A TEXT BOOK OF

BIG DATA ANALYTICS

(PROGRAM ELECTIVE – VIII 702(A))

FOR
SEMESTER – VII
FINAL YEAR B. TECH COURSE IN
COMPUTER ENGINEERING

Strictly According to New Revised Credit System Syllabus
of Dr. Babasaheb Ambedkar Technological University (DBATU),
Lonere, (Dist. Raigad) Maharashtra,
(w.e.f. June 2020-21)

NITIN N. SAKHARE
M. E. (Comp. Networks)
Assistant Professor,
Computer Engineering Department
Vishwakarma Institute of Information Technology
Kondhwa (Bk.), PUNE

Price ₹ 240.00

NIRALI PRAKASHAN

N1156

SYLLABUS

Unit 1 : Introduction to Big Data **[6 Hrs]**

Why Big Data and Where did it come from?, Characteristics of Big, Challenges and applications of Big Data, Enabling Technologies for Big Data, Big Data Stack, Big Data distribution packages.

Unit 2 : Big Data Platforms **[7 Hrs]**

Overview of Apache Spark, HDFS, YARN, MapReduce, MapReduce Programming Model with Spark, MapReduce Example: Word Count, Page Rank etc, CAP Theorem, Eventual Consistency, Consistency Trade-O-s, ACID and BASE, Zookeeper and Paxos, Cassandra, Cassandra Internals, HBase, HBase Internals.

Unit 3 : Big Data Streaming Platforms **[6 Hrs]**

Big Data Streaming Platforms for Fast Data, Streaming Systems, Big Data Pipelines for Real-Time computing, Spark Streaming, Kafka, Streaming Ecosystem.

Unit 4 : Big Data Applications **[6 Hrs]**

Overview of Big Data Machine Learning, Mahout, Big Data Machine learning Algorithms in Mahout- kmeans, Naive Bayes etc. Machine learning with Spark, Machine Learning Algorithms in Spark, Spark MLlib, Deep Learning for Big Data, Graph Processing: Pregel, Giraph, Spark GraphX

Unit 5 : Database for the Modern Web **[7 Hrs]**

Introduction to mongoDB key features, Core server tools, MongoDB through the JavaScript' shell, Creating and querying through Indexes, Document-oriented, principles of schema design, Constructing queries on databases, collections and documents, MongoDBquery language

CONTENTS

UNIT I

INTRODUCTION TO BIG DATA

1.1 INTRODUCTION

- Now We are living in Big Data Era.
- Few years ago, Systems or Organizations or Applications were using all Structured Data only Structured Data means In the form of Rows and Columns). It was very easy to use Relational Data Bases (RDBMS) and old Tools to store, manage, process and report this Data.
- However recently, Nature of Data is changed. And Systems or Organizations or Applications are generating huge amount of Data in variety of formats at very fast rate.
- That means Data is not simple Structured Data(Not in the form of simple Rows and Columns). It does not have any proper format, just Raw Data without any format. It is "very difficult or not possible" to use Old Technologies, Traditional Relational Databases and Tools to store, manage, process and report this Data. Traditional Databases cannot Store, Process and Analysis this kind of Data.
- Then how to solve this problem? Here Big Data Solutions come into picture.
- Big Data Solutions solve all these problems very easily.
- Let us start with understanding What is Big Data and How important it is in our life.
- We don't have a straightforward definition to Big Data. However, we will try to answer this question in different ways.
- In Simple Words, Big Data is a technique to solve data problems that are not solvable using Traditional Databases and Tools.
- In other way, Big Data means not just huge amount of Data. Big Data means huge amount of data generating at very fast rate in different formats.
- Big Data is a Technique to "Store, Process, Manage, Analysis and Report" a huge amount of variety data, at the required speed, and within the required time to allow Real-time Analysis and Reaction.
- The term Big Data refers to all the data that is being generated across the globe at an unprecedented rate. This data could be either structured or unstructured. Today's business enterprises owe a huge part of their success to an economy that is firmly knowledge-oriented.
- Data drives the modern organizations of the world and hence making sense of this data and unraveling the various patterns and revealing unseen connections within the vast sea of data becomes critical and a hugely rewarding endeavor indeed.
- There is a need to convert Big Data into Business Intelligence that enterprises can readily deploy. Better data leads to better decision making and an improved way to strategize for organizations regardless of their size, geography, market share, customer segmentation and such other categorizations. Hadoop is the platform of choice for working with extremely large volumes of data.
- **Big Data** is a blanket term for the non-traditional strategies and technologies needed to gather, organize, process, and gather insights from large datasets. While the problem of working with data that exceeds the computing power or storage of a single computer is not new, the pervasiveness, scale, and value of this type of computing has greatly expanded in recent years.

Why Data is Important

- We are living in Data Era or Information Era. Data is most important factor for all Organizations for the following reasons or benefits:
  - ➢ Data is useful in Decision Making.
  - ➢ To know Customer Preferences so that Organizations can improve their Business.
  - ➢ Getting the Right Information for Business.
  - ➢ By analyzing Data, We can optimize our systems.
  - ➢ More Data, More Analysis, More Results, More Profits.

- ➢ Data is effective in improving Business Value
- ➢ Data Analysis provides Customer Likes and Dislikes information

1.2 WHY BIG DATA IS SO IMPORTANT

- Now-a-Days, Big data is very important for Organizations or Companies form Medium-Size to Large-Size, because it enables them to gather, store, manage, and manipulate "Extremely Large Amounts of Data, Extremely High Velocity of Data and Extremely Wide Variety of Data":
  - ➢ At the right speed
  - ➢ At the right time
  - ➢ To get the required Business Value
- By following this Big Data 4Vs Paradigm, we will get lot of benefits as shown below:

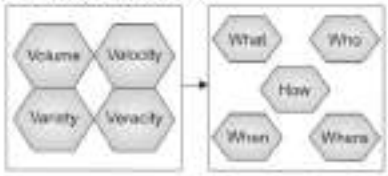


Fig. 1.1: BigData 4Vs – Business Value

- By using those Big Data 4Vs Paradigm, Organizations can get many befits by understanding "What, Who, When, Where, How" kind of questions:
  - ➢ What business decisions need to be made?
  - ➢ What insight can we derive from the information?
  - ➢ How accurate is that data in predicting business value?
  - ➢ Who could benefit from the information that we are capturing?
  - ➢ When do they need to know in order to make a more informed decision?
  - ➢ How to improve our business value?
  - ➢ How to improve our profits?
  - ➢ Where do we have more Profits?
- 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:
  - ➢ **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.
  - ➢ **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.
  - ➢ **Understand the Market Conditions** : By analyzing big data you can get a better understanding of current market conditions. For example, by analyzing customers' purchasing behaviors, a company can find out the products that are sold the most and produce products according to this trend. By this, it can get ahead of its competitors.
  - ➢ **Control Online Reputation:** Big data tools can do sentiment analysis. Therefore, you can get feedback about who is saying what about your company. If you want to monitor and improve the online presence of your business then, big data tools can help in all this.

Using Big Data Analytics to Boost Customer Acquisition and Retention

- The customer is the most important asset any business depends on. There is no single business that can claim success without first having to establish a solid customer base. 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.

Using Big Data Analytics to Solve Advertisers Problem and Offer Marketing Insights

- Big data analytics can help change all business operations. This includes the ability to match customer expectation, changing company's product line and of

course ensuring that the marketing campaigns are powerful.

Big Data Analytics As a Driver of Innovations and Product Development

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

Best Examples of Big Data

- The best examples of big data can be found both in the public and private sector. From targeted advertising, education, and already mentioned massive industries (healthcare, insurance, manufacturing or banking), to real-life scenarios, in guest service or entertainment. by the year 2020, 1.7 megabytes of data will be generated every second for every person on the planet, the potential for data-driven organizational growth in the hospitality sector is enormous.
- Big data can serve to deliver benefits in some surprising areas.

Big Data in Education Industry

- Following are some of the fields in education industry that have been transformed by big data motivated changes
  - ➢ Customized and Dynamic Learning Programs:
  - ➢ Reframing course material:
  - ➢ Grading Systems:
  - ➢ Career prediction:

Big Data in Insurance Industry

- The insurance industry holds importance not only for individuals but also business companies. The reason insurance holds a significant place is because it supports people during times of adversities and uncertainties. The data collected from these sources are of varying formats and change at tremendous speeds.

Collecting Information

- As big data refers to gathering data from disparate sources, this feature creates a crucial use case for the insurance industry to pounce on.

  Example: When a customer intends to buy a car insurance Kenya, the companies can obtain information from which they can calculate the safety levels for driving in the buyer's vicinity and his past driving records. On basis of this they can effectively calculate cost of car insurance as well.

Gaining Customer Insight

- Determining customer experience and making customers the center of a company's attraction is of prime importance to organizations.

Fraud Detection

- Insurance frauds are a common incidence. Big data use case for reducing fraud is highly effective.

Threat Mapping

- When an insurance agency sells an insurance, they want to be aware of all the possibilities of things going unfavorably with their customer, making them file a claim.

Big Data in Government Industry

- Along with many other areas, big data in government can have an enormous impact local, national and global. With so many complex issues on the table today, governments have their work cut out trying to make sense of all the information they receive and make vital decisions that affect millions of people.
- Governments, be it of any country, come face to face with a very huge amount of data on almost daily basis. Reason being, they have to keep track of various records and databases regarding the citizens. The proper study and analysis of this data helps the Governments in endless ways. Few of them are:

Welfare Schemes:

Cyber Security:

PDF - Big Data Applications in the Government Sector: A Comparative Analysis among Leading Countries

Big Data in Banking Sector

- The amount of data in banking sectors is skyrocketing every second. According to GDC prognosis, this data is estimated to grow 700% by 2020.

Study and Analysis of Big Data can Help Detect :

- The misuse of credit cards
- Misuse of debit cards
- Venture credit hazard treatment
- Business clarity
- Customer statistics alteration
- Money laundering
- Risk Mitigation

Real-Time Big Data Analytics Tools

- More and more tools offer the possibility of real-time processing of Big Data.

Storm

- Storm, which is now owned by Twitter, is a real-time distributed computation system.

Cloudera

- Cloudera offers the Cloudera Enterprise RTQ tools that offers real-time, interactive analytical queries of the data stored in HBase or HDFS.

GridGrain

- GridGrain is an enterprise open source grid computing made for Java. It is compatible with Hadoop DFS and it offers a substitute to Hadoop's MapReduce.

SpaceCurve

- The technology that SpaceCurve is developing can discover underlying patterns in multidimensional geodata.

Big Data: Data Formats

- In Big Data 3V Paradigm, one V refers to Variety. It means generating or getting data in different formats.
- In Data Era, We, Systems, Devices or Organizations are generating or getting the following types of Data Formats.

Structured Data

- Structured Data means Data that is in the form of Rows and Columns. So it is very easy to store even in Relational Databases.
- In Simple words, Anything which possible to store in the form of Rows and Columns that is Structured Data.

  For Example:- Relational DBs Data(Online Subscription, Transactional Data etc).

Empno	Empname	Sal	Deptno
1001	Abc	24500	10
1002	Xyz	45000	20
1003	Pqr	10000	30

Semi-Structured Data

- Semi-Structured Data means Data that is formatted in some way. But it is not formatted in the form of Rows and Columns. It is possible to store in Relational Databases, but bit complex to manage and provide very less performance.

For Example:

Log Files

- In Log Files, Columns are separated by using "Whitespace" characters (Which are characters used to align things either horizontally or vertically. For instance, space or Tab space, next line etc).

Observe the Following JBoss Server Log File:

> 09:20:01,054 INFO [org.jboss.modules] (main) JBoss Modules version 1.3
>
> 09:20:01,652 INFO [org.jboss.as.process.Host Controller.status] (main) JBAS012017: Starting process 'Host Controller'
>
> 09:20:05,079 INFO [org.jboss.as.process.Server: myserver.status] (ProcessController-threads - 10) JBAS012017: Starting process 'Server: myserver'
>
> 17:01:58,833 INFO [org.jboss.as.process] (Shutdown thread) JBAS012016: Shutting down process controller
>
> 17:02:03,408 INFO [org.jboss.as.process.Host Controller.status] (Shutdown thread) JBAS012018: Stopping process 'Host Controller'
>
> 17:02:15,246 INFO [org.jboss.as.process.Server: myserver.status] (ProcessController-threads - 9) JBAS012018: Stopping process 'Server: myserver'
>
> 17:03:02,990 INFO [org.jboss.as.process.Server.myserver.status] (reaper for Server: myserver) JBAS012010: Process 'Server: myserver' finished with an exit status of 0
>
> 17:03:13,170 INFO [org.jboss.as.process.Host Controller.status] (reaper for Host Controller) JBAS012010: Process 'Host Controller' finished with an exit status of 0
>
> 17:03:13,195 INFO [org.jboss.as.process] (Shutdown thread) JBAS012015: All processes finished; exiting

- If we observe above log file, first column (contains "timestamp") is separated by some Whitespaces with 2nd column (Contains Logging level). It is semi-formatted, not fully formatted text.

XML Documents

- Observe the following XML Document. It is also semi-formatted with XML start and end tags.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <employees>
```

```
3       <emp id> = <"1001">
4           <empname>Abc</empname>
5           <sal>24500</sal>
6           <deptno>10</deptno>
7       </emp>
8   </employees>
```

Un-Structured Data

- Un-Structured Data means Data that is not formatted in any way. It is not possible to store data in Relational Databases.
- For Example:- Audio files, Videos, Call Centre Executive Typed Text, Photos, Sensor Data, Web Data, Mobile Data, GPS Data, Social Media Data etc are Un-Structured Data.
- If we open any image file (for instance, jpeg file) in any text editor, we can see all binary data, which is not at all formatted any form.
- Now-a-Days, People, Machines, Devices, Organizations and Internet are generating Multi-Structured Data that means combination of Structured Data, Semi-Structured Data and Un-Structured Data. It is not at all possible to store and manage this kind of Data using Traditional Old Technologies, Databases and Tools.

  Multi-Structured Data = Structured Data + Semi-structured Data + Un-Structured Data
- Here Big Data solutions solve this problem in efficient and cost-effective way.

Big Data Advantages

- If we use Big Data solutions to store, manage, process and report our Data, we will get the following benefits:
  - Store Data of all types and sizes at low cost.
  - Efficiently Store, Process and Manage our Data.
  - Provides Cost-effective way to mange our Data.
  - Provides Better Performance Solutions.
  - Provides Highly Scalable Solutions.
  - Produces Right Business Value.
  - Increase Productivity.
  - Increase Profits.

Big Data Solutions

- The following is the list of Most Popular Big Data Solutions available in the market.
  - Apache Hadoop Big Data Solution.
  - Amazon Web Services (AWS) Big Data Solutions.
  - Google Cloud Big Data Solutions.
  - Microsoft Big Data Solutions.
  - Cloud Era Big Data Solutions.
  - IBM Big Data Solutions.
  - Oracle Big Data Solutions.

Big Data Use Cases

- Most of the Organizations are using or moving to Big Data. So it is not possible to list out all those Big Data Organizations or Customers here.
- We will provide only some popular Organizations who are using and benefiting from Big Data Solutions.

Facebook

- Facebook is one of the popular Social Networking Website. World-wide, Around 1000 million users are using Facebook Application. It is collecting around 500TB (Tera Bytes) per Day from Users Subscription, User Likes, Posts, Relations Information, Audios, Videos, Pictures etc.

Google

- Google is also using their Big Data Cloud Platform to mange their applications data like Gmail, Google+, Google Search Engine, YouTube etc.

Adhar India

- In India, UIDAI (Unique Identification Authority Of India) manages all Adhar Card information. It is also using Big Data solutions to manage that huge amount of Data.

RedBus

- RedBus is India's largest online Bus Ticket and Hotel Booking organization. It is also using Big Data Solutions to manage that huge amount of Data with very high traffic rate.

eBay and Amazon

- Two World famous online shopping giants eBay and Amazon are also using Big Data solutions to mange their Customer Data, products information etc.

Airline Industry

- A lot of Airlines (For Example:- British Airways, Singapore Airlines etc.) today are using Big Data solutions to store and mange their aircraft and customers information.

Yahoo

- Yahoo is also using their Big Data Cloud Platform solutions to mange their applications data like Yahoo Mail, Yahoo Search Engine, Flickr etc.

Safari Books Online

- Safari Books Online is an online subscription service for Individuals and Organizations to access their online Books, Tutorials, Videos.

New York Stock Exchange

- The New York Stock Exchange is one the famous Stock Exchanges in the World. It generates about 5 TB (Tera Bytes) of data per day.

Big Data is Data with has the Following Three Characteristics:

- Extremely Large Volumes of Data.
- Extremely High Velocity of Data.
- Extremely Wide Variety of Data.

1.3 BIG DATA CHARACTERISTICS

The following three are known as "Big Data Characteristics".

1. Volume
2. Velocity
3. Variety

1. Volume:

- Volume means "How much Data is generated". Now-a-days, Organizations or Human Beings or Systems are generating or getting very vast amount of Data say TB(Tera Bytes) to PB(Peta Bytes) to Exa Byte(EB) and more.
- The name Big Data itself is related to a size which is enormous. Size of data plays a very crucial role in determining value out of data. Also, whether a particular data can actually be considered as a Big Data or not, is dependent upon the volume of data. Hence, 'Volume' is one characteristic which needs to be considered while dealing with Big Data.

  VOLUME = Very large amount of data

2. Velocity:

- Velocity means "How fast produce Data", Now-a-days, Organizations or Human Beings or Systems are generating huge amounts of Data at very fast rate. The term "**velocity**" refers to the speed of generation of data. How fast the data is generated and processed to meet the demands, determines real potential in the data.
- Big Data Velocity deals with the speed at which data flows in from sources like business processes, application logs, networks, and social media sites, sensors, Mobile devices, etc. The flow of data is massive and continuous.

  VELOCITY = Produce data at very fast rate

3. Variety:

- Variety means "Different forms of Data". Now-a-days, Organizations or Human Beings or Systems are generating very huge amount of data at very fast rate in different formats. We will discuss in details about different formats of Data soon. Variety refers to heterogeneous sources and the nature of data, both structured and unstructured.
- During earlier days, spreadsheets and databases were the only sources of data considered by most of the applications. Nowadays, data in the form of emails, photos, videos, monitoring devices, PDFs, audio, etc. are also being considered in the analysis applications. This variety of unstructured data poses certain issues for storage, mining and analyzing data.

  VARIETY = Produce data in different formats

Big Data refers to 3V (VVV) Paradigm:

Fig. 1.2 : BigData-3Vs

- Three "Vs" Paradigm (Volume, Velocity, Variety) of Big Data was defined by "Doug Laney" in 2001.
- If our Organization's Data is in this 3Vs Paradigm, that means we are in Big Data Problems. So we should use some Big Data Solutions to solve our problems.
- These 3Vs Paradigm is not enough to get better value from our Big Data. There is another V (4th V), which is most important for every Big Data problem.

4th V : Veracity

- Veracity means "The Quality or Correctness or Accuracy of Captured Data". Out of 4Vs, it is most important V for any Big Data Solutions. Because without Correct Information or Data, there is no use of storing large amount of data at fast rate and different formats. That data should give correct business value.
- This refers to the inconsistency which can be shown by the data at times, thus hampering the process of being able to handle and manage the data effectively.

  VERACITY = The correctness of data

So this 4th V answers the following questions:

1. How accurate is that data in predicting business value?
2. Do the results of a big data analysis actually make sense?

- Big Data 4Vs In Simple Terminology:

  V(Volume) : The Amount of Data

  V(Variety) : The number of Type of Data

  V(Velocity) : The Speed of Data Processing

  V(Veracity) : The Correctness of Data

1.4 CHALLENGES AND APPLICATIONS OF BIG DATA

1.4.1 Challenges of Big Data

- The handling of big data is very complex. Some challenges faced during its integration include uncertainty of data Management, big data talent gap, getting data into a big data structure, syncing across data sources, getting useful information out of the big data, volume, skill availability, solution cost etc.

1. The Uncertainty of Data Management:

- One disruptive facet of big data management is the use of a wide range of innovative data management tools and frameworks whose designs are dedicated to supporting operational and analytical processing. The NoSQL (not only SQL) frameworks are used that differentiate it from traditional relational database management systems and are also largely designed to fulfill performance demands of big data applications such as managing a large amount of data and quick response times.
- There are a variety of NoSQL approaches such as hierarchical object representation (such as JSON, XML and BSON) and the concept of a key-value storage. The wide range of NoSQL tools, developers and the status of the market are creating uncertainty with the data management.

2. Talent Gap in Big Data:

- It is difficult to win the respect from media and analysts in tech without being bombarded with content touting the value of the analysis of big data and corresponding reliance on a wide range of disruptive technologies.
- The new tools evolved in this sector can range from traditional relational database tools with some alternative data layouts designed to maximize access speed while reducing the storage footprints, NoSQL data management frameworks, in-memory analytics, and as well as the broad Hadoop ecosystem.
- The reality is that there is a lack of skills available in the market for big data technologies. The typical expert has also gained experience through tool implementation and its use as a programming model, apart from the big data management aspects.

3. Getting Data into Big Data Structure:

- It might be obvious that the intent of a big data management involves analyzing and processing a large amount of data. There are many people who have raised expectations considering analyzing huge data sets for a big data platform. They also may not be aware of the complexity behind the transmission, access, and delivery of data and information from a wide range of resources and then loading these data into a big data platform.
- The intricate aspects of data transmission, access and loading are only part of the challenge. The requirement to navigate transformation and extraction is not limited to conventional relational data sets.

4. Syncing Across Data Sources:

- Once you import data into big data platforms you may also realize that data copies migrated from a wide range of sources on different rates and schedules can rapidly get out of the synchronization with the originating system. This implies that the data coming from one source is not out of date as compared to the data coming from another source. It also means the commonality of data definitions, concepts, metadata and the like.

- The traditional data management and data warehouses, the sequence of data transformation, extraction and migrations all arise the situation in which there are risks for data to become unsynchronized.

5. Extracting Information from the Data in Big Data Integration:

- The most practical use cases for big data involve the availability of data, augmenting existing storage of data as well as allowing access to end-user employing business intelligence tools for the purpose of the discovery of data. This business intelligence must be able to connect different big data platforms and also provide transparency of the data consumers to eliminate the requirement of custom coding.
- At the same time, if the number of data consumers grow, then one can provide a need to support an increasing collection of many simultaneous user accesses. This increment of demand may also spike at any time in reaction to different aspects of business process cycles. It also becomes a challenge in big data integration to ensure the right-time data availability to the data consumers.

6. Miscellaneous Challenges:

- Other challenges may occur while integrating big data. Some of the challenges include integration of data, skill availability, solution cost, the volume of data, the rate of transformation of data, veracity and validity of data.
- The ability to merge data that is not similar in source or structure and to do so at a reasonable cost and in time. It is also a challenge to process a large amount of data at a reasonable speed so that information is available for data consumers when they need it. The validation of data set is also fulfilled while transferring data from one source to another or to consumers as well.
- This is all about the big data integration and some challenges that one can face during the implementation. These points must be considered and should be taken care of if you are going to manage any big data platform.

7. Dealing with Data Growth

- The most obvious challenge associated with big data is simply storing and analyzing all that information. In its Digital Universe report, IDC estimates that the amount of information stored in the world's IT systems is doubling about every two years. By 2020, the total amount will be enough to fill a stack of tablets that reaches from the earth to the moon 6.6 times. And enterprises have responsibility or liability for about 85 percent of that information.
- Much of that data is unstructured, meaning that it doesn't reside in a database. Documents, photos, audio, videos and other unstructured data can be difficult to search and analyze.
- It's no surprise, then, that the IDG report found, "Managing unstructured data is growing as a challenge – rising from 31 percent in 2015 to 45 percent in 2016."
- In order to deal with data growth, organizations are turning to a number of different technologies. When it comes to storage, converged and hyper converged infrastructure and software-defined storage can make it easier for companies to scale their hardware. And technologies like compression, deduplication and tiering can reduce the amount of space and the costs associated with big data storage.
- On the management and analysis side, enterprises are using tools like NoSQL databases, Hadoop, Spark, big data analytics software, business intelligence applications, artificial intelligence and machine learning to help them comb through their big data stores to find the insights their companies need.

8. Generating Insights in a Timely Manner

- Of course, organizations don't just want to store their big data they want to use that big data to achieve business goals. According to the New Vantage Partners survey, the most common goals associated with big data projects included the following:
  - ➢ Decreasing expenses through operational cost efficiencies
  - ➢ Establishing a data-driven culture
  - ➢ Creating new avenues for innovation and disruption
  - ➢ Accelerating the speed with which new capabilities and services are deployed
  - ➢ Launching new product and service offerings
- All of those goals can help organizations become more competitive but only if they can extract insights from

their big data and then act on those insights quickly. PwC's Global Data and Analytics Survey 2016 found, "Everyone wants decision-making to be faster, especially in banking, insurance, and healthcare."

- To achieve that speed, some organizations are looking to a new generation of ETL and analytics tools that dramatically reduce the time it takes to generate reports. They are investing in software with real-time analytics capabilities that allows them to respond to developments in the marketplace immediately.

9. Recruiting and Retaining Big Data Talent

- But in order to develop, manage and run those applications that generate insights, organizations need professionals with big data skills. That has driven up demand for big data experts and big data salaries have increased dramatically as a result.
- The 2017 Robert Half Technology Salary Guide reported that big data engineers were earning between $135,000 and $196,000 on average, while data scientist salaries ranged from $116,000 to $163, 500. Even business intelligence analysts were very well paid, making $118,000 to $138,750 per year.
- In order to deal with talent shortages, organizations have a couple of options. First, many are increasing their budgets and their recruitment and retention efforts. Second, they are offering more training opportunities to their current staff members in an attempt to develop the talent they need from within. Third, many organizations are looking to technology.
- They are buying analytics solutions with self-service and/or machine learning capabilities. Designed to be used by professionals without a data science degree, these tools may help organizations achieve their big data goals even if they do not have a lot of big data experts on staff.

10. Integrating Disparate Data Sources

- The variety associated with big data leads to challenges in data integration. Big data comes from a lot of different places enterprise applications, social media streams, email systems, employee-created documents, etc. Combining all that data and reconciling it so that it can be used to create reports can be incredibly difficult. Vendors offer a variety of ETL and data integration tools designed to make the process easier, but many enterprises say that they have not solved the data integration problem yet.
- In response, many enterprises are turning to new technology solutions. In the IDG report, 89 percent of those surveyed said that their companies planned to invest in new big data tools in the next 12 to 18 months. When asked which kind of tools they were planning to purchase, integration technology was second on the list, behind data analytics software.

11. Validating Data

- Closely related to the idea of data integration is the idea of data validation. Often organizations are getting similar pieces of data from different systems, and the data in those different systems doesn't always agree. For example, the ecommerce system may show daily sales at a certain level while the Enterprise Resource Planning (ERP) system has a slightly different number. Or a hospital's Electronic Health Record (EHR) system may have one address for a patient, while a partner pharmacy has a different address on record.
- The process of getting those records to agree, as well as making sure the records are accurate, usable and secure, is called data governance. And in the At Scale 2016 Big Data Maturity Survey, the fastest-growing area of concern cited by respondents was data governance.
- Solving data governance challenges is very complex and is usually requires a combination of policy changes and technology. Organizations often set up a group of people to oversee data governance and write a set of policies and procedures. They may also invest in data management solutions designed to simplify data governance and help ensure the accuracy of big data stores and the insights derived from them.

12. Securing Big Data

- Security is also a big concern for organizations with big data stores. After all, some big data stores can be attractive targets for hackers or Advanced Persistent Threats (APTs).
- However, most organizations seem to believe that their existing data security methods are sufficient for their big data needs as well. In the IDG survey, less than half of those surveyed (39 percent) said that they were using additional security measure for their big data repositories or analyses. Among those who do use additional measures, the most popular include identity and access control (59 percent), data encryption (52 percent) and data segregation (42 percent).

13. Organizational Resistance

- It is not only the technological aspects of big data that can be challenging people can be an issue too.

- In the New Vantage Partners survey, 85.5 percent of those surveyed said that their firms were committed to creating a data-driven culture, but only 37.1 percent said they had been successful with those efforts. When asked about the impediments to that culture shift, respondents pointed to three big obstacles within their organizations:
  (i) Insufficient organizational alignment (4.6 percent)
  (ii) Lack of middle management adoption and understanding (41.0 percent)
  (iii) Business resistance or lack of understanding (41.0 percent)
- In order for organizations to capitalize on the opportunities offered by big data, they are going to have to do some things differently. And that sort of change can be tremendously difficult for large organizations.

1.4.2 Applications of Big Data

- The primary goal of big data analytics is to help companies make more informed business decisions by enabling data scientists, predictive modelers, and other analytics professionals to analyze large volumes of transactional data, as well as other forms of data that may be untapped by more conventional Business Intelligence(BI) programs.
- That could include web server logs and Internet click-stream data, social media content and social network activity reports, text from customer emails and survey responses, mobile phone call detail records and machine data captured by sensors and connected to the Internet of Things.

Big Data Applications:

Fig. 1.3

- Big data has found many applications in various fields today. The major fields where big data is being used are as follows :

Government

- Big data analytics has proven to be very useful in the government sector. Big data analysis played a large role in Barack Obama's successful 2012 re-election campaign. Also most recently, Big data analysis was majorly responsible for the BJP and its allies to win a highly successful Indian General Election 2014. The Indian Government utilizes numerous techniques to ascertain how the Indian electorate is responding to government action, as well as ideas for policy augmentation.

Social Media Analytics

- The advent of social media has led to an outburst of big data. Various solutions have been built in order to analyze social media activity like IBM's Cognos Consumer Insights, a point solution running on IBM's BigInsights Big Data platform, can make sense of the chatter. Social media can provide valuable real-time insights into how the market is responding to products and campaigns. With the help of these insights, the companies can adjust their pricing, promotion, and campaign placements accordingly. Before utilizing the big data there needs to be some preprocessing to be done on the big data in order to derive some intelligent and valuable results. Thus to know the consumer mindset the application of intelligent decisions derived from big data is necessary.

Technology

- The technological applications of big data comprise of the following companies which deal with huge amounts of data every day and put them to use for business decisions as well.
- For example, eBay.com uses two data warehouses at 7.5 peta bytes and 40PB as well as a 40PB Hadoop cluster for search, consumer recommendations, and merchandising. Inside eBay's 90PB data warehouse.
- Amazon.com handles millions of backend operations every day as well as queries from more than half a million third-party sellers. The core technology that keeps Amazon running is Linux-based and as of 2005, they had the world's three largest Linux databases, with capacities of 7.8 TB, 18.5 TB, and 24.7 TB.

- Facebook handles 50 billion photos from its user base. Windermere Real Estate uses anonymous GPS signals from nearly 100 million drivers to help new home buyers determine their typical drive times to and from work throughout various times of the day.

Fraud Detection

- For businesses whose operations involve any type of claims or transaction processing, fraud detection is one of the most compelling Big Data application examples. Historically, fraud detection on the fly has proven an elusive goal. In most cases, fraud is discovered long after the fact, at which point the damage has been done and all that's left is to minimize the harm and adjust policies to prevent it from happening again.
- Big Data platforms that can analyze claims and transactions in real time, identifying large-scale patterns across many transactions or detecting anomalous behavior from an individual user, can change the fraud detection game.

Call Center Analytics

- Now we turn to the customer-facing Big Data application examples, of which call center analytics are particularly powerful. What's going on in a customer's call center is often a great barometer and influencer of market sentiment, but without a Big Data solution, much of the insight that a call center can provide will be overlooked or discovered too late.
- Big Data solutions can help identify recurring problems or customer and staff behavior patterns on the fly not only by making sense of time/quality resolution metrics but also by capturing and processing call content itself.

Banking

- The use of customer data invariably raises privacy issues. By uncovering hidden connections between seemingly unrelated pieces of data, big data analytics could potentially reveal sensitive personal information. Research indicates that 62% of bankers are cautious in their use of big data due to privacy issues. Further, outsourcing of data analysis activities or distribution of customer data across departments for the generation of richer insights also amplifies security risks.
- Such as customer's earnings, savings, mortgages, and insurance policies ended up in the wrong hands. Such incidents reinforce concerns about data privacy and discourage customers from sharing personal information in exchange for customized offers.

Agriculture

- A biotechnology firm uses sensor data to optimize crop efficiency. It plants test crops and runs simulations to measure how plants react to various changes in condition. Its data environment constantly adjusts to changes in the attributes of various data it collects, including temperature, water levels, soil composition, growth, output, and gene sequencing of each plant in the test bed. These simulations allow it to discover the optimal environmental conditions for specific gene types.

Marketing

- Marketers have begun to use facial recognition software to learn how well their advertising succeeds or fails at stimulating interest in their products. A recent study published in the Harvard Business Review looked at what kinds of advertisements compelled viewers to continue watching and what turned viewers off. Among their tools was "a system that analyses facial expressions to reveal what viewers are feeling."
- The research was designed to discover what kinds of promotions induced watchers to share the ads with their social network, helping marketers create ads most likely to "go viral" and improve sales.

Smart Phones

- Perhaps more impressive, people now carry facial recognition technology in their pockets. Users of I Phone and Android smart phones have applications at their fingertips that use facial recognition technology for various tasks. For example, Android users with the remember app can snap a photo of someone, then bring up stored information about that person based on their image when their own memory lets them down a potential boon for salespeople.

Telecoms

- Now a day's big data is used in different fields. In telecom also it plays a very good role. Operators face an uphill challenge when they need to deliver new, compelling, revenue-generating services without overloading their networks and keeping their running costs under control.
- The market demands new set of data management and analysis capabilities that can help service providers

make accurate decisions by taking into account customer, network context and other critical aspects of their businesses. Most of these decisions must be made in real time, placing additional pressure on the operators.

- Real-time predictive analytics can help leverage the data that resides in their multitude systems, make it immediately accessible and help correlate that data to generate insight that can help them drive their business forward.

Healthcare

- Traditionally, the healthcare industry has lagged behind other industries in the use of big data, part of the problem stems from resistance to change providers are accustomed to making treatment decisions independently, using their own clinical judgment, rather than relying on protocols based on big data. Other obstacles are more structural in nature.
- This is one of the best place to set an example for Big Data Application. Even within a single hospital, payor, or pharmaceutical company, important information often remains siloed within one group or department because organizations lack procedures for integrating data and communicating findings.
- Health care stakeholders now have access to promising new threads of knowledge. This information is a form of "big data," so called not only for its sheer volume but for its complexity, diversity, and timelines.
- Pharmaceutical industry experts, payers, and providers are now beginning to analyze big data to obtain insights. Recent technologic advances in the industry have improved their ability to work with such data, even though the files are enormous and often have different database structures and technical characteristics.

1.5 ENABLING TECHNOLOGIES FOR BIG DATA

- The big data analytics technology is a combination of several techniques and processing methods. What makes them effective is their collective use by enterprises to obtain relevant results for strategic management and implementation.
- In spite of the investment enthusiasm, and ambition to leverage the power of data to transform the enterprise, results vary in terms of success. Organizations still struggle to forge what would be consider a "data-driven" culture. Of the executives who report starting such a project, only 40.2% report having success. Big transformations take time, and while the vast majority of firms aspire to being "data-driven", a much smaller percentage have realized this ambition. Cultural transformations seldom occur overnight.
- At this point in the evolution of big data, the challenges for most companies are not related to technology. The biggest impediments to adoption relate to cultural challenges: organizational alignment, resistance or lack of understanding, and change management.
- Here are some key technologies that enable Big Data for Businesses:

Fig. 1.4

1. **Predictive Analytics**

- One of the prime tools for businesses to avoid risks in decision making, predictive analytics can help businesses. Predictive analytics hardware and software solutions can be utilized for discovery, evaluation and deployment of predictive scenarios by processing big data. Such data can help companies to be prepared for what is to come and help solve problems by analyzing and understanding them.

2. **NoSQL Databases**

- These databases are utilized for reliable and efficient data management across a scalable number of storage nodes. NoSQL databases store data as relational database tables, JSON docs or key-value pairings.

3. **Knowledge Discovery Tools**

- These are tools that allow businesses to mine big data (structured and unstructured) which is stored on multiple sources. These sources can be different file systems, APIs, DBMS or similar platforms. With search and knowledge discovery tools, businesses can isolate and utilize the information to their benefit.

4. **Stream Analytics**

- Sometimes the data an organization needs to process can be stored on multiple platforms and in multiple formats. Stream analytics software is highly useful for filtering, aggregation, and analysis of such big data. Stream analytics also allows connection to external data sources and their integration into the application flow.

5. **In-Memory Data Fabric**

- This technology helps in distribution of large quantities of data across system resources such as Dynamic RAM, Flash Storage or Solid State Storage Drives which in turn enables low latency access and processing of big data on the connected nodes.

6. **Distributed Storage**

- A way to counter independent node failures and loss or corruption of big data sources, distributed file stores contain replicated data. Sometimes the data is also replicated for low latency quick access on large computer networks. These are generally non-relational databases.

7. **Data Virtualization**

- It enables applications to retrieve data without implementing technical restrictions such as data formats, the physical location of data, etc. Used by Apache Hadoop and other distributed data stores for real-time or near real-time access to data stored on various platforms, data virtualization is one of the most used big data technologies.

8. **Data Integration**

- A key operational challenge for most organizations handling big data is to process terabytes (or peta bytes) of data in a way that can be useful for customer deliverables. Data integration tools allow businesses to streamline data across a number of big data solutions such as Amazon EMR, Apache Hive, Apache Pig, Apache Spark, Hadoop, MapReduce, MongoDB and Couchbase.

9. **Data Preprocessing**

- These software solutions are used for manipulation of data into a format that is consistent and can be used for further analysis. The data preparation tools accelerate the data sharing process by formatting and cleansing unstructured data sets. A limitation of data preprocessing is that all its tasks cannot be automated and require human oversight, which can be tedious and time-consuming.

10. **Data Quality**

- An important parameter for big data processing is the data quality. The data quality software can conduct cleansing and enrichment of large data sets by utilizing parallel processing. This software is widely used for getting consistent and reliable outputs from big data processing.
- **In conclusion**, Big Data is already being used to improve operational efficiency, and the ability to make informed decisions based on the very latest up-to-the-moment information is rapidly becoming the mainstream norm.

1.6 BIG DATA STACK

The Data Layer

- At the bottom of the stack are technologies that store masses of raw data, which comes from traditional sources like OLTP databases, and newer, less structured sources like log files, sensors, web analytics.

document and media archives. Increasingly, storage happens in the cloud or on virtualized local resources. Organizations are moving away from legacy storage, towards commoditized hardware, and more recently to managed services like Amazon S3.

Data Storage Systems

A few examples:

- **Hadoop HDFS :** The classic big data file system. It became popular due to its robustness and limitless scale on commodity hardware. However, it requires a specialized skill set and complex integration of a myriad open source components.
- **Amazon S3 :** Create buckets and load data using a variety of integrations, with 99.999999999% guaranteed durability. S3 is simple, secure, and provides a quick and cheap solution for storing limitless amounts of big data.
- **MongoDB :** A mature open source document-based database, built to handle data at scale with proven performance. However, some have criticized its use as a first-class data storage system, due to its limited analytical capabilities and no support for transactional data.

The Data Ingestion and Integration Layer

- To create a big data store, you'll need to import data from its original sources into the data layer. In many cases, to enable analysis, you'll need to ingest data into specialized tools, such as data warehouses. This won't happen without a data pipeline. You can leverage a rich ecosystem of big data integration tools, including powerful open source integration tools, to pull data from sources, transform it, and load it to a target system of your choice.

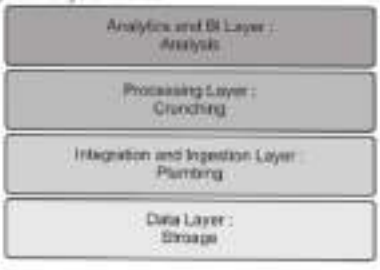


Fig. 1.5

Big Data Ingestion Tools

A few examples:

- **Stitch :** A lightweight ETL (Extract, Transform, Load) tool which pulls data from multiple pre-integrated data sources, transforms and cleans it as necessary. Stitch is easy to setup, seamless and integrates with dozens of data sources. However, it does not support data transformations.
- **Blendo :** A cloud data integration tool that lets you connect data sources with a few clicks, and pipe them to Amazon Redshift, PostgreSQL, MS SQL Server and Panoply's automated data warehouse. Blendo provides schemas and optimization for email marketing, eCommerce and other big data use cases.
- **Apache Kafka :** An open source streaming messaging bus that can creates a feed from your data sources, partitions the data, and streams it to a passive listener. Apache Kafka is a mature and powerful solution used in production at huge scale. However it is complex to implement, and uses a messaging paradigm most data engineers are not familiar with.

The Data Processing Layer

- Thanks to the plumbing, data arrives at its destination. You now need a technology that can crunch the numbers to facilitate analysis. Analysts and data scientists want to run SQL queries against your big data, some of which will require enormous computing power to execute. The data processing layer should optimize the data to facilitate more efficient analysis, and provide a compute engine to run the queries. Data warehouse tools are optimal for processing data at scale, while a data lake is more appropriate for storage, requiring other technologies to assist when data needs to be processed and analyzed.

Data Processing Tools

A few examples:

- **Apache Spark :** Like the old Map/Reduce but over 100X faster. Runs parallelized queries on unstructured, distributed data in Hadoop, Mesos, Kubernetes and elsewhere. Spark also provides a SQL interface, but is not natively a SQL engine.
- **PostgreSQL :** Many organizations pipe their data to good old Postgres to facilitate queries. PostgreSQL can be scaled by sharding or partitioning and is very reliable.

- **Amazon Redshift** : Darling of the data industry, a cloud-based peta byte-scale data warehouse offering blazing query speeds and can be used as a relational database.

Analytics and BI Layer

- You've bought the groceries, whipped up a cake and baked it now you get to eat it! The data layer collected the raw materials for your analysis, the integration layer mixed them all together, the data processing layer optimized, organized the data and executed the queries. The analytics & BI is the real thing using the data to enable data-driven decisions.
- Using the technology in this layer, you can run queries to answer questions the business is asking, slice and dice the data, build dashboards and create beautiful visualizations, using one of many advanced BI tools. Your objective? Answer business questions and provide actionable data which can help the business.

BI / Analytics Tools

A few examples:

- **Tableau** : Powerful BI and data visualization tool, which connects to your data and allows you to drill down, perform complex analysis, and build charts and dashboards.
- **Chartio** : Cloud BI service allowing you to connect data sources, explore data, build SQL queries and transform the data as needed, and create live auto-refreshing dashboards.
- **Looker** : Cloud-based BI platform that lets you query and analyze large data sets via SQL define metrics once set up visualizations that tell a story with your data.

1.7 BIG DATA DISTRIBUTION PACKAGE

- Hadoop is the open source software framework at the heart of much of the Big Data and analytics revolution. It provides solutions for enterprise data storage and analytics with almost unlimited scalability. Since its release in 2011 it has rapidly grown in popularity and a strong ecosystem of distributors, vendors and consultants has emerged to support its use across industry.
- At its core, Hadoop is an Open Source system, which, among other considerations, means it is essentially free for anyone to use. However, the requirement for it to be aligned to the needs of individual organizations has resulted in the emergence of many commercial distributions. These generally come packaged with support or additional features designed to streamline its deployment or allow users to build additional analytics, security or data handling into their framework.
- Competition in this market is fierce and the landscape is constantly shifting. For example all the top distributions now include the Apache Spark parallel processing framework, whereas a few years ago this was not the case. The growing prominence of Spark has resulted in many vendors increasing the resources dedicated to Spark deployment and support.
- One important factor to consider in choosing a Hadoop distribution is whether you want an on-premises or cloud-based solution. If there is no room to compromise when it comes to maintaining complete control and ownership of your data, an on-site solution still theoretically offers the highest level of security. In recent years, though, cloud solutions have become less expensive, more flexible and easier to scale.
- Most of the vendor products here can be installed on a cloud or on-premises. However, some cannot be run on-site. These are generally products from web service providers, such as Amazon or Microsoft, running either Hadoop distributions from other, platform-focused vendors such as Hortonworks or MapR, or their own distributions.
- Beyond that, all of the top distributions have subtle differences which could make them more or less suitable for your business. Here's a non-exhaustive guide to some of the most popular on the market today.

Cloudera

- Cloudera was the first vendor to offer Hadoop as a package and continues to be a leader in the industry. Its Cloudera CDH distribution, which contains all the open source components, is the most popular Hadoop distribution. Cloudera is known for acting quickly to innovate with additions to the core framework – it was the first to offer SQL-for-Hadoop with its Impala query engine. Other additions include user interface, security and interfaces for integration with third party applications. It offers support for the whole of the

distribution through its Cloudera Enterprise subscription service.

Hortonworks

- Hortonworks' platform is entirely open source in fact the company is known for making acquisitions of other companies with useful code and releasing it into the open source community. What some have seen as a start of a trend towards consolidation in the market has prompted a growth in popularity of Hortonworks' product. Recently Pivotal stopped development of its own distribution and both Amazon and IBM are now offering Hortonworks as options on their own platforms, alongside their own Hadoop distributions. Hortonworks' platform is also at the core of the Open Data Platform Initiative a group looking to simplify and standardize specifications in the Big Data ecosphere. In the long run this is likely to mean it will become even more widely supported.

MapR

- Like Hortonworks and Cloudera, MapR is a platform-focused provider, rather than a managed service provider, like Amazon or Microsoft (see below). MapR integrates its own database system MapR-DB which it claims is between four and seven times faster than the stock Hadoop database - HBase running on competing distributions. Due to its power and speed MapR is often seen as a good choice for the biggest of Big Data projects.

Amazon Elastic Map Reduce

- Amazon offers a cloud-only Hadoop-as-a-service platform through its Amazon Web Services arm. A key advantage of the pay-as-you-go model offered by cloud-only service providers is the scalability offered, with storage and data processing able to be ramped up or wound down as demands change. Amazon has recently announced that customers can now use the Apache Flink stream processing framework for real-time data analytics on the platform, along with other popular tools such as Kafka and Presto. It also seamlessly connects (as you would expect) with Amazon's other cloud services infrastructure such as EC2 for cloud processing, Amazon S3 and DynamoDB for storage and AWS IoT to collect data from Internet of Things-enabled devices.

Microsoft

- Microsoft's Azure HDInsight platform is a cloud-only service which offers managed installations of several open source Hadoop distributions including Hortonworks, Cloudera and MapR. It integrates them with its own Azure Data Lake platform to offer a complete solution for cloud-based storage and analytics. As well as the core Hadoop framework, HDInsights provides Spark, Hive, Kafka and Storm cloud services, and its own cloud security framework.

Altiscale

- Acquired recently by SAP for $125 million, Altiscale is another company offering cloud-based, managed Hadoop as-a-service. It continues to offer its Altiscale Data Cloud product, which includes additional operational services like automation, security, scaling and performance-tuning alongside the core Hadoop framework. Data Cloud also provides managed Spark, Hive and Pig services like most of the other products here but unlike the other as-a-service offerings, uses its own Hadoop distribution rather than that of one of the platform-focused vendors such as Hortonworks or MapR.

EXERCISE

1. Define the term Big Data.
2. Why Big Data is so important?
3. State and explain various sources of big data.
4. Give the examples of Big Data formats.
5. What are the advantages of Big Data?
6. Mention any 5 use cases of Big Data.
7. Describe the characteristics of Big Data with suitable examples.
8. Mention any 6 challenges of Big Data.
9. Describe any 5 applications of Big Data.
10. Explain in brief any 3 key technologies that enable Big Data for businesses.
11. Draw and explain the Big Data Stack.
12. Mention various Big Data Distribution package.

❖ ❖ ❖

UNIT II

BIG DATA PLATFORMS

2.1 INTRODUCTION TO BIG DATA PLATFORMS

- Big data platform is a type of IT solution that combines the features and capabilities of several big data application and utilities within a single solution.
- It is an enterprise class IT platform that enables organization in developing, deploying, operating and managing a big data infrastructure /environment. Big data platform generally consists of big data storage, servers, database, big data management, business intelligence and other big data management utilities. It also supports custom development, querying and integration with other systems. The primary benefit behind a big data platform is to reduce the complexity of multiple vendors/ solutions into a one cohesive solution. Big data platform are also delivered through cloud where the provider provides an all inclusive big data solutions and services.
- Big Data Platform is integrated IT solution for Big Data management which combines several software system, software tools and hardware to provide easy to use tools system to enterprises.
- It is a single one-stop solution for all Big Data needs of an enterprise irrespective of size and data volume. Big Data Platform is enterprise class IT solution for developing, deploying and managing Big Data.
- There are several Open source and commercial Big Data Platform in the market with varied features which can be used in Big Data environment.

Features of Big Data Platform

Here are most important features of any good Big Data Analytics Platform:

- Big Data platform should be able to accommodate new platforms and tool based on the business requirement. Because business needs can change due to new technologies or due to change in business process.
- It should support linear scale-out.
- It should have capability for rapid deployment.
- It should support variety of data format.
- Platform should provide data analysis and reporting tools.
- It should provide real-time data analysis software.
- It should have tools for searching the data through large data sets.

2.2 OVERVIEW OF APACHE SPARK

2.2.1 Introduction to Apache Spark

- Apache Spark is a data processing framework that can quickly perform processing tasks on very large data sets, and can also distribute data processing tasks across multiple computers, either on its own or in tandem with other distributed computing tools. These two qualities are key to the worlds of big data and machine learning, which require the marshaling of massive computing power to crunch through large data stores. Spark also takes some of the programming burdens of these tasks off the shoulders of developers with an easy-to-use API that abstracts away much of the grunt work of distributed computing and big data processing.
- From its humble beginnings in the AMPLab at U.C. Berkeley in 2009, Apache Spark has become one of the key big data distributed processing frameworks in the world. Spark can be deployed in a variety of ways, provides native bindings for the Java, Scala, Python, and R programming languages, and supports SQL, streaming data, machine learning, and graph processing. You'll find it used by banks, telecommunications companies, games companies, governments, and all of the major tech giants such as Apple, Facebook, IBM, and Microsoft.

2.2.2 Apache Spark Architecture

- At a fundamental level, an Apache Spark application consists of two main components: a driver, which converts the user's code into multiple tasks that can be distributed across worker nodes, and executors, which

run on those nodes and execute the tasks assigned to them. Some form of cluster manager is necessary to mediate between the two.

- Out of the box, Spark can run in a standalone cluster mode that simply requires the Apache Spark framework and a JVM on each machine in your cluster. However, it's more likely you'll want to take advantage of a more robust resource or cluster management system to take care of allocating workers on demand for you. In the enterprise, this will normally mean running on Hadoop YARN (this is how the Cloudera and Hortonworks distributions run Spark jobs), but Apache Spark can also run on Apache Mesos, Kubernetes, and Docker Swarm.
- If you seek a managed solution, then Apache Spark can be found as part of Amazon EMR, Google Cloud Dataproc, and Microsoft Azure HDInsight. Databricks, the company that employs the founders of Apache Spark, also offers the Databricks Unified Analytics Platform, which is a comprehensive managed service that offers Apache Spark clusters, streaming support, integrated web-based notebook development, and optimized cloud I/O performance over a standard Apache Spark distribution.
- Apache Spark builds the user's data processing commands into a Directed Acyclic Graph, or DAG. The DAG is Apache Spark's scheduling layer; it determines what tasks are executed on what nodes and in what sequence.
- Spark has become the framework of choice when processing big data, overtaking the old MapReduce paradigm that brought Hadoop to prominence.
- The first advantage is speed. Spark's in-memory data engine means that it can perform tasks up to one hundred times faster than MapReduce in certain situations, particularly when compared with multi-stage jobs that require the writing of state back out to disk between stages. In essence, MapReduce creates a two-stage execution graph consisting of data mapping and reducing, whereas Apache Spark's DAG has multiple stages that can be distributed more efficiently. Even Apache Spark jobs where the data cannot be completely contained within memory tend to be around 10 times faster than their MapReduce counterpart.
- The second advantage is the developer-friendly Spark API. As important as Spark's speedup is, one could argue that the friendliness of the Spark API is even more important.

2.2.3 Spark Core

- In comparison to MapReduce and other Apache Hadoop components, the Apache Spark API is very friendly to developers, hiding much of the complexity of a distributed processing engine behind simple method calls. The canonical example of this is how almost 50 lines of MapReduce code to count words in a document can be reduced to just a few lines of Apache Spark (here shown in Scala):

```
val textFile =
sparkSession.sparkContext.textFile("hdfs:///tmp/word
s")
val counts = textFile.flatMap(line => line.split(" "))
             .map(word => (word, 1))
             .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs:///tmp/words_agg")
```

- By providing bindings to popular languages for data analysis like Python and R, as well as the more enterprise-friendly Java and Scala, Apache Spark allows everybody from application developers to data scientists to harness its scalability and speed in an accessible manner.

2.2.4 Spark RDD

- At the heart of Apache Spark is the concept of the Resilient Distributed Dataset (RDD), a programming abstraction that represents an immutable collection of objects that can be split across a computing cluster. Operations on the RDDs can also be split across the cluster and executed in a parallel batch process, leading to fast and scalable parallel processing.
- RDDs can be created from simple text files, SQL databases, NoSQL stores (such as Cassandra and MongoDB), Amazon S3 buckets, and much more besides. Much of the Spark Core API is built on this RDD concept, enabling traditional map and reduce functionality, but also providing built-in support for joining data sets, filtering, sampling, and aggregation.
- Spark runs in a distributed fashion by combining a driver core process that splits a Spark application into tasks and distributes them among many executor processes that do the work. These executors can be scaled up and down as required for the application's needs.

2.2.5 Spark SQL

- Originally known as Shark, Spark SQL has become more and more important to the Apache Spark project. It is likely the interface most commonly used by today's developers when creating applications. Spark SQL is focused on the processing of structured data, using a data frame approach borrowed from R and Python (in Pandas). But as the name suggests, Spark SQL also provides a SQL2003-compliant interface for querying data, bringing the power of Apache Spark to analysts as well as developers.
- Alongside standard SQL support, Spark SQL provides a standard interface for reading from and writing to other data stores including JSON, HDFS, Apache Hive, JDBC, Apache ORC, and Apache Parquet, all of which are supported out of the box. Other popular stores Apache Cassandra, MongoDB, Apache HBase, and many others can be used by pulling in separate connectors from the Spark Packages ecosystem.
- Selecting some columns from a data frame is as simple as this line:

```
citiesDF.select("name", "pop")
```

- Using the SQL interface, we register the dataframe as a temporary table, after which we can issue SQL queries against it:

```
citiesDF.createOrReplaceTempView("cities")
spark.sql("SELECT name, pop FROM cities")
```

- Behind the scenes, Apache Spark uses a query optimizer called Catalyst that examines data and queries in order to produce an efficient query plan for data locality and computation that will perform the required calculations across the cluster. In the Apache Spark 2.x era, the Spark SQL interface of data frames and datasets (essentially a typed data frame that can be checked at compile time for correctness and take advantage of further memory and compute optimizations at run time) is the recommended approach for development. The RDD interface is still available, but recommended only if your needs cannot be addressed within the Spark SQL paradigm.

2.2.6 Spark MLlib

- Apache Spark also bundles libraries for applying machine learning and graph analysis techniques to data at scale. Spark MLlib includes a framework for creating machine learning pipelines, allowing for easy implementation of feature extraction, selections, and transformations on any structured dataset. MLlib comes with distributed implementations of clustering and classification algorithms such as k-means clustering and random forests that can be swapped in and out of custom pipelines with ease. Models can be trained by data scientists in Apache Spark using R or Python, saved using MLlib, and then imported into a Java-based or Scala-based pipeline for production use.

2.2.7 Spark GraphX

- Spark GraphX comes with a selection of distributed algorithms for processing graph structures including an implementation of Google's PageRank. These algorithms use Spark Core's RDD approach to modeling data; the GraphFrames package allows you to do graph operations on data frames, including taking advantage of the Catalyst optimizer for graph queries.

2.2.8 Spark Streaming

- Spark Streaming was an early addition to Apache Spark that helped it gain traction in environments that required real-time or near real-time processing. Previously, batch and stream processing in the world of Apache Hadoop were separate things. You would write MapReduce code for your batch processing needs and use something like Apache Storm for your real-time streaming requirements. This obviously leads to disparate codebases that need to be kept in sync for the application domain despite being based on completely different frameworks, requiring different resources, and involving different operational concerns for running them.
- Spark Streaming extended the Apache Spark concept of batch processing into streaming by breaking the stream down into a continuous series of microbatches, which could then be manipulated using the Apache Spark API. In this way, code in batch and streaming operations can share (mostly) the same code, running on the same framework, thus reducing both developer and operator overhead. Everybody wins.
- A criticism of the Spark Streaming approach is that microbatching, in scenarios where a low-latency response to incoming data is required, may not be able to match the performance of other streaming-capable

frameworks like Apache Storm, Apache Flink, and Apache Apex, all of which use a pure streaming method rather than microbatches.

2.2.9 Structured Streaming

- Structured Streaming (added in Spark 2.x) is to Spark Streaming what Spark SQL was to the Spark Core APIs: A higher-level API and easier abstraction for writing applications. In the case of Structure Streaming, the higher-level API essentially allows developers to create infinite streaming data frames and datasets. It also solves some very real pain points that users have struggled with in the earlier framework, especially concerning dealing with event-time aggregations and late delivery of messages. All queries on structured streams go through the Catalyst query optimizer, and can even be run in an interactive manner, allowing users to perform SQL queries against live streaming data.
- Structured Streaming originally relied on Spark Streaming's microbatching scheme of handling streaming data. But in Spark 2.3, the Apache Spark team added a low-latency Continuous Processing Mode to Structured Streaming, allowing it to handle responses with latencies as low as 1ms, which is very impressive. As of Spark 2.4, Continuous Processing is still considered experimental. While Structured Streaming is built on top of the Spark SQL engine, Continuous Streaming supports only a restricted set of queries.
- Structured Streaming is the future of streaming applications with the platform, so if you're building a new streaming application, you should use Structured Streaming. The legacy Spark Streaming APIs will continue to be supported, but the project recommends porting over to Structured Streaming, as the new method makes writing and maintaining streaming code a lot more bearable.

2.3 HDFS

- Hadoop File System was developed using distributed file system design. It is run on commodity hardware. Unlike other distributed systems, HDFS is highly fault tolerant and designed using low-cost hardware.
- HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing.

2.3.1 Features of HDFS

- It is suitable for the distributed storage and processing.
- Hadoop provides a command interface to interact with HDFS.
- The built-in servers of namenode and datanode help users to easily check the status of cluster.
- Streaming access to file system data.
- HDFS provides file permissions and authentication.

2.3.2 HDFS Architecture

- Given below is the architecture of a Hadoop File System.

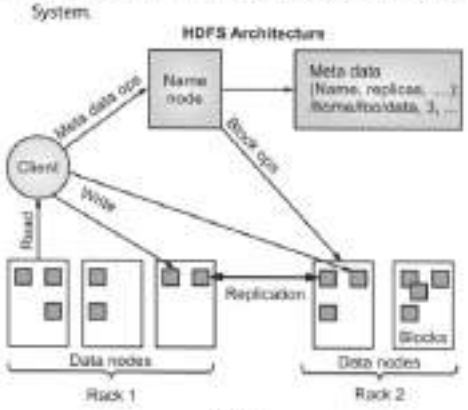


Fig. 2.1

- HDFS follows the master-slave architecture and it has the following elements.

2.3.3 Namenode

- The namenode is the commodity hardware that contains the GNU/Linux operating system and the namenode software. It is a software that can be run on commodity hardware. The system having the namenode acts as the master server and it does the following tasks –
  - ➢ Manages the file system namespace.
  - ➢ Regulates client's access to files.
  - ➢ It also executes file system operations such as renaming, closing, and opening files and directories.

2.3.4 Datanode

- The datanode is a commodity hardware having the GNU/Linux operating system and datanode software. For every node (Commodity hardware/System) in a cluster, there will be a datanode. These nodes manage the data storage of their system.
- Datanodes perform read-write operations on the file systems, as per client request.
- They also perform operations such as block creation, deletion, and replication according to the instructions of the namenode.

2.3.5 Block

- Generally the user data is stored in the files of HDFS. The file in a file system will be divided into one or more segments and/or stored in individual data nodes. These file segments are called as blocks. In other words, the minimum amount of data that HDFS can read or write is called a Block. The default block size is 64MB, but it can be increased as per the need to change in HDFS configuration.

2.3.6 Goals of HDFS

- **Fault Detection and Recovery :** Since HDFS includes a large number of commodity hardware, failure of components is frequent. Therefore HDFS should have mechanisms for quick and automatic fault detection and recovery.
- **Huge Datasets :** HDFS should have hundreds of nodes per cluster to manage the applications having huge datasets.
- **Hardware at Data :** A requested task can be done efficiently, when the computation takes place near the data. Especially where huge datasets are involved, it reduces the network traffic and increases the throughput.

2.3.7 HDFS Operations

Starting HDFS

- Initially you have to format the configured HDFS file system, open namenode (HDFS server), and execute the following command.

```
$ hadoop namenode -format
```

- After formatting the HDFS, start the distributed file system. The following command will start the namenode as well as the data nodes as cluster.

```
$ start-dfs.sh
```

Listing Files in HDFS

- After loading the information in the server, we can find the list of files in a directory, status of a file, using 'ls'. Given below is the syntax of **ls** that you can pass to a directory or a filename as an argument.

```
$ $HADOOP_HOME/bin/hadoop fs -ls <args>
```

Inserting Data into HDFS

- Assume we have data in the file called file.txt in the local system which is ought to be saved in the hdfs file system. Follow the steps given below to insert the required file in the Hadoop file system.

Step 1

You have to create an input directory.

```
$ $HADOOP_HOME/bin/hadoop fs -mkdir
/user/input
```

Step 2

Transfer and store a data file from local systems to the Hadoop file system using the put command.

```
$ $HADOOP_HOME/bin/hadoop fs -put
/home/file.txt /user/input
```

Step 3

You can verify the file using ls command.

```
$ $HADOOP_HOME/bin/hadoop fs -ls /user/input
```

Retrieving Data from HDFS

- Assume we have a file in HDFS called **outfile**. Given below is a simple demonstration for retrieving the required file from the Hadoop file system.

Step 1

Initially, view the data from HDFS using **cat** command.

```
$ $HADOOP_HOME/bin/hadoop fs -cat
/user/output/outfile
```

Step 2

Get the file from HDFS to the local file system using **get** command.

```
$ $HADOOP_HOME/bin/hadoop fs -get
/user/output/ /home/hadoop_tp/
```

Shutting Down the HDFS

You can shut down the HDFS by using the following command.

```
$ stop-dfs.sh
```

2.3.8 HDFS Command References

- There are many more commands in "**$HADOOP_HOME/bin/hadoop fs**" than are demonstrated here, although these basic operations will get you started. Running ./bin/hadoop dfs with no additional arguments will list all the commands that can be run with the FsShell system. Furthermore, **$HADOOP_HOME/bin/hadoop fs -help** commandName will display a short usage summary for the operation in question, if you are stuck.
- A table of all the operations is shown below. The following conventions are used for parameters –

"<path>" means any file or directory name.

"<path>..." means one or more file or directory names.

"<file>" means any filename.

"<src>" and "<dest>" are path names in a directed operation.

"<localSrc>" and "<localDest>" are paths as above, but on the local file system.

All other files and path names refer to the objects inside HDFS.

Sr. No	Command & Description
1	**-ls <path>** Lists the contents of the directory specified by path, showing the names, permissions, owner, size and modification date for each entry.
2	**-lsr <path>** Behaves like -ls, but recursively displays entries in all subdirectories of path.
3	**-du <path>** Shows disk usage, in bytes, for all the files which match path; filenames are reported with the full HDFS protocol prefix.
4	**-dus <path>** Like -du, but prints a summary of disk usage of all files/directories in the path.
5	**-mv <src><dest>** Moves the file or directory indicated by src to dest, within HDFS.
6	**-cp <src> <dest>** Copies the file or directory identified by src to dest, within HDFS.
7	**-rm <path>** Removes the file or empty directory identified by path.
8	**-rmr <path>** Removes the file or directory identified by path. Recursively deletes any child entries (i.e., files or subdirectories of path).
9	**-put <localSrc> <dest>** Copies the file or directory from the local file system identified by localSrc to dest within the DFS.
10	**-copyFromLocal <localSrc> <dest>** Identical to -put
11	**-moveFromLocal <localSrc> <dest>** Copies the file or directory from the local file system identified by localSrc to dest within HDFS, and then deletes the local copy on success.
12	**-get [-crc] <src> <localDest>** Copies the file or directory in HDFS identified by src to the local file system path identified by localDest.
13	**-getmerge <src> <localDest>** Retrieves all files that match the path src in HDFS, and copies them to a single, merged file in the local file system identified by localDest.
14	**-cat <filen-ame>** Displays the contents of filename on stdout.
15	**<copyToLocal <src> <localDest>** Identical to -get
16	**-moveToLocal <src> <localDest>** Works like -get, but deletes the HDFS copy on success.
17	**-mkdir <path>** Creates a directory named path in HDFS. Creates any parent directories in path that are missing (e.g., mkdir -p in Linux).
18	**-setrep [-R] [-w] rep <path>** Sets the target replication factor for files identified by path to rep. (The actual replication factor will move toward the target over time)
19	**-touchz <path>** Creates a file at path containing the current time as a timestamp. Fails if a file already exists at path, unless the file is already size 0.
20	**-test -[ezd] <path>** Returns 1 if path exists; has zero length; or is a directory or 0 otherwise.
21	**-stat [format] <path>** Prints information about path. Format is a string which accepts file size in blocks (%b), filename (%n), block size (%o), replication (%r), and modification date (%y, %Y).

22	**-tail [-f] <file2name>** Shows the last 1KB of file on stdout.
23	**-chmod [-R] mode,mode,... <path>...** Changes the file permissions associated with one or more objects identified by path.... Performs changes recursively with R. mode is a 3-digit octal mode, or (augo)+/-(rwxX). Assumes if no scope is specified and does not apply an umask.
24	**-chown [-R] [owner][:[group]] <path>...** Sets the owning user and/or group for files or directories identified by path.... Sets owner recursively if -R is specified.
25	**-chgrp [-R] group <path>...** Sets the owning group for files or directories identified by path.... Sets group recursively if -R is specified.
26	**-help <cmd-name>** Returns usage information for one of the commands listed above. You must omit the leading '-' character in cmd.

2.4 YARN

- YARN stands for "Yet Another Resource Negotiator". It was introduced in Hadoop 2.0 to remove the bottleneck on Job Tracker which was present in Hadoop 1.0. YARN was described as a "Redesigned Resource Manager" at the time of its launching, but it has now evolved to be known as large-scale distributed operating system used for Big Data processing.

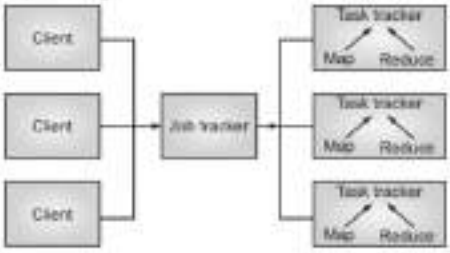


Fig. 2.2 : Hadoop 1.0 architecture

- YARN architecture basically separates resource management layer from the processing layer. In Hadoop 1.0 version, the responsibility of Job tracker is split between the resource manager and application manager.

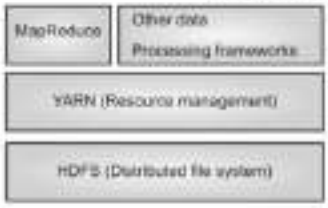


Fig. 2.3 : Hadoop 2.0

- YARN also allows different data processing engines like graph processing, interactive processing, stream processing as well as batch processing to run and process data stored in HDFS (Hadoop Distributed File System) thus making the system much more efficient. Through its various components, it can dynamically allocate various resources and schedule the application processing. For large volume data processing, it is quite necessary to manage the available resources properly so that every application can leverage them.

YARN Features: YARN gained popularity because of the following features-

- **Scalability:** The scheduler in Resource manager of YARN architecture allows Hadoop to extend and manage thousands of nodes and clusters.
- **Compatability:** YARN supports the existing map-reduce applications without disruptions thus making it compatible with Hadoop 1.0 as well.
- **Cluster Utilization:** Since YARN supports Dynamic utilization of cluster in Hadoop, which enables optimized Cluster Utilization.
- **Multi-Tenancy:** It allows multiple engine access thus giving organizations a benefit of multi-tenancy.

Hadoop YARN Architecture

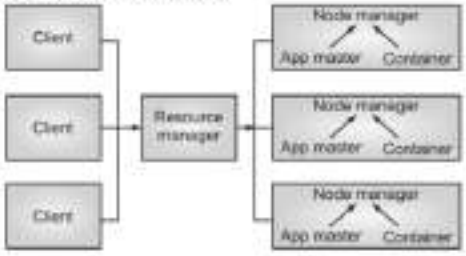


Fig. 2.4 : Hadoop YARN Architecture

The main components of YARN architecture include:

- **Client:** It submits map-reduce jobs.
- **Resource Manager:** It is the master daemon of YARN and is responsible for resource assignment and management among all the applications. Whenever it receives a processing request, it forwards it to the corresponding node manager and allocates resources for the completion of the request accordingly. It has two major components:
  - **Scheduler:** It performs scheduling based on the allocated application and available resources. It is a pure scheduler, means it does not perform other tasks such as monitoring or tracking and does not guarantee a restart if a task fails. The YARN scheduler supports plugins such as Capacity Scheduler and Fair Scheduler to partition the cluster resources.
  - **Application Manager:** It is responsible for accepting the application and negotiating the first container from the resource manager. It also restarts the Application Manager container if a task fails.
- **Node Manager:** It take care of individual node on Hadoop cluster and manages application and workflow and that particular node. Its primary job is to keep-up with the Node Manager. It monitors resource usage, performs log management and also kills a container based on directions from the resource manager. It is also responsible for creating the container process and start it on the request of Application master.
- **Application Master:** An application is a single job submitted to a framework. The application manager is responsible for negotiating resources with the resource manager, tracking the status and monitoring progress of a single application. The application master requests the container from the node manager by sending a Container Launch Context(CLC) which includes everything an application needs to run. Once the application is started, it sends the health report to the resource manager from time-to-time.
- **Container:** It is a collection of physical resources such as RAM, CPU cores and disk on a single node. The containers are invoked by Container Launch Context(CLC) which is a record that contains information such as environment variables, security tokens, dependencies etc.

Application Workflow in Hadoop YARN:

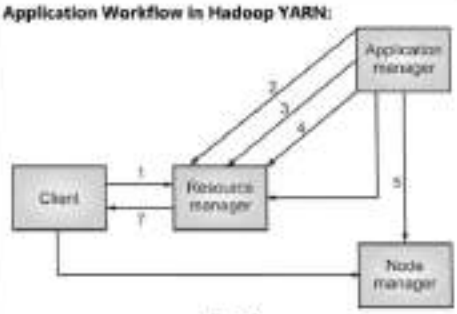


Fig. 2.5

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

2.5 MAPREDUCE

- One of the three components of Hadoop is Map Reduce. The first component of Hadoop that is, Hadoop Distributed File System (HDFS) is responsible for storing the file. The second component that is, Map Reduce is responsible for processing the file.
- Suppose there is a word file containing some text. Let us name this file as sample.txt. Note that we use Hadoop to deal with huge files but for the sake of easy explanation over here, we are taking a text file as an example. So, let's assume that this sample.txt file contains few lines as text. The content of the file is as follows:

```
1. Hello I am GeeksforGeeks
2. How can I help you
3. How can I assist you
4. Are you an engineer
5. Are you looking for coding
6. Are you looking for interview questions
7. What are you doing these days
8. What are your strengths
```

- Hence, the above eight lines are the content of the file. Let's assume that while storing this file in Hadoop, HDFS broke this file into four parts and named each part as first.txt, second.txt, third.txt, and fourth.txt. So, you can easily see that the above file will be divided into four equal parts and each part will contain two lines. First two lines will be in the file first.txt, next two lines in second.txt, next two in third.txt and the last two lines will be stored in fourth.txt. All these files will be stored in Data Nodes and the Name Node will contain the metadata about them. All this is the task of HDFS.
- Now, suppose a user wants to process this file. Here is what Map-Reduce comes into the picture. Suppose this user wants to run a query on this sample.txt. So, instead of bringing sample.txt on the local computer, we will send this query on the data. To keep a track of our request, we use Job Tracker (a master service). Job Tracker traps our request and keeps a track of it.
- Now suppose that the user wants to run his query on sample.txt and want the output in result_output file. Let the name of the file containing the query is query.jar. So, the user will write a query like:

```
$hadoop jar query.jar DriverCode sample.txt
result.output
```

- **query.jar** : query file that needs to be processed on the input file.
- **sample.txt:** input file.
- **result.output:** directory in which output of the processing will be received.
- So, now the Job Tracker traps this request and asks Name Node to run this request on sample.txt. Name Node then provides the metadata to the Job Tracker. Job Tracker now knows that sample.txt is stored in first.txt, second.txt, third.txt, and fourth.txt. As all these four files have three copies stored in HDFS, so the Job Tracker communicates with the ***Task Tracker*** (a slave service) of each of these files but it communicates with only one copy of each file which is residing nearest to it.

Note: Applying the desired code on local first.txt, second.txt, third.txt and fourth.txt is a process., This process is called **Map**.

- In Hadoop terminology, the main file sample.txt is called input file and its four sub files are called input splits. So, in Hadoop the number of mappers for an input file are equal to number of input splits of this input file. In the above case, the input file sample.txt has four input splits hence four mappers will be running to process it. The responsibility of handling these mappers is of Job Tracker.
- Note that the task trackers are slave services to the Job Tracker. So, in case any of the local machines breaks down then the processing over that part of the file will stop and it will halt the complete process. So, each task tracker sends heartbeat and its number of slots to Job Tracker in every 3 seconds. This is called the status of *Task Trackers*. In case any task tracker goes down, the Job Tracker then waits for 10 heartbeat times, that is, 30 seconds, and even after that if it does not get any status, then it assumes that either the task tracker is dead or is extremely busy. So it then communicates with the task tracker of another copy of the same file and directs it to process the desired code over it. Similarly, the slot information is used by the Job Tracker to keep a track of how many tasks are being currently served by the task tracker and how many more tasks can be assigned to it. In this way, the Job Tracker keeps track of our request.
- Now, suppose that the system has generated output for individual first.txt, second.txt, third.txt, and fourth.txt. But this is not the user's desired output. To produce the desired output, all these individual outputs have to be merged or reduced to a single output. This reduction of multiple outputs to a single one is also a process which is done by REDUCER. In Hadoop, as many reducers are there, those many number of output files are generated. By default, there is always one reducer per cluster.

Note: Map and Reduce are two different processes of the second component of Hadoop, that is, Map Reduce. These are also called phases of Map Reduce. Thus we can say that Map Reduce has two phases. Phase 1 is Map and Phase 2 is Reduce.

Functioning of Map Reduce

- Now, let us move back to our sample.txt file with the same content. Again it is being divided into four input splits namely, first.txt, second.txt, third.txt, and fourth.txt. Now, suppose we want to count number of each word in the file. That is the content of the file looks like:

Hello I am GeeksforGeeks
How can I help you
How can I assist you
Are you an engineer
Are you looking for coding
Are you looking for interview questions
what are you doing these days
what are your strengths
Then the output of the 'word count' code will be like:
Hello - 1
I - 1
am - 1
geeksforgeeks - 1
How - 2 (How is written two times in the entire file)
Similarly
Are - 3
are - 2
..and so on

- Thus in order to get this output, the user will have to send his query on the data. Suppose the query 'word count' is in the file wordcount.jar. So, the query will look like:

```
J$hadoop jar wordcount.jar DriverCode sample.txt
result.output
```

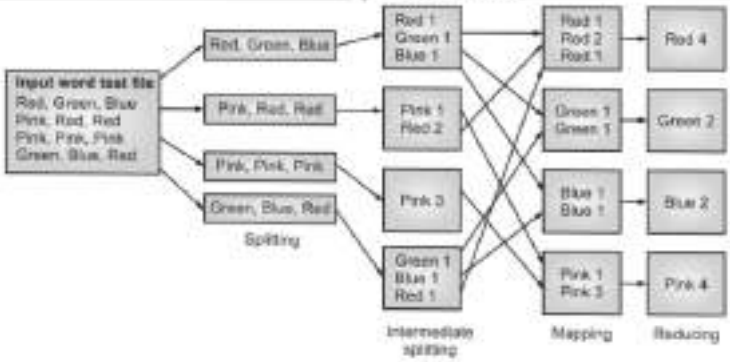


Fig. 2.6

Types of File Format in Hadoop

- Now, as we know that there are four input splits, so four mappers will be running. One on each input split. But, Mappers don't run directly on the input splits. It is because the input splits contain text but mappers don't understand the text. Mappers understand (key, value) pairs only. Thus the text in input splits first needs to be converted to (key, value) pairs. This is achieved by Record Readers. Thus we can also say that as many numbers of input splits are there, those many numbers of record readers are there.
- In Hadoop terminology, each line in a text is termed as a 'record'. How record reader converts this text into (key, value) pair depends on the format of the file. In Hadoop, there are four formats of a file. These formats are Predefined Classes in Hadoop.

Four Types of Formats are:

1. TextInputFormat.
2. KeyValueTextInputFormat.
3. SequenceFileInputFormat.
4. SequenceFileAsTextInputFormat.

- By default, a file is in TextInputFormat. Record reader reads one record(line) at a time. While reading, it doesn't consider the format of the file. But, it converts each record into (key, value) pair depending upon its format. For the time being, let's assume that the first input split first.txt is in TextInputFormat. Now, the record reader working on this input split converts the record in the form of (byte offset, entire line). For example first.txt has the content:

```
Hello I am GeeksforGeeks
How can I help you
```

- So, the output of record reader has two pairs (since two records are there in the file). The first pair looks like (0, Hello I am geeksforgeeks) and the second pair looks like (26, How can I help you). Note that the second pair has the byte offset of 26 because there are 25 characters in the first line and the newline operator (\n) is also considered a character. Thus, after the record reader as many numbers of records is there, those many numbers of (key, value) pairs are there. Now, the mapper will run once for each of these pairs. Similarly, other mappers are also running for (key, value) pairs of different input splits. Thus in this way,

Hadoop breaks a big task into smaller tasks and executes them in parallel execution.

Shuffling and Sorting

- Now, the mapper provides an output corresponding to each (key, value) pair provided by the record reader. Let us take the first input split of first.txt. The two pairs so generated for this file by the record reader are (0, Hello I am GeeksforGeeks) and (26, How can I help you). Now mapper takes one of these pair at a time and produces output like (Hello, 1), (I, 1), (am, 1) and (GeeksforGeeks, 1) for the first pair and (How, 1), (can, 1), (I, 1), (help, 1) and (you, 1) for the second pair. Similarly, we have outputs of all the mappers. Note that this data contains duplicate keys like (I, 1) and further (how, 1) etc. These duplicate keys also need to be taken care of. This data is also called Intermediate Data. Before passing this intermediate data to the reducer, it is first passed through two more stages, called Shuffling and Sorting.
  1. **Shuffling Phase:** This phase combines all values associated to an identical key. For eg, (Are, 1) is there three times in the input file. So after the shuffling phase, the output will be like (Are, [1,1,1]).
  2. **Sorting Phase:** Once shuffling is done, the output is sent to the sorting phase where all the (key, value) pairs are sorted automatically. In Hadoop sorting is an automatic process because of the presence of an inbuilt interface called WritableComparableInterface.
- After the completion of the shuffling and sorting phase, the resultant output is then sent to the reducer. Now, if there are n (key, value) pairs after the shuffling and sorting phase, then the reducer runs n times and thus produces the final result in which the final processed output is there. In the above case, the resultant output after the reducer processing will get stored in the directory result.output as specified in the query code written to process the query on the data.

2.6 MAPREDUCE PROGRAMMING MODEL WITH SPARK

- Dealing with massive amount of data sets requires efficient processing methods that will reduce run-time and cost in addition to overcome memory constraints. Parallel Processing is one of the most used methods by data scientists when it comes to compute and data-intensive tasks, where a task is broken up to multiple parts with a software tool and each part is distributed to a processor, then each processor will perform the assigned part. Finally, the parts are reassembled to deliver the final solution or execute the task.
- Please note that **Parallel Processing** should not be confused with **Multiprocessing** in where multiple processors or cores are working on solving different tasks, instead of parts of the same task as in **parallel processing**.
- When we perform Parallel Processing there are several points that we should take care of to ensure the success of the process:
  1. **Load Balancing Issue:** We need to ensure each machine will get an equal share when dividing the data into chunks so non of the machines is being overloaded or underutilized.
  2. **Critical Path Problem:** Managing the time required by each machine to finish the job, if any of the machines failed to deliver the job in time, the whole task gets delayed.
  3. **Reliability:** There should be a mechanism to solve the fail of one of the machines, because if one machine failed to provide output the whole task will fail.
  4. **Aggregation of the Result:** An aggregation approach should be designed to generate the final result after getting the input form different machines.
- So, performing parallel processing is not an easy task, that's when MapReduce framework comes in handy, which allows us to write code logic to run in parallel without bothering about the previous mentioned issues.

MapReduce Programming Model

Let's start by explaining some terms:

- **MapReduce:** Is a programming model that allows us to perform parallel processing across Big Data using a large number of nodes (multiple computers).
- **Cluster Computing:** Nodes are homogeneous and located on the same local network.
- **Grid Computing:** Nodes are heterogeneous (different hardware) and located geographically far from each other.

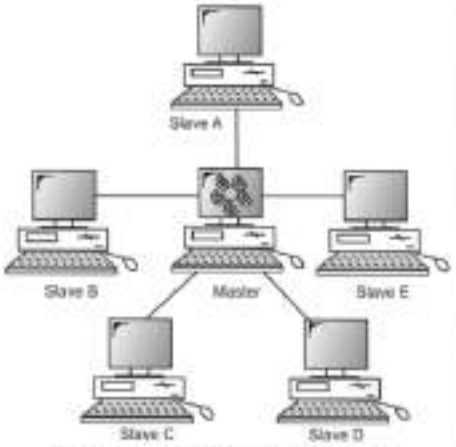


Fig. 2.7 (a) : Moving data to the processing unit (Traditional Approch)

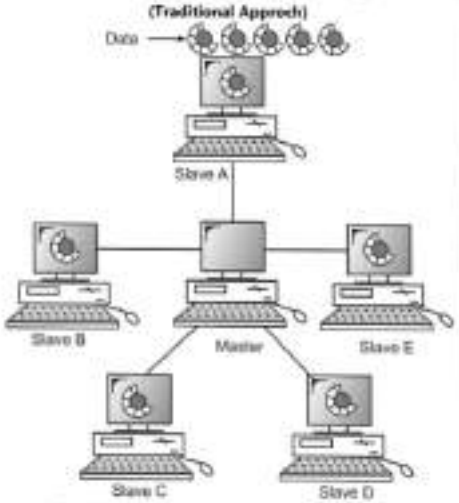


Fig. 2.7 (b) : Moving processing unit to the data (MapReduce Approch)

- Another big advantage offered by MapReduce is what we call **Data Locality.** Simply put, data locality is bringing the processing unit to the data(i.e. performing computation process on the site where the data is being saved) instead of sending the data to the unit, which will significantly reduce the network traffic and consequently running time.

MapReduce programming Steps:

- **Input Split:** In this step raw input data is being divided into chunks called input splits. Each chunk will be an input of a single map, typically an input split will have the size between 16 MP- 64 MB (it's controllable by the user). Data input of this step will from the form (key1, Value1).
- **Mapping:** A node who is assigned a map function takes the input and emits a set of (key2, value2) pairs. One of the nodes in the cluster is special, Master Node it assigns the work to the worker nodes -Slave nodes- and make sure that the job is done by these slaves, it's also responsible for saving the location and size of each intermediate file produced by each map task (more about this later).
- **Shuffling:** In this step the output of the mapping function is being grouped by keys and redistributed in a way that all data with the same key are located on the same node. The output of this step will be (k2, list(v2)).
- **Reducing:** Nodes now process each group of output data by aggregating values of shuffle phase output. The final output will be from the shape of (list (k3, v3)).
- Example of MapReduce algorithm to perform word frequency task, where the input is a text.
- We only need to care about assigning the Map function and the Reduce function the rest of steps will be handled automatically. For the above word frequency example pseudo code would be:

```
function map(String name, String document):
    // name: document name
    // document: document contents
    for each word w in document:
        emit (w, 1)
function reduce(String word, Iterator partialCounts):
    // word: a word
    // partialCounts : a list of aggregated partial counts
    sum = 0
    for each pc in partialCounts :
        sum += pc
    emit (word, sum)
```

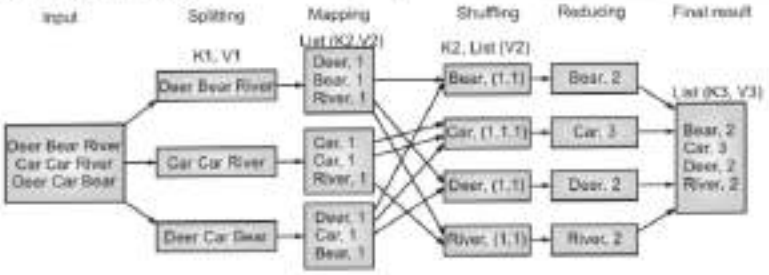


Fig. 2.8 : Overall MapReduce word count process

Combiner

- In the example above where MapReduce framework is applied to find word frequencies in a document (usually very huge document), Map function will emit (word, 1) format for each word, even if the same word appears on the same node twice, so if Deer occurs twice on the same node, the map function will emit (Deer, 1) twice, instead of (Deer, 2).
- By using a combiner on each mapper server, we reduce the data by combining similar keys, the combiner will make shuffling and sorting easier.

MapReduce Reliability

- Since MapReduce program uses hundreds or thousands of machines, it must tolerate machine failures gracefully. As mentioned earlier there is one master node (machine) which is responsible to track the progress of worker nodes. If no response is received from a worker in a certain amount of time, the master will mark the worker as failed. If the machine failed in the map phase, map tasks will be executed again because the results of a map task are stored on the local disk of the node. However, a completed reduce task will not be re-executed because its output is saved in a global file system.
- In this article i will briefly talk about **Hadoop** and **Spark**, as they are build upon MapReduce programming model. (Spark will be explained in details in a future article)

Hadoop

- Hadoop is an open source, Java based framework, uses MapReduce programming model for fast information storage and processing big data, it is being managed by *Apache Software Foundation*.
- Hadoop performs fault tolerance by storing a replicated copy of the data stored in any of the nods in the cluster, so it ensures to re-execute the process if the node broke down. Also , running complex queries in Hadoop is very fast and it won't take seconds due to the using of distributed file system (HDFS) and MapReduce programming for parallel processing.
- As Hadoop is an open-source framework with no license to be purchased, the cost will be significantly low.

Spark

- Spark is also an open source cluster computing platform like Hadoop. However Spark extends MapReduce model to efficiently support more types of computations (like interactive queries and stream processing), this model used by Spark called Resilient Distributed Dataset (RDD).

Spark/Hadoop

There are two prominent differences between Spark and Hadoop:

1. Spark is faster than Hadoop due to in-memory data engine(Ability to run computations in memory)
2. Spark offers simple APIs in Python, Java, Scala and SQL, and rich build-in libraries(like: Sparl SQL, Spark Streaming, MLlib, GraphX). Where several lines written in Hadoop can be summarized to few lines in Spark Python (Pyspark).

2.7 MAPREDUCE EXAMPLE: WORD COUNT, PAGERANK

2.7.1 Word Count

- In Hadoop, **MapReduce** is a computation that decomposes large manipulation jobs into individual tasks that can be executed in parallel across a cluster of servers. The results of tasks can be joined together to compute final results.
- MapReduce consists of two steps:
  1. **Map Function :** It takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (Key-Value pair).

Example – (Map function in Word Count)

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

  2. **Reduce Function :** Takes the output from Map as an input and combines those data tuples into a smaller set of tuples.

Example – (Reduce function in Word Count)

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

Work Flow of the Program

Workflow of MapReduce consists of five steps:

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

Now Let's See the Word Count Program in Java

- Fortunately, we don't have to write all of the above steps, we only need to write the splitting parameter, Map function logic, and Reduce function logic. The rest of the remaining steps will execute automatically.
- Make sure that Hadoop is installed on your system with the Java SDK.

Steps

1. Open Eclipse> File > New > Java Project >(Name it - MRProgramsDemo) > Finish.
2. Right Click > New > Package (Name it - PackageDemo) > Finish.
3. Right Click on Package > New > Class (Name it - WordCount).
4. Add Following Reference Libraries:

Right Click on Project > Build Path> Add External

```
/usr/lib/hadoop-0.20/hadoop-core.jar
Usr/lib/hadoop-0.20/lib/Commons-cli-1.2.jar
```

5. Type the following code:

```
package PackageDemo;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import
org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import
org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
public class WordCount
```

```
{
public static void main(String [] args) throws Exception
{
Configuration c=new Configuration();
String[] files=new
GenericOptionsParser(c,args).getRemainingArgs();
Path input=new Path(files[0]);
Path output=new Path(files[1]);
Job j=new Job(c,"wordcount");
j.setJarByClass(WordCount.class);
j.setMapperClass(MapForWordCount.class);
j.setReducerClass(ReduceForWordCount.class);
j.setOutputKeyClass(Text.class);
j.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(j, input);
FileOutputFormat.setOutputPath(j, output);
System.exit(j.waitForCompletion(true)?0:1);
}
public static class MapForWordCount extends
Mapper&lt;LongWritable, Text, Text, IntWritable&gt;
{
public void map(LongWritable key, Text value, Context
con) throws IOException, InterruptedException
{
String line = value.toString();
String[] words=line.split(",");
for(String word: words )

Text outputKey = new Text(word.toUpperCase().trim());
IntWritable outputValue = new IntWritable(1);
con.write(outputKey, outputValue);



ublic static class ReduceForWordCount extends
educer&lt;Text, IntWritable, Text, IntWritable&gt;

ublic void reduce(Text word,
erable&lt;IntWritable&gt; values, Context con)
rows IOException, InterruptedException

sum = 0;
for(IntWritable value : values)
{
sum += value.get();
}
con.write(word, new IntWritable(sum));
}
}
}
```

The above program consists of three classes:

Driver class (Public, void, static, or main; this is the entry point).

The Map class which extends the public class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT> and implements the Map function.

The Reduce class which extends the public class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT> and implements the Reduce function.

6. Make a jar file

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

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

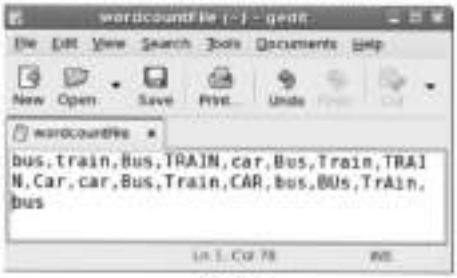


Fig. 2.10

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

```
[training@localhost ~]$ hadoop fs -put
wordcountFile wordCountFile
```

8. Run the jar file:

```
(Hadoop jar jarfilename.jar
packageName.ClassName PathToInputTextFile
PathToOutputDirectry)
[training@localhost ~]$ hadoop jar
MRProgramsDemo.jar PackageDemo.WordCount
wordCountFile MRDir1
```

9. Open the result:

```
[training@localhost ~]$ hadoop fs -ls MRDir1
```

```
Found 3 items
-rw-r--r--   1 training supergroup          0 2016-02-23 03:36 /user/training/MRDir1/_SUCCESS
drwxr-xr-x   - training supergroup          0 2016-02-23 03:36 /user/training/MRDir1/_logs
-rw-r--r--   1 training supergroup         20 2016-02-23 03:36 /user/training/MRDir1/part-r-00000
[training@localhost ~]$ hadoop fs -cat MRDir1/part-r-00000
BUS    7
CAR    4
TRAIN  6
```

2.7.2 PageRank

- PageRank is a way of measuring the importance of website pages. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.
- PageRank (or PR in short) is a recursive algorithm developed by Google founder *Larry Page* to assign a real number to each page in the Web so they can be ranked based on these scores, where the higher the score of a page, the more "important" it is.
- To understand how these importance scores are being calculated, I want you to think of how would you decide if a person on Twitter is important or not? The first thing you would probably check is the number of his/her followers, where the more followers this person has, the higher likelihood he/she is famous(important), then you would maybe check how important this person's followers are, if the president for example is following him/her. The same idea is being applied in *PageRank* to find the importance score of a page, if a movie-page's link is being displayed on many other pages, we say that these pages are pointing or voting to that specify movie-page, and thus:
- The importance of a page depends on the number of other pages pointing (votes) to it.
- You might argue that if what we all do is counting the number of in-links for each page to find their importance, the clothes-seller in the early mentioned example, could simply fool the *PageRank* algorithm (searching engine) to believe that his page is important by creating a "spam farm" of a million pages, each of which linked to his clothes-page. However, as in Twitter, the president following a person gives more weight to the importance measurement than a normal follower, pages in the created "spam farm" would not be given much importance by *PageRank*, since other pages would not link to them, and the spammer will not be able to fool Google.
- In the general case, the PageRank value for any page u can be expressed as:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

i.e. the PageRank value for a page u is dependent on the PageRank values for each page v contained in the set Bu (the set containing all pages linking to page u), divided by the number L(v) of links from page v.

- Suppose consider a small network of four web pages: A, B, C and D. Links from a page to itself, or multiple

outbound links from one single page to another single page, are ignored. PageRank is initialized to the same value for all pages. In the original form of PageRank, the sum of PageRank over all pages was the total number of pages on the web at that time, so each page in this example would have an initial value of 1.

$$PR(A) = \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)}$$

- The damping factor (generally set to 0.85) is subtracted from 1 (and in some variations of the algorithm, the result is divided by the number of documents (N) in the collection) and this term is then added to the product of the damping factor and the sum of the incoming PageRank scores. That is,

$$PR(A) = \frac{1-d}{N} + d\left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \ldots\right)$$

- So any page's PageRank is derived in large part from the PageRanks of other pages. The damping factor adjusts the derived value downward.

PageRank Using MapReduce

- As you can imagine the number of pages on the Web is enormously huge and using a simple approach to recursively update Ranking of millions of pages will be so expensive and time consuming. MapReduce approach will tackle the problem by taking the advantage of running on a cluster (parallelization) and scaled up to very large link-graphs (large number of pages). I've discussed MapReduce frame work at length in a earlier post –
- Now that we know how PageRank calculates the ranking of pages, we can start writing the algorithm on MapReduce. Before we start coding, I would like to mention that the code is written in Pyspark (Python 3).

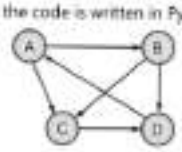


Fig. 2.11

- Let's start by creating a *Spark Session and an adjacency* list representation of the directed graph in Fig. 2.11. Where **'A B C'** means that A node (page) out-links to B & C.

```
import findspark
findspark.init()
findspark.find()
import pyspark
findspark.find()
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession
conf =
pyspark.SparkConf().setAppName('appName').setMaste
r('local')
sc = pyspark.SparkContext(conf=conf)
spark = SparkSession(sc)

# Adjacency list
links = sc.textFile('links.txt')
links.collect()
Out[2]:
['A B C', 'B C D', 'C D', 'D A']
```

- Create key/value pairs, where the key is the name of the page and the value is out-links from the page (Di) and Initiate PageRank values(Ri) as 1/Number of pages.

```
# Key/value pairs
links = links.map(lambda x: (x.split(' ')[0], x.split('
')[1:]))
print(links.collect())

# Find node count
N = links.count()
print(N)

# Create and initialize the ranks
ranks = links.map(lambda node: (node[0],1.0/N))
print(ranks.collect())
[('A', ['B', 'C']), ('B', ['C', 'D']), ('C', ['D']), ('D',
['A'])]
4
[('A', 0.25), ('B', 0.25), ('C', 0.25), ('D', 0.25)]
```

- **Map:** For each node i, calculate vote (Ri/Di) for each out-link of i and propagate to adjacent nodes.
- **Reduce:** For each node i, sum the upcoming votes and update Rank value (Ri).

- Repeat this Map-Reduce step until Rank values converge (stable or within a margin).

```
ITERATIONS=20
for i in range(ITERATIONS):
  # Join graph info with rank info and propagate to all
neighbors rank scores (rank/(number of neighbors)
  # And add up ranks from all in-coming edges
  ranks = links.join(ranks).flatMap(lambda x :
[(i, float(x[1][1])/len(x[1][0])) for i in x[1][0]])\
  reduceByKey(lambda x,y: x+y)
  print(ranks.sortByKey().collect())
[('A', 0.25), ('B', 0.125), ('C', 0.25), ('D', 0.375)]
[('A', 0.375), ('B', 0.125), ('C', 0.1875), ('D', 0.3125)]
[('A', 0.3125), ('B', 0.1875), ('C', 0.25), ('D', 0.25)]
[('A', 0.25), ('B', 0.15625), ('C', 0.25), ('D', 0.34375)]
[('A', 0.34375), ('B', 0.125), ('C', 0.203125), ('D',
0.328125)]
[('A', 0.328125), ('B', 0.171875), ('C', 0.234375),
('D',0.265625)]
[('A', 0.265625), ('B', 0.1640625), ('C', 0.25), ('D',
0.3203125)]
[('A', 0.3203125), ('B', 0.1328125), ('C', 0.21484375),
('D',0.33203125)]
[('A', 0.33203125), ('B', 0.16015625), ('C', 0.2265625),
('D',0.28125)]
[('A', 0.28125), ('B', 0.166015625), ('C', 0.24609375),
('D',0.306640625)]
[('A', 0.306640625), ('B', 0.140625), ('C',
0.2236328125), ('D',0.3291015625)]
[('A', 0.3291015625), ('B', 0.1533203125),
('C',0.2236328125), ('D', 0.2939453125)]
[('A', 0.2939453125), ('B', 0.16455078125),
('C',0.2412109375), ('D', 0.30029296875)]
[('A', 0.30029296875), ('B', 0.14697265625),
('C',0.229248046875), ('D', 0.323486328125)]
[('A', 0.323486328125), ('B', 0.150146484375),
('C',0.2236328125), ('D', 0.302734375)]
('A', 0.302734375), ('B', 0.1617431640625),
'C',0.23681640625), ('D', 0.2987060546875)]
('A', 0.2987060546875), ('B', 0.1513671875),
'C',0.23223876953125), ('D', 0.31768798828125)]
('A', 0.31768798828125), ('B', 0.14935302734375),
'C',0.22503662109375), ('D', 0.30792236328125)]
('A', 0.30792236328125), ('B', 0.158843994140625),
('A', 0.299713134765625), ('B', 0.153961181640625),
('C',0.2333831787109375), ('D',
0.3129425048828125)]
```

2.8 CAP THEOREM

- **The CAP Theorem**, originally introduced as the CAP principle, can be used to explain some of the competing requirements in a distributed system with replication. It is a tool used to makes system designers aware of the trade-offs while designing networked shared-data systems.
- The three letters in CAP refer to three desirable properties of distributed systems with replicated data: **Consistency** (among replicated copies), **Availability** of the system for read and write operations) and **Partition tolerance** in the face of the nodes in the system being partitioned by a network fault).
- The CAP theorem states that it is not possible to guarantee all three of the desirable properties – consistency, availability, and partition tolerance at the same time in a distributed system with data replication.

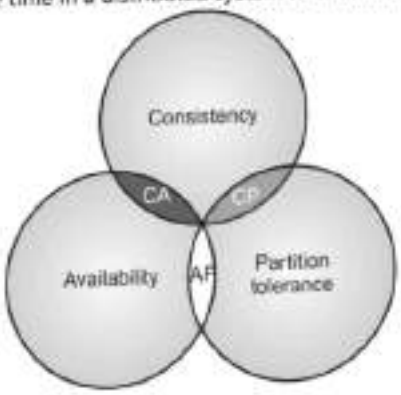


Fig. 2.12

- The theorem states that networked shared-data systems can only strongly support two of the following three properties:

1. **Consistency**

- Consistency means that the nodes will have the same copies of a replicated data item visible for various transactions.
- A guarantee that every node in a distributed cluster returns the same, most recent, successful write. Consistency refers to every client having the same view of the data. There are various types of consistency models.
- Consistency in CAP refers to sequential consistency, a [illegible] form of consistency. This condition states

that all nodes see the same data at the same time. Simply put, performing a read operation will return the value of the most recent write operation causing all nodes to return the same data.

- A system has consistency if a transaction starts with the system in a consistent state, and ends with the system in a consistent state. In this model, a system can (and does) shift into an inconsistent state during a transaction, but the entire transaction gets rolled back if there is an error during any stage in the process. In the image, we have 2 different records ("Bulbasaur" and "Pikachu") at different timestamps.
- The output on the third partition is "Pikachu", the latest input. However, the nodes will need time to update and will not be Available on the network as often.

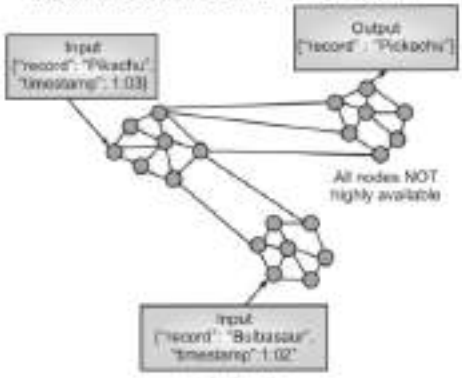


Fig. 2.13

2. Availability

- Availability means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed. Every non-failing node returns a response for all read and write requests in a reasonable amount of time.
- The key word here is every. To be available, every node on (either side of a network partition) must be able to respond in a reasonable amount of time. This condition states that every request gets a response on success/failure. Achieving availability in a distributed system requires that the system remains operational 100% of the time. Every client gets a response, regardless of the state of any individual node in the system. This metric is trivial to measure: either you can submit read/write commands, or you cannot. Hence, the databases are time independent as the nodes need to be available online at all times. This means that, unlike the previous example, we do not know if "Pikachu" or "Bulbasaur" was added first. The output could be either one. Hence why, high availability isn't feasible when analyzing streaming data at high frequency.

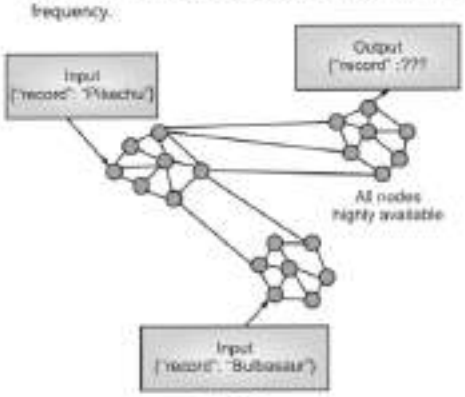


Fig. 2.14

3. Partition Tolerant

- Partition tolerance means that the system can continue operating if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other. That means, the system continues to function and upholds its consistency guarantees in spite of network partitions.
- Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals. This condition states that the system continues to run, despite the number of messages being delayed by the network between nodes.
- A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network. Data records are sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages. When dealing with modern distributed systems, Partition Tolerance is not an option. It's a necessity.

Hence, we have to trade between Consistency and Availability.

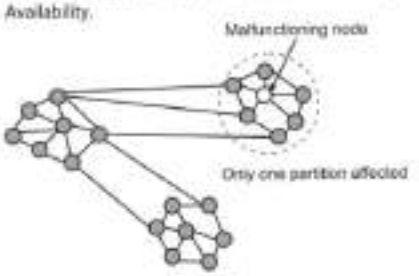


Fig. 2.15

- The use of the word consistency in CAP and its use in ACID do not refer to the same identical concept.
- In CAP, the term consistency refers to the consistency of the values in different copies of the same data item in a replicated distributed system. In ACID, it refers to the fact that a transaction will not violate the integrity constraints specified on the database schema.

2.9 EVENTUAL CONSISTENCY

- Eventual consistency is a theoretical guarantee that, provided no new updates to an entity are made, all reads of the entity will eventually return the last updated value. The Internet Domain Name System (DNS) is a well-known example of a system with an eventual consistency model. DNS servers do not necessarily reflect the latest values but, rather, the values are cached and replicated across many directories over the Internet. It takes a certain amount of time to replicate modified values to all DNS clients and servers. However, the DNS system is a very successful system that has become one of the foundations of the Internet. It is highly available and has proven to be extremely scalable, enabling name lookups to over a hundred million devices across the entire Internet.
- Fig. 2.16 illustrates the concept of replication with eventual consistency. Although replicas are always available to read, some replicas may be inconsistent with the latest write on the originating node, at a particular moment in time. In the diagram, Node A is the originating node and node B and C are the replicas.

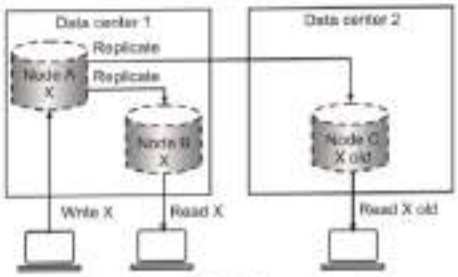


Fig. 2.16

Eventual Consistency

Strong Consistency

- In contrast, traditional relational databases have been designed based on the concept of strong consistency, also called immediate consistency. This means that data viewed immediately after an update will be consistent for all observers of the entity.
- To have strong consistency, developers must compromise on the scalability and performance of their application. Simply put, data has to be locked during the period of update or replication process to ensure that no other processes are updating the same data.
- A conceptual view of the deployment topology and replication process with strong consistency is shown in below Fig. 2.17. In this diagram, you can see how replicas always have values consistent with the originating node, but are not accessible until the update finishes.

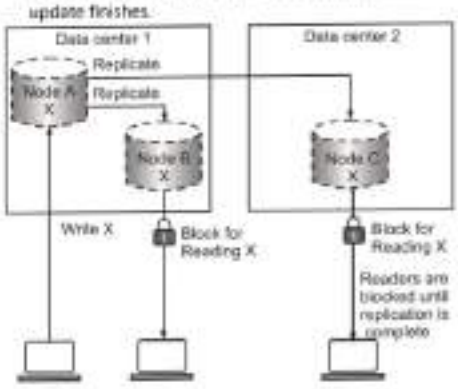


Fig. 2.17

Use Cases:

- Non-relational databases have become popular recently, especially for web applications that require high-scalability and performance with high-availability. Non-relational databases let developers choose an optimal balance between strong consistency and eventual consistency for each application. This allows developers to combine the benefits of both worlds.
- For example, information such as "knowing who in your buddy list is online at given time" or "knowing how many users have +1'd your post" are use cases where strong consistency is not required. Scalability and performance can be provided for these use cases by leveraging eventual consistency. Use cases which require strong consistency include information such as "whether or not a user finished the billing process" or "the number of points a game player earned during a battle session".
- To generalize the examples just given, use cases with very large numbers of entities often suggest that eventual consistency is the best model. If there are a very large number of results in a query, then the user experience may not be affected by the inclusion or exclusion of specific entities. On the other hand, use cases with a small number of entities and a narrow context suggest that strong consistency is required. The user experience will be affected because the context will make users aware of which entities should be included or excluded.

2.10 CONSISTENCY TRADE – OFFS

- Once you start distributing processing then you are forced into a trade-off between the availability of the system and consistency of its data. Given how important availability is to most commercial systems then some inconsistency over data has to be tolerated to ensure that requests can always be processed. However, more often than not, once you start exploring the requirements in more detail then this trade-off doesn't feel so bad.

Identifying the Trade-Offs

- The key trade-offs in a distributed system are best expressed by Eric Brewer's CAP theorem who suggested that you can only provide two of the following three guarantees at the same time:
  1. **Consistency :** Every node sees the same data at the same time.
  2. **Availability :** Every request receives a response.
  3. **Tolerance to Network Partitions :** The system continues to operate despite the kind of arbitrary message loss or failure associated with separate networks.
- Tightly-contained systems such as clustered databases can provide consistency and availability but they have to operate within the same network environment. Consistency implies a need for transaction locking which can be implemented in a more distributed environment, but this will come at the cost of availability. Distributed systems such as DNS or web caches that offer genuine guarantees over availability do so at the cost of some data consistency.
- Given that network partitions are a fact of life for larger distributed systems, you have a choice between relaxing consistency or availability. The challenges of maintaining state are such that systems need to tolerate a degree of inconsistency if they are to ensure that requests can always be answered. Thus, eventual consistency becomes a necessary side-effect of ensuring availability.

ACID V/s BASE

- A key benefit of embracing eventual consistency is that it removes the need to distribute transactions. These have a significant impact on the scalability, performance and availability of systems at scale. They are also very complex to implement.
- This can be difficult for developers to understand, particularly when they have been raised on the certainties of ACID transactions (Atomic, Consistent, Isolated, Durable) where clustered database servers take away all the pain of ensuring data consistency. The fact that two key systems can have a different view of the state of a customer account sets their heads spinning as they try to work out how to resolve the various paradoxes and race conditions this can create.
- ACID's scruffier cousin is BASE (Basic Availability, Soft-state and Eventually consistent) which suggests that data updates can be more relaxed. It's fine to use stale data and not a problem to provide approximate answers. This does not necessarily mean that the hard-

cart has been primed as it can be surprising how often you can live without immediate consistency.

Race Conditions Don't Always Exist in the Business World

- Most of the time, a few seconds of inconsistency is unlikely to make a difference to the underlying process. Developers are often quick to accept them at face value and build in unnecessary cross checking that is not really required by the business requirements.
- This can be illustrated by a simple ecommerce example with distributed billing and fulfilment:
  - ➢ If the order has been shipped then don't allow the user to cancel
  - ➢ Don't let the goods ship if an order has been cancelled.
- On the face of it, a race condition happens when two different users submit orders to ship and cancel at the same time. However, if you drill deeper things become a little more open.
  - ➢ Distance selling regulations mean that the user should be able to cancel after the order has shipped.
  - ➢ A refund doesn't have to be issued the moment an order is cancelled.
  - ➢ Most cancellations happen very soon after the order has been placed rather than a few days later.
- The point is that it's not always safe to assume that everything needs to be enforced and processed immediately. Long-running processes are actually quite common once you get under the skin of them and true race conditions are relatively rare in business processing.

Avoid Getting Bogged Down in Edge Cases

- Distributed systems are often created to meet complex, distributed business processes. Eventual consistency reflects real-world business processes where different actors collaborate on a system over a protracted period. You rarely get cascades of critical events happening in the short time it takes for a distributed platform to achieve consistency.
- Developers using a distributed system have to be aware of which trade-offs have been made. If a system emphasises consistency at the expense of availability then a client will have to manage when to write data. In more available systems they can assume that a write will be expected but will have to be prepared for inconsistencies over the subsequent data reads.
- Eventual consistency does inevitably carry a small risk of concurrency problems such as dirty reads. However, once you fully explore your business processes and understand the life cycle of your data these normally turn out to be very small risks that are associated with edge cases. You can usually allow some level of tolerance so the entire platform does not have to be designed for a tiny fraction of problem transactions that may never actually occur.

2.11 ZOOKEEPER AND PAXOS

2.11.1 ZooKeeper

- ZooKeeper is a coordination service for distributed systems. By providing a robust implementation of a few basic operations, ZooKeeper simplifies the implementation of many advanced patterns in distributed systems.

ZooKeeper as a Distributed File System

- One way of getting to know ZooKeeper is to think of it as a distributed file system. In fact, the way information in ZooKeeper is organized is quite similar to a file system. At the top there is a root simply referred to as. Below the root there are nodes referred to as zNodes, short for ZooKeeper Node, but mostly a term used to avoid confusion with computer nodes. A zNode may act as both a file containing binary data and a directory with more zNodes as sub nodes. As most file systems, each zNode has some meta data. This meta data includes read and write permissions and version information.
- Unlike an ordinary distributed file system, ZooKeeper supports the concepts of ephemeral zNodes and sequential zNodes. An ephemeral zNode is a node that will disappear when the session of its owner ends. A typical use case for ephemeral nodes is when using ZooKeeper for discovery of hosts in your distributed system. Each server can then publish its IP address in an ephemeral node, and should a server loose connectivity with ZooKeeper and fail to reconnect within the session timeout, then its information is deleted.

- Sequential nodes are nodes whose names are automatically assigned a sequence number suffix. This suffix is strictly growing and assigned by ZooKeeper when the zNode is created. An easy way of doing leader election with ZooKeeper is to let every server publish its information in a zNode that is both sequential and ephemeral. Then, whichever server has the lowest sequential zNode is the leader. If the leader or any other server for that matter, goes offline, its session dies and its ephemeral node is removed, and all other servers can observe who is the new leader.
- The pattern with every node creating a sequential and ephemeral zNode is effectively organizing all the nodes in a queue that is observable to all. This is not only useful for leader election, it may just as well be generalized to distributed locks for any purpose with any number of nodes inside the lock.

ZooKeeper as a Message Queue

- Speaking of observing changes, another key feature of ZooKeeper is the possibility of registering watchers on zNodes. This allows clients to be notified of the next update to that zNode. With the use of watchers one can implement a message queue by letting all clients interested in a certain topic register a watcher on a zNode for that topic, and messages regarding that topic can be broadcast to all the clients by writing to that zNode.
- An important thing to note about watchers though, is that they're always one shot, so if you want further updates to that zNode you have to re-register them. This implies that you might loose an update in between receiving one and re-registering, but you can detect this by utilizing the version number of the zNode. If, however, every version is important, then sequential zNodes is the way to go.
- ZooKeeper gives guarantees about ordering. Every update is part of a total ordering. All clients might not be at the exact same point in time, but they will all see every update in the same order. It is also possible to do writes conditioned on a certain version of the zNode so that if two clients try to update the same zNode based on the same version, only one of the updates will be successful. This makes it easy to implement distributed counters and perform partial updates to node data. ZooKeeper even provides a mechanism for submitting multiple update operations in a batch so that they may be executed atomically, meaning that either all or none of the operations will be executed. If you store data structures in ZooKeeper that need to be consistent over multiple zNodes, then the multi-update API is useful; however, it is still not as powerful as ACID transactions in traditional SQL databases. You can't say: "Begin Transaction", as you still have to specify the expected pre-state of each zNode you rely on.

Don't Replace Your Distributed File System and Message Queue

- Although it might be tempting to have one system for everything, you're bound to run into some issues if you try to replace your file servers with ZooKeeper. The first issue is likely to be the zNode limit imposed by the jute.maxbuffer-setting. This is a limit on the size of each zNode, and the default value is one megabyte. In general, it is not recommended to change that setting, simply because ZooKeeper was not implemented to be a large data store. The exception to the rule, which we've experienced at Found, is when a client with many watchers has lost connection to ZooKeeper, and the client library - in this case Curator - attempts to recreate all the watchers upon reconnection. Since the same setting also applies to all messages sent to and from ZooKeeper, we had to increase it to allow Curator to reconnect smoothly for these clients.
- Similarly, you are likely to end up with throughput issues if you use ZooKeeper when what you really need is a message queue, as ZooKeeper is all about correctness and consistency first and speed second. That said, it is still pretty fast when operating normally.
- At Found we use ZooKeeper extensively for discovery, resource allocation, leader election and high priority notifications. Our entire service is built up of multiple systems reading and writing to ZooKeeper.
- One example of such a system is our customer console, the web application that our customers use to create and manage Elasticsearch clusters hosted by Found. One can also think of the customer console as the customers window into ZooKeeper. When a customer creates a new cluster or makes a change to an existing one, this is stored in ZooKeeper as a pending plan change.

- The next step is done by the Constructor, which has a watch in ZooKeeper for new plans. The Constructor implements the plan by deciding how many Elasticsearch instances are required and if any of the existing instances may be reused. The Constructor then updates the instance list for each Elasticsearch server accordingly and waits for the new instances to start.
- On each server running Elasticsearch instances, we have a small application that monitors the servers' instance lists in ZooKeeper and start or stops LXC containers with Elasticsearch instances as needed. When an Elasticsearch instance starts, we use a plugin inside Elasticsearch to report the IP and port to ZooKeeper and discover other Elasticsearch instances to form a cluster with.
- The Constructor waits for the Elasticsearch instances to report back through ZooKeeper with their IP address and port and uses this information to connect with each instance and to ensure they have formed a cluster successfully. And of course, if this does not happen within a certain timeout, then the Constructor will begin rolling back the changes. A common issue that may lead to new nodes having trouble starting is a mis configured Elasticsearch plugin or a plugin that requires more memory than anticipated.
- To provide our customers with high availability and easy failover we have a proxy in front of the Elasticsearch clusters. It is also crucial that this proxy forwards traffic to the correct server, whether changes are planned or not. By monitoring information reported to ZooKeeper by each Elasticsearch instance, our proxy is able to detect whether it should divert traffic to other instances or block traffic altogether to prevent detoration of information in an unhealthy cluster.
- We also use ZooKeeper for leader election among services where this is required. One such example is our backup service. The actual backups are made with the Snapshot and Restore API in Elasticsearch, while the scheduling of the backups is done externally. We decided to co-locate the scheduling of the backups with each Elasticsearch instance. Thus, for customers that pay for high availability, the backup service is also highly available. However, if there are no live nodes in a cluster there is no point in attempting a backup. Since we only want to trigger one backup per cluster and not one per instance, there is a need for coordinating the backup schedulers. This is done by letting them elect a leader for each of the clusters.
- With this many systems relying on ZooKeeper, we need a reliable low latency connection to it. Hence, we run one ZooKeeper cluster per region. While having both client and server in the same region goes a long way in terms of network reliability, you should still anticipate intermittent glitches, especially when doing maintenance to the ZooKeeper cluster itself. At Found, we've learned first hand that it's very important to have a clear idea of what information a client should maintain a local cache of, and what actions a client system may perform while not having a live connection to ZooKeeper.

How Does It Work?

- Three or more independent servers form a ZooKeeper cluster and elect a master. The master receives all writes and publishes changes to the other servers in an ordered fashion. The other servers provide redundancy in case the master fails and offload the master of read requests and client notifications. The concept of ordering is important in order to understand the quality of service that ZooKeeper provides. All operations are ordered as they are received and this ordering is maintained as information flows through the ZooKeeper cluster to other clients, even in the event of a master node failure. Two clients might not have the exact same point in time view of the world at any given time, but they will observe all changes in the same order.

The CAP Theorem

- Consistency, Availability and Partition tolerance are the the three properties considered in the CAP theorem. The theorem states that a distributed system can only provide two of these three properties. ZooKeeper is a CP system with regard to the CAP theorem. This implies that it sacrifices availability in order to achieve consistency and partition tolerance. In other words, if it cannot guarantee correct behavior it will not respond to queries.

Consistency Algorithm

- Although ZooKeeper provides similar functionality to the Paxos algorithm, the core consensus algorithm of ZooKeeper is not Paxos. The algorithm used in ZooKeeper is called ZAB, short for ZooKeeper Atomic Broadcast. Like Paxos, it relies on a quorum for durability. The differences can be summed up as: only one promoter at a time, whereas Paxos may have many promoters of issues concurrently; a much stronger focus on a total ordering of all changes, and every election of a new leader is followed by a synchronization phase before any new changes are accepted. If you want to read up on the specifics of the algorithm, I recommend the paper: "Zab: High-performance broadcast for primary-backup systems".
- What everyone running ZooKeeper in production needs to know, is that having a quorum means that more than half of the number of nodes are up and running. If your client is connecting with a ZooKeeper server which does not participate in a quorum, then it will not be able to answer any queries. This is the only way ZooKeeper is capable of protecting itself against split brains in case of a network partition.
- As much as we love ZooKeeper, we have become so dependent of it that we're also taking care to avoid pushing its limits. Just because we need to send a piece of information from A to B and they both use ZooKeeper does not mean that ZooKeeper is the solution. In order to allow sending anything through ZooKeeper, the value and the urgency of the information has to be high enough in relation to the cost of sending it (size and update frequency).
  - ➢ **Application Logs :** As annoying as it is to be miss logs when you try to debug something, logs are usually the first thing you're willing to sacrifice when your system is pushed to the limit. Hence, ZooKeeper is not a good fit, you actually want something with looser consistency requirements.
  - ➢ **Binaries :** These fellas are just too big and would require tweaking ZooKeeper settings to the point where a lot of corner cases nobody has ever tested are likely to happen. Instead we store binaries on S3 and keep the URL's in ZooKeeper.
  - ➢ **Metrics :** It may work very well to start off with, but in the long run it would pose a scaling issue. If we had been sending metrics through ZooKeeper, it would simply be too expensive to have a comfortable buffer between required and available capacity. That goes for metrics in general, with the exemption being two critical metrics that are also used for application logic: the current disk and memory usage of each node. The latter is used by the proxies to stop indexing when the customer exceeds their disk quota and the first one will at some stage in the future be used to upgrade customers plans when needed.

Atomic Broadcast

- At the heart of ZooKeeper is an atomic messaging system that keeps all of the servers in sync.

Guarantees, Properties, and Definitions

- The specific guarantees provided by the messaging system used by ZooKeeper are the following:

Reliable Delivery

- If a message, m, is delivered by one server, it will be eventually delivered by all servers.

Total Order

- If a message is delivered before message b by one server, a will be delivered before b by all servers. If a and b are delivered messages, either a will be delivered before b or b will be delivered before a.

Causal Order

- If a message b is sent after a message a has been delivered by the sender of b, a must be ordered before b. If a sender sends c after sending b, c must be ordered after b.
- The ZooKeeper messaging system also needs to be efficient, reliable, and easy to implement and maintain. We make heavy use of messaging, so we need the system to be able to handle thousands of requests per second. Although we can require at least k+1 correct servers to send new messages, we must be able to recover from correlated failures such as power outages. When we implemented the system we had little time and few engineering resources, so we needed a protocol that is accessible to engineers and is easy to implement. We found that our protocol satisfied all of these goals.

- Our protocol assumes that we can construct point-to-point FIFO channels between the servers. While similar services usually assume message delivery that can lose or reorder messages, our assumption of FIFO channels is very practical given that we use TCP for communication. Specifically we rely on the following property of TCP:

Ordered Delivery

- Data is delivered in the same order it is sent and a message m is delivered only after all messages sent before m have been delivered. (The corollary to this is that if message m is lost all messages after m will be lost.)

No Message After Close

- Once a FIFO channel is closed, no messages will be received from it.
- FLP proved that consensus cannot be achieved in asynchronous distributed systems if failures are possible. To ensure we achieve consensus in the presence of failures we use timeouts. However, we rely on times for liveness not for correctness. So, if timeouts stop working (clocks malfunction for example) the messaging system may hang, but it will not violate its guarantees.
- When describing the ZooKeeper messaging protocol we will talk of packets, proposals, and messages:

Packet

- A sequence of bytes sent through a FIFO channel

Proposal

- A unit of agreement. Proposals are agreed upon by exchanging packets with a quorum of ZooKeeper servers. Most proposals contain messages, however the NEW_LEADER proposal is an example of a proposal that does not correspond to a message.

Message

- A sequence of bytes to be atomically broadcast to all ZooKeeper servers. A message put into a proposal and agreed upon before it is delivered.
- As stated above, ZooKeeper guarantees a total order of messages, and it also guarantees a total order of proposals. ZooKeeper exposes the total ordering using a ZooKeeper transaction id (zxid). All proposals will be stamped with a zxid when it is proposed and exactly reflects the total ordering. Proposals are sent to all ZooKeeper servers and committed when a quorum of them acknowledge the proposal. If a proposal contains a message, the message will be delivered when the proposal is committed. Acknowledgement means the server has recorded the proposal to persistent storage. Our quorums have the requirement that any pair of quorum must have at least one server in common. We ensure this by requiring that all quorums have size $(n/2+1)$ where n is the number of servers that make up a ZooKeeper service.
- The zxid has two parts: the epoch and a counter. In our implementation the zxid is a 64-bit number. We use the high order 32-bits for the epoch and the low order 32-bits for the counter. Because it has two parts represent the zxid both as a number and as a pair of integers, (*epoch, count*). The epoch number represents a change in leadership. Each time a new leader comes into power it will have its own epoch number. We have a simple algorithm to assign a unique zxid to a proposal: the leader simply increments the zxid to obtain a unique zxid for each proposal. *Leadership activation will ensure that only one leader uses a given epoch, so our simple algorithm guarantees that every proposal will have a unique id.*

ZooKeeper Messaging Consists of Two Phases:

1. **Leader Activation**

- In this phase a leader establishes the correct state of the system and gets ready to start making proposals.
- Leader activation includes leader election. We currently have two leader election algorithms in ZooKeeper: Leader Election and Fast Leader Election (Auth Fast Leader Election is a variant of Fast Leader Election that uses UDP and allows servers to perform a simple form of authentication to avoid IP spoofing). ZooKeeper messaging doesn't care about the exact method of electing a leader has long as the following holds:
  - The leader has seen the highest zxid of all the followers.
  - A quorum of servers have committed to following the leader.
- Of these two requirements only the first, the highest zxid among the followers needs to hold for correct operation. The second requirement, a quorum of followers, just needs to hold with high probability. We are going to recheck the second requirement, so if a

failure happens during or after the leader election and quorum is lost, we will recover by abandoning leader activation and running another election.

- After leader election a single server will be designated as a leader and start waiting for followers to connect. The rest of the servers will try to connect to the leader. The leader will sync up with followers by sending any proposals they are missing, or if a follower is missing too many proposals, it will send a full snapshot of the state to the follower.
- There is a corner case in which a follower that has proposals, U, not seen by a leader arrives. Proposals are seen in order, so the proposals of U will have a zxids higher than zxids seen by the leader. The follower must have arrived after the leader election, otherwise the follower would have been elected leader given that it has seen a higher zxid. Since committed proposals must be seen by a quorum of servers, and a quorum of servers that elected the leader did not see U, the proposals of you have not been committed, so they can be discarded. When the follower connects to the leader, the leader will tell the follower to discard U.
- A new leader establishes a zxid to start using for new proposals by getting the epoch, e, of the highest zxid it has seen and setting the next zxid to use to be (e+1, 0), after the leader syncs with a follower, it will propose a NEW_LEADER proposal. Once the NEW_LEADER proposal has been committed, the leader will activate and start receiving and issuing proposals.
- It all sounds complicated but here are the basic rules of operation during leader activation:
  - A follower will ACK the NEW_LEADER proposal after it has synced with the leader.
  - A follower will only ACK a NEW_LEADER proposal with a given zxid from a single server.
  - A new leader will COMMIT the NEW_LEADER proposal when a quorum of followers have ACKed it.
  - A follower will commit any state it received from the leader when the NEW_LEADER proposal is COMMIT.
  - A new leader will not accept new proposals until the NEW_LEADER proposal has been COMMITED.
- If leader election terminates erroneously, we don't have a problem since the NEW_LEADER proposal will not be committed since the leader will not have quorum. When this happens, the leader and any remaining followers will timeout and go back to leader election.

2. Active Messaging

- In this phase a leader accepts messages to propose and coordinates message delivery.
- ZooKeeper is a holistic protocol. We do not focus on individual proposals, rather look at the stream of proposals as a whole. Our strict ordering allows us to do this efficiently and greatly simplifies our protocol. Leadership activation embodies this holistic concept. A leader becomes active only when a quorum of followers (The leader counts as a follower as well. You can always vote for yourself.) has synced up with the leader, they have the same state. This state consists of all of the proposals that the leader believes have been committed and the proposal to follow the leader, the NEW_LEADER proposal. (Hopefully you are thinking to yourself, Does the set of proposals that the leader believes has been committed included all the proposals that really have been committed? The answer is yes. Below, we make clear why.)
- Leader Activation does all the heavy lifting. Once the leader is coronated he can start blasting out proposals. As long as he remains the leader no other leader can emerge since no other leader will be able to get a quorum of followers. If a new leader does emerge, it means that the leader has lost quorum, and the new leader will clean up any mess left over during her leadership activation.
- ZooKeeper messaging operates similar to a classic two-phase commit.

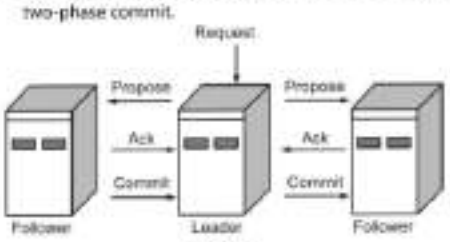


Fig. 2.18

- All communication channels are FIFO, so everything is done in order. Specifically the following operating constraints are observed:

- ➢ The leader sends proposals to all followers using the same order. Moreover, this order follows the order in which requests have been received. Because we use FIFO channels this means that followers also receive proposals in order.
- ➢ Followers process messages in the order they are received. This means that messages will be ACKed in order and the leader will receive ACKs from followers in order, due to the FIFO channels. It also means that if message m has been written to non-volatile storage, all messages that were proposed before m have been written to non-volatile storage.
- ➢ The leader will issue a COMMIT to all followers as soon as a quorum of followers have ACKed a message. Since messages are ACKed in order, COMMITs will be sent by the leader as received by the followers in order.
- ➢ COMMITs are processed in order. Followers deliver a proposals message when that proposal is committed.

2.11.2 Paxos in Hadoop

The Consensus Problem

- Suppose you have a collection of computers and want them all to agree on something. This is what consensus is about; consensus means agreement.
- Consensus comes up frequently in distributed systems design. There are various reasons why we want it: to agree on who gets access to a resource (mutual exclusion), agree on who is in charge (elections), or to agree on a common ordering of events among a collection of computers (e.g., what action to take next, or state machine replication).
- Replication may be the most common use of consensus. We set up collections (clusters) of servers, all of which will have replicated content. This provides fault tolerance: if any server dies, others are still running. It also provides scale: clients can read data from any available server although writing data will generally require the use of all servers and not scale as well. For many applications, though, reads far outnumber writes. To keep the content replicated, we have two design choices. We can funnel all write requests through one coordinator, which will ensure in-order delivery to all replicas or we can send updates to any system but that system will coordinate those updates with its replicas to ensure that all updates are applied in the same order on each replica. In the first case, we need consensus to elect that coordinator. In the second case, we need to run a consensus algorithm for each update to ensure that everyone agrees on the order.
- The consensus problem can be stated in a basic, generic manner: One or more systems may propose some value. How do we get a collection of computers to agree on exactly one of those proposed values?

The Formal Properties for Asynchronous Consensus are:

- **Validity:** Only the proposed values can be decided. If a process decides on some value, v, then some process must have proposed v.
- **Uniform Agreement:** No two correct processes (those that do not crash) can decide on different values.
- **Integrity:** Each process can decide a value at most once.
- **Termination:** All processes will eventually decide on a result.

Paxos

- Paxos is an algorithm that is used to achieve consensus among a distributed set of computers that communicate via an asynchronous network. One or more clients proposes a value to Paxos and we have consensus when a majority of systems running Paxos agrees on one of the proposed values. Paxos is widely used and is legendary in computer science since it is the first consensus algorithm that has been rigorously proved to be correct.
- Paxos simply selects a single value from one or more values that are proposed to it and lets everyone know what that value is. A run of the Paxos protocol results in the selection of single proposed value. If you need to use Paxos to create a replicated log (for a replicated state machine, for example), then you need to run Paxos repeatedly. This is called multi-Paxos. There are some optimizations that could be implemented for multi-Paxos but we will not discuss those here.
- Paxos provides abort able consensus. This means that some processes abort the consensus if there is contention while others decide on the value. Those processes that decide have to agree on the same

value. Aborting allows a process to terminate rather than be blocked indefinitely. When a client proposes a value to Paxos, it is possible that the proposed value might fail if there was a competing concurrent proposal that won. The client will then have to propose the value again to another run of the Paxos algorithm.

Our Assumptions for the Algorithm are:

- **Concurrent Proposals:** One or more systems may propose a value concurrently. If only one system would propose a value then it is clear what the consensus would be. With multiple systems, we need to select one from among those values.
- **Validity:** The chosen value that is agreed upon must be one of the proposed values. The servers cannot just choose a random number.
- **Majority Rule:** Once a majority of Paxos servers agrees on one of the proposed values, we have consensus on that value. This also implies that a majority of servers need to be functioning for the algorithm to run. To survive m failures, we will need 2m+1 systems.
- **Asynchronous Network:** The network is unreliable and asynchronous: messages may get lost or arbitrarily delayed. The network may also get partitioned.
- **Fail-Stop Faults:** Systems may exhibit fail-stop faults. They may restart but need to remember their previous state to make sure they do not change their mind. Failures are not Byzantine.
- **Unicasts:** Communication is point-to-point. There is no mechanism to multicast a message atomically to the set of Paxos servers.
- **Announcement:** Once consensus is reached, the results can be made known to everyone.

Need of Distributed Consensus Protocol

- We have an environment of multiple systems (nodes), connected by a network, where one or more of these nodes may concurrently propose a value (e.g., perform an operation on a server, select a coordinator, add to a log, whatever...). We need a protocol to choose exactly one value in cases where multiple competing values may be proposed.
- We will call the processes that are proposing values proposers. We will also have processes called acceptors that will accept these values and help figure out which one value will be chosen.

Multiple Proposers, Single Acceptor

- The simplest attempt to design a consensus protocol will use a single acceptor. We can elect one of our systems to take on this role. Multiple proposers will concurrently propose various values to a single acceptor. The acceptor chooses one of these proposed values. Unfortunately, this does not handle the case where the acceptor crashes. If it crashes after choosing a value, we will not know what value has been chosen and will have to wait for the acceptor to restart. We want to design a system that will be functional if the majority of nodes are running.

Fault Tolerance: Multiple Acceptors

- To achieve fault tolerance, we will use a collection of acceptors. Each proposal will be sent to at least a majority of these acceptors. If a quorum, or majority, of these acceptors chooses a particular proposed value (proposal) then that value will be considered chosen. Even if some acceptors crash, others are up so we will still know the outcome. If an acceptor crashes after it accepted a proposal, other acceptors know that a value was chosen.
- If an acceptor simply accepts the first proposed value it receives and cannot change its mind (i.e., it chooses that value), it is possible that we may have no majority of accepted values. Depending on the order in which messages arrive at various acceptors, each acceptor may choose a different value or small groups of acceptors may choose different values such that there is no majority of acceptors that chose the same proposed values. This tells us that acceptors may need to change their mind about which accepted value to choose. We want a value to be chosen only when a majority of acceptors accept that value.
- Since we know that an acceptor may need to change its mind, we might consider just having an accepter accept any value given by any proposer. A majority of acceptors may accept one value, which means that value is chosen. However another server may also tell a majority of acceptors to accept another value. Some acceptors will receive both of these proposals because to have a majority means that both sets of proposals have at least one acceptor in common. That means at least one acceptor had to change its mind about what value is ultimately chosen, violating the integrity

property. Once we choose a value, there is no going back.

- To fix this, instead of just having a proposer propose a value, we will first ask it to contact a majority of acceptors and check whether a value has already been chosen. If it has, then the proposer must propose that chosen value to the other acceptors. To implement this, we will need a two phase protocol: check first and then provide a value.
- Asking a proposer to check is not sufficient. Another proposer may come along after the checking phase and propose a different value. What if that second value is accepted by a majority and then the acceptors receive requests from the first proposer to accept its value? We again end up with two chosen values. The protocol needs to be modified to ensure that once we accept a value, we will abort competing proposals. To do this, Paxos will impose an ordering on proposals. Newer proposals will take precedence over older ones. If that first proposer tries to finish its protocol, its requests will fail.

Paxos Working

The Case of Characters

Paxos has Three Entities:

1. **Proposers:** Receive requests (values) from clients and try to convince acceptors to accept their proposed values.
2. **Acceptors:** Accept certain proposed values from proposers and let proposers know if something else was accepted. A response from an acceptor represents a vote for a particular proposal.
3. **Learners:** Announce the outcome.

- In practice, a single node may run proposer, acceptor, and learner roles. It is common for Paxos to coexist with the service that requires consensus (e.g., distributed storage) on a set of replicated servers, with each server taking on all three roles rather than using separate servers dedicated to Paxos. For the sake of discussing the protocol, however, we consider these to be independent entities.

What Paxos Does

- A client sends a request to any Paxos proposer. The proposer then runs a two-phase protocol with the acceptors. Paxos is a majority-wins protocol. A majority avoids split-brain problems and ensures that if you made a proposal and asked over 50% of the systems if somebody else made a proposal and they all reply no then you know for certain that no other system could have asked over 50% of the systems received the same answer. Because of this Paxos requires a majority of its servers to be running for the algorithm to terminate. A majority ensures that there is at least one node in common from one majority to another if servers die and restart. The system requires 2m+1 servers to tolerate the failure of m servers. As we shall see, Paxos requires a majority of acceptors. We can have one or a smaller number of proposers and learners.
- Paxos acceptors cannot forget what they accepted, so they need to keep track of the information they received from proposers by writing it to stable storage. This is storage such as flash memory or disk, whose contents can be retrieved even if the process or system is restarted.

The Paxos Protocol (Initial Version)

Paxos is a two-phase protocol, meaning that the proposers interact with the acceptors twice. At a high level:

Phase 1

A proposer asks all the working acceptors whether anyone already received a proposal. If the answer is no, propose a value.

Phase 2

If a majority of acceptors agree to this value then that is our consensus.

- Let's look at the protocol in a bit more detail now. Right now, we will examine the mostly failure-free case. Later, we will augment the algorithm to correct for other failures.
- When a proposer receives a client request to reach consensus on a value, the proposer must create a proposal number. This number must have two properties:
  (i) It must be unique. No two proposers can come up with the same number. An easy way of doing this is to use a global process identifier for the least significant bits of the number. For example, instead of an ID=12, node 3 will generate ID=12.3 and node 2 will generate 12.2.

(ii) It must be bigger than any previously used identifier used in the cluster. A proposer may use an incrementing counter or use a nanosecond-level timestamp to achieve this. If the number is not bigger than one previously used, the proposer will find out by having its proposal rejected and will have to try again.

Phase 1: PREPARE-PROMISE

- A proposer receives a consensus request for a VALUE from a client. It creates a unique proposal number, ID, and sends a PREPARE(ID) message to at least a majority of acceptors.
- Each acceptor that receives the PREPARE message looks at the ID in the message and decides:

```
Is this ID bigger than any round I have previously received?
  If yes
    store the ID number, max_id = ID
    respond with a PROMISE message
  If no
    do not respond (or respond with a "fail" message)
```

- If the proposer receives a PROMISE response from a majority of acceptors, it now knows that a majority of them are willing to participate in this proposal. The proposer can now proceed with getting consensus. Each of these acceptors made a promise that no other proposal with a smaller number can make it to consensus.

Phase 2: PROPOSE-ACCEPT

- If a proposer received a PROMISE message from the majority of acceptors, it now has to tell the acceptors to accept that proposal. If not, it has to start over with another round of Paxos.
- In this phase of the protocol, the proposer tells all the acceptors (that are live) what value to accept. It sends:

```
PROPOSE(ID, VALUE)
```

- to a majority or all of the acceptors. Each acceptor now decides whether to accept the proposal. It accepts the proposal if the ID number of the proposal is still the largest one that it has seen. Recall that an acceptor promised not to accept proposals from PREPARE messages with smaller numbers but can and will accept proposals with higher numbers. The logic the acceptor uses is:

```
Is the ID the largest I have seen so far, max_id == N?
  if yes
    reply with an ACCEPTED message & send
    ACCEPTED(ID, VALUE) to all learners
  if no
    do not respond (or respond with a "fail" message)
```

- The ACCEPTED message can be sent back to the proposer as well as to the learners so they can propagate the value to wherever its action is needed (e.g., append to a log, modify a replicated database, ...). When the proposer or learner receives a majority of accept messages then it knows that consensus has been reached on the value.
- To summarize, in the first phase, the proposer finds out that no promises have been made to higher numbered proposals. In the second phase, the proposer asks the acceptors to accept the proposal with a specific value. As long as no higher numbered proposals have arrived during this time, the acceptor responds back that the proposal has been accepted.

Handling Failures and Fixing the Protocol

- Suppose a proposer sends a PREPARE message with a lower ID than was previously promised by the majority of acceptors. Some acceptors, those that did not receive the earlier higher ID, might reply with a PROMISE. But a majority will not and the proposer will get fail messages (negative acknowledgements) or simply time out waiting and have to retry with a higher ID.
- On the other hand, suppose some proposer now sends a PREPARE message with a higher ID (higher_ID). The acceptors only promised to ignore messages with lower numbers. We need to modify what the acceptor does in Phase 1.
- If the acceptor has not yet accepted any proposal (that is, it responded with a PROMISE to a past proposal but not an ACCEPTED), it will simply respond back to the proposer with a PROMISE. However, if the acceptor has already accepted an earlier message it responds to the proposer with a PROMISE that contains the accepted ID and its corresponding value.
- The acceptor needs to keep track of proposals that it has accepted (in phase 2). If it already sent an ACCEPT message it cannot change its mind but it will inform

the proposer that it already accepted an earlier proposal. The logic for the acceptor in the first phase is now:

Acceptor Receives a PREPARE(ID) Message:

```
is this ID bigger than any round I have previously received?
  if yes
    store the ID number, max_id = ID
    respond with a PROMISE(ID) message
  if no
      did I already accept a proposal?
      if yes
        respond with a PROMISE(ID, accepted_ID, accepted_VALUE) message
      if no
        do not respond (or respond with a "fail" message)
```

- When the proposer receives responses from acceptors at the end of phase 1, it needs to get a majority of responses for its proposal ID to continue with the protocol. It also has to look through each one of those responses to see if there were any accepted proposals. If there were none, then the proposer is free to propose its own value. If there were, then proposer is obligated to pick the value corresponding to the highest-numbered accepted proposal. This is how other acceptors get to find out about accepted proposals that they may have missed. The logic at the proposer at the start of phase 2 is now:

Proposer Receives PROMISE(ID, [VALUE]) Messages:

```
do I have PROMISE responses from a majority of acceptors?
if yes
  do any responses contain accepted values (from other proposals)?
  if yes
    pick the value with the highest accepted ID
    send PROPOSE(ID, accepted_VALUE) to at least a majority of acceptors
  if no
    we can use our proposed value
    send PROPOSE(ID, VALUE) to at least a majority of acceptors
```

Our Full Paxos Protocol Now Looks like this:

Phase 1a: Proposer (PREPARE)

A proposer initiates a PREPARE message, picking a unique, ever-incrementing value.

```
ID = cnt++;
send PREPARE(ID)
```

Phase 1b: Acceptor (PROMISE)

An acceptor receives a PREPARE(ID) message:

```
if (ID <= max_id)
  do not respond (or respond with a "fail" message)
else
  max_id = ID    // save highest ID we've seen so far
  if (proposal_accepted == true) // was a proposal already accepted?
  respond: PROMISE(ID, accepted_ID, accepted_VALUE)
  else
    respond: PROMISE(ID)
```

Phase 2a: Proposer (PROPOSE)

- The proposer now checks to see if it can use its proposal or if it has to use the highest-numbered one it received from among all responses:

```
did I receive PROMISE responses from a majority of acceptors?
if yes
  do any responses contain accepted values (from other proposals)?
  if yes
    val = accepted_VALUE    // value from PROMISE message with the highest accepted ID
  if no
    val = VALUE    // we can use our proposed value
  send PROPOSE(ID, val) to at least a majority of acceptors
```

Phase 2b: Acceptor (ACCEPT)

- Each acceptor receives a PROPOSE(ID, VALUE) messag from a proposer. If the ID is the highest number it h processed then accept the proposal and propagate t value to the proposer and to all the learners.

```
if (ID == max_id) // is the ID the largest I have seen so far?
   proposal_accepted = true
                        // note that we accepted a proposal
   accepted_ID = ID  // save the accepted proposal number
   accepted_VALUE = VALUE
                         // save the accepted proposal data
   respond: ACCEPTED(ID, VALUE) to the proposer and all learners
else
   do not respond (or respond with a "Fail" message)
```

- If a majority of acceptors accept ID, value then consensus is reached. Consensus is on the value, not necessarily the ID.

Failure Examples

Acceptor Fails in Phase 1

- Suppose an acceptor fails during phase 1. That means it will not return a PROMISE message. As long as the proposer still gets responses from a majority of acceptors, the protocol can continue to make progress.

Acceptor Fails in Phase 2

- Suppose an acceptor fails during phase 2. That means it will not be able to send back an ACCEPTED message. This is also not a problem as long as enough of the acceptors are still alive and will respond so that the proposer or learner receives responses from a majority of acceptors.

Proposer Fails in the Prepare Phase

- If the proposer fails before it sent any messages, then it is the same as if it did not run at all.
- What if the proposer fails after sending one or more PREPARE msgs? An acceptor would have sent PROMISE responses back but no ACCEPT messages would follow, so there would be no consensus. Some other node will eventually run its version of Paxos and run as a proposer, picking its own ID. If the higher ID number works, then the algorithm runs. Otherwise, the proposer would have its request rejected and have to pick a higher ID number.
- What if the proposer fails during the ACCEPT phase? At least one ACCEPT message was sent. Some another node proposes a new message with PREPARE(higher-ID). The acceptor responds by telling that proposer that an earlier proposal was already accepted:

```
PROMISE(higher-ID, <old_ID, Value>)
```

- If a proposer gets any PROMISE responses with a value then it must choose the response with the highest accepted ID and change its own value. It sends out:

```
ACCEPT(higher-ID, Value)
```

- If proposer fails in the ACCEPT phase, any proposer that takes over finishes the job of propagating the old value that was accepted.
- Suppose a majority of acceptors receive PREPARE messages for some ID followed by PROPOSE messages. That means a majority of acceptors accepted for that ID. No PREPARE messages with lower IDs can now be accepted by a majority. To do so would require a majority of promises for the lower numbered ID but we already made promises for the higher numbered ID. No PROPOSE messages with higher IDs and different values will be accepted by a majority either. At least one acceptor will know the ID and corresponding value that it accepted, which it will propagate back to the proposer. You can have proposals with higher IDs, but the proposer is obligated to give them the same value. If a proposer sends:

```
PREPARE(high-ID)
```

- At least one acceptor will respond back with the previously accepted ID and value:

```
PROMISE(high-ID, accepted-ID, value)
```

- The proposer will have to return back a PROPOSE(high-ID, value). This is how proposers & learners can learn about what was accepted and ultimately create a majority outcome.

Proposer Fails in the ACCEPT Phase

- Here, a proposer that takes over does not know it is taking over a pending consensus. It simply proposes a value but does not realize that the consensus protocol was already in progress. There are two cases to consider here:
- The proposer does not get any responses from a majority of acceptors that contain an old proposal ID

and corresponding value. That means there has been no majority agreement yet. The proposer then just executes normally and finishes its protocol.

- The proposer that takes over knows it is taking over a pending consensus because it gets at least one response that contains an accepted proposal # and value. It just executes using that previous value and finishes the protocol.

The Leader

- With Paxos, there is a chance of reaching livelock where the algorithm makes no progress. There are approaches to break this livelock, such as using random exponentially-increasing delays. However, a common solution is to select a single proposer to handle all incoming requests. This elected proposer will be called the leader. Selecting a leader requires running an election among the proposers. While we can run Paxos to select a leader, we can also ensure that we do not encounter livelock while running an election to choose a leader so we can avoid livelock. We can use an algorithm such as the Bully algorithm to choose a leader. Note that Paxos is still designed to be fault tolerant. The leader is not a requirement and requests may still be made via other proposers or other proposers may step in at any time.
- Bully algorithm recap: A node that starts an election sends its server ID to all of its peers. If it gets a response from any peer with a higher ID, it will not be the leader. If all responses have lower IDs than it becomes the leader. If a node receives an election message from a node with a lower-numbered ID, then the node starts its own election. Eventually, the node with the highest ID will be the winner.

Engineering Paxos for the Real World

- Paxos defines a protocol for single-value distributed consensus but does not consider other factors that are needed to get Paxos to run in real environments. Some of these factors are:

Single Run V/s. Multi-Run

- Paxos yields consensus for a single value. Much of the time, we need to make consensus decisions repeatedly, such as keeping a replicated log or state machine synchronized. For this we need to run Paxos multiple times. This environment is called multi-Paxos.

Group Management

- The cluster of systems that are running Paxos needs to be administered. We need to be able to add systems to the group, remove them, and detect if any processes, entire systems, or network links are dead. Each proposer needs to know the set of acceptors so it can communicate with them and needs to know the learners (if those are present). Paxos is a fault-tolerant protocol but the mechanisms for managing the group are outside of its scope. We can turn to something like the group membership service of Isis virtual synchrony to track this.

Byzantine Failures

- We assumed that none of the systems running Paxos suffer Byzantine failures. That is, either they run and communicate correctly or they stay silent. In real life, however, Byzantine failures do exist. We can guard against network problems with mechanisms such as checksums or, if we fear malicious interference, digital signatures. However, we do need to worry about a misbehaving proposer that may inadvertantly set its proposal ID to infinity (e.g., INFINITY in math.h in C or math.inf in Python if using floats; otherwise INT_MAX in C or sys.maxint in Python). This puts the Paxos protocol into a state where acceptors will have to reject any other proposal.

Location of Servers

- Possibly the most common use of Paxos is in implementing replicated state machines such as a distributed storage system. To ensure that replicas are consistent, incoming operations must be processed in the same order on all systems. A common way of doing this is to use a replicated log. That is, each of the servers will maintain a log that is sequenced identically to the logs on the other servers. A consensus algorithm will decide the next value that goes on the log. Then, each server simply processes the log in order and applies the requested operations.
- In these environments, and most others, each server also serves as a Paxos node, running the functions of proposer, acceptor, and learner. A client can send a message to any server, which invokes some operation that updates the replicated state machine (e.g., replicated storage service). In this case, the proposer

will often be co-resident with the server that the client contacted. The request from the user is the value for a proposal that will be originated by that node. The proposer on that node manages its proposal ID numbers and sends proposals to all the acceptors it can reach, including itself. These acceptors accept the requests via the Paxos protocol. When the node knows that consensus has been reached, the operation can be applied to the log and propagated to the other servers. If another node tries to propose something concurrently, one of the proposals will be told that another proposal has already been accepted and it will get a different value so it will carry out the Paxos protocol to achieve consensus on that accepted value. It will then have to try again later, running Paxos again, to get its own value into the log.

2.12 CASSANDRA

- Apache Cassandra is a highly scalable, high-performance distributed database designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. It is a type of NoSQL database. Let us first understand what a NoSQL database does.

NoSQL Database

- A NoSQL database (sometimes called as Not Only SQL) is a database that provides a mechanism to store and retrieve data other than the tabular relations used in relational databases. These databases are schema-free, support easy replication, have simple API, eventually consistent, and can handle huge amounts of data.
- The primary objective of a NoSQL database is to have
  - Simplicity of design,
  - Horizontal scaling, and
  - Finer control over availability.
- NoSql databases use different data structures compared to relational databases. It makes some operations faster in NoSQL. The suitability of a given NoSQL database depends on the problem it must solve.

NoSQL vs. Relational Database

- The following table lists the points that differentiate a relational database from a NoSQL database.

Relational Database	NoSql Database
Supports powerful query language.	Supports very simple query language.
It has a fixed schema.	No fixed schema.
Follows ACID (Atomicity, Consistency, Isolation, and Durability).	It is only "eventually consistent".
Supports transactions.	Does not support transactions.

- Besides Cassandra, we have the following NoSQL databases that are quite popular –
  - **Apache HBase :** HBase is an open source, non-relational, distributed database modeled after Google's BigTable and is written in Java. It is developed as a part of Apache Hadoop project and runs on top of HDFS, providing BigTable-like capabilities for Hadoop.
  - **MongoDB :** MongoDB is a cross-platform document-oriented database system that avoids using the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas making the integration of data in certain types of applications easier and faster.

What is Apache Cassandra?

- Apache Cassandra is an open source, distributed and decentralized/distributed storage system (database), for managing very large amounts of structured data spread out across the world. It provides highly available service with no single point of failure.

Listed below are some of the notable points of Apache Cassandra –

- It is scalable, fault-tolerant, and consistent.
- It is a column-oriented database.
- Its distribution design is based on Amazon's Dynamo and its data model on Google's Bigtable.
- Created at Facebook, it differs sharply from relational database management systems.
- Cassandra implements a Dynamo-style replication model with no single point of failure, but adds a more powerful "column family" data model.

➢ Cassandra is being used by some of the biggest companies such as Facebook, Twitter, Cisco, Rackspace, ebay, Twitter, Netflix, and more.

Features of Cassandra

Cassandra has become so popular because of its outstanding technical features. Given below are some of the features of Cassandra:

- **Elastic Scalability :** Cassandra is highly scalable; it allows to add more hardware to accommodate more customers and more data as per requirement.
- **Always on Architecture :** Cassandra has no single point of failure and it is continuously available for business-critical applications that cannot afford a failure.
- **Fast Linear-Scale Performance :** Cassandra is linearly scalable, i.e., it increases your throughput as you increase the number of nodes in the cluster. Therefore it maintains a quick response time.
- **Flexible Data Storage :** Cassandra accommodates all possible data formats including: structured, semi-structured, and unstructured. It can dynamically accommodate changes to your data structures according to your need.
- **Easy Data Distribution :** Cassandra provides the flexibility to distribute data where you need by replicating data across multiple data centers.
- **Transaction Support :** Cassandra supports properties like Atomicity, Consistency, Isolation, and Durability (ACID).
- **Fast Writes :** Cassandra was designed to run on cheap commodity hardware. It performs blazingly fast writes and can store hundreds of terabytes of data, without sacrificing the read efficiency.

History of Cassandra

- Cassandra was developed at Facebook for inbox search.
- It was open-sourced by Facebook in July 2008.
- Cassandra was accepted into Apache Incubator in March 2009.
- It was made an Apache top-level project since February 2010.

The design goal of Cassandra is to handle big data workloads across multiple nodes without any single point of failure. Cassandra has peer-to-peer distributed system across its nodes, and data is distributed among all the nodes in a cluster.

- All the nodes in a cluster play the same role. Each node is independent and at the same time interconnected to other nodes.
- Each node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster.
- When a node goes down, read/write requests can be served from other nodes in the network.

Data Replication in Cassandra

- In Cassandra, one or more of the nodes in a cluster act as replicas for a given piece of data. If it is detected that some of the nodes responded with an out-of-date value, Cassandra will return the most recent value to the client. After returning the most recent value, Cassandra performs a **read repair** in the background to update the stale values.
- The following Fig. 2.19 shows a schematic view of how Cassandra uses data replication among the nodes in a cluster to ensure no single point of failure.

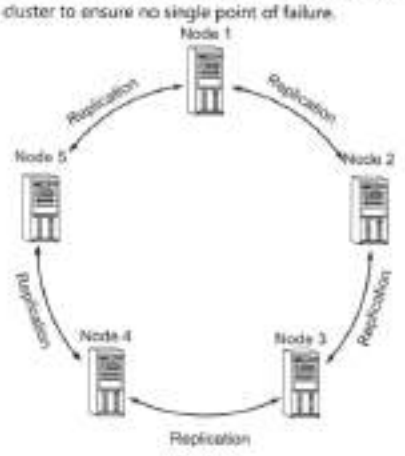


Fig. 2.19

Note : Cassandra uses the **Gossip Protocol** in the background to allow the nodes to communicate with each other and detect any faulty nodes in the cluster.

Components of Cassandra

The key components of Cassandra are as follows –

- **Node :** It is the place where data is stored.
- **Data Center :** It is a collection of related nodes.
- **Cluster :** A cluster is a component that contains one or more data centers.
- **Commit Log :** The commit log is a crash-recovery mechanism in Cassandra. Every write operation is written to the commit log.
- **Mem-Table :** A mem-table is a memory-resident data structure. After commit log, the data will be written to the mem-table. Sometimes, for a single-column family, there will be multiple mem-tables.
- **SSTable :** It is a disk file to which the data is flushed from the mem-table when its contents reach a threshold value.
- **Bloom Filter :** These are nothing but quick, nondeterministic, algorithms for testing whether an element is a member of a set. It is a special kind of cache. Bloom filters are accessed after every query.

Cassandra Query Language

- Users can access Cassandra through its nodes using Cassandra Query Language (CQL). CQL treats the database **(Keyspace)** as a container of tables. Programmers use **cqlsh:** a prompt to work with CQL or separate application language drivers.
- Clients approach any of the nodes for their read-write operations. That node (coordinator) plays a proxy between the client and the nodes holding the data.

Write Operations

- Every write activity of nodes is captured by the **commit logs** written in the nodes. Later the data will be captured and stored in the **mem-table.** Whenever the mem-table is full, data will be written into the **SStable** data file. All writes are automatically partitioned and replicated throughout the cluster. Cassandra periodically consolidates the SSTables, discarding unnecessary data.

Read Operations

- During read operations, Cassandra gets values from the mem-table and checks the bloom filter to find the appropriate SSTable that holds the required data.
- The data model of Cassandra is significantly different from what we normally see in an RDBMS. This chapter provides an overview of how Cassandra stores its data.

Cluster

- Cassandra database is distributed over several machines that operate together. The outermost container is known as the Cluster. For failure handling, every node contains a replica, and in case of a failure, the replica takes charge. Cassandra arranges the nodes in a cluster, in a ring format, and assigns data to them.

Keyspace

- Keyspace is the outermost container for data in Cassandra. The basic attributes of a Keyspace in Cassandra are –
  - ➢ **Replication Factor :** It is the number of machines in the cluster that will receive copies of the same data.
  - ➢ **Replica Placement Strategy :** It is nothing but the strategy to place replicas in the ring. We have strategies such as **simple strategy** (rack-aware strategy), **old network topology strategy** (rack-aware strategy), and **network topology strategy** (datacenter-shared strategy).
  - ➢ **Column Families :** Keyspace is a container for a list of one or more column families. A column family, in turn, is a container of a collection of rows. Each row contains ordered columns. Column families represent the structure of your data. Each keyspace has at least one and often many column families.

The Syntax of Creating a Keyspace is as Follows :

```
CREATE KEYSPACE Keyspace name
WITH replication = {'class': 'SimpleStrategy',
'replication_factor' : 3};
```

The following illustration shows a schematic view of a Keyspace.

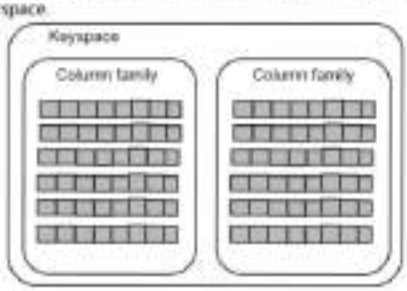


Fig. 2.20

Column Family

- A column family is a container for an ordered collection of rows. Each row, in turn, is an ordered collection of columns. The following table lists the points that differentiate a column family from a table of relational databases.

Relational Table	Cassandra Column Family
A schema in a relational model is fixed. Once we define certain columns for a table, while inserting data, in every row all the columns must be filled at least with a null value.	In Cassandra, although the column families are defined, the columns are not. You can freely add any column to any column family at any time.
Relational tables define only columns and the user fills in the table with values.	In Cassandra, a table contains columns, or can be defined as a super column family.

A Cassandra column family has the following attributes –

- **keys_cached** – It represents the number of locations to keep cached per SSTable.
- **rows_cached** – It represents the number of rows whose entire contents will be cached in memory.
- **preload_row_cache** – It specifies whether you want to pre-populate the row cache.

Note : Unlike relational tables where a column family's schema is not fixed, Cassandra does not force individual rows to have all the columns.

The following Fig. 2.21 shows an example of a Cassandra column family.

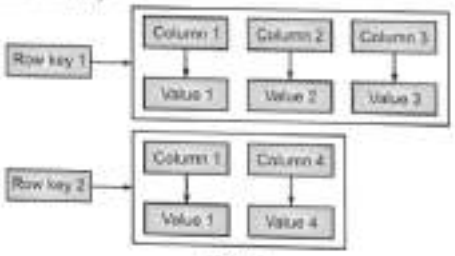


Fig. 2.21

Column

- A column is the basic data structure of Cassandra with three values, namely key or column name, value, and a time stamp. Given below is the structure of a column.

Column		
name : byte[]	value : byte[]	clock : clock[]

Super Column

- A super column is a special column, therefore, it is also a key-value pair. But a super column stores a map of sub-columns.
- Generally column families are stored on disk in individual files. Therefore, to optimize performance, it is important to keep columns that you are likely to query together in the same column family, and a super column can be helpful here. Given below is the structure of a super column.

Super Column	
name : byte[]	Cols : map<byte[ ], column>

Data Models of Cassandra and RDBMS

- The following table lists down the points that differentiate the data model of Cassandra from that of an RDBMS.

RDBMS	Cassandra
RDBMS deals with structured data.	Cassandra deals with unstructured data.
It has a fixed schema.	Cassandra has a flexible schema.
In RDBMS, a table is an array of arrays. (ROW x COLUMN)	In Cassandra, a table is a list of "nested key-value pairs". (ROW x COLUMN key x COLUMN value)
Database is the outermost container that contains data corresponding to an application.	Keyspace is the outermost container that contains data corresponding to an application.
Tables are the entities of a database.	Tables or column families are the entity of a key space.
Row is an individual record in RDBMS.	Row is a unit of replication in Cassandra.
Column represents the attributes of a relation.	Column is a unit of storage in Cassandra.
RDBMS supports the concepts of foreign keys, joins.	Relationships are represented using collections.

2.13 CASSANDRA INTERNALS

Cassandra is a NoSQL database that belongs to the Column Family NoSQL database category. It's an Apache project and it has an Enterprise version maintained by DataStax. Cassandra is written in Java and it's mainly used for time-series data such as metrics, IoT (Internet of Things), logs, chat messages, and so on. It is able to handle a massive amount of writes and reads and scale to thousands of nodes. Let's list out here some important Cassandra characteristics. Basically, Cassandra...

- Mix ideas from Google's Big Table and Amazon's Dynamo.
- Is based on peer-to-peer architecture. Every node is equal (can perform both reads and writes), therefore there is no master or slave node, that is, there is no master single point of failure.
- Does automatic partitioning and replication.
- Has tunable write and read consistency for both read and write operations.
- Is able to horizontally scale keeping linear scalability for both reads and writes.
- Handles inter-node communication through the Gossip protocol.
- Handles client communication through the CQL (Cassandra Query Language), which is very similar to SQL.

Coordinator

- When a request is sent to any Cassandra node, this node acts as a proxy for the application (actually, the Cassandra driver) and the nodes involved in the request flow. This proxy node is called as the coordinator. The coordinator is responsible for managing the entire request path and to respond back to the client.

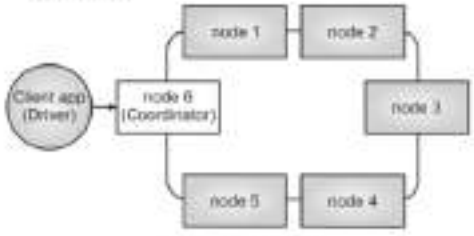


Fig. 2.22 : Coordinator

- Besides, sometimes when the coordinator forwards a write request to the replica nodes, they may happen to be unavailable at this very moment. In this case, the coordinator plays an important role implementing a mechanism called Hinted Handoff, which will be described in details later.

Partitioner

- Basically, for each node in the Cassandra cluster (Cassandra ring) is assigned a range of tokens as shown in Fig. 2.23 for a Six-node cluster (with imaginary tokens, of course).

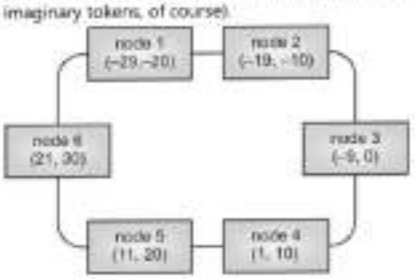


Fig. 2.23 : Token Ranges

- Cassandra distributes data across the cluster using a Consistent Hashing algorithm and, starting from version 1.2, it also implements the concept of virtual nodes (vnodes), where each node owns a large number of small token ranges in order to improve token reorganization and avoid hotspots in the cluster, that is, some nodes storing much more data than the others. Virtual nodes also allow to add and remove nodes in the cluster more easily and manages the token assignment automatically for you so that you can enjoy a nice coffee when adding or removing a node instead of calculating and assigning new token ranges for each node (which is a very error-prone operation, by the way).
- Well, that said, the partitioner is the component responsible for determining how to distribute the data across the nodes in the cluster given the partition key of a row. Basically, it is a hash function for computing a token given the partition key.
- Once the partitioner applies the hash function to the partition key and gets the token, it knows exactly which node is going to handle the request.

- Let's consider a simple example: suppose a request is issued to node6 (that is, node6 is the coordinator for this request) with a row containing the partition key "jorge_acetozi". Suppose the partitioner applies the hash function to the partition key "jorge_acetozi" and gets the token -17. As Fig. 2.24 shows, node2 token ranges include -17, so this node will be the one handling the request.

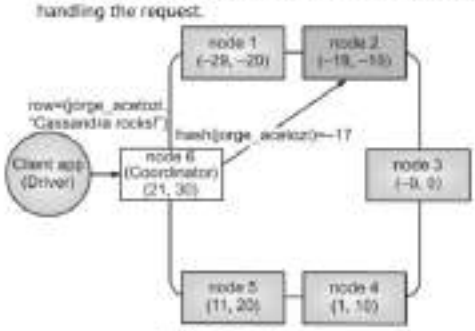


Fig. 2.24 : Partitioner

- Cassandra offers three types of partitioners: Murmur Partitioner (which is the default), Random Partitioner, and Byte Ordered Partitioner.

Replication

- Life would be much easier if...
  - ➢ Nodes never fail
  - ➢ Networks had no latency
  - ➢ People did not stumble on cables
  - ➢ Amazon did not restart your instances
  - ➢ Full GC meant "Full Guitar Concert"
- And so on. Unfortunately, these things happen all the time and you already chose a software engineer career (your mother used to advise you to study hard and to become a doctor, but you chose to keep playing Counter-Strike instead. Now you are a software engineer, know what AK-47 means and have to care about stuff like that).
- Fortunately, Cassandra offers automatic data replication and keeps your data redundant throughout different nodes in the cluster. This means that (in certain levels) you can even resist to node failure scenarios and your data would still be safe and available. But everything comes at a price, and the price of replication is consistency.

Replication Strategy

- Basically, the coordinator uses the Replication Strategy to find out which nodes will be the replica nodes for a given request.

There are Two Replication Strategies Available:

1. **Simple Strategy:** Used for a single data center deployment (not recommended for production environment). It doesn't consider the network topology. Basically, it just takes the partitioner's decision (that is, the node that will handle the request first based on the token range) and places the remaining replicas clockwise in relation to this node. For example, in Fig. 2.24, if the table replication factor was three, which nodes would have been chosen by the Simple Strategy to act as replicas (besides node2, which was already chosen by the partitioner)? That's correct, node3 and node4! What if the replication factor was 4? Well, then node5 would also be included.
2. **Network Topology Strategy:** Used for multiple data centers deployment (recommended for production environment). It also takes the partitioner's decision and places the remaining replicas clockwise, but it also takes into consideration the rack and data centers configuration.

Replication Factor

- When you create a table (Column Family) in Cassandra, you specify the replication factor. The replication factor is the number of replicas that Cassandra will hold for this table in different nodes. If you specify REPLICATION_FACTOR=3, then your data will be replicated to three different nodes throughout the cluster. That provides fault tolerance and resilience because even if some nodes fail your data would still be safe and available.

Write Consistency Level

- Do you still remember that when the client sends a request to a Cassandra node, this node is called a coordinator and acts as a proxy between the client and the replica nodes?
- Well, when you write to a table in Cassandra (inserting data, for example), you can specify the write consistency level. The write consistency level is the number of replica nodes that have to acknowledge the coordinator that its local insert was successful (success here means that the data was appended to the commit

log and written to the memtable). As soon as the coordinator gets WRITE_CONSISTENCY_LEVEL success acknowledgments from the replica nodes, it returns success back to the client and doesn't wait for the remaining replicas to acknowledge success.

- For example, if an application issue an insert request with WRITE_CONSISTENCY_LEVEL=TWO to a table that is configured with REPLICATION_FACTOR=3, the coordinator will only return success to the application when two of the three replicas acknowledge success. Of course, this doesn't mean that the third replica will not write the data too; it will, but at this point, the coordinator would already have sent success back to the client.
- There are many different types of write consistency levels you can specify in your write request. From the less consistent to full consistency: ANY, ONE, TWO, THREE, QUORUM, LOCAL_QUORUM, EACH_QUORUM, ALL.

Write Flow Example

- For simplicity, suppose a write request is issued to a six-node Cassandra cluster with the following characteristics:
  - WRITE_CONSISTENCY_LEVEL=TWO
  - TABLE_REPLICATION_FACTOR=3
  - REPLICATION_STRATEGY=Simple Strategy
- First, the client sends the write request to the Cassandra cluster using the driver. We haven't discussed the role of the driver in this post (maybe in another post), but it plays a very important role as well. The driver is responsible for a lot of features such as asynchronous IO, parallel execution, request pipelining, connection pooling, auto node discovery, automatic reconnection, token awareness, and so on. For example, by using a driver that implements a token-aware policy, the driver reduces network hops by sending requests directly to the node that owns the data instead of sending it to a "random" coordinator.
- As soon as the coordinator gets the write request, it applies the partitioner hash function to the partition key and uses the configured Replication Strategy in order to determine the TABLE_REPLICATION_FACTOR replica nodes that will actually write the data (in this sentence, replace TABLE_REPLICATION_FACTOR with the number 3). Fig. 2.25 shows the replica nodes (in green) that will handle the write request.

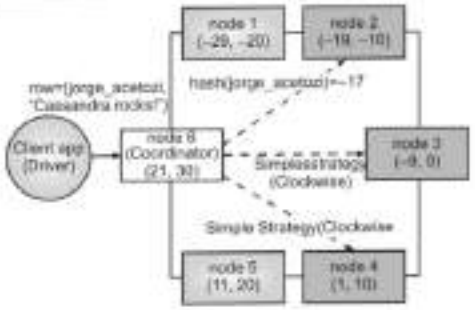


Fig. 2.25 : Replica Nodes

- Now, before the coordinator forwards the write request to all the three replica nodes, it will ask to the Failure Detector component how many of these replica nodes are actually available and compare it to the WRITE_CONSISTENCY_LEVEL provided in the request. If the number of replica nodes available is less than the WRITE_CONSISTENCY_LEVEL provided, the Failure Detector will immediately throw an Exception.
- For our example, suppose the three replica nodes are available (that is, the Failure Detector will allow the request to continue) such as shown in Fig. 2.26. Now, the coordinator will asynchronously forward the write request to all the replica nodes (in these case, the three replica nodes that were figured in the first step). As soon as WRITE_CONSISTENCY_LEVEL replica nodes acknowledge success (node2 and node4), the coordinator returns success back to the driver.

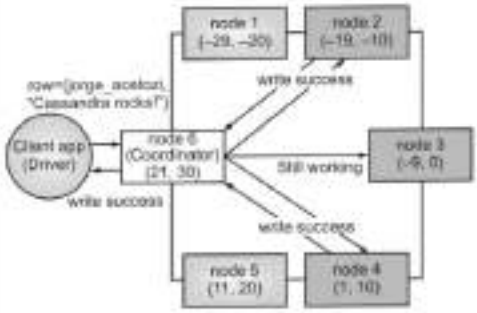


Fig. 2.26 : Write Success

- If the WRITE_CONSISTENCY_LEVEL for this request was THREE (or ALL), the coordinator would have to wait until node3 acknowledges success too, and of course that this write request would be slower.

So, basically...

- Do you need fault tolerance and high availability? Use replication.
- Just bear in mind that using replication means you will pay with consistency (for most of the cases, this is not a problem, Availability is often more important than consistency).
- If consistency is not an issue for your domain, perfect. If it is, just increase the consistency level, but then you will pay with higher latency.
- If you want fault tolerance and high availability, strong consistency and low latency, then you should be the client, not the software engineer (Lol).

Hinted Handoff

- Suppose in the last example that only two of three replica nodes were available. In this case, the Failure Detector would still allow the request to continue as the number of available replica nodes is not less than the WRITE_CONSISTENCY_LEVEL provided. In this case, the coordinator would behave exactly as described before but there would be one additional step. The coordinator would write locally the hint (the write request blob along with some metadata) in the disk (hints directory) and would keep the hint there for three hours (by default) waiting for the replica node to become available again. If the replica node recovers within this period, the coordinator will send the hint to the replica node so that it can update itself and become consistent with the other replicas. If the replica node is offline for more than three hours, then a read repair is needed. This process is referred as Hinted Handoff.

Write Internals

In short, when a write request reaches a node, mainly two things happen:

1. The write request is appended to the commit log in the disk. This ensures data durability (the write request data would permanently survive even in a node failure scenario).
2. The write request is sent to the memtable (a structure stored in the memory). When the memtable is full, the data is flushed to a SSTable on disk using sequential I/O and the data in the commit log is purged.

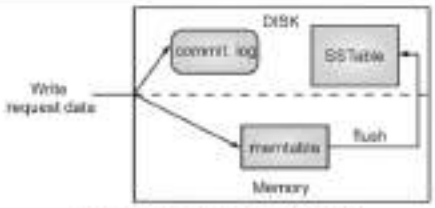


Fig. 2.27 : Cassandra Node Internals

2.14 HBASE

- HBase is a data model that is similar to Google's big table designed to provide quick random access to huge amounts of structured data. This tutorial provides an introduction to HBase, the procedures to set up HBase on Hadoop File Systems, and ways to interact with HBase shell. It also describes how to connect to HBase using java, and how to perform basic operations on HBase using java.
- Since 1970, RDBMS is the solution for data storage and maintenance related problems. After the advent of big data, companies realized the benefit of processing big data and started opting for solutions like Hadoop.
- Hadoop uses distributed file system for storing big data, and MapReduce to process it. Hadoop excels in storing and processing of huge data of various formats such as arbitrary, semi-, or even unstructured.

Limitations of Hadoop

- Hadoop can perform only batch processing, and data will be accessed only in a sequential manner. That means one has to search the entire dataset even for the simplest of jobs.
- A huge dataset when processed results in another huge data set, which should also be processed sequentially. At this point, a new solution is needed to access any point of data in a single unit of time (random access).

Hadoop Random Access Databases

- Applications such as HBase, Cassandra, couchDB, Dynamo, and MongoDB are some of the databases that store huge amounts of data and access the data in a random manner.

What is HBase?

- HBase is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable.
- HBase is a data model that is similar to Google's big table designed to provide quick random access to huge amounts of structured data. It leverages the fault tolerance provided by the Hadoop File System (HDFS).

- It is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System.
- One can store the data in HDFS either directly or through HBase. Data consumer reads/accesses the data in HDFS randomly using HBase. HBase sits on top of the Hadoop File System and provides read and write access.

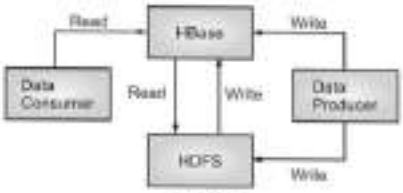


Fig. 2.28

HBase and HDFS

HDFS	HBase
HDFS is a distributed file system suitable for storing large files.	• HBase is a database built on top of the HDFS.
HDFS does not support fast individual record lookups.	HBase provides fast lookups for larger tables.
It provides high latency batch processing; no concept of batch processing.	It provides low latency access to single rows from billions of records (Random access).
It provides only sequential access of data.	HBase internally uses Hash tables and provides random access, and it stores the data in indexed HDFS files for faster lookups.

Storage Mechanism in HBase

- HBase is a **column-oriented database** and the tables in it are sorted by row. The table schema defines only column families, which are the key value pairs. A table have multiple column families and each column family can have any number of columns. Subsequent column values are stored contiguously on the disk. Each cell value of the table has a timestamp. In short, in an HBase:
  - Table is a collection of rows.
  - Row is a collection of column families.
  - Column family is a collection of columns.
  - Column is a collection of key value pairs.

Given below is an example schema of table in HBase.

Rowid	Column Family			Column Family			Column Family			Column Family		
	col1	col2	col3	col1	col2	col3	col1	col2	col3	col1	col2	col3
1												
2												
3												

Column Oriented and Row Oriented

Column-oriented databases are those that store data tables as sections of columns of data, rather than as rows of data. Shortly, they will have column families.

Row-Oriented Database	Column-Oriented Database
It is suitable for Online Transaction Process (OLTP).	It is suitable for Online Analytical Processing (OLAP).
Such databases are designed for small number of rows and columns.	Column-oriented databases are designed for huge tables.

- The following Fig. 2.29 shows column families in a column-oriented database:

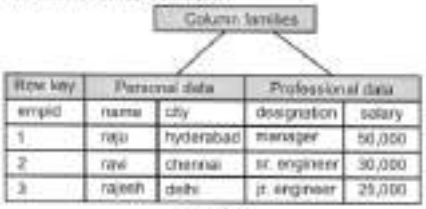


Fig. 2.29

HBase and RDBMS

HBase	RDBMS
HBase is schema-less, it doesn't have the concept of fixed columns schema; defines only column families.	An RDBMS is governed by its schema, which describes the whole structure of tables.
It is built for wide tables. HBase is horizontally scalable.	It is thin and built for small tables. Hard to scale.
No transactions are there in HBase.	RDBMS is transactional.
It has de-normalized data.	It will have normalized data.
It is good for semi-structured as well as structured data.	It is good for structured data.

Features of HBase

- HBase is linearly scalable.
- It has automatic failure support.
- It provides consistent read and writes.

- It integrates with Hadoop, both as a source and a destination.
- It has easy java API for client.
- It provides data replication across clusters.

Where to Use HBase

- Apache HBase is used to have random, real-time read/write access to Big Data.
- It hosts very large tables on top of clusters of commodity hardware.
- Apache HBase is a non-relational database modeled after Google's Bigtable. Bigtable acts up on Google File System, likewise Apache HBase works on top of Hadoop and HDFS.

Applications of HBase

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

HBase History

Year	Event
Nov 2006	Google released the paper on BigTable.
Feb 2007	Initial HBase prototype was created as a Hadoop contribution.
Oct 2007	The first usable HBase along with Hadoop 0.15.0 was released.
Jan 2008	HBase became the sub project of Hadoop.
Oct 2008	HBase 0.18.1 was released.
Jan 2009	HBase 0.19.0 was released.
Sept 2009	HBase 0.20.0 was released.
May 2010	HBase became Apache top-level project.

- In HBase, tables are split into regions and are served by the region servers. Regions are vertically divided by column families into "Stores". Stores are saved as files in HDFS. Shown below is the architecture of HBase.

Note: The term 'store' is used for regions to explain the storage structure.

- HBase has three major components: the client library, a master server, and region servers. Region servers can be added or removed as per requirement.

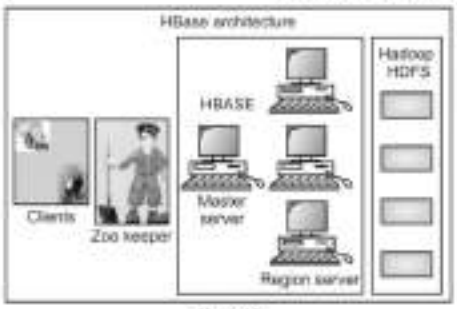


Fig. 2.30

Master Server

The master server -

- Assigns regions to the region servers and takes the help of Apache ZooKeeper for this task.
- Handles load balancing of the regions across region servers. It unloads the busy servers and shifts the regions to less occupied servers.
- Maintains the state of the cluster by negotiating the load balancing.
- Is responsible for schema changes and other metadata operations such as creation of tables and column families.

Regions

- Regions are nothing but tables that are split up and spread across the region servers.

Region Server

The region servers have regions that -

- Communicate with the client and handle data-related operations.
- Handle read and write requests for all the regions under it.
- Decide the size of the region by following the region size thresholds.

When we take a deeper look into the region server, it contain regions and stores as shown below:

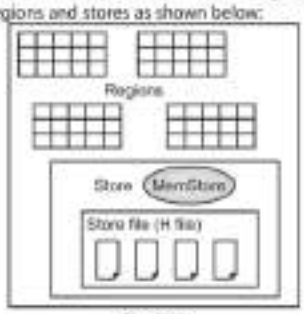


Fig. 2.31

- The store contains memory store and HFiles. Memstore is just like a cache memory. Anything that is entered into the HBase is stored here initially. Later, the data is transferred and saved in Hfiles as blocks and the memstore is flushed.

Zookeeper

- Zookeeper is an open-source project that provides services like maintaining configuration information, naming, providing distributed synchronization, etc.
- Zookeeper has ephemeral nodes representing different region servers. Master servers use these nodes to discover available servers.
- In addition to availability, the nodes are also used to track server failures or network partitions.
- Clients communicate with region servers via zookeeper.
- In pseudo and standalone modes, HBase itself will take care of zookeeper.

2.15 HBASE INTERNALS

- **HBase** is a high-reliability, high-performance, column-oriented, scalable distributed storage system that uses HBase technology to build large-scale structured storage clusters on inexpensive PC Servers. The goal of HBase is to store and process large amounts of data, specifically to handle large amounts of data consisting of thousands of rows and columns using only standard hardware configurations.
- Different from MapReduce's offline batch computing framework, HBase is random access storage and retrieval data platform, which makes up for the shortcomings of HDFS that cannot access data randomly.
- It is suitable for business scenarios where real-time requirements are not very high. HBase stores Byte arrays, which don't mind data types, allowing dynamic, flexible data models.

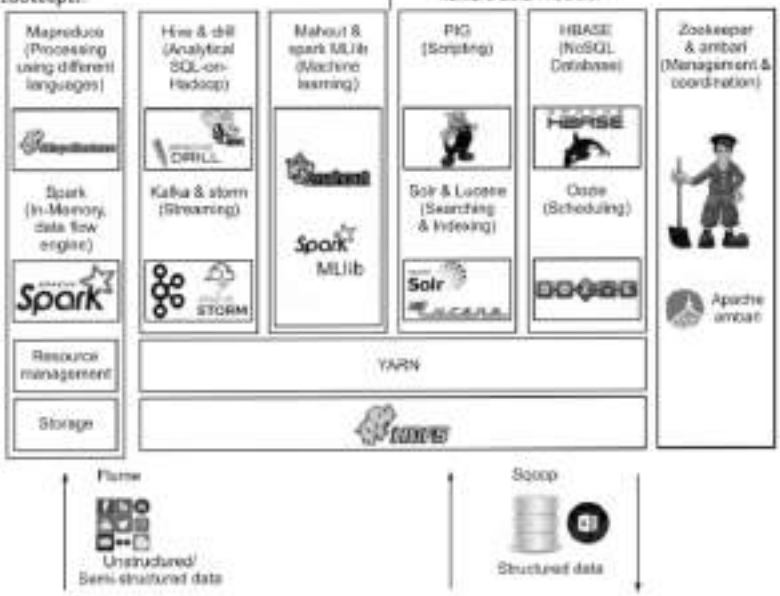


Fig. 2.32

Hadoop Ecosystem (Credit: Edureka.com)

- The Fig. 2.32 above depicts the various layers of the Hadoop 2.0 ecosystem. Hbase located on the structured storage layer.
- HDFS provides high-reliability low-level storage support for HBase.
- MapReduce provides high-performance batch processing capability for HBase. ZooKeeper provides

stable services and failover mechanism for HBase. Pig and Hive provide HBase for high-level language support for data statistics processing. Sqoop provides HDB with available RDBMS data import function, which makes it very convenient to migrate business data from a traditional database to HBase.

2.15.1 HBase Architecture

Design Idea

- HBase is a distributed database that uses ZooKeeper to manage clusters and HDFS as the underlying storage.
- At the architectural level, it consists of HMaster (Leader elected by Zookeeper) and multiple HRegionServers.
- The underlying architecture is shown in the following Fig. 2.33:

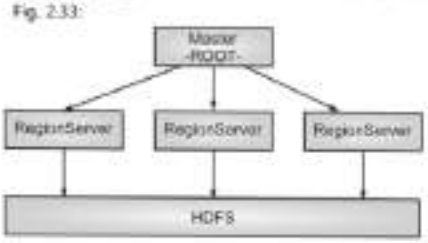


Fig. 2.33

- In the concept of HBase, HRegionServer corresponds to one node in the cluster, one HRegionServer is responsible for managing multiple HRegions, and one HRegion represents a part of the data of a table.
- In HBase, a table may require a lot of HRegions to store data, and the data in each HRegion is not disorganized.
- When HBase manages HRegion, it will define a range of Rowkey for each HRegion. The data falling within a defined scope will be handed over to a specific Region, thus distributing the load to multiple nodes, thus taking advantage of the advantages of distributed and characteristic.
- Also, HBase will automatically adjust the location of the Region. If an HRegionServer is overheated, that is, a large number of requests fall on the HRegion managed by the HRegionServer, HBase will move the HRegion to other nodes that are relatively idle, ensuring that the cluster environment is fully utilized.

Basic Architecture

- **HBase** consists of HMaster and HRegionServer and also follows the master-slave server architecture. HBase divides the logical table into multiple data blocks, HRegion, and stores them in HRegionServer.
- **HMaster** is responsible for managing all HRegionServers. It does not store any data itself, but only stores the mappings (metadata) of data to HRegionServer.
- All nodes in the cluster are coordinated by Zookeeper and handle various issues that may be encountered during HBase operation. The basic architecture of HBase is shown in Fig. 2.34.
- **Client :** Use HBase's RPC mechanism to communicate with HMaster and HRegionServer, submit requests and get results. For management operations, the client performs RPC with HMaster. For data read and write operations, the client performs RPC with HRegionServer.
- **Zookeeper:** By registering the status information of each node in the cluster to ZooKeeper, HMaster can sense the health status of each HRegionServer at any time, and can also avoid the single point problem of HMaster.
- **HMaster:** Manage all HRegionServers, tell them which HRegions need to be maintained, and monitor the health of all HRegionServers. When a new HRegionServer logs in to HMaster, HMaster tells it to wait for data to be allocated. When an HRegion dies, HMaster marks all HRegions it is responsible for as unallocated and then assigns them to other HRegionServers. HMaster does not have a single point problem. HBase can start multiple HMasters. Through the Zookeeper's election mechanism, there is always one HMaster running in the cluster, which improves the availability of the cluster.
- **HRegion:** When the size of the table exceeds the preset value, HBase will automatically divide the table into different areas, each of which contains a subset of all the rows in the table. For the user, each table is a collection of data, distinguished by a primary key (RowKey). Physically, a table is split into multiple blocks, each of which is an HRegion. We use the table name + start/end primary key to distinguish each HRegion. One HRegion will save a piece of continuous data in a table. A complete table data is stored in multiple HRegions.

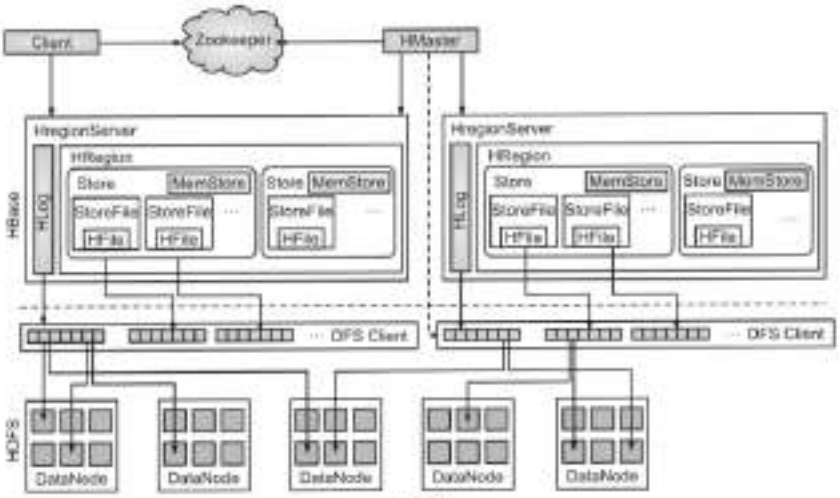


Fig. 2.34

- **HRegionServer:** All data in HBase is generally stored in HDFS from the bottom layer. Users can obtain this data through a series of HRegionServers. Generally, only one HRegionServer is running on one node of the cluster, and the HRegion of each segment is only maintained by one HRegionServer. HRegionServer is mainly responsible for reading and writing data to the HDFS file system in response to user I/O requests. It is the core module in HBase. HRegionServer internally manages a series of HRegion objects, each HRegion corresponding to a continuous data segment in the logical table. HRegion is composed of multiple HStores. Each HStore corresponds to the storage of one column family in the logical table. It can be seen that each column family is a centralized storage unit. Therefore, to improve operational efficiency, it is preferable to place columns with common I/O characteristics in one column family.
- **HStore:** It is the core of HBase storage, which consists of MemStore and StoreFiles. MemStore is a memory buffer. The data written by the user will first be put into MemStore. When MemStore is full, Flush will be a StoreFile (the underlying implementation is HFile). When the number of StoreFile files increases to a certain threshold, the Compact merge operation will be triggered, merge multiple StoreFiles into one StoreFile, and perform version merge and data delete operations during the merge process. Therefore, it can be seen that HBase only adds data, and all update and delete operations are performed in the subsequent Compact process, so that the user's write operation can be returned as soon as it enters the memory, ensuring the high performance of HBase I/O. When StoreFiles Compact, it will gradually form a larger and larger StoreFile. When the size of a single StoreFile exceeds a certain threshold, the Split operation will be triggered. At the same time, the current HRegion will be split into two HRegions, and the parent HRegion will go offline. The two sub-HRegions are assigned to the corresponding HRegionServer by HMaster so that the load pressure of the original HRegion is shunted to the two HRegions.
- **HLog:** Each HRegionServer has an HLog object, which is a pre-written log class that implements the Write Ahead Log. Each time a user writes data to MemStore, it also writes a copy of the data to the HLog file. The HLog file is periodically scrolled and deleted, and the old file is deleted (data that has been persisted to the

StoreFile). When HMaster detects that an HRegionServer is terminated unexpectedly by the Zookeeper, HMaster first processes the legacy HLog file, splits the HLog data of different HRegions, puts them into the corresponding HRegion directory, and then redistributes the invalid HRegions. In the process of loading HRegion, HRegionServer of these HRegions will find that there is a history HLog needs to be processed so the data in Replay HLog will be transferred to MemStore, then Flush to StoreFiles to complete data recovery.

Root and Meta

- All HRegion metadata of HBase is stored in the .META. table. As HRegion increases, the data in the .META table also increases and splits into multiple new HRegions.
- To locate the location of each HRegion in the .META table, the metadata of all HRegions in the .META the table is stored in the -ROOT-table, and finally, the location information of the ROOT-table is recorded by Zookeeper.
- Before all clients access user data, they need to first access Zookeeper to obtain the location of -ROOT-, then access the -ROOT-table to get the location of the .META table, and finally determine the location of the user data according to the information in the META table, as follows: The Fig. 2.35 shows.

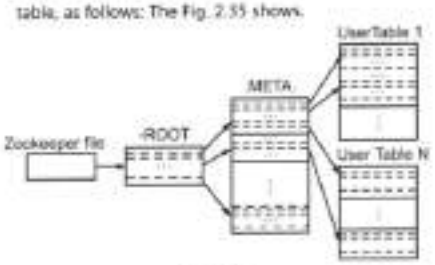


Fig. 2.35

- The -ROOT-table is never split. It has only one HRegion, which guarantees that any HRegion can be located with only three jumps. To speed up access, all regions of the .META table are kept in memory.
- The client caches the queried location information, and the cache does not actively fail. If the client still does not have access to the data based on the cached information, then ask the Region server of the relevant .META table to try to obtain the location of the data. If it still fails, ask where the .META table associated with the -ROOT-table is.
- Finally, if the previous information is all invalid, the data of HRegion is relocated by ZooKeeper. So if the cache on the client is entirely invalid, you need to go back and forth six times to get the correct HRegion.

HBase Data Model

- HBase is a distributed database similar to BigTable. It is sparse long-term storage (on HDFS), multi-dimensional, and sorted mapping tables. The index of this table is the row keyword, column keyword, and timestamp. HBase data is a string, no type.

Row key	Time Stamp	Column "Contents:"	Column "anchor:"		Column "mime:"
"com.cnn.www"	t9		"anchor:cnnsi.com"	"CNN"	
	t8		"anchor:my.look.ca"	"CNN.com"	
	t6	"<html>..."			"text/html"
	t5	"<html>..."			
	t3	"<html>..."			

- Think of a table as a large mapping. You can locate specific data by row key, row key + timestamp or row key + column (column family: column modifier). Since HBase is sparsely storing data, some columns can be blank. The above table gives the logical storage logical view of the com.cnn.www website. There is only one row of data in the table.
- The unique identifier of the row is "com.cnn.www", and there is a time for each logical modification of this row of data. The stamp corresponds to the corresponding.
- There are four columns in the table: contents: HTML, anchor:cnnsi.com, anchor:my.look.ca, mime: type and

each column give the column family to which it belongs.

- The row key (RowKey) is the unique identifier of the data row in the table and serves as the primary key for retrieving records.
- There are only three ways to access rows in a table in HBase: access via a row key, range access for a given row key, and full table scan.
- The row key can be any string (maximum length 64KB) and stored in lexicographical order. For rows that are often read together, the fundamental values need to be carefully designed so that they can be stored collectively.

HBase Read and Write Process

- The Fig. 2.36 below is the HRegionServer data storage relationship diagram. As mentioned above, HBase uses MemStore and StoreFile to store updates to the table. The data is first written to HLog and MemStore when it is updated. The data in the MemStore is sorted.

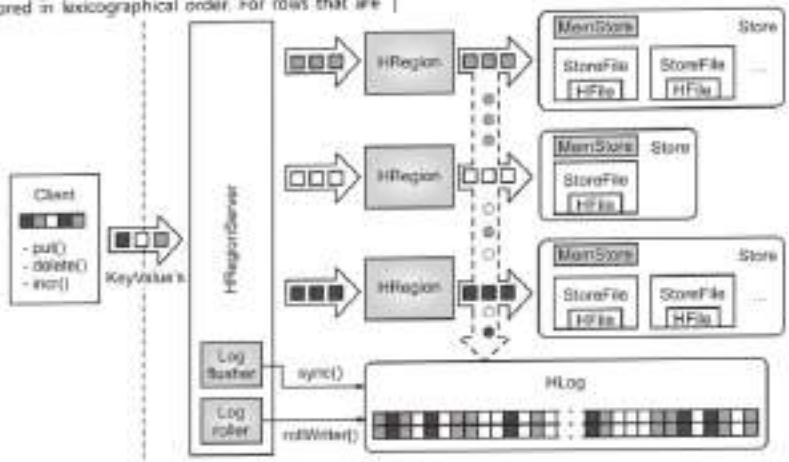


Fig. 2.36

- When the MemStore accumulates to a certain threshold, a new MemStore is created, and the old MemStore is added to the Flush queue, and a separate thread is flushed to the disk to become a StoreFile. At the same time, the system will record a CheckPoint in Zookeeper, indicating that the data changes before this time have been persisted. When an unexpected system occurs, the data in the MemStore may be lost.
- In this case, HLog is used to recover the data after Check Point.
- StoreFile is read-only and cannot be modified once created. Therefore, the update of HBase is an additional operation. When the StoreFile in a store reaches a certain threshold, a merge operation is performed, and the modifications of the same key are merged to form a large StoreFile. When the size of the StoreFile reaches a certain threshold, the StoreFile is split and divided into two StoreFiles.

Write Operation Flow

- **Step 1:** The client sends a write data request to the HRegionServer through the scheduling of the Zookeeper, and writes the data in the HRegion.
- **Step 2:** The data is written to the MemStore of HRegion until the MemStore reaches the preset threshold.
- **Step 3:** The data in MemStore is Flushed into a StoreFile.
- **Step 4:** As the number of StoreFile files increases, when the number of the StoreFile files increases to a

certain threshold, the Compact merge operation is triggered, and multiple StoreFiles are merged into one StoreFile, and version merge and data deletion are performed at the same time.

- **Step 5:** StoreFiles gradually forms a larger and larger StoreFile through the continuous Compact operation.
- **Step 6:** After the size of a single StoreFile exceeds a certain threshold, the Split operation is triggered to split the current HRegion into two new HRegions. The parent HRegion will go offline, and the two sub-HRegions from the new Split will be assigned to the corresponding HRegionServer by HMaster so that the pressure of the original HRegion can be shunted to the two HRegions.

Read Operation Flow

- **Step 1:** The client accesses Zookeeper, finds the -ROOT-table, and obtains the .META. table information.
- **Step 2:** Search from the .META. table to obtain the HRegion information of the target data, to find the corresponding HRegionServer.
- **Step 3:** Obtain the data you need to find through HRegionServer.
- **Step 4:** The memory of the HRegionserver is divided into two parts: MemStore and BlockCache. MemStore is mainly used to write data, and BlockCache is mainly used to read data. Read the request first to the MemStore to check the data, check the BlockCache check, and then check the StoreFile, and put the read result into the BlockCache.

HBase Usage Scenarios

- **Semi-Structured or Unstructured Data:** For data structure fields that are not well defined or cluttered, it is difficult to extract data according to a concept suitable for HBase. If more fields are stored as the business grows, the RDBMS needs to be down to maintain the change table structure, and HBase supports dynamic additions.
- **The Records are Very Sparse:** how many columns of the RDBMS row are fixed, and empty columns waste storage space. Columns with empty HBase are not stored, which saves space and improves read performance.
- **Multi-Version Data:** Values that are located according to the RowKey and column identifiers can have any number of version values (timestamps are different), so it is very convenient to use HBase for data that needs to store the change history.
- **A Large Amount of Data:** When the amount of data is getting larger and larger, the RDBMS database can't hold up, and there is a read-write separation strategy. Through a master, it is responsible for write operations, and multiple slaves are responsible for reading operations, and the server cost is doubled. As the pressure increases, the Master can't hold it. At this time, the library is divided, and the data with little correlation is deployed separately. Some join queries cannot be used, and the middle layer needs to be used. As the amount of data increases further, the record of a table becomes larger and larger, and the query becomes very slow.
- Therefore, it is necessary to divide the table, for example, by modulo the ID into multiple tables to reduce the number of records of a single table. People who have experienced these things know how to toss the process.
- HBase is simple, just add new nodes to the cluster, HBase will automatically split horizontally, and seamless integration with Hadoop ensures data reliability (HDFS) and high performance of massive data analysis (MapReduce).

HBase Map Reduce

- The relationship between Table and Region in HBase is somewhat similar to the relationship between File and Block in HDFS. Since HBase provides APIs for interacting with MapReduce, such as TableInputFormat and TableOutputFormat, HBase data tables can be directly used as input and output of Hadoop MapReduce, which facilitates the development of MapReduce applications, and does not need to pay attention to the processing of HBase system itself Detail.

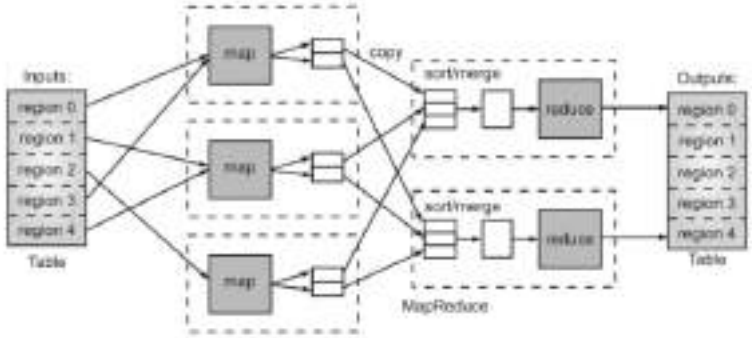


Fig. 2.37

EXERCISE

1. Define the term Big Data platform. Name some popular Big Data platforms.
2. Explain the features of Big Data Platform.
3. What is Apache Spark? Give the overview.
4. Draw and explain the Apache Spark Architecture.
5. Write short note on following
   a. Spark Core
   b. Spark RDD
   c. Spark SQL
   d. Spark MLlib
   e. Spark Streaming
   f. Structured Streaming
6. What is HDFS? What are the features of HDFS?
7. Draw and explain the HDFS architecture.
8. What are the goals of HDFS?
9. List and describe various HDFS operations.
10. What is YARN? Draw and explain the YARN architecture.
11. Mention the features of YARN.
12. What are the different components of YARN.
13. Draw and explain the application workflow in HADOOP YARN.
14. What is MapReduce? Explain the components of MapReduce.
15. Give the different file formats in HADOOP.
16. Draw and explain the MapReduce programming model with SPARK.
17. How MapReduce is used for word count problem ?
18. How MapReduce is used for page ranking problem?
19. Explain the CAP theorem in brief.
20. What is eventual consistency?
21. Describe the consistency trade-offs?
22. What is ZooKeeper? Explain the functionality of ZooKeeper in Hadoop ecosystem.
23. What is Paxos in Hadoop? How it can be used to solve the consensus problem?
24. What is the need of distributed consensus protocol?
25. Describe the working phases of Paxos.
26. What is Cassandra? Differentiate between NoSQL vs. Relational Database
27. Describe the features of Cassandra.

28. Explain data replication in Cassandra.
29. What are the components of Cassandra?
30. Differentiate between RDBMS and Cassandra.
31. What is HBase? Differentiate HDFS and HBase.
32. Differentiate HBase and RDBMS.
33. Explain the features of HBase.
34. Draw and explain the HBase architecture.
35. Write short note on
    a. Read Operation flow in HBase
    b. Write Operation flow in HBase.

UNIT III

BIG DATA STREAMING PLATFORMS

3.1 INTRODUCTION TO BIG DATA STREAMING PLATFORMS

- Streaming data platforms bring together not just low-latency analysis of information, but the important aspect of being able to integrate data between different sources.
- Data-driven strategies are becoming a greater part of an organization's DNA. Executive management is embracing how data can be used to create, sustain, and strengthen competitive advantage. Disruptive companies are building business models based on data that other organizations leave behind. Employees are growing into the role of data analysts as part of their day-to-day responsibilities, and companies are introducing new data sources to continue this trend.
- Specifically, data-driven strategies can be seen in the way that organizations are taking advantage of modern technical architectures in mobility, cloud, and device sensors, and integrating that information into new ways of doing business. The use of location-based mobile apps, optimized supply chains for online retail applications, and the introduction of the internet of things, increased the focus on low-latency data collection, transformation, and analytics.
- With the rise of importance of data-driven organizations and the focus on low-latency decision making, the speed of analytics increased almost as rapidly as the ability to collect information. This is where the world of streaming data platforms comes into play. These modern data management platforms bring together not just the low-latency analysis of information, but the important aspect of being able to integrate information from operation systems, mobile applications, databases, and the Internet of Things in real-time/near real-time.
- The true key to streaming data platforms, and the applications they support, is the integration of technical and business data sources in real time. Without this level of streaming data acquisition, or data integration, the analytics that data-driven strategies and the business models built on those strategies cannot match the promise of the business stakeholders who are looking to create new business value and increase competitive advantage.
- As the business models and strategies of data-driven organizations drive real-time applications, many technologists may ask the following question:
- We can already perform analytics in near real-time, so why do we need a streaming data platform?
- The simple answer is that the low-latency many streaming applications require and that streaming data platforms provide is associated with the acquisition and integration of data sources as opposed to simply the processing of the data into analytics.
- While some organizations can perform analytics in real time, their ability to collect and integrate that information at the same speed detracts from the low-latency concept. For example, imagine you are an organization with a predictive maintenance application for your connected fleet vehicles. As part of your predictive maintenance app, you collect both vehicle usage information and geolocation information from the vehicles.
- While it can be a relatively easy task to perform near real-time analytics, once all the information from your connected fleet is landed in your data warehouse or data lake of choice, the real issue is getting the information to the point where you can do the analysis. Having a delay in data acquisition associated with a predictive maintenance application means that you now know that your fleet vehicle is broken down by the side of the road (in a best-case scenario) or along a single vehicle access "lane" such as a rail line, airport runway, or mining access road (in a worst-case scenario).

- You need to have the information from and about your connected fleet as the data is created on the sensor-equipped vehicle in time to make corrective actions before there is disabled vehicle or that disabled vehicle takes a greater toll on your business operations than simply a single truck, train, plane, or ship.
- Major sources of information to place this streaming data come from various sources, including:
  - ➢ Event, product, and process stage reference data
  - ➢ Operational support applications (i.e. ERP, supply chain, customer care)
  - ➢ Curated business information in data warehouses and data marts
- Much of this contextual information resides in traditional data management platforms, which operate on a different operational tempo and access speed faster than the real-time nature of the applications and strategies of data-driven organizations. This information can be accessed and matched with streaming event information using a technique referred to as CDC (Change Data Capture).
- CDC techniques have been used for years. CDC is the concept of using the transaction and event logs associated with relational database management platforms, or the operational support applications themselves, and using that information to either replicate or access information without causing the underlying data to be disturbed or the core operations of the operational support application to be interrupted. Streaming data platforms use new technical approaches to CDC to improve the speed and availability of the contextual information from those operational platforms in a streaming data platform environment.

3.2 BIG DATA STREAMING PLATFORMS FOR FAST DATA

- Fast data is real-time data that typically comes in from streaming such as through Internet of Things (IoT) technologies and event-driven applications and is analyzed quickly to make rapid business decisions. The point of any analytics is to gain deeper, real-time insights into a situation or problem with an eye towards solving it or producing a better outcome. Because streaming analytics gives visibility into what's happening right now, it can:
  - ➢ Enable decision makers to act sooner
  - ➢ Manage Key Performance Indicators (KPIs) daily rather weekly or monthly
  - ➢ Understand the root cause of problems more quickly
  - ➢ Detect patterns that emerge across diverse data sets such as sales figures and weather variations

Business Requirements for Streaming Data Analytics

- Streaming data analytics can mean several things, depending on context. For Zoomdata, we provide a platform for people to explore and interact with data while it updates in near-real-time. To be relevant and useful across a broad range of users and scenarios, customers have guided us to build a modern BI platform that fulfills the following requirements for streaming data analytics.

Stream Data in Near-Real-Time

- Zoomdata delivers fresh data to the dashboard without requiring users to do anything: no pressing F5 or hitting the "refresh" button. Unlike in-stream analytics for automated, split-second decision-making (for example, to open or shut a valve), streaming data analytics for BI aggregates a variety of data for human decision-making. Human eyes can't differentiate between split-second updates, and different use cases require different rates of refresh; one data source could be updated every few seconds, for example, whereas every minute or so would work great for another data source. Refresh rates in Zoomdata are configurable.

Interactivity, Exploration, and Dynamic Calculations

- Exploring live data should offer the same rich experience as working with historical data. To understand what's going on across a variety of data points to become situationally aware -- business users need to dynamically filter, sort, group, and drill-down on live data. They also want to build new charts, create new derived fields, and even calculate new measures on-the-fly. That's a tall order when the data is constantly rolling in, and Zoomdata does it.

Sometimes You Want to Stop Time

- For the same reasons video surveillance is recorded, sometimes you want to pause, rewind, and playback data streams so you can better see, understand, and communicate events. What's even more powerful, however, is delivering the Data DVR as a standard

- You need to have the information from and about your connected fleet as the data is created on the sensor-equipped vehicle in time to make corrective actions before there is disabled vehicle or that disabled vehicle takes a greater toll on your business operations than simply a single truck, train, plane, or ship.
- Major sources of information to place this streaming data come from various sources, including:
  - ➢ Event, product, and process stage reference data
  - ➢ Operational support applications (i.e. ERP, supply chain, customer care)
  - ➢ Curated business information in data warehouses and data marts
- Much of this contextual information resides in traditional data management platforms, which operate on a different operational tempo and access speed faster than the real-time nature of the applications and strategies of data-driven organizations. This information can be accessed and matched with streaming event information using a technique referred to as CDC (Change Data Capture).
- CDC techniques have been used for years. CDC is the concept of using the transaction and event logs associated with relational database management platforms, or the operational support applications themselves, and using that information to either replicate or access information without causing the underlying data to be disturbed or the core operations of the operational support application to be interrupted. Streaming data platforms use new technical approaches to CDC to improve the speed and availability of the contextual information from those operational platforms in a streaming data platform environment.

3.2 BIG DATA STREAMING PLATFORMS FOR FAST DATA

- Fast data is real-time data that typically comes in from streaming such as through Internet of Things (IoT) technologies and event-driven applications and is analyzed quickly to make rapid business decisions. The point of any analytics is to gain deeper, real-time insights into a situation or problem with an eye towards solving it or producing a better outcome. Because streaming analytics gives visibility into what's happening right now, it can:
  - ➢ Enable decision makers to act sooner
  - ➢ Manage Key Performance Indicators (KPIs) daily rather weekly or monthly
  - ➢ Understand the root cause of problems more quickly
  - ➢ Detect patterns that emerge across diverse data sets such as sales figures and weather variations

Business Requirements for Streaming Data Analytics

- Streaming data analytics can mean several things, depending on context. For Zoomdata, we provide a platform for people to explore and interact with data while it updates in near-real-time. To be relevant and useful across a broad range of users and scenarios, customers have guided us to build a modern BI platform that fulfills the following requirements for streaming data analytics.

Stream Data in Near-Real-Time

- Zoomdata delivers fresh data to the dashboard without requiring users to do anything: no pressing F5 or hitting the "refresh" button. Unlike in-stream analytics for automated, split-second decision-making (for example, to open or shut a valve), streaming data analytics for BI aggregates a variety of data for human decision-making. Human eyes can't differentiate between split-second updates, and different use cases require different rates of refresh; one data source could be updated every few seconds, for example, whereas every minute or so would work great for another data source. Refresh rates in Zoomdata are configurable.

Interactivity, Exploration, and Dynamic Calculations

- Exploring live data should offer the same rich experience as working with historical data. To understand what's going on across a variety of data points to become situationally aware -- business users need to dynamically filter, sort, group, and drill-down on live data. They also want to build new charts, create new derived fields, and even calculate new measures on-the-fly. That's a tall order when the data is constantly rolling in, and Zoomdata does it.

Sometimes You Want to Stop Time

- For the same reasons video surveillance is recorded, sometimes you want to pause, rewind, and playback data streams so you can better see, understand, and communicate events. What's even more powerful, however, is delivering the Data DVR as a standard

feature for interactive, visual dashboards so you can better understand what preceded current events and why.

- There is one attribute common to all data streams: time. Every event from a heartbeat, to a purchase, or an airplane departure, can be associated with a point in time. Zoomdata holds several patents related to streaming data visualization, and how we work with time is a key part of our enabling technology.
- Real-time analytics can keep you up-to-date on what's happening right now, such as how many people are currently reading your new blog post and whether someone just liked your latest Facebook status. For most use cases, real time is a nice-to-have feature that won't provide any crucial insights. However, sometimes real time is a must.
- Let's say that you run a big ad agency. Real-time analytics can keep you posted on whether your latest online ad campaign that your client paid tons of money for is actually working, and if not, you can make immediate changes before the budget gets spent any further. Another use case is providing real-time analytics for your own app it looks good, and your users may require it.
- There are quite a few real-time platforms out there. A lot of them are newcomers, and the differences between them aren't clear at all. The least we can do, is present all the options for you to choose from, so here are five real-time streaming platforms for Big Data.

1. **Apache Flink**

- Flink is an open-source streaming platform capable of running near real-time, fault tolerate processing pipelines, scalable to millions of events per second. Flink enables the execution of batch and stream processing.

2. **Apache Spark**

- Spark is an open-source data-processing framework that is really hot at the moment. Because Spark runs in-memory on clusters, and it isn't tied to Hadoop's MapReduce two-stage paradigm, it has lightning-fast performance. Spark can run as a standalone or on top of Hadoop YARN, where it can read data directly from HDFS. In addition to its in-memory processing, graph processing, and machine learning, Spark can also handle streaming. Companies like Yahoo, Intel, Baidu, Trend Micro, and Groupon are already using it.

3. **Apache Storm**

- Storm is a distributed real-time computation system that claims to do for streaming what Hadoop did for batch processing. It can be used for real-time analytics, machine learning, continuous computation, and more. The cool thing is that it was designed to be used with any programming language. It runs on top of Hadoop YARN and can be used with Flume to store data on HDFS. Storm is already used by the likes of WebMD, Yelp, and Spotify.

4. **Apache Samza**

- Samza is a distributed stream-processing framework that is based on Apache Kafka and YARN. It provides a simple callback-based API that's similar to MapReduce, and it includes snapshot management and fault tolerance in a durable and scalable way.

5. **Amazon Kinesis**

- Kinesis is Amazon's service for real-time processing of streaming data on the cloud. It's deeply integrated with other Amazon services via connectors, such as S3, Redshift, and DynamoDB, for a complete Big Data architecture. Kinesis also includes Kinesis Client Library (KCL) that allows you to build applications and use stream data for dashboards, alerts, or even dynamic pricing.

3.3 STREAMING SYSTEMS

- Streaming data is becoming a core component of enterprise data architecture due to the explosive growth of data from non-traditional sources such as IoT sensors, security logs and web applications.
- Streaming technologies are not new, but they have considerably matured in recent years. The industry is moving from painstaking integration of open-source Spark/Hadoop frameworks, towards full stack solutions that provide an end-to-end streaming data architecture built on the scalability of cloud data lakes. Streaming Data is data that is generated continuously by thousands of data sources, which typically send in the data records simultaneously, and in small sizes (order of Kilobytes). Streaming data includes a wide variety of data such as log files generated by customers using your mobile or web applications, ecommerce purchases, in-game player activity, information from social networks, financial trading floors, or geospatial services, and telemetry from connected devices or instrumentation in data centers.
- This data needs to be processed sequentially and incrementally on a record-by-record basis or over

sliding time windows, and used for a wide variety of analytics including correlations, aggregations, filtering, and sampling. Information derived from such analysis gives companies visibility into many aspects of their business and customer activity such as –service usage (for metering/billing), server activity, website clicks, and geo-location of devices, people, and physical goods – and enables them to respond promptly to emerging situations. For example, businesses can track changes in public sentiment on their brands and products by continuously analyzing social media streams, and respond in a timely fashion as the necessity arises.

- Streaming data refers to data that is **continuously generated**, usually in **high volumes** and at **high velocity**. A streaming data source would typically consist of a stream of logs that record events as they happen – such as a user clicking on a link in a web page, or a sensor reporting the current temperature.

Common Examples of Streaming Data Include:

- ➢ IoT sensors
- ➢ Server and security logs
- ➢ Real-time advertising
- ➢ Click-stream data from apps and websites

- In all of these cases we have end devices that are continuously generating thousands or millions of records, forming a data stream – unstructured or semi-structured form, most commonly JSON or XML key-value pairs. Here's an example of how a single streaming event would look – in this case the data we are looking at is a website session

- Sensors in transportation vehicles, industrial equipment, and farm machinery send data to a streaming application. The application monitors performance, detects any potential defects in advance, and places a spare part order automatically preventing equipment down time.
- A financial institution tracks changes in the stock market in real time, computes value-at-risk, and automatically rebalances portfolios based on stock price movements.
- A real-estate website tracks a subset of data from consumers' mobile devices and makes real-time property recommendations of properties to visit based on their geo-location.
- A solar power company has to maintain power throughput for its customers, or pay penalties. It implemented a streaming data application that monitors of all of panels in the field, and schedules service in real time, thereby minimizing the periods of low throughput from each panel and the associated penalty payouts.
- A media publisher streams billions of click stream records from its online properties, aggregates and enriches the data with demographic information about users, and optimizes content placement on its site, delivering relevancy and better experience to its audience.
- An online gaming company collects streaming data about player-game interactions, and feeds the data into its gaming platform. It then analyzes the data in real-time, offers incentives and dynamic experiences to engage its players.

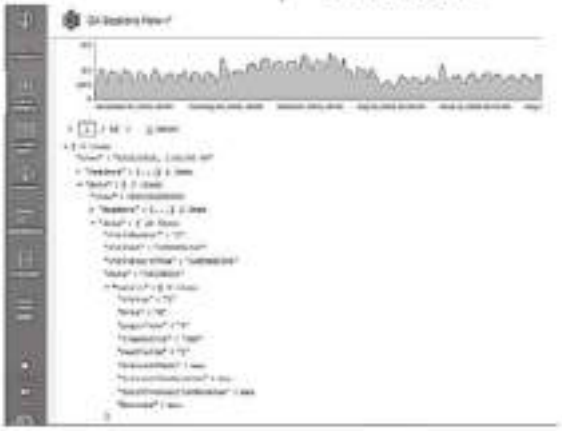


Fig. 3.1

- A single streaming source will generate massive amounts of these events every minute. In its raw form, this data is very difficult to work with as the lack of schema and structure makes it difficult to query with SQL-based analytic tools. Instead, data needs to be processed, parsed and structured before any serious analysis can be done.

Streaming Data Architecture

- A streaming data architecture is a framework of software components built to ingest and process large volumes of streaming data from multiple sources. While traditional data solutions focused on writing and reading data in batches, a streaming data architecture consumes data immediately as it is generated, persists it to storage, and may include various additional components per use case – such as tools for real-time processing, data manipulation and analytics.
- Streaming architectures need to be able to account for the unique characteristics of data streams, which tend to generate massive amounts of data (terabytes to peta bytes) that is at best semi-structured and requires significant pre-processing and ETL to become useful.
- Stream processing is a complex challenge rarely solved with a single database or ETL tool – hence the need to 'architect' a solution consisting of multiple building blocks. Part of the thinking behind Up solver is that many of these building blocks can be combined and replaced with declarative functions within the platform, and we will demonstrate how this approach manifests within each part of the streaming data supply chain.
- Stream processing used to be a 'niche' technology used only by a small subset of companies. However, with the rapid growth of SaaS, IoT and machine learning, organizations across industries are now dipping their feet into streaming analytics. It's difficult to find a modern company that doesn't have an app or a website; as traffic to these digital assets grows, and with increasing appetite for complex and real-time analytics, the need to adopt modern data infrastructure is quickly becoming mainstream.
- While traditional batch architectures can be sufficient at smaller scales, stream processing provides several benefits that other data platforms cannot:
  - ➢ **Able to Deal with Never-Ending Streams of Events :** Some data is naturally structured this way. Traditional batch processing tools require stopping the stream of events, capturing batches of data and combining the batches to draw overall conclusions. In stream processing, while it is challenging to combine and capture data from multiple streams, it lets you derive immediate insights from large volumes of streaming data.
  - ➢ **Real-Time or Near-Real-Time Processing :** Most organizations adopt stream processing to enable real time data analytics. While real time analytics is also possible with high performance database systems, often the data lends itself to a stream processing model.
  - ➢ **Detecting Patterns in Time-Series Data :** Detecting patterns over time, for example looking for trends in website traffic data, requires data to be continuously processed and analyzed. Batch processing makes this more difficult because it breaks data into batches, meaning some events are broken across two or more batches.
  - ➢ **Easy Data Scalability :** Growing data volumes can break a batch processing system, requiring you to provision more resources or modify the architecture. Modern stream processing infrastructure is hyper-scalable, able to deal with Gigabytes of data per second with a single stream processor. This allows you to easily deal with growing data volumes without infrastructure changes.

The Components of a Streaming Architecture

- Most streaming stacks are still built on an assembly line of open-source and proprietary solutions to specific problems such as stream processing, storage, data integration and real-time analytics. At Up solver we've developed a modern platform that combines most building blocks and offers a seamless way to transform streams into analytics-ready datasets.
- Whether you go with a modern data lake platform or a traditional patchwork of tools, your streaming architecture must include these four key building blocks:

1. The Message Broker / Stream Processor

- This is the element that takes data from a source, called a producer, translates it into a standard message format, and streams it on an ongoing basis. Other components can then listen in and consume the messages passed on by the broker.
- The first generation of message brokers, such as RabbitMQ and Apache ActiveMQ, relied on the Message Oriented Middleware (MOM) paradigm. Later, hyper-performant messaging platforms (often called stream processors) emerged which are more suitable for a streaming paradigm. Two popular stream processing tools are Apache Kafka and Amazon Kinesis Data Streams.

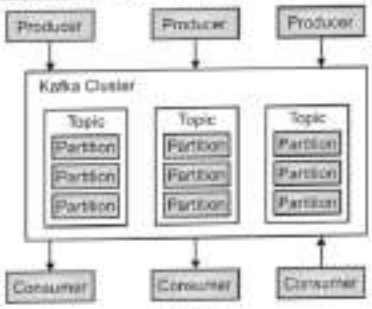


Fig. 3.2

- Unlike the old MoM brokers, streaming brokers support very high performance with persistence, have massive capacity of a Gigabyte per second or more of message traffic, and are tightly focused on streaming with little support for data transformations or task scheduling (although Confluent's KSQL offers the ability to perform basic ETL in real-time while storing data in Kafka).
- You can learn more about message brokers in our article on analyzing Apache Kafka data, as well as these comparisons between Kafka and RabbitMQ and between Apache Kafka and Amazon Kinesis.

2. Batch and Real-Time ETL Tools

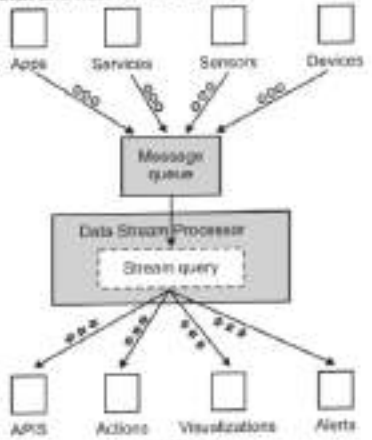


Fig. 3.3

- Data streams from one or more message brokers need to be aggregated, transformed and structured before data can be analyzed with SQL-based analytics tools. This would be done by an ETL tool or platform receives queries from users, fetches events from message queues and applies the query, to generate a result – often performing additional joins, transformations on aggregations on the data. The result may be an API call, an action, a visualization, an alert, or in some cases a new data stream.

3. Data Analytics / Serverless Query Engine

- After streaming data is prepared for consumption by the stream processor, it must be analyzed to provide value. There are many different approaches to streaming data analytics. Here are some of the tools most commonly used for streaming data analytics.

4. Streaming Data Storage

- With the advent of low cost storage technologies, most organizations today are storing their streaming event data. Here are several options for storing streaming data, and their pros and cons.

Streaming Data Storage Option	Pros	Cons
In a database or data warehouse : for example, PostgreSQL or Amazon Redshift	Easy SQL-based data analysis.	Hard to scale and manage. If cloud-based, storage is expensive.
In the message broker : For example, using Kafka persistent storage	Agile, no need to structure data into tables. Easy to set up, no additional components.	Data retention is an issue since Kafka storage is up to 10X more expensive compared to data lake storage. Kafka performance is best for reading recent (cached) data (cached)
In a data lake : for example, Amazon S3	Agile, no need to structure data into tables. Low cost storage.	High latency, makes real time analysis difficult. Difficult to perform SQL-based analysis.

- A data lake is the most flexible and inexpensive option for storing event data, but it is often very technically involved to build and maintain one. We've written before about the challenges of building a data lake and maintaining lake storage best practices, including the need to ensure exactly-once processing, partitioning the data, and enabling backfill with historical data. It's easy to just dump all your data into object storage; creating an operational data lake can often be much more difficult.
- Up solver's data lake ETL platform reduces time-to-value for data lake projects by automating stream ingestion, schema-on-read, and metadata extraction. This allows data consumers to easily prepare data for analytics tools and real time analysis. To learn more, you can check out our Product page.

Modern Streaming Architecture

- In modern streaming data deployments, many organizations are adopting a full stack approach rather than relying on patching together open-source technologies. The modern data platform is built on business-centric value chains rather than IT-centric coding processes, wherein the complexity of traditional architecture is abstracted into a single self-service platform that turns event streams into analytics-ready data.
- The idea behind Up solver is to act as the centralized data platform that automates the labor-intensive parts of working with streaming data: message ingestion, batch and streaming ETL, storage management and preparing data for analytics.

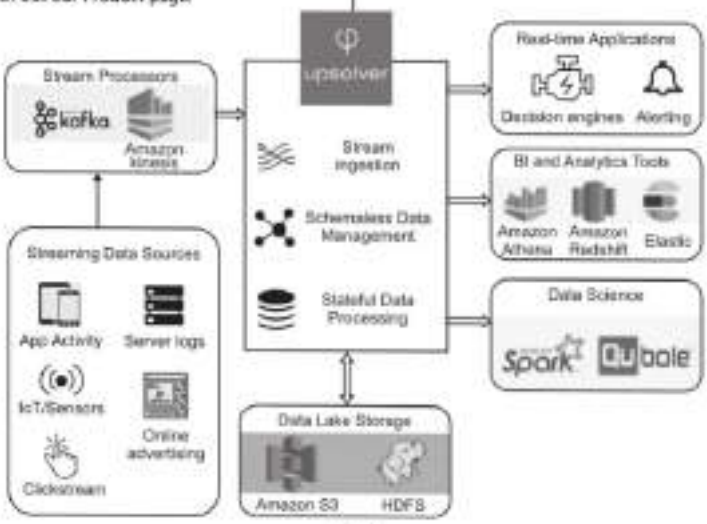


Fig. 3.4

Benefits of a Modern Streaming Architecture:

- Streaming data processing is beneficial in most scenarios where new, dynamic data is generated on a continual basis. It applies to most of the industry segments and big data use cases. Companies generally begin with simple applications such as collecting system logs and rudimentary processing like rolling min-max computations. Then, these applications evolve to more sophisticated near-real-time processing. Initially, applications may process data streams to produce simple reports, and perform simple actions in response, such as emitting alarms when key measures exceed certain thresholds. Eventually, those applications perform more sophisticated forms of data analysis, like applying machine learning algorithms, and extract deeper insights from the data. Over time, complex, stream and event processing algorithms, like decaying time windows to find the most recent

popular movies, are applied, further enriching the insights.

- ➢ Can eliminate the need for large data engineering projects
- ➢ Performance, high availability and fault tolerance built in
- ➢ Newer platforms are cloud-based and can be deployed very quickly with no upfront investment
- ➢ Flexibility and support for multiple use cases

The Future of Streaming Data in 2019 and Beyond

- Streaming data architecture is in constant flux. Three trends we believe will be significant in 2019 and beyond:
  1. **Fast Adoption of Platforms that Decouple Storage and Compute** : Streaming data growth is making traditional data warehouse platforms too expensive and cumbersome to manage. Data lakes are increasingly used, both as a cheap persistence option for storing large volumes of event data, and as a flexible integration point, allowing tools outside the streaming ecosystem to access streaming data.
  2. **From Table Modeling to Schemaless Development** : Data consumers don't always know the questions they will ask in advance. They want to run an interactive, iterative process with as little initial setup as possible. Lengthy table modeling, schema detection and metadata extraction are a burden.
  3. **Automation of Data Plumbing** : Organizations are becoming reluctant to spend precious data engineering time on data plumbing, instead of activities that add value, such as data cleansing or enrichment. Increasingly, data teams prefer full stack platforms that reduce time-to-value, over tailored home-grown solutions.

Streaming and Analytics Use Cases

- Stream processing systems like Apache Kafka and Confluent bring real-time data and analytics to life. While there are use cases for data streaming in every industry, this ability to integrate, analyze, troubleshoot, and/or predict data in real-time, at massive scale, opens up new use cases. Not only can organizations use past data or batch data in storage, but gain valuable insights on data in motion.

Typical Uses Cases Include:

- Location data
- Fraud detection
- Real-time stock trades
- Marketing, sales, and business analytics
- Customer/user activity
- Monitoring and reporting on internal IT systems
- Log Monitoring: Troubleshooting systems, servers, devices, and more
- SIEM (Security Information and Event Management): analyzing logs and real-time event data for monitoring, metrics, and threat detection
- Retail/warehouse inventory: inventory management across all channels and locations, and providing a seamless user experience across all devices
- Ride share matching: Combining location, user, and pricing data for predictive analytics - matching riders with the best drivers in term of proximity, destination, pricing, and wait times
- Machine learning and A.I.: By combining past and present data for one central nervous system, predictive analytics bring new possibilities.

As long as there is any type of data to be processed, stored, or analyzed, a stream processing system like Apache Kafka can help leverage your data to produce numerous use cases. It's open source software that anyone can use for free.

If you don't have the manpower or expertise to build your own stream processing applications, Confluent makes it easy to get started with virtually any type of data without the hassle of building, configuring, or managing your own applications.

3.4 BIG DATA PIPELINES FOR REAL TIME COMPUTING

- Defined by 3Vs that are velocity, volume, and variety of the data, big data sits in the separate row from the regular data. Though big data was the buzzword since last few years for data analysis, the new fuss about big data analytics is to build up real-time big data pipeline. In a single sentence, to build up an efficient big data analytic system for enabling organizations to make decisions on the fly.
- In a real-time big data pipeline, you need to consider factors like real-time fraud analysis, log analysis, predicting errors to measure the correct business decisions.

- Hence, to process such high-velocity massive data on a real-time basis, highly reliable data processing system is the demand of the hour. There are many open source tools and technologies available in the market to perform real-time big data pipeline operations. In this blog, we will discuss the most preferred ones – Apache Hadoop, Apache Spark, and Apache Kafka.
- It is estimated that by 2020 approximately 1.7 megabytes of data will be created every second. This results in an increasing demand for real-time and streaming data analysis. For historical data analysis descriptive, prescriptive, and predictive analysis techniques are used. On the other hand, for real-time data analysis, streaming data analysis is the choice. The main benefit of real-time analysis is one can analyze and visualize the report on a real-time basis.
- Through real-time big data pipeline, we can perform real-time data analysis which enables below capabilities:
  - Helps to make operational decisions.
  - The decisions built out of the results will be applied to business processes, different production activities and transactions in real time.
  - It can be applied to prescriptive or pre-existing models.
  - Helps to generate historical and current data concurrently.
  - Generates alerts based on predefined parameters.
  - Monitors constantly for changing transactional data sets in real time.
- A real-time big data pipeline should have some essential features to respond to business demands, and besides that, it should not cross the cost and usage limit of the organization.

Features That a Big Data Pipeline System Must Have:

- **High Volume Data Storage:** The system must have a robust big data framework like Apache Hadoop.
- **Messaging System:** It should have publish-subscribe messaging support like Apache Kafka.
- **Predictive Analysis Support:** The system should support various machine learning algorithms. Hence it must have required library support like Apache Spark MLlib.
- **The Flexible Backend to Store Result Data:** The processed output must be stored in some database. Hence, a flexible database preferably NoSQL data should be in place.
- **Reporting and Visualization Support:** The system must have some reporting and visualization tool like Tableau.
- **Alert Support:** The system must be able to generate text or email alert, and related tool support must be in place.

There are some key points that we need to measure while selecting a tool or technology for building a big data pipeline which is as follows:

- Components
- Parameters

Components of a Big Data Pipeline are:

- The messaging system.
- Message distribution support to various nodes for further data processing.
- Data analysis system to derive decisions from data.
- Data storage system to store results and related information.
- Data representation and reporting tools and alerts system.

Important Parameters that a Big Data Pipeline System Must Have :

- Compatible with big data
- Low latency
- Scalability
- A diversity that means it can handle various use cases
- Flexibility
- Economic

The choice of technologies like Apache Hadoop, Apache Spark, and Apache Kafka address the above aspects. Hence, these tools are the preferred choice for building a real-time big data pipeline.

In a Big Data Pipeline System, the Two Core Processes are :

1. The messaging system
2. The data ingestion process

- The messaging system is the entry point in a big data pipeline and Apache Kafka is a publish-subscribe messaging system work as an input system. For

messaging. Apache Kafka provide two mechanisms utilizing its APIs –

1. Producer
2. Subscriber

- Using the Priority queue, it writes data to the producer. Then the data is subscribed by the listener. It could be a Spark listener or any other listener. Apache Kafka can handle high-volume and high-frequency data.
- Once the data is available in a messaging system, it needs to be ingested and processed in a real-time manner. Apache Spark makes it possible by using its streaming APIs. Also, Hadoop MapReduce processes the data in some of the architecture.
- Apache Hadoop provides an ecosystem for the Apache Spark and Apache Kafka to run on top of it. Additionally, it provides persistent data storage through its HDFS. Also for security purpose, Kerberos can be configured on the Hadoop cluster. Since components such as Apache Spark and Apache Kafka run on a Hadoop cluster, thus they are also covered by this security features and enable a robust big data pipeline system.
- There are two types of architecture followed for the making of real-time big data pipeline:
  1. Lambda architecture
  2. Kappa architecture

1. Lambda Architecture

There are mainly three purposes of Lambda architecture –

(i) Ingest

(ii) Process

(iii) Query real-time and batch data

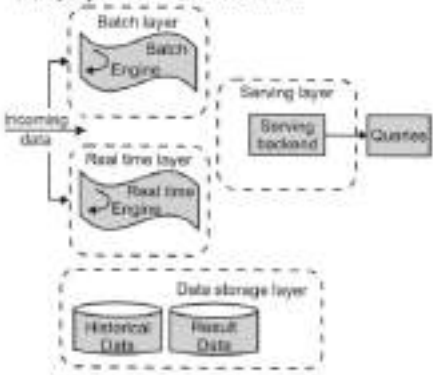


Fig. 3.5

- Single data architecture is used for the above three purposes. This architecture consists of three layers of lambda architecture
  (i) Speed layer
  (ii) Serving layer
  (iii) Batch layer
- These layers mainly perform real-time data processing and identify if any error occurs in the system.

How does Lambda Architecture Work?

- From the input source data enters into the system and routed to the batch layer and speed layer. The input source could be a pub-sub messaging system like Apache Kafka.
- Apache Hadoop sits at the batch layer and along with playing the role of persistent data storage performs the two most important functions:
  (i) Manages the master dataset
  (ii) Pre-compute the batch views
- Serving layer indexes the batch views which enables low latency querying. NoSQL database is used as a serving layer.
- Speed layer deals with the real-time data only. Also in case of any data error or missing of data during data streaming it manages high latency data updates. Hence, batch jobs running in Hadoop layer will compensate that by running MapReduce job at regular intervals. As a result speed layer provides real-time results to a serving layer. Usually, Apache Spark works as the speed layer.
- Finally, a merged result is generated which is the combination of real-time views and batch views.
- Apache Spark is used as the standard platform for batch and speed layer. This facilitates the code sharing between the two layers.

2. Kappa Architecture

- Kappa architecture is comprised of two layers instead of three layers as in the Lambda architecture. These layers are –
  (i) Streaming data.
  (ii) Real-time layer/Stream processing
  (iii) Serving layer

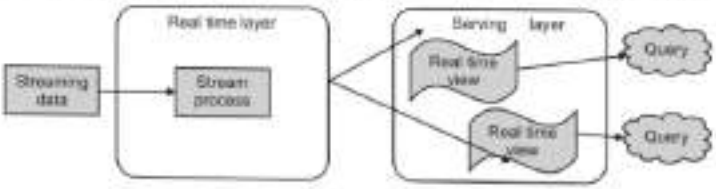


Fig. 3.6

The Flow of Kappa Architecture

- In this case, the incoming data is ingested through the real-time layer via a messaging system like Apache Kafka.
- In the real-time layer or streaming process data is processed. Usually, Apache Spark is used in this layer as it supports both batch and stream data processing.
- The output result from the real-time layer is sent to the serving layer which is a backend system like a NoSQL database.
- Apache Hadoop provides the eco-system for Apache Spark and Apache Kafka.
- The main benefit of Kappa architecture is that it can handle both real-time and continuous data processing through a single stream process engine.
- If Big Data pipeline is appropriately deployed it can add several benefits to an organization. As it can enable real-time data processing and detect real-time fraud, it helps an organization from revenue loss. Big data pipeline can be applied in any business domains, and it has a huge impact towards business optimization.

3.5 SPARK STREAMING

- Apache Spark Streaming is a scalable fault-tolerant streaming processing system that natively supports both batch and streaming workloads. Spark Streaming is an extension of the core Spark API that allows data engineers and data scientists to process real-time data from various sources including (but not limited to) Kafka, Flume, and Amazon Kinesis. This processed data can be pushed out to file systems, databases, and live dashboards. Its key abstraction is a Discretized Stream or, in short, a DStream, which represents a stream of data divided into small batches. DStreams are built on RDDs, Spark's core data abstraction. This allows Spark Streaming to seamlessly integrate with any other Spark components like MLlib and Spark SQL. Spark Streaming is different from other systems that either have a processing engine designed only for streaming, or have similar batch and streaming APIs but compile internally to different engines. Spark's single execution engine and unified programming model for batch and streaming lead to some unique benefits over other traditional streaming systems. When data continuously arrives in a sequence of unbound, then it is called a data stream. Input data is flowing steadily, and it is divided by streaming. Further processing of data is done after it is divided into discrete units. The analysing of data and processing data at low latency is called stream processing.
- In 2013, Apache Spark was added with Spark Streaming. There are many sources from which the Data ingestion can happen such as: TCP Sockets, Amazon Kinesis, Apache Flume and Kafka. With the help of sophisticated algorithms, processing of data is done. A high-level function such as window, join, reduce and map are used to express the processing. Live Dashboards, Databases and file systems are used to push the processed data to file systems.

Four Major Aspects of Spark Streaming

1. Fast recovery from failures and stragglers
2. Better load balancing and resource usage
3. Combining of streaming data with static datasets and interactive queries
4. Native integration with advanced processing libraries (SQL, machine learning, graph processing)

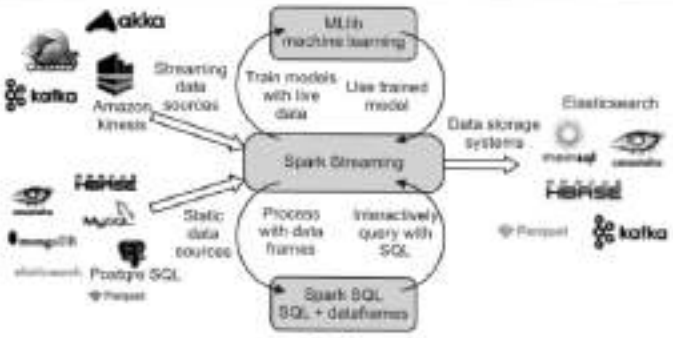


Fig. 3.7

- This unification of disparate data processing capabilities is the key reason behind Spark Streaming's rapid adoption. It makes it very easy for developers to use a single framework to satisfy all their processing needs.

Working of Stream

- Following are the internal working. Spark streaming divides the live input data streams into batches. Spark Engine is used to process these batches to generate final stream batches as a result.
- Data in the stream is divided into small batches and is represented by Apache Spark Discretized Stream (Spark DStream). Spark RDDs is used to build DStreams, and this is the core data abstraction of Spark. Any components of Apache Spark such as Spark SQL and Spark MLib can be easily integrated with the Spark Streaming seamlessly.
- Spark Streaming helps in scaling the live data streams. It is one of the extensions of the core Spark API. It also enables processing of fault-tolerant stream and high-throughput. The use of Spark Streaming does Real-time processing and streaming of live data. Major Top Companies in the world are using the service of Spark Streaming such as Pinterest, Netflix and Uber.
- Spark Streaming also provides an analysis of data in real-time. Live and Fast processing of data are performed on the single platform of Spark Streaming.
- Spark Streaming can be used to stream real-time data from different sources, such as Facebook, Stock Market, and Geographical Systems, and conduct powerful analytics to encourage businesses.

There are five significant aspects of Spark Streaming which makes it so unique, and they are:

1. **Integration**

   Advanced Libraries like graph processing, machine learning, SQL can be easily integrated with it.

2. **Combination**

   The data which is getting streamed can be done in conjunction with interactive queries and also static datasets.

3. **Load Balancing**

   Spark Streaming has a perfect balancing of load, which makes it very special.

4. **Resource Usage**

   Spark Streaming use the available resource in a very optimum way.

5. **Recovery from Stragglers and Failures**

   Spark Streaming can quickly recover from any kinds of failures or straggler.

Need for Streaming in Apache Spark

- Continuous operator model is used while designing the system for processing streams traditionally to process the data. The working of the system is as follows:
  - ➢ Data sources are used to stream the data. The different kinds of Data sources are IoT device, system telemetry data, live logs and many more. These streaming data are ingested into data ingestion systems such as Amazon Kinesis, Apache Kafka and many more.

- ➢ On a cluster, parallel processing is done on the data.
- ➢ Downstream systems such as Kafka, Cassandra, HBase are used to pass the results.

- A set of worker nodes runs some continuous operators. The processing of records of streamed data is done one at a time. The documents are then forwarded to the next operators in the pipeline.
- Source Operators are used to receiving Data from ingestion systems. Sink Operators are used to giving output to the downstream system.
- Some operators are continuous. These are a natural and straightforward model. When it comes to Analytics of complex data at real-time, which is done at a large scale, traditional architecture faces some challenges in the modern world, and they are:

Fast Failure Recovery

- In today's system failures are quickly accompanied and accommodated by recovering lost information by computing the missing info in parallel nodes. Thus, it makes the recovery even faster compared to traditional systems.

Load Balancer

- Load balancer helps to allocate resource and data among the node in a more efficient manner so that no resource is waiting or doing nothing but the data is evenly distributed throughout the nodes.

Unification of Interactive, Batch and Streaming Workloads

- One can also interact with streaming data by making queries to the streaming data. It can also be combined with static datasets. One cannot do ad-hoc queries using new operators because it is not designed for continuous operators. Interactive, streaming and batch queries can be combined by using a single-engine.

SQL Queries and Analytics with ML

- Developing systems with common database command made developer life easy to work in collaboration with other systems. The community widely accepts SQL queries. Where the system provides module and libraries for machine learning that can be used for advance analytical purpose.

Spark Streaming Overview

- Spark Streaming uses a set of RDDs which is used to process the real-time data. Hence, Spark Streaming is generally used commonly for treating real-time data stream. Spark Streaming provides fault-tolerant and high throughput processing of live streams of data. It is an extra feature that comes with core spark API.

Spark Streaming Features

- **Business Analysis:** With the use of Spark Streaming, one can also learn the behaviour of the audience. These learning can later be used in the decision-making of businesses.
- **Integration:** Real-time and Batch processing is integrated with Spark.
- **Fault Tolerance :** The unique ability of the Spark is that it can recover from the failure efficiently.
- **Speed:** Low Latency is achieved by Spark.
- **Scaling:** Nodes can be scaled easily up to hundreds by Spark.

Spark Streaming Fundamentals

1. Streaming Context

- In Spark the data stream is consumed and managed by Streaming Context. It creates an object of Receiver which is produced by registering an Input streaming. Thus it is the main Spark functionality that becomes a critical entry point to the system as it provides many contexts that provide a default workflow for different sources like Akka Actor, Twitter and ZeroMQ.
- A spark context object represents the connection with a spark cluster. Where the Spark Streaming object is created by a Streaming Context object, accumulators, RDDs and broadcast variables can also be created a Spark Context object.

2. Checkpoints, Broadcast Variables and Accumulators

Checkpoints

- Checkpoint works similar to Checkpoints which stores the state of the systems the same as in the games. Where, in this case, Checkpoints helps in reducing the loss of resources and make the system more resilient to system breakdown. A checkpoint methodology is a better way to keep track of and save the states of the system so that at the time of recovery, it can be easily pulled back.

Broadcast Variables

- Instead of providing the complete copy of tasks to the network Nodes, it always catches a read-only variable which is responsible for acknowledging the nodes of different task present and thus reducing transfer and computation cost by individual nodes. So it can provide a significant input set more efficiently. It also uses advanced algorithms to distribute the broadcast variable to different nodes in the network; thus, the communication cost is reduced.

Accumulators

- Accumulators are variables which can be customized for different purposes. But there also exist already defined Accumulators like counter and sum Accumulators. There is also tracking Accumulators that keeps track of each node, and some extra features can also be added into it. Numeric Accumulators support many digital functions which are also supported by Spark. A custom-defined Accumulators can also be created demanded by the user.

DStream

- DStream means Discretized Stream. Spark Streaming offers the necessary abstraction, which is called Discretized Stream (DStream). DStream is a data which streams continuously. From a source of data, DStream is received. It may also be obtained from a stream of processed data. Transformation of input stream generates processed data stream.
- After a specified interval, data is contained in an RDD. Endless series of RDDs represents a DStream.

Caching

- Developers can use DStream to cache the stream's data in memory. This is useful if the data is computed multiple times in the DStream. It can be achieved by using the persist() method on a DStream.
- Duplication of data is done to ensure the safety of having a resilient system that can resist and failure in the system thus having an ability to tolerate faults in the system (such as Kafka, Sockets, Flume etc.)

Spark Streaming Advantage and Architecture

- Processing of one data stream at a time can be cumbersome at times; hence Spark Streaming discretize the data into small sub batches which are easily manageable. That's because Spark workers get buffers of data in parallel accepted by Spark Streaming receiver. And hence the whole system runs the batches in parallel and then accumulates the final results. Then these short tasks are processed in batches by Spark engine, and the results are provided to other systems.
- In Spark Streaming architecture, the computation is not statically allocated and loaded to a node but based on the data locality and availability of the resources. It is thus reducing loading time as compared to previous traditional systems. Hence the use of data locality principle, it is also easier for fault detection and its recovery.
- Data nodes in Spark are usually represented by RDD that is Resilient Distribution Dataset.

Goals of Spark Streaming

Following are the Goals achieved by Spark architecture.

1. **Dynamic Load Balancing**

- This is one of the essential features of Spark Streaming where data streams are dynamically allocated by the load balancer, which is responsible for allocation data and computation of resources using specific rules defined in it. The main goal of load balancing is to balance the workload efficiently across the workers and put everything in a parallel way such that there is no wastage of resources available. And also responsible for dynamically allocating resource to the worker nodes in the system.

2. **Failure and Recovery**

- As in the traditional system, when there occurs an operation failure, the whole system has to recompute that part to get the lost information back. But the problem comes when one node is handling all this recovery and making the entire system to wait for its completion. Whereas in Spark the lost information is computed by other free nodes and bring back the system to track without any extra waiting like in the traditional methods.
- And also the failed task is distributed evenly on all the nodes in the system to recompute and bring back it from failure faster than the traditional method.

3. **Batches and Interactive Query**

- Set of RDDs in Spark are called to be DStream in Spark that provides a relation between Streaming workloads and batches. These batches are stored in Spark's memory, which provides an efficient way to query the data present in it.

- The best part of Spark is that it includes a wide variety of libraries that can be used when required by the spark system. Few names of the libraries are MLlib for machine learning, SQL for data query, GraphX and Data Frame whereas Dataframe and questions can be converted to equivalent SQL statements by DStreams.

4. Performance

- As the spark system uses parallel distributions of the task that improve its throughput capacity and thus leveraging the sparks engine that capable of achieving low latency as low as up to few 100 milliseconds.

How do Spark Streaming works?

- The data in the stream is divided into small batches which are called DStreams in the Spark Streaming. It is a sequence of RDDs internally. Spark APIs are used by RDDS to process the data and shipments are returned as a result. The API of Spark Streaming is available in Python, Java and Scala. Many features are lacking in the recently introduced Python API in Spark 1.2.
- Stateful computations are called a state that is maintained by the Spark Streaming based on the incoming data in the stream. The data that flows in the stream is processed within a time frame. This time frame is to be specified by the developer, and it is to be allowed by Spark Streaming. The time window is the time frame within which the work should be completed. The time window is updated within a time interval which is also known as the sliding interval in the window.

Spark Streaming Sources

- Receiver object which is related with an input DStream, stores data received, in Sparks Memory for processing.

Built-in streaming has two categories

1. Basic Source

Sources available in Streaming API, e.g. Socket Connection and File System.

2. Advanced Source

Advanced level of sources is Kinesis, Flume & Kafka etc.

Streaming Operations

There are two types of operations which are supported by Spark RDDS, and they are -

1. Output Operations in Apache Spark

- Output Operations are used to push out the data of the DStream into an external system such as a file system or a database. Output Operations allows transformed data to be consumed by the external systems. All the DStreams Transformation are actually executed by the triggering, which is done by the external systems.

These are the current Output operations:

- foreachRDD(func), [suffix]), saveAsHadoopFiles(prefix, [suffix]), saveAsObjectFiles(prefix, [suffix])"prefix-TIME_IN_MS[.suffix]", saveAsTextFiles(prefix, print()
- RDDs lazily execute output Operations. Inside the DStream Operations of Output, RDD Actions are taken forcefully to be processed of the received data. The execution of Output Operations is done one-at-a-time. Spark applications define the order of the performance of the output operations.

2. Spark Transformation

- Spark transformation also changes the data from the DStream as RDDs support it in Spark. Just as Spark RDD's, many alterations are supported by DStream.

Following are the most common Transformation operations:

- Window(), updateStateByKey(), transform(), [numTasks]), cogroup(otherStream, [numTasks]), join(otherStream, reduceByKey(func, [numTasks]), countByValue(), reduce(), union(otherStream), count(), repartition(numPartitions), filter(), flatMap(), map().

3.6 KAFKA

- In Big Data, an enormous volume of data is used. Regarding data, we have two main challenges.The first challenge is how to collect large volume of data and the second challenge is to analyze the collected data. To overcome those challenges, you must need a messaging system.
- Kafka is designed for distributed high throughput systems. Kafka tends to work very well as a replacement for a more traditional message broker. In comparison to other messaging systems, Kafka has better throughput, built-in partitioning, replication and inherent fault-tolerance, which makes it a good fit for large-scale message processing applications.

What is a Messaging System?

- A Messaging System is responsible for transferring data from one application to another, so the applications can focus on data, but not worry about how to share it. Distributed messaging is based on the

concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging system. Two types of messaging patterns are available – one is point to point and the other is publish-subscribe (pub-sub) messaging system. Most of the messaging patterns follow **pub-sub**.

Point to Point Messaging System

- In a point-to-point system, messages are persisted in a queue. One or more consumers can consume the messages in the queue, but a particular message can be consumed by a maximum of one consumer only. Once a consumer reads a message in the queue, it disappears from that queue. The typical example of this system is an Order Processing System, where each order will be processed by one Order Processor, but Multiple Order Processors can work as well at the same time. The following diagram depicts the structure.

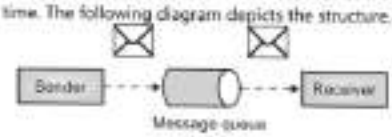


Fig. 3.8

Publish-Subscribe Messaging System

- In the publish-subscribe system, messages are persisted in a topic. Unlike point-to-point system, consumers can subscribe to one or more topic and consume all the messages in that topic. In the Publish-Subscribe system, message producers are called publishers and message consumers are called subscribers. A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.

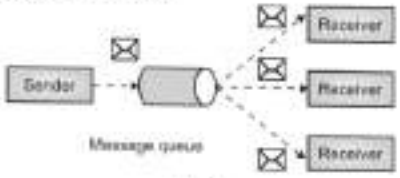


Fig. 3.9

- Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables you to pass messages from one end-point to another. Kafka is suitable for both offline and online message consumption. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. Kafka is built on top of the ZooKeeper synchronization service. It integrates very well with Apache Storm and Spark for real-time streaming data analysis.

Benefits

Following are a few benefits of Kafka :

- **Reliability** – Kafka is distributed, partitioned, replicated and fault tolerance.
- **Scalability** – Kafka messaging system scales easily without down time..
- **Durability** – Kafka uses Distributed commit log which means messages persists on disk as fast as possible, hence it is durable..
- **Performance** – Kafka has high throughput for both publishing and subscribing messages. It maintains stable performance even many TB of messages are stored.

Kafka is very fast and guarantees zero downtime and zero data loss.

Use Cases

Kafka can be used in many Use Cases. Some of them are listed below –

- **Metrics** – Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.
- **Log Aggregation Solution** – Kafka can be used across an organization to collect logs from multiple services and make them available in a standard format to multiple consumers.
- **Stream Processing** – Popular frameworks such as Storm and Spark Streaming read data from a topic, processes it, and write processed data to a new topic where it becomes available for users and applications. Kafka's strong durability is also very useful in the context of stream processing.

Need for Kafka

- Kafka is a unified platform for handling all the real-time data feeds. Kafka supports low latency message delivery and gives guarantee for fault tolerance in the presence of machine failures. It has the ability to handle a large number of diverse consumers. Kafka is very fast, performs two million writes/sec. Kafka persists all data to the disk, which essentially means that all the writes go to the page cache of the OS (RAM). This makes it very efficient to transfer data from page cache to a network socket.
- Before moving deep into the Kafka, you must aware of the main terminologies such as topics, brokers, producers and consumers. The following diagram illustrates the main terminologies and the table describes the diagram components in detail.

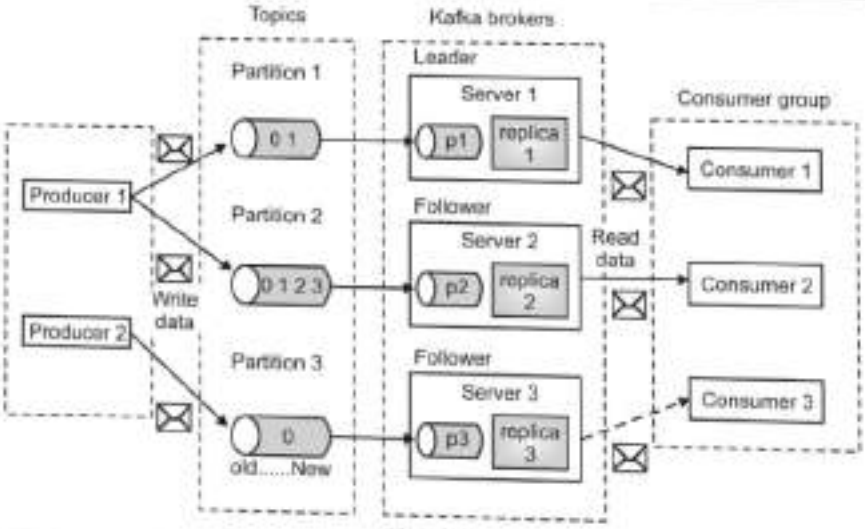


Fig. 3.10

- In the above diagram, a topic is configured into three partitions. Partition 1 has two offset factors 0 and 1. Partition 2 has four offset factors 0, 1, 2, and 3. Partition 3 has one offset factor 0. The id of the replica is same as the id of the server that hosts it.
- Assume, if the replication factor of the topic is set to 3, then Kafka will create 3 identical replicas of each partition and place them in the cluster to make available for all its operations. To balance a load in cluster, each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time.

S. No.	Components and Description
1.	**Topics** A stream of messages belonging to a particular category is called a topic. Data is stored in topics. Topics are split into partitions. For each topic, Kafka keeps a mini-mum of one partition. Each such partition contains messages in an immutable ordered sequence. A partition is implemented as a set of segment files of equal sizes.
2.	**Partition** Topics may have many partitions, so it can handle an arbitrary amount of data.
3.	**Partition Offset** Each partitioned message has a unique sequence id called as offset.
4.	**Replicas of Partition** Replicas are nothing but backups of a partition. Replicas are never read or write data. They are used to prevent data loss.
5.	**Brokers** Brokers are simple system responsible for maintaining the published data. Each broker may have zero or more partitions per topic. Assume, if there are N partitions in a topic and N number of brokers, each broker will have one partition. Assume if there are N partitions in a topic and more than N brokers (n + m), the first N broker will have one partition and the next M broker will not have any partition for that particular topic. Assume if there are N partitions in a topic and less than N brokers (n-m), each broker will have one or more partition sharing among them. This scenario is not recommended due to unequal load distribution among the broker.
6.	**Kafka Cluster** Kafka's having more than one broker are called as Kafka cluster. A Kafka cluster can be expanded without downtime. These clusters are used to manage the persistence and replication of message data.
7.	**Producers** Producers are the publisher of messages to one or more Kafka topics. Producers send data to Kafka brokers. Every time a producer pub-lishes a message to a broker, the broker simply appends the message to the last segment file. Actually, the message will be appended to a partition. Producer can also send messages to a partition of their choice.
8.	**Consumers** Consumers read data from brokers. Consumers subscribes to one or more topics and consume published messages by pulling data from the brokers.
9.	**Leader** Leader is the node responsible for all reads and writes for the given partition. Every partition has one server acting as a leader.
10.	**Follower** Node which follows leader instructions are called as follower. If the leader fails, one of the follower will automatically become the new leader. A follower acts as normal consumer, pulls messages and up-dates its own data store.

- Take a look at the following illustration. It shows the cluster diagram of Kafka.

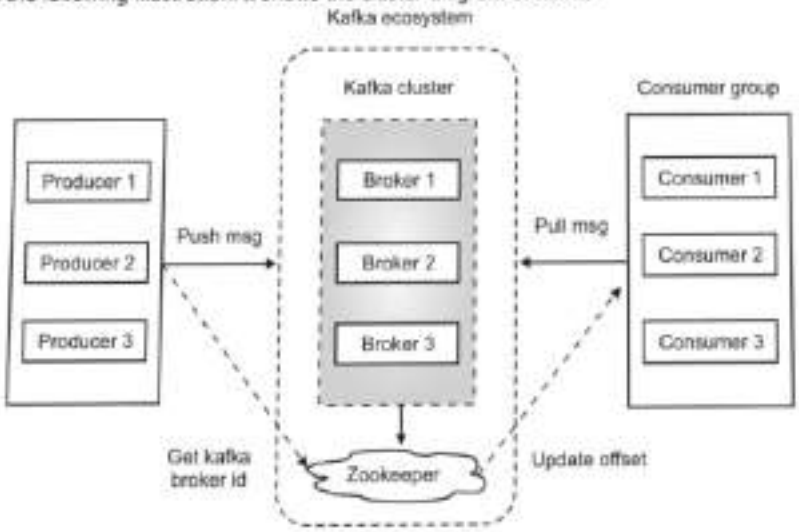


Fig. 3.11

The following table describes each of the components shown in the above diagram.

S. No	Components and Description
1.	**Broker** Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state. One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each broker can handle TB of messages without performance impact. Kafka broker leader election can be done by ZooKeeper.
2.	**ZooKeeper** ZooKeeper is used for managing and coordinating Kafka broker. ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system. As per the notification received by the Zookeeper regarding presence or failure of the broker then pro-ducer and consumer takes decision and starts coordinating their task with some other broker.
3.	**Producers** Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker. Kafka producer doesn't wait for acknowledgements from the broker and sends messages as fast as the broker can handle.
4.	**Consumers** Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset. If the consumer acknowledges a particular message offset, it implies that the consumer has consumed all prior messages. The consumer issues an asynchronous pull request to the broker to have a buffer of bytes ready to consume. The consumers can rewind or skip to any point in a partition simply by supplying an offset value. Consumer offset value is notified by ZooKeeper.

- As of now, we discussed the core concepts of Kafka. Let us now throw some light on the workflow of Kafka.
- Kafka is simply a collection of topics split into one or more partitions. A Kafka partition is a linearly ordered sequence of messages, where each message is identified by their index (called as offset). All the data in a Kafka cluster is the disjointed union of partitions. Incoming messages are written at the end of a partition and messages are sequentially read by consumers. Durability is provided by replicating messages to different brokers.
- Kafka provides both pub-sub and queue based messaging system in a fast, reliable, persisted, fault-tolerance and zero downtime manner. In both cases, producers simply send the message to a topic and consumer can choose any one type of messaging

system depending on their need. Let us follow the steps in the next section to understand how the consumer can choose the messaging system of their choice.

Workflow of Pub-Sub Messaging

Following is the step wise workflow of the Pub-Sub Messaging –

- Producers send message to a topic at regular intervals.
- Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second partition.
- Consumer subscribes to a specific topic.
- Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper ensemble.
- Consumer will request the Kafka in a regular interval (like 100 Ms) for new messages.
- Once Kafka receives the messages from producers, it forwards these messages to the consumers.
- Consumer will receive the message and process it.
- Once the messages are processed, consumer will send an acknowledgement to the Kafka broker.
- Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper, the consumer can read next message correctly even during server outrages.
- This above flow will repeat until the consumer stops the request.
- Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

Workflow of Queue Messaging / Consumer Group

- In a queue messaging system instead of a single consumer, a group of consumers having the same Group ID will subscribe to a topic. In simple terms, consumers subscribing to a topic with same Group ID are considered as a single group and the messages are shared among them. Let us check the actual workflow of this system.
  - ➢ Producers send message to a topic in a regular interval.
  - ➢ Kafka stores all messages in the partitions configured for that particular topic similar to the earlier scenario.
  - ➢ A single consumer subscribes to a specific topic, assume Topic-01 with Group ID as Group-1.
  - ➢ Kafka interacts with the consumer in the same way as Pub-Sub Messaging until new consumer subscribes the same topic, Topic-01 with the same Group ID as Group-1.
  - ➢ Once the new consumer arrives, Kafka switches its operation to share mode and shares the data between the two consumers. This sharing will go on until the number of con-sumers reach the number of partition configured for that particular topic.
  - ➢ Once the number of consumer exceeds the number of partitions, the new consumer will not receive any further message until any one of the existing consumer unsubscribes. This scenario arises because each consumer in Kafka will be assigned a minimum of one partition and once all the partitions are assigned to the existing consumers, the new consumers will have to wait.
  - ➢ This feature is also called as Consumer Group. In the same way, Kafka will provide the best of both the systems in a very simple and efficient manner.

Role of ZooKeeper

- A critical dependency of Apache Kafka is Apache Zookeeper, which is a distributed configuration and synchronization service. Zookeeper serves as the coordination interface between the Kafka brokers and consumers. The Kafka servers share information via a Zookeeper cluster. Kafka stores basic metadata in Zookeeper such as information about topics, brokers, consumer offsets (queue readers) and so on.
- Since all the critical information is stored in the Zookeeper and it normally replicates this data across its ensemble, failure of Kafka broker / Zookeeper does not affect the state of the Kafka cluster. Kafka will restore the state, once the Zookeeper restarts. This gives zero downtime for Kafka. The leader election between the Kafka broker is also done by using Zookeeper in the event of leader failure.

Advantages of Kafka

- So, here we are listing out some of the advantages of Kafka. Basically, these Kafka advantages are making Kafka ideal for our data lake implementation. So, let's start learning advantages of Kafka in detail.

Fig. 3.12

Kafka Pros and Cons – Kafka Advantages

1. **High-Throughput**

   Without having not so large hardware, Kafka is capable of handling high-velocity and high-volume data. Also, able to support message throughput of thousands of messages per second.

2. **Low Latency**

   It is capable of handling these messages with the very low latency of the range of milliseconds, demanded by most of the new use cases.

3. **Fault-Tolerant**

   One of the best advantages is Fault Tolerance. There is an inherent capability in Kafka, to be resistant to node/machine failure within a cluster.

4. **Durability**

   Here, durability refers to the persistence of data/messages on disk. Also, messages replication is one of the reasons behind durability, hence messages are never lost.

5. **Scalability**

   Without incurring any downtime on the fly by adding additional nodes, Kafka can be scaled-out. Moreover, inside the Kafka cluster, the message handling is fully transparent and these are seamless.

6. **Distributed**

   The distributed architecture of Kafka makes it scalable using capabilities like replication and partitioning.

7. **Message Broker Capabilities**

   Kafka tends to work very well as a replacement for a more traditional message broker. Here, a message broker refers to an intermediary program, which translates messages from the formal messaging protocol of the publisher to the formal messaging protocol of the receiver.

8. **High Concurrency**

   Kafka is able to handle thousands of messages per second and that too in low latency conditions with high throughput. In addition, it permits the reading and writing of messages into it at high concurrency.

9. **By Default Persistent**

   As we discussed above that the messages are persistent, that makes it durable and reliable.

10. **Consumer Friendly**

    It is possible to integrate with the variety of consumers using Kafka. The best part of Kafka is, it can behave or act differently according to the consumer, that it integrates with because each customer has a different ability to handle these messages, coming out of Kafka. Moreover, Kafka can integrate well with a variety of consumers written in a variety of languages.

11. **Batch Handling Capable (ETL Like Functionality)**

    Kafka could also be employed for batch-like use cases and can also do the work of a traditional ETL, due to its capability of persists messages.

12. **Variety of Use Cases**

    It is able to manage the variety of use cases commonly required for a Data Lake. For Example log aggregation, web activity tracking, and so on.

13. **Real-Time Handling**

    Kafka can handle real-time data pipeline. Since we need to find a technology piece to handle real-time messages from applications, it is one of the core reasons for Kafka as our choice.

Disadvantages of Kafka

Fig. 3.13

Cons of Kafka – Apache Kafka Disadvantages

- It is good to know Kafka's limitations even if its advantages appear more prominent than its disadvantages. However, consider it only when advantages are too compelling to omit. Here is one more condition that some disadvantages might be more relevant for a particular use case but not really linked to ours. So, here we are listing out some of the disadvantage associated with Kafka:

1. **No Complete Set of Monitoring Tools**

- It is seen that it lacks a full set of management and monitoring tools. Hence, enterprise support staff felt anxious or fearful about choosing Kafka and supporting it in the long run.

2. **Issues with Message Tweaking**

- As we know, the broker uses certain system calls to deliver messages to the consumer. However, Kafka's performance reduces significantly if the message needs some tweaking. So, it can perform quite well if the message is unchanged because it uses the capabilities of the system.

3. **Not Support Wildcard Topic Selection**

There is an issue that Kafka only matches the exact topic name, that means it does not support wildcard topic selection. Because that makes it incapable of addressing certain use cases.

4. **Lack of Pace**

There can be a problem because of the lack of pace, while API's which are needed by other languages are maintained by different individuals and corporates.

5. **Reduces Performance**

In general, there are no issues with the individual message size. However, the brokers and consumers start compressing these messages as the size increases. Due to this, when decompressed, the node memory gets slowly used. Also, compress happens when the data flow in the pipeline. It affects throughput and also performance.

6. **Behaves Clumsy**

Sometimes, it starts behaving a bit clumsy and slow, when the number of queues in a Kafka cluster increases.

7. **Lacks Some Messaging Paradigms**

Some of the messaging paradigms are missing in Kafka like request/reply, point-to-point queues and so on. Not always but for certain use cases, it sounds problematic.

3.7 STREAMING ECOSYSTEM

- "Big-data" is one of the most inflated buzzword of the last years. Technologies born to handle huge datasets and overcome limits of previous products are gaining popularity outside the research environment. The following list would be a reference of this world. It's still incomplete and always will be.

3.7.1 Getting Data into HDFS

- Most of the big data originates outside the Hadoop cluster. These tools will help you get data into HDFS.

Table 3.1 : Tools for Getting Data into HDFS

Tool	Remarks
Flume	Gathers data from multiple sources and gets it into HDFS.
Scribe	Distributed log gatherer, originally developed by Facebook. It hasn't been updated recently.
Chukwa	Data collection system
Sqoop	Transfers data between Hadoop and Relational Databases (RDBMS)
Kafka	Distributed publish-subscribe system.

3.7.2 Compute Frameworks

Table 3.2 : Hadoop Compute Frameworks

Tool	Remarks
Map reduce	Original distributed compute framework of Hadoop
YARN	Next generation MapReduce, available in Hadoop version 2.0
Weave	Simplified YARN programming
Cloudera SDK	Simplified MapReduce programming

3.7.3 Querying Data in HDFS

Table 3.3 : Querying Data in HDFS

Tool	Remarks
Java MapReduce	Native mapreduce in Java
Hadoop Streaming	Map Reduce in other languages (Ruby, Python)
Pig	Pig provides a higher level data flow language to process data. Pig scripts are much more compact than Java Map Reduce code.
Hive	Hive provides an SQL layer on top of HDFS. The data can be queried using SQL rather than writing Java Map Reduce code.
Cascading Lingual	Executes ANSI SQL queries as Cascading applications on Apache Hadoop clusters.
Stinger / Tez	Next generation Hive.
Hadapt	Provides SQL support for Hadoop. (commercial product)
Greenplum HAWQ	Relational database with SQL support on top of Hadoop HDFS. (commercial product)
Cloudera Search	Text search on HDFS
Impala	Provides real time queries over Big Data. Developed by Cloudera.
Presto	Developed by Facebook, provides fast SQL querying over Hadoop

3.7.4 SQL on Hadoop / HBase

Table 3.4 : SQL Querying Data in HDFS

Tool	Remarks
Hive	Hive provides an SQL layer on top of HDFS. The data can be queried using SQL rather than writing Java Map Reduce code.
Stinger / Tez	Next generation Hive.
Hadapt	Provides SQL support for Hadoop. (commercial product)
Greenplum HAWQ	Relational database with SQL support on top of Hadoop HDFS. (commercial product)
Impala	Provides real time queries over Big Data. Developed by Cloudera.
Presto	Developed by Facebook, provides fast SQL querying over Hadoop
Phoenix	SQL layer over HBase. Developed by SalesForce.com.
Spire	SQL layer over HBase. Developed by DrawnToScale.com.
Citus Data	Relational database with SQL support on top of Hadoop HDFS. (commercial product)
Apache Drill	Interactive analysis of large scale data sets.
Presto	Developed by Facebook, provides fast SQL querying over Hadoop

3.7.5 Real Time Querying

Table 3.5 : Real Time Queries

Tool	Remarks
Apache Drill	Interactive analysis of large scale data sets.
Impala	Provides real time queries over Big Data. Developed by Cloudera.

3.7.6 Stream Processing

Table 3.6 : Stream Processing Tools

Tool	Remarks
Storm	Fast stream processing developed by Twitter.
Apache S4	
Samza	
Malhar	Massively scalable, fault-tolerant, stateful native Hadoop platform, developed by DataTorrent

3.7.7 NoSQL Stores

Table 3.7 : NoSQL Stores for Big Data

Tool	Remarks
HBase	NoSQL built on top of Hadoop.
Cassandra	NoSQL store (does not use Hadoop).
Redis	Key value store.
Amazon SimpleDB	Offered by Amazon on their environment.
Voldemort	Distributed key value store developed by LinkedIn.
Accumulo	A NoSQL store developed by NSA (yes, that agency!).

3.7.8 Hadoop in the Cloud

Table 3.8 : Hadoop in the Cloud

Tool	Remarks
Amazon Elastic Map Reduce (EMR)	On demand Hadoop on Amazon Cloud.
Hadoop on Rackspace	On demand and managed Hadoop at Rackspace
Hadoop on Google Cloud	Hadoop runs on Google Cloud
Whirr	Tool to easily spin up and manage Hadoop clusters on cloud services like Amazon / RackSpace.

3.7.9 Work Flow Tools / Schedulers

Table 3.9 : Work Flow Tools

Tool	Remarks
Oozie	Orchestrates map reduce jobs.
Azkaban	
Cascading	Application framework for Java developers to develop robust Data Analytics and Data Management applications on Apache Hadoop.
Scalding	Scala library that makes it easy to specify Hadoop MapReduce jobs. Scalding is built on top of Cascading.
Lipstick	Pig work flow visualization

3.7.10 Serialization Frameworks

Table 3.10 : Serialization Frameworks

Tool	Remarks
Avro	Data serialization system.
Trevni	Column file format.
Protobuf	Popular serialization library (not a Hadoop project).
Parquet	columnar storage format for Hadoop

3.7.11 Monitoring Systems

Table 3.11 : Tools for Monitoring Hadoop

Tool	Remarks
Hue	Developed by Cloudera.
Ganglia	Overall host monitoring system. Hadoop can publish metrics to Ganglia.
Open TSDB	Metrics collector and visualizer.
Nagios	IT infrastructure monitoring.

3.7.12 Applications / Platforms

Table 3.12 : Applications that run on top of Hadoop

Tool	Remarks
Mahout	Recommendation engine on top of Hadoop.
Giraph	Fast graph processing on top of Hadoop.
Lily	Lily unifies Apache HBase, Hadoop and Solr into a comprehensively integrated, interactive data platform

3.7.13 Distributed Coordination

Table 3.13 : Distributed Coordination

Tool	Remarks
Zookeeper	ZooKeeper is a centralized service for maintaining configuration information, naming, and providing distributed synchronization.
Book keeper	Distributed logging service based on ZooKeeper.

3.7.14 Data Analytics Tools

Table 3.14 : Data Analytics on Hadoop

Tool	Remarks
R language	Software environment for statistical computing and graphics.
RHIPE	Integrates R and Hadoop.

3.7.15 Distributed Message Processing

Table 3.15 : Distributed Message Processing

Tool	Remarks
Kafka	Distributed publish-subscribe system.
Akka	Distributed messaging system with actors.
RabbitMQ	Distributed MQ messaging system.

3.7.16 Business Intelligence (BI) Tools

Table 3.16 : Business Intelligence (BI) Tools

Tool	Remarks
Datameer	
Tableau	
Pentaho	
SiSense	
SumoLogic	

3.7.17 YARN-Based Frameworks

Table 3.17 : YARN-Based Frameworks

Tool	Remarks
Samza	
Spark	Spark on YARN
Malhