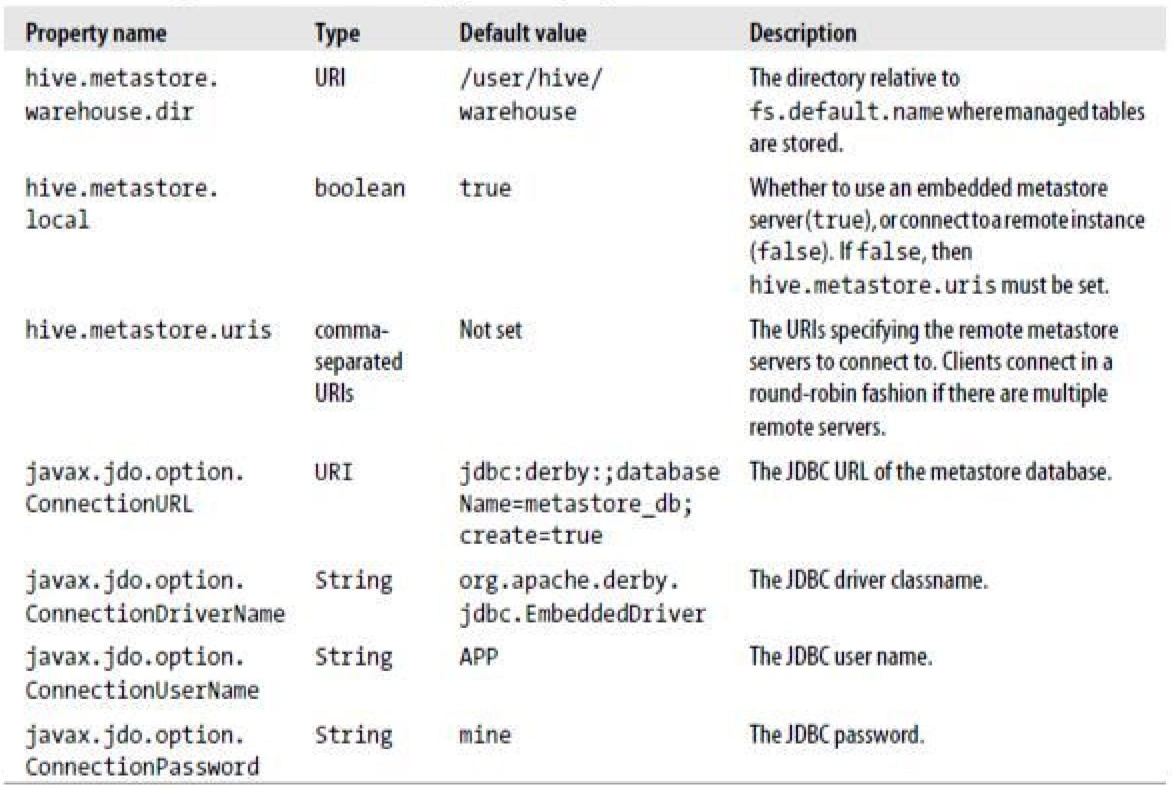
- To support multiple sessions (and therefore multiple users) is to use a standalone database.
- This configuration is referred to as a *local metastore*, since the metastore service still runs in the same process as the Hive service, but connects to a database running in a separate process, either on the same machine or on a remote machine.

3. Remote metastore

- o Where one or more metastore servers run in separate processes to the Hive service.

- o This brings better manageability and security, since the database tier can be completely firewalled off, and the clients no longer need the database credentials.

Table 12-1. Important metastore configuration properties



Property name	Type	Default value	Description
`hive.metastore.warehouse.dir`	URI	`/user/hive/warehouse`	The directory relative to `fs.default.name` where managed tables are stored.
`hive.metastore.local`	boolean	`true`	Whether to use an embedded metastore server (`true`), or connect to a remote instance (`false`). If `false`, then `hive.metastore.uris` must be set.
`hive.metastore.uris`	comma-separated URIs	Not set	The URIs specifying the remote metastore servers to connect to. Clients connect in a round-robin fashion if there are multiple remote servers.
`javax.jdo.option.ConnectionURL`	URI	`jdbc:derby:;databaseName=metastore_db;create=true`	The JDBC URL of the metastore database.
`javax.jdo.option.ConnectionDriverName`	String	`org.apache.derby.jdbc.EmbeddedDriver`	The JDBC driver classname.
`javax.jdo.option.ConnectionUserName`	String	`APP`	The JDBC user name.
`javax.jdo.option.ConnectionPassword`	String	`mine`	The JDBC password.

