ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY

Kanyakumari Main Road, near Anjugramam, Palkulam, Anjugramam, Tamil Nadu 629401

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ADD ON COURSE

BIG DATA ANALYTICS

COURSE MATERIAL

BIG DATA ANALYTICS

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

This definition clearly answers the "What is Big Data?" question – Big Data refers to complex and large data sets that have to be processed and analyzed to uncover valuable information that can benefit businesses and organizations.

However, there are certain basic tenets of Big Data that will make it even simpler to answer what is Big Data:

- It refers to a massive amount of data that keeps on growing exponentially with time.
- It is so voluminous that it cannot be processed or analyzed using conventional data processing techniques.
- It includes data mining, data storage, data analysis, data sharing, and data visualization.
- The term is an all-comprehensive one including data, data frameworks, along with the tools and techniques used to process and analyze the data.

The History of Big Data

Although the concept of big data itself is relatively new, the origins of large data sets go back to the 1960s and '70s when the world of data was just getting started with the first data centers and the development of the relational database.

Around 2005, people began to realize just how much data users generated through Facebook, YouTube, and other online services. Hadoop (an open-source framework created specifically to store and analyze big data sets) was developed that same year. NoSQL also began to gain popularity during this time.

The development of open-source frameworks, such as Hadoop (and more recently, Spark) was essential for the growth of big data because they make big data easier to work with and cheaper to store. In the years since then, the volume of big data has skyrocketed. Users are still generating huge amounts of data—but it's not just humans who are doing it.

With the advent of the Internet of Things (IoT), more objects and devices are connected to the internet, gathering data on customer usage patterns and product performance. The emergence of machine learning has produced still more data.

While big data has come far, its usefulness is only just beginning. Cloud computing has expanded big data possibilities even further. The cloud offers truly elastic scalability, where developers can simply spin up ad hoc clusters to test a subset of data.

Benefits of Big Data and Data Analytics

- Big data makes it possible for you to gain more complete answers because you have more information.
- More complete answers mean more confidence in the data—which means a completely different approach to tackling problems.

Types of Big Data

Now that we are on track with what is big data, let's have a look at the types of big data:

a) Structured

Structured is one of the types of big data and By structured data, we mean data that can be processed, stored, and retrieved in a fixed format. It refers to highly organized information that can be readily and seamlessly stored and accessed from a database by simple search engine algorithms. **For instance, the employee table in a company database will be structured as the employee details, their job positions, their salaries, etc.,** will be present in an organized manner.

b) Unstructured

Unstructured data refers to the data that lacks any specific form or structure whatsoever. This makes it very difficult and time-consuming to process and analyze unstructured data. Email is an example of unstructured data. Structured and unstructured are two important types of big data.

c) Semi-structured

Semi structured is the third type of big data. Semi-structured data pertains to the data containing both the formats mentioned above, that is, structured and unstructured data. To be precise, it refers to the data that although has not been classified under a particular repository (database), yet contains vital information or tags that segregate individual elements within the data. Thus we come to the end of types of data.

Characteristics of Big Data

Back in 2001, Gartner analyst Doug Laney listed the **3 'V's of Big Data – Variety, Velocity, and**

Volume. Let's discuss the characteristics of big data. These characteristics, isolated, are enough to know what big data is. Let's look at them in depth:

a) Variety

Variety of Big Data refers to structured, unstructured, and semi-structured data that is gathered from multiple sources. While in the past, data could only be collected from spreadsheets and databases, today data comes in an array of forms such as emails, PDFs, photos, videos, audios, SM posts, and so much more. Variety is one of the important characteristics of big data.

Velocity

Velocity essentially refers to the speed at which data is being created in real-time. In a broader prospect, it comprises the rate of change, linking of incoming data sets at varying speeds, and activity bursts.

b) Volume

Volume is one of the characteristics of big data. We already know that Big Data indicates huge 'volumes' of data that is being generated on a daily basis from various sources like social media platforms, business processes, machines, networks, human interactions, etc. Such a large amount of data is stored in data warehouses. Thus comes to the end of characteristics of big data.

Why is Big Data Important?

The importance of big data does not revolve around how much data a company has but how a company utilizes the collected data. Every company uses data in its own way; the more efficiently a company uses its data, the more potential it has to grow. The company can take data from any source and analyze it to find answers which will enable:

1. **Cost Savings**: Some tools of Big Data like Hadoop and Cloud-Based Analytics can bring cost advantages to business when large amounts of data are to be stored and these tools also help in identifying more efficient ways of doing business.
2. **Time Reductions: The** high speed of tools like Hadoop and in-memory analytics can easily identify new sources of data which helps businesses analyzing data immediately and make quick decisions based on the learning.
3. **Understand the market conditions**: By analyzing big data you can get a better understanding of current market conditions. For example, by analyzing customers' purchasing behaviors, a company can find out the products that are sold the most and produce products according to this trend. By this, it can get ahead of its competitors.

4. **Control online reputation:** Big data tools can do sentiment analysis. Therefore, you can get feedback about who is saying what about your company. If you want to monitor and improve the online presence of your business, then, big data tools can help in all this.
5. **Using Big Data Analytics to Boost Customer Acquisition and Retention**

   The customer is the most important asset any business depends on. There is no single business that can claim success without first having to establish a solid customer base. However, even with a customer base, a business cannot afford to disregard the high competition it faces. If a business is slow to learn what customers are looking for, then it is very easy to begin offering poor quality products. In the end, loss of clientele will result, and this creates an adverse overall effect on business success. The use of big data allows businesses to observe various customer related patterns and trends. Observing customer behavior is important to trigger loyalty.
6. **Using Big Data Analytics to Solve Advertisers Problem and Offer Marketing Insights**

   Big data analytics can help change all business operations. This includes the ability to match customer expectation, changing company's product line and of course ensuring that the marketing campaigns are powerful.
7. **Big Data Analytics As a Driver of Innovations and Product Development**

   Another huge advantage of big data is the ability to help companies innovate and redevelop their products.

Business Intelligence vs Big Data

Although Big Data and Business Intelligence are two technologies used to analyze data to help companies in the decision-making process, there are differences between both of them. They differ in the way they work as much as in the type of data they analyze.

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).

A Big Data solution differs in many aspects to BI to use. These are the main differences between

Big Data and Business Intelligence:

1. In a Big Data environment, information is stored on a distributed file system, rather than on a central server. It is a much safer and more flexible space.
2. Big Data solutions carry the processing functions to the data, rather than the data to the functions. As the analysis is centered on the information, it´s easier to handle larger amounts of information in a more agile way.
3. Big Data can analyze data in different formats, both structured and unstructured. The volume of unstructured data (those not stored in a traditional database) is growing at levels much higher than the structured data. Nevertheless, its analysis carries different challenges. Big Data solutions solve them by allowing a global analysis of various sources of information.
4. 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.
5. 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.

Big Data vs 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.

Hadoop Environment Big Data Analytics

Hadoop is changing the perception of handling Big Data especially the unstructured data. Let's know how Apache Hadoop software library, which is a framework, plays a vital role in handling Big 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

Activities performed on Big Data

- **Store** – Big data need to be collected in a seamless repository, and it is not necessary to store in a single physical database.
- **Process** – The process becomes more tedious than traditional one in terms of cleansing, enriching, calculating, transforming, and running algorithms.
- **Access** – There is no business sense of it at all when the data cannot be searched, retrieved easily, and can be virtually showcased along the business lines.

Classification of analytics

Descriptive analytics

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 make decisions based on the computational findings of algorithmic models.

Generating automated decisions or recommendations requires specific and unique algorithmic models and clear direction from those utilizing the analytical technique. A recommendation cannot be generated without knowing what to look for or what problem is desired to be solved. In this way, prescriptive analytics begins with a problem.

Example

A Training Manager uses predictive analysis to discover that most learners without a particular

skill will not complete the newly launched course. What could be done? Now prescriptive analytics can be of assistance on the matter and help determine options for action. Perhaps an algorithm can detect the learners who require that new course, but lack that particular skill, and send an automated recommendation that they take an additional training resource to acquire the missing skill.

The accuracy of a generated decision or recommendation, however, is only as good as the quality of data and the algorithmic models developed. What may work for one company's training needs may not make sense when put into practice in another company's training department. Models are generally recommended to be tailored for each unique situation and need.

Descriptive vs Predictive vs Prescriptive Analytics

Descriptive Analytics is focused solely on historical data.

You can think of Predictive Analytics as then using this historical data to develop statistical models that will then forecast about future possibilities.

Prescriptive Analytics takes Predictive Analytics a step further and takes the possible forecasted outcomes and predicts consequences for these outcomes.

What Big Data Analytics Challenges?

1. Need for Synchronization Across Disparate Data Sources

As data sets are becoming bigger and more diverse, there is a big challenge to incorporate them into an analytical platform. If this is overlooked, it will create gaps and lead to wrong messages and insights.

2. Acute Shortage Of Professionals Who Understand Big Data Analysis

The analysis of data is important to make this voluminous amount of data being produced in every minute, useful. With the exponential rise of data, a huge demand for big data scientists and Big Data analysts has been created in the market. It is important for business organizations to hire a data scientist having skills that are varied as the job of a data scientist is multidisciplinary. Another major challenge faced by businesses is the shortage of professionals who understand Big Data analysis. There is a sharp shortage of data scientists in comparison to the massive amount of data being produced.

3. Getting Meaningful Insights Through The Use Of Big Data Analytics

It is imperative for business organizations to gain important insights from Big Data analytics, and also it is important that only the relevant department has access to this information. A big challenge faced by the companies in the Big Data analytics is mending this wide gap in an effective manner.

4. Getting Voluminous Data Into The Big Data Platform

It is hardly surprising that data is growing with every passing day. This simply indicates that business organizations need to handle a large amount of data on daily basis. The amount and variety of data available these days can overwhelm any data engineer and that is why it is considered vital to make data accessibility easy and convenient for brand owners and managers.

5. Uncertainty Of Data Management Landscape

With the rise of Big Data, new technologies and companies are being developed every day. However, a big challenge faced by the companies in the Big Data analytics is to find out which technology will be best suited to them without the introduction of new problems and potential risks.

