UNIT I - Big Data

UNIT 1 INTRODUCTION **9 Hrs.**

Introduction to Big Data - Challenges of Conventional Systems - Nature of Data - Small data - Medium data - Big Data - Small data vs Big data - Sources of Big Data - Big Data Characteristics - Big Data Analytics - Importance of Big Data, Big Data in the Enterprise - Big Data Enterprise Model - Building a Big Data Platform - Big data in Social and Behavioral sciences.

Introduction

What is Data?

Data can be defined as a representation of facts, concepts, or instructions in a formalized manner.

Table 1.1 Characteristics of Data

Accuracy	Is the information correct in every detail?
Completeness	How comprehensive is the information?
Reliability	Does the information contradict other trusted resources?
Relevance	Do you really need this information?
Timeliness	How up- to-date is information? Can it be used for real-time reporting?

Differences between Small Data, Medium Data and Big Data

Data can be small, medium or big.

Small data is data in a volume and format that makes it accessible, informative and actionable.

Medium data refers to data sets that are too large to fit on a single machine but don't require enormous clusters of thousands.

Big data is extremely large data sets that may be analysed computationally to reveal patterns, trends, and associations, especially relating to human behaviour and interactions.

Table 1.2 Small Data and Big Data Comparison Table

Basis of Comparison	Small Data	Big Data
Definition	Data that is 'small' enough for human .In a volume and format that makes it accessible, informative and actionable	Data sets that are so large or complex that traditional data processing applications cannot deal with them
Data Source	● Data from traditional enterprise systems like	● Purchase data from point-of-sale

	○ Enterprise resource planning ○ Customer relationship management(CRM)	● Clickstream data from websites ● GPS stream data – Mobility data sent to a server ● Social media – Facebook, Twitter
Volume	Most cases in a range of tens or hundreds of GB.Some case few TBs (1 TB=1000 GB)	More than a few Terabytes (TB)
Velocity (Rate at which data appears)	● Controlled and steady data flow ● Data accumulation is slow	● Data can arrive at very fast speeds. ● Enormous data can accumulate within very short periods of time
Variety	Structured data in tabular format with fixed schema and semi-structured data in JSON or XML format	High variety data sets which include Tabular data,Text files, Images, Video, Audio, XML,JSON,Logs,Sensor data etc.
Veracity (Quality of data)	Contains less noise as data collected in a controlled manner.	Usually, the quality of data not guaranteed. Rigorous data validation is required before processing.
Value	Business Intelligence, Analysis, and Reporting	Complex data mining for prediction, recommendation, pattern finding, etc.
Time Variance	Historical data equally valid as data represent solid business interactions	In some cases, data gets older soon(Eg fraud detection).
Data Location	Databases within an enterprise, Local servers, etc.	Mostly in distributed storages on Cloud or in external file systems.
Infrastructure	Predictable resource allocation.Mostly vertically scalable hardware	More agile infrastructure with a horizontally scalable architecture. Load on the system varies a lot.

Introduction to Big Data

Big data is data that exceeds the processing capacity of conventional database systems. The data is too big, moves too fast, or doesn't fit the strictures of your database architectures. To gain value from this data, you must choose an alternative way to process it. 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 that treats ways to analyze, systematically extract information from, or otherwise deal with data sets that are too large or complex to be dealt with by traditional dataprocessing application software. – Wikipedia

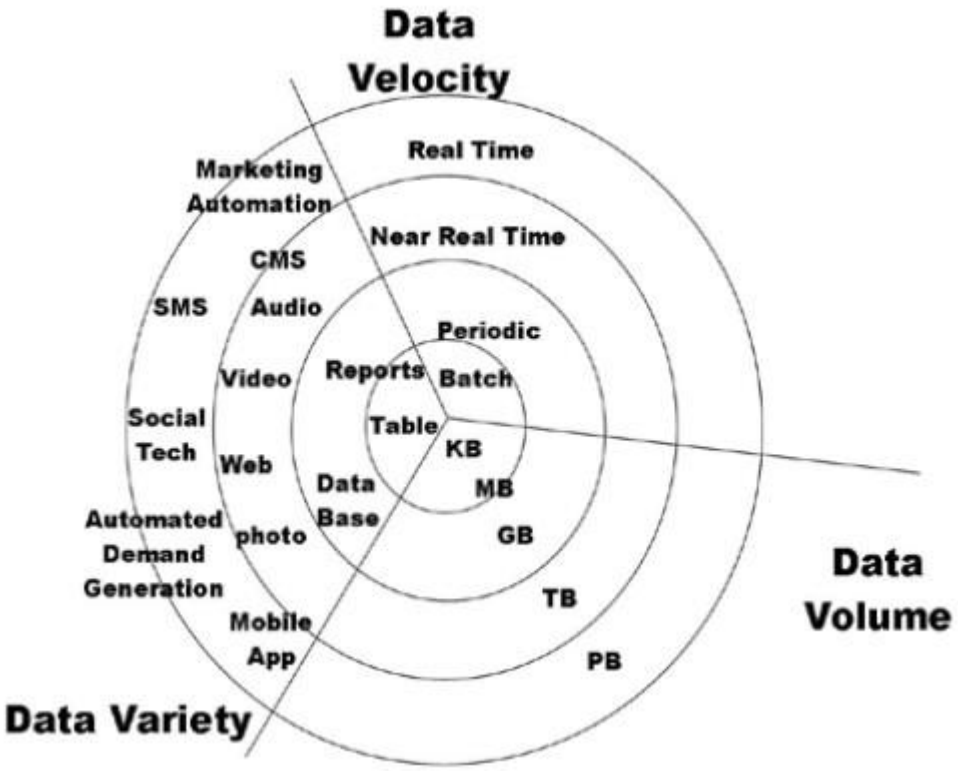


Fig. 1.1 3Vs of Big Data

Examples of Big Data:

The New York Stock Exchange generates about one terabyte of new trade data per day.

The statistic shows that 500+terabytes of new data get ingested into the databases of social media site Facebook, every day. This data is mainly generated in terms of photo and video uploads, message exchanges, putting comments etc.

A single Jet engine can generate 10+terabytes of data in 30 minutes of flight time. With many thousand flights per day, generation of data reaches up to many Petabytes.

Table 1.3 Examples of Data Volumes

Unit	Value	Example
Kilobytes (KB)	1,000 bytes	a paragraph of a text document
Megabytes (MB)	1,000 Kilobytes	a small novel
Gigabytes (GB)	1,000 Megabytes	Beethoven's 5th Symphony
Terabytes (TB)	1,000 Gigabytes	all the X-rays in a large hospital
Petabytes (PB)	1,000 Terabytes	half the contents of all US academic research libraries
Exabytes (EB)	1,000 Petabytes	about one fifth of the words people have ever spoken

Zettabytes (ZB)	1,000 Exabytes	as much information as there are grains of sand on all the world's beaches
Yottabytes (YB)	1,000 Zettabytes	as much information as there are atoms in 7,000 human bodies

Advantages of using Big Data

1. Improved business processes
2. Fraud detection
3. Improved customer service
4. Better decision-making
5. Increased productivity
6. Reduce costs
7. Improved customer service
8. Fraud detection
9. Increased revenue
10. Increased agility
11. Greater innovation
12. Faster speed to market

Disadvantages of Big Data

1. Privacy and security concerns
2. Need for technical expertise
3. Need for talent
4. Data quality
5. Need for cultural change
6. Compliance
7. Cybersecurity risks
8. Rapid change
9. Hardware needs
10. Costs
11. Difficulty integrating legacy systems

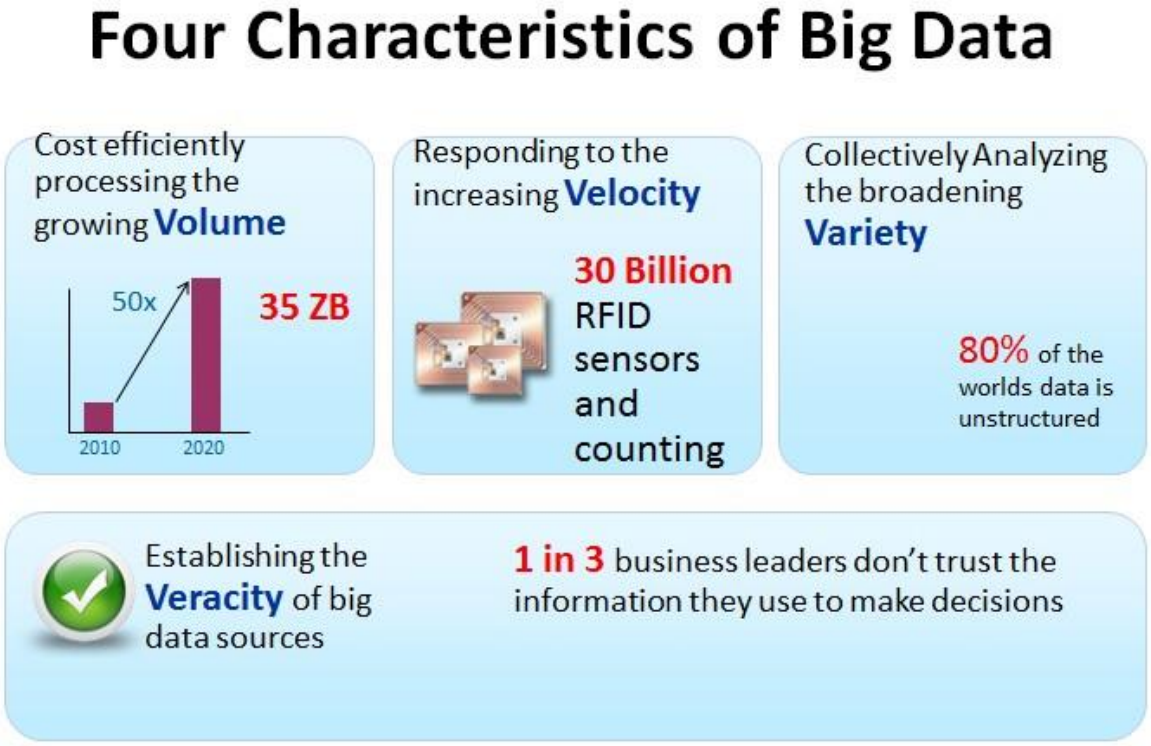


Fig 1.2 Four Characteristics of Big Data

Characteristics of Big Data (3 Vs of Big Data) 3Vs

of Big Data = Volume, Velocity and Variety.

1. Volume:

Volume refers to the sheer size of the ever-exploding data of the computing world. It raises the question about the quantity of data collected from different sources over the Internet

2. Velocity:

Velocity refers to the processing speed. It raises the question of at what speed the data is processed. The speed is measured by the use of the data in a specific time period.In Big Data velocity data flows in from sources like machines, networks, social media, mobile phones etc.There is a massive and continuous flow of data. This determines the potential of data that how fast the data is generated and processed to meet the demands.

3. Variety:

Variety: Variety refers to the types of data. In Big Data the raw data always collected in variety. The raw data can be structured, unstructured, and semi structured. This is because the data is collected from various sources.It also refers to heterogeneous sources.

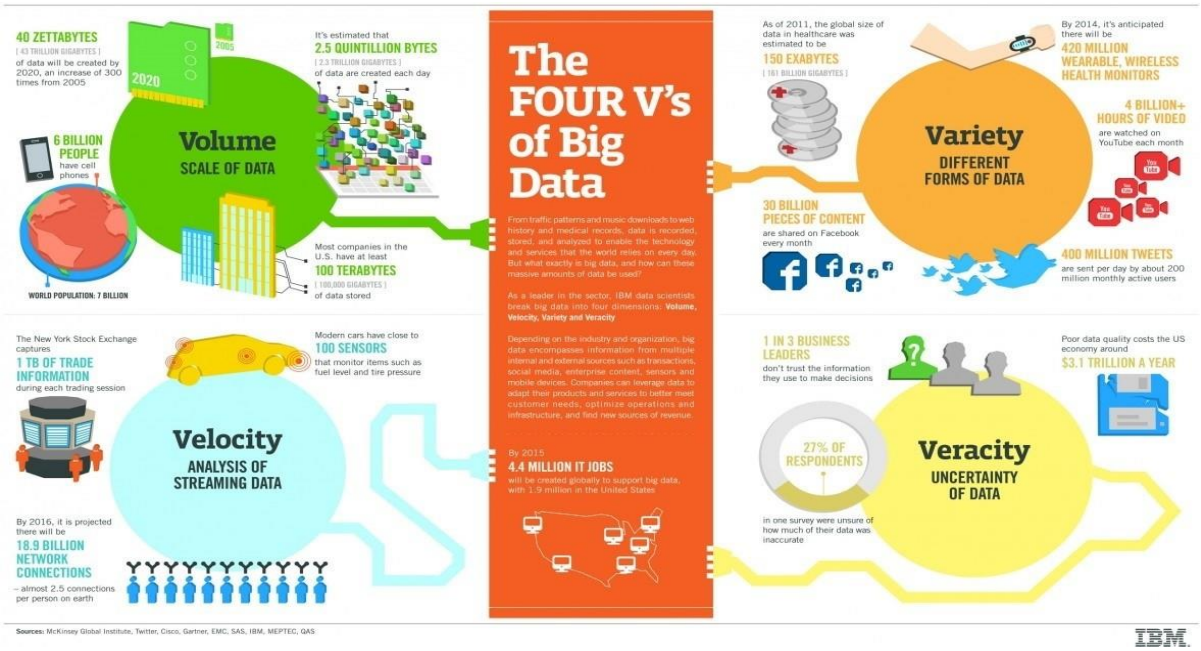


Fig. 1.3 Four Vs of Big Data 4Vs

of Big Data = Volume, Velocity, Variety, Veracity

4. Veracity:

Veracity is all about the trust score of the data. If the data is collected from trusted or reliable sources then the data neglect this rule of big data.It refers to inconsistencies and uncertainty in data, that is data which is available can sometimes get messy and quality and accuracy are difficult to control.Big Data is also variable because of the multitude of data dimensions resulting from multiple disparate data types and sources.Example: Data in bulk could create confusion whereas less amount of data could convey half or Incomplete Information. **5Vs of Big Data = Volume, Velocity, Variety, Veracity,**

5. Value:

Value refers to purpose, scenario or business outcome that the analytical solution has to address.Does the data have value, if not is it worth being stored or collected?The analysis needs to be performed to meet the ethical considerations.

6Vs of Big Data = Volume, Velocity, Variety, Veracity, Variability

6. Variability

This refers to establishing if the contextualizing structure of the data stream is regular and dependable even in conditions of extreme unpredictability.It defines the need to get meaningful data considering all possible circumstances.

7Vs of Big Data =Volume, Velocity, Variety, Veracity, Variability, Visualization

7. Visualization:

Visualization is critical in today's world. Using charts and graphs to visualize large amounts of complex data is much more effective in conveying meaning than spreadsheets and reports chock-full of numbers and formulas.

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

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 allencompassing 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.

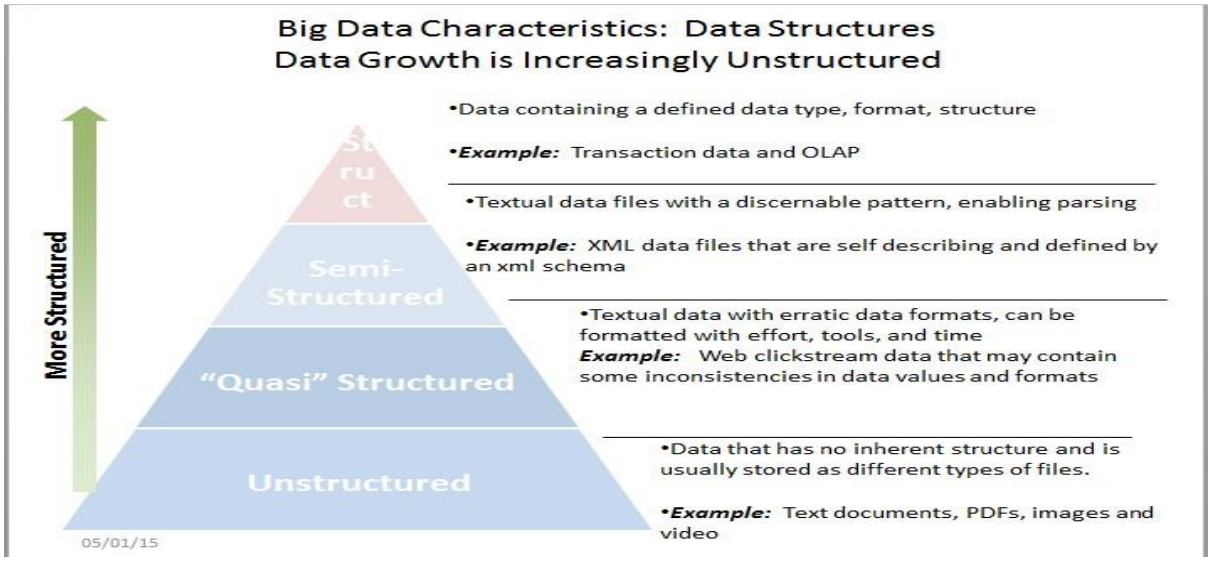


Fig. 1.4. Nature of Data Structured

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. (We know that people buy more Pop-Tarts at Walmart when a storm is predicted.)"

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 behavior. Every time someone enters a search term on Google that is data that can be captured for future benefit. The GPS info on our smartphones 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

Classification of Types of Big Data

The following classification was developed by the Task Team on Big Data, in June 2013.

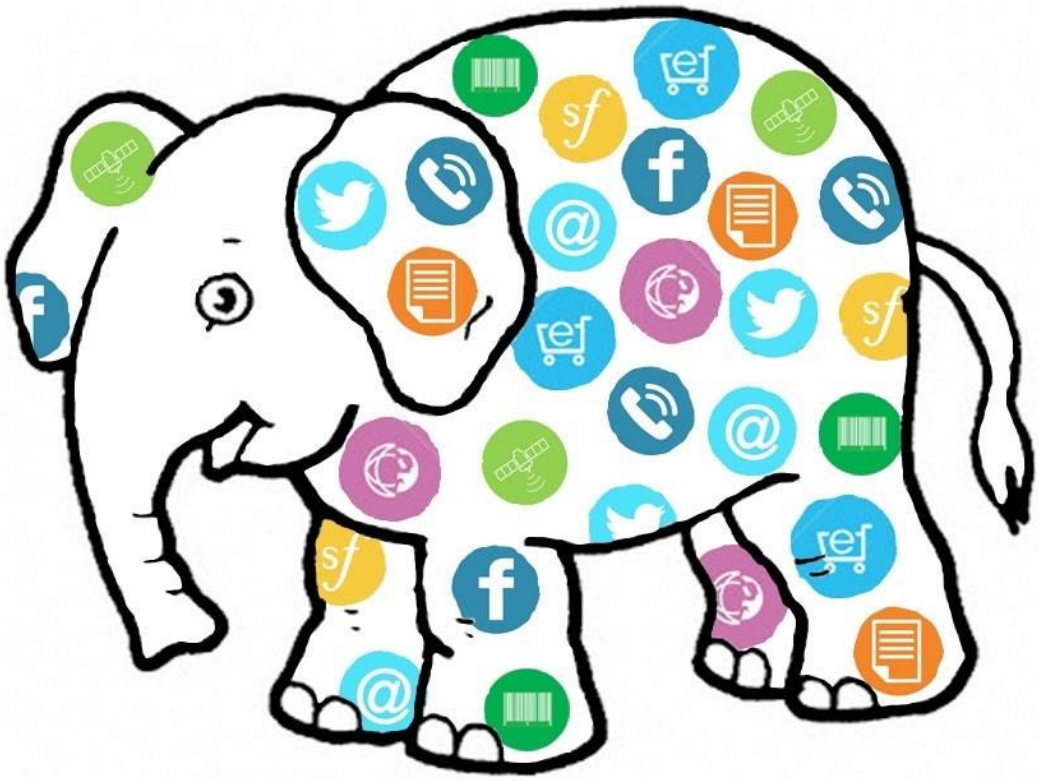


Fig. 1.5 Sources of Big Data

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.

1100. Social Networks: Facebook, Twitter, Tumblr etc.

1200. Blogs and comments

1300. Personal documents

1400. Pictures: Instagram, Flickr, Picasa etc.

1500. Videos: Youtube etc.

1600. Internet searches

1700. Mobile data content: text messages

1800. User-generated maps

1900. 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. (Some sources belonging to this class may fall into the category of "Administrative data").

21. Data produced by Public Agencies

    2110. Medical records

22. Data produced by businesses

    2210. Commercial transactions

    2220. Banking/stock records

    2230. E-commerce

    2240. 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.

31. Data from sensors

    311. Fixed sensors

    3111. Home automation

    3112. Weather/pollution sensors

    3113. Traffic sensors/webcam

    3114. Scientific sensors

    3115. Security/surveillance videos/images

    312. Mobile sensors (tracking)

    3121. Mobile phone location

    3122. Cars

    3123. Satellite images

32. Data from computer systems

    3210. Logs

    3220. Web logs

BIG DATA ENTERPRISE ARCHITECTURE

The 5 V's of Big Data:

Too often in the hype and excitement around Big Data, the conversation gets complicated very quickly. Data scientists and technical experts bandy around terms like Hadoop, Pig, Mahout, and Sqoop, making us wonder if we're talking about information architecture or a Dr. Seuss book. Business executives who want to leverage the value of Big Data analytics in their organisation can get lost amidst this highly-technical and rapidly-emerging ecosystem. In an effort to simplify Big Data, many experts have referenced the "3 V's": Volume, Velocity, and Variety. In other words, is information being generated at a high volume (e.g. terabytes per day), with a rapid rate of change, encompassing a broad range of sources including both structured and unstructured data? If the answer is yes then it falls into the Big Data category along with sensor data from the "internet of things", log files, and social media streams. The ability to understand and manage these sources, and then integrate them into the larger Business Intelligence ecosystem can provide previously unknown insights from data and this understanding leads to the "4th V" of Big Data – Value.

There is a vast opportunity offered by Big Data technologies to discover new insights that drive significant business value. Industries are seeing data as a market differentiator and have started reinventing themselves as "data companies", as they realise that information has become their biggest asset. This trend is prevalent in industries such as telecommunications, internet search firms, marketing firms, etc. who see their data as a key driver for monetisation and growth. Insights such as footfall traffic patterns from mobile devices have been used to assist city planners in designing more efficient traffic flows. Customer sentiment analysis through social media and call logs have given new insights into customer satisfaction. Network performance patterns have been analysed to discover new ways to drive efficiencies. Customer usage patterns based on web click-stream data have driven innovation for new products and services to increase revenue. The list goes on.

Key to success in any Big Data analytics initiative is to first identify the business needs and opportunities, and then select the proper fit-for-purpose platform. With the array of new Big Data technologies emerging at a rapid pace, many technologists are eager to be the first to test the latest Dr. Seuss-termed platform. But each technology has a unique specialisation, and might not be aligned to the business priorities. In fact, some identified use cases from the business might be best suited by existing technologies such as a data warehouse while others require a combination of existing technologies and new Big Data systems.

With this integration of disparate data systems comes the 5th V – Veracity, i.e. the correctness and accuracy of information. Behind any information management practice lies the core doctrines of Data Quality, Data Governance, and Metadata Management, along with considerations for Privacy and Legal concerns. Big Data needs to be integrated into the entire information landscape, not seen as a stand-alone effort or a stealth project done by a handful of Big Data experts.

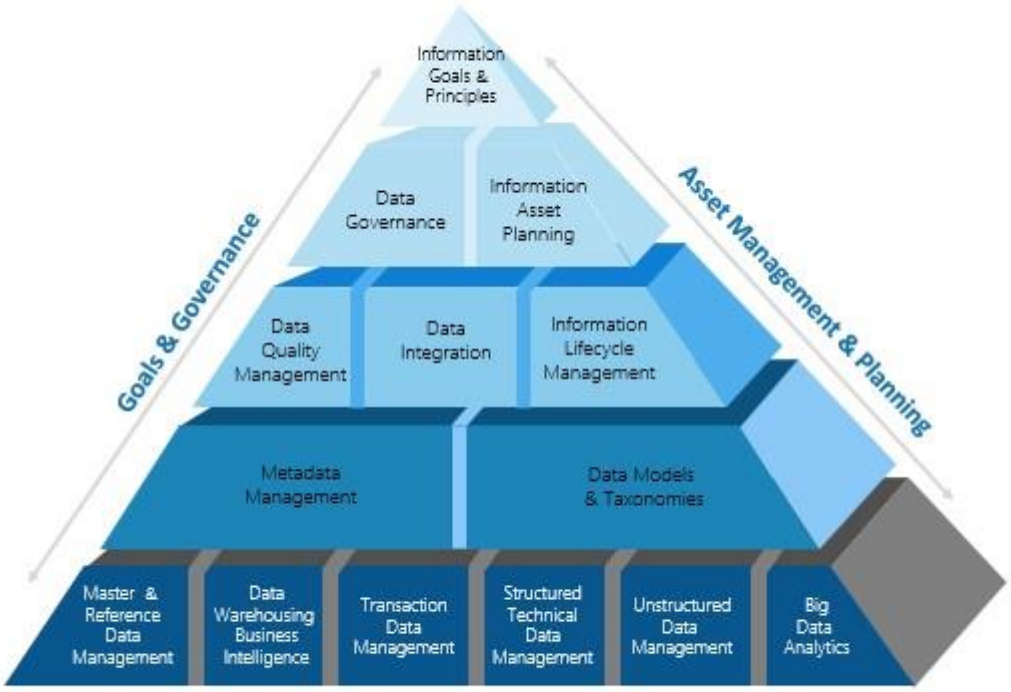


Fig. 1.6 Enterprise Architects Information Management Framework

In the excitement and hype around Big Data analytics, it's easy to see this emerging technology as a "silver bullet" that can magically generate new insights solely through powerful technology and smart data scientists. As in any age of change, however, core principles still apply, and in order to gain insights from Big Data, you need to make sure your "little data" is correct. Many of the "golden nuggets" of discovery are obtained through an intersection of Big Data analytics with traditional sources such as a data warehouses or master data management hubs.

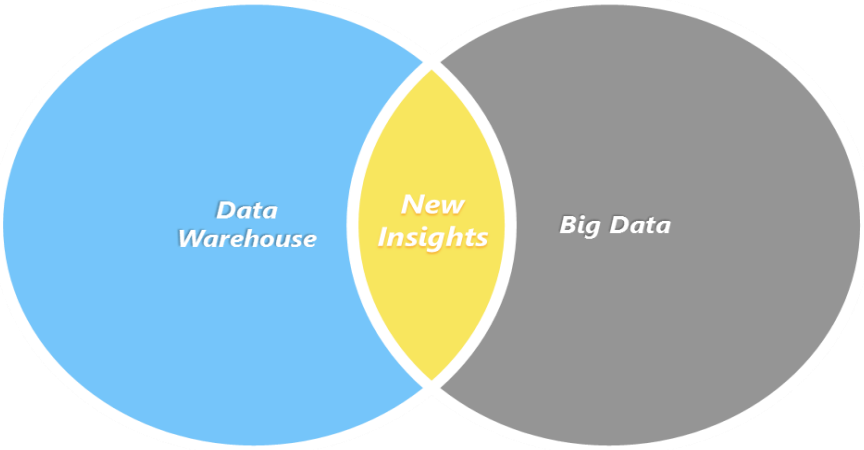


Fig.1.7 The intersection of Big Data analytics with traditional sources such as a data warehouse

Customer sentiment analysis is a common use-case for Big Data analytics—i.e. what are our customers saying about our products in social media and/or call log records? And how can we

leverage this information to improve our business? Unless you have a robust 'single source of record' for customer information, new discoveries from Big Data analytics will be of little use. Was it Jane R. Doe or Jane P. Doe complaining about the new luxury sedan model? With data properly managed within an information management framework, the full value of Big Data becomes apparent and "golden nuggets" of information can appear. For example, not only did Jane R. Doe complain about the new luxury sedan, but she had five service calls about her transmission. She has purchased five high-priced sedans from us in the past ten years and has an income of over $750,000. Jane R. Doe recently followed our competitor on Twitter and has asked several questions about new features. It might be worth having a representative call her personally.

Big Data analytics is an exciting development in the field of information management and, if used properly, can generate a wealth of opportunity. In order to discover the "golden nuggets" in your organisation, remember these guiding principles:

•Start with your business goals and drivers and align them to fit-for-purpose technologies (not the other way around)

•Integrate your Big Data initiatives with core information management practices

•Build your information management practice on a core framework that includes data governance, data quality management, data quality, and the other principles that create a trusted source of information

Lastly, have fun—this is an exciting time to be in information management. New technologies are emerging almost daily that can add significant value to your organisation, particularly in the Big Data space.

A big data architecture is designed to handle the ingestion, processing, and analysis of data that is too large or complex for traditional database systems. The threshold at which organizations enter into the big data realm differs, depending on the capabilities of the users and their tools. For some, it can mean hundreds of gigabytes of data, while for others it means hundreds of terabytes. Over the years, the data landscape has changed. What you can do, or are expected to do, with data has changed. The cost of storage has fallen dramatically, while the means by which data is collected keeps growing. Some data arrives at a rapid pace, constantly demanding to be collected and observed. Other data arrives more slowly, but in very large chunks, often in the form of decades of historical data. You might be facing an advanced analytics problem, or one that requires machine learning. These are challenges that big data architectures seek to solve.

Big data solutions typically involve one or more of the following types of workload:

- Batch processing of big data sources at rest.
- Real-time processing of big data in motion.
- Interactive exploration of big data.
- analytics and machine learning. Predictive

Consider big data architectures when you need to:

- process data in volumes too large for a traditional database. Store and
- unstructured data for analysis and reporting. Transform
- Capture, process, and analyze unbounded streams of data in real time, or with low latency.

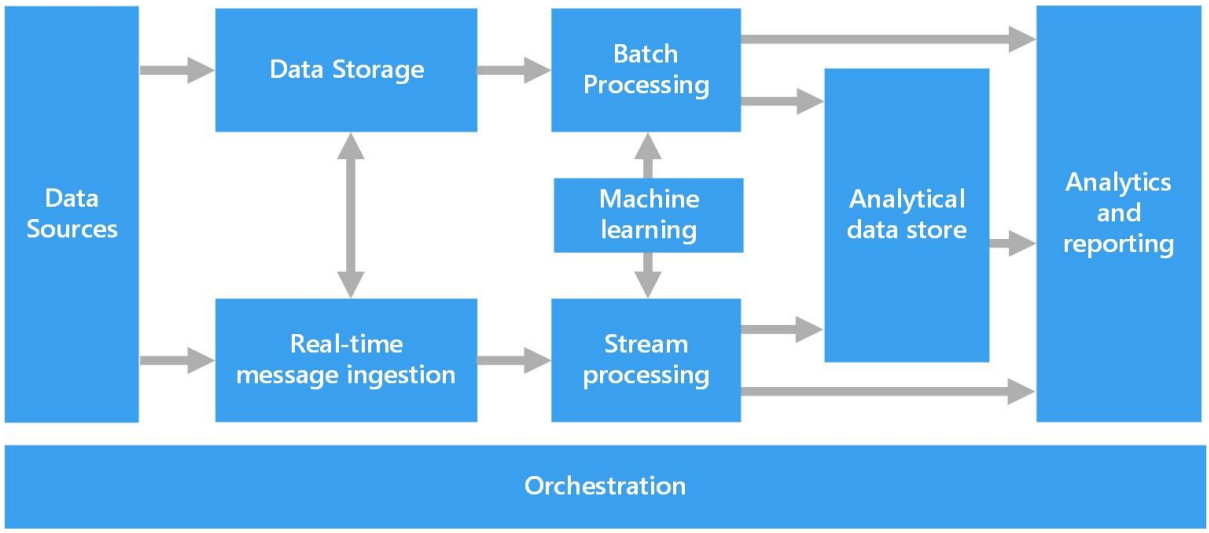


Fig. 1.8 Components of a Big Data Architecture

Components of a Big Data Architecture: Logical components that fit into a big data architecture

Most big data architectures include some or all of the following components:

Data sources: All big data solutions start with one or more data sources. Examples includeApplication data stores, such as relational databases.Static files produced by applications, such as web server log files.Real-time data sources, such as IoT devices.

Data storage: Data for batch processing operations is typically stored in a distributed file store that can hold high volumes of large files in various formats. This kind of store is often called a data lake. Options for implementing this storage include Azure Data Lake Store or blob containers in Azure Storage.

Real-time message ingestion: If the solution includes real-time sources, the architecture must include a way to capture and store real-time messages for stream processing. This might be a simple data store, where incoming messages are dropped into a folder for processing. However, many solutions need a message ingestion store to act as a buffer for messages, and to support scale-out processing, reliable delivery, and other message queuing semantics. This portion of a streaming architecture is often referred to as stream buffering. Options include Azure Event Hubs, Azure IoT Hub, and Kafka.

Real-time message ingestion:

If the solution includes real-time sources, the architecture must include a way to capture and store real-time messages for stream processing. This might be a simple data store, where incoming messages are dropped into a folder for processing. However, many solutions need a

message ingestion store to act as a buffer for messages, and to support scale-out processing, reliable delivery, and other message queuing semantics. This portion of a streaming architecture is often referred to as stream buffering. Options include Azure Event Hubs, Azure IoT Hub, and Kafka.

Stream Processing:

After capturing real-time messages, the solution must process them by filtering, aggregating, and otherwise preparing the data for analysis. The processed stream data is then written to an output sink. Azure Stream Analytics provides a managed stream processing service based on perpetually running SQL queries that operate on unbounded streams. You can also use open source Apache streaming technologies like Storm and Spark Streaming in an HDInsight cluster.

Analytical Data Store:

Many big data solutions prepare data for analysis and then serve the processed data in a structured format that can be queried using analytical tools. The analytical data store used to serve these queries can be a Kimball-style relational data warehouse, as seen in most traditional business intelligence (BI) solutions. Alternatively, the data could be presented through a low-latency NoSQL technology such as HBase, or an interactive Hive database that provides a metadata abstraction over data files in the distributed data store. Azure Synapse Analytics provides a managed service for large-scale, cloud-based data warehousing.HDInsight supports Interactive Hive, HBase, and Spark SQL, which can also be used to serve data for analysis.

Analysis and Reporting:

The goal of most big data solutions is to provide insights into the data through analysis and reporting. To empower users to analyze the data, the architecture may include a data modeling layer, such as a multidimensional OLAP cube or tabular data model in Azure Analysis Services. It might also support self-service BI, using the modeling and visualization technologies in Microsoft Power BI or Microsoft Excel. Analysis and reporting can also take the form of interactive data exploration by data scientists or data analysts. For these scenarios, many Azure services support analytical notebooks, such as Jupyter, enabling these users to leverage their existing skills with Python or R. For large-scale data exploration, you can use Microsoft R Server, either standalone or with Spark.

Orchestration:

Most big data solutions consist of repeated data processing operations, encapsulated in workflows, that transform source data, move data between multiple sources and sinks, load the processed data into an analytical data store, or push the results straight to a report or dashboard. To automate these workflows, you can use an orchestration technology such Azure Data Factory or Apache Oozie and Sqoop

BIG DATA ANALYTICS

Big Data Analytics are the natural result of four major global trends

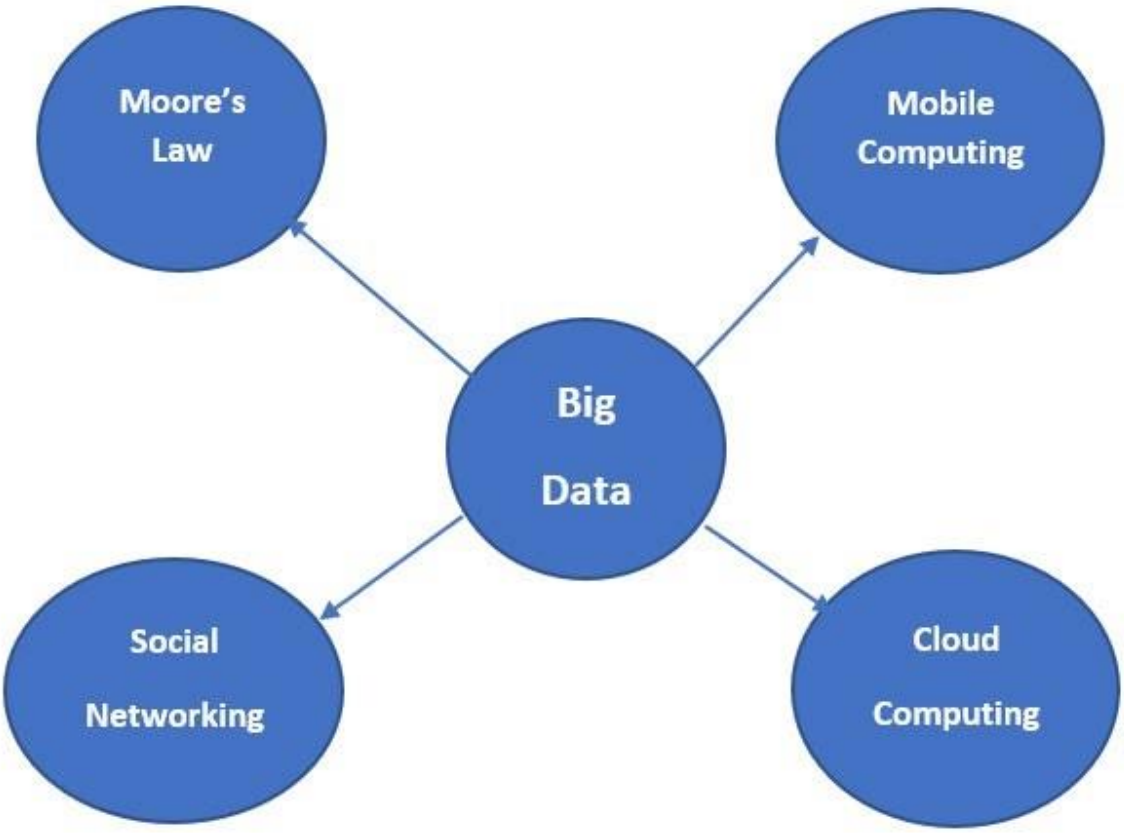


Fig 1.9 Four Major Global Trends

Moore's Law – Basically says that technology always gets cheaper

Mobile Computing – Smart Phone or Mobile Phone in your hand

Social Networking – Facebook, Foursquare, Pinterest (American Image sharing social media service), etc.

Cloud Computing – You don't have to buy hardware or software. Just rent or lease it.

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.

APPLICATIONS – Where it is used

1. Life Sciences:

Clinical research is a slow and expensive process, with trials failing for a variety of reasons. Advanced analytics, artificial intelligence (AI) and the Internet of Medical Things (IoMT) unlocks the potential of improving speed and efficiency at every stage of clinical research by delivering more intelligent, automated solutions.

2. Banking:

Financial institutions gather and access analytical insight from large volumes of unstructured data in order to make sound financial decisions. Big data analytics allows them to access the information they need when they need it, by eliminating overlapping, redundant tools and systems.

3. Manufacturing:

For manufacturers, solving problems is nothing new. They wrestle with difficult problems on a daily basis - from complex supply chains, to motion applications, to labor constraints and equipment breakdowns. That's why big data analytics is essential in the manufacturing industry, as it has allowed competitive organizations to discover new cost saving opportunities and revenue opportunities.

4. Health Care:

Big data is a given in the health care industry. Patient records, health plans, insurance information and other types of information can be difficult to manage – but are full of key insights once analytics are applied. That's why big data analytics technology is so important to heath care. By analyzing large amounts of information – both structured and unstructured – quickly, health care providers can provide lifesaving diagnoses or treatment options almost immediately.

5. Government:

Certain government agencies face a big challenge: tighten the budget without compromising quality or productivity. This is particularly troublesome with law enforcement agencies, which are struggling to keep crime rates down with relatively scarce resources. And that's why many agencies use big data analytics; the technology streamlines operations while giving the agency a more holistic view of criminal activity.

6. Retail:

Customer service has evolved in the past several years, as savvier shoppers expect retailers to understand exactly what they need, when they need it. Big data analytics technology helps retailers meet those demands. Armed with endless amounts of data from customer loyalty programs, buying habits and other sources, retailers not only have an in-depth understanding of their customers, they can also predict trends, recommend new products – and boost profitability.

Big Data Enterprise Model

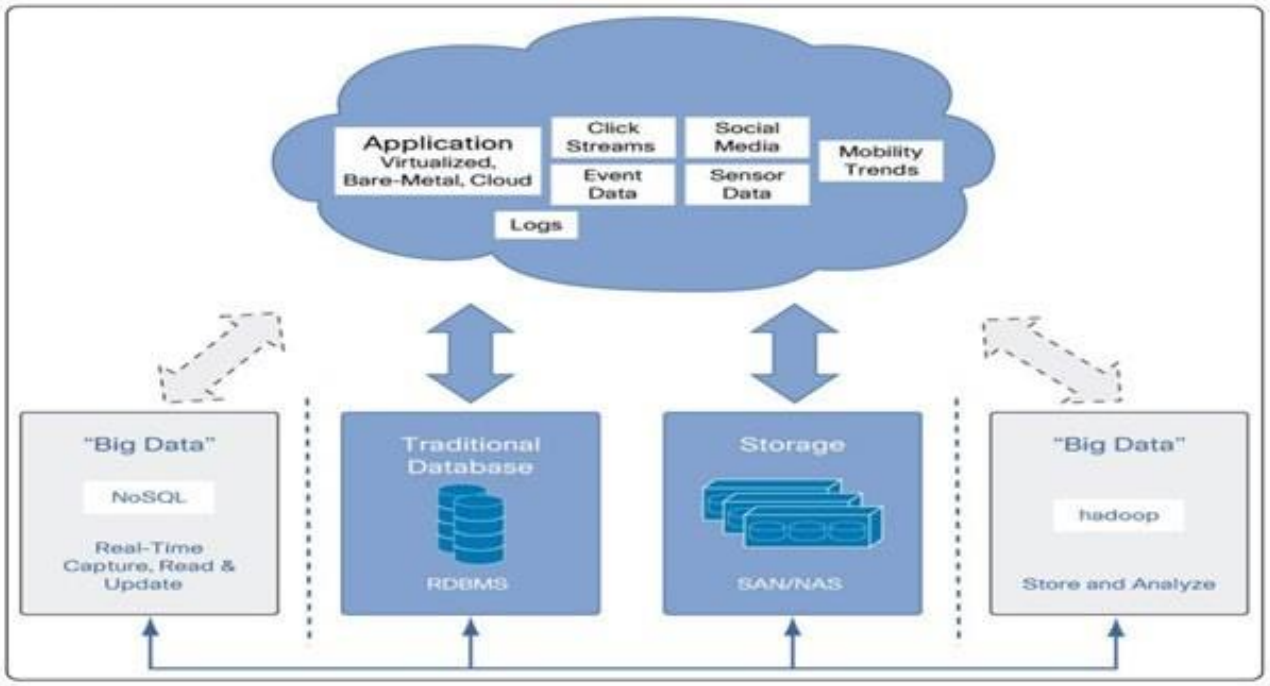


Fig. 1.10 Big Data Enterprise Model

The requirements of traditional enterprise data models for application, database, and storage resources have grown over the years, and the cost and complexity of these models has increased along the way to meet the needs of big data. This rapid change has prompted changes in the fundamental models that describe the way that big data is stored, analyzed, and accessed.

The new models are based on a scaled-out, shared-nothing architecture, bringing new challenges to enterprises to decide what technologies to use, where to use them, and how. One size no longer fits all, and the traditional model is now being expanded to incorporate new building blocks that address the challenges of big data with new information processing frameworks purpose-built to meet big data's requirements. However, these purpose-built systems also must meet the inherent requirement for integration into current business models, data strategies, and network infrastructures.