5.5 Comparison with traditional databases

Schema on Read versus Schema on Write

- In a traditional database, a table's schema is enforced at data load time.
- Hive does not verify the data when it is loaded but it is verified when the query is issued
- Traditional db takes longer time to load data
- Schema on read makes for a very fast

initial load **Query time performance**:

- Schema on write makes query time performance faster

- Schema on read makes longer time for query execution

Transactions:

Hive doesnot support transactions.

Hive does not support updates(or deletes).

Indexes:

Release 0.7.0 introduced indexes, which can speed up queries

Locking:

Release 0.7.0 introduces table and partitional level locking in hive.

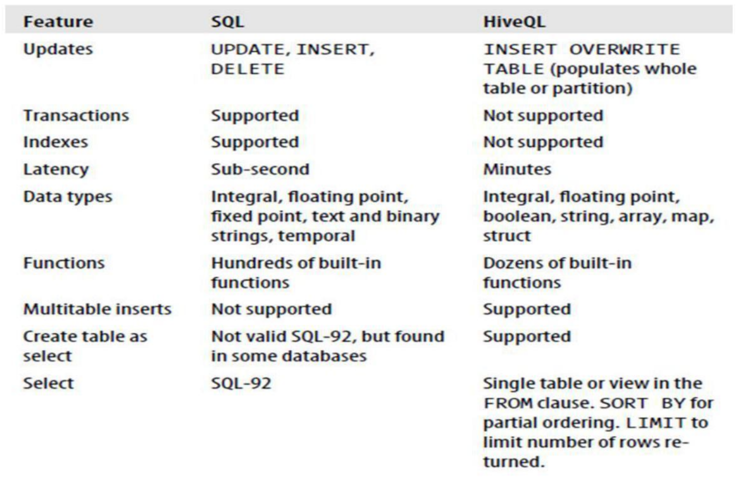
Locks are managed transparently by zookeeper.

5.6 Hive QL

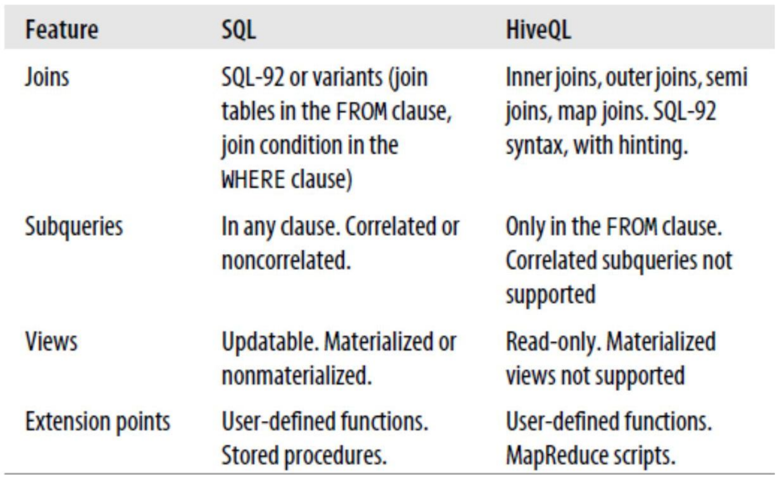
HiveQL is Hive's SQL dialect.

It does not provide the full features of SQL-92 language constructs.

The main difference between HiveQL and SQL are *Table : A high-level comparison of SQL and HiveQL*



Feature	SQL	HiveQL
Updates	UPDATE, INSERT, DELETE	INSERT OVERWRITE TABLE (populates whole table or partition)
Transactions	Supported	Not supported
Indexes	Supported	Not supported
Latency	Sub-second	Minutes
Data types	Integral, floating point, fixed point, text and binary strings, temporal	Integral, floating point, boolean, string, array, map, struct
Functions	Hundreds of built-in functions	Dozens of built-in functions
Multitable inserts	Not supported	Supported
Create table as select	Not valid SQL-92, but found in some databases	Supported
Select	SQL-92	Single table or view in the FROM clause. SORT BY for partial ordering. LIMIT to limit number of rows re-turned.