6. Data Storage and Quality

Business organizations are growing at a rapid pace. With the tremendous growth of the companies and large business organizations, increases the amount of data produced. The storage of this massive amount of data is becoming a real challenge for everyone. Popular data storage options like data lakes/ warehouses are commonly used to gather and store large quantities of unstructured and structured data in its native format. The real problem arises when a data lakes/ warehouse try to combine unstructured and inconsistent data from diverse sources, it encounters errors. Missing data, inconsistent data, logic conflicts, and duplicates data all result in data quality challenges.

7. Security and Privacy Of Data

Once business enterprises discover how to use Big Data, it brings them a wide range of possibilities

and opportunities. However, it also involves the potential risks associated with big data when it comes to the privacy and the security of the data. The Big Data tools used for analysis and storage utilizes the data disparate sources. This eventually leads to a high risk of exposure of the data, making it vulnerable. Thus, the rise of voluminous amount of data increases privacy and security concerns.

Terminologies Used In Big Data Environments

- **As-a-service infrastructure**

Data-as-a-service, software-as-a-service, platform-as-a-service – all refer to the idea that rather than selling data, licenses to use data, or platforms for running Big Data technology, it can be provided "as a service", rather than as a product. This reduces the upfront capital investment necessary for customers to begin putting their data, or platforms, to work for them, as the provider bears all of the costs of setting up and hosting the infrastructure. As a customer, as-a-service infrastructure can greatly reduce the initial cost and setup time of getting Big Data initiatives up and running.

- **Data science**

Data science is the professional field that deals with turning data into value such as new insights or predictive models. It brings together expertise from fields including statistics, mathematics, computer science, communication as well as domain expertise such as business knowledge. Data scientist has recently been voted the No 1 job in the U.S., based on current demand and salary and career opportunities.

- **Data mining**

Data mining is the process of discovering insights from data. In terms of Big Data, because it is so large, this is generally done by computational methods in an automated way using methods such as decision trees, clustering analysis and, most recently, machine learning. This can be thought of as using the brute mathematical power of computers to spot patterns in data which would not be visible to the human eye due to the complexity of the dataset.

- **Hadoop**

Hadoop is a framework for Big Data computing which has been released into the public domain as open source software, and so can freely be used by anyone. It consists of a number of modules all tailored for a different vital step of the Big Data process – from file storage (Hadoop File System

– HDFS) to database (HBase) to carrying out data operations (Hadoop MapReduce – see below). It has become so popular due to its power and flexibility that it has developed its own industry of retailers (selling tailored versions), support service providers and consultants.

- **Predictive modelling**

At its simplest, this is predicting what will happen next based on data about what has happened previously. In the Big Data age, because there is more data around than ever before, predictions are becoming more and more accurate. Predictive modelling is a core component of most Big Data initiatives, which are formulated to help us choose the course of action which will lead to the most desirable outcome. The speed of modern computers and the volume of data available means that predictions can be made based on a huge number of variables, allowing an ever-increasing number of variables to be assessed for the probability that it will lead to success.

- **MapReduce**

MapReduce is a computing procedure for working with large datasets, which was devised due to difficulty of reading and analysing really Big Data using conventional computing methodologies. As its name suggest, it consists of two procedures – mapping (sorting information into the format needed for analysis – i.e. sorting a list of people according to their age) and reducing (performing an operation, such checking the age of everyone in the dataset to see who is over 21).

- **NoSQL**

NoSQL refers to a database format designed to hold more than data which is simply arranged into tables, rows, and columns, as is the case in a conventional relational database. This database format has proven very popular in Big Data applications because Big Data is often messy, unstructured and does not easily fit into traditional database frameworks.

- **Python**

Python is a programming language which has become very popular in the Big Data space due to its ability to work very well with large, unstructured datasets (see Part II for the difference between structured and unstructured data). It is considered to be easier to learn for a data science beginner than other languages such as R (see also Part II) and more flexible.

- **R Programming**

R is another programming language commonly used in Big Data, and can be thought of as more specialized than Python, being geared towards statistics. Its strength lies in its powerful handling of structured data. Like Python, it has an active community of users who are constantly expanding

and adding to its capabilities by creating new libraries and extensions.

- **Recommendation engine**

A recommendation engine is basically an algorithm, or collection of algorithms, designed to match an entity (for example, a customer) with something they are looking for. Recommendation engines used by the likes of Netflix or Amazon heavily rely on Big Data technology to gain an overview of their customers and, using predictive modelling, match them with products to buy or content to consume. The economic incentives offered by recommendation engines has been a driving force behind a lot of commercial Big Data initiatives and developments over the last decade.

- **Real-time**

Real-time means "as it happens" and in Big Data refers to a system or process which is able to give data-driven insights based on what is happening at the present moment. Recent years have seen a large push for the development of systems capable of processing and offering insights in real-time (or near-real-time), and advances in computing power as well as development of techniques such as machine learning have made it a reality in many applications today.

- **Reporting**

The crucial "last step" of many Big Data initiative involves getting the right information to the people who need it to make decisions, at the right time. When this step is automated, analytics is applied to the insights themselves to ensure that they are communicated in a way that they will be understood and easy to act on. This will usually involve creating multiple reports based on the same data or insights but each intended for a different audience (for example, in-depth technical analysis for engineers, and an overview of the impact on the bottom line for c-level executives).

- **Spark**

Spark is another open source framework like Hadoop but more recently developed and more suited to handling cutting-edge Big Data tasks involving real time analytics and machine learning. Unlike Hadoop it does not include its own file system, though it is designed to work with Hadoop's HDFS or a number of other options. However, for certain data related processes it is able to calculate at over 100 times the speed of Hadoop, thanks to its in-memory processing capability. This means it is becoming an increasingly popular choice for projects involving deep learning, neural networks and other compute-intensive tasks.

- **Structured Data**

Structured data is simply data that can be arranged neatly into charts and tables consisting of rows,

columns or multi-dimensioned matrixes. This is traditionally the way that computers have stored data, and information in this format can easily and simply be processed and mined for insights. Data gathered from machines is often a good example of structured data, where various data points – speed, temperature, rate of failure, RPM etc. – can be neatly recorded and tabulated for analysis.

- **Unstructured Data**

Unstructured data is any data which cannot easily be put into conventional charts and tables. This can include video data, pictures, recorded sounds, text written in human languages and a great deal more. This data has traditionally been far harder to draw insight from using computers which were generally designed to read and analyze structured information. However, since it has become apparent that a huge amount of value can be locked away in this unstructured data, great efforts have been made to create applications which are capable of understanding unstructured data – for example visual recognition and natural language processing.

- **Visualization**

Humans find it very hard to understand and draw insights from large amounts of text or numerical data – we can do it, but it takes time, and our concentration and attention is limited. For this reason effort has been made to develop computer applications capable of rendering information in a visual form – charts and graphics which highlight the most important insights which have resulted from our Big Data projects. A subfield of reporting (see above), visualizing is now often an automated process, with visualizations customized by algorithm to be understandable to the people who need to act or take decisions based on them.

Basic availability, Soft state and Eventual consistency

Basic availability implies continuous system availability despite network failures **and** tolerance to temporary in**consistency**.

Soft state refers to **state** change without input which is required for **eventual consistency**.

Eventual consistency means that if no further updates are made to a given updated data**base** item for long enough period of time , all users will see the same value for the updated item.

Top Analytics Tools

*** R** is a language for statistical computing and graphics. It also used for big data analysis. It provides a wide variety of statistical tests.

Features:

- Effective data handling and storage facility,
- It provides a suite of operators for calculations on arrays, in particular, matrices,
- It provides coherent, integrated collection of big data tools for data analysis
- It provides graphical facilities for data analysis which display either on-screen or on hardcopy

*** Apache Spark** is a powerful open source big data analytics tool. It offers over 80 high-level operators that make it easy to build parallel apps. It is used at a wide range of organizations to process large datasets.

Features:

- It helps to run an application in Hadoop cluster, up to 100 times faster in memory, and ten times faster on disk
- It offers lighting Fast Processing
- Support for Sophisticated Analytics
- Ability to Integrate with Hadoop and Existing Hadoop Data

*** Plotly** is an analytics tool that lets users create charts and dashboards to share online.

Features:

- Easily turn any data into eye-catching and informative graphics
- It provides audited industries with fine-grained information on data provenance
- Plotly offers unlimited public file hosting through its free community plan

*** Lumify** is a big data fusion, analysis, and visualization platform. It helps users to discover connections and explore relationships in their data via a suite of analytic options.

Features:

- It provides both 2D and 3D graph visualizations with a variety of automatic layouts

- It provides a variety of options for analyzing the links between entities on the graph
- It comes with specific ingest processing and interface elements for textual content, images, and videos
- It spaces feature allows you to organize work into a set of projects, or workspaces
- It is built on proven, scalable big data technologies

*** IBM SPSS Modeler** is a predictive big data analytics platform. It offers predictive models and delivers to individuals, groups, systems and the enterprise. It has a range of advanced algorithms and analysis techniques.

Features:

- Discover insights and solve problems faster by analyzing structured and unstructured data
- Use an intuitive interface for everyone to learn
- You can select from on-premises, cloud and hybrid deployment options
- Quickly choose the best performing algorithm based on model performance

*** MongoDB** is a NoSQL, document-oriented database written in C, C++, and JavaScript. It is free to use and is an open source tool that supports multiple operating systems including Windows Vista (and later versions), OS X (10.7 and later versions), Linux, Solaris, and FreeBSD.

Its main features include Aggregation, Adhoc-queries, Uses BSON format, Sharding, Indexing, Replication, Server-side execution of javascript, Schemaless, Capped collection, MongoDB management service (MMS), load balancing and file storage.

Features:

- Easy to learn.
- Provides support for multiple technologies and platforms.
- No hiccups in installation and maintenance.
- Reliable and low cost.

NoSQL

NoSQL is a non-relational DMS, that does not require a fixed schema, avoids joins, and is easy to scale. NoSQL database is used for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example companies like Twitter, Facebook, Google that collect terabytes of user data every single day.

