SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

School of Computing

Department of Computer Science and Engineering

UNIT I - Big Data – SBS1608

SYLLABUS

Introduction to BigData Platform – Challenges of Conventional Systems - Intelligent data analysis Nature of Data - Analytic Processes and Tools - Analysis vs Reporting - Modern Data Analytic Tools.

1. Introduction to Big Data Platform

Introduction to Big Data:

Big Data has to deal with large and complex datasets that can be structured, Semi-structured, or unstructured and will typically not fit into memory to be processed.

Big data is a field

- ways to analyze,
- systematically extract information or
- deal with data sets that are too large or complex to be dealt with by traditional data-processing application software

Big data is a phrase like software testing

- describes about capturing, storing, querying, updating
- and analyzing of huge data sets that are so voluminous and complex
- that traditional data-processing application software are inadequate to deal with them

Why Big Data?

- Cost / Time Reduction
- Faster and Better Decision Making
- New Products and Services

A big data platform is a tool that has been developed by data management vendors with an aim of increasing the scalability, availability, performance, and security of organizations that are driven using big data. The platform is designed to handle voluminous data that is multi-structured in real time. Big data platform - consists of big data storage, servers, database, big data management, business intelligence and other big data management utilities. It supports custom development, querying and integration with other systems. Primary benefit is reducing the complexity of multiple vendors/ solutions into a one cohesive solution. Big data platform are also delivered through cloud where the provider provides big data solutions and services.

New analytic applications drive the requirements for a big data platform -

- •Integrate and manage the full variety, velocity and volume of data
- •Apply advanced analytics to information in its native form
- •Visualize all available data for ad-hoc analysis
- •Development environment for building new analytic applications
- •Workload optimization and scheduling
- •Security and Governance

Augments open source Hadoop with enterprise capabilities:

- –Enterprise-class storage
- –Security
- –Performance Optimization
- –Enterprise integration
- –Development tooling

–Analytic Accelerators
–Application and industry accelerators
–Visualization

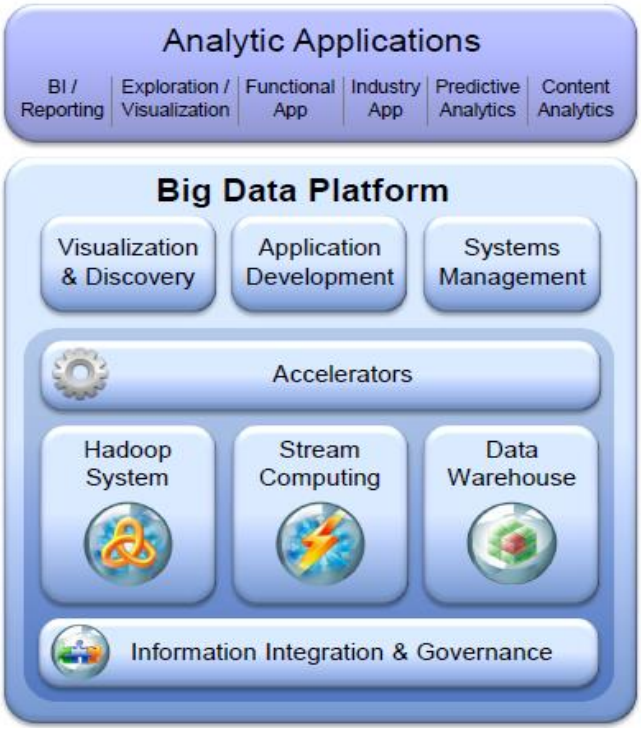


Fig. 1.1 Big Data Platform

Workload Optimization:
Adaptive MapReduce

- •Algorithm to optimize execution time of multiple small and large jobs
- •Performance gains of 30% reduce overhead of task startup Hadoop System Scheduler
- •Identifies small and large jobs from prior experience
- •Sequences work to reduce overhead

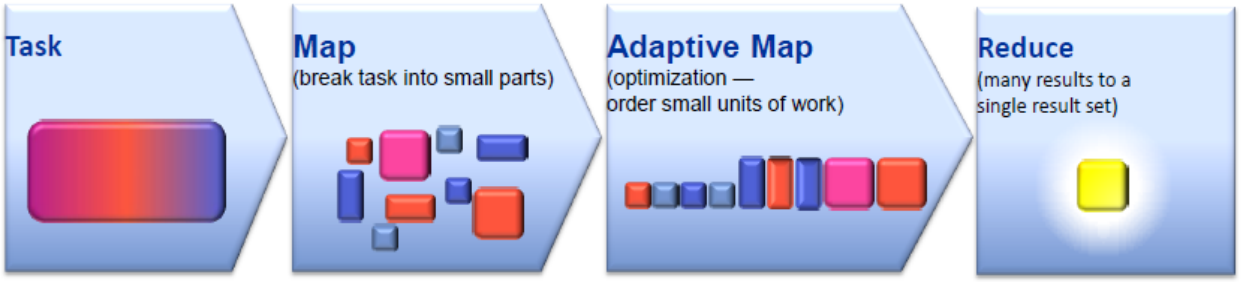


Fig. 1.2 MapReduce

Big Data Platform - Stream Computing:

Built to analyze data in motion

- •Multiple concurrent input streams
- •Massive scalability

Process and analyze a variety of data

- •Structured, unstructured content, video, audio
- •Advanced analytic operators

Big Data Platform - Data Warehousing:

Workload optimized systems

–Deep analytics appliance

–Configurable operational analytics appliance

–Data warehousing software

Capabilities

•Massive parallel processing engine

•High performance OLAP

•Mixed operational and analytic workloads

Big Data Platform - Information Integration and Governance

Integrate any type of data to the big data platform

–Structured

–Unstructured

–Streaming

Governance and trust for big data

–Secure sensitive data

–Lineage and metadata of new big data sources

–Lifecycle management to control data growth

–Master data to establish single version of the truth

Leverage purpose-built connectors for multiple data sources:

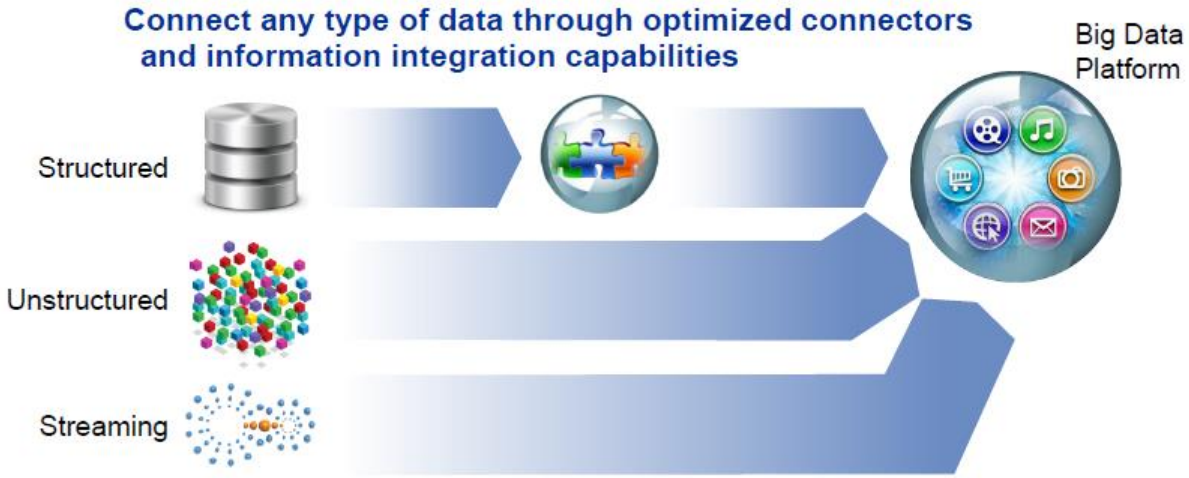


Fig. 1.3 Multiple Data source

- Massive volume of structured data movement

  •2.38 TB / Hour load to data warehouse

  •High-volume load to Hadoop file system

- Ingest unstructured data into Hadoop file system
- Integrate streaming data sources

Big Data Platform - User Interfaces

•Business Users

•Visualization of a large volume and wide variety of data

•Developers
•Similarity in tooling and languages
•Mature open source tools with enterprise capabilities
•Integration among environments

Administrators

•Consoles to aid in systems management

Big Data Platform –Accelerators:

Analytic accelerators

–Analytics, operators, rule sets

Industry and Horizontal Application Accelerators

–Analytics
–Models
–Visualization / user interfaces
–Adapters

Big Data Platform - Analytic Applications:

Big Data Platform is designed for analytic application development and integration.
BI/Reporting – Cognos BI, Attivio
Predictive Analytics – SPSS, G2, SAS
Exploration/Visualization – BigSheets, Datameer
Instrumentation Analytics – Brocade, IBM GBS
Content Analytics – IBM Content Analytics
Functional Applications – Algorithmics, Cognos Consumer Insights, Clickfox, i2, IBM GBS

Industry Applications – TerraEchos, Cisco, IBM GBS

2. Challenges of Conventional System

Fundamental challenges

- How to store
- How to work with voluminous data sizes,
- and more important, how to understand data and turn it into a competitive advantage.

How about Conventional system technology?

• CPU Speeds:
  – 1990 - 44 MIPS at 40 MHz
  – 2000 - 3,561 MIPS at 1.2 GHz
  – 2010 - 147,600 MIPS at 3.3 GHz
• RAM Memory
  – 1990 – 640K conventional memory (256K extended memory recommended)
  – 2000 – 64MB memory
  – 2010 - 8-32GB (and more)
• Disk Capacity
  – 1990 – 20MB
  – 2000 - 1GB
  – 2010 – 1TB

• Disk Latency (speed of reads and writes) – not much improvement in last 7-10 years, currently around 70 – 80MB / sec

How long it will take to read 1TB of data?

- 1TB (at 80Mb / sec):
- – 1 disk - 3.4 hours
- – 10 disks - 20 min
- – 100 disks - 2 min
- – 1000 disks - 12 sec

What do we care about when we process data?

• Handle partial hardware failures without going down:

– If machine fails, we should be switch over to stand by machine

– If disk fails – use RAID or mirror disk

• Able to recover on major failures:

– Regular backups

– Logging

– Mirror database at different site

• Capability:

– Increase capacity without restarting the whole system

– More computing power should equal to faster processing

• Result consistency:

– Answer should be consistent (independent of something failing) and returned in reasonable amount of time

3. Intelligent Data Analysis

Intelligent Data Analysis (IDA) is one of the hot issues in the field of artificial intelligence and information. Intelligent data analysis reveals implicit, previously unknown and potentially valuable information or knowledge from large amounts of data. Intelligent data analysis is also a kind of decision support process. Based on artificial intelligence, machine learning, pattern recognition, statistics, database and visualization technology mainly, IDA automatically extracts useful information, necessary knowledge and interesting models from a lot of online data in order to help decision makers make the right choices. The process of IDA generally consists of the following three stages: (1) data preparation; (2) rule finding or data mining; (3) result validation and explanation. Data preparation involves selecting the required data from the relevant data source and integrating this into a data set to be used for data mining. Rule finding is working out rules contained in the data set by means of certain methods or algorithms. Result validation requires examining these rules, and result explanation is giving intuitive, reasonable and understandable descriptions using logical reasoning.

As the goal of intelligent data analysis is to extract useful knowledge, the process demands a combination of extraction, analysis, conversion, classification, organization, reasoning, and so on. It is challenging and fun working out how to choose appropriate methods to resolve the difficulties encountered in the process. Intelligent data analysis methods and tools, as well as the authenticity of obtained results pose us continued challenges.

4. Nature of Data

Big data is a term thrown around in a lot of articles, and for those who understand what big data means that is fine, but for those struggling to understand exactly what big data is, it

can get frustrating. There are several definitions of big data as it is frequently used as an all-encompassing term for everything from actual data sets to big data technology and big data analytics. However, this article will focus on the actual types of data that are contributing to the ever growing collection of data referred to as big data. Specifically we focus on the data created outside of an organization, which can be grouped into two broad categories: structured and unstructured.

Structured Data

1. Created

Created data is just that; data businesses purposely create, generally for market research. This may consist of customer surveys or focus groups. It also includes more modern methods of research, such as creating a loyalty program that collects consumer information or asking users to create an account and login while they are shopping online.

2. Provoked

A Forbes Article defined provoked data as, "Giving people the opportunity to express their views." Every time a customer rates a restaurant, an employee, a purchasing experience or a product they are creating provoked data. Rating sites, such as Yelp, also generate this type of data.

3. Transacted

Transactional data is also fairly self-explanatory. Businesses collect data on every transaction completed, whether the purchase is completed through an online shopping cart or in-store at the cash register. Businesses also collect data on the steps that lead to a purchase online. For example, a customer may click on a banner ad that leads them to the product pages which then spurs a purchase. As explained by the Forbes article, "Transacted data is a powerful way to understand exactly what was bought, where it was bought, and when. Matching this type of data with other information, such as weather, can yield even more insights.

4. Compiled

Compiled data is giant databases of data collected on every U.S. household. Companies like Acxiom collect information on things like credit scores, location, demographics, purchases and registered cars that marketing companies can then access for supplemental consumer data.

5. Experimental

Experimental data is created when businesses experiment with different marketing pieces and messages to see which are most effective with consumers. You can also look at experimental data as a combination of created and transactional data.

Unstructured Data

People in the business world are generally very familiar with the types of structured data mentioned above. However, unstructured is a little less familiar not because there's less of it, but before technologies like NoSQL and Hadoop came along, harnessing unstructured data wasn't possible. In fact, most data being created today is unstructured. Unstructured data, as the name suggests, lacks structure. It can't be gathered based on clicks, purchases or a barcode, so what is it exactly?

6. Captured

Captured data is created passively due to a person's behaviour. Every time someone enters a search term on Google that is data that can be captured for future benefit. The GPS info on our smart phones is another example of passive data that can be captured with big data technologies.

7. User-generated

User-generated data consists of all of the data individuals are putting on the Internet every day. From tweets, to Facebook posts, to comments on news stories, to videos put up on YouTube, individuals are creating a huge amount of data that businesses can use to better target consumers and get feedback on products.

Big data is made up of many different types of data. The seven listed above comprise types of external data included in the big data spectrum. There are, of course, many types of internal data that contribute to big data as well, but hopefully breaking down the types of data helps you to better see why combining all of this data into big data is so powerful for business.
Sources of Big Data:

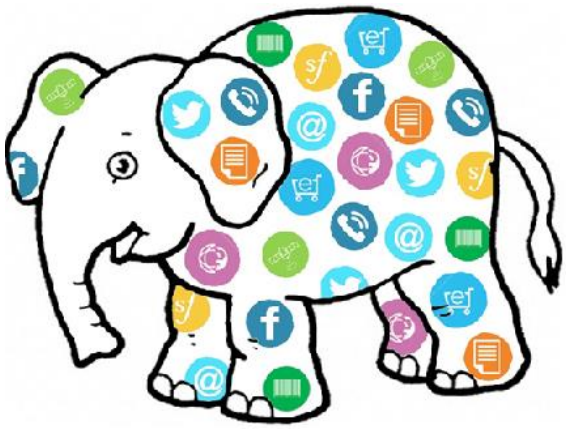


Fig. 1.4 Sources of Big Data

Classification of Types of Big Data

The following classification was developed by the Task Team on Big Data, in June 2013. Comments and feedback are welcome.

1. Social Networks (human-sourced information): this information is the record of human experiences, previously recorded in books and works of art, and later in photographs, audio and video. Human-sourced information is now almost entirely digitized and stored everywhere from personal computers to social networks. Data are loosely structured and often ungoverned.

- Social Networks: Facebook, Twitter, Tumblr etc.
- Blogs and comments
- Personal documents
- Pictures: Instagram, Flickr, Picasa etc.
- Videos: YouTube etc.
- Internet searches
- Mobile data content: text messages
- User-generated maps
- E-Mail

2. Traditional Business systems (process-mediated data): these processes record and monitor business events of interest, such as registering a customer, manufacturing a product, taking an order, etc. The process-mediated data thus collected is highly structured and includes transactions, reference tables and relationships, as well as the metadata that sets its context. Traditional business data is the vast majority of what IT managed and processed, in both operational and BI systems. Usually structured and stored in relational database systems.

- Data produced by Public Agencies

- Medical records
- Data produced by businesses
  - Commercial transactions
  - Banking/stock records
  - E-commerce
  - Credit cards

3. Internet of Things (machine-generated data): derived from the phenomenal growth in the number of sensors and machines used to measure and record the events and situations in the physical world. The output of these sensors is machine-generated data, and from simple sensor records to complex computer logs, it is well structured. As sensors proliferate and data volumes grow, it is becoming an increasingly important component of the information stored and processed by many businesses. Its well-structured nature is suitable for computer processing, but its size and speed is beyond traditional approaches.

- Data from sensors
  - Fixed sensors
    - ✓ Home automation
    - ✓ Weather/pollution sensors
    - ✓ Traffic sensors/webcam
    - ✓ Scientific sensors
    - ✓ Security/surveillance videos/images
- Mobile sensors (tracking)
  - Mobile phone location
  - Cars
  - Satellite images
- Data from computer systems
  - Logs
  - Web logs

5. Analytic Processes and Tool

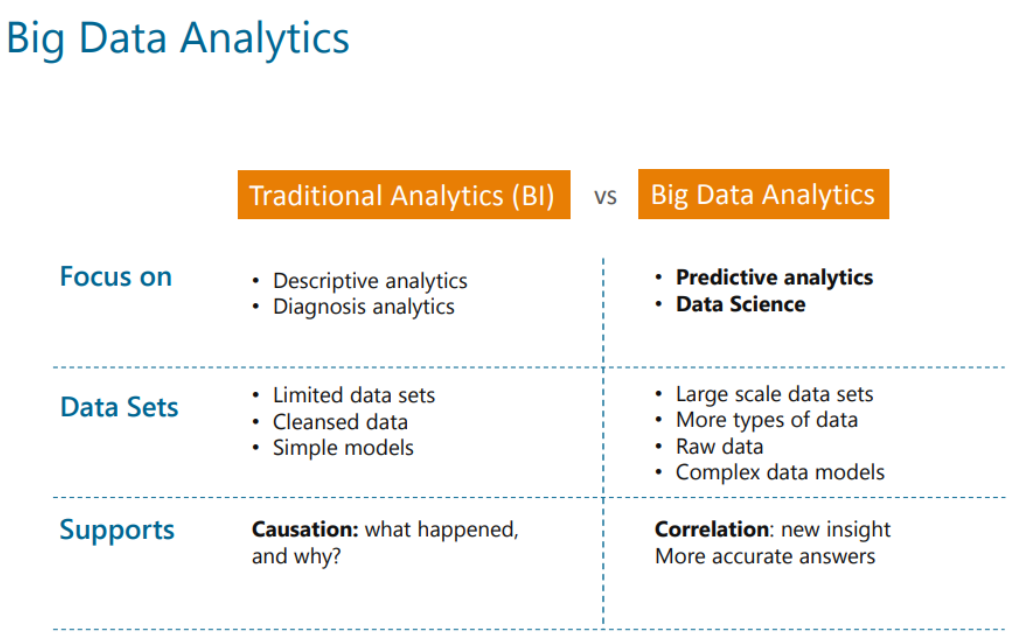


Fig. 1.5 Traditional Analytic Vs Big Data Analytics

Open Source Big Data Tools

Based on the popularity and usability we have listed the following ten open source tools as the best open source big data tools

1. Hadoop

Apache Hadoop is the most prominent and used tool in big data industry with its enormous capability of large-scale processing data. This is 100% open source framework and runs on commodity hardware in an existing data center. Furthermore, it can run on a cloud infrastructure. Hadoop consists of four parts:

- Hadoop Distributed File System: Commonly known as HDFS, it is a distributed file system compatible with very high scale bandwidth.
- MapReduce: A programming model for processing big data.
- YARN: It is a platform used for managing and scheduling Hadoop's resources in Hadoop infrastructure.
- Libraries: To help other modules to work with Hadoop.

2. Apache Spark

Apache Spark is the next hype in the industry among the big data tools. The key point of this open source big data tool is it fills the gaps of Apache Hadoop concerning data processing. Interestingly, Spark can handle both batch data and real-time data. As Spark does in-memory data processing, it processes data much faster than traditional disk processing. This is indeed a plus point for data analysts handling certain types of data to achieve the faster outcome.

Apache Spark is flexible to work with HDFS as well as with other data stores, for example with OpenStack Swift or Apache Cassandra. It's also quite easy to run Spark on a single local system to make development and testing easier. Spark Core is the heart of the project, and it facilitates many things like

- distributed task transmission
- scheduling
- I/O functionality

Spark is an alternative to Hadoop's MapReduce. Spark can run jobs 100 times faster than Hadoop's MapReduce.

3. Apache Storm

Apache Storm is a distributed real-time framework for reliably processing the unbounded data stream. The framework supports any programming language. The unique features of Apache Storm are:

- Massive scalability
- Fault-tolerance
- "fail fast, auto restart" approach
- The guaranteed process of every tuple
- Written in Clojure
- Runs on the JVM
- Supports direct acrylic graph(DAG) topology

- Supports multiple languages
- Supports protocols like JSON

Storm topologies can be considered similar to MapReduce job. However, in case of Storm, it is real-time stream data processing instead of batch data processing. Based on the topology configuration, Storm scheduler distributes the workloads to nodes. Storm can interoperate with Hadoop's HDFS through adapters if needed which is another point that makes it useful as an open source big data tool.

4. Cassandra

Apache Cassandra is a distributed type database to manage a large set of data across the servers. This is one of the best big data tools that mainly process structured data sets. It provides highly available service with no single point of failure. Additionally, it has certain capabilities which no other relational database and any NoSQL database can provide. These capabilities are:

- Continuous availability as a data source
- Linear scalable performance
- Simple operations
- Across the data centers easy distribution of data
- Cloud availability points
- Scalability
- Performance

Apache Cassandra architecture does not follow master-slave architecture, and all nodes play the same role. It can handle numerous concurrent users across data centers. Hence, adding a new node is no matter in the existing cluster even at its up time.

5. RapidMiner

RapidMiner is a software platform for data science activities and provides an integrated environment for:

- Preparing data
- Machine learning
- Text mining
- Predictive analytics
- Deep learning
- Application development
- Prototyping

This is one of the useful big data tools that support different steps of machine learning, such as:

- Data preparation
- Visualization
- Predictive analytics
- Model validation
- Optimization
- Statistical modelling

- Evaluation
- Deployment

RapidMiner follows a client/server model where the server could be located on-premise, or in a cloud infrastructure. It is written in Java and provides a GUI to design and execute workflows. It can provide 99% of an advanced analytical solution.

6. MongoDB

MongoDB is an open source NoSQL database which is cross-platform compatible with many built-in features. It is ideal for the business that needs fast and real-time data for instant decisions. It is ideal for the users who want data-driven experiences. It runs on MEAN software stack, NET applications and, Java platform.
Some notable features of MongoDB are:

- It can store any type of data like integer, string, array, object, Boolean, date etc.
- It provides flexibility in cloud-based infrastructure.
- It is flexible and easily partitions data across the servers in a cloud structure.
- MongoDB uses dynamic schemas. Hence, you can prepare data on the fly and quickly. This is another way of cost saving.

7. R Programming Tool

This is one of the widely used open source big data tools in big data industry for statistical analysis of data. The most positive part of this big data tool is – although used for statistical analysis, as a user you don't have to be a statistical expert. R has its own public library CRAN (Comprehensive R Archive Network) which consists of more than 9000 modules and algorithms for statistical analysis of data.

R can run on Windows and Linux server as well inside SQL server. It also supports Hadoop and Spark. Using R tool one can work on discrete data and try out a new analytical algorithm for analysis. It is a portable language. Hence, an R model built and tested on a local data source can be easily implemented in other servers or even against a Hadoop data lake.

8. Neo4j

Hadoop may not be a wise choice for all big data related problems. For example, when you need to deal with large volume of network data or graph related issue like social networking or demographic pattern, a graph database may be a perfect choice. Neo4j is one of the big data tools that is widely used graph database in big data industry. It follows the fundamental structure of graph database which is interconnected node-relationship of data. It maintains a key-value pattern in data storing.
Notable features of Neo4j are:

- It supports ACID transaction
- High availability
- Scalable and reliable
- Flexible as it does not need a schema or data type to store data
- It can integrate with other databases
- Supports query language for graphs which is commonly known as Cypher.

9. Apache SAMOA

Apache SAMOA is among well-known big data tools used for distributed streaming algorithms for big data mining. Not only data mining it is also used for other machine learning tasks such as:

- Classification
- Clustering
- Regression
- Programming abstractions for new algorithms

It runs on the top of distributed stream processing engines (DSPEs). Apache Samoa is a pluggable architecture and allows it to run on multiple DSPEs which include

- Apache Storm
- Apache S4
- Apache Samza
- Apache Flink

Due to below reasons, Samoa has got immense importance as the open source big data tool in the industry:

- You can program once and run it everywhere
- Its existing infrastructure is reusable. Hence, you can avoid deploying cycles.
- No system downtime
- No need for complex backup or update process

10. HPCC

High-Performance Computing Cluster (HPCC) is another among best big data tools. It is the competitor of Hadoop in big data market. It is one of the open source big data tools under the Apache 2.0 license. Some of the core features of HPCC are:

- Helps in parallel data processing
- Open Source distributed data computing platform
- Follows shared nothing architecture
- Runs on commodity hardware
- Comes with binary packages supported for Linux distributions
- Supports end-to-end big data workflow management
- The platform includes:

Thor: for batch-oriented data manipulation, their linking, and analytics
Roxie: for real-time data delivery and analytics

- Implicitly a parallel engine
- Maintains code and data encapsulation
- Extensible
- Highly optimized
- Helps to build graphical execution plans
- It compiles into C++ and native machine code

6. Analysis Vs Reporting

Reporting

Reporting is the first step of working with data when it comes to marketing. Reporting is really about the collection and organization of data points to start the storytelling process (more on story-telling later). Yet, to plant a seed, storytelling is really the core of reporting when it's done well. The data should come together into an organized visual format, allowing you to see changes against time or other relevant variables to show what has happened.

Good reporting should be organized with clear time parameters and have a clear visual presentation, so you can start to gain understanding of where things are as they pertain to your marketing efforts.

Analysis

Analysis is the step that should happen after the reports have been created. Analysis is the process of searching the reports and data to start to tell a more complex story. Analysis would look for the interactions between various data points to see how they influence each other. This search for correlation, or for the cause-and-effect relationships that exist inside of the data, is the basis of good analysis. To find, test, and confirm a true cause-and-effect relationship within the data would mark a successful analysis of the data.

Sometimes there's not enough data to truly do analysis in your existing data set. This would mean that to do true analysis you would have to gather data from outside of your data set. For example, if you were doing some analysis on your web data, you might have to gather reports on your social media channels or referral channels to see a bigger picture of the data and get an idea of how it's influenced by outside sources.

7. Modern Data Analytic Tools

The growing demand and importance of data analytics in the market have generated many openings worldwide. It becomes slightly tough to shortlist the top data analytics tools as the open source tools are more popular, user-friendly and performance oriented than the paid version. There are many open source tools which doesn't require much/any coding and manages to deliver better results than paid versions e.g. – R programming in data mining and Tableau public, Python in data visualization. Below is the list of top 10 of data analytics tools, both open source and paid version, based on their popularity, learning and performance.

1. R Programming

R is the leading analytics tool in the industry and widely used for statistics and data modelling. It can easily manipulate your data and present in different ways. It has exceeded SAS in many ways like capacity of data, performance and outcome. R compiles and runs on a wide variety of platforms viz. -UNIX, Windows and MacOS. It has 11,556 packages and allows you to browse the packages by categories. R also provides tools to automatically install all packages as per user requirement, which can also be well assembled with Big data.

2. Tableau Public:

Tableau Public is a free software that connects any data source be it corporate Data Warehouse, Microsoft Excel or web-based data, and creates data visualizations, maps, dashboards etc. with real-time updates presenting on web. They can also be shared through social media or with the client. It allows the access to download the file in different formats. If you want to see the power of tableau, then we must have very good data source. Tableau's Big

Data capabilities makes them important and one can analyze and visualize data better than any other data visualization software in the market.

3. Python

Python is an object-oriented scripting language which is easy to read, write, maintain and is a free open source tool. It was developed by Guido Van Rossum in late 1980's which supports both functional and structured programming methods.Python is easy to learn as it is very similar to JavaScript, Ruby, and PHP. Also, Python has very good machine learning libraries viz. Scikitlearn, Theano, Tensorflow and Keras. Another important feature of Python is that it can be assembled on any platform like SQL server, a MongoDB database or JSON. Python can also handle text data very well.

4. SAS:

Sas is a programming environment and language for data manipulation and a leader in analytics, developed by the SAS Institute in 1966 and further developed in 1980's and 1990's. SAS is easily accessible, manageable and can analyze data from any sources. SAS introduced a large set of products in 2011 for customer intelligence and numerous SAS modules for web, social media and marketing analytics that is widely used for profiling customers and prospects. It can also predict their behaviours, manage, and optimize communications.

5. Apache Spark

The University of California, Berkeley's AMP Lab, developed Apache in 2009. Apache Spark is a fast large-scale data processing engine and executes applications in Hadoop clusters 100 times faster in memory and 10 times faster on disk. Spark is built on data science and its concept makes data science effortless. Spark is also popular for data pipelines and machine learning models development.

Spark also includes a library – MLlib, that provides a progressive set of machine algorithms for repetitive data science techniques like Classification, Regression, Collaborative Filtering, Clustering, etc.

6. Excel

Excel is a basic, popular and widely used analytical tool almost in all industries. Whether you are an expert in Sas, R or Tableau, you will still need to use Excel. Excel becomes important when there is a requirement of analytics on the client's internal data. It analyzes the complex task that summarizes the data with a preview of pivot tables that helps in filtering the data as per client requirement. Excel has the advance business analytics option which helps in modelling capabilities which have prebuilt options like automatic relationship detection, a creation of DAX (Data Analysis Expressions) measures and time grouping.

7. RapidMiner:

RapidMiner is a powerful integrated data science platform developed by the same company that performs predictive analysis and other advanced analytics like data mining, text analytics, machine learning and visual analytics without any programming. RapidMiner can incorporate with any data source types, including Access, Excel, Microsoft SQL, Tera data, Oracle, Sybase, IBM DB2, Ingres, MySQL, IBM SPSS, Dbase etc. The tool is very powerful

that can generate analytics based on real-life data transformation settings, i.e. you can control the formats and data sets for predictive analysis.

8. KNIME

KNIME Developed in January 2004 by a team of software engineers at University of Konstanz. KNIME is leading open source, reporting, and integrated analytics tools that allow you to analyze and model the data through visual programming, it integrates various components for data mining and machine learning via its modular data-pipelining concept.

9. QlikView

QlikView has many unique features like patented technology and has in-memory data processing, which executes the result very fast to the end users and stores the data in the report itself. Data association in QlikView is automatically maintained and can be compressed to almost 10% from its original size. Data relationship is visualized using colours – a specific colour is given to related data and another colour for non-related data.

10. Splunk:

Splunk is a tool that analyzes and searches the machine-generated data. Splunk pulls all text-based log data and provides a simple way to search through it, a user can pull in all kind of data, and perform all sort of interesting statistical analysis on it, and present it in different formats.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

School of Computing

Department of Computer Science and Engineering

UNIT II - Big Data – SBS1608

SYLLABUS

Introduction – distributed file system – Big Data and its importance, Four Vs, Drivers for Big data, Big data analytics, Big data applications. Algorithms using map reduce.

1. Distributed File System

In Big Data,

- deal with multiple clusters (computers) often.

- One of the main advantages of Big Data
  - is that it goes beyond the capabilities of one single super powerful server with extremely high computing power.
- The whole idea of Big Data
  - is to distribute data across multiple clusters
  - and to make use of computing power of each cluster (node) to process information.
- Distributed file system
  - is a system that can handle accessing data across multiple clusters (nodes)

Two main purposes of using files:

1. Permanent storage of information on a secondary storage media.
2. Sharing of information between applications.

A file system is a process that manages how and where data on storage disk, typically a hard disk drive (HDD),

- is stored, accessed and managed.
- is a logical disk component that manages a disk's internal operations as it relates to a computer and
- is abstract to a human user

- In short,
  - controls how data is stored and retrieved

File Systems – Types

FAT File System

- File Allocation Table
- is used by the operating system
- to locate files on a disk
- A file may be divided into many sections and scattered around the disk due to fragmentation
- FAT keeps track of all pieces of a file
- FAT does not support local and folder security
- A user logged on a computer locally has full access to the files and folders in FAT partitions of the computer.

FAT32 File System

- File Allocation Table
- FAT32 is an advanced version of FAT file system.
- It can be used on drives from 512 MB to 2TB in size.
- One of the most important features of FAT and FAT32

- is that they offer compatibility with operating systems other than Windows 2000 also.

NTFS File System

- New Technology File System
- Windows 2000 professional fully supports NTFS
- NTFS provides file and folder security
- Files and folders are safer than FAT
- Security is maintained by assigning NTFS permissions to files and folders.
- Security is maintained at the local level and the network level.
- The permissions can be assigned to individual files and folders.

Distributed File System

- is a method of storing and accessing files
- In a distributed file system,
  - one or more central servers store files that can be accessed,
  - with proper authorization rights,
  - by any number of remote clients in the network
- Purpose is to allow users of physically distributed computers to share data and storage resources by using a common file system.
- A typical configuration for a DFS is a collection of workstations and mainframes connected by a local area network (LAN).

Distributed file system works as follows:

- Distribution:
  - Distribute blocks of data sets across multiple nodes.
  - Each node has its own computing power; which gives the ability of DFS to parallel processing data blocks.
- Replication:
  - Distributed file system will also replicate data blocks on different clusters by copy the same pieces of information into multiple clusters on different racks.
- This will help to achieve the following:
  - Fault Tolerance: recover data block in case of cluster failure or Rack failure.
  - High Concurrency: avail same piece of data to be processed by multiple clients at the same time.

Distributed file system - advantages:

- Scalability: can scale up the infrastructure by adding more racks or clusters to your system.
- Fault Tolerance: Data replication will help to achieve fault tolerance in the following cases:
  - Cluster is down
  - Rack*(*collection of servers*)* is down
  - Rack is disconnected from the network.
  - Job failure or restart
- High Concurrency: utilize the compute power of each node to handle multiple client requests (in a parallel way) at the same time.

Distributed File Systems – Features

1. Transparency

Structure transparency - Clients should not know the number or locations of file servers and the storage devices.

Access transparency - Both local and remote files should be accessible in the same way. The file system should automatically locate an accessed file and transport it to the clients' site.

Naming transparency - The name of the file should give no hint as to the location of the file. The name of the file must not be changed when moving from one node to another.

Replication transparency - If a file is replicated on multiple nodes, both the existence of multiple copies and their locations should be hidden from the clients.

2. User mobility - Automatically bring the users environment (e.g. users home directory) to the node where the user logs in.

3. Performance - is measured as the average amount of time needed to satisfy client requests. This time includes CPU time + time for accessing secondary storage + network access time.

4. Simplicity and ease of use - User interface to the file system be simple and number of commands should be as small as possible.

5. Scalability - Growth of nodes and users should not seriously disrupt service.

6. High availability - A distributed file system should continue to function in the face of partial failures such as a link failure, a node failure, or a storage device crash.

7. High reliability - Probability of loss of stored data should be minimized. System should automatically generate backup copies of critical files.

8. Data integrity - Concurrent access requests from multiple users who are competing to access the file must be properly synchronized by the use of some form of concurrency control mechanism. Atomic transactions can also be provided.

9. Security - Users should be confident of the privacy of their data.

10. Heterogeneity - There should be easy access to shared data on diverse platforms (e.g. UNIX workstation, Wintel platform etc).

File Replication

- High availability is a desirable feature of a good distributed file system and file replication is the primary mechanism for improving file availability.
- A replicated file is a file that has multiple copies, with each file on a separate file server.

File Replication – Advantages

1. Increased Availability - Alternate copies of a replicated data can be used when the primary copy is unavailable.

2. Increased Reliability - Due to the presence of redundant data files in the system, recovery from catastrophic failures (e.g. hard drive crash) becomes possible.

3.Improved response time - It enables data to be accessed either locally or from a node to which access time is lower than the primary copy access time.

4. Reduced network traffic - If a file's replica is available with a file server that resides on a client's node, the client's access request can be serviced locally, resulting in reduced network traffic.

5. Improved system throughput - Several clients request for access to a file can be serviced in parallel by different servers, resulting in improved system throughput.

6. Better scalability - Multiple file servers are available to service client requests since due to file replication. This improves scalability.

Replication Transparency

- Replication of files should be transparent to the users so that multiple copies of a replicated file appear as a single logical file to its users.

This calls for the assignment of a single identifier/name to all replicas of a file.

- In addition, replication control should be transparent, i.e., the number and locations of replicas of a replicated file should be hidden from the user.

Thus replication control must be handled automatically in a user-transparent manner.

Multicopy Update Problem

Maintaining consistency among copies when a replicated file is updated is a major design issue of a distributed file system that supports file replication.

1. Read-only replication

- In this case the update problem does not arise. This method is too restrictive.

2. Read-Any-Write-All Protocol

- A read operation on a replicated file is performed by reading any copy of the file and a write operation by writing to all copies of the file.
- Before updating any copy, all copies need to be locked, then they are updated, and finally the locks are released to complete the write.

Disadvantage: A write operation cannot be performed if any of the servers having a copy of the replicated file is down at the time of the write operation.

3. Available-Copies Protocol

- A read operation on a replicated file is performed by reading any copy of the file and a write operation by writing to all available copies of the file.
- Thus if a file server with a replica is down, its copy is not updated.
- When the server recovers after a failure, it brings itself up to date by copying from other servers before accepting any user request.

4. Primary-Copy Protocol

- For each replicated file, one copy is designated as the primary copy and all the others are secondary copies.
- Read operations can be performed using any copy, primary or secondary.
- But write operations are performed only on the primary copy.
- Each server having a secondary copy updates its copy either by receiving notification of changes from the server having the primary copy or by requesting the updated copy from it.

Distributed File System

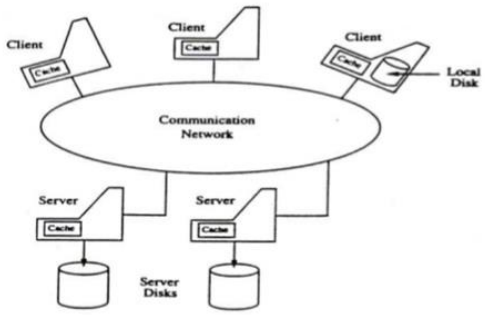


Fig. 2.1 Distributed File System

Distributed File System – Why?

- because they make it easier to distribute documents to multiple clients
- they provide a centralized storage system so that client machines are not using their resources to store files
- It is important – because whenever we work with big data sets
- system will not able to run the data sets
- Because of RAM, Hard Disc etc.
- So, there is a need of distributed file system where - can distribute the task in some other machine and process the job

2. Big Data and its importance

Big data is high-volume, high-velocity and high-variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making. Big Data is a broad term for data sets so large or complex that they are difficult to process using traditional data processing applications. Challenges include analysis, capture, data curation, search, sharing, storage, transfer, visualization, and information privacy.

- Big Data helps the organizations to create new growth opportunities and entirely new categories of companies that can combine and analyze industry data.
- These companies have ample information about the products and services, buyers and suppliers, consumer preferences that can be captured and analyzed
- The importance of big data does not revolve around how much data a company has but how a company utilises the collected data.
- Every company uses data in its own way; the more efficiently a company uses its data, the more potential it has to grow.
- The company can take data from any source and analyze it to find answers which will enable:
- Cost Savings: when large amounts of data are to be stored, big data tools help in identifying more efficient ways of doing business.
- Time Reductions: The high speed of tools can easily identify new sources of data which helps businesses analyzing data immediately and make quick decisions based on the learning.
- Understand the market conditions: By analyzing big data you can get a better understanding of current market conditions. For example, by analyzing customers'

purchasing behaviors, a company can find out the products that are sold the most and produce products according to this trend. By this, it can get ahead of its competitors.

- Control online reputation: Big data tools can do sentiment analysis. Therefore, you can get feedback about who is saying what about your company. If you want to monitor and improve the online presence of your business, then, big data tools can help in all this.

- Using Big Data Analytics to Boost Customer Acquisition and Retention: The customer is the most important asset any business depends on. There is no single business that can claim success without first having to establish a solid customer base. The use of big data allows businesses to observe various customer related patterns and trends. Observing customer behavior is important to trigger loyalty.

- Using Big Data Analytics to Solve Advertisers Problem and Offer Marketing Insights: Big data analytics can help change all business operations. This includes the ability to match customer expectation, changing company's product line and of course ensuring that the marketing campaigns are powerful.
- Big Data Analytics as a Driver of Innovations and Product Development: Another huge advantage of big data is the ability to help companies innovate and redevelop their products.

Big Data – The Opportunity is Now

In the past, technology platforms were built to address either structured OR unstructured data. The value and means of unifying and/or integrating these data types had yet to be realized, and the computing environments to efficiently process high volumes of disparate data were not yet commercially available.

Large content repositories house unstructured data such as documents, and companies often store a great deal of structured information corporate systems like Oracle, SAP and Net Suite and others. Today's organizations, however, are utilizing, sharing and storing more information in varying formats, including:

- e-mail and Instant Messaging
- Collaborative Intranets and Extranets
- Public websites, wikis, and blogs
- Social media channels
- Video and audio files
- Data from industrial sensors, wearables and other monitoring devices

This unstructured data adds up to as much as 85% of the information that businesses store. Regardless of the size of your business or the industry you are in, you have Big Data. The ability to extract high value from this data to enable innovation and competitive gain is the purpose of Big Data analytics. Conducting analytics on large sets of data, business users and executives are able to see patterns and trends in performance, new relationships between data sets and potentially new sources of revenue.

