



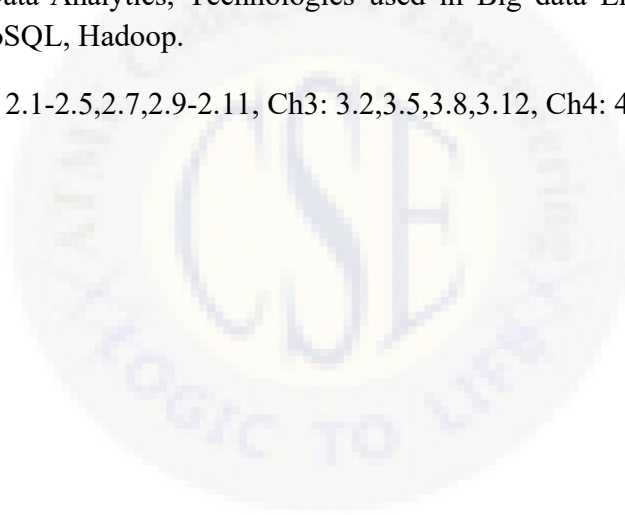
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

**SUBJECT TITLE AND CODE :BIG DATA
ANALYTICS(BCS714D)**

MODULE 1

Classification of data, Characteristics, Evolution and definition of Big data, What is Big data, Why Big data, Traditional Business Intelligence Vs Big Data, Typical data warehouse and Hadoop environment. Big Data Analytics: What is Big data Analytics, Classification of Analytics, Importance of Big Data Analytics, Technologies used in Big data Environments, Few Top Analytical Tools , NoSQL, Hadoop.

TB1: Ch 1: 1.1, Ch2: 2.1-2.5,2.7,2.9-2.11, Ch3: 3.2,3.5,3.8,3.12, Ch4: 4.1,4.2



MODULE-1

Classification of data, Characteristics, Evolution and definition of Big data, What is Big data, Why Big data, Traditional Business Intelligence Vs Big Data, Typical data warehouse and Hadoop environment. Big Data Analytics: What is Big data Analytics, Classification of Analytics, Importance of Big Data Analytics, Technologies used in Big data Environments, Few Top Analytical Tools , NoSQL, Hadoop. TB1: Ch 1: 1.1, Ch2: 2.1-2.5, 2.7, 2.9-2.11, Ch3: 3.2, 3.5, 3.8, 3.12, Ch4: 4.1, 4.2

What is DATA?

Data is a raw fact and is defined as individual facts, such as numbers, words, measurements, observations or just descriptions of things. For example, data might include individual prices, weights, addresses, ages, names, temperatures, dates, or distances.

There are two main types of data:

1. Quantitative data is provided in numerical form, like the weight, volume, or cost of an item.
2. Qualitative data is descriptive, but non-numerical, like the name, gender, or eye color of a person.

Characteristics of Data

The key characteristics of the data are:

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

2. Condition: The condition of data deals with the state of data, that is, "Can one use this data as is for analysis?" or "Does it require cleansing for further enhancement and enrichment?"

3. Context: The context of data deals with "Where has this data been generated?" "Why was this data generated?" "How sensitive is this data?" "What are the events associated with this data?" and so on. The other following are six other characteristics of data which are discussed below:

1. Accuracy
2. Validity
3. Reliability
4. Timeliness
5. Relevance
6. Completeness

1. Accuracy

Data should be sufficiently accurate for the intended use and should be captured only once, although it may have multiple uses. Data should be captured at the point of activity.

2. Validity

Data should be recorded and used in compliance with relevant requirements, including the correct application of any rules or definitions. This will ensure consistency between periods and with similar organizations, measuring what is intended to be measured.

3. Reliability

Data should reflect stable and consistent data collection processes across collection points and over time. Progress toward performance targets should reflect real changes rather than variations in data collection approaches or methods. Source data is clearly identified and readily available from manual, automated, or other systems and records.

4. Timeliness

Data should be captured as quickly as possible after the event or activity and must be available for the intended use within a reasonable time period. Data must be available quickly and frequently enough to support information needs and to influence service or management decisions.

5. Relevance

Data captured should be relevant to the purposes for which it is to be used. This will require a periodic review of requirements to reflect changing needs.

6. Completeness

Data requirements should be clearly specified based on the information needs of the organization and data collection processes matched to these requirements.

Classification of data

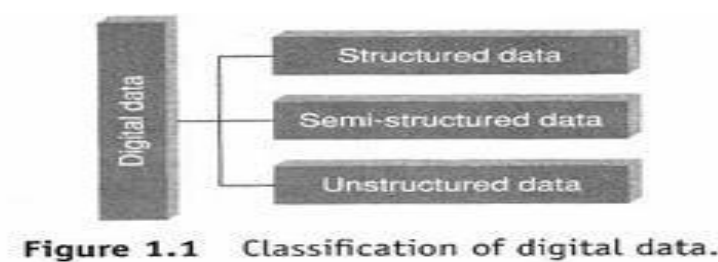


Figure 1.1 Classification of digital data.

1. Structured Data:

- Structured data refers to any data that resides in a fixed field within a record or file.
- Having a particular Data Model.

- Meaningful data
- Data arranged in a row and column.
- Structured data has the advantage of being easily entered, stored, queried and analysed.
- E.g.: Relational Data Base, Spread sheets.
- Structured data is often managed using Structured Query Language (SQL)

Sources of Structured Data:

- SQL Databases
- Spreadsheets such as Excel
- OLTP Systems
- Online forms
- Sensors such as GPS or RFID tags
- Network and Web server logs
- Medical devices

Ease of working with structured data:

1.Insert/update/delete: The Data Manipulation Language (DML)operations provide the required ease with data input, storage, access, process, analysis, etc.

2.Security: Information security is ensured through strong encryption and tokenization, protecting data throughout its lifecycle. Access control measures ensure only authorized individuals can decrypt and view sensitive information, maintaining compliance.

3.Indexing: An index is a data structure that speeds up the data retrieval operations (primarily the SELECT DML statement) at the cost of additional writes and storage space, but the benefits that ensue in search operation are worth the additional writes and storage space.

4. Scalability: The storage and processing capabilities of the traditional RDBMS can be easily scaled up by increasing the horsepower of the database server (increasing the primary and secondary or peripheral storage capacity, processing capacity of the processor, etc.).

5.Transaction processing: RDBMS has support for Atomicity,

Consistency, Isolation, and Durability (ACID) properties of transaction.

Atomicity: A transaction is atomic, means that either it happens in its entirety or none of it at all.

Consistency: The database moves from one consistent state to another consistent state. In other words, if the same piece of information is stored at two or more places, they are in complete agreement.

Isolation: The resource allocation to the transaction happens such that the transaction gets the impression

that it is the only transaction happening in isolation

Durability: All changes made to the database during a transaction are permanent and that accounts for the durability of the transaction.

1. Unstructured data:

- Unstructured data can not readily classify and fit into a neat box
- Also called unclassified data.
- Which does not confirm to any data model.
- Business rules are not applied.
- Indexing is not required.

E.g.: photos and graphic images, videos, streaming instrument data, webpages, Pdf files, PowerPoint presentations, emails, blog entries

Sources of Unstructured Data:

- Web pages
- Images (JPEG, GIF, PNG, etc.)
- Videos
- Memos
- Reports
- Word documents and PowerPoint presentations
- Surveys

HOW TO DEAL WITH UNSTRUCTURED DATA



1. Data mining: First, we deal with large data sets. Second, we use methods at the intersection of artificial intelligence, machine learning, statistics, and database systems to unearth consistent patterns in large data sets and/or systematic relationships between variables. It is the analysis step of the "knowledge discovery in databases" process.

Few popular data mining algorithms are as follows:

- Association rule mining: It is also called "market basket analysis" or "affinity analysis". It is used to determine "What goes with what?" It is about when you buy a product, what is the other product that you are likely to purchase with it. For example, if you pick up bread from the grocery, are you likely to pick eggs or cheese to go with it.
- Regression analysis: It helps to predict the relationship between two variables. The variable whose value needs to be predicted is called the dependent variable and the variables which are used to predict the value are referred to as the independent variables.
- Collaborative filtering: It is about predicting a user's preference or preferences based on the preferences of a group of users.

2. Text analytics or text mining: Compared to the structured data stored in relational databases, text is largely unstructured, amorphous, and difficult to deal with algorithmically. Text mining is the process of extracting high quality and meaningful information (through devising of patterns and trends by means of statistical pattern learning) from text. It includes tasks such as text categorization, text clustering, sentiment analysis, concept/entity extraction, etc.

3. Natural language processing (NLP): It is related to the area of human computer interaction. It is about enabling computers to understand human or natural language input.

4. Noisy text analytics: It is the process of extracting structured or semi-structured information from noisy unstructured data such as chats, blogs, wikis, emails, message boards, text messages, etc. The noisy unstructured data usually comprises one or more of the following: Spelling mistakes, abbreviations, acronyms, non standard words, missing punctuation, missing letter case, filler words such as "uh", "um", etc.

2. Semi-structured data:

- Self-describing data.
- Metadata (Data about data).
- Also called quiz data: data in between structured and semi structured.
- It is a type of structured data but not followed data model.
- Data which does not have rigid structure.
- E.g.: E-mails, word processing software.
- XML and other markup language are often used to manage semi structured Data.

Sources of semi-structured Data:

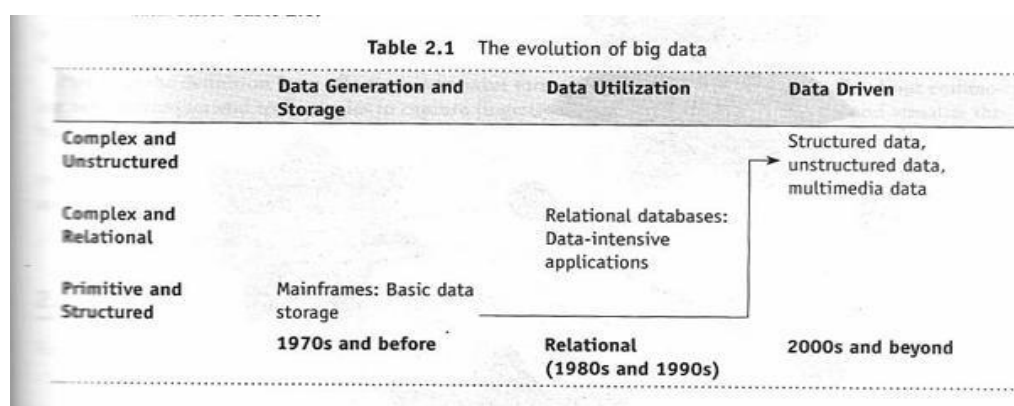
- E-mails
- XML and other markup languages
- Binary executables
- TCP/IP packets
- Zipped files
- Integration of data from different sources
- Web pages

It has the following features:

1. It does not conform to the data models that one typically associates with relational databases or any other form of data tables.
2. It uses tags to segregate semantic elements.
3. Tags are also used to enforce hierarchies of records and fields within data.
4. There is no separation between the data and the schema. The amount of structure used is dictated by the purpose at hand.
5. In semi-structured data, entities belonging to the same class and also grouped together need not necessarily have the same set of attributes. And if at all, they have the same set of attributes, the order of attributes may not be similar and for all practical purposes it is not important as well.

Evolution of Big Data:

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



Definition of Big data

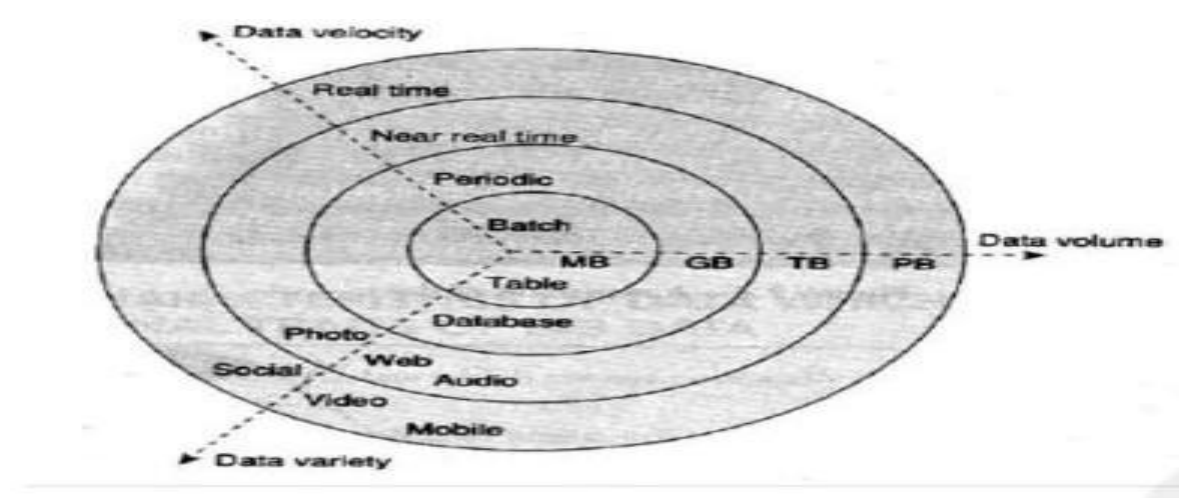
Big data is high-volume, high-velocity, and high-variety information assets that demand cost effective, innovative forms of information processing for enhanced insight and decision making.

or

Big Data is a collection of data that is huge in volume, yet growing exponentially with time. It is a data with so large size and complexity that none of traditional data management tools can store it or process it efficiently. Big data is also a data but with huge size.

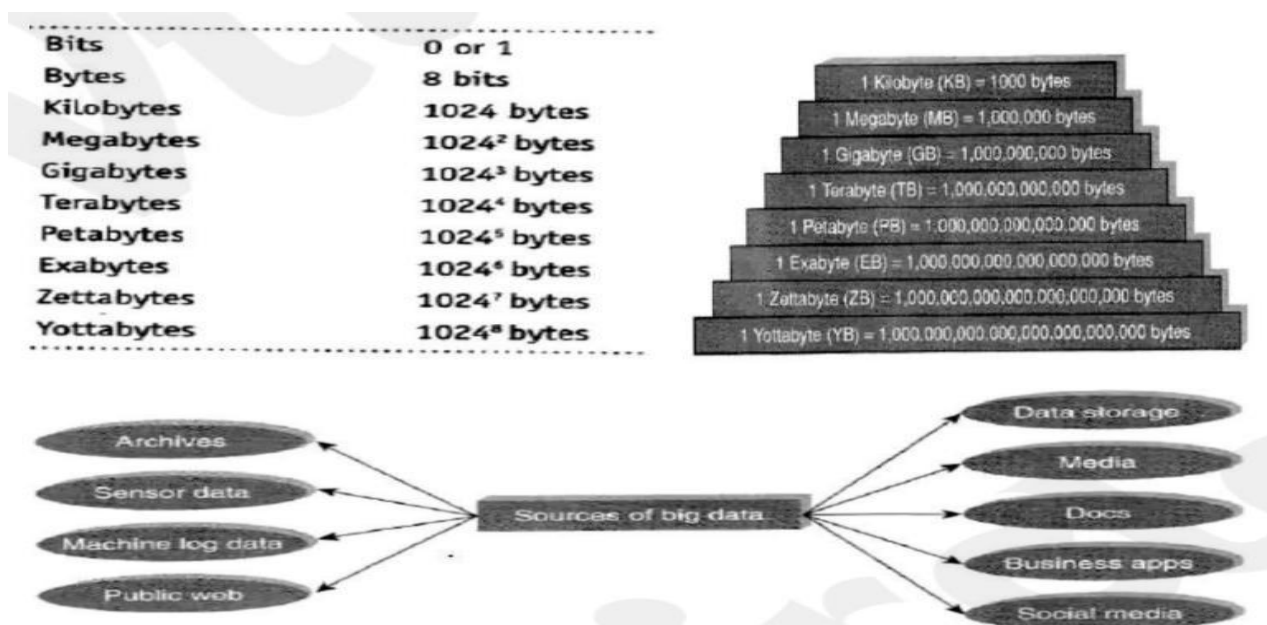
What is Big data?

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



**Bits → Bytes → Kilobytes → Megabytes → Gigabytes → Terabytes
→ Petabytes → Exabytes → Zettabytes → Yottabytes**

The name Big Data itself is related to an enormous size. Big Data is a vast 'volume' of data generated from many sources daily, such as business processes, machines, social media platforms, networks, human interactions, and many more.



1. Typical internal data sources:

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

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

2. External data sources: Data residing outside an organization's firewall like Public Web: Wikipedia, weather, regulatory, compliance, census, etc.

3. Both (internal + external data sources)

- Sensor data: Car sensors, smart electric meters, office buildings, air conditioning units, refrigerators, and so on.
- Machine log data: Event logs, application logs, Business process logs, audit logs, click stream data, etc.
- Social media: Twitter, blogs, Facebook, LinkedIn, YouTube, Instagram, etc.
- Business apps: ERP, CRM, HR, Google Docs, and so on.
- Media: Audio, Video, Image, Podcast, etc.
- Docs: Comma separated value (CSV), Word Documents, PDF, XLS, PPT, and so on.

2. Velocity:

We have moved from batch processing to real time processing.

Batch → Periodic → Near real time → Real-time processing

3. Variety:

Variety deals with a wide range of data types and sources of data like Structured data, semi-structured data and unstructured data.

Other characteristics of data

1. Veracity and Validity:

Veracity refers to biases, noise and abnormality in data. Validity refers to accuracy and correctness of the data.

2. Volatility:

Volatility of data deals with, how long is the data valid and how long should it be stored. There is some data that is required for long-term decisions and remains valid for longer periods of time. However, there are also pieces of data that quickly become obsolete minutes after their generation.

3. Variability:

Data flows can be highly inconsistent with periodic peaks.

Why Big data?



It helps organizations to:

- To understand Where, When and Why their customers buy
- Protect the company's client base with improved loyalty programs
- Seizing cross-selling and up selling opportunities
- Provide targeted promotional information
- Optimize Workforce planning and operations
- Improve inefficiencies in the company's supply chain

- Predict market trends
- Predict future needs
- Make companies more innovative and competitive
- It helps companies to discover new sources of revenue

Importance of Big data

1. Cost Savings

Big Data tools like Apache Hadoop, Spark, etc. bring cost-saving benefits to businesses when they have to store large amounts of data. These tools help organizations in identifying more effective ways of doing business.

2. Time-Saving

Real-time in-memory analytics helps companies to collect data from various sources. Tools like Hadoop help them to analyze data immediately thus helping in making quick decisions based on the learnings.

3. Understand the market conditions

Big Data analysis helps businesses to get a better understanding of market situations. For example, analysis of customer purchasing behavior helps companies to identify the products sold most and thus produces those products accordingly. This helps companies to get ahead of their competitors.

4. Social Media Listening

Companies can perform sentiment analysis using Big Data tools. These enable them to get feedback about their company, that is, who is saying what about the company. Companies can use big data tools to improve their online presence.

5. Boost Customer Acquisition and Retention

Understanding customer needs is crucial for business success, as neglecting them can lead to loss of clientele and hinder growth. Big data analytics helps identify customer trends and behaviors, enabling businesses to make informed decisions and stay competitive.

6. Solve Advertisers Problem and Offer Marketing Insights

Big data analytics shapes all business operations. It enables companies to fulfill customer expectations. Big data analytics helps in changing the company's product line. It ensures powerful marketing campaigns.

7. The driver of Innovations and Product Development

Big data makes companies capable to innovate and redevelop their products.

Challenges with Big data



1. Managing massive amounts of data

It's in the name—big data is big. Most companies are increasing the amount of data they collect daily. Eventually, the storage capacity a traditional data center can provide will be inadequate, which worries many business leaders. Forty-three percent of IT decision-makers in the technology sector worry about this data influx overwhelming their infrastructure. To handle this challenge, companies are migrating their IT infrastructure to the cloud. Cloud storage solutions can scale dynamically as more storage is needed. Big data software is designed to store large volumes of data that can be accessed and queried quickly.

2. Integrating data from multiple sources

The data itself presents another challenge to businesses. There is a lot, but it is also diverse because it can come from a variety of different sources. A business could have analytics data from multiple websites, sharing data from social media, user information from CRM software, email data, and more. None of this data is structured the same but may have to be integrated and reconciled to gather necessary insights and create reports. To deal with this challenge, businesses use data integration software, ETL software, and business intelligence software to map disparate data sources into a common structure and combine them so they can generate accurate reports.

3. Ensuring data quality

Analytics and machine learning processes that depend on big data to run also depend on clean, accurate data to generate valid insights and predictions. If the data is corrupted or incomplete, the results may not be what you expect. But as the sources, types, and quantity of data increase, it can be hard to determine if the data has the quality you need for accurate insights. Fortunately, there are solutions for this. Data governance applications will help organize, manage, and secure the data you use in your big data projects while also validating data sources against what you expect them to be and cleaning up corrupted and incomplete data sets. Data quality software can also be used specifically for the task of validating and cleaning your data before it is processed.

4. Keeping data secure

Many companies handle data that is sensitive, such as:

- Company data that competitors could use to take a bigger market share of the industry
- Financial data that could give hackers access to accounts

If a business handles sensitive data, it will become a target of hackers. To protect this data from attack, businesses often hire cybersecurity professionals who keep up to date on security best practices and techniques to secure their systems. Whether you hire a consultant or keep it in-house, you need to ensure that data is encrypted, so the data is useless without an encryption key. Add identity and access authorization control to all resources so only the intended users can access it. Implement endpoint protection software so malware can't infect the system and real-time monitoring to stop threats immediately if they are detected.

5. Selecting the right big data tools

Fortunately, when a business decides to start working with data, there is no shortage of tools to help them do it. At the same time, the wealth of options is also a challenge. Big data software comes in many varieties, and their capabilities often overlap. How do you make sure you are choosing the right big data tools? Often, the best option is to hire a consultant who can determine which tools will fit best with what your business wants to do with big data. A big data professional can look at your current and future needs and choose an enterprise data streaming or ETL solution that will collect data from all your data sources and aggregate it. They can configure your cloud services and scale dynamically based on workloads. Once your system is set up with big data tools that fit your needs, the system will run seamlessly with very little maintenance. Thinking about hiring a data analytics company to help your business implement a big data strategy? Browse our list of top data analytics companies, and learn more about their services in our hiring guide.

6. Scaling systems and costs efficiently

If you start building a big data solution without a well-thought-out plan, you can spend a lot of money storing and processing data that is either useless or not exactly what your business needs. Big data is big, but it doesn't mean you have to process all of your data. When your business begins a data project, start with goals in mind and strategies for how you will use the data you have available to reach those goals. The team involved in implementing a solution needs to plan the type of data they need and the schemas they will use before they start building the system so the project doesn't go in the wrong direction. They also need to create policies for purging old data from the system once it is no longer useful.

7. Lack of skilled data professionals

One of the big data problems that many companies run into is that their current staff have never worked with big data before, and this is not the type of skill set you build overnight. Working with untrained personnel can result in dead ends, disruptions of workflow, and errors in processing. There are a few ways to solve this problem. One is to hire a big data specialist and have that specialist manage and train your data team until they are up to speed. The specialist can either be hired on as a full-time employee or as a consultant who trains your team and moves on, depending on your budget. Another option, if you have time to prepare ahead, is to offer training to your current team members so they will have the skills once

your big data project is in motion. A third option is to choose one of the self-service analytics or business intelligence solutions that are designed to be used by professionals who don't have a data science background.

8. Organizational resistance

Another way people can be a challenge to a data project is when they resist change. The bigger an organization is, the more resistant it is to change. Leaders may not see the value in big data, analytics, or machine learning. Or they may simply not want to spend the time and money on a new project. This can be a hard challenge to tackle, but it can be done. You can start with a smaller project and a small team and let the results of that project prove the value of big data to other leaders and gradually become a data-driven business. Another option is placing big data experts in leadership roles so they can guide your business towards transformation.

Traditional Business Intelligence Vs Big Data

- BI (Business Intelligence) is a set of processes, architectures, and technologies that convert raw data into meaningful information that drives profitable business actions. It is a suite of software and services to transform data into actionable intelligence and knowledge.
- Traditional BI methodology is based on the principle of grouping all business data into a central server. Typically, this data is analyzed in offline mode, after storing the information in an environment called Data Warehouse. The data is structured in a conventional relational database with an additional set of indexes and forms of access to the tables (multidimensional cubes).
- In traditional BI environment, all the enterprise's data is housed in a central server whereas in a big data environment data resides in a distributed file system. The distributed file system scales by scaling in or out horizontally as compared to typical database server that scales vertically.
- In traditional BI, data is generally analyzed in an offline mode whereas in big data, it is analyzed in both real time as well as in offline mode.
- Traditional BI is about structured data and it is here that data is taken to processing functions (move data to code) whereas big data is about variety: Structured, semi-structured, and unstructured data and here the processing functions are taken to the data (move code to data).
- Data processed by Big Data solutions can be historical or come from real-time sources. Thus, companies can make decisions that affect their business in an agile and efficient way.
- Big Data technology uses parallel mass processing (MPP) concepts, which improves the speed of analysis. With MPP many instructions are executed simultaneously, and since the various jobs are divided into several parallel execution parts, at the end the overall results are reunited and presented. This allows you to analyze large volumes of information quickly.

Typical data warehouse

- Big Data has become the reality of doing business for organizations today.
- There is a boom in the amount of structured as well as raw data that floods every organization daily.
- If this data is managed well, it can lead to powerful insights and quality decision making.
- Big data analytics is the process of examining large data sets containing a variety of data types to discover some knowledge in databases, to identify interesting patterns and establish relationships to solve problems, market trends, customer preferences, and other useful information.
- Companies and businesses that implement Big Data Analytics often reap several business benefits.
- Companies implement Big Data Analytics because they want to make more informed business decisions.
- A data warehouse (DW) is a collection of corporate information and data derived from operational systems and external data sources.
- A data warehouse is designed to support business decisions by allowing data consolidation, analysis and reporting at different aggregate levels.
- Data is populated into the Data Warehouse through the processes of extraction, transformation and loading (ETL tools).
- Data analysis tools, such as business intelligence software, access the data within the warehouse.



Fig: Typical Data Warehouse

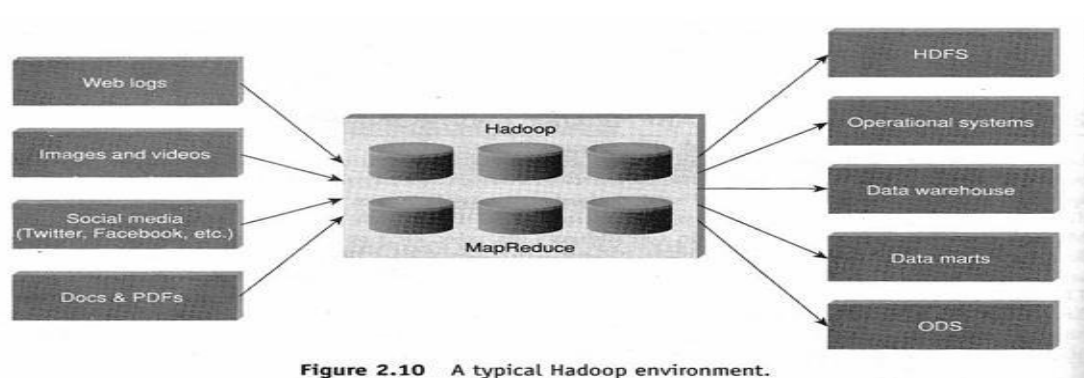
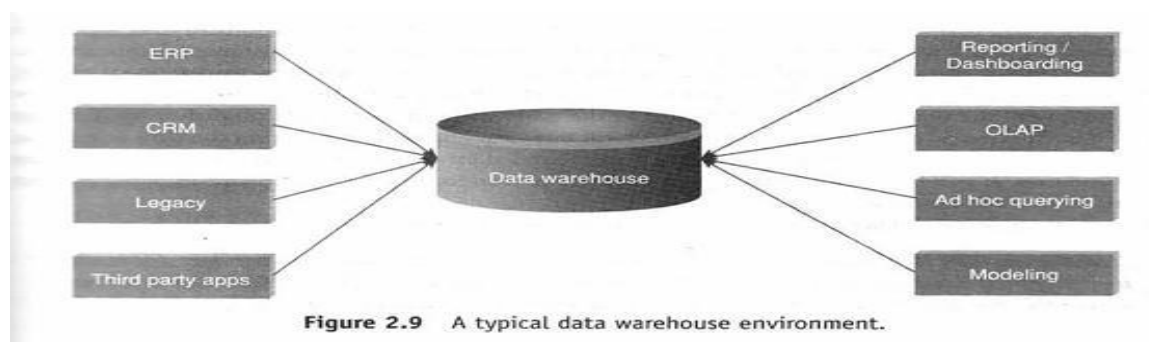
Typical Hadoop Environment

- Hadoop is changing the perception of handling Big Data especially the unstructured data.
- Apache Hadoop enables surplus data to be streamlined for any distributed processing system across clusters of computers using simple programming models.
- It truly is made to scale up from single servers to a large number of machines, each and every offering local computation, and storage space.
- Instead of depending on hardware to provide high-availability, the library itself is built to detect and handle breakdowns at the application layer, so providing an extremely available service along with a cluster of computers, as both versions might be vulnerable to failures.

Hadoop Community Package Consists of

- File system and OS level abstractions
- A MapReduce engine (either MapReduce or YARN)
- The Hadoop Distributed File System (HDFS)

- Java ARchive (JAR) files
- Scripts needed to start Hadoop
- Source code, documentation and a contribution section



Big data analytics is the process of examining large data sets containing a variety of data types to discover some knowledge in databases, to identify interesting patterns and establish relationships to solve problems, market trends, customer preferences, and other useful information.

What Big data Analytics Entails?

1. Classify analytics into basic, operationalized, advanced and monetized
2. Classify analytics into Analytics 1.0, Analytics 2.0 and Analytics 3.0

Classify analytics into basic, operationalized, advanced and monetized:

1. Basic analytics: This primarily is slicing and dicing of data to help with basic business insights. This is about reporting on historical data, basic visualization, etc.
2. Operationalized analytics: It is operationalized analytics if it gets woven into the enterprise's business processes.
3. Advanced analytics: This largely is about forecasting for the future by way of predictive and prescriptive modeling.
4. Monetized analytics: This is analytics in use to derive direct business revenue.

Classify analytics into Analytics 1.0, Analytics 2.0 and Analytics 3.0

Analytics 1.0:

- Era: mid 1950s to 2009
- Descriptive statistics (report on events, occurrences, etc. of the past)
- Key questions asked: What happened? Why did it happen?
- Data from legacy systems, ERP, CRM, and 3rd party applications.
- Small and structured data sources. Data stored in enterprise data warehouses or data marts.
- Data was internally sourced.
- Relational databases

Analytics 2.0:

- Era: 2005 to 2012
- Descriptive statistics + predictive statistics (use data from the past to make predictions for the future)
- Key questions asked: What will happen? Why will it happen?
- Big data is being taken up seriously. Data is mainly unstructured, arriving at a much higher pace. This fast flow of data entailed that the influx of big volume data had to be stored and processed rapidly, often on massive parallel servers running Hadoop.
- Data was often externally sourced. Database appliances, Hadoop clusters, SQL to Hadoop environments, etc.

Analytics 3.0:

- Era: 2012 to present
- Descriptive + predictive + prescriptive statistics (use data from the past to make prophecies for the future and at the same time make recommendations to leverage the situation to one's advantage)
- Key questions asked: What will happen? When will it happen? Why will it happen? What should be the action taken to take advantage of what will happen?
- A blend of big data and data from legacy systems, ERP, CRM, and 3rd party applications. A blend of big data and traditional analytics to yield insights and offerings with speed and impact.
- Data is both being internally and externally sourced.
- In memory analytics, in database processing, agile analytical methods, machine learning techniques, etc.

Descriptive analytics is a statistical method that is used to search and summarize historical data in order to identify patterns or meaning. Data aggregation and data mining are two techniques used in descriptive analytics to discover historical data. Data is first gathered and sorted by data aggregation in order to make the datasets more manageable by analysts. Data mining describes the next step of the analysis and involves a search of the data to identify patterns and meaning. Identified patterns are analyzed to discover the specific ways that learners interacted with the learning content and within the learning environment.

Advantages:

- Quickly and easily report on the Return on Investment (ROI) by showing how performance achieved business or target goals.
- Identify gaps and performance issues early - before they become problems.
- Identify specific learners who require additional support, regardless of how many students or employees there are.
- Identify successful learners in order to offer positive feedback or additional resources.
- Analyze the value and impact of course design and learning resources.

Predictive analytics

Predictive Analytics is a statistical method that utilizes algorithms and machine learning to identify trends in data and predict future behaviors. The software for predictive analytics has moved beyond the realm of statisticians and is becoming more affordable and accessible for different markets and industries, including the field of learning & development. For online learning specifically, predictive analytics is often found incorporated in the Learning Management System (LMS), but can also be purchased separately as specialized software. For the learner, predictive forecasting could be as simple as a dashboard located on the main screen after logging in to access a course. Analyzing data from past and current progress, visual indicators in the dashboard could be provided to signal whether the employee was on track with training requirements.

Advantages:

- Personalize the training needs of employees by identifying their gaps, strengths, and weaknesses; specific learning resources and training can be offered to support individual needs.
- Retain Talent by tracking and understanding employee career progression and forecasting what skills and learning resources would best benefit their career paths. Knowing what skills employees need also benefits the design of future training.
- Support employees who may be falling behind or not reaching their potential by offering intervention support before their performance puts them at risk.
- Simplified reporting and visuals that keep everyone updated when predictive forecasting is required.

Prescriptive Analytics

Prescriptive analytics is a statistical method used to generate recommendations and support decision-making based on the computational outcomes of algorithmic models. It focuses on determining the best possible course of action by analyzing available data and predicted future outcomes. To generate automated decisions or recommendations, prescriptive analytics requires specialized algorithmic models and clear objectives defined by the users of the analytical technique. Without a clearly identified problem or goal, meaningful recommendations cannot be produced; therefore, prescriptive analytics always begins with problem identification.

For example, a training manager may use predictive analytics to identify that most learners who lack a specific skill are unlikely to complete a newly launched course. Prescriptive analytics can then be applied to determine appropriate actions to address this issue. An algorithm might identify learners who need the course but do not possess the required skill and automatically recommend an additional training resource to help them acquire it. However, the effectiveness and accuracy of such recommendations depend heavily on the quality of data and the robustness of the algorithmic models used. What works effectively for one organization's training requirements may not be suitable for another. Therefore, prescriptive analytics models should be customized to fit the specific needs and context of each organization.

Descriptive vs Predictive vs Prescriptive Analytics

Descriptive analytics focuses exclusively on analyzing historical data to understand what has happened in the past. It summarizes past events and provides insights through reports, dashboards, and visualizations. Predictive analytics builds upon descriptive analytics by using historical data to develop statistical and machine learning models that forecast future outcomes and possibilities. Prescriptive analytics takes predictive analytics one step further by analyzing the forecasted outcomes and recommending appropriate actions. It predicts the consequences of different decisions and suggests the best possible course of action to achieve desired objectives.

Why Big Data Analytics Is Important

1. Reactive – Business Intelligence

Business Intelligence (BI) enables organizations to make faster and more effective decisions by delivering the right information to the right people at the right time in the appropriate format. BI primarily focuses on the analysis of past or historical data and presents the results through enterprise dashboards, alerts, notifications, and reports. It supports both pre-defined reports and ad hoc queries, helping organizations understand past performance and trends.

2. Reactive – Big Data Analytics

In reactive big data analytics, analysis is performed on very large datasets. However, the approach remains reactive because it relies on static data and focuses mainly on understanding past events rather than predicting future outcomes or recommending actions.

3. Proactive Analytics

Proactive analytics supports future-oriented decision-making by using techniques such as data mining, predictive modeling, text mining, and statistical analysis. Although it aims to predict future trends, it typically relies on traditional database management systems. As a result, it faces limitations related to storage capacity and processing power when handling extremely large datasets.

4. Proactive – Big Data Analytics

Proactive big data analytics involves analyzing massive volumes of data ranging from terabytes to petabytes and exabytes to extract relevant information. It uses high-performance analytics to gain rapid insights and solve complex problems by leveraging large-scale data processing and advanced analytical techniques.

Technologies Used in Big Data Environments

1. In-Memory Analytics

Accessing data from non-volatile storage such as hard disks is a slow process, especially when large volumes of data are involved. Traditional approaches involve pre-processing and storing aggregated data to reduce disk access, but this requires anticipating analytical needs in advance. In-memory analytics addresses this challenge by storing relevant data directly in Random Access Memory (RAM). This eliminates the need to access data from secondary storage, resulting in faster data access, quicker deployment, improved insights, and minimal IT involvement.

2. In-Database Processing

In-database processing, also known as in-database analytics, integrates analytical capabilities directly into the database system. Typically, data from various Online Transaction Processing (OLTP) systems is cleaned using ETL processes and stored in an Enterprise Data Warehouse (EDW) or data marts. Traditionally, this data is exported to external analytical tools for processing. In-database processing allows computations to be performed within the database itself, eliminating data movement and significantly reducing processing time. Many leading database vendors provide this capability for large enterprises.

3. Symmetric Multiprocessor System (SMP)

In a Symmetric Multiprocessor (SMP) system, two or more identical processors share a single common main memory and are controlled by a single operating system. Each processor has full access to all input/output devices and includes its own high-speed cache memory. SMP systems are tightly coupled multiprocessing systems where processors are connected through a system bus.

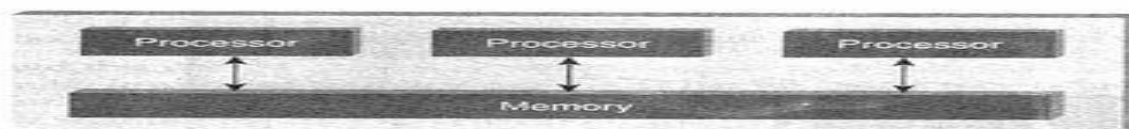
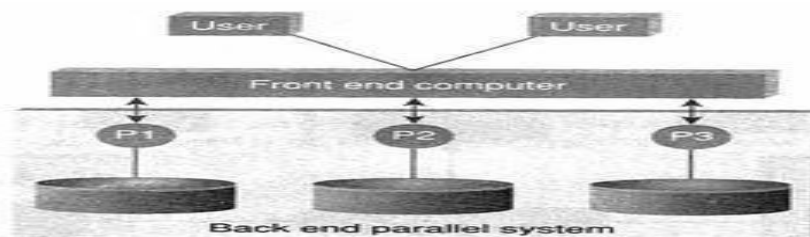
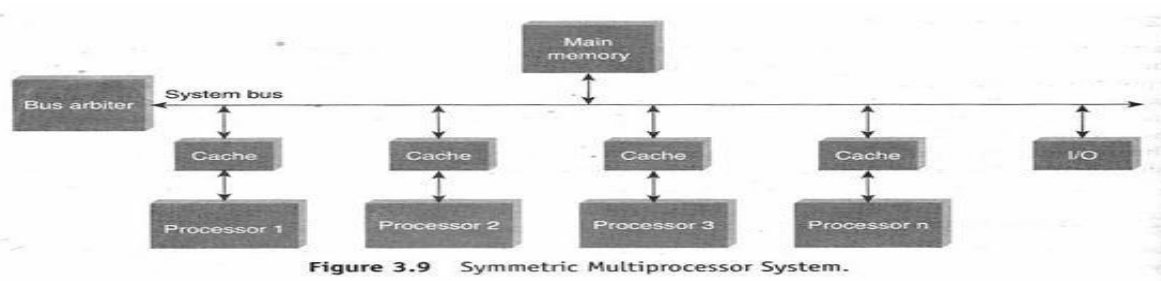
4. Massively Parallel Processing (MPP)

Massively Parallel Processing (MPP) involves multiple processors working in parallel to execute different parts of the same program. Each processor has its own operating system and dedicated memory. The processors communicate using a messaging interface. MPP systems are more complex to program because applications must be carefully divided to ensure proper communication between processing units. Unlike SMP systems, MPP systems do not share memory or a single operating system and are considered loosely coupled systems.

5. Difference between Parallel and Distributed Systems

A parallel database system is a tightly coupled system in which multiple processors cooperate closely to process database queries. In such systems, the processors either share a common memory or communicate with each other through message passing mechanisms. The primary objective of a parallel database system is to improve performance and reduce query execution time by dividing tasks among multiple processors working simultaneously.

In contrast, a distributed database system is a loosely coupled system composed of multiple independent machines connected through a network. Each machine can run its own applications and serve its local users independently. The data in a distributed database system is spread across several machines, which means that answering a single user query often requires accessing data from multiple nodes. This architecture emphasizes scalability, data availability, and resource sharing rather than tightly coordinated processing.