Big Data Components

Two main building blocks are being added to the enterprise stack to accommodate big data:

- Hadoop: Provides storage capability through a distributed, shared-nothing file system, and analysis capability through MapReduce

- NoSQL: Provides the capability to capture, read, and update, in real time, the large influx of unstructured data and data without schemas; examples include click streams, social media, log files, event data, mobility trends, and sensor and machine data

Building a Big Data Platform

Big Data Platform - Hadoop System:

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

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

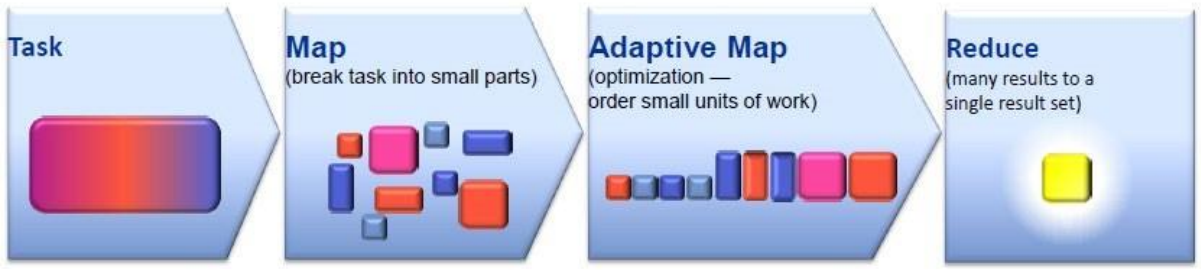


Fig. 1.11 Workload Optimization

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

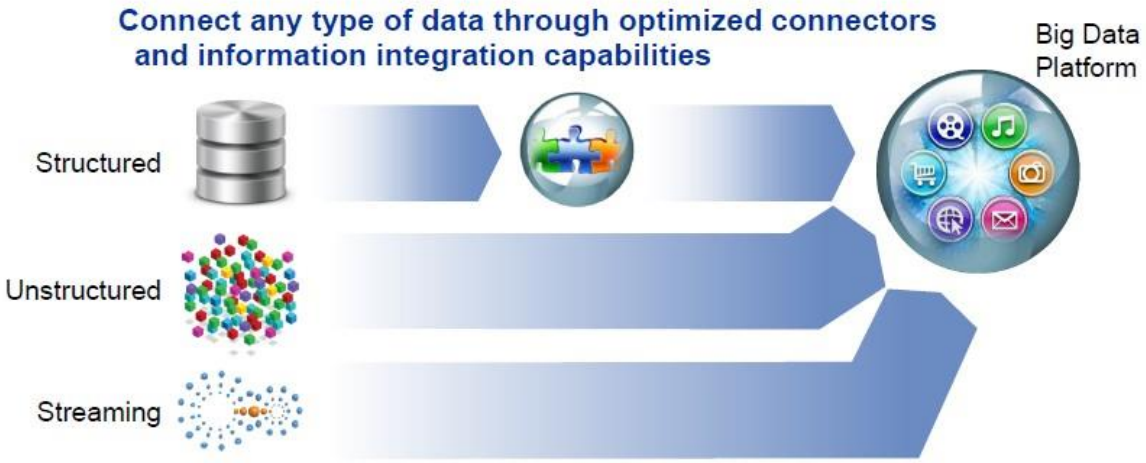


Fig 1.12 Connects different types of data

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

Fig 1.13 Big Data Platform

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:

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

Big Data Analytics for Social and Behavioral Sciences

What Social &Behavioral Sciences Tell Us?

•Social science networks have widespread application in various fields

•Most of the analyses techniques have come from Sociology, Statistics and Mathematics

•For a comprehensive introduction to social network analysis

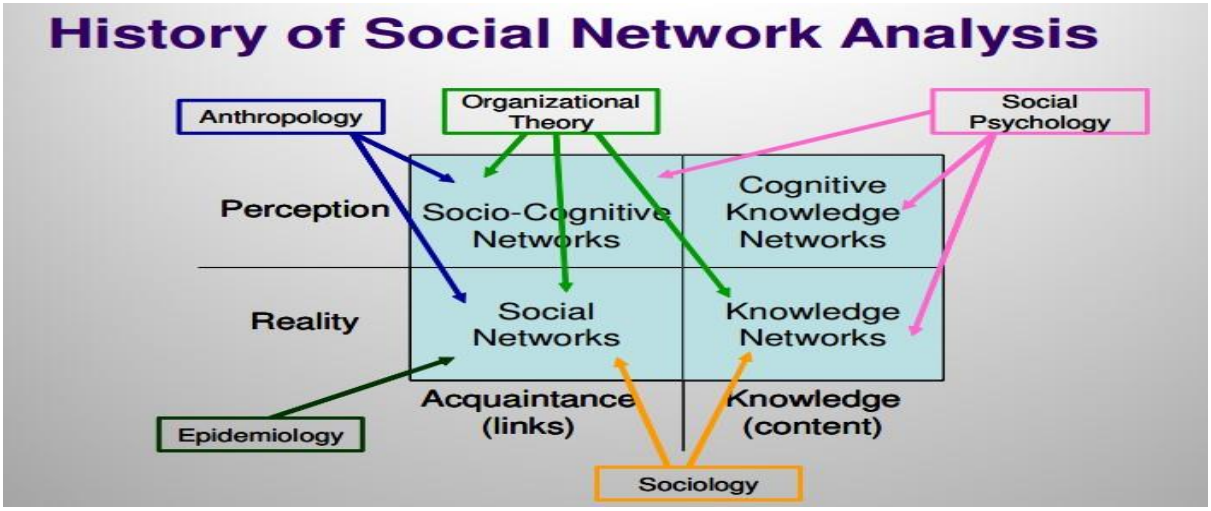


Fig 1.14 History of Social Network Analysis

Why do we create and sustain networks?

•Theories of self-interest

•Theories of social and resource exchange

•Theories of mutual interest and collective action

•Theories of contagion

•Theories of balance

•Theories of homophily

•Theories of proximity

•Theories of co-evolution

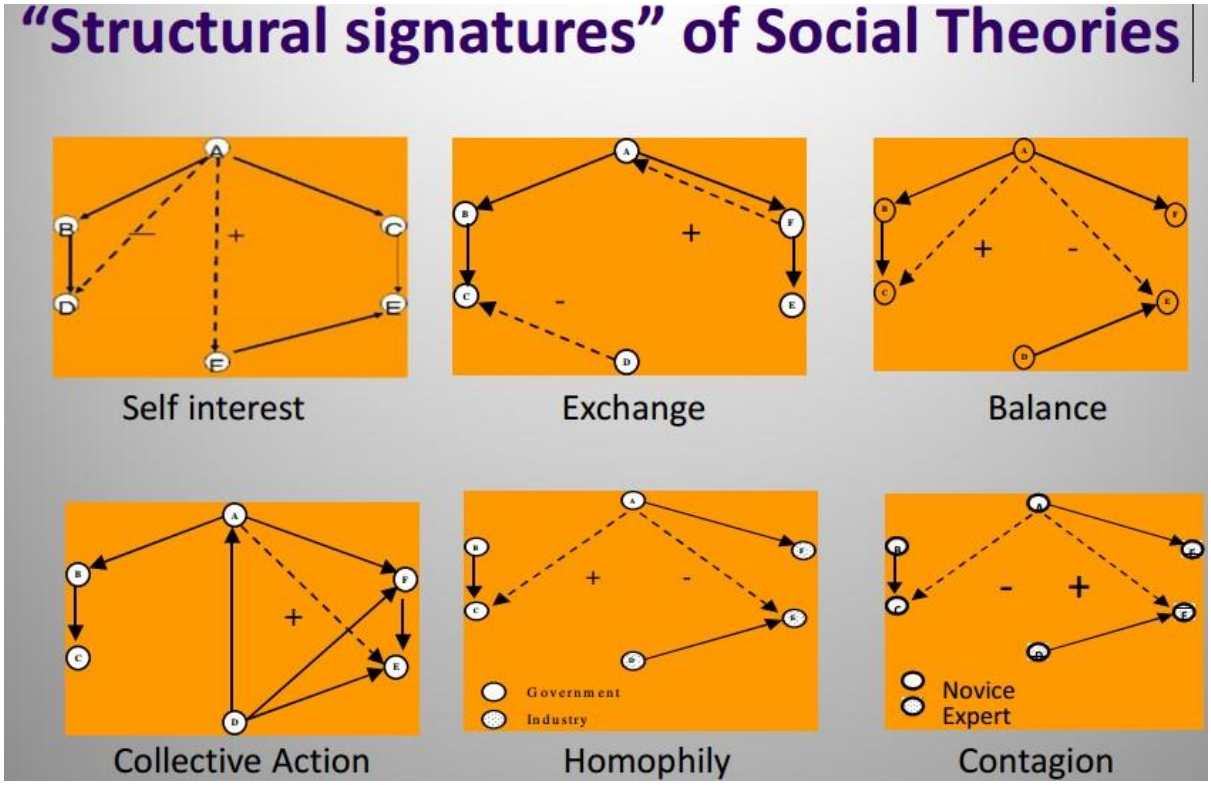


Fig 1.15 "Structural Signatures" of Social Theories

Application Successes

- Numerous in Social Sciences
- Google – PageRank
- LinkedIn – expanding your Cognitive Social Network
  - Making you aware that you're more connected and closer than you think you are
- Expertise discovery in organizations
  - experts, 'authorities'
  - connected individuals, 'hubs'
- Knowledge Well-Rapid-Response teams in emergency management
- Information flow in organizations

- Twitter – real time information dissemination Etc.

UNIT _ II _ SIT1606 _ BIG DATA

Syllabus

Hadoop and Databases - Typical Datacenter Architecture - Adding Hadoop to the Mix - Key Benefit ·Flexibility: Complex Data Processing - HDFS - Hadoop Infrastructure - Architecture - Different in Data Model and Computing Model - HDFS Files and Blocks, Components of HDFS - Hadoop framework - HDFS - Map Reduce Framework - Data Loading techniques - Hadoop Cluster Architecture - Hadoop Configuration files - Hadoop Cluster modes - Single Node - Multi Node - Fully distributed node.

UNIT-II

HDFS, HADOOP AND HADOOP INFRASTRUCTURE

HDFS, Hadoop and Hadoop Infrastructure

Hadoop

- Big Data Technology
- Distributed processing of large data sets
- Open source
- Map Reduce- Simple Programming model **Why Hadoop?**
- Handles any data type
  - Structured/unstructured
  - Schema/no Schema
  - High volume/Low volume
  - All kinds of analytic applications
- Grows with business
- Proven with Petabyte scale
- Capacity & Performance grows
- Leverages commodity hardware to mitigate costs **Hadoop Features**
- 100% Apache open source
- No Vendor locking
- Rich Eco system & community development
- To Derive compute value of all data
- More affordable cost-effective platform

Hadoop and Databases

RDMS (Relational Database Management System): RDBMS is an information management system, which is based on a data model. In RDBMS tables are used for information storage. Each row of the table represents a record and column represents an attribute of data. Organization of data and their manipulation processes are different in RDBMS from other databases. RDBMS ensures ACID (atomicity, consistency, integrity, durability) properties required for designing a database. The purpose of RDBMS is to store, manage, and retrieve data as quickly and reliably as possible.

Hadoop: It is an open-source software framework used for storing data and running applications on a group of commodity hardware. It has large storage capacity and high processing power. It can manage multiple concurrent processes at the same time. It is used in predictive analysis, data mining and machine learning. It can handle both structured and unstructured form of data. It is more flexible in storing, processing, and managing data than traditional RDBMS. Unlike traditional systems, Hadoop enables multiple analytical processes on the same data at the same time. It supports scalability very flexibly.

HDFS, Hadoop and Hadoop Infrastructure

Below in Table 2.1 of differences between Data Science and Data Visualization: **Table 2.1 RDBMS Vs Hadoop**

S.NO.	RDBMS	HADOOP
1.	Structured database approach	Structured and Unstructured database approach
2.	Traditional row-column based databases, basically used for data storage, manipulation and retrieval.	An open-source software used for storing data and running applications or processes concurrently.
3.	It is best suited for OLTP environment.	It is best suited for BIG data.
4.	Interactive OLAP analytics	Scalability of storage/Compute
5.	Multistep ACID transactions	Complex data processing
6.	Stored in the form of tables	Distributed file system
7.	It is less scalable than Hadoop.	It is highly scalable.
8.	SQL – Update and Access data	MapReduce Programming model
9.	100% SQL compliant	Both SQL & NoSQL
10.	Data normalization is required in RDBMS.	Data normalization is not required in Hadoop.
11.	It stores transformed and aggregated data.	It stores huge volume of data.
12.	It has no latency in response.	It has some latency in response.
13.	The data schema of RDBMS is static type.	The data schema of Hadoop is dynamic type.
14.	High data integrity available.	Low data integrity available than RDBMS.

15.	Cost is applicable for licensed software.	Free of cost, as it is an open source software.

Typical Data Centre Architecture

- Schema _on_write (RDBMS)
  - Schema must be created before any data can be loaded
  - An explicit load operation has to take place which transforms data into DB internal structure.
  - New columns must be added explicitly and loaded into the database. (Figure 2.1)

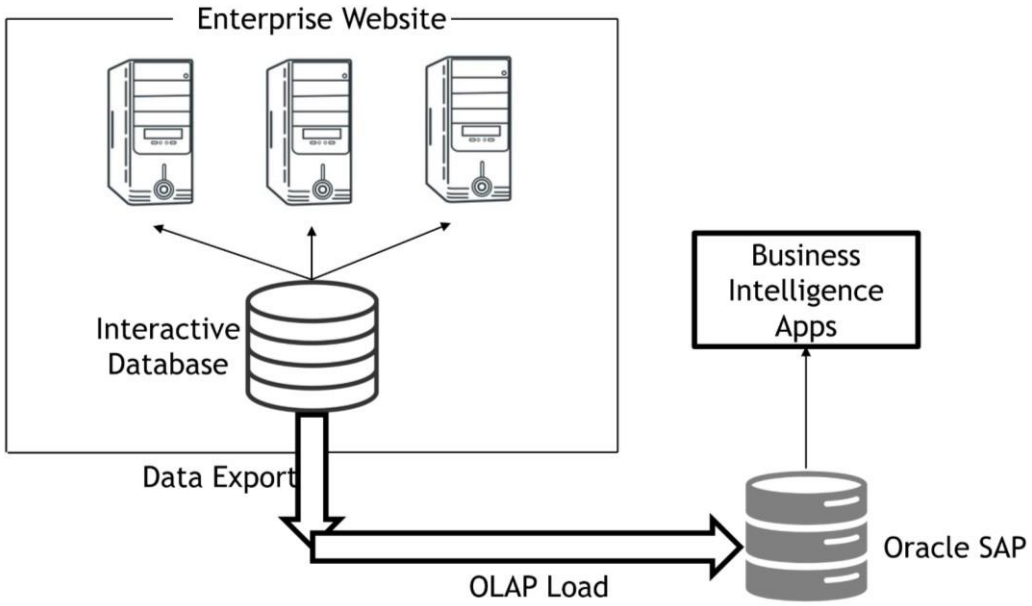


Fig. 2.1 Traditional Data Centre Architecture

Adding Hadoop to the Mix

- Schema _on_write (Hadoop)
  - Data is simply copied to the storage, no transformation is needed
  - A serializer / Deserilalizer is applied during read time to extract the required columns.
  - New data can start flowing at any time and will appear. (Figure 2.2)

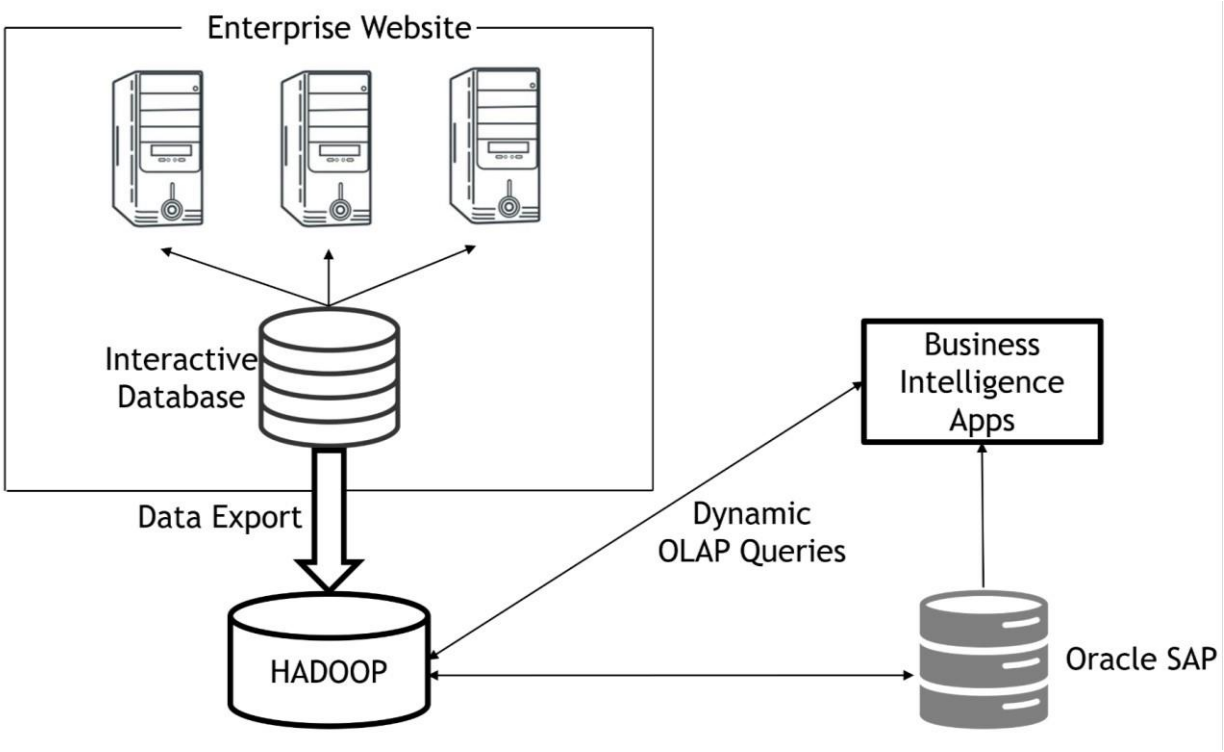


Fig. 2.2 Adding Hadoop Mix to the Traditional Data Centre Architecture

Hadoop Key Benefits

1. Scalable

2. Cost effective

3. Flexible

4. Fast

5. Resilient to failure

1. Scalable

Hadoop is a highly scalable storage platform, because it can store and distribute very large data sets across hundreds of inexpensive servers that operate in parallel. Unlike traditional relational database systems (RDBMS) that can't scale to process large amounts of data, Hadoop enables businesses to run applications on thousands of nodes involving thousands of terabytes of data.

2. Cost effective

Hadoop also offers a cost-effective storage solution for businesses' exploding data sets. The problem with traditional relational database management systems is that it is extremely cost prohibitive to scale to such a degree in order to process such massive volumes of data. In an effort to reduce costs, many companies in the past would have had to down-sample data and classify it based on certain assumptions as to which data was the most valuable. The raw data would be deleted, as it would be too costprohibitive to keep. While this approach may have worked in the short term, this meant that when business priorities changed,

the complete raw data set was not available, as it was too expensive to store. Hadoop, on the other hand, is designed as a scale-out architecture that can affordably store all of a company's data for later use. The cost savings are staggering: instead of costing thousands to tens of thousands of pounds per terabyte, Hadoop offers computing and storage capabilities for hundreds of pounds per terabyte.

3. Flexible

Hadoop enables businesses to easily access new data sources and tap into different types of data (both structured and unstructured) to generate value from that data. This means businesses can use Hadoop to derive valuable business insights from data sources such as social media, email conversations or clickstream data. In addition, Hadoop can be used for a wide variety of purposes, such as log processing, recommendation systems, data warehousing, market campaign analysis and fraud detection.

4. Fast

Hadoop's unique storage method is based on a distributed file system that basically 'maps' data wherever it is located on a cluster. The tools for data processing are often on the same servers where the data is located, resulting in much faster data processing. If you're dealing with large volumes of unstructured data, Hadoop is able to efficiently process terabytes of data in just minutes, and petabytes in hours.

5. Resilient to failure

A key advantage of using Hadoop is its fault tolerance. When data is sent to an individual node, that data is also replicated to other nodes in the cluster, which means that in the event of failure, there is another copy available for use.

The MapR distribution goes beyond that by eliminating the NameNode and replacing it with a distributed No NameNode architecture that provides true high availability. Our architecture provides protection from both single and multiple failures.

When it comes to handling large data sets in a safe and cost-effective manner, Hadoop has the advantage over relational database management systems, and its value for any size business will continue to increase as unstructured data continues to grow.

Flexibility: Complex Data processing

- Java Map Reduce
- Streaming Map Reduce
- Crunch
- Pig Latin
- Hive
- Oozie

- **Java MapReduce**

- Most Flexible and well performed, but tedious development cycles ⦿ **Streaming MapReduce**
- Develop using any programming language of your choice, but slightly lower performance and less flexibility than native Java MapReduce.

⦿ **Crunch**

- A library for multi stage MapReduce pipelines in Java ⦿ **Pig Latin**
- A High-level language suitable for batch data flow workloads developed by yahoo

⦿ **Hive**

- Hive is a data warehouse infrastructure tool to process structured data in Hadoop.
- Hive is an SQL Based tool that builds over Hadoop to process the data. ⦿ **Oozie**
- Oozie is a workflow scheduler system to manage Apache Hadoop jobs. **Hadoop Infrastructure**

The 2 infrastructure models of Hadoop are:

⦿ **Data Model**

⦿ **Computing Model**

Traditional Database Model Vs Hadoop Data Model

Table 2.2 Traditional Database Model Vs Hadoop Data Model

DISTRIBUTED DATABASE MODEL	HADOOP DATA MODEL
Deals with tables and relations	Deals with flat files in any format
Must have a schema for data	No schema
Data fragmentation and partitioning	Files are divided automatically into blocks

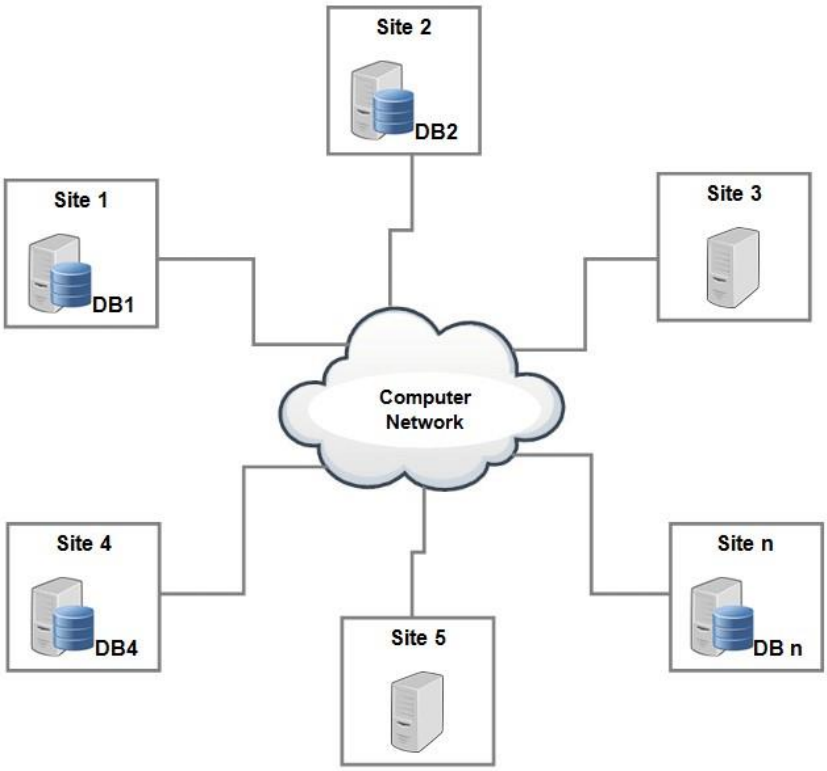


Fig. 2.3 Distributed Database Model

A distributed database consists of a network of many interconnected physical databases that spread across various geographical locations. The separate databases are periodically synchronized for ensuring all have consistent data. (Figure 2.3)

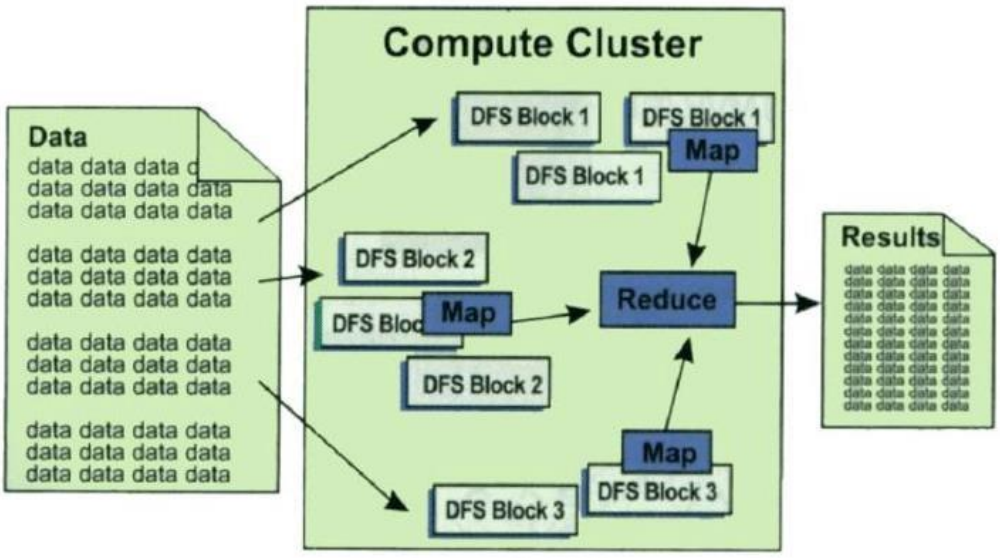


Fig. 2.4 Hadoop Data Model

A Hadoop data model consists of a Blocks of Distributed File Systems under a specific Compute Cluster, where the DFS blocks are interconnected with communication channel. This makes sure that the centre is in one place for the data sent to be mapped and reduced. (Figure 2.4)

Traditional Vs Computing Model

Table 2.3 Traditional Vs Computing Model

DISTRIBUTED DATABASES	HADOOP
Notion of a transaction	Notion of a job divided into tasks
Distributed transaction with ACID properties	MapReduce computing model where every task is either a map or reduce service

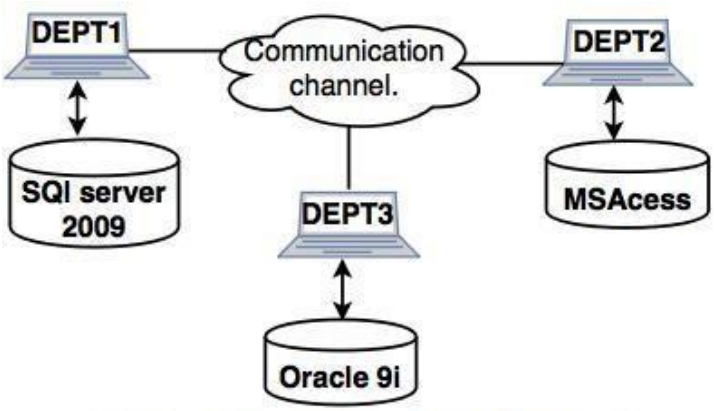


Fig. 2.5 Distributer Databases

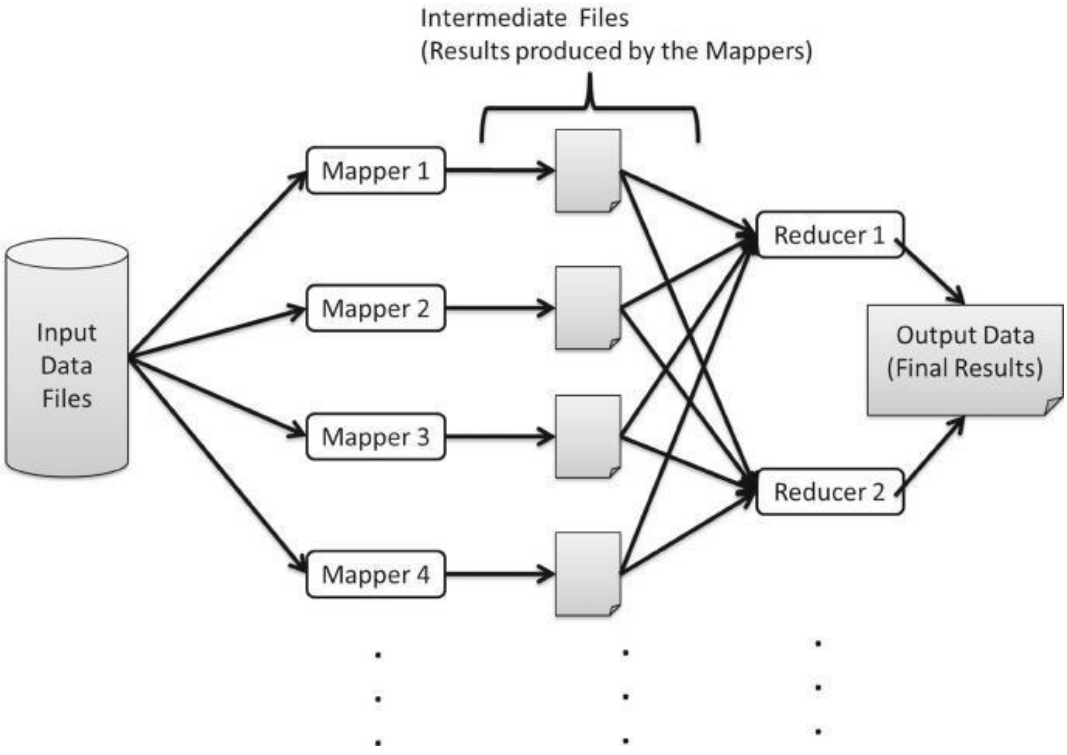


Fig. 2.6 Hadoop – Computing Model

Hadoop Architecture

What is Architecture of Hadoop?

- Hadoop is the open-source framework of Apache Software Foundation, which is used to store and process large unstructured datasets in the distributed environment.
- Data is first distributed among different available clusters then it is processed.
- Hadoop biggest strength is that it is scalable in nature means it can work on a single node to thousands of nodes without any problem.
- Hadoop framework is based on Java programming and it runs applications with the help of MapReduce which is used to perform parallel processing and achieve the entire statistical analysis on large datasets.
- Distribution of large datasets to different clusters is done on the basis of Apache Hadoop software library using easy programming models.
- Organizations are now adopting Hadoop for the purpose of reducing the cost of data storage.
- It will lead to the analytics at an economical cost which will maximize the business profitability.
- For a good Hadoop architectural design, you need to take in considerations – good computing power, storage, and networking. **Hadoop Architecture**
- Hadoop ecosystem consists of various components such as Hadoop Distributed File System (HDFS), Hadoop MapReduce, Hadoop Common, HBase, YARN, Pig, Hive, and others. ⦿ Hadoop components which play a vital role in its architecture are-
    - Hadoop Distributed File System (HDFS)
    - Hadoop MapReduce
- Hadoop works on the master/slave architecture for distributed storage and distributed computation.
- NameNode is the master and the DataNodes are the slaves in the distributed storage.
- The Job Tracker is the master and the Task Trackers are the slaves in the distributed computation.
- The slave nodes are those which store the data and perform the complex computations.
- Every slave node comes with a Task Tracker daemon and a DataNode synchronizes the processes with the Job Tracker and NameNode respectively.
- In Hadoop architectural setup, the master and slave systems can be implemented in the cloud or on-site premise. (Figure 2.7, 2.8)

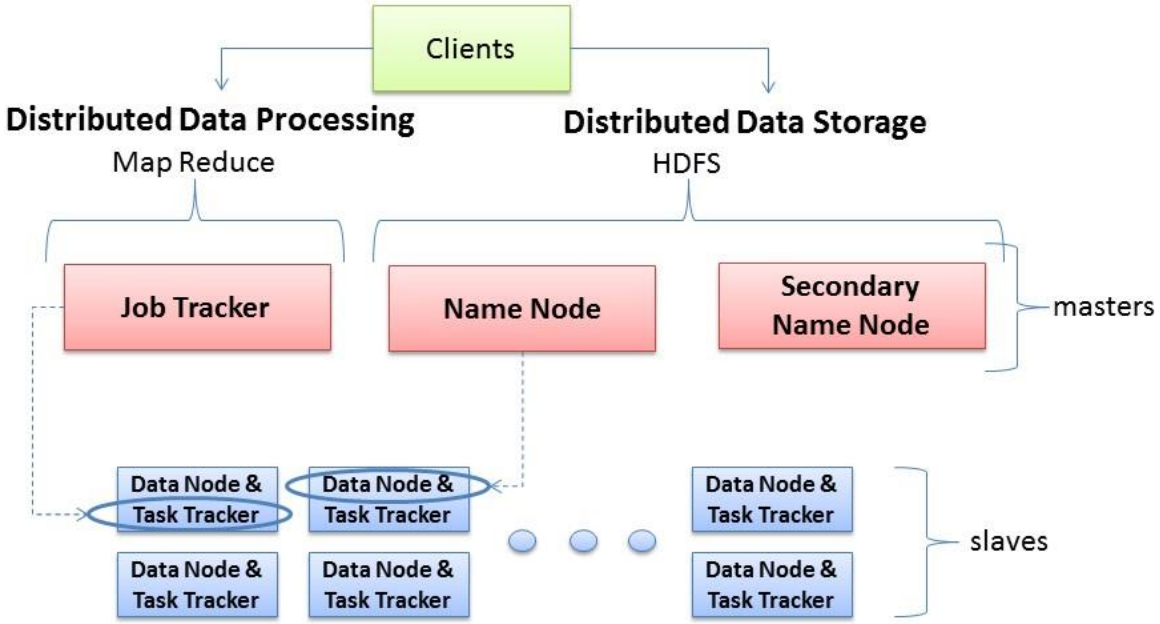


Fig. 2.7 Hadoop Architecture

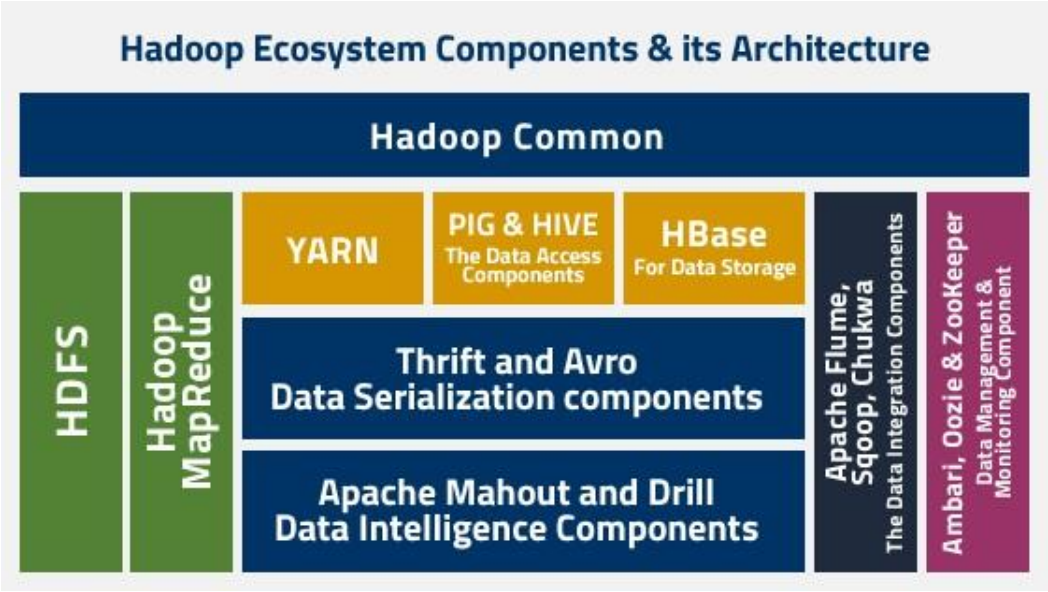


Fig. 2.8 Hadoop Ecosystem Components & its Architecture

Hadoop Framework

There are 2 Main layers:

- HDFS
- Execution Engineer

HDFS

Role of HDFS in Hadoop Architecture

-
-

  HDFS is used to split files into multiple blocks. Each file is replicated when it is stored in Hadoop cluster.

  The default size of that block of data is 64 MB but it can be extended up to 256 MB as per the requirement.

- HDFS stores the application data and the file system metadata on two different servers.
- NameNode is used to store the file system metadata while and application data is stored by the DataNode.
- To ensure the data reliability and availability to the highest point, HDFS replicates the file content many times.
- NameNode and DataNode communicate with each other by using TCP protocols.
- Hadoop architecture performance depends upon Hard-drives throughput and the network speed for the data transfer. (Figure 2.9)

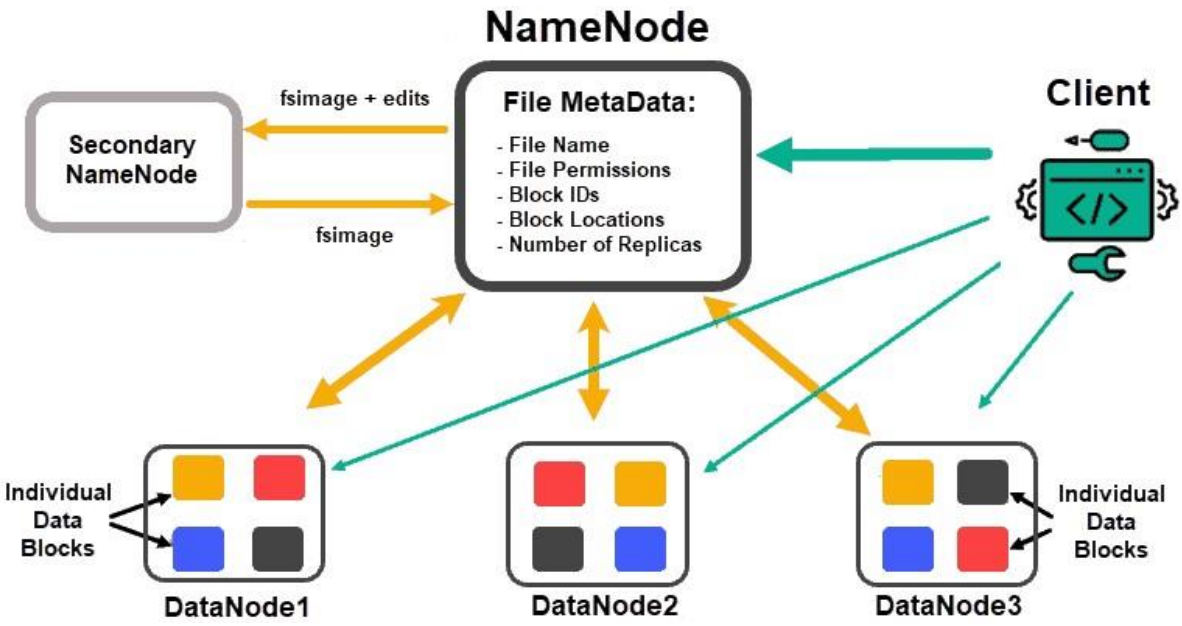


Fig. 2.9 HDFS What

is HDFS in Hadoop

-
-

-
-
- The Hadoop Distributed File System is a java-based file, developed by Apache Software Foundation with the purpose of providing versatile, resilient, and clustered approach to manage files in a Big Data environment using commodity servers.

  HDFS used to store a large amount of data by placing them on multiple machines as there are hundreds and thousands of machines connected together.

  The goal is to store a smaller number of larger files rather than the greater number of small files.

  HDFS provides high reliability of data because it used to replicate data into three different copies- two are saved in one group and third one in another.

  HDFS is scalable in nature as it can be extended to 200 PB of storage where a single cluster contains 4500 servers, supporting billions of blocks and files.

How HDFS Works

- Hadoop works on a master node which is NameNode and multiple slave nodes which are DataNodes on a commodity cluster.
- As all the nodes are present in the same rack in the data center, data is broken into different blocks that are distributed among different nodes for the storage.
- These blocks are replicated across nodes to make data available in case of a failure. (Figure 2.10)

-
-

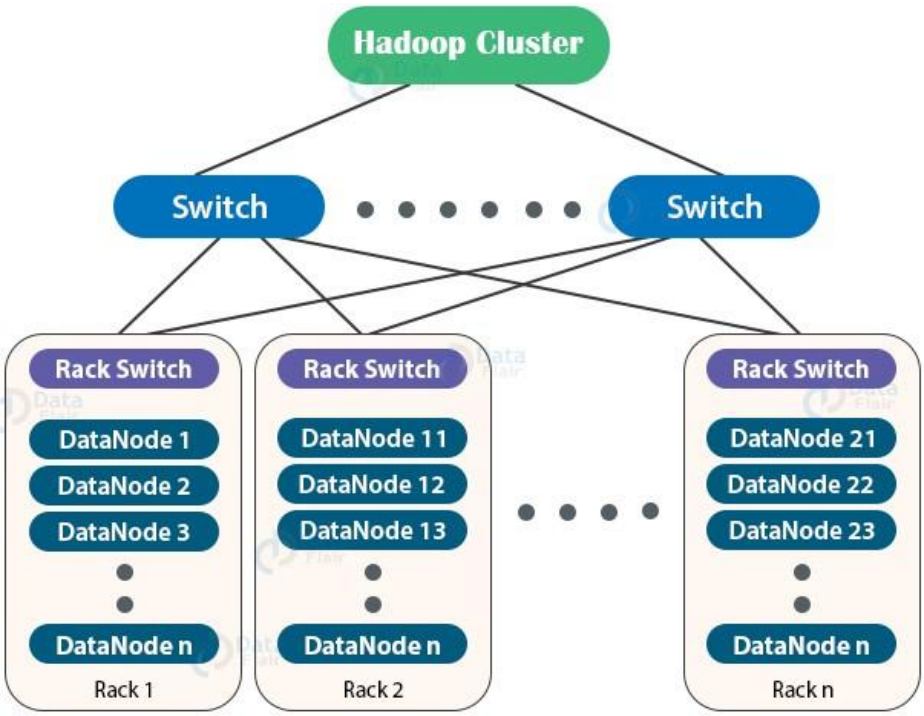


Fig. 2.10 HDFS Working

The NameNode is known as a smart node in the cluster.

- NameNode knows which data node contains which blocks and where data node has been placed in the clusters.

  The NameNode also contains the access authority which is given to files to perform read, write, create, delete and replication of various blocks between the data nodes.

  The NameNode is connected with the data nodes in the loosely coupled fashion.

  This provides the feature of scalability in real time means it can add or delete nodes as per the requirement.

  Data nodes used to communicate every time with the NameNode to check whether the certain task is completed or not.

-
-
- Communication between these two ensures that NameNode knows the status of each data node all the time.
- If one of the data nodes is not functioning properly then NameNode can assign that task to another node.
- To operate normally, data nodes in the same block communicate with each other.
- The NameNode is replicated to overcome system failure and is the most critical part of the whole system.
- Data nodes are not considered to be smart, but flexible in nature and data blocks are replicated across various nodes and the NameNode is used to manage the access.
- To gain the maximum efficiency, replication mechanism is used and all the nodes of the cluster are placed into a rack. Rack Id is used by the NameNode to track data nodes in the cluster.
- A heartbeat message is put through to ensure that the data nodes and the NameNode are still connected.
- When the heartbeat is no longer available, then the NameNode detach that data node from the cluster and works in the normal manner.
- When the heartbeat comes, data node is added back to the cluster.
- Transaction log and the checksum are used to maintain the data integrity.
- Transaction log used to keep track of every operation and help them in auditing and rebuilding the file system, in case of an exception.
- Checksum validates the content in HDFS.
- When a user requests a file, it verifies the checksum of that content.
- If checksum validation matches then they can access it.
- If the checksum reports an error, then the file is hidden to avoid tampering.
- The performance also depends on where the data is stored, so it is stored on local disk in the commodity servers.

-
-

-

-

  To ensure that one server failure doesn't corrupt the whole file, data blocks are replicated in various data nodes.

  The degree of replication and the number of data nodes are managed at a time when the cluster is implemented.

-

-

Features of HDFS

- Fault-Tolerant
- Scalability
- Data Availability
- Data Reliability
- Replication

Description to Features of HDFS

- **Fault-Tolerant**
  - HDFS is highly fault-tolerant.
  - HDFS replicates and stores data in three different locations.
  - So, in the case of corruption or unavailability, data can be accessed from the previous location.
- **Scalability**
  - Scalability means adding or subtracting the cluster from HDFS environment.
  - Scaling is done by the two ways – vertical or horizontal.
  - In vertical scaling, you can add up any number of nodes to the cluster but there is some downtime.
    In horizontal scaling, there is no downtime; you can add any number of nodes in the cluster in real time. ⦿ **Data Availability**
  - Data is replicated and stored on different nodes due to which data is available all the time.
  - In case of network, node or some hardware failure, data is accessible without any trouble from a different source or from a different node. ⦿ **Data Reliability**
  - HDFS provides highly reliable data storage, as data are divided into blocks and each block is replicated in the cluster which made data reliable.
  - If one node contains the data is down, it can be accessed from others because HDFS creates three replicas of each block.
  - When there is no loss of data in case of failure then it is highly reliable.
- **Replication**
  - Data replication is one of the unique and important features of HDFS.
  - HDFS used to create replicas of data in the different cluster.