Big Data for All

Big Data initiatives are rated as "extremely important" or "important" to 93% of companies over $250M, The opportunity to amass and capitalize on Big Data is available to any organization, large or small. The data likely exists already, distributed amongst a collection of internal repositories and files shares and/or external data sources. Storing and managing large

amounts of data has become more affordable and manageable, enabling organizations to take full advantage of their assets. Leveraging a Big Data analytics solution can help you unlock the strategic value of this information by allowing you to:

- Understand where, when and why your customers buy
- Protect your client base with improved loyalty programs
- Seize cross selling and up selling opportunities
- Provide targeted promotional information to your prospects and existing clients
- Optimize Workforce planning and operations
- Improve inefficiencies in your supply chain
- Predict market trends and future needs
- Become more innovative and competitive
- Discover new sources of revenue

The Big Data Challenge and Opportunity

Big Data analytics provides organizations an opportunity for disruptive change and growth. In most cases, however, the data sets are too large, move too fast or are too complex for the traditional computing environment, which creates a significant challenge. The technologies are available; however, an investment of time, money and resources will be necessary to fully implement a Big Data solution. Is it worth it? The options are limited—invest in the platform, technologies and expertise to leverage your data, or continue along the path of the status quo. Enterprise content and data specialists, such as General Networks, can help you to define and quantify your Big Data goals and objectives.

- Artificial intelligence (AI) and big data penetrates manufacturing floors in the form of robots, sensors, and machinery that make production systems faster and more efficient.
- The IoT and big data integrates systems and has many analytic applications.
  - One application is data analysis in manufacturing, which can be utilized
  - to improve efficiency, to reduce errors and identify faulty products on the production line, before they can be released to business partners or customers.
  - Another strong case for big data analytics in smart manufacturing is their application in predictive maintenance processes.
  - Systems put in place can send out repair alerts and preventive maintenance reminders to stop equipment breakdowns before they occur.
  - Integration of sensors for condition-based monitoring can also be integrated to monitor equipment performance and health in real time, improving overall equipment effectiveness on the factory floor.
- Sales and operations planning processes can begin to be automated by using big data analytics to review historical loads, customer data, and changes to major projects which will help companies optimize their plant loading.
- Big Data application in key industries
  - In rapidly evolving industries, big data enables businesses to solve today's manufacturing challenges and to gain a competitive edge.
  - With big data and analytics, companies have got a chance to make better real-time decisions about asset usage and operations scheduling.
  - In such a way, this data helps chemical and oil/gas manufacturers optimize production levels, reduce waste, improve accuracy, and manage energy consumption.

  - Video surveillance technology is another big manifestation of the industrial data presence.
  - Video analytics can help analyze the video streams of those cameras to provide real-time alerting, as well as operational insights for maintenance purposes.
- Career opportunities in Big Data are numerous and identifying which is more suitable for any given individual depends on interests, career path, skills and abilities.
- Some well-known Big Data career paths are as follows:
    - Database Administrator
    - Database Developer
    - Data Analyst
    - Data Scientist
    - Big Data Engineer
    - Data Modeler
- Whether a fresher or an experienced professional looking to enhance the career prospects, learn Big Data tools, trends and techniques from industry experts for significant career growth in Big Data field.

"Data is useless without the skill to analyze it."

- From a career point of view, there are so many options available, in terms of domain as well as nature of job.
    - Big Data Analytics Business Consultant
    - Big Data Analytics Architect
    - Big Data Engineer
    - Big Data Solution Architect
    - Big Data Analyst
    - Analytics Associate
    - Business Intelligence and Analytics Consultant
    - Metrics and Analytics Specialist

Big Data Analytics career is deep and one can choose from the 3 types of data analytics -

- Prescriptive Analytics
- Predictive Analytics
- Descriptive Analytics.

- Insufficient understanding and acceptance of big data
  - companies fail to know even the basics: what big data actually is, what its benefits are, what infrastructure is needed, etc.
  - Without a clear understanding, a big data adoption project risks in failure.
  - Companies may waste lots of time and resources on things they don't even know how to use
  - organize numerous trainings and workshops
- Confusing variety of big data technologies
  - can be easy to get lost in the variety of big data technologies now available on the market.
  - Finding the answers can be tricky.
  - And it's even easier to choose a wrong one, if exploring the ocean of technological opportunities without a clear view of what is needed.
  - trying to seek professional help would be the right way to go.
- Paying loads of money

  - big data adoption - lots of expenses.
  - have to mind the costs of new hardware, new hires (administrators and developers), electricity and so on.
  - although the needed frameworks are open-source, still need to pay for the development, setup, configuration and maintenance of new software.
  - If cloud-based big data solution, still need to hire staff and pay for *cloud* services, big data solution development as well as setup and maintenance of needed frameworks.
  - to solve - properly analyzing the needs and choosing a corresponding course of action.
- Complexity of managing data quality
  - data from diverse sources
  - unreliable data
  - there are techniques dedicated to cleansing data.
  - Also big data needs to have a proper model
- Dangerous big data security holes
  - big data adoption projects put security off till later stages
  - the precaution against big data security challenges is putting security first.
  - important at the stage of designing the solution's architecture.
  - If the big data security is not considered from the very start, it'll bite when least expected.
- Tricky process of converting big data into valuable insights
  - an example: the super-cool big data analytics looks at what item pairs people buy (say, a needle and thread) solely based on your historical data about customer behavior. But in the store, if required products are not available - as a result, may lose revenue and maybe some loyal customers too.
  - Also, - shop has both items and even offers a 15% discount if both are bought together.
  - idea here is that - to create a proper system of factors and data sources, whose analysis will bring the needed insights, and ensure that nothing falls out of scope.
  - Such a system should often include external sources, even if it may be difficult to obtain and analyze external data.
- Troubles of upscaling
  - ability to grow.
  - lies in the complexity of scaling up so that the system's performance doesn't decline and can stay within budget.
  - the first and foremost precaution for challenges like this is a decent architecture of the big data solution.
  - Another highly important thing to do is designing your big data algorithms while keeping future upscaling in mind.
  - But besides that, you also need to plan for the system's maintenance and support so that any changes related to data growth are properly attended to.
  - And on top of that, holding systematic performance audits can help identify weak spots and address on time.

- Most of the reviewed challenges can be dealt with – if the big data solution has a decent, well-organized and thought-through architecture.
- Also, companies should:
  - Hold workshops for employees to ensure big data adoption.
  - Carefully select technology stack.
  - Mind costs and plan for future upscaling.
  - Remember that data isn't 100% accurate but still manage its quality.
  - Dig deep and wide for actionable insights.
  - Never neglect big data security.

3. The 4 V's of Big Data

The general consensus of the day is that there are specific attributes that define big data. In most big data circles, these are called the four V's: volume, variety, velocity, and veracity.

Volume

The main characteristic that makes data "big" is the sheer volume. It makes no sense to focus on minimum storage units because the total amount of information is growing exponentially every year. In 2010, Thomson Reuters estimated in its annual report that it believed the world was "awash with over 800 exabytes of data and growing." For that same year, EMC, a hardware company that makes data storage devices, thought it was closer to 900 exabytes and would grow by 50 percent every year. No one really knows how much new data is being generated, but the amount of information being collected is huge.

Variety

Variety is one the most interesting developments in technology as more and more information is digitized. Traditional data types (structured data) include things on a bank statement like date, amount, and time. These are things that fit neatly in a relational database.

Structured data is augmented by unstructured data, which is where things like Twitter feeds, audio files, MRI images, web pages, web logs are put — anything that can be captured and stored but doesn't have a meta model (a set of rules to frame a concept or idea — it defines a class of information and how to express it) that neatly defines it.

Unstructured data is a fundamental concept in big data. The best way to understand unstructured data is by comparing it to structured data. Think of structured data as data that is well defined in a set of rules. For example, money will always be numbers and have at least two decimal points; names are expressed as text; and dates follow a specific pattern.

With unstructured data, on the other hand, there are no rules. A picture, a voice recording, a tweet — they all can be different but express ideas and thoughts based on human understanding. One of the goals of big data is to use technology to take this unstructured data and make sense of it.

Veracity

Veracity refers to the trustworthiness of the data. Can the manager rely on the fact that the data is representative? Every good manager knows that there are inherent discrepancies in all the data collected.

Velocity

Velocity is the frequency of incoming data that needs to be processed. Think about how many SMS messages, Facebook status updates, or credit card swipes are being sent on a particular telecom carrier every minute of every day, and you'll have a good appreciation of velocity. A streaming application like Amazon Web Services Kinesis is an example of an application that handles the velocity of data.

Value

It may seem painfully obvious to some, but a real objective is critical to this mashup of the four V's. Will the insights you gather from analysis create a new product line, a cross-sell opportunity, or a cost-cutting measure? Or will your data analysis lead to the discovery of a critical causal effect that result in a cure to a disease?

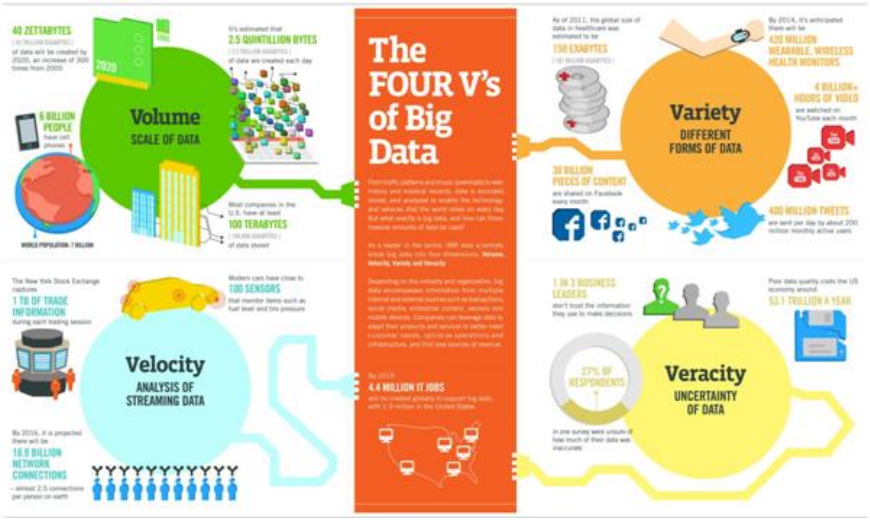


Fig. 2.2 Four V's of Big Data

4. Drivers for Big Data

Big Data emerged in the last decade from a combination of business needs and technology innovations. A number of companies that have Big Data at the core of their strategy have become very successful at the beginning of the 21st century. Famous examples include Apple, Amazon, Facebook and Netflix.

A number of business drivers are at the core of this success and explain why Big Data has quickly risen to become one of the most coveted topics in the industry. Six main business drivers can be identified:

1. The digitization of society;
2. The plummeting of technology costs;
3. Connectivity through cloud computing;
4. Increased knowledge about data science;
5. Social media applications;
6. The upcoming Internet-of-Things (IoT).

A high-level overview of each of these business drivers are explored here. Each of these adds to the competitive advantage of enterprises by creating new revenue streams by reducing the operational costs.

1. The digitization of society

Big Data is largely consumer driven and consumer oriented. Most of the data in the world is generated by consumers, who are nowadays 'always-on'. Most people now spend 4-6 hours per day consuming and generating data through a variety of devices and (social) applications. With every click, swipe or message, new data is created in a database somewhere around the world. Because everyone now has a smart phone in their pocket, the data creation sums to incomprehensible amounts. Some studies estimate that 60% of data was generated within the last two years, which is a good indication of the rate with which society has digitized.

2. The plummeting of technology costs

Technology related to collecting and processing massive quantities of diverse (high variety) data has become increasingly more affordable. The costs of data storage and processors keep declining, making it possible for small businesses and individuals to become involved with Big Data. For storage capacity, the often-cited Moore's Law still holds that the storage density (and therefore capacity) still doubles every two years. The plummeting of technology costs has been depicted in the figure below.

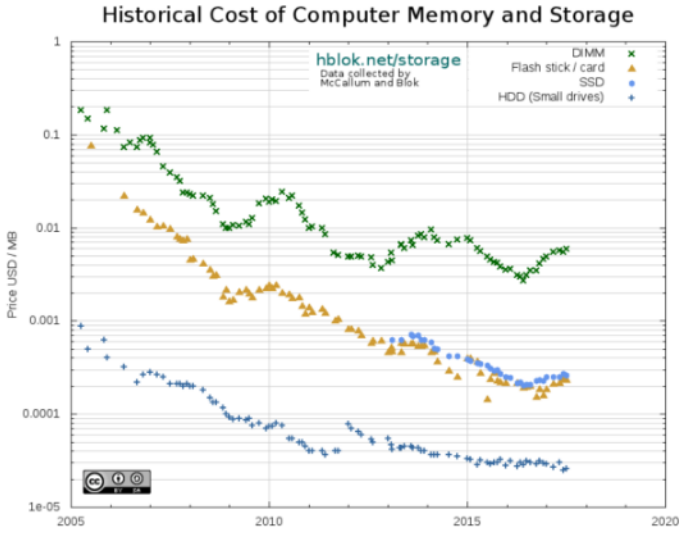


Fig. 2.3 Historical Costs of Computer Memory

Besides the plummeting of the storage costs, a second key contributing factor to the affordability of Big Data has been the development of open source Big Data software frameworks. The most popular software framework (nowadays considered the standard for Big Data) is Apache Hadoop for distributed storage and processing. Due to the high availability of these software frameworks in open sources, it has become increasingly inexpensive to start Big Data projects in organizations.

3. Connectivity through cloud computing

Cloud computing environments (where data is remotely stored in distributed storage systems) have made it possible to quickly scale up or scale down IT infrastructure and facilitate

a pay-as-you-go model. This means that organizations that want to process massive quantities of data (and thus have large storage and processing requirements) do not have to invest in large quantities of IT infrastructure. Instead, they can license the storage and processing capacity they need and only pay for the amounts they actually used. As a result, most of Big Data solutions leverage the possibilities of cloud computing to deliver their solutions to enterprises.

4. Increased knowledge about data science

In the last decade, the term data science and data scientist have become tremendously popular. In October 2012, Harvard Business Review called the data scientist "sexiest job of the 21st century" and many other publications have featured this new job role in recent years. The demand for data scientist (and similar job titles) has increased tremendously and many people have actively become engaged in the domain of data science.

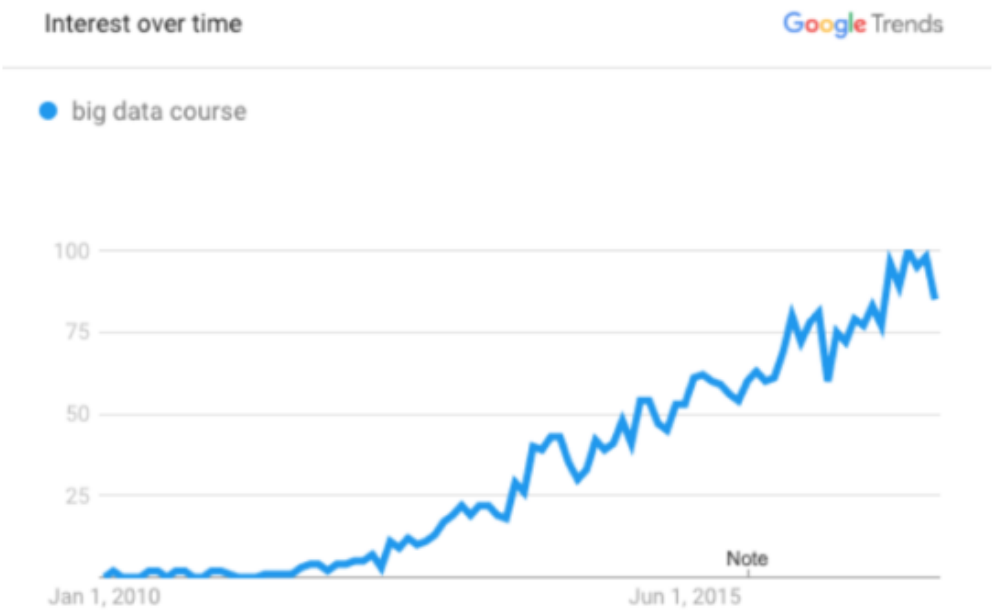


Fig. 2.4 Increased knowledge about data science

As a result, the knowledge and education about data science has greatly professionalized and more information becomes available every day. While statistics and data analysis mostly remained an academic field previously, it is quickly becoming a popular subject among students and the working population.

5. Social media applications

Everyone understands the impact that social media has on daily life. However, in the study of Big Data, social media plays a role of paramount importance. Not only because of the sheer volume of data that is produced everyday through platforms such as Twitter, Facebook, LinkedIn and Instagram, but also because social media provides nearly real-time data about human behaviour.

Social media data provides insights into the behaviours, preferences and opinions of 'the public' on a scale that has never been known before. Due to this, it is immensely valuable to anyone who is able to derive meaning from these large quantities of data. Social media data can be used to identify customer preferences for product development, target new customers

for future purchases, or even target potential voters in elections. Social media data might even be considered one of the most important business drivers of Big Data.

6. The upcoming internet of things (IoT)

The Internet of things (IoT) is the network of physical devices, vehicles, home appliances and other items embedded with electronics, software, sensors, actuators, and network connectivity which enable these objects to connect and exchange data. It is increasingly gaining popularity as consumer goods providers start including 'smart' sensors in household appliances. Whereas the average household in 2010 had around 10 devices that connected to the internet, this number is expected to rise to 50 per household by 2020. Examples of these devices include thermostats, smoke detectors, televisions, audio systems and even smart refrigerators.

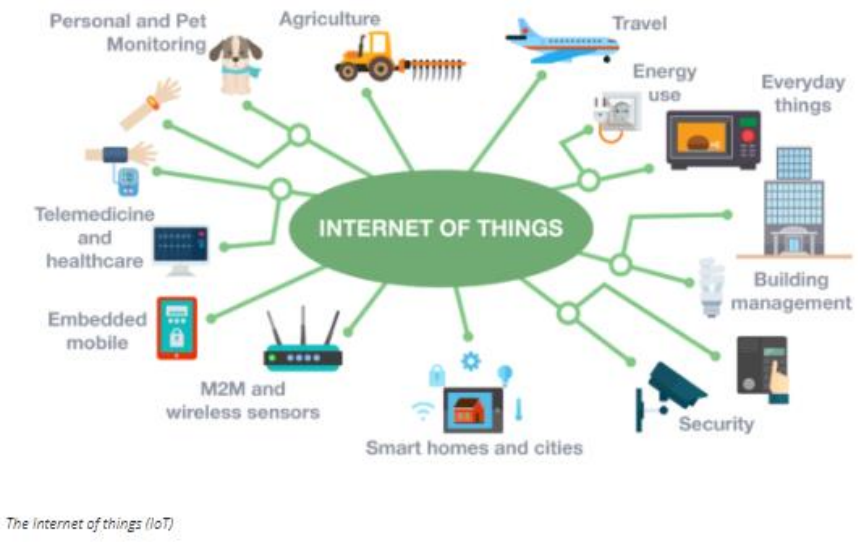


Fig. 2.5 The Internet of Things (IoT)

5. Big data Analytics

Big data analytics is the use of advanced analytic techniques against very large, diverse data sets that include structured, semi-structured and unstructured data, from different sources, and in different sizes from terabytes to zettabytes.

Big data is a term applied to data sets whose size or type is beyond the ability of traditional relational databases to capture, manage and process the data with low latency. Big data has one or more of the following characteristics: high volume, high velocity or high variety. Artificial intelligence (AI), mobile, social and the Internet of Things (IoT) are driving data complexity through new forms and sources of data. For example, big data comes from sensors, devices, video/audio, networks, log files, transactional applications, web, and social media — much of it generated in real time and at a very large scale.

Analysis of big data allows analysts, researchers and business users to make better and faster decisions using data that was previously inaccessible or unusable. Businesses can use advanced analytics techniques such as text analytics, machine learning, predictive analytics, data mining, statistics and natural language processing to gain new insights from previously untapped data sources independently or together with existing enterprise data.

Fig. 2.6 Big Data Analytics

6. Big Data Applications

Primary goal of Big Data applications

- to help companies make more informative business decisions by analyzing large volumes of data

- It could include web server logs, Internet click stream data, social media content and activity reports, text from customer emails, mobile phone call details and machine data captured by multiple sensors.
- Organisations from different domain are investing in Big Data applications,
  - for examining large data sets to uncover all hidden patterns, unknown correlations, market trends, customer preferences and other useful business information.
- Healthcare
- Banking
- Ecommerce
- Education
- Agriculture
- Media & Entertainment
- Digital Marketing
- Social Media Sector
- Airline Industry
- National Security
- Government Sector
- Tourism
- Restaurants
- Fast-food Industry
- Casino Business
- National Security
- Disaster Management
- Customer Oriented Service
- Cloud Computation
- Telecommunication

Big Data Applications – Healthcare

- Role of big data in medical was not mentionable.
- But data science is dominating to improve healthcare nowadays.
- Data science not only introduced to identify treatment but also improved the process of rendering healthcare.
- Big data has a great impact on reducing waste of money and time.
- Alongside this, governments are using big data to develop new infrastructures and emergency medical services.

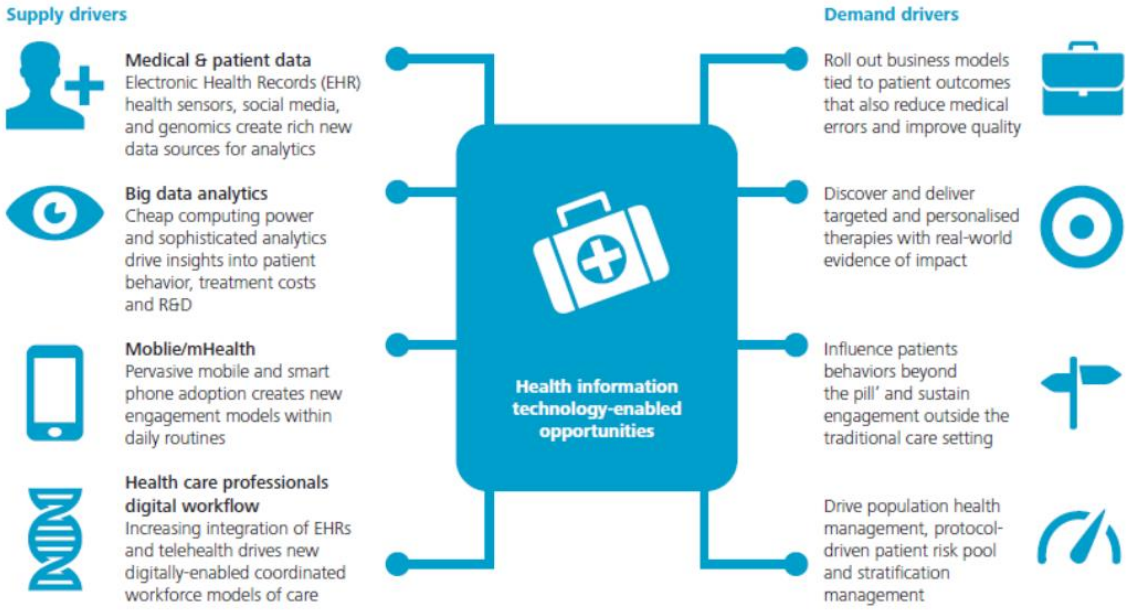


Fig. 2.7 Big Data in Healthcare

Interpretation

- Enables shift managers to predict the required doctors at specific times and introduced EHR (Electronic Healthcare Records) to keep track of patient's records.
- Using wearable digital devices, big data can monitor patients and send reports to the associated doctors.
- Big data can evaluate symptoms and identify any many diseases at the early stages.
- Can keep the sensitive records secured and store huge amount of data efficiently.
- Availability of medical database has also played a major role.
- Diseases like AIDS and Cancer can cause life easily.
- Big data can save lives by analyzing the behavior and health condition of the patients.
- Big data applications can also foretell the location where there is a chance of dengue or malaria spreading.

Big Data Applications – Banking Sector

- Valuable properties are kept in the bank for ensuring security.
- But a bank has to go through a lot of strategies to keep your wealth safe and well maintained.
- In each bank, big data is being used for many years.
- From cash collection to financial management, big data is making banks more efficient in every sector.

- Big data applications in the banking sector have lessened customer's hassle and generated revenue for the banks.

Fig. 2.8 Big Data in Banking

Interpretation

- Using clustering techniques banks can take important decisions.
- It can identify the new branch locations where the demand is high.
- Association rule is applied in banking sectors to predict the amount of cash needed to be present in a branch at the specific time of every year.
- Banking platforms are digital now, and all operations can be done from home, which is a blessing of data science.
- Machine learning and AI are being used by many banks to detect fraudulent activities and report to the related personnel.
- Data science has made it easy to handle, store and analyze this massive amount of data and ensure its security as well for the banks.

Big Data Applications – Ecommerce

- Ecommerce is one of the legit ways through which people can earn online.
- Basically small to large businesses compete with each other in the eCommerce industry.
- Ecommerce not only enjoy the benefits of operating online but also faces many challenges to achieve the business objectives.
- Big data in eCommerce can provide competitive advantages by providing insights and analytical reports.

Fig. 2.9 Big Data in eCommerce

Interpretation

- Can collect data and customer requirements even before the official operation has started.
- Creates a high performing marketing model and set a startup apart from the existing and become successful.
- Ecommerce owners can identify the most viewed products and the pages that appeared the maximum number of time.
- Evaluates customers' behavior and suggests similar products.
- It increases the number of sales and generates revenue.
- If any product is added to cart but was not ultimately bought by a customer, big data can automatically send a promotional offer to that particular customer.
- Big data applications can generate a sorted report depending on the visitor's age, gender, location, and so on.

Big Data Applications – Agriculture

- In agriculture, Big data is playing an influential role to enhance the performance of the firms.
- Goal
  - to minimize the firm's loss and
  - increase the generation of necessary food grains for the citizens of the nations.
- Data science has helped a lot to introduce digital and futuristic methods to the existing agricultural traditions.
- Uses of big data make us able to meet the required amount of yearly production and remove the necessity of importing goods as well.

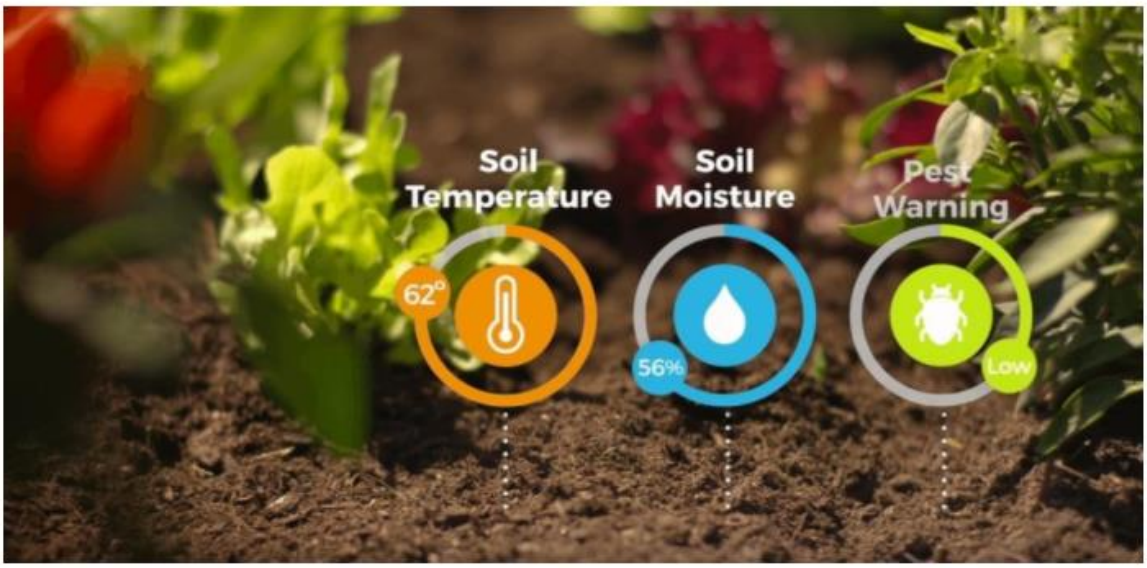


Fig. 2.10 Big Data in Agriculture

Interpretation

- Using big data, the whole process from harvesting to distribution process of agricultural products like paddy, wheat, vegetables, and so on can automate the watering system of the firm.
- Farmers can get enough time to concentrate on more important factors.
- Big data can take data from the past years and can suggest the pesticides that work best under certain conditions.
- Enables the firm's owners to use the same land for several purposes and data science applications can generate production throughout the year without any interval.
- While smart technologies are collecting data directly from the fields, advanced algorithms and data science can drive fantastic decision-making abilities.

Big Data Applications – Media and Entertainment

- Media companies and entertainment sectors need to drive digital transformation to distribute their products and contents as fast as possible at the present market.
- The availability of searching and accessing any content anywhere with any device becomes a widespread practice.
- It can even help to figure out the views or likes of an artist to measure the popularity in the digital media sector.

Fig. 2.11 Big Data in Media & Entertainment

- Helps to gather the information and demands of the individual.
- Identifying the device and the most effective time to view data for analysis later.
- Can be used to identify the reason behind subscribing and unsubscribing a content and the interest in particular content.
- Big data applications help to set the ad target group for media companies.
- Can generate additional new features analyzing public demand.
- Even an artist can choose the placement where he wants to promote his performance.
- Depending on the popularity he/she can choose the devices, screen size and also OS to place his song or video.

Big Data Applications – Digital Marketing

- Marketing trends for the business have completely changed.
- Digital marketing is the key to make any business successful.
- Not only the big companies run marketing promotional activities
- but also the small entrepreneurs run successful advertising campaigns on social media platforms and promote their products.
- Big data has made digital marketing really powerful
- an essential part of any business.

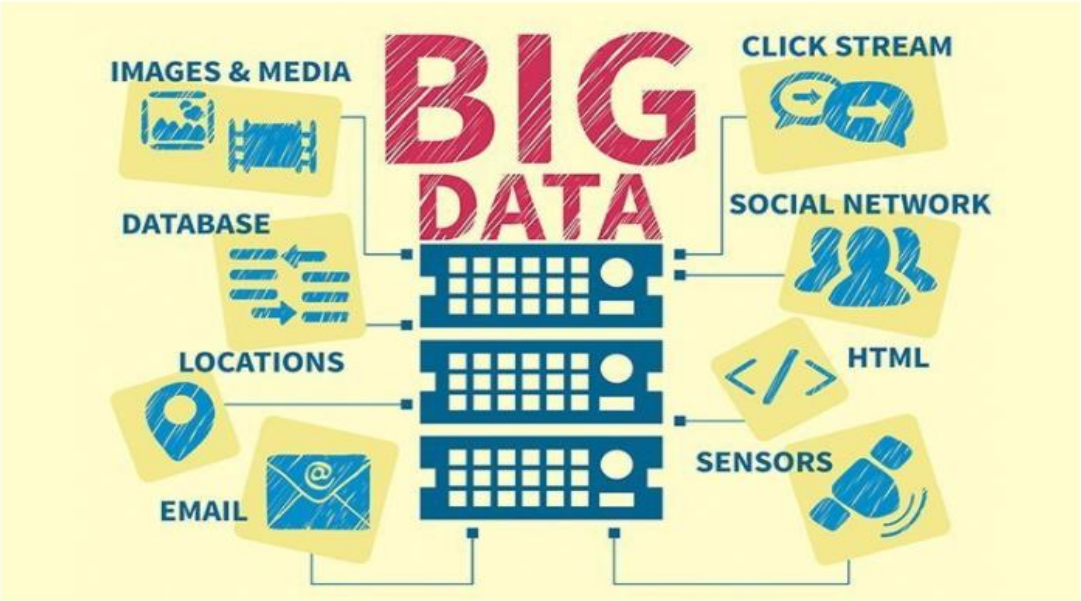


Fig. 2.12 Big Data in Digital Marketing

Interpretation

- Analyzes market, competitors and evaluate the business goal.
- identify the opportunities
- find the existing social media users and target them based on demographics, gender, income, age and interests.
- Generates reports after every ad campaign that includes the performance, audience engagements, and what could be done for generating better results also.
- Data science - transform customers into loyal clients.
- Focuses on highly searched topics - to rank business' website higher on Google.
- Using the existing audience database - target similar clients and earn the profits.

Big Data Applications – Social Media Sector

- Social media - popular digital media sector.
- Platform of social media marketing - completely depends upon the application of big data.
- Although it is not permitted to use all type of information in social media, it is important for proper maintenance and user satisfaction.

Interpretation

- provide opportunities for digital marketers to reach their customer audience directly through social media with the help of AI.
- ability for keyword analysis - makes it effective
- can be used to have a better understanding of the user's comfort and decision making.
- helps to analyze the preference, behavior and peak timing of a customer for staying relevant and competitive.

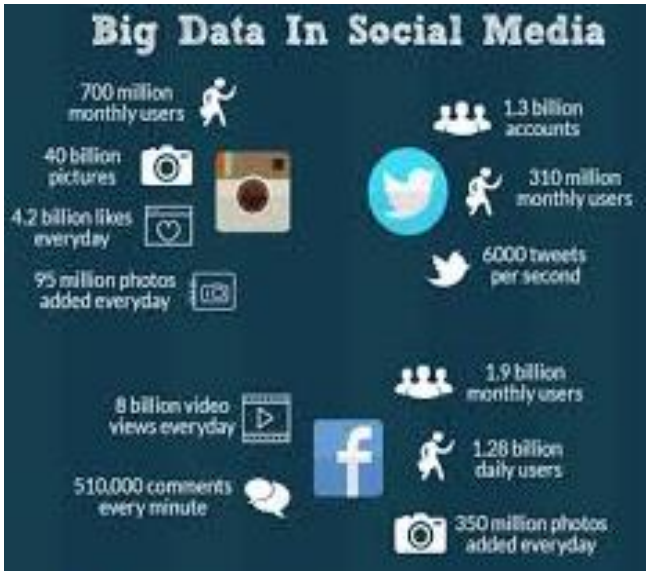


Fig. 2.13 Big Data in Social Media Sector

Big Data Applications – Airline Industry

- Airline industry has the best utilization of big data as it provides them with a minute to minute operational data.
- It helps with the gathered information about customer service, ticketing, weather forecast, etc.
- A small airline - take decisions for customer satisfaction and
- meet demands with the help of big data.

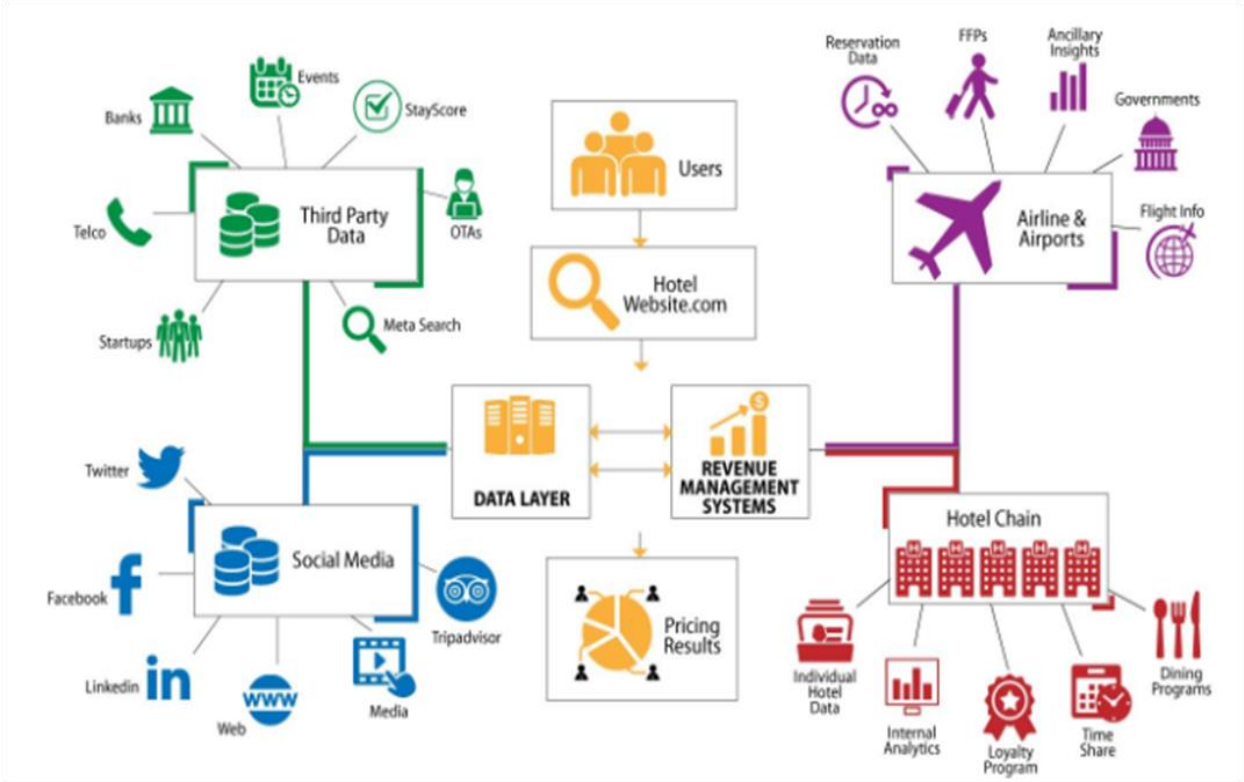


Fig. 2.14 Big Data in Airline Industry

Interpretation

- used for the smarter maintenance of aircraft by comparing operating costs, fuel quantity and costs, etc.

- improve the safety security of flights by capturing flight incident data and can strengthen aviation chain links.
- enhance customer service and customer's buying habits by analyzing past information.
- determine air traffic control, in-flight telemetry data information to have a comfortable flight.
- check real-time baggage status so that the no customer's baggage gets lost and suffers.

Big Data Applications – Ensure National Security

- Technology has shaped our lives and made better with its enormous possibilities.
- Big data is responsible for the success of these products.
- In many police forces, big data is used to improve their workflow and operations all around the world.
- Developed countries have implemented big data in their social and security activities a long ago,
- But underdeveloped countries have also started receiving the benefits of using big data already.

Fig. 2.15 Big Data in National Security - threat to India's national security

Interpretation

- The governments collect the information of all citizens, and this data is stored into a database for many purposes.
- Data science - extract meaningful information alongside a hidden relationship between datasets.
- Can evaluate the density of the population in a specific location and identify the possible threatening situations even before anything has occurred.
- Security officers can use this dataset to find any criminal and detect fraudulent activities in any area of the country.
- Besides, related personnel can predict the potential outspread of any virus or diseases and take necessary actions to prevent.

Big Data Applications – Government Sector

- The government needs to handle various local, national and global complex issue daily.
- The application of big data can leave an enormous impact on this sector by collecting all the information about millions of people that helps to take any decision considering locals.

It allows us to analyze the impact and opinion of any decision and to decide if any change is needed or not.

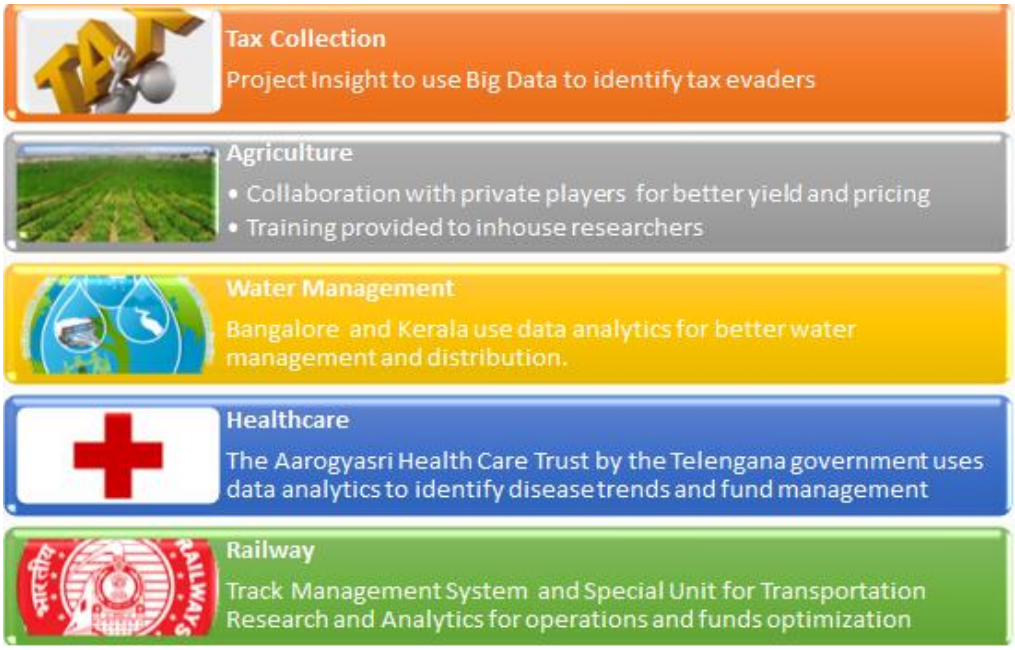


Fig. 2.16 Big Data in Government Sector

Interpretation

- The government can access daily functional information considering particular indecent or topic.
- identify the areas that need attention and analyze to improve the current situation.
- Governments easily reach to public demand and act accordingly.
- monitor the decisions taken by the government and evaluate the results
- Besides, can predict any terrorist attack and take necessary action to prevent unwanted conditions.

Big Data Applications – Tourism

- The tourism business - based upon the interest of an area toward the tourist group
- how they present the most desired package of tours within public demands.
- Modern tourists are more likely to use digital world rather than agencies.
- Big data helps to gather the knowledge of tourists all around the world about places

Interpretation

- Helps to gather the information of public demand by analyzing the data travelers provide on social media
- Some of the devices can gather credit or debit card information for quick purchase and quick identification of the traveler.
- Airlines can plan effectively by the data of passengers and their luggage throughout the journey and provide services accordingly.
- Based on the information of Geo-location, traffic, and weather, travel agencies can send offers and benefits suitable for the particular customer.
- Big data can help to provide security by using block chain technology.