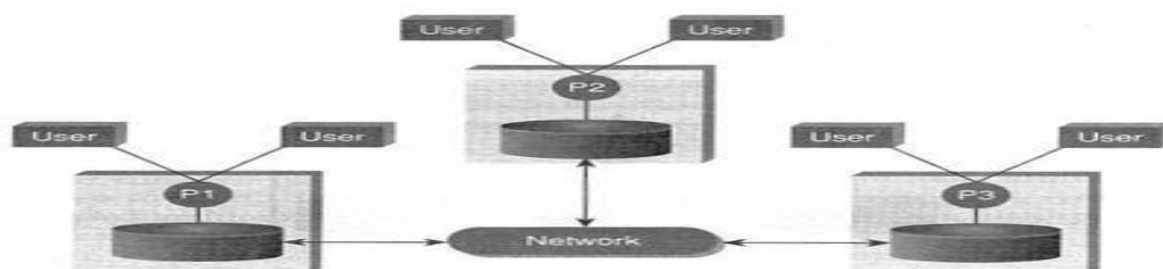


Figure 3.12 Distributed system.

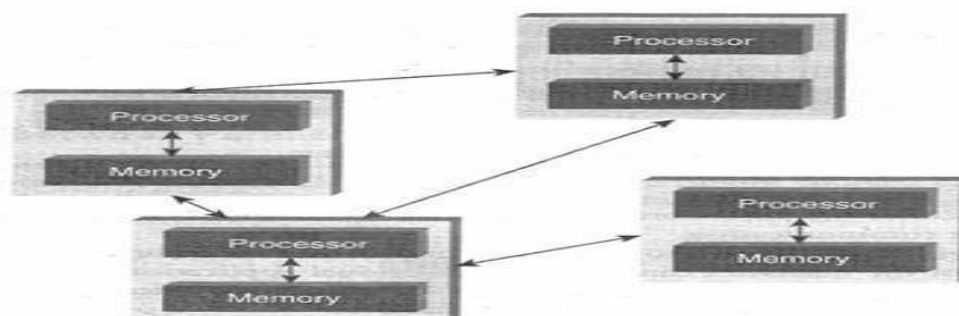


Figure 3.13 Distributed system.

6. Shared Nothing Architecture

The three most common types of architecture for multiprocessor high transaction rate systems are:

1. Shared Memory (SM).
2. Shared Disk (SD).
3. Shared Nothing (SN).

- In shared memory architecture, a common central memory is shared by multiple processors

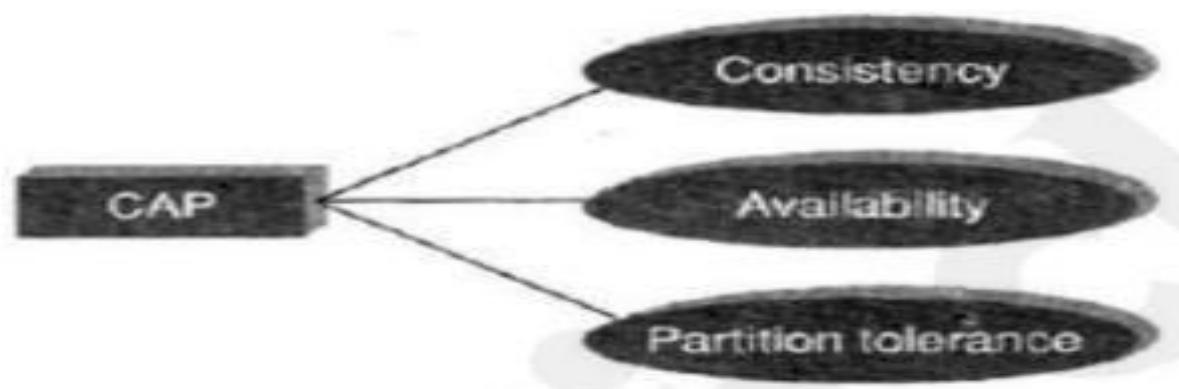
Advantages of a "Shared Nothing Architecture"

1. Fault Isolation: In a Shared Nothing Architecture, each node operates independently without shared memory or disk, ensuring faults are isolated to the failing node. Failures affect only message exchanges, preventing system wide disruptions.
2. Scalability: Assume that the disk is a shared resource. It implies that the controller and the disk bandwidth are also shared. Synchronization will have to be implemented to maintain a consistent shared state. This would mean that different nodes will have to take turns to access the critical data. This imposes a limit on how many nodes can be added to the distributed shared disk system, thus compromising on scalability.

7. CAP Theorem

The CAP theorem is also called the Brewer's Theorem. It states that in a distributed computing environment (a collection of interconnected nodes that share data), it is impossible to provide the following guarantees. At best you can have two of the following three - one must be sacrificed.

1. Consistency
2. Availability
3. Partition tolerance



CAP Theorem

1. Consistency implies that every read fetches the last write.
2. Availability implies that reads and writes always succeed. In other words, each non-failing node will return a response in a reasonable amount of time.
3. Partition tolerance implies that the system will continue to function when network partition occurs.

When to choose consistency over availability and vice-versa...

1. Choose availability over consistency when your business requirements allow some flexibility around when the data in the system synchronizes.
2. Choose consistency over availability when your business requirements demand atomic reads and

Examples of databases that follow one of the possible three combinations

1. Availability and Partition Tolerance (AP)
2. Consistency and Partition Tolerance (CP)
3. Consistency and Availability (CA)

Few top Analytical tools

There is no dearth of analytical tools in the market.

1. MS Excel
2. Statistical Analysis System(SAS)
3. IBM Statistical Package for the Social Sciences(SPSS) Modeler
4. Statistica
5. Salford systems
6. World Programming System(WPS)

Open Source Analytics Tools

1. R analytics
2. Weka

NoSQL (NOT ONLY SQL)

The term NoSQL was first coined by Carlo Strozzi in 1998 to name his lightweight, open-source, relational database that did not expose the standard

SQL interface.

Few features of NoSQL databases are as follows: .

1. They are open sources
2. They are nonrelational
3. They are distributed
4. They are schema less
5. They are cluster friendly
6. They are born out of 21st century web applications.

Where is it Used?

- NoSQL databases are widely used in big data and other real-time web applications.
- NoSQL. databases is used to stock log data which can then be pulled for analysis.
- It is used to store social media data and all such data which cannot be stored and analyzed comfortably in RDBMS.



Figure 4.1 Where to use NoSQL?



Figure 4.2 What is NoSQL?

What is it?

▪ NoSQL stands for Not Only SQL. These are non-relational, open source, distributed databases. They are hugely popular today owing to their ability to scale out or scale horizontally and the adeptness at dealing with a rich variety of data: structured, semi-structured and unstructured data.

1. Are non-relational: They do not adhere to relational data model, In fact, they are either key-value pairs or document-oriented or column-oriented or graph-based databases.
2. Are distributed: They are distributed meaning the data is distributed across several nodes in a cluster constituted of low-cost commodity hardware.
3. Offer no support for ACID properties (Atomicity, Consistency, Isolation, and Durability): They do not offer support for ACID properties of transactions. On the contrary, they have adherence to Brewer's CAP (Consistency, Availability, and Partition tolerance) theorem and are often seen compromising on consistency in favor of availability and partition tolerance.
4. Provide no fixed table schema: NoSQL databases are becoming increasingly popular owing to their support for flexibility to the schema. They do not mandate for the data to strictly adhere to any schema structure at the time of storage.

Types of NoSQL Databases:

Document databases: These databases store data as semi-structured documents, such as JSON or XML, and can be queried using document-oriented query languages. Eg. MongoDB

Sample Document in Document Database

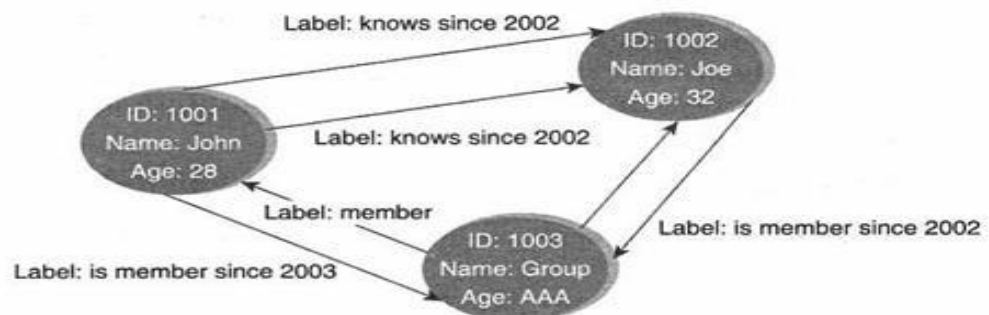
```
{  
  "Book Name": "Fundamentals of Business Analytics",  
  "Publisher": "Wiley India",  
  "Year of Publication": "2011"  
}
```

Key-value stores: These databases store data as key-value pairs, and are optimized for simple and fast read/write operations. Eg. Redis, Coherence

Column-family stores: These databases store data as column families, which are sets of columns that are treated as a single entity. They are optimized for fast and efficient querying of large amounts of data. Eg., Big Table

Key	Value
First Name	Simmonds
Last Name	David

Graph databases: These databases store data as nodes and edges, and are designed to handle complex relationships between data. Eg., Amazon Neptune



Refer Table 4.1 for popular schema-less databases.

Why NoSQL?

1. It has scale out architecture instead of the monolithic architecture of relational databases.
2. It can house large volumes of structured, semi-structured, and unstructured data.
3. Dynamic schema: NoSQL database allows insertion of data without a predefined schema. In other words, it facilitates application changes in real time, which thus supports faster development, easy code integration, and requires less database administration.
4. Auto-sharding: It automatically spreads data across an arbitrary number of servers. The application in question is more often not even aware of the composition of the server pool. It balances the load of data and query on the available servers; and if and when a server goes down, it is quickly replaced without any major activity disruptions.
5. Replication: It offers good support for replication which in turn guarantees high availability, fault tolerance, and disaster recovery.

Advantages of NoSQL (NOT ONLY SQL)



Figure 4.4 Advantages of NoSQL.

Advantages of NoSQL (NOT ONLY SQL)

1. Can easily scale up and down: NoSQL database supports scaling rapidly and elastically and even allows to scale to the cloud.

(a) Cluster scale: It allows distribution of database across 100+ nodes often in multiple data centers.

(b) Performance scale: It sustains over 100,000+ database reads and writes per second.

(c) Data scale: It supports housing of 1 billion+ documents in the database.

2. Doesn't require a pre-defined schema: NoSQL does not require any adherence to pre-defined schema. It is pretty flexible. For example, if we look at MongoDB, the documents (equivalent of records in RDBMS) in a collection (equivalent of table in RDBMS) can have different sets of key-value pairs.

3. Cheap, easy to implement: Deploying NoSQL properly allows for all of the benefits of scale, high availability, fault tolerance, etc. while also lowering operational costs.

4. Relaxes the data consistency requirement: NoSQL databases have adherence to CAP theorem (Consistency, Availability, and Partition tolerance).

Most of the NoSQL databases compromise on consistency in favor of availability and partition tolerance. However, they do go for eventual consistency.

5. Data can be replicated to multiple nodes and can be partitioned:

There are two terms that we will discuss here:

(a) Sharding: Sharding is when different pieces of data are distributed across multiple servers. NoSQL databases support auto-sharding; this means that they can natively and automatically spread data across an arbitrary number of servers, without requiring the application to even be aware of the composition of the server pool. Servers can be added or removed from the data layer without application downtime. This would mean that data and query load are automatically balanced across servers, and when a server goes down, it can be quickly and transparently replaced with no application disruption.

(b) Replication: Replication is when multiple copies of data are stored across the cluster and even across data centers. This promises high availability and fault tolerance.

Use of NoSQL in industry

Use of NoSQL in industry



Figure 4.6 Use of NoSQL in industry.

NO SQL VENDORS

Company	Product	Most Widely Used by
Amazon	DynamoDB	LinkedIn, Mozilla
Facebook	Cassandra	Netflix, Twitter, eBay
Google	BigTable	Adobe Photoshop

SQL VERSUS NOSQL

Refer Table 4.3 for few salient differences between SQL and NoSQL.

Table 4.3 SQL versus NoSQL	
SQL	NoSQL
Relational database	Non-relational, distributed database
Relational model	Model-less approach
Pre-defined schema	Dynamic schema for unstructured data
Table based databases	Document-based or graph-based or wide column store or key-value pairs databases
Vertically scalable (by increasing system resources)	Horizontally scalable (by creating a cluster of commodity machines)
Uses SQL	Uses UnQL (Unstructured Query Language)
Not preferred for large datasets	Largely preferred for large datasets
Not a best fit for hierarchical data	Best fit for hierarchical storage as it follows the key-value pair of storing data similar to JSON (Java Script Object Notation)
Emphasis on ACID properties	Follows Brewer's CAP theorem
Excellent support from vendors	Relies heavily on community support
Supports complex querying and data keeping needs	Does not have good support for complex querying
Can be configured for strong consistency	Few support strong consistency (e.g., MongoDB), some others can be configured for eventual consistency (e.g., Cassandra)
Examples: Oracle, DB2, MySQL, MS SQL, PostgreSQL, etc.	Examples: MongoDB, HBase, Cassandra, Redis, Neo4j, CouchDB, Couchbase, Riak, etc.

NewSQL

There is yet another new term doing the rounds - "NewSQL". So, what is NewSQL and how is it different from SQL and NoSQL? We need a database that has the same scalable performance of NoSQL systems for On Line Transaction Processing (OLTP) while still maintaining the ACID guarantees of a traditional database. This new modern RDBMS is called NewSQL. It supports relational data model and uses SQL

as their primary interface.

Characteristics of NewSQL

NewSQL is based on the shared nothing architecture with a SQL interface for application interaction.

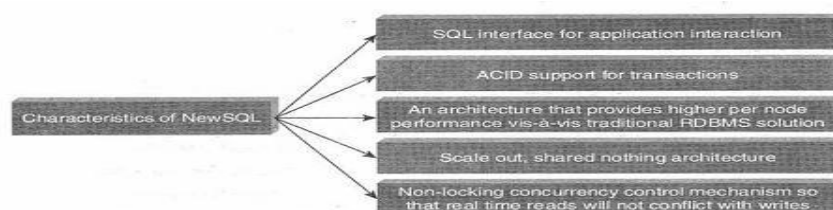


Figure 4.7 Characteristics of NewSQL.

Comparison of SQL, NoSQL, and NewSQL

NewSQL is based on the shared nothing architecture with a SQL interface for application interaction.

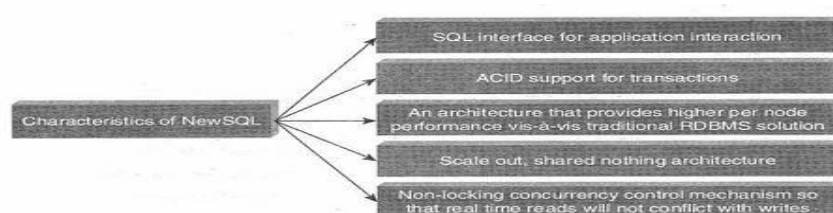


Figure 4.7 Characteristics of NewSQL.

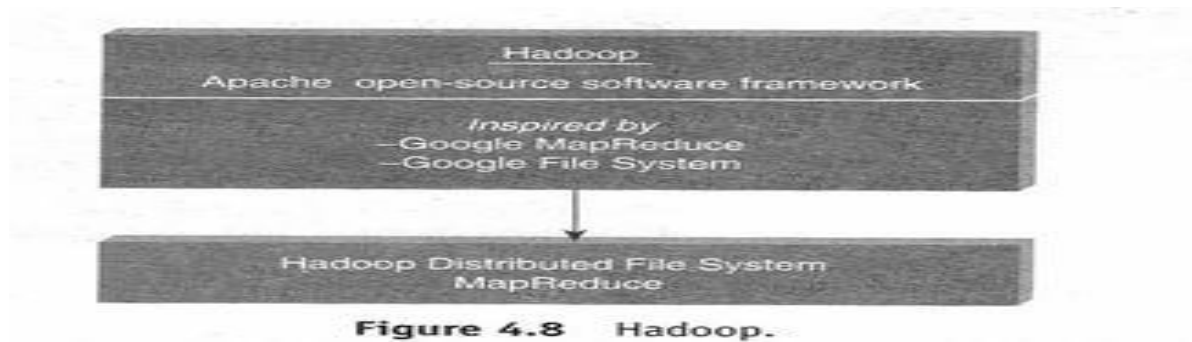
Comparison of SQL, NoSQL, and NewSQL

Table 4.4 Comparative study of SQL, NoSQL and NewSQL

	SQL	NoSQL	NewSQL
Adherence to ACID properties	Yes	No	Yes
OLTP/OLAP	Yes	No	Yes
Schema rigidity	Yes	No	Maybe
Adherence to data model	Adherence to relational model		
Data Format Flexibility	No	Yes	Maybe
Scalability	Scale up Vertical Scaling	Scale out Horizontal Scaling	Scale out
Distributed Computing	Yes	Yes	Yes
Community Support	Huge	Growing	Slowly growing

Hadoop

- Hadoop is an open-source project of the Apache foundation.
- It is a framework written in Java, originally developed by Doug Cutting in 2005 who named it after his son's toy elephant who was working with Yahoo then.
- It was created to support distribution for "Nutch", the text search engine.
- Hadoop uses Google's MapReduce and Google File System technologies as its foundation.
- Hadoop is now a core part of the computing infrastructure for companies such as Yahoo, Facebook, LinkedIn, Twitter, etc.



Features of Hadoop

1. It is optimized to handle massive quantities of structured, semi-structured, and unstructured data, using commodity hardware, that is, relatively inexpensive computers.
 2. Hadoop has a shared nothing architecture.
 3. It replicates its data across multiple computers so that if one goes down, the data can still be processed from another machine that stores its replica.
 4. Hadoop is for high throughput rather than low latency. It is a batch operation handling massive quantities of data; therefore the response time is not immediate.
 5. It complements On-Line Transaction Processing (OLTP) and On-Line Analytical Processing (OLAP). However, it is not a replacement for a relational database management system.
 6. It is NOT good when work cannot be parallelized or when there are dependencies within the data.
- It is NOT good for processing small files. It works best with huge data files and datasets.

Key advantages of Hadoop

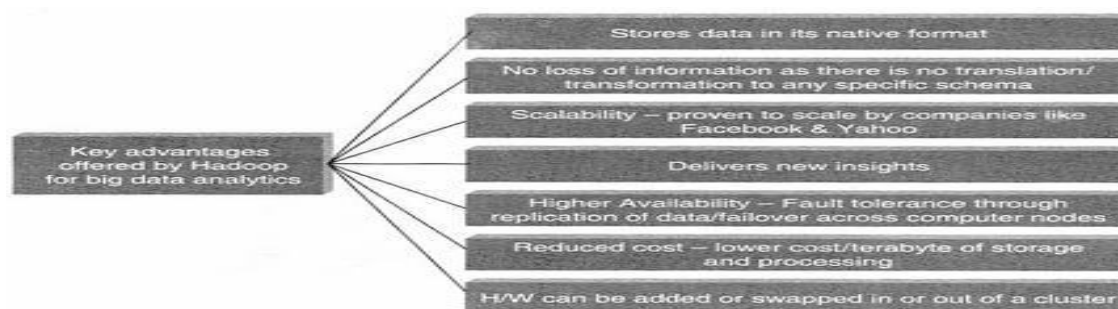


Figure 4.9 Key advantages of Hadoop.

1. Stores data in its native format: Hadoop's data storage framework (HDFS - Hadoop Distributed File System) can store data in its native format. There is no structure that is imposed while keying in data or storing data. HDFS is pretty much schema-less. It is only later when the data needs to be processed that structure is imposed on the raw data.
2. Scalable: Hadoop can store and distribute very large datasets (involving thousands of terabytes of data) across hundreds of inexpensive servers that operate in parallel.
3. Cost-effective: Owing to its scale-out architecture, Hadoop has a much reduced cost/terabyte of storage and processing.
4. Resilient to failure: Hadoop is fault-tolerant. It practices replication of data diligently which means whenever data is sent to any node, the same data also gets replicated to other nodes in the cluster, thereby ensuring that in the event of a node failure, there will always be another copy of data available for use.
5. Flexibility: One of the key advantages of Hadoop is its ability to work with all kinds of data: structured, semi-structured, and unstructured data. It can help derive meaningful business insights from email conversations, social media data, click-stream data, etc. It can be put to several purposes such as log analysis, data mining, recommendation systems, market campaign analysis, etc.
6. Fast: Processing is extremely fast in Hadoop as compared to other conventional systems owing to the "move code to data" paradigm.

Hadoop has a shared-nothing architecture.

Versions of Hadoop

There are two versions of Hadoop available:

1. Hadoop 1.0
2. Hadoop 2.0

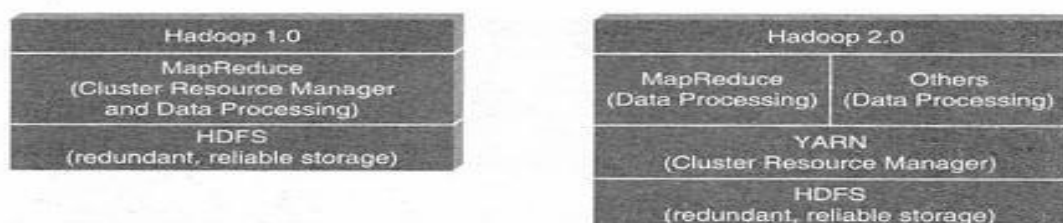


Figure 4.10 Versions of Hadoop.

1. Hadoop 1.0:

It has 2 main parts:

1. Data storage framework: It is a general-purpose file system called Hadoop Distributed File System (HDFS). HDFS is schema-less. It simply stores data files. These data files can be in just about any format. The idea is to store files as close to their original form as possible. This in turn provides the business units and the organization the much needed flexibility and agility without being overly worried by what it can implement.
2. Data processing framework: This is a simple functional programming model initially popularized by Google as MapReduce. It essentially uses two functions: the MAP and the REDUCE functions to process data. The "Mappers" take in a set of key-value pairs and generate intermediate data (which is another list of key-value pairs). The "Reducers" then act on this input to produce the output data. The two functions seemingly work in isolation from one another, thus enabling the processing to be highly distributed in a highly-parallel, fault-tolerant, and scalable way.

There were, however, a few limitations of Hadoop 1.0. They are as follows:

1. The first limitation was the requirement for MapReduce programming expertise along with proficiency required in other programming languages, notably Java.
2. It supported only batch processing which although is suitable for tasks such as log analysis, large-scale data mining projects but pretty much unsuitable for other kinds of projects.
3. One major limitation was that Hadoop 1.0 was tightly computationally coupled with MapReduce, which meant that the established data management vendors were left with two options: Either rewrite their functionality in MapReduce so that it could be executed in Hadoop or extract the data from HDFS and process it outside of Hadoop. None of the options were viable as it led to process inefficiencies caused by the data being moved in and out of the

Hadoop cluster.

2. Hadoop 2.0:

In Hadoop 2.0, HDFS continues to be the data storage framework. However, a new and separate resource management framework called Yet Another Resource Negotiator (YARN) has been added. Any application capable of dividing itself into parallel tasks is supported by YARN. YARN coordinates the allocation of subtasks of the submitted application, thereby further enhancing the flexibility, scalability, and efficiency of the applications. It works by having an Application Master in place of the erstwhile JobTracker, running applications on resources governed by a new NodeManager (in place of the erstwhile TaskTracker). Application Master is able to run any application and not just MapReduce. This, in other words, means that the MapReduce Programming expertise is no longer required. Furthermore, it not only supports batch processing but also real-time processing. MapReduce is no longer the only data processing option; other alternative data processing functions such as data standardization, master data management can now be performed natively in

HDFS.

Overview of Hadoop Ecosystems

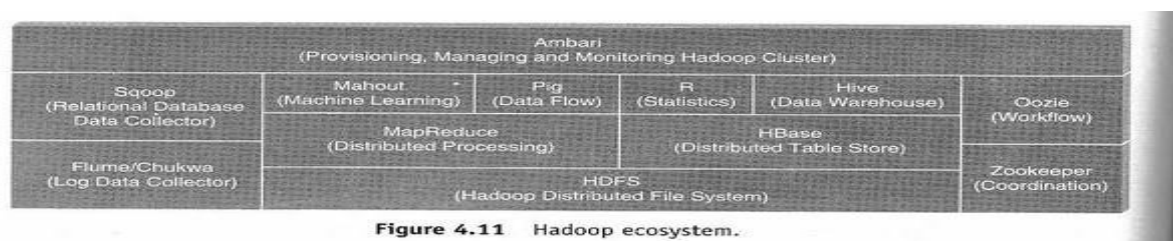


Figure 4.11 Hadoop ecosystem.

The components of the Hadoop ecosystem are shown in Figure. There are components available in the Hadoop ecosystem for data ingestion, processing, and analysis.

Data Ingestion → Data Processing → Data Analysis

Components that help with Data Ingestion are:

1. Sqoop
2. Flume

Components that help with Data Processing are:

1. MapReduce
2. Spark

Components that help with Data Analysis are:

1. Pig
2. Hive
3. Impala

HDFS

It is the distributed storage unit of Hadoop. It provides streaming access to file system data as well as file permissions and authentication. It is based on GFS (Google File System). It is used to scale a single cluster node to hundreds and thousands of nodes. It handles large datasets running on commodity hardware. HDFS is highly fault-tolerant. It stores files across multiple machines. These files are stored in redundant fashion to allow for data recovery in case of failure.

HBase

It stores data in HDFS. It is the first non-batch component of the Hadoop Ecosystem. It is a database on top of HDFS. It provides a quick random access to the stored data. It has very low latency compared to HDFS. It is a NoSQL database, is non-relational and is a column-oriented database. A table can have thousands of columns. A table can have multiple rows. Each row can have several column families. Each column family can have several columns. Each column can have several key values. It is based on Google BigTable. This is widely used by Facebook, Twitter, Yahoo, etc.

Difference between HBase and Hadoop/HDFS

1. HDFS is the file system whereas HBase is a Hadoop database.
2. HDFS is WORM (Write once and read multiple times or many times). Latest versions support appending of data but this feature is rarely used. However, HBase supports real-time random read and write.
3. HDFS is based on Google File System (GFS) whereas HBase is based on Google Big Table.
4. HDFS supports only full table scan or partition table scan. Hbase supports random small range scan or table scan.
5. Performance of Hive on HDFS is relatively very good but for HBase it becomes 4-5 times slower.
6. The access to data is via MapReduce job only in HDFS whereas in HBase the access is via Java APIs, Rest, Avro, Thrift APIs.
7. HDFS does not support dynamic storage owing to its rigid structure whereas HBase supports dynamic storage.
8. HDFS has high latency operations whereas HBase has low latency operations.
9. HDFS is most suitable for batch analytics whereas HBase is for real-time analytics.

Hadoop Ecosystem Components for Data Ingestion

1. Sqoop: Sqoop stands for SQL to Hadoop. Its main functions are:
 - a) Importing data from RDBMS such as MySQL, Oracle, DB2, etc. to Hadoop file system (HDFS, HBase, Hive).
 - b) Exporting data from Hadoop File system (HDFS, HBase, Hive) to RDBMS (MySQL, Oracle, DB2).

Uses of Sqoop

- a) It has a connector-based architecture to allow plug-ins to connect to external systems such as MySQL, Oracle, DB2, etc.
- b) It can provision the data from external system on to HDFS and populate tables in Hive and HBase.
- c) It integrates with Oozie allowing you to schedule and automate import and export tasks.

2. Flume: Flume is an important log aggregator (aggregates logs from different machines and places them in HDFS) component in the Hadoop ecosystem. Flume has been developed by Cloudera. It is designed for high volume ingestion of event-based data into Hadoop. The default destination in Flume (called as sink in flume parlance) is HDFS. However it can also write to HBase or Solr.

Hadoop Ecosystem Components for Data Processing

1. MapReduce: It is a programming paradigm that allows distributed and parallel processing of huge datasets. It is based on Google MapReduce. Google released a paper on MapReduce programming paradigm in 2004 and that became the genesis of Hadoop processing model. The MapReduce framework gets the input data from HDFS. There are two main phases: Map phase and the

Reduce phase. The map phase converts the input data into another set of data (key-value pairs). This new intermediate dataset then serves as the input to the reduce phase. The reduce phase acts on the datasets to combine (aggregate and consolidate) and reduce them to a smaller set of tuples. The result is then stored back in HDFS.

2. Spark: It is both a programming model as well as a computing model. It is an open-source big data processing framework. It was originally developed in 2009 at UC Berkeley's AmpLab and became an open-source project in 2010. It is written in Scala. It provides in-memory computing for Hadoop. In Spark, workloads execute in memory rather than on disk owing to which it is much faster (10 to 100 times) than when the workload is executed on disk. However, if the datasets are too large to fit into the available system memory, it can perform conventional disk-based processing. It serves as a potentially faster and more flexible alternative to MapReduce. It accesses data from HDFS (Spark does not have its own distributed file system) but bypasses the MapReduce processing. Spark can be used with Hadoop coexisting smoothly with MapReduce (sitting on top of Hadoop YARN) or used independently of Hadoop (standalone). As a programming model, it works well with Scala, Python (it has API connectors for using it with Java or Python) or R programming language. The following are the Spark libraries:

- a) Spark SQL: Spark also has support for SQL. Spark SQL uses SQL to help query data stored in disparate applications.
- b) Spark streaming: It helps to analyze and present data in real time.
- c) MLlib: It supports machine learning such as applying advanced statistical operations on data in Spark Cluster.
- d) GraphX: It helps in graph parallel computation.

Spark and Hadoop are usually used together by several companies. Hadoop was primarily designed to house unstructured data and run batch processing operations on it. Spark is used extensively for its high speed in memory computing and ability to run advanced real-time analytics where the two together have been giving good results.

Hadoop Ecosystem Components for Data Analysis

1. Pig: It is a high-level scripting language used with Hadoop. It serves as an alternative to MapReduce. It has two parts:

(a) Pig Latin: It is SQL-like scripting language. Pig Latin scripts are translated into MapReduce jobs which can then run on YARN and process data in the HDFS cluster. It was initially developed by Yahoo. It is immensely popular with developers who are not comfortable with MapReduce. However, SQL developers may have a preference for Hive.

How it works? There is a "Load" command available to load the data from "HDFS" into Pig. Then one can perform functions such as grouping, filtering, sorting, joining etc. The processed or computed data can then be either displayed on screen or placed back into HDFS. It gives you a platform for building data flow for ETL (Extract, Transform and Load), processing and analyzing huge data sets.

(b) Pig runtime: It is the runtime environment.

2. Hive:

Hive is a data warehouse software project built on top of Hadoop. Three main tasks performed by Hive are summarization, querying and analysis. It supports queries written in a language called HQL or HiveQL which is a declarative SQL-like language. It converts the SQL-style queries into MapReduce jobs which are then executed on the Hadoop platform.

Difference between Hive and RDBMS

Both Hive and traditional databases such as MySQL, MS SQL Server, PostgreSQL support SQL interface. However, Hive is better known as a dataware house (D/W) rather than a database.

Let us look at the difference between Hive and traditional databases as regards the schema.

1. Hive enforces schema on Read Time whereas RDBMS enforces schema on Write Time. In RDBMS, at the time of loading/inserting data, the table's schema is enforced. If the data being loaded does not conform to the schema then it is rejected. Thus, the schema is enforced on write (loading the data into the database). Schema on write takes longer to load the data into the database; however it makes up for it during data retrieval with a good query time performance. However, Hive does not enforce the schema when the data is being loaded into the D/W. It is enforced only when the data is being read/retrieved. This is called schema on read. It definitely makes for fast initial load as the data load or insertion operation is just a file copy or move.

2. Hive is based on the notion of write once and read many times whereas the RDBMS is designed for read and write many times.

3. Hadoop is a batch-oriented system. Hive, therefore, is not suitable for OLTP (Online Transaction Processing) but, although not ideal, seems closer to OLAP (Online Analytical Processing). The reason being that there is quite a latency between issuing a query and receiving a reply as the query written in HiveQL will be converted to MapReduce jobs which are then executed on the Hadoop cluster. RDBMS is suitable for housing day-to-day transaction data and supports all OLTP operations with frequent insertions, modifications (updates), deletions of the data.

4. Hive handles static data analysis which is non-real-time data. Hive is the data warehouse of Hadoop. There are no frequent updates to the data and the query response time is not fast. RDBMS is suited for handling dynamic data which is real time.

5. Hive can be easily scaled at a very low cost when compared to RDBMS. Hive uses HDFS to store data, thus it cannot be considered as the owner of the data, while on the other hand RDBMS is the owner of the data responsible for storing, managing and manipulating it in the database.

6. Hive uses the concept of parallel computing, whereas RDBMS uses serial computing.

	Hadoop	RDBMS
Data Variety	Used for structured, semi-structured and unstructured data. Hadoop supports a variety of data formats in real time such as XML, JSON, and text-based flat file formats.	Used for structured data
Data Storage	Usually datasets of size terabytes, petabytes	Usually datasets of size gigabytes
Querying	HiveQL	SQL
Query Response	In Hadoop, there is latency due to batch processing.	In RDBMS, query response time is immediate.
Schema	Schema required on read	Schema required on write
Speed	Writes are faster compared to reads as there is no adherence to schema required at the time of inserting or writing data. Schema is enforced at read time. Hadoop is designed for write once read many times. It does not work for random reading and writing of a few records like RDBMS.	Reads are very fast (supported by building indexes on required columns). RDBMS is designed for read and write many times.
Cost	Apache Hadoop is open-source, large-scale, distributed, scalable, data intensive computing.	Available as proprietary RDBMS such as Oracle, MS SQL Server, IBM DB2, etc. Also open-source RDBMS are available such as MySQL, PostgreSQL, etc.
Use Cases	Analytics, data discovery	OLTP (Online Transaction Processing). Mainly used to store and process day-to-day business data.

Table 4.5 (Continued)

	Hadoop	RDBMS
Throughput	High	Low
Scalability	Horizontal (Hadoop scales by adding nodes to a Hadoop cluster of easily available commodity machines).	Vertical: RDBMS scales vertically by increasing the horsepower (CPU, Hard Disk Capacity, RAM, etc.) of the machine.
Hardware	Commodity/Utility Hardware	High End Servers
Integrity	Low	High. Obeys ACID properties A – Atomicity C – Consistency I – Integrity D – Durability

Difference between Hive and HBase

1. Hive is a MapReduce-based SQL engine that runs on top of Hadoop. HBase is a key-value NoSQL database that runs on top of HDFS.
2. Hive is for batch processing of big data. HBase is for real-time data streaming.

Impala

It is a high-performance SQL engine that runs on Hadoop cluster. It is ideal for interactive analysis. It has very low latency measured in milliseconds. It supports a dialect of SQL called Impala SQL.

ZooKeeper

It is a coordination service for distributed applications.

Oozie

It is a workflow scheduler system to manage Apache Hadoop jobs.

Mahout

It is a scalable machine learning and data mining library.

Chukwa

It is a data collection system for managing large distributed systems.

Ambari

It is a web-based tool for provisioning, managing, and monitoring Apache

Hadoop clusters.

Hadoop Distributions

Hadoop is an open-source Apache project. Anyone can freely download the core aspects of Hadoop. The core aspects of Hadoop include the following:

1. Hadoop Common
2. Hadoop Distributed File System (HDFS)
3. Hadoop YARN (Yet Another Resource Negotiator)
4. Hadoop MapReduce

There are few companies such as IBM, Amazon Web Services, Microsoft, Teradata, Hortonworks, Cloudera, etc. that have packaged Hadoop into a more easily consumable distributions or services. Although each of these companies have a slightly different strategy, the key essence remains its ability to distribute data and workloads across potentially thousands of servers thus making big data manageable data. A few Hadoop distributions are given in Figure below.

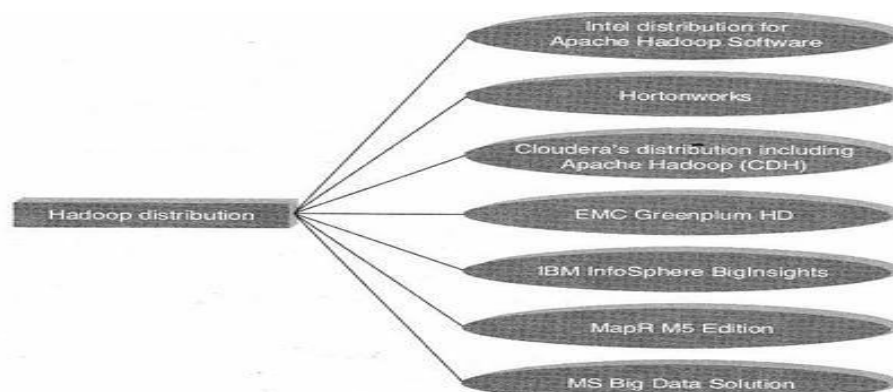


Figure 4.12 Hadoop distributions.

Difference between Hadoop and SQL

Feature	Hadoop	SQL
Technology	Modern	Traditional
Volume	Usually in Petabytes	Usually in Gigabytes
Operations	Storage, processing, retrieval and pattern extraction from data	Storage, processing, retrieval and pattern mining of data
Fault Tolerance	Hadoop is highly fault tolerant	SQL has good fault tolerance
Storage	Stores data in the form of key-value pairs, tables, hash maps, etc. in distributed systems	Stores structured data in tabular format with fixed schema in cloud
Scaling	Linear	Non-linear
Providers	Cloudera, Hortonworks, AWS etc. provide Hadoop systems	Well-known industry leaders in SQL systems are Microsoft, SAP, Oracle etc.
Data Access	Batch-oriented data access	Interactive and batch-oriented data access
Cost	It is open source and systems can be cost-effectively scaled	It is licensed and costs a fortune to buy a SQL server; additional charges emerge if storage runs out
Time	Statements are executed very quickly	SQL syntax is slow when executed on millions of rows
Optimization	Stores data in HDFS and processes through MapReduce with huge optimization techniques	Does not have any advanced optimization techniques

Feature	Hadoop	SQL
Structure	Dynamic schema; capable of storing and processing log data, real-time data, images, videos, sensor data (structured and unstructured)	Static schema; capable of storing data only in tabular format (structured)
Data Update	Write once, read many times	Read and write multiple times
Integrity	Low	High
Interaction	SQL systems can read and write data to Hadoop systems	Hadoop uses JDBC (Java Database Connectivity) to communicate with SQL systems
Hardware	Uses commodity hardware	Uses proprietary hardware
Training	Learning Hadoop for entry-level as well as seasoned professionals is moderately hard	Learning SQL is easy even for entry-level professionals

Integrated Hadoop Systems Offered by Leading Market Vendors

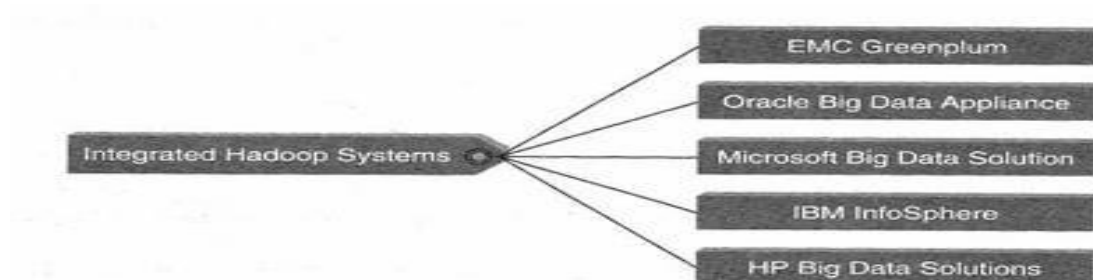
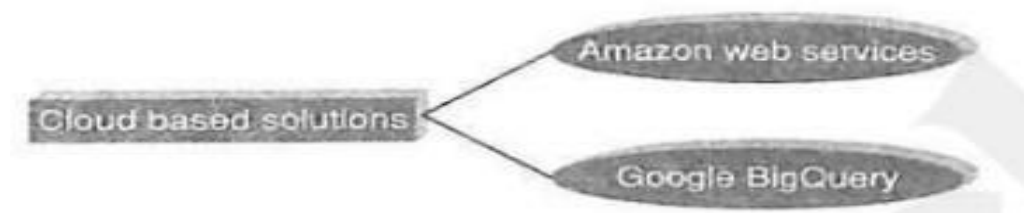


Figure 4.13 Integrated Hadoop systems.