Feature	SQL	HiveQL
Joins	SQL-92 or variants (join tables in the FROM clause, join condition in the WHERE clause)	Inner joins, outer joins, semi joins, map joins. SQL-92 syntax, with hinting.
Subqueries	In any clause. Correlated or noncorrelated.	Only in the FROM clause. Correlated subqueries not supported
Views	Updatable. Materialized or nonmaterialized.	Read-only. Materialized views not supported
Extension points	User-defined functions. Stored procedures.	User-defined functions. MapReduce scripts.

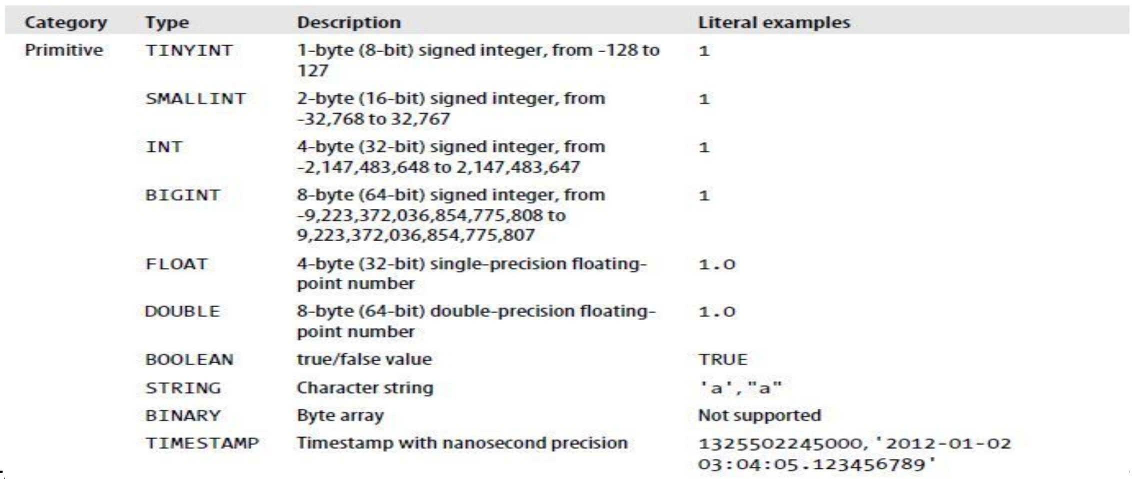
Data Types

- Hive supports both primitive and complex data types.
- Primitives include

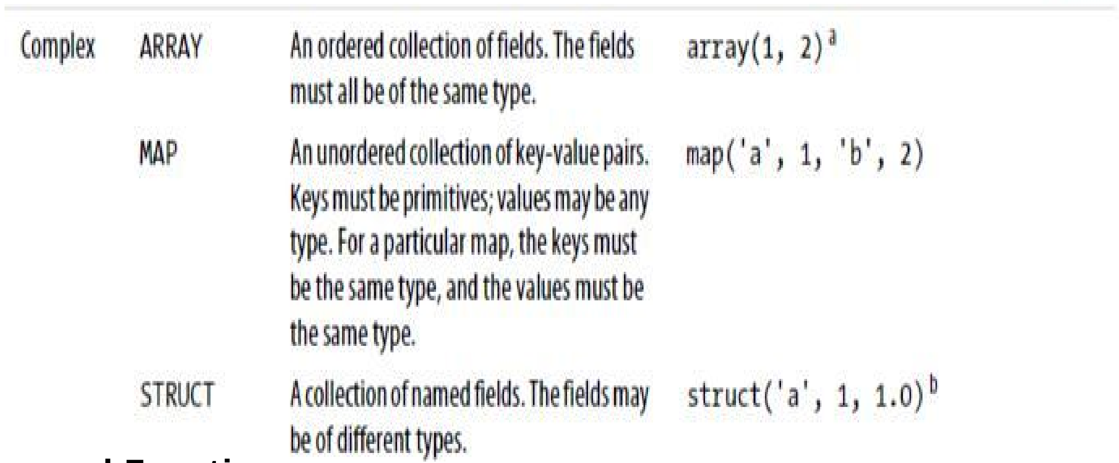
  1) numeric
  2) boolean
  3) string,
  4) timestamp