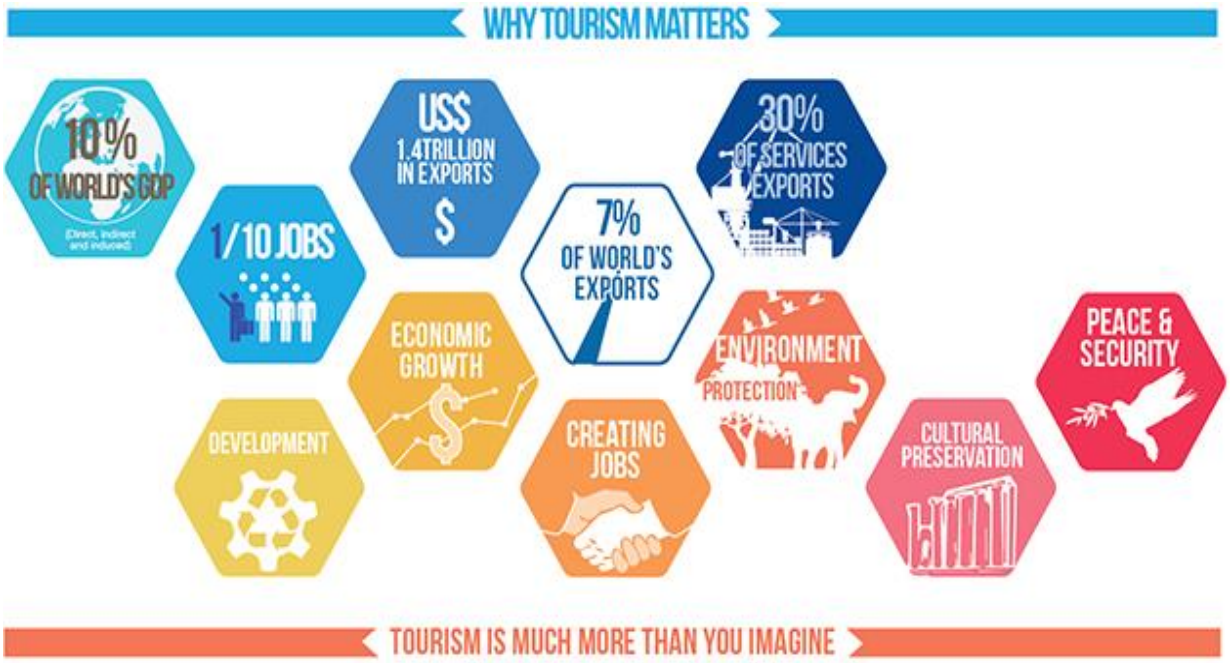


Fig. 2.17 Big Data in Tourism

Big Data Applications – Restaurants

Interpretation

- Collects data from customers and stored into a database to find new possibilities and identify opportunities.
- Evaluate data to predict customer behavior along with their food taste and demand.
- Identify companion products like if someone buys chicken, he/she is more likely to buy Pepsi also.
- Data mining - find hidden patterns and similarities that help the restaurants to determine their potential customers.
- Image processing and Machine learning identify the most wanted place in restaurants - for marketing purpose.
- stock management system are making life easier for the managers to keep track of the resources.

Big Data Applications – Fast Food Industry

Interpretation

- Using a populated database containing demographic, interest, and behavior, fast food companies try to bring changes in the food menu.
- Predict the number of food lovers at the specific time of the day – can be prepared according to the demand.
- They have all the information about their customer, which help to design the marketing strategy and follow trends.
- When the queue is long, the data science applications automatically show only the foods that can be prepared within a short time.
- Can evaluate the performance of a branch and fix the locations where the new offices should be opened to increase profit.

Big Data Applications – Casino Business

- profitable - proper and eye-catching establishments; focus on the decoration and interior design to attract the gamblers.

- most crucial - is maximizing profit where big data play most critical role.

Interpretation

- identify the most popular games, and casinos can increase the number of similar machines to engage more customers.
- identify places where people love to pass most of the time.
- can engage people and motivate them to come again and again
- When you give more money to the customers than what you earn - to shut your casino business. Big data ensures your profit.
- If any games are not popular Big data applications will automatically detect these games and help you to evaluate their performance.

Big Data Applications – Disaster Management

- Every year natural calamities like hurricane, floods and earthquakes cause huge damage and many lives.

Interpretation

- Can identify the potential disasters by evaluating temperature, water level, wind pressure, and other related factors.
- Weather forecasters can analyze data collected from Satellite and Radar. They can examine the weather conditions every 12 hours.
- Can identify the water level and possibility of flood in any specific area in a particular time of a year. Actions like excavations can be done before the flood attacks.
- Even, earthquakes can be brought under the monitor of natural disaster management specialists, and they can warn people as well.

Big Data Applications – Customer Oriented Service

Interpretation

- Identifies customer requirements, what they want and focuses on delivering the best service to accomplish their demand.
- Analyzes customer's behavior, interest and follows their trends to produce customer-oriented products.
- Introduces sustaining and efficiency innovation to deliver better products at lower costs.
- Can collect many influential data from the customers and use the insight while designing a marketing model for promotion.
- Finds the similarity between clients and their needs. So targeting based advertising campaigns can be conducted easily.

To conclude,

- can understand the importance of big data applications in real life.
- Even though a few days ago, the enormous impact was not visible but now with the recent development of AI, advanced algorithms, data mining techniques, and Image processing are helping big data to become more useful than ever.
- In every division of our life, the uses of big data have added an extra advantage.
- In the coming days, many changes and advancement of existing systems will be introduced.
- Undoubtedly, big data will be part of this evolution process and play the most influential role as well.
- If you want to be a successful data scientist then you must know the viable usages of big data in the modern era.

7. Algorithms using map reduce

MapReduce is a Distributed Data Processing Algorithm, introduced by Google. MapReduce Algorithm is mainly inspired by Functional Programming model. MapReduce algorithm is mainly useful to process huge amount of data in parallel, reliable and efficient way in cluster environments. It is similar to "Divide and Conquer" algorithm. It uses Divide and Conquer technique to process large amount of data. It divides input task into smaller and manageable sub-tasks (They should be executable independently) to execute them in-parallel.

MapReduce

- is a programming model for writing applications
- that can process Big Data in parallel on multiple nodes.
- MapReduce provides analytical capabilities
- for analyzing huge volumes of complex data.

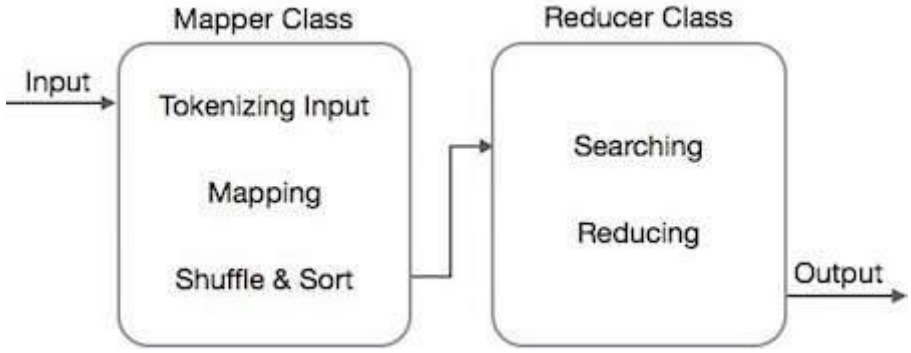


Fig. 2.18 MapReduce

Algorithms using MapReduce

The MapReduce algorithm contains two important tasks, namely

1. Map and
2. Reduce

- The map task is done by means of Mapper Class
- The reduce task is done by means of Reducer Class.
- Mapper class takes the input, tokenizes it, maps and sorts it.
- The output of Mapper class is used as input by Reducer class, which in turn searches matching pairs and reduces them.
- MapReduce implements various mathematical algorithms to divide a task into small parts and assign them to multiple systems.
- In technical terms, MapReduce algorithm helps in sending the Map & Reduce tasks to appropriate servers in a cluster.

Sorting

- Sorting is one of the basic MapReduce algorithms to process and analyze data.
- MapReduce implements sorting algorithm to automatically sort the output key-value pairs from the mapper by their keys.
- Sorting methods are implemented in the mapper class itself.
- In the Shuffle and Sort phase, after tokenizing the values in the mapper class, the Context class (user-defined class) collects the matching valued keys as a collection.
- To collect similar key-value pairs (intermediate keys), the Mapper class takes the help of RawComparator class to sort the key-value pairs.
- The set of intermediate key-value pairs for a given Reducer is automatically sorted before they are presented to the Reducer.

Searching

- Searching plays an important role in MapReduce algorithm. It helps in the combiner phase (optional) and in the Reducer phase.

Example - find out the details of employee who draws the highest salary in a employee dataset.

- Let us assume we have employee data in four different files − A, B, C, and D. Let us also assume there are duplicate employee records in all four files because of importing the employee data from all database tables repeatedly. See the following illustration.

Fig. 2.19 MapReduce Example

- The Map phase processes each input file and provides the employee data in key-value pairs (<k, v> : <emp name, salary>). See the following illustration.
- The combiner phase (searching technique) will accept the input from the Map phase as a key-value pair with employee name and salary.
- Using searching technique, the combiner will check all the employee salary to find the highest salaried employee in each file.

Fig. 2.20 MapReduce Example – key,value pair

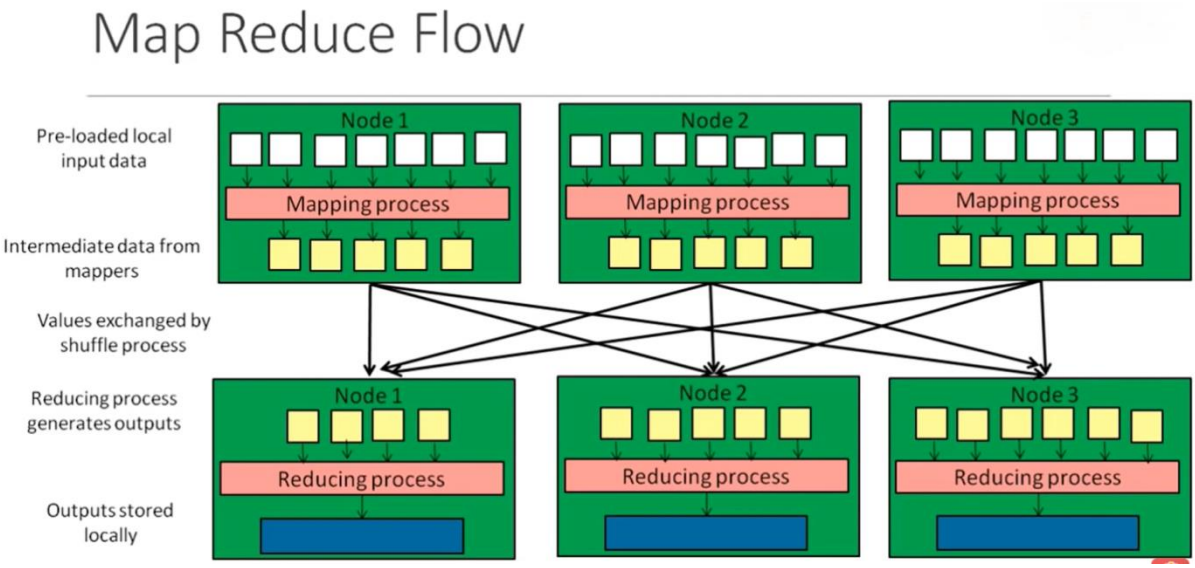


Fig. 2.21 MapReduce Flow

Problem Statement:

- Count the number of occurrences of each word available in a DataSet.

Input DataSet

```
1  Red Blue Red Blue Green Red Blue Green
2  White Black
3  Red White Black
4  Orange Green
5  Red Blue Red
6  Blue Green Red Blue
7  Green White Black
```

Output

```
1  Black = 3
2  Blue = 6
3  Green = 5
4  Orange = 1
5  Red = 7
6  White = 3
```

Final Ouput

MapReduce – Map Function (Split Step)

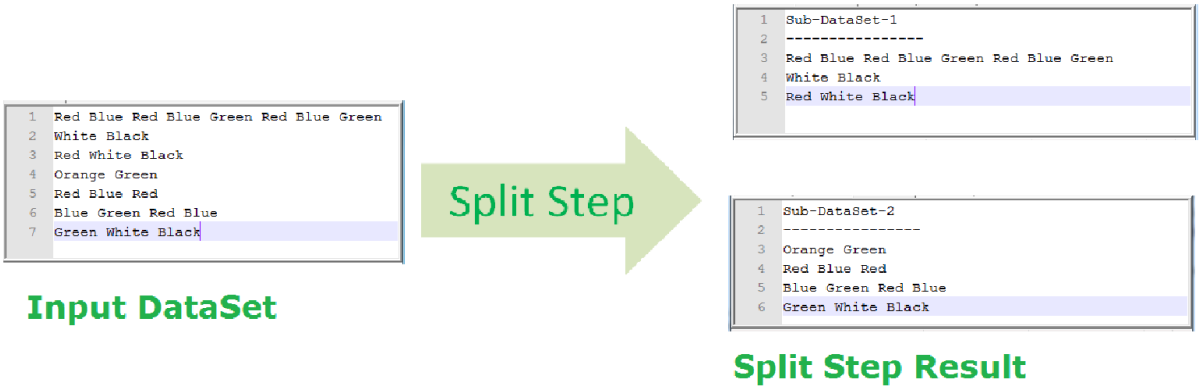


Fig. 2.22 MapReduce - Map Function (Split Step)

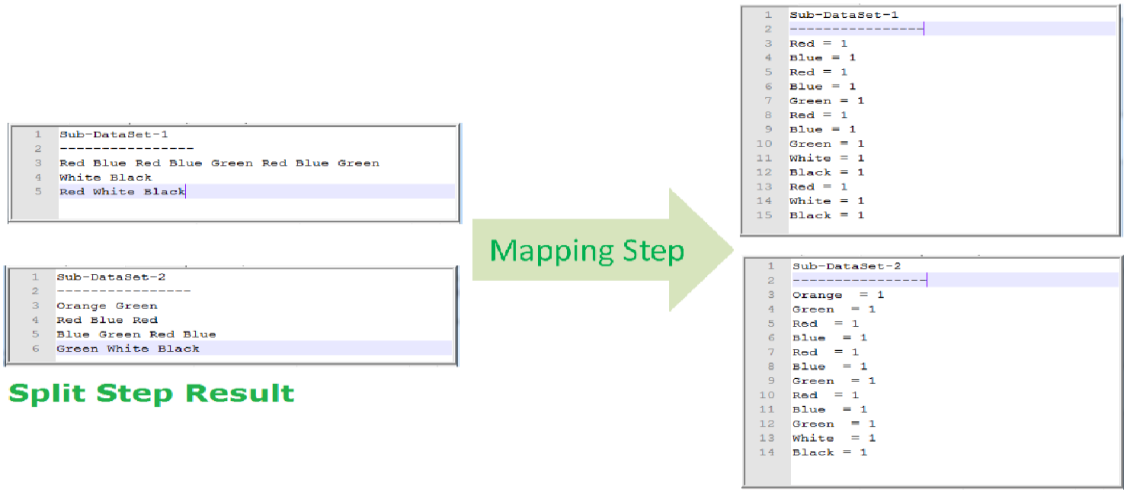


Fig. 2.22 MapReduce - Map Function (Mapping Step)

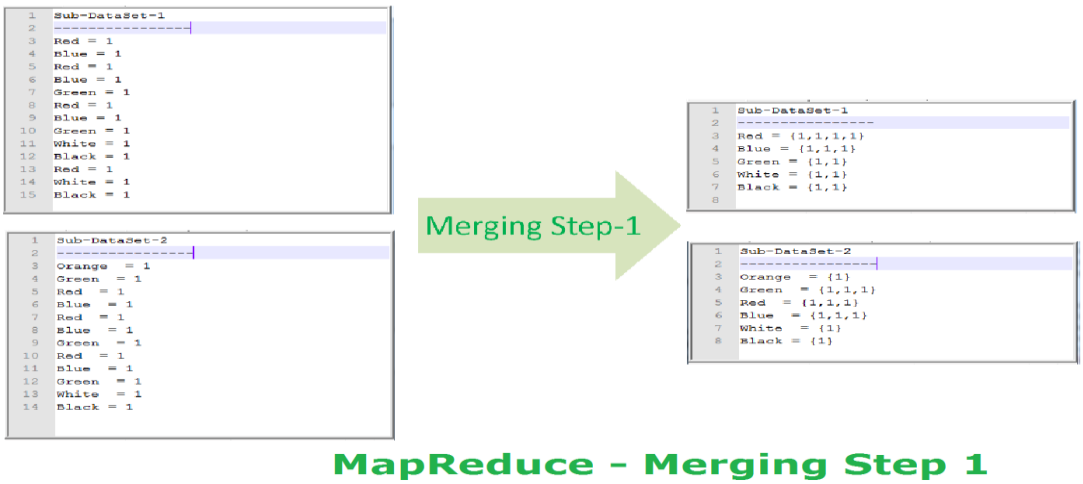


Fig. 2.23 MapReduce - Merging Step1

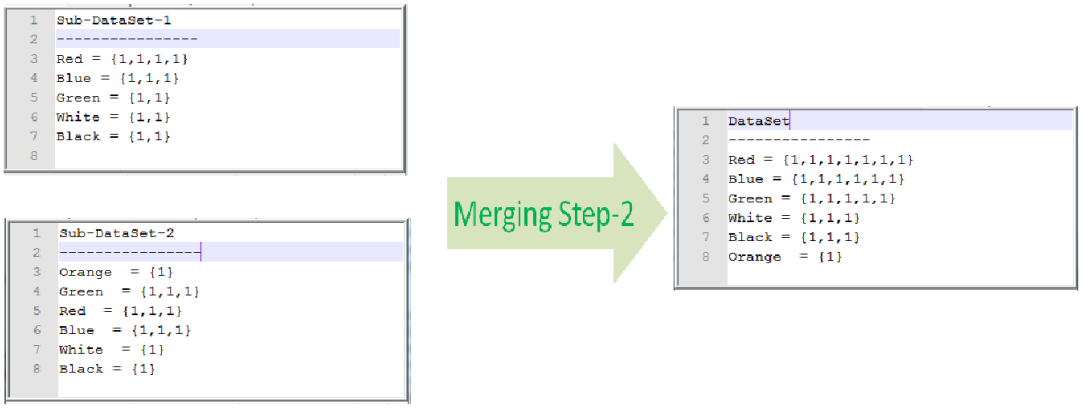


Fig. 2.24 MapReduce - Merging Step2

MapReduce – Shuffle Function (Sorting Step)

Fig. 2.25 MapReduce - Shuffle Function (Sorting Step)

MapReduce – Reduce Function (Reduce Step)

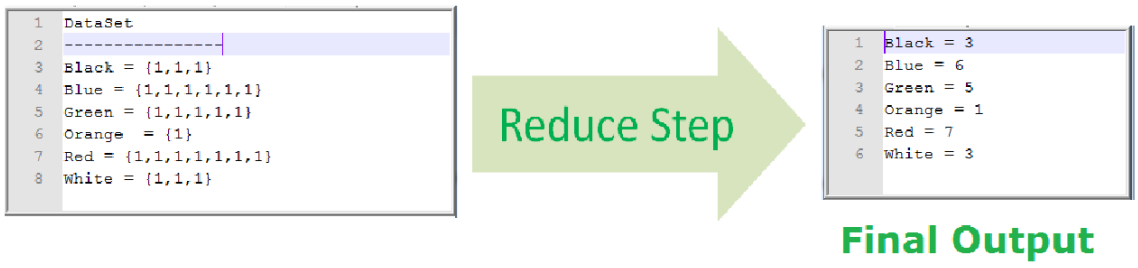


Fig. 2.26 MapReduce - Reduce Function (Reduce Step)

MapReduce 3 Step Process With WordCount Example

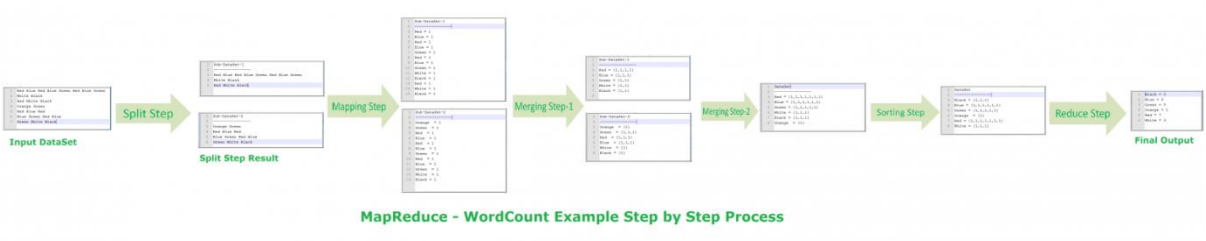


Fig. 2.27 MapReduce - 3 Step Process With WordCount Example

MapReduce – Word Count Problem with another example

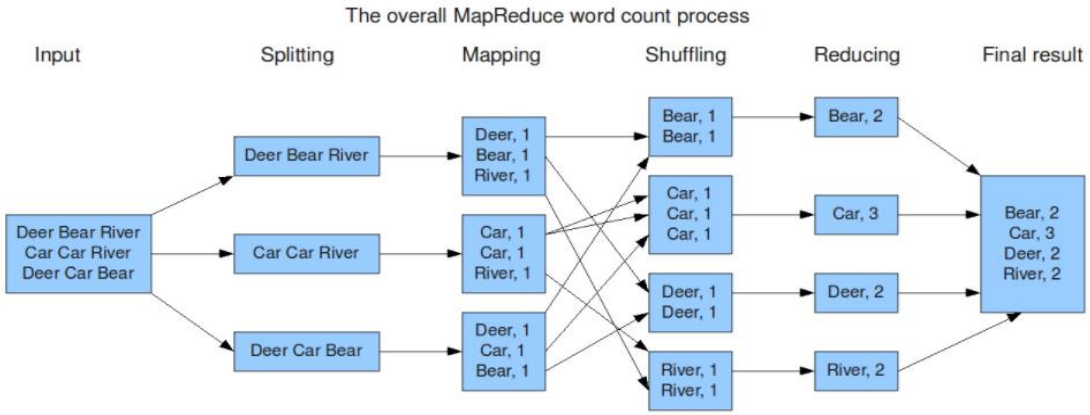


Fig. 2.28 MapReduce - Word Count Problem with another example

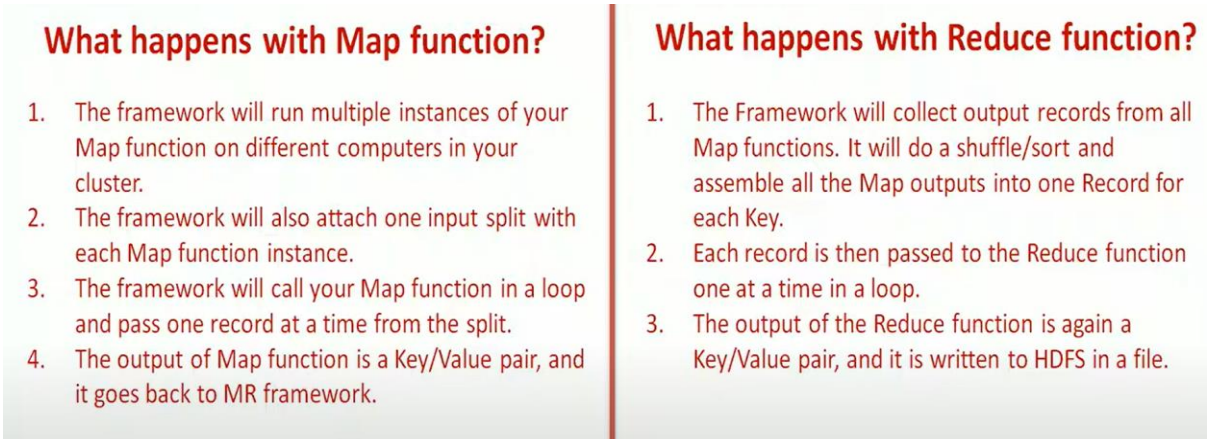


Fig. 2.29 Map function Vs Reduce function

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

School of Computing

Department of Computer Science and Engineering

UNIT III - Big Data – SBS1608

SYLLABUS

History of Hadoop - The Hadoop Distributed File System – Components of Hadoop-Analyzing the Data with Hadoop- Scaling Out- Hadoop Streaming- Design of HDFS-Java interfaces to HDFS- How Map Reduce Works-Anatomy of a Map Reduce Job run-Failures-Job Scheduling-Shuffle and Sort – Task execution – Map Reduce Features

1. History of Hadoop

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

In January 2008, Hadoop was made its own top-level project at Apache, confirming its success and its diverse, active community. By this time, Hadoop was being used by many other companies besides Yahoo!, such as Last.fm, Facebook, and the New York Times. In one well-publicized feat, the New York Times used Amazon's EC2 compute cloud to crunch through 4 terabytes of scanned archives from the paper, converting them to PDFs for the Web. The processing took less than 24 hours to run using 100 machines, and the project probably wouldn't have been embarked upon without the combination of Amazon's pay-by-the-hour model (which allowed the NYT to access a large number of machines for a short period) and Hadoop's easy-to-use parallel programming model.

In April 2008, Hadoop broke a world record to become the fastest system to sort an entire terabyte of data. Running on a 910-node cluster, Hadoop sorted 1 terabyte in 209 seconds (just under 3.5 minutes), beating the previous year's winner of 297 seconds. In November of the same year, Google reported that its MapReduce implementation sorted 1terabyte in 68 seconds. Then, in April 2009, it was announced that a team at Yahoo! had used Hadoop to sort 1 terabyte in 62 seconds. The trend since then has been to sort even larger volumes of data at ever faster rates. In the 2014 competition, a team from Databricks were joint winners of the Gray Sort benchmark. They used a 207-node Spark cluster to sort 100 terabytes of data in 1,406 seconds, a rate of 4.27 terabytes per minute.

Today, Hadoop is widely used in mainstream enterprises. Hadoop's role as a general purpose storage and analysis platform for big data has been recognized by the industry, and this fact is reflected in the number of products that use or incorporate Hadoop in some way. Commercial Hadoop support is available from large, established enterprise vendors, including EMC, IBM, Microsoft, and Oracle, as well as from specialist Hadoop companies such as Cloudera, Hortonworks, and MapR.

HADOOP BASICS:

- Need to process huge datasets on large clusters of computers
- Very expensive to build reliability into each application
- Nodes fail every day
  - Failure is expected, rather than exceptional
  - The number of nodes in a cluster is not constant
- Need a common infrastructure
  - Efficient, reliable, easy to use
  - Open Source, Apache Licence

Key Benefit & Flexibility

Client-server Concept

•Client sends requests to one or more servers which in turn accepts, processes them and return the requested information to the client.

•A server might run software which listens on particular ipand port number for requests

Examples:

Server -web server

Client –web browser

- Very Large Distributed File System
  10K nodes, 100 million files, 10PB
- Assumes Commodity Hardware
  Files are replicated to handle hardware failure
  Detect failures and recover from them
- Optimized for Batch Processing
  Data locations exposed so that computations can move to where data resides
  Provides very high aggregate bandwidth

2. The Hadoop Distributed File System (HDFS)

HDFS cluster has two types of nodes operating in a master-worker pattern:

i. A namenode – the master

ii. A number of datanodes – workers

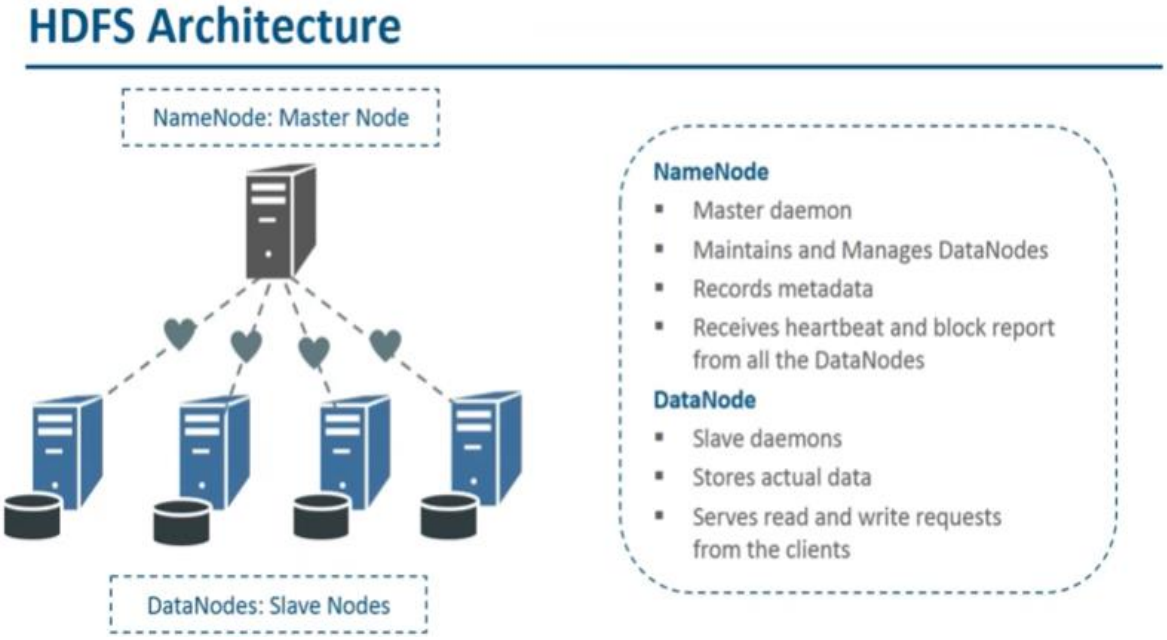


Fig. 3.1 HDFS Architecture

Namenode – manages the filesystem namespace

- It maintains the file system tree and the metadata for all the files and directories in the tree
- The 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, because this information is reconstructed from datanodes when the system starts

Datanodes – workhorses of the filesystem

- They store and retrieve blocks when they are told to (by clients or namenode)
- And they report back to the namenode periodically with lists of blocks that they are storing
- Without the namenode, the filesystem cannot be used
- If the machine running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes.
- For this reason, it is important to make the namenode resilient to failure

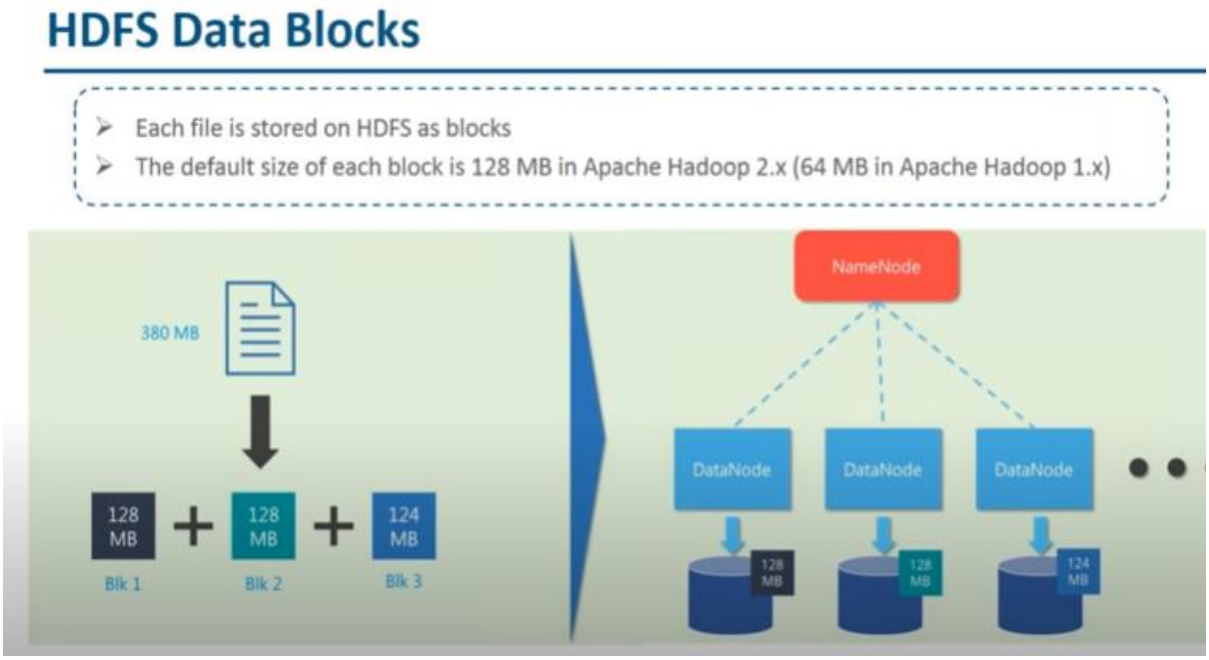


Fig. 3.2 HDFS Data Blocks

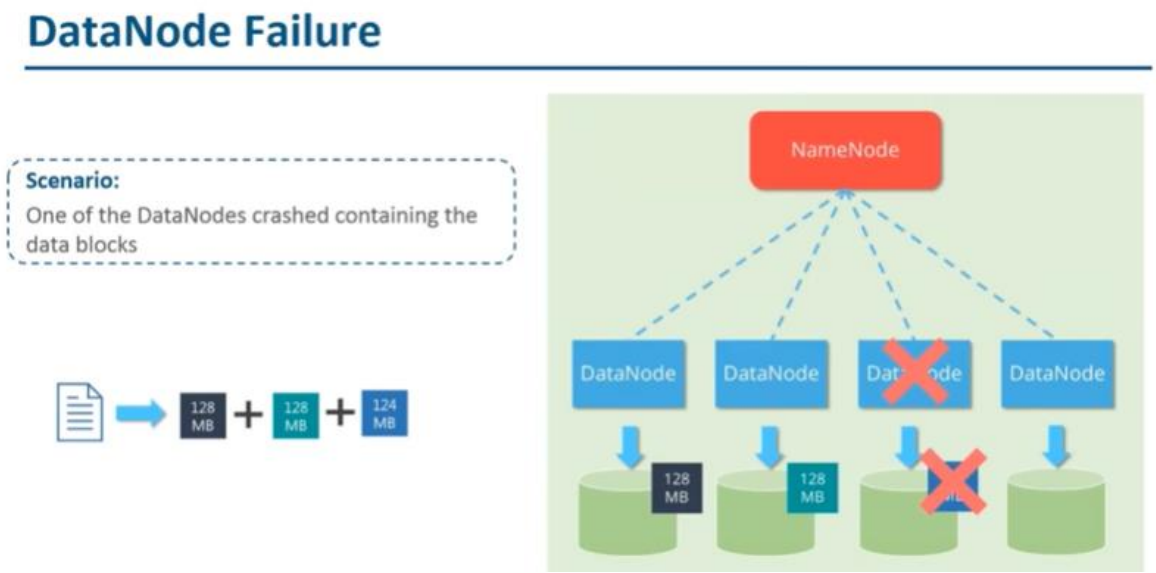


Fig. 3.3 DataNode Failure

- It is important to make the namenode resilient to failure;

Hadoop provides two mechanisms for this

1. The first way is to back up the files – Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems
2. Secondary namenode – does not act as a namenode. Its main role is to periodically merge the namespace

Block Caching

- Normally, a datanode reads blocks from disk, but for frequently accessed files the blocks may be explicitly cached in the datanode's memory – block cache
- By default, a block is cached in only one datanode's memory
- Job schedulers (for MapReduce, Spark and other frameworks) can take advantage of cached blocks by running tasks on the datanode where a block is cached, for increased read performance
- Users or applications instruct the namenode - which files to cache and for how long – by adding a cache directive to a cache pool
- Cache pools are an administrative grouping for managing cache permission and resource usage

HDFS Federation

- The namenode keeps a reference to every file and block in the filesystem in memory which means that on very large clusters with many files, memory becomes the limiting factor for scaling
- HDFS Federation – introduced in the 2.x release series; allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace
- For e.g., one namenode might manage all the files rooted under /user and a second namenode might handle files under /share
- Under federation, each namenode manages a namespace volume,
  - which is made up of the metadata for the namespace and
  - a block pool containing all the blocks for the files in the namespace
- Namespace volumes are independent of each other, which means namenodes do not communicate with one another
- Failure of one namenode does not affect the availability of the namespace managed by other namenodes
- Block pool storage is not partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools

HDFS High Availability

- The combination of replicating namenode metadata on multiple filesystems and using the secondary namenode to create checkpoints protects against data loss, but it does not provide high availability of the filesystem
- The namenode is still a single point of failure (SPOF)
- If it fails, 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
- In such an event, the whole Hadoop system would effectively be out of service until a new namenode could be brought online

- To recover from a failed namenode in this situation, an administrator starts a new primary namenode with one of the filesystem metadata replicas and configures datanodes and clients to use this new namenode
- The new namenode is not able to serve requests until it has –
    - Loaded its namespace image into memory
    - Replayed its edit log
    - 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 – can be 30 minutes or more
- The long recovery time is a problem for routine maintenance
- Hadoop remedied this situation by adding support for HDFS high availability
- In this implementation, there are a pair of namenodes in an active-standby configuration
- In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption
- Few architectural changes are needed –
    - The namenodes must use highly available shared storage to share the edit log
    - Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode's memory and not on disk
    - Clients must be configured to handle namenode failover, using a mechanism that is transparent to users
    - The secondary namenode's role is subsumed by the standby, which takes periodic checkpoints of the active namenode's namespace

3. Components of Hadoop

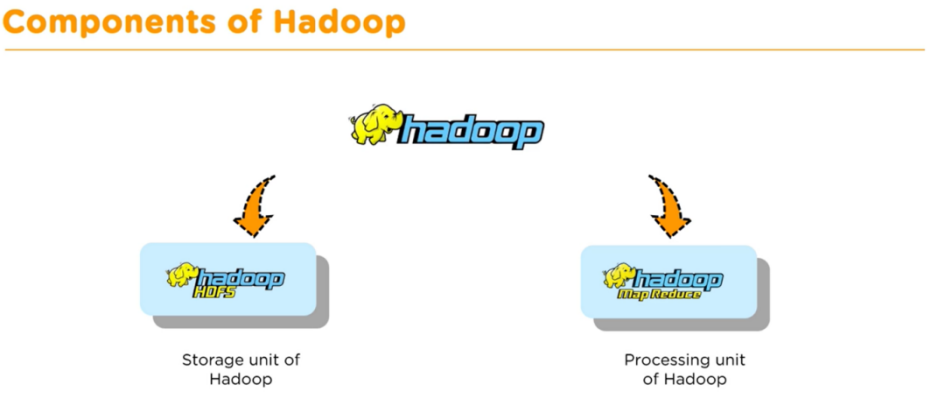


Fig. 3.4 Components of Hadoop

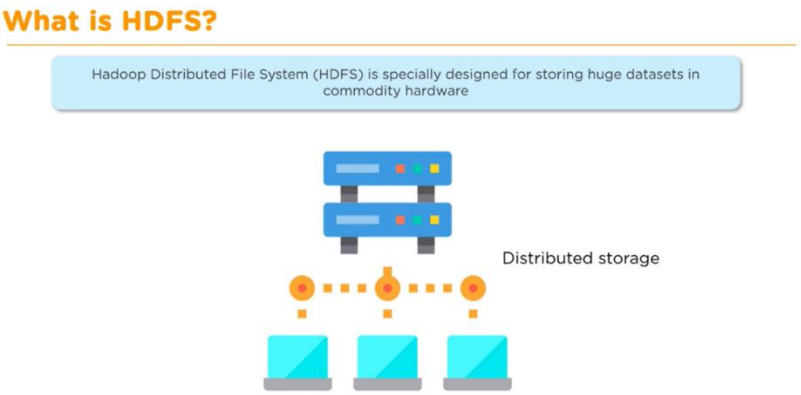


Fig. 3.5 HDFS Architecture

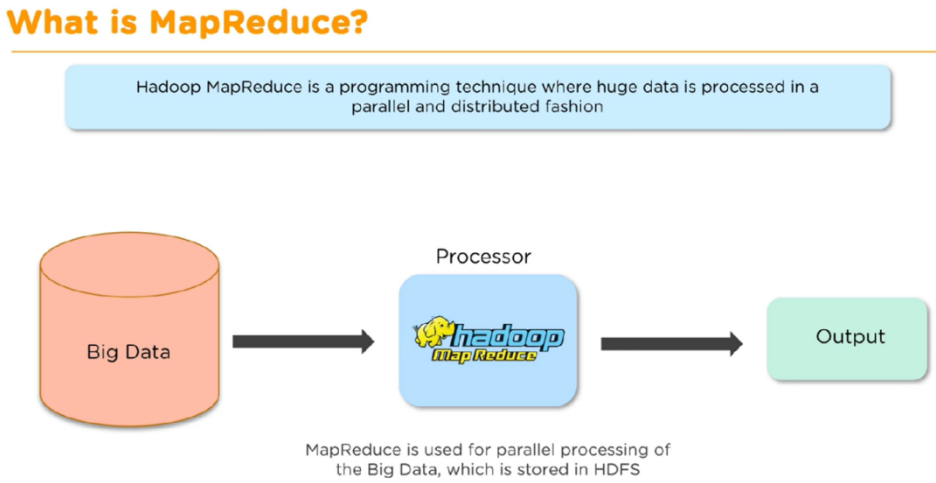


Fig. 3.6 MapReduce

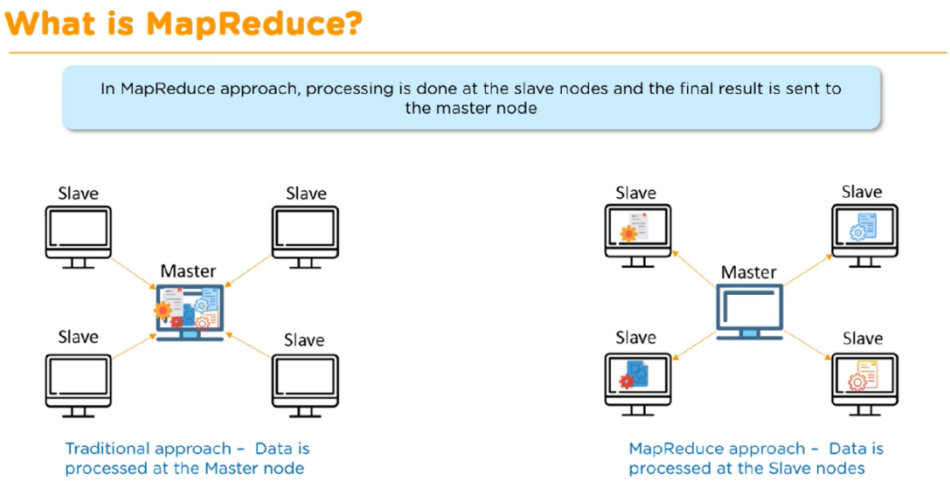


Fig. 3.7 MapReduce

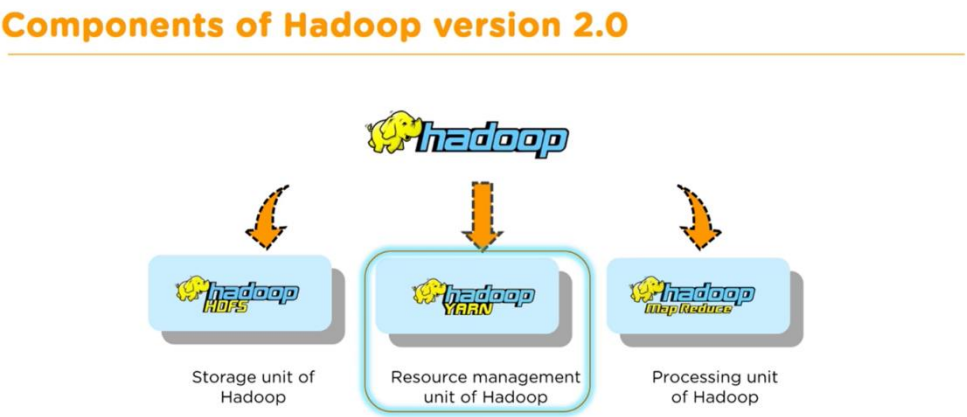


Fig. 3.8 Components of Hadoop

Fig. 3.9 YARN

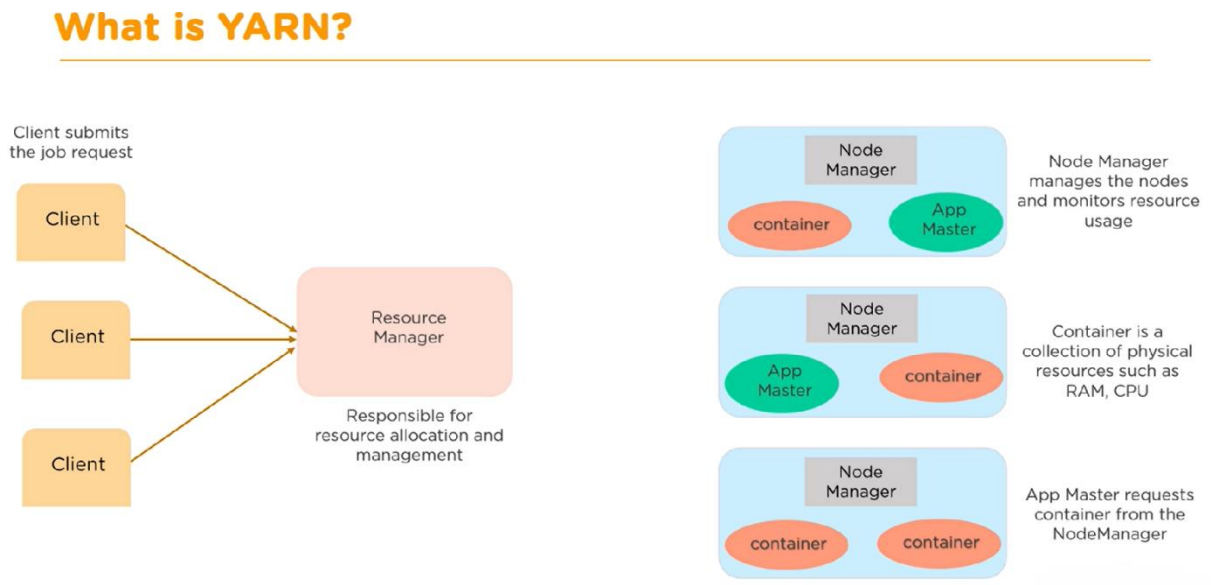


Fig. 3.10 YARN

- Yarn which is short for Yet Another Resource Negotiator.
- It is like the operating system of Hadoop as it monitors and manages the resources.
- Yarn came into the picture with the launch of Hadoop 2.x in order to allow different workloads.
- It handles the workloads like stream processing, interactive processing, and batch processing over a single platform.
- Yarn has two main components – Node Manager and Resource Manager.