- As if one node goes down it can be accessed from other because every data block has three replicas created.
- This is why, there is no chance of data loss.

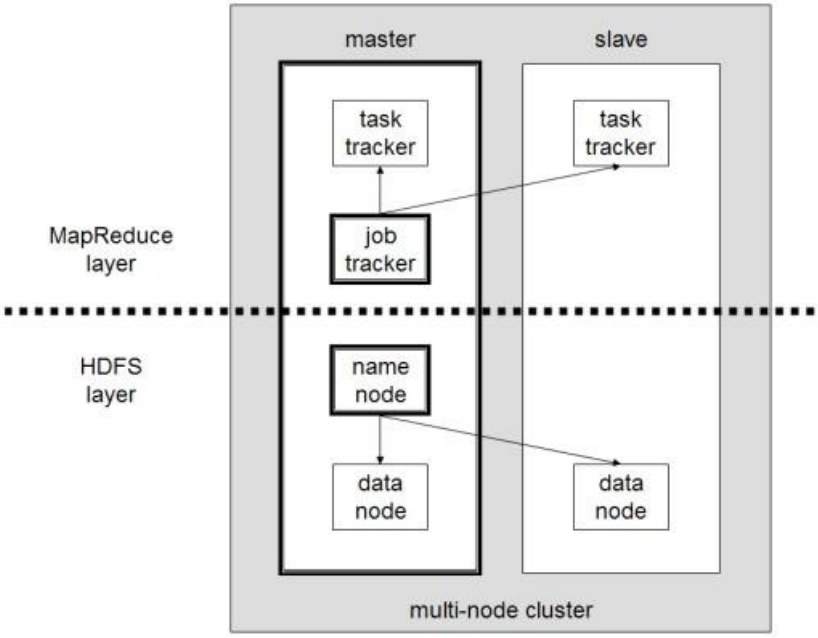


Fig. 2.11 HDFS Layer

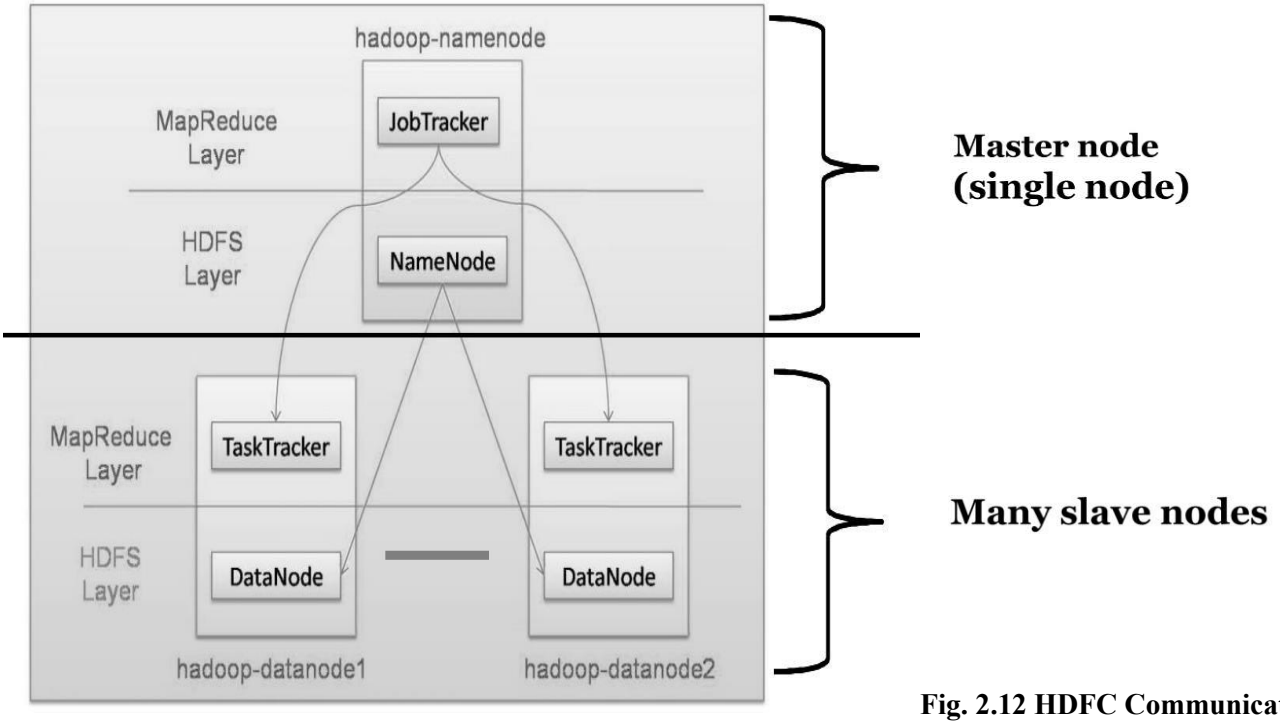


Fig. 2.12 HDFC Communication

Components of HDFS

NameNode

- Single node in cluster

- Brain of HDFS
- Hadoop employs master/slave architecture for both distributed storage and distributed computation
- Distributed Storage system is HDFS
- NameNode is the master of HDFS that directs slave DataNode to perform low level tasks. (Figure 2.13)

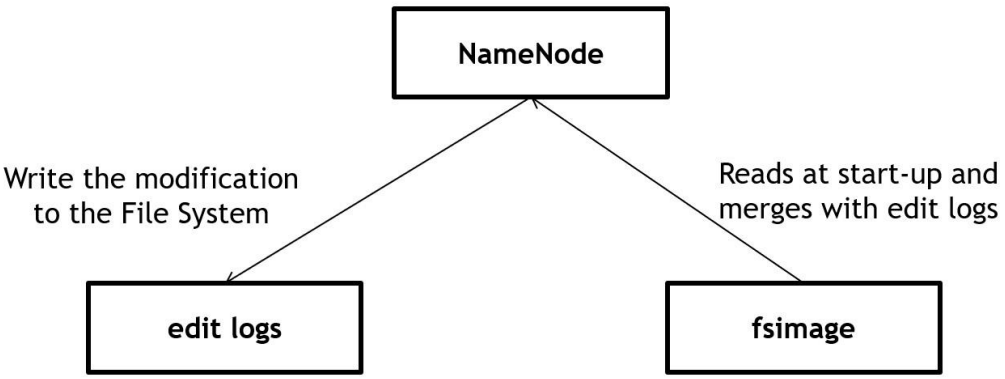


Fig. 2.13 Components of HDFS NameNode

as book Keeper of HDFS

- NameNode list of all blocks of file and list of all data nodes that contains the block
- It keeps track of how files are stored into file blocks, which nodes store these blocks and overall health of HDFS
- The function of NameNode is memory and IO intensive
- The server hosting NameNode doesn't store any user data or perform any computation **Name node – Drawback**
- Name node is the Single point of failure in Hadoop cluster
- For other data nodes, if their host node fails due to h/w or s/w reasons, the Hadoop cluster will continue smoothly

Data Nodes

- Each slave machine in the cluster will host a data node to perform grunt work of Distributed File System
- Reading and writing HDFS blocks to actual files on local file system
- During r/w a HDFS file, the file is broken into blocks and NameNode will tell the client which data node each block resides in.

- Client communicates directly with the DataNodes to process local files corresponding to the blocks
- Data node may communicate with other DataNodes to replicate data blocks for redundancy

Data Replication

- HDFS is designed to reliably store very large files across machines in a large cluster.
- It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size.
- The blocks of a file are replicated for fault tolerance.
- The block size and replication factor are configurable per file.
- An application can specify the number of replicas of a file.
- The replication factor can be specified at file creation time and can be changed later.
- Files in HDFS are write-once and have strictly one writer at any time.
- The NameNode makes all decisions regarding replication of blocks. **Heartbeat and Block report of data nodes**
- NameNode periodically receives a Heartbeat and a Block report from each of the DataNodes in the cluster
- Receipt of a Heartbeat implies that the DataNode is functioning properly
- A Block report contains a list of all blocks on a DataNode (Figure 2.14, 2.15)

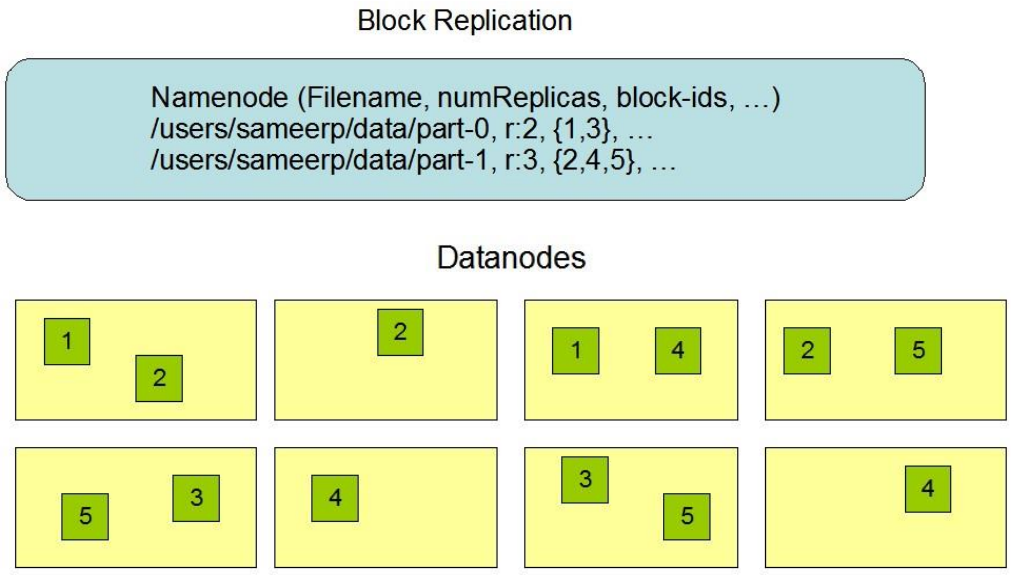


Fig. 2.14 Block Replication

Fig. 2.15 DataNodes

Replication factor

When the replication factor is three, HDFS's placement policy is to put:

1. one replica:
    - on the local machine if the writer is on a DataNode,
    - otherwise on a random DataNode,
2. another replica on a node in a different (remote) rack,
3. last on a different node in the same remote rack.

This policy cuts the inter-rack write traffic which generally improves write performance. The chance of rack failure is far less than that of node failure. **Replication factor >3**

- If the replication factor is greater than 3, the placement of the 4th and following replicas are determined randomly while keeping the number of replicas per rack below the upper limit (which is basically (replicas - 1) / racks + 2)

DataNodes and NameNodes

- DataNode constantly report to NameNode
- Upon initialization, each DataNode informs NameNode of the blocks it is currently storing
- After mapping, the DataNode continually poll the NameNode to provide information regarding local changes and receive instructions to create, move or delete blocks from local disk. **Secondary Name Node (SNN)**
- SNN is an assistant daemon for maintaining the state of clusters HDFS
- Like NameNode, each cluster has on SNN
- SNN does not receive or record any real-time changes to HDFS
- SNN communicates with NameNode to take snapshots of HDFS meta data at intervals defined by cluster configuration
- NameNode is a single point of failure for Hadoop clusters and SNN snapshots help minimize the downtime and loss of data
- NameNode failure requires human intervention to reconfigure the cluster to use SNN as primary NameNode (Figure 2.16)

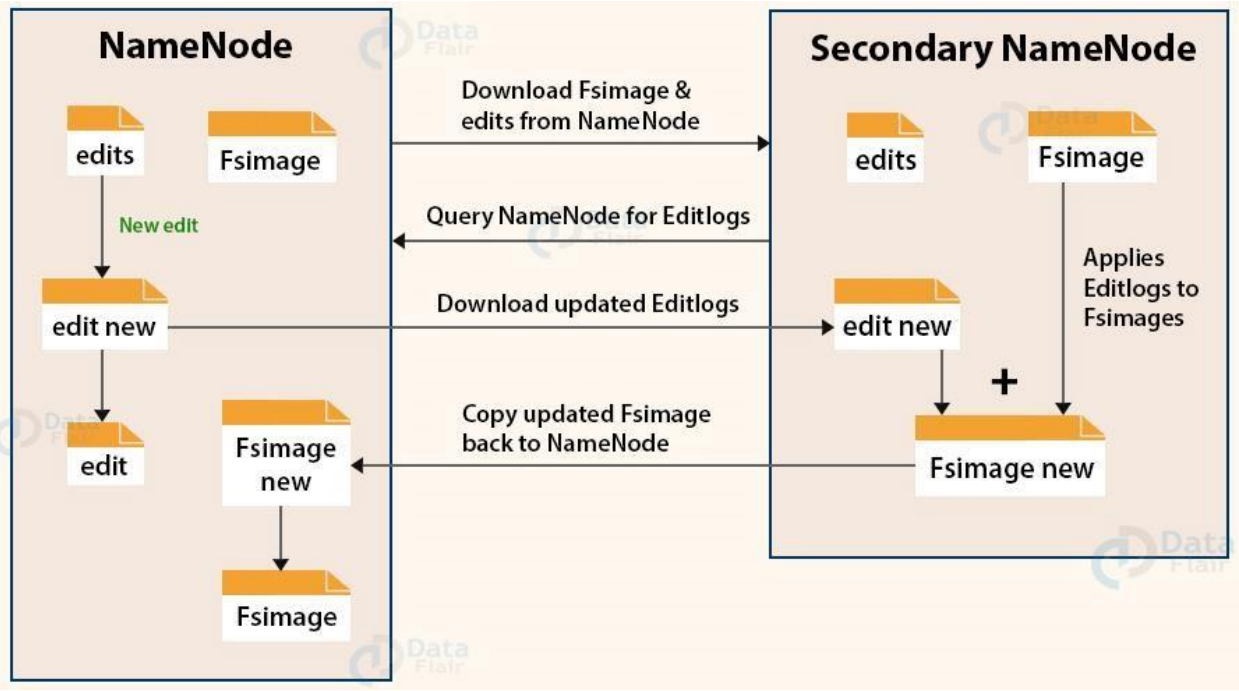


Fig. 2.16 NameNode Interaction Job

Tracker

- Job tracker is a liaison between the client application and Hadoop
- Once the client submits the code to the cluster the JobTracker determines the execution plan by determining which files to process
- Assigns nodes to different tasks and monitors all tasks as they are running
- If a task fails, the job tracker will automatically relaunch the task on a different node **Task Tracker**
- Job tracker is the master overseeing the overall execution of MapReduce job
- Task trackers manages the execution of individual tasks
- Task tracker can spawn multiple JVMs to handle many map/reduce tasks in parallel
- Main responsibility is to constantly communicate with Job tracker
- If Job tracker fails to receive a heartbeat from TaskTracker within the specified amount of time, it will assume task tracker has crashed and will resubmit the corresponding tasks to other nodes in the cluster. (Figure 2.17)

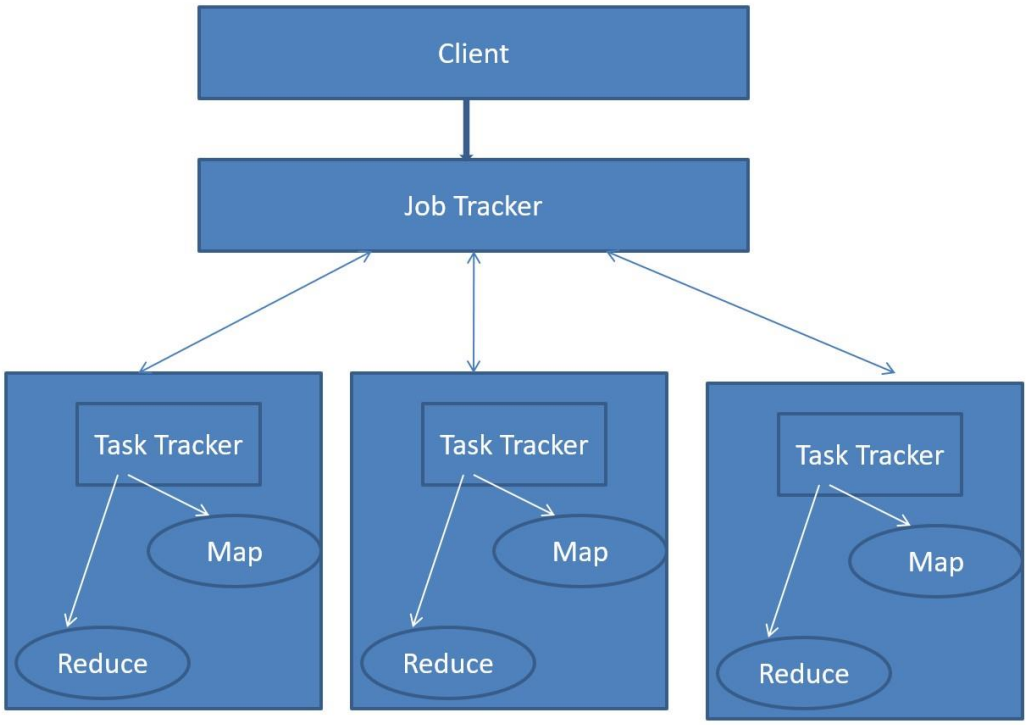


Fig. 2.17 Task Tracker

Topology of Hadoop cluster

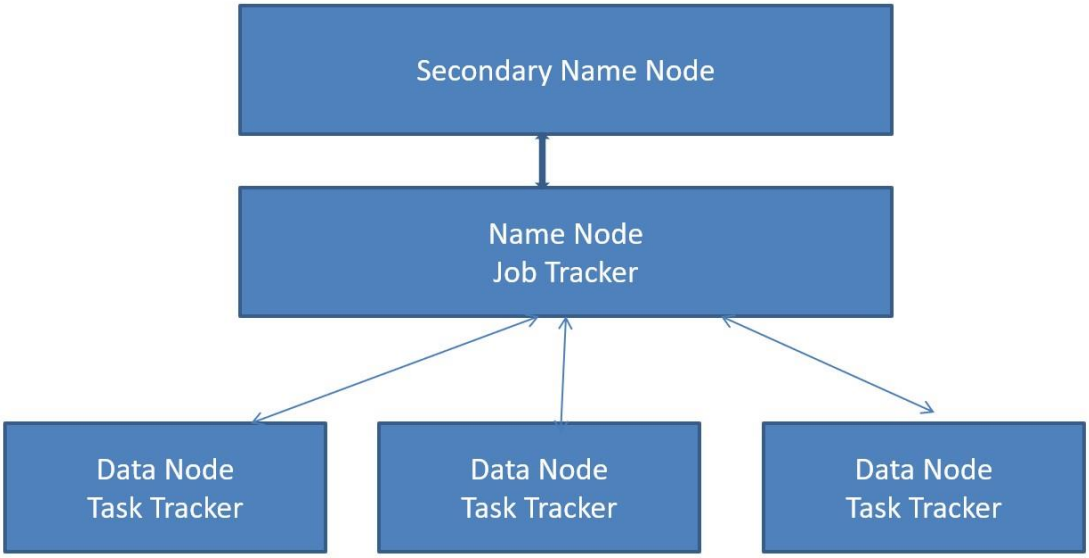


Fig. 2.18 Topology of Hadoop cluster

HDFS

- HDFS is the first building block of Hadoop cluster
- HDFS breaks incoming files into blocks and stores them redundantly across the cluster
- To efficiently process massive amounts of data, it is important to move
- A single large file splits into blocks and blocks are distributed among the nodes of Hadoop cluster
- The blocks used in HDFS are large 128 MB or more compared to small blocks associated with traditional file systems
- This allows the system to scale without increasing the size and complexity of HDFS metadata
- Files in HDFS are write once files
- Input data is loaded into HDFS and processed by MapReduce framework. (Figure 2.18) **HDFS Files and Blocks**
- Files are stored as a collection of Blocks
- Blocks: The minimum amount of data that can be read or written
- Typically, 64MB of unit size is default
- Block details are stored on 3 nodes
    - The Name node – Manages meta-data about files & Blocks
    - SNN – Holds backup of Name node data
    - Data Node – Store and serve blocks

- Multiple copies of a block are stored

Data Loading techniques

HDFS READS

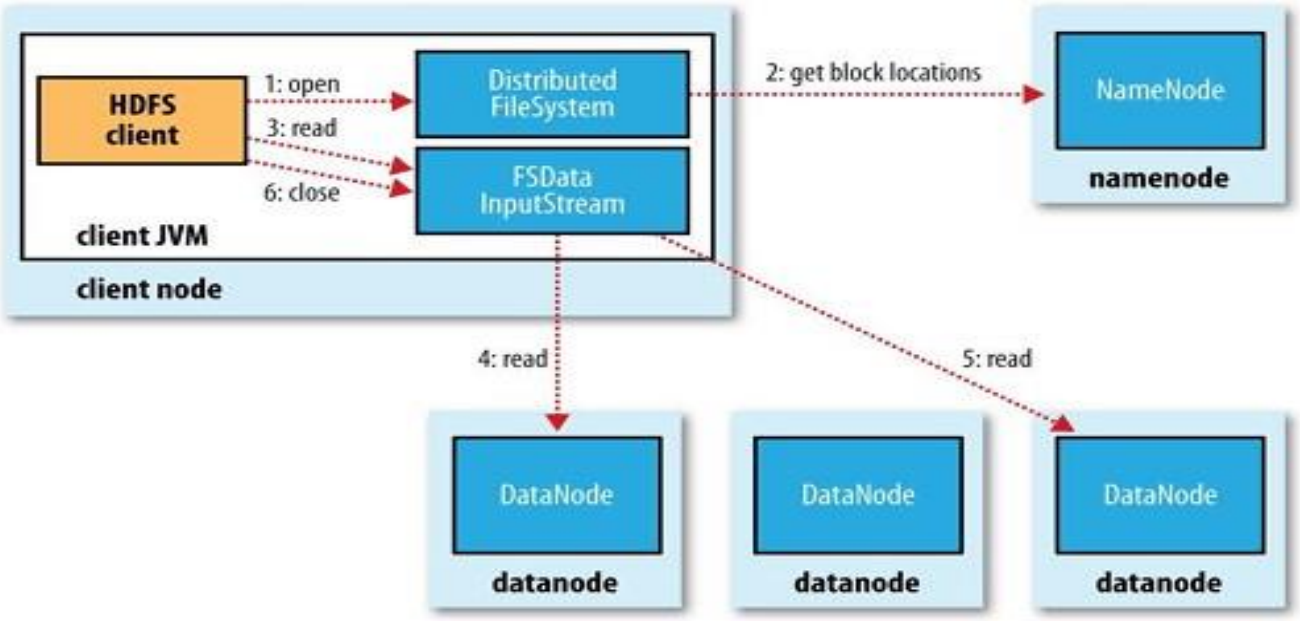


Fig. 2.19 HDFS Reads

HDFS Writes

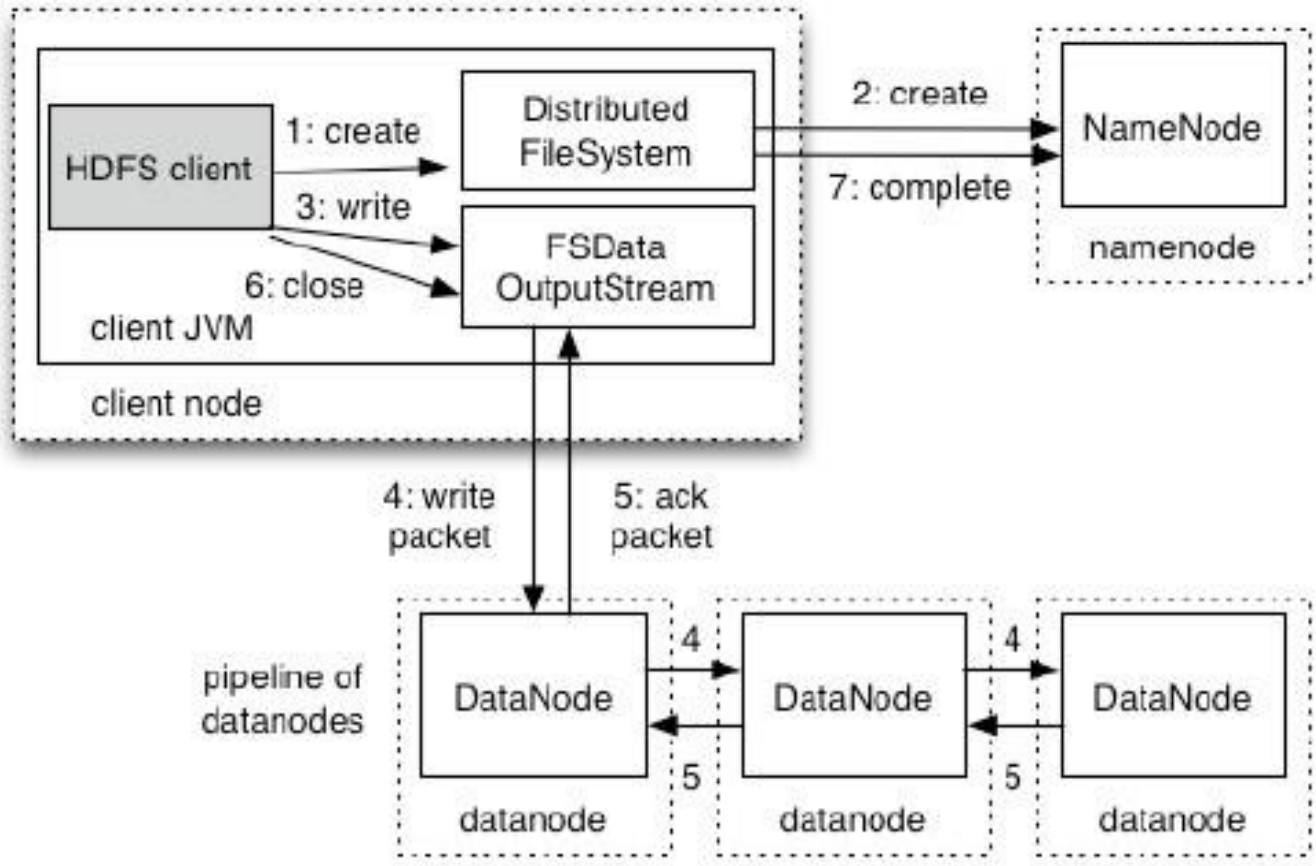


Fig. 2.20 HDFS Write In

conclusion, HDFS empowers Hadoop functionality.

HDFS provides highly reliable data storage despite of any hardware failure.

It is highly fault-tolerant, provides high data availability, and high scalability. (Figure 2.19, 2.20)

MapReduce

Role of MapReduce in Hadoop Architecture

- The job is the top-level unit of MapReduce working and each job contains one or more Map or Reduce tasks.
- The execution of a job starts when it is submitted to the Job Tracker of MapReduce which specifies the map, combines, and reduce functions along with the location of input and output data.
- When the job is received, the job tracker searches the number of splits based on input path and select Task Trackers based on their network locality to the data sources.
- Task Tracker extracts information from the splits as the processing begins in Map phase.

Records are parsed by the "Input Format" and generate key-value pairs in the memory buffer when Map function is provoked.

- Combine function is used to sort all the splits from the memory buffer.
- After the completion of a map task, Task Tracker gives a message to the Job Tracker.
- Job Tracker then gives a message to selected Task Tracker to start the reduce phase.
- Now Task Tracker sorts the key-value pairs for each key after reading it.
- At last reduce function is invoked and all the values are collected into one output file.

MapReduce

- Map reduce is a programming model for expressing distributed computations at a massive scale •

  It is an execution framework for organizing and performing such computation. (Figure 2.21)

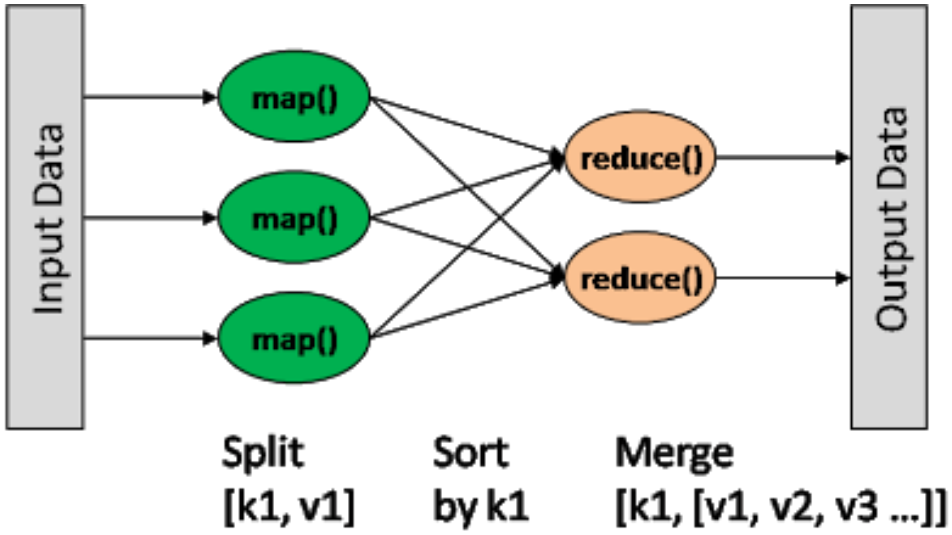


Fig. 2.21 MapReduce Typical

Large Data Problem

- Map
  - Iterate over large number of records
  - Extract data of interest
  - Shuffle and sort intermediate results

 Reduce

  - Aggregate intermediate results
  - Generate final output

⦿

Map Reduce model

- Basic notion of MapReduce is to divide a task into subtasks, handle subtasks in parallel and aggregate the results of sub tasks to form the final output
- Programs written in MapReduce are automatically parallelized and programmers not need to be concerned about the implementation details of parallel processing **MapReduce functions**
- Map()
- Reduce()
- Map Phase:
    - Reads input(in parallel) and distributes data to reducers
    - Auxiliary phase such as sorting partitioning and combining values can also take place between map and reduce phases.
- Map Reduce programs are used to process large files
- Input and Output for map and reduce functions are expressed in the form of (Key, Value) pairs
- Hadoop MapReduce program has a component called Driver
- Driver is responsible for initializing the job with its configuration details, specifying the mapper and reducer classes for the job informing the Hadoop platform to execute the code on the specified input file and controlling the location which the output files are placed.

What is MapReduce in Hadoop?

- The heart of Apache Hadoop is Hadoop MapReduce.
- It's a programming model used for processing large datasets in parallel across hundreds or thousands of Hadoop clusters on commodity hardware.
- The framework does all the work, you just need to put the business logic into the MapReduce.
- All the work is divided into the smaller works and assigned to the slave by a master who is submitted by the user.
- Hadoop MapReduce is designed by using a different approach.
- Different kinds of lists as input is presented to MapReduce for processing and it coverts those lists into output which is also in the form lists.
- Hadoop MapReduce parallel processing has made Hadoop more efficient and powerful.
- Jobs are divided into small parts by MapReduce and each of them resides in parallel on the cluster.

- All the problems are divided into a larger number of small chunks and each of the chunks are processed individually for the output.

  For the final output, these are further processed.
- Hadoop MapReduce can be scaled hundreds to thousands of computers across the clusters.
- Different computers in the clusters used to process the jobs which could not be processed by a large one.
- Map-Reduce is the component of Hadoop and used for processing the data.
- These two transform the lists of input data elements by providing those key-pair values and then back into the lists of output data elements by combining the key pair values.
- A small phase of Shuffle and Sort also come during the Map and Reduce phase in MapReduce.
- Mapper and Reducer execution across a data set is known as MapReduce Job.
- Mapper and Reducer is an execution of two processing layer.
- Input data, the MapReduce program, and configuration information are what a MapReduce Job contains.
- So, if the client wants a MapReduce Job to execute, he needs to provide input data, write a MapReduce program, and provide some configuration information.
- Execution of a chunk of data in Mapper or Reducer phase is known as the Task or Task-InProgress.
- The attempt of any instance to execute a task on a node is known as Task Attempt.
- There's possibility task cannot be executed due to machine failure.
- Then the task is rescheduled another node.
- Rescheduling of the task can only be done for 4 times.
- If any job fails more than the 4 times then it is considered to be a failed job.

Working Process of Map and Reduce

- User-written function at mapper is used for processing of input data in mapper.
- All the business logic is put into the mapper level because it is a place where all the heavy processing is done.
- A number of mappers are always greater than the reducers.
- The output which is produced by the mapper is intermediate data and it is input for the reducer.
- User-written function at reducer is used to process the intermediate data and after this final output is generated.

-
- The output generated in reducer phase is stored in HDFS and replication is done as per usual. (Figure 2.22)

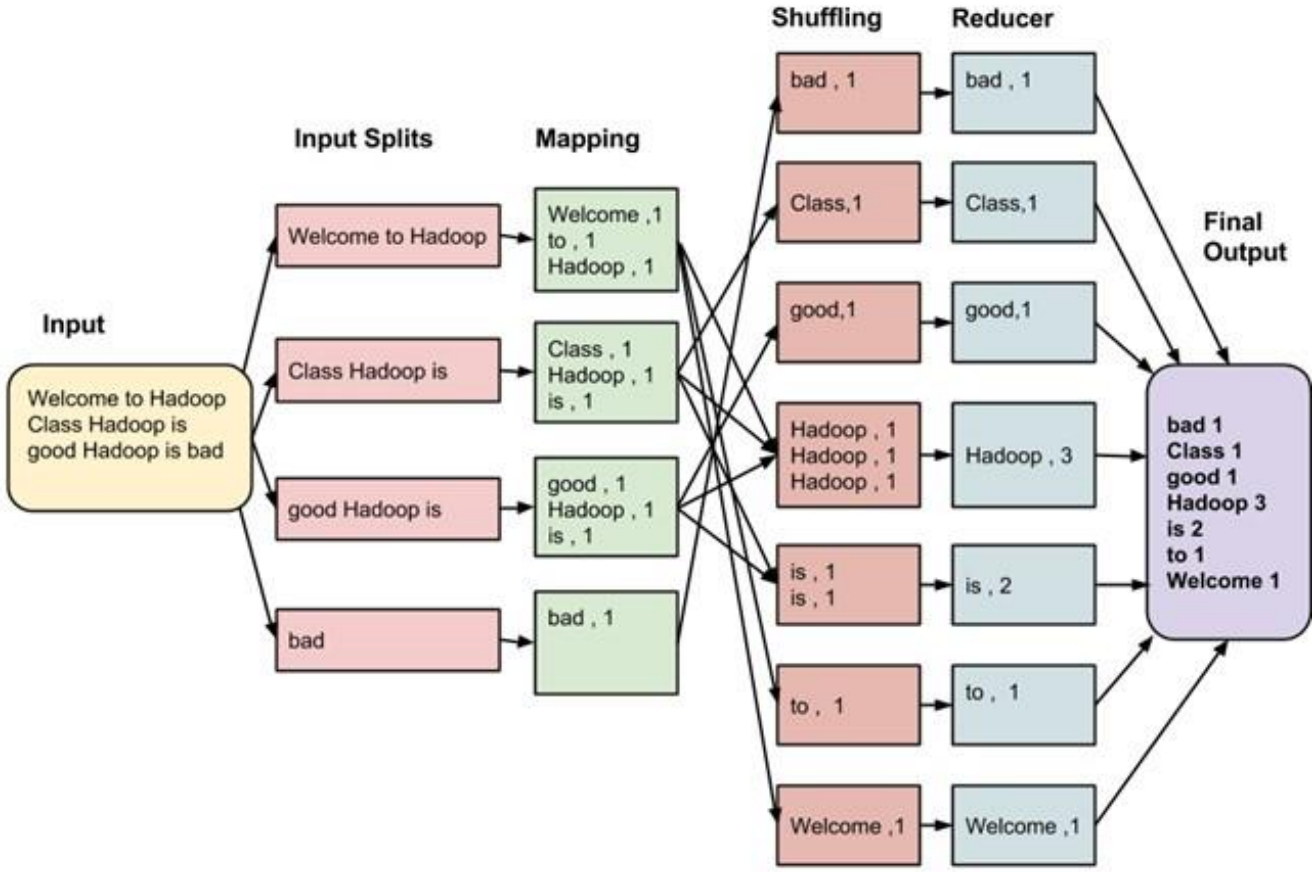


Fig. 2.22 Working of MapReduce

Dataflow in MapReduce

- Mapper is the first phase where datasets are splits into chunks.
- Mapper works on a one chunk at a time.
- The output from the mapper phase is written to the local disk machine where it is running.
- An output created by the mapper phase is known as Intermediate output.
- Intermediate output by mapper phase is written to the local disk always.
- When the mapper phase is done, the intermediate output is transferred to reducer node.
- Hence, the transferring of intermediate data from the mapper to reducer is known as Shuffle.
- If the shuffle phase will not happen then the reducer will not have any input to work on.
- The entire keys which are generated during mapper are sorted by MapReduce.

- It starts even before reducers and all the intermediate key-value pairs are generated by the mapper, are sorted by key and not by the value.
- Sorting helps reducers to start a new reduce task at the right time.

  Reduce task is started by the reducers when the key is sorted and input data is different from the previous.
- Key-value pairs are taken as input in very reduce task and key-value pairs as output are generated.
- If reducers are specified as zero then no shuffling and sorting are performed and which makes the mapper phase faster. (Figure 2.24) **Map Input**

- Map(inkey, invalue)-> list(intermediate key, intermediate value)
- Purpose of map phase is to organize data in preparation for processing done in reduce phase
- Input to map function is in the form of (key, value) pairs, even though the input to MapReduce program is a file or files.

Map

- Value: data record
- Key: Offset of data record from the beginning of the file
- Output: Collection of Key value pairs which are inputs for reduce function
- E.g. Wordcount Problem
- Mapper can run several identical mappers in parallel **Reduce**
- Reduce function processes intermediate values for particular key generated by map function and generates the output
- One to one mapping between keys and reducers
- Several reducers can run in parallel. Independent of each other
- No. of reducers decided by the user
- Default no. of reducers is 1 (Figure 2.23)

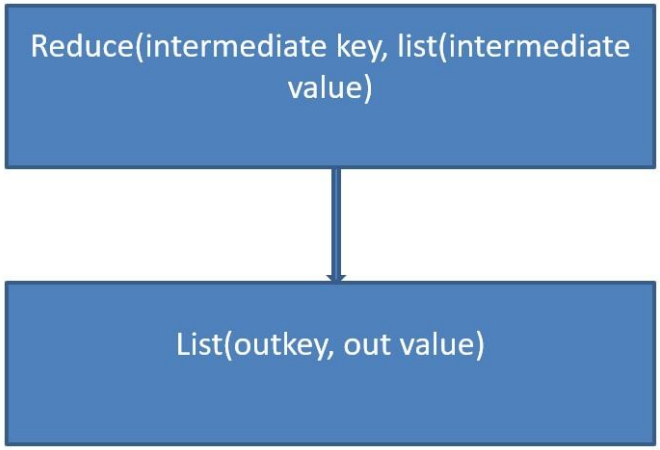


Fig. 2.23 Reduce

Map Reduce Example

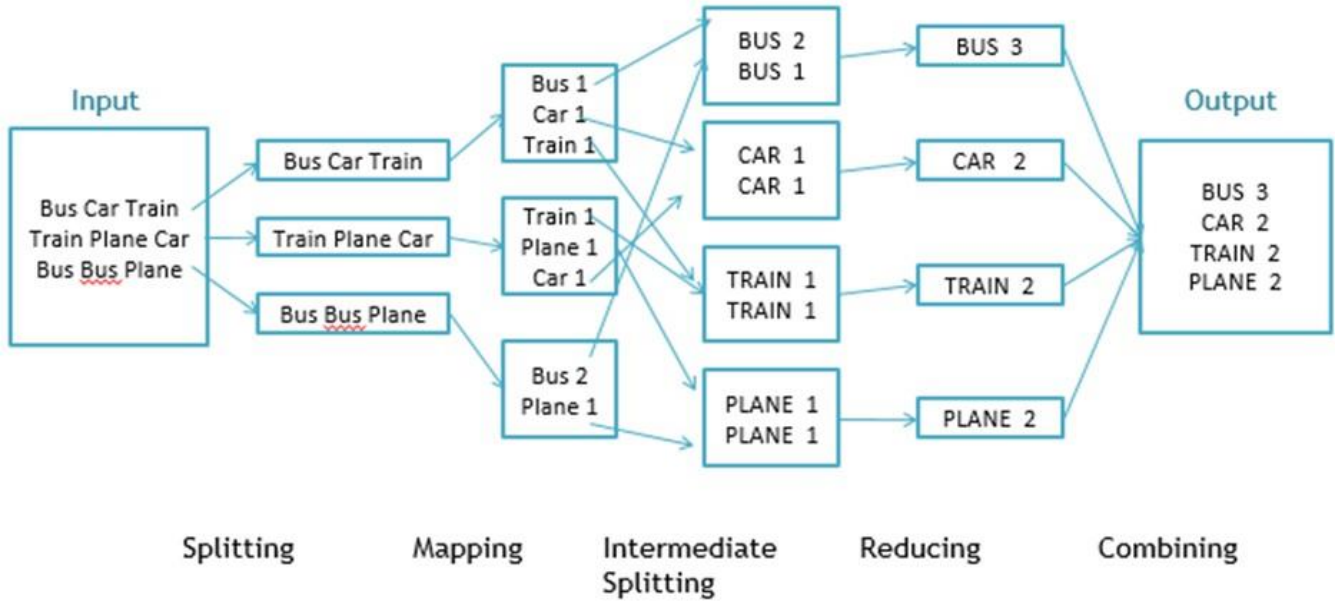


Fig. 2.24 MapReduce Example

Conclusion

- Hadoop MapReduce enables a high degree of parallelism, scalability, and fault tolerance.
- It is a versatile tool for data processing and it will help enterprises to gain importance.
- If you want to become a Hadoop administrator, it is mandatory to be familiar with the role and functionality of MapReduce.

Data Loading Techniques & Analysis

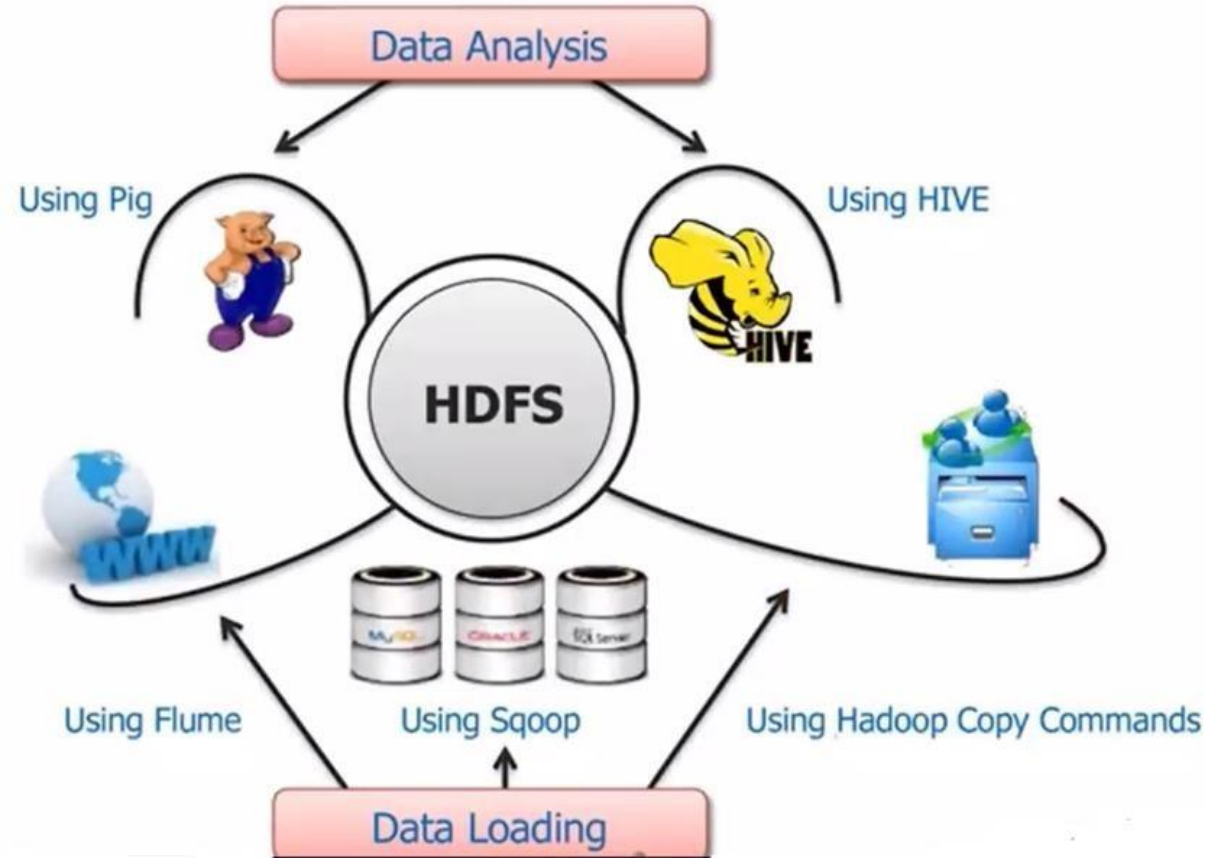


Fig. 2.25 Data Loading Techniques & Analysis

Data Loading using Flume

Flume is a distributed, reliable and available service for efficiently collecting, aggregating and moving large amounts of streaming event data (Figure 2.26).