- complex data types include

  1) arrays
  2) maps
  3) structures



Category	Type	Description	Literal examples
Primitive	TINYINT	1-byte (8-bit) signed integer, from -128 to 127	1
	SMALLINT	2-byte (16-bit) signed integer, from -32,768 to 32,767	1
	INT	4-byte (32-bit) signed integer, from -2,147,483,648 to 2,147,483,647	1
	BIGINT	8-byte (64-bit) signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	1
	FLOAT	4-byte (32-bit) single-precision floating-point number	1.0
	DOUBLE	8-byte (64-bit) double-precision floating-point number	1.0
	BOOLEAN	true/false value	TRUE
	STRING	Character string	'a', "a"
	BINARY	Byte array	Not supported
	TIMESTAMP	Timestamp with nanosecond precision	1325502245000, '2012-01-02 03:04:05.123456789'



Complex	ARRAY	An ordered collection of fields. The fields must all be of the same type.	`array(1, 2)`[a]
	MAP	An unordered collection of key-value pairs. Keys must be primitives; values may be any type. For a particular map, the keys must be the same type, and the values must be the same type.	`map('a', 1, 'b', 2)`
	STRUCT	A collection of named fields. The fields may be of different types.	`struct('a', 1, 1.0)`[b]

Operators and Functions

- Operators
  1) arithmetic
  2) relational
  3) logical
- Categories of Functions
  1) mathematical
  2) statistical
  3) string
  4) date
  5) conditional
  6) aggregate
  7) Functions working with XML and JSON

5.7 Tables

A Hive table is logically made up of the data being stored and the associated metadata describing the layout of the data in the table.

The data typically resides in HDFS, although it may reside in any Hadoop filesystem, including the local filesystem or S3.

Hive stores the metadata in a relational database—and
not in HDFS, **Managed Tables and External Tables**

When you create a table in Hive, by default Hive will manage the data, which means that Hive moves the data into its warehouse directory.

For example: CREATE TABLE managed_table (dummy STRING);

LOAD DATA INPATH '/user/tom/data.txt' INTO table managed_table; will *move* the file *hdfs://user/tom/data.txt* into Hive's warehouse directory for the managed_table table, which is *hdfs://user/hive/warehouse/managed_table*

If the table is later dropped, using:

DROP TABLE managed_table;

then the table, including its metadata *and its data*, is deleted.

An external table behaves differently. You control the creation and deletion of the data.

The location of the external data is specified at table creation time:

CREATE EXTERNAL TABLE external_table (dummy STRING)

LOCATION '/user/tom/external_table';

LOAD DATA INPATH '/user/tom/data.txt' INTO TABLE external_table;

Partitions and Buckets

Hive organizes tables into *partitions*, a way of dividing a table into coarse-grained parts based on the value of a *partition column*, such as date. Tables or partitions may further be subdivided into *buckets*, to give extra structure to the data that may be used for more efficient queries.

Partitions

- Hive organizes tables in to partitions.

- A way of dividing a table in to related parts based on the value of a partition column ex: date, city, dept

- Faster to do queries on slices of the data

- Tables or partitions are sub-divided into buckets , to provide extra structure to the data that may be used for more efficient querying

- Bucketing works based on the value of hash function of some column of a table. Ex:

  A table named Tab1 contains employee data such as id, name, dept, and yoj , need to retrieve the details of all employees who joined in 2012.

- A query searches the whole table for the required information.

- if you partition the employee data with the year and store it in a separate file, it reduces the query processing time.

  The following file contains employee data table. /tab1/employeedata/file1 id, name, dept, yoj

  1, gopal, TP,2012
  2, kiran,HR,2012
  3, kaleel,SC,2013
  4,Prasanth,SC,20

The above data is partitioned into two files using year.

```
/tab1/employeedata/2012/file2/tab1/employeedata/2013/file3
1, gopal, TP, 2012                3, kaleel,SC, 2013

2, kiran, HR, 2012                4, Prasanth, SC, 2013
```

Adding a Partition :-
We can add partitions to a table by altering the
table hive> ALTER TABLE employee

> ADD PARTITION (year='2013')

> location '/2012/part2012';