Node Manager

- It is Yarn's per-node agent and takes care of the individual compute nodes in a Hadoop cluster.
- It monitors the resource usage like CPU, memory etc. of the local node and intimates the same to Resource Manager.

Resource Manager

- It is responsible for tracking the resources in the cluster and scheduling tasks like map-reduce jobs.
- Also, there is
    - Application Master and

    - Scheduler
- in Yarn.
- Also, there is
    - Application Master and
    - Scheduler
- in Yarn.

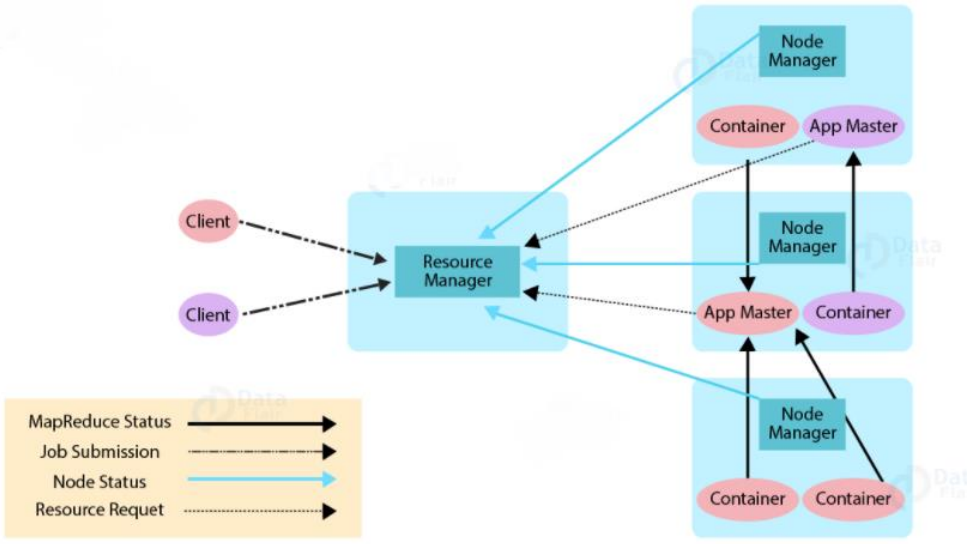


Fig. 3.11 YARN

Application Master has two functions and they are:-

- Negotiating resources from Resource Manager
- Working with NodeManager to monitor and execute the sub-task.
  Following are the functions of Resource Scheduler:-
- It allocates resources to various running applications
- But it does not monitor the status of the application.
- So in the event of failure of the task, it does not restart the same.
- Another concept Container.
- It is nothing but a fraction of NodeManager capacity i.e. CPU, memory, disk, network etc.

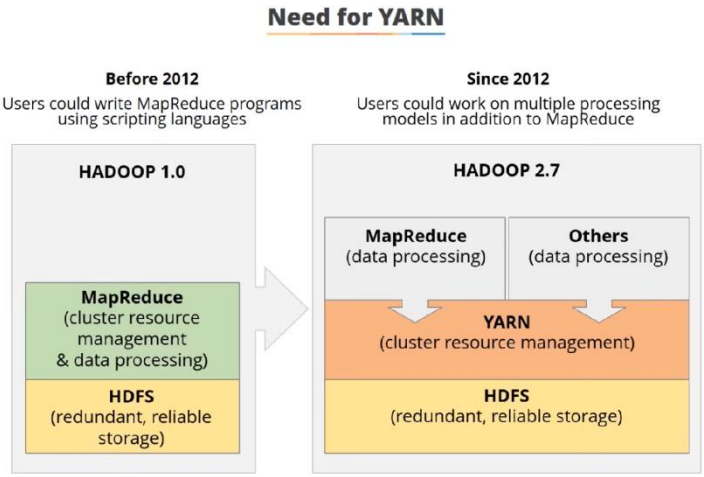


Fig. 3.12 Need for YARN

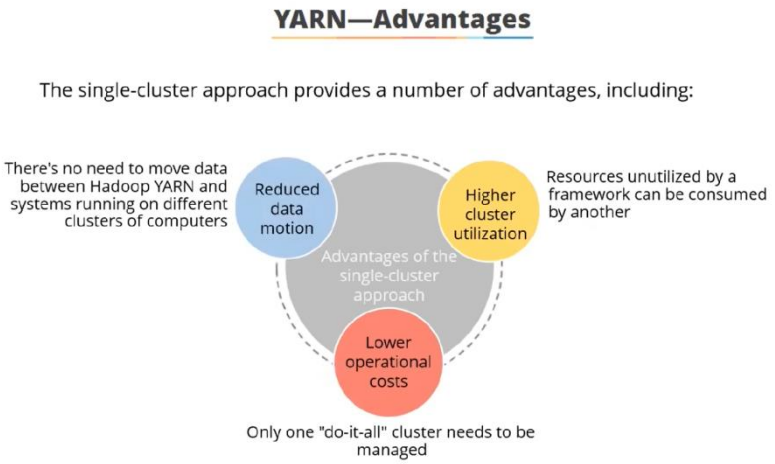


Fig. 3.13 YARN – Advantages

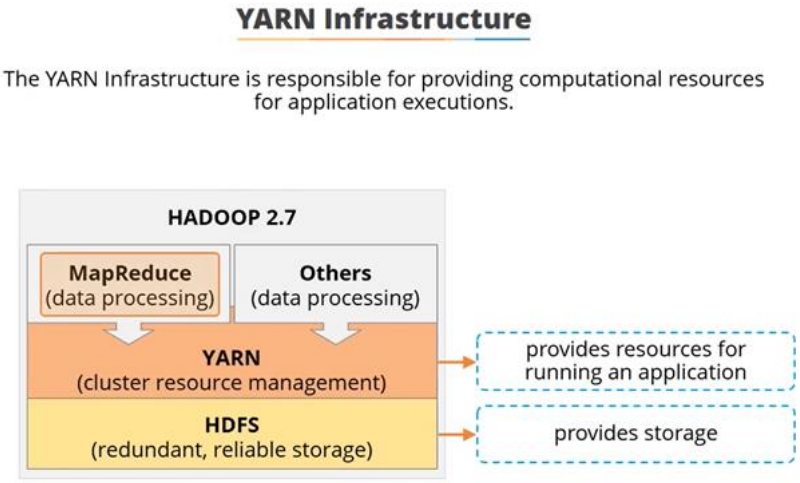


Fig. 3.14 YARN Infrastructure

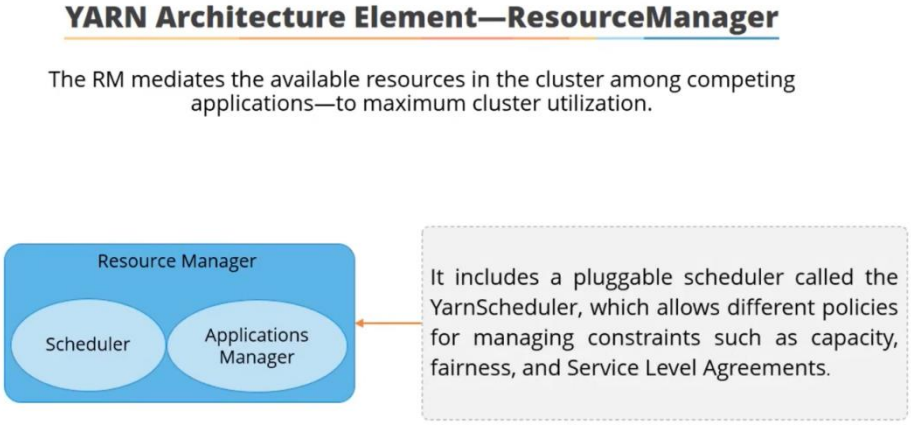


Fig. 3.15 YARN – Resource Manager

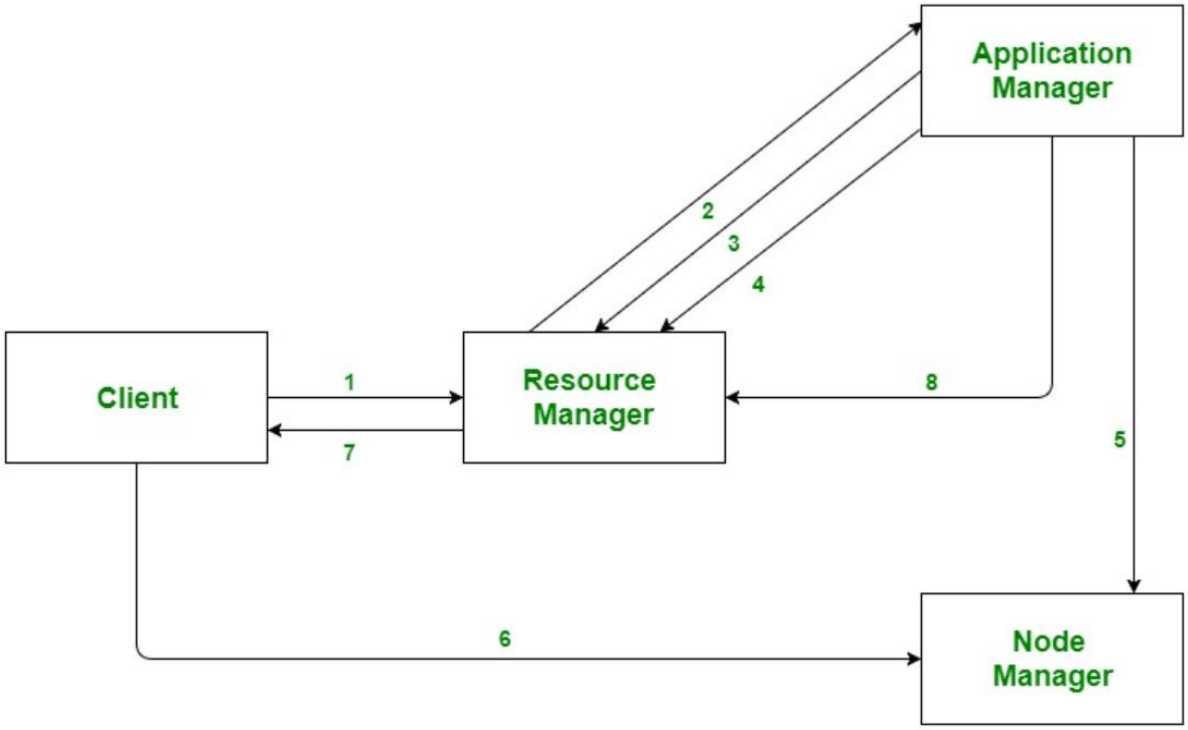


Fig. 3.16 Application workflow in Hadoop YARN

1. Client submits an application
2. The Resource Manager allocates a container to start the Application Manager
3. The Application Manager registers itself with the Resource Manager
4. The Application Manager negotiates containers from the Resource Manager
5. The Application Manager notifies the Node Manager to launch containers
6. Application code is executed in the container
7. Client contacts Resource Manager/Application Manager to monitor application's status
8. Once the processing is complete, the Application Manager un-registers with the Resource Manager

The objective of Apache Hadoop ecosystem components is to have an overview of what are the different components of Hadoop ecosystem that make Hadoop so powerful and due to which several Hadoop job roles are available now. We will also learn about Hadoop ecosystem components like HDFS and HDFS components, MapReduce, YARN, Hive, Apache Pig, Apache HBase and HBase components, HCatalog, Avro, Thrift, Drill, Apachemahout, Sqoop, ApacheFlume, Ambari, Z ookeeper and Apache Oozie to deep dive into Big Data Hadoop and to acquire master level knowledge of the Hadoop Ecosystem.

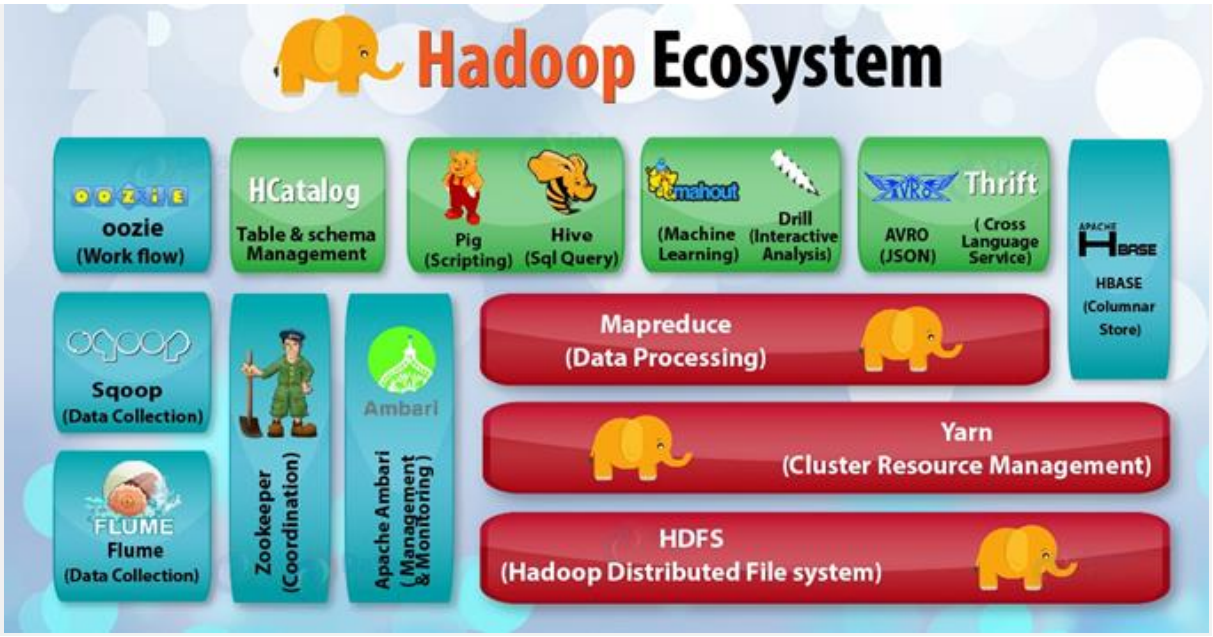


Fig. 3.17 Hadoop Ecosystem and Their Components

The list of Hadoop Components is discussed in this section one by one in detail.

Hadoop Distributed File System

It is the most important component of Hadoop Ecosystem. HDFS is the primary storage system of Hadoop. Hadoop distributed file system (HDFS) is a java based file system that provides scalable, fault tolerance, reliable and cost efficient data storage for Big data. HDFS is a distributed filesystem that runs on commodity hardware. HDFS is already configured with default configuration for many installations. Most of the time for large clusters configuration is needed. Hadoop interact directly with HDFS by shell-like commands.

HDFS Components:

There are two major components of Hadoop HDFS- NameNode and DataNode. Let's now discuss these Hadoop HDFS Components-

i. NameNode

It is also known as Master node. NameNode does not store actual data or dataset. NameNode stores Metadata i.e. number of blocks, their location, on which Rack, which Datanode the data is stored and other details. It consists of files and directories.

Tasks of HDFS NameNode

- Manage file system namespace.
- Regulates client's access to files.
- Executes file system execution such as naming, closing, opening files and directories.

ii. DataNode

It is also known as *Slave*. HDFS Datanode is responsible for storing actual data in HDFS. Datanode performs read and write operation as per the request of the clients. Replica block of Datanode consists of 2 files on the file system. The first file is for data and second file is for recording the block's metadata. HDFS Metadata includes checksums for data. At startup, each Datanode connects to its corresponding Namenode and does handshaking. Verification of

namespace ID and software version of DataNode take place by handshaking. At the time of mismatch found, DataNode goes down automatically.
Tasks of HDFS DataNode

- DataNode performs operations like block replica creation, deletion, and replication according to the instruction of NameNode.
- DataNode manages data storage of the system.

This was all about HDFS as a Hadoop Ecosystem component.

MapReduce

Hadoop MapReduce is the core Hadoop ecosystem component which provides data processing. MapReduce is a software framework for easily writing applications that process the vast amount of structured and unstructured data stored in the Hadoop Distributed File system.

MapReduce programs are parallel in nature, thus are very useful for performing large-scale data analysis using multiple machines in the cluster. Thus, it improves the speed and reliability of cluster this parallel processing.

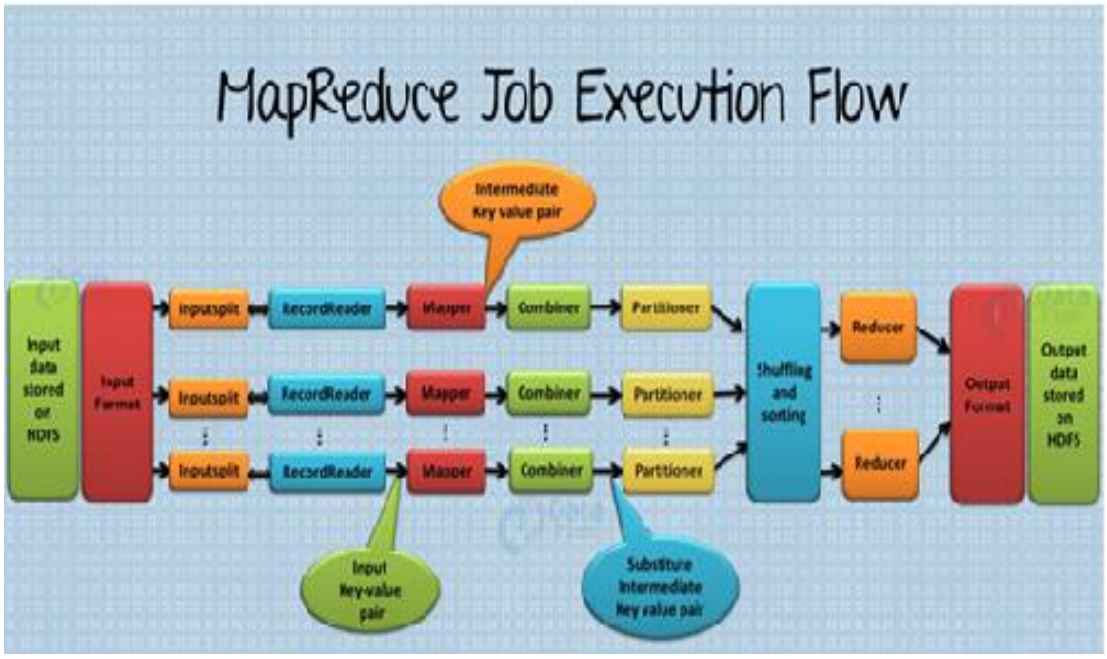


Fig. 3.18 Hadoop MapReduce

Working of MapReduce

Hadoop Ecosystem component 'MapReduce' works by breaking the processing into two phases:

- Map phase
- Reduce phase

Each phase has key-value pairs as input and output. In addition, programmer also specifies two functions: map function and reduce function. Map function takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Reduce function takes the output from the Map as an input and combines those data tuples based on the key and accordingly modifies the value of the key.

Features of MapReduce

- Simplicity – MapReduce jobs are easy to run. Applications can be written in any language such as java, C++, and python.
- Scalability – MapReduce can process petabytes of data.
- Speed – By means of parallel processing problems that take days to solve, it is solved in hours and minutes by MapReduce.
- Fault Tolerance – MapReduce takes care of failures. If one copy of data is unavailable, another machine has a copy of the same key pair which can be used for solving the same subtask.

YARN

` Hadoop YARN (Yet Another Resource Negotiator) is a Hadoop ecosystem component that provides the resource management. Yarn is also one the most important component of Hadoop Ecosystem. YARN is called as the operating system of Hadoop as it is responsible for managing and monitoring workloads. It allows multiple data processing engines such as real-time streaming and batch processing to handle data stored on a single platform.

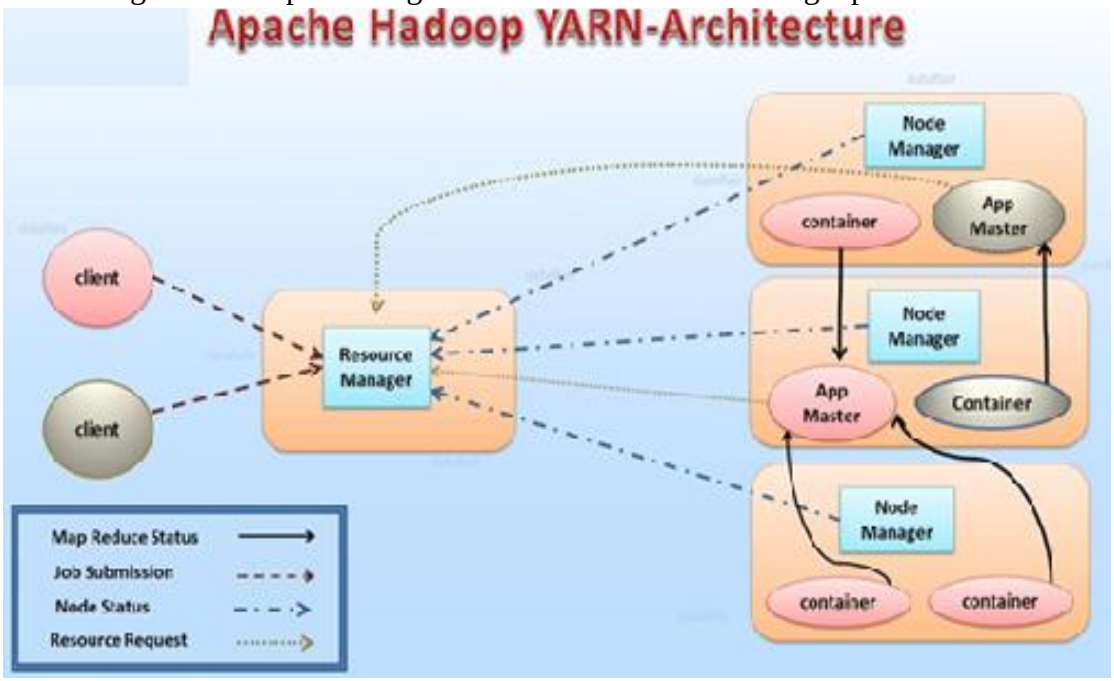


Fig. 3.19 Hadoop Yarn Diagram

YARN has been projected as a data operating system for Hadoop2. Main features of YARN are:

- Flexibility – Enables other purpose-built data processing models beyond MapReduce (batch), such as interactive and streaming. Due to this feature of YARN, other applications can also be run along with Map Reduce programs in Hadoop2.
- Efficiency – As many applications run on the same cluster, hence, efficiency of Hadoop increases without much effect on quality of service.
- Shared – Provides a stable, reliable, secure foundation and shared operational services across multiple workloads. Additional programming models such as graph processing and iterative modelling are now possible for data processing.

Hive

The Hadoop ecosystem component, Apache Hive, is an open source data warehouse system for querying and analyzing large datasets stored in Hadoop files. Hive do three main functions: data summarization, query, and analysis.Hive use language called HiveQL (HQL), which is similar to SQL. HiveQL automatically translates SQL-like queries into MapReduce jobs which will execute on Hadoop.

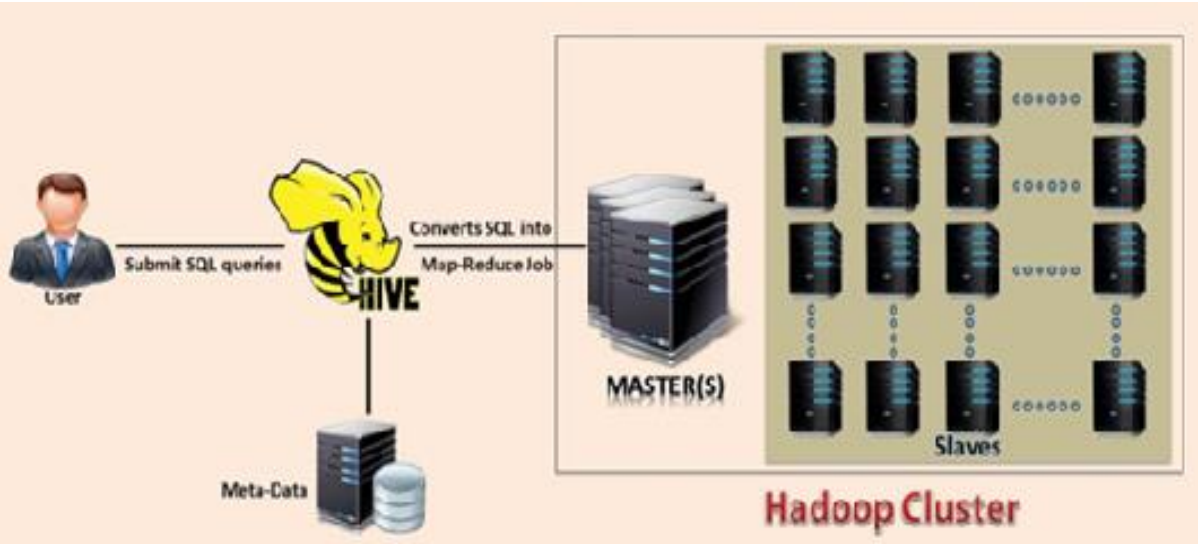


Fig. 3.20 Hive Diagram

Main parts of Hive are:

- Metastore – It stores the metadata.
- Driver – Manage the lifecycle of a HiveQL statement.
- Query compiler – Compiles HiveQL into Directed Acyclic Graph (DAG).
- Hive server – Provide a thrift interface and JDBC/ODBC server.

Pig

Apache Pig is a high-level language platform for analyzing and querying huge dataset that are stored in HDFS. Pig as a component of Hadoop Ecosystem uses Pig Latin language. It is very similar to SQL. It loads the data, applies the required filters and dumps the data in the required format. For Programs execution, pig requires Java runtime environment.

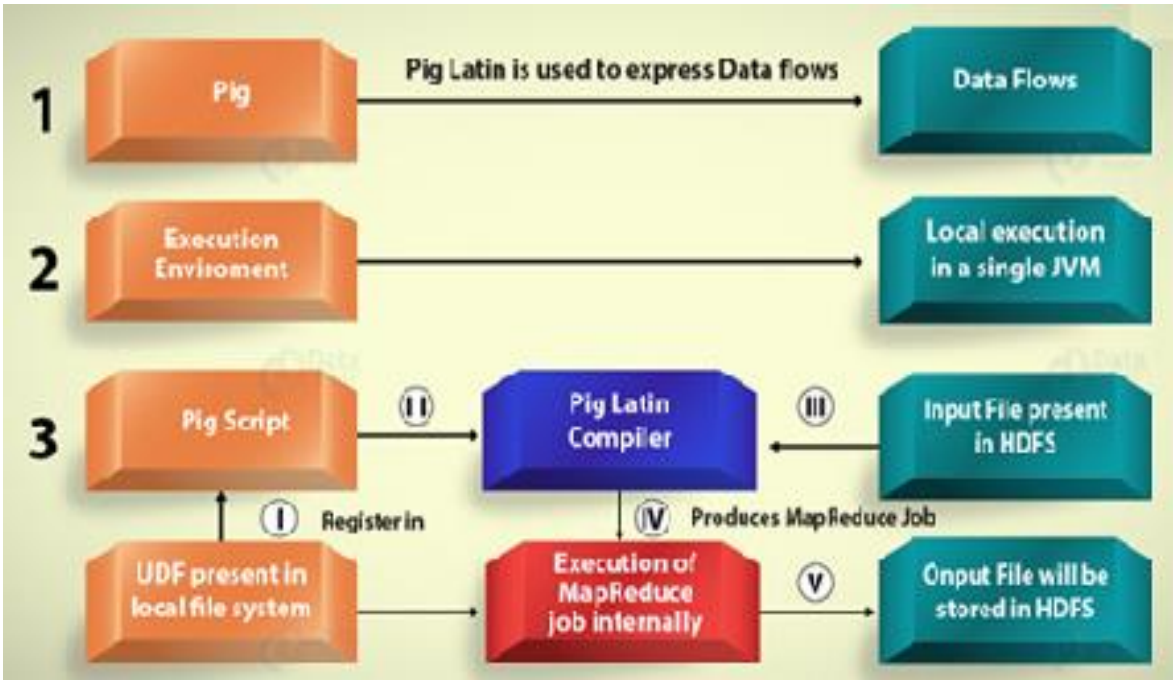


Fig. 3.21 Pig Diagram

Features of Apache Pig:

- Extensibility – For carrying out special purpose processing, users can create their own function.
- Optimization opportunities – Pig allows the system to optimize automatic execution. This allows the user to pay attention to semantics instead of efficiency.
- Handles all kinds of data – Pig analyzes both structured as well as unstructured.

HBase

Apache HBase is a Hadoop ecosystem component which is a distributed database that was designed to store structured data in tables that could have billions of row and millions of columns. HBase is scalable, distributed, and NoSQL database that is built on top of HDFS. HBase, provide real-time access to read or write data in HDFS.

Components of Hbase

There are two HBase Components namely- HBase Master and RegionServer.

i. HBase Master

It is not part of the actual data storage but negotiates load balancing across all RegionServer.

- Maintain and monitor the Hadoop cluster.
- Performs administration (interface for creating, updating and deleting tables.)
- Controls the failover.
- HMaster handles DDL operation.

ii. RegionServer

It is the worker node which handles read, writes, updates and delete requests from clients. Region server process runs on every node in Hadoop cluster. Region server runs on HDFS DateNode.

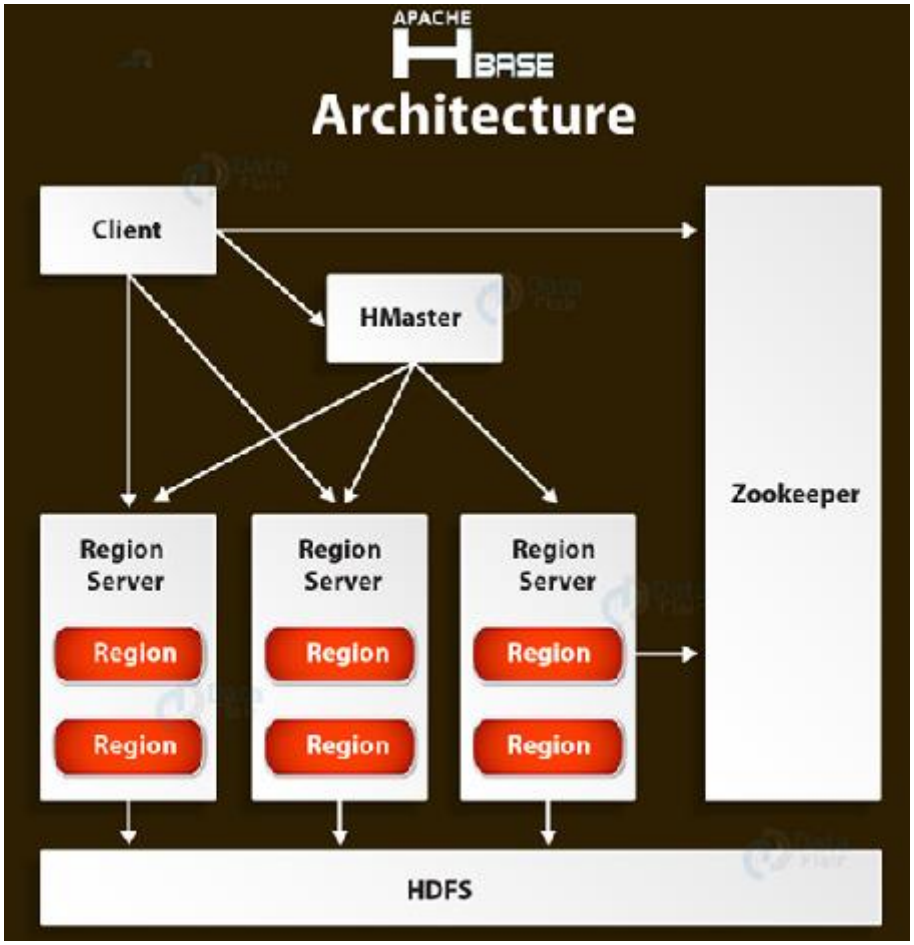


Fig. 3.22 HBase Diagram

HCatalog

It is a table and storage management layer for Hadoop. HCatalog supports different components available in Hadoop ecosystems like MapReduce, Hive, and Pig to easily read and write data from the cluster. HCatalog is a key component of Hive that enables the user to store their data in any format and structure.
By default, HCatalog supports RCFile, CSV, JSON, sequenceFile and ORC file formats.
Benefits of HCatalog:

- Enables notifications of data availability.
- With the table abstraction, HCatalog frees the user from overhead of data storage.
- Provide visibility for data cleaning and archiving tools.

Avro

Avro is a part of Hadoop ecosystem and is a most popular Data serialization system. Avro is an open source project that provides data serialization and data exchange services for Hadoop. These services can be used together or independently. Big data can exchange programs written in different languages using Avro.Using serialization service programs can serialize data into files or messages. It stores data definition and data together in one message or file making it easy for programs to dynamically understand information stored in Avro file or message.

Avro schema – It relies on schemas for serialization/deserialization. Avro requires the schema for data writes/read. When Avro data is stored in a file its schema is stored with it, so that files may be processed later by any program.
Dynamic typing – It refers to serialization and deserialization without code generation. It complements the code generation which is available in Avro for statically typed language as an optional optimization.
Features provided by Avro:

- Rich data structures.
- Remote procedure call.
- Compact, fast, binary data format.
- Container file, to store persistent data.

Thrift

It is a software framework for scalable cross-language services development. Thrift is an interface definition language for RPC(Remote procedure call) communication. Hadoop does a lot of RPC calls so there is a possibility of using Hadoop Ecosystem componet Apache Thrift for performance or other reasons.

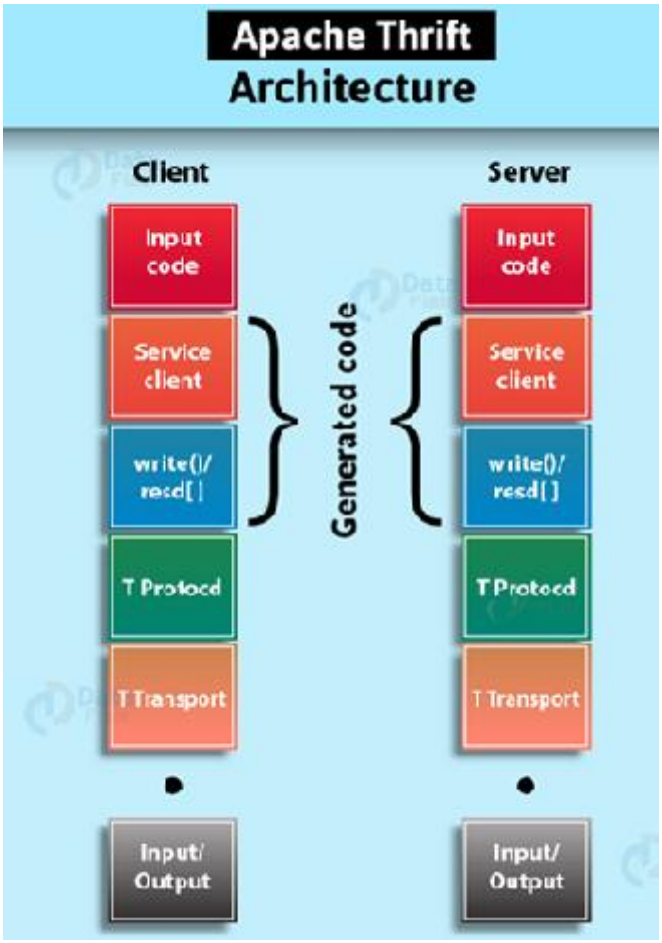


Fig. 3.23 Thrift Diagram

Apache Drill

The main purpose of the Hadoop Ecosystem Component is large-scale data processing including structured and semi-structured data. It is a low latency distributed query engine that

is designed to scale to several thousands of nodes and query petabytes of data. The drill is the first distributed SQL query engine that has a schema-free model.

Application of Apache drill

The drill has become an invaluable tool at cardlytics, a company that provides consumer purchase data for mobile and internet banking. Cardlytics is using a drill to quickly process trillions of record and execute queries.

Features of Apache Drill:

The drill has specialized memory management system to eliminates garbage collection and optimize memory allocation and usage. Drill plays well with Hive by allowing developers to reuse their existing Hive deployment.

- Extensibility – Drill provides an extensible architecture at all layers, including query layer, query optimization, and client API. We can extend any layer for the specific need of an organization.
- Flexibility – Drill provides a hierarchical columnar data model that can represent complex, highly dynamic data and allow efficient processing.
- Dynamic schema discovery – Apache drill does not require schema or type specification for data in order to start the query execution process. Instead, drill starts processing the data in units called record batches and discover schema on the fly during processing.
- Drill decentralized metadata – Unlike other SQL Hadoop technologies, the drill does not have centralized metadata requirement. Drill users do not need to create and manage tables in metadata in order to query data.

Apache Mahout

Mahout is open source framework for creating scalable machine learning algorithm and data mining library. Once data is stored in Hadoop HDFS, mahout provides the data science tools to automatically find meaningful patterns in those big data sets.

Algorithms of Mahout are:

- Clustering – Here it takes the item in particular class and organizes them into naturally occurring groups, such that item belonging to the same group are similar to each other.
- Collaborative filtering – It mines user behaviour and makes product recommendations (e.g. Amazon recommendations)
- Classifications – It learns from existing categorization and then assigns unclassified items to the best category.
- Frequent pattern mining – It analyzes items in a group (e.g. items in a shopping cart or terms in query session) and then identifies which items typically appear together.

Apache Sqoop

Sqoop imports data from external sources into related Hadoop ecosystem components like HDFS, Hbase or Hive. It also exports data from Hadoop to other external sources. Sqoop works with relational databases such as teradata, Netezza, oracle, MySQL.

Features of Apache Sqoop:

- Import sequential datasets from mainframe – Sqoop satisfies the growing need to move data from the mainframe to HDFS.
- Import direct to ORC files – Improves compression and light weight indexing and improve query performance.

- Parallel data transfer – For faster performance and optimal system utilization.
- Efficient data analysis – Improve efficiency of data analysis by combining structured data and unstructured data on a schema on reading data lake.
- Fast data copies – from an external system into Hadoop.