Cloud-Based Hadoop Solutions

Amazon Web Services holds out a comprehensive, end-to-end portfolio of cloud computing services to help manage big data. The aim is to achieve this and more along with retaining the emphasis on reducing costs, scaling to meet demand, and accelerating the speed of innovation. The Google Cloud Storage connector for Hadoop empowers one to perform MapReduce jobs directly on data in Google Cloud Storage, without the need to copy it to local disk and running it in the Hadoop Distributed File System (HDFS). The connector simplifies Hadoop deployment, and at the same time reduces cost and provides performance comparable to HDFS, all this while increasing reliability by eliminating the single point of failure of the name node.



MODULE-2

Introduction to Hadoop: Introducing hadoop, Why hadoop, Why not RDBMS, RDBMS Vs Hadoop, History of Hadoop, Hadoop overview, Use case of Hadoop, HDFS (Hadoop Distributed File System), Processing data with Hadoop, Managing resources and applications with Hadoop YARN (Yet Another Resource Negotiator). Introduction to Map Reduce Programming: Introduction, Mapper, Reducer, Combiner, Partitioner, Searching, Sorting, Compression.

TB1: Ch 5: 5.1-5.8, 5.10-5.12, Ch 8: 8.1 - 8.8

Introducing Hadoop

- Hadoop is an open-source framework designed for distributed storage and processing of large datasets using clusters of computers.
- It divides large data into smaller parts and processes them across multiple computers at the same time.
- It is part of the Apache Software Foundation and is widely used for Big Data processing.

Need for Hadoop

Why can't we use traditional databases?

Big Data Problems: Companies like Google, Facebook, and Amazon generate terabytes of data every second.

Slow Processing: A single computer cannot handle such huge data efficiently. Storage Limitations: Traditional databases have limits on how much data they can store.

Hadoop helps to store and process large-scale data efficiently and quickly.

Hadoop has 3 main components:

HDFS (Hadoop Distributed File System) – Like Google Drive, it stores data across multiple computers.

MapReduce – Like group work, it processes data in parallel and then combines results.

YARN (Yet Another Resource Negotiator) – Like a manager, it assigns tasks to different computers.

Why Hadoop?

The key consideration is:

Its capability to handle massive amounts of data, different categories of data - fairly quickly.

The other considerations are:

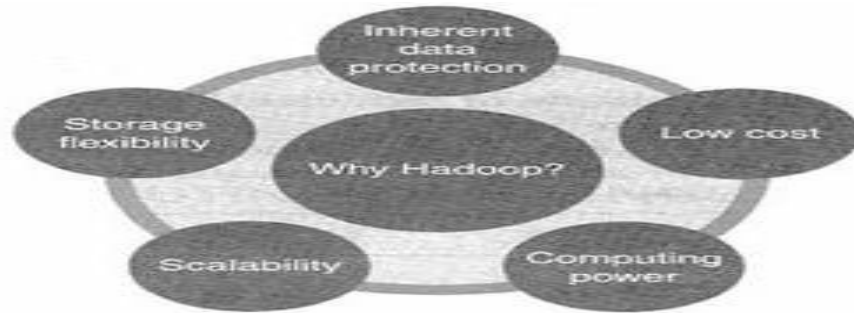


Figure 5.2 Key considerations of Hadoop.

Low cost:

Hadoop is an open-source framework and uses commodity hardware (commodity hardware is relatively inexpensive and low-cost, widely available, general-purpose computers) to store enormous quantities of data.

Computing power:

Hadoop is based on distributed computing model which processes very large volumes of data fairly quickly. The more the number of computing nodes, the more the processing power at hand.

Scalability:

This boils down to simply adding nodes as the system grows and requires much less administration.

Storage flexibility:

Unlike the traditional relational databases, in Hadoop data need not be pre-processed before storing it. Hadoop provides the convenience of storing as much data as one needs and also the added flexibility of deciding later as to how to use the stored data. In Hadoop, one can store unstructured data like images, videos, and free-form text.

Inherent data protection:

Hadoop protects data and executing applications against hardware failure. If a node fails, it automatically redirects the jobs that had been assigned to this node to the other functional and available nodes and ensures that distributed computing does not fail. It goes a step further to store multiple copies (replicas) of the data on various nodes across the cluster.

Why not RDBMS?

- RDBMS is not suitable for storing and processing large files, images, and videos.
- RDBMS is not a good choice when it comes to advanced analytics involving machine learning.
- It calls for huge investment as the volume of data shows an upward trend.

Difference between RDBMS and Hadoop

Parameter	RDBMS	Hadoop
System	RDBMS stands for Relational Database Management System. It follows a centralized, relational model.	Hadoop follows a node-based, distributed flat structure.
Data	RDBMS is suitable mainly for structured data stored in tables with rows and columns.	Hadoop supports structured, semi-structured, and unstructured data. It can handle XML, JSON, text files, logs, and more.
Processing	RDBMS is mainly used for OLTP (Online Transaction Processing).	Hadoop is used for analytical processing and big data processing.
Choice	RDBMS is preferred when data consistency and strong relationships between data are required.	Hadoop is preferred for big data processing where strict relationships and consistency are not mandatory.
Processor / Hardware	RDBMS requires expensive hardware or high-end processors to store and process large volumes of data.	Hadoop runs on commodity hardware; each node needs only a processor, network card, and multiple hard disks.
Cost	The cost of storage in RDBMS is high, approximately \$10,000 to \$14,000 per terabyte.	The cost of storage in Hadoop is comparatively low, around \$4,000 per terabyte.

PARAMETERS	RDBMS	HADOOP
System	Relational Database Management System.	Node Based Flat Structure.
Data	Suitable for structured data.	Suitable for structured, unstructured data. Supports variety of data formats in real time such as XML, JSON, text based flat file formats, etc.
Processing	OLTP	Analytical, Big Data Processing
Choice	When the data needs consistent relationship.	Big Data processing, which does not require any consistent relationships between data.
Processor	Needs expensive hardware or high-end processors to store huge volumes of data.	In a Hadoop Cluster, a node requires only a processor, a network card, and few hard drives.
Cost	Cost around \$10,000 to \$14,000 per terabytes of storage.	Cost around \$4,000 per terabytes of storage.

When to choose between NoSQL and Hadoop?

Choose Hadoop for large-scale batch processing, big data analytics, and cost-effective distributed storage with HDFS.

NoSQL is better for real-time transactions, flexible schema, and low-latency applications like web apps and IoT.

Distributed Computing Challenges:

Although there are several challenges with distributed computing, we will focus on two major challenges.

Hardware Failure In a distributed system, several servers are networked together. This implies that more often than not, there may be a possibility of hardware failure. And when such a failure does happen, how does one retrieve the data that was stored in the system? Just to explain further - a regular hard disk may fail once in 3 years. And when you have 1000 such hard disks, there is a possibility of at least a few being down every day.

Hadoop has an answer to this problem in Replication Factor (RF). Replication Factor connotes the number of data copies of a given data item/data block stored across the network.

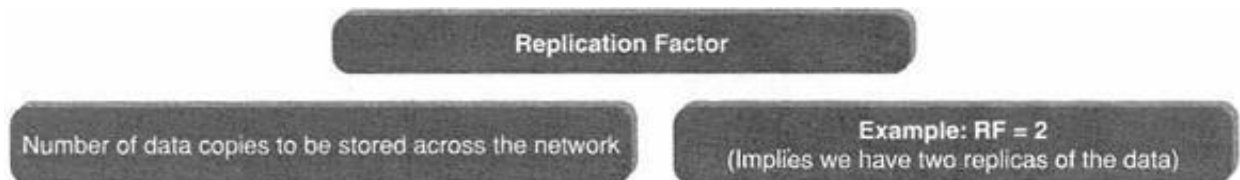


Figure 5.5 Replication factor.

How to Process This Gigantic Store of Data?

In a distributed system, the data is spread across the network on several machines. A key challenge here is to integrate the data available on several machines prior to processing it. Hadoop solves this problem by using MapReduce Programming. It is a programming model to process the data (MapReduce programming will be discussed a little later).

History of Hadoop

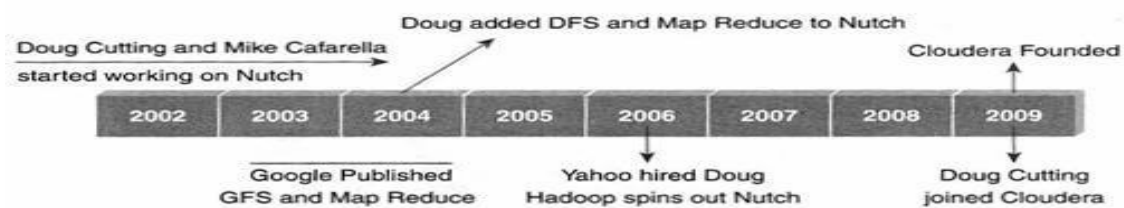


Figure 5.6 Hadoop history.

The image is a timeline depicting key events in the history of Hadoop and related technologies. Here's a breakdown of the events shown:

1. 2002 - Doug Cutting and Mike Cafarella started working on Nutch, an open-source web crawler.
2. 2003 - The development of Nutch continued.
3. 2004 - Google published papers on Google File System (GFS) and MapReduce, which inspired further developments in distributed computing.
4. 2004-2005 - Doug Cutting added Distributed File System (DFS) and MapReduce to Nutch, enhancing its capabilities for large-scale data processing.
5. 2006 - Yahoo hired Doug Cutting, and Hadoop was spun out from Nutch, becoming a separate open-source project focused on distributed data storage and processing.
6. 2007-2008 - Hadoop continued to evolve and gain adoption.

7. 2009 - Cloudera was founded, a company focused on commercializing Hadoop solutions.

8. 2009 - Doug Cutting joined Cloudera, further contributing to the development and expansion of Hadoop-based technologies.

Hadoop Overview

Open-source software framework to store and process massive amounts of data in a distributed fashion on large clusters of commodity hardware. Basically, Hadoop accomplishes two tasks:

1. Massive data storage.
2. Faster data processing.

Key Aspects of Hadoop

Open source software: It is free to download, use and contribute to.

Framework: Means everything that you will need to develop and execute and application is provided - programs, tools, etc..

Distributed: Divides and stores data across multiple computers. Computation/Processing is done in parallel across multiple connected nodes.

Massive storage: Stores colossal amounts of data across nodes of low-cost commodity hardware.

Faster processing: Large amounts of data is processed in parallel, yielding quick response.

Hadoop components

Hadoop Core Components

1. HDFS:

- (a) Storage component.
- (b) Distributes data across several nodes.
- (c) Natively redundant.

2. MapReduce:

- (a) Computational framework.
- (b) Splits a task across multiple nodes.
- (c) Processes data in parallel.

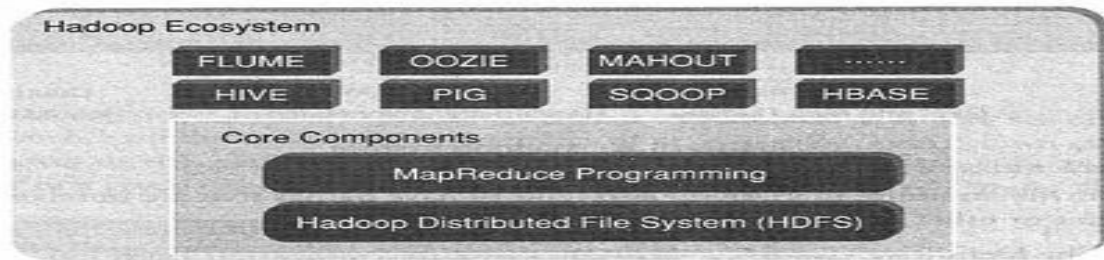


Figure 5.8 Hadoop components.

Hadoop Ecosystem: Hadoop Ecosystem are support projects to enhance the functionality of Hadoop Core Components. The Eco Projects are as follows:

1. HIVE
2. PIG
3. SPOOP
4. HBASE
5. FLUME
6. OOZIE
7. MAHOUT

Hadoop Conceptual Layer

It is conceptually divided into:

Data Storage Layer which stores huge volumes of data and

Data Processing Layer which processes data in parallel to extract richer and meaningful insights from data.



Figure 5.9 Hadoop conceptual layer.

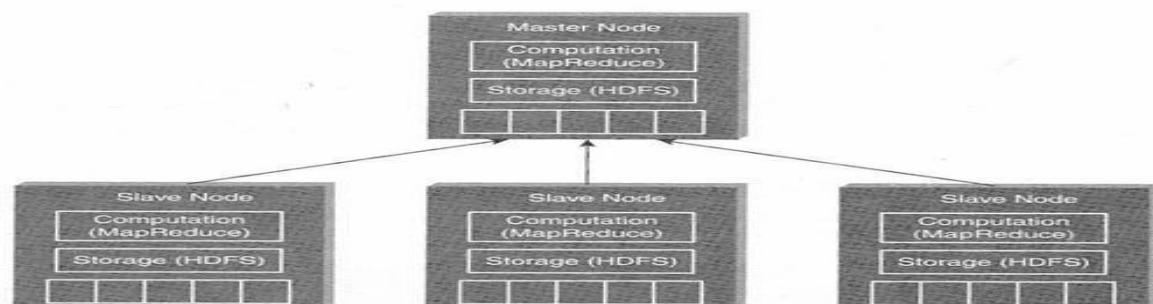


Figure 5.10 Hadoop high-level architecture.
Reference: Hadoop in Practice, Alex Holmes.

High-Level Architecture of Hadoop

- Hadoop is a distributed Master-Slave Architecture.

- Master node is known as NameNode and slave nodes are known as

DataNodes.

- Figure depicts the Master-Slave Architecture of Hadoop Framework.

The key components of the Master Node are:

1. Master HDFS: Its main responsibility is partitioning the data storage across the slave nodes. It also keeps track of locations of data on DataNodes.
2. Master MapReduce: It decides and schedules computation task on slave nodes.

Use case of Hadoop

ClickStream Data

ClickStream data (mouse clicks) helps you to understand the purchasing behavior of customers. ClickStream analysis helps online marketers to optimize their product web pages, promotional content, etc. to improve their business.

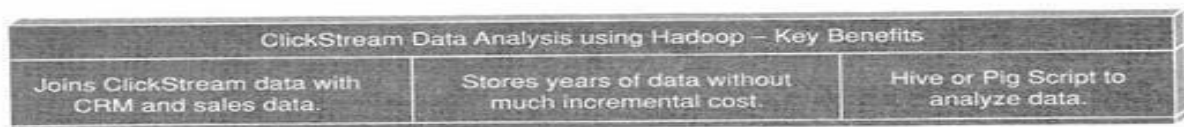


Figure 5.11 ClickStream data analysis.

The ClickStream analysis, as shown in above figure, using Hadoop provides three key benefits:

1. Hadoop helps to join ClickStream data with other data sources such as Customer Relationship Management Data (Customer Demographics Data, Sales Data, and Information on Advertising Campaigns). This additional data often provides the much needed information to understand customer behavior.
2. Hadoop's scalability property helps you to store years of data without ample incremental cost. This helps you to perform temporal or year over year analysis on ClickStream data which your competitors may miss.
3. Business analysts can use Apache Pig or Apache Hive for website analysis. With these tools, you can organize ClickStream data by user session, refine it, and feed it to visualization or analytics tools.

HDFS(Hadoop Distributed file system)

Some key Points of Hadoop Distributed File System are as follows:

1. Storage component of Hadoop.
2. Distributed File System.
3. Modeled after Google File System.

4. Optimized for high throughput (HDFS leverages large block size and moves computation where data is stored).
5. You can create multiple copies of a file as per configuration, ensuring reliability. This replication enhances fault tolerance for both software and hardware failures.
6. Re-replicates data blocks automatically on nodes that have failed.
7. You can realize the power of HDFS when you perform read or write on large files (gigabytes and larger).
8. It operates above native file systems like ext3 and ext4, as illustrated in the figure below. This abstraction enables additional functionality and flexibility in data management.

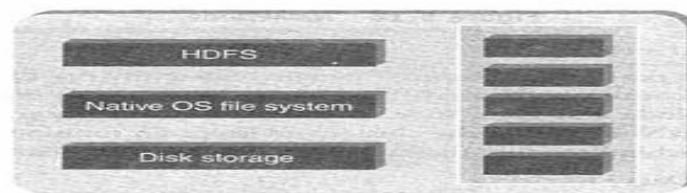


Figure 5.13 Hadoop Distributed File System.



Figure 5.14 Hadoop Distributed File System – key points.

The figure highlights key aspects of the Hadoop Distributed File System (HDFS). It mentions that HDFS uses a block-structured file system, has a default replication factor of 3 for fault tolerance, and a default block size of 64 MB for efficient storage and processing. Figure below describes Hadoop Distributed File System Architecture.

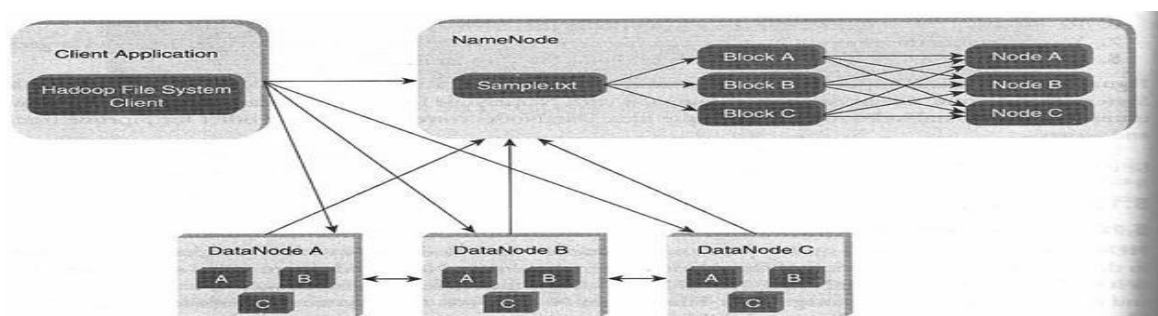


Figure 5.15 Hadoop Distributed File System Architecture.
Reference: Hadoop in Practice, Alex Holmes.



Figure 5.16 NameNode.

The figure illustrates the Hadoop Distributed File System (HDFS) architecture, showing how a file is stored and managed across multiple nodes.

Key Components:

Client Application

The client interacts with HDFS through the Hadoop File System Client to read/write data.

NameNode

Stores metadata about files (e.g., locations of blocks, filenames).Manages file-to-block mapping (e.g., Sample.txt is split into Block A, Block B, and Block C). Assigns blocks to different DataNodes for storage.

DataNodes

Actual storage locations for blocks. Each block is replicated across multiple DataNodes (ensuring fault tolerance).The figure shows three DataNodes (A, B, and C), each storing different copies of Blocks A, B, and C based on the replication factor.

Working of HDFS:

A file is split into fixed-size blocks (e.g., 64MB or 128MB). These blocks are distributed across different DataNodes for redundancy.The NameNode manages block locations but does not store actual data.The Hadoop File System Client communicates with the NameNode to fetch block locations and then retrieves data from DataNodes.This design ensures high availability, fault tolerance, and parallel processing for large-scale data applications.

HDFS Daemons

NameNode

- HDFS breaks a large file into smaller pieces called blocks.
- NameNode uses a rack ID to identify DataNodes in the rack.
- A rack is a collection of DataNodes within the cluster.
- NameNode keeps tracks of blocks of a file as it is placed on various DataNodes. NameNode manages file-related operations such as read, write, create, and delete. Its main job is managing the File System

Namespace.

- A file system namespace is collection of files in the cluster.
- NameNode stores HDFS namespace. File system namespace includes mapping of blocks to file, file properties and is stored in a file called FsImage.
- NameNode uses an EditLog (transaction log) to record every transaction that happens to the file system metadata. Refer Figure below.
- When NameNode starts up, it reads FsImage and EditLog from disk and applies all transactions from the EditLog to in-memory representation of the FsImage.
- Then it flushes out new version of FsImage on disk and truncates the old EditLog because the changes are updated in the FsImage.
- There is a single NameNode percluster.

DataNode

- There are multiple DataNodes per cluster.
- During Pipeline read and write DataNodes communicate with each other.
- A DataNode also continuously sends "heartbeat" message to NameNode to ensure the connectivity between the NameNode and DataNode.
- In case there is no heartbeat from a DataNode, the NameNode replicates that DataNode within the cluster and keeps on running as if nothing had happened.

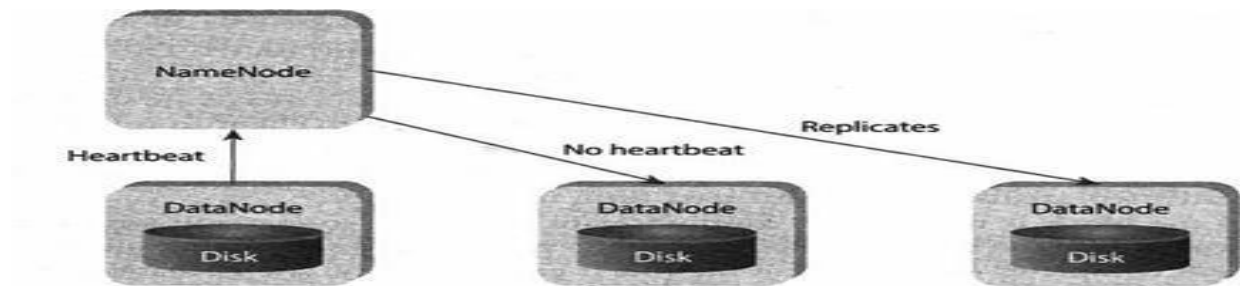


Figure 5.17 NameNode and DataNode Communication.

Heartbeat Mechanism:

Active DataNodes send heartbeats to the NameNode to indicate they are functioning. If a DataNode stops sending heartbeats (fails or disconnects), the NameNode assumes it is unavailable.

Data Replication:

When a DataNode fails (no heartbeat detected), the NameNode triggers replication of its data to another DataNode.

This ensures fault tolerance and high availability of data in HDFS.

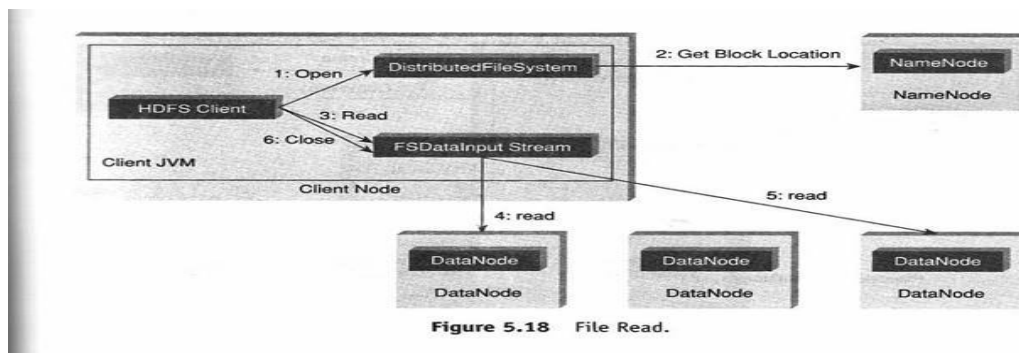
The mechanism helps detect node failures and automatically redistribute data, ensuring Hadoop's reliability and resilience.

Secondary NameNode

- The Secondary NameNode takes a snapshot of HDFS metadata at intervals specified in the Hadoop configuration.
- Since the memory requirements of Secondary NameNode are the same as NameNode, it is better to run NameNode and Secondary NameNode on different machines.
- In case of failure of the NameNode, the Secondary NameNode can be configured manually to bring up the cluster. However, the Secondary NameNode does not record any real-time changes that happen to the HDFS metadata.

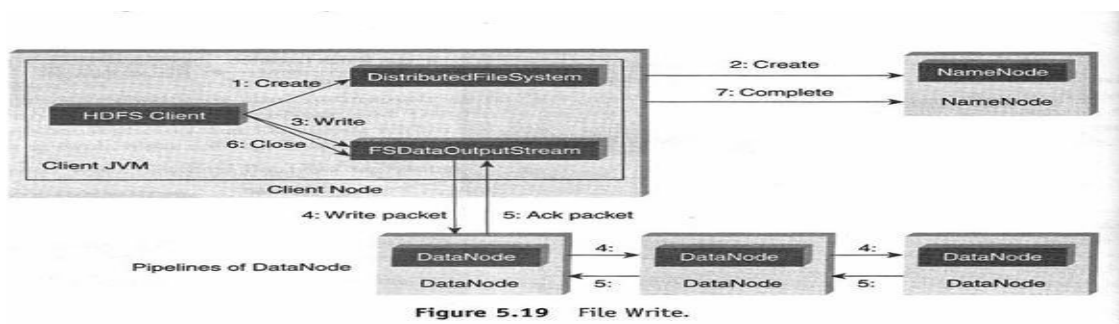
Anatomy of File read

The figure below describes the anatomy of file read



1. The client opens the file that it wishes to read from by calling `open()` on the `DistributedFileSystem`.
2. `DistributedFileSystem` communicates with the `NameNode` to get the location of data blocks. `NameNode` returns with the addresses of the `DataNodes` that the data blocks are stored on. Subsequent to this, the `DistributedFileSystem` returns an `FSDDataInputStream` to client to read from the file.
3. Client then calls `read()` on the stream `DFSInputStream`, which has addresses of the `DataNodes` for the first few blocks of the file, connects to the closest `DataNode` for the first block in the file.
4. Client calls `read()` repeatedly to stream the data from the `DataNode`.
5. When end of the block is reached, `DFSInputStream` closes the connection with the `DataNode`. It repeats the steps to find the best `DataNode` for the next block and subsequent blocks.
6. When the client completes the reading of the file, it calls `close()` on the `FSDDataInputStream` to close the connection.

Anatomy of File write



The steps involved in anatomy of File Write are as follows:

1. The client calls `create()` on `DistributedFileSystem` to create a file.
2. An RPC(Remote Procedure Call) call to the `NameNode` happens through the `DistributedFileSystem` to create a new file. The `NameNode` performs various checks to create a new file (checks whether such a file exists or not). Initially, the `NameNode` creates a file without associating any data blocks to the file. The `DistributedFileSystem` returns an `FSDDataOutputStream` to the client to perform write.
3. As the client writes data, data is split into packets by `DFSOutputStream`, which is then written to an internal queue, called data queue. `DataStream` consumes the data queue. The `DataStream` requests the `NameNode` to allocate new blocks by selecting a list of suitable `DataNodes` to store replicas. This list

of DataNodes makes a pipeline. Here, we will go with the default replication factor of three, so there will be three nodes in the pipeline for the first block.

4. DataStreamer streams the packets to the first DataNode in the pipeline. It stores packet and forwards it to the second DataNode in the pipeline. In the same way, the second DataNode stores the packet and forwards it to the third DataNode in the pipeline.

5. In addition to the internal queue, DFSOutputStream also manages an "Ack queue" of packets that are waiting for the acknowledgement by DataNodes. A packet is removed from the "Ack queue" only if it is acknowledged by all the DataNodes in the pipeline.

6. When the client finishes writing the file, it calls close() on the stream.

7. This flushes all the remaining packets to the DataNode pipeline and waits for relevant acknowledgments before communicating with the NameNode to inform the client that the creation of the file is complete.

Replica Placement Strategy

Hadoop Default Replica Placement Strategy: As per the Hadoop Replica Placement Strategy, first replica is placed on the same node as the client. Then it places second replica on a node that is present on different rack. It places the third replica on the same rack as second, but on a different node in the rack. Once replica locations have been set, a pipeline is built. This strategy provides good reliability. Figure below describes the typical replica pipeline.

Working with HDFS Commands

Objective: To get the list of directories and files at the root of HDFS.

`hadoop fs -ls /`

The command `hadoop fs -ls /` is used in Hadoop Distributed File System (HDFS) to list the contents of the root directory (/).

`hadoop fs` → Interface to interact with the Hadoop filesystem.

`-ls` → Lists files and directories in the specified path.

`/` → Refers to the root directory in HDFS.

Objective: To get the list of complete directories and files of HDFS.

`hadoop fs -ls -R /`

The command `hadoop fs -ls -R /` is used to recursively list all files and directories in Hadoop Distributed File System (HDFS) starting from the root (/). This command is useful for searching files, checking storage structure, and debugging file locations in HDFS.

`hadoop fs` → Interface to interact with HDFS.

-ls → Lists files and directories in the specified path.

-R (Recursive) → Lists files and directories recursively, including all subdirectories and their contents.

/ → Specifies the root directory of HDFS.

Objective: To create a directory (say, sample) in HDFS.

`hadoop fs -mkdir /sample`

The command `hadoop fs -mkdir /sample` is used to create a new directory named sample in the Hadoop Distributed File System (HDFS) root (/). It organizes data in HDFS by creating folders, stores files in structured directories and Prepares directories for Hadoop jobs.

`hadoop fs` → Interface to interact with HDFS.

`-mkdir` → Creates a new directory.

`/sample` → Specifies the directory path to be created in HDFS.

Objective: To copy a file from local file system to HDFS.

`hadoop fs -put /root/sample/test.txt /sample/test.txt`

The command `hadoop fs -put /root/sample/test.txt /sample/test.txt` is used to upload a file from the local file system to the Hadoop Distributed File System (HDFS).

`hadoop fs` → Interface to interact with HDFS.

`-put` → Copies a file from the local file system to HDFS.

`/root/sample/test.txt` → The source file located in the local file system.

`/sample/test.txt` → The destination path in HDFS where the file will be stored.

Use Cases:

Uploading data files to HDFS for processing with Hadoop/Spark. Transferring log files, CSVs, or datasets for big data analysis. Storing files in a distributed manner for fault tolerance and scalability.

Objective: To copy a file from HDFS to local file system.

`hadoop fs -get /sample/test.txt /local/path/testsample.txt`

`hadoop fs` → Interface to interact with HDFS.

`-get` → Downloads a file from HDFS to the local file system.

`/sample/test.txt` → Source file in HDFS.

`/local/path/testsample.txt` → Destination path in the local file system where the file will be stored.

Use Cases:

Retrieving processed results from HDFS to local storage. Copying data files from HDFS for analysis on a local machine. Exporting files for backup or sharing.

Objective: To copy a file from local file system to HDFS via `copyFromLocal` command.

```
hadoop fs -copyFromLocal /root/sample/test.txt /sample/testsample.txt
```

`hadoop fs -copyFromLocal`: Copies a local file to the Hadoop Distributed File System (HDFS).

`/root/sample/test.txt`: The source file (on the local file system).

`/sample/testsample.txt`: The destination file (on HDFS).

Objective: To copy a file from Hadoop file system to local file system via `copyToLocal` command.

```
hadoop fs -copyToLocal /sample/test.txt /root/sample/testsample1.txt
```

`hadoop fs -copyToLocal`: Copies a file from HDFS to the local file system.

`/sample/test.txt`: The source file (on HDFS).

`/root/sample/testsample1.txt`: The destination file (on the local system).

Objectives To display the contents of an HDFS file on console.

```
hadoop fs -cat /sample/test.txt
```

Displays the contents of `/sample/test.txt` (stored in HDFS) on the terminal.

Objective: To copy a file from one directory to another on HDFS.

```
hadoop fs -cp /sample/test.txt /sample1/
```

`hadoop fs -cp` → Copies a file within HDFS.

`/sample/test.txt` → Source file in HDFS.

`/sample1/` → Destination directory in HDFS. The trailing `/` ensures it is treated as a directory.

Objective: To remove a directory from HDFS.

```
hadoop fs -rm -r /sample1
```

`hadoop fs -rm` → Removes files from HDFS.

`-r` → Recursively deletes a directory and its contents.

`/sample1` → The target directory to be deleted.

Special Features of HDFS

1. Data Replication: There is absolutely no need for a client application to track all blocks. It directs the client to the nearest replica to ensure high performance.

2. Data Pipeline: A client application writes a block to the first DataNode in the pipeline. Then this DataNode takes over and forwards the data to the next node in the pipeline. This process continues for all the data blocks, and subsequently all the replicas are written to the disk.

PROCESSING DATA WITH HADOOP

MapReduce Programming is a software framework. MapReduce Programming helps you to process massive amounts of data in parallel. In MapReduce Programming, the input dataset is split into independent chunks. Map tasks process these independent chunks completely in a parallel manner. The output produced by the map tasks serves as intermediate data and is stored on the local disk of that server. The output of the mappers are automatically shuffled and sorted by the framework. MapReduce Framework sorts the output based on keys. This sorted output becomes the input to the reduce tasks. Reduce task provides reduced output by combining the output of the various mappers. Job inputs and outputs are stored in a file system. MapReduce framework also takes care of the other tasks such as scheduling, monitoring, re-executing failed tasks, etc.

Hadoop Distributed File System and MapReduce Framework run on the same set of nodes. This configuration allows effective scheduling of tasks on the nodes where data is present (Data Locality). This in turn results in very high throughput.

There are two daemons associated with MapReduce Programming. A single master JobTracker per cluster and one slave TaskTracker per cluster-node. The JobTracker is responsible for scheduling tasks to the TaskTrackers, monitoring the task, and re-executing the task just in case the TaskTracker fails. The TaskTracker executes the task.

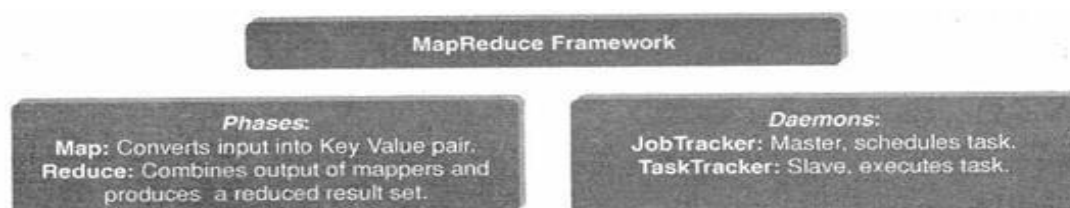


Figure 5.21 MapReduce Programming phases and daemons.

The MapReduce functions and input/output locations are implemented via the MapReduce applications. These applications use suitable interfaces to construct the job. The application and the job parameters together are known as job configuration. Hadoop job client submits job (jar/executable, etc.) to the JobTracker. Then it is the responsibility of Job Tracker to schedule tasks to the slaves. In addition to scheduling, it also monitors the task and provides status information to the job-client.

Mapreduce Daemons:

1. JobTracker: It provides connectivity between Hadoop and your application. When you submit code to cluster, Job Tracker creates the execution plan by deciding which task to assign to which node. It also monitors all the running tasks. When a task fails, it automatically re-schedules the task to a different node after a predefined number of retries. Job Tracker is a master daemon responsible for executing overall MapReduce job. There is a single Job Tracker per Hadoop cluster.

2. TaskTracker: This daemon is responsible for executing individual tasks that is assigned by the Job Tracker. There is a single TaskTracker per slave and spawns multiple Java Virtual Machines (JVMs) to handle multiple map or reduce tasks in parallel. TaskTracker continuously sends heartbeat message to JobTracker. When the Job Tracker fails to receive a heartbeat from a TaskTracker, the Job Tracker assumes that the TaskTracker has failed and resubmits the task to another available node in the cluster. Once the client submits a job to the Job Tracker, it partitions and assigns diverse MapReduce tasks for each Task Tracker in the cluster.

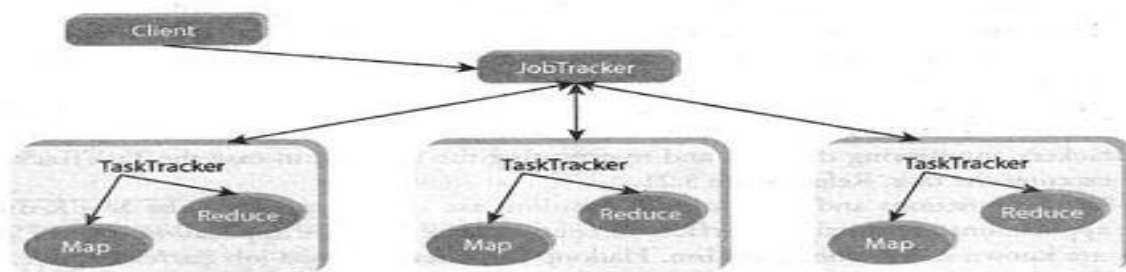


Figure 5.22 JobTracker and TaskTracker interaction.

How does Mapreduce work?

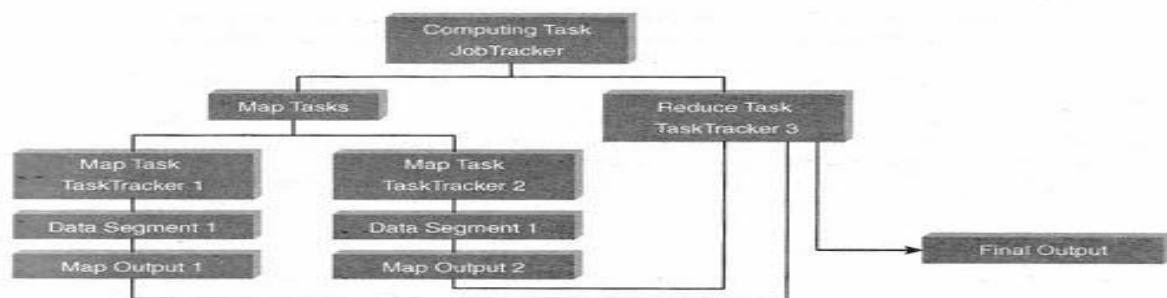


Figure 5.23 MapReduce programming workflow.

The above figure shows how the mapreduce programming works which is fundamental to Hadoop's data processing framework. Here's a step-by-step explanation of the workflow illustrated in the figure:

1. Computing Task (JobTracker):

The JobTracker is responsible for managing and scheduling jobs across the Hadoop cluster. It assigns Map Tasks and Reduce Tasks to available TaskTrackers (worker nodes).

2. Map Tasks Execution:

The JobTracker assigns Map Tasks to different TaskTrackers (worker nodes). Each TaskTracker processes a data segment (a portion of the input data). The map function runs on these segments to generate intermediate results called Map Outputs.

3. Reduce Task Execution: The intermediate Map Outputs are sent to a Reduce Task, which runs on another TaskTracker (TaskTracker 3). The Reducer aggregates, sorts, and processes the intermediate results to generate the final output.

4. Final Output Generation: The processed data from the Reduce Task is then written to the final output location.

- Job Tracker is the master node that assigns and monitors tasks.
- TaskTrackers execute individual Map and Reduce tasks.
- Map tasks process raw data and generate intermediate outputs.
- Reduce task processes the intermediate outputs and produces the final result.
- This model enables parallel processing, making it efficient for handling large-scale data. This figure effectively represents the distributed nature of Hadoop's MapReduce framework, where tasks are divided and executed across multiple nodes to enhance performance and scalability.

MAPREDUCE PROGRAMMING ARCHITECTURE

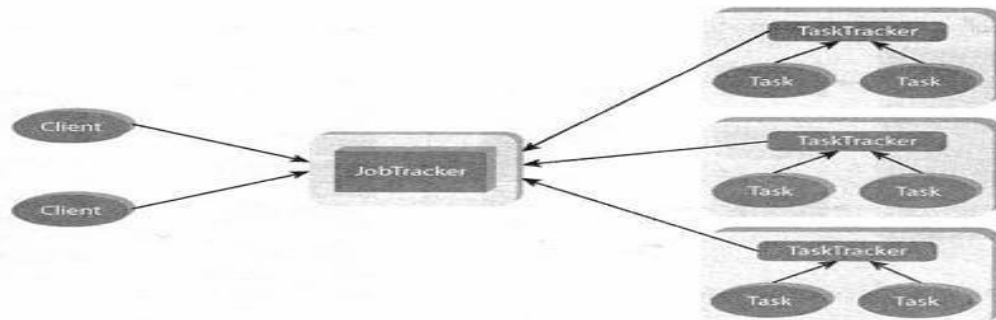


Figure 5.24 MapReduce programming architecture.

Figure above depicts Job the mapreduce programming architecture. The following steps describe how MapReduce performs its task.

1. First, the input dataset is split into multiple pieces of data (several small subsets).
2. Next, the framework creates a master and several workers processes and executes the worker processes remotely.
3. Several map tasks work simultaneously and read pieces of data that were assigned to each map task. The map worker uses the map function to extract only those data that are present on their server and generates key/value pair for the extracted data.
4. Map worker uses partitioner function to divide the data into regions. Partitioner decides which reducer should get the output of the specified mapper.
5. When the map workers complete their work, the master instructs the reduce workers to begin their work. The reduce workers in turn contact the map workers to get the key/value data for their partition. The data thus received is shuffled and sorted as per keys.
6. Then it calls reduce function for every unique key. This function writes the output to the file.
7. When all the reduce workers complete their work, the master transfers the control to the user program.