Renaming a Partition

hive> ALTER TABLE employee PARTITION (year='1203')

> RENAME TO PARTITION (Yoj='1203');

Show Partition

hive> show partitions employee;

Dropping a Partition

hive> ALTER TABLE employee DROP [IF EXISTS]

> PARTITION (year='1203');

Buckets

- It is a mechanism to query and examine random samples of data

- Break data into a set of buckets based on a hash function of a ―bucket column

- Capability to execute queries on a sub-set of random data

- Doesn't automatically enforce bucketing

- User is required to specify the number of buckets by setting # of reducer Create and use table with Buckets

  hive>create table post-count(user String,
  count Int) >clustered by (user) into 5
  buckets;

  Use the clustered by clause to specify columns to bucket on and the number of buckets.

Storage Formats

- There are two dimensions that govern table storage in Hive: the *row format* and the *file format*. The row format dictates how rows, and the fields in a particular row, are stored

- The file format dictates the container format for fields in a row. The simplest format is a plain text file, but there are row-oriented and column-oriented binary formats available,too.

- The default storage format: Delimited text

- When you create a table with no ROW FORMAT or STORED AS clauses, the default format is delimited text, with a row per line.

- The default row delimiter is not a tab character, but the Control-A character from the set of ASCII control codes

- The choice of Control-A, sometimes written as ^A in documentation, came about since it is less likely to be a part of the field text than a tab character.

The default collection item delimiter is a Control-B character, used to delimit items in an ARRAY or STRUCT, or key-value pairs in a MAP. The default map key delimiter is a Control-C character, used to delimit the key and value in a MAP. Rows in a table are delimited by a newline character.

- Importing Data
    - o INSERT OVERWRITE TABLE
- a table with data from another Hive table using an INSERT statement i.e
- Example

```
hive>  INSERT  OVERWRITE  TABLE
        target >SELECT col1, col2

        >FROM
        source;
```

Another way

```
Hive> FROM source

        >INSERT OVERWRITE TABLE target

        >SELECT col1, col2;
```

- Multitable insert
- it's possible to have multiple INSERT clauses in the same query.

- *multitable insert is more efficient than* multiple INSERT statements,

- the source table need only be scanned once to produce the multiple, disjoint outputs hive> FROM records2

```
>INSERT         OVERWRITE         TABLE
stations_by_year
>SELECT year, COUNT(DISTINCT station)
>GROUP BY year

>INSERT         OVERWRITE         TABLE
records_by_year
>SELECT year, COUNT(1)
>GROUP BY year
>INSERT            OVERWRITE            TABLE
good_records_by_year
>SELECT year, COUNT(1)
>WHERE temperature != 9999
>AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR
quality = 9)
>GROUP BY year;
```

- There is a single source table (records2), but three tables to hold the results from three different queries over the source.

    - o CREATE TABLE...AS SELECT

- To store the output of a Hive query in a new table.

- The new table's column definitions are derived from the columns retrieved by the

SELECT
clause
Example

CREATE TABLE target
AS

SELECT col1, col2

FROM source;

- Alter TableAlter the attributes of a table such as changing its table name, changing column names, adding columns, and deleting or replacing columns.

Syntax

ALTER TABLE name RENAME TO new_name

ALTER TABLE name ADD COLUMNS (col_spec[, col_spec ...])

ALTER TABLE name DROP [COLUMN] column_name

ALTER TABLE name CHANGE column_name new_name
new_type ALTER TABLE name REPLACE COLUMNS
(col_spec[, col_spec ...]) Examples

hive> ALTER TABLE employee RENAME TO emp;
hive>ALTER TABLE employee ADD COLUMNS (dept
string); hive>ALTER TABLE employee DROP dept;

hive> ALTER TABLE employee CHANGE name ename
String; hive> ALTER TABLE employee CHANGE salary
salary Double;

hive> ALTER TABLE employee REPLACE COLUMNS (eid INT empid Int, ename STRING name String);

hive> DROP TABLE IF EXISTS employee;

- Select Statements

SELECT statement is used to retrieve the data from a table. WHERE clause works similar to a condition. It filters the data using the condition and gives you a finite result. Syntax:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...