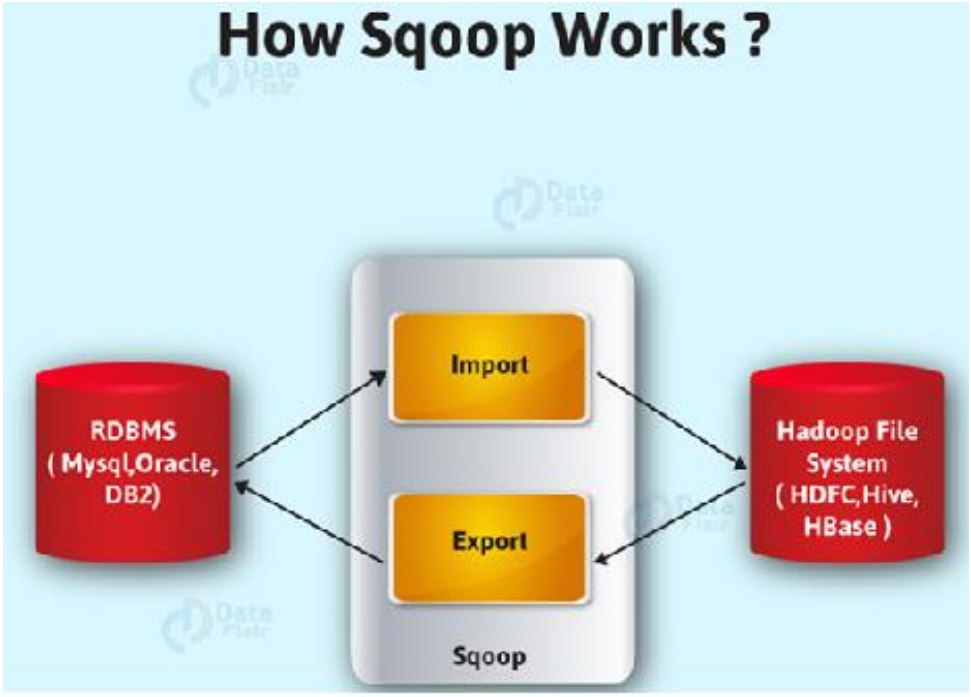


Fig. 3.24 Apache Sqoop Diagram

Apache Flume

Flume efficiently collects, aggregate and moves a large amount of data from its origin and sending it back to HDFS. It is fault tolerant and reliable mechanism. This Hadoop Ecosystem component allows the data flow from the source into Hadoop environment. It uses a simple extensible data model that allows for the online analytic application. Using Flume, we can get the data from multiple servers immediately into hadoop.

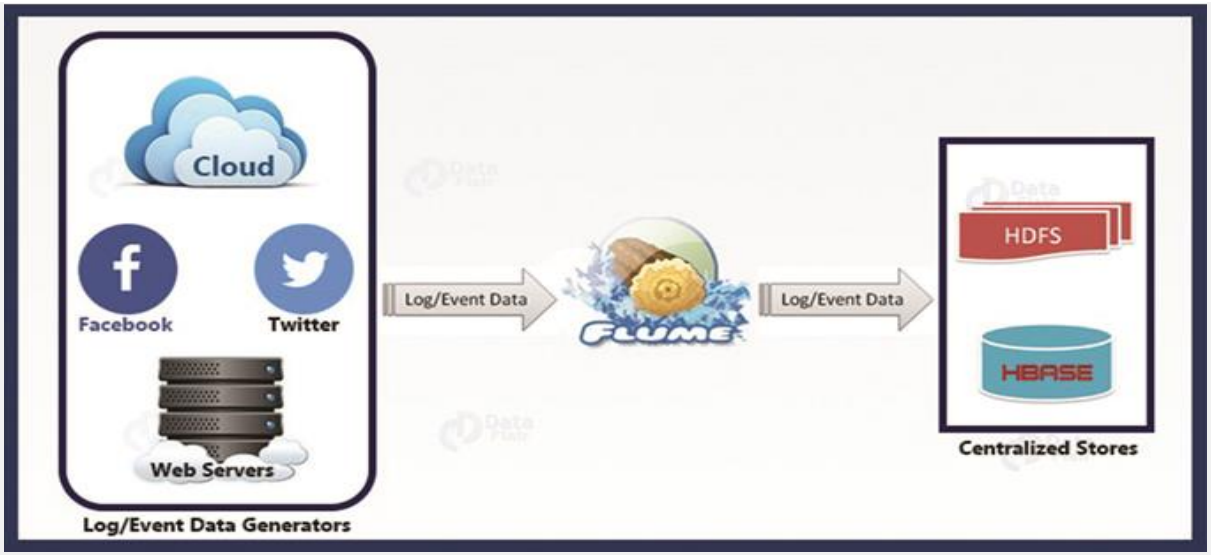


Fig. 3.25 Apache Flume

Ambari

Ambari, another Hadoop ecosystem component, is a management platform for provisioning, managing, monitoring and securing apache Hadoop cluster. Hadoop management gets simpler as Ambari provide consistent, secure platform for operational control.

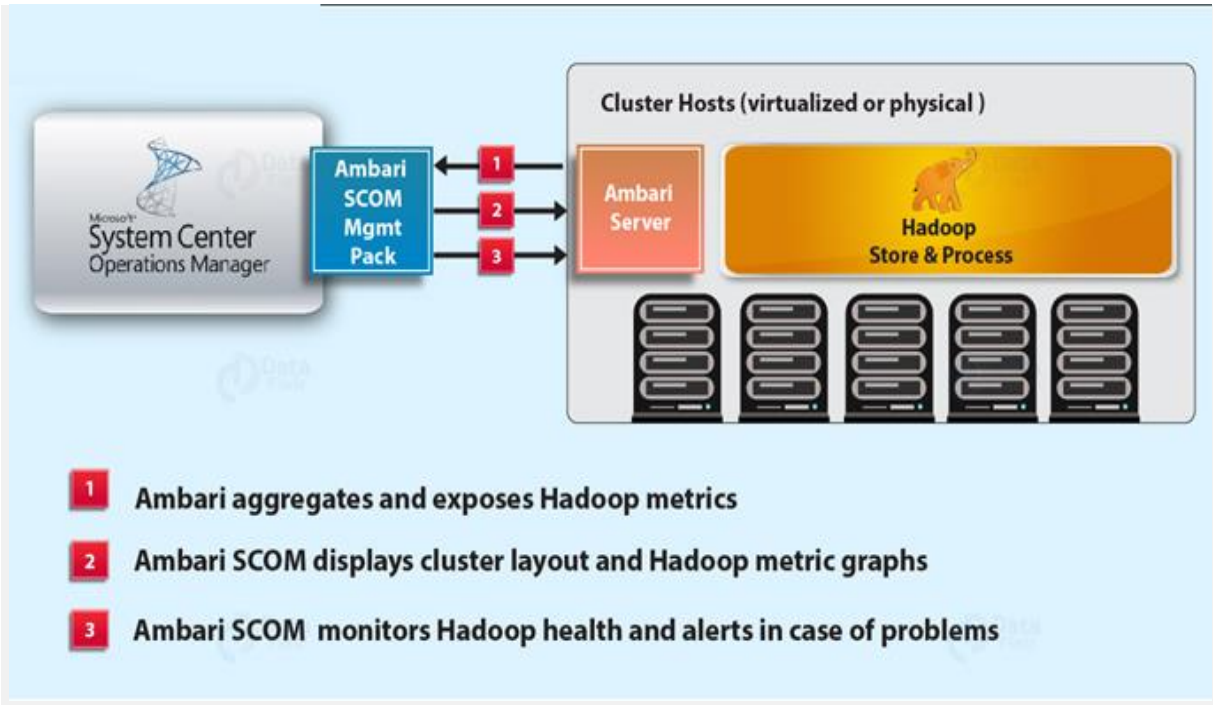


Fig. 3.26 Ambari Diagram

Features of Ambari:

- Simplified installation, configuration, and management – Ambari easily and efficiently create and manage clusters at scale.
- Centralized security setup – Ambari reduce the complexity to administer and configure cluster security across the entire platform.
- Highly extensible and customizable – Ambari is highly extensible for bringing custom services under management.
- Full visibility into cluster health – Ambari ensures that the cluster is healthy and available with a holistic approach to monitoring.

Zookeeper

Apache Zookeeper is a centralized service and a Hadoop Ecosystem component for maintaining configuration information, naming, providing distributed synchronization and providing group services. Zookeeper manages and coordinates a large cluster of machines.

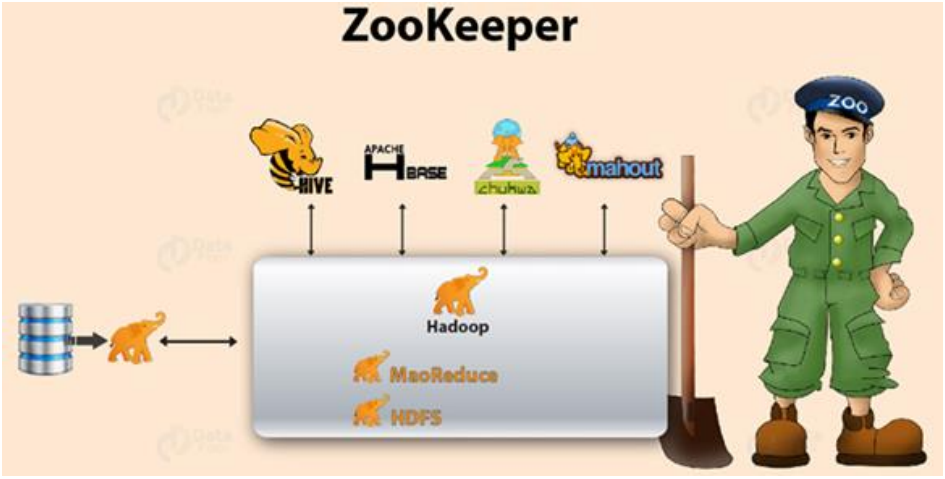


Fig. 3.27 ZooKeeper Diagram

Features of Zookeeper:

- Fast – Zookeeper is fast with workloads where reads to data are more common than writes. The ideal read/write ratio is 10:1.
- Ordered – Zookeeper maintains a record of all transactions.

Oozie

It is a workflow scheduler system for managing apache Hadoop jobs. Oozie combines multiple jobs sequentially into one logical unit of work. Oozie framework is fully integrated with apache Hadoop stack, YARN as an architecture center and supports Hadoop jobs for apache MapReduce, Pig, Hive, and Sqoop.

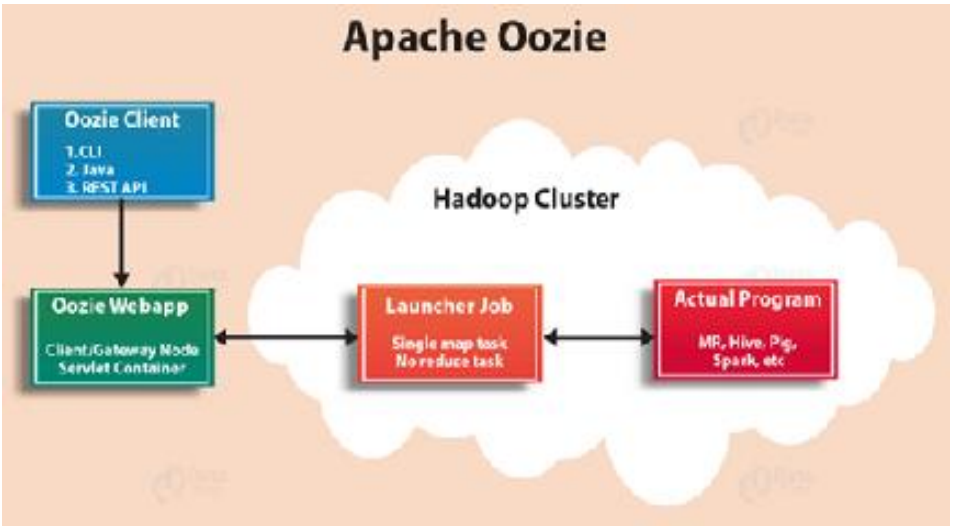


Fig. 3.28 Oozie Diagram

In Oozie, users can create Directed Acyclic Graph of workflow, which can run in parallel and sequentially in Hadoop. Oozie is scalable and can manage timely execution of thousands of workflow in a Hadoop cluster. Oozie is very much flexible as well. One can easily start, stop, suspend and rerun jobs. It is even possible to skip a specific failed node or rerun it in Oozie.

There are two basic types of Oozie jobs:

- Oozie workflow – It is to store and run workflows composed of Hadoop jobs e.g., MapReduce, pig, Hive.
- Oozie Coordinator – It runs workflow jobs based on predefined schedules and availability of data.

This was all about Components of Hadoop Ecosystem

4. Analyzing the Data with Hadoop

Map and Reduce

- MapReduce works by breaking the processing into two phases:
  - the map phase and the reduce phase.
- Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer.
- The programmer also specifies two functions: the map function and the reduce function.
- The input to our map phase is the raw NCDC data.
- A text input format is chosen that gives 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 is simple.

- year and the air temperature is pulled out, since these are the only fields - interested in.
- In this case, the map function is just a data preparation phase, setting up the data in such a way that the reducer function can do its work on it: finding the maximum temperature for each year.
- The map function is also a good place to drop bad records: here temperatures are filtered out - that are missing, suspect, or erroneous.

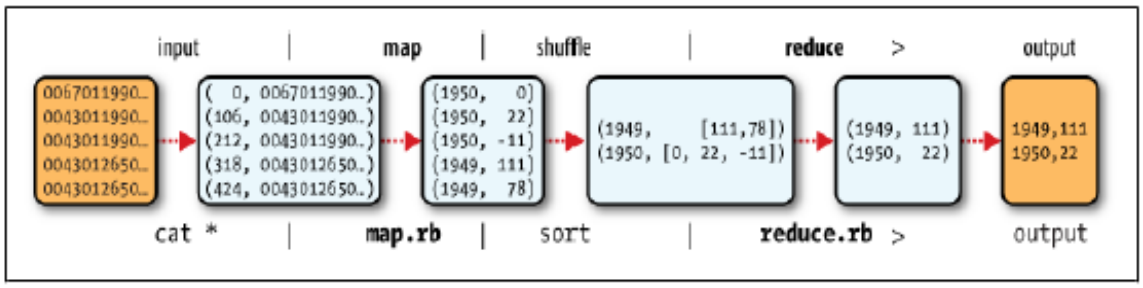


Fig. 3.29 MapReduce Logical Dataflow

Java MapReduce

- Having run through how the MapReduce program works, the next step is to express it in code.
- Need three things: a map function, a reduce function, and some code to run the job.

Example 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.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
```

```
public class MaxTemperatureMapper extends MapReduceBase
implements Mapper<LongWritable, Text, Text, IntWritable> {
private static final int MISSING = 9999;
public void map(LongWritable key, Text value,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {
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]")) {
output.collect(new Text(year), new IntWritable(airTemperature)); } } }
```

Example Reducer for maximum temperature example

```
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
public class MaxTemperatureReducer extends MapReduceBase
implements Reducer<Text, IntWritable, Text, IntWritable> {
public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {
int maxValue = Integer.MIN_VALUE;
while (values.hasNext()) {
maxValue = Math.max(maxValue, values.next().get());
}
output.collect(key, new IntWritable(maxValue));
}
}
```

Example Application to find the maximum temperature in the weather dataset

```
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
```

```
public class MaxTemperature {
public static void main(String[] args) throws IOException {
if (args.length != 2) {
System.err.println("Usage: MaxTemperature<input path><output path>");
System.exit(-1); }
JobConf conf = new JobConf(MaxTemperature.class);
conf.setJobName("Max temperature");
FileInputFormat.addInputPath(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
conf.setMapperClass(MaxTemperatureMapper.class);
conf.setReducerClass(MaxTemperatureReducer.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
JobClient.runJob(conf); } }
```

5. Scaling Out

- data flow for large inputs.
- For simplicity, the examples so far have used files on the local filesystem.
- However, to scale out,
  - data should be stored in a distributed filesystem, typically HDFS,
  - to allow Hadoop to move the MapReduce computation
  - to each machine hosting a part of the data.
- 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, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine.
- On the other hand,
- 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
- Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the data locality optimization.
- It should now be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node.
- Map tasks write their output to the local disk, not to HDFS.

Why is this?

- Map output is intermediate output: it’s processed by reduce tasks to produce the final output, and once the job is complete the map output can be thrown away.
- So storing it in HDFS, with replication, would be overkill.
- If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output.

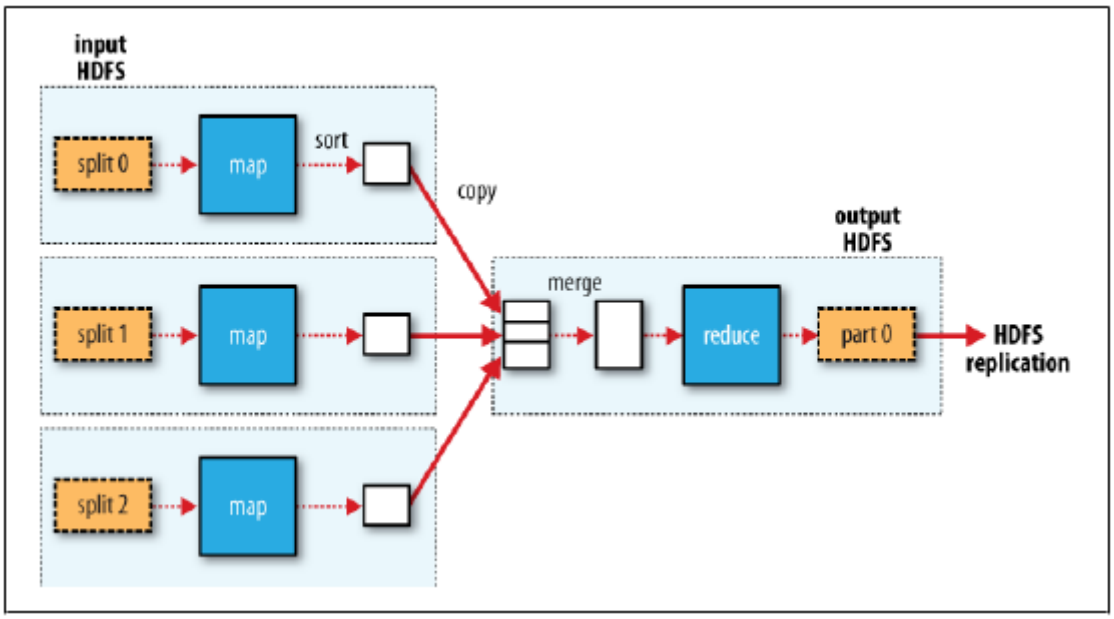


Fig. 3.30 MapReduce Data flow with the single reduce task

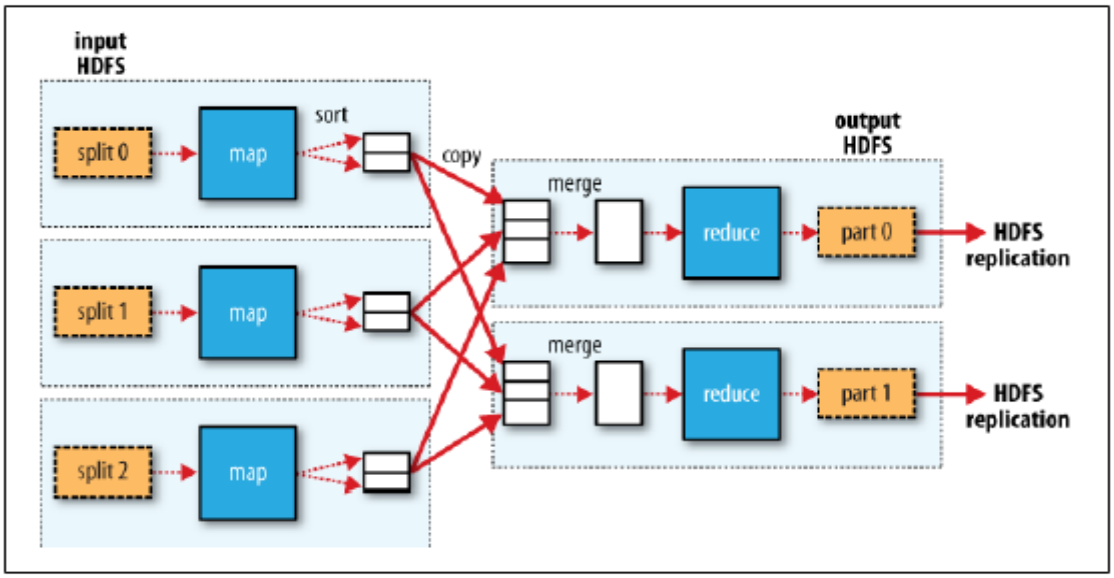


Fig. 3.31 MapReduce Data flow with the multiple reduce tasks

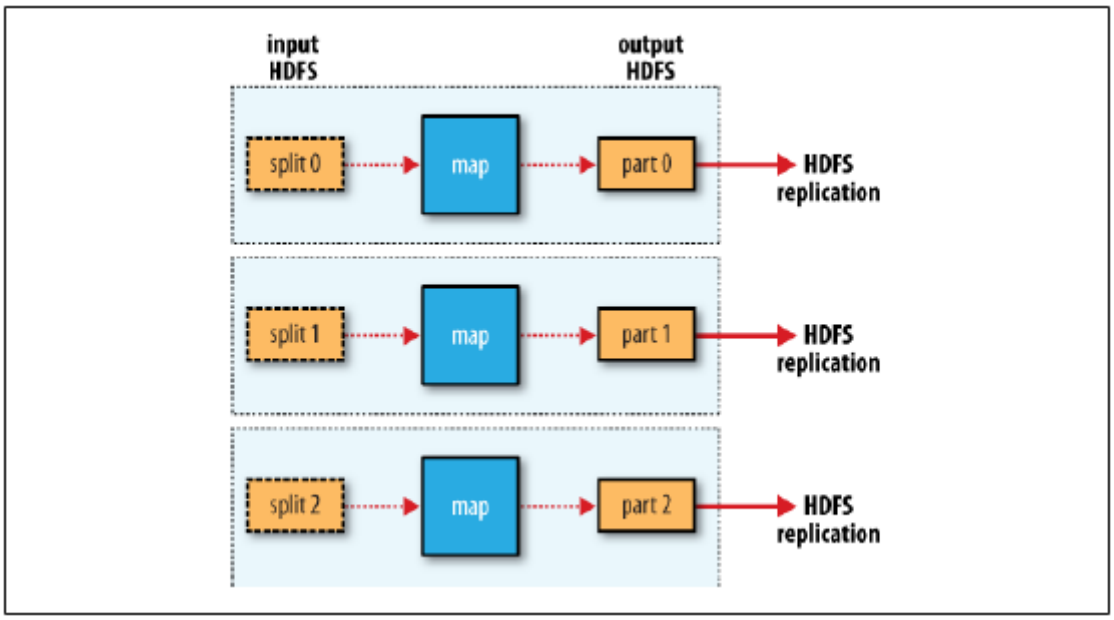


Fig. 3.32 MapReduce Data flow with the no reduce task

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, if at all.
- In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer
- The contract for the combiner function constrains the type of function that may be used.
- This is best illustrated with an example.
- Suppose that for 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)
- since 25 is the maximum value in the list.

A combiner function can be used - 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
- If 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.
- But 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 a combiner function can be used in the MapReduce job.

6. Hadoop Streaming

- Hadoop provides an API to MapReduce that allows writing map and reducing functions in languages other than Java.
- Hadoop Streaming uses Unix standard streams as the interface between Hadoop and the program, so any language can be used that can read standard input and write to standard output to write the MapReduce program.
- Streaming is naturally suited for text processing and when used in text mode, it has a line-oriented view of data.
- Map input data is passed over standard input to the map function, which processes it line by line and writes lines to standard output.
- A map output key-value pair is written as a single tab-delimited line.
- Input to the reduce function is in the same format—a tab-separated key-value pair—passed over standard input.
- The reduce function reads lines from standard input, which the framework guarantees are sorted by key, and writes its results to standard output.
- ------Ruby, Python

7. Design of HDFS

Distributed File System

- File Systems that manage the storage across a network of machines
- Since they are network based, all the complications of network programming occur
- This makes DFS more complex than regular disk file systems
  - For example, one of the biggest challenges is making the filesystem tolerate node failure without suffering data loss
- HDFS – a file system designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware
- Very Large files
  - Hundreds of megabytes, gigabytes or terabytes in size
  - There are Hadoop clusters running today 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, and then various analyses are performed on that dataset over time
- Commodity Hardware
  - Hadoop doesn't require expensive, highly reliable hardware.
  - It is designed to run on clusters of commodity hardware(commonly available hardware that can be obtained from multiple vendors)
  - Chance of node failure is high, atleast for large clusters
  - HDFS is designed to carry on working without interruption to the user in the face of such failure

Areas where HDFS is not good fit today

- Low-latency data access
  - Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS
  - HDFS is designed for delivering a high throughput of data and this may be at the expense of latency
  - Hbase – better choice of low-latency access
- Lots of small files
  - Because the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode
  - As a rule of thumb, each file, directory and block takes about 150 bytes.
  - So, e.g.,if one million files, each taking one block, would need at least 300 MB of memory
- Multiple writers, arbitrary file modifications
  - Files in HDFS may be written to by a single writer
  - Writers are always made at the end of the file, in append-only fashion
  - There is no support for multiple writers or for modifications at arbitrary offsets in the file

8. Java interfaces to HDFS

Reading Data from a Hadoop URL

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

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

IOUtils:org.apache.hadoop.io – Generic i/o code for reading and writing data to HDFS.

It is a utility class (handy tool) for I/O related functionality on HDFS.

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

setURLStreamHandlerFactory() method on URL with a
instance of FsUrlStreamHandlerFactory

- This method can be called only once per JVM, so it is typically executed in a static block
- This limitation means that if some other part of your program – perhaps a third-party component outside our control – sets a URLStreamHandlerFactory, - won't be able to use this approach for reading data from Hadoop

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).
- Path - a Hadoop filesystem URI, such as *hdfs://localhost/user/tom/quangle.txt*.
- FileSystem is a general filesystem API, so the first step is to retrieve an instance for the filesystem we want to use—HDFS in this case.
- There are three static factory methods for getting a FileSystem instance:
  public static FileSystem get(Configuration conf) throws IOException
  public static FileSystem get(URI uri, Configuration conf) throws IOException
  public static FileSystem get(URI uri, Configuration conf, String user) throws IOException
- A Configuration object encapsulates a client or server's configuration, which is set using configuration files read from the classpath, such as *conf/core-site.xml*.
- The first method returns the default filesystem (as specified in the file *conf/core-site.xml*, or the default local filesystem if not specified there).
- The second uses the given URI's scheme and authority to determine the filesystem to use, falling back to the default filesystem if no scheme is specified in the given URI.
- The third retrieves the filesystem as the given user, which is important in the context of security

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

```
package org.apache.hadoop.util;
public interface Progressable {
    public void progress();
}
```

- As an alternative to creating a new file, you can append to an existing file using the append() method (there are also some other overloaded versions):
  public FSDataOutputStream append(Path f) throws IOException

Directories

- FileSystem provides a method to create a directory:
  public booleanmkdirs(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'smkdirs() method.
- It returns true if the directory (and all parent directories) was (were) successfully created.
- Often, you don't 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.
- The FileStatus class encapsulates filesystem metadata for files and directories, including file length, block size, replication, modification time, ownership, and permission information.
- The method getFileStatus() on FileSystem provides a way of getting a FileStatus object for a single file or directory.

Deleting Data

- Use the delete() method on FileSystem 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.
- A nonempty directory is only deleted, along with its contents, if recursive is true (otherwise an IOException is thrown).

9. How MapReduce Works

10. Anatomy of a MapReduce Job run

Anatomy of a MapReduce Job Run

- Can run a MapReduce job with a single method call:
  submit () on a Job object
- Can also call
  waitForCompletion() – submits the job if it hasn't been submitted already, then waits for it to finish

At the highest level, there are five independent entries:

- The client, which submits the MapReduce job
- The YARN resource manager, which coordinates the allocation of computer resources on the cluster
- The YARN node managers, which launch and monitor the compute containers on machines in the cluster
- The MapReduce application master, which coordinates the tasks running the MapReduce job. The application master and the MapReduce tasks run in containers that are scheduled by the resource manager and managed by the node managers.
- The DFS (normally HDFS), which is used for sharing job files between the other entities.

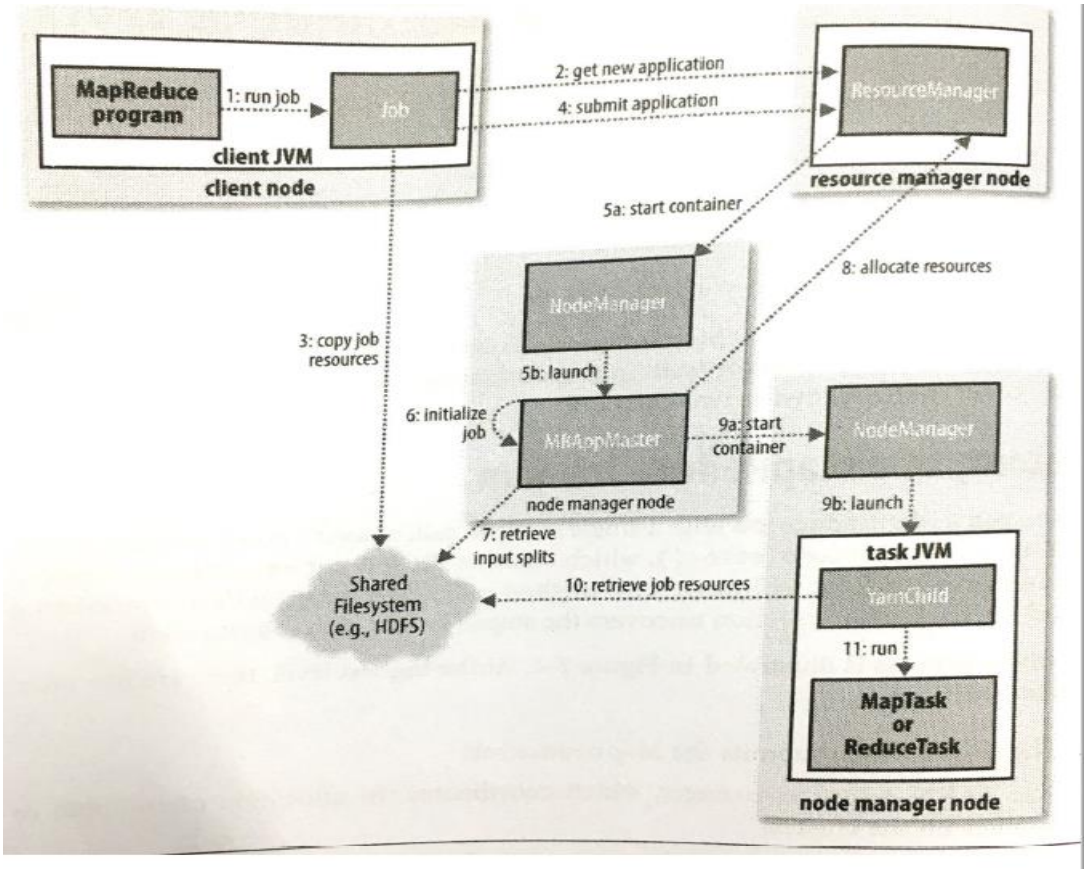


Fig. 3.33 Anatomy of MapReduce Job Run

Job Submission

- The *submit()* method on job creates an internal *JobSubmitter* instance and calls *submitJobInternal()* on it (step 1 in figure)
- Having submitted the job, *waitForCompletion()* polls the job's progress once per second and reports the progress to the console if it has changed since last report
- When the job completes successfully, 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 *JobSubmitter* does the following
- Asks the resource manager for a new application ID, used for the MapReduce job ID(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 of the input paths don't exist, for example), 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 shared filesystem in a directory named after the job ID (step 3). The job JAR is copied with a high replication factor so that there are lots of copies across the cluster for the node managers to access when they run tasks for the job.
- Submits the job by calling *submitApplication()* on the resource manager (step 4)

Job Initialization

- When the resource manager receives a call to its *submitApplication()* method, it handsoff the request to the YARN scheduler.
- The scheduler allocates a container, and the resource manager then launches the application master's process there, under the node manager's management (steps 5a and 5b)
- The application master for MapReduce jobs is a Java application whose main class is*MRAppMaster*.
- It initializes the job by creating a number of bookkeeping objects to keep track of the job's progress, as it will receive progress and completion reports from the task (step 6)
- Next, it retrieves the input splits computed in the client from the shared filesystem (step 7)
- It then creates a map task object for each split, as well as a number of reduce task objects determined by the *mapreduce.job.reduces* property (set by the *setNumReduceTasks()* method on Job). Tasks are given IDs at this point
- The application master must decide how to run the tasks that make up the MapReduce job
- If the job is small, the application master may choose to run the tasks in the same JVM as itself.
- This happens when it judges that the overhead of allocating and running tasks in new containers outweighs the gain to be had in running them in parallel, compared to running them sequentially on one node. Such a job is said to be uberized or run as an uber task.

What qualifies as a small job?

- By default,
- a small job is one that has
  - less than 10 mappers,
  - only one reducer and
  - an input size that is less than the size of one HDFS block
- Before it can run the task, it localizes the resources that the task needs, including the job configuration and JAR file and any files from the distributed cache (step 10).
- Finally it runs the map or reduce task (step 11)
- The YarnChild runs in a dedicated JVM, so that any bugs in the user-defined map and reduce functions don't affect the node manager – by causing it to crash or hang.

Streaming

- Streaming runs special map and reduce tasks for the purpose of launching the user-supplied executable and communicating with it

Job Completion

- When the application master receives a notification that the last task for a job is complete, it changes the status for the job to "successful"
- Finally, on job completion, the application master and the task containers clean up their working state

11. Failures

- In the real world, user code is buggy, processes crash and machines fail

One of the major benefits of using Hadoop is

- its ability to handle such failures and allow the job to complete successfully

- Failure of any of the following entities – considered
  - The task
  - The application master
  - The node manager
  - The resource manager

Task Failure

- The most common occurrence of this failure is when user code in the map or reduce task throws a runtime exception
- If this happens, the task JVM reports the error back to its parent application master before it exits
- The error ultimately makes it into the user logs
- The application master marks the task attempt as failed, and frees up the container so its resources are available for another task
- Another failure – sudden exit of the task JVM
- In this case, the node manager notices that the process has exited and informs the application master so it can mark the attempt as failed
- When the application master is notified of a task attempt that has failed, it will reschedule the execution of the task

Application Master Failure

- Just like MapReduce tasks are given several attempts to succeed, applications in YARN are retired in the event of failure
- The maximum number of attempts to run a MapReduce application master is controlled by the *mapreduce.am.max.attempts*property
- The default value is 2, so if a MapReduce application master fails twice it will not be tried again and the job will fail
- YARN imposes a limit for the maximum number of attempts for any YARN application
- The limit is set by *yarn.resourcemanager.am.max.attempts*and defaults to 2

Node Manager Failure

- If a node manager fails by crashing or running very slowly, it will stop sending heartbeats to the resource manager
- The resource manager will notice a node manager that has stopped sending heartbeats if it hasn't received one for 10 minutes; - remove it from its pool of nodes to schedule containers on
- Any task or application master running on the failed node manager will be recovered
- Node managers may be blacklisted if the number of failures for the application is high

Resource Manager Failure

- Failure of the resource manager is serious because without it, neither jobs nor task containers can be launched
- In the default configuration, the resource manager is a single point of failure, since in the event of machine failure, all running jobs fail – and can't be recovered
- To achieve High Availability (HA), it is necessary to run a pair of resource managers in an active-standby configuration
- If the active resource manager fails, then the standby can take over without a significant interruption to the client
- Information about all the running applications is stored in a highly available state store, so that the standby can recover the core state of the failed active resource manager.

- Node manager information is not stored in the state store since it can be reconstructed relatively quickly by the new resource manager as the node managers send their first heartbeats
- When the new resource manager starts, it reads the application information from the state store, then restarts the application masters for all applications running on the cluster
- The transition of a resource manager from standby to active is handled by a failover controller
- Clients and node managers must be configured to handle resource manager failover, since there are now two possible resource managers to communicate with
- They try connecting to each resource manager in a round-robin fashion until they find the active one
- If the active fails, then they will retry until the standby becomes active

12. Job Scheduling

- Early versions of Hadoop had a very simple approach to scheduling users' jobs: they ran in order of submission, using a FIFO scheduler.
- Each job would use the whole cluster - so jobs had to wait for their turn.
- The problem of sharing resources fairly between users requires a better scheduler.
- Production jobs need to complete in a timely manner, while allowing users who are making smaller ad hoc queries to get results back in a reasonable time.
- Later on, the ability to set a job's priority was added, via the *mapred.job.priority*property or the *setJobPriority()* method on JobClient (both of which take one of the values *VERY_HIGH, HIGH, NORMAL, LOW, VERY_LOW*).
- When the job scheduler is choosing the next job to run, it selects one with the highest priority.
- However, with the FIFO scheduler, priorities do not support *preemption*
- so a high-priority job can still be blocked by a long-running low priority job that started before the high-priority job was scheduled.
- MapReduce in Hadoop comes with a choice of schedulers.
- The default is the original FIFO queue-based scheduler, and
- there are also multiuser schedulers called
    - the Fair Scheduler and
    - the Capacity Scheduler.

The Fair Scheduler

- aims to give every user a fair share of the cluster capacity over time.
- If a single job is running, it gets all of the cluster.
- As more jobs are submitted, free task slots are given to the jobs in such a way as to give each user a fair share of the cluster.
- A short job belonging to one user will complete in a reasonable time even while another user's long job is running, and the long job will still make progress.
- Jobs are placed in pools, and by default, each user gets their own pool.
- It is also possible to define custom pools with guaranteed minimum capacities defined in terms of the number of map and reduce slots, and to set weightings for each pool.

- The Fair Scheduler supports preemption, so if a pool has not received its fair share for a certain period of time, then the scheduler will kill tasks in pools running over capacity in order to give the slots to the pool running under capacity.
- The Fair Scheduler is a *"contrib"* module.
- To enable it, place its JAR file on Hadoop's classpath, by copying it from Hadoop's *contrib/fairscheduler*directory to the *lib* directory.
- Then set the mapred.jobtracker.taskScheduler property to:

  org.apache.hadoop.mapred.FairScheduler

The Capacity Scheduler

- The Capacity Scheduler takes a slightly different approach to multiuser scheduling.
- A cluster is made up of a number of queues (like the Fair Scheduler's pools), which may be hierarchical (so a queue may be the child of another queue), and each queue has an allocated capacity.
- This is like the Fair Scheduler, except that within each queue, jobs are scheduled using FIFO scheduling (with priorities).
- In effect, the Capacity Scheduler allows users or organizations (defined using queues) to simulate a separate MapReduce cluster with FIFO scheduling for each user or organization.
- The Fair Scheduler enforces fair sharing within each pool, so running jobs share the pool's resources.

13.Shuffle & Sort

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
- In many ways, the shuffle is the heart of MapReduce and is where the "magic" happens.

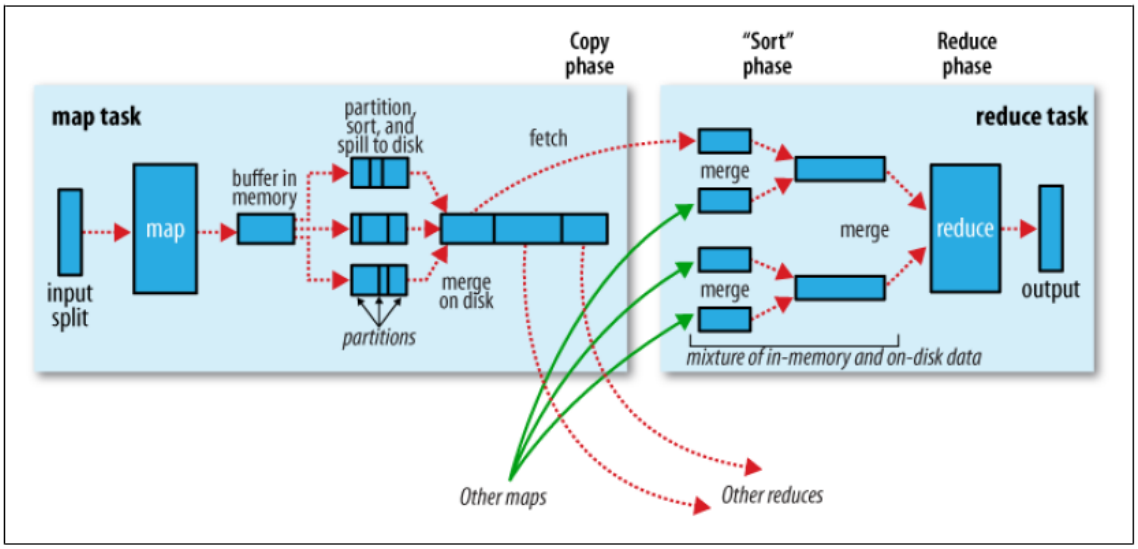


Fig. 3.34 MapReduce

14. Task Execution

Speculative Execution

- The MapReduce model is to break jobs into tasks and run the tasks in parallel to make the overall job execution time smaller than it would otherwise be if the tasks ran sequentially.
- This makes job execution time sensitive to slow-running tasks, as it takes only one slow task to make the whole job take significantly longer than it would have done otherwise.
- When a job consists of hundreds or thousands of tasks, the possibility of a few straggling tasks is very real.
- Tasks may be slow for various reasons, including hardware degradation or software mis-configuration, but the causes may be hard to detect since the tasks still complete successfully.
- Hadoop doesn't try to diagnose and fix slow-running tasks; instead, it tries to detect when a task is running slower than expected and launches another, equivalent, task as a backup.
- This is termed speculative execution of tasks.
- It's important to understand that speculative execution does not work by launching two duplicate tasks at about the same time so they can race each other.
- This would be wasteful of cluster resources.
- When a task completes successfully, any duplicate tasks that are running are killed since they are no longer needed.
- So if the original task completes before the speculative task, then the speculative task is killed; on the other hand, if the speculative task finishes first, then the original is killed.
- Speculative execution is an optimization, not a feature to make jobs run more reliably.
- If there are bugs that sometimes cause a task to hang or slow down, then relying on speculative execution to avoid these problems is unwise, and won't work reliably, since the same bugs are likely to affect the speculative task.
- Speculative execution is turned on by default.
- It can be enabled or disabled independently for map tasks and reduce tasks, on a cluster-wide basis, or on a per-job basis.

15. MapReduce Features

Hadoop MapReduce

MapReduce is a programming model suitable for processing of huge data. Hadoop is capable of running MapReduce programs written in various languages: Java, Ruby, Python, and C++. MapReduce programs are parallel in nature, thus are very useful for performing large-scale data analysis using multiple machines in the cluster.

MapReduce programs work in two phases:

1. Map phase
2. Reduce phase.

An input to each phase is key-value pairs. In addition, every programmer needs to specify two functions: map function and reduce function.

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

Let's understand this with an example –

Consider you have following input data for your Map Reduce Program