MapReduce Example

The famous example for MapReduce Programming is Word Count. For example, consider you need to count the occurrences of similar words across 50 files. You can achieve this using MapReduce Programming. Refer Figure.

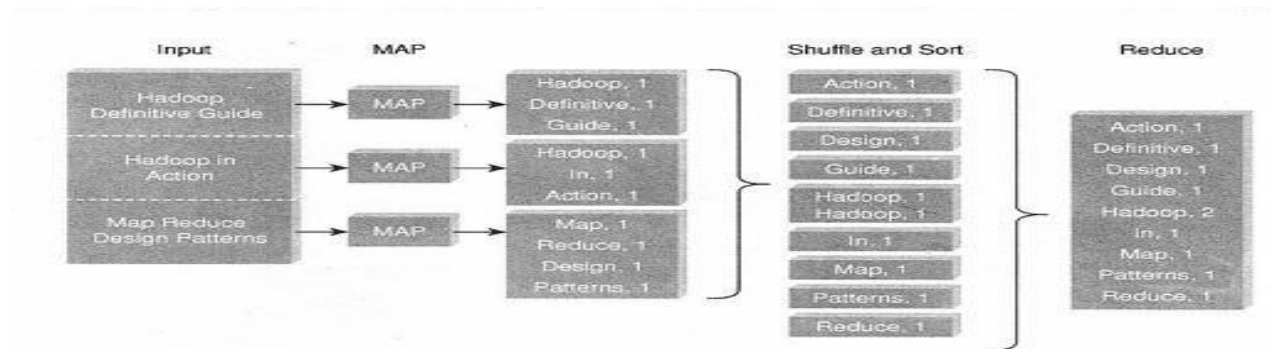


Figure 5.25 Wordcount example.

Word Count MapReduce Programming using Java

The MapReduce Programming requires three things.

1. Driver Class: This class specifies Job Configuration details.
2. Mapper Class: This class overrides the Map Function based on the problem statement.
3. Reducer Class: This class overrides the Reduce Function based on the problem statement.

Wordcounter.java: Driver Program

```
package com.app;

import java.io.IOException;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCounter {

    public static void main(String[] args) throws IOException, InterruptedException,
    ClassNotFoundException {

        // Create a new Job

        Job job = Job.getInstance();

        job.setJobName("wordcounter");
```

```
// Set the Jar file containing this class
job.setJarByClass(WordCounter.class);

// Set Mapper and Reducer classes
job.setMapperClass(WordCounterMap.class);
job.setReducerClass(WordCounterRed.class);

// Set output key-value types
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

// Set input and output paths
FileInputFormat.addInputPath(job, new Path("/sample/word.txt"));
FileOutputFormat.setOutputPath(job, new Path("/sample/wordcount"));

// Execute the job and wait for completion
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

Mapper Class: WordCounterMap.java

```
package com.app;

import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WordCounterMap 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 {
        // Convert line to lowercase and split by non-word characters
        String[] words = value.toString().toLowerCase().split("\\W+");
```

```
for (String w : words) {  
    if (!w.isEmpty()) {  
        word.set(w);  
        context.write(word, one); // Emit (word, 1)  
    }  
}  
}  
}
```

Reducer Class: WordCounterRed.java

```
package com.app;  
  
import java.io.IOException;  
  
import org.apache.hadoop.io.IntWritable;  
  
import org.apache.hadoop.io.Text;  
  
import org.apache.hadoop.mapreduce.Reducer;  
  
public class WordCounterRed extends Reducer<Text, IntWritable, Text,  
IntWritable> {  
  
    public void reduce(Text key, Iterable<IntWritable> values, Context context)  
        throws IOException, InterruptedException {  
  
        int sum = 0;  
  
        // Sum up all counts for each word  
        for (IntWritable val : values) {  
            sum += val.get();  
        }  
  
        // Emit (word, total_count)  
        context.write(key, new IntWritable(sum));  
    }  
}
```

Difference between SQL and Mapreduce

MANAGING RESOURCES AND APPLICATIONS WITH HADOOP YARN (YET ANOTHER RESOURCE NEGOTIATOR)

Apache Hadoop YARN is a sub-project of Hadoop 2.x. Hadoop 2.x is YARN-based architecture. It is a general processing platform. YARN is not constrained to MapReduce only. You can run multiple applications in Hadoop 2.x in which all applications share a common resource management. Now Hadoop can be used for various types of processing such as Batch, Interactive, Online, Streaming, Graph, and others.

Limitations of Hadoop 1.0 Architecture

In Hadoop 1.0, HDFS and MapReduce are Core Components, while other components are built around the core.

1. Single NameNode is responsible for managing entire namespace for Hadoop Cluster.
2. It has a restricted processing model which is suitable for batch-oriented MapReduce jobs.
3. Hadoop MapReduce is not suitable for interactive analysis.
4. Hadoop 1.0 is not suitable for machine learning algorithms, graphs, and other memory intensive algorithms.
5. MapReduce is responsible for cluster resource management and data processing.

In this Architecture, map slots might be "full", while the reduce slots are empty and vice versa. This causes resource utilization issues. This needs to be improved for proper resource utilization.

HDFS Limitation

NameNode saves all its file metadata in main memory. Although the main memory today is not as small and as expensive as it used to be two decades ago, still there is a limit on the number of objects that one can have in the memory on a single NameNode. The NameNode can quickly become overwhelmed with load on the system increasing.

In Hadoop 2.x, this is resolved with the help of HDFS Federation.

Hadoop 2: HDFS

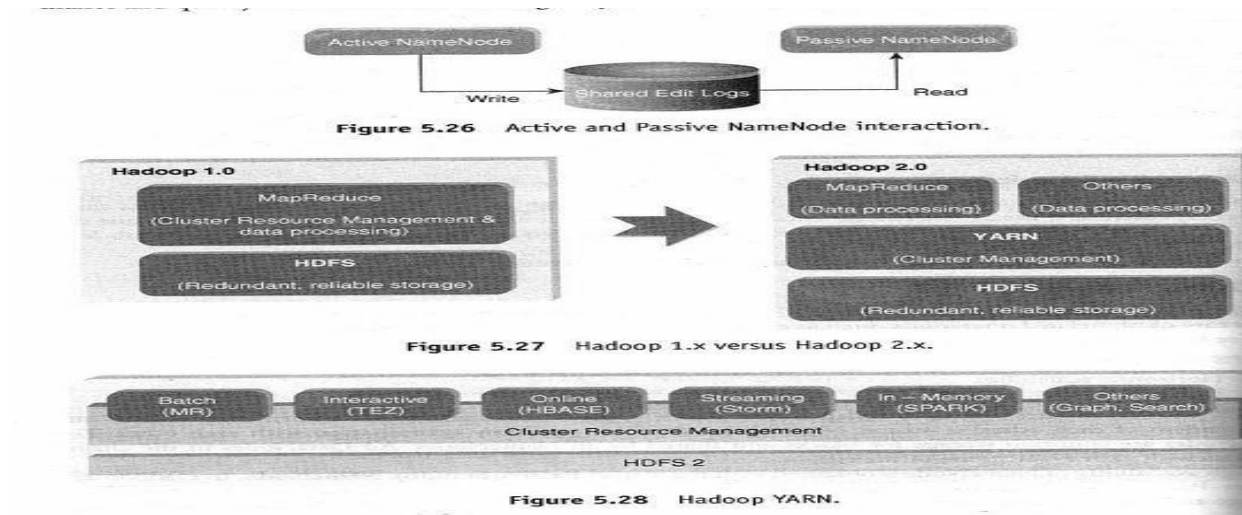
HDFS 2 consists of two major components: (a) namespace, (b) blocks storage service. Namespace service takes care of file-related operations, such as creating files, modifying files, and directories. The block storage service handles data node cluster management, replication.

HDFS 2 Features

1. Horizontal scalability.
2. High availability.

HDFS Federation uses multiple independent NameNodes for horizontal scalability. NameNodes are independent of each other. It means, NameNodes does not need any coordination with each other. The DataNodes are common storage for blocks and shared by all NameNodes. All DataNodes in the cluster

registers with each NameNode in the cluster. High availability of NameNode is obtained with the help of Passive Standby NameNode. In Hadoop 2.x, Active-Passive NameNode handles failover automatically. All namespace edits are recorded to a shared NFS storage and there is a single writer at any point of time. Passive NameNode reads edits from shared storage and keeps updated metadata information. In case of Active NameNode failure, Passive NameNode becomes an Active NameNode automatically. Then it starts writing to the shared storage. Figure below describes the Active-Passive NameNode interaction.



Hadoop 2 YARN: Taking Hadoop beyond Batch YARN helps us to store all data in one place. We can interact in multiple ways to get predictable performance and quality of services. This was originally architected by Yahoo. Refer Figure below.

Fundamental Idea

The fundamental idea behind this architecture is splitting the Job Tracker responsibility of resource management and Job Scheduling/Monitoring into separate daemons. Daemons that are part of YARN Architecture are described below.

1. A Global ResourceManager: Its main responsibility is to distribute resources among various applications in the system. It has two main components:

(a) Scheduler: The pluggable scheduler of ResourceManager decides allocation of resources to various running applications. The scheduler is just that, a pure scheduler, meaning it does NOT monitor or track the status of the application.

(b) ApplicationManager: ApplicationManager does the following:

- Accepting job submissions.
- Negotiating resources (container) for executing the application specific

ApplicationMaster. Restarting the ApplicationMaster in case of failure.

2. NodeManager: This is a per-machine slave daemon. NodeManager responsibility is launching the application containers for application execution. NodeManager monitors the resource usage such as memory, CPU, disk, network, etc. It then reports the usage of resources to the global ResourceManager.
3. Per-application ApplicationMaster: This is an application-specific entity. Its responsibility is to negotiate required resources for execution from the ResourceManager. It works along with the NodeManager for executing and monitoring component tasks.

Basic Concepts

Application:

1. Application is a job submitted to the framework.
2. Example - MapReduce Job.

Container:

1. Basic unit of allocation.
2. Fine-grained resource allocation across multiple resource types (Memory, CPU, disk, network, etc.)
 - (a) container_0 = 2GB, 1 CPU
 - (b) container_1 = 1GB, 6 CPU
3. Replaces the fixed map/reduce slots.

YARN ARCHITECTURE

The figure below shows YARN architecture.

The steps involved in YARN architecture is as shown below:

1. A client program submits the application which includes the necessary specifications to launch the application-specific ApplicationMaster itself.

2. The ResourceManager launches the ApplicationMaster by assigning some container.
3. The ApplicationMaster, on boot-up, registers with the ResourceManager. This helps the client program to query the ResourceManager directly for the details.
4. During the normal course, ApplicationMaster negotiates appropriate resource containers via the resource-request protocol.
5. On successful container allocations, the ApplicationMaster launches the container by providing the container launch specification to the NodeManager.
6. The NodeManager executes the application code and provides necessary information such as progress, status, etc. to its ApplicationMaster via an application-specific protocol.
7. During the application execution, the client that submitted the job directly communicates with the ApplicationMaster to get status, progress updates, etc. via an application-specific protocol.
8. Once the application has been processed completely, ApplicationMaster deregisters with the ResourceManager and shuts down, allowing its own container to be repurposed.

INTRODUCTION TO MAPREDUCE PROGRAMMING

Introduction

In MapReduce Programming, Jobs (Applications) are split into a set of map tasks and reduce tasks. Then these tasks are executed in a distributed fashion on Hadoop cluster. Each task processes small subset of data that has been assigned to it. This way, Hadoop distributes the load across the cluster. MapReduce job takes a set of files that is stored in HDFS (Hadoop Distributed File System) as input. Map task takes care of loading, parsing, transforming, and filtering. The responsibility of reduce task is grouping and aggregating data that is produced by map tasks to generate final output. Each map task is broken into the following phases:

1. RecordReader.
2. Mapper.
3. Combiner.
4. Partitioner.

The output produced by map task is known as intermediate keys and values. These intermediate keys and values are sent to reducer. The reduce tasks are broken into the following phases:

1. Shuffle.
2. Sort.
3. Reducer.
4. Output Format.

Hadoop assigns map tasks to the DataNode where the actual data to be processed resides. This ensures data locality. Data locality means that data is not moved over network; only computational code is moved to process data which saves network bandwidth.

Mapper

A mapper maps the input key-value pairs into a set of intermediate key-value pairs. Maps are individual tasks that have the responsibility of transforming input records into intermediate key-value pairs.

1. **RecordReader:** RecordReader converts a byte-oriented view of the input (as generated by the Input-Split) into a record-oriented view and presents it to the Mapper tasks. It presents the tasks with keys and values. Generally the key is the positional information and value is a chunk of data that constitutes the record.
2. **Map:** Map function works on the key-value pair produced by Record Reader and generates zero or more intermediate key-value pairs. The MapReduce decides the key-value pair based on the context.
3. **Combiner:** It is an optional function but provides high performance in terms of network bandwidth and disk space. It takes intermediate key-value pair provided by mapper and applies user-specific aggregate function to only that mapper. It is also known as local reducer.
4. **Partitioner:** The partitioner takes the intermediate key-value pairs produced by the mapper, splits them into shard, and sends the shard to the particular reducer as per the user-specific code. Usually, the key with same values goes to the same reducer. The partitioned data of each map task is written to the local disk of that machine and pulled by the respective reducer.

Reducer

The primary chore of the Reducer is to reduce a set of intermediate values (the ones that share a common key) to a smaller set of values. The Reducer has three primary phases: Shuffle and Sort, Reduce, and Output Format.

1. **Shuffle and Sort:** This phase takes the output of all the partitioners and downloads them into the local machine where the reducer is running. Then these individual data pipes are sorted by keys which produce larger data list. The main purpose of this sort is grouping similar words so that their values can be easily iterated over by the reduce task.
2. **Reduce:** The reducer takes the grouped data produced by the shuffle and sort phase, applies reduce function, and processes one group at a time. The reduce function iterates all the values associated with that key. Reducer function provides various operations such as aggregation, filtering, and combining data. Once it is done, the output (zero or more key-value pairs) of reducer is sent to the output format.
3. **Output Format:** The output format separates key-value pair with tab (default) and writes it out to a file using record writer. Figure describes the chores of Mapper, Combiner, Partitioner, and Reducer for the word count problem. The Word Count problem has been discussed under "Combiner" and "Partitioner".



Combiner

It is an optimization technique for MapReduce Job. Generally, the reducer class is set to be the combiner class. The difference between combiner class and reducer class is as follows:

1. Output generated by combiner is intermediate data and it is passed to the reducer.
2. Output of the reducer is passed to the output file on disk. The sections have been designed as follows:

Objective: What is it that we are trying to achieve here?

Input Data: What is the input that has been given to us to act upon?

Act: Output:

The actual statement/command to accomplish the task at hand. The result/output as a consequence of executing the statement.

Objective

To write a MapReduce program that counts the occurrences of words in a given text file and uses a Combiner for optimization.

1. Input Data

The input file contains sentences such as:

Welcome to Hadoop Session

Introduction to Hadoop

Introducing Hive

Hive Session

Pig Session

2. MapReduce Execution Flow

A. Mapper Phase

The Mapper reads the input file and splits it into words. It emits key-value pairs in the form (word, 1).

Example output from Mapper:

(Hadoop, 1)

(Session, 1)

(Introduction, 1)

(Hadoop, 1)

(Introducing, 1)

(Hive, 1)

(Hive, 1)

(Session, 1)

(Pig, 1)

(Session, 1)

B. Combiner Phase (Optimization)

The Combiner runs after the Mapper to locally aggregate word counts before passing data to the Reducer.

This reduces the amount of data transferred across the network.

Example output from Combiner:

(Hadoop, 2)

(Session, 3)

(Hive, 2)

(Pig, 1)

(Introduction, 1)

(Introducing, 1)

C. Shuffle and Sort Phase

The framework groups all occurrences of the same word together.

D. Reducer Phase

The Reducer receives the grouped data and sums up the counts.

Example output from Reducer:

Hadoop 2

Hive 2

Introducing 1

Introduction 1

Pig 1

Session 3

3. Driver Program Implementation

The Combiner class is set in the driver program:

```
job.setCombinerClass(WordCounterRed.class);
```

The input and output paths are configured:

```
FileInputFormat.addInputPath(job, new Path("/mapreducedemos/lines.txt"));
```

```
FileOutputFormat.setOutputPath(job,newPath("/mapreducedemos/output/wordcount/"));
```

The MapReduce job is executed using:

```
hadoop jar <jar_name> <driver_class> <input_path> <output_path>
```

4. Output Storage

The reducer output is stored in part-r-00000 under the output directory:/mapreducedemos/output/wordcount/part-r-00000

Partitioner

- The partitioning phase happens after the map phase and before the reduce phase.
- The number of partitions equals the number of reducers.
- The default partitioner in Hadoop is the hash partitioner, but custom partitioners can be implemented.

Objective of the Exercise

- Implement a MapReduce program to count word occurrences.
- Use a custom partitioner to divide words based on their starting alphabet.
- This ensures that words beginning with the same letter are sent to the same reducer.

Input Data Example

- Welcome to Hadoop Session
- Introduction to Hadoop
- Introducing Hive

- Hive Session

- Pig Session

Implementation of the Custom Partitioner (WordCountPartitioner.java)

- Extends Partitioner<Text, IntWritable>.
- Extracts the first letter of each word.
- Assigns a partition number based on this letter.
- Uses a switch-case structure for letters A–Z, with a default partition for non-alphabet characters.

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class WordCountPartitioner extends Partitioner<Text, IntWritable>
{
    @Override
    public int getPartition(Text key, IntWritable value, int numPartitions) {
        String word = key.toString();
        char alphabet = word.toUpperCase().charAt(0);
        int partitionNumber = 0;
        switch (alphabet) {
            case 'A': partitionNumber = 1; break;
            case 'B': partitionNumber = 2; break;
            case 'C': partitionNumber = 3; break;
            case 'D': partitionNumber = 4; break;
            case 'E': partitionNumber = 5; break;
            case 'F': partitionNumber = 6; break;
            case 'G': partitionNumber = 7; break;
            case 'H': partitionNumber = 8; break;
            case 'I': partitionNumber = 9; break;
            case 'J': partitionNumber = 10; break;
```

```

case 'K': partitionNumber = 11; break;
case 'L': partitionNumber = 12; break;
case 'M': partitionNumber = 13; break;
case 'N': partitionNumber = 14; break;
case 'O': partitionNumber = 15; break;
case 'P': partitionNumber = 16; break;
case 'Q': partitionNumber = 17; break;
case 'R': partitionNumber = 18; break;
case 'S': partitionNumber = 19; break;
case 'T': partitionNumber = 20; break;
case 'U': partitionNumber = 21; break;
case 'V': partitionNumber = 22; break;
case 'W': partitionNumber = 23; break;
case 'X': partitionNumber = 24; break;
case 'Y': partitionNumber = 25; break;
case 'Z': partitionNumber = 26; break;
default: partitionNumber = 0; // Non-alphabet words
}
return partitionNumber;
}
}

```

Sets 27 reducers (one for each letter A-Z and a default).

Uses `setPartitionerClass` to specify the custom partitioner.

```
job.setNumReduceTasks(27);
```

```
job.setPartitionerClass(WordCountPartitioner.class);
```

// Input and Output Paths

```
FileInputFormat.addInputPath(job, new Path("/mapreducedemos/lines.txt"));
```

```
FileOutputFormat.setOutputPath(job,newPath("/mapreducedemos/output/wordcountpartitioner/"));
```

- The output directory contains 27 partitioned files.

- Each partition file corresponds to words starting with a specific letter.
- Example: part-r-00008 corresponds to words starting with 'H'.

Each line is broken into words:

- Welcome
- to
- Hadoop
- Session
- Introduction
- to
- Hadoop
- Introducing
- Hive
- Hive
- Session
- Pig
- Session

After mapping, each word gets a count of 1:(Words emitted by the mapper)

- Welcome → 1
- to → 1
- Hadoop → 1
- Session → 1
- Introduction → 1
- to → 1
- Hadoop → 1
- Introducing → 1
- Hive → 1
- Hive → 1
- Session → 1
- Pig → 1

- Session → 1

Each word is assigned to a reducer based on its first letter.

Each word is assigned to a reducer according to the **first letter of the word**. The partition number is calculated based on the alphabetical position of the first letter.

Word	First Letter	Partition Number
Welcome	W	23
to	T	20
Hadoop	H	8
Session	S	19
Introduction	I	9
Introducing	I	9
Hive	H	8
Pig	P	16

In this partitioning method, all words starting with the same letter are sent to the **same reducer**. For example, both *Introduction* and *Introducing* start with the letter **I**, so they are assigned to **partition 9**. Similarly, *Hadoop* and *Hive* start with **H**, so they go to **partition 8**. This ensures that similar keys are grouped together during the reduce phase.

Each reducer generates an output file with words starting with a specific letter. File: part-r-00008 (Reducer for 'H')

Hadoop 2

Hive 2

File: part-r-00009 (Reducer for 'I')

Introduction 1

Introducing 1

File: part-r-00016 (Reducer for 'P')

Pig 1

File: part-r-00019 (Reducer for 'S')

Session 3

File: part-r-00020 (Reducer for 'T')

to 2

File: part-r-00023 (Reducer for 'W')

Welcome 1

Final Output Directory

/mapreducedemos/output/wordcountpartitioner/

|— part-r-00008 (Words starting with H)

|— part-r-00009 (Words starting with I)

|— part-r-00016 (Words starting with P)

|— part-r-00019 (Words starting with S)

|— part-r-00020 (Words starting with T)

|— part-r-00023 (Words starting with W)

Searching

Objective

Implement a MapReduce program to search for a specific keyword in a file. The program will output lines containing the keyword along with file name and position.

Input Data (student.csv)

1001,John,45

1002,Jack,39

1003,Alex,44

1004,Smith,38

1005,Bob,33

Each row contains ID, Name, and Score.

The keyword to search is "Jack".

The MapReduce job consists of three components:

Driver Program (WordSearcher.java)

Mapper Class (WordSearchMapper.java)

Reducer Class (WordSearchReducer.java)

(a) Driver Program (WordSearcher.java)

This configures the Hadoop job and specifies the Mapper, Reducer, and input/output paths

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordSearcher {

    public static void main(String[] args) throws IOException, InterruptedException,
    ClassNotFoundException {

        Configuration conf = new Configuration();
        Job job = new Job(conf, "Word Search");
        job.setJarByClass(WordSearcher.class);
        job.setMapperClass(WordSearchMapper.class);
        job.setReducerClass(WordSearchReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        // Setting the keyword to search
        job.getConfiguration().set("keyword", "Jack");

        // Set input and output paths
        FileInputFormat.setInputPaths(job, new Path("/mapreduce/student.csv"));
        FileOutputFormat.setOutputPath(job, new Path("/mapreduce/output/search/"));
        job.setNumReduceTasks(1);

        // Run the job and exit
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

}

}

□ Configures the job, sets the keyword, and specifies input and output paths.

□ Uses only one reducer for simplicity.

(b) Mapper Class (WordSearchMapper.java)

The Mapper scans each line, searches for the keyword, and outputs matching lines.

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class WordSearchMapper extends Mapper<LongWritable, Text, Text, Text> {

    static String keyword;

    static int pos = 0;

    @Override

    protected void setup(Context context) throws IOException, InterruptedException {

        // Get the keyword from the configuration

        Configuration configuration = context.getConfiguration();

        keyword = configuration.get("keyword");

    }

    @Override

    protected void map(LongWritable key, Text value, Context context) throws

        IOException, InterruptedException {

        // Get file name

        FileSplit fileSplit = (FileSplit) context.getInputSplit();

        String fileName = fileSplit.getPath().getName();

        // Track word position

        Integer wordPos;
```

```
pos++;
// If line contains the keyword, write output
if (value.toString().contains(keyword)) {
    wordPos = pos;
    context.write(value, new Text(fileName + "," + wordPos.toString()));
}
}
}
```

- Extracts the keyword from the job configuration (setup method).
- Searches each line (map method) and writes matches with file name & position.

(c) Reducer Class (WordSearchReducer.java)

The Reducer simply writes the filtered results from the Mapper.

```
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class WordSearchReducer extends Reducer<Text, Text, Text, Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws
        IOException, InterruptedException {
        for (Text value : values) {
            context.write(key, value);
        }
    }
}
```

- Receives filtered results and writes them as final output.
- No additional processing needed since the Mapper does the filtering.

After running the job, the output is stored in Hadoop HDFS at:

/mapreduce/output/search/part-r-00000

Content of part-r-00000:

1002,Jack,39 student.csv,5

□ The program correctly identifies and outputs the row that contains "Jack".

Sorting

Objective to write a Mapreduce program to sort data by student name.

Input data (stored in student.csv file)

1001,John,45

1002,Jack,39

1003,Alex,44

1004,Smith,38

1005,Bob,33

```
import java.io.IOException;
```

```
import org.apache.hadoop.conf.Configuration;
```

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

```
import org.apache.hadoop.io.LongWritable;
```

```
import org.apache.hadoop.io.NullWritable;
```

```
import org.apache.hadoop.io.Text;
```

```
import org.apache.hadoop.mapreduce.Job;
```

```
import org.apache.hadoop.mapreduce.Mapper;
```

```
import org.apache.hadoop.mapreduce.Reducer;
```

```
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
```

```
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

```
public class SortStudNames {
```

```
// Mapper Class
```

```
public static class SortMapper extends Mapper<LongWritable, Text, Text, Text> {
```

```
protected void map(LongWritable key, Text value, Context context) throws
```

```
IOException, InterruptedException {
```

```
String[] token = value.toString().split(",");
```

```

context.write(new Text(token[1]), new Text(token[0] + "," + token[2])); // Key:
Name, Value: ID,Score
}
}

// Reducer Class

public static class SortReducer extends Reducer<Text, Text, NullWritable, Text> {
    public void reduce(Text key, Iterable<Text> values, Context context) throws
        IOException, InterruptedException {
        for (Text details : values) {
            context.write(NullWritable.get(), new Text(key.toString() + "," +
                details.toString()));
        }
    }
}

// Driver Class

public static void main(String[] args) throws IOException, InterruptedException,
    ClassNotFoundException {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "Sort Students by Name");
    job.setJarByClass(SortStudNames.class);
    job.setMapperClass(SortMapper.class);
    job.setReducerClass(SortReducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);
    job.setOutputKeyClass(NullWritable.class);
    job.setOutputValueClass(Text.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}

```

```
}
```

```
}
```

Output:

Alex,1003,44

Bob,1005,33

Jack,1002,39

John,1001,45

Smith,1004,38

Command to run the program:

```
hadoop jar StudentSort.jar SortStudNames /mapreduce/student.csv /mapreduce/output
```

After execution, check output:

```
hdfs dfs -cat /mapreduce/output/part-r-000000
```

COMPRESSION

In MapReduce programming, you can compress the MapReduce output file. Compression provides two benefits as follows:

1. Reduces the space to store files.
2. Speeds up data transfer across the network.

You can specify compression format in the Driver Program as shown below:

```
conf.setBoolean("mapred.output.compress", true);
```

```
conf.setClass("mapred.output.compression.codec",GzipCodec.class,CompressionCodec.class);
```

Here, codec is the implementation of a compression and decompression algorithm.

GzipCodec is the compression algorithm for gzip. This compresses the output file.

MODULE-3

Introduction to MongoDB: What is MongoDB, Why MongoDB, Terms used in RDBMS and MongoDB, Data Types in MongoDB, MongoDB Query Language.

TB1: Ch 6: 6.1-6.5

3.1 What is MongoDB?

MongoDB is

1. Cross-platform.
2. Open source.
3. Non-relational.
4. Distributed.
5. NoSQL.
6. Document-oriented data store.

3.2 WHY MongoDB?

Few of the major challenges with traditional RDBMS are dealing with large volumes of data, rich variety of data-particularly unstructured data, and meeting up to the scale needs of enterprise data. The need is for a database that can scale out or scale horizontally to meet the scale requirements, has flexibility with respect to schema, is fault tolerant, is consistent and partition tolerant, and can be easily distributed over a multitude of nodes in a cluster.

1. Full index support
2. Rich query language
3. Auto sharding
4. Document oriented
5. High performance
6. Fast in-place updates
7. Replication
8. Easy scalability
9. High availability

3.2.1 Using Java Script Object Notation(JSON)

- JSON is extremely expressive.
- MongoDB actually does not use JSON but BSON – it is Binary JSON. It is an open standard. It is used to store complex data structures.

Let us trace the journey from .csv to XML to JSON:

Let us look at how data is stored in .csv file.

Assume that this data is about the employees of an organization named "XYZ". As we can see below, the column values are separated using commas and the rows are separated by a carriage return.

John, Mathews, +123 4567 8900

Andrews, Symmonds, +4567890 1234

Mable, Mathews, +789 1234 5678

However it can be made slightly more legible by adding column heading. FirstName, LastName, ContactNo

John, Mathews, +1234567 8900

Andrews, Symmonds, +4567890 1234

Mable. Mathews, +789 12345678

Challenges with CSV Format

1. Flat Structure: CSV works best with flat, non-repeating data.
2. Multiple Values Problem: Some employees have multiple Office and Home contact numbers. Some have multiple email addresses (2, 3, or more).
3. Merge Complexity: When different departments use different CSV formats, merging becomes tedious and error-prone. Field inconsistency and missing/extra columns further complicate consolidation.

Why XML is Not Ideal for Simpler Use Cases

Pros:

- XML supports complex and hierarchical data structures.
- Suitable for highly structured data formats.

Cons:

- Too verbose and heavy for simple data exchange.
- Requires definition of data structure using schemas or DTDs.
- Overkill for lightweight, frequently changing employee records.

JSON as an Effective Alternative

1. Extensible and lightweight.
2. Handles arrays/lists of data naturally (e.g., multiple contacts or emails).
3. Easy to read and write.
4. Excellent for web applications and APIs.

```
{  
  "FirstName": "John",  
  "LastName": "Mathews",  
  "ContactNo": ["+123 45678900", "+123 4444 5555"],  
  "Emails": ["john@example.com", "john.mathews@work.com"]  
},  
{  
  "FirstName": "Andrews",  
  "LastName": "Symmonds",  
  "ContactNo": ["+456 7890 1234", "+456 6666 7777"]  
},  
{  
  "FirstName": "Mable",  
  "LastName": "Mathews",  
  "ContactNo": ["+789 1234 5678"]  
}
```

JSON is very expressive. It provides the much needed ease to store and retrieve documents in their real form. The binary form of JSON is BSON. BSON is an open standard. In most cases it consumes less space as compared to the text-based JSON. There is yet another advantage with BSON. It is much easier and quicker to convert BSON to a programming language's native data format. There are MongoDB drivers available for a number of programming languages such as C, C++, Ruby, PHP, Python, C#, etc., and each works slightly differently. Using the basic binary format enables the native data structures to be built quickly for each language without going through the hassle of first processing JSON.

3.2.2 Creating or generating a Unique key

- Each JSON document should have a unique identifier.
- It is the `_id` key.

- It is similar to the primary key in relational databases.
- This facilitates search for documents based on the unique identifier.
- An index is automatically built on the unique identifier.
- It is your choice to either provide unique values yourself or have the mongo shell generate the same.

3.2.2.1 Database

It is a collection of collections. In other words, it is like a container for collections. It gets created the first time that your collection makes a reference to it. This can also be created on demand. Each database gets its own set of files on the file system. A single MongoDB server can house several databases.

3.2.2.2 Collection

A collection is analogous to a table of RDBMS. A collection is created on demand. It gets created the first time that you attempt to save a document that references it. A collection exists within a single database. A collection holds several MongoDB documents. A collection does not enforce a schema. This implies that documents within a collection can have different fields. Even if the documents within a collection have same fields, the order of the fields can be different.

3.2.2.3 Document

A document is analogous to a row/record/tuple in an RDBMS table. A document has a dynamic schema. This implies that a document in a collection need not necessarily have the same set of fields/key-value pairs. Shown in Figure below is a collection by the name "students" containing three documents.

3.2.3 Support for Dynamic Queries

MongoDB has extensive support for dynamic queries. This is in keeping with traditional RDBMS wherein we have static data and dynamic queries. CouchDB, another document-oriented, schema-less NoSQL data-base and MongoDB's biggest competitor, works on quite the reverse philosophy. It has support for dynamic data and static queries.

MongoDB: Dynamic Queries, Static Data MongoDB allows you to build dynamic queries using a rich and expressive query language. You can query on any field, including nested documents and arrays. This aligns with the traditional RDBMS approach, where the data structure (schema) is fixed, and the queries are dynamic.

Example: You can construct queries at runtime, filter based on various fields, and use advanced operators (\$gt, \$in, \$or, etc.).

CouchDB: Static Queries, Dynamic Data

CouchDB uses MapReduce views to query data.

Once a view (which is essentially a query) is defined, it becomes static—you cannot alter it dynamically without redefining the view. The data is more flexible, and each document can have a completely different structure. This allows more flexibility in storing heterogeneous data but limits ad-hoc querying unless pre-defined.

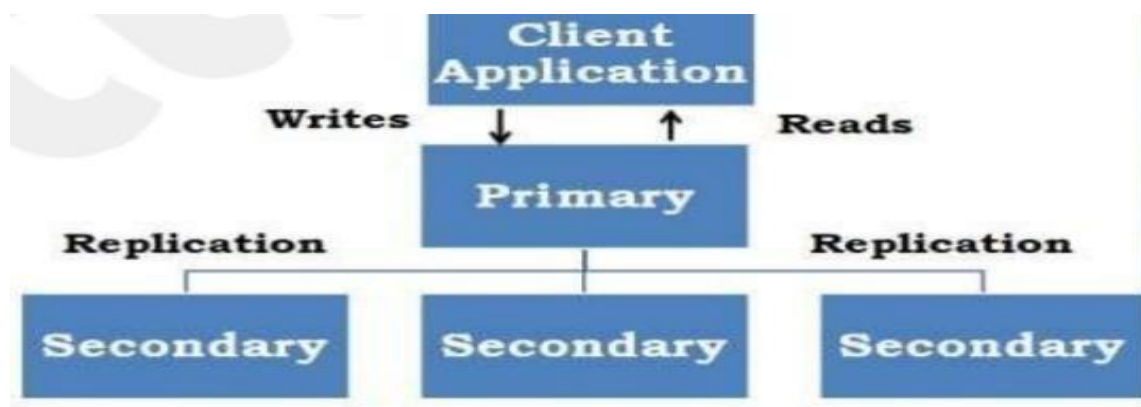
3.2.4 Storing Binary Data

MongoDB provides GridFS to support the storage of binary data. It can store up to 4 MB of data. This usually suffices for photographs (such as a profile picture) or small audio clips. However, if one wishes to store movie clips, MongoDB has another solution. It stores the metadata (data about data along with the context information) in a collection called "file". It then breaks the data into small pieces called chunks and stores it in the "chunks" collection. This process takes care about the need for easy scalability.

3.2.5 Replication

Why replication?

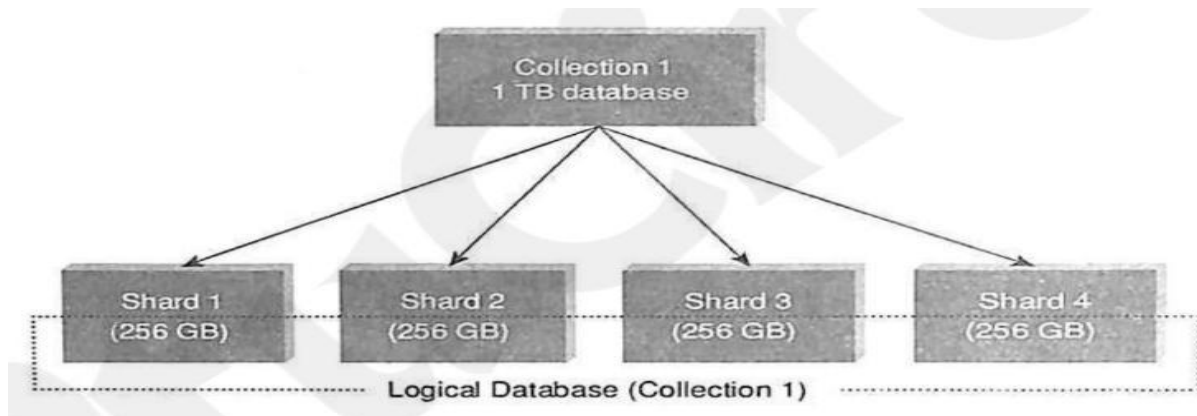
It provides data redundancy and high availability. It helps to recover from hardware failure and service interruptions. In MongoDB, the replica set has a single primary and several secondaries. Each write request from the client is directed to the primary. The primary logs all write requests into its Oplog (operations log). The Oplog is then used by the secondary replica members to synchronize their data. This way there is strict adherence to consistency. Refer Figure. The clients usually read from the primary. However, the client can also specify a read preference that will then direct the read operations to the secondary.



3.2.6 Sharding

Sharding is akin to horizontal scaling. It means that the large dataset is divided and distributed over multiple servers or shards. Each shard is an independent database and collectively they would constitute a logical database. The prime advantages of sharding are as follows:

1. Sharding reduces the amount of data that each shard needs to store and manage. For example, if the dataset was 1 TB in size and we were to distribute this over four shards, each shard would house just 256 GB data. As the cluster grows, the amount of data that each shard will store and manage will decrease.
2. Sharding reduces the number of operations that each shard handles. For example, if we were to insert data, the application needs to access only that shard which houses that data.



3.2.7 Updating Information In-Place

MongoDB updates the information in-place. This implies that it updates the data wherever it is available. It does not allocate separate space and the indexes remain unaltered. MongoDB is all for lazy-writes. It writes to the disk once every second. Reading and writing to disk is a slow operation as compared to reading and writing from memory. The fewer the reads and writes that we perform to the disk, the better is the performance. This makes MongoDB faster than its other competitors who write almost immediately to the disk. However, there is a tradeoff. MongoDB makes no guarantee that data will be stored safely on the disk.

3.3 Terms Used in RDBMS and MongoDB (Structural Differences)

RDBMS Term	MongoDB Term	Description
Database	Database	A logical grouping of data. Both RDBMS and MongoDB use the term <i>database</i> to organize data at the highest level.
Table	Collection	A table in RDBMS stores data in rows, whereas a collection in MongoDB stores multiple documents.
Row	Document	A row in RDBMS corresponds to a document in MongoDB. A document is a JSON-like structure stored in BSON format and contains data in key–value pairs.
Column	Field	A column in RDBMS represents an attribute of data, which is referred to as a field in MongoDB.

RDBMS Term	MongoDB Term	Description
Primary Key	<code>_id</code> Field	In MongoDB, every document contains a unique <code>_id</code> field that acts as the primary key and uniquely identifies the document.

Relationships and Joins

RDBMS Term	MongoDB Equivalent	Description
Foreign Key	Reference (Manual <code>\$lookup</code>) or	In MongoDB, relationships are represented by storing references to documents in other collections instead of using foreign key constraints.
JOIN Operation	Embedding or <code>\$lookup</code>	RDBMS uses JOIN operations to retrieve related data from multiple tables. MongoDB achieves similar functionality either by embedding related documents within a document or by using the <code>\$lookup</code> aggregation stage.

Indexing and Querying

RDBMS Term	MongoDB Equivalent	Description
Index	Index	Both RDBMS and MongoDB use indexes to improve query performance. MongoDB supports single-field, compound, text, and geospatial indexes.
SQL Query (SELECT, WHERE, etc.)	Find Query (db.collection.find())	RDBMS uses SQL queries such as SELECT and WHERE, whereas MongoDB uses a JavaScript-like query syntax with the <code>find()</code> method.
Aggregate Functions (SUM, COUNT)	AVG, Aggregation Pipeline	MongoDB replaces SQL aggregate functions with its aggregation framework, using stages such as <code>\$group</code> , <code>\$sum</code> , <code>\$avg</code> , and <code>\$count</code> .

Transactions and Consistency

RDBMS Term	MongoDB Equivalent	Description
ACID Transactions	Multi-document Transactions	RDBMS supports ACID compliance by default. MongoDB introduced multi-document transactions in version 4.0 to provide similar transactional behavior across multiple documents and collections.

RDBMS Term	MongoDB Equivalent	Description
Commit & Rollback	Session-Based Transactions	In MongoDB, commit and rollback operations are handled through session-based transactions, allowing changes to be committed or aborted as a single unit of work.

Session-based operation refers to processes where a user's interactions are tracked across multiple requests using a unique session ID. It allows continuity, like maintaining login state or a shopping cart, throughout a user's visit.