         FROM table_reference

        [WHERE where_condition]

         [GROUP BY col_list]

        [HAVING having_condition]

        [CLUSTER BY col_list |

         [DISTRIBUTE BY col_list]

        [SORT BY col_list]]

        [LIMIT number];
```

Examples:

```
hive> SELECT * FROM employee WHERE
eid=1205; hive> SELECT * FROM employee
WHERE
salary>=40000;

hive> SELECT Id, Name, Dept FROM employee ORDER
BY DEPT;
 hive> SELECT Dept,count(*) FROM employee GROUP BY
DEPT;
 hive> FROM records2
          SELECT year, temperature

          DISTRIBUTE BY year
          SORT BY year ASC, temperature
          DESC;
hive> SELECT * FROM employee LIMIT 4;
```

5.8 querying data

Sorting and Aggregating

Sorting data in Hive can be achieved by use of a standard ORDER BY clause, but there is a catch. ORDER BY produces a result that is totally sorted, as expected, but to do so it sets the number of reducers to one, making it very inefficient for large datasets.

In some cases, you want to control which reducer a particular row goes to, typically so you can perform some subsequent aggregation. This is what Hive's DISTRIBUTE BY clause does. Here's an example to sort the weather dataset by year and temperature

SORT BY produces a sorted file per reducer.

```
hive> FROM records2
> SELECT year, temperature
> DISTRIBUTE BY year
```

> **SORT BY year ASC, temperature DESC;** 1949 111 1949 78 1950 22
1950 0 1950 -11

MapReduce Scripts

Using an approach like Hadoop Streaming, the TRANSFORM, MAP, and REDUCE clauses make it possible to invoke an external script or program from Hive.

Example: Python script to filter out poor quality weather records

```
#!/usr/bin/env python
import re

import sys

for line in sys.stdin:

(year, temp, q) = line.strip().split()



if (temp != "9999" and re.match("[01459]", q)):

print "%s\t%s" % (year, temp)
```

We can use the script as follows:

hive> **ADD FILE /path/to/is_good_quality.py;**

hive> **FROM records2**

> **SELECT TRANSFORM(year, temperature, quality)**

> **USING 'is_good_quality.py'**

> **AS year,**
temperature; 1949
111 1949 78 1950 0

1950 22

1950 -11
Before running the query, we need to register the script with Hive. This is so Hive knows to ship the file to the Hadoop cluster

The query itself streams the year, temperature, and quality fields as a tab-separated line to the *is_good_quality.py* script, and parses the tab-separated output into year and temperature fields to form the output of the query.

This example has no reducers. If we use a nested form for the query, we can specify a map and a reduce function. This time we use the MAP and REDUCE keywords, but SELECT TRANSFORM in both cases would have the same result. The source for the *max_temperature_reduce.py* script is shown in Example
FROM
(FROMrecos
2

MAP year, temperature,
quality USING
'is_good_quality.py'

AS year, temperature)
map_output **REDUCE** year,
temperature USING
'max_temperature_reduce.py'
AS year, temperature;

Views

- A view is a sort of —virtual table that is defined by a SELECT statement.
- Views can be used to present data to users in a different way to the way it is actually stored on disk.
- Views may also be used to restrict users access to particular subsets of tables that they are authorized to see.
  - First create table and then insert data into it

```
hive> create table posts(id int,name string,sal double)
> row format delimited
> fields terminated by ','
      stored as textfile;
```

  - Create View

```
hive> create view posts_name as
    > select name from posts;
hive> create view first_id as
        select * from posts whereid=1;
hive> create view max_sal as
        select name ,max(sal) from posts;
        Show views
```

hive> show tables;

- Altering views

hive> alter view first_id rename to 1stid;

- Drop a view

hive> drop view 1stid;

Joins

- JOIN is a clause that is used for combining specific fields from two tables by using values common to each one.
- used to combine records from two or more tables in the database.
- similar to SQL JOINS.
- There are different types of joins given as follows:
  - JOIN
  - LEFT OUTER JOIN
  - RIGHT OUTER JOIN
  - FULL OUTER JOIN
- Can join multiple tables
- Default join Is Inner join
  - Rows are joined where the keys match

- Rows that do not have matches are not included in the result

The simplest kind of join is the inner join, where each match in the input tables results in a row in the output table it is being joined to (things): hive> **SELECT * FROM sales;**

```
joe      2
Hank     4
Ali      0
Eve      3
Hank     2
```

hive> **SELECT * FROM things;**

```
2   Tie
4   Coat
3   Hat
   1      Scarf
```

hive> **SELECT sales.*, things.***

> **FROM sales JOIN things ON (sales.id = things.id);**

```
Joe     2 2 Tie

Hank 2 2 Tie

Eve     3 3 Hat

Hank  4 4 Coat
```

- The table in the FROM clause (sales) is joined with the table in the JOIN clause (things), using the predicate in the ON clause

- Hive only supports equijoins, which means that only equality can be used in the join predicate, which here matches on the id column in both tables.

- the row for Ali did not appear in the output, since the ID of the item she purchased was not present in the things table

Left Outer Join

Outer joins allow you to find non matches in the tables being joined.

If we change

the join type to LEFT OUTER JOIN, then the query will return a row for every row in the left table (sales), even if there is no corresponding row in the table it is being joined to (things):

```
hive> SELECT * FROM sales;

Joe     2
Hank    4
Ali     0
Eve     3
Hank    2

hive> SELECT * FROM things;

2   Tie

4   Coat

3   Hat
        1   Scarf
```

The row for Ali is now returned, and the columns from the things table are NULL, since there is no match.

i.e Row from the first table are included whether they have a match or not. Columns from the unmatched(second) table are set to null.

Right Outer Join

- Opposite of Left Outer Join, Rows from the second table are included no matter what. Columns from the unmatched (first) table are set to null.
- all items from the things table are included, even those that weren't purchased by anyone
  (a scarf):

```
hive> SELECT * FROM sales;

Joe    2
Hank   4
Ali    0
Eve    3
Hank   2

hive> SELECT * FROM things;

2   Tie

4   Coat

3   Hat
        1   Scarf
```

We can perform left outer join on the two tables as follows:

```
hive> SELECT sales.*, things.*
```

```
> FROM sales RIGHT OUTER JOIN things ON (sales.id =
things.id); NULL NULL 1 Scarf

Joe   2 2 Tie
Han     2     2
k         Tie
Eve   3 3 Hat
              Coa
Hak  4 4 t
```

Full Outer Join

- Rows from both sides are includes. For unmatched rows the columns from the other table are set to null
- In full outer join, the output has a row for each row from both tables in the join:

```
hive> SELECT * FROM sales;
Joe     2
Hank  4
Ali      0
Eve     3
```

We can perform left outer join on the two tables as follows:

```
hive> SELECT sales.*, things.*
```

```
 Ali

 Joe

 HanEve

Hank
```

> **FROM sales LEFT OUTER JOIN**

```
0 NULL NULL

2 2 Tie

2 2 Tie

3 3 Hat

4 4 Coat
```

Subqueries

- A subquery is a SELECT statement that is embedded in another SQL statement.
- Hive has limited support for subqueries

The query finds the mean maximum temperature for every year and weather station:

```
SELECT station, year, AVG(max_temperature)

FROM (

SELECT station, year, MAX(temperature) AS
max_temperature FROM records2

WHERE temperature != 9999

AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality =
9)
GROUP BY station, year