```
Welcome to Hadoop Class
Hadoop is good
Hadoop is bad
```

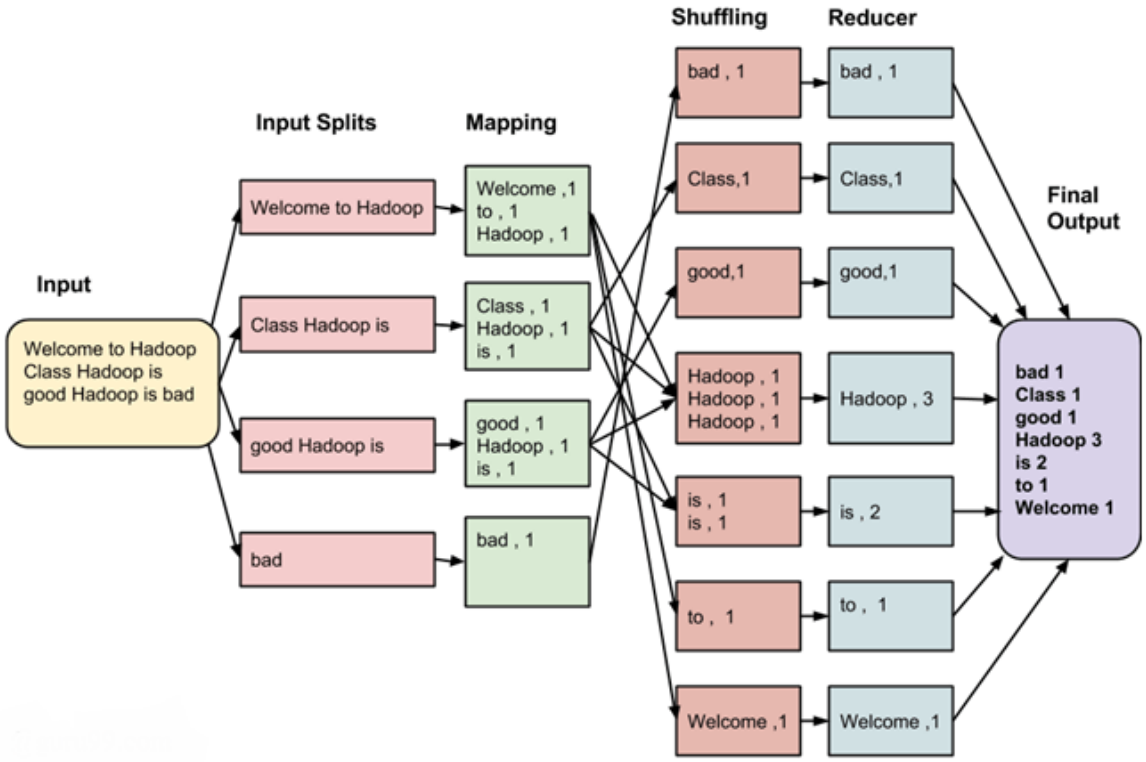


Fig. 3.35 MapReduce Architecture

The final output of the MapReduce task is

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

The data goes through the following phases

Input Splits:

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

Mapping

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

Shuffling

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

Reducing

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

MapReduce Architecture explained -

- One map task is created for each split which then executes map function for each record in the split.
- It is always beneficial to have multiple splits because the time taken to process a split is small as compared to the time taken for processing of the whole input. When the splits are smaller, the processing is better to load balanced since we are processing the splits in parallel.
- However, it is also not desirable to have splits too small in size. When splits are too small, the overload of managing the splits and map task creation begins to dominate the total job execution time.
- For most jobs, it is better to make a split size equal to the size of an HDFS block (which is 64 MB, by default).
- Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
- Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of HDFS store operation.
- Map output is intermediate output which is processed by reduce tasks to produce the final output.
- Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill.
- In the event of node failure, before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.
- Reduce task doesn't work on the concept of data locality. An output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
- On this machine, the output is merged and then passed to the user-defined reduce function.

- Unlike the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes). So, writing the reduce output

How MapReduce Organizes Work?

Hadoop divides the job into tasks. There are two types of tasks:

1. Map tasks (Splits & Mapping)
2. Reduce tasks (Shuffling, Reducing)

as mentioned above.

The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called a

1. Jobtracker: Acts like a master (responsible for complete execution of submitted job)
2. Multiple Task Trackers: Acts like slaves, each of them performing the job

For every job submitted for execution in the system, there is one Jobtracker that resides on Namenode and there are multiple tasktrackers which reside on Datanode.

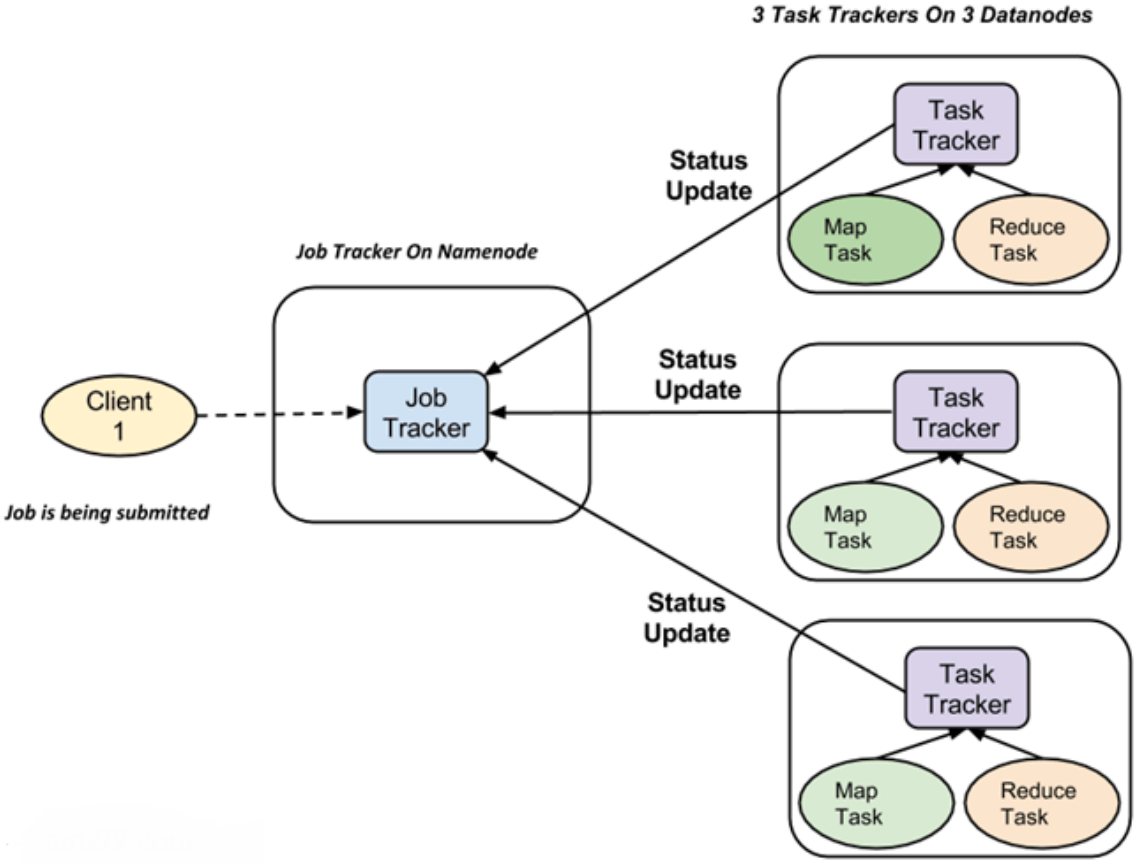


Fig. 3.34 HDFS Architecture

- A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
- It is the responsibility of job tracker to coordinate the activity by scheduling tasks to run on different data nodes.
- Execution of individual task is then to look after by task tracker, which resides on every data node executing part of the job.
- Task tracker's responsibility is to send the progress report to the job tracker.

- In addition, task tracker periodically sends 'heartbeat' signal to the Jobtracker so as to notify him of the current state of the system.
- Thus job tracker keeps track of the overall progress of each job. In the event of task failure, the job tracker can reschedule it on a different task tracker.

Word Count Program with MapReduce and Java

In Hadoop, MapReduce is a computation that decomposes large manipulation jobs into individual tasks that can be executed in parallel across a cluster of servers. The results of tasks can be joined together to compute final results.

MapReduce consists of 2 steps:

- Map Function – It takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (Key-Value pair).
- Reduce Function – Takes the output from Map as an input and combines those data tuples into a smaller set of tuples.

Example – (Map function in Word Count)

Input	Set of data	Bus, Car, bus, car, train, car, bus, car, train, bus, TRAIN,BUS, buS, caR, CAR, car, BUS, TRAIN
Output	Convert into another set of data (Key,Value)	(Bus,1), (Car,1), (bus,1), (car,1), (train,1), (car,1), (bus,1), (car,1), (train,1), (bus,1), (TRAIN,1),(BUS,1), (buS,1), (caR,1), (CAR,1), (car,1), (BUS,1), (TRAIN,1)

Example – (Reduce function in Word Count)

Input (output of Map function)	Set of Tuples	(Bus,1), (Car,1), (bus,1), (car,1), (train,1), (car,1), (bus,1), (car,1), (train,1), (bus,1), (TRAIN,1),(BUS,1), (buS,1), (caR,1), (CAR,1), (car,1), (BUS,1), (TRAIN,1)
Output	Converts into smaller set of tuples	(BUS,7), (CAR,7), (TRAIN,4)

Work Flow of the Program

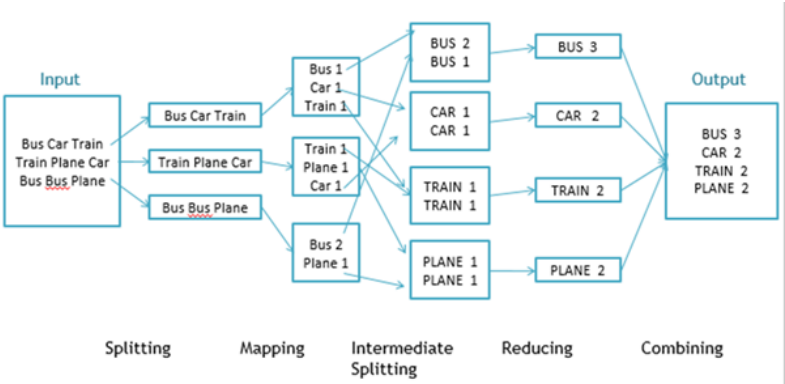


Fig. 3.35 Work Flow of the Program

Workflow of MapReduce consists of 5 steps:

1. Splitting – The splitting parameter can be anything, e.g. splitting by space, comma, semicolon, or even by a new line ('\n').
2. Mapping – as explained above.
3. Intermediate splitting – the entire process in parallel on different clusters. In order to group them in "Reduce Phase" the similar KEY data should be on the same cluster.
4. Reduce – it is nothing but mostly group by phase.
5. Combining – The last phase where all the data (individual result set from each cluster) is combined together to form a result.

Steps

1. Open Eclipse> File > New > Java Project >(Name it – MRProgramsDemo) > Finish.
2. Right Click > New > Package (Name it - PackageDemo) > Finish.
3. Right Click on Package > New > Class (Name it - WordCount).
4. Add Following Reference Libraries:
    1. Right Click on Project > Build Path> Add External
        1. */usr/lib/hadoop-0.20/*hadoop-core.jar
        2. *Usr/lib/hadoop-0.20/lib/*Commons-cli-1.2.jar

5. Type the following code:

```
packagePackageDemo;
importjava.io.IOException;
importorg.apache.hadoop.conf.Configuration;
importorg.apache.hadoop.fs.Path;
importorg.apache.hadoop.io.IntWritable;
importorg.apache.hadoop.io.LongWritable;
importorg.apache.hadoop.io.Text;
importorg.apache.hadoop.mapreduce.Job;
importorg.apache.hadoop.mapreduce.Mapper;
importorg.apache.hadoop.mapreduce.Reducer;
importorg.apache.hadoop.mapreduce.lib.input.FileInputFormat;
```

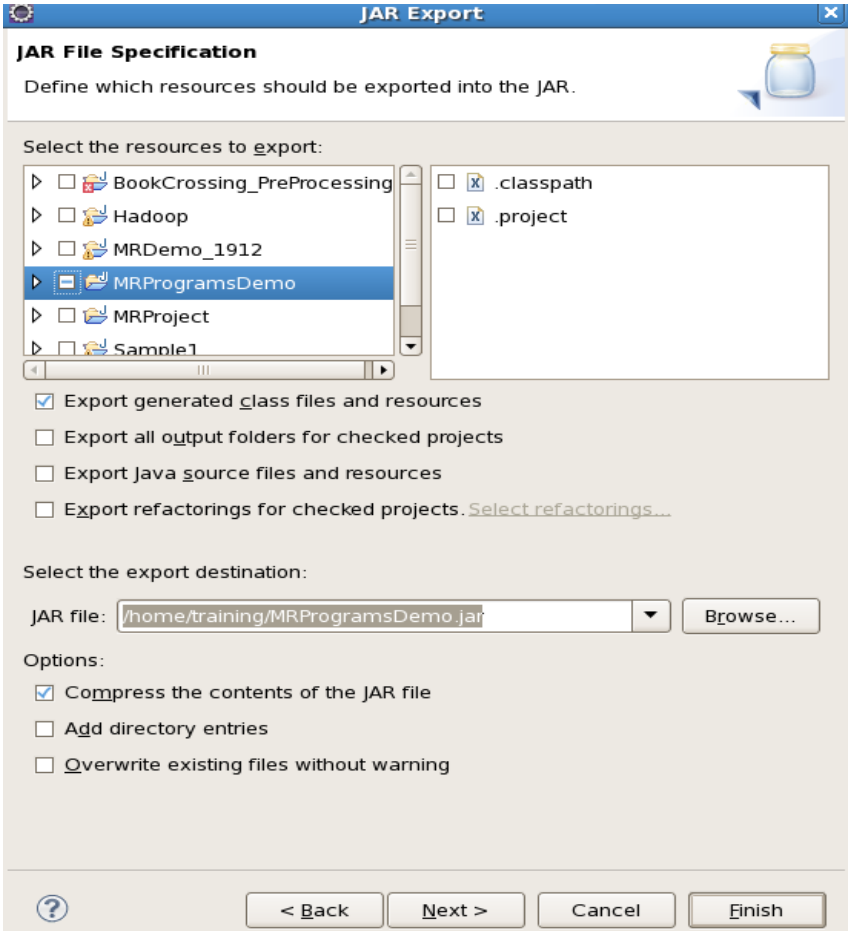
```
importorg.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
importorg.apache.hadoop.util.GenericOptionsParser;
publicclassWordCount {
publicstaticvoidmain(String [] args) throwsException
{
Configurationc=newConfiguration();
String[] files=newGenericOptionsParser(c,args).getRemainingArgs();
Pathinput=newPath(files[0]);
Pathoutput=newPath(files[1]);
Jobj=newJob(c,"wordcount");
j.setJarByClass(WordCount.class);
j.setMapperClass(MapForWordCount.class);
j.setReducerClass(ReduceForWordCount.class);
j.setOutputKeyClass(Text.class);
j.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(j, input);
FileOutputFormat.setOutputPath(j, output);
System.exit(j.waitForCompletion(true)?0:1); }
publicstaticclassMapForWordCountextendsMapper<LongWritable, Text, Text, IntWritable>{
public void map(LongWritablekey, Textvalue, Contextcon) throwsIOException,
InterruptedException
{
Stringline=value.toString();
String[] words=line.split(",");
for(Stringword: words )
{
TextoutputKey=newText(word.toUpperCase().trim());
IntWritableoutputValue=newIntWritable(1);
con.write(outputKey, outputValue); }
}
}
public static classReduceForWordCountextendsReducer<Text, IntWritable, Text,
IntWritable>
{
public void reduce(Textword, Iterable<IntWritable>values, Contextcon) throwsIOException,
InterruptedException
{
Int sum=0;
for(IntWritablevalue : values)
  {
sum+=value.get();
  }
con.write(word, newIntWritable(sum)); }
}
}
```

The above program consists of three classes:

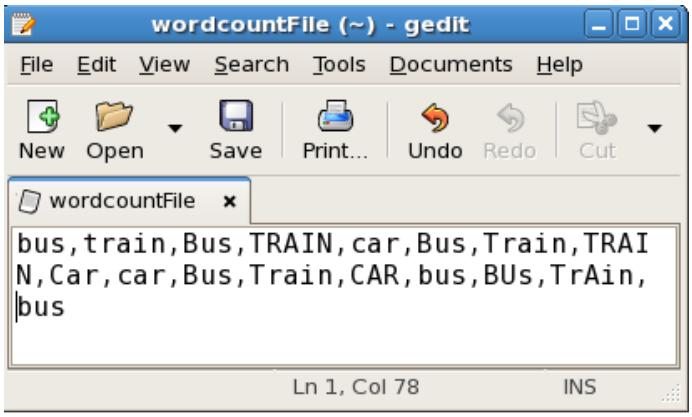
- Driver class (Public, void, static, or main; this is the entry point).
- The Map class which extends the public class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT> and implements the Map function.
- The Reduce class which extends the public class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT> and implements the Reduce function.

6. Make a jar file

Right Click on Project> Export> Select export destination as Jar File > next> Finish.



7. Take a text file and move it into HDFS format:



To move this into Hadoop directly, open the terminal and enter the following commands:

```
[training@localhost ~]$ hadoop fs -putwordcountFilewordCountFile
```

8. Run the jar file:
(Hadoop jar jarfilename.jar packageName.ClassName PathToInputTextFilePathToOutputDirectry)

```
[training@localhost ~]$ hadoop jar MRProgramsDemo.jar PackageDemo.WordCountwordCountFile MRDir1
```

9. Open the result:

```
[training@localhost ~]$ hadoop fs -ls MRDir1
Found 3 items
-rw-r--r--1 training supergroup 02016-02-2303:36 /user/training/MRDir1/_SUCCESS
drwxr-xr-x - training supergroup 02016-02-2303:36 /user/training/MRDir1/_logs
-rw-r--r--1 training supergroup 202016-02-2303:36 /user/training/MRDir1/part-r-00000
[training@localhost ~]$ hadoop fs -cat MRDir1/part-r-00000
BUS 7
CAR 4
TRAIN 6
```

Basic Command and Syntax for MapReduce

Java Program:

```
$HADOOP_HOME/bin/hadoop jar / org.myorg.WordCount /input-directory /output-directory
```

Other Scripting languages:

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar -input myInputDirs -output myOutputDir -mapper /bin/cat -reducer /bin/wc
```

Input parameter mentions the input files and the output parameter mentions the output directory. Mapper and Reducer mention the algorithm for Map function and Reduce function respectively. These have to be mentioned in case Hadoop streaming API is used i.e; the mapper and reducer are written in scripting language. The commands remain the same as for Hadoop. The jobs can also be submitted using jobs command in Hadoop. All the parameters for the specific task has to be mentioned in a file called 'job-file' and submitted to Hadoop using the following command. The following are the commands that are useful when a job file is submitted:

```
hadoop job -list Displays all the ongoing jobs
hadoop job-status Prints the map and reduce completion percentage
hadoop job -kill Kills the corresponding job.
hadoop job -set-priority Set priority for Queued jobs
```

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

School of Computing

Department of Computer Science and Engineering

UNIT IV - Big Data – SBS1608

UNIT IV

Setting up a Hadoop Cluster - Cluster specification - Cluster Setup and Installation - Hadoop Configuration-Security in Hadoop - Administering Hadoop – HDFS - Monitoring-Maintenance

1. Setting Up a Hadoop Cluster

This explains how to set up Hadoop to run on a cluster of machines. Running HDFS and MapReduce on a single machine is great for learning about these systems, but to do useful work they need to run on multiple nodes. There are a few options when it comes to getting a Hadoop cluster, from building your own to running on rented hardware, or using an offering that provides Hadoop as a service in the cloud.

2. Cluster Specification

Hadoop is designed to run on commodity hardware. That means that you are not tied to expensive, proprietary offerings from a single vendor; rather, you can choose standardized, commonly available hardware from any of a large range of vendors to build your cluster. "Commodity" does not mean "low-end." Low-end machines often have cheap components, which have higher failure rates than more expensive (but still commodity class) machines. When you are operating tens, hundreds, or thousands of machines, cheap components turn out to be a false economy, as the higher failure rate incurs a greater maintenance cost. On the other hand, large database class machines are not recommended either, since they don't score well on the price/performance curve. And even though you would need fewer of them to build a cluster of comparable performance to one built of mid-range commodity hardware, when one did fail it would have a bigger impact on the cluster, since a larger proportion of the cluster hardware would be unavailable.

Hardware specifications rapidly become obsolete, but for the sake of illustration, a typical choice of machine for running a Hadoop datanode and tasktracker in mid-2010 would have the following specifications:

Processor
2 quad-core 2-2.5GHz CPUs
Memory
16-24 GB ECC RAM*
Storage
4 × 1TB SATA disks
Network
Gigabit Ethernet

While the hardware specification for your cluster will assuredly be different, Hadoop is designed to use multiple cores and disks, so it will be able to take full advantage of more powerful hardware. The bulk of Hadoop is written in Java, and can therefore run on any platform with a JVM, although there are enough parts that harbor Unix assumptions (the control scripts, for example) to make it unwise to run on a non-Unix platform in production. In fact, Windows operating systems are not supported production platforms. How large should your cluster be? There isn't an exact answer to this question, but the beauty of Hadoop is that you can start with a small cluster (say, 10 nodes) and grow it as your storage and computational

needs grow. In many ways, a better question is this: how fast does my cluster need to grow? You can get a good feel for this by considering storage capacity.

For example, if your data grows by 1 TB a week, and you have three-way HDFS replication, then you need an additional 3 TB of raw storage per week. Allow some room for intermediate files and logfiles (around 30%, say), and this works out at about one machine (2010 vintage) per week, on average. In practice, you wouldn't buy a new machine each week and add it to the cluster. The value of doing a back-of-the-envelope calculation like this is that it gives you a feel for how big your cluster should be: in this example, a cluster that holds two years of data needs 100 machines.

For a small cluster (on the order of 10 nodes), it is usually acceptable to run the namenode and the jobtracker on a single master machine (as long as at least one copy of the namenode's metadata is stored on a remote filesystem). As the cluster and the number of files stored in HDFS grow, the namenode needs more memory, so the namenode and jobtracker should be moved onto separate machines. The secondary namenode can be run on the same machine as the namenode, but again for reasons of memory usage (the secondary has the same memory requirements as the primary), it is best to run it on a separate piece of hardware, especially for larger clusters. Machines running the namenodes should typically run on 64-bit hardware to avoid the 3 GB limit on Java heap size in 32-bit architectures.

Network Topology

A common Hadoop cluster architecture consists of a two-level network topology, as illustrated in figure 1. Typically there are 30 to 40 servers per rack, with a 1 GB switch for the rack (only three are shown in the diagram), and an uplink to a core switch or router (which is normally 1 GB or better). The salient point is that the aggregate bandwidth between nodes on the same rack is much greater than that between nodes on different racks.

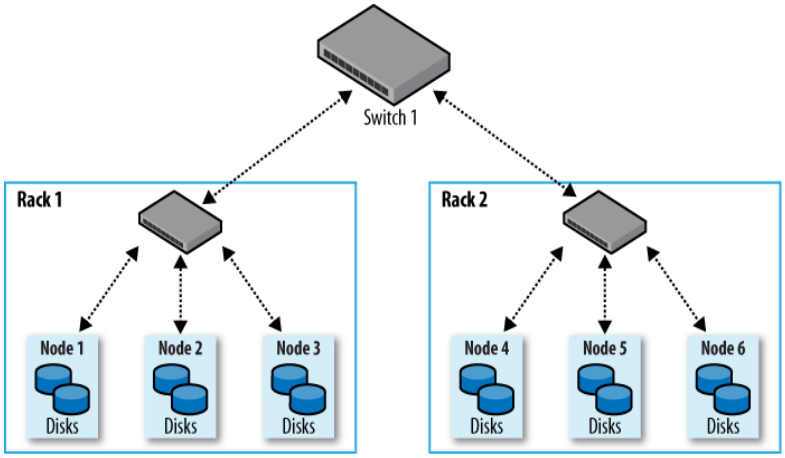


Fig. 4.1 Typical two-level network architecture for a Hadoop cluster

Rack awareness

To get maximum performance out of Hadoop, it is important to configure Hadoop so that it knows the topology of your network. If your cluster runs on a single rack, then there is nothing more to do, since this is the default. However, for multirack clusters, you need to map nodes to racks. By doing this, Hadoop will prefer within-rack transfers (where there is more bandwidth available) to off-rack transfers when placing MapReduce tasks on nodes. HDFS will be able to place replicas more intelligently to trade-off performance and resilience. Network locations such as nodes and racks are represented in a tree, which reflects the network "distance" between locations. The namenode uses the network location when determining

where to place block replicas; the jobtracker uses network location to determine where the closest replica is as input for a map task that is scheduled to run on a tasktracker. For the network in figure 4.1, the rack topology is described by two network locations, say, /switch1/rack1 and /switch1/rack2. Since there is only one top-level switch in this cluster, the locations can be simplified to /rack1 and /rack2. The Hadoop configuration must specify a map between node addresses and network locations.The map is described by a Java interface, DNSToSwitchMapping, whose signature is:

```
public interface DNSToSwitchMapping {
public List<String> resolve(List<String> names);
}
```

The names parameter is a list of IP addresses, and the return value is a list of corresponding network location strings. The topology.node.switch.mapping.implconfiguration property defines an implementation of the DNSToSwitchMappinginterface that the namenode and the jobtracker use to resolve worker node network locations. For the network in our example, we would map node1, node2, and node3 to /rack1, and node4, node5, and node6 to /rack2. Most installations don't need to implement the interface themselves, however, since the default implementation is ScriptBasedMapping, which runs a user-defined script to determine the mapping. The script's location is controlled by the property topology.script.file.name. The script must accept a variable number of arguments that are the hostnames or IP addresses to be mapped, and it must emit the corresponding network locations to standard output, separated by whitespace. The Hadoop wiki has an example at http://wiki.apache.org/hadoop/topology_rack_awareness_scripts. If no script location is specified, the default behaviour is to map all nodes to a single network location, called /default-rack.

3. Cluster Setup and Installation

The next steps are to get it racked up and install the software needed to run Hadoop. There are various ways to install and configure Hadoop.

To ease the burden of installing and maintaining the same software on each node, it is normal to use an automated installation method like Red Hat Linux's Kickstart or Debian's Fully Automatic Installation. These tools allow you to automate the operating system installation by recording the answers to questions that are asked during the installation process (such as the disk partition layout), as well as which packages to install. Crucially, they also provide hooks to run scripts at the end of the process, which are invaluable for doing final system tweaks and customization that is not covered by the standard installer. The following sections describe the customizations that are needed to run Hadoop. These should all be added to the installation script.

Installing Java

Java 6 or later is required to run Hadoop. The latest stable Sun JDK is the preferred option, although Java distributions from other vendors may work, too. The following command confirms that Java was installed correctly:

```
% java -version
java version "1.6.0_12"
Java(TM) SE Runtime Environment (build 1.6.0_12-b04)
Java HotSpot(TM) 64-Bit Server VM (build 11.2-b01, mixed mode)
```

Creating a Hadoop User

It's good practice to create a dedicated Hadoop user account to separate the Hadoop installation from other services running on the same machine. Some cluster administrators choose to make this user's home directory an NFSmounted drive, to aid with SSH key distribution (see the following discussion). The NFS server is typically outside the Hadoop cluster. If you use NFS, it is worth considering autofs, which allows you to mount the NFS filesystem on demand, when the system accesses it. Autofs provides some protection against the NFS server failing and allows you to use replicated filesystems for failover. There are other NFS gotchas to watch out for, such as synchronizing UIDs and GIDs. For help setting up NFS on Linux, refer to the HOW TO at http://nfs.sourceforge.net/nfs-howto/index.html.

Installing Hadoop

Download Hadoop from the Apache Hadoop releases page (http://hadoop.apache.org/core/releases.html), and unpack the contents of the distribution in a sensible location, such as /usr/local (/opt is another standard choice). Note that Hadoop is not installed in the hadoopuser's home directory, as that may be an NFS-mounted directory:

```
% cd /usr/local
% sudo tar xzf hadoop-x.y.z.tar.gz
```

The owner of the Hadoop files will be changed to be the hadoopuser and group:

```
% sudochown -R hadoop:hadoophadoop-x.y.z
```

Testing the Installation

Once the installation file is created, you are ready to test it by installing it on the machines in your cluster. This will probably take a few iterations as you discover kinks in the install. When it's working, you can proceed to configure Hadoop and give it a test run. This process is documented in the following sections.

SSH Configuration

The Hadoop control scripts rely on SSH to perform cluster-wide operations. For example, there is a script for stopping and starting all the daemons in the cluster. Note that the control scripts are optional—cluster-wide operations can be performed by other mechanisms, too (such as a distributed shell). To work seamlessly, SSH needs to be set up to allow password-less login for the hadoopuser from machines in the cluster. The simplest way to achieve this is to generate a public/private key pair, and it will be shared across the cluster using NFS.

First, generate an RSA key pair by typing the following in the hadoopuser account:

```
% ssh-keygen -t rsa -f ~/.ssh/id_rsa
```

Even though we want password-less logins, keys without passphrases are not considered good practice (it's OK to have an empty passphrase when running a local pseudodistributed cluster), so we specify a passphrase when prompted for one. We shall use ssh-agent to avoid the need to enter a password for each connection. The private key is in the file specified by the -f option, ~/.ssh/id_rsa, and the public key is stored in a file with the same name with .pub appended, ~/.ssh/id_rsa.pub.

Next we need to make sure that the public key is in the ~/.ssh/authorized_keysfile on all the machines in the cluster that we want to connect to. If the hadoopuser's home directory is an NFS filesystem, as described earlier, then the keys can be shared across the cluster by typing:

```
% cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

If the home directory is not shared using NFS, then the public keys will need to be shared by some other means.

Test that you can SSH from the master to a worker machine by making sure sshagent is running and then run ssh-add to store your passphrase. You should be able to ssh to a worker without entering the passphrase again.

4. Hadoop Configuration

There are a handful of files for controlling the configuration of a Hadoop installation; the most important ones are listed in Table 4.1.

Table 4.1 Hadoop configuration files

Filename	Format	Description
hadoop-env.sh	Bash script	Environment variables that are used in the scripts to run Hadoop.
core-site.xml	Hadoop configuration XML	Configuration settings for Hadoop Core, such as I/O settings that are common to HDFS and MapReduce.
hdfs-site.xml	Hadoop configuration XML	Configuration settings for HDFS daemons: the namenode, the secondary namenode, and the datanodes.
mapred-site.xml	Hadoop configuration XML	Configuration settings for MapReduce daemons: the jobtracker, and the tasktrackers.
masters	Plain text	A list of machines (one per line) that each run a secondary namenode.
slaves	Plain text	A list of machines (one per line) that each run a datanode and a tasktracker.
hadoop-metrics.properties	Java Properties	Properties for controlling how metrics are published in Hadoop
log4j.properties	Java Properties	Properties for system logfiles, the namenode audit log, and the task log for the tasktracker child process

These files are all found in the conf directory of the Hadoop distribution. The configuration directory can be relocated to another part of the filesystem (outside the Hadoop installation, which makes upgrades marginally easier) as long as daemons are started with the --config option specifying the location of this directory on the local filesystem.

Configuration Management

Hadoop does not have a single, global location for configuration information. Instead, each Hadoop node in the cluster has its own set of configuration files, and it is up to administrators to ensure that they are kept in sync across the system. Hadoop provides a rudimentary facility for synchronizing configuration using rsync; alternatively, there are parallel shell tools that can help do this, like dshor pdsh.

Hadoop is designed so that it is possible to have a single set of configuration files that are used for all master and worker machines. The great advantage of this is simplicity, both conceptually (since there is only one configuration to deal with) and operationally (as the Hadoop scripts are sufficient to manage a single configuration setup).

For some clusters, the one-size-fits-all configuration model breaks down. For example, if you expand the cluster with new machines that have a different hardware specification to the existing ones, then you need a different configuration for the new machines to take advantage of their extra resources.

In these cases, you need to have the concept of a class of machine, and maintain a separate configuration for each class. Hadoop doesn't provide tools to do this, but there are several excellent tools for doing precisely this type of configuration management, such as Chef, Puppet, cfengine, and bcfg2.

For a cluster of any size, it can be a challenge to keep all of the machines in sync: consider what happens if the machine is unavailable when you push out an update—who

ensures it gets the update when it becomes available? This is a big problem and can lead to divergent installations, so even if you use the Hadoop control scripts for managing Hadoop, it may be a good idea to use configuration management tools for maintaining the cluster. These tools are also excellent for doing regular maintenance, such as patching security holes and updating system packages.

Control scripts

Hadoop comes with scripts for running commands, and starting and stopping daemons across the whole cluster. To use these scripts (which can be found in the bin directory), you need to tell Hadoop which machines are in the cluster. There are two files for this purpose, called masters and slaves, each of which contains a list of the machine hostnames or IP addresses, one per line. The masters file is actually a misleading name, in that it determines which machine or machines should run a secondary namenode. The slaves file lists the machines that the datanodes and tasktrackers should run on. Both masters and slaves files reside in the configuration directory, although the slaves file may be placed elsewhere (and given another name) by changing the HADOOP_SLAVES setting in hadoop-env.sh. Also, these files do not need to be distributed to worker nodes, since they are used only by the control scripts running on the namenode or jobtracker. You don't need to specify which machine (or machines) the namenode and jobtracker runs on in the masters file, as this is determined by the machine the scripts are run on. For example, the start-dfs.sh script, which starts all the HDFS daemons in the cluster, runs the namenode on the machine the script is run on. In slightly more detail, it:

1. Starts a namenode on the local machine (the machine that the script is run on)
2. Starts a datanode on each machine listed in the slaves file
3. Starts a secondary namenode on each machine listed in the masters file

There is a similar script called start-mapred.sh, which starts all the MapReduce daemons in the cluster. More specifically, it:

1. Starts a jobtracker on the local machine
2. Starts a tasktracker on each machine listed in the slaves file

Note that a master is not used by the MapReduce control scripts.

Also provided are stop-dfs.sh and stop-mapred.sh scripts to stop the daemons started by the corresponding start script. These scripts start and stop Hadoop daemons using the hadoop-daemon.sh script. If you use the aforementioned scripts, you shouldn't call hadoop-daemon.sh directly. But if you need to control Hadoop daemons from another system or from your own scripts, then the hadoop-daemon.sh script is a good integration point. Likewise, hadoopdaemons.sh (with an "s") is handy for starting the same daemon on a set of hosts.

Depending on the size of the cluster, there are various configurations for running the master daemons: the namenode, secondary namenode, and jobtracker. On a small cluster (a few tens of nodes), it is convenient to put them on a single machine; however, as the cluster gets larger, there are good reasons to separate them.

The namenode has high memory requirements, as it holds file and block metadata for the entire namespace in memory. The secondary namenode, while idle most of the time, has a comparable memory footprint to the primary when it creates a checkpoint. For filesystems with a large number of files, there may not be enough physical memory on one machine to run both the primary and secondary namenode. The secondary namenode keeps a copy of the latest checkpoint of the filesystem metadata that it creates. Keeping this (stale) backup on a different

node to the namenode allows recovery in the event of loss (or corruption) of all the namenode's metadata files.

On a busy cluster running lots of MapReduce jobs, the jobtracker uses considerable memory and CPU resources, so it should run on a dedicated node. Whether the master daemons run on one or more nodes, the following instructions apply:

• Run the HDFS control scripts from the namenode machine. The masters file should contain the address of the secondary namenode.

• Run the MapReduce control scripts from the jobtracker machine. When the namenode and jobtracker are on separate nodes, their slaves files need to be kept in sync, since each node in the cluster should run a datanode and a tasktracker.

Environment Settings

In this section, we consider how to set the variables in hadoop-env.sh.

Memory

By default, Hadoop allocates 1000 MB (1 GB) of memory to each daemon it runs. This is controlled by the HADOOP_HEAPSIZE setting in hadoop-env.sh. In addition, the task tracker launches separate child JVMs to run map and reduce tasks in, so we need to factor these into the total memory footprint of a worker machine.

The maximum number of map tasks that will be run on a tasktracker at one time is controlled by the mapred.tasktracker.map.tasks.maximumproperty, which defaults to two tasks. There is a corresponding property for reduce tasks, mapred.tasktracker.reduce.tasks.maximum, which also defaults to two tasks. The memory given to each of these child JVMs can be changed by setting the mapred.child.java.optsproperty. The default setting is -Xmx200m, which gives each task 200 MB of memory. The default configuration therefore uses 2,800 MB of memory for a worker machine.

Table 4.2 Worker node memory calculation

JVM	Default memory used (MB)	Memory used for 8 processors, 400 MB per child (MB)
Datanode	1,000	1,000
Tasktracker	1,000	1,000
Tasktracker child map task	2 × 200	7 × 400
Tasktracker child reduce task	2 × 200	7 × 400
Total	2,800	7,600

The number of tasks that can be run simultaneously on a tasktracker is governed by the number of processors available on the machine. Because MapReduce jobs are normally I/O-bound, it makes sense to have more tasks than processors to get better utilization. The amount of oversubscription depends on the CPU utilization of jobs you run, but a good rule of thumb is to have a factor of between one and two more tasks (counting both map and reduce tasks) than processors.

For example, if you had 8 processors and you wanted to run 2 processes on each processor, then you could set each of mapred.tasktracker.map.tasks.maximum and mapred.tasktracker.reduce.tasks.maximum to 7 (not 8, since the datanode and the tasktracker each take one slot).

If you also increased the memory available to each child task to 400 MB, then the total memory usage would be 7,600 MB (see Table 4.2).

Whether this Java memory allocation will fit into 8 GB of physical memory depends on the other processes that are running on the machine. If you are running Streaming or Pipes programs, this allocation will probably be inappropriate (and the memory allocated to the child should be dialed down), since it doesn't allow enough memory for users' (Streaming or Pipes) processes to run. The thing to avoid is processes being swapped out, as this it leads to severe performance degradation. The precise memory settings are necessarily very cluster-dependent and can be optimized over time with experience gained from monitoring the memory usage across the cluster. Tools like Ganglia are good for gathering this information. Hadoop also provides settings to control how much memory is used for MapReduce operations. For the master node, each of the namenode, secondary namenode, and jobtracker daemons uses 1,000 MB by default, a total of 3,000 MB.

Java

The location of the Java implementation to use is determined by the JAVA_HOME setting in hadoop-env.sh or from the JAVA_HOME shell environment variable, if not set in hadoopenv. sh. It's a good idea to set the value in hadoop-env.sh, so that it is clearly defined in one place and to ensure that the whole cluster is using the same version of Java.

System logfiles

System logfiles produced by Hadoop are stored in $HADOOP_INSTALL/logs by default. This can be changed using the HADOOP_LOG_DIR setting in hadoop-env.sh. It's a good idea to change this so that logfiles are kept out of the directory that Hadoop is installed in, since this keeps logfiles in one place even after the installation directory changes after an upgrade. A common choice is /var/log/hadoop, set by including the following line in hadoop-env.sh:

```
export HADOOP_LOG_DIR=/var/log/hadoop
```

The log directory will be created if it doesn't already exist (if not, confirm that the Hadoop user has permission to create it). Each Hadoop daemon running on a machine produces two logfiles. The first is the log output written via log4j. This file, which ends in .log, should be the first port of call when diagnosing problems, since most application log messages are written here. The standard Hadoop log4j configuration uses a Daily Rolling File Appender to rotate logfiles. Old logfiles are never deleted, so you should arrange for them to be periodically deleted or archived, so as to not run out of disk space on the local node. The second logfile is the combined standard output and standard error log. This logfile, which ends in .out, usually contains little or no output, since Hadoop uses log4j for logging. It is only rotated when the daemon is restarted, and only the last five logs are retained. Old logfiles are suffixed with a number between 1 and 5, with 5 being the oldest file.

Logfile names (of both types) are a combination of the name of the user running the daemon, the daemon name, and the machine hostname. For example, hadoop-tomdatanode-sturges.local.log.2008-07-04 is the name of a logfile after it has been rotated.

This naming structure makes it possible to archive logs from all machines in the cluster in a single directory, if needed, since the filenames are unique. The username in the logfile name is actually the default for the HADOOP_IDENT_STRING setting in hadoop-env.sh. If you wish to give the Hadoop instance a different identity for the purposes of naming the logfiles, change HADOOP_IDENT_STRING to be the identifier you want.

SSH settings

The control scripts allow you to run commands on (remote) worker nodes from the master node using SSH. It can be useful to customize the SSH settings, for various reasons. For example, you may want to reduce the connection timeout (using the ConnectTimeoutoption) so the control scripts don't hang around waiting to see whether a dead node is going to respond. Obviously, this can be taken too far. If the timeout is too low, then busy nodes will be skipped, which is bad.

Another useful SSH setting is StrictHostKeyChecking, which can be set to no to automatically add new host keys to the known hosts files. The default, ask, is to prompt the user to confirm they have verified the key fingerprint, which is not a suitable setting in a large cluster environment.

To pass extra options to SSH, define the HADOOP_SSH_OPTS environment variable in hadoop-env.sh. The Hadoop control scripts can distribute configuration files to all nodes of the cluster using rsync. This is not enabled by default, but by defining the HADOOP_MASTER setting in hadoop-env.sh, worker daemons will rsync the tree rooted at HADOOP_MASTER to the local node's HADOOP_INSTALL whenever the daemon starts up.

What if you have two masters—a namenode and a jobtracker on separate machines? You can pick one as the source and the other can rsync from it, along with all the workers. In fact, you could use any machine, even one outside the Hadoop cluster, to rsync from. Because HADOOP_MASTER is unset by default, there is a bootstrapping problem: how do we make sure hadoop-env.sh with HADOOP_MASTER set is present on worker nodes? For small clusters, it is easy to write a small script to copy hadoop-env.sh from the master to all of the worker nodes. For larger clusters, tools like dshcan do the copies in parallel. Alternatively, a suitable hadoop-env.sh can be created as a part of the automated installation script (such as Kickstart). When starting a large cluster with rsyncing enabled, the worker nodes can overwhelm the master node with rsync requests since the workers start at around the same time. To avoid this, set the HADOOP_SLAVE_SLEEP setting to a small number of seconds, such as 0.1, for one-tenth of a second. When running commands on all nodes of the cluster, the master will sleep for this period between invoking the command on each worker machine in turn.

Important Hadoop Daemon Properties

Hadoop has a bewildering number of configuration properties. In this section, we address the ones that you need to define (or at least understand why the default is appropriate) for any real-world working cluster. These properties are set in the Hadoop site files: core-site.xml, hdfs-site.xml, and mapred-site.xml. Example 1 shows a typical example set of files. Notice that most are marked as final, in order to prevent them from being overridden by job configurations.

Example 1. A typical set of site configuration files

```
<?xml version="1.0"?>
<!-- core-site.xml -->
<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://namenode/</value>
<final>true</final>
</property>
```

```
</configuration>
<?xml version="1.0"?>
<!-- hdfs-site.xml -->
<configuration>
<property>
<name>dfs.name.dir</name>
<value>/disk1/hdfs/name,/remote/hdfs/name</value>
<final>true</final>
</property>
<property>
<name>dfs.data.dir</name>
<value>/disk1/hdfs/data,/disk2/hdfs/data</value>
<final>true</final>
</property>
<property>
<name>fs.checkpoint.dir</name>
<value>/disk1/hdfs/namesecondary,/disk2/hdfs/namesecondary</value>
<final>true</final>
</property>
</configuration>
<?xml version="1.0"?>
<!-- mapred-site.xml -->
<configuration>
<property>
<name>mapred.job.tracker</name>
<value>jobtracker:8021</value>
<final>true</final>
</property>
<property>
<name>mapred.local.dir</name>
<value>/disk1/mapred/local,/disk2/mapred/local</value>
<final>true</final>
</property>
<property>
<name>mapred.system.dir</name>
<value>/tmp/hadoop/mapred/system</value>
<final>true</final>
</property>
<property>
<name>mapred.tasktracker.map.tasks.maximum</name>
<value>7</value>
<final>true</final>
</property>
<property>
<name>mapred.tasktracker.reduce.tasks.maximum</name>
<value>7</value>
```

<final>true</final>
</property>
<property>
<name>mapred.child.java.opts</name>
<value>-Xmx400m</value>
<!-- Not marked as final so jobs can include JVM debugging options -->
</property>
</configuration>

HDFS

To run HDFS, you need to designate one machine as a namenode. In this case, the property fs.default.name is an HDFS filesystem URI, whose host is the namenode's hostname or IP address, and port is the port that the namenode will listen on for RPCs. If no port is specified, the default of 8020 is used. The fs.default.name property also doubles as specifying the default filesystem. The default filesystem is used to resolve relative paths, which are handy to use since they save typing (and avoid hardcoding knowledge of a particular namenode's address). For example, with the default filesystem defined in Example 9-1, the relative URI /a/b is resolved to hdfs://namenode/a/b.

There are a few other configuration properties you should set for HDFS: those that set the storage directories for the namenode and for datanodes. The property dfs.name.dirspecifies a list of directories where the namenode stores persistent filesystem metadata (the edit log and the filesystem image). A copy of each of the metadata files is stored in each directory for redundancy. It's common to configure dfs.name.dirso that the namenode metadata is written to one or two local disks, and a remote disk, such as an NFS-mounted directory. Such a setup guards against failure of a local disk and failure of the entire namenode, since in both cases the files can be recovered and used to start a new namenode. (The secondary namenode takes only periodic checkpoints of the namenode, so it does not provide an up-to-date backup of the namenode.)

You should also set the dfs.data.dirproperty, which specifies a list of directories for a datanode to store its blocks. Unlike the namenode, which uses multiple directories for redundancy, a datanode round-robins writes between its storage directories, so for performance you should specify a storage directory for each local disk. Read performance also benefits from having multiple disks for storage, because blocks will be spread across them, and concurrent reads for distinct blocks will be correspondingly spread across disks. Finally, you should configure where the secondary namenode stores its checkpoints of the filesystem. The fs.checkpoint.dirproperty specifies a list of directories where the checkpoints are kept. Like the storage directories for the namenode, which keep redundant copies of the namenode metadata, the checkpointed filesystem image is stored in each checkpoint directory for redundancy.
Table 4.3 summarizes the important configuration properties for HDFS.

Table 4.3. Important HDFS daemon properties

Property name	Type	Default value	Description
fs.default.name	URI	file:///	The default filesystem. The URI defines the hostname and port that the namenode's RPC server runs on. The default port is 8020. This property should be set in *core-site.xml*.
dfs.name.dir	comma-separated directory names	${hadoop.tmp.dir}/dfs/name	The list of directories where the namenode stores its persistent metadata. The namenode stores a copy of the metadata in each directory in the list.
dfs.data.dir	comma-separated directory names	${hadoop.tmp.dir}/dfs/data	A list of directories where the datanode stores blocks. Each block is stored in only one of these directories.
fs.checkpoint.dir	comma-separated directory names	${hadoop.tmp.dir}/dfs/namesecondary	A list of directories where the secondary namenode stores checkpoints. It stores a copy of the checkpoint in each directory in the list.

A list of directories where the secondary namenode stores checkpoints. It stores a copy of the checkpoint in each directory in the list.

MapReduce

To run MapReduce, you need to designate one machine as a jobtracker, which on small clusters may be the same machine as the namenode. To do this, set the mapred.job.trackerproperty to the hostname or IP address and port that the jobtracker will listen on. Note that this property is not a URI, but a host-port pair, separated by a colon. The port number 8021 is a common choice. During a MapReduce job, intermediate data and working files are written to temporary local files. Since this data includes the potentially very large output of map tasks, you need to ensure that the mapred.local.dirproperty, which controls the location of local temporary storage, is configured to use disk partitions that are large enough. The mapred.local.dirproperty takes a comma-separated list of directory names, and you should use all available local disks to spread disk I/O. Typically, you will use the same disks and partitions (but different directories) for MapReduce temporary data as you use for datanode block storage, as governed by the dfs.data.dirproperty.

MapReduce uses a distributed filesystem to share files (such as the job JAR file) with the tasktrackers that run the MapReduce tasks. The mapred.system.dirproperty is used to specify a directory where these files can be stored. This directory is resolved relative to the default filesystem (configured in fs.default.name), which is usually HDFS.

Finally, you should set the mapred.tasktracker.map.tasks.maximum and mapred.tasktracker.reduce.tasks.maximum properties to reflect the number of available cores on the tasktracker machines and mapred.child.java.optsto reflect the amount of memory available for the tasktracker child JVMs. Table 4.4 summarizes the important configuration properties for HDFS.

Table 4.4. Important MapReduce daemon properties

Property name	Type	Default value	Description
mapred.job.tracker	hostname and port	local	The hostname and port that the jobtracker's RPC server runs on. If set to the default value of local, then the jobtracker is run in-process on demand when you run a MapReduce job (you don't need to start the jobtracker in this case, and in fact you will get an error if you try to start it in this mode).
mapred.local.dir	comma-separated directory names	${hadoop.tmp.dir}/mapred/local	A list of directories where the MapReduce stores intermediate data for jobs. The data is cleared out when the job ends.
mapred.system.dir	URI	${hadoop.tmp.dir}/mapred/system	The directory relative to fs.default.name where shared files are stored, during a job run.
mapred.task tracker.map.tasks. maximum	int	2	The number of map tasks that may be run on a tasktracker at any one time.
mapred.task tracker.reduce.tasks. maximum	int	2	The number of reduce tasks that may be run on a tasktracker at any one time.
mapred.child.java.opts	String	-Xmx200m	The JVM options used to launch the tasktracker child process that runs map and reduce tasks. This property can be set on a per-job basis, which can be useful for setting JVM properties for debugging, for example.

Hadoop Daemon Addresses and Ports

Hadoop daemons generally run both an RPC server (Table 4.5) for communication between daemons and an HTTP server to provide web pages for human consumption (Table 4. 6). Each server is configured by setting the network address and port number to listen on. By specifying the network address as 0.0.0.0, Hadoop will bind to all addresses on the machine. Alternatively, you can specify a single address to bind to. A port number of 0 instructs the server to start on a free port: this is generally discouraged, since it is incompatible with setting cluster-wide firewall policies.

Table 4.5 RPC server properties

Property name	Default value	Description
fs.default.name	file:///	When set to an HDFS URI, this property determines the namenode's RPC server address and port. The default port is 8020 if not specified.
dfs.datanode.ipc.address	0.0.0.0:50020	The datanode's RPC server address and port.
mapred.job.tracker	local	When set to a hostname and port, this property specifies the jobtracker's RPC server address and port. A commonly used port is 8021.
mapred.task.tracker.report.address	127.0.0.1:0	The tasktracker's RPC server address and port. This is used by the tasktracker's child JVM to communicate with the tasktracker. Using any free port is acceptable in this case, as the server only binds to the loopback address. You should change this setting only if the machine has no loopback address.

In addition to an RPC server, datanodes run a TCP/IP server for block transfers. The server address and port is set by the dfs.datanode.addressproperty, and has a default value of 0.0.0.0:50010.

Table 4.6 HTTP server properties

Property name	Default value	Description
mapred.job.tracker.http.address	0.0.0.0:50030	The jobtracker's HTTP server address and port.
mapred.task.tracker.http.address	0.0.0.0:50060	The tasktracker's HTTP server address and port.
dfs.http.address	0.0.0.0:50070	The namenode's HTTP server address and port.
dfs.datanode.http.address	0.0.0.0:50075	The datanode's HTTP server address and port.
dfs.secondary.http.address	0.0.0.0:50090	The secondary namenode's HTTP server address and port.

There are also settings for controlling which network interfaces the datanodes and tasktrackers report as their IP addresses (for HTTP and RPC servers). The relevant properties are dfs.datanode.dns.interface and mapred.tasktracker.dns.interface, both of which are set to default, which will use the default network interface. You can set this explicitly to report the address of a particular interface.

User Account Creation

Once you have a Hadoop cluster up and running, you need to give users access to it. This involves creating a home directory for each user and setting ownership permissions on it:

% hadoop fs -mkdir /user/username

% hadoop fs -chownusername:username/user/username

This is a good time to set space limits on the directory. The following sets a 1 TB limit on the given user directory:

% hadoopdfsadmin -setSpaceQuota 1t /user/username

5. Security

Early versions of Hadoop assumed that HDFS and MapReduce clusters would be used by a group of cooperating users within a secure environment. The measures for restricting access were designed to prevent accidental data loss, rather than to prevent unauthorized access to data. For example, the file permissions system in HDFS prevents one user from accidentally wiping out the whole filesystem from a bug in a program, or by mistakenly typing hadoop fs -rmr /, but it doesn't prevent a malicious user from assuming root's identity to access or delete any data in the cluster.

In security parlance, what was missing was a secure authentication mechanism to assure Hadoop that the user seeking to perform an operation on the cluster is who they claim to be and therefore trusted. HDFS file permissions provide only a mechanism for authorization, which controls what a particular user can do to a particular file. For example, a file may only be readable by a group of users, so anyone not in that group is not authorized to read it. However, authorization is not enough by itself, since the system is still open to abuse via spoofing by a malicious user who can gain network access to the cluster.

It's common to restrict access to data that contains personally identifiable information (such as an end user's full name or IP address) to a small set of users (of the cluster) within the organization, who are authorized to access such information. Less sensitive (or anonymized) data may be made available to a larger set of users. It is convenient to host a mix of datasets with different security levels on the same cluster (not least because it means the datasets with

lower security levels can be shared). However, to meet regulatory requirements for data protection, secure authentication must be in place for shared clusters. This is the situation that Yahoo! faced in 2009, which led a team of engineers there to implement secure authentication for Hadoop. In their design, Hadoop itself does not manage user credentials, since it relies on Kerberos, a mature open-source network authentication protocol, to authenticate the user. In turn, Kerberos doesn't manage permissions. Kerberos says that a user is who they say they are; it's Hadoop's job to determine whether that user has permission to perform a given action.
Kerberos and Hadoop

At a high level, there are three steps that a client must take to access a service when using Kerberos, each of which involves a message exchange with a server:
1. Authentication. The client authenticates itself to the Authentication Server and receives a timestamped Ticket-Granting Ticket (TGT).
2. Authorization. The client uses the TGT to request a service ticket from the Ticket Granting Server.
3. Service Request. The client uses the service ticket to authenticate itself to the server that is providing the service the client is using. In the case of Hadoop, this might be the namenode or the jobtracker.
Together, the Authentication Server and the Ticket Granting Server form the Key Distribution Center(KDC). The process is shown graphically in Figure 4.2.

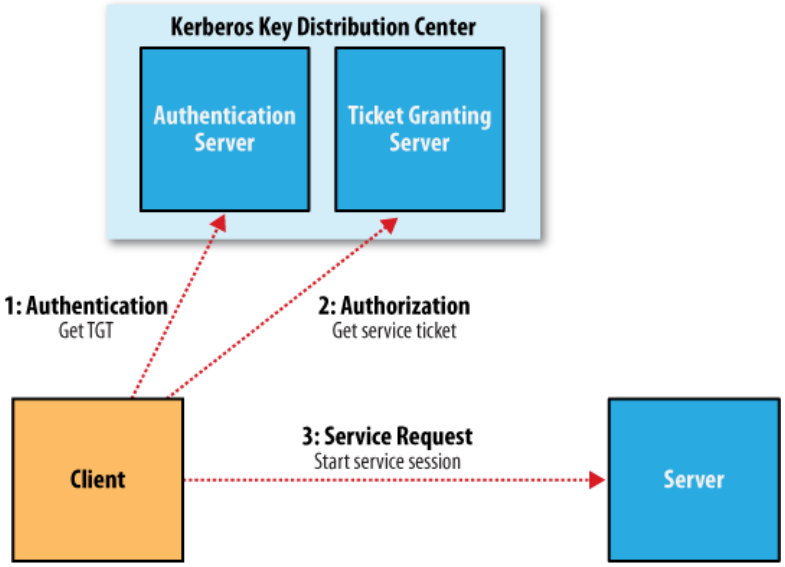


Fig. 4.2 The three-step Kerberos ticket exchange protocol

The authorization and service request steps are not user-level actions: the client performs these steps on the user's behalf. The authentication step, however, is normally carried out explicitly by the user using the kinit command, which will prompt for a password. However, this doesn't mean you need to enter your password every time you run a job or access HDFS, since TGTs last for 10 hours by default (and can be renewed for up to a week). It's common to automate authentication at operating system login time, thereby providing single sign-on to Hadoop.

In cases where you don't want to be prompted for a password (for running an unattended MapReduce job, for example), you can create a Kerberos keytabfile using the ktutilcommand. A keytab is a file that stores passwords and may be supplied to kinit with the -t option.

6. Administering Hadoop

In this, the procedures are discussed to keep a cluster running smoothly.

HDFS

Persistent Data Structures

As an administrator, it is invaluable to have a basic understanding of how the components of HDFS—the namenode, the secondary namenode, and the datanodes— organize their persistent data on disk. Knowing which files are which can help you diagnose problems or spot that something is awry.

Namenode directory structure

A newly formatted namenode creates the following directory structure:

${dfs.name.dir}/current/VERSION
/edits
/fsimage
/fstime

The dfs.name.dirproperty is a list of directories, with the same contents mirrored in each directory. This mechanism provides resilience, particularly if one of the directories is an NFS mount, as is recommended. The VERSION file is a Java properties file that contains information about the version of HDFS that is running. Here are the contents of a typical file:

#Tue Mar 10 19:21:36 GMT 2009
namespaceID=134368441
cTime=0
storageType=NAME_NODE
layoutVersion=-18

The layoutVersionis a negative integer that defines the version of HDFS's persistent data structures. This version number has no relation to the release number of the Hadoop distribution. Whenever the layout changes, the version number is decremented (for example, the version after −18 is −19). When this happens, HDFS needs to be upgraded, since a newer namenode (or datanode) will not operate if its storage layout is an older version.

The namespaceID is a unique identifier for the filesystem, which is created when the filesystem is first formatted. The namenode uses it to identify new datanodes, since they will not know the namespaceIDuntil they have registered with the namenode. The Timeproperty marks the creation time of the namenode's storage. For newly formatted storage, the value is always zero, but it is updated to a timestamp whenever the filesystem is upgraded. The storageType indicates that this storage directory contains data structures for a namenode. The other files in the namenode's storage directory are edits, fsimage, and fstime. These are all binary files, which use Hadoop Writable objects as their serialization format. To understand what these files are for, we need to dig into the workings of the namenode a little more.

The filesystem image and edit log

When a filesystem client performs a write operation (such as creating or moving a file), it is first recorded in the edit log. The namenode also has an in-memory representation of the filesystem metadata, which it updates after the edit log has been modified. The in-memory metadata is used to serve read requests. The edit log is flushed and synced after every write before a success code is returned to the client. For namenodes that write to multiple directories, the write must be flushed and synced to every copy before returning successfully. This ensures that no operation is lost due to machine failure. The fsimagefile is a persistent checkpoint of the filesystem metadata. However, it is not updated for every filesystem write operation, since

writing out the fsimagefile, which can grow to be gigabytes in size, would be very slow. This does not compromise resilience, however, because if the namenode fails, then the latest state of its metadata can be reconstructed by loading the fsimagefrom disk into memory, then applying each of the operations in the edit log. In fact, this is precisely what the namenode does when it starts up.

The edits file would grow without bound. Though this state of affairs would have no impact on the system while the namenode is running, if the namenode were restarted, it would take a long time to apply each of the operations in its (very long) edit log. During this time, the filesystem would be offline, which is generally undesirable.

The solution is to run the secondary namenode, whose purpose is to produce checkpoints of the primary's in-memory filesystem metadata. The checkpointing process proceeds as follows (and is shown schematically in Figure 1.1):

1. The secondary asks the primary to roll its edits file, so new edits go to a new file.
2. The secondary retrieves fsimageand edits from the primary (using HTTP GET).
3. The secondary loads fsimageinto memory, applies each operation from edits, then creates a new consolidated fsimagefile.
4. The secondary sends the new fsimageback to the primary (using HTTP POST).
5. The primary replaces the old fsimagewith the new one from the secondary, and the old edits file with the new one it started in step 1. It also updates the fstimefile to record the time that the checkpoint was taken.

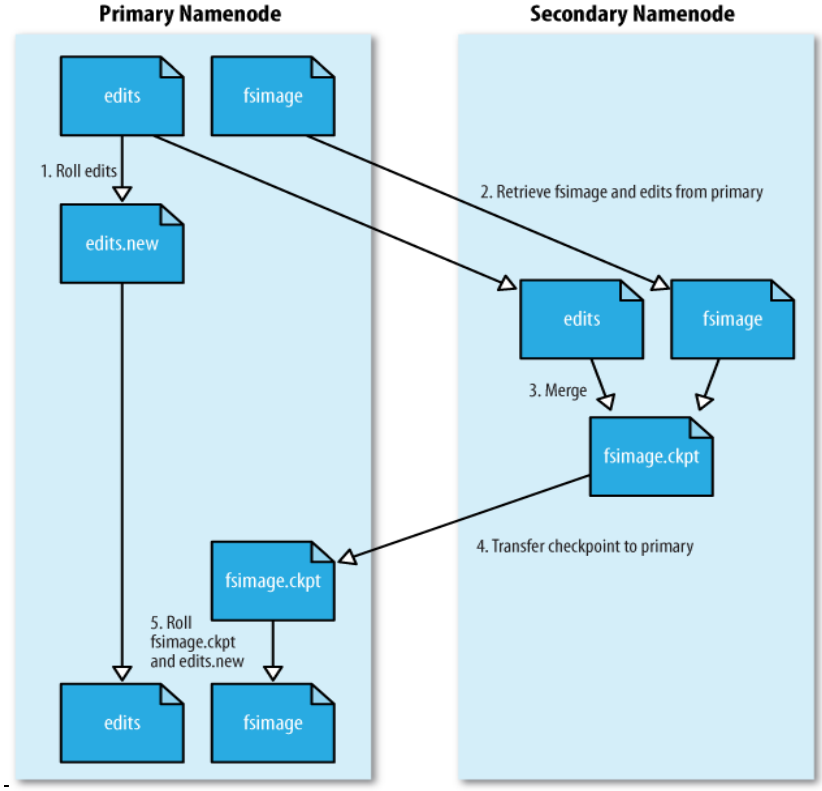


Fig. 4.3 The checkpointing process

At the end of the process, the primary has an up-to-date fsimagefile and a shorter edits file (it is not necessarily empty, as it may have received some edits while the checkpoint was being taken). It is possible for an administrator to run this process manually while the namenode is in safe mode, using the hadoopdfsadmin -saveNamespace command. This procedure makes it clear why the secondary has similar memory requirements to the primary

(since it loads the fsimage into memory), which is the reason that the secondary needs a dedicated machine on large clusters.

The schedule for checkpointing is controlled by two configuration parameters. The secondary namenode checkpoints every hour (fs.checkpoint.periodin seconds) or sooner if the edit log has reached 64 MB (fs.checkpoint.sizein bytes), which it checks every five minutes.

Secondary namenode directory structure

A useful side effect of the checkpointing process is that the secondary has a checkpoint at the end of the process, which can be found in a subdirectory called previous.checkpoint. This can be used as a source for making (stale) backups of the namenode's metadata:

${fs.checkpoint.dir}/current/VERSION
/edits
/fsimage
/fstime
/previous.checkpoint/VERSION
/edits
/fsimage
/fstime

The layout of this directory and of the secondary's current directory is identical to the namenode's. This is by design, since in the event of total namenode failure (when there are no recoverable backups, even from NFS), it allows recovery from a secondary namenode. This can be achieved either by copying the relevant storage directory to a new namenode, or, if the secondary is taking over as the new primary namenode, by using the -importCheckpoint option when starting the namenode daemon. The –importCheckpoint option will load the namenode metadata from the latest checkpoint in the directory defined by the fs.checkpoint.dir property, but only if there is no metadata in the dfs.name.dir directory, so there is no risk of overwriting precious metadata.

Datanode directory structure

Unlike namenodes, datanodes do not need to be explicitly formatted, since they create their storage directories automatically on startup. Here are the key files and directories:

${dfs.data.dir}/current/VERSION
/blk_<id_1>
/blk_<id_1>.meta
/blk_<id_2>
/blk_<id_2>.meta
/...
/blk_<id_64>
/blk_<id_64>.meta
/subdir0/
/subdir1/
/...
/subdir63/

A datanode'sVERSION file is very similar to the namenode's:

#Tue Mar 10 21:32:31 GMT 2009
namespaceID=134368441
storageID=DS-547717739-172.16.85.1-50010-1236720751627
cTime=0

storageType=DATA_NODE
layoutVersion=-18

The namespaceID, cTime, and layoutVersionare all the same as the values in the namenode (in fact, the namespaceIDis retrieved from the namenode when the datanode first connects). The storageIDis unique to the datanode (it is the same across all storage directories) and is used by the namenode to uniquely identify the datanode. The storageTypeidentifies this directory as a datanode storage directory.

The other files in the datanode'scurrent storage directory are the files with the blk_prefix. There are two types: the HDFS blocks themselves (which just consist of the file's raw bytes) and the metadata for a block (with a .meta suffix). A block file just consists of the raw bytes of a portion of the file being stored; the metadata file is made up of a header with version and type information, followed by a series of checksums for sections of the block.

When the number of blocks in a directory grows to a certain size, the datanode creates a new subdirectory in which to place new blocks and their accompanying metadata. It creates a new subdirectory every time the number of blocks in a directory reaches 64 (set by the dfs.datanode.numblocksconfiguration property). The effect is to have a tree with high fan-out, so even for systems with a very large number of blocks, the directories will only be a few levels deep. By taking this measure, the datanode ensures that there is a manageable number of files per directory, which avoids the problems that most operating systems encounter when there are a large number of files (tens or hundreds of thousands) in a single directory. If the configuration property dfs.data.dirspecifies multiple directories (on different drives), blocks are written to each in a round-robin fashion. Note that blocks are not replicated on each drive on a single datanode: block replication is across distinct datanodes.

Safe Mode

When the namenode starts, the first thing it does is load its image file (fsimage) into memory and apply the edits from the edit log (edits). Once it has reconstructed a consistent in-memory image of the filesystem metadata, it creates a new fsimagefile (effectively doing the checkpoint itself, without recourse to the secondary namenode) and an empty edit log. Only at this point does the namenode start listening for RPC and HTTP requests. However, the namenode is running in safe mode, which means that it offers only a read-only view of the filesystem to clients.

The locations of blocks in the system are not persisted by the namenode— this information resides with the datanodes, in the form of a list of the blocks it is storing. During normal operation of the system, the namenode has a map of block locations stored in memory. Safe mode is needed to give the datanodes time to check in to the namenode with their block lists, so the namenode can be informed of enough block locations to run the filesystem effectively. If the namenode didn't wait for enough datanodes to check in, then it would start the process of replicating blocks to new datanodes, which would be unnecessary in most cases (since it only needed to wait for the extra datanodes to check in), and would put a great strain on the cluster's resources. Indeed, while in safe mode, the namenode does not issue any block replication or deletion instructions to datanodes.

Safe mode is exited when the minimal replication condition is reached, plus an extension time of 30 seconds. The minimal replication condition is when 99.9% of the blocks in the whole filesystem meet their minimum replication level (which defaults to one, and is set by dfs.replication.min, see Table 4.7).

When you are starting a newly formatted HDFS cluster, the namenode does not go into safe mode since there are no blocks in the system.

Table 4.7 Safe mode properties

Property name	Type	Default value	Description
dfs.replication.min	int	1	The minimum number of replicas that have to be written for a write to be successful.
dfs.safemode.threshold.pct	float	0.999	The proportion of blocks in the system that must meet the minimum replication level defined by dfs.replication.min before the namenode will exit safe mode. Setting this value to 0 or less forces the namenode not to start in safe mode. Setting this value to more than 1 means the namenode never exits safe mode.
dfs.safemode.extension	int	30,000	The time, in milliseconds, to extend safe mode by after the minimum replication condition defined by dfs.safemode.threshold.pct has been satisfied. For small clusters (tens of nodes), it can be set to 0.

Entering and leaving safe mode

To see whether the namenode is in safe mode, you can use the dfsadmincommand:

```
% hadoopdfsadmin -safemode get
Safe mode is ON
```

The front page of the HDFS web UI provides another indication of whether the namenode is in safe mode.

Sometimes you want to wait for the namenode to exit safe mode before carrying out a command, particularly in scripts. The wait option achieves this:

```
hadoopdfsadmin -safemode wait
# command to read or write a file
```

An administrator has the ability to make the namenode enter or leave safe mode at any time. It is sometimes necessary to do this when carrying out maintenance on the cluster or after upgrading a cluster to confirm that data is still readable. To enter safe mode, use the following command:

```
% hadoopdfsadmin -safemode enter
Safe mode is ON
```

You can use this command when the namenode is still in safe mode while starting up to ensure that it never leaves safe mode. Another way of making sure that the namenode stays in safe mode indefinitely is to set the property dfs.safemode.threshold.pct to a value over one.

You can make the namenode leave safe mode by using:

```
% hadoopdfsadmin -safemode leave
Safe mode is OFF
```

Audit Logging

HDFS has the ability to log all filesystem access requests, a feature that some organizations require for auditing purposes. Audit logging is implemented using log4j logging at the INFO level, and in the default configuration it is disabled, as the log threshold is set to WARN in log4j.properties:

```
log4j.logger.org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit=WARN
```

You can enable audit logging by replacing WARN with INFO, and the result will be a log line written to the namenode's log for every HDFS event. Here's an example for a list status request on /user/tom:

```
2009-03-13 07:11:22,982 INFO org.apache.hadoop.hdfs.server.namenode.FSNamesystem.
audit: ugi=tom,staff,adminip=/127.0.0.1 cmd=listStatussrc=/user/tom dst=null
```

perm=null

It is a good idea to configure log4j so that the audit log is written to a separate file and isn't mixed up with the namenode's other log entries. An example of how to do this can be found on the Hadoop wiki at http://wiki.apache.org/hadoop/HowToConfigure.

Tools

dfsadmin

The dfsadmintool is a multipurpose tool for finding information about the state of HDFS, as well as performing administration operations on HDFS. It is invoked as hadoopdfsadmin. Commands that alter HDFS state typically require superuser privileges. The available commands to dfsadmin are described in Table 4.8

Table 4.8 dfsadmin commands

Command	Description
`-help`	Shows help for a given command, or all commands if no command is specified.
`-report`	Shows filesystem statistics (similar to those shown in the web UI) and information on connected datanodes.
`-metasave`	Dumps information to a file in Hadoop's log directory about blocks that are being replicated or deleted, and a list of connected datanodes.
`-safemode`	Changes or query the state of safe mode.
`-saveNamespace`	Saves the current in-memory filesystem image to a new *fsimage* file and resets the *edits* file. This operation may be performed only in safe mode.
`-refreshNodes`	Updates the set of datanodes that are permitted to connect to the namenode.
`-upgradeProgress`	Gets information on the progress of an HDFS upgrade or forces an upgrade to proceed.
`-finalizeUpgrade`	Removes the previous version of the datanodes' and namenode's storage directories. Used after an upgrade has been applied and the cluster is running successfully on the new version.
`-setQuota`	Sets directory quotas. Directory quotas set a limit on the number of names (files or directories) in the directory tree. Directory quotas are useful for preventing users from creating large numbers of small files, a measure that helps preserve the namenode's memory (recall that accounting information for every file, directory, and block in the filesystem is stored in memory).
`-clrQuota`	Clears specified directory quotas.
`-setSpaceQuota`	Sets space quotas on directories. Space quotas set a limit on the size of files that may be stored in a directory tree. They are useful for giving users a limited amount of storage.
`-clrSpaceQuota`	Clears specified space quotas.
`-refreshServiceAcl`	Refreshes the namenode's service-level authorization policy file.

Filesystem check (fsck)

Hadoop provides an fsckutility for checking the health of files in HDFS. The tool looks for blocks that are missing from all datanodes, as well as under- or over-replicated blocks. Here is an example of checking the whole filesystem for a small cluster:

```
% hadoopfsck /
.....Status: HEALTHY
Total size: 511799225 B
Total dirs: 10
Total files: 22
Total blocks (validated): 22 (avg. block size 23263601 B)
Minimally replicated blocks: 22 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 3
Average block replication: 3.0
Corrupt blocks: 0
```

Missing replicas: 0 (0.0 %)
Number of data-nodes: 4
Number of racks: 1

The filesystem under path '/' is HEALTHY fsckrecursively walks the filesystem namespace, starting at the given path (here the filesystem root), and checks the files it finds. It prints a dot for every file it checks. To check a file, fsckretrieves the metadata for the file's blocks and looks for problems or inconsistencies. Note that fsckretrieves all of its information from the namenode; it does not communicate with any datanodes to actually retrieve any block data.Most of the output from fsckis self-explanatory, but here are some of the conditions it looks for:

Over-replicated blocks

These are blocks that exceed their target replication for the file they belong to. Over-replication is not normally a problem, and HDFS will automatically delete excess replicas.

Under-replicated blocks

These are blocks that do not meet their target replication for the file they belong to. HDFS will automatically create new replicas of under-replicated blocks until they meet the target replication. You can get information about the blocks being replicated (or waiting to be replicated) using hadoopdfsadmin -metasave.

Misreplicated blocks

These are blocks that do not satisfy the block replica placement policy. For example, for a replication level of three in a multirack cluster, if all three replicas of a block are on the same rack, then the block is misreplicated since the replicas should be spread across at least two racks for resilience.

A misreplicated block is not fixed automatically by HDFS (at the time of this writing). As a workaround, you can fix the problem manually by increasing the replication of the file the block belongs to (using hadoop fs -setrep), waiting until the block gets replicated, then decreasing the replication of the file back to its original value.

Corrupt blocks

These are blocks whose replicas are all corrupt. Blocks with at least one noncorrupt replica are not reported as corrupt; the namenode will replicate the noncorrupt replica until the target replication is met.

Missing replicas

These are blocks with no replicas anywhere in the cluster.

Corrupt or missing blocks are the biggest cause for concern, as it means data has been lost. By default, fsckleaves files with corrupt or missing blocks, but you can tell it to perform one of the following actions on them:

• Move the affected files to the /lost+founddirectory in HDFS, using the -move option. Files are broken into chains of contiguous blocks to aid any salvaging efforts you may attempt.

• Delete the affected files, using the -delete option. Files cannot be recovered after being deleted.

The fscktool provides an easy way to find out which blocks are in any particular file. For example:

% hadoopfsck /user/tom/part-00007 -files -blocks -racks
/user/tom/part-00007 25582428 bytes, 1 block(s): OK
0. blk_-3724870485760122836_1035 len=25582428 repl=3 [/default-rack/10.251.43.2:50010, /default-rack/10.251.27.178:50010, /default-rack/10.251.123.163:50010]

This says that the file /user/tom/part-00007 is made up of one block and shows the datanodes where the blocks are located. The fsckoptions used are as follows:

• The -files option shows the line with the filename, size, number of blocks, and its health (whether there are any missing blocks).

• The -blocks option shows information about each block in the file, one line per block.

• The -racks option displays the rack location and the datanode addresses for each block.

Running hadoopfsckwithout any arguments displays full usage instructions.

7. Monitoring

Monitoring is an important part of system administration. In this section, we look at the monitoring facilities in Hadoop and how they can hook into external monitoring systems. The purpose of monitoring is to detect when the cluster is not providing the expected level of service. The master daemons are the most important to monitor: the namenodes (primary and secondary) and the jobtracker. Failure of datanodes and tasktrackers is to be expected, particularly on larger clusters, so you should provide extra capacity so that the cluster can tolerate having a small percentage of dead nodes at any time. In addition to the facilities described next, some administrators run test jobs on a periodic basis as a test of the cluster's health. There is lot of work going on to add more monitoring capabilities to Hadoop. For example, Chukwa is a data collection and monitoring system built on HDFS and MapReduce, and excels at mining log data for finding large-scale trends.

Logging

All Hadoop daemons produce logfiles that can be very useful for finding out what is happening in the system.

Setting log levels

When debugging a problem, it is very convenient to be able to change the log level temporarily for a particular component in the system. Hadoop daemons have a web page for changing the log level for any log4j log name, which can be found at /logLevelin the daemon's web UI. By convention, log names in Hadoop correspond to the classname doing the logging, although there are exceptions to this rule, so you should consult the source code to find log names. For example, to enable debug logging for the JobTrackerclass, we would visit the jobtracker's web UI at http://jobtracker-host:50030/logLevel and set the log name org.apache.hadoop.mapred.JobTrackerto level DEBUG.

The same thing can be achieved from the command line as follows:

```
% hadoopdaemonlog -setleveljobtracker-host:50030 \
org.apache.hadoop.mapred.JobTracker DEBUG
```

Log levels changed in this way are reset when the daemon restarts, which is usually what you want. However, to make a persistent change to a log level, simply change the log4j.properties file in the configuration directory. In this case, the line to add is:

```
log4j.logger.org.apache.hadoop.mapred.JobTracker=DEBUG
```

Getting stack traces

Hadoop daemons expose a web page (/stacks in the web UI) that produces a thread dump for all running threads in the daemon's JVM. For example, you can get a thread dump for a jobtracker from http://jobtracker-host:50030/stacks.

Metrics

The HDFS and MapReduce daemons collect information about events and measurements that are collectively known as metrics. For example, datanodes collect the

following metrics (and many more): the number of bytes written, the number of blocks replicated, and the number of read requests from clients (both local and remote).

Metrics belong to a context, and Hadoop currently uses “dfs”, “mapred”, “rpc”, and “jvm” contexts. Hadoop daemons usually collect metrics under several contexts. For example, datanodes collect metrics for the “dfs”, “rpc”, and “jvm” contexts.

A context defines the unit of publication; you can choose to publish the “dfs” context, but not the “jvm” context, for instance. Metrics are configured in the conf/hadoopmetrics. properties file, and, by default, all contexts are configured so they do not publish their metrics. This is the contents of the default configuration file (minus thecomments):

dfs.class=org.apache.hadoop.metrics.spi.NullContext
mapred.class=org.apache.hadoop.metrics.spi.NullContext
jvm.class=org.apache.hadoop.metrics.spi.NullContext
rpc.class=org.apache.hadoop.metrics.spi.NullContext

Each line in this file configures a different context and specifies the class that handles the metrics for that context. The class must be an implementation of the MetricsCon text interface; and, as the name suggests, the NullContextclass neither publishes nor updates metrics.

The other implementations of MetricsContext are covered in the following sections.

FileContext

FileContextwrites metrics to a local file. It exposes two configuration properties: fileName, which specifies the absolute name of the file to write to, and period, for the time interval (in seconds) between file updates. Both properties are optional; if not set, the metrics will be written to standard output every five seconds. Configuration properties apply to a context name and are specified by appending the property name to the context name (separated by a dot). For example, to dump the “jvm” context to a file, we alter its configuration to be the following:

jvm.class=org.apache.hadoop.metrics.file.FileContext
jvm.fileName=/tmp/jvm_metrics.log

In the first line, we have changed the “jvm” context to use a FileContext, and in the second, we have set the “jvm” context’s fileNameproperty to be a temporary file. Here are two lines of output from the logfile, split over several lines to fit the page:

```
jvm.metrics: hostName=ip-10-250-59-159, processName=NameNode, sessionId=, ↵
gcCount=46, gcTimeMillis=394, logError=0, logFatal=0, logInfo=59, logWarn=1, ↵
memHeapCommittedM=4.9375, memHeapUsedM=2.5322647, memNonHeapCommittedM=18.25, ↵
memNonHeapUsedM=11.330269, threadsBlocked=0, threadsNew=0, threadsRunnable=6, ↵
threadsTerminated=0, threadsTimedWaiting=8, threadsWaiting=13
jvm.metrics: hostName=ip-10-250-59-159, processName=SecondaryNameNode, sessionId=, ↵
gcCount=36, gcTimeMillis=261, logError=0, logFatal=0, logInfo=18, logWarn=4, ↵
memHeapCommittedM=5.4414062, memHeapUsedM=4.46756, memNonHeapCommittedM=18.25, ↵
memNonHeapUsedM=10.624519, threadsBlocked=0, threadsNew=0, threadsRunnable=5, ↵
threadsTerminated=0, threadsTimedWaiting=4, threadsWaiting=2
```

FileContext can be useful on a local system for debugging purposes, but is unsuitable on a larger cluster since the output files are spread across the cluster, which makes analyzing them difficult.

8. Maintenance

Routine Administration Procedures

Metadata backups

If the namenode's persistent metadata is lost or damaged, the entire filesystem is rendered unusable, so it is critical that backups are made of these files. You should keep multiple copies of different ages (one hour, one day, one week, and one month, say) to protect against corruption, either in the copies themselves or in the live files running on the namenode. A straightforward way to make backups is to write a script to periodically archive the secondary namenode'sprevious.checkpointsubdirectory (under the directory defined by the fs.checkpoint.dirproperty) to an offsite location. The script should additionally test the integrity of the copy. This can be done by starting a local namenode daemon and verifying that it has successfully read the fsimageand edits files into memory (by scanning the namenode log for the appropriate success message, for example).

Data backups

Although HDFS is designed to store data reliably, data loss can occur, just like in any storage system, and thus a backup strategy is essential. With the large data volumes that Hadoop can store, deciding what data to back up and where to store it is a challenge. The key here is to prioritize your data. The highest priority is the data that cannot be regenerated and that is critical to the business; however, data that is straightforward to regenerate, or essentially disposable because it is of limited business value, is the lowest priority, and you may choose not to make backups of this category of data. It's common to have a policy for user directories in HDFS. For example, they may have space quotas and be backed up nightly. Whatever the policy, make sure your users know what it is, so they know what to expect. The distcp tool is ideal for making backups to other HDFS clusters (preferably running on a different version of the software, to guard against loss due to bugs in HDFS) or other Hadoop filesystems (such as S3 or KFS), since it can copy files in parallel. Alternatively, you can employ an entirely different storage system for backups, using one of the ways to export data from HDFS.

Filesystem check (fsck)

It is advisable to run HDFS's fsck tool regularly (for example, daily) on the whole filesystem to proactively look for missing or corrupt blocks.

Filesystem balancer

Run the balancer tool regularly to keep the filesystem datanodes evenly balanced.

Commissioning and Decommissioning Nodes

As an administrator of a Hadoop cluster, you will need to add or remove nodes from time to time. For example, to grow the storage available to a cluster, you commission new nodes. Conversely, sometimes you may wish to shrink a cluster, and to do so, you decommission nodes. It can sometimes be necessary to decommission a node if it is misbehaving, perhaps because it is failing more often than it should or its performance is noticeably slow. Nodes normally run both a datanode and a tasktracker, and both are typically commissioned or decommissioned in tandem.

Commissioning new nodes

Although commissioning a new node can be as simple as configuring the hdfs site.xml file to point to the namenode and the mapred-site.xml file to point to the jobtracker, and starting the datanode and jobtracker daemons, it is generally best to have a list of authorized nodes. It is a potential security risk to allow any machine to connect to the namenode and act as a datanode, since the machine may gain access to data that it is not authorized to see.

Furthermore, since such a machine is not a real datanode, it is not under your control, and may stop at any time, causing potential data loss. (Imagine what would happen if a number of such nodes were connected, and a block of data was present only on the "alien" nodes?) This scenario is a risk even inside a firewall, through misconfiguration, so datanodes (and tasktrackers) should be explicitly managed on all production clusters.

Datanodes that are permitted to connect to the namenode are specified in a file whose name is specified by the dfs.hostsproperty. The file resides on the namenode's local filesystem, and it contains a line for each datanode, specified by network address (as reported by the datanode—you can see what this is by looking at the namenode's web UI). If you need to specify multiple network addresses for a datanode, put them on one line, separated by whitespace. Similarly, tasktrackers that may connect to the jobtracker are specified in a file whose name is specified by the mapred.hostsproperty. In most cases, there is one shared file, referred to as the include file, that both dfs.hostsand mapred.hosts refer to, since nodes in the cluster run both datanode and tasktracker daemons.

To add new nodes to the cluster:

1. Add the network addresses of the new nodes to the include file.
2. Update the namenode with the new set of permitted datanodes using this command:

% hadoopdfsadmin -refreshNodes

3. Update the slaves file with the new nodes, so that they are included in future operations performed by the Hadoop control scripts.
4. Start the new datanodes.
5. Restart the MapReduce cluster.
6. Check that the new datanodes and tasktrackers appear in the web UI.

HDFS will not move blocks from old datanodes to new datanodes to balance the cluster.

Decommissioning old nodes

Although HDFS is designed to tolerate datanode failures, this does not mean you can just terminate datanodesen masse with no ill effect. With a replication level of three, for example, the chances are very high that you will lose data by simultaneously shutting down three datanodes if they are on different racks. The way to decommission datanodes is to inform the namenode of the nodes that you wish to take out of circulation, so that it can replicate the blocks to other datanodes before the datanodes are shut down. With tasktrackers, Hadoop is more forgiving. If you shut down a tasktracker that is running tasks, the jobtracker will notice the failure and reschedule the tasks on other tasktrackers.

The decommissioning process is controlled by an exclude file, which for HDFS is set by the dfs.hosts.excludeproperty and for MapReduce by the mapred.hosts.excludeproperty. It is often the case that these properties refer to the same file. The exclude file lists the nodes that are not permitted to connect to the cluster. The rules for whether a tasktracker may connect to the jobtracker are simple: a tasktracker may connect only if it appears in the include file and does not appear in the exclude file. An unspecified or empty include file is taken to mean that all nodes are in the include file. For HDFS, the rules are slightly different. If a datanode appears in both the include and the exclude file, then it may connect, but only to be decommissioned. Table 4.9 summarizes the different combinations for datanodes. As for tasktrackers, an unspecified or empty include file means all nodes are included.

Table 4.9 HDFS include and exclude file precedence

Node appears in include file	Node appears in exclude file	Interpretation
No	No	Node may not connect.
No	Yes	Node may not connect.
Yes	No	Node may connect.
Yes	Yes	Node may connect and will be decommissioned.

To remove nodes from the cluster:
1. Add the network addresses of the nodes to be decommissioned to the exclude file. Do not update the include file at this point.
2. Restart the MapReduce cluster to stop the tasktrackers on the nodes being decommissioned.
3. Update the namenode with the new set of permitted datanodes, with this command:
% hadoopdfsadmin -refreshNodes
4. Go to the web UI and check whether the admin state has changed to "Decommission In Progress" for the datanodes being decommissioned. They will start copying their blocks to other datanodes in the cluster.
5. When all the datanodes report their state as "Decommissioned," then all the blocks have been replicated. Shut down the decommissioned nodes.
6. Remove the nodes from the include file, and run: % hadoopdfsadmin -refreshNodes
7. Remove the nodes from the slaves file.

Upgrades

Upgrading an HDFS and MapReduce cluster requires careful planning. The most important consideration is the HDFS upgrade. If the layout version of the filesystem has changed, then the upgrade will automatically migrate the filesystem data and metadata to a format that is compatible with the new version. As with any procedure that involves data migration, there is a risk of data loss, so you should be sure that both your data and metadata is backed up.

Part of the planning process should include a trial run on a small test cluster with a copy of data that you can afford to lose. A trial run will allow you to familiarize yourself with the process, customize it to your particular cluster configuration and toolset, and iron out any snags before running the upgrade procedure on a production cluster. A test cluster also has the benefit of being available to test client upgrades on. Upgrading a cluster when the filesystem layout has not changed is fairly straightforward: install the new versions of HDFS and MapReduce on the cluster (and on clients at the same time), shut down the old daemons, update configuration files, then start up the new daemons and switch clients to use the new libraries. This process is reversible, so rolling back an upgrade is also straightforward. After every successful upgrade, you should perform a couple of final cleanup steps:
• Remove the old installation and configuration files from the cluster.
• Fix any deprecation warnings in your code and configuration.
HDFS data and metadata upgrades

Upgrade to a new version of HDFS and it expects a different layout version, then the namenode will refuse to run. A message like the following will appear in its log:
File system image contains an old layout version -16.An upgrade to version -18 is required. Please restart NameNode with -upgrade option.

The most reliable way of finding out whether you need to upgrade the filesystem is by performing a trial on a test cluster. An upgrade of HDFS makes a copy of the previous version's metadata and data. Doing an upgrade does not double the storage requirements of the cluster, as the datanodes use hard links to keep two references (for the current and previous version) to the same block of data. This design makes it straightforward to roll back to the previous version of the filesystem, should you need to. You should understand that any changes made to the data on the upgraded system will be lost after the rollback completes. You can keep only the previous version of the filesystem: you can't roll back several versions. Therefore, to carry out another upgrade to HDFS data and metadata, you will need to delete the previous version, a process called finalizing the upgrade. Once an upgrade is finalized, there is no procedure for rolling back to a previous version. In general, you can skip releases when upgrading (for example, you can upgrade from release 0.18.3 to 0.20.0 without having to upgrade to a 0.19.x release first), but in some cases, you may have to go through intermediate releases. The release notes make it clear when this is required.

You should only attempt to upgrade a healthy filesystem. Before running the upgrade, do a full fsck. As an extra precaution, you can keep a copy of the fsckoutput that lists all the files and blocks in the system, so you can compare it with the output of running fsckafter the upgrade.

It's also worth clearing out temporary files before doing the upgrade, both from the MapReduce system directory on HDFS and local temporary files. With these preliminaries out of the way, here is the high-level procedure for upgrading a cluster when the filesystem layout needs to be migrated:

1. Make sure that any previous upgrade is finalized before proceeding with another upgrade.
2. Shut down MapReduce and kill any orphaned task processes on the tasktrackers.
3. Shut down HDFS and backup the namenode directories.
4. Install new versions of Hadoop HDFS and MapReduce on the cluster and on clients.
5. Start HDFS with the -upgrade option.
6. Wait until the upgrade is complete.
7. Perform some sanity checks on HDFS.
8. Start MapReduce.
9. Roll back or finalize the upgrade (optional).

While running the upgrade procedure, it is a good idea to remove the Hadoop scripts from your PATH environment variable. This forces you to be explicit about which version of the scripts you are running. It can be convenient to define two environment variables for the new installation directories; in the following instructions, we have defined OLD_HADOOP_INSTALL and NEW_HADOOP_INSTALL.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

School of Computing

Department of Computer Science and Engineering

UNIT V - Big Data – SBS1608

SYLLABUS

Applications on Big Data Using Pig and Hive – Data processing operators in Pig – Hive services – HiveQL – Querying Data in Hive - fundamentals of HBase and ZooKeeper.

1. Applications on Big Data Using Pig and Hive

2. Data Processing Operators in Pig

Loading and Storing Data

How to load data from external storage for processing in Pig is discussed here. Storing the results is straightforward, too. Here's an example of using PigStorage to store tuples as plain-text values separated by a colon character:

grunt>STORE A INTO 'out' USING PigStorage(':');

grunt>cat out

Joe:cherry:2

Ali:apple:3

Joe:banana:2

Eve:apple:7

Filtering Data

Once you have some data loaded into a relation, the next step is often to filter it to remove the data that you are not interested in. By filtering early in the processing pipeline, you minimize the amount of data flowing through the system, which can improve efficiency.

FOREACH...GENERATE

We have already seen how to remove rows from a relation using the FILTER operator with simple expressions and a UDF. The FOREACH...GENERATE operator is used to act on every row in a relation. It can be used to remove fields or to generate new ones. In this example, we do both:

grunt>DUMP A;

(Joe,cherry,2)

(Ali,apple,3)

(Joe,banana,2)

(Eve,apple,7)

grunt>B = FOREACH A GENERATE $0, $2+1, 'Constant';

grunt>DUMP B;

(Joe,3,Constant)

(Ali,4,Constant)

(Joe,3,Constant)

(Eve,8,Constant)

Here we have created a new relation B with three fields. Its first field is a projection of the first field ($0) of A. B's second field is the third field of A ($2) with one added to it. B's third field is a constant field (every row in B has the same third field) with the char array value Constant. The FOREACH...GENERATE operator has a nested form to support more complex processing. In the following example, we compute various statistics for the weather dataset:

```
-- year_stats.pig

REGISTER pig-examples.jar;

DEFINE isGoodcom.hadoopbook.pig.IsGoodQuality();

records = LOAD 'input/ncdc/all/19{1,2,3,4,5}0*'

USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,16-19,88-92,93-93')

AS (usaf:chararray, wban:chararray, year:int, temperature:int, quality:int);

grouped_records = GROUP records BY year PARALLEL 30;

year_stats = FOREACH grouped_records {

uniq_stations = DISTINCT records.usaf;

good_records = FILTER records BY isGood(quality);

GENERATE FLATTEN(group), COUNT(uniq_stations) AS station_count,

COUNT(good_records) AS good_record_count, COUNT(records) AS record_count;

} DUMP year_stats;
```

First, various fields from the input dataset are loaded into the records relation. Next we group records by year. Notice the PARALLEL keyword for setting the number of reducers to use; this is vital when running on a cluster. Then we process each group using a nested FOREACH...GENERATE operator. The first nested statement creates a relation for the distinct USAF identifiers for stations using the DISTINCT operator. The second nested statement creates a relation for the records with "good" readings using the FILTER operator and a UDF. The final nested statement is a GENERATE statement (a nested FOREACH...GENERATE must always have a GENERATE statement as the last nested statement) that generates the summary fields of interest using the grouped records, as well as the relations created in the nested block. Running it on a few years of data, we get the following:

(1920,8L,8595L,8595L)

(1950,1988L,8635452L,8641353L)

(1930,121L,89245L,89262L)

(1910,7L,7650L,7650L)

(1940,732L,1052333L,1052976L)

The fields are year, number of unique stations, total number of good readings, and total number of readings. We can see how the number of weather stations and readings grew over time.

STREAM

The STREAM operator allows you to transform data in a relation using an external program or script. It is named by analogy with Hadoop Streaming, which provides a similar capability for MapReduce. STREAM can use built-in commands with arguments. Here is an example that uses the UNIX cut command to extract the second field of each tuple in A. Note that the command and its arguments are enclosed in back ticks:

```
grunt>C = STREAM A THROUGH `cut -f 2`;

grunt>DUMP C;

(cherry)

(apple)

(banana)

(apple)
```

The STREAM operator uses PigStorage to serialize and deserialize relations to and from the program's standard input and output streams. Tuples in A are converted to tab delimited lines that are passed to the script. The output of the script is read one line at a time and split on tabs to create new tuples for the output relation C. You can provide a custom serializer and deserializer, which implement PigToStream and StreamToPigrespectively (both in the org.apache.pig package), using the DEFINE command. Pig streaming is most powerful when you write custom processing scripts. The following Python script filters out bad 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)
```

To use the script, you need to ship it to the cluster. This is achieved via a DEFINE clause, which also creates an alias for the STREAM command. The STREAM statement can then refer to the alias, as the following Pig script shows:

```
-- max_temp_filter_stream.pig

DEFINE is_good_quality `is_good_quality.py`

SHIP ('ch11/src/main/python/is_good_quality.py');
```

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = STREAM records THROUGH is_good_quality
AS (year:chararray, temperature:int);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
MAX(filtered_records.temperature);
DUMP max_temp;
```

Grouping and Joining Data

Joining datasets in MapReduce takes some work on the part of the programmer, whereas Pig has very good built-in support for join operations, making it much more approachable. Since the large datasets that are suitable for analysis by Pig (and MapReduce in general) are not normalized, joins are used more in frequently in Pig than they are in SQL.

JOIN

Let's look at an example of an inner join. Consider the relations A and B:

```
grunt>DUMP A;
(2,Tie)
(4,Coat)
(3,Hat)
(1,Scarf)
grunt>DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

We can join the two relations on the numerical (identity) field in each:

```
grunt>C = JOIN A BY $0, B BY $1;
grunt>DUMP C;
(2,Tie,Joe,2)
(2,Tie,Hank,2)
(3,Hat,Eve,3)
```

(4,Coat,Hank,4)

This is a classic inner join, where each match between the two relations corresponds to a row in the result. (It's actually an equijoin since the join predicate is equality.) The result's fields are made up of all the fields of all the input relations. You should use the general join operator if all the relations being joined are too large to fit in memory. If one of the relations is small enough to fit in memory, there is a special type of join called a fragment replicate join, which is implemented by distributing the small input to all the mappers and performing a map-side join using an in-memory lookup table against the (fragmented) larger relation. There is a special syntax for telling Pig to use a fragment replicate join:

grunt>C = JOIN A BY $0, B BY $1 USING "replicated";

The first relation must be the large one, followed by one or more small ones (all ofwhich must fit in memory).Pig also supports outer joins using a syntax that is similar to SQL's. For example:

grunt>C = JOIN A BY $0 LEFT OUTER, B BY $1;

grunt>DUMP C;

(1,Scarf,,)

(2,Tie,Joe,2)

(2,Tie,Hank,2)

(3,Hat,Eve,3)

(4,Coat,Hank,4)

COGROUP

JOIN always gives a flat structure: a set of tuples. The COGROUP statement is similar to JOIN, but creates a nested set of output tuples. This can be useful if you want to exploit the structure in subsequent statements:

grunt>D = COGROUP A BY $0, B BY $1;

grunt>DUMP D;

(0,{},{(Ali,0)})

(1,{(1,Scarf)},{})

(2,{(2,Tie)},{(Joe,2),(Hank,2)})

(3,{(3,Hat)},{(Eve,3)})

(4,{(4,Coat)},{(Hank,4)})

COGROUP generates a tuple for each unique grouping key. The first field of each tuple is the key, and the remaining fields are bags of tuples from the relations with a matching key. The first bag contains the matching tuples from relation A with the same key. Similarly, the second bag contains the matching tuples from relation B with the same key. If for a particular key a relation has no matching key, then the bag for that relation is empty. For example, since

no one has bought a scarf (with ID 1), the second bag in the tuple for that row is empty. This is an example of an outer join, which is the default type for COGROUP. It can be made explicit using the OUTER keyword, making this COGROUP statement the same as the previous one:

D = COGROUP A BY $0 OUTER, B BY $1 OUTER;

You can suppress rows with empty bags by using the INNER keyword, which gives the COGROUP inner join semantics. The INNER keyword is applied per relation, so the following only suppresses rows when relation A has no match (dropping the unknown product 0 here):

grunt>E = COGROUP A BY $0 INNER, B BY $1;

grunt>DUMP E;

(1,{(1,Scarf)},{})

(2,{(2,Tie)},{(Joe,2),(Hank,2)})

(3,{(3,Hat)},{(Eve,3)})

(4,{(4,Coat)},{(Hank,4)})

We can flatten this structure to discover who bought each of the items in relation A:

grunt>F = FOREACH E GENERATE FLATTEN(A), B.$0;

grunt>DUMP F;

(1,Scarf,{})

(2,Tie,{(Joe),(Hank)})

(3,Hat,{(Eve)})

(4,Coat,{(Hank)})

Using a combination of COGROUP, INNER, and FLATTEN (which removes nesting)

it’s possible to simulate an (inner) JOIN:

grunt>G = COGROUP A BY $0 INNER, B BY $1 INNER;

grunt>H = FOREACH G GENERATE FLATTEN($1), FLATTEN($2);

grunt>DUMP H;

(2,Tie,Joe,2)

(2,Tie,Hank,2)

(3,Hat,Eve,3)

(4,Coat,Hank,4)

This gives the same result as JOIN A BY $0, B BY $1.
If the join key is composed of several fields, you can specify them all in the BY clauses of the JOIN or COGROUP statement. Make sure that the number of fields in each BY clause is the

same. Here's another example of a join in Pig, in a script for calculating the maximum temperature for every station over a time period controlled by the input:

```
-- max_temp_station_name.pig

REGISTER pig-examples.jar;

DEFINE isGoodcom.hadoopbook.pig.IsGoodQuality();

stations = LOAD 'input/ncdc/metadata/stations-fixed-width.txt'

USING com.hadoopbook.pig.CutLoadFunc('1-6,8-12,14-42')

AS (usaf:chararray, wban:chararray, name:chararray);

trimmed_stations = FOREACH stations GENERATE usaf, wban,

com.hadoopbook.pig.Trim(name);

records = LOAD 'input/ncdc/all/191*'

USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,88-92,93-93')

AS (usaf:chararray, wban:chararray, temperature:int, quality:int);

filtered_records = FILTER records BY temperature != 9999 AND is Good(quality);

grouped_records = GROUP filtered_records BY (usaf, wban) PARALLEL 30;

max_temp = FOREACH grouped_records GENERATE FLATTEN (group),

MAX(filtered_records.temperature);

max_temp_named = JOIN max_temp BY (usaf, wban), trimmed_stations BY (usaf, wban)

PARALLEL 30;

max_temp_result = FOREACH max_temp_named GENERATE $0, $1, $5, $2;

STORE max_temp_result INTO 'max_temp_by_station';
```

We group the filtered weather records by station ID and aggregate by maximum temperature, before joining with the stations. Finally, we project out the fields we want in the final result: USAF, WBAN, station name, maximum temperature.

Here are a few results for the 1910s:

```
228020 99999 SORTAVALA 322

029110 99999 VAASA AIRPORT 300

040650 99999 GRIMSEY 378
```

This query could be made more efficient by using a fragment replicate join, as the station metadata is small.

CROSS

Pig Latin includes the cross-product operator (also known as the cartesian product), which joins every tuple in a relation with every tuple in a second relation (and with every tuple in further relations if supplied). The size of the output is the product of the size of the inputs, potentially making the output very large:

```
grunt>I = CROSS A, B;

grunt>DUMP I;

(2,Tie,Joe,2)

(2,Tie,Hank,4)

(2,Tie,Ali,0)

(2,Tie,Eve,3)

(2,Tie,Hank,2)

(4,Coat,Joe,2)

(4,Coat,Hank,4)

(4,Coat,Ali,0)

(4,Coat,Eve,3)

(4,Coat,Hank,2)

(3,Hat,Joe,2)

(3,Hat,Hank,4)

(3,Hat,Ali,0)

(3,Hat,Eve,3)

(3,Hat,Hank,2)

(1,Scarf,Joe,2)

(1,Scarf,Hank,4)

(1,Scarf,Ali,0)

(1,Scarf,Eve,3)

(1,Scarf,Hank,2)
```

When dealing with large datasets, you should try to avoid operations that generate intermediate representations that are quadratic (or worse) in size. Computing the cross product of the whole input dataset is rarely needed, if ever. For example, at first blush one might expect that calculating pairwise document similarity in a corpus of documents would require every document pair to be generated before calculating their similarity. However, if one starts with the insight that most document pairs have a similarity score of zero (that is, they are unrelated), then we can find a way to a better algorithm.

In this case, the key idea is to focus on the entities that we are using to calculate similarity (terms in a document, for example) and make them the center of the algorithm. In practice, we also remove terms that don't help discriminate between documents (stop words), and this reduces the problem space still further. Using this technique to analyze a set of roughly one million (106) documents generates in the order of one billion (109) intermediate pairs, rather than the one trillion (1012) produced by the naïve approach (generating the cross-product of the input) or the approach with no stop word removal.

GROUP

Although COGROUP groups the data in two or more relations, the GROUP statement groups the data in a single relation. GROUP supports grouping by more than equality of keys: you can use an expression or user-defined function as the group key. For example, consider the following relation A:

grunt>DUMP A;

(Joe,cherry)

(Ali,apple)

(Joe,banana)

(Eve,apple)

Let's group by the number of characters in the second field:

grunt>B = GROUP A BY SIZE($1);

grunt>DUMP B;

(5L,{(Ali,apple),(Eve,apple)})

(6L,{(Joe,cherry),(Joe,banana)})

GROUP creates a relation whose first field is the grouping field, which is given the alias group. The second field is a bag containing the grouped fields with the same schema as the original relation (in this case, A).There are also two special grouping operations: ALL and ANY. ALL groups all the tuples in a relation in a single group, as if the GROUP function was a constant:

grunt>C = GROUP A ALL;

grunt>DUMP C;

(all,{(Joe,cherry),(Ali,apple),(Joe,banana),(Eve,apple)})

Note that there is no BY in this form of the GROUP statement. The ALL grouping is commonly used to count the number of tuples in a relation. The ANY keyword is used to group the tuples in a relation randomly, which can be useful for sampling.

Sorting Data

Relations are unordered in Pig. Consider a relation A:

grunt>DUMP A;

(2,3)

(1,2)

(2,4)

There is no guarantee which order the rows will be processed in. In particular, when retrieving the contents of A using DUMP or STORE, the rows may be written in any order. If you want to impose an order on the output, you can use the ORDER operator to sort a relation by one or more fields. The default sort order compares fields of the same type using the natural ordering, and different types are given an arbitrary, but deterministic, ordering (a tuple is always "less than" a bag, for example).The following example sorts A by the first field in ascending order and by the second field in descending order:

grunt>B = ORDER A BY $0, $1 DESC;

grunt>DUMP B;

(1,2)

(2,4)

(2,3)

Any further processing on a sorted relation is not guaranteed to retain its order. For example:

grunt>C = FOREACH B GENERATE *;

Even though relation C has the same contents as relation B, its tuples may be emitted in any order by a DUMP or a STORE. It is for this reason that it is usual to perform the ORDER operation just before retrieving the output. The LIMIT statement is useful for limiting the number of results, as a quick and dirty way to get a sample of a relation; prototyping (the ILLUSTRATE command) should be preferred for generating more representative samples of the data. It can be used immediately after the ORDER statement to retrieve the first n tuples. Usually, LIMIT will select any n tuples from a relation, but when used immediately after an ORDER statement, the order is retained (in an exception to the rule that processing a relation does not retain its order):

grunt>D = LIMIT B 2;

grunt>DUMP D;

(1,2)

(2,4)

If the limit is greater than the number of tuples in the relation, all tuples are returned(so LIMIT has no effect).Using LIMIT can improve the performance of a query because Pig tries to apply the limit as early as possible in the processing pipeline, to minimize the amount of data that needs to be processed. For this reason, you should always use LIMIT if entire output is not required.

Combining and Splitting Data

Sometimes you have several relations that you would like to combine into one. For this, the UNION statement is used. For example:

```
grunt>DUMP A;

(2,3)

(1,2)

(2,4)

grunt>DUMP B;

(z,x,8)

(w,y,1)

grunt>C = UNION A, B;

grunt>DUMP C;
(z,x,8)

(w,y,1)

(2,3)

(1,2)

(2,4)
```

C is the union of relations A and B, and since relations are unordered, the order of the tuples in C is undefined. Also, it's possible to form the union of two relations with different schemas or with different numbers of fields, as we have done here. Pig attempts to merge the schemas from the relations that UNION is operating on. In this case, they are incompatible, so C has no schema:

```
grunt>DESCRIBE A;

A: {f0: int,f1: int}

grunt>DESCRIBE B;

B: {f0: chararray,f1: chararray,f2: int}

grunt>DESCRIBE C;

Schema for C unknown.
```

If the output relation has no schema, your script needs to be able to handle tuples that vary in the number of fields and/or types. The SPLIT operator is the opposite of UNION; it partitions a relation into two or more relations.

3. 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.

cli

The command line interface to Hive (the shell). This is the default service.

Hive server

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).

hwi

The Hive Web Interface.

jar

The Hive equivalent to hadoop jar, a convenient way to run Java applications that includes both Hadoop and Hive classes on the class path.

metastore

By default, the metastore is run in the same process as the Hive service. Using this service, it is possible to run the metastore as a standalone (remote) process. Set theMETASTORE_PORT environment variable to specify the port the server will listen on.

The Hive Web Interface (HWI)

As an alternative to the shell, you might want to try Hive's simple web interface. Start it using the following commands:

% export ANT_LIB=/path/to/ant/lib

% hive --service hwi

(You only need to set the ANT_LIB environment variable if Ant's library is not found in /opt/ant/lib on your system.) Then navigate to http://localhost:9999/hwi in your browser. From there, you can browse Hive database schemas and create sessions for issuing commands and queries.

It's possible to run the web interface as a shared service to give users within an organization access to Hive without having to install any client software. There are more details on the Hive Web Interface on the Hive wiki at http://wiki.apache.org/hadoop/Hive/HiveWebInterface.

4. HiveQL

Hive's SQL dialect, called HiveQL, does not support the full SQL-92 specification. There are a number of reasons for this. Being a fairly young project, it has not had time to provide the full repertoire of SQL-92 language constructs. More fundamentally,SQL-92 compliance has never been an explicit project goal; rather, as an open source project, features

were added by developers to meet their users' needs. Furthermore, Hive has some extensions that are not in SQL-92, which have been inspired by syntax from other database systems, notably MySQL. In fact, to a first-order approximation, HiveQL most closely resembles MySQL's SQL dialect. Some of Hive's extensions to SQL-92 were inspired by MapReduce, such as multi table inserts and the TRANSFORM, MAP, and REDUCE clauses.

It turns out that some SQL-92 constructs that are missing from HiveQL are easy to work around using other language features, so there has not been much pressure to implement them. For example, SELECT statements do not (at the time of writing) support a HAVING clause in HiveQL, but the same result can be achieved by adding a subquery in the FROM clause. Table 5.1 provides a high-level comparison of SQL and HiveQL.

Table 5.1 A high-level comparison of SQL and HiveQL

Feature	SQL	HiveQL	References
Updates	UPDATE, INSERT, DELETE	INSERT OVERWRITE TABLE (populates whole table or partition)	"INSERT OVERWRITE TABLE" "Updates, Transactions, and Indexes"
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	"Data Types"
Functions	Hundreds of built-in functions	Dozens of built-in functions	"Operators and Functions"
Multitable inserts	Not supported	Supported	"Multitable insert"
Create table as select	Not valid SQL-92, but found in some databases	Supported	"CREATE TABLE...AS SELECT"
Select	SQL-92	Single table or view in the FROM clause. SORT BY for partial ordering. LIMIT to limit number of rows returned. HAVING not supported.	"Querying Data"
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.	"Joins"
Subqueries	In any clause. Correlated or noncorrelated.	Only in the FROM clause. Correlated subqueries not supported	"Subqueries"
Views	Updatable. Materialized or nonmaterialized.	Read-only. Materialized views not supported	"Views"
Extension points	User-defined functions. Stored procedures.	User-defined functions. MapReduce scripts.	"User-Defined Functions" "MapReduce Scripts"

Data Types

Hive supports both primitive and complex data types. Primitives include numeric, boolean, and string types. The complex data types include arrays, maps, and structs. Hive's data types are listed in Table 5.2. Note that the literals shown are those used from within HiveQL; they are not the serialized form used in the table's storage format.

Table 5.2 Hive data types

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"
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]

[a] The literal forms for arrays, maps, and structs are provided as functions. That is, `array()`, `map()`, and `struct()` are built-in Hive functions.

[b] From Hive 0.6.0. The columns are named `col1`, `col2`, `col3`, etc.

Primitive types

Compared to traditional databases, Hive supports only a small set of primitive data types. There is currently no support for temporal types (dates and times), although there are functions for converting UNIX timestamps (stored as integers) to strings, which makes most common date operations tractable using Hive. Hive's primitive types correspond roughly to Java's, although some names are influenced by MySQL's type names (some of which, in turn, overlap with SQL-92). There are four signed integral types: TINYINT, SMALLINT, INT, and BIGINT, which are equivalent to Java's byte, short, int, and long primitive types, respectively; they are 1-byte, 2-byte,4-byte, and 8-byte signed integers.

Hive's floating-point types, FLOAT and DOUBLE, correspond to Java's float and double, which are 32-bit and 64-bit floating point numbers. Unlike some databases, there is no option to control the number of significant digits or decimal places stored for floating point values. Hive supports a BOOLEAN type for storing true and false values. There is a single Hive data type for storing text, STRING, which is a variable-length character string. Hive's STRING type is like VARCHAR in other databases, although there is no declaration of the maximum number of characters to store with STRING. (The theoretical maximum size STRING that may be stored is 2GB, although in practice it maybe inefficient to materialize such large values. Sqoop has large object support.)

Conversions

Primitive types form a hierarchy, which dictates the implicit type conversions that Hive will perform. For example, a TINYINT will be converted to an INT, if an expression expects an INT; however, the reverse conversion will not occur and Hive will return an error unless the CAST operator is used. The implicit conversion rules can be summarized as follows. Any integral numeric type can be implicitly converted to a wider type. All the integral numeric types, FLOAT, and (perhaps surprisingly) STRING can be implicitly converted to DOUBLE. TINYINT, SMALLINT, and INT can all be converted to FLOAT. BOOLEAN types cannot be converted to anyother type. You can perform explicit type conversion using CAST. For example, CAST ('1' AS INT)will convert the string '1' to the integer value 1. If the cast fails—as it does in CAST ('X'AS INT), for example—then the expression returns NULL.

Complex types

Hive has three complex types: ARRAY, MAP, and STRUCT. ARRAY and MAP are like their names in Java, while a STRUCT is a record type which encapsulates a set of named fields. Complex types permit an arbitrary level of nesting. Complex type declarations must specify the type of the fields in the collection, using an angled bracket notation, as illustrated in this table definition which has three columns, one for each complex type:

```
CREATE TABLE complex (
col1 ARRAY<INT>,
col2 MAP<STRING, INT>,
col3 STRUCT<a:STRING, b:INT, c:DOUBLE>
);
```

If we load the table with one row of data for ARRAY, MAP, and STRUCT shown in the "Literal examples" column in Table 12-3, then the following query demonstrates the field access or operators for each type:

```
hive>SELECT col1[0], col2['b'], col3.c FROM complex;
1 2 1.0
```

Operators and Functions

The usual set of SQL operators is provided by Hive: relational operators (such as x ='a' for testing equality, x IS NULL for testing nullity, x LIKE 'a%' for pattern matching), arithmetic operators (such as x + 1 for addition), and logical operators (such as x OR y for logical OR). The operators match those in MySQL, which deviates from SQL-92since || is logical OR, not string concatenation. Use the concat function for the latter in both MySQL and Hive.

Hive comes with a large number of built-in functions—too many to list here—divided into categories including mathematical and statistical functions, string functions, date functions (for operating on string representations of dates), conditional functions, aggregate functions, and functions for working with XML (using the xpathfunction) and JSON.

You can retrieve a list of functions from the Hive shell by typing SHOW FUNCTIONS.# To get brief usage instructions for a particular function, use the DESCRIBE command:

hive>DESCRIBE FUNCTION length;

length(str) - Returns the length of str

5. Querying Data in Hive

This section discusses how to use various forms of the SELECT statement to retrieve data from Hive.

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. When a globally sorted result is not required—and in many cases it isn't—then you can use Hive's nonstandard extension, SORT BY instead. SORT BY produces a sorted file per reducer.

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, in such a way to ensure that all the rows for a given year end up in the same reducer partition:

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

A follow-on query would be able to use the fact that each year's temperatures were grouped and sorted (in descending order) in the same file. If the columns for SORT BY and DISTRIBUTE BY are the same, you can use CLUSTER BY as shorthand for specifying both.

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. Suppose we want to use a script to filter out rows that don't meet some condition, such as the script in Example, which removes poor quality readings.

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 SELECTTRANSFORM in both cases would have the same result. The source for the max_temperature_reduce.py script is shown in Example:

```
FROM (

FROM records2

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;
```

Joins

One of the nice things about using Hive, rather than raw MapReduce, is that it makes performing commonly used operations very simple. Join operations are a case in point, given how involved they are to implement in MapReduce.

Inner joins

The simplest kind of join is the inner join, where each match in the input tables results in a row in the output. Consider two small demonstration tables: sales, which lists the names of people and the ID of the item they bought; and things, which lists the itemID and its name:

```
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 an inner join on the two tables as follows:

```
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. In Hive, you can join on multiple columns in the join predicate by specifying a series of expressions, separated by AND keywords. You can also join more than two tables by supplying additional JOIN...ON... clauses in the query. Hive is intelligent about trying to minimize the

number of MapReduce jobs to perform the joins. A single join is implemented as a single MapReduce job, but multiple joins can be performed in less than one MapReduce job per join if the same column is used in the join condition. You can see how many MapReduce jobs Hive will use for any particular query by prefixing it with the EXPLAIN keyword:

EXPLAIN

SELECT sales.*, things.*

FROM sales JOIN things ON (sales.id = things.id);

The EXPLAIN output includes many details about the execution plan for the query, including the abstract syntax tree, the dependency graph for the stages that Hive will execute, and information about each stage. Stages may be MapReduce jobs or operations such as file moves. For even more detail, prefix the query with EXPLAIN EXTENDED. Hive currently uses a rule-based query optimizer for determining how to execute a query, but it's likely that in the future a cost-based optimizer will be added.

Outer joins

Outer joins allow you to find non matches in the tables being joined. In the current example, when we performed an inner join, 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. 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 sales.*, things.*

>FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);

Ali 0 NULL NULL
Joe 2 2 Tie

Hank 2 2 Tie

Eve 3 3 Hat

Hank 4 4 Coat

Notice that the row for Ali is now returned, and the columns from the things table are NULL, since there is no match. Hive supports right outer joins, which reverses the roles of the tables relative to the left join. In this case, all items from the things table are included, even those that weren't purchased by anyone (a scarf):

hive>SELECT sales.*, things.*

>FROM sales RIGHT OUTER JOIN things ON (sales.id = things.id);

NULL NULL 1 Scarf

Joe 2 2 Tie

Hank 2 2 Tie

Eve 3 3 Hat

Hank 4 4 Coat

Finally, there is a full outer join, where the output has a row for each row from both tables in the join:

hive>SELECT sales.*, things.*

>FROM sales FULL OUTER JOIN things ON (sales.id = things.id);

Ali 0 NULL NULL

NULL NULL 1 Scarf

Joe 2 2 Tie

Hank 2 2 Tie

Eve 3 3 Hat

Hank 4 4 Coat

Semi joins

Hive doesn't support IN subqueries (at the time of writing), but you can use a LEFT SEMIJOIN to do the same thing. Consider this IN subquery, which finds all the items in the things table that are in the sales table:

SELECT *

FROM things

WHERE things.id IN (SELECT id from sales);

We can rewrite it as follows:

hive>SELECT *

>FROM things LEFT SEMI JOIN sales ON (sales.id = things.id);

2 Tie

3 Hat

4 Coat

There is a restriction that we must observe for LEFT SEMI JOIN queries: the right table (sales) may only appear in the ON clause. It cannot be referenced in a SELECT expression, for example.

Map joins

If one table is small enough to fit in memory, then Hive can load the smaller table into memory to perform the join in each of the mappers. The syntax for specifying a mapjoin is a hint embedded in an SQL C-style comment:

SELECT /*+ MAPJOIN(things) */ sales.*, things.*

FROM sales JOIN things ON (sales.id = things.id);

The job to execute this query has no reducers, so this query would not work for a RIGHT or FULL OUTER JOIN, since absence of matching can only be detected in an aggregating (reduce) step across all the inputs. Map joins can take advantage of bucketed tables, since a mapper working on a bucket of the left table only needs to load the corresponding buckets of the right table to perform the join. The syntax for the join is the same as for the in-memory case above; however, you also need to enable the optimization with:

SET hive.optimize.bucketmapjoin=true;

Subqueries

A subquery is a SELECT statement that is embedded in another SQL statement. Hive has limited support for subqueries, only permitting a subquery in the FROM clause of a SELECT statement. Other databases allow subqueries almost anywhere that an expression is valid, such as in the list of values to retrieve from a SELECT state mentor in the WHERE clause. Many uses of subqueries can be rewritten as joins, so if you find yourself writing a subquery where Hive does not support it, then see if it can be expressed as a join. For example, an IN sub query can be written as a semi join, or an inner join. The following 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 sub query is used to find the maximum temperature for each station/date combination, then 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 sub query like it does a table, which is why the sub query must be given an alias (mt). The columns of the sub query have to be given unique names so that the outer query can refer to them.

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. Often, the data from existing tables is simplified or aggregated in a particular way that makes it convenient for further processing. Views may also be used to restrict users’ access to particular subsets of tables that they are authorized to see. In Hive, a view is not materialized to disk when it is created; rather, the view’s SELECT statement is executed when the statement that refers

to the view is run. If a view performs extensive transformations on the base tables, or is used frequently, then you may choose to manually materialize it by creating a new table that stores the contents of the view. We can use views to rework the query from the previous section for finding the mean maximum temperature for every year and weather station. First, let's create view for valid records, that is, records that have a particular quality value:

```
CREATE VIEW valid_records

AS

SELECT *

FROM records2

WHERE temperature != 9999

AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9);
```

When we create a view, the query is not run; it is simply stored in the metastore. Views are included in the output of the SHOW TABLES command, and you can see more details about a particular view, including the query used to define it, by issuing the DESCRIBEEXTENDED view_name command.Next, let's create a second view of maximum temperatures for each station and year. It is based on the valid_recordsview:

```
CREATE VIEW max_temperatures (station, year, max_temperature)

AS

SELECT station, year, MAX(temperature)

FROM valid_records

GROUP BY station, year;
```

In this view definition, we list the column names explicitly. We do this since the maximum temperature column is an aggregate expression, and otherwise Hive would create a column alias for us (such as _c2). We could equally well have used an AS clause in the SELECT to name the column.

With the views in place, we can now use them by running a query:

```
SELECT station, year, AVG (max_temperature)

FROM max_temperatures

GROUP BY station, year;
```

The result of the query is the same as running the one that uses a sub query and, in particular, the number of MapReduce jobs that Hive creates is the same for both: two in each case, one for each GROUP BY. This example shows that Hive can combine a query on a view into a sequence of jobs that is equivalent to writing the query without using a view. In other words, Hive won't needlessly materialize a view even at execution time. Views in Hive are read-only, so there is no way to load or insert data into an underlying base table via a view.

6. FUNDAMENTALS OF HBASE AND ZOOKEEPER

HBase

HBase is a distributed column-oriented database built on top of HDFS. HBase is the Hadoop application to use when you require real-time read/write random-access to very large datasets. Although there are countless strategies and implementations for database storage and retrieval, most solutions—especially those of the relational variety—are not built with very large scale and distribution in mind. Many vendors offer replication and partitioning solutions to grow the database beyond the confines of a single node, but these add-ons are generally an afterthought and are complicated to install and maintain. They also come at some severe compromise to the RDBMS feature set. Joins, complex queries, triggers, views, and foreign-key constraints become prohibitively expensive to run on a scaled RDBMS or do not work at all. HBase comes at the scaling problem from the opposite direction. It is built from the ground-up to scale linearly just by adding nodes. HBase is not relational and does not support SQL, but given the proper problem space, it is able to do what an RDBMS cannot: host very large, sparsely populated tables on clusters made from commodity hardware.

The canonical HBase use case is the web table, a table of crawled web pages and their attributes (such as language and MIME type) keyed by the web page URL. The web table is large, with row counts that run into the billions. Batch analytic and parsing MapReduce jobs are continuously run against the web table deriving statistics and adding new columns of verified MIME type and parsed text content for later indexing by a search engine. Concurrently, the table is randomly accessed by crawlers running at various rates updating random rows while random web pages are served in real time as users click on a website's cached-page feature. Next discussion is about building general distributed applications using Hadoop's distributed coordination service, called ZooKeeper.

Writing distributed applications is hard. It's hard primarily because of partial failure. When a message is sent across the network between two nodes and the network fails, the sender does not know whether the receiver got the message. It may have gotten through before the network failed, or it may not have. Or perhaps the receiver's process died. The only way that the sender can find out what happened is to reconnect to the receiver and ask it. This is partial failure: when we don't even know if an operation failed.

ZooKeeper can't make partial failures go away, since they are intrinsic to distributed systems. It certainly does not hide partial failures, either. But what ZooKeeper - give you a set of tools to build distributed applications that can safely handle partial failures.

ZooKeeper also has the following characteristics:

ZooKeeper is simple

ZooKeeper is, at its core, a stripped-down file system that exposes a few simple operations, and some extra abstractions such as ordering and notifications.

ZooKeeper is expressive

The ZooKeeper primitives are a rich set of building blocks that can be used to build a large class of coordination data structures and protocols. Examples include: distributed queues, distributed locks, and leader election among a group of peers.

ZooKeeper is highly available

ZooKeeper runs on a collection of machines and is designed to be highly available, so applications can depend on it. ZooKeeper can help you avoid introducing single points of failure into your system, so you can build a reliable application.

ZooKeeper facilitates loosely coupled interactions

ZooKeeper interactions support participants that do not need to know about one another. For example, ZooKeeper can be used as a rendezvous mechanism so that processes that otherwise doesn't know of each other's existence (or network details) can discover and interact with each other. Coordinating parties may not even be contemporaneous; since one process may leave a message in ZooKeeper that is read by another after the first has shut down.

ZooKeeper is a library

ZooKeeper provides an open source, shared repository of implementations and recipes of common coordination patterns. Individual programmers are spared the burden of writing common protocols themselves (which are often difficult to get right). Over time, the community can add to and improve the libraries, which is to everyone's benefit.

ZooKeeper is highly performant, too. At Yahoo!, where it was created, ZooKeeper's throughput has been benchmarked at over 10,000 operations per second for write dominant workloads. For workloads where reads dominate, which is the norm, the throughput is several times higher.