Performance and Scaling

RDBMS Term	MongoDB Equivalent	Description
Vertical Scaling	Horizontal Scaling	RDBMS typically scales vertically by upgrading existing hardware resources, whereas MongoDB scales horizontally by distributing data across multiple nodes using sharding.
Partitioning	Sharding	In MongoDB, data is distributed across multiple servers using sharding to improve performance and scalability.
Replication (Master-Slave)	Replication (Replica Sets)	Both RDBMS and MongoDB support replication, but MongoDB uses replica sets to provide automatic failover, high availability, and redundancy.

Backup and Recovery

RDBMS Term	MongoDB Equivalent	Description
Backup & Restore (mysqldump, pg_dump)	mongodump & mongorestore	MongoDB provides utilities such as mongodump and mongorestore to create backups and restore data, similar to dump and restore tools used in RDBMS.
Point-in-Time Recovery	Oplog Replay	In MongoDB, the oplog (operations log) allows point-in-time recovery by replaying operations in replica sets to restore data to a specific moment.

Data Integrity and Constraints

RDBMS Term	MongoDB Equivalent	Description
Triggers	Change Streams	RDBMS uses triggers to automatically execute actions in response to data changes. MongoDB provides change streams to notify applications of data changes in real time.

Security and Authentication

RDBMS Term	MongoDB Equivalent	Description
User Roles & Permissions	Role-Based Access Control (RBAC)	MongoDB uses role-based access control to provide fine-grained security by assigning specific privileges to users based on their roles.
Authentication (LDAP, Kerberos, etc.)	Authentication with LDAP, Kerberos, SCRAM	MongoDB supports multiple authentication mechanisms, including LDAP, Kerberos, and SCRAM-SHA, to verify user identities securely.
Encryption	Encryption at Rest & In Transit	MongoDB supports TLS/SSL for encrypting data in transit and uses AES-256 encryption to secure data stored at rest.

3.3.1 Create Database

Creating a Database

Syntax:

```
use DATABASE_Name
```

Example:

To create a database named myDB, use:

```
use myDB
```

Output:

```
switched to db myDB
```

Verifying the Current Database

To check which database you are currently using:

```
db
```

Output:

```
myDB
```

Listing All Databases

To list all existing databases:

```
show dbs
```

Output (example):

```
admin (empty)
```

```
local 0.078GB
```

```
test 0.078GB
```

- The newly created database (e.g., myDB) does not appear in the list from show dbs until it contains at least one document.
- The default database in MongoDB is test.

If no database is explicitly created, any collections inserted will be stored in the test database.

3.3.2 Drop Database

Syntax to Drop a Database

```
db.dropDatabase();
```

Steps to Drop a Specific Database (e.g., "myDB")

1. First, switch to the database you want to drop:
2. use myDB;
3. Then execute the drop command:
4. db.dropDatabase();

Confirmation Message

After running the command, MongoDB returns:

```
{ "dropped" : "myDB", "ok" : 1 }
```

- Always ensure you are in the correct database before executing

```
db.dropDatabase().
```

- If no database is selected, MongoDB will drop the default database test.

Data Types in MongoDB

String	Must be UTF-8 valid. Most commonly used data type.
Integer	Can be 32-bit or 64-bit (depends on the server).
Boolean	To store a true/false value.
Double	To store floating point (real values).
Min/Max keys	To compare a value against the lowest or highest BSON elements.
Arrays	To store arrays or list or multiple values into one key.
Timestamp	To record when a document has been modified or added.
Null	To store a NULL value. A NULL is a missing or unknown value.
Date	To store the current date or time in Unix time format. One can create object of date and pass day, month and year to it.
Object ID	To store the document's id.
Binary data	To store binary data (images, binaries, etc.).
Code	To store javascript code into the document.
Regular expression	To store regular expression.

1. String

In MongoDB, strings must be UTF-8 encoded.

Ex: { "name": "John Doe" }

2. Integer

MongoDB differentiate between 32-bit(Int32) and 64-bit(Int64) integer

Ex: { "age": 25 }

3. Double

Ex: { "Salary": 1900.54 }

4. Boolean

Ex: { "isActive": true }

5. Array

Stores multiple values in a single field

Ex: { "skills": ["JavaScript", "Python", "MongoDB"] }

6. Object(Embedded document or subdocument)

Stores a document inside another document.

Ex : { "name": "Alice", "address": { "city": "New York", "zip": "10001" } }

7. ObjectId

A 12-byte identifier (timestamp, machine ID, process ID, and counter).

Ex: { "_id": ObjectId("507f1f77bcf86cd799439011") }

8. Date

Default format: `ISODate("YYYY-MM-DDTHH:MM:SSZ")`.

Ex: `{ "createdAt": ISODate("2024-03-25T10:30:00Z") }`

9. Null

Ex: `{ "deletedAt": null }`

10. Binary Data

Stores binary data such as images, audio, or encrypted data.

Ex : `{ "profilePicture": BinData(0, "base64EncodedData") }`

11. Regular Expression

Stores and queries strings using regex patterns.

Ex: `{ "pattern": /mongodb/i }`

12. JavaScript Code

Ex: `{ "script": function() { return "Hello MongoDB"; } }`

13. JavaScript with Scope

Similar to javascript, but allows defining scope (variables) for the script.

Ex: `{ "script": { "$code": "function(x) { return x * 2; }", "$scope": { "x": 10 } } }`

14. Timestamp

Stores a high-precision timestamp (used for internal MongoDB operations).

Ex: `{ "createdAt": Timestamp(1618928492, 1) }`

15. Decimal128

High-precision 128-bit decimal numbers (useful for financial calculations).

Ex: `{ "amount": NumberDecimal("1234.5678") }`

16. MinKey & MaxKey

MinKey: Represents the lowest possible value in MongoDB (useful for sorting).

MaxKey: Represents the highest possible value.

Ex : `{ "lowestValue": MinKey(), "highestValue": MaxKey() }`

Useful MongoDB Shell Commands

To report the name of the current database:

`db`

Example Output:

```
test
```

To display the list of all databases:

```
show dbs
```

Example Output:

```
admin (empty)
```

```
local 0.078GB
```

```
myDB1 0.078GB
```

To switch to a new database (e.g., myDB1):

```
use myDB1
```

Output:

```
switched to db myDB1
```

To display the list of collections (tables) in the current database:

```
show collections
```

Example Output:

```
system.indexes
```

```
system.js
```

To display the current version of the MongoDB server:

```
db.version()
```

Example Output:

```
2.6.1
```

Consider a table “Students” with the following columns:

1. StudRoll No

2. StudName

3. Grade

4. Hobbies

5. DOJ

Before we get into the details of CRUD operations in MongoDB, let us look at how the statements are written in RDBMS and MongoDB.

	RDBMS	MongoDB
Insert	Insert into Students (StudRollNo, StudName, Grade, Hobbies, DOJ) Values ('S101', 'Simon David', 'VII', 'Net Surfing', '10-Oct-2012')	db.Students.insert({_id:1, StudRollNo: 'S101', StudName: 'Simon David', Grade: 'VII', Hobbies: 'Net Surfing', DOJ: '10-Oct-2012'})
Update	Update Students set Hobbies = 'Ice Hockey' where StudRollNo = 'S101'	db.Students.update({StudRollNo: 'S101'}, {\$set: {Hobbies: 'Ice Hockey'}})
	Update Students Set Hobbies = 'Ice Hockey'	db.Students.update({}, {\$set: {Hobbies: 'Ice Hockey'}}, {multi:true})
Delete	Delete from Students where StudRollNo = 'S101'	db.Students.remove ({StudRollNo : 'S101'})
	Delete From Students	db.Students.remove({})
Select	Select * from Students	db.Students.find() db.Students.find().pretty()
	Select * from students where StudRollNo = 'S101'	db.Students.find({StudRollNo: 'S101'})
	Select StudRollNo, StudName, Hobbies from Students	db.Students.find({}, {StudRollNo:1, StudName:1, _id:0})
	Select StudRollNo, StudName, Hobbies from Students where StudRollNo = 'S101'	db.Students.find({StudRollNo: 'S101'}, {StudRollNo : 1, StudName: 1, Hobbies : 1, _id:0})

RDBMS	MongoDB
Select StudRollNo, StudName, Hobbies From Students Where Grade = 'VII' and Hobbies = 'Ice Hockey'	db.Students.find({Grade: 'VII' , Hobbies: 'Ice Hockey'}, {StudRollNo : 1, StudName: 1, Hobbies : 1, _id:0})
Select StudRollNo, StudName, Hobbies From Students Where Grade = 'VII' or Hobbies = 'Ice Hockey'	db.Students.find({ \$or: [{Grade: 'VII' , Hobbies: 'Ice Hockey'}], StudRollNo : 1, StudName: 1, Hobbies : 1, _id:0})
Select * From Students Where StudName like 'S%'	db.Students.find({ StudName: /^S/}).pretty()

3.5 MongoDB Query Language

CRUD (Create, Read Update, and Delete) operations in MongoDB

Create → Creation of data is done using insert() or update() or save() method.

Read → Reading the data is performed using the find() method.

Update → Update to data is accomplished using the update() method with
UPSERT set to false.

Delete → a document is Deleted using the remove() method.

Creating and Dropping Collections

Creating a Collection

Objective: Create a collection named "Person".

Step 1 – View existing collections:

show collections

Example output:

Students

food

system.indexes

system.js

Step 2 – Create the new collection:

```
db.createCollection("Person")
```

Output:

```
{ "ok" : 1 }
```

Outcome – View updated collections:

show collections

Example output after creation:

Person

Students

food

system.indexes

system.js

Dropping a Collection

Objective: Drop the collection named "food".

Step 1 – Check current collections:

show collections

Example output:

Person

Students

food

system.indexes

system.js

Step 2 – Drop the collection:

```
db.food.drop()
```

Output:

true

Outcome – View updated collections:

```
show collections
```

Example output after dropping:

Person

Students

system.indexes

system.js

3.5.1 Insert Method

1. Create Collection and Insert Document

Objective: Create a collection named Students and insert a document.

Check existing collections:

```
show collections
```

Insert first document:

```
db.Students.insert({_id:1, StudName:"Michelle Jacintha", Grade:"VII",  
Hobbies:"Internet Surfing"})
```

Verify insertion:

```
db.Students.find().pretty()
```

2. Insert Another Document

Insert second document:

```
db.Students.insert({_id:2, StudName:"Mabel  
Hobbies:"Baseball"})
```

Verify with pretty():

```
db.Students.find().pretty()
```

```
Mathews", Grade:"VII",
```

3. Conditional Insert/Update with upsert

Objective: Insert Aryan David only if not already in the collection. If present, update his hobbies.

Check current documents:

```
db.Students.find().pretty()
```

Insert using upsert:

```
db.Students.update({_id:3}, {StudName:"Aryan David", Grade:"VII",  
$set:{Hobbies:"Skating"}}, {upsert:true})
```

Confirm insertion:

```
db.Students.find().pretty()
```

4. Update Existing Document

Objective: Update Aryan David's hobbies from "Skating" to "Chess".

Update using upsert (will update if exists, insert otherwise):

```
db.Students.update({_id:3}, {StudName:"Aryan David", Grade:"VII",  
$set:{Hobbies:"Chess"}}, {upsert:true})
```

5. Insert Document Using save()

Objective: Insert Vamsi Bapat without specifying _id.

Save document:

```
db.Students.save({StudName:"Vamsi Bapat", Grade:"VII"})
```

Check final documents:

```
db.Students.find().pretty()
```

3.5.2 save() method

Inserts a new document if no document with the specified _id exists. If the document exists, it replaces the existing one.

Objective:

Insert the document of "Hersch Gibbs" into the Students collection using the update() method with the upsert option.

Step 1: Check existing documents in the "Students" collection

Shows the existing documents with their `_id`, `StudName`, `Grade`, and `Hobbies`.

Step 2: Use update with `upsert: false`

```
db.Students.update(  
  { _id:4, StudName:"Hersch Gibbs", Grade:"VII"},  
  { $set: { Hobbies: "Graffiti" } },  
  { upsert: false }  
);
```

- No document is inserted because a document with `_id:4` doesn't exist.
- Result shows `nUpserted: 0` meaning no document was inserted.

Step 3: Use update with `upsert: true`

```
db.Students.update(  
  { _id:4, StudName:"Hersch Gibbs", Grade:"VII"},  
  { $set: { Hobbies: "Graffiti" } },  
  { upsert: true }  
);
```

- A new document with `_id:4` is inserted.
- Result shows `nUpserted: 1`, meaning one document was inserted.

Step 4: Confirm the new document

`db.Students.find()` now shows the new document of "Hersch Gibbs" with the `Hobbies: "Graffiti"` included.

3.5.3 Add a new field to an existing document-Update Method

Syntax of update method

```
db.students.update(  
  { Age: { $gt: 18 } }, // Update Criteria (which documents to update)  
  { $set: { Status: "A" } }, // Update Action (what to update/set)  
  { multi: true } // Update Option (update multiple documents)  
)
```

Objective:

Add a new field "Location" with value "Newark" to the document with `_id:4` in the "Students" collection.

Input:

Check the document with `_id:4` before updating:

```
db.Students.find({_id:4}).pretty();
```

Output:

```
{
  "_id": 4,
  "Grade": "VII",
  "StudName": "Hersch Gibbs",
  "Hobbies": "Graffiti"
}
```

Act:

Add the new field "Location" with the value "Newark":

```
db.Students.update(
  {_id:4},
  {$set: {Location: "Newark"}}
);
```

Output shows:

```
{
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
}
```

Outcome:

Confirm the new field has been added:

```
db.Students.find({_id:4}).pretty();
```

Output:

```
{
  "_id": 4,
  "Grade": "VII",
  "StudName": "Hersch Gibbs",
  "Hobbies": "Graffiti",
  "Location": "Newark"
}
```

3.5.4 Removing an Existing Field from an Existing Document – Remove

Method

Objective:

To remove the field "Location" with the value "Newark" from a document with `_id: 4` in the Students collection.

Input:

Inspect the current document:

```
db.Students.find({_id:4}).pretty()
```

Output:

```
{
  "_id": 4,
  "Grade": "VII",
  "StudName": "Hersch Gibbs",
  "Hobbies": "Graffiti",
  "Location": "Newark"
}
```

Act:

Execute the update command to remove the "Location" field:

```
db.Students.update({_id:4}, { $unset: { Location: "Newark" } })
```

This uses:

- update to modify the document,
- \$unset to remove the "Location" field (value is ignored, key matters).

Output:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Outcome:

Verify the document again:

```
db.Students.find({_id:4}).pretty()
```

Result:

```
{
  "_id": 4,
  "Grade": "VII",
  "StudName": "Hersch Gibbs",
  "Hobbies": "Graffiti"
}
```

The "Location" field has been successfully removed.

1. Removing a Document with remove() Method

```
db.Students.remove({Age: {$gt: 18}})
```

- Removes all documents in the Students collection where the Age is greater than 18.

2. Removing a Field from a Document with \$unset

```
db.Students.update({_id: 4}, {$unset: {Location: "Newark"}})
```

- Removes the Location field from the document with _id: 4.

3. Updating Documents with \$set

```
db.Students.update({_id: 4}, {$set: {Grade: "X"}})
```

- Updates the Grade field of the document with _id: 4 to "X".

4. Adding New Fields

```
db.Students.update({_id: 4}, {$set: {Sports: "Football"}})
```

- Adds a new field Sports with the value "Football" to the document with _id: 4.

5. Using multi: true to Update Multiple Documents

```
db.Students.update(  
  {Grade: "VII"},  
  {$set: {Sports: "Cricket"}},  
  {multi: true}  
)
```

- Updates all documents where Grade is "VII" by setting Sports to "Cricket".

6. Replacing an Entire Document

```
db.Students.replaceOne(  
  {_id: 4},  
  {  
    _id: 4,  
    Grade:"X",  
    StudName: "Hersch Gibbs",  
    Hobbies: "Graffiti",  
    Sports: "Football"  
  }  
)
```

- Replaces the entire document with _id: 4.

7. Upsert Operation (update + upsert: true)

```
db.Students.update(  
  {_id: 5},  
  {$set: {StudName: "Paul Adams", Grade: "VIII"}},  
  {upsert: true}  
)
```

- If a document with _id: 5 doesn't exist, MongoDB inserts it with the specified fields.

3.5.7 Finding Elements Based on Some Criteria – findOne() Method

Objective:

To retrieve a single document from the "Students" collection where a specific field (e.g., Grade) matches a value.

Command:

```
db.Students.findOne({Grade: "VII"})
```

- This will return only one document (not all matches).
- Equivalent in SQL:

```
SELECT * FROM Students WHERE Grade = 'VII' LIMIT 1;
```

Finding Specific Elements – find() with Projections

Objective:

Retrieve only selected fields from matching documents.

Command:

```
db.Students.find({Grade: "VII"}, {StudName: 1, _id: 0})
```

- This will return only the StudName of students in Grade "VII".
- `_id: 0` hides the `_id` field.
- Equivalent in SQL:

```
SELECT StudName FROM Students WHERE Grade = 'VII';
```

Sorting Results – sort() Method

Objective:

Sort documents in ascending or descending order by a specified field.

Commands:

```
db.Students.find().sort({Grade: 1}) // Ascending
```

```
db.Students.find().sort({Grade: -1}) // Descending
```

- Equivalent in SQL:

```
SELECT * FROM Students ORDER BY Grade ASC;
```

```
SELECT * FROM Students ORDER BY Grade DESC;
```

Limiting Results – limit() Method

Objective:

Retrieve only a certain number of documents.

Command:

```
db.Students.find().limit(3)
```

- Returns the first 3 documents from the collection.
- Equivalent in SQL:

```
SELECT * FROM Students LIMIT 3;
```

3.5.5 Finding Documents based on Search Criteria - Find Method

Objective: To search for documents from the "Students" collection based on certain search criteria. Input: Check the documents in the "Students" collection before proceeding.



```

> db.Students.find({}).pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}

{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}

{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}

{
  "_id" : ObjectId("5464849859ad1ab07d489b7f"),
  "StudName" : "Vamsi Sapat",
  "Grade" : "VI"
}

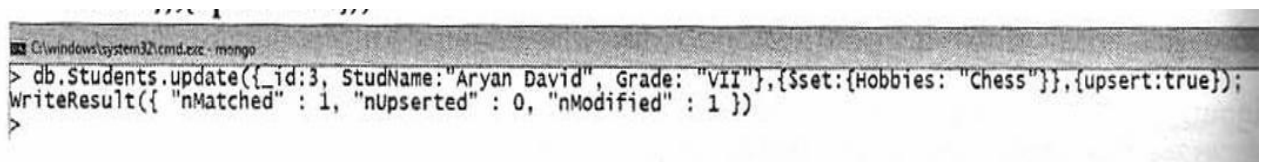
{
  "_id" : 2,
  "StudName" : "Habel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}

```

Act: Find the document wherein the "StudName" has value "Aryan David".

```
db.Students.find({StudName:"Aryan David"});
```

Outcome:



```

> db.Students.update({_id:3, StudName:"Aryan David", Grade: "VII"},{$set:{Hobbies: "Chess"}},{upsert:true});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>

```

RDBMS equivalent:

Select *

From Students

Where StudName like 'Aryan David';

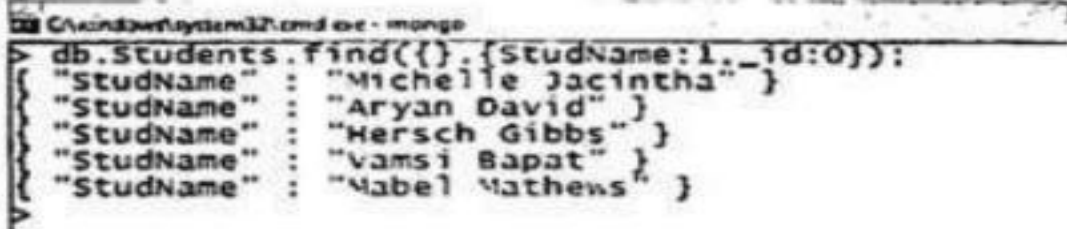
Objective: To display only the StudName from all the documents of the Student's collection. The identifier "_id" should be suppressed and NOT displayed.

Act:

```
db.Students.find({}, {StudName: 1, _id: 0});
```

Outcome:

Outcome:



```

db.Students.find({}, {StudName: 1, _id: 0}):
{"StudName": "Michelle Jacintha"}
{"StudName": "Aryan David"}
{"StudName": "Hersch Gibbs"}
{"StudName": "Vamsi Bapat"}
{"StudName": "Mabel Mathews"}

```

RDBMS equivalent:

Select StudName

From Students;

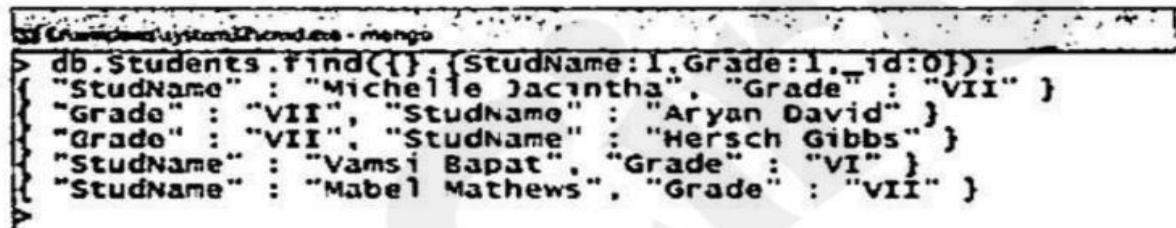
Objective: To display only the StudName and Grade from all the documents of the Students collection. The identifier _id should be suppressed and NOT displayed.

Act:

```
db.Students.find({}, {StudName: 1, Grade: 1, _id: 0});
```

Outcome:

Outcome:



```

db.Students.find({}, {StudName: 1, Grade: 1, _id: 0}):
{"StudName": "Michelle Jacintha", "Grade": "VII"}
{"Grade": "VII", "StudName": "Aryan David"}
{"Grade": "VII", "StudName": "Hersch Gibbs"}
{"StudName": "Vamsi Bapat", "Grade": "VI"}
{"StudName": "Mabel Mathews", "Grade": "VII"}

```

RDBMS equivalent:

Select StudName, Grade

From Students;

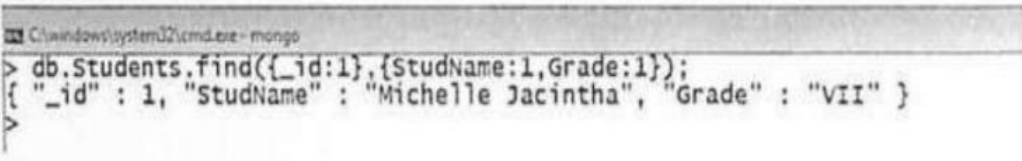
Objective: To display the StudName, Grade as well the identifier, id from the document of the Students collection where the _id column is 1.

Act:

```
db.Students.find({_id:1},{StudName:1,Grade:1});
```

Outcome:

Outcome:



```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({_id:1},{StudName:1,Grade:1});
{'_id' : 1, 'StudName' : 'Michelle Jacintha', 'Grade' : 'VII' }
```

RDBMS equivalent:

Select StudRoll No, StudName, Grade

From Students

Where StudRollNo = '1';

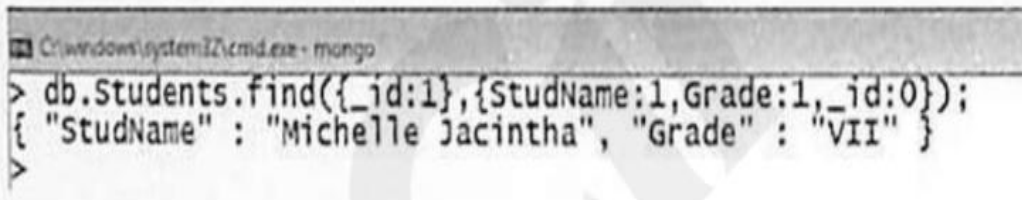
Objective: To display the StudName and Grade from the document of the Students collection where the _id column is 1. The id field should NOT be displayed.

Act:

```
db.Students.find({_id:1}, {StudName:1,Grade:1,_id:0});
```

Outcome:

Outcome:



```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({_id:1},{StudName:1,Grade:1,_id:0});
{'StudName' : 'Michelle Jacintha', 'Grade' : 'VII' }
```

RDBMS equivalent:

Select StudName, Grade

From Students

Where StudRollNo like '1';

Relational operators available to use in the search criteria:

\$eq	→ equal to
\$ne	→ not equal to
\$gte	→ greater than or equal to
\$lte	→ less than or equal to
\$gt	→ greater than
\$lt	→ less than

Objective: To find those documents where the Grade is set to 'VII'.

Act:

```
db.Students.find({Grade: {$eq:'VII'}}).pretty();
```

Outcome:



```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({Grade:{$eq:'VII'}}).pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
```

RDBMS Equivalent:

Select *

From Students

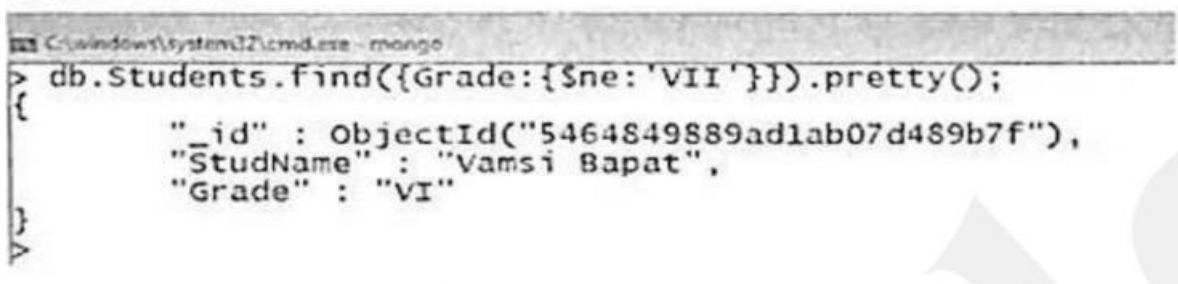
Where Grade like 'VII';

Objective: To find those documents where the Grade is NOT set to 'VII'.

Act:

```
db.Students.find({Grade: {$ne: 'VII'}}).pretty();
```

Outcome:



```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({Grade:{$ne:'VII'}}).pretty();
{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}
```

RDBMS Equivalent:

Select *

From Students

Where Grade <> 'VII';

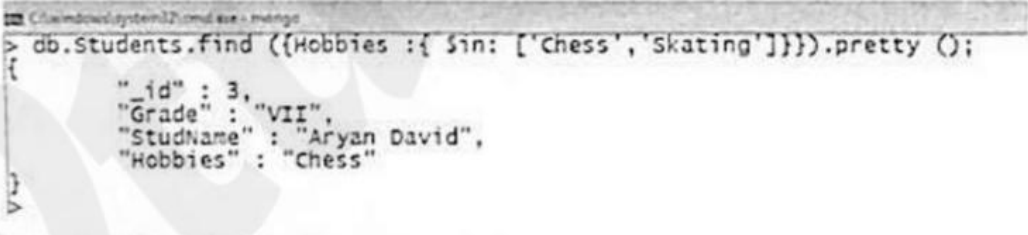
Objective: To find those documents from the Students collection where the Hobbies is set to either 'Chess' or is set to 'Skating'.

Act:

```
db.Students.find ({Hobbies : { $in: ['Chess', 'Skating'] }}).pretty ();
```

Outcome:

Outcome:



```

> db.Students.find ({Hobbies : { $in: ['Chess', 'Skating'] }}).pretty ();
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}

```

RDBMS Equivalent:

Select *

From Students

Where Hobbies in ('Chess', 'Skating');

Objective: To find those documents from the Students collection where the Hobbies is set neither to 'Chess' nor is set to 'Skating'.

Act:

```
db.Students.find ({Hobbies : { $nin: ['Chess', 'Skating'] }}).pretty ();
```

Outcome:

Outcome:



```

> db.Students.find ({Hobbies : { $nin: ['Chess', 'Skating'] }}).pretty ();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : ObjectId("5464849689ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}

```

RDBMS Equivalent:

Select *

From Students

Where Hobbies not in ('Chess', 'Skating');

Objective: To find those documents from the Students collection where the Hobbies is set to 'Graffiti' and the StudName is set to 'Hersch Gibbs' (AND condition).

Act:

```
db.Students.find({Hobbies:'Graffiti', StudName: 'Hersch Gibbs'}).pretty();
```

RDBMS Equivalent:

Select *

From Students

Where Hobbies like 'Graffiti' and StudName like 'Hersch Gibbs';

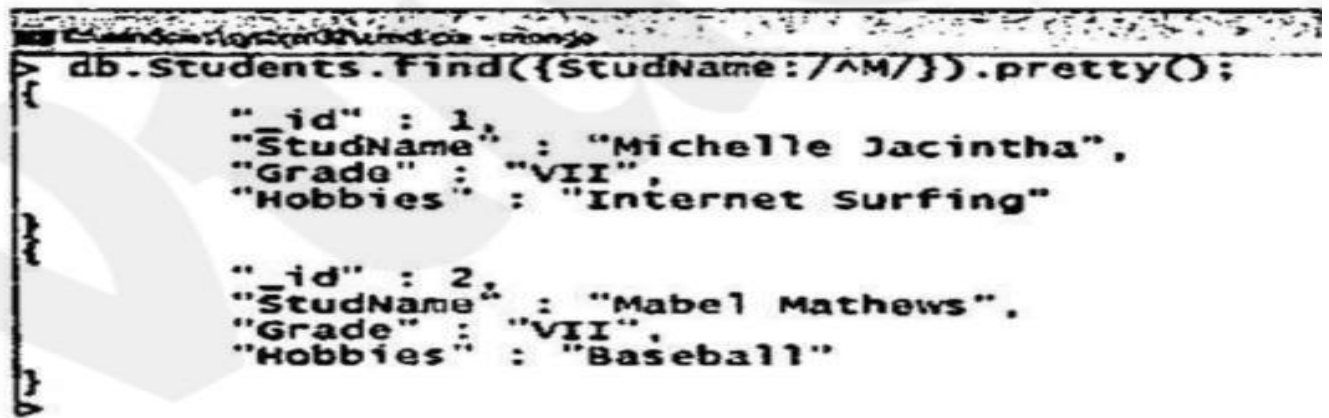
Objective: To find documents from the Students collection where the StudName begins with "M".

Act:

```
db.Students.find({StudName:/^M/}).pretty();
```

Outcome:

Outcome:



```

> db.Students.find({StudName:/^M/}).pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet surfing"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}

```

RDBMS Equivalent:

Select *

From Students

Where StudName like 'M%';

Objective: To find documents from the Students collection where the StudName ends in "s".

Act:

```
db.Students.find({StudName:/s$/}).pretty();
```

Outcome:

RDBMS Equivalent:

Select *

From Students

Where StudName like '%s';

Objective: To find documents from the Students collection where the StudName has an "e" in any position.

Act:

```
db.Students.find({StudName:/e/}).pretty();
```

OR

```
db.Students.find({StudName:/.*e.*/}).pretty();
```

OR

```
db.Students.find({StudName: {$regex:"e"}}).pretty();
```

Outcome:

RDBMS Equivalent:

Select *

From Students

Where StudName like '%e%';

Objective: To find documents from the Students collection where the StudName ends in "a".

Act:

```
db.Students.find({StudName: {$regex:"a$"}}).pretty();
```

RDBMS Equivalent:

Select *

From Students

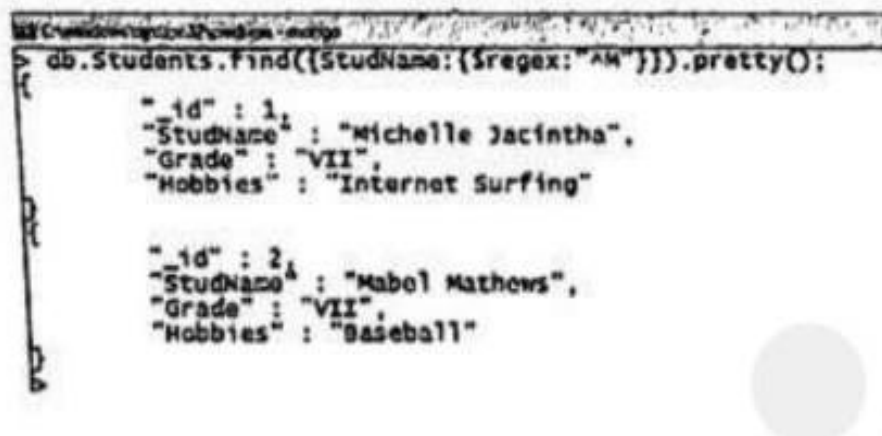
Where StudName like "%a";

Objective: To find documents from the Students collection where the StudName begins with "M".

Act:

```
db.Students.find({StudName:{$regex:"^M"}}).pretty();
```

Outcome:



RDBMS Equivalent:

```
Select *
```

```
From Students
```

```
Where StudName like 'M%';
```

3.5.6 Dealing with NULL Values

Objective:

To add or manage a field (Location) with a NULL value in documents of the Students collection.

- NULL indicates a missing or unknown value.
- This is useful when we don't know the value at the moment but may update it later.

Step 1: Viewing Existing Documents

To view specific documents before updating:

```
db.Students.find({$or: [{_id: 3}, {_id: 4}]})
```

Step 2: Adding NULL Values

To insert a NULL value in the "Location" field:

```
db.Students.update({_id: 3}, {$set: {Location: null}});
```

```
db.Students.update({_id: 4}, {$set: {Location: null}});
```

RDBMS Equivalent:

```
UPDATE Students SET Location = NULL WHERE StudRollNo IN (3, 4);
```

Step 3: Searching for NULL Values

To find documents where Location is NULL or does not exist:

```
db.Students.find({Location: {$eq: null}});
```

RDBMS Equivalent:

```
SELECT * FROM Students WHERE Location IS NULL;
```

Step 4: Removing Fields with NULL Values

To remove the Location field where it's NULL:

```
db.Students.update({_id: 3}, {$unset: {Location: null}});
```

```
db.Students.update({_id: 4}, {$unset: {Location: null}});
```

Step 5: Confirming the Change

To verify that the fields have been removed:

```
db.Students.find()
```

- NULL values can be assigned using \$set.
- NULL fields can be removed using \$unset.
- Documents with NULL or missing fields can be queried with \$eq: null.

3.5.7 Count, Limit, Sort, and Skip

Objective: To find the number of documents in the Students collection.

Act:

```
db.Students.count()
```

Objective: To find the number of documents in the Students collection wherein the Grade is VII.

Act:

```
db.Students.count({Grade:"VII"});
```

Objective: To retrieve the first 3 documents from the Students collection wherein the Grade is VII.

Act:

```
db.Students.find({Grade:"VII"}).limit(3).pretty();
```

Outcome:

```

> db.Students.find({Grade:"VII"}).limit(3).pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet surfing"
}
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}

```

RDBMS Equivalent:

Select *

From Students

Where Grade like "VII" and rownum <4;

Objective: To sort the documents from the Students collection in the ascending order of StudName.

Act:

```
db.Students.find().sort({StudName:1}).pretty();
```

Outcome:

```

> db.Students.find().sort({StudName:1}).pretty();
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet surfing"
}
{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}

```

RDBMS Equivalent:

Select *

From Students

Order by StudName asc;

Objective: To sort the documents from the Students collection in the descending order of StudName.

Act:

```
db.Students.find().sort({StudName:-1}).pretty();
```

Outcome:



```

C:\mongodb\bin>mongo
MongoDB shell version: 2.6.10
> use test
switched to db test
> db.Students.find().sort({StudName:-1}).pretty();
{
  "_id" : ObjectId("5464849589ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}

{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}

{
  "_id" : 2,
  "StudName" : "Habel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}

{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}

{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
  
```

RDBMS Equivalent:

Select *

From Students

Order by StudName desc;

Objective: To sort the documents from the Students collection first on Grade in ascending order and then on Hobbies in descending order.

Act:

```
db.Students.find().sort((Grade:1, Hobbies:-1)).pretty();
```

Outcome:



```

C:\windows\system32\cmd.exe - mongo
> db.Students.find().sort({Grade:1, Hobbies:-1}).pretty();
{
  "_id" : ObjectId("5464849869ad1ab07d449b7f"),
  "StudName" : "Vansh Bapat",
  "Grade" : "VI",
  "Hobbies" : "Internet Surfing"
},
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
},
{
  "_id" : 4,
  "StudName" : "Hersch Gibbs",
  "Grade" : "VII",
  "Hobbies" : "Graffiti"
},
{
  "_id" : 3,
  "StudName" : "Aryan David",
  "Grade" : "VII",
  "Hobbies" : "Chess"
},
{
  "_id" : 2,
  "StudName" : "Nabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}

```

3.5.8 Arrays

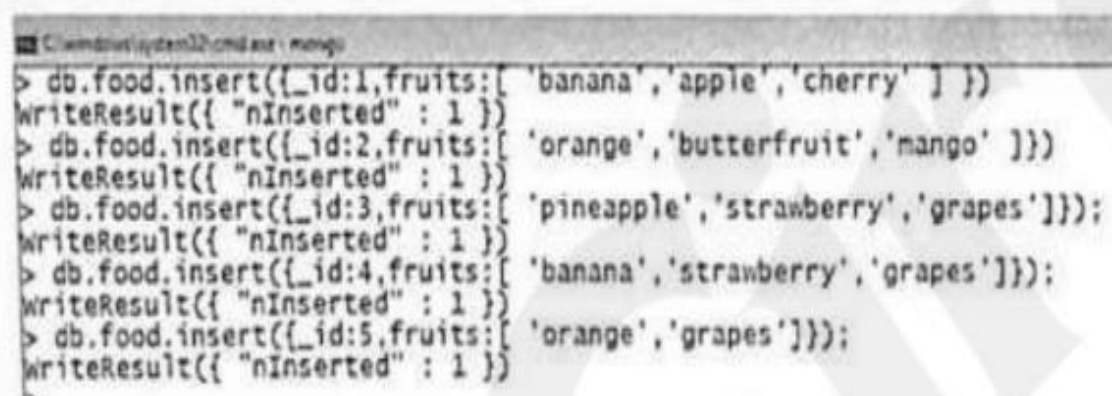
Objective: To create a collection by the name "food" and then insert documents into the "food" collection. Each document should have a "fruits" array.

Act:

```

db.food.insert({_id:1,fruits:[ 'banana','apple', 'cherry' ] })
db.food.insert({_id:2,fruits:[ 'orange','butterfruit','mango' ]})
db.food.insert({_id:3,fruits:[ 'pineapple','strawberry','grapes']});
db.food.insert({_id:4,fruits:[ 'banana','strawberry','grapes']});
db.food.insert({_id:5,fruits:[ 'orange','grapes']});

```



```

C:\windows\system32\cmd.exe - mongo
> db.food.insert({_id:1,fruits:[ 'banana','apple','cherry' ] })
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:2,fruits:[ 'orange','butterfruit','mango' ]})
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:3,fruits:[ 'pineapple','strawberry','grapes']});
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:4,fruits:[ 'banana','strawberry','grapes']});
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:5,fruits:[ 'orange','grapes']});
WriteResult({ "nInserted" : 1 })

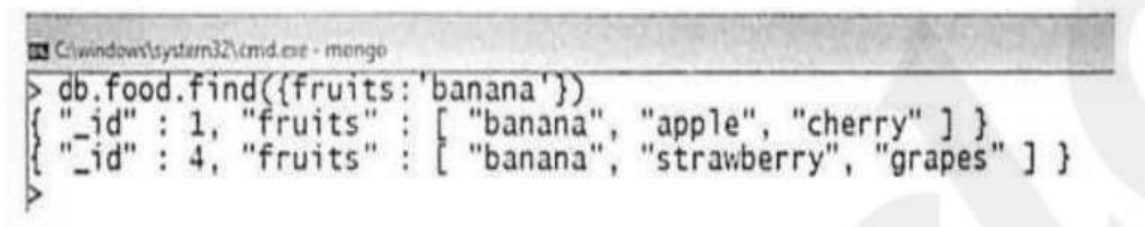
```

Objective: To find those documents from the "food" collection which has the "fruits" array having "banana", as an element.