) mt
GROUP BY station, year;
```

- The subquery is used to find the maximum temperature for each station/date combination,
- the outer query uses the AVG aggregate function to find the average of the maximum temperature readings for each station/date combination.
- The outer query accesses the results of the subquery like it does a table, which is why the subquery must be given an alias (mt).
- The columns of the subquery have to be given unique names so that the outer query can refer to them.

5.9 User-Defined functions

- Write the query that can't be expressed easily using built-in functions.
- Write a User-Defined Function(UDF) .
- Easy to plug in own processing code and invoke it from a Hive Query.
- There are 3 types of UDF in Hive

1) Regular **UDF**s

Operates on a single row and produces a single row as its output. Ex: Mathematical and String functions

2) **UDAF** (User-defined aggregate functions)

- works on multiple input rows and creates a single output row.
- Aggregate functions include such functions as COUNT and MAX.

3) **UDTF**s (user-defined table-generating functions)

- operates on a single row and produces multiple rows—a table—as output

5.9 User-Defined functions

- Write the query that can’t be expressed easily using built-in functions.

- Write a User-Defined Function(UDF) .

- Easy to plug in own processing code and invoke it from a Hive Query.

- There are 3 types of UDF in Hive

2) Regular **UDF**s

Operates on a single row and produces a single row as its output. Ex: Mathematical and String functions

4) **UDAF** (User-defined aggregate functions)

- works on multiple input rows and creates a single output row.

- Aggregate functions include such functions as COUNT and MAX.

5) **UDTF**s (user-defined table-generating functions)

- operates on a single row and produces multiple rows—a table—as output

UNIT-VI
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. Which of the following command sets the value of a particular configuration variable [ ]

   A) Set-v B) set <key>=<value> C) set D) reset

2. Which of the following operator executes a shell command from the Hive shell? [ ]

   A) | B) ! C^ D) +

3. Which of the following will remove the resource(s) from the distributed cache? [ ]

   A) Delete FILE[S] <filepath>*

   B) Delete JAR[S]<filepath>*

   C) Delete ARCHIVE[S]<filepath>*

   D) All

4. _______ is a shell utility which can be used to run Hive queries in either interactive or batch mode. [ ]

   A) $HIVE/bin/hive

   B) $HIVE_HOME/hive

   C) $HIVE_HOME/bin/hive

   D) All

5. Which of the following is a command line option? [ ]

   A) –d,-define <key=value>

   B) –e,-define<key=value>

   C) –f,-define<key=value>

   D) None

6. Hive uses__________ for logging [ ]

   A)logj4 B) log41 C) log4i D) log4j

7. Hive Server2 introduced in HIVE 0.11 has new CLI called [ ]

   A) BeeLine B) SQLLine C)HIVELine D) CLILine

8. Hcatalog is installed with HIVE, starting with HIVE relase [ ]

   A) 0.10.0 B) 0.9.0 C)0.11.0 D)0.1.20

9. _____ supports a new command shell Beeline that works with HIVE Server2. [ ]

A) HiveServer2 B) HiveServer3 C) HiveServer4 D) None

10. In _______ mode HiveServer2 only accepts valid Thrift calls. [ ]

A) Remote B) HTTP C) Embedded D) Interactive

11. Hive specific commands can be run from Beeline, When the Hive _____ driver is used. [ ]

A) ODBC B) JDBC C) ODBC-JDBC D) ALL

12. The ___ allow users to read or write Avro data s Hive Table [ ]

A) AvroSerde B) HiveSerde C) SQLSerde D) None

13. Starting in Hive_____ the Avro schema can be inferred from the hive table schema. [ ]

A) 0.14 B) 0.12 C) 0.13 D) 0.11

14. Which of the following data type is supported by HIVE [ ]

A) map B) record C) string D) enum

15. which of the following data type is converted to Array prior to Hive 0.12.0 [ ]

A) map B) long C) float D) bytes

16.Avro-backed tables can simply be created by using _________ in a DDL statement. [ ]

A) "STORED AS AVERO" C. –STORED AS AVROHIVE

B) –STORED AS HIVE D. –STORED AS SERED

17. Types that may be null must be defined as a ___________ of that type and NULL within AVRO. [ ]

A) Union B) intersection C) Set D) All

18. use_____ and embed the schema in the create statement [ ]

A) schema.literal B) schema.lit C) row.literal D) All

19. Serialization of string columns uses a____ to form unique column values.

A) Footer B) STRIPES C) Dictionary D) Index

20. Hive uses________ -Style escaping within the strings [ ]

A) C B) JAVA C) python D) Scala

SECTION-B

SUBJECTIVE QUESTIONS

1. What is hive? List the features of hive?
2. List out hive Services
3. What is metastore? What are different types of metastores?
4. What are megastore configuration properties?
5. Comapre the SQL and HIVEQL
6. List out Hive Data Types?
7. Explain about partitions and buckets?
8. Outline about Querying Data?
9. What are user-defined functions?
10. Explain joins?
11. Explain about HIVEQL in Hadoop System
12. Illustrate the HIVE Shell?
13. Describe about the tables in HIVE.
14. Explain about HIVE architecture?
15. Compare HIVE with traditional database?
16. Elaborate on HIVE QL data manipulation and queries in details
17. Discuss about the relationship between HIVE clients and HIVE Services with a neat diagram?
18. Explain in detail about Map side and Reduce Side joins.