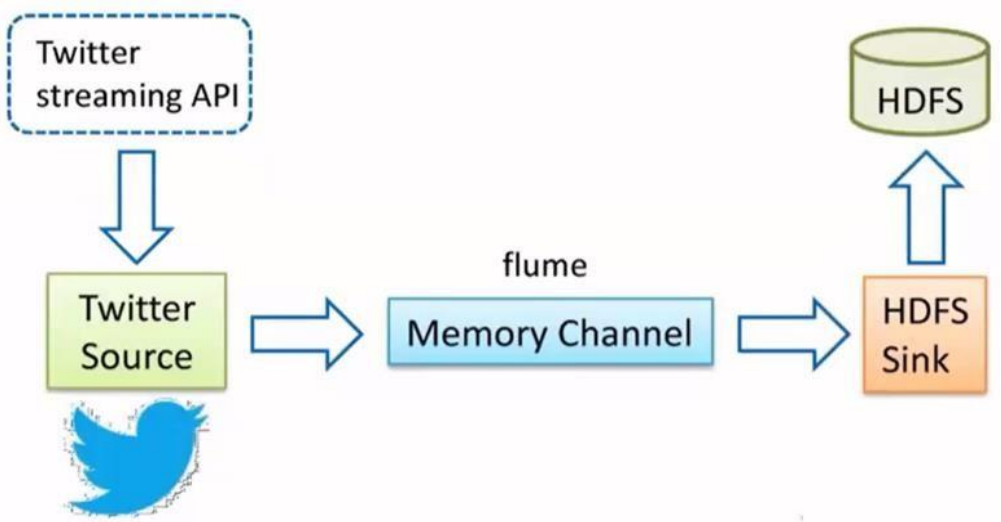


Fig. 2.26 Data Loading using Flume

Hadoop Cluster Architecture

- Cluster: Loosely/Tightly connected computers work together as a single system
- Hadoop Cluster: Storing & analysing large amount of unstructured data in distributed environment
- These clusters run on low cost on commodity computers

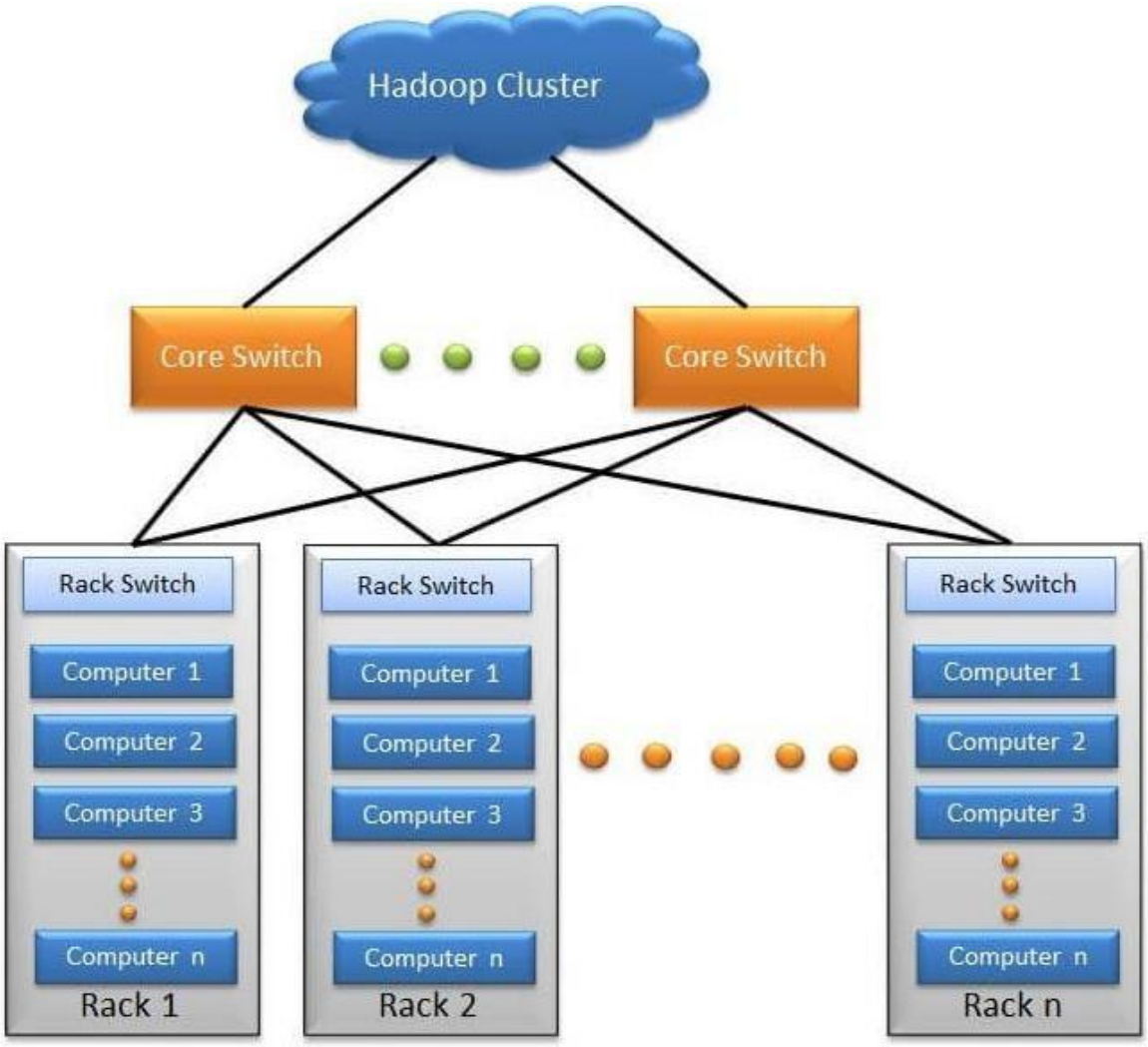


Fig. 2.27 Hadoop Cluster Architecture

Hadoop Cluster architecture as Share nothing systems

- Nodes share only the networks that connects them
- Large Hadoop clusters are arranged in several racks
- Network traffic between different nodes in the same rack is much more desirable than network traffic across the racks
- E.g. Hadoop cluster set up for 4500 nodes (Figure 2.27)

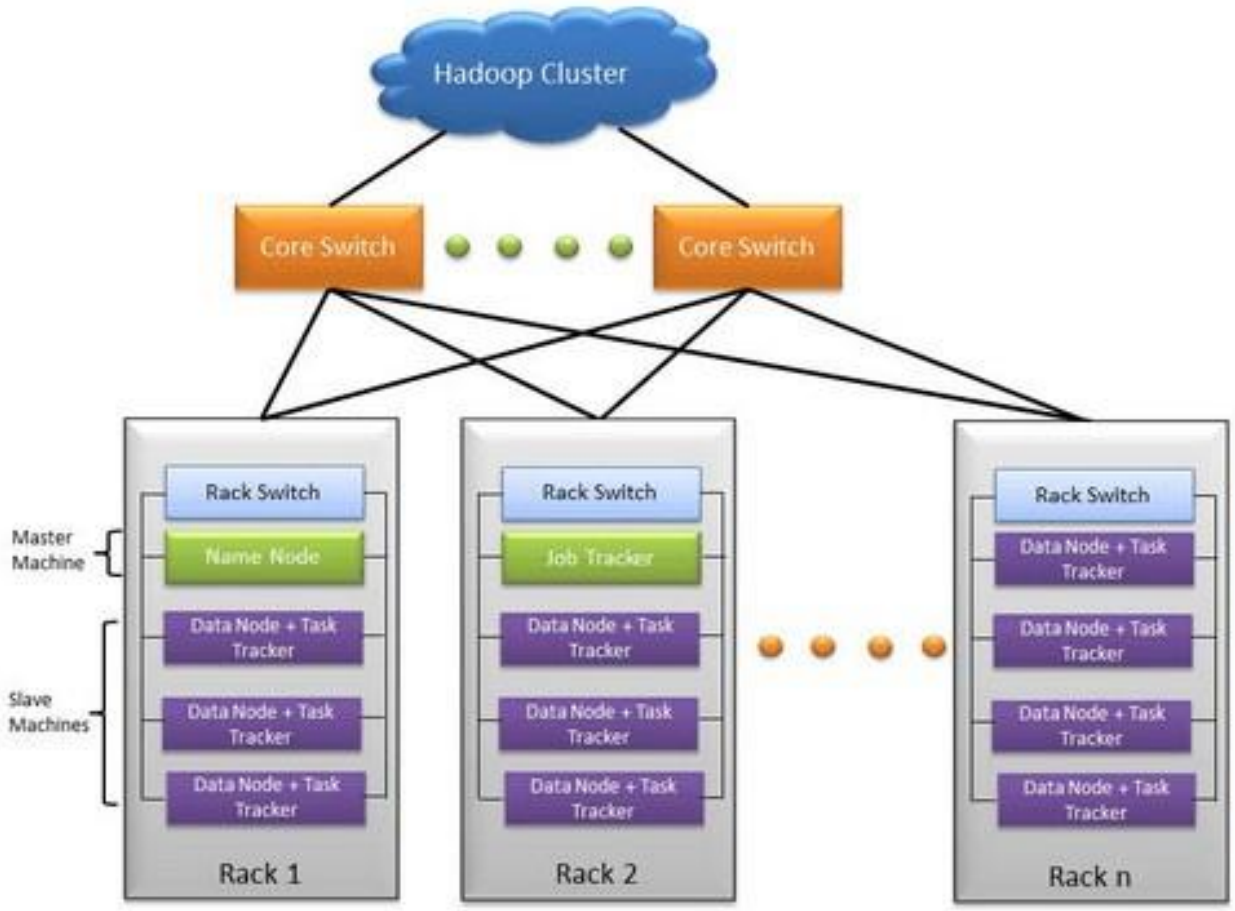


Fig. 2.28 Hadoop Cluster architecture as Share nothing systems

- In this Hadoop cluster (Figure 2.28)
  - 110 different racks
    - Each rack has 40 slave machines

Rack Description

- At the top of each rack there is a rack switch
- Each slave machine has cables coming out from both the ends
- Cables are connected to rack switch at the top which means that top rack switch will have around 80 ports
- There are global 8 core switches
- The rack which has uplinks connected to core switches and hence connecting all other racks with uniform bandwidth forming the cluster
- In the cluster, few machines act as NameNodes and JobTracker
- They are referred to as Masters

- Masters have different configuration for more DRAM and CPU and less storage
- The majority of machine acts as data node and Task trackers are referred as slaves
- These slave nodes have lots of local disk storage and moderate amounts of DRAM and CPU

Hadoop configuration files

- Hadoop configuration is driven by two types of important configuration files:
  - Read-only default configuration - src/core/core-default.xml, src/hdfs/hdfsdefault.xml and src/mapred/mapred-default.xml.
  - Site-specific configuration - *conf/core-site.xml*, *conf/hdfs-site.xml* and *conf/mapredsite.xml*.

Configuring the Environment of the Hadoop Daemons

Table 2.4 Hadoop Configuration Options

Daemon	Configuration Options
NameNode	HADOOP_NAMENODE_OPTS
DataNode	HADOOP_DATANODE_OPTS
SecondaryNamenode	HADOOP_SECONDARYNAMENODE_OPTS
JobTracker	HADOOP_JOBTRACKER_OPTS
TaskTracker	HADOOP_TASKTRACKER_OPTS

Configuring the Hadoop Daemons

Table 2.5 Specified in conf/core-site.xml

Parameter	Value
fs.default.name	URI of NameNode.

Table 2.6 conf/hdfs-site.xml

Parameter	Description
dfs.name.dir	Path on the local filesystem where the NameNode stores the namespace and transactions logs persistently.
dfs.data.dir	Comma separated list of paths on the local filesystem of a DataNode where it should store its blocks.

Table 2.7 conf/mapred-site.xml

Parameter	Value
mapred.job.tracker	Host or IP and port of JobTracker.
mapred.system.dir	Path on the HDFS where the MapReduce framework stores system files e.g. /hadoop/mapred/system/.
mapred.local.dir	Comma-separated list of paths on the local filesystem where temporary MapReduce data is written.
mapred.tasktracker.{map|reduce}.tasks.maximum	The maximum number of MapReduce tasks, which are run simultaneously on a given TaskTracker, individually.
dfs.hosts/dfs.hosts.exclude	List of permitted/excluded DataNodes.
mapred.hosts/mapred.hosts.exclude	List of permitted/excluded TaskTrackers.
mapred.queue.names	Comma separated list of queues to which jobs can be submitted.
mapred.acls.enabled	Boolean, specifying whether checks for queue ACLs and job ACLs are to be done for authorizing users for doing queue operations and job operations

Table 2.8 conf/mapred-queue-acls.xml

Parameter	Value
mapred.queue.*queue-name*.acl-submit-job	List of users and groups that can submit jobs to the specified *queue-name*.
mapred.queue.*queue-name*.acladminister-jobs	List of users and groups that can view job details, change the priority or kill jobs that have been submitted to the specified *queue-name*.

Introduction to Apache Hadoop

- With the continuous business growth and start-ups flourishing up, the need to store a large amount of data has also increased rapidly.
- To meet the growth of business and gain profits the companies would fetch tools to analyse the Big Data.
- To meet the growing demand, Apache Software Foundation launched Hadoop, a tool to store, analyse, and process Big Data.

What is Hadoop?

HDFS, Hadoop and Hadoop Infrastructure

- Hadoop is an open source Java-based framework for big data processing.

- It is a tool used to store, analyse and process Big Data in the distributed environment.
- The Hadoop is an open source project of Apache Software Foundation and was originally created by Yahoo in 2006.
- Since then, this open source project has brought revolution in Big Data analytics and taken over the Big Data market.
- In simple terms, Apache Hadoop is a tool used to handle big data.
- It is used to work on large sets of data distributed over a number of computers using some programming languages.
- Apache Hadoop is easily scalable and you can scale a number of machines through a single server.
- Apache Hadoop runs on Java and is an open source framework used to process and store a large amount of data on a huge cluster.
- Hadoop framework is composed of four main components (Figure 2.29):
- Hadoop Common
- Hadoop Distributed File System
- YARN
- MapReduce

The Apache framework is composed of the following components:

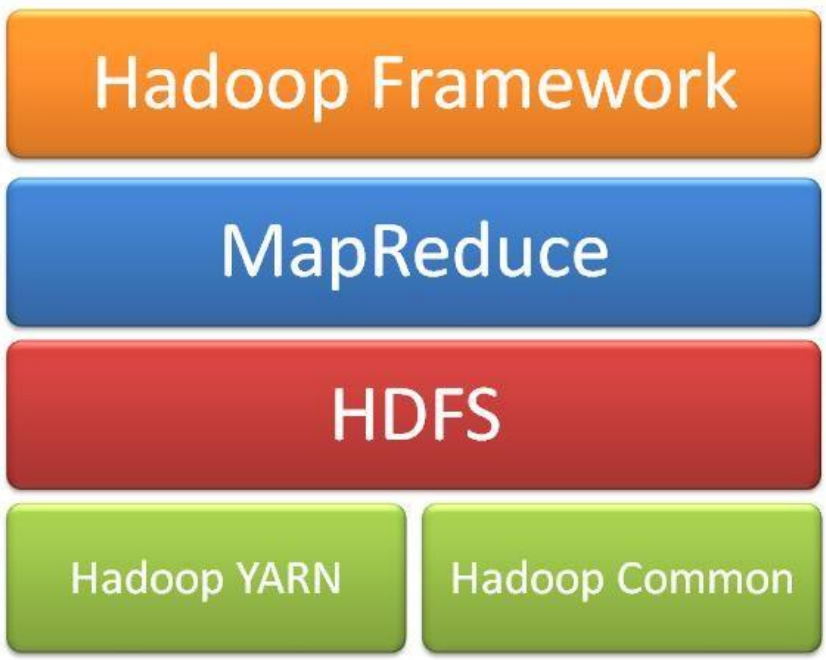


Fig. 2.29 Hadoop framework Components

- **Hadoop Common:**
  - It refers to the common Java utilities and libraries that support Hadoop modules.
- **Hadoop Distributed File Systems:**
  - It is the primary storage system that Hadoop applications use.
  - It is a distributed file system that enables you to have an access to applications data.
- **Hadoop MapReduce:**
  - Hadoop MapReduce is a software framework used for the parallel processing of big data.
- **Hadoop YARN:**
  - YARN is the resource management technology used by Hadoop.
  - It is responsible for resource management and job scheduling **How does Hadoop Work?**

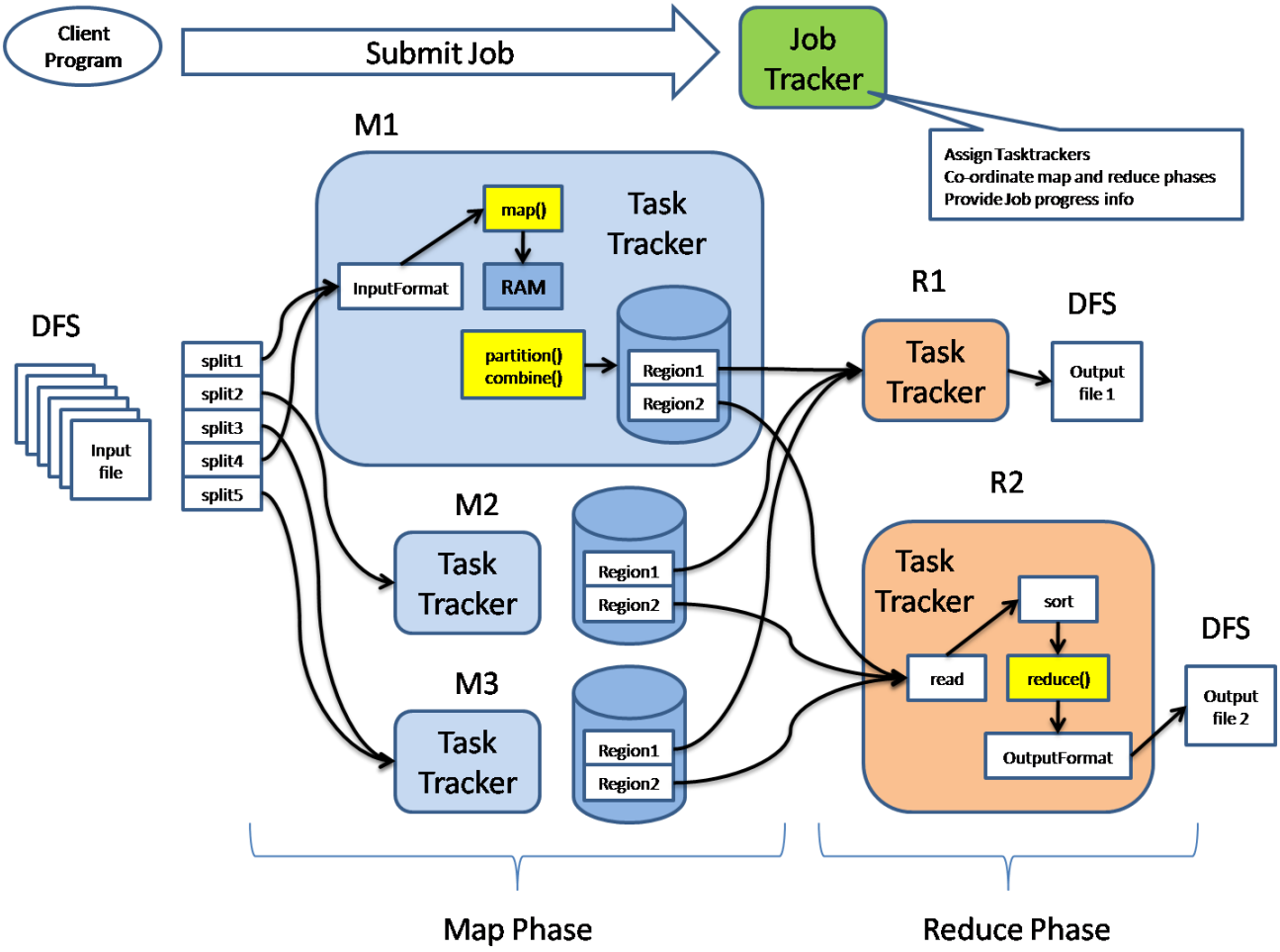


Fig. 2.30 Working of Hadoop

- The working of Hadoop is a three-stage procedure. Let's understand how exactly Hadoop works (Figure 2.30):
- **Stage 1:** The job is submitted to the Hadoop job client for the required process with following details –
  - Input and Output file location in the distributed file system
  - The Java classes with the implementation of Map and Reduce functions ▪ Job configuration with the different parameter set ⦿ **Stage 2:**
  - Hadoop job client transfers job along with job configuration to the JobTracker.
  - The JobTracker is then responsible to perform for configuration distribution to slaves, tasks scheduling, and monitoring, submitting status update back to the job client.
- **Step 3:**
  - At different nodes, TaskTrackers then execute the tasks according to MapReduce implementation.
  - The output generated by the Reduce function is stored on the distributed file system in output files.

Features of Apache Hadoop

- Scalability: Apache Hadoop uses distributed processing of local data, this allows the data to be stored, processed, and analyzed at a large scale.
- Reliability: In Apache Hadoop, the data is auto-replicated and hence can generate a redundant copy of data when it comes to system failures. Thus, Apache Hadoop has fault tolerance feature.
- Flexibility: Apache Hadoop does not follow the traditional relational database rules. It can store information and data in any format such as structured, unstructured, and semi-structured.
- Cost-effectiveness: Apache Hadoop is open source and is free of cost. This makes it costeffective and available for all.
- Compatibility: Apache Hadoop, being a Java-based framework, is compatible with all the platforms.

Hadoop Cluster Modes

- Standalone mode
- Single node cluster/Pseudo distributed mode
- Multi node cluster/ Fully Distributed mode

Standalone mode

- Default mode
- HDFS is not utilized in this mode
- Local file system is used for input and output
- Used for debugging purpose
- No custom configuration is required in 3 Hadoop files
    - mapred- site.xml
    - core-site.xml
    - hdfs-site.xml
- Standalone mode is much faster than pseudo distributed mode **Pseudo distributed mode**
- Configuration is required in given 3 files
- Replication factor is one for HDFS
- Here, one node is used as Master node/data node/Job tracker/Task tracker
- Used for real code to test in HDFS
- Pseudo distributed cluster is a cluster where all daemons are running on one node itself **Fully Distributed mode/Multiple node cluster**
- Production phase
- This mode involves the code running on an actual Hadoop cluster.
- It is mode in which you see the actual power of Hadoop, when you run your code against a very large input on 1000s of servers.
- It is always difficult to debug a MapReduce program as you have Mappers running on different machine with different piece of input.
- You can never know where the Mappers are going to run eventually.
- Also, with large inputs, it is likely that the data will be irregular in its format.

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

COMMON TO CSE AND IT

UNIT – III – SIT1606 – BIG DATA

UNIT 3 HADOOP MAP REDUCE FRAMEWORK **9 Hrs.**

Relationship between MapReduce and HDFS - Clients, Data Nodes, and HDFS Storage - MapReduce workloads. Hadoop framework - Hadoop data types - Hadoop map reduce Paradigm - Map and Reduce Tasks - Map reduce Execution framework - Partitioners and Combiners - Input formats (Input Splits and Records, Text Input, Binary Input, Multiple Inputs)- Output Formats (TextOutput, BinaryOutPut, Multiple Output)- Hadoop Mapreduce programming - Advanced Map reduce concepts - Counters, Custom Writables - Unit testing framework - Error Handling - Tuning - Advanced Map reduce.

HADOOP MAPREDUCE FRAMEWORK

MAPREDUCE

- MapReduce is a programming framework of Hadoop that is used for parallel processing of data. MapReduce is the processing engine of Hadoop that processes and computes vast volumes of data.
- It has 2 components: Map and Reduce phase which is shown in Fig. 3.1.

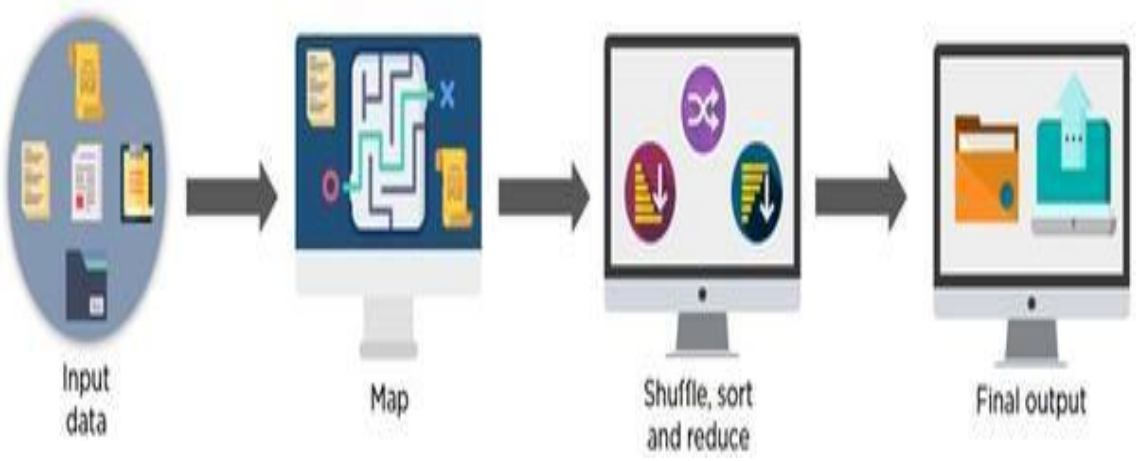


Fig. 3.1 Components of Hadoop

- The generalized workflow of MapReduce is shown in Fig. 3.2.

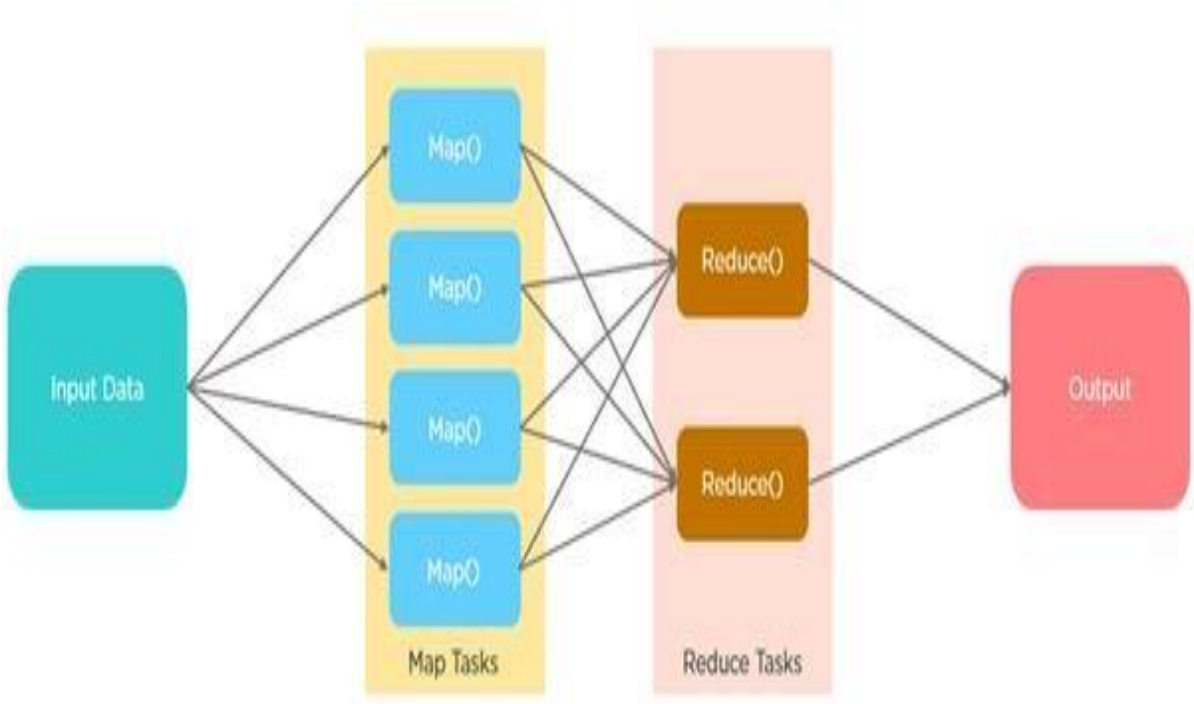


Fig. 3.2 Generalized Workflow of MapReduce

- MapReduce performs parallel processing in the manner shown in Fig. 3.3.

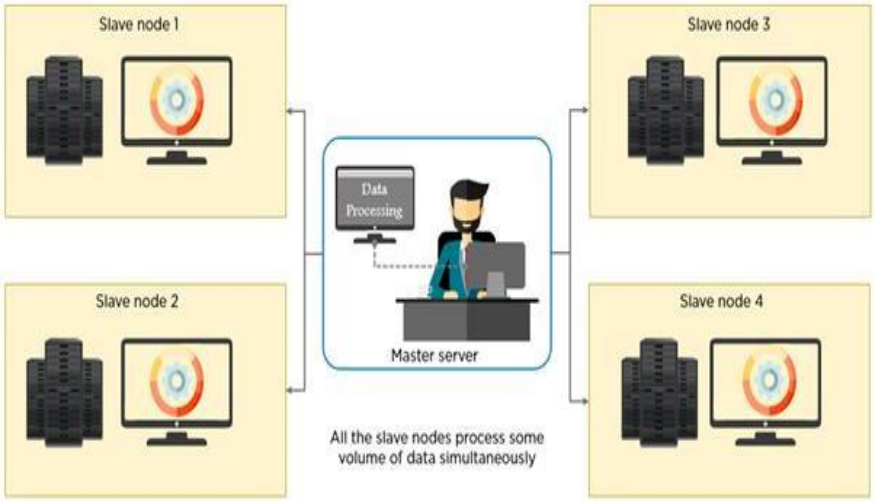


Fig. 3.3 Parallel Processing of MapReduce •

The detailed workflow of MapReduce is shown in the Fig. 3.4:

Hadoop MapReduce Framework

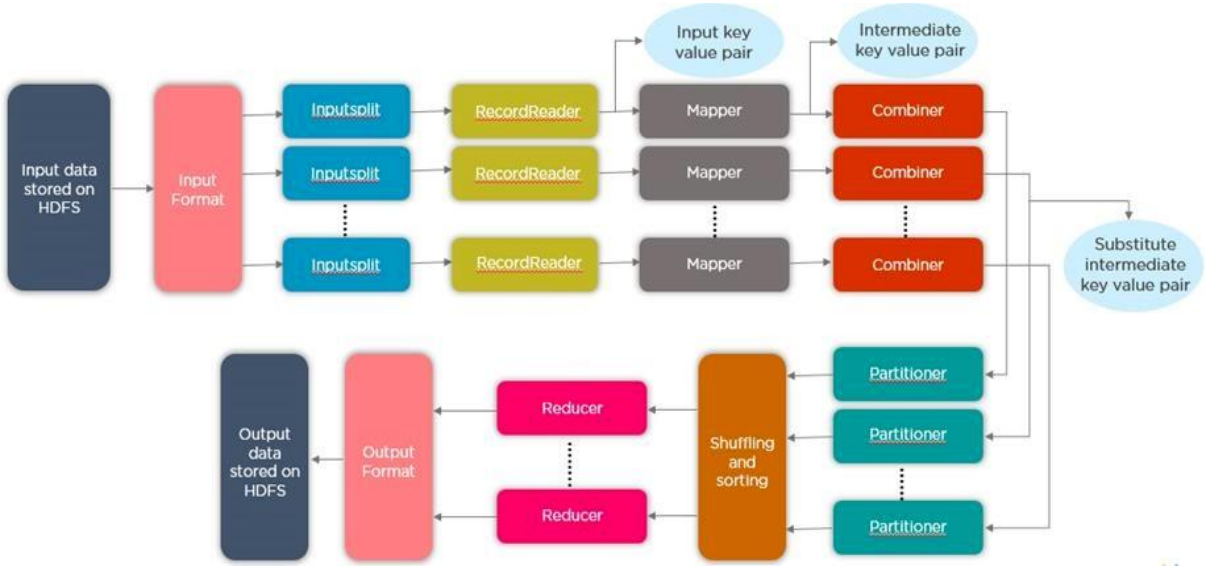


Fig. 3.4 Detailed Workflow of MapReduce

- So, to conclude, MapReduce does parallel processing of data in Hadoop.

HDFS – HADOOP DISTRIBUTED FILE SYSTEM

- HDFS provides high throughput access to application data and is suitable for applications that have large data sets.
- The NameNode executes file system namespace operations like opening, closing, and renaming files and directories.
- The DataNodes are responsible for serving read and write requests from the file system's clients.

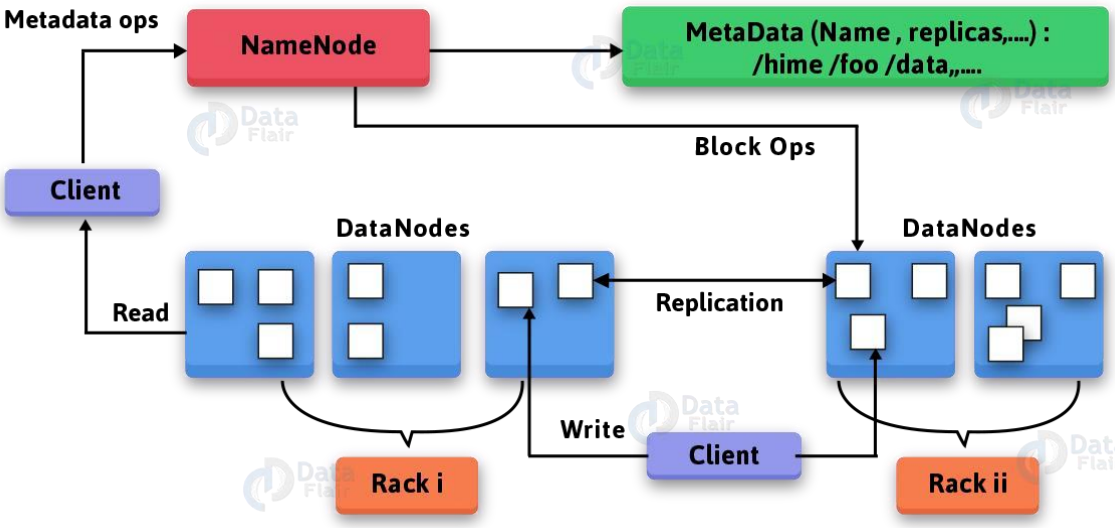


Fig. 3.5 Hadoop Distributed File System

Working of HDFS

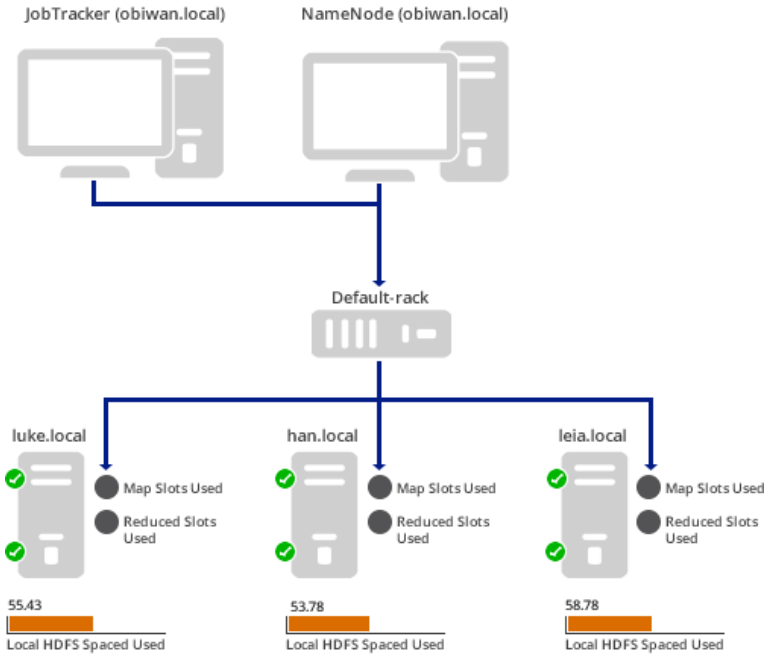


Fig. 3.6 Working of HDFS

RELATIONSHIP BETWEEN MAPREDUCE AND HDFS

- We compare the Hadoop software framework as a computer, the MapReduce is the same as software, and the HDFS is the same as hardware.
- MapReduce is a framework that is used by Hadoop to process the data residing with HDFS.
- HDFS store data in each block which size is 64MB or 128MB and MapReduce can interaction with HDFS and operates the data in HDFS.
- The configuration which we set when we set the environment of Hadoop. The below diagram will show the structure of Hadoop

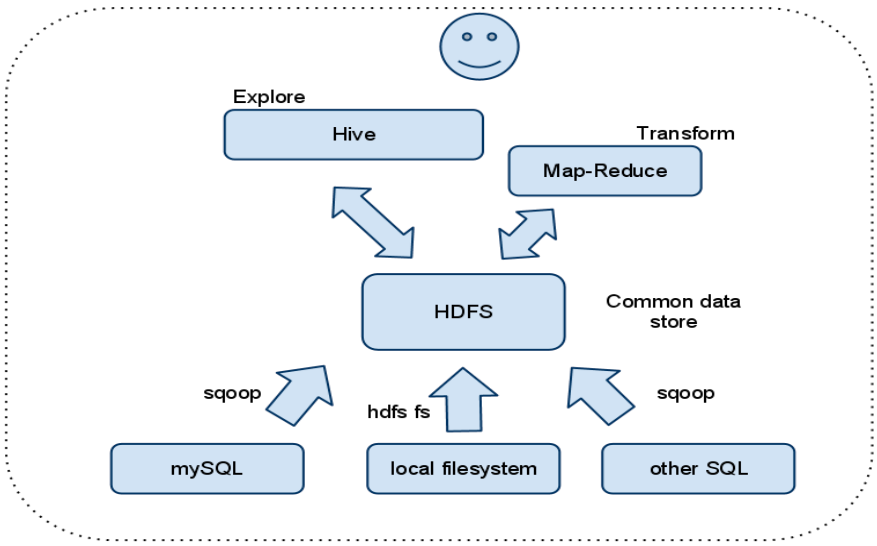


Fig. 3.7 Relationship between MapReduce and HDFS

- HDFS is a distributed file system that provides high throughput access to application data
- MapReduce is a software framework that processes big data on large clusters reliably.
- Hadoop supports distributed processing of large data sets across clusters of computers.
- HDFS and MapReduce are two modules in Hadoop architecture.

HDFS vs MapReduce

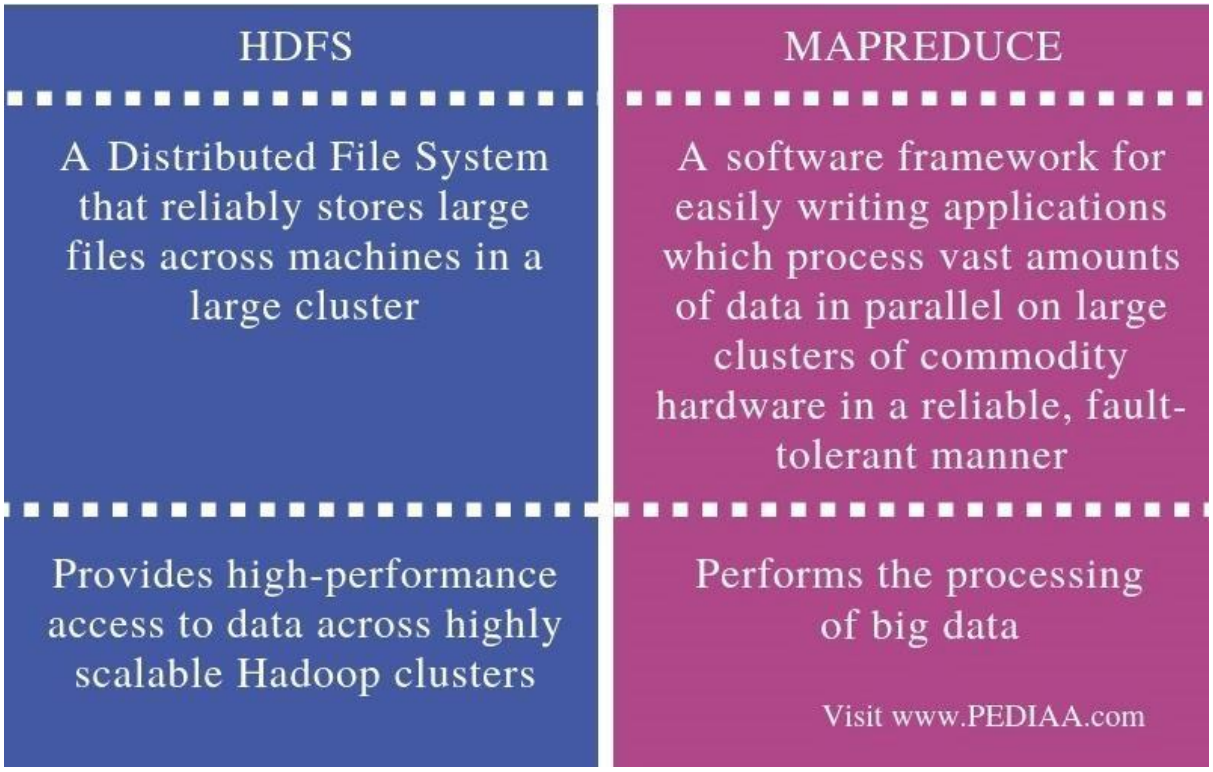


Fig. 3.8 HDFS vs. MapReduce

- Another difference between HDFS and MapReduce is that the HDFS provides highperformance access to data across highly scalable Hadoop clusters while MapReduce performs the processing of big data.

HDFS CLIENT

- Client in Hadoop refers to the Interface used to communicate with the Hadoop Filesystem.
- There are different types of Clients available with Hadoop to perform different tasks.
- The basic filesystem client HDFS DFS is used to connect to a Hadoop Filesystem and perform basic file related tasks.
- It uses the ClientProtocol to communicate with a NameNode daemon, and connects directly to DataNodes to read/write block data.
- These Clients can be invoked using their respective CLI (Command Line Interface) commands from a node where Hadoop is installed and has the necessary configurations

and libraries required to connect to a Hadoop Filesystem. Such nodes are often referred as Hadoop Clients.

- For example, if I just write an HDFS command on the Terminal, is it still a "client"?
- Technically, Yes. If you are able to access the FS using the HDFS command, then the node has the configurations and libraries required to be a Hadoop Client.

HDFS DATANODE

- A DataNode stores data in the Hadoop Filesystem.
- A functional filesystem has more than one DataNode, with data replicated across them.
- On startup, a DataNode connects to the NameNode; spinning until that service comes up. It then responds to requests from the NameNode for filesystem operations.
- Client applications can talk directly to a DataNode, once the NameNode has provided the location of the data.
- Similarly, MapReduce operations farmed out to TaskTracker instances near a DataNode, talk directly to the DataNode to access the files.
- TaskTracker instances can be deployed on the same servers of the host DataNode instances, so that MapReduce operations are performed close to the data.
- DataNode instances can talk to each other, which is what they do when they are replicating data.
- Data is designed to replicate across multiple servers, rather than multiple disks on the same server.
- An ideal configuration is for a server to have a DataNode, a TaskTracker, and then physical disks with one TaskTracker slot per CPU.
- This will allow every TaskTracker 100% of a CPU, and separate disks to read and write data.