Act:

```
db.food.find({fruits:'banana'})
```

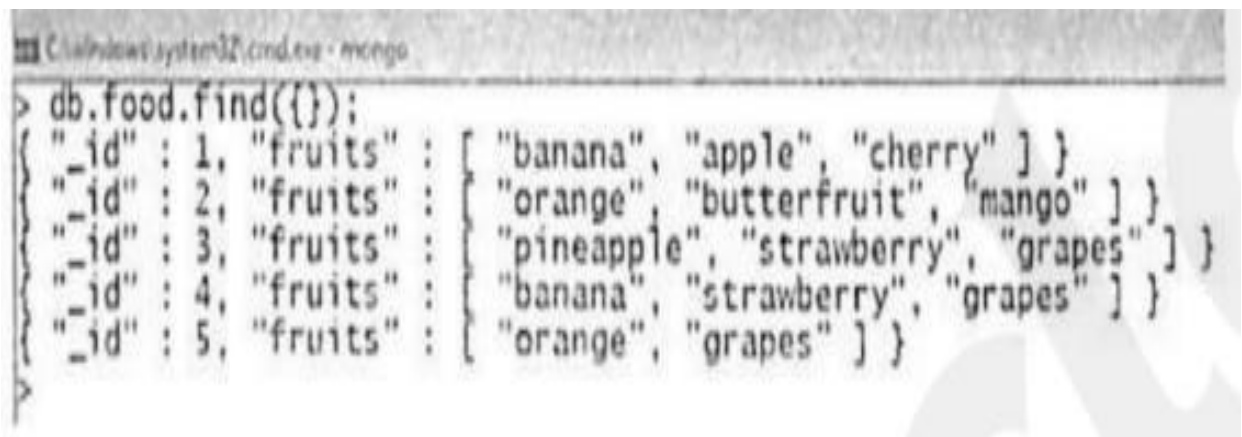
Outcome:



```
C:\windows\system32\cmd.exe - mongo
> db.food.find({fruits:'banana'})
{ "_id" : 1, "fruits" : [ "banana", "apple", "cherry" ] }
{ "_id" : 4, "fruits" : [ "banana", "strawberry", "grapes" ] }
```

3.5.8.1 Update on the Array

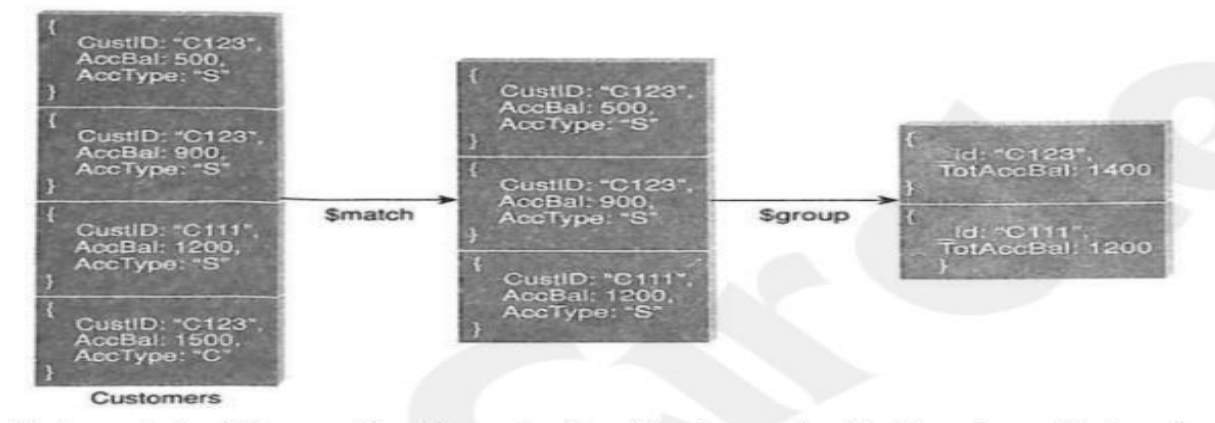
Before we begin the update operations on the "fruits" array of the documents of "food" collection, let us take a look at the documents that we have in the "food" collection:



```
C:\windows\system32\cmd.exe - mongo
> db.food.find({});
{ "_id" : 1, "fruits" : [ "banana", "apple", "cherry" ] }
{ "_id" : 2, "fruits" : [ "orange", "butterfruit", "mango" ] }
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "banana", "strawberry", "grapes" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
```

3.5.9 Aggregate Function

Objective: Consider the collection "Customers" as given below. It has four documents. We would like to filter out those documents where the "AccType" has a value other than "S". After the filter, we should be left with three documents where the "AccType": "S". It is then required to group the documents on the basis of CustID and sum up the "AccBal" for each unique "CustID". This is similar to the output received with group by clause in RDBMS. Once the groups have been formed [as per the example below, there will be only two groups: (a) "CustID" : "C123" and (b) "CustID" : "C111"], filter and display that group where the "TotAccBal" column has a value greater than 1200.



Let us start off by creating the collection "Customers" with the above displayed four documents:

```
db.Customers.insert([ {CustID:"C123",AccBal:500,AccType:"S"},
  {CustID:"C123", AccBal: 900, AccType:"S"},
  {CustID:"C111", AccBal: 1200, AccType:"S"},
  {CustID:"C123", AccBal: 1500, AccType:"C"}]);
```

To confirm the presence of four documents in the "Customers" collection, use the below syntax:

```
db.Customers.find().pretty();
```

To group on "CustID" and compute the sum of "AccBal", use the below syntax:

```
db.Customers.aggregate({$group: {_id:"$CustID",TotAccBal:{$sum:"$AccBal" }}});
```

In order to first filter on "AccType:S" and then group it on "CustID" and then compute the sum of "AccBal", use the below syntax:

```
db.Customers.aggregate( { $match: {AccType: "S" } },
  {$group: { _id: "$CustID",TotAccBal: { $sum : "$AccBal" } } });
```

In order to first filter on "AccType:S" and then group it on "CustID" and then to compute the sum of "AccBal" and then filter those documents wherein the "TotAccBal" is greater than 1200, use the below syntax:

```
db.Customers.aggregate( { $match : {AccType : "S" } },
  {$group: { _id: "$CustID",TotAccBal: { $sum: "$AccBal" } } }, { $match: {TotAccBal : { $gt: 1200 } } });
```

To group on "CustID" and compute the average of the "AccBal" for each group:

```
db.Customers.aggregate({ $group: { _id: "$CustID", TotAccBal : { $avg: "$AccBal" } } });
```

To group on "CustID" and determine the maximum "AccBal" for each group:

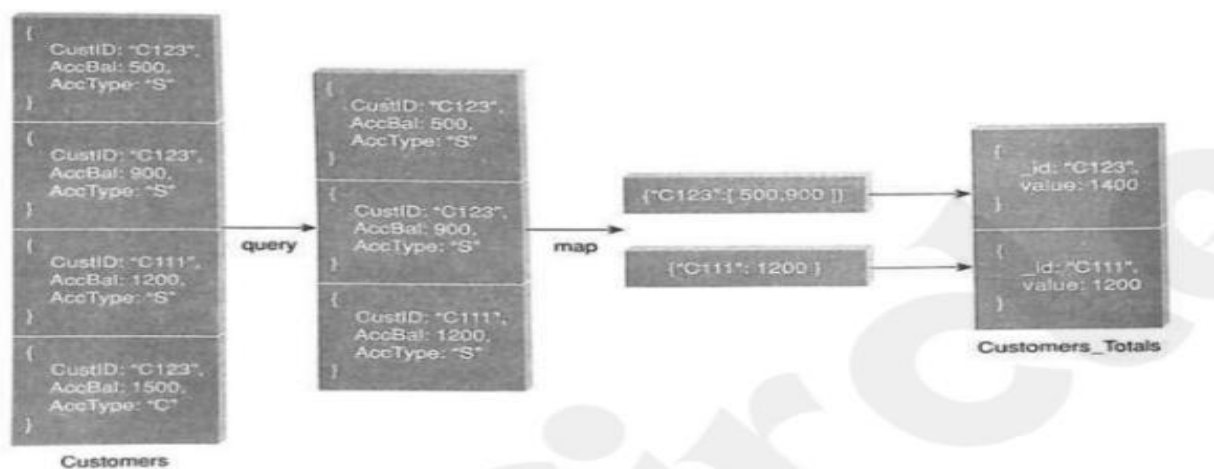
```
db.Customers.aggregate({ $group: { _id: "$CustID", TotAccBal: { $max : "$AccBal" } } });
```

To group on "CustID" and determine the minimum "AccBal" for each group:

```
db.Customers.aggregate({ $group: { _id: "$CustID", TotAccBal: { $min : "$AccBal" } } });
```

3.5.10 MapReduce Function

Objective: Consider the collection "Customers" below. There are four documents. Run a query to filter out those documents where the key "AccType" has a value other than "S". Then for each unique CustID, prepare a list of AccBal values. For example, for CustID: "C123", the AccBals are 500,900. This task will be assigned to the mapper function. The output from the mapper function serves as the input to the reducer function. The reducer function then aggregates the AccBal for each CustID. For example, for CustID: "C123", the value is 1400, etc.



Given below is the syntax that we will use to accomplish the objective.

```
db.Customers.mapReduce (
  map → function() { emit (this.CustID, this.AccBal); },
  reduce → function(key, values) { return Array.sum (values); },
  {
    query → query: { AccType: "S" },
    output → out: "Customer_Totals"
  }
)
```

Map Function

```
var map=function(){
```

```
emit (this. CustID, this.AccBal );}
```

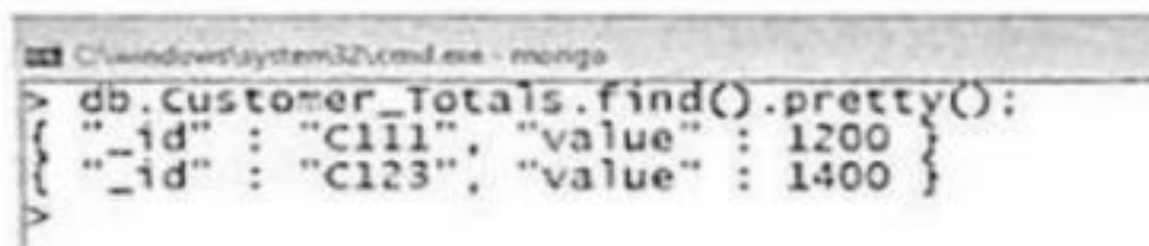
Reduce Function

```
var reduce = function (key, values) { return Array.sum(values); }
```

To execute the query

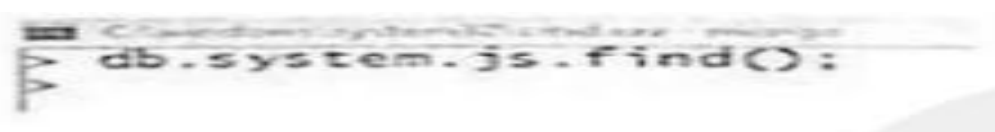
```
db.Customers.mapReduce(map,reduce,{out:"Customer_Totals",query:{AccType:"S"}});
```

The output as archived in "Customer_Totals" collection:



3.5.11 Java Script Programming

Objective: To compute the factorial of a given positive number. The user is required to create a function by the name "factorial" and insert it into the "system.js" collection. Before we proceed, a quick check on what is contained in the "system.js" collection:



Act:

```
db.system.js.insert({_id:"factorial",
value:function(n)
{
if (n==1)
return 1;
else
return n* factorial(n-1);
}
});
```

To execute the function "factorial", use the eval() method.

```
db.eval("factorial(3)");
```

```
db.eval("factorial(5)");
```

```
db.eval("factorial(1)");
```

3.5.12 Cursors in MongoDB

Objective: To create a collection named "alphabets" and insert 26 documents into it. Each document contains:

- `_id`: Numeric values from 1 to 26.
- `alphabet`: Alphabet letters from 'a' to 'z'.

Insertion Syntax:

Example insert statement:

```
db.alphabets.insert({_id:1, alphabet:"a"});
```

We must perform 26 such insertions to cover all lowercase alphabets.

Using Cursor to Iterate:

Using find():

```
var cursor = db.alphabets.find();
```

This retrieves all documents in the collection.

- If find() is not assigned to a variable, MongoDB shell automatically prints only the first 20 documents, followed by "Type "it" for more". To print the remaining, type: it

Manual Cursor Iteration Methods:

Method 1: hasNext()

Returns a Boolean indicating if more documents exist in the cursor.

Usage:

```
while (cursor.hasNext()) {  
  printjson(cursor.next());  
}
```

Method 2: next()

- Retrieves the next document in the cursor.

Using forEach Loop:

Alternative to hasNext():

```
db.alphabets.find().forEach(function (myrec) {  
  print("Alphabet is: " + myrec.alphabet);  
});
```

Verification:

To confirm the presence of all 26 documents:

```
db.alphabets.find()
```

3.5.13 Indexes

Sample Data:

Collection:books

Contains 5 documents with fields like:

- `_id`, Category, Bookname, Author, Qty, Price, Pages

Example Categories:

- Machine Learning
- Web Mining
- Python
- Visualization

Creating an Index

To create an index on the Category field in the books collection:

```
db.books.ensureIndex({"Category": 1});
```

Checking Index Status

1. To check index stats:

```
db.books.stats();
```

Shows count, storage size, index count, index sizes, etc.

Example:

```
"indexes" : 2,  
"indexSizes" : {  
  "_id_" : 8176,
```

```
"Category_1" : 8176
```

```
}
```

2. To list all indexes:

```
db.books.getIndexes();
```

- Shows keys and names of all indexes.

Using Index in Queries

To force use of a specific index:

```
db.books.find({"Category": "Web Mining"}).hint({"Category": 1});
```

To understand how indexes are used:

```
db.books.find({"Category": "Web Mining"}).hint({"Category":1}).explain();
```

- cursor: "BtreeCursor Category_1"
- indexOnly: false (unless it's a covered index)
- nscanned: Number of documents scanned
- nscannedObjects: Number of actual documents examined

Covered Index

Covered index = All fields in query are part of the index.

Example:

```
db.books.find({"Category": "Web Mining"}, {"Category": 1, _id: 0})
```

```
.hint({"Category": 1})
```

```
.explain();
```

- indexOnly: true
- _id field is explicitly excluded to allow the index to fully satisfy the query.

To have a covered index, only indexed fields should be returned (projected). MongoDB skips retrieving the full document if the index contains all the queried fields, improving performance.

3.5.14 mongoimport

Purpose:

The mongoimport command is used to import data into MongoDB from:

- CSV (Comma-Separated Values)
- TSV (Tab-Separated Values)

- JSON (JavaScript Object Notation)

Objective:

Import a CSV file named sample.txt located in the D: drive into the MongoDB collection SampleJSON within the test database. Contents of sample.txt:

_id,FName,LName

1,Samuel,Jones

2,Virat,Kumar

3,Raul,"A Simpson"

4,,"Andrew Simon"

Command to Import CSV File:

Run the following command in the command prompt:

```
mongoimport --db test --collection SampleJSON --type csv --headerline --file  
d:\sample.txt
```

- --db test → Target database
- --collection SampleJSON → Target collection
- --type csv → Input file type
- --headerline → Use the first line of the CSV file as field names
- --file → Path to the input file

Successful Output Message:

connected to: 127.0.0.1

imported 4 objects

Verifying the Import in Mongo Shell:

Steps:

1. Start Mongo shell
2. Switch to test database:
3. use test
4. View collections:
5. show collections
6. Query the data:

```
7. db.SampleJSON.find().pretty();
```

Sample Output of JSON Documents:

```
{ "_id": 1, "FName": "Samuel", "LName": "Jones" }
```

```
{ "_id": 2, "FName": "Virat", "LName": "Kumar" }
```

```
{ "_id": 3, "FName": "Raul", "LName": "A Simpson" }
```

```
{ "_id": 4, "FName": "", "LName": "Andrew Simon" }
```

- Quoted values (e.g., "A Simpson") are correctly interpreted.
- Empty fields are stored as empty strings (e.g., FName in record 4).
- --headerline ensures correct mapping of CSV headers to MongoDB fields.

3.5.15 mongoexport

Purpose:

The mongoexport command is used at the command prompt to export MongoDB JSON documents into:

- CSV (Comma-Separated Values),
- TSV (Tab-Separated Values), or
- JSON (JavaScript Object Notation) formats.

Objective:

Export data from the Customers collection in the test database into a CSV file named Output.txt in the D: drive. Sample Data in MongoDB (Customers Collection):

```
{ "_id": ObjectId("..."), "CustID": "C123", "AccBal": 500, "AccType": "S" }
```

```
{ "_id": ObjectId("..."), "CustID": "C123", "AccBal": 900, "AccType": "S" }
```

```
{ "_id": ObjectId("..."), "CustID": "C111", "AccBal": 1200, "AccType": "S" }
```

```
{ "_id": ObjectId("..."), "CustID": "C123", "AccBal": 1500, "AccType": "C" }
```

Steps to Export the Data:

Step 1: Create fields.txt file

This file should contain the field names exactly as they appear in the

MongoDB collection, one per line:

CustID

AccBal

AccType

Important: Field names are case-sensitive. Only one field name should be placed per line.

Step 2: Run the Export Command

```
mongoexport --db test --collection Customers --csv --fieldFile d:\fields.txt --out  
d:\output.txt
```

- --db test → Specifies the database.
- --collection Customers → Target collection.
- --csv → Specifies the output format.
- --fieldFile → Points to the list of fields to include.
- --out → Output file location.

Expected Command Line Output:

```
connected to: 127.0.0.1
```

```
exported 4 records
```

Final Output File (Output.txt in D: Drive):

```
CustID,AccBal,AccType
```

```
"C123",500.0,"S"
```

```
"C123",900.0,"S"
```

```
"C111",1200.0,"S"
```

```
"C123",1500.0,"C"
```

3.5.16 Automatic Generation of Unique Numbers for the `_id` Field

This technique is useful to automatically assign a unique, incrementing ID to each new document inserted into a collection.

Step 1: Initialize the Counter Document

Insert an initial document in a new collection named usercounters:

```
db.usercounters.insert(  
  {  
    _id: "empid",  
    seq: 0  
  }  
)
```

- `_id` is a custom name (e.g., "empid") used to identify the sequence.
- `seq` is initialized to 0.

Step 2: Create a JavaScript Function `getnextseq`

This function will find and increment the sequence value using `findAndModify()`.

```
function getnextseq(name) {  
  var ret = db.usercounters.findAndModify({  
    query: { _id: name },  
    update: { $inc: { seq: 1 } },  
    new: true  
  });  
  return ret.seq;  
}
```

- `findAndModify()` atomically finds the document and increments `seq` by 1.
- `new: true` returns the modified document after the update.
- Returns the incremented `seq` value.

Step 3: Use `getnextseq()` While Inserting New Documents

Use the `getnextseq()` function when inserting into a collection (e.g., `users`) to auto-assign a unique `_id`:

```
db.users.insert(  
  {  
    _id: getnextseq("empid"),  
    Name: "sarah jane"  
  }  
)
```

- The `_id` will now have an auto-incremented value based on the "empid" sequence in `usercounters`.

Benefits:

- Ensures unique and sequential `_id` values.
- Useful in applications needing custom ID schemes (e.g., employee numbers, customer IDs).

MODULE-4

Introduction to Hive: What is Hive, Hive Architecture, Hive data types, Hive file formats, Hive Query Language (HQL), RC File implementation, User Defined Function (UDF). Introduction to Pig: What is Pig, Anatomy of Pig, Pig on Hadoop, Pig Philosophy, Use case for Pig, Pig Latin Overview, Data types in Pig, Running Pig, Execution Modes of Pig, HDFS Commands, Relational Operators, Eval Function, Complex Data Types, Piggy Bank, User Defined Function, Pig Vs Hive. TB1: Ch 9: 9.1-9.6,9.8, Ch 10: 10.1 - 10.15, 10.22

4.1 What is Hive?

Hive is a **data warehousing tool** that runs on top of the Hadoop ecosystem. It is designed to manage and query large volumes of data stored in Hadoop Distributed File System (HDFS). Hive provides a SQL-like language called **HiveQL (HQL)**, which allows users to perform data analysis without writing complex MapReduce programs.

Hive is Suitable For

Hive is mainly suitable for **data warehousing applications** where large datasets need to be analyzed rather than updated frequently. It is optimized for **batch processing** and works efficiently on huge volumes of **immutable data**, meaning data whose structure does not change once it is created. Hive is commonly used to analyze data such as **web logs, application logs, and historical data**, where fast read and analysis are more important than real-time processing.

Hive is used to process structured data in Hadoop.

The three main tasks performed by Apache Hive are:

1. Summarization
2. Querying
3. Analysis

Facebook initially created Hive component to manage their ever-growing volumes of log data. Later Apache software foundation developed it as open- source and it came to be known as Apache Hive.

Hive makes use of the following:

1. HDFS for Storage.
2. MapReduce for execution.
3. Stores metadata/schemas in an RDBMS.

Hive provides HQL (Hive Query Language) or HiveQL which is similar to SQL.

Hive compiles SQL queries into MapReduce jobs and then runs the job in the

Hadoop Cluster. It is designed to support OLAP (Online Analytical Processing).

Hive provides extensive data type functions and formats for data summarization and analysis.

Note:

1. Hive is not RDBMS.
2. It is not designed to support OLTP (Online Transaction Processing).
3. It is not designed for real-time queries.
4. It is not designed to support row-level updates.

4.1.1 History of Hive and recent releases of Hive

4.1.2 Hive Features

1. It is similar to SQL.
2. HQL is easy to code.
3. Hive supports rich data types such as structs, lists and maps.
4. Hive supports SQL filters, group-by and order-by clauses.
5. Custom Types, Custom Functions can be defined.

4.1.3 Hive Integration and Work Flow

Figure shows depicts the flow of log file analysis.

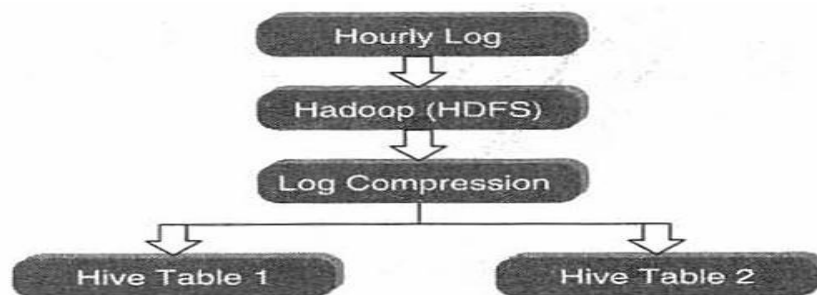


Figure 9.4 Flow of log analysis file.

Hourly Log Data can be stored directly into HDFS and then data cleansing is performed on the log file. Finally, Hive table(s) can be created to query the log file.

4.1.4 Hive Data Units

1. Databases: The namespace for tables.
 2. Tables: Set of records that have similar schema.
 3. Partitions: Logical separations of data based on classification of given information as per specific attributes. Once hive has partitioned the data based on a specified key, it starts to assemble the records into specific folders as and when the records are inserted.
 4. Buckets (or Clusters): Similar to partitions but uses hash function to segregate data and determines the cluster or bucket into which the record should be placed.
- Partitioning tables changes how Hive structures the data storage.
 - Hive will create subdirectories reflecting the partitioning structure.
 - Although partitioning helps in enhancing performance and is recommended, having too many partitions may prove detrimental for few queries.
 - Bucketing is another technique of managing large datasets. If we partition the dataset based on customer_ID, we would end up with far too many partitions. Instead, if we bucket the customer table and use customer_id as the bucketing column, the value of this column will be hashed by a user-defined number into buckets.

When to Use Partitioning/Bucketing?

Bucketing works well when the field has high cardinality (cardinality is the number of values a column or field can have) and data is evenly distributed among buckets. Partitioning works best when the cardinality of the partitioning field is not too high. Partitioning can be done on multiple fields with an order (Year/Month/ Day) whereas bucketing can be done on only one field. Figure shows how these data units are arranged in a Hive Cluster.

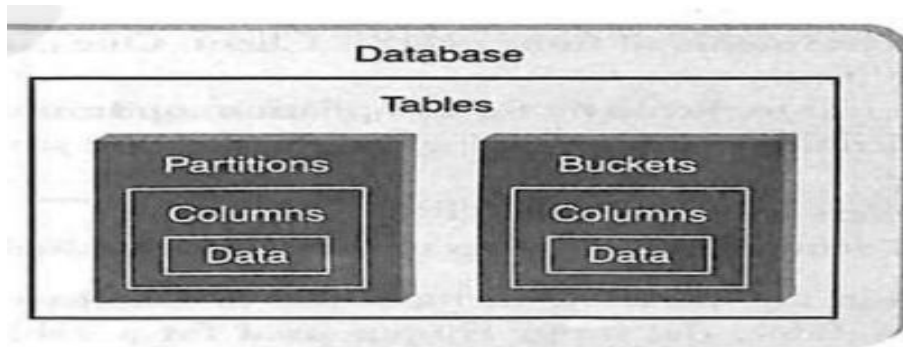


Figure below describes the semblance of Hive structure with database.

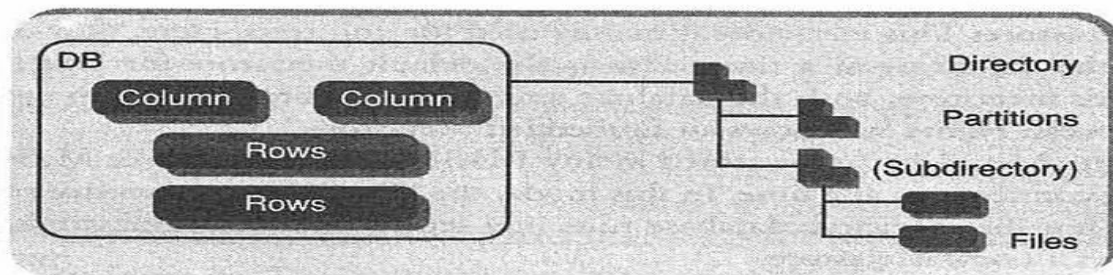


Figure 9.6 Semblance of Hive structure with database.

A database contains several tables. Each table is constituted of rows and columns. In Hive, tables are stored as a folder and partition tables are stored as a sub-directory. Bucketed tables are stored as a file.

4.2 HIVE ARCHITECTURE

The below figure shows Hive Architecture.

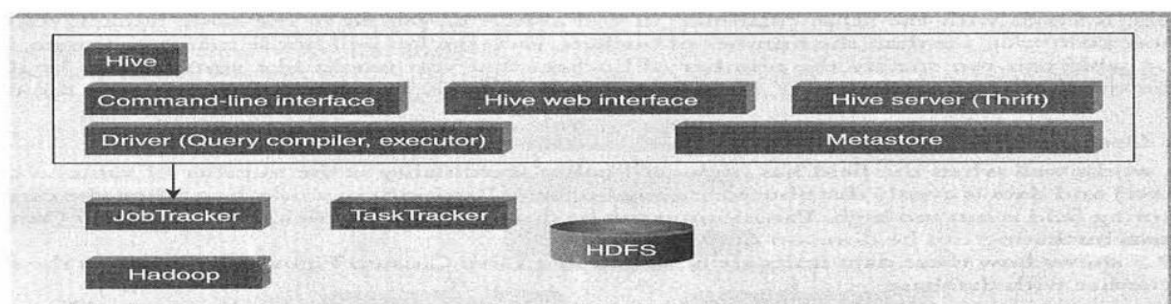


Figure 9.7 Hive architecture.

The various parts are as follows:

1. Hive Command-Line Interface (Hive CLI): The most commonly used interface to interact with Hive.
2. Hive Web Interface: It is a simple Graphic User Interface to interact with Hive and to execute query.
3. Hive Server: This is an optional server. This can be used to submit Hive Jobs from a remote client.
4. JDBC/ODBC: Jobs can be submitted from a JDBC Client. One can write a Java code to connect to Hive and submit jobs on it.

5. Driver: Hive queries are sent to the driver for compilation, optimization and execution.

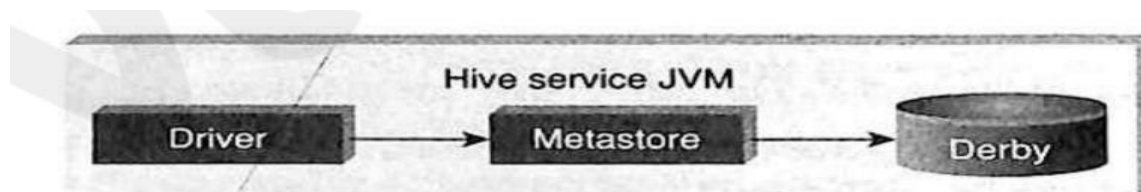
6. Metastore: Hive table definitions and mappings to the data are stored in a Metastore. A Metastore consists of the following:

- Metastore service: Offers interface to the Hive.
- Database: Stores data definitions, mappings to the data and others.

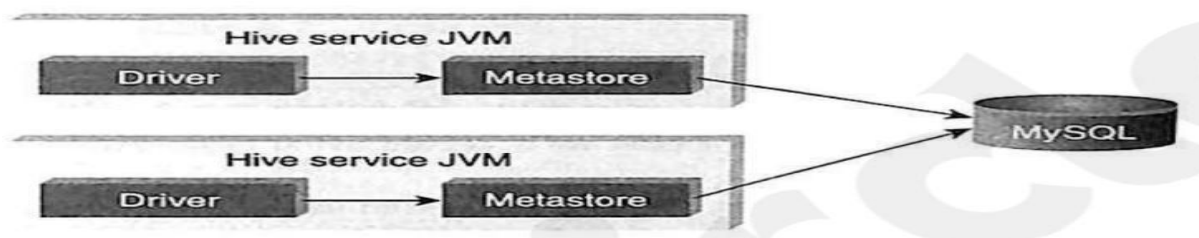
The metadata which is stored in the metastore includes IDs of Database, IDs of Tables, IDs of Indexes, etc., the time of creation of a Table, the Input Format used for a Table, the Output Format used for a Table, etc. The metastore is updated whenever a table is created or deleted from Hive.

There are three kinds of metastore.

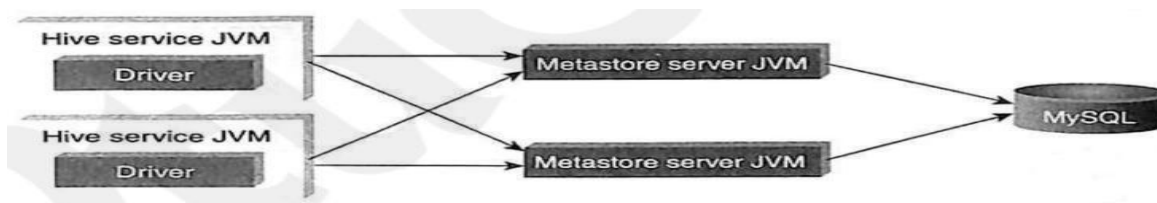
1. Embedded Metastore: This metastore is mainly used for unit tests. Here, only one process is allowed to connect to the metastore at a time. This is used for default setup in Hive. It is Apache Derby Database. In this metastore, both the database and the metastore service run embedded in the main Hive Server process. Figure below shows an Embedded Metastore.



2. Local Metastore: Metadata can be stored in any RDBMS component like MySQL. Local metastore allows multiple connections at a time. In this mode, the Hive metastore service runs in the main Hive Server process, but the metastore database runs in a separate process, and can be on a separate host. Figure below shows a Local Metastore.



3. Remote Metastore: In this, the Hive driver and the metastore interface run on different JVMs (which can run on different machines as well) as in Figure below. This way the database can be fire-walled from the Hive user and also database credentials are completely isolated from the users of Hive.



4.3 HIVE Datatypes

4.3.1 PRIMITIVE DATATYPES

Numeric Data Type	
TINYINT	1-byte signed integer
SMALLINT	2-byte signed integer
INT	4-byte signed integer
BIGINT	8-byte signed integer
FLOAT	4-byte single-precision floating-point
DOUBLE	8-byte double-precision floating-point number
String Types	
STRING	
VARCHAR	Only available starting with Hive 0.12.0
CHAR	Only available starting with Hive 0.13.0
Strings can be expressed in either single quotes (') or double quotes (")	
Miscellaneous Types	
BOOLEAN	
BINARY	Only available starting with Hive

4.3.2 COLLECTION DATATYPES

Collection Data Types	
STRUCT	Similar to 'C' struct. Fields are accessed using dot notation. E.g.: struct('John', 'Doe')
MAP	A collection of key-value pairs. Fields are accessed using [] notation. E.g.: map('first', 'John', 'last', 'Doe')
ARRAY	Ordered sequence of same types. Fields are accessed using array index. E.g.: array('John', 'Doe')

4.4 HIVE FILE FORMAT

4.4.1. Text File

The default file format is text file. In this format, each record is a line in the file. In text file, different control characters are used as delimiters. The delimiters are ^A (octal 001, separates all fields), ^B (octal 002, separates the elements in the array or struct), ^C (octal 003, separates key-value pair), and \n. The term field is used when overriding the default delimiter. The supported text files are CSV and TSV. JSON or XML documents too can be specified as text file.

Types of Supported Text File Formats

Even though the default format is a plain text file, several structured formats are also treated as text files:

CSV (Comma-Separated Values)

Fields are separated by commas

Example: Alice,22,Delhi

TSV (Tab-Separated Values)

Fields are separated by a tab (\t)

Example: Bob\t23\tMumbai

JSON (JavaScript Object Notation)

Structured text format often used to store objects and arrays.

Example: {"name": "Charlie", "age": 30, "city": "Chennai"}

XML (eXtensible Markup Language)

Uses tags to structure data

Example: <person><name>Diana</name><age>28</age><city>Pune</city></p erson>

Even though JSON and XML look very different from CSV/TSV, they are still text files because they store data in readable text form.

4.4.2. Sequential File

Sequential files are flat files that store binary key-value pairs. It includes compression support which reduces the CPU, I/O requirement.

4.4.3. RCFile (Record Columnar File)

RCFile stores the data in Column Oriented Manner which ensures that Aggregation operation is not an expensive operation. For example, consider a table which contains four columns as shown in Table below.

C1	C2	C3	C4
11	12	13	14
21	22	23	24
31	32	33	34
41	42	43	44
51	52	53	54

Instead of only partitioning the table horizontally like the row-oriented DBMS (row-store), RCFile partitions this table first horizontally and then vertically to serialize the data. Based on the user-specified value, first the table is partitioned into multiple row groups horizontally. Depicted in Table-2, Table-1 is partitioned into two row groups by considering three rows as the size of each row group.

Row Group 1				Row Group 2			
C1	C2	C3	C4	C1	C2	C3	C4
11	12	13	14	41	42	43	44
21	22	23	24	51	52	53	54
31	32	33	34				

Table 9.3 Table in RCFile Format

Row Group 1	Row Group 2
11, 21, 31;	41, 51;
12, 22, 32;	42, 52;
13, 23, 33;	43, 53;
14, 24, 34;	44, 54;

4.5 HIVE QUERY LANGUAGE

Hive query language provides basic SQL like operations. Below are few of the tasks which HQL can do easily.

1. Create and manage tables and partitions.
2. Support various Relational, Arithmetic, and Logical Operators.
3. Evaluate functions.
4. Download the contents of a table to a local directory or result of queries to HDFS directory.

4.5.1. DDL (Data Definition Language) Statements

These statements are used to build and modify the tables and other objects in the database. The DDL commands are as follows:

1. Create/Drop/Alter Database
2. Create/Drop/Truncate Table
3. Alter Table/Partition/Column
4. Create/Drop/Alter View
5. Create/Drop/Alter Index
6. Show
7. Describe

4.5.2. DML (Data Manipulation Language) Statements

These statements are used to retrieve, store, modify, delete, and update data in database. The DML commands are as follows:

1. Loading files into table.
2. Inserting data into Hive Tables from queries.

Note: Hive 0.14 supports update, delete, and transaction operations.

4.5.3. Starting Hive Shell

Steps to Start Hive Shell:

1. Navigate to the installation path of Hive.
2. Open the terminal and type the command: `hive`
3. When executed, the terminal initializes Hive and you might see log messages related to SLF4J bindings and configurations.
4. After successful initialization, the `hive>` prompt appears, meaning Hive Shell is ready to accept commands. Understanding the Log Output:

SLF4J Binding Messages:

These messages indicate that Hive is loading its logging configuration files and choosing the appropriate logging framework. If multiple bindings are found, SLF4J selects one to use for logging output.

4.5.4. Database

A Database in Hive is a logical container used to organize tables. It helps manage and group related data sets within the Hive ecosystem.

Objective:

To create a database named STUDENTS with a comment for clarity and custom properties for identification.

Command (Act):

```
CREATE DATABASE IF NOT EXISTS STUDENTS
```

```
COMMENT 'STUDENT Details'
```

```
WITH DBPROPERTIES ('creator' = 'JOHN');
```

IF NOT EXISTS: Prevents errors if the database already exists.

COMMENT: Describes the purpose of the database.

WITH DBPROPERTIES: Allows setting key-value properties for the database (metadata).

Outcome:

Hive creates the database STUDENTS if it doesn't already exist.

Displays:

```
OK
```

```
Time taken: 0.536 seconds
```

hive>

The hive> prompt returns, ready for the next command.

Explanation of the syntax:

IF NOT EXIST: It is an optional clause. The create database statement with “IF Not EXISTS” clause creates a database if it does not exist. However, if the database already exists then it will notify the user that a database with the same name already exists and will not show any error message.

COMMENT: This is to provide short description about the database.

WITH DBPROPERTIES: It is an optional clause. It is used to specify any properties of database in the form of (key, value) separated pairs. In the above example, "Creator" is the "Key" and "JOHN" is the value. We can use "SCHEMA" in place of "DATABASE" in this command.

Note: We have not specified the location where the Hive database will be created. By default all the Hive databases will be created under default warehouse directory (set by the property `hive.metastore.warehouse.dir`) as `/user/hive/warehouse/database_name.db`. But if we want to specify our own location, then the **LOCATION** clause can be specified. This clause is optional.

Objective:

To display a list of all databases available in Hive.

Command (Act):

SHOW DATABASES;

Outcome:

Lists all databases present in the Hive metastore.

Example output:

students

Time taken: 0.082 seconds, Fetched: 22 row(s)

The hive> prompt returns, ready for the next command.

Additional Notes:

The **SHOW DATABASES** command lists all databases.

The word **SCHEMAS** can be used instead of **DATABASES**:

SHOW SCHEMAS;

The command supports the **LIKE** clause for filtering results based on patterns:

* matches multiple characters.

? matches a single character.

Examples with LIKE Clause:

SHOW DATABASES LIKE 'Stu*'; Lists all databases whose names start with "Stu".

SHOW DATABASES LIKE 'Stud???'; Lists all databases whose names start with "Stud" followed by exactly 3 characters.

Objective:

To view details about an existing database in Hive.

Command (Act):

DESCRIBE DATABASE STUDENTS;

This command displays the database name, comment (description), and the directory location in HDFS where the database is stored.

Outcome (Sample Output):

students STUDENT Details

hdfs://volgalnx010.ad.infosys.com:9000/user/hive/warehouse/students.

db root USER

Time taken: 0.03 seconds, Fetched: 1 row(s)

students: The name of the database.

STUDENT Details: The comment provided during creation.

hdfs://.../students.db: The HDFS directory path where the database is located.

root USER: Owner information.

This command is useful for confirming database metadata and verifying storage paths.

Objective:

To describe the extended properties of a database.

Command (Act):

DESCRIBE DATABASE EXTENDED STUDENTS;

This command displays additional information such as DB properties (defined using DBPROPERTIES during creation), along with standard database metadata.

Outcome (Sample Output):

students STUDENT Details

hdfs://volgalnx010.ad.infosys.com:9000/user/hive/warehouse/students.

db root USER {creator=JOHN}

Time taken: 0.027 seconds, Fetched: 1 row(s)

Name: students

Comment: STUDENT Details

Location: HDFS path where the database is stored

Owner: root, role: USER

Properties: {creator=JOHN} (user-defined property)

DESCRIBE DATABASE EXTENDED reveals DBPROPERTIES, unlike the basic DESCRIBE DATABASE command.

SCHEMA can be used instead of DATABASE:

DESCRIBE SCHEMA EXTENDED STUDENTS;

DESC is a shorthand for DESCRIBE:

DESC DATABASE EXTENDED STUDENTS;

This command is useful when you need a complete overview of database metadata and user-defined properties.

Objective:

To alter the database properties.

Command (Act):

ALTER DATABASE STUDENTS SET DBPROPERTIES ('edited-by' = 'JAMES');

The ALTER DATABASE command is used to:

1. Add new key-value pairs into DBPROPERTIES.
2. Set the owner or role for the database.

Important: Hive does not allow unsetting (removing) existing DB properties.

Outcome (Sample Output):

hive> ALTER DATABASE STUDENTS SET DBPROPERTIES ('edited-by' = 'JAMES');

OK Time taken: 0.086 seconds

This confirms that the property was added successfully.

Verification:

You can verify the updated properties using the command:

```
DESCRIBE DATABASE EXTENDED STUDENTS;
```

This will now include:

```
{creator=JOHN, edited-by=JAMES}
```

Objective:

To make a database the current working database.