SQL

Structured Query language (SQL) **pronounced as "S-Q-L" or sometimes as "See-Quel**" is the standard language for dealing with Relational Databases. A relational database defines relationships in the form of tables.

SQL programming can be effectively used to insert, search, update, delete database records.

Comparison of SQL and NoSQL

Parameter	**SQL**	**NOSQL**
Definition	SQL databases are primarily called RDBMS or Relational Databases	NoSQL databases are primarily called as Non-relational or distributed database
Design for	Traditional RDBMS uses SQL syntax and queries to analyze and get the data for further insights. They are used for OLAP systems.	NoSQL database system consists of various kind of database technologies. These databases were developed in response to the demands presented for the development of the modern application.
Query Language	Structured query language (SQL)	No declarative query language
Type	SQL databases are table based databases	NoSQL databases can be document based, key-value pairs, graph databases
Schema	SQL databases have a predefined schema	NoSQL databases use dynamic schema for unstructured data.
Ability to scale	SQL databases are vertically scalable	NoSQL databases are horizontally scalable
Examples	Oracle, Postgres, and MS-SQL.	MongoDB, Redis, , Neo4j, Cassandra, Hbase.
Best suited for	An ideal choice for the complex query intensive environment.	It is not good fit complex queries.

Hierarchical data storage	SQL databases are not suitable for hierarchical data storage.	More suitable for the hierarchical data store as it supports key-value pair method.
Variations	One type with minor variations.	Many different types which include key-value stores, document databases, and graph databases.

Development Year	It was developed in the 1970s to deal with issues with flat file storage	Developed in the late 2000s to overcome issues and limitations of SQL databases.
Open-source	A mix of open-source like Postgres & MySQL, and commercial like Oracle Database.	Open-source
Consistency	It should be configured for strong consistency.	It depends on DBMS as some offers strong consistency like MongoDB, whereas others offer only offers eventual consistency, like Cassandra.
Best Used for	RDBMS database is the right option for solving ACID problems.	NoSQL is a best used for solving data availability problems
Importance	It should be used when data validity is super important	Use when it's more important to have fast data than correct data
Best option	When you need to support dynamic queries	Use when you need to scale based on changing requirements
Hardware	Specialized DB hardware (Oracle Exadata, etc.)	Commodity hardware
Network	Highly available network (Infiniband, Fabric Path, etc.)	Commodity network (Ethernet, etc.)
Storage Type	Highly Available Storage (SAN, RAID, etc.)	Commodity drives storage (standard HDDs, JBOD)
Best features	Cross-platform support, Secure and free	Easy to use, High performance, and Flexible tool.
Top Companies Using	Hootsuite, CircleCI, Gauges	Airbnb, Uber, Kickstarter

Average salary	The average salary for any professional SQL Developer is $84,328 per year in the U.S.A.	The average salary for "NoSQL developer" ranges from approximately $72,174 per year
ACID vs. BASE Model	ACID(Atomicity, Consistency, Isolation, and Durability) is a standard for RDBMS	Base (Basically Available, Soft state, Eventually Consistent) is a model of many NoSQL systems

RDBMS Versus Hadoop

Criteria	Hadoop	RDBMS
Schema	Based on 'Schema on Read'.	Based on 'Schema on Write'.
Data Type	Structured, Semi-Structured and Unstructured data.	Structured Data.
Speed	Writes are Fast.	Reads are Fast.
Cost	Open source framework, free of cost.	Licensed software, Paid.
Application	Data discovery, Storage and processing of Unstructured data.	OLTP and complex ACID transaction.

Distributed Computing Challenges

Designing a distributed system does not come as easy and straight forward. A number of challenges need to be overcome in order to get the ideal system. The major challenges in distributed systems are listed below:

1. Heterogeneity:

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity (that is, variety and difference) applies to all of the following:

- Hardware devices: computers, tablets, mobile phones, embedded devices, etc.
- Operating System: Ms Windows, Linux, Mac, Unix, etc.
- Network: Local network, the Internet, wireless network, satellite links, etc.
- Programming languages: Java, C/C++, Python, PHP, etc.
- Different roles of software developers, designers, system managers

Different programming languages use different representations for characters and data structures such as arrays and records. These differences must be addressed if programs written in different languages are to be able to communicate with one another. Programs written by different developers cannot communicate with one another unless they use common standards, for example, for network communication and the representation of primitive data items and data structures in messages. For this to happen, standards need to be agreed and adopted – as have the Internet protocols. **Middleware**: The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. Most middleware is implemented over the Internet protocols, which

themselves mask the differences of the underlying networks, but all middleware deals with the differences in operating systems and hardware

Heterogeneity and mobile code: The term mobile code is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.

2. Transparency:

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. In other words, distributed systems designers must hide the complexity of the systems as much as they can. Some terms of transparency in distributed systems are:

Access Hide differences in data representation and how a resource is accessed **Location** Hide where a resource is located **Migration** Hide that a resource may move to another location **Relocation** Hide that a resource may be moved to another location while in use **Replication** Hide that a resource may be copied in several places **Concurrency** Hide that a resource may be shared by several competitive users **Failure** Hide the failure and recovery of a resource **Persistence** Hide whether a (software) resource is in memory or a disk

3. Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs. If the well-defined interfaces for a system are published, it is easier for developers to add new features or replace sub-systems in the future. Example: Twitter and Facebook have API that allows developers to develop their own software interactively.

4. Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed

very frequently when it gets close to the deadline time. For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems.

5. Security

Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components: ***confidentiality*** (protection against disclosure to unauthorized individuals) ***integrity*** (protection against alteration or corruption), ***availability*** for the authorized (protection against interference with the means to access the resources).

6. Scalability

Distributed systems must be scalable as the number of user increases. The scalability is defined by B. Clifford Neumann as

A system is said to be scalable if it can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity

Scalability has 3 dimensions:

- Size
  - Number of users and resources to be processed. Problem associated is overloading
- Geography
  - Distance between users and resources. Problem associated is communication reliability
- Administration
  - As the size of distributed systems increases, many of the system needs to be controlled. Problem associated is administrative mess

7. Failure Handling

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation. The handling of failures is particularly difficult.

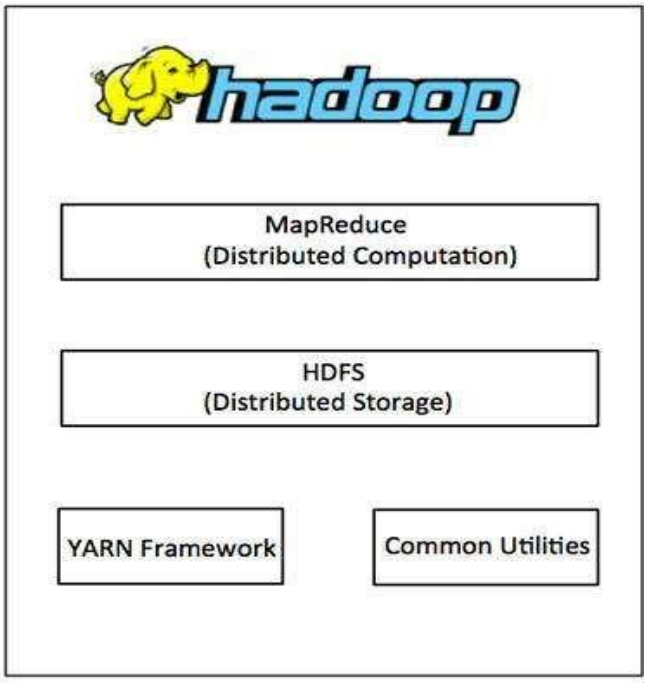
Hadoop Overview

Hadoop is an Apache open source framework written in java that allows distributed processing of large datasets across clusters of computers using simple programming models. The Hadoop framework application works in an environment that provides distributed *storage* and *computation* across clusters of computers. Hadoop is designed to scale up from single server to thousands of machines, each offering local computation and storage.

Hadoop Architecture

At its core, Hadoop has two major layers namely −

- Processing/Computation layer (MapReduce), and
- Storage layer (Hadoop Distributed File System).



MapReduce

MapReduce is a parallel programming model for writing distributed applications devised at Google for efficient processing of large amounts of data (multi-terabyte data-sets), on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. The MapReduce program runs on Hadoop which is an Apache open-source framework.

Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is based on the Google File System (GFS) and provides a distributed file system that is designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. It is highly fault-tolerant and is designed to be deployed on low-cost hardware. It provides high throughput access to application data and is suitable for applications having large datasets.

Apart from the above-mentioned two core components, Hadoop framework also includes the following two modules −

- **Hadoop Common** − These are Java libraries and utilities required by other Hadoop modules.
- **Hadoop YARN** − This is a framework for job scheduling and cluster resource management.

How Does Hadoop Work?

It is quite expensive to build bigger servers with heavy configurations that handle large scale processing, but as an alternative, you can tie together many commodity computers with single-CPU, as a single functional distributed system and practically, the clustered machines can read the dataset in parallel and provide a much higher throughput. Moreover, it is cheaper than one high-end server. So this is the first motivational factor behind using Hadoop that it runs across clustered and low-cost machines.

Hadoop runs code across a cluster of computers. This process includes the following core tasks that Hadoop performs −

- Data is initially divided into directories and files. Files are divided into uniform sized blocks of 128M and 64M (preferably 128M).
- These files are then distributed across various cluster nodes for further processing.
- HDFS, being on top of the local file system, supervises the processing.
- Blocks are replicated for handling hardware failure.
- Checking that the code was executed successfully.
- Performing the sort that takes place between the map and reduce stages.

- Sending the sorted data to a certain computer.
- Writing the debugging logs for each job.

Advantages of Hadoop

- Hadoop framework allows the user to quickly write and test distributed systems. It is efficient, and it automatic distributes the data and work across the machines and in turn, utilizes the underlying parallelism of the CPU cores.
- Hadoop does not rely on hardware to provide fault-tolerance and high availability (FTHA), rather Hadoop library itself has been designed to detect and handle failures at the application layer.
- Servers can be added or removed from the cluster dynamically and Hadoop continues to operate without interruption.
- Another big advantage of Hadoop is that apart from being open source, it is compatible on all the platforms since it is Java based.

Processing Data with Hadoop - Managing Resources and Applications with Hadoop YARN

Yarn divides the task on resource management and job scheduling/monitoring into separate daemons. There is one Resource Manager and per-application Application Master. An application can be either a job or a DAG of jobs.
The Resource Manger have two components – Scheduler and Application Manager.

The **scheduler** is a pure scheduler i.e. it does not track the status of running application. It only allocates resources to various competing applications. Also, it does not restart the job after failure due to hardware or application failure. The scheduler allocates the resources based on an abstract notion of a container. A container is nothing but a fraction of resources like CPU, memory, disk, network etc.
Following are the tasks of Application Manager: -

- Accepts submission of jobs by client.
- Negotiates first container for specific Application Master.
- Restarts the container after application failure.

Below are the responsibilities of Application Master

- Negotiates containers from Scheduler
- Tracking container status and monitoring its progress.

Yarn supports the concept of Resource Reservation via Reservation System. In this, a user can fix a number of resources for execution of a particular job over time and temporal constraints. The Reservation System makes sure that the resources are available to the job until its completion. It also performs admission control for reservation.

Yarn can scale beyond a few thousand nodes via Yarn Federation. YARN Federation allows to wire multiple sub-cluster into the single massive cluster. We can use many independent clusters together for a single large job. It can be used to achieve a large scale system.

Let us summarize how **Hadoop** works step by step:

- Input data is broken into blocks of size **128 Mb** and then blocks are moved to different nodes.
- Once all the blocks of the data are stored on data-nodes, the user can process the data.
- Resource Manager then schedules the program (submitted by the user) on individual nodes.
- Once all the nodes process the data, the output is written back to HDFS.

Interacting with Hadoop Ecosystem

Hadoop Ecosystem Hadoop has an ecosystem that has evolved from its three core components processing, resource management, and storage. In this topic, you will learn the components of the Hadoop ecosystem and how they perform their roles during Big Data processing. The
Hadoop ecosystem is continuously growing to meet the needs of Big Data. It comprises the following twelve components:

- HDFS(Hadoop Distributed file system)
- HBase
- Sqoop
- Flume
- Spark
- Hadoop MapReduce
- Pig
- Impala

- Hive
- Cloudera Search
- Oozie
- Hue.

Let us understand the role of each component of the Hadoop ecosystem.

Components of Hadoop Ecosystem

Let us start with the first component HDFS of Hadoop Ecosystem.

HDFS (HADOOP DISTRIBUTED FILE SYSTEM)

- HDFS is a storage layer for Hadoop.
- HDFS is suitable for distributed storage and processing, that is, while the data is being stored, it first gets distributed and then it is processed.
- HDFS provides Streaming access to file system data.
- HDFS provides file permission and authentication.
- HDFS uses a command line interface to interact with Hadoop.

So what stores data in HDFS? It is the HBase which stores data in HDFS.

HBase

- HBase is a NoSQL database or non-relational database.
- HBase is important and mainly used when you need random, real-time, read, or write access to your Big Data.
- It provides support to a high volume of data and high throughput.
- In an HBase, a table can have thousands of columns.

INTRODUCTION TO MONGODB AND MAPREDUCE PROGRAMMING

MongoDB is a cross-platform, document-oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Collection

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

The following table shows the relationship of RDBMS terminology with MongoDB.

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents

Primary Key	Primary Key (Default key _id provided by MongoDB itself)
Database Server and Client	
mysqld/Oracle	mongod
mysql/sqlplus	mongo

Sample Document

Following example shows the document structure of a blog site, which is simply a comma separated key value pair.

```
{
  _id:  ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by: 'tutorials point',
  url:  'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100,
  comments: [
    {
      user:'user1',
      message:  'My  first  comment',
      dateCreated:  new  Date(2011,1,20,2,15),
      like: 0
    },
```

```
    {
      user:'user2',
      message: 'My second comments',
      dateCreated: new Date(2011,1,25,7,45),
      like: 5
    }
  ]
}
```

_id is a 12 bytes' hexadecimal number which assures the uniqueness of every document. You can provide _id while inserting the document. If you don't provide then MongoDB provides a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental VALUE.

Any relational database has a typical schema design that shows number of tables and the relationship between these tables. While in MongoDB, there is no concept of relationship.

Advantages of MongoDB over RDBMS

- **Schema less** − MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- Structure of a single object is clear.
- No complex joins.
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Tuning.
- **Ease of scale-out** − MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.

- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

Why Use MongoDB?

- **Document Oriented Storage** − Data is stored in the form of JSON style documents.
- Index on any attribute
- Replication and high availability
- Auto-Sharding
- Rich queries
- Fast in-place updates
- Professional support by MongoDB

Where to Use MongoDB?

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

MongoDB supports many datatypes. Some of them are −

- **String** − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** − This type is used to store a boolean (true/ false) value.
- **Double** − This type is used to store floating point values.
- **Min/ Max keys** − This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** − This type is used to store arrays or list or multiple values into one key.
- **Timestamp** − ctimestamp. This can be handy for recording when a document has been

modified or added.

- **Object** − This datatype is used for embedded documents.
- **Null** − This type is used to store a Null value.
- **Symbol** − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** − This datatype is used to store the document's ID.
- **Binary data** − This datatype is used to store binary data.
- **Code** − This datatype is used to store JavaScript code into the document.
- **Regular expression** − This datatype is used to store regular expression.

The find() Method

To query data from MongoDB collection, you need to use MongoDB's **find()** method.

Syntax

The basic syntax of **find()** method is as follows −

```
>db.COLLECTION_NAME.find()
```

find() method will display all the documents in a non-structured way.

Example

Assume we have created a collection named mycol as −

```
> use sampleDB
switched to db sampleDB
> db.createCollection("mycol")
{ "ok" : 1 }
>
```

And inserted 3 documents in it using the insert() method as shown below −

```
> db.mycol.insert([
        {
                title: "MongoDB Overview",
                description: "MongoDB is no SQL database",
                by: "tutorials point",
                url: "http://www.tutorialspoint.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 100
        },
        {
                title: "NoSQL Database",
                description: "NoSQL database doesn't have tables",
                by: "tutorials point",
                url: "http://www.tutorialspoint.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 20,
                comments: [
```

```
                        {
                                user:"user1",
                                message: "My first comment",
                                dateCreated: new Date(2013,11,10,2,35),
                                like: 0
                        }
                ]
        }
])
```

Following method retrieves all the documents in the collection −

```
> db.mycol.find()
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534c"), "title" : "MongoDB Overview", "description" : "MongoDB is no SQL database", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 100 }
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534d"), "title" : "NoSQL Database", "description" : "NoSQL database doesn't have tables", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 20, "comments" : [ { "user" : "user1", "message" : "My first comment", "dateCreated" : ISODate("2013-12-09T21:05:00Z"), "like" : 0 } ] }
>
```

The pretty() Method

To display the results in a formatted way, you can use pretty() method.

Syntax

>db.COLLECTION_NAME.find().pretty()

Example

Following example retrieves all the documents from the collection named mycol and arranges them in an easy-to-read format.

```
> db.mycol.find().pretty()
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
```

```
}
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534d"),
        "title" : "NoSQL Database",
        "description" : "NoSQL database doesn't have tables",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 20,
        "comments" : [
                {
                        "user" : "user1",
                        "message" : "My first comment",
                        "dateCreated" : ISODate("2013-12-09T21:05:00Z"),
                        "like" : 0
                }
        ]
}
```

The findOne() method

Apart from the find() method, there is **findOne()** method, that returns only one document.

Syntax

>db.COLLECTIONNAME.findOne()

Example

Following example retrieves the document with title MongoDB Overview.

```
> db.mycol.findOne({title: "MongoDB Overview"})
{
        "_id" : ObjectId("5dd6542170fb13eec3963bf0"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
```

```
}
```

RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

Operation	**Syntax**	**Example**	**RDBMS Equivalent**
Equality	{<key>:{$eg;<value>}}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50

Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50
Values in an array	{<key>:{$in:[<value1>, <value2>,……<valueN>]}}	db.mycol.find({"name":{$in:["Raj", "Ram", "Raghu"]}}).pretty()	Where name matches any of the value in :["Raj", "Ram", "Raghu"]

Values not in an array	{<key>:{$nin:<value>}}	db.mycol.find({"name":{$nin:["Ramu", "Raghav"]}}).pretty()	Where name values is not in the array :["Ramu", "Raghav"] or, doesn't exist at all

AND in MongoDB

Syntax

To query documents based on the AND condition, you need to use $and keyword. Following is the basic syntax of AND −

>db.mycol.find({ $and: [{<key1>:<value1>}, { <key2>:<value2>}] })

Example

Following example will show all the tutorials written by 'tutorials point' and whose title is 'MongoDB Overview'.

```
> db.mycol.find({$and:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()
{
        "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
        "title" : "MongoDB Overview",
        "description" : "MongoDB is no SQL database",
        "by" : "tutorials point",
        "url" : "http://www.tutorialspoint.com",
        "tags" : [
                "mongodb",
                "database",
                "NoSQL"
        ],
        "likes" : 100
}
>
```

For the above given example, equivalent where clause will be **' where by = 'tutorials point' AND title = 'MongoDB Overview' '**. You can pass any number of key, value pairs in find clause.

ORin MongoDB

Syntax

To query documents based on the OR condition, you need to use **$or** keyword. Following is the basic syntax of **OR** −

```
>db.mycol.find(
  {
    $or: [
      {key1: value1}, {key2:value2}
    ]
  }
).pretty()
```

Example

Following example will show all the tutorials written by 'tutorials point' or whose title is 'MongoDB Overview'.

```
>db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()
{
  "_id": ObjectId(7df78ad8902c),
  "title": "MongoDB Overview",
  "description": "MongoDB is no sql database",
  "by": "tutorials point",
  "url": "http://www.tutorialspoint.com",
  "tags": ["mongodb", "database", "NoSQL"],
  "likes": "100"
}
>
```

Using AND and OR Together

Example

The following example will show the documents that have likes greater than 10 and whose title is either 'MongoDB Overview' or by is 'tutorials point'. Equivalent SQL where clause is **'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')'**

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},
  {"title": "MongoDB Overview"}]}).pretty()
{
  "_id": ObjectId(7df78ad8902c),
  "title": "MongoDB Overview",
  "description": "MongoDB is no sql database",
  "by": "tutorials point",
  "url": "http://www.tutorialspoint.com",
  "tags": ["mongodb", "database", "NoSQL"],
  "likes": "100"
}
>
```

NOR in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword. Following is the basic syntax of **NOT** −

```
>db.COLLECTION_NAME.find(
        {
                Exam    $not: [
                ple
                        ]
        }
)
```

{key1:
value1},
{key2:value2
}

Assume we have inserted 3 documents in the collection **empDetails** as shown below −

```
db.empDetails.insertMany(
        [
                {
                        First_Name: "Radhika",
                        Last_Name: "Sharma",
                        Age: "26",
                        e_mail: "radhika_sharma.123@gmail.com",
                        phone: "9000012345"
                },
                {
                        First_Name: "Rachel",
                        Last_Name: "Christopher",
                        Age: "27",
                        e_mail: "Rachel_Christopher.123@gmail.com",
                        phone: "9000054321"
                },
                {
                        First_Name: "Fathima",
                        Last_Name: "Sheik",
                        Age: "24",
                        e_mail: "Fathima_Sheik.123@gmail.com",
                        phone: "9000054321"
                }
        ]
)
```

Following example will retrieve the document(s) whose first name is not "Radhika" and last name is not "Christopher"

```
> db.empDetails.find(
        {
```

```
                $nor:[
                        40
                        {"First_Name": "Radhika"},
                        {"Last_Name": "Christopher"}
                ]
        }
).pretty()
{
        "_id" : ObjectId("5dd631f270fb13eec3963bef"),
        "First_Name" : "Fathima",
        "Last_Name" : "Sheik",
        "Age" : "24",
        "e_mail" : "Fathima_Sheik.123@gmail.com",
        "phone" : "9000054321"
}
```

NOT in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword following is the basic syntax of **NOT** −

```
>db.COLLECTION_NAME.find(
        {
                $NOT: [
                        {key1: value1}, {key2:value2}
                ]
}
).pretty()
```

Example

Following example will retrieve the document(s) whose age is not greater than 25

```
> db.empDetails.find( { "Age": { $not: { $gt: "25" } } } )
{
        "_id" : ObjectId("5dd6636870fb13eec3963bf7"),
        "First_Name" : "Fathima",
        "Last_Name" : "Sheik",
        "Age" : "24",
        "e_mail" : "Fathima_Sheik.123@gmail.com",
        "phone" : "9000054321"
}
```

MapReduce:

MapReduce addresses the challenges of distributed programming by providing an abstraction that isolates the developer from system-level details (e.g., locking of data structures, data starvation issues in the processing pipeline, etc.). The programming model specifies simple and well-defined interfaces between a small number of components, and therefore is easy for the programmer to reason about. MapReduce maintains a separation of what computations are to be performed and how those computations are actually carried out on a cluster of machines.

The first is under the control of the programmer, while the second is exclusively the responsibility of the execution framework or "runtime". The advantage is that the execution framework only needs to be designed once and verified for correctness—thereafter, as long as the developer expresses computations in the programming model, code is guaranteed to behave as expected. The upshot is that the developer is freed from having to worry about system-level details (e.g., no more debugging race conditions and addressing lock contention) and can instead focus on algorithm or application design.

MapReduce represents the first widely-adopted step away from the von Neumann model that has served as the foundation of computer science over the last half plus century. Valiant called this a bridging model, a conceptual bridge between the physical implementation of a machine and the software that is to be executed on that machine. Until recently, the von Neumann model has served us well: Hardware designers focused on efficient implementations of the von Neumann model and didn't have to think much about the actual software that would run on the machines.

Similarly, the software industry developed software targeted at the model without worrying about the hardware details. The result was extraordinary growth: chip designers churned out successive generations of increasingly powerful processors, and software engineers were able to develop applications in high-level languages that exploited those processors.

MapReduce can be viewed as the first breakthrough in the quest for new abstractions that allow us to organize computations, not over individual machines, but over entire clusters. As Barroso puts it, the datacenter is the computer. MapReduce is certainly not the first model of parallel computation that has been proposed. The most prevalent model in theoretical computer science, which dates back several decades, is the PRAM. MAPPERS AND REDUCERS Key-value pairs form the basic data structure in MapReduce. Keys and values may be primitives such as integers, floating point values, strings, and raw bytes, or they may be arbitrarily

complex structures (lists, tuples, associative arrays, etc.). Programmers typically need to define their own custom data types, although a number of libraries such as Protocol Buffers,5 Thrift,6 and Avro7 simplify the task. Part of the design of MapReduce algorithms involves imposing the key-value structure on arbitrary datasets. For a collection of web pages, keys may be URLs and values may be the actual HTML content. For a graph, keys may represent node ids and values may contain the adjacency lists of those nodes (see Chapter 5 for more details). In some algorithms, input keys are not particularly

meaningful and are simply ignored during processing, while in other cases input keys are used to uniquely identify a datum (such as a record id). In Chapter 3, we discuss the role of complex keys and values in the design of various algorithms. In MapReduce, the programmer defines a mapper and a reducer with the following signatures: map: (k1, v1) → [(k2, v2)] reduce: (k2, [v2]) → [(k3, v3)] The convention [. . .] is used throughout this book to denote a list.

The input to a MapReduce job starts as data stored on the underlying distributed file system (see Section 2.5). The mapper is applied to every input key-value pair (split across an arbitrary number of files) to generate an arbitrary number of intermediate key-value pairs. The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs.8 Implicit between the map and reduce phases is a distributed "group by" operation on intermediate keys. Intermediate data arrive at each reducer in order, sorted by the key. However, no ordering relationship is guaranteed for keys across different reducers. Output key-value pairs from each reducer are written persistently back onto the distributed file system (whereas intermediate key-value pairs are transient and not preserved). The output ends up in r files on the distributed file system, where r is the number of reducers. For the most part, there is no need to consolidate reducer output, since the r files often serve as input to yet another MapReduce job. Figure 2.2 illustrates this two-stage processing structure. A simple word count algorithm in MapReduce is shown in Figure 2.3. This algorithm counts the number of

occurrences of every word in a text collection, which may be the first step in, for example,

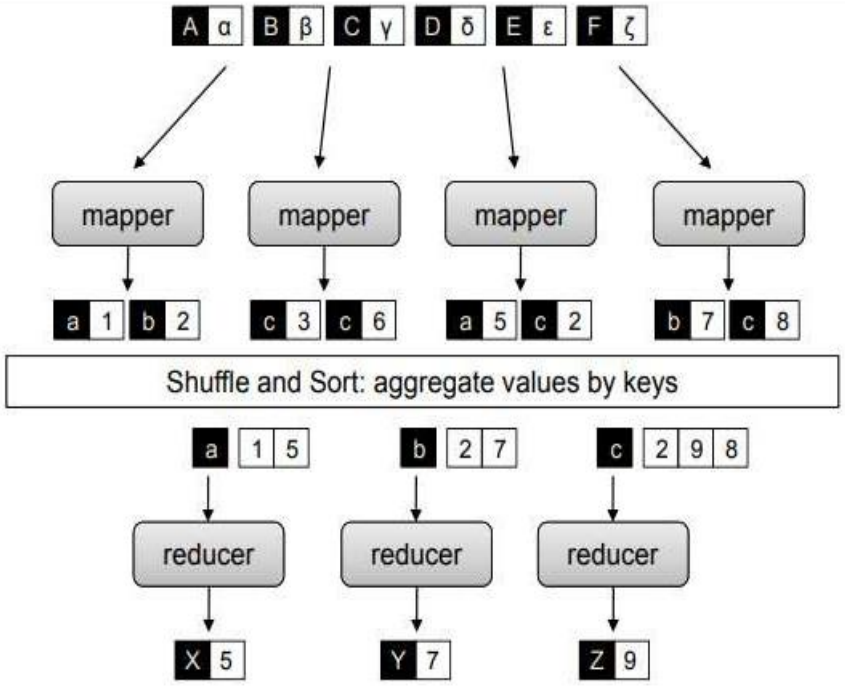


Figure 2.2: Simplified view of MapReduce. Mappers are applied to all input key-value pairs, which generate an arbitrary number of intermediate key-value pairs. Reducers are applied to all values associated with the same key. Between the map and reduce phases lies a barrier that involves a large distributed sort and group by.

building a unigram language model (i.e., probability

MAPREDUCE BASICS

Distribution over words in a collection). Input key-values pairs take the form of (docid, doc) pairs stored on the distributed file system, where the former is a unique identifier for the document, and the latter is the text of the document itself. The mapper takes an input key-value pair, tokenizes the document, and emits an intermediate key-value pair for every word: the word itself serves as the key, and the integer one serves as the value (denoting that we've seen the word once).

The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Therefore, in our word count algorithm, we simply need to sum up all counts (ones) associated with each word. The reducer does exactly this, and emits final keyvalue pairs with the word as the key, and the count as the value. Final output is written to the distributed file system, one file per reducer. Words within each file will be sorted

by alphabetical order, and each file will contain roughly the same number of words. The partitioner, which we discuss later in Section 2.4, controls the assignment of words to reducers. The output can be examined by the programmer or used as input to another MapReduce program.

There are some differences between the Hadoop implementation of MapReduce and Google's implementation.9 In Hadoop, the reducer is presented with a key and an iterator over all values associated with the particular key. The values are arbitrarily ordered. Google's implementation allows the programmer to specify a secondary sort key for ordering the values (if desired)—in which case values associated with each key would be presented to the developer's reduce code in sorted order. Later in Section 3.4 we discuss how to overcome this limitation in Hadoop to perform secondary sorting. Another difference: in Google's implementation the programmer is not allowed to change the key in the reducer. That is, the reducer output key must be exactly the same as the reducer input key. In Hadoop, there is no such restriction, and the reducer can emit an arbitrary number of output key-value pairs (with different keys).

To provide a bit more implementation detail: pseudo-code provided in this book roughly mirrors how MapReduce programs are written in Hadoop. Mappers and reducers are objects that implement the Map and Reduce methods, respectively. In Hadoop, a mapper object is initialized for each map task (associated with a particular sequence of key-value pairs called an input split) and the Map method is called on each key-value pair by the execution framework. In configuring a MapReduce job, the programmer provides a hint on the number of map tasks to run, but the execution framework (see next section) makes the final determination based on the physical layout of the data The situation is similar for the reduce phase: a reducer object is initialized for each reduce task, and the Reduce method is called once per intermediate key. In contrast with the number of map tasks, the programmer can precisely specify the number of reduce tasks.

Mappers and reducers can express arbitrary computations over their inputs. However, one must generally be careful about use of external resources since multiple mappers or reducers may be contending for those resources. For example, it may be unwise for a mapper to query an external SQL database, since that would introduce a scalability bottleneck on the number of map tasks that could be run in parallel (since they might all be simultaneously querying the database).10 In general, mappers can emit an arbitrary number of intermediate key-value pairs, and they need not be of the same type as the input key-value pairs. Similarly, reducers can emit an arbitrary number of final key-value pairs, and they can differ in type from the intermediate key-value pairs. Although not permitted in functional programming, mappers and reducers can

have side effects. This is a powerful and useful feature: Such algorithms can be understood as having side effects that only change state that is internal to the mapper or reducer. While the correctness of such algorithms may be more difficult to guarantee (since the function's behavior depends not only on the current input but on previous inputs), most potential synchronization problems are avoided since internal state is private only to individual mappers and reducers.

It may be useful for mappers or reducers to have external side effects, such as writing files to the distributed file system. Since many mappers and reducers are run in parallel, and the distributed file system is a shared global resource, special care must be taken to ensure that such operations avoid synchronization conflicts. One strategy is to write a temporary file that is renamed upon successful completion of the mapper or reducer.

In addition to the "canonical" MapReduce processing flow, other variations are also possible. MapReduce programs can contain no reducers, in which case mapper output is directly written to disk (one file per mapper). For embarrassingly parallel problems, e.g., parse a large text collection or independently analyze a large number of images, this would be a common pattern. The converse—a MapReduce program with no mappers—is not possible, although in some cases it is useful for the mapper to implement the identity function and simply pass input key-value pairs to the reducers. This has the effect of sorting and regrouping the input for reduce-side processing. Similarly, in some cases it is useful for the reducer to implement the identity function, in which case the program simply sorts and groups mapper output. Finally, running identity mappers and reducers has the effect of regrouping and resorting the input data (which is sometimes useful).

Although in the most common case, input to a MapReduce job comes from data stored on the distributed file system and output is written back to the distributed file system, any other system that satisfies the proper abstractions can serve as a data source or sink. With Google's MapReduce implementation, BigTable,a sparse, distributed, persistent multidimensional sorted map, is frequently used as a source of input and as a store of MapReduce output. HBase is an open-source BigTable clone and has similar capabilities. Also, Hadoop has been integrated with existing MPP (massively parallel processing) relational databases, which allows a programmer to write MapReduce jobs over database rows and dump output into a new database table. Finally, in some

cases MapReduce jobs may not consume any input at all (e.g., computing π) or may only consume a small amount of data (e.g., input parameters to many instances of processor intensive simulations running in parallel).

PARTITIONERS AND COMBINERS

We have thus far presented a simplified view of MapReduce. There are two additional elements that complete the programming model: partitioners and combiners. Partitioners are responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. In other words, the partitioner specifies the task to which an intermediate key-value pair must be copied. Within each reducer, keys are processed in sorted order (which is how the "group by" is implemented). The simplest partitioner involves computing the hash value of the key and then taking the mod of that value with the number of reducers. This assigns approximately the same number of keys to each reducer (dependent on the quality of the hash function). Note, however, that the partitioner only considers the key and ignores the value—therefore, a roughly-even partitioning of the key space may nevertheless yield large differences in the number of key-values pairs sent to each reducer (since different keys may have different numbers of associated values). This imbalance in the amount of data associated with each key is relatively common in many text processing applications due to the Zipfian distribution of word occurrences.

Combiners are an optimization in MapReduce that allow for local aggregation before the shuffle and sort phase. Furthermore, all these key-value pairs need to be copied across the network, and so the amount of intermediate data will be larger than the input collection itself. This is clearly inefficient. One solution is to perform local aggregation on the output of each mapper, i.e., to compute a local count for a word over all the documents processed by the mapper. With this modification (assuming the maximum amount of local aggregation possible), the number of intermediate key-value pairs will be at most the number of unique words in the collection times the number of mappers (and typically far smaller because each mapper may not encounter every word).

smaller because each mapper may not encounter every word). The combiner in MapReduce supports such an optimization. One can think of combiners as "mini-reducers" that take place on the output of the mappers, prior to the shuffle and sort phase. Each combiner operates in isolation and therefore does not have access to intermediate output from other mappers. The combiner is provided keys and values associated with each key (the same types as the mapper output keys and values). Critically, one cannot assume that a combiner will have the opportunity to process all

values associated with the same key. The combiner can emit any number of key-value pairs, but the keys and values must be of the same type as the mapper output (same as the reducer input).12 In cases where an operation is both associative and commutative (e.g., addition or multiplication), reducers can directly serve as combiners. In general, however, reducers and combiners are not interchangeable.

In many cases, proper use of combiners can spell the difference between an impractical algorithm and an efficient algorithm. This topic will be discussed in Section 3.1, which focuses on various techniques for local aggregation. It suffices to say for now that a combiner can significantly reduce the amount of data that needs to be copied over the network, resulting in much faster algorithms. The complete MapReduce model is shown in Figure 2.4. Output of the mappers are processed by the combiners, which perform local aggregation to cut down on the number of intermediate key-value pairs. The partitioner determines which reducer will be responsible for processing a particular key, and the execution framework uses this information to copy the data to the right location during the shuffle and sort phase.13 Therefore, a complete MapReduce job consists of code for the mapper, reducer, combiner, and partitioner, along with job configuration parameters. The execution framework handles everything else.

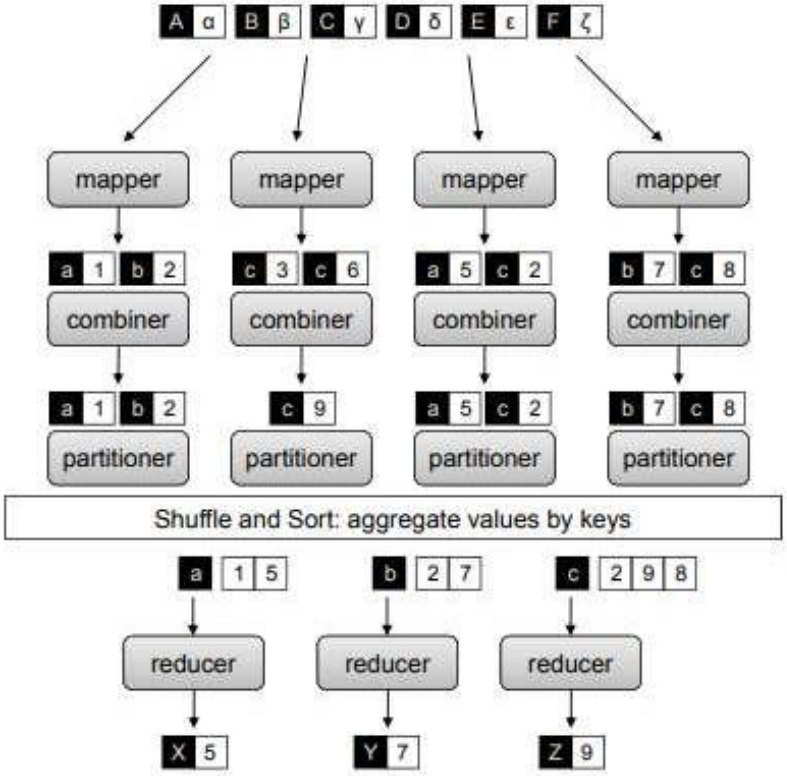


Figure 2.4: Complete view of MapReduce, illustrating combiners and partitioners in addition to mappers and reducers. Combiners can be viewed as "mini-reducers" in the map phase. Partitioners determine which reducer is responsible for a particular key.

SECONDARY SORTING

MapReduce sorts intermediate key-value pairs by the keys during the shuffle and sort phase, which is very convenient if computations inside the reducer rely on sort order (e.g., the order inversion design pattern described in the previous section). However, what if in addition to sorting by key, we also need to sort by value? Google's MapReduce implementation provides built-in functionality for (optional) secondary sorting, which guarantees that values arrive in sorted order. Hadoop, unfortunately, does not have this capability built in.

Consider the example of sensor data from a scientific experiment: there are m sensors each taking readings on continuous basis, where m is potentially a large number. A dump of the sensor data might look something like the following, where rx after each timestamp represents the actual sensor readings (unimportant for this discussion, but may be a series of values, one or more complex records, or even raw bytes of images).

(t1, m1, r80521)
(t1, m2, r14209)
(t1, m3, r76042) ...
(t2, m1, r21823)
(t2, m2, r66508)
(t2, m3, r98347)

Suppose we wish to reconstruct the activity at each individual sensor over time. A MapReduce program to accomplish this might map over the raw data and emit the sensor id as the intermediate key, with the rest of each record as the value:

$m1 \rightarrow (t1, r80521)$

This would bring all readings from the same sensor together in the reducer. However, since MapReduce makes no guarantees about the ordering of values associated with the same key, the sensor readings will not likely be in temporal order. The most obvious solution is to buffer all the readings in memory and then sort by timestamp before additional processing. However, it should be apparent by now that any in-memory buffering of data introduces a potential scalability bottleneck. What if we are working with a high frequency sensor or sensor readings over a long period of time? What if the sensor readings themselves are large complex objects? This approach may not scale in these cases—the reducer would run out of memory trying to buffer all values associated with the same key.

This is a common problem, since in many applications we wish to first group together data one way (e.g., by sensor id), and then sort within the groupings another way (e.g., by time). Fortunately, there is a general purpose solution, which we call the "value-to-key conversion" design pattern. The basic idea is to move part of the value into the intermediate key to form a composite key, and let the MapReduce execution framework handle the sorting. In the above example, instead of emitting the sensor id as the key, we would emit the sensor id and the timestamp as a composite key: $(m1, t1) \rightarrow (r80521)$

The sensor reading itself now occupies the value. We must define the intermediate key sort order to first sort by the sensor id (the left element in the pair) and then by the timestamp (the right element in the pair). We must also implement a custom partitioner so that all pairs associated with the same sensor are shuffled to the same reducer. Properly orchestrated, the key-value pairs will

be presented to the reducer in the correct sorted order: (m1, t1) → [(r80521)] (m1, t2) → [(r21823)] (m1, t3) → [(r146925)] . . .

However, note that sensor readings are now split across multiple keys. The reducer will need to preserve state and keep track of when readings associated with the current sensor end and the next sensor begin.9 The basic tradeoff between the two approaches discussed above (buffer and inmemory sort vs. value-to-key conversion) is where sorting is performed. One can explicitly implement secondary sorting in the reducer, which is likely to be faster but suffers from a scalability bottleneck.10 With value-to-key conversion, sorting is offloaded to the MapReduce execution framework. Note that this approach can be arbitrarily extended to tertiary, quaternary, etc. sorting. This pattern results in many more keys for the framework to sort, but distributed sorting is a task that the MapReduce runtime excels at since it lies at the heart of the programming model.

INDEX COMPRESSION

We return to the question of how postings are actually compressed and stored on disk. This chapter devotes a substantial amount of space to this topic because index compression is one of the main differences between a "toy" indexer and one that works on real-world collections. Otherwise, MapReduce inverted indexing algorithms are pretty straightforward.

Let us consider the canonical case where each posting consists of a document id and the term frequency. A na¨ıve implementation might represent the first as a 32-bit integer9 and the second as a 16-bit integer. Thus, a postings list might be encoded as follows: [(5, 2),(7, 3),(12, 1),(49, 1),(51, 2), . . .]

where each posting is represented by a pair in parentheses. Note that all brackets, parentheses, and commas are only included to enhance readability; in reality the postings would be represented as a long stream of integers. This na¨ıve implementation would require six bytes per posting. Using this scheme, the entire inverted index would be about as large as the collection itself. Fortunately, we can do significantly better. The first trick is to encode differences between document ids as opposed to the document ids themselves. Since the postings are sorted by document ids, the differences (called d-gaps) must be positive integers greater than zero. The above postings list, represented with d-gaps, would be: [(5, 2),(2, 3),(5, 1),(37, 1),(2, 2)

Of course, we must actually encode the first document id. We haven't lost any information, since

the original document ids can be easily reconstructed from the d-gaps. However, it's not obvious that we've reduced the space requirements either, since the largest possible d-gap is one less than the number of documents in the collection. This is where the second trick comes in, which is to represent the d-gaps in a way such that it takes less space for smaller numbers. Similarly, we want to apply the same techniques to compress the term frequencies, since for the most part they are also small values. But to understand how this is done, we need to take a slight detour into compression techniques, particularly for coding integers.

Compression, in general, can be characterized as either lossless or lossy: it's fairly obvious that loseless compression is required in this context. To start, it is important to understand that all compression techniques represent a time–space tradeoff. That is, we reduce the amount of space on disk necessary to store data, but at the cost of extra processor cycles that must be spent coding and decoding data. Therefore, it is possible that compression reduces size but also slows processing. However, if the two factors are properly balanced (i.e., decoding speed can keep up with disk bandwidth), we can achieve the best of both worlds: smaller and faster.

POSTINGS COMPRESSION

Having completed our slight detour into integer compression techniques, we can now return to the scalable inverted indexing algorithm shown in Figure 4.4 and discuss how postings lists can be properly compressed. As we can see from the previous section, there is a wide range of choices that represent different tradeoffs between compression ratio and decoding speed. Actual performance also depends on characteristics of the collection, which, among other factors, determine the distribution of d-gaps. B¨uttcher et al. [30] recently compared the performance of various compression techniques on coding document ids. In terms of the amount of compression that can be obtained (measured in bits per docid), Golomb and Rice codes performed the best, followed by γ codes, Simple-9, varInt, and group varInt (the least space efficient). In terms of raw decoding speed, the order was almost the reverse: group varInt was the fastest, followed by varInt.14 Simple-9 was substantially slower, and the bit-aligned codes were even slower than that. Within the bit-aligned codes, Rice codes were the fastest, followed by γ, with Golomb codes being the slowest (about ten times slower than group varInt).

Let us discuss what modifications are necessary to our inverted indexing algorithm if we were to adopt Golomb compression for d-gaps and represent term frequencies with γ codes. Notethat this represents a space-efficient encoding, at the cost of slower decoding compared to alternatives.

Whether or not this is actually a worthwhile tradeoff in practice is not important here: use of Golomb codes serves a pedagogical purpose, to illustrate how one might set compression parameters.

Coding term frequencies with γ codes is easy since they are parameterless. Compressing d-gaps with Golomb codes, however, is a bit tricky, since two parameters are required: the size of the document collection and the number of postings for a particular postings list (i.e., the document frequency, or df). The first is easy to obtain and can be passed into the reducer as a constant. The df of a term, however, is not known until all the postings have been processed—and unfortunately,

the parameter must be known before any posting is coded. At first glance, this seems like a chicken-and-egg problem. A two-pass solution that involves first buffering the postings (in memory) would suffer from the memory bottleneck we've been trying to avoid in the first place.

To get around this problem, we need to somehow inform the reducer of a term's df before any of its postings arrive. This can be solved with the order inversion design pattern introduced in Section 3.3 to compute relative frequencies. The solution is to have the mapper emit special keys of the form ht, ∗i to communicate partial document frequencies. That is, inside the mapper, in addition to emitting intermediate key-value pairs of the following form:

(tuple ht, docidi,tf f)

we also emit special intermediate key-value pairs like this:

(tuple ht, ∗i, df e)

to keep track of document frequencies associated with each term. In practice, we can accomplish this by applying the in-mapper combining design pattern (see Section 3.1). The mapper holds an in-memory associative array that keeps track of how many documents a term has been observed in (i.e., the local document frequency of the term for the subset of documents processed by the mapper). Once the mapper has processed all input records, special keys of the form ht, ∗i are emitted with the partial df as the value.

To ensure that these special keys arrive first, we define the sort order of the tuple so that the special symbol ∗ precedes all documents (part of the order inversion design pattern). Thus, for each term,

the reducer will first encounter the ht, ∗i key, associated with a list of values representing partial df values originating from each mapper. Summing all these partial contributions will yield the term's df, which can then be used to set the Golomb compression parameter b. This allows the postings to be incrementally compressed as they are encountered in the reducer—memory bottlenecks are eliminated since we do not need to buffer postings in memory.

Once again, the order inversion design pattern comes to the rescue. Recall that the pattern is useful when a reducer needs to access the result of a computation (e.g., an aggregate statistic) before it encounters the data necessary to produce that computation. For computing relative frequencies, that bit of information was the marginal. In this case, it's the document frequency.

PARALLEL BREADTH-FIRST SEARCH

One of the most common and well-studied problems in graph theory is the single-source shortest path problem, where the task is to find shortest paths from a source node to all other nodes in the graph (or alternatively, edges can be associated with costs or weights, in which case the task is to compute lowest-cost or lowest-weight paths). Such problems are a staple in undergraduate algorithm courses, where students are taught the solution using Dijkstra's algorithm. However, this famous algorithm assumes sequential processing—how would we solve this problem in parallel, and more specifically, with MapReduce?

Dijkstra(G, w, s)
2: d[s] ← 0
3: for all vertex v ∈ V do
4: d[v] ← ∞
5: Q ← {V }
6: while Q 6= ∅ do
7: u ← ExtractMin(Q)
8: for all vertex v ∈ u.AdjacencyList do
9: if d[v] > d[u] + w(u, v) then
10: d[v] ← d[u] + w(u, v)

Pseudo-code for Dijkstra's algorithm, which is based on maintaining a global priority queue of nodes with priorities equal to their distances from the source node. At each iteration, the algorithm expands the node with the shortest distance and updates distances to all reachable nodes. As a

refresher and also to serve as a point of comparison, Dijkstra's algorithm is shown in Figure 5.2, adapted from Cormen, Leiserson, and Rivest's classic algorithms textbook [41] (often simply known as CLR). The input to the algorithm is a directed, connected graph G = (V, E) represented with adjacency lists, w containing edge distances such that $w(u, v) \geq 0$, and the source node s. The algorithm begins by first setting distances to all vertices d[v], $v \in V$ to ∞, except for the source node, whose distance to itself is zero.

The algorithm maintains Q, a global priority queue of vertices with priorities equal to their distance values d Dijkstra's algorithm operates by iteratively selecting the node with the lowest current distance from the priority queue (initially, this is the source node). At each iteration, the algorithm "expands" that node by traversing the adjacency list of the selected node to see if any of those nodes can be reached with a path of a shorter distance. The algorithm terminates when the priority queue Q is empty, or equivalently, when all nodes have been considered. Note that the algorithm as presented in Figure 5.2 only computes the shortest distances. The actual paths can be recovered by storing "backpointers" for every node indicating a fragment of the shortest path.

A sample trace of the algorithm running on a simple graph is shown in Figure 5.3 (example also adapted from CLR). We start out in (a) with n1 having a distance of zero (since it's the source) and all other nodes having a distance of ∞. In the first iteration (a), n1 is selected as the node to expand (indicated by the thicker border). After the expansion, we see in (b) that n2 and n3 can be reached at a distance of 10 and 5, respectively. Also, we see in (b) that n3 is the next node selected for expansion. Nodes we have already considered for expansion are shown in black. Expanding n3, we see in (c) that the distance to n2 has decreased because we've found a shorter path. The nodes that will be expanded next, in order, are n5, n2, and n4. The algorithm terminates with the end state shown in (f), where we've discovered the shortest distance to all nodes.

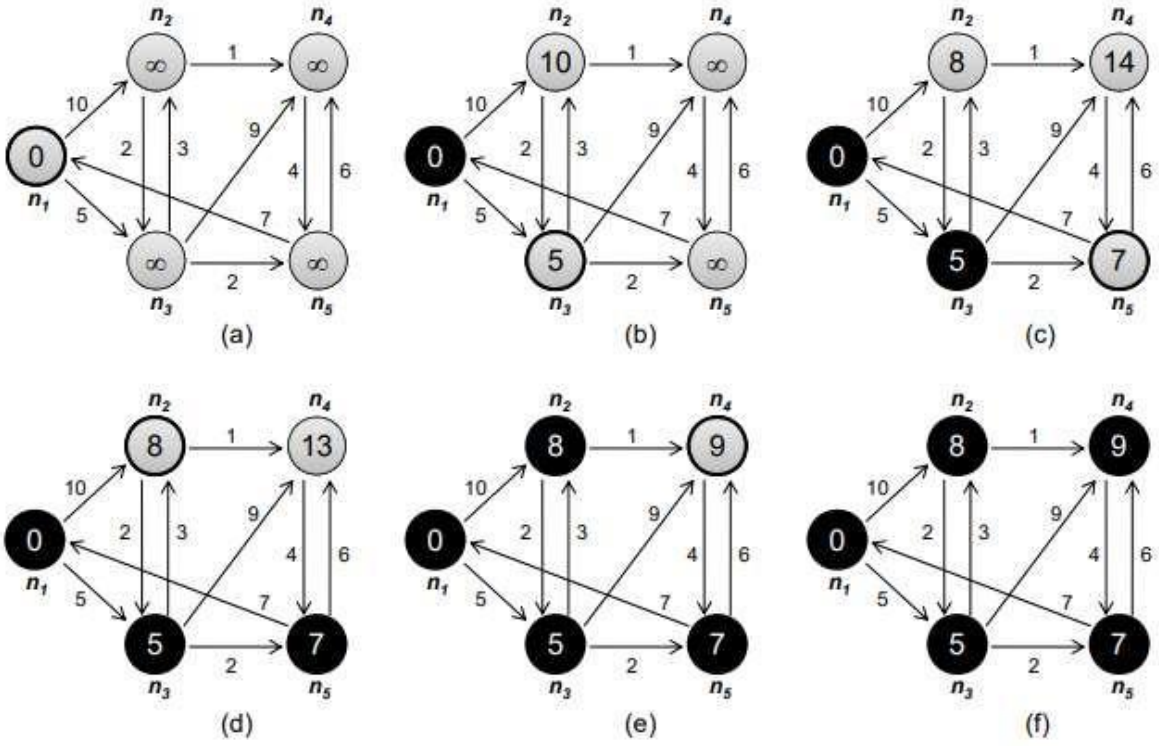


Figure 5.3: Example of Dijkstra's algorithm applied to a simple graph with five nodes, with n as the source and edge distances as indicated. Parts (a)–(e) show the running of the algorithm at each iteration, with the current distance inside the node. Nodes with thicker borders are those being expanded; nodes that have already been expanded are shown in black.

The key to Dijkstra's algorithm is the priority queue that maintains a globallysorted list of nodes by current distance. This is not possible in MapReduce, as the programming model does not provide a mechanism for exchanging global data. Instead, we adopt a brute force approach known as parallel breadth-first search. First, as a simplification let us assume that all edges have unit distance (modeling, for example, hyperlinks on the web). This makes the algorithm easier to understand, but we'll relax this restriction later.

The intuition behind the algorithm is this: the distance of all nodes connected directly to the source node is one; the distance of all nodes directly connected to those is two; and so on. Imagine water rippling away from a rock dropped into a pond— that's a good image of how parallel breadth-first search works. However, what if there are multiple paths to the same node? Suppose we wish to compute the shortest distance to node n. The shortest path must go through one of the nodes in M that contains an outgoing edge to n: we need to examine all m ∈ M to find ms, the node with the shortest distance. The shortest distance to n is the distance to ms plus one.

Pseudo-code for the implementation of the parallel breadth-first search algorithm is provided in Figure 5.4. As with Dijkstra's algorithm, we assume a connected, directed graph represented as

adjacency lists. Distance to each node is directly stored alongside the adjacency list of that node, and initialized to ∞ for all nodes except for the source node. In the pseudo-code, we use n to denote the node id (an integer) and N to denote the node's corresponding data structure (adjacency list and current distance). The algorithm works by mapping over all nodes and emitting a key-value pair for each neighbor on the node's adjacency list. The key contains the node id of the neighbor, and the value is the current distance to the node plus one. This says: if we can reach node n with a distance d, then we must be able to reach all the nodes that are connected to n with distance d + 1.

After shuffle and sort, reducers will receive keys corresponding to the destination node ids and distances corresponding to all paths leading to that node. The reducer will select the shortest of these distances and then update the distance in the node data structure.

h iteration corresponds to a MapReduce job. The first time we run the algorithm, we "discover" all nodes that are connected to the source. The second iteration, we discover all nodes connected to those, and so on. Each iteration of the algorithm expands the "search frontier" by one hop, and, eventually, all nodes will be discovered with their shortest distances (assuming a fully-connected graph). Before we discuss termination of the algorithm, there is one more detail required to make the parallel breadth-first search algorithm work. We need to "pass along" the graph structure from one iteration to the next. This is accomplished by emitting the node data structure itself, with the node id as a key (Figure 5.4, line 4 in the mapper). In the reducer, we must distinguish the node data structure from distance values (Figure 5.4, lines 5–6 in the reducer), and update the minimum distance in the node data structure before emitting it as the final value. The final output is now ready to serve as input to the next iteration.

So how many iterations are necessary to compute the shortest distance to all nodes? The answer is the diameter of the graph, or the greatest distance between any pair of nodes. This number is surprisingly small for many real-world problems: the saying "six degrees of separation" suggests that everyone on the planet is connected to everyone else by at most six steps (the people aperson knows are one step away, people that they know are two steps away, etc.). If this is indeed true, then parallel breadthfirst search on the global social network would take at most six MapReduce iterations.

class Mapper

2: method Map(nid n, node N) 3:

d ← N.Distance

4: Emit(nid n, N) . Pass along graph structure

```
5: for all nodeid m ∈ N.AdjacencyList do
6: Emit(nid m, d + 1) . Emit distances to reachable nodes
1: class Reducer
2: method Reduce(nid m, [d1, d2, . . .])
3: dmin ← ∞
4: M ← ∅
5: for all d ∈ counts [d1, d2, . . .] do
6: if IsNode(d) then
7: M ← d . Recover graph structure
8: else if d < dmin then . Look for shorter distance
9: dmin ← d
10: M.Distance ← dmin . Update shortest distance
11: Emit(nid m, node M)
```

Figure 5.4: Pseudo-code for parallel breath-first search in MapReduce: the mappers emit distances to reachable nodes, while the reducers select the minimum of those distances for each destination node. Each iteration (one MapReduce job) of the algorithm expands the "search frontier" by one hop.

For more serious academic studies of "small world" phenomena in networks, we refer the reader to a number of publications [61, 62, 152, 2]. In practical terms, we iterate the algorithm until there are no more node distances that are ∞. Since the graph is connected, all nodes are reachable, and since all edge distances are one, all discovered nodes are guaranteed to have the shortest distances (i.e., there is not a shorter path that goes through a node that hasn't been discovered).

The actual checking of the termination condition must occur outside of MapReduce. Typically, execution of an iterative MapReduce algorithm requires a nonMapReduce "driver" program, which submits a MapReduce job to iterate the algorithm, checks to see if a termination condition has been met, and if not, repeats. Hadoop provides a lightweight API for constructs called "counters", which, as the name suggests, can be used for counting events that occur during execution, e.g., number of corrupt records, number of times a certain condition is met, or anything that the programmer desires. Counters can be defined to count the number of nodes that have distances of ∞: at the end of the job, the driver program can access the final counter value and check to see if another iteration is necessary.

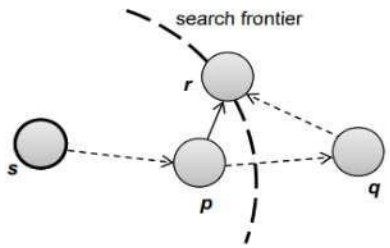


Figure 5.5: In the single source shortest path problem with arbitrary edge distances, the shortest path from source s to node r may go outside the current search frontier, in which case we will not find the shortest distance to r until the search frontier expands to cover q.

Finally, as with Dijkstra's algorithm in the form presented earlier, the parallel breadth-first search algorithm only finds the shortest distances, not the actual shortest paths. However, the path can be straightforwardly recovered. Storing "backpointers" at each node, as with Dijkstra's algorithm, will work, but may not be efficient since the graph needs to be traversed again to reconstruct the path segments. A simpler approach is to emit paths along with distances in the mapper, so that each node will have its shortest path easily accessible at all times. The additional space requirements for shuffling these data from mappers to reducers are relatively modest, since for the most part paths (i.e., sequence of node ids) are relatively short.

Up until now, we have been assuming that all edges are unit distance. Let us relax that restriction and see what changes are required in the parallel breadth-first search algorithm. The adjacency lists, which were previously lists of node ids, must now encode the edge distances as well.

The graph structure is represented with adjacency lists, which is part of some larger node data structure that may contain additional information (variables to store intermediate output, features of the nodes). In many cases, features are attached to edges as well (e.g., edge weights).

The graph structure is represented with adjacency lists, which is part of some larger node data structure that may contain additional information (variables to store intermediate output, features of the nodes). In many cases, features are attached to edges as well (e.g., edge weights).

In addition to computations, the graph itself is also passed from the mapper to the reducer. In the reducer, the data structure corresponding to each node is updated and written back to disk.

Graph algorithms in MapReduce are generally iterative, where the output of the previous iteration serves as input to the next iteration. The process is controlled by a non-MapReduce driver program that checks for termination.

For parallel breadth-first search, the mapper computation is the current distance plus edge distance (emitting distances to neighbors), while the reducer computation is the Min function

(selecting the shortest path). As we will see in the next section, the Map Reduce algorithm for PageRank works in much the same way