Map Task (HDFS Data Localization):

- The unit of input for a map task is an HDFS data block of the input file. The map taskfunctions most efficiently if the data block it has to process is available locally on the node on which the task is scheduled. This approach is called HDFS data localization.
- An HDFS data locality miss occurs if the data needed by the map task is not available locally. In such a case, the map task will request the data from another node in the cluster: an operation that is expensive and time consuming, leading to inefficiencies and, hence, delay in job completion.

CLIENTS, DATA NODES, AND HDFS STORAGE:

Input data is uploaded to the HDFS file system in either of following two ways:

1. An HDFS client has a large amount of data to place intoHDFS.
2. An HDFS client is constantly streaming data intoHDFS.

- Both these scenarios have the same interaction with HDFS, except that in the streaming case, the client waits for enough data to fill a data block before writing to HDFS.
- Data is stored in HDFS in large blocks, generally 64 to 128 MB or more in size.
- This storage approach allows easy parallel processing of data.

HDFS-SITE.XML

<property>
<name>dfs.block.size</name>
<value>134217728</value> ç 128MB Block size
</property>
OR
<property>
<name>dfs.block.size</name>
<value>67108864</value> ç 64MB Block size (Default is this value is not set)
</property>

Block Replication Factor:

- During the process of writing to HDFS, the blocks are generally replicated to multiple data nodes for redundancy.
- The number of copies, or the replication factor, is set to a default of 3 and can be modified by the cluster administrator as below:

HDFS-SITE.XML
<property>
<name>dfs.replication</name>
<value>3</value>
</property>
<property>
When the replication factor is three, HDFS's placement policy is to:

- Put one replica on one node in the localrack,
- Another on a node in a different (remote)rack,
- Last on a different node in the same remoterack.

- When a new data block is stored on a data node, the data node initiates a replication process to replicate the data onto a second data node.

- The second data node, in turn, replicates the block to a third data node, completing the replication of the block.
- With this policy, the replicas of a file do not evenly distribute across the racks.
- One third of replicas are on one node, two thirds of replicas are on one rack, and the other third are evenly distributed across the remaining racks.
- This policy improves write performance without compromising data reliability or read performance.

HADOOP DATA TYPES

Serialization:

- Serialization is the process of converting object data into byte stream data for transmission over a network across different nodes in a cluster or for persistent data storage.

Deserialization:

- Deserialization is the reverse process of serialization and converts byte stream data into object data for reading data from HDFS.
- Hadoop provides Writables for serialization and deserialization purpose.

Writable and WritableComparable Interfaces:

- To provide mechanisms for serialization and deserialization of data, Hadoop provided two important interfaces Writable and WritableComparable.
- Writable interface specification is as follows:

```
package org.apache.hadoop.io; import java.io.DataInput;
import java.io.DataOutput; import java.io.IOException; public interface Writable
{
void write (DataOutput out) throws IOException; void readFields(DataInput in) throws
IOException;
}
```

- WritableComparable interface is subinterface of Hadoop's Writable and Java's Comparable interfacesand its specification is shown below:

```
public interface WritableComparable extends Writable, Comparable
{
}
```

- The standard java.lang.Comparable Interface contains single method compareTo() method for comparing the operators passed to it.

```
public interface Comparable
```

```
{
public int compareTo(Object obj);
}
```

- The compareTo() method returns 1, 0 , or 1 depending on whether the compared object is less than, equal to, or greater than the current object.
- The above two interfaces are provided in org.apache.hadoop.io package

Constraints on Keyvalues in Mapreduce

- Hadoop data types used in Mapreduce for key or value fields must satisfy two constraints.
  - Any data type used for a Value field in mapper or reducer input/output must implementWritable Interface.
  - Any data type used for a Key field in mapper or reducer input/output must implement WritableComparable interface along with Writable interface to compare the keys of this type with each other for sorting purposes

Writable Classes – Hadoop Data Types

- Hadoop provides classes that wrap the Java primitive types and implement the WritableComparableand Writable Interfaces. They are provided in the org.apache.hadoop.io package.
- All the Writable wrapper classes have a get() and a set() method for retrieving and storing the wrapped value.

Primitive Writable Classes

- These are Writable Wrappers for Java primitive data types and they hold a single primitive value that can be set either at construction or via a setter method.
- All these primitive writable wrappers have get() and set() methods to read or write the wrapped value. Below is the list of primitive writable data types available in Hadoop.
  - BooleanWritable
  - ByteWritable
  - IntWritable
  - VIntWritable
  - FloatWritable
  - LongWritable
  - VLongWritable
  - DoubleWritable

- In the above list VIntWritable and VLongWritable are used for variable length Integer types and variable length long types respectively.
- Serialized sizes of the above primitive writable data types are same as the size of actual java data type. So, the size of IntWritable is 4 bytes and LongWritable is 8 bytes.

Array Writable Classes

- Hadoop provided two types of array writable classes, one for singledimensionaland another for twodimensional arrays.
- But the elements of these arrays must be other writable objects like IntWritable or LongWritable only but not the java native data types like int or float.
    - ArrayWritable
    - TwoDArrayWritable

Map Writable Classes

- Hadoop provided below MapWritable data types which implement java.util.Map interface
    - AbstractMapWritable – This is abstract or base class for other MapWritable classes.
    - MapWritable – This is a general-purpose map mapping Writable keys to Writable values.
    - SortedMapWritable – This is a specialization of the MapWritable class that also implements the SortedMap interface.

Other Writable Classes

- NullWritable→NullWritable is a special type of Writable representing a null value. No bytes are read or written when a data type is specified as NullWritable. So, in Mapreduce, a key or a value can be declared as a NullWritable when we don't need to use that field.
- ObjectWritable→This is a generalpurposegeneric object wrapper which can store any objects like Java primitives, String, Enum, Writable, null, or arrays.
- Text→Text can be used as the Writable equivalent of java.lang.String and It's max size is 2 GB. Unlike java's String data type, Text is mutable in Hadoop.
- BytesWritable →BytesWritable is a wrapper for an array of binary data.
- GenericWritable → It is similar to ObjectWritable but supports only a few types. User need to subclass this GenericWritable class and need to specify the types to support.

Example Program to Test Writables

- Let's write a WritablesTest.java program to test most of the data types mentioned above in this post with get (), set (), getBytes(), getLength(), put(), containsKey(), keySet() methods.

```
import org.apache.hadoop.io. * ;  import java.util.* ; public class WritablesTest
{
public static class TextArrayWritable extends ArrayWritable
{
public TextArrayWritable()
{
super (Text.class) ;
} }
public static class IntArrayWritable extends ArrayWritable
{
public IntArrayWritable()
{
super (IntWritable.class) ;
} }
public static void main (String [] args)
{
IntWritable i1 = new IntWritable(2) ;
IntWritable i2 = new IntWritable() ;
i2.set (5) ;
IntWritable i3 = new IntWritable () ;
i3.set (i2.get ()) ;
System.out.printf("Int Writables Test I1: %d, I2: %d, I3: %d", i1.get (), i2.get (), i3.get ()) ;
BooleanWritable bool1 = new BooleanWritable() ; bool1.set(true);
ByteWritable byte1 = new ByteWritable((byte)7) ;
System.out.printf("\n Boolean Value: %s Byte Value: %d", bool1.get (), byte1.get ()) ; Text t
= new Text("hadoop") ;
Text t2 = new Text () ; t2.set("pig") ;
System.out.printf("\n t: %s, t.legth: %d, t2: %s, t2.length: %d \n", t.toString(), t.getLength(),
t2.getBytes(), t2.getBytes().length);
ArrayWritable a = new ArrayWritable(IntWritable.class) ;
```

```
a.set(new IntWritable[]{new IntWritable(10), new IntWritable(20), new IntWritable(30)});
ArrayWritable b = new ArrayWritable(Text.class);
b.set (new Text [] {new Text("Hello"), new Text("Writables"), new Text ("World !!!")}) ;
for (IntWritable i: (IntWritable[]) a.get()) System.out.println(i) ; for (Text i: (Text [])
b.get())
System.out.println(i) ;
IntArrayWritableia = new IntArrayWritable();
ia.set (newIntWritable[]{new IntWritable(100), new IntWritable(300), new
IntWritable(500)}) ;
IntWritable[] ivalues = (IntWritable[])ia.get() ; for
(IntWritablei:ivalues)
System.out.println(i);
MapWritable m = new MapWritable() ; IntWritable key1 = new IntWritable(1) ;  NullWritable
value1 = NullWritable.get() ;
m.put(key1, value1) ;
m.put(new VIntWritable(2), new LongWritable(163));
m.put(new VIntWritable(3), new Text("Mapreduce"));
System.out.println(m.containsKey(key1));  System.out.println(m.get(new
VIntWritable(3)));
m.put(new LongWritable(1000000000), key1);
Set<Writable> keys = m.keySet() ;
For(Writable w: keys)
System.out.println(m.get(w)) ;
}
}
```

MAPREDUCE PARADIGM

- Splits input files into blocks (typically of 64MB each)
- Operated on key/value pairs
- Mapper filter & transform input data
- Reducers aggregate mappers output
- Efficient way to process the cluster:
    - Move code to data
    - Run code on all machines

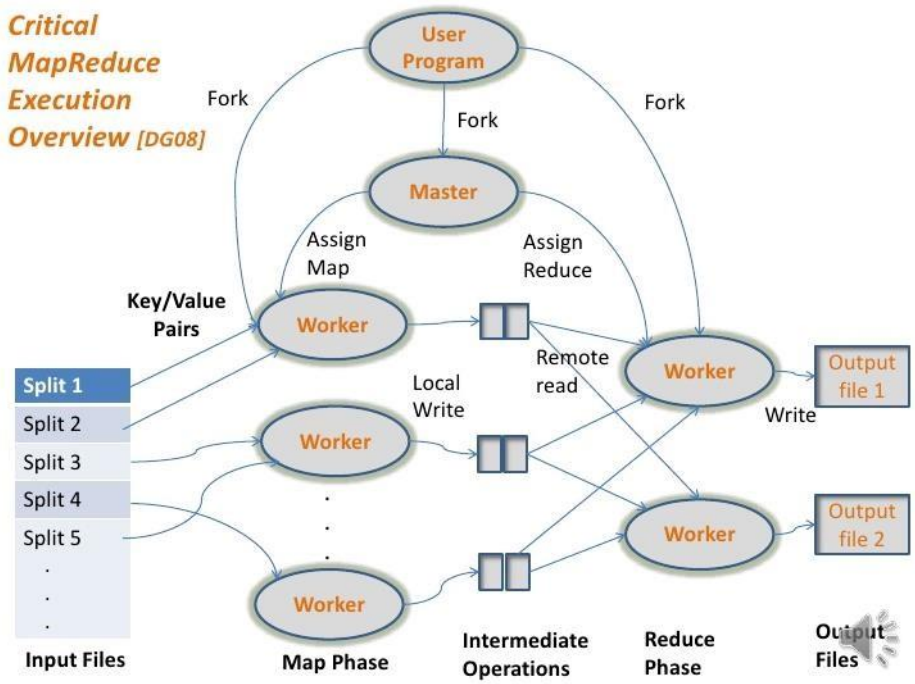


Fig. 3.9 MapReduce Paradigm

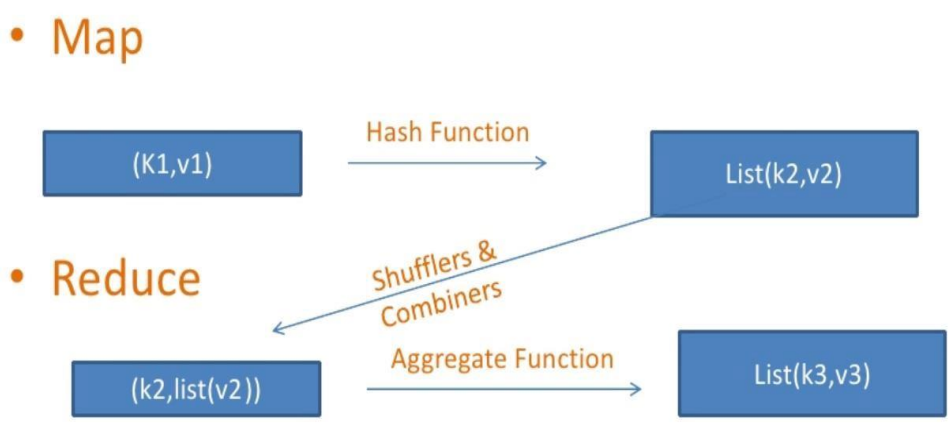


Fig. 3.10 Map and Reduce Phase

Job Scheduling in Hadoop

- One map task for each block of the input file
    - Applies user-defined map function to each record in the block ○ Record = <key, value>
- User-defined number of reduce tasks
    - Each reduce task is assigned a set of record groups
    - For each group, apply user-defined reduce function to the record values in that group

- Reduce tasks read from every map task
    - Each read returns the record groups for that reduce task

Visual understanding of Dataflow in Hadoop

- Map tasks write their output to local disk
    - Output available after map task has completed
- Reduce tasks write their output to HDFS
    - Once job is finished, next job's map tasks can be scheduled, and will read input from HDFS
- Therefore, fault tolerance is simple: simply re-run tasks on failure
    - No consumers see partial operator output

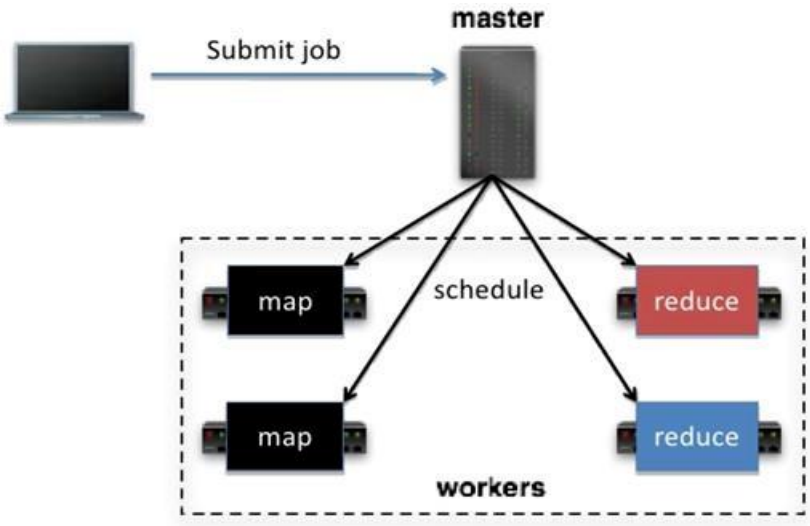


Fig. 3.11 Step1 of Dataflow in Hadoop

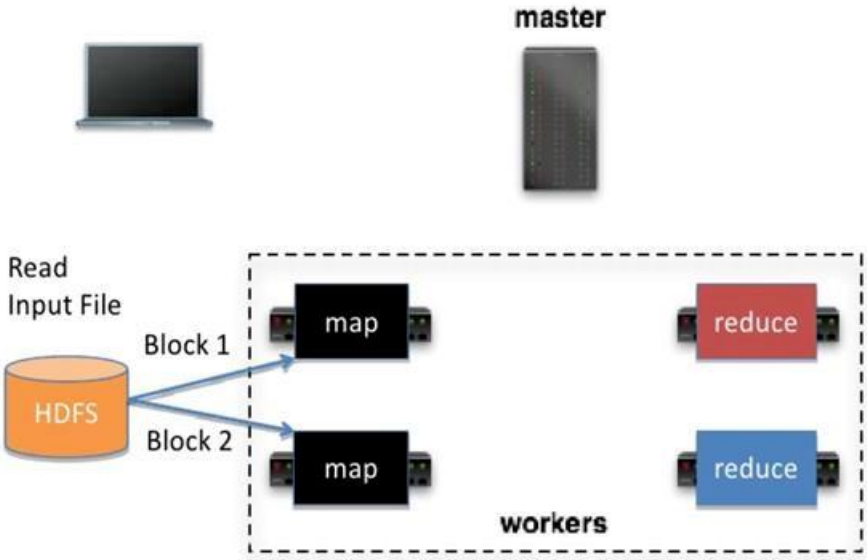


Fig. 3.12 Step2 of Dataflow in Hadoop

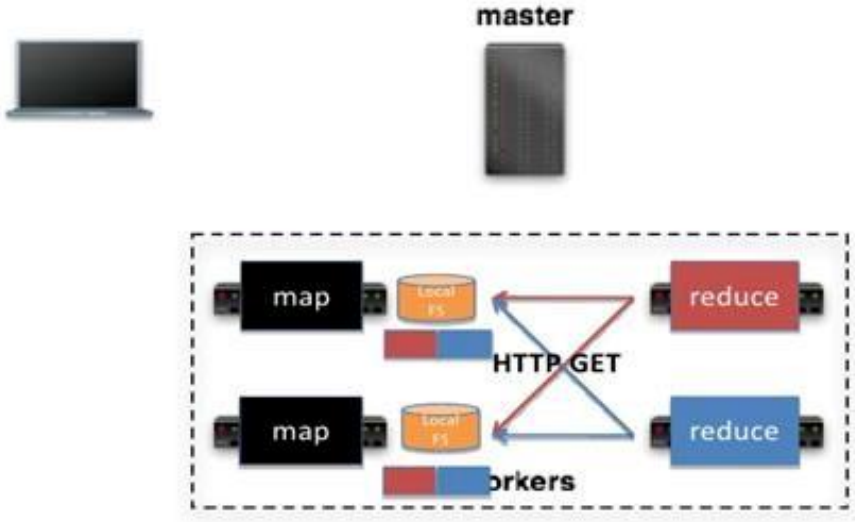


Fig. 3.13 Step 3 of Dataflow in Hadoop

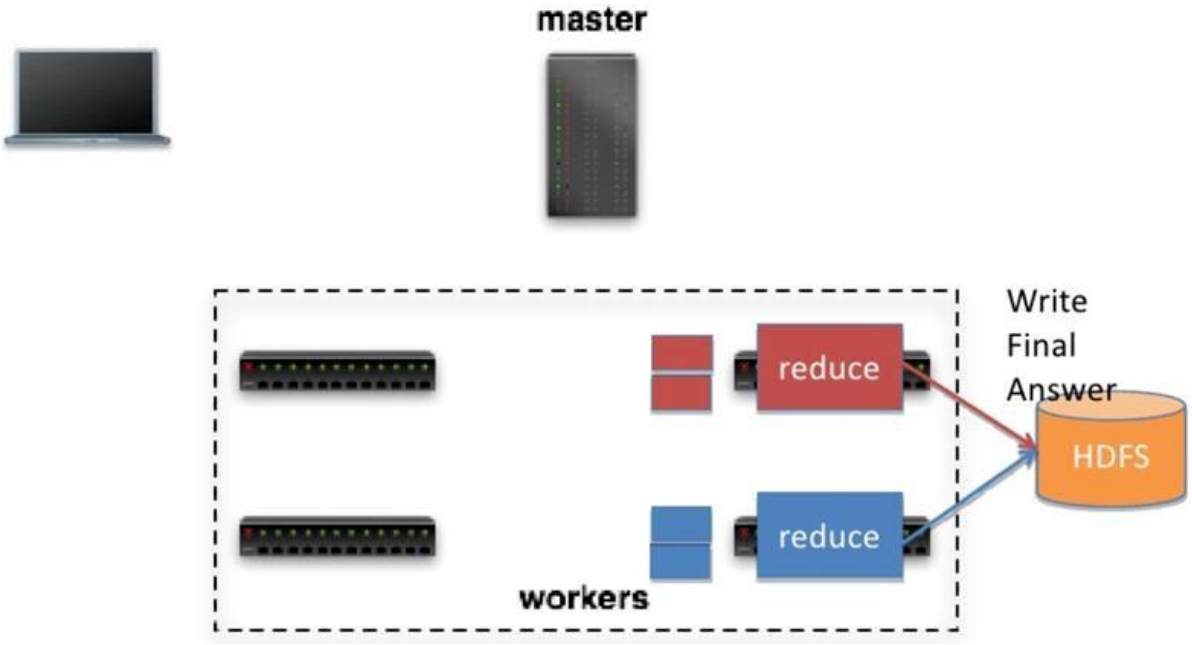


Fig. 3.14 Step 4 of Dataflow in Hadoop

- Terms are borrowed from Functional Language (e.g., Lisp)

Sum of Squares:

- (map square '(1 2 3 4))
    - Output: (1 4 9 16) ○ [processes each record sequentially and independently]
- (reduce + '(1 4 9 16))
    - (+16 (+9 (+4 1))) ○ Output: 30 ○ [processes set of all records in batches]
- Let's consider a sample application: Wordcount

o You are given a huge dataset and asked to list the count for each of the words in each of the documents therein o Process individual records to generate intermediate key/value pairs.

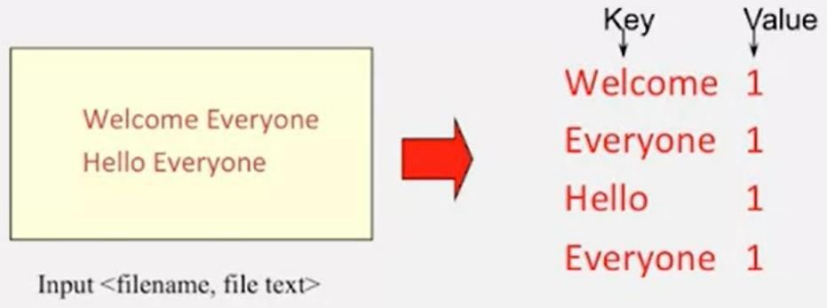


Fig. 3.15 Individual Record Processing

o Parallelly process individual records to generate intermediate key/value pairs.

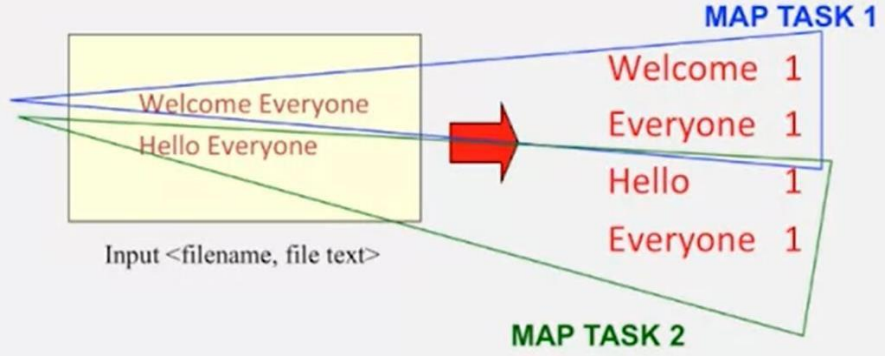


Fig. 3.16 Parallel Processing of Individual Record o Parallelly process a large number of individual records to generate intermediate key/value pairs.

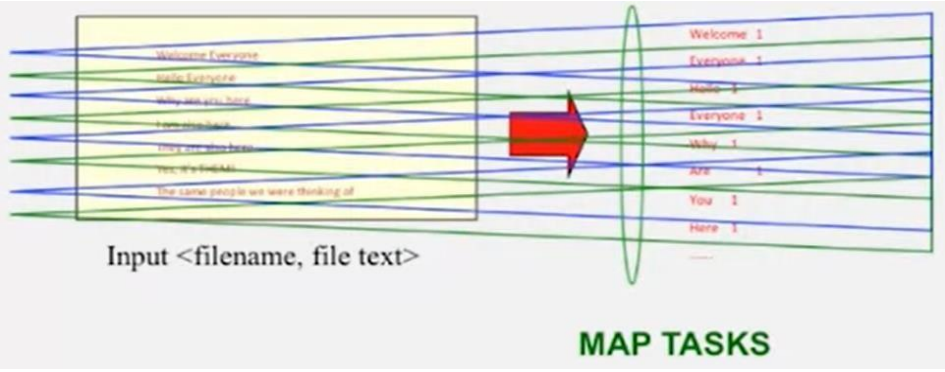


Fig. 3.17 Parallel Processing of Large Number of Records o Reduce processes and merges all intermediate values associated per key

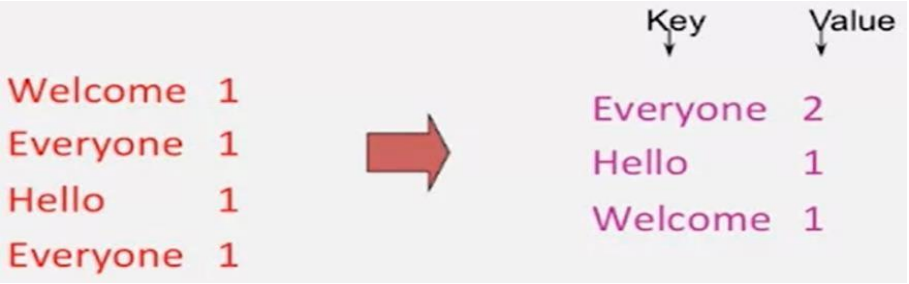


Fig. 3.18 Reduce Process

- Each key assigned to one Reduce
- Parallelly processes and merges all intermediate values by partitioning keys

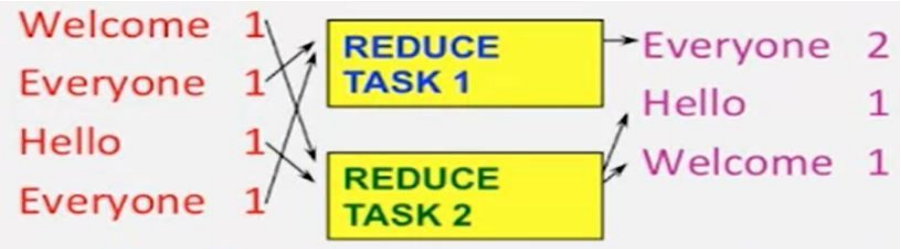


Fig. 3.19 Merge the Intermediate Values

- Popular: Hash partitioning, i.e., key is assigned to reduce # - hash(key)%number of reduce servers

Hadoop Code – Map

```
public static class MapClass extends MapReduceBase
            implements Mapper<LongWritable, Text, Text,
              IntWritable> {
  private final static IntWritable one =
    new IntWritable(1);
  private Text word = new Text();

  public void map( LongWritable key, Text value,
            OutputCollector<Text, IntWritable> output,
            Reporter reporter)
    throws IOException {
    String line = value.toString();
    StringTokenizer itr = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
    }
  }
} // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
```

Fig. 3.20 Hadoop Code - Map

Hadoop Code – Reduce

```
public static class ReduceClass 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 sum = 0;
            while (values.hasNext()) {
              sum += values.next().get();
            }
            output.collect(key, new IntWritable(sum));
  }
} // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
```

Fig. 3.21 Hadoop Code - Reduce

Hadoop Code – Driver

```
// Tells Hadoop how to run your Map-Reduce job
public void run (String inputPath, String outputPath)
            throws Exception {
  // The job. WordCount contains MapClass and Reduce.
  JobConf conf = new JobConf(WordCount.class);
  conf.setJobName("mywordcount");
  // The keys are words
  (strings) conf.setOutputKeyClass(Text.class);
  // The values are counts (ints)
  conf.setOutputValueClass(IntWritable.class);
  conf.setMapperClass(MapClass.class);
  conf.setReducerClass(ReduceClass.class);
  FileInputFormat.addInputPath(
            conf, newPath(inputPath));
  FileOutputFormat.setOutputPath(
            conf, new Path(outputPath));
  JobClient.runJob(conf);
} // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
```

Fig. 3.22 Hadoop Code - Driver

MAPREDUCE WITH COMBINER AND PARTITIONER

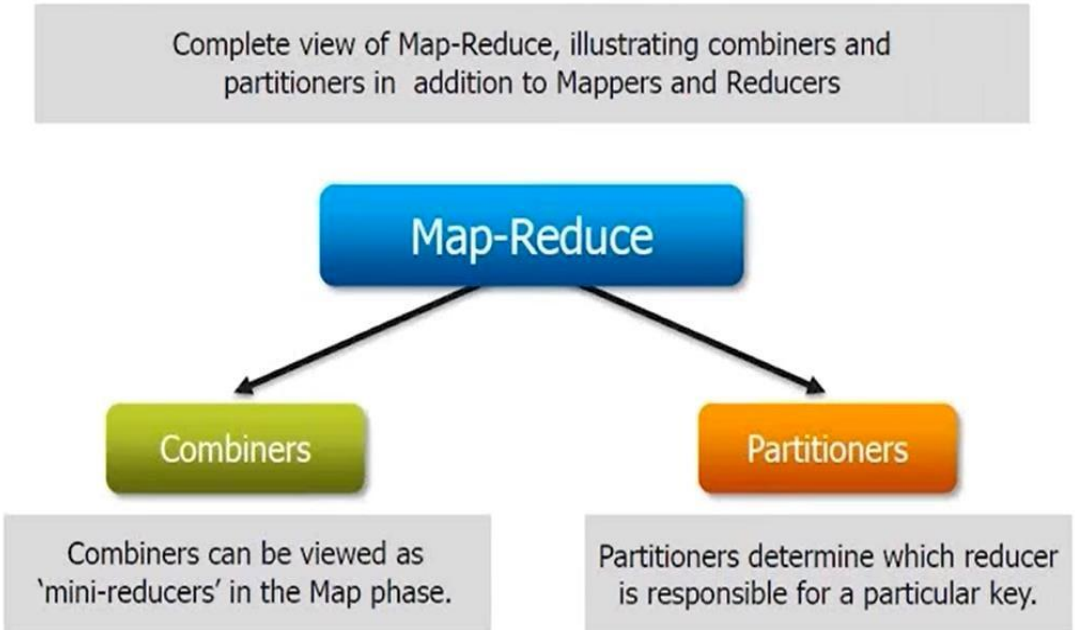


Fig. 3.23 Combiner and Partitioners

MAPREDUCE COMBINERS

- A Combiner, also known as a semi-reducer, is an optional class that operates by accepting the inputs from the Map class and thereafter passing the output key-value pairs to the Reducer class.

MapReduce with and without Combiner

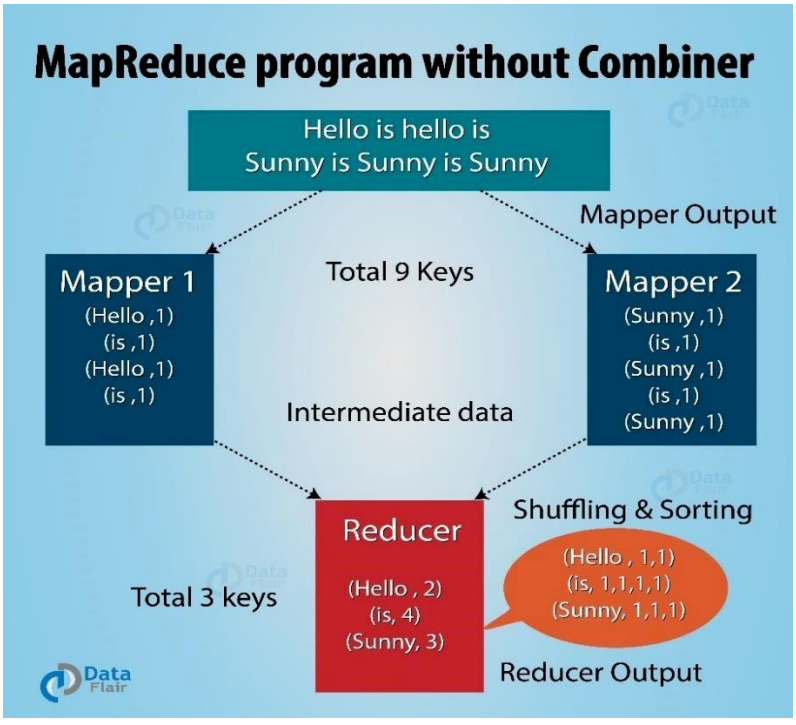


Fig. 3.24 MapReduce without Combiner

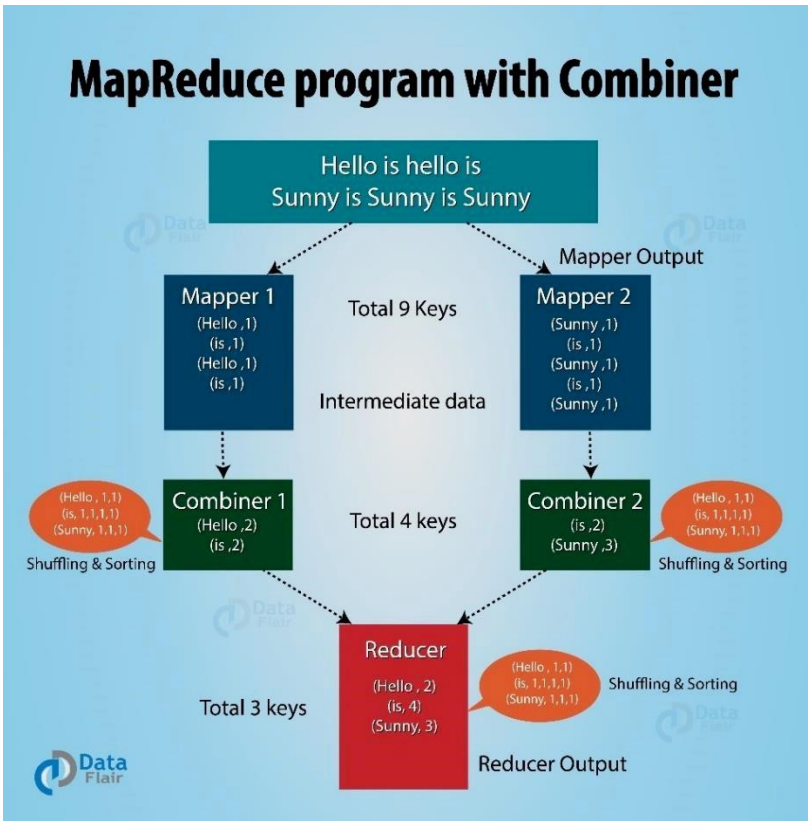


Fig. 3.25 MapReduce with Combiner

- The main function of a Combiner is to summarize the map output records with the same key. The output key-value collection of the combiner will be sent over the network to the actual Reducer task as input.

Combiner

- The Combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Map and Reduce. Usually, the output of the map task is large and the data transferred to the reduce task is high.
- The following MapReduce task diagram shows the COMBINER PHASE.

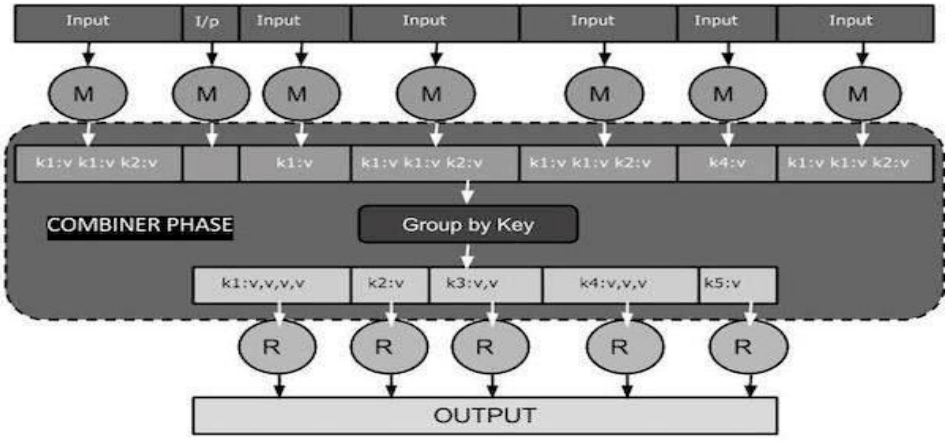


Fig. 3.26 Combiner Phase

- Here is a brief summary on how MapReduce Combiner works − ○ A combiner does not have a predefined interface and it must implement the Reducer interface's reduce method.
    - A combiner operates on each map output key. It must have the same output key-value types as the Reducer class.
    - A combiner can produce summary information from a large dataset because it replaces the original Map output.
- Although, Combiner is optional yet it helps segregating data into multiple groups for Reduce phase, which makes it easier to process.

MapReduce Combiner Implementation

- The following example provides a theoretical idea about combiners. Let us assume we have the following input text file named input.txt for MapReduce.

<1, What do you mean by Object>

<2, What do you know about Java>

<3, What is Java Virtual Machine>

<4, How Java enabled High Performance>

- The important phases of the MapReduce program with Combiner are discussed below.

Record Reader

- This is the first phase of MapReduce where the Record Reader reads every line from the input text file as text and yields output as key-value pairs.
- Input − Line by line text from the input file.
- Output − Forms the key-value pairs. The following is the set of expected key-value pairs.

<1, What do you mean by Object>

<2, What do you know about Java>

<3, What is Java Virtual Machine>

<4, How Java enabled High Performance>

Map Phase

- The Map phase takes input from the Record Reader, processes it, and produces the

- output as another set of key-value pairs.

  Input − The following key-value pair is the input taken from the Record Reader.

  <1, What do you mean by Object>

  <2, What do you know about Java>

  <3, What is Java Virtual Machine>

  <4, How Java enabled High Performance>

- The Map phase reads each key-value pair, divides each word from the value using StringTokenizer, treats each word as key and the count of that word as value.
- The following code snippet shows the Mapper class and the map function.

```
public static class TokenizerMapper extends Mapper<Object,
        Text, Text, IntWritable>
{
 private final static IntWritable one = new IntWritable(1);  private
Text word = new Text();
 public void map(Object key, Text value, Context context) throws IOException,
InterruptedException
 {
  StringTokenizer itr = new StringTokenizer(value.toString());
while (itr.hasMoreTokens())
  {
word.set(itr.nextToken());  context.write(word,
one);
  }
 }
```

- }
- Output − The expected output is as follows –

```
<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>
<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>
<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>
<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>
```

Combiner Phase

The Combiner phase takes each key-value pair from the Map phase, processes it, and produces the output as key-value collection pairs.

-

```
<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>
<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>
<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>
<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>
```

- The Combiner phase reads each key-value pair, combines the common words as key and Input − The following key-value pair is the input taken from the Map phase. values as collection.
- Usually, the code and operation for a Combiner is similar to that of a Reducer.
- Following is the code snippet for Mapper, Combiner and Reducer class declaration.

  job.setMapperClass(TokenizerMapper.class);

  job.setCombinerClass(IntSumReducer.class);

  job.setReducerClass(IntSumReducer.class);

- Output − The expected output is as follows –

```
<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,1,1,1>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>
```

Reducer Phase

- The Reducer phase takes each key-value collection pair from the Combiner phase, processes it, and passes the output as key-value pairs.

-
-
-

```
<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,1,1,1>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>
```

- The Reducer phase reads each key -value pair. Following is the code snippet for the Combiner.
  Note that the Combiner functionality is same as the Reducer.

  Input − The following key-value pair is the input taken from the Combiner phase.

```
public static class IntSumReducer extends
Reducer<Text,IntWritable,Text,IntWritable>
{
private IntWritable result = new IntWritable();
```

```
public void reduce(Text key, Iterable<IntWritable> values, Context context)throws
IOException, InterruptedException
{
    int sum = 0;
    for (IntWritableval : values)
    {
     sum += val.get();
} result.set(sum);
context.write(key, result);
  }
}
```

- Output − The expected output from the Reducer phase is as follows –

```
<What,3> <do,2> <you,2> <mean,1> <by,1> <Object,1>
<know,1> <about,1> <Java,3>
<is,1> <Virtual,1> <Machine,1>
<How,1> <enabled,1> <High,1> <Performance,1>
```

Record Writer

- This is the last phase of MapReduce where the Record Writer writes every key-value pair from the Reducer phase and sends the output as text.
- Input − Each key-value pair from the Reducer phase along with the Output format.
- Output − It gives you the key-value pairs in text format. Following is the expected output.

```
What            3
do              2
you             2
mean            1
by              1
Object          1
know            1
about           1
Java            3
is              1
Virtual         1
Machine         1
How             1
enabled         1
High            1
Performance     1
```

MAPREDUCE –PARTITIONER

- A partitioner works like a condition in processing an input dataset.
- The partition phase takes place after the Map phase and before the Reduce phase.
- The number of partitioners is equal to the number of reducers.
- That means a partitioner will divide the data according to the number of reducers.
- Therefore, the data passed from a single partitioner is processed by a single Reducer.

Partitioner

- A partitioner partitions the key-value pairs of intermediate Map-outputs.
- It partitions the data using a user-defined condition, which works like a hash function.
- The total number of partitions is same as the number of Reducer tasks for the job.
- Let us take an example to understand how the partitioner works.

MapReduce Partitioner Implementation

- For the sake of convenience, let us assume we have a small table called Employee with the following data.
- We will use this sample data as our input dataset to demonstrate how the partitioner works.

Table 3.1 Employee Dataset

Id	Name	Age	Gender	Salary
1201	gopal	45	Male	50,000

1202	manisha	40	Female	50,000
1203	khalil	34	Male	30,000
1204	prasanth	30	Male	30,000
1205	kiran	20	Male	40,000
1206	laxmi	25	Female	35,000
1207	bhavya	20	Female	15,000
1208	reshma	19	Female	15,000
1209	kranthi	22	Male	22,000
1210	Satish	24	Male	25,000
1211	Krishna	25	Male	25,000
1212	Arshad	28	Male	20,000
1213	lavanya	18	Female	8,000

- We have to write an application to process the input dataset to find the highest salaried employee by gender in different age groups (for example, below 20, between 21 to 30, above 30).
- The above data is saved as input.txt in the "/home/hadoop/hadoopPartitioner" directory and given as input.
- Based on the given input, following is the algorithmic explanation of the program.
- **Map Tasks**
    - The map task accepts the key-value pairs as input while we have the text data in a text file. The input for this map task is as follows −
    - **Input** − The key would be a pattern such as "any special key + filename + line number" (example: key = @input1) and the value would be the data in that line (example: value = 1201 \t gopal \t 45 \t Male \t 50000).
    - **Method** − The operation of this map task is as follows –

- Read the **value** (record data), which comes as input value from the argument list in a string.
- Using the split function, separate the gender and store in a string variable.

  String [] str = value.toString().split("\t", -3);

  String gender=str[3];
- Send the gender information and the record data **value** as output keyvalue pair from the map task to the **partition task** context.write (new Text(gender), new Text(value));
- Repeat all the above steps for all the records in the text file.

o **Output** − You will get the gender data and the record data value as key-value pairs.

- **Partitioner Task**

o The partitioner task accepts the key-value pairs from the map task as its input. Partition implies dividing the data into segments. According to the given conditional criteria of partitions, the input key-value paired data can be divided into three parts based on the age criteria.

o **Input** − The whole data in a collection of key-value pairs.
- key = Gender field value in the record.
- value = Whole record data value of that gender. o

Method − The process of partition logic runs as follows.
- Read the age field value from the input key-value pair o

Check the age value with the following conditions.
- Age less than or equal to 20
- Age Greater than 20 and Less than or equal to 30.
- Age Greater than 30.

```
if(age<=20)
{      return
0;
}     else     if(age>20     &&
age<=30)
{
    return 1 % numReduceTasks;
} else {      return 2 %
numReduceTasks;
```

```
}
```

- **Output** − The whole data of key-value pairs are segmented into three collections of key-value pairs. The Reducer works individually on each collection.

- **Reduce Tasks**
  - The number of partitioner tasks is equal to the number of reducer tasks. Here we have three partitioner tasks and hence we have three Reducer tasks to be executed.
  - **Input** − The Reducer will execute three times with different collection of keyvalue pairs.
    - key = gender field value in the record.
    - value = the whole record data of that gender. ○ **Method** −

The following logic will be applied on each collection.

- Read the Salary field value of each record.
- String [] str = val.toString ().split("\t", -3); Note:
- str [4] have the salary field value.