Command (Act):

```
USE STUDENTS;
```

Outcome (Example):

```
hive> USE STUDENTS;
```

```
OK
```

```
Time taken: 0.02 seconds
```

There is no direct command to display the current active database in Hive.

However, to make the command prompt display the current database name as a suffix, use the following setting:

```
set hive.cli.print.current.db=true;
```

This will help you keep track of which database you're working in during a Hive session.

Objective:

To drop a database.

Command (Act):

```
DROP DATABASE STUDENTS;
```

Hive stores databases in the warehouse directory (e.g., /user/hive/warehouse). Deleting a database only works if no tables exist inside it — this is the default RESTRICT mode.

To drop a database along with all its tables:

```
DROP DATABASE STUDENTS CASCADE;
```

The CASCADE option deletes the database along with all contained tables.

RESTRICT (default) prevents deletion if any tables are present.

Complete Syntax:

```
DROP DATABASE [IF EXISTS] database_name [RESTRICT | CASCADE];
```

4.5.5 Tables

Hive provides two kinds of table:

1. Internal or Managed Table
2. External Table

4.5.5.1 Managed Table

1. Hive stores the Managed tables under the warehouse folder under Hive.
2. The complete life cycle of table and data is managed by Hive.
3. When the internal table is dropped, it drops the data as well as the metadata.

When you create a table in Hive, by default it is internal or managed table. If one needs to create an external table, one will have to use the keyword

“EXTERNAL”.

Objective:

To create a managed table named STUDENT.

Command (Act):

```
CREATE TABLE IF NOT EXISTS STUDENT (  
rollno INT,  
name STRING,  
gpa FLOAT  
)
```

```
ROW FORMAT DELIMITED
```

```
FIELDS TERMINATED BY '\t';
```

Explanation:

CREATE TABLE IF NOT EXISTS ensures the table is only created if it doesn't already exist.

This is a managed table – Hive manages both the data and metadata.

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' specifies that fields are separated by tab characters in the input data.

Outcome Example:

Table creation completes successfully.

Execution time may vary (e.g., 0.355 seconds in the sample output).

Objective:

To describe the structure and metadata of the STUDENT table.

Command (Act):

DESCRIBE STUDENT;

Outcome:

Lists the columns in the STUDENT table with their respective data types:

rollno int

name string

gpa float

Note:

Hive stores managed tables in the warehouse directory (e.g.,
/user/hive/warehouse/students.db).

The table data resides inside a subdirectory (like student inside students.db).

To Check if a Table is Managed or External

Command:

DESCRIBE FORMATTED STUDENT;

Details:

Displays complete metadata of the table.

One of the fields will be:

Table Type: MANAGED_TABLE – if Hive manages both data and metadata.

Table Type: EXTERNAL_TABLE – if data is managed outside Hive.

4.5.5.2 External or Self-Managed Table

1. When the table is dropped, it retains the data in the underlying location.

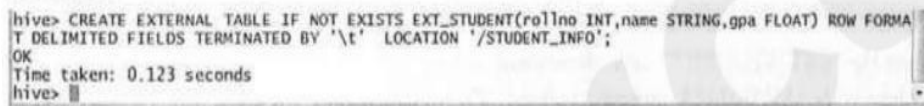
2. External keyword is used to create an external table.
3. Location needs to be specified to store the dataset in that particular location.

Objective: To create external table named 'EXT_STUDENT'.

Act:

```
CREATE EXTERNAL TABLE IF NOT EXISTS EXT_STUDENT (rollno
INT,name STRING,gpa FLOAT) ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t' LOCATION '/STUDENT_INFO';
```

Outcome:



```
hive> CREATE EXTERNAL TABLE IF NOT EXISTS EXT_STUDENT(rollno INT,name STRING,gpa FLOAT) ROW FORM
T DELIMITED FIELDS TERMINATED BY '\t' LOCATION '/STUDENT_INFO';
OK
Time taken: 0.123 seconds
hive>
```

Note: Hive creates the external table in the specified location.

4.5.5.3 Loading Data into Table from File

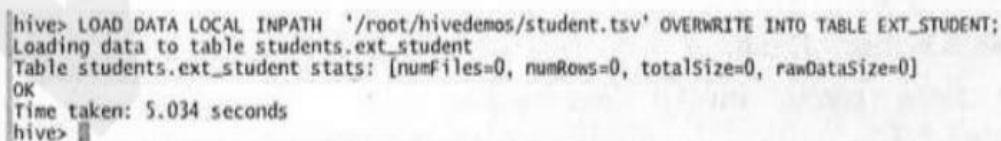
Objective: To load data into the table from file named student.tsv.

Act:

```
LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' OVERWRITE
INTO TABLE EXT_STUDENT;
```

Note: Local keyword is used to load the data from the local file system. In this case, the file is copied. To load the data from HDFS, remove local key word from the statement. In this case, the file is moved from the original location.

Outcome:



```
hive> LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' OVERWRITE INTO TABLE EXT_STUDENT;
Loading data to table students.ext_student
Table students.ext_student stats: [numFiles=0, numRows=0, totalSize=0, rawDataSize=0]
OK
Time taken: 5.034 seconds
hive>
```

4.5.5.4 Collection Data Types

Objective: To work with collection data types.

Input:

1001,John,Smith:Jones,Mark1!45:Mark2!46:Mark3!43

1002,Jack,Smith:Jones,Mark1!46:Mark2!47:Mark3!42

Act:

```
CREATE TABLE STUDENT_INFO(rollno INT,name String, sub
ARRAY<STRING>,marks MAP<STRING,INT>)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY ':'
MAP KEYS TERMINATED BY '!';
LOAD DATA LOCAL INPATH '/root/hivedemos/studentinfo.csv' INTO
STUDENT_INFO;
```

Outcome:

```
hive> CREATE TABLE STUDENT_INFO (rollno INT,name String, sub ARRAY<STRING>,marks MAP<STRING,INT>)
> ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
> COLLECTION ITEMS TERMINATED BY ':'
> MAP KEYS TERMINATED BY '!';
OK
Time taken: 0.112 seconds
hive>

hive> LOAD DATA LOCAL INPATH '/root/hivedemos/studentinfo.csv' INTO TABLE STUDENT_INFO;
Loading data to table students.student_info
Table students.student_info stats: [numFiles=1, totalSize=109]
OK
Time taken: 0.397 seconds
hive>
```

4.5.5.5 Querying Table

Objective: To retrieve the student details from "EXT_STUDENT" table.

Act:

```
SELECT * from EXT_STUDENT;
```

Outcome:

```
hive> select * from EXT_STUDENT;
OK
1001      John      3.0
1002      Jack      4.0
1003      Smith     4.5
1004      Scott     4.2
1005      Joshi     3.5
1006      Alex      4.5
1007      David     4.2
1008      James     4.0
1009      John      3.0
1010      Joshi     3.5
Time taken: 0.054 seconds, Fetched: 10 row(s)
hive>
```

Objective: Querying Collection Data Types.

Act:

```
SELECT * from STUDENT_INFO;
```

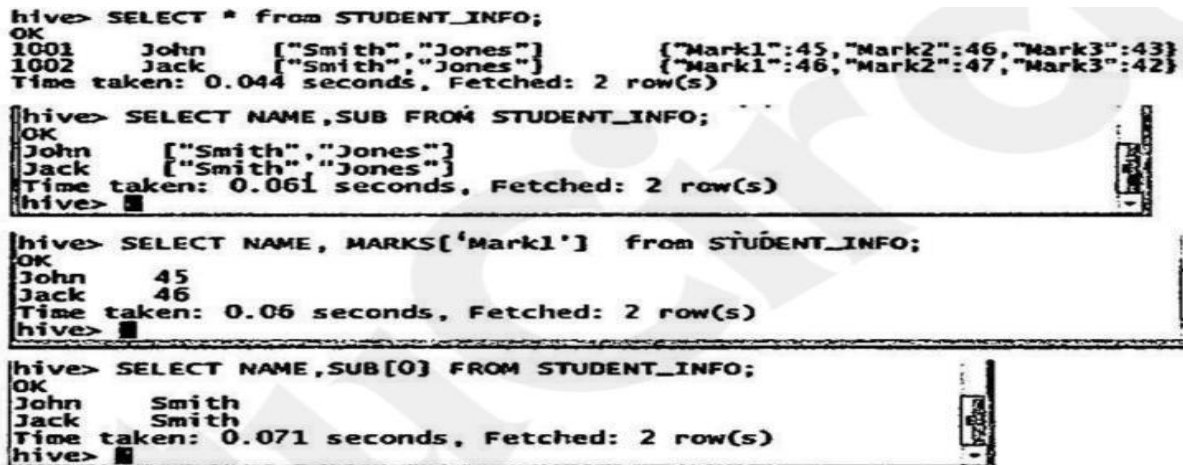
```
SELECT NAME,SUB FROM STUDENT_INFO;
```

```
// To retrieve value of Mark1
```

```
SELECT NAME, MARKS['Mark1'] from STUDENT_INFO;-
```

```
// To retrieve subordinate (array) value
```

```
SELECT NAME,SUB[0] FROM STUDENT_INFO;
```



```
hive> SELECT * from STUDENT_INFO;
OK
1001   John   ["Smith","Jones"]   {"Mark1":45,"Mark2":46,"Mark3":43}
1002   Jack   ["Smith","Jones"]   {"Mark1":46,"Mark2":47,"Mark3":42}
Time taken: 0.044 seconds, Fetched: 2 row(s)

hive> SELECT NAME,SUB FROM STUDENT_INFO;
OK
John   ["Smith","Jones"]
Jack   ["Smith","Jones"]
Time taken: 0.061 seconds, Fetched: 2 row(s)

hive> SELECT NAME, MARKS['Mark1'] from STUDENT_INFO;
OK
John   45
Jack   46
Time taken: 0.06 seconds, Fetched: 2 row(s)

hive> SELECT NAME,SUB[0] FROM STUDENT_INFO;
OK
John   Smith
Jack   Smith
Time taken: 0.071 seconds, Fetched: 2 row(s)
hive>
```

4.5.6 Partitions

In Hive, without partitioning, queries scan the entire dataset, leading to high I/O and slow performance. Partitioning divides data into subdirectories (like by year or region), so Hive reads only relevant parts. This reduces I/O, speeds up queries, and improves MapReduce job efficiency. Partition is of two types:

1. **STATIC PARTITION:** It is upon the user to mention the partition (the segregation unit) where the data from the file is to be loaded.
2. **DYNAMIC PARTITION:** The user is required to simply state the column, basis which the partitioning will take place. Hive will then create partitions basis the unique values in the column on which partition is to be carried out.

Note:

1. **STATIC PARTITIONING** implies that the user controls everything from defining the PARTITION column to loading data into the various partitioned folders.
2. If **STATIC** partition is done over the STATE column and assume by mistake the data for state "B" is placed inside the partition for state "A", our query for data for state "B" is bound to return zilch records. The reason is obvious. A Select fired on **STATIC** partition just takes into consideration the partition name, and does not consider the data held inside the partition.
3. **DYNAMIC PARTITIONING** means Hive will intelligently get the distinct values for partitioned column and segregate data into respective partitions. There is no manual intervention. By default, dynamic partitioning is enabled in Hive. Also by default it is strictly implying that one is required to do

one level of STATIC partitioning before Hive can perform DYNAMIC partitioning inside this STATIC segregation unit. In order to go with full dynamic partitioning, we have to set below property to non-strict in Hive.

```
hive> set hive.exec.dynamic.partition.mode=nonstrict
```

4.5.6.1 Static Partition

Static partitions comprise columns whose values are known at compile time.

Objective: To create static partition based on "gpa" column.

Act:

```
CREATE TABLE IF NOT EXISTS STATIC_PART_STUDENT (rollno INT,
name STRING) PARTITIONED BY (gpa FLOAT) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t';
```

Outcome:

Outcome:

```
hive> CREATE TABLE IF NOT EXISTS STATIC_PART_STUDENT(rollno INT,name STRING) PARTITIONED BY (gpa FLOAT) ROW
FORMAT DELIMITED FIELDS TERMINATED BY '\t';
OK
Time taken: 0.105 seconds
hive>
```

4.5.6.2 Dynamic Partition

Dynamic partition have columns whose values are known only at Execution Time.

Objective: To create dynamic partition on column date.

Act:

```
CREATE TABLE IF NOT EXISTS DYNAMIC_PART_STUDENT(rollno
INT,name STRING) PARTITIONED BY (gpa FLOAT) ROW FORMAT
DELIMITED FIELDS TERMINATED BY '\t';
```

Outcome:

```
hive> CREATE TABLE IF NOT EXISTS DYNAMIC_PART_STUDENT(rollno INT,name STRING) PARTITIONED BY (gpa FLOAT) R
OW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
OK
Time taken: 0.166 seconds
hive>
```

Objective: To load data into a dynamic partition table from table.

Act:

```
SET hive.exec.dynamic.partition = true;
```

```
SET hive.exec.dynamic.partition.mode = nonstrict;
```

Note: The dynamic partition strict mode requires at least one static partition column. To turn this off, set `hive.exec.dynamic.partition.mode=nonstrict`

```
INSERT OVERWRITE TABLE DYNAMIC_PART_STUDENT PARTITION (gpa)
```

```
SELECT rollno,name,gpa from EXT_STUDENT;
```

Outcome:

Outcome:

Contents of directory `/user/hive/warehouse/students.db/dynamic_part_student`

Goto `/user/hive/warehouse/studen` go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
gpa=3.0	dir				2015-02-24 23:16	rw-r--r--	root	supergroup
gpa=3.5	dir				2015-02-24 23:16	rw-r--r--	root	supergroup
gpa=4.0	dir				2015-02-24 23:16	rw-r--r--	root	supergroup
gpa=4.2	dir				2015-02-24 23:16	rw-r--r--	root	supergroup
gpa=4.5	dir				2015-02-24 23:16	rw-r--r--	root	supergroup

Go back to DFS browser

4.5.7 Bucketing

Bucketing is similar to partition. However, there is a subtle difference between partition and bucketing. In a partition, you need to create partition for each unique value of the column. This may lead to situations where you may end up with thousands of partitions. This can be avoided by using Bucketing in which you can limit the number of buckets that will be created. A bucket is a file whereas a partition is a directory.

Objective: To learn the concept of bucket in hive.

Act:

```
CREATE TABLE IF NOT EXISTS STUDENT (rollno INT,name STRING,grade
```

```
FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

```
LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' INTO TABLE
```

```
STUDENT;
```

Set below property to enable bucketing.

```
set hive.enforce.bucketing=true;
```

```
// To create a bucketed table having 3 buckets
```

```
CREATE TABLE IF NOT EXISTS STUDENT_BUCKET (rollno INT,name
```

```
STRING,grade FLOAT)
```

CLUSTERED BY (grade) into 3 buckets;

// Load data to bucketed table

FROM STUDENT

INSERT OVERWRITE TABLE STUDENT_BUCKET

SELECT rollno,name,grade;

// To display content of first bucket

SELECT DISTINCT GRADE FROM STUDENT_BUCKET

TABLESAMPLE(BUCKET 1 OUT OF 3 ON GRADE);

4.5.8 Views

In Hive, view support is available only in version starting from 0.6. Views are purely logical object.

Objective: To create a view table named “STUDENT_VIEW”.

Act:

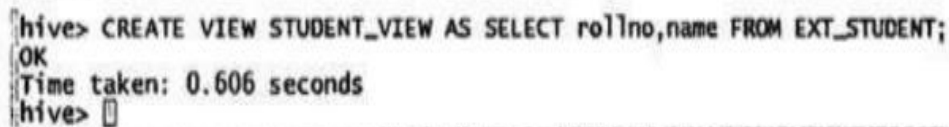
CREATE VIEW STUDENT_VIEW AS SELECT rollno, name FROM

EXT_STUDENT;

Outcome:

Outcome:

Outcome:



```
hive> CREATE VIEW STUDENT_VIEW AS SELECT rollno,name FROM EXT_STUDENT;  
OK  
Time taken: 0.606 seconds  
hive> 
```

Objective: Querying the view "STUDENT_VIEW".

Act:

SELECT * FROM STUDENT_VIEW LIMIT 4;

Outcome:

Outcome:

```
hive> SELECT * FROM STUDENT_VIEW LIMIT 4;
OK
1001      John
1002      Jack
1003      Smith
1004      Scott
Time taken: 0.279 seconds, Fetched: 4 row(s)
```

Objective: To drop the view "STUDENT_VIEW".

Act:

```
DROP VIEW STUDENT_VIEW;
```

Outcome:

Outcome:

```
hive> DROP VIEW STUDENT_VIEW;
OK
Time taken: 0.452 seconds
hive>
```

4.5.9 Sub-Query

In Hive, sub-queries are supported only in the FROM clause (Hive 0.12). You need to specify name for sub-query because every table in a FROM clause has a name. The columns in the sub-query select list should have unique names. The columns in the subquery select list are available to the outer query just like columns of a table.

Objective: Write a sub-query to count occurrence of similar words in the file.

Act:

```
CREATE TABLE docs (line STRING);
```

```
LOAD DATA LOCAL INPATH '/root/hivedemos/lines.txt' OVERWRITE
```

```
INTO TABLE docs;
```

```
CREATE TABLE word_count AS
```

```
SELECT word, count(1) AS count FROM
```

```
(SELECT explode (split (line, ' ')) AS word FROM docs) w
```

```
GROUP BY word
```

```
ORDER BY word;
```

```
SELECT * FROM word_count;
```

4.5.10 Joins

Joins in Hive is similar to the SQL Join.

Objective: To create JOIN between Student and Department tables where we use RollNo from both the tables as the join key.

Act:

```
CREATE TABLE IF NOT EXISTS STUDENT(rollno INT,name STRING,gpa
FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' OVERWRITE
INTO TABLE STUDENT;
CREATE TABLE IF NOT EXISTS DEPARTMENT(rollno INT,deptno int,name
STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
LOAD DATA LOCAL INPATH '/root/hivedemos/department.tsy'
OVERWRITE INTO TABLE DEPARTMENT;
SELECT a.rollno, a.name, a.gpa, b.deptno FROM STUDENT a JOIN
DEPARTMENT b ON a.rollno = b.rollno;
```

4.5.11 Aggregation

Hive supports aggregation functions like avg, count, etc.

Objective: To write the average and count aggregation functions.

Act:

```
SELECT avg(gpa) FROM STUDENT;
SELECT count(*) FROM STUDENT;
```

Outcome:

```
hive> SELECT avg(gpa) FROM STUDENT;
OK
3.8399999961853027
Time taken: 28.639 seconds, Fetched: 1 row(s)
hive>
hive> SELECT count(*) FROM STUDENT;
OK
10
Time taken: 26.218 seconds, Fetched: 1 row(s)
hive>
```

4.5.12 Group By and Having

Data in a column or columns can be grouped on the basis of values contained therein by using "Group By". "Having" clause is used to filter out groups NOT meeting the specified condition.

Objective: To write group by and having function.

Act:

```
SELECT rollno, name,gpa FROM STUDENT GROUP BY rollno,name,gpa  
HAVING gpa > 4.0;
```

4.6 RCFILE IMPLEMENTATION

RCFile (Record Columnar File) is a data placement structure that determines how to store relational tables on computer clusters.

Objective: To work with RCFILE Format.

Act:

```
CREATE TABLE STUDENT_RC( rollno int, name string,gpa float ) STORED  
AS RCFILE; INSERT OVERWRITE table STUDENT_RC SELECT * FROM  
STUDENT; SELECT SUM(gpa) FROM STUDENT_RC;
```

Outcome:

4.8 USER-DEFINED FUNCTION (UDF)

In Hive, you can use custom functions by defining the User-Defined Function (UDF).

Objective: Write a Hive function to convert the values of a field to

uppercase.

Act:

```
package com.example.hive.udf;

import org.apache.hadoop.hive.ql.exec.Description;
import org.apache.hadoop.hive.ql.exec.UDF;

@Description(
    name="SimpleUDFExample")
public final class MyLowerCase extends UDF {
    public String evaluate(final String word) {
        return word.toLowerCase();
    }
}
```

Note: Convert this Java Program into Jar.

ADD JAR /root/hivedemos/UpperCase.jar;

CREATE TEMPORARY FUNCTION touppercase AS

'com.example.hive.udf.MyUpperCase';

SELECT TOUPPERCASE(name) FROM STUDENT;

INTRODUCTION TO PIG

4.9 What is PIG?

Apache Pig is a platform for data analysis. It is an alternative to MapReduce Programming. Pig was developed as a research project at Yahoo.

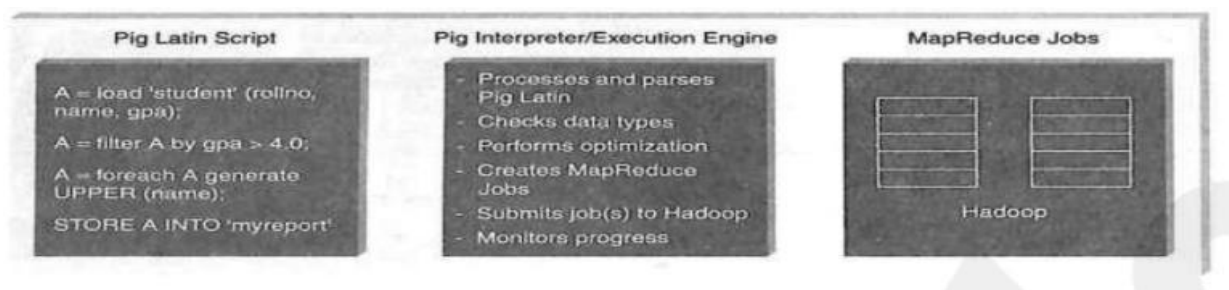
Key Features of Pig

1. It provides an engine for executing data flows (how your data should flow). Pig processes data in parallel on the Hadoop cluster.
2. It provides a language called "Pig Latin" to express data flows.
3. Pig Latin contains operators for many of the traditional data operations such as join, filter, sort, etc.
4. It allows users to develop their own functions (User Defined Functions) for reading, processing, and writing data.

4.10 The Anatomy of PIG

The main components of Pig are as follows:

1. Data flow language (Pig Latin).
2. Interactive shell where you can type Pig Latin statements (Grunt).
3. Pig interpreter and execution engine.



4.11 PIG on Hadoop

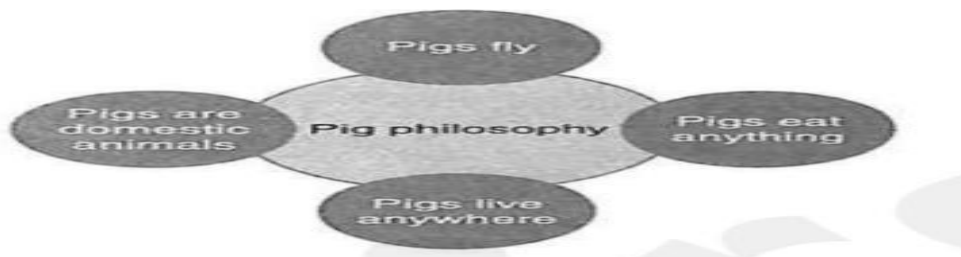
Pig runs on Hadoop. Pig uses both Hadoop Distributed File System and MapReduce Programming. By default, Pig reads input files from HDFS. Pig stores the intermediate data (data produced by MapReduce jobs) and the output in HDFS. However, Pig can also read input from and place output to other sources. Pig supports the following:

1. HDFS commands.
2. UNIX shell commands.
3. Relational operators.
4. Positional parameters.
5. Common mathematical functions.
6. Custom functions.
7. Complex data structures.

4.12 PIG Philosophy

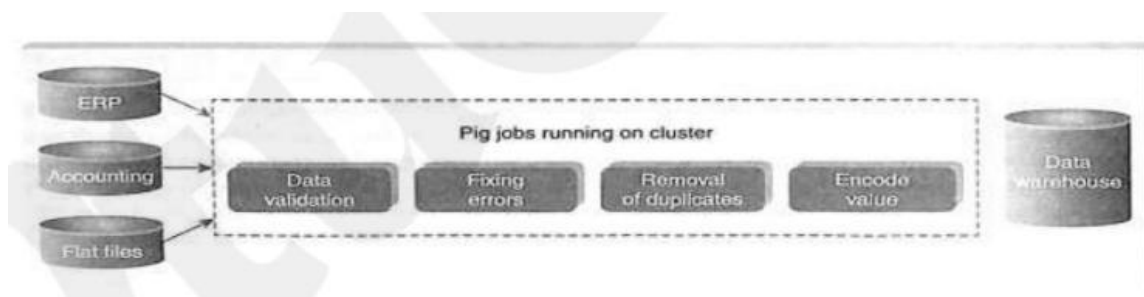
Figure below describes the Pig philosophy.

1. Pigs Eat Anything: Pig can process different kinds of data such as structured and unstructured data.
2. Pigs Live Anywhere: Pig not only processes files in HDFS, it also processes files in other sources such as files in the local file system.
3. Pigs are Domestic Animals: Pig allows you to develop user-defined functions and the same can be included in the script for complex operations.
4. Pigs Fly: Pig processes data quickly.



4.13 USE CASE FOR PIG: ETL PROCESSING

Pig is widely used for "ETL" (Extract, Transform, and Load). Pig can extract data from different sources such as ERP, Accounting, Flat Files, etc. Pig then makes use of various operators to perform transformation on the data and subsequently loads it into the data warehouse. Refer Figure below.



4.14 PIG Latin Overview

4.14.1 Pig Latin Statements

1. Pig Latin statements are basic constructs to process data using Pig.
2. Pig Latin statement is an operator.
3. An operator in Pig Latin takes a relation as input and yields another relation as output.
4. Pig Latin statements include schemas and expressions to process data.
5. Pig Latin statements should end with a semi-colon. Pig Latin Statements are generally ordered as follows:
 1. LOAD statement that reads data from the file system.
 2. Series of statements to perform transformations.
 3. DUMP or STORE to display/store result.

The following is a simple Pig Latin script to load, filter, and store "student" data.

```
A = load 'student' (rollno, name, gpa);
A = filter A by gpa > 4.0;
A = foreach A generate UPPER (name);
STORE A INTO 'myreport'
```

Note: In the above example A is a relation and NOT a variable.

4.14.2 Pig Latin Keywords

Keywords are reserved. It cannot be used to name things.

LOAD: Loads data from a file or data source into a relation.

STORE: Stores a relation to a file or data source.

GROUP: Groups rows in a relation based on certain fields.

JOIN: Combines two relations based on a common field.

FILTER: Selects rows based on a condition.

FOREACH: Applies a transformation to each row in a relation.

DISTINCT: Removes duplicate rows from a relation.

DEFINE: Defines user-defined functions (UDFs).

AS: Assigns an alias to a relation or field.

4.14.3 Pig Latin: Identifiers

1. Identifiers are names assigned to fields or other data structures.
2. It should begin with a letter and should be followed only by letters, numbers, and underscores.

The below table shows the list of valid and invalid identifiers.

4.14.4 Pig Latin: Comments

In Pig Latin two types of comments are supported:

1. Single line comments that begin with "--".
2. Multiline comments that begin with "/* and end with */".

4.14.5 Pig Latin: Case Sensitivity

1. Keywords are not case sensitive such as LOAD, STORE, GROUP, FOREACH, DUMP, etc.
2. Relations and paths are case-sensitive.
3. Function names are case sensitive such as PigStorage, COUNT.

4.14.6 Operators in Pig Latin

Table below describes operators in Pig Latin.

Arithmetic	Comparison	Null	Boolean
+	==	IS NULL	AND
-	!=	IS NOT NULL	OR
*	<		NOT
/	>		
%	<=		
	>=		

4.15 Simple Data Types

Table below describes simple data types supported in Pig. In Pig, fields of unspecified types are considered as an array of bytes which is known as bytearray.

Null: In Pig Latin, NULL denotes a value that is unknown or is non-existent.

4.16 Running PIG

We can run Pig in 2 ways:

1. Interactive Mode
2. Batch Mode

1. Interactive Mode:

Command to start Pig shell:

```
pig
```

- Pig runs in Grunt shell, an interactive environment for executing Pig Latin scripts.
- Warnings show deprecated configuration parameters (e.g., mapred.job.tracker, fs.default.name) and suggest alternatives like mapreduce.jobtracker.address, fs.defaultFS.

Loading and Viewing Data in Pig

Pig Latin Command to load data:

```
A = LOAD '/pigdemo/student.tsv' AS (rollno, name, gpa);
```

Command to display the data:

```
DUMP A;
```

Output (Sample Data):

```
(1001, John, 3.0)
```

```
(1002, Jack, 4.0)
```

```
(1003, Smith, 4.5)
```

```
(1004, Scott, 4.2)
```

```
(1005, Joshi, 3.5)
```

1. The path /pigdemo/student.tsv refers to an HDFS location.
 2. DUMP outputs the dataset to the console.
 3. Each tuple contains: rollno (int), name (string), gpa (float).
2. Batch Mode: We need to create "Pig Script" to run pig in batch mode. Write Pig Latin statements in a file and save it with .pig extension.

4.17 EXECUTION MODES OF PIG

We can execute pig in two modes:

1. Local Mode.
2. MapReduce Mode.

Local Mode

To run pig in local mode, we need to have your files in the local file system.

Syntax:

```
pig -x local filename
```

MapReduce Mode

To run pig in MapReduce mode, we need to have access to a Hadoop Cluster to read/write file. This is the default mode of Pig.

Syntax:

```
pig filename
```

4.18 HDFS COMMANDS in Pig (Grunt Shell)

Objective: Learn how to perform HDFS operations directly from the Pig Grunt shell.

Command Example:

```
grunt> fs -mkdir /piglatindemos;
```

Explanation:

The fs keyword allows HDFS commands within the Grunt shell.

-mkdir creates a new directory in HDFS at the specified path.

Outcome: A directory named /piglatindemos is created in HDFS.

4.19 Relational Operators

1. FILTER Operator in Pig\Purpose:

The FILTER operator is used to select tuples (records) from a relation based on specified conditions.

Objective:

Select students whose GPA is greater than 4.0.

Input:

Relation Student with schema:

(rollno:int, name:chararray, gpa:float)

Commands (Act):

```
A = LOAD '/pigdemo/student.tsv' AS (rollno:int, name:chararray,  
gpa:float);
```

```
B = FILTER A BY gpa > 4.0;
```

```
DUMP B;
```

Output:

```
(1003, Smith, 4.5) (1004, Scott, 4.2)
```

FILTER is used to apply conditions and reduce the dataset.

It helps in extracting meaningful subsets of data for further processing.

2. FOREACH Operator in Pig

Purpose:

The FOREACH operator is used for data transformation based on columns of a relation.

Objective:

Display the names of all students in uppercase.

Input:

Relation Student with schema:

(rollno:int, name:chararray, gpa:float)

Commands (Act):

```
A = LOAD '/pigdemo/student.tsv' AS (rollno:int, name:chararray,  
gpa:float);
```

```
B = FOREACH A GENERATE UPPER(name);
```

```
DUMP B;
```

Output:

(JOHN)

(JACK)

(SMITH)

(SCOTT)

(JOSHI)

FOREACH ... GENERATE is used to apply transformations to each row.

Functions like UPPER() can be used to modify specific fields.

3. GROUP Operator in Pig

Purpose:

The GROUP operator is used to group data based on a column.

Objective:

Group tuples of students based on their GPA.

Input:

Relation Student with schema:

(rollno:int, name:chararray, gpa:float)

Commands (Act):

```
A = LOAD '/pigdemo/student.tsv' AS (rollno:int, name:chararray,gpa:float);
```

B = GROUP A BY gpa;

DUMP B;

Output:

(3.0, {(1001, John, 3.0), (1001, John, 3.0)})

(3.5, {(1005, Joshi, 3.5), (1005, Joshi, 3.5)})

(4.0, {(1008, James, 4.0), (1002, Jack, 4.0)})

(4.2, {(1007, David, 4.2), (1004, Scott, 4.2)})

(4.5, {(1006, Alex, 4.5), (1003, Smith, 4.5)})

GROUP A BY <column> creates groups where all tuples with the same value in the specified column are grouped together.

Each group is represented as (group_key, bag_of_tuples).

4. DISTINCT Operator in Pig

Purpose:

The DISTINCT operator is used to remove duplicate tuples from a relation.

Important Note:

It operates on entire tuples, not on individual fields.

Objective:

Remove duplicate tuples of students.

Input:

Relation Student with schema:

(rollno:int, name:chararray, gpa:float) and data:

(1001, John, 3.0)

(1002, Jack, 4.0)

(1003, Smith, 4.5)

(1004, Scott, 4.2)

(1005, Joshi, 3.5)

(1006, Alex, 4.5)

(1007, David, 4.2)

(1008, James, 4.0)

(1001, John, 3.0) <-- Duplicate

(1005, Joshi, 3.5) <-- Duplicate

Commands (Act):

```
A = LOAD '/pigdemo/student.tsv' AS (rollno:int, name:chararray,  
gpa:float);
```

```
B = DISTINCT A;
```

```
DUMP B;
```

Output (unique tuples only):

(1001, John, 3.0)

(1002, Jack, 4.0)

(1003, Smith, 4.5)

(1004, Scott, 4.2)

(1003, Smith, 4.5)

1. ORDER BY helps sort data globally, unlike GROUP which organizes data into bins.

2. The default is ascending order. To sort in descending order, use:

```
B = ORDER A BY name DESC;
```

7. JOIN Operator in Pig

Purpose:

The JOIN operator is used to combine two or more relations based on values in a common field (typically a key like rollno). Pig always performs an inner join by default.

Objective:

Join two relations—student and department—based on the rollno column.

Input Schemas:

```
Student (rollno:int, name:chararray, gpa:float) Department (rollno:int, deptno:int, deptname:chararray)
```

Pig Latin Script (Act):

```
A = LOAD '/pigdemo/student.tsv' AS (rollno:int, name:chararray gpa:float);
```

```
B = LOAD '/pigdemo/department.tsv' AS (rollno:int, deptno:int, deptname:chararray);
```

```
C = JOIN A BY rollno, B BY rollno;
```

DUMP C;

Output:

(1001, John, 3.0, 1001, 101, B.E.)
(1001, John, 3.0, 1001, 101, B.E.)
(1002, Jack, 4.0, 1002, 102, B.Tech)
(1003, Smith, 4.5, 1003, 103, M.Tech)
(1004, Scott, 4.2, 1004, 104, MCA)
(1005, Joshi, 3.5, 1005, 105, MBA)
(1005, Joshi, 3.5, 1005, 105, MBA)
(1006, Alex, 4.5, 1006, 101, B.E.)
(1007, David, 4.2, 1007, 104, MCA)
(1008, James, 4.0, 1008, 102, B.Tech)

1. The JOIN operator merges rows from two datasets where the join keys match.
2. Duplicate entries in either relation will result in multiple joined records (Cartesian effect for those keys).
3. Default join is inner join — only records with matching keys in both relations are included.

Syntax:

C = JOIN A BY key1, B BY key2;

8. UNION in Pig Latin

Objective:

To merge the contents of two relations student and department.

Input Data:

Student relation:

student.tsv Format: rollno:int, name:chararray, gpa:float

Department relation:

department.tsv Format: rollno:int, deptno:int, deptname:chararray

Pig Latin Script:

A = LOAD '/pigdemo/student.tsv' AS (rollno:int, name:chararray,
gpa:float);

```
B = LOAD '/pigdemo/department.tsv' AS (rollno:int, deptno:int,
deptname:chararray);
A_proj = FOREACH A GENERATE rollno, name AS info1, gpa AS info2;
B_proj = FOREACH B GENERATE rollno, deptname AS info1, deptno AS
info2;
C = UNION A_proj, B_proj
STORE C INTO '/pigdemo/uniondemo';
DUMP C;
```

- UNION in Pig is used to combine rows from two or more relations having the same schema.

- Since student and department have different schemas, we first project both into a common structure with rollno, a chararray field, and a numeric field.

Output Note:

- STORE saves the merged data into HDFS.
- The files inside /pigdemo/uniondemo represent distributed parts of the output.

9. SPLIT

It is used to partition a relation into two or more relations.

Objective: To partition a relation based on the GPAs acquired by the students.

- GPA = 4.0, place it into relation X.
- GPA is < 4.0, place it into relation Y.

Input:

Student (rollno:int,name:chararray,gpa:float)

Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
SPLIT A INTO X IF gpa==4.0, Y IF gpa<=4.0;
DUMP X;
```

Output: Relation X

```
(1002,Jack,4.0)
(1008,James,4.0)
[root@volga1nx010 pigdemos]#
```

Output: Relation Y

```
(1001,John,3.0)
(1002,Jack,4.0)
(1005,Joshi,3.5)
(1008,James,4.0)
(1001,John,3.0)
(1005,Joshi,3.5)
[root@volga1nx010 pigdemos]#
```

10. SAMPLE

It is used to select random sample of data based on the specified sample size.

Objective: To depict the use of SAMPLE.

Input:

Student (rollno:int,name:chararray,gpa:float)

Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
```

```
B = SAMPLE A 0.01;
```

```
DUMP B;
```

4.20 EVAL FUNCTION**4.20.1 AVG**

AVG is used to compute the average of numeric values in a single column bag.

Objective: To calculate the average marks for each student.

Input:

Student (studname:chararray,marks:int)

Act:

```
A = load '/pigdemo/student.csv' USING PigStorage(',') as
```

```
(studname:chararray,marks:int);
```

```
B=GROUP A BY studname;
```

```
C= FOREACH B GENERATE A.studname, AVG(A.marks);
```

```
DUMP C;
```

Output:

```
((Jack), (Jack), (Jack), (Jack)), 39.75)
((John), (John), (John), (John)), 39.0)
[root@volgalnx010 pigdemos]#
```

4.20.2 MAX

MAX is used to compute the maximum of numeric values in a single column bag.

Objective: To calculate the maximum marks for each student.

Input: Student (studname:chararray,marks:int)

Act:

A= load '/pigdemo/student.csv' USING PigStorage (",") as

(studname:chararray, marks:int);

B = GROUP A BY studname;

C = FOREACH B GENERATE A.studname, MAX(A.marks);

DUMP C;

Output:

```
((Jack), (Jack), (Jack), (Jack)), 46)
((John), (John), (John), (John)), 45)
[root@volgalnx010 pigdemos]#
```

4.20.3 COUNT

COUNT is used to count the number of elements in a bag.

Objective: To count the number of tuples in a bag.

Input:

Student (studname:chararray,marks:int)

Act:

A= load '/pigdemo/student.csv' USING PigStorage (",") as

(studname:chararray, marks:int);

B = GROUP A BY studname;

C = FOREACH B GENERATE A.studname, COUNT(A);

DUMP C;

Output:

```
({(Jack),(Jack),(Jack),(Jack)},4)
({(John),(John),(John),(John)},4)
[root@volgalnx010 pigdemos]#
```

10.21 COMPLEX DATA TYPES**10.21.1 TUPLE**

A TUPLE is an ordered collection of fields.

Objective: To use the complex data type “Tuple” to load data.

Input:

(John,12)

(Jack,13)

(James,7)

(Joseph,5)

(Smith,8)

(Scott,12)

Act:

```
A=LOAD /root/pigdemos/studentdata.tsv
```

AS

```
(t1:tuple(tla:chararray,t1b:int),t2:tuple(t2a:chararray,t2b:int));
```

```
B = FOREACH A GENERATE t1.tla, t1.t1b,t2.$0,t2.$1;
```

```
DUMP B;
```

Output:

```
(John,12,Jack,13)
(James,7,Joseph,5)
(Smith,8,Scott,12)
[root@volgalnx010 pigdemos]#
```

Objective: To depict the complex data type “map”.

Input:

John [city#Bangalore]

Jack [city#Pune]

James [city#Chennai]

Act:

A= load '/root/pigdemos/studentcity.tsv' Using PigStorage as

(studname:chararray,m:map[chararray]);

B = foreach A generate m#'city' as CityName:chararray;

DUMP B

Output:

(Bangalore)

(Pune)

(Chennai)

[root@volgalnx010 pigdemos]#

10.22 PIGGY BANK

Pig user can use Piggy Bank functions in Pig Latin script and they can also share their functions in Piggy Bank.

Objective: To use Piggy Bank string UPPER function.

Input: Student (rollno:int,name:chararray,gpa:float)

Act:

register '/root/pigdemos/piggybank-0.12.0.jar';

A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);

upper = foreach A generate

org.apache.pig.piggybank.evaluation.string.UPPER(name);

DUMP upper;

Output:

(JOHN)

(JACK)

(SMITH)

(SCOTT)

(JOSHI)

(ALEX)

(DAVID)

(JAMES)

(JOHN)

(JOSHI)

[root@volgalnx010 pigdemos]#

10.23 USER-DEFINED FUNCTIONS (UDF)

Pig allows you to create your own function for complex analysis.

Objective: To depict user-defined function.

Java Code to convert name into uppercase:

```
package myudfs;

import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;

public class UPPER extends EvalFunc<String>
{
    public String exec(Tuple input) throws IOException {
        if (input==null || input.size() == 0)
            return null;
        try{
            String str = (String)input.get(0);
            return str.toUpperCase();
        } catch (Exception e){
            throw WrappedIOException.wrap ("Caught
            exception processing input row ", e);
        }
    }
}
```

Note: Convert above java class into jar to include this function into your code.

Input:

Student (rollno:int,name:chararray,gpa:float)

Act:

```
register /root/pigdemos/myudfs.jar;
```

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
```

```
B = FOREACH A GENERATE myudfs.UPPER(name);
```

DUMP B;

Output:

```
(JOHN)
(JACK)
(SMITH)
(SCOTT)
(JOSHI)
(ALEX)
(DAVID)
(JAMES)
(JOHN)
(JOSHI)
[root@volgalnx010 pigdemos]#
```

PIG VERSUS HIVE

Features	Pig	Hive
Used By	Programmers and Researchers	Analyst
Used For	Programming	Reporting
Language	Procedural data flow language	SQL Like
Suitable For	Semi - Structured	Structured
Schema/Types	Explicit	Implicit
UDF Support	YES	YES
Join/Order/Sort	YES	YES
DFS Direct Access	YES (Implicit)	YES (Explicit)
Web Interface	YES	NO
Partitions	YES	NO
Shell	YES	YES

MODULE-5

Spark and Big Data Analytics: Spark, Introduction to Data Analysis with Spark. Text, Web Content and Link Analytics: Introduction, Text Mining, Web Mining, Web Content and Web Usage Analytics, Page Rank, Structure of Web and Analyzing a Web Graph.