- Check the salary with the max variable. If str[4] is the max salary, then assign str[4] to max, otherwise skip the step.

```
if(Integer.parseInt(str[4])>max)
{
   max=Integer.parseInt(str[4]);
}
```

○ Repeat Steps 1 and 2 for each key collection (Male & Female are the key collections). After executing these three steps, you will find one max salary from the Male key collection and one max salary from the Female key collection.

```
context.write(new Text(key), new IntWritable(max));
```

- **Output** − Finally, you will get a set of key-value pair data in three collections of different age groups. It contains the max salary from the Male collection and the max salary from the Female collection in each age group respectively. ○ After executing the Map, the Partitioner, and the Reduce tasks, the three collections of key-value pair data are stored in three different files as the output.
- All the three tasks are treated as MapReduce jobs. The following requirements and specifications of these jobs should be specified in the Configurations –

- Job name
- Input and Output formats of keys and values
- Individual classes for Map, Reduce, and Partitioner tasks

```
Configuration conf = getConf();
//Create Job Jobjob = new Job(conf, "topsal");
job.setJarByClass(PartitionerExample.class);
// File Input and Output paths FileInputFormat.setInputPaths(job, new
Path(arg[0]));
FileOutputFormat.setOutputPath(job,new Path(arg[1])); //Set
Mapper class and Output format for key-value pair.
job.setMapperClass(MapClass.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);
//set partitioner statement
job.setPartitionerClass(CaderPartitioner.class);
//Set Reducer class and Input/Output format for key-value pair.
job.setReducerClass(ReduceClass.class); //Number of Reducer
tasks. job.setNumReduceTasks(3);
//Input and Output format for data
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
job.setOutputKeyClass(Text.class); job.setOutputValueClass(Text.class);
```

○ I/P Mapper: Data set ○ O/P to Mapper

→ I/P to Partioner:

- Female → record
- Male →record ○ O/P

Partioner I/P to Reducer

- Age < 20 → Female → Record
- Age < 20 → Male → Record
- Age > 20 &&< 30 → Female → Record
- Age > 20 &&< 30 → Male → Record
- Age > → 30 → Female → Record

Age > 30 Male Record ○ O/P Reducer

- Max(Age < → 20 Female → Record)
- Max(Age < → 20 Male → Record)
- Max(Age > 20 &&< 30 → Female → Record)
- Max(Age > 20 &&< 30 → Male → Record)
- Max(Age > → 30 Female → Record)
- Max(Age > 30 → Male → Record)

HADOOP INPUT FORMATS

Input Splits, Input Formats and Record Reader:

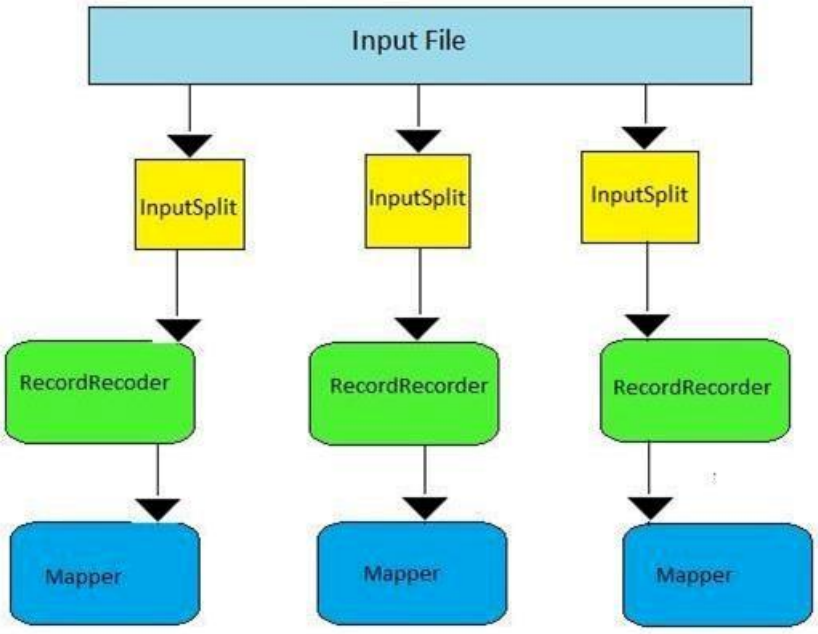


Fig. 3.27 Input Splits and Record Reader

- On job startup, each input file is broken into splits and each map processes a single split. Each split is further divided into records of key/value pairs which are processed by map tasks one record at a time.
- To get split details of an input file, Hadoop provides an **InputSplit class in org.apache.hadoop.mapreduce package** and its implementation is as follows.

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

Input Splits

- From the above two methods, programmer can get length of a split and storage locations.
- A good input split size is equal to the HDFS block size.
- But if the splits are too smaller than the default HDFS block size, then managing splits and creation of map tasks becomes an overhead than the job execution time.
- But these file splits need not be taken care by Mapreduce programmer because Hadoop provides **InputFormat class in org.apache.hadoop.mapreduce package** for the below two responsibilities.
    - To provide details on how to split an input file into the splits.
    - To create a **RecordReader** class that will generate the series of key/value pairs from a split.
- To meet these two requirements, Hadoop provides below implementation for InputFormat class with two methods.

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

Key value paring in Hadoop Mapreduce

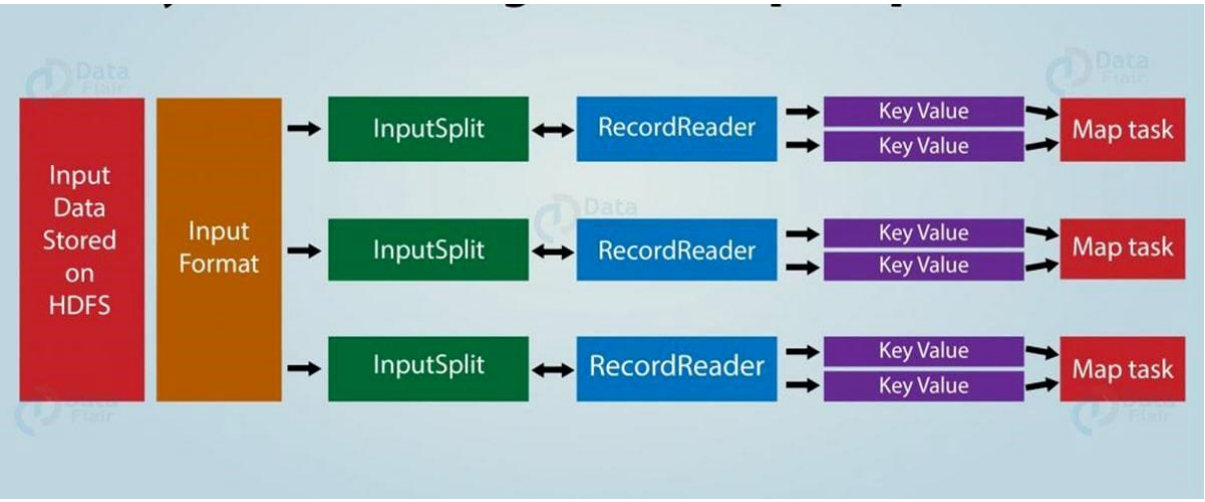


Fig. 3.28 Key Value Paring in Hadoop MapReduce

- Thus, Record reader creates key/value pairs from input splits and writes on to Context, which will be shared with Mapper class. Mapper class's run () method retrieves these key/value

pairs from context by calling getCurrentKey() and getCurrentValue() methods and passes onto map() method for further processing of the record.

- Mapper's run () method:

```
public void run (Context context) throws IOException, InterruptedException
{
 setup(context);
 while (context.nextKeyValue())
 {
 map(context.getCurrentKey(), context.getCurrentValue(), context);
 }
cleanup(context);
}
```

- Thus, finally key/value pairs from each input record are sent to map() task.

Built-in Hadoop Input Formats:

- Hadoop provided some built in InputFormat implementations in the org.apache.hadoop.mapreduce.lib.input package:
- FileInputFormat: Base class for all file-based InputFormat implementations.
- Some of the important sub classes of the FileInputFormat class are:
    - TextInputFormat
    - KeyValueTextInputFormat
    - FixedLengthInputFormat
    - NLineInputFormat
    - CombineFileInputFormat
    - MultiFileInputFormat
    - SequenceFileInputFormat
    - SequenceFileAsTextInputFormat
    - SequenceFileAsBinaryInputFormat
    - MultipleInputs
    - DBInputFormat

- **TextInputFormat :**

The default InputFormat class when no other class is specified. It treats the input files as text files.

- **KeyValueTextInputFormat :**

  An InputFormat for plain text files. Files are broken into lines. Each line is divided into key and value parts by a separator byte. If no such a byte exists, the key will be the entire line and value will be empty.

- **FixedLengthInputFormat :**

  An input format to read input files with fixed length records. These need not be text files and can be binary files. Users must configure the record length property by calling: ***FixedLengthInputFormat.setRecordLength(conf, recordLength);***

- **NLineInputFormat :**

  It splits **N** lines of input as one split which will be fed to a single map task. It can be used in applications, that splits the input file such that by default, one line is fed as a value to one map task, and key is the offset. i.e. (k,v) is (LongWritable, Text).

- **CombineFileInputFormat :**

  This input file format is suitable for processing huge number of small files. CombineFileInputFormat packs many small files into each split so that each mapper has more to process. Thus it can improve the efficiency of mapreduce job by making less number of map tasks to process huge number of small files.

- **MultiFileInputFormat :**

  An abstract InputFormat class that returns MultiFileSplit's in **getSplits()** method from the files under the input paths.

- **SequenceFileInputFormat :**

  Hadoop specific Binary file format for efficient file processing.

- **SequenceFileAsTextInputFormat :**

  SequenceFileAsTextInputFormat is a sub class of SequenceFileInputFormat. This class is similar to SequenceFileInputFormat, except it generates SequenceFileAsTextRecordReader which converts the input keys and values to their String forms by calling **toString()** method.

- **SequenceFileAsBinaryInputFormat :**

  SequenceFileAsBinaryInputFormat is another sub class of SequenceFileInputFormat. It is an input format for reading keys, values from Sequence Files in binary (raw) format.

- **MultipleInputs** :

This class supports MapReduce jobs that have **multiple input paths** with a **different InputFormat** and Mapper for each path.

- **DBInputFormat :**

  A InputFormat that reads input data from an SQL table. DBInputFormat emits LongWritables containing the record number as key and DBWritables as value.

  The SQL query, and input class can be using one of the two setInput() methods

Prevent Input File Splitting

- If we don't want files to be split, so that a single mapper can process each input file in its entirety, we can override the isSplitable() method of FileInputFormat class to return false.
- For example, here's a non splittableFileInputFormat implementation.

```
Import org.apache.hadoop.fs.Path;

Import org.apache.hadoop,.mapreduce.JobContext;

Import org.apache.hadoop.mapreduce.lib.input.FileInputFormat; public

class NonSplittableFileInputFormat extends FileInputFormat   {

@Override

protected Boolean isSplitable(JobContext context, Path file)    { return

false;

}
}
```

Input File Format Class Hierarchy

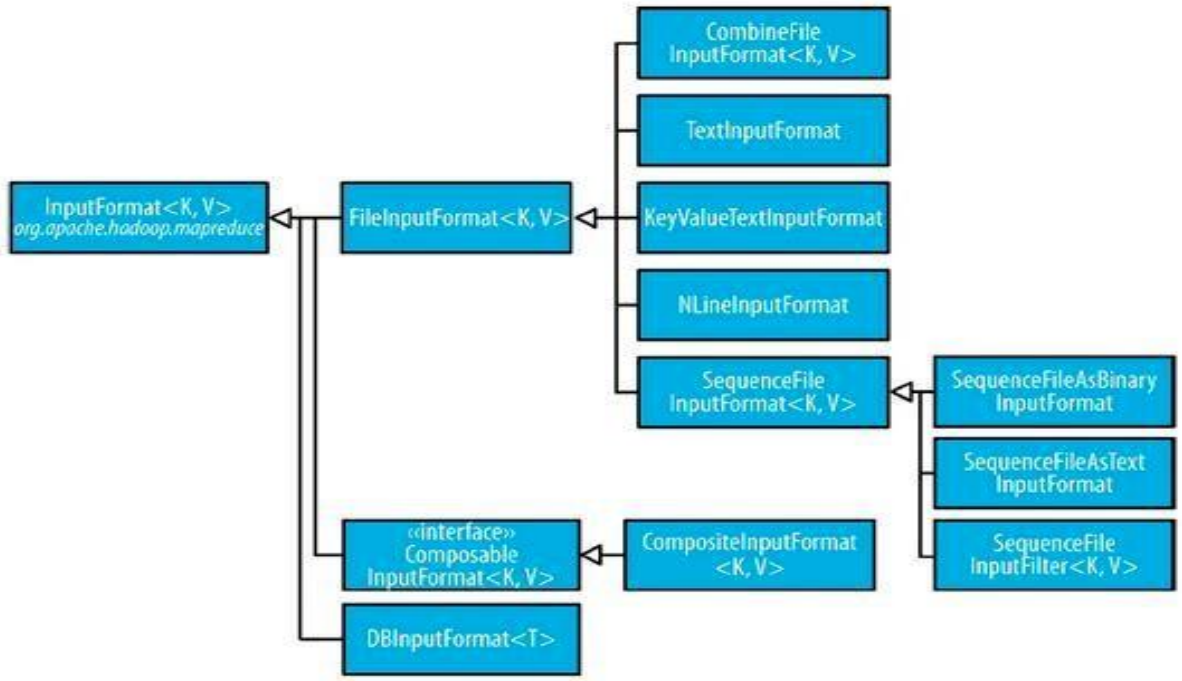


Fig. 3.29 Class Hierarchy of Input File Format

Hadoop Output Formats

- Hadoop provides output formats that corresponding to each input format. All hadoop output formats must implement the interface org.apache.hadoop.mapreduce.OutputFormat.
    - OutputFormat describes the output-specification for a Map-Reduce job. Based on Output specification, Mapreduce job checks that the output directory doesn't already exist.
    - OutputFormat provides the RecordWriter implementation to be used to write out the output files of the job.

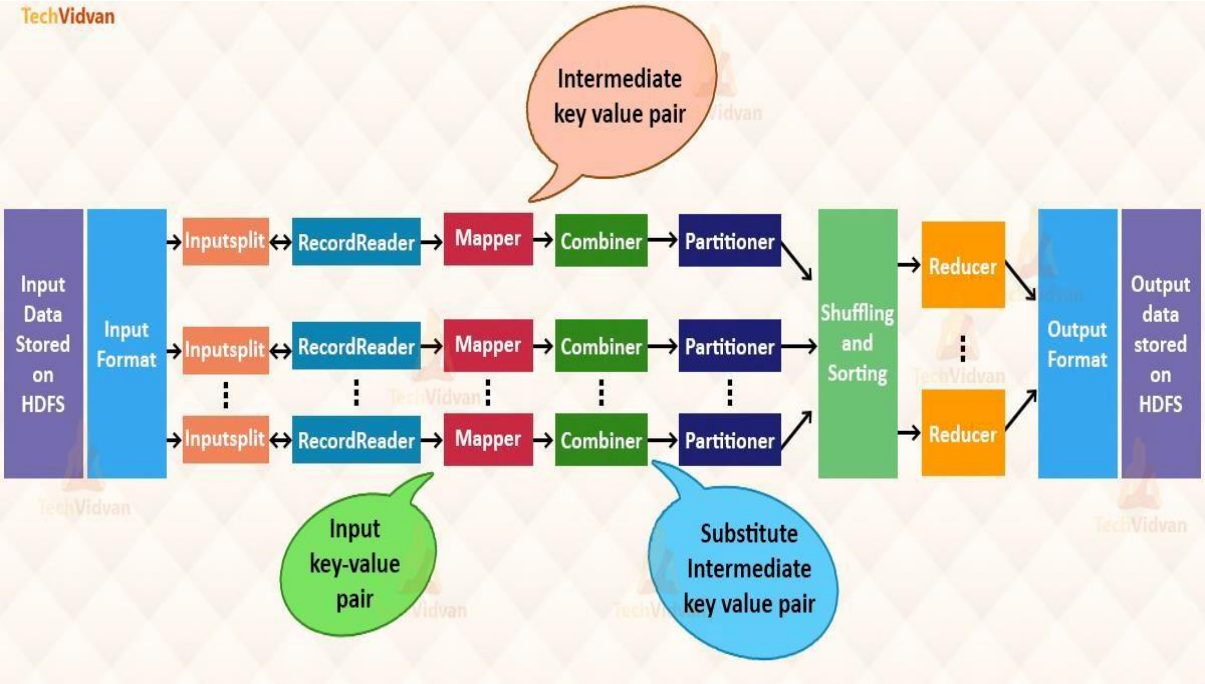


Fig. 3.30 Output File Format

- These two requirements of the OutputFormat are accomplished with below two methods in the interface.

```
public abstract void checkOutputSpecs(JobContext context) throws
IOException, InterruptedException
{
}
```

- This method checks that output directory doesn't exist already and throws an exception when it already exists, so that output is not overwritten.

```
public abstract RecordWriter<K,V> getRecordWriter(TaskAttemptContextcontext)throw
s IOException, InterruptedException
{
}
```

- This method Gets the RecordWriter for the given task.
- org.apache.hadoop.mapreduce.RecordWriter<K,V> class implementations are used to write the output <key, value> pairs to an output file.

Built-In Hadoop Output Formats

- Hadoop provided some built in InputFormat implementations in the org.apache.hadoop.mapreduce.lib.output package:
- FileOutputFormat ○ Base class for all file-based OutputFormat implementations. Some of the important sub classes of the FileOutputFormat class are:
- TextOutputFormat ○ The default output format provided by hadoop is TextOuputFormat and it writes records as lines of text. If file output format is not specified explicitly, then text files are created as output files.
    - Output Key-value pairs can be of any format because TextOutputFormat converts these into strings with toString() method. Output key-value pairs are tab delimited by default.
    - For reading these output text files as input, KeyValueTextInputFormat is best suitable, since it breaks input lines into key value pairs based on a separator character.
- SequenceFileOutputFormat
    - This output format class is useful to write out sequence files which is a best option when the output files need to be fed into another mapreduce jobs as input files, since these are compressed and compact.
- SequenceFileAsBinaryOutputFormat ○ SequenceFileAsBinaryOutputFormat is a direct subclass of SequenceFileOutputFormat and it is counterpart for SequenceFileAsBinaryInputFormat. It writes keys and values to Sequence Files in binary format.
- MapFileOutputFormat
    - It is also a direct subclass of FileOutputFormat and it is used to write output as Map files.
- MultipleOutputs
    - The MultipleOutputs class is used to write output data to multiple outputs. Below are the two main use cases of MultipleOutputs.
        - Job output can be written to additional outputs other than the default output. Each additional output, or named output, may be configured with its own OutputFormat, with its own key class and value class.

▪ Write data to different files provided by user ○ MultipleOutputs supports counters to count the number records written to each output name. But these are disabled by default.

Usage pattern for job submission:

```
Job job = new Job();
FileInputFormat.setInputPath(job, InPath);
FileOutputFormat.setOutputPath(job, OutPath);
job.setMapperClass(MultipleOutputMap.class);
job.setReducerClass(MultipleOutputReduce.class);
...
// Defines additional single text-based output 'text' for the job
MultipleOutputs.addNamedOutput(job, "text", TextOutputFormat.class, LongWritable.class, Text.class);
// Defines additional sequence-file based output 'sequence' for the job
MultipleOutputs.addNamedOutput(job, "seq", SequenceFileOutputFormat.class, LongWritable.class, Text.class);
...
job.waitForCompletion(true);
...
```

Usage in Reducer: MultipleOutputs

MultipleOutputFormat

- It is an abstract class which is extended by ***MultipleTextOutputFormat*** and ***MultipleSequenceFileOutputFormat***. This abstract class extends the **FileOutputFormat**,
- The main advantage of this format is the ability to write the output data to different output files. Instead of the default output file convention **part-m-xxxxx** or **part-rxxxxx** , output files can be written with names of our choice. Below are the three scenarios where output file names can be changed.
- If there is at least one reducer in the mapreduce job, output can be written to different files depending on the actual keys.

- If the mapreduce job is a map only job, then job can use the output file name that is either a part of the input file name or any name derived from it.
- If it is a map only job, job can use the output file name that depends on both the keys and the input file name.
- **MultipleSequenceFileOutputFormat**

  This is also a sub class of **MultipleOutputFormat** class. Using this format, the output data can be written to different output files in Sequence file format.
- **MultipleTextOutputFormat**

  This is also a sub class of **MultipleOutputFormat** class. Using this format, the output data can be written to different output files in Text output format.
- **LazyOutputFormat**

  By Default, **FileOutputFormat** creates the output files even if a single output record is not emitted from reducers. Thus,Mapreduce jobs create empty output files some times. This can be avoided with **LazyOutputFormat** in which output files are created only when the first output is emitted from the reducers. It is used in conjunction with **org.apache.hadoop.mapreduce.lib.output.MultipleOutputs** to recreate the behavior of *org.apache.hadoop.mapred.lib.MultipleTextOutputFormat* of the old Hadoop API.
- **DBOutputFormat**

  This output format is used to write an output into SQL tables using mapreduce jobs. **DBOutputFormat** accepts <key,value> pairs, where key has a type extending **DBWritable**. Returned DBOutputFormat. **DBRecordWriter** writes only the key to the database with a batch SQL query.

- **NullOutputFormat**

  NullOutputFormat writes nothing to output directory. It Consumes all outputs and put them in /dev/null. We can suppress the key or value in the output using a NullWritable type. Both can be suppressed if the output file format is NullOutputFormat.

Hadoop Output Format Hierarchy

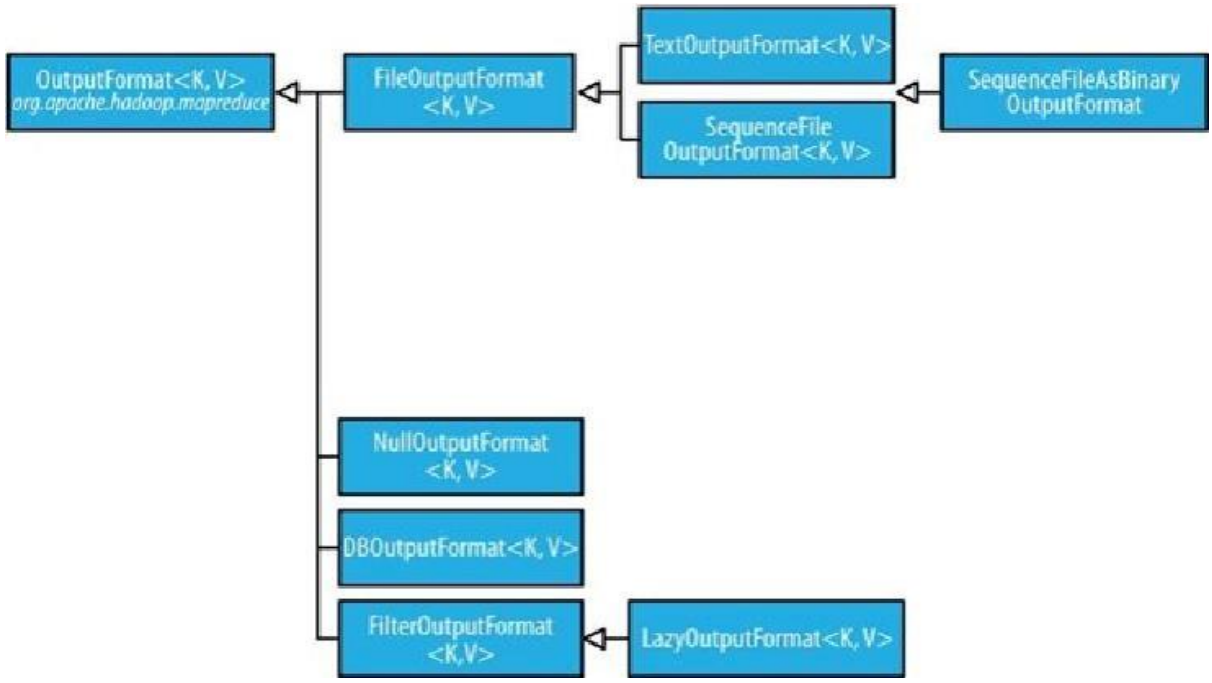


Fig. 3.31 Output File Format Hierarchy

Input / Output Formats

Table 3.2 Input Format vs. Output Format

org.apache.hadoop.mapreduce.lib.input	org.apache.hadoop.mapreduce.OutputFormat.
InputSplit class in org.apache.hadoop.mapreduce getLength() getLocations()	
InputFormat provides the Record Reader class that will generate the series of key/value pairs from a split.	OutputFormat provides theRecordWriter implementation to be used towrite out the output files of the job.
Mapper class's run () method retrieves these key/value pairs from context by callinggetCurrentKey() and getCurrentValue() methods and passes onto map() method for further processing of the record.	checkOutputSpecs() - Based on Output specification, Mapreduce job checks that the output directory RecordWriter<K,V> - used to write out the output files of the job.
TextInputFormat The default InputFormat class	TextOutputFormat It writes records as lines of text.

KeyValueTextInputFormat An InputFormat for plain text files. Files are broken into lines. Each line is divided into key and value parts by a separator byte	MultipleOutputFormat class. Using this format the output data can be written to different output files in Text output format.
FixedLengthInputFormat An input format to read input files with	
fixed length records.	
NLineInputFormat It splits N lines of input as one split which will be fed to a single map task.	NullOutputFormat NullOutputFormat writes nothing to output directory.
CombineFileInputFormat This input file format is suitable for processing huge number of small files. CombineFileInputFormat packs many small files into each split so that each mapper has more to process.	MapFileOutputFormat It is used to write output as Map files.
SequenceFileInputFormat Hadoop specific Binary file format for efficient file processing	SequenceFileOutputFormat This output format class is useful to write out sequence files
SequenceFileAsTextInputFormat which converts the input keys and values to their String forms by calling toString() method	MultipleTextOutputFormat This is also a sub class of MultipleOutputFormat class. Using this format the output data can be written to different output files in Text output format.
SequenceFileAsBinaryInputFormat: It is an input format for reading keys, values from Sequence Files in binary (raw) format	SequenceFileAsBinaryOutputFormat It writes keys and values to Sequence Files in binary format.
MultipleInputs This class supports MapReduce jobs that have multiple input paths with a different InputFormat	MultipleOutputs The MultipleOutputs class is used to write output data to multiple outputs
DBInputFormat A InputFormat that reads input data from an SQL table	DBOutputFormat This output format is used to write an output into SQL tables using mapreduce jobs. DBRecordWriter writes only the key to the database with a batch SQL query

HADOOP COUNTERS

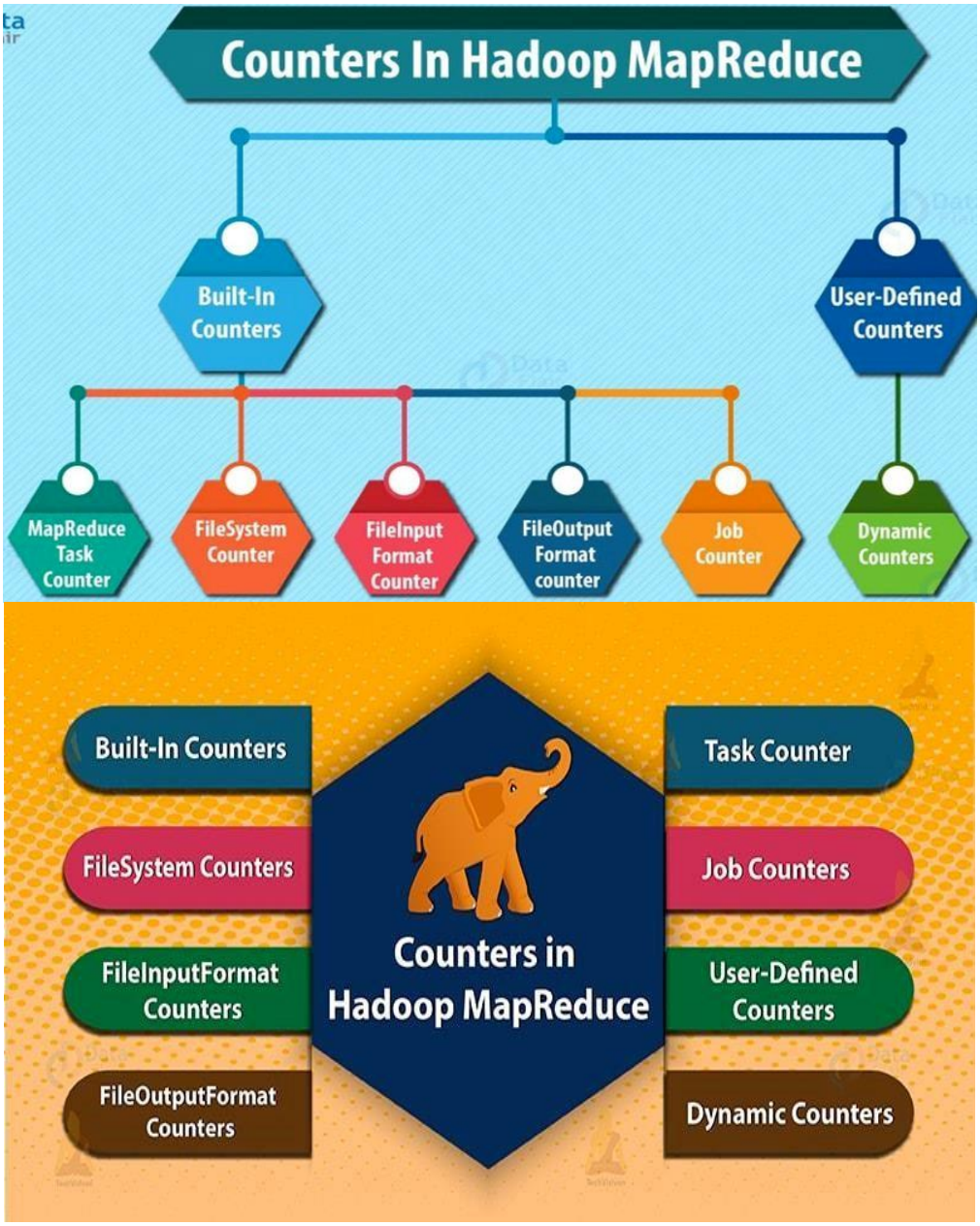


Fig. 3.32 Hadoop Counters

- MapReduce is the core component of Hadoop which provides data processing.
- MapReduce works by breaking the processing into two phases; Map phase and Reduce phase.

Hadoop MapReduce Framework

- The map is the first phase of processing, where we specify all the complex logic/business rules/costly code, whereas the Reduce phase is the second phase of processing, where we specify light-weight processing like aggregation/ summation.
- In Hadoop, MapReduce Framework has certain elements such as Counters, Combiners, and Partitioners, which play a key role in improving the performance of data processing.

What are Hadoop Counters?

- Hadoop Counters provides a way to measure the progress or the number of operations that occur within map/reduce job.
- Counters in Hadoop MapReduce are a useful channel for gathering statistics about the MapReduce job: for quality control or for application-level.
- They are also useful for problem diagnosis.
- Counters represent Hadoop global counters, defined either by the MapReduce framework or applications.
- Each Hadoop counter is named by an "Enum" and has a long for the value.
- Counters are bunched into groups, each comprising of counters from a particular Enum class.
- Hadoop Counters validate that:
    - The correct number of bytes was read and written. o The correct number of tasks was launched and successfully ran.
    - The amount of CPU and memory consumed is appropriate for our job and cluster nodes.
- There are basically 2 types of MapReduce Counters:
    - Built-In Counters in MapReduce
    - User-Defined Counters/Custom counters in MapReduce

Built-In Counters in MapReduce

- Hadoop maintains some built-in Hadoop counters for every job and these report various metrics, like, there are counters for the number of bytes and records, which allow us to confirm that the expected amount of input is consumed and the expected amount of output is produced.
- Hadoop Counters are divided into groups and there are several groups for the built-in counters.

- Each group either contains task counters (which are updated as task progress) or job counter (which are updated as a job progress).
- There are several groups for the Hadoop built-in Counters:
    a) MapReduce Task Counter in Hadoop
    b) FileSystem Counters
    c) FileInputFormat Counters in Hadoop
    d) FileOutputFormat counters in MapReduce
    e) MapReduce Job Counters

MapReduce Task Counter in Hadoop

- Hadoop Task counter collects specific information (like number of records read and written) about tasks during its execution time.
- For example, the MAP_INPUT_RECORDS counter is the Task Counter which counts the input records read by each map task.
- Hadoop Task counters are maintained by each task attempt and periodically sent to the application master so they can be globally combined.

FileSystem Counters

- Hadoop FileSystem Counters in Hadoop MapReduce gather information like a number of bytes read and written by the file system. Below are the name and description of the file system counters:
    - FileSystem bytes read– The number of bytes read by the filesystem by map and reduce tasks.
    - FileSystem bytes written– The number of bytes written to the filesystem by map and reduce tasks.

FileInputFormat Counters in Hadoop

- FileInputFormat Counters in Hadoop MapReduce gather information of a number of bytes read by map tasks via FileInputFormat.

FileOutputFormat counters in MapReduce

- FileOutputFormat counters in Hadoop MapReduce gathers information of a number of bytes written by map tasks (for map-only jobs) or reduce tasks via FileOutputFormat.

MapReduce Job Counters

- MapReduce Job counter measures the job-level statistics, not values that change while a task is running.
- For example, TOTAL_LAUNCHED_MAPS, count the number of map tasks that were launched over the course of a job (including tasks that failed).
- Application master maintains MapReduce Job counters, so these Hadoop Counters don't need to be sent across the network, unlike all other counters, including userdefined ones.

User-Defined Counters/Custom Counters in Hadoop MapReduce

- In addition to MapReduce built-in counters, MapReduce allows user code to define a set of counters, which are then incremented as desired in the mapper or reducer.
- For example, in Java, 'enum' is used to define counters.
- A job may define an arbitrary number of 'enums', each with an arbitrary number of fields.
- The name of the enum is the group name, and the enum's fields are the counter names.

Dynamic Counters in Hadoop MapReduce

- Java enum's fields are defined at compile time, so we cannot create new counters in Hadoop MapReduce at runtime using enums.
- To do so, we use dynamic counters in Hadoop MapReduce, one that is not defined at compile time using java enum.

MapReduce Counters: Conclusion

- Counters check whether the correct number of bytes is read or written, the correct number of tasks are launched and successfully run.
- Hence, Hadoop maintains built-in counters and user-defined counters to measure the progress that occurs within MapReduce job.

Custom Writables in Hadoop

HOW TO CREATE A CUSTOM WRITABLE FOR HADOOP

- We have gone through other Hadoop MapReduce examples, we will have noticed the use of "Writable" data types such as LongWritable, IntWritable, Text, etc…

- All values in used in Hadoop MapReduce must implement the Writable interface.
- Although we can do a lot with the primitive Writables already available with Hadoop, there are often times when we want to transmit a variety of data and/or data types from Mapper to Reducer.
- Sometimes it is possible to convert all these data into strings and concatenate them to result in a single key or single value. However, this can get very messy, and is not recommended.
- Implementing Writable requires implementing two methods, readFields(DataInput in) and write(DataOutput out).
- Writables that are used as keys in MapReduce jobs must also implement Comparable (or simply WritableComparable).
- Overriding the toString() method is not necessary, but can be very helpful when storing your output data as text in HDFS.
- Below is an example of a custom Writable that is used to store both gender and login information.
- An example of using this might be to calculate login statistics based on gender.
- Notice that this custom class, GenderLoginWritable, does not implement Comparable or WritableComparable, so it can only be used as a value in the MapReduce framework, not as a key.

```
package
com.bigdatums.hadoop.mapreduce;  import
org.apache.hadoop.io.IntWritable; import
org.apache.hadoop.io.Writable;  import
java.io.DataInput; import
java.io.DataOutput; import
java.io.IOException;

public class GenderLoginWritable implements Writable {
private IntWritable male;      private IntWritable female;
private     IntWritablemaleLogins;                  private
IntWritablefemaleLogins;
```

```
   public GenderLoginWritable() {
male = new IntWritable(0);         female =
new IntWritable(0);         maleLogins = new
IntWritable(0);         femaleLogins = new
IntWritable(0);    } public
GenderLoginWritable(IntWritable male,
IntWritablemaleLogins, IntWritable female,
IntWritablefemaleLogins) {         this.male =
male;         this.female = female;
this.maleLogins = maleLogins;
this.femaleLogins = femaleLogins;
   }
public      IntWritablegetMale()      {
return male;
   }

   public      IntWritablegetFemale()      {
return female;
   }

   public      IntWritablegetMaleLogins()      {
return maleLogins;
   }

   public      IntWritablegetFemaleLogins()      {
return femaleLogins;
   }

   public     void     setMale(IntWritable     male)     {
this.male = male;
   }
```

```
   public void setFemale(IntWritable female) {
this.female = female;
   }

   public void setMaleLogins(IntWritablemaleLogins) {
this.maleLogins = maleLogins;
   }
 public void setFemaleLogins(IntWritablefemalelogins) {
this.femaleLogins = femalelogins;
   }

   public void readFields(DataInput in) throws IOException
{      male.readFields(in);      female.readFields(in);
maleLogins.readFields(in);
femaleLogins.readFields(in);
   }

public void write(DataOutput out) throws IOException
{      male.write(out);      female.write(out);
maleLogins.write(out);      femaleLogins.write(out);
   }

@Override
public String toString() {
return male.toString() + "\t" + maleLogins.toString() + "\t" + female.toString() + "\t" +
femaleLogins.toString();
}
}
```

Unit Testing Framework

- Hadoop MapReduce jobs have a unique code architecture that follows a specific template with specific constructs.

- This architecture raises interesting issues when doing test-driven development (TDD) and writing unit tests.
- This is a real-world example using MRUnit, Mockito, and PowerMock. It is
    - using MRUnit to write JUnit tests for hadoop MR applications,
    - using PowerMock& Mockito to mock static methods,
    - mocking-out business-logic contained in another class,
    - verifying that mocked-out business logic was called (or not)
    - testing counters,
    - testing statements in a log4j conditional block, and
    - handling exceptions in tests.
- With MRUnit, you can craft test input, push it through your mapper and/or reducer, and verify it's output all in a JUnit test. As do other JUnit tests, this allows you to debug your code using the JUnit test as a driver.
- A map/reduce pair can be tested using MRUnit'sMapReduceDriver. A combiner can be tested using MapReduceDriver as well. A PipelineMapReduceDriver allows you to test a workflow of map/reduce jobs. Currently, partitioners do not have a test driver under MRUnit.
- MRUnit allows you to do TDD and write light-weight unit tests which accommodate Hadoop's specific architecture and constructs.
- In the following example, we're processing road surface data used to create maps. The input contains both linear surfaces (describing a stretch of the road) and intersections (describing a road intersection). This mapper takes a collection of these mixed surfaces as input, discards anything that isn't a linear road surface, i.e., intersections, and then processes each road surface and writes it out to HDFS. We want to keep count and eventually print out how many non-road surfaces are input. For debugging purposes, we will additionally print out how many road surfaces were processed.

```
public class MergeAndSplineMapper extends Mapper<LongWritable, BytesWritable,
LongWritable, BytesWritable> {
private static Logger LOG = Logger.getLogger(MergeAndSplineMapper.class);
enumSurfaceCounters {
ROADS, NONLINEARS, UNKNOWN
```

```
}
@Override
public void map(LongWritable key, BytesWritable value, Context context) throws
IOException, InterruptedException {
// A list of mixed surface types
LinkSurfaceMaplsm = (LinkSurfaceMap) BytesConverter.bytesToObject(value.getBytes());
List<RoadSurface>mixedSurfaces = lsm.toSurfaceList();
for (RoadSurfacesurface :mixedSurfaces)  {
Long surfaceId = surface.getNumericId();
Enums.SurfaceTypesurfaceType = surface.getSurfaceType();
if ( surfaceType.equals(SurfaceType.INTERSECTION)  )  {
// Ignore non-linear surfaces.
context.getCounter(SurfaceCounters.NONLINEARS).increment(1);
continue; }
else if ( !surfaceType.equals(SurfaceType.ROAD) ) {
// Ignore anything that wasn't an INTERSECTION or ROAD, ie any future additions.
context.getCounter(SurfaceCounters.UNKNOWN).increment(1); continue;
}
PopulatorPreprocessor.processLinearSurface(surface); //
Write out the processed linear surface.
lsm.setSurface(surface);
context.write(new LongWritable(surfaceId), new
BytesWritable(BytesConverter.objectToBytes(lsm)));  if
(LOG.isDebugEnabled()) {
context.getCounter(SurfaceCounters.ROADS).increment(1);
}
}
} }
```

We've written the following unit test for our class using MRUnit, Mockito, and PowerMock.

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(PopulatorPreprocessor.class)  public
classMergeAndSplineMapperTest {
privateMapDriver<LongWritable, BytesWritable, LongWritable, BytesWritable>mapDriver;
```

```
@Before
public voidsetUp() {
MergeAndSplineMapper mapper = newMergeAndSplineMapper();
mapDriver = newMapDriver<LongWritable, BytesWritable, LongWritable,
BytesWritable>(); mapDriver.setMapper(mapper);
}
@Test public voidtestMap_INTERSECTION()
throwsIOException {
LinkSurfaceMaplsm = newLinkSurfaceMap();
RoadSurfacers = newRoadSurface(Enums.RoadType.INTERSECTION);
byte[] lsmBytes = append(lsm, rs);
PowerMockito.mockStatic(PopulatorPreprocessor.class);
mapDriver.withInput(newLongWritable(1234567), newBytesWritable(lsmBytes));
mapDriver.runTest();
Assert.assertEquals("ROADS count incorrect.", 0,
mapDriver.getCounters().findCounter(SurfaceCounters.ROADS).getValue());
Assert.assertEquals("NONLINEARS count incorrect.", 1,
mapDriver.getCounters().findCounter(SurfaceCounters.NONLINEARS).getValue());
Assert.assertEquals("UNKNOWN count incorrect.", 0, mapDriver.getCounters().
findCounter(SurfaceCounters.UNKNOWN).getValue());
PowerMockito.verifyStatic(Mockito.never());
PopulatorPreprocessor.processLinearSurface(rs);
}
@Test public voidtestMap_ROAD()
throwsIOException {
LinkSurfaceMaplsm = newLinkSurfaceMap();
RoadSurfacers = newRoadSurface(Enums.RoadType.ROAD);
byte[] lsmBytes = append(lsm, rs);
// save logging level since we are modifying it.
Level originalLevel = Logger.getRootLogger().getLevel();
Logger.getRootLogger().setLevel(Level.DEBUG);
PowerMockito.mockStatic(PopulatorPreprocessor.class);
mapDriver.withInput(newLongWritable(1234567), newBytesWritable(lsmBytes));
```

```
mapDriver.withOutput(newLongWritable(1000000), newBytesWritable(lsmBytes));
mapDriver.runTest();

Assert.assertEquals("ROADS count incorrect.", 1,
mapDriver.getCounters().findCounter(SurfaceCounters.ROADS).getValue());
Assert.assertEquals("NONLINEARS count incorrect.", 0,
mapDriver.getCounters().findCounter(SurfaceCounters.NONLINEARS).getValue());
Assert.assertEquals("UNKNOWN count incorrect.", 0,
mapDriver.getCounters().findCounter(SurfaceCounters.UNKNOWN).getValue());
PowerMockito.verifyStatic(Mockito.times(1));
PopulatorPreprocessor.processLinearSurface(rs);
// set logging level back to it's original state so as not to affect other tests
Logger.getRootLogger().setLevel(originalLevel);
}
```

Let's take a look at the first test, testMap_INTERSECTION().

testMap_INTERSECTION

Our objective is to verify

1. SurfaceCounters.*NONLINEARS* is incremented.
2. The for-loop continues,i.e., PopulatorPreprocessor.*processLinearSurface*(surface) is **never** called.
3. SurfaceCounters.*ROADS* and SurfaceCounters.*UNKNOWN* are **not** incremented. Since this is a mapper, we start by defining and initializing a mapper driver. Note that the four type-parameters defined for the MapDriver must match our class under test, i.e., MergeAndSplineMapper.

```
private MapDriver<LongWritable, BytesWritable, LongWritable, BytesWritable>mapDriver;

    @Before
    public void setUp() {
MergeAndSplineMapper mapper = new MergeAndSplineMapper();
mapDriver = new MapDriver<LongWritable, BytesWritable, LongWritable,
BytesWritable>(); mapDriver.setMapper(mapper);
    }
```

Throwing IOException on the unit test method signature

The mapper could throw an IOException. In JUnit tests you can handle exceptions thrown by the calling code by catching them or throwing them. Keep in mind that we are not specifically testing exceptions. I prefer not to catch the exception and have the unit test method throw it. If the unit test method encounters the exception, the test will fail. Which is what we want. Trying to catch exceptions in unit tests, when you are not specifically testing exception handling, can lead to unnecessary clutter, logic, maintenance, when you can simply throw the exception to fail the test.

@Test

public void testMap_INTERSECTION() throws IOException {

Initialize the test input to drive the test. In order to hit the if-block we want to test, we have to ensure the surface type is of RoadType.*INTERSECTION*.

LinkSurfaceMaplsm = new LinkSurfaceMap();

RoadSurfacers = new RoadSurface(Enums.RoadType.INTERSECTION); byte[] lsmBytes = append(lsm, rs);

We use PowerMock[3] to mock out a static call to the PopulatorPreprocessor class. PopulatorPreprocessor is a separate class containing business logic and is tested by it's own JUnit test. At the class level, we set-up PowerMock with the @RunWith annotation and tell it which classes to mock; in this case one, PopulatorPreprocessor. With @PrepareForTest we tell PowerMock which classes have static methods that we want to mock. PowerMock supports both EasyMock and Mockito, since we're using Mockito, you'll see references to PowerMockito. We mock the static class by calling PowerMockito.*mockStatic*.

@RunWith(PowerMockRunner.class)

@PrepareForTest(PopulatorPreprocessor.class)

PowerMockito.mockStatic(PopulatorPreprocessor.class); Set the previously created test input and run the mapper:

mapDriver.withInput(new LongWritable(1234567), new BytesWritable(lsmBytes));

mapDriver.runTest();

Verify the output. SurfaceCounters.*NONLINEARS* is incremented once, and SurfaceCounters.*ROADS* and SurfaceCounters.*UNKNOWN* are not incremented. A quick review – with JUnit's *assertEquals*, the first parameter, a String, which is optional, is the assertion error message. The second parameter is the *expected* value and the third parameter is

the *actual value*. *assertEquals* prints out a nice error message of the form "expected: <x> but was: <y>." So if the second assertion were to fire, e.g., we could get the error message "java.lang.AssertionError: NONLINEARS count incorrect. expected:<1> but was:<0>."

```
Assert.assertEquals("ROADS count incorrect.", 0,
mapDriver.getCounters().findCounter(SurfaceCounters.ROADS).getValue());
Assert.assertEquals("NONLINEARS count incorrect.", 1,
mapDriver.getCounters().findCounter(SurfaceCounters.NONLINEARS).getValue());
Assert.assertEquals("UNKNOWN count incorrect.", 0,
mapDriver.getCounters().findCounter(SurfaceCounters.UNKNOWN).getValue());
```

Verify that PopulatorPreprocessor.*processLinearSurface*(surface) **has not** been called, by using the following PowerMock/Mockito syntax.

```
PowerMockito.verifyStatic(Mockito.never());
    PopulatorPreprocessor.processLinearSurface(rs);
```

testMap_ROAD

In our second test, testMap_ROAD(). Our Objective is to verify:

1. SurfaceCounters.*ROADS* is incremented.
2. That PopulatorPreprocessor.*processLinearSurface*(surface) **is** called.
3. SurfaceCounters.*NONLINEARS* and SurfaceCounters.*UNKNOWN* are **not** incremente d.

The setup is identical to the first test with a couple of exceptions.

1. Specifying a Road type in our input data.

```
RoadSurfacers = newRoadSurface(Enums.RoadType.ROAD);
```

2. Setting the log4j debug level.

Interestingly, in our source code we only want to count road surfaces when debug level is set in the log4j logger. To test this,first we save the original logging level, then we retrieve the Root logger and set the level to *DEBUG* .

```
Level originalLevel = Logger.getRootLogger().getLevel();
Logger.getRootLogger().setLevel(Level.DEBUG)
```

At the end of the test, we revert to the original logging level so as not to affect other tests

```
Logger.getRootLogger().setLevel(originalLevel);
```

Once again, let's verify the output. SurfaceCounters. *ROADS* is incremented once, and SurfaceCounters. *NONLINEARS* and SurfaceCounters.*UNKNOWN* are not incremented.

```
Assert.assertEquals("ROADS count incorrect.", 1,
```

mapDriver.getCounters().findCounter(SurfaceCounters.*ROADS*).getValue());

Assert.*assertEquals*("NONLINEARS count incorrect.", 0,

mapDriver.getCounters().findCounter(SurfaceCounters.*NONLINEARS*).getValue());

Assert.*assertEquals*("UNKNOWN count incorrect.", 0,
mapDriver.getCounters().findCounter(SurfaceCounters.*UNKNOWN*).getValue());

Verify that PopulatorPreprocessor.*processLinearSurface*(surface) **has** been called **once**, by using the following PowerMock/Mockito syntax.

```
PowerMockito.verifyStatic(Mockito.times(1));
    PopulatorPreprocessor.processLinearSurface(rs);
```

Testing A REDUCER

The same principles would apply as in testing a mapper. The difference being that we would want to create a ReducerDriver, and populate it with our reducer class under test as shown below.

```
privateReduceDriver<LongWritable, BytesWritable, LongWritable, BytesWritable>reduceDriver;
  @Before public
voidsetUp() {
        MyReducer reducer = newMyReducer ();
    reduceDriver = newReduceDriver<LongWritable, BytesWritable, LongWritable, BytesWritable>();
        reduceDriver.setReducer(reducer);
  }
```

MAVEN Pom Dependencies

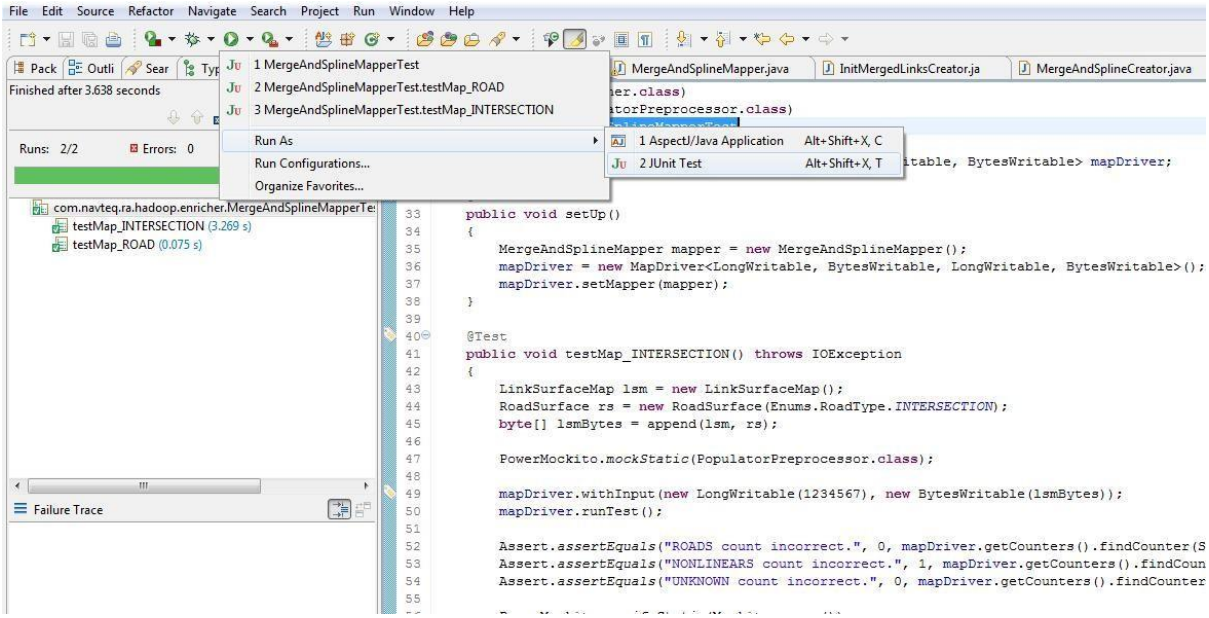
In addition to JUnit 4, you'll have to include the following dependencies in your maven pom.xml. On the PowerMock web page[3], take note of the supported versions of Mockito.

```
<dependency>
        <groupId>org.apache.mrunit</groupId>
        <artifactId>mrunit</artifactId>
        <version>0.8.0-incubating</version>
        <scope>test</scope>
    </dependency>
<dependency>
```

```
            <groupId>org.mockito</groupId>
            <artifactId>mockito-all</artifactId>
            <version>1.9.0-rc1</version>
            <scope>test</scope>
        </dependency>
    <dependency>
            <groupId>org.powermock</groupId>
            <artifactId>powermock-module-junit4</artifactId>
            <version>1.4.12</version>
            <scope>test</scope>
        </dependency>
    <dependency>
            <groupId>org.powermock</groupId>
            <artifactId>powermock-api-mockito</artifactId>
            <version>1.4.12</version>
            <scope>test</scope>
</dependency>
```

Running In Eclipse

The test is run just as any other JUnit test would be run. Here's an example of the test running inside Eclipse.



Summary

MRUnit provides a powerful and light-weight approach to do test-driven development. A nice side effect is that it helps move you to better code coverage than was previously possible.

Exception Handling (Handling Failures in Hadoop, Mapreduce)

- 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 your job to complete successfully.
- We need to consider the failure of any of the following entities the task, the application master, the node manager, and 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 mode is the sudden exit of the task JVM perhaps there is a JVM bug that causes the JVM to exit for a particular set of circumstances exposed by the MapReduce user code.
- 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.
- Hanging tasks are dealt with differently. The application master notices that it hasn't received a progress update for a while and proceeds to mark the task as failed.
- The task JVM process will be killed automatically after this period. The timeout period after which tasks are considered failed is normally 10 minutes and can be configured on a per-job basis (or a cluster basis) by setting the mapreduce.task.timeout property to a value in milliseconds.
- Setting the timeout to a value of zero disables the timeout, so long-running tasks are never marked as failed.

- In this case, a hanging task will never free up its container, and over time there may be cluster slowdown as a result. This approach should therefore be avoided, and making sure that a task is reporting progress periodically should suffice.
- When the application master is notified of a task attempt that has failed, it will reschedule execution of the task. The application master will try to avoid rescheduling the task on a node manager where it has previously failed. Furthermore, if a task fails four times, it will not be retried again. This value is configurable. The maximum number

  of attempts to run a task is controlled by the mapreduce.map.maxattempts property for map tasks and mapreduce.reduce.maxattempts for reduce tasks. By default, if any task fails four times (or whatever the maximum number of attempts is configured to), the whole job fails.
- For some applications, it is undesirable to abort the job if a few tasks fail, as it may be possible to use the results of the job despite some failures. In this case, the maximum percentage of tasks that are allowed to fail without triggering job failure can be set for the job. Map tasks and reduce tasks are controlled independently, using the mapreduce.map.failures.maxpercent and mapreduce.reduce.failures.maxpercent properties.
- A task attempt may also be killed, which is different from it failing. A task attempt may be killed because it is a speculative duplicate or because the node manager it was running on failed and the application master marked all the task attempts running on it as killed.
- Killed task attempts do not count against the number of attempts to run the task (as set by mapreduce.map.maxattempts and mapreduce.reduce.maxattempts), because it wasn't the task's fault that an attempt was killed.
- Application Master Failure
- Just like MapReduce tasks are given several attempts to succeed (in the face of hardware or network failures), applications in YARN are retried 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 master running on the cluster, and individual applications may not exceed this limit. The limit is set by yarn.resourcemanager.am.max-attempts and defaults to 2, so if you want to

increase the number of MapReduce application master attempts, you will have to increase the YARN setting on the cluster, too.

- The way recovery works is as follows. An application master sends periodic heartbeats to the resource manager, and in the event of application master failure, the resource manager will detect the failure and start a new instance of the master running in a new container which is managed by a node manager.
- In the case of the MapReduce application master, it will use the job history to recover the state of the tasks that were already run by the application so they don't have to be rerun.

  Recovery is enabled by default, but can be disabled by setting yarn.app.mapreduce.am.job.recovery.enable to false.

- Node Manager Failure
- If a node manager fails by crashing or running very slowly, it will stop sending heartbeats to the resource manager (or send them very infrequently). The resource manager will notice a node manager that has stopped sending heartbeats if it hasn't received one for 10 minutes (this is configured, in milliseconds, via the yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms property) and 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 using the mechanisms described in the previous two sections. In addition, the application master arranges for map tasks that were run and completed successfully on the failed node manager to be rerun if they belong to incomplete jobs, since their intermediate output residing on the failed node manager's local filesystem may not be accessible to the reduce task.
- Node managers may be blacklisted if the number of failures for the application is high, even if the node manager itself has not failed. Blacklisting is done by the application master, and for MapReduce the application master will try to reschedule tasks on different nodes if more than three tasks fail on a node manager. The user may set the threshold with the mapreduce.job.maxtaskfailures.per.tracker job property.
- Note : Note that the resource manager does not do blacklisting across applications (at the time of writing), so tasks from new jobs may be scheduled on bad nodes even if they have been blacklisted by an application master running an earlier job.
- 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 (unlikely) 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 (backed by ZooKeeper or HDFS), so that the standby can recover the core state of the failed active resource manager.

Tuning of Mapreduce program

- Most important process of mapreduce program is shuffling of outputs produced by map function .So we need to concentrate mainly on map phase for better optimization of mapreduce programshence, We should do the following things for optimization :-
- We should give as much as more memory to shuffle process , but also need to keep in mind that we also give sufficient memory to map and reduce function .
- Amount of memory given to JVM for map reduce tasks is set by :- mapred.child.java.opts , we need to make this value more as much as we can. Map Side optimization: -
- optimization can be done by minimizing the multi spills ,, which can be controlled by
- io.sort.*
- We can use counters to check about the count of spill records
- We should increaseio.sort.mb :- which is used as amount of memory buffer used while sorting out the map output • io.sort.spill.percent :- threshold value for using memory buffer , afterwards records started to spill.Default value :- 0.80.
- io.sort.factor :- property which help to merge output streams by map. We should increase this upto 100 for optimization.
- mapred.compress.map.output :- we should compress the ouptput of mapper phase , it saves space as ell increase transfer of data between tasks.

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 – SIT1606

UNIT 5 HADOOP PROJECT ENVIRONMENT **9 Hrs.**

HBase: Introduction to HBase, Client API's and their features, Available Client, HBase Architecture, MapReduce Integration. HBase: Advanced Usage, Schema Design, Advance Indexing, Coprocessors, Hadoop 2.0-MRv2 - YARN - NameNode High Availability, HDFS Federation, MRv2, YARN, Running MRv1 in YARN, Upgrade your existing MRv1 code to MRv2, Programming in YARN framework-cover Apache Oozie Workflow Scheduler for Hadoop

Hadoop Project Environment

5.1 Introduction to HBase

HBase is a distributed column-oriented database built on top of the Hadoop file system. Base is a data model that is similar to Google's big table designed to provide quick random access to huge amounts of structured data.It leverages the fault tolerance provided by the Hadoop File System (HDFS).It is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System.One can store the data in HDFS either directly or through HBase.Data consumer reads and accesses the data in HDFS randomly using HBase.

Need for Hbase is as follows:

- Hadoop can perform only batch processing
- data will be accessed only in a sequential manner.
- That means one has to search the entire dataset even for the simplest of jobs.
- a new solution is needed to access any point of data in a single unit of time (random access)

HBase sits on top of the Hadoop File System and provides read and write access. Figure 5.1 shows the Hbase read and write

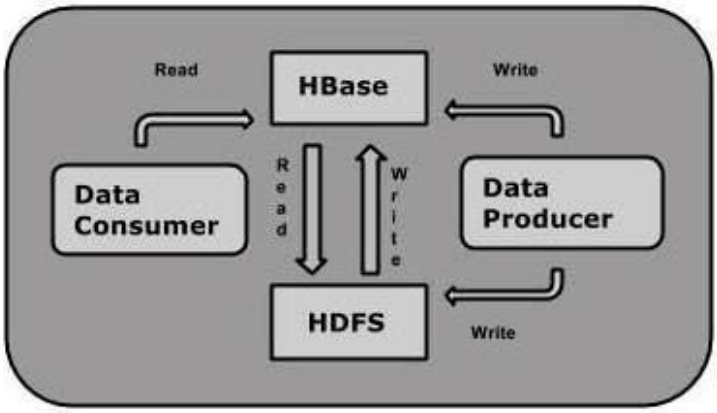


Figure 5.1 Hbase Read/Write

Relational DBMS Vs Column oriented database

In a row-oriented indexed system, the primary key is the rowid that is mapped from indexed data. In Hbase, HBase is built on top of HDFS where the HBase data is stored in HFiles and HFiles are stored on HDFS. The purpose for introducing HBase was to enable random access to data for interactive querying which was not supported by pure HDFS implementation.

Column-oriented vs Row-oriented storages

Column-oriented Database	**Row oriented Database**
When the situation comes to process and analytics we use this approach. Such as **Online Analytical Processing** and it's applications.	**Online Transactional process** such as banking and finance domains use this approach. Online banking, Online airline ticket booking Sending a text message, Order entry Add a book to shopping cart
The amount of data that can able to store in this model is very huge like in terms of petabytes	It is designed for a small number of rows and columns.

DB design is subject oriented. Example: Database design changes with subjects like sales, marketing, purchasing, etc.	DB design is application oriented. Example: Database design changes with industry like Retail, Airline, Banking, etc.

Storage Mechanism in HBase

The data storage model is shown on figure 5.2.

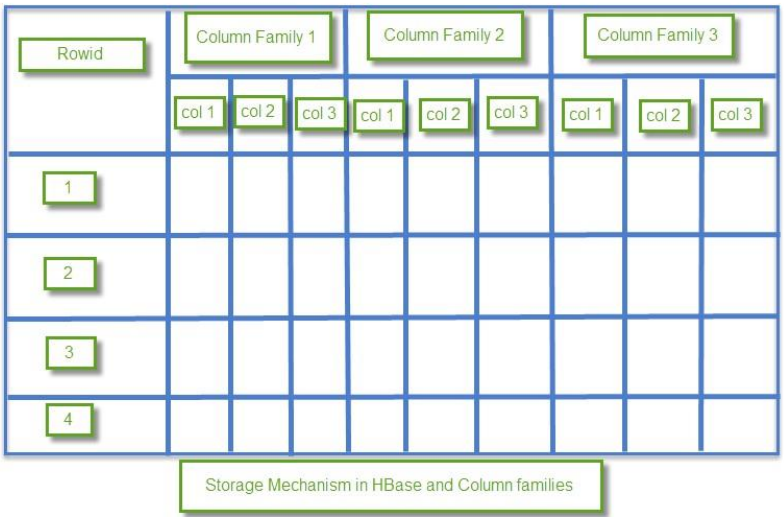


Figure 5.2 Hbase Data model

HBase is a **column-oriented database** and the tables in it are sorted by row. The table schema defines only column families, which are the key value pairs. Table is a collection of rows. Row is a collection of column families and column family is a collection of columns.
Column is a collection of key value pairs.

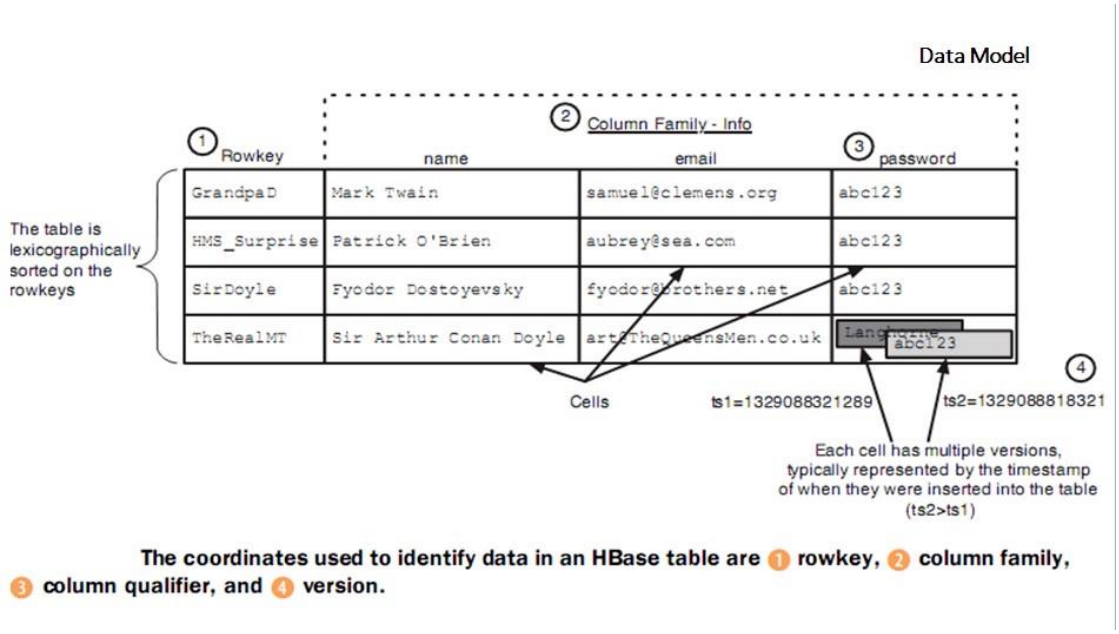


Figure 5.3 Sample Hbase Column database

Features of HBase

The important feature of Hbase is as follows:

- HBase is linearly scalable :You can add any number of columns anytime.
- It has automatic failure support.
- Integrations with Map/Reduce framework: All the commands and java codes internally implement Map/ Reduce to do the task.
- fundamentally, it's a platform for storing and retrieving data with random access.
- It doesn't care about data types(storing an integer in one row and a string in another for the same column).
- It doesn't enforce relationships within your data.
- It provides consistent read and writes.
- It integrates with Hadoop, both as a source and a destination.
- It has easy java API for client.
- It provides data replication across clusters.

Applications of HBase

It is used whenever there is a need to write heavy applications. HBase is used whenever we need to provide fast random access to available data. Companies such as Facebook, Twitter, Yahoo, and Adobe use HBase internally.

5.2HBase Architecture

HBase is a column-oriented database and data is stored in tables.The tables are sorted by RowId. As shown below figure 5.5, HBase has RowId, which is the collection of several column families that are present in the table.The column families that are present in the schema are key-value pairs. If we observe in detail each column family having multiple numbers of columns.The column values stored into disk memory. Each cell of the table has its own Metadata like timestamp and other information.

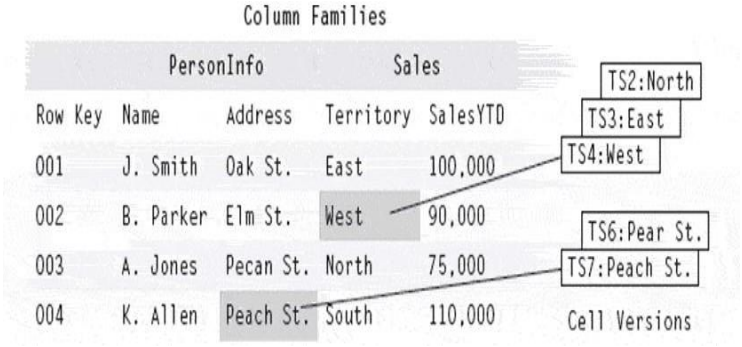


Figure 5.4 Sample Hbase database

In the above table , two column families: PersonInfo and Sales are present. PersonInfo column family has two column qualifiers: Name and Address and Sales column family has two column qualifiers: Territory and SalesYTD. All values have a timestamp and Territory for row key 002 has multiple values that have changed over time (same for Address of row key 004)

The above can be represented as this JSON -like map:

```
{"001" :  {"PersonInfo" :
                {"Name" :
                        {TS1: "J. Smith"}
                 "Address" :
                        {TS1: "Oak St"}}
           "Sales" :
                {"Territory" :
                        {TS1: "East"}
                 "SalesYTD" :
                        {TS1: "100,000"}}}
 "002" :  {"PersonInfo" :
                {"Name" :
                        {TS2: "B. Parker"}
                  "Address" :
                        {TS2: "Elm St."}}
          "Sales" :
                {"Territory" :
                        {TS4: "West"
                         TS3: "East"
                         TS2: "North"}
                 "SalesYTD" :
                        {TS2: "90,000"}}}
…}
```

key terms representing table schema **Table**:

Collection of rows present.

Row: Collection of column families.

Column Family: Collection of columns.

Column: Collection of key-value pairs.

Namespace: Logical grouping of tables.

Cell: A {row, column, version} tuple exactly specifies a cell definition in HBase.

HBase Architecture and its Important Components are shown in figure 5.6.

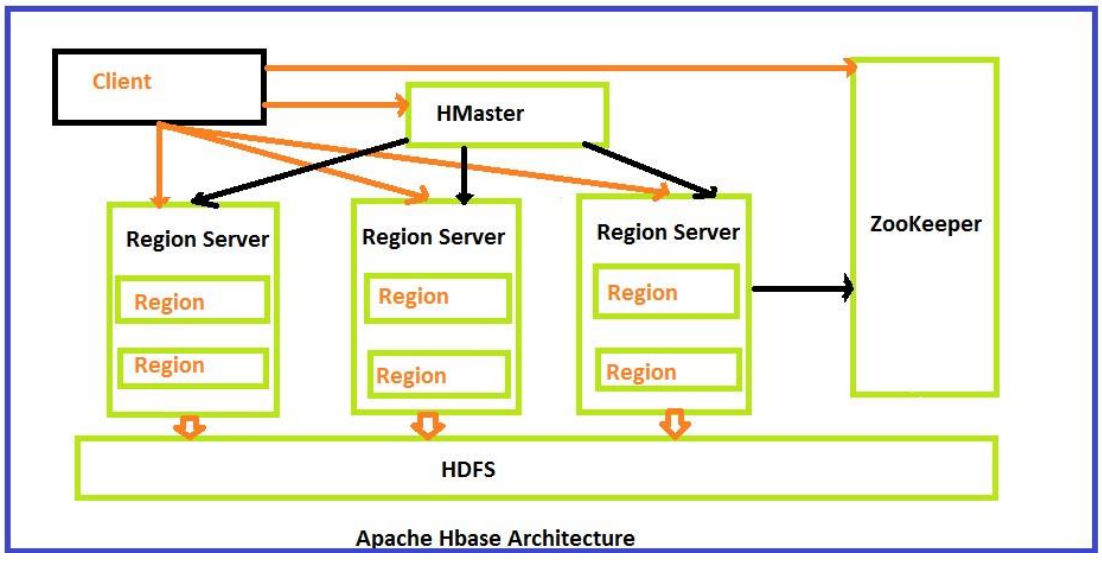


Figure 5.5Hbase Architecture

HBase architecture consists mainly of four components

- HMaster
- HRegionserver

- HRegions
- Zookeeper
- HDFS

HMaster:

HMaster is the implementation of a Master server in HBase architecture.It acts as a monitoring agent to monitor all Region Server instances present in the cluster and acts as an interface for all the metadata changes. In a distributed cluster environment, Master runs on NameNode. Master runs several background threads.

Roles performed by HMaster in HBase.

- provides admin performance and distributes services to different region servers.
- assigns regions to region servers.
- controlling load balancing and failover to handle the load over nodes present in the cluster.
- When a client wants to change any schema and to change any Metadata operations, HMaster takes responsibility for these operations.

Some of the methods exposed by HMaster Interface are primarily Metadata oriented methods.

- Table (createTable, removeTable, enable, disable)
- ColumnFamily (add Column, modify Column)
- Region (move, assign)
- The client communicates in a bi-directional way with both HMaster and ZooKeeper.
- For read and write operations, it directly contacts with HRegion servers.
- HMaster assigns regions to region servers and in turn, check the health status of region servers.

Hbase Region Servers

- receives writes , and read requests from the client
- it assigns the request to a specific region, where the actual column family resides.
- Client can directly communicates the region server.

- The client requires HMaster help when operations related to metadata and schema changes are required.
- It is responsible for serving and managing regions or data that is present in a distributed cluster.
- The region servers run on Data Nodes present in the Hadoop cluster.

Hbase Region Servers

HMaster can get into contact with multiple HRegion servers and performs the following functions.

- Hosting and managing regions
- Splitting regions automatically
- Handling read and writes requests
- Communicating with the client directly

HBase Regions:

HRegions are the basic building elements of HBase cluster that consists of the distribution of tables and are comprised of Column families. It contains multiple stores, one for each column family. It consists of mainly two components, which are Memstore and Hfile.

ZooKeeper:

In HBase, Zookeeper is a centralized monitoring server which maintains configuration information and provides distributed synchronization. Distributed synchronization is to access the distributed applications running across the cluster with the responsibility of providing coordination services between nodes.

Services provided by ZooKeeper

- Maintains Configuration information
- Provides distributed synchronization
- Client Communication establishment with region servers
- Provides ephemeral nodes for which represent different region servers
- Master servers usability of ephemeral nodes for discovering available servers in the cluster
- To track server failure and network partitions

Master and HBase slave nodes (region servers) registered themselves with ZooKeeper. The client needs access to ZK(zookeeper) quorum configuration to connect with master and region servers.During a failure of nodes that present in HBase cluster, ZKquoram will trigger error messages, and it starts to repair the failed nodes.

HDFS:-

HDFS is a Hadoop distributed file system, as the name implies it provides a distributed environment for the storage and it is a file system designed in a way to run on commodity hardware. It stores each file in multiple blocks and to maintain fault tolerance, the blocks are replicated across a Hadoop cluster.HDFS provides a high degree of fault –tolerance and runs on cheap commodity hardware. By adding nodes to the cluster and performing processing & storing by using the cheap commodity hardware, it will give the client better results as compared to the existing one.

HBase Read and Write Data

Hbase read and write operations are shown in figure 5.6

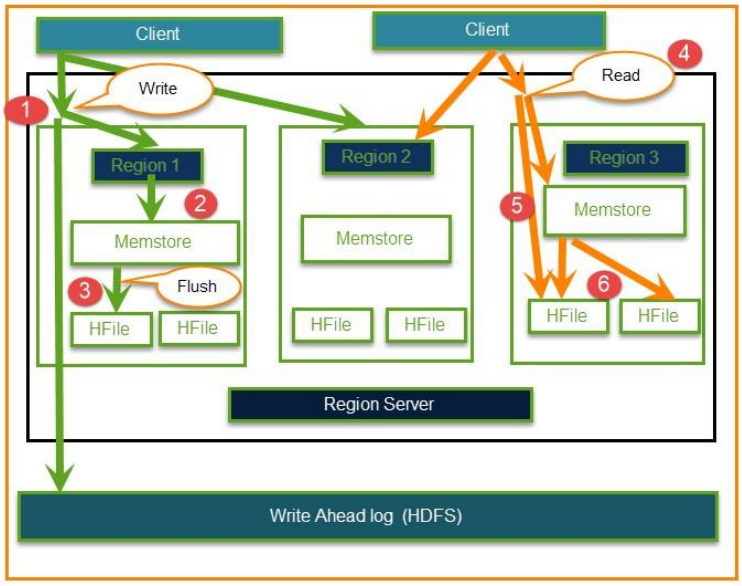


Figure 5.6Hbase Read and Write operation

Step 1) Client wants to write data and in turn first communicates with Regions server and then regions

Step 2) Regions contacting memstore for storing associated with the column family

Step 3) First data stores into Memstore, where the data is sorted and after that, it flushes into HFile. The main reason for using Memstore is to store data in a Distributed file system based on Row Key. Memstore will be placed in Region server main memory while HFiles are written into HDFS.

Step 4) Client wants to read data from Regions

Step 5) In turn Client can have direct access to Mem store, and it can request for data.
Step 6) Client approaches HFiles to get the data. The data are fetched and retrieved by the Client.

Memstore holds in-memory modifications to the store. The hierarchy of objects in HBase Regions is as shown from top to bottom in below table.

Table	HBase table present in the HBase cluster
Region	HRegions for the presented tables
Store	It stores per ColumnFamily for each region for the table
Memstore	• Memstore for each store for each region for the table • It sorts data before flushing into HFiles • Write and read performance will increase because of sorting
StoreFile	StoreFiles for each store for each region for the table
Block	Blocks present inside StoreFiles

5.3 Client API's and their features

HBase Client API – HTable,Put, Get, Delete, Result

In order to perform CRUD operations on HBase tables, we use Client API for Hbase.

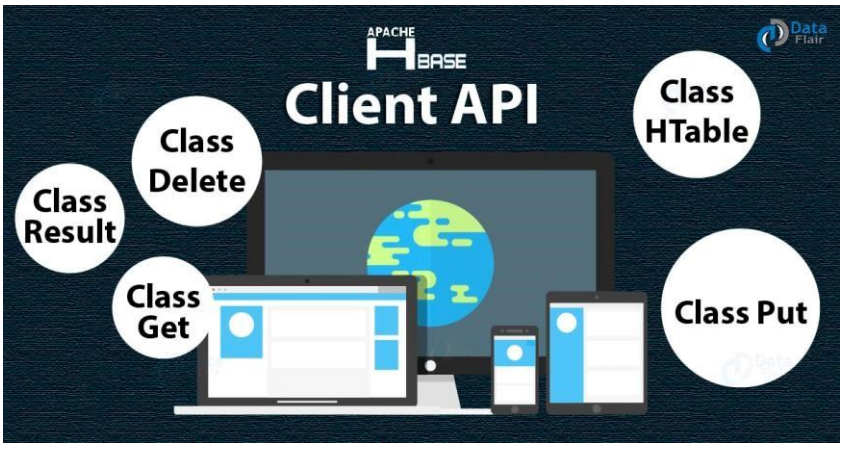


Figure 5.7Hbase Client API

To perform CRUD operations on HBase tables we use Java client API for HBase.Since HBase has a Java Native API and it is written in Java thus it offers programmatic access to DML (Data Manipulation Language).

Class HBase Configuration

- This class adds HBase configuration files to a Configuration. It belongs to the org.apache.hadoop.hbase package.

Method

- static org.apache.hadoop.conf.Configuration create()
- To create a Configuration with HBase resources, we use this method.

Class HTable in HBase Client API

- An HBase internal class which represents an HBase table is HTable.
- Basically, to communicate with a single HBase table, we use this implementation of a table.
- It belongs to the **org.apache.hadoop.hbase.client** class.

a. Constructors

i. HTable() ii. HTable(TableNametableName, ClusterConnection connection, ExecutorService pool)

• We can create an object to access an HBase table, by using this constructor. b.

Methods

i. void close()

Basically, to release all the resources of the HTable, we use this method.

ii. void delete(Delete delete)

The method "void delete(Delete delete)" helps to delete the specified cells/row.

iii. boolean exists(Get get)

As specified by Get, it is possible to test the existence of columns in the table, with this method.

iv. Result get(Get get)

This method retrieves certain cells from a given row.

v. org.apache.hadoop.conf.ConfigurationgetConfiguration()

It returns the Configuration object used by this instance. vi.

TableNamegetName()

This method returns the table name instance of this table.

vii. HTableDescriptorgetTableDescriptor()

It returns the table descriptor for this table. viii.

byte[] getTableName()

This method returns the name of this table.

ix. void put(Put put)

We can insert data into the table, by using this method.

In order to perform put operations for a single row, we use this class. This class belongs to the **org.apache.hadoop.hbase.client** package.

a. Constructors

i. Put(byte[] row) ii. Put(byte[] rowArray,

introwOffset, introwLength)

However, to make a copy of the passed-in row key to keep local, we use it.

iii. Put(byte[] rowArray, introwOffset, introwLength, long ts)

We can make a copy of the passed-in row key to keep local, by using this constructor.

iv. Put(byte[] row, long ts)

Basically, to create a Put operation for the specified row, using a given timestamp, we use it.

b. Methods

i. Put add(byte[] family, byte[] qualifier, byte[] value)

The method "Put add(byte[] family, byte[] qualifier, byte[] value)" adds the specified column and value to this Put operation.

ii. Put add(byte[] family, byte[] qualifier, long ts, byte[] value) With the specified timestamp, it adds the specified column and value, as its version to this Put operation.

iii. Put add(byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value) This method adds the specified column and value, with the specified timestamp as its version to this Put operation.

iv. Put add(byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value) With the specified timestamp, it adds the specified column and value, as its version to this Put operation.

To perform Get operations on a single row, we use this class. It belongs to the **org.apache.hadoop.hbase.client** package.

a. Constructor

i. Get(byte[] row)

It is possible to create a Get operation for the specified row, by using this constructor. **ii.**

Get(Get get)

b. Methods

i. Get addColumn(byte[] family, byte[] qualifier)

To retrieves the column from the specific family with the specified qualifier, this method helps.

ii. Get addFamily(byte[] family)

Whereas this one helps to retrieves all columns from the specified family.

In order to perform delete operations on a single row, we use this Class.

Instantiate a Delete object with the row to delete, to delete an entire row.

It belongs to the org.apache.hadoop.hbase.client package.

a. Constructor

i. Delete(byte[] row)

To create a delete operation for the specified row, we use it. **ii.**

Delete(byte[] rowArray, introwOffset, introwLength)

This constructor creates a Delete operation for the specified row and timestamp.

iii. Delete(byte[] rowArray, introwOffset, introwLength, long ts)

Basically, the constructor "Delete(byte[] rowArray, introwOffset, introwLength, long ts)" creates a Delete operation.

iv. Delete(byte[] row, long timestamp)

Again the constructor "Delete(byte[] row, long timestamp)" also creates a Delete operation. **b.**

Methods

i. Delete addColumn(byte[] family, byte[] qualifier)

This method helps to delete the latest version of the specified column. **ii.**

Delete addColumns(byte[] family, byte[] qualifier, long timestamp)

However, to delete all versions of the specified column we use this method,

especially, With a timestamp less than or equal to the specified timestamp.

iii. Delete addFamily(byte[] family)

The method "Delete addFamily(byte[] family)" deletes all versions of all columns of the specified family.

iv. Delete addFamily(byte[] family, long timestamp)

Again, with a timestamp less than or equal to the specified timestamp, this method also deletes all columns of the specified family.

In order to get a single row result of a Get or a Scan query, we use class result HBase Client API.

a. Constructors

i.Result()

With no KeyValue payload, it is possible to create an empty Result; returns null if you call raw Cells(), by using this constructor.

b.Methods i. byte[] getValue(byte[] family, byte[] qualifier)

Basically, in order to get the latest version of the specified column, we use this method.

ii. byte[] getRow()

Moreover, to retrieve the row key which corresponds to the row from which this Result was created, we use this method.

So, this was all about HBase Client API. Hope you like our explanation.

Client API Features:-Filters

HBase filters are a powerful feature that can greatly enhance your effectiveness when working with data stored in tables. The two prominent read functions for HBase are get() and scan(), both supporting either direct access to data or the use of a start and end key, respectively. These include column families, column qualifiers, timestamps or ranges, as well as version number.

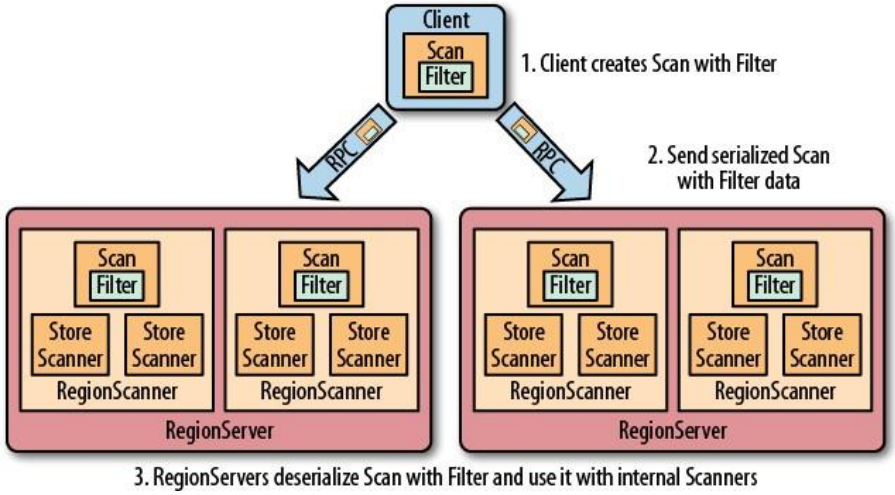


Fig- 5.8 Shows the filters created on the client side, sent through the RPC, and executed on the server side

The filter hierarchy

The lowest level in the filter hierarchy is the Filter interface, and the abstractFilterBase class that implements an empty shell, or skeleton, that is used by the actual filter classes to avoid having the same boilerplate code in each ofthem.

You define a new instance of the filter you want to apply and hand it to the Get or Scan instances,using:

```
setFilter(filter)
```

Comparison operators

As CompareFilter-based filters add one more feature to the base FilterBaseclass, namely the compare() operation, it has to have a user-supplied operator type that defines how the result of the comparison isinterpreted.

Table5.1 The possible comparison operators for CompareFilter-based filters

Operator	**Description**
LESS	Match values less than the provided one.
LESS_OR_EQUAL	Match values less than or equal to the provided one.
EQUAL	Do an exact match on the value and the provided one.
NOT_EQUAL	Includeeverything that does not match the providedvalue.

GREATER_OR_EQUAL	Match values that are equal to or greater than the provided one.
GREATER	Only include values greater than the provided one.
NO_OP	Exclude everything.

The comparison operators define what is included, or excluded, when the filter is applied. This allows you to select the data that you want as either a range, subset, or exact and single match.

Comparators

The second type that you need to provide to CompareFilter-related classes is a *comparator*, which is needed to compare various values and keys in different ways. They are derived from WritableByteArrayComparable, which implements Writable, and Comparable. You do not have to go into the details if you just want to use an implementation provided by HBase and listed in the Table .The constructors usually take the control value, that is, the one to compare each table valueagainst.

Table-5.2 The HBase-supplied comparators, used with CompareFilter-based filters

Comparator	**Description**
BinaryComparator	Uses Bytes.compareTo() to compare the current with the provided value.
BinaryPrefixComparator	Similar to the above, but does a lefthand, prefix- based match usingBytes.compareTo().
NullComparator	Does not compare against an actual value but whether a given one is null, or notnull.
BitComparator	Performs a bitwise comparison, providing a BitwiseOp class with AND,OR,and XOR operators.
RegexStringComparator	Given a regular expression at instantiation this comparator does a pattern match on the table data.
SubstringComparator	Treats the value and table data as Stringinstances and performs a contains()check.

FILTERROW() AND BATCHMODE

A filter using filterRow() to filter out an entire row, or filterRow(List) to modify the final list of included values, *must* also override the hasRowFilter() function to return true.Figure shows the logical flow of the filter methods for a single row. There is a more fine-grained process to apply the filters on a column level, which is not relevant in this context.

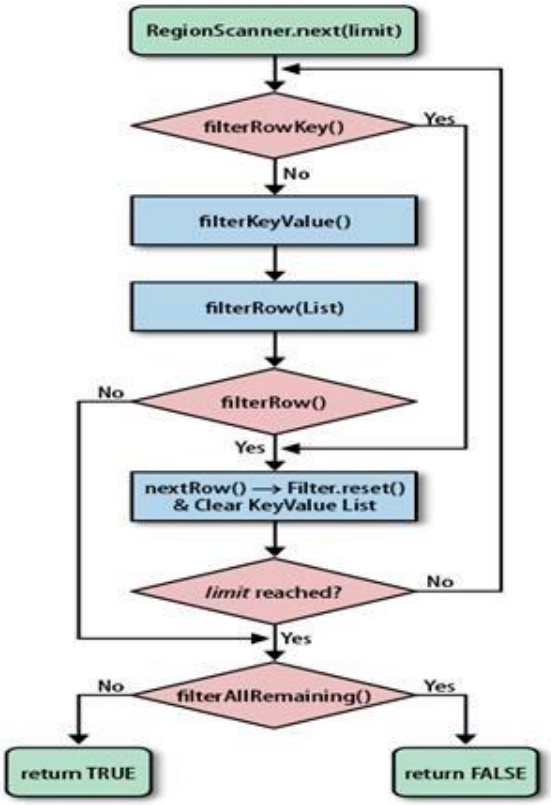


Figure 5.9 logical flow of the filter methods

5.4 Advanced Usage of HBASE:

It is important to have a good understanding of how to design tables, row keys, column names, and so on, to take full advantage of the architecture.

Key Design

HBase has two fundamental *key* structures: the *row key* and the *column key*. Both can be used to convey meaning, by either the data they store, or by exploiting their sorting order. In the following sections, we will use these keys to solve commonly found problems when designing storage solutions.

Concepts

The first concept to explain in more detail is the logical layout of a table, compared to on-disk storage. HBase's main unit of separation within a table is the *column family*— not the actual columns as expected from a column-oriented database in their traditional sense. Figure - shows the fact that, although you store cells in a table format logically, in reality these rows are stored as linear sets of the actual cells, which in turn contain all the vital information inside them.

The top-left part of the figure shows the logical layout of your data—you have rows and columns. The columns are the typical HBase combination of a column family name and a column qualifier, forming the *column key*. The rows also have a *row key* so that you can address all columns in one logicalrow.

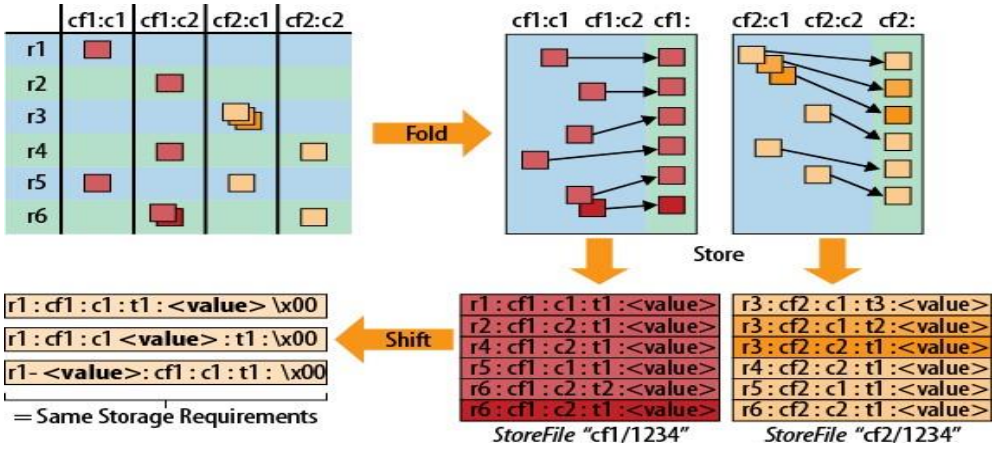


Figure 5.10Hbase cell

HBASE Design Schema

With HBase, you have a ―query-first‖ schema design; all possible queries should be identified first, and the schema model designed accordingly. You should design your HBase schema to take advantage of the strengths of HBase. Think about your access patterns, and design your schema so that the data that is read together is stored together. Remember that HBase is designed for clustering.

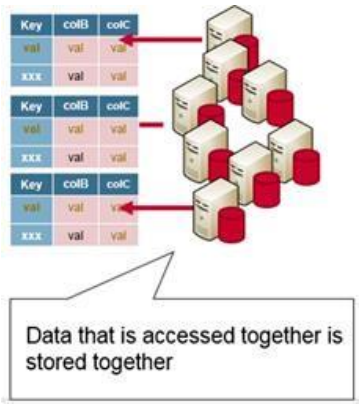
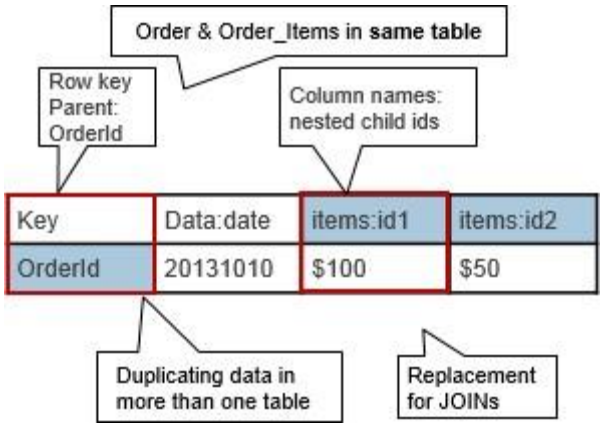


Figure 5.11 Data Access

- Distributed data is stored and accessedtogether
- It is query-centric, so focus on how the data isread
- Design for thequestions

Parent-Child Relationship–Nested Entity

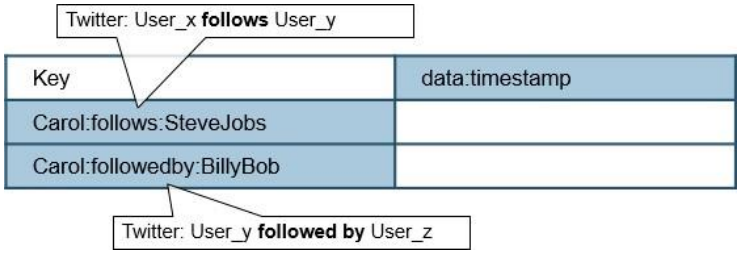
Here is an example of denormalization in HBase, if your tables exist in aone-tomanyrelationship, it‘s possible to model it in HBase as a single row. In the example below, theorder and related line items are stored together and can be read together with a get on the row key. This makes the reads a lot faster than joining tables together.



The rowkey corresponds to the parent entity id, the OrderId. There is one column family for the order data, and one column family for the order items. The Order Items arenested, the Order Item IDs are put into the column names and any non-identifying attributes are put into thevalue.This kind of schema design is appropriate when the only way you get at the childentities is via the parententity.

Self-Join Relationship – HBase

- A self-join is a relationship in which both match fields are defined in the sametable.
- Consider a schema for twitter relationships, where the queries are: which users does userX follow, and which users follow userX? Here‘s a possible solution: The userids are put in a composite row key with the relationship type as a separator. For example, Carol follows Steve Jobs and Carol is followed by BillyBob. This allows for row key scans for everyone carol:follows or carol:followedby
- Below is the example Twittertable:



Schema Design Exploration:

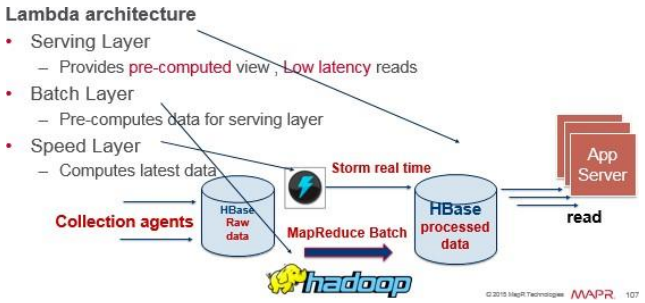
- Raw data from HDFS orHBase
- MapReduce for data transformation and ETL from rawdata.
- Use bulk import from MapReduce toHBase
- Serve data for online reads fromHBase

Designing for reads means aggressively de-normalizing data so that the data that is read together is stored together.

Data Access Pattern

The batch layer precomputes the batch views. In the batch view, you read the results from a precomputed view. The precomputed view is indexed so that it can be accessed quickly with random reads.The serving layer indexes the batch view and loads it up so it can be efficiently queried to get particular values out of the view. A serving layer database only requires batch updates and random reads. The serving layer updates whenever the batch layer finishes precomputing a batch view.

You can do stream-based processing with Storm and batch processing with Hadoop. The speed layer only produces views on recent data, and is for functions computed on data in the few hours not covered by the batch.



5.4 Advance Indexing In HBase

In HBase, the row key provides the same data retrieval benefits as a primary index. So, when you create a secondary index, use elements that are different from the row key. Secondary indexes allow you to have a secondary way to read an HBase table. They provide a way to efficiently access records by means of some piece of information other than the primary key. Secondary indexes require additional cluster space and processing because the act of creating a secondary index requires both space and processing cycles to update. A method of index maintenance, called Diff-Index, can help IBM® Big SQL to create secondary indexes for HBase, maintain those indexes, and use indexes to speed up queries.

Why is this important?

With secondary indexing, I can either find a single row to find all of the rows that contain an attribute with a specific value. Like everything in HBase, its stored in sort order so it becomes rather trivial for the client to fetch several rows and join them in sort order, or to take the intersection if we are trying to find the records that meet a specific qualification. (e.g. find all of Bob's employees who live in Cleveland, OH. Or find the average cost of repairing a Volvo S80 that was involved in a front end collision.)As more people want to use HBase like a database and apply SQL, using secondary indexing makes filtering and doing data joins much more efficient. One just takes the intersection of the indexed qualifiers specified, and then apply the unindexed qualifiers as filters further reducing theresulset

Problems in Indexing

This design appears to mimic the concept of the column families, where like data is stored in a column family file. So that like data can be accessed quickly. But like the column families, we run in to the same problem... too many column families can be a bad thing when it comes to compaction. Depending on the number of columns to be indexed, the size of the index table could easily be larger than the base table. (Especially when you add in geo-spatial indexing.)

Co-Processor

The idea of HBase Coprocessors was inspired by Google‘s BigTable coprocessors. which Google developed to bring computing parallelism to BigTable. They have the following characteristics:

- Arbitrary code can run at each tablet in tableserver
- High-level call interface forclients
- Calls are addressed to rows or ranges of rows and the coprocessor client library resolves them to actual locations;
- Calls across multiple rows are automatically split into multiple parallelized RPC
- Provides a very flexible model for building distributed services
- Automatic scaling, load balancing, request routing for applications

Back to HBase, we definitely want to support efficient computational parallelism as well, beyond what HadoopMapReduce can provide. In addition, exciting new features can be built on top of it, for example secondary indexing, complex filtering (push down predicates), and access control. Coprocessors can be loaded globally on all tables and regions hosted by the region server, these are known as system coprocessors; or the administrator can specify which coprocessors should be loaded on all regions for a table on a per-table basis, these are known as tablecoprocessors. In order to support sufficient flexibility for potential coprocessor behaviors, two different aspects of extension are provided by the framework. One is the observer, which are like triggers in conventional databases, and the other is the endpoint, dynamic RPC endpoints that resemble stored procedures.

Observers

The idea behind observers is that we can insert user code by overriding upcall methods

provided by the coprocessor framework. The callback functions are executed from core HBase code when certain events occur.The coprocessor framework handles all of the details of invoking callbacks during various base HBase activities; the coprocessor need only insert the desired additional or alternate functionality.

The RegionObserver interface provides callbacks for:

- preOpen, postOpen: Called before and after the region is reported as online to themaster.
- preFlush, postFlush: Called before and after the memstore is flushed into a new storefile.
- preGet, postGet: Called before and after a client makes a Getrequest.
- preExists, postExists: Called before and after the client tests for existence using aGet.
- prePut and postPut: Called before and after the client stores avalue.
- preDelete and postDelete: Called before and after the client deletes a value etc.

Endpoint

As mentioned previously, observers can be thought of like database triggers. Endpoints, on the other hand, are more powerful, resembling stored procedures. One can invoke an endpoint at any time from theclient.The endpoint implementation will then be executed remotely at the target region or regions, and results from those executions will be returned to theclient.Endpoint is an interface for dynamic RPC extension. The endpoint implementation is installed on the server side and can then be invoked with HBase RPC. The client library provides convenience methods for invoking such dynamicinterfaces.

In order to build and use your own endpoint, you need to:

- Have a new protocol interface which extendsCoprocessorProtocol.
- Implement the Endpoint interface. The implementation will be loaded into and executed from the regioncontext.
- Extend the abstract class BaseEndpointCoprocessor. This convenience class hides some internal details that the implementer need not necessary be concerned about, such as coprocessor framework classloading.
- On the client side, the Endpoint can be invoked by two new HBase clientAPIs:

Executing against a singleregion:

```
o HTableInterface.coprocessorProxy(Class<T> protocol, byte[]row)
```

Executing over a range of regions

```
o HTableInterface.coprocessorExec(Class<T> protocol, startKey,byte[]
   byte[] endKey, Batch.Call<T,R> callable)
```

Coprocessor Management

After you have a good understanding of how coprocessors work in HBase, you can start to build your own experimental coprocessors, deploy them to your HBase cluster, and observe the new behaviors.

Build Your Own Coprocessor

We now assume you have your coprocessor code ready, compiled and packaged as a jar file.

Coprocessor Deployment

Currently we provide two options for deploying coprocessor extensions: load from configuration, which happens when the master or region servers start up; or load from table attribute, dynamic loading when the table is (re)opened. Because most users will set table attributes by way of the_alter‘ command of the HBase shell, let‘s call this load from shell.

Load from Configuration

When a region is opened, the framework tries to read coprocessor class names supplied as the configuration entries:

- hbase.coprocessor.region.classes: for RegionObservers andEndpoints
- hbase.coprocessor.master.classes: for MasterObservers
- hbase.coprocessor.wal.classes: forWALObservers

Here is an example of the hbase-site.xml where one RegionObserver is configured for all the HBase tables:

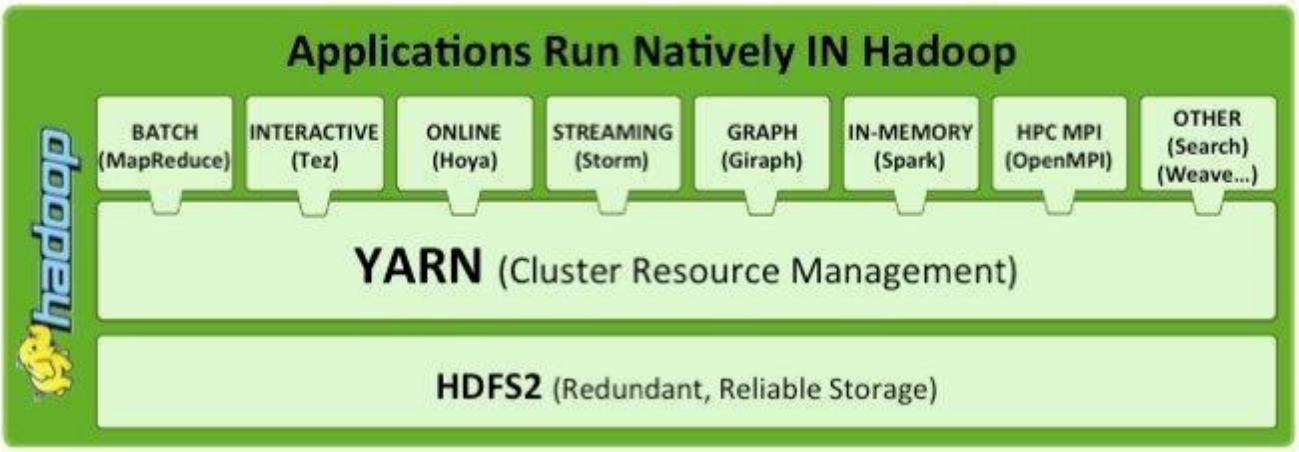
```
<property>
```

```
<name>hbase.coprocessor.region.classes</name>

<value>org.apache.hadoop.hbase.coprocessor.AggregateImplementation</value>
```

5.5 Hadoop 2.0

Apache Hadoop 2.0 represents a generational shift in the architecture of Apache Hadoop. With YARN, Apache Hadoop is recast as a significantly more powerful platform – one that takes Hadoop beyond merely batch applications to taking its position as a _data operating system‘ where HDFS is the file system and YARN is the operating system.YARN is a re-architecture of Hadoop that allows multiple applications to run on the same platform. With YARN, applications run ―in‖ Hadoop, instead of ―on‖ Hadoop:



The fundamental idea of YARN is to split up the two major responsibilities of the JobTracker and TaskTracker into separate entities. In Hadoop 2.0, the JobTracker and TaskTracker no longer exist and have been replaced by threecomponents:

- **ResourceManager:** a scheduler that allocates available resources in the cluster amongst the competingapplications.
- **NodeManager:** runs on each node in the cluster and takes direction from the ResourceManager. It is responsible for managing resources available on a singlenode.
- **ApplicationMaster:** an instance of a framework-specific library, an ApplicationMaster runs a specific YARN job and is responsible for negotiating resources from theResourceManager and also working with the NodeManager to execute and monitor Containers.

The actual data processing occurs within the Containers executed by the ApplicationMaster. A Container grants rights to an application to use a specific amount of resources (memory, cpu etc.) on a specific host.

YARN is not the only new major feature of Hadoop 2.0. HDFS has undergone a major transformation with a collection of new features that include:

- **NameNode HA:** automated failover with a hot standby and resiliency for the NameNode masterservice.
- **Snapshots:** point-in-time recovery for backup, disaster recovery and protection against useerrors.
- **Federation:** a clear separation of namespace and storage by enabling generic block storagelayer.

MRv2-YARN

The new architecture introduced in hadoop-0.23, divides the two major functions of the JobTracker: resource management and job life-cycle management into separate components.The new ResourceManager manages the global assignment of compute resources to applications and the per-application ApplicationMaster manages the application‘s scheduling and coordination. An application is either a single job in the sense of classic MapReduce jobs or a DAG of such jobs.The

ResourceManager and per-machine NodeManager daemon, which manages the user processes on that machine, form the computation fabric.The per-application ApplicationMaster is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks. The fundamental idea of YARN is to split up the functionalities of resource management and job scheduling/monitoring into separate daemons. The idea is to have a global ResourceManager
(*RM*) and per-application ApplicationMaster (*AM*). An application is either a single job or a DAG ofjobs.The ResourceManager and the NodeManager form the data-computation framework. The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system.The NodeManager is the per-machine framework agent who is responsible for containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager/Scheduler.The per-application ApplicationMaster is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks.

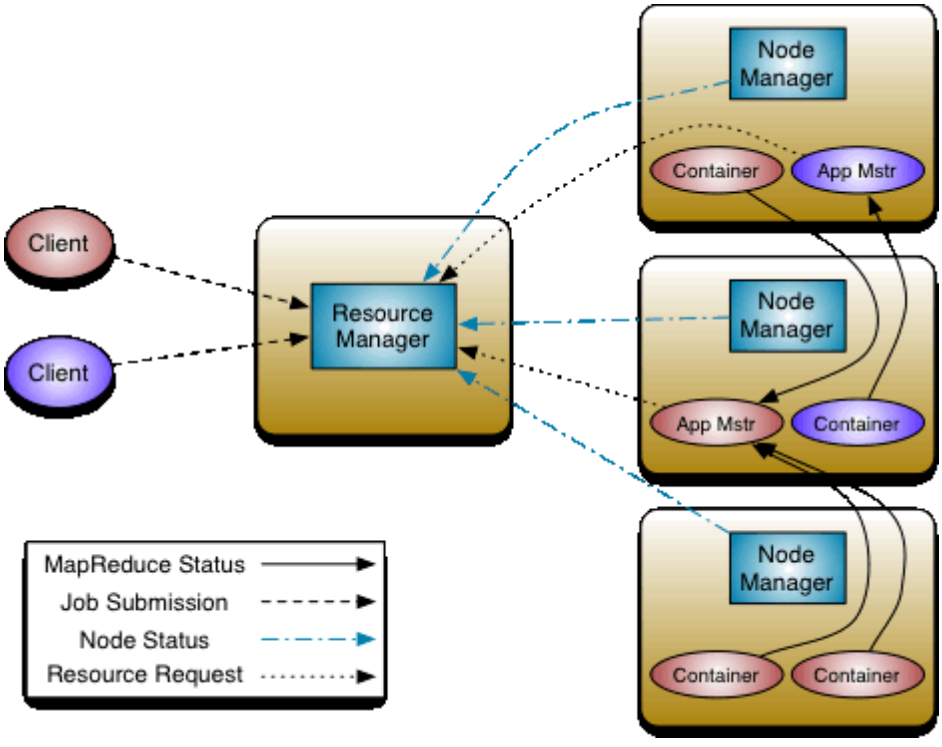


Figure 5.12 YARN workflow

The ResourceManager has two main components: Scheduler and ApplicationsManager. The Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is pure scheduler in the sense that it performs no monitoring or tracking of status for the application. Also, it offers no guarantees about restarting failed tasks either due to application failure or hardware failures.The Scheduler performs its scheduling function based the resource requirements of the applications; it does so based on the abstract notion of a resource *Container* which incorporates elements such as memory, cpu, disk, network etc.

5.6 NameNode High Availablity

Prior to Hadoop 2.0.0, the NameNode was a single point of failure (SPOF) in an HDFS cluster. Each cluster had a single NameNode, and if that machine or process became unavailable, the cluster as a whole would be unavailable until the NameNode was either restarted or brought up on a separate machine.

This impacted the total availability of the HDFS cluster in two major ways:

- In the case of an unplanned event such as a machine crash, the cluster would be unavailable until an operator restarted theNameNode.
- Planned maintenance events such as software or hardware upgrades on the NameNode machine would result in windows of cluster downtime.

The HDFS High Availability feature addresses the above problems by providing the option of running two redundant NameNodes in the same cluster in an Active/Passive configuration with a hot standby.This allows a fast failover to a new NameNode in the case that a machine crashes, or a graceful administrator-initiated failover for the purpose of planned maintenance.

Architecture

In a typical HA cluster, two separate machines are configured as NameNodes. At any point in time, exactly one of the NameNodes is in an *Active* state, and the other is in a *Standby* state. The Active NameNode is responsible for all client operations in the cluster, while the Standby is simply acting as a slave, maintaining enough state to provide a fast failover if necessary.In order for the Standby node to keep its state synchronized with the Active node, the current implementation requires that the two nodes both have access to a directory on a shared storage device (eg an NFS

mount from a NAS). This restriction will likely be relaxed in future versions.When any namespace modification is performed by the Active node, it durably logs a record of the modification to an edit log file stored in the shared directory. The Standby node is constantly watching this directory for edits, and as it sees the edits, it applies them to its own namespace.In the event of a failover, the Standby will ensure that it has read all of the edits from the shared storage before promoting itself to the Active state. This ensures that the namespace state is fully synchronized before a failover occurs.

In order to provide a fast failover, it is also necessary that the Standby node have up-todate information regarding the location of blocks in the cluster. In order to achieve this, the DataNodes are configured with the location of both NameNodes, and send block location information and heartbeats toboth.

Hardware resources

In order to deploy an HA cluster, you should prepare the following:

- **NameNode machines - the machines on which you run the Active and Standby NameNodes should have equivalent hardware to each other, and equivalent** hardware to what would be used in a non-HAcluster.
- **Shared storage** - you will need to have a shared directory which both NameNode machines can have read/write access to. Typically this is a remote filer whichsupportsNFS and is mounted on each of the NameNode machines. Currently only a single shared edits directory is supported.Thus, the availability of the system is limited by the availability of this shared edits directory, and therefore in order to remove all single points of failure there needs to be redundancy for the shared edits directory.

5.7HDFS Federation

HDFS Federation improves the existing **HDFS** architecture through a clear separation of namespace and storage, enabling generic block storage layer. It enables support for multiple namespaces in the cluster to improve scalability and isolation

This guide provides an overview of the HDFS Federation feature and how to configure and manage the federated cluster.

Background

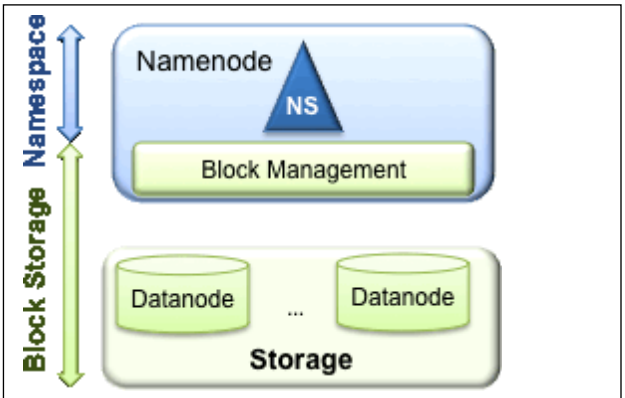


Figure 5.13 Block storage

HDFS has two main layers:

- **Namespace**
  - Consists of directories, files andblocks.
  - It supports all the namespace related file system operations such as create,delete, modify and list files anddirectories.
- **Block Storage Service**, which has two parts:
  - Block Management (performed in theNamenode)
    - Provides Datanode cluster membership by handling registrations,and periodic heartbeats.
    - Processes block reports and maintains location ofblocks.
    - Supports block related operations such as create, delete, modify andget blocklocation.
    - Manages replica placement, block replication for under replicatedblocks, and deletes blocks that are over replicated.
  - Storage - is provided by Datanodes by storing blocks on the local file systemand allowing read/writeaccess.

The prior HDFS architecture allows only a single namespace for the entire cluster. In that configuration, a single Namenode manages the namespace. HDFS Federation addresses this limitation by adding support for multiple Namenodes/namespaces to HDFS.

Key Benefits

- Namespace Scalability - Federation adds namespace horizontal scaling. Large deployments or deployments using lot of small files benefit from namespace scaling by allowing more Namenodes to be added to thecluster.
- Performance - File system throughput is not limited by a single Namenode. Adding more Namenodes to the cluster scales the file system read/writethroughput.
- Isolation - A single Namenode offers no isolation in a multi user environment. For example, an experimental application can overload the Namenode and slow down production critical applications. By using multiple Namenodes, different categories of applications and users can be isolated to different namespaces.

Federation configuration is **backward compatible** and allows existing single Namenode configurations to work without any change. The new configuration is designed such that all the nodes in the cluster have the same configuration without the need for deploying different configurations based on the type of the node in the cluster.

Configuration:

Step 1: Add the dfs.nameservices parameter to your configuration and configure it with a list of comma separated NameServiceIDs. This will be used by the Datanodes to determine the Namenodes in the cluster.

Step 2: For each Namenode and Secondary Namenode/BackupNode/Checkpointer add the following configuration parameters suffixed with the corresponding NameServiceID into the common configuration file:

Formatting Namenodes

Step 1: Format a Namenode using the following command:

[hdfs]$ $HADOOP_PREFIX/bin/hdfsnamenode -format [-clusterId<cluster_id>]

Choose a unique cluster_id which will not conflict other clusters in your environment. If a cluster_id is not provided, then a unique one is auto generated.

Step 2: Format additional Namenodes using the following command:

[hdfs]$ $HADOOP_PREFIX/bin/hdfsnamenode -format -clusterId<cluster_id>

Upgrading from an older release and configuring federation

Older releases only support a single Namenode. Upgrade the cluster to newer release in order to enable federation During upgrade you can provide a ClusterID as follows:

[hdfs]$ $HADOOP_PREFIX/bin/hdfs start namenode --config $HADOOP_CONF_DIR -upgrade -clusterId<cluster_ID>

If cluster_id is not provided, it is auto generated. **Adding a new Namenode to an existing HDFScluster** Perform the followingsteps:

- Add dfs.nameservices to theconfiguration.
- Update the configuration with the NameServiceID suffix. Configuration key names changed post release 0.20. You must use the new configuration parameter names inorder to usefederation.
- Add the new Namenode related config to the configurationfile.
- Propagate the configuration file to the all the nodes in thecluster.
- Start the new Namenode andSecondary/Backup.
- Refresh the Datanodes to pickup the newly added Namenode by running thefollowing command against all the Datanodes in thecluster:

  [hdfs]$ $HADOOP_PREFIX/bin/hdfsdfsadmin -refreshNameNodes <datanode_host_name>:<datanode_rpc_port>

Managing the cluster

Starting and stopping cluster

To start the cluster run the followingcommand:

[hdfs]$ $HADOOP_PREFIX/sbin/start-dfs.sh To stop the cluster run the followingcommand:

[hdfs]$ $HADOOP_PREFIX/sbin/stop-dfs.sh

Balancer

The Balancer has been changed to work with multiple Namenodes. The Balancer can be run using the command:

[hdfs]$ $HADOOP_PREFIX/sbin/hadoop-daemon.sh start balancer [-policy <policy>]

Decommissioning

Decommissioning is similar to prior releases. The nodes that need to be decomissioned are added to the exclude file at all of the Namenodes. Each Namenode decommissions its Block Pool.When all the Namenodes finish decommissioning a Datanode, the Datanode is considered decommissioned.

Step 1: To distribute an exclude file to all the Namenodes, use the following command:

[hdfs]$ $HADOOP_PREFIX/sbin/distribute-exclude.sh <exclude_file>

Step 2: Refresh all the Namenodes to pick up the new exclude file:

[hdfs]$ $HADOOP_PREFIX/sbin/refresh-namenodes.sh

The above command uses HDFS configuration to determine the configured Namenodes in the cluster and refreshes them to pick up the new exclude file.

Cluster Web Console

Similar to the Namenode status web page, when using federation a Cluster Web Console is available to monitor the federated cluster at http://<any_nn_host:port>/dfsclusterhealth.jsp. Any Namenode in the cluster can be used to access this web page.

Migrating from MapReduce 1 (MRv1) to MapReduce 2

MapReduce from Hadoop 1 (MapReduce MRv1) has been split into two components. The cluster resource management capabilities have become YARN (Yet Another Resource Negotiator), while the MapReduce-specific capabilities remainMapReduce.In the MapReduce MRv1 architecture, the cluster was managed by a service called the JobTracker. TaskTracker services

lived on each host and would launch tasks on behalf of jobs. The JobTracker would serve information about completed jobs.

In MapReduce MRv2, the functions of the JobTracker have been split between three services. The ResourceManager is a persistent YARN service that receives and runs applications (a MapReduce job is an application) on the cluster.It contains the scheduler, which, as previously, is pluggable. The MapReduce-specific capabilities of the JobTracker have been moved into the MapReduce Application Master, one of which is started to manage each MapReduce job and terminated when the jobcompletes.The JobTracker function of serving information about completed jobs has been moved to the JobHistory Server. The TaskTracker has been replaced with the NodeManager, a YARN service that manages resources and deployment on a host. It is responsible for launching containers, each of which can house a map or reduce task.Nearly all jobs written for MRv1 will be able to run without any modifications on an MRv2 cluster.

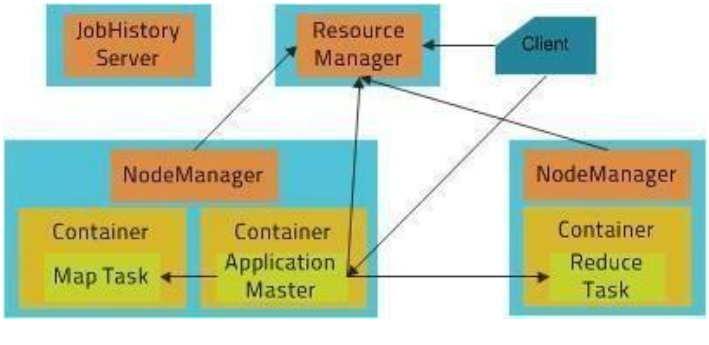


Figure 5.14 MRV2

The new architecture has its advantages. First, by breaking up the JobTracker into a few different services, it avoids many of the scaling issues faced by MapReduce in Hadoop 1.More importantly, it makes it possible to run frameworks other than MapReduce on a Hadoop cluster.
For example, Impala can also run on YARN and share resourceswith MapReduce.

Configuration Migration

Since MapReduce 1 functionality has been split into two components, MapReduce cluster configuration options have been split into YARN configuration options, which go in yarn- site.xml, and MapReduce configuration options, which go in mapred-site.xml. Many have been given new names to reflect the shift. As JobTrackers and TaskTrackers no longer exist in MRv2, all

configuration options pertaining to them no longer exist, although many have corresponding options for the ResourceManager, NodeManager, and JobHistoryServer.

A minimal configuration required to run MRv2 jobs on YARN is: yarn-site.xml configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>you.hostname.com</value>
  </property>

  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
</configuration>
```

mapred-site.xml configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

Below is a table with HA-related configurations used in MRv1 and their equivalents in YARN:

MRv1	YARN / MRv2	Comment
mapred.jobtrackers.*name*	yarn.resourcemanager.ha.rm-ids	
mapred.ha.jobtracker.id	yarn.resourcemanager.ha.id	Unlike in MRv1, this must be configured in YARN.
mapred.jobtracker.*rpc-address.name.id*	yarn.resourcemanager.*rpc-address.id*	YARN/ MRv2 has different RPC ports for different functionalities. Each port-related configuration must be suffixed with an id. Note that there is no <name> in YARN.

mapred.ha.jobtracker.*rpc-address.name.id*	yarn.resourcemanager.ha.admin.address	
mapred.ha.fencing.methods	yarn.resourcemanager.ha.fencer	Not required to be specified
mapred.client.failover.*	None	Not required
	yarn.resourcemanager.ha.enabled	Enable HA
mapred.jobtracker.restart.recover	yarn.resourcemanager.recovery.enabled	Enable recovery of jobs after failover
	yarn.resourcemanager.store.class	org.apache .hadoop.yarn
MRv1	**YARN / MRv2**	**Comment**
		.server.resourcemanager .recovery .ZKRMStateStore
mapred.ha.automatic-failover.enabled	yarn.resourcemanager.ha.automatic-failover.enabled	Enable automatic failover
mapred.ha.zkfc.port	yarn.resourcemanager.ha.automatic-failover.port	
mapred.job.tracker	yarn.resourcemanager.cluster.i	

5.8 Programming in YARN Framework

Under Hadoop 2.0, MapReduce is but one instance of a YARN application, where YARN has taken center stage as the ―operating system‖ of Hadoop. BecauseYARN allows*any* application to run on equal footing with MapReduce, it opened the floodgates for a new generation of software applications with these kinds of features:

More programming models. Because YARN supports any application that can divide itself into parallel tasks,they are no longer shoehorned into the palette of ―mappers,‖―combiners,‖and ―reducers.‖ This in turn supports complex data-flow applications like ETL and ELT, and iterative programs like massively-parallel machine learning and modeling.

Integration of native libraries. Because YARN has robust support for *any* executable – not limited to MapReduce, and not even limited to Java – application vendors with a large mature code base have a clear path to Hadoop integration.

Support for large reference data. YARN automatically ―localizes‖ and caches large reference datasets, making them available to all nodes for ―data local‖ processing. This supports legacy functions like address standardization, which require large reference data sets that cannot be accessed from the Hadoop Distributed File System (HDFS) by the legacy libraries. Of course, MapReduce isn‘t the only option for processing data at scale using Hadoop. Toolslike Pig (a large scale query and analysis system), Hive (a data warehousing application) and others have been available for some time. These tools can express transforms and analysis using more accessible constructs: Hive uses HQL, a language similar to SQL.

Pig provides a script language (Pig Latin) to create MapReduce jobs. Business analysts familiar with conventional tools like SQL and SAS should be able to use these tools to write programs to solve large data problems on Hadoop clusters **The Value of Visual Application Development Tools**

A new generation of ―visual design‖application development tools could help solve thesecoding problems. By running as native YARN applications and side-stepping the need for MapReduce, some of these programs eliminate coding altogether.Other tools reduce coding by generating MapReduce code or by generating scripts like Pig. Visual designers are powerful for several reasons:

- Increased level of abstraction: Instead of thinking about classes and methods, users see operations, data, and outcomes.
- Fast―what-if‖:Thedrag-and-connectinterfacesupportsquicktry/observe/adjustcycles.
- Automatic optimization: Scaling and efficiency arebuilt-in.
- High-levelpalette:High-levelconstructslike―standardizeaddress‖,―deduplicateconsumers‖, or―parsenames‖ areoften directlyon thedesignerpalette.

How does this look in practice? Here‘s an illustration that shows how three competing approaches differ:

- MapReduce written inJava
- Pig scripts developed fromscratch
- A visually-designed process running a native YARN ETL application. The application is from RedPoint Global, but comparable approaches can be seen in Talend andActian.

Using these three approaches, we conducted a ―Word Count‖ test on 30,000 files (20 gigabytes) of Project Gutenberg books. This test reads lines of text, breaks them into words, and creates a concordance (list of words and the number of times each occurs).Our Hadoop cluster was small—only four nodes—but was large enough to demonstrate the concepts and tradeoffs **MapReduce:**

Set-up time: While flexible, MapReduce had the longest learning curve and required significant coding skills—both as a Java programmer and a MapReduce specialist—to prepare the test. Performance: It took 3 hours 20 minutes to run the test initially due to the ―small files problem‖ that is familiar to seasoned MapReduce programmers.This problem occurs when reading large collections of small files, because MapReduce's default behavior is to assign a mapper task to each file. This results in a huge number of tasks.To address this issue, we created a custom InputFormat class to read multiple files at once. This reduced our run time to 58 minutes. Then we tuned the split sizes and mapper task limit appropriately, which dropped the run time to about six minutes.
Comments: Each performance improvement came at a cost. Overall, nearly a full day of programmer time was spent optimizing the original code.

Pig:

Set-up time: Learning Pig was fairly easy. It was pretty natural to create the coding for this test. However to make a common adjustment in the code—changing the set of whitespace separators to include punctuation—required the addition of a ―User defined function‖ or UDF which had to be written in Java.

Pig is generally easy enough to use by people who aren't professional programmers but who know how to write scripting languages like JavaScript or Visual Basic.

Performance: The results were not stellar: run time was close to 15 minutes.

Comments: While coding took less time, Java programming was ultimately required to meet the test requirements.

Sample Pig script without the User Defined Function:

```
SET pig.maxCombinedSplitSize 67108864
SET pig.splitCombination true
A = LOAD '/testdata/pg/*/*/*';
B = FOREACH A GENERATE FLATTEN(TOKENIZE((chararray)$0)) AS word;
C = FOREACH B GENERATE UPPER(word) AS word;
D = GROUP C BY word;
E = FOREACH D GENERATE COUNT(C) AS occurrences, group;
F = ORDER E BY occurrences DESC;
STORE F INTO '/user/cleonardi/pg/pig-count';
```

YARN-enabled ETL/ELT designer:

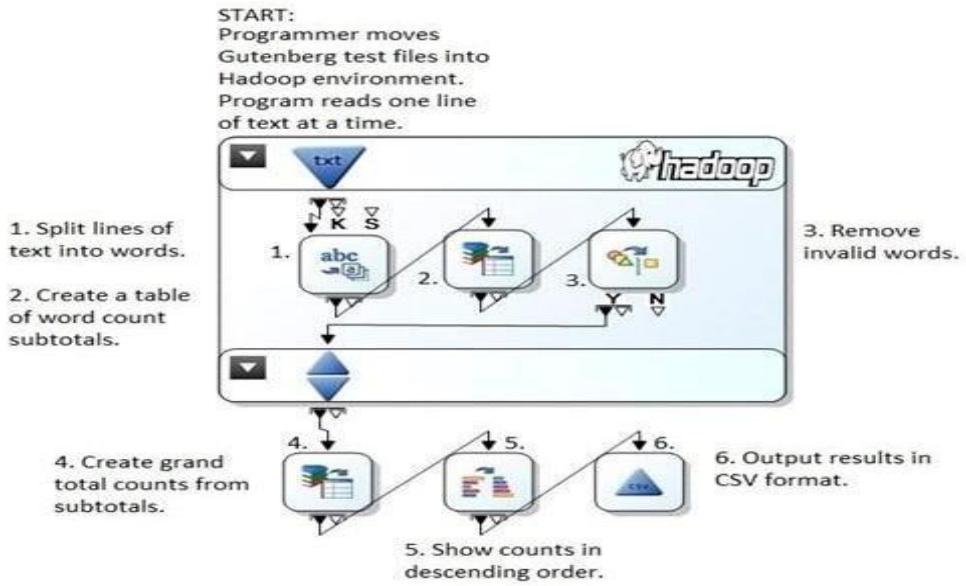
Set-up time: The tool is designed to have a shorter learning curve than even Pig scripting. Dragging tools like ―Delimited Input‖, ―Summarize‖ and ―Tokenize‖ from the palette and configuring them is designed to be discoverable and intuitive, and the resulting diagram has a one-to-one correspondence between icons and operations. There‘s no need for coding or learning a language like Java or Pig.

The visual design covers the input file format, tokenizing and counting steps. The resulting data flow graph contains seven icons along with a grouping construct that shows what executes ―inside‖Hadoop.Each icon represents a stepin the data transformation.

Performance: The run time for this data flow is just over three minutes with no tuning.

*Comments:*Because there is no code to manage, and editing is done visually, running ―what if‖ scenarios is quick for non-programmers.

Once the data flow is designed, it can be stored and saved for later use. In addition, the logic can be captured into a ―macro‖ for sharing and reuse between multiple data flows.



5.9 Oozie workflow scheduler for Hadoop

*Oozie*is a framework that helps automate this process and codify this work into repeatable units or workflows that can be reused over time without the need to write any new code or steps.

Apache Oozie is an open source project based on Java™ technology that simplifies the process of creating workflows and managing coordination among jobs. In principle, Oozie offers the ability to combine multiple jobs sequentially into one logical unit ofwork.One advantage of the Oozie framework is that it is fully integrated with the Apache Hadoop stack and supports Hadoop jobs for Apache MapReduce, Pig, Hive, andSqoop.

In addition, it can be used to schedule jobs specific to a system, such as Java programs.In practice, there are different types of Oozie jobs:

- *Oozie Workflow* jobs — Represented as directed acyclical graphs to specify a sequence of actions to beexecuted.
- *Oozie Coordinator* jobs — Represent Oozie workflow jobs triggered by time and data availability.
- *Oozie Bundle*— Facilitates packaging multiple coordinator and workflow jobs, and makes it easier to manage the life cycle of those jobs.

How does Oozie work?

An Oozie workflow is a collection of actions arranged in a directed acyclic graph (DAG). This graph can contain two types of nodes: control nodes and action nodes. *Control nodes*, which are used to define job chronology, provide the rules for beginning and ending a workflow and control the workflow execution path with possible decision points known as fork and joinnodes.

Action nodes are used to trigger the execution of tasks. In particular, an action node can be a MapReduce job, a Pig application, a file system task, or a Java application. (The shell and ssh actions have been deprecated).Oozie is a native Hadoop stack integration that supports all types of Hadoop jobs and is integrated with the Hadoop stack.Figure shown below illustrates a sample Oozie workflow that combines six action nodes (Pig scrip, MapReduce jobs,Java code,and HDFStask) andfive control nodes (Start, Decisioncontrol, Fork, Join, and End). Oozie workflows can be also parameterized. When submitting a workflow job, values for the parameters must be provided. If the appropriate parameters are used, several identical workflow jobs can occurconcurrently.

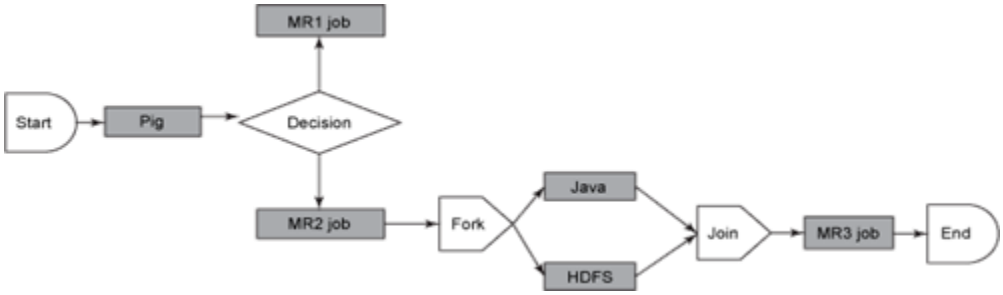


Figure 5.16 :sample Oozie workflow

In practice, it is sometimes necessary to run Oozie workflows on regular time intervals, but in coordination with other conditions, such as the availability of specific data or the completion of any other events or tasks.

Oozie in action

Use an Oozie workflow to run a recurring job. Oozie workflows are written as an XML file representing a directed acyclic graph. Let's look at the following simple workflow example that chains two MapReduce jobs. The first job performs an initial ingestion of the data and the second job merges data of a given type.

Simple example of Oozie workflow

```
<workflow-app xmlns='uri:oozie:workflow:0.1' name='SimpleWorkflow'> <start
        to='ingestor'/>
        <action name='ingestor'>
                </java>
                        <job-tracker>${jobTracker}</job-tracker>
                        <name-node>${nameNode}</name-node>
                        <configuration>
                                <property>
                                        <name>mapred.job.queue.name</name>
                                        <value>default</value>
                                </property>
                        </configuration>
                            <arg>${driveID}</arg>
                    </java>
                    <ok to='merging'/>
                    <error to='fail'/>
            </action>
            <fork name='merging'>
                    <pathstart='mergeT1'/>
                    <pathstart='mergeT2'/>
            </fork>
            <action name='mergeT1'>
                    <java>
                        <job-tracker>${jobTracker}</job-tracker>
                        <name-node>${nameNode}</name-node>
                        <configuration>
                                <property>
                                        <name>mapred.job.queue.name</name>
                                        <value>default</value>
                                </property>
                        </configuration>
                        <arg>-drive</arg>
                        <arg>${driveID}</arg>
                        <arg>-type</arg>
                        <arg>T1</arg>
                        <ok to='completed'/>
                        <error to='fail'/>
            </action>
```

```
<action name='mergeT2'>
    <java>
        <job-tracker>${jobTracker}</job-tracker>
        <name-node>${nameNode}</name-node>
        <configuration>
            <property>
                <name>mapred.job.queue.
                name</name>
                <value>default</value>
            </property>
        </configuration> <main-
class>com.navteq.assetmgmt.hdfs.mer ge.MergerLoader</main-
class>
        <arg>-drive</arg>
        <arg>${driveID}</arg>
        <arg>-type</arg>
        <arg>T2</arg>
    </java>
    <ok to='completed'/>
    <error to='fail'/>
</action>
<join name='completed' to='end'/>
<kill name='fail'>
    <message>Java failed,
error
message[${wf:errorMessage(wf:lastErrorNo
de())}]</message>
</kill>
<end name='end'/> </workflow-app>
```


Reference :

1.WA Gmob, “Big Data and Hadoop”, Kindle Edition, 2013

2. Eric Miller, “A Overview of Map Reduce and its impact on Distributed Data”, Kindle Edition, 2012.

3. Strata, “ Big Data Now”, O’Reily Media Inc., Kindle Edition, 2012. https://www.guru99.com/hbase-architecture-data-flow-usecases.html

https://data-flair.training/blogs/hbase-client-api/ https://intellipaat.com/blog/tutorial/hbase-tutorial/installation/ https://www.tutorialspoint.com/hbase/hbase_create_data.htm

http://jcsites.juniata.edu/faculty/Rhodes/smui/hbase.htm