TB2: Ch5: 5.2,5.3, Ch 9: 9.1-9.4

5.1 SPARK

Apache Spark is a fast and general compute engine. Apache Spark powers the analytics applications up to 100 times faster. It supports HDFS compatible data. Spark has a simple and expressive programming model. Expressive program model implements a number of mathematical and logic operations with smaller and easier written codes which a compiler as well as programmer can understand easily. The model, therefore, gives programming ease for a wide range of applications. Applications of expressive codes are in analytics, Extract Transform Load (ETL), Machine Learning (ML), stream processing and graph computations. Spark runs on both Windows and UNIX-like systems, such as Linux and Mac OS. Java is essential for running Spark applications. Executing a spark application on a computer system therefore requires setting a JDK path using JAVA_HOME environment variable or system variable.

5.1.1 Introduction to Big Data Tool-Spark

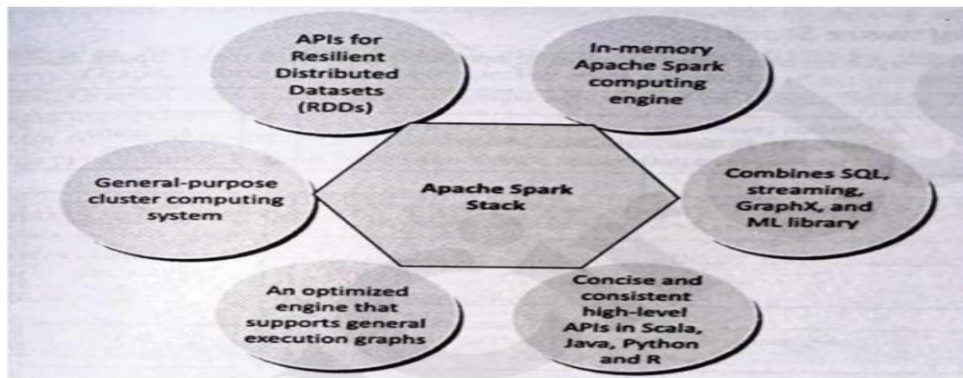
Figure below shows the main components in the Apache software Foundation's Spark framework, which includes data storage, API's and resources management bonded with functions in Spark core.

Main components of Spark Architecture are:

1. Spark HDFS file system for data storage: Storage is at an HDFS or Hadoop compatible data source such as HDFS, HBase, Cassandra, Ceph or at the objectstore S3.
2. Spark standard API enables the creation of applications using Scala, Java, Python and R.
3. Spark resource management can be a stand-alone server or it can be on a distributed computing framework, such as YARN or Mesos. Ceph refers to an open source, scalable object, block, file storing unified system. Ceph provides for dynamic replication and redistribution. Ceph provides HDFS compatible mechanisms for Big data in petabytes or exabytes. An application uniquely accesses the object, block and file stores in a system using Ceph. Ceph is highly reliable. Apache Mesos is an open source project developed at University of Berkeley, and now passed to Apache. It manages computing clusters. Its implementation is in C++. Apache released a stable version 1.3.0 of Mesos in 2017. Apache Mesos enables fine-grained sharing of CPU, RAM, IOs and other frameworks. Mesos offers them resource. Each resource contains a list of agents. Each agent has the Hadoop and Mapreduce executors. S3(Simple Storage Service) refers to an Amazon we service on the cloud names S3 which provides Object stores for data.

5.1.1.1 Features of Spark

Figure below shows the main features of the Spark.

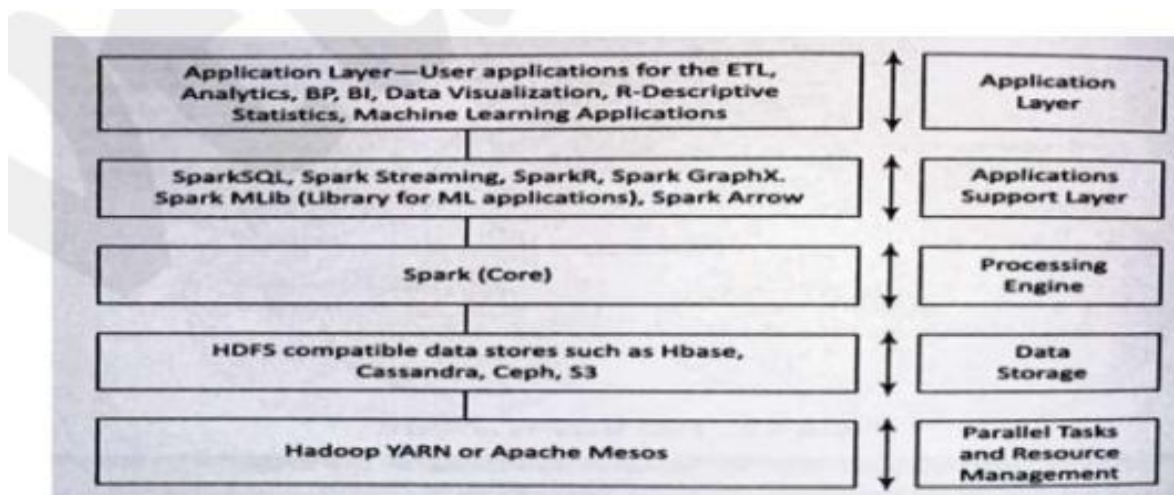


The features of Spark:

1. Spark provisions for creating applications that use the complex data. In-memory Apache Spark computing engine enables up to 100 times performance with respect to Hadoop.
2. Execution engine uses both in-memory and on-disk computing. Intermediate results save in-memory and spill over to disk.
3. Data uploading from an Object Store for immediate use as a Spark object instance. Spark service interface sets up the Object Store.
4. Provides high performance when an application accesses memory cache from the disk.
5. Contains API to define Resilient Distributed Datasets (RDDs). RDD is a programming abstraction. RDD is the core concept in Spark framework. RDD represents a collection of Object Stores distributed across many compute nodes for parallel processing. Spark stores data in RDD on different partitions. A table has partitions into columns or rows. Similarly, an RDD can also be considered as a table in a database that can hold any type of data. RDDs are also fault tolerant.
6. Processes any kind of data, namely structured, semi-structured or unstructured data arriving from various sources.
7. Supports many new functions in addition to Map and Reduce functions.
8. Optimizes data processing performance by slowing the evaluation of Big Data queries.
9. Provides concise and consistent APIs in Scala, Java and Python. Spark codes are in Scala and run on JVM environment.
10. Provides powerful tool to analyze data interactively using shell which is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python. The tool also provides for learning the usages of API.

5.1.1.2 Spark Software Stack

Figure below shows a five-layer architecture for running applications when using Spark stack.



The main components of Spark stack are SQL, Streaming, R, GraphX, MLlib and Arrow at the applications support layer. Spark core is the processing engine. Data Store provides the data to the processing engine. Hadoop, YARN or Mesos facilitates the parallel running of the tasks and the management and scheduling of the resources.

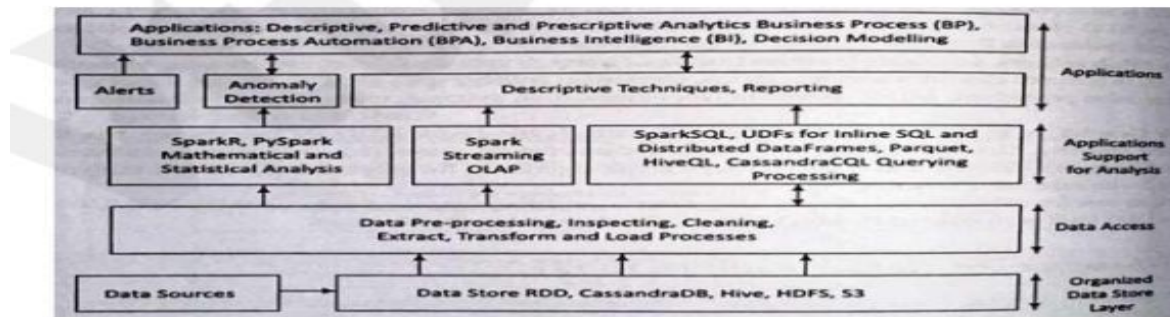
Spark Stack

Spark stack imbibes generality to Spark. Grouping of the following forms Spark stack: Spark SQL for the structured data. The SQL runs the queries on Spark data in the traditional business HQL queries also run in Spark SQL. Runs UDFS for inline SQL, distributed DataFrames, Parquet, Hive and analytics and visualization applications. Spark SQL enables Spark datasets to use JDBC or ODBC APL Cassandra Data Stores. Spark Streaming is for processing real-time streaming data. Processing is based on micro-batches style of computing and processing. Streaming uses the DStream which is basically a series of RDDs, to process the real-time data. SparkR is an R package used as light-weight front end for Apache Spark from R. Spark API uses SparkR through the RDD class. A user can interactively run the jobs from the R shell on a cluster. An RDD API is in the distributed lists in R. Spark MLlib is Spark's scalable machine learning library. It consists of common learning algorithms and utilities. MLlib includes classification, regression, clustering, collaborative filtering, dimensionality reduction and optimization primitives. MLlib applies in recommendation systems, clustering and classification using Spark. Spark GraphX is an API for graphs. GraphX extends the Spark RDD by introducing the Resilient Distributed Property. GraphX computations use fundamental operators (e.g., subgraph, join Vertices and aggregateMessages). GraphX uses a collection of graph algorithms for programming. Graph analytics tasks are created with ease using GraphX. Spark Arrow for columnar in-memory analytics and enabling usages of vectorised UDFS (VUDFS). The Arrow enables high performance Python UDFs for SerDe and data pipelines.

5.2 INTRODUCTION TO DATA ANALYSIS WITH SPARK

"Analysis of data is a process of inspecting, cleaning, transforming and modelling data with the goal of discovering useful information, suggesting conclusions and supporting decision-making." For examples, consider a car company. Assume that company sells five models of cars. Each model is available in 16 different colours and shades. Sales are processed through a large number of showrooms, all over the country. An analyses of annual sales and annual profits model-wise, colour-wise, region-wise and showroom-wise help in discovering useful information and making suggestive conclusions. The company

does predictive analytics from the results of the analyses and decides the future strategies for manufacturing, sales and out-reaches to customers on the basis of the results of the analytics. Figure below shows the steps between acquisition of data from different sources, applications of the analyzed data, and application support by Spark for the analyses.



Following are the steps for analyzing the data:

1. Data Storage: Store of data from the multiple sources after acquisition. The Big Data storage may be in HDFS compatible files, Cassandra, Hive, HDFS or S3.
2. Data pre-processing: This step requires:
 - (a) dropping out of range, inconsistent and outlier values,
 - (b) filtering unreliable, irrelevant and redundant information,
 - (c) data cleaning, editing, reduction and/or wrangling,
 - (d) data-validation, transformation or transcoding.
3. Extract, transform and Load (ETL)) for the analysis
4. Mathematical and statistical analysis of the data obtained after querying relevant data needing the analysis, or OLAP,
5. Applications of analyzed data, for example, descriptive, predictive and prescriptive analytics, business processes (BPS), business process automation (BPA), business intelligence (BI), decision modelling and knowledge discovery.

5.2.1 Spark SQL

Spark SQL is a component of Spark Big Data Stack. Spark SQL components are DataFrames (SchemaRDDs), SQLContext and JDBC server. Spark SQL at Spark does the following:

1. Runs SQL like scripts for query processing, using catalyst optimizer and tungsten execution engine
2. Processes structured data
3. Provides flexible APIs for support for many types of data sources.
4. ETL operations by creating ETL pipeline on the data from different file-formats, such as JSON, Parquet, Hive, Cassandra and then run ad-hoc querying.

Spark SQL has the following features for analysis:

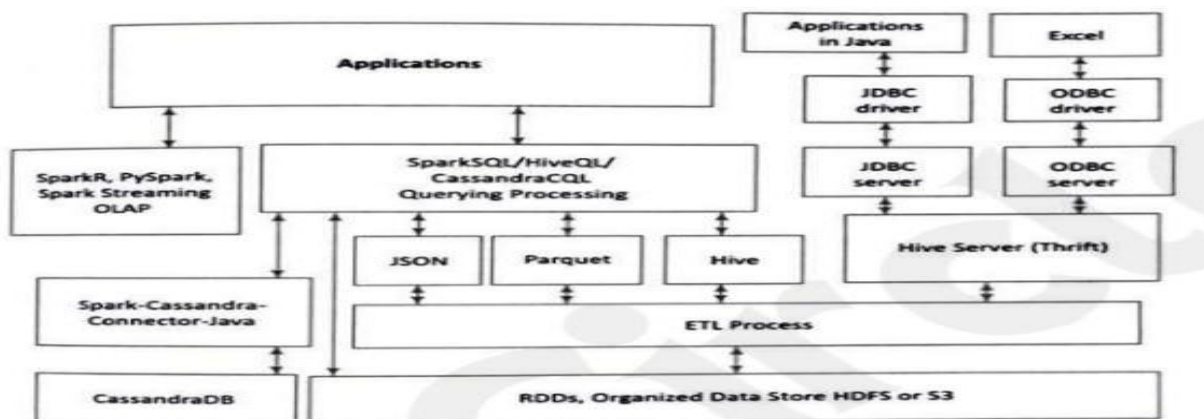
1. SparkR, PySpark, Python, Java and other language support for coding for data analysis.
2. Provisioning of JDBC and ODBC APIs: Applications in Java and Microsoft programs (such as Excel) need to connect to databases using JDBC (Object Database Connectivity) and ODBC (Object Database Connectivity). Spark APIS enable that connectivity,
3. Spark SQL enables users to exit their data from different formats, such as Hive, JSON and Parquet, and then transform that into required formats for ad hoc querying.

[Ad hoc query is a query just for this purpose' or query 'on the fly'. For example, var

newToyQuery = "SELECT * FROM table WHERE ID = "+ toy_puzzleId. The result will be different each time this code executes, depending on the value of toy_puzzleId.

4. Spark SQL processing support inclusion of Hive. Hive support enables the use of Hive tables, database and data warehouse, UDFs and SerDe (serialization and deserialization).
5. Spark SQL supports HiveQL and Cassandra CQL for query processing.
6. Spark Streaming for support to OLTP and structured streaming.

Figure below shows the connectivity of applications with Spark SQL which connects to Object Stores in different formats.



JDBC Server An application reads the data tables in RDBMS using a JDBC client (JDBC API at the application). Many applications in Java connect to databases using JDBC driver and server. Spark SQL API provides JDBC connectivity. Command for using JDBC server is as follows:

```
./sbin/start-thriftserver.sh -- master sparkMaster
```

Hive Server (Thrift) enables a remote Hive client or JDBC driver to send a request to Hive and the server sends response to that. The requests can be in Scala, Java, Python or R. JSON, Hive, Parquet Objects HDFS is highly reliable for very long running queries. However, 10 operations are slow, Columnar storage is a solution for faster IOS. Columnar storage stores the data portion, presently required for the IOs. Load only columns access during processing. Also, a columnar object Data Store can be compressed or encoded according to the data type. Also, executions of different columns or column partitions can be

in parallel at the data nodes. Hive RC file records store in columns and can be partitioned into row groups. An ORC file consists of row-groups row data called stripes. ORC enables concurrent reads of the same file using separate RecordReaders.' Metadata stored using protocol buffers for addition and removal of fields. Parquet is a nested hierarchical columnar storage concept. Apache Parquet file is a columnar storage file. The file uses an HDFS block. The block saves the file for running big long queries on Big Data. Each file compulsorily consists of metadata, though a file need not consist of data. An application retrieves the columnar data quickly from Parquet files. Apache Parquet three projects specify the usages of files for query processing or applications.

The projects are (1) parquet-format for specifying formats and Thrift definitions of metadata, (ii) parquet-mr for implementing the sub-modules in the core components for reading and writing a nested, column-oriented data stream, and (iii) parquet-compatibility for compatibility for read-write in multiple languages. Spark DataFrame (SchemaRDD) A DataFrame is a distributed collection of data organized into nane columns. DataFrame can be used for transformation using filter, join, or groupby aggregation functions. An RDD method converts Spark DataFrames to RDDs. Each RDD consists of a number of row objects. Creating Spark Dataframe (SchemaRDD) from Parquet and JSON Objects Data Frames can be created from different data sources. Examples of data sources are JSON datasets, Hive tables, Parquet row Creating Spark DataFrame (SchemaRDD) from Parquet and JSON Objects DataFrames can be groups, structured data files, external databases and existing RDDs.

The following example explains the creation and usages of DataFrames from the Parquet and JSON objects:

EXAMPLE

Assume a table toyPuzzleTypeCostTbl with four columns. Figure below shows the sample table toy PuzzleTypeCostTbl. The columns are puzzle type, puzzle code, number of puzzle pieces and puzzle cost. DataFrame1 named toyPuzzleTypeCodes consists of columns 1 and 2. DataFrame2 named toyPuzzleCodesCost consists of Columns 2 and 4. The table consists of multiple rows in each row group. The dashed lines point to the columns in the two data frames.

	puzzleType	puzzleCode	puzzlePieces	puzzleCost
Row Group1	puzzle_Garden	10725	100	1.35
	puzzle_Garden	10825	200	1.35
	puzzle_Garden	10975	400	1.35
Row Group2	...	...	...	...
	puzzle_Jungle	31047	300	2.85
	puzzle_Jungle	31047	300	2.85
Row Group3	...	...	...	...
	puzzle_School	81409	800	0.90
	puzzle_Forest	...	...	...

DataFrame toyPuzzleTypeCodes Columns 1 and 2

DataFrame toyPuzzleCodesCost Columns 2 and 4

(i) Create a sqlContext from a given SparkContext 'sc'.

(ii) Create a four columns DataFrame using the Parquet file named "toyPuzzleTypeCostTbl". (iii) How will a DataFrame create using a JSON file format file "toyPuzzleTypeCostTbl"? Create two DataFrames using Java and SqlContext, one with columns for puzzle type and puzzle code, and the other for puzzle code and cost.

(iv) How will two DataFrames join using puzzle code as a join key to create a DataFrame of three columns, 1, 2 and 4?

SOLUTION

(i) The following statement creates sqlContext from Spark Context sc:

```
SqlContext sqlContext = new org.apache.spark.sql.SQLContext (sc)
```

(ii) The following statement creates a DataFrame named toyTypeCost:

```
DataFrame toyType Cost = sqlContext.
```

```
parquet Pile ("toyPuzzleTypeCost Tbl")
```

[DataFrame created will have four columns.]

(iii) DataFrame creates using Load() method at the sqlContext.

```
#To display the contents of Table: "toyPuzzleTypeCostTbl"
```

```
spark.sql ("SELECT * FROM toyPuzzleTypeCost Tbl ")
```

```
#To create DataFrame toyPuzzleTypeCost Tbl
```

```
DataFrame toyPuzzleTypeCostTbl = sqlContext.Load
```

```
("toyPuzzleTypeCost Tbl", "json")
```

The following statements create two DataFrames using DataFrame

toyPuzzleTypeCostTbl. One frame of two columns, 1 and 2, puzzle type and puzzle code:

```
DataFrame toyTypeCodes = toyPuzzleTypeCostTbl.
```

```
select (toyPuzzleTypeCostTbl ['puzzleType'], toyPuzzleTypeCostTbl  
['puzzleCode'])
```

Second frame of two columns, 2 and 4, puzzle code and puzzle cost

```
DataFrame toyCodesCost = toyPuzzleTypeCostTbl.
```

```
select (toyPuzzleTypeCostTbl ['puzzleCode'], toyPuzzleTypeCostTbl ['puzzleCost'])
```

(iv) DataFrame1 (toyTypeCodes) has two columns, 1 and 2, as puzzle Type and puzzle Code, respectively. The frame joins dataframe2 (toyCodesCost) column 4 as puzzle Cost and resultant is a joined new. dataframe toy TypeCodes Cost consisting of columns 1, 2 and 4. Join key is puzzle Code. Following statements use join() method and joining key.

Format of the statement is

```
dataFrame.join (dataframe1, dataframe2.col ("JoinKey") = dataframeNew
("JoinKey"))
```

and the statement is

```
dataFrame.join(toyTypeCodes, toyCodesCost.col ("puzzleCode"). equalTo
(toyTypeCodes Cost ("puzzleCode"))))
```

Using HiveQL for Spark SQL Spark SQL programming provides two contexts, SQLContext and HiveContext. While using HiveContext, then, commands access the Hive Server only and use HiveQL commands. SQLContext is a subset of Spark SQL, SQLContext does not need the HiveServer (Thrift) Therefore, when needing access to the HiveServer, specify the HiveContext. HiveQL is recommended for Spark SQL. Many resources are available in Hive readily and can directly be used. Use of Aggregation and Statistical Functions Aggregation functions can be used for analysis. Hive consists of count (*), count (expr); sum (col), sum (DISTINCT col), avg (col), avg (DISTINCT col), min (col) and DOUBLE max(col) (Table 4.10). The statistical functions stdev(), sampleStdev(), variance sample Variance() can be used for analysis with DataFrames in input.

Consider the example below to find sum of the sales of the Jaguar Land Rover model and trace the showroom that recorded the best sales.

EXAMPLE

Consider the annual car sales data in the following format:

CarShowroomsCumulativeYearlySales						
Car ShowroomID (csID)	Date (DT) mmddyy	Jagaur Land Rover Sale (JLRDS)	Hexa Sale (HDS)	Zest Sales (ZDS)	Nexon Sale (NDS)	Safari Storme Sale (SSDS)
220	----	--	---	----	---	---
10	121217	49	34	164	115	38
122	121217	40	141	123	37	88
-	-	-	-	-	-	-
-	-	-	-	-	-	-

- (i) Find the Jagaur Land Rover annual sale figure over all showrooms.
- (ii) Find the showroom ID and Land Rover sales figure for the showroom giving maximum Jaguar Land Rover sales.

SOLUTION

- (i) The following query returns the sum of the Jagaur Land Rover annual sale in column 3 of the table:

```
SELECT sum (JLRDS) FROM CarShowroomsCumulativeYearlySales
```

- (ii) The following nested query statement returns the csID and maximum annual sale value for Land Rover:

```
SELECT CSID, saleJL (max (map (csID, ID, JLRDS, saleJL))) FROM
```

CarShowroomsCumulativeYearlySales

5.2.2 Using Python Advanced Features with Spark SQL

Python is a general purpose, interpreted, interactive, object oriented and high level programming language. Python defines the basic data types, containers, lists, dictionaries, sets, tuples, functions and classes. Python Standard Library is very extensive. The libraries for regular expressions, documentation generation, unit testing, web browsers, threading, databases, CGI, email, image manipulation and a lot of other functionalities are available in Python. Python programming is a strong combination of performance and features in the same bundle of codes. Spark SQL binds with Python easily. Python has the expressive program statements. Spark SQL features together with Python help a programmer to build challenging applications for Big Data. The following example explains the use of PySpark, Python along with the Spark SQL:

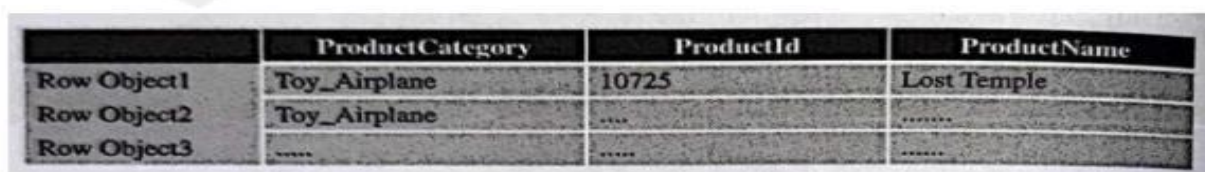
Example

Questions:

1. How is HiveContext and Spark SQL used?
2. How does PySpark use a row object?
3. Assume a JSON file, toyTypeProductTbl of row objects.

Do a file load and query sent to the file: toyPuzzleProduct

The table (toyPuzzleProduct) contains:



	ProductCategory	ProductId	ProductName
Row Object1	Toy_Airplane	10725	Lost Temple
Row Object2	Toy_Airplane	10725	Lost Temple
Row Object3	Toy_Airplane	10725	Lost Temple

Solution:

Import Spark SQL using:

```
import Spark SQL
```

Importing a row object using HiveContext:

```
from pyspark.sql import HiveContext, toyPuzzleTypeCost
```

Load the JSON file and query it:

```
input = hiveCtx.jsonFile(toyTypeProductTbl)
```

```
input.registerTempTable("toyPuzzleProduct")
```

Query to find Product_ID name:

```
Product_ID_name = hiveCtx.sql(
```

```
"SELECT ID, name FROM toyPuzzleProduct ORDER BY ProductCategory"
```

```
)
```

5.2.2.1 Python Libraries for Analysis

NumPy and SciPy are open source downloadable libraries for numerical (Num) analysis and scientific (Sci) computations in Python (Py). Python has open source library packages, NumPy, SciPy, Scikit-learn, Pandas and StatsModel, which are widely used for data analysis. Python library, matplotlib functions plot the mathematical functions. Spark added a Python API support for UDFS. The functions take one row at a time. That requires overhead (additional codes) for SerDe. Earlier data pipelines first defined the UDFS in Java or Scala, and then invoked them from Python. Spark 2.3 provisions for vectorized UDFS (VUDFs) and Apache Arrow facilitates VUDFs which enables high performance Python UDFS for SerDe and data pipelines.

NumPy NumPy includes

- (i) N-dimensional array object, array and vector mathematics;
- (ii) linear algebraic functions, Fourier transform and random number functions;
- (iii) sophisticated (broadcasting) functions; and (iv) tools for integrating with C/C++ and Fortran codes.

NumPy provides multi-dimensional efficient containers of generic data and definitions of arbitrary data types. NumPy integrates easily with a wide variety of databases. NumPy provides import, export (load/save) elements in the arrays, indexing, subsetting and slicing of the arrays, scalar and vector mathematics (such as files, creation of arrays, inspection of properties, copying, sorting and reshaping, addition and removal of X, power, sqrt, sin, log, ceil - round up to nearest int, floor-round down up to the nearest int, round -round to nearest integer). NumPy also provides statistical functions. Table below gives the examples of NumPy functions for data analysis problems.

Table: Examples of NumPy Functions for Data Analysis Problems

Function	Description
<code>np.loadtxt('file.txt')</code>	Loads data from a text file.
<code>np.genfromtxt('file.csv', delimiter=',')</code>	Loads a CSV file with comma as the delimiter between records.
<code>np.savetxt('file.txt', arr, delimiter=' ')</code>	Saves an array to a text file with values separated by spaces.
<code>np.mean(arr, axis=0)</code>	Returns the mean of the array along a specified axis.
<code>np.sum(arr)</code>	Returns the sum of all elements in the array.
<code>np.min(arr)</code>	Returns the minimum value of the array.
<code>np.max(arr, axis=0)</code>	Returns the maximum value along a specified axis.
<code>np.var(arr)</code>	Returns the variance of the array.

Function	Description
<code>np.std(arr, axis=0)</code>	Returns the standard deviation along a specified axis.
<code>arr.sort()</code>	Sorts the elements of the array in ascending order.
<code>np.add(arr1, arr2)</code>	Performs element-wise (vector) addition of two arrays.
<code>np.corrcoef(arr)</code>	Returns the correlation coefficient of the array.

SciPy adds on top of NumPy. It includes MATLAB files and special functions, such as routines for numerical integration and optimization. SciPy defines some useful functions for computing distances between a set of points. SciPy includes

- (i) interactions with NumPy,
- (ii) creation of dense and open mesh grids,
- (iii) shape manipulation functions,
- (iv) polynomial and vectoring functions,
- (v) real and imaginary functions, and casting an object to a data type, and (vi) matrix creation and matrices routines and usages of sparse matrices. Table below gives few examples of SciPy functions for scientific computational problems.

Table: Examples of NumPy Functions for Data Analysis Problems

Function	Description
<code>np.loadtxt('file.txt')</code>	Loads data from a text file.
<code>np.genfromtxt('file.csv', delimiter=',')</code>	Loads a CSV file with comma as the delimiter between records.
<code>np.savetxt('file.txt', arr, delimiter=' ')</code>	Saves an array to a text file with values separated by spaces.
<code>np.mean(arr, axis=0)</code>	Returns the mean of the array along a specified axis.
<code>np.sum(arr)</code>	Returns the sum of all elements in the array.
<code>np.min(arr)</code>	Returns the minimum value of the array.
<code>np.max(arr, axis=0)</code>	Returns the maximum value along a specified axis.
<code>np.var(arr)</code>	Returns the variance of the array.
<code>np.std(arr, axis=0)</code>	Returns the standard deviation along a specified axis.
<code>arr.sort()</code>	Sorts the elements of the array in ascending order.
<code>np.add(arr1, arr2)</code>	Performs element-wise (vector) addition of two arrays.

Function	Description
<code>np.corrcoef(arr)</code>	Returns the correlation coefficient of the array.

Panda Panda derives its name from usages of a data structure called Panel. The first three characters Pan in Panda stand for the term 'panel'. The next two characters da in Panda stand for data. The Panda package considers three data structures: Series, DataFrame and Panel. DataFrame is a container for Series. Panel is a container for DataFrame objects. The Panel objects can be datasets. The data need not be labeled. This means datasets, objects and DataFrames can be placed into a inserted or removed similar to as in a dictionary. DataFrame may be a DataFrame of statistical or observed Panda data structure without labels. Panel is a container for three-dimensional data. Panel is a widely used term in econometrics. Three axes describe operations involving panel data. For example, panel data in econometric analysis. Items can be considered as along axis 0 of an inside DataFrame (set of columns). Index (rows) of each of the DataFrames correspond to axis 1 (major axis). Columns of each of the DataFrames correspond to axis 2 (minor axis) Panel 4D consists of labels as 0 - axis, items - axis 1, major axis - axis 2 and minor axis - axis 3. Figure below shows the main features of Pandas package.



Pandas package includes the following provisions:

1. Database style DataFrames merge, join, and concatenation of objects
2. RPy interface for R functions plus additional functions
3. Panda ecosystem has statistics, machine learning, integrated development environment (IDE), API and several out of core features
4. SQL like features: SELECT, WHERE, GROUPBY, JOIN, UNION, UPDATE and DELETE
5. GroupBy feature of split-apply-combine with the steps as: (i) an object such as table, file or document splits into groups, (ii) iterate through the groups and select a group for aggregation, transformation and/or

filtration. The instance method can be dispatched and applied in a manner similar to the aggregation/transformation function

6. Size mutability, which means that columns can be inserted and deleted from DataFrame and higher dimensional objects

7. Slicing and dicing a collection of DataFrame objects. The names of axes can be somewhat arbitrary in a Panel. (Arbitrary means the axes need not be named a1, a2, ..., an, and can be year, car model, sales, ...). If slicing function slices the first dimension, the lower dimension objects are obtained.

5.2.2.2 User-Defined Functions (UDFs)

The functions take one row at a time. This requires overhead for SerDe. Data exchanges take place between Python and JVM. Earlier the data pipeline (between data and application) defined the UDFs in Java or Scala, and then invoked them from Python while using Python libraries for analysis or other application. SparkSQL UDFS enable registering of themselves in Python, Java and Scala. The SQL calls the UDFS. This is a very popular way to expose advanced functionality to SQL users. User codes call the registered UDFS into the SQL statements without

writing the detailed codes. The following example demonstrates how a UDF is created in PySpark.

EXAMPLE

Create a UDF, `udfCost Plus ()` in pandas. The table column puzzle Cost creates using `jigsaw puzzle_info.txt` from an RDD. Write a UDF which increases the costs in the column, `puzzle_cost USD` by 10%. (The UDF takes one row at a time as input.)

SOLUTION

Following are the Python statements

```
from pyspark.sql.functions import udf
```

[Use udf to define a row at a time udf.]

```
@udfCostPlus ('float')
```

[Input and output costs are two values both for a single float variable, v.]

```
def plusTenPercent (v) ;
```

```
return v + 0.1 x v;
```

```
df.withColumn ('v4', puzzle_cost USD (df.v))
```

[Data Frame df has v4 as puzzleCost in the fourth column.]

Dataset at Example consists of car sales data. Column 1 represents car showroom ID. The ID key is has the Jaguar Land Rover sales figures in more than 300 rows for more than 300 dates. Sales figures of four present in all date fields on which the sales were recorded during the year. Corresponding to an ID, column 3 other models are in columns 4, 5, 6 and 7. The product sales analysis is widely performed in many businesses. Writing a UDF for analysis of sales once and using it for different products whenever and

wherever desired reduces coding efforts. This also, integrates new functions (UDFs) in higher level language with their lower level language implementations. Also, writing UDFs are helpful when built-in functionalities in a currently used tool needs additional functionalities.

A UDF using aggregation function `max()` calculates the Land Rover sales and traces the showroom giving maximum Jaguar Land Rover sales. The UDF is of great help to perform similar analyses on different models of the car. Java class can be created user defined method (UDF) product `SalesAnalysis ()`. Python scripts can also create the UDF to find that sales point ID and total yearly sale for that sales point from which the total is highest for a product. The UDF will be reusable not only for car sales analysis but also for analysis of sales of many companies, such as ACVM or Toy Company. Python provides a register function: `hiveContext (sc).registerFunction ()`. The command can be `hiveContext (sc). registerFunction ("csIDn1", int: bestYearlySalesModel1, LongType (), ("csIDn2", int: best YearlySalesModel2, LongType (), ...). csIDn1 car-showroom ID1, bestYearlySalesModel1 is best yearly sale of model 1.`

5.2.2.3 Vectorized User Defined Functions (VUDFS)

Python UDFS express data in detail. Therefore, Python UDFs, block-level UDFS with block-level arguments and return types, conversions or transformations are widely used in ETL or ML applications. Spark Arrow facilitates columnar in-memory analytics, which results in high performance of Python UDFS, SerDe and data pipelines. VUDFs use series data structure (meaning one-dimensional array or tuples). Spark 2.3 (2018) provisions for using vectorized UDFS (VUDFs). Apache Arrow 0.8.0 (release date December 18, 2017) facilitates usages of VUDFS. Pandas UDF, `pandas_UDF` uses the function to create a VUDF with

- (i) `pandas.Series` as input to the UDF,
- (ii) `pandas.Series` as output from the UDF,
- (iii) no grouping using `GroupBy`,
- (iv) output size same as input, and
- (v) returns the same data types as specified type in return `pandas.Series`.

The following example explains the use of VUDF.

EXAMPLE

Create a vectorized UDF (VUDF). First define a `pandas_UDFCostPlus` for increasing cost `puzzle_cost_USD` of toys in `puzzle_Costs` RDD created from `jigsaw puzzle_info.txt`.

SOLUTION

Following are the Python statements from `pyspark.sql.functions` import `pandas_udf`:

```
from pyspark.sql.functions import pandas_udf
```

```
[Use pandas_udf to define a vectorized udf.]
```

```
@pandas_udf Cost Plus ('float')
```

```
[Input/output are both a pandas.Series of elements with data type float.]
```

```
def vectorized_plusTenPercent (v):
```

```
return v4 + 0.1
```

```
df.withColumn('v4', vectorized_plusTenPercent(df.v))
```

[Use udf to define a DataFrame vudf.]

5.2.2.4 Grouped Vectorized UDFS (GVUDFS)

Grouped Vectorized UDFS (GVUDFS) use Panda library split-apply-combine pattern in data analysis. The GVUDF group function operates on all the data for a group, such as operate on all the data, "for each carshowroom, compute yearly sales". GVUDF steps are:

1. Splits a Spark DataFrame into groups based on the conditions specified in the groupby operator
2. Applies a vectorized user-defined function (pandas.DataFrame -> pandas.DataFrame) to each group
3. Combines into new group

4. Returns the results as a new Spark DataFrame Pandas GVUDF, pandas_GVUDF,

- (i) uses the function similar to pandas_VUDF,

- (ii) pandas.DataFrame as input to the GVUDF,

- (ii) pandas.DataFrame as output from the

GVUDF,

- (iii) grouping semantics defined using clause GroupBy,

- (iv) output size can be

any and can be grouped and can be distinct from input, and

- (v) returns data type is a

StructType. The type defines a column name and the type of the returned pandas. DataFrame.

The following example explains GVUDF for adding 10% in a cost of group of rows for toy products.

EXAMPLE

Add 10% cost in each value of item cost in a group of rows. Use GVUDF to define a DataFrame costTen Percet Plus GVUDF.

SOLUTION

Following are the Python statements from pyspark.sql.functions import pandas_udf:

```
from pyspark.sql.functions import pandas_udf
```

[Use pandas_udf to define a grouped vectorized udf.]

```
@pandas_udf(df.schema)
```

```
# Input/output are both a pandas. DataFrame  
def costTenPercet Plus (pdf):  
    return pdf.assign (v=add (pdf.v + 0.1xpdf.v))  
df.groupby('id').apply (costTenpercet Plus)
```

5.2.3 Data Analysis Operations

Examples of operations required in the above analysis are given below:

1. Filtering single and multiple columns
2. Creating a top-ten list with values or percentages
3. Setting up sub-totals
4. Creating multiple-field criteria filters
5. Creating unique lists from repeating field data
6. Finding duplicate data with specialized arrays, and using the remove duplicates command removing outliers
7. Multiple key sorting
8. Counting the number of unique items in a list
9. Using SUMIF and COUNTIF functions
10. Working with database functions, such as DSUM and DMAX
11. Converting lists to tables.

5.2.3.1 Removing Outliers for Data Quality Improvement for Analysis

Outliers are data which appear as they do not belong to the data set. The Outliers are generally results of human data-entry errors, programming bugs, some transition effect or phase lag in stabilizing the data vale to the true value. The actual outliers need to be removed from the data set. For example, missing decimal in the cost of toy US\$ 1,85 will make the cost 100 times more for a single toy. The result will thus be affected by a smal or large amount. When valid data is identified as an outlier, then also the results are affected. The statistical mean is computed from the product of each observed value v of Values with probability (or weight) P and then taking the average. The variance equals the

difference of a value with respect to the mean, then square that, and then average the results. Standard deviation is just the square root of the variance The following example explains the Python codes for removing outliers.

EXAMPLE

How will you remove outliers in values in a column?

SOLUTION

Transform RDD string or other to numeric data so that statistical methods compute and remove those who have larger distanceNumerics.

```
distanceNumerics = distances.map(PySpark string: float (string))
```

```
stats = distanceNumerics.stats()
```

[stats() means a statistical function, such as mean(), stdev()]

```
statdev = std.stdev ()
```

```
mean = stats.mean ()
```

```
reasonableDistances = distance Numerics.filter (PySpark values: maths.fabs (values- mean) < 3 xstdev)
```

[Assume that distances that are reasonable are less than three times the standard deviation. Distance means difference with respect to mean or peak value.]

```
print reasonableDistances.collect ()
```

[Print the values within reasonable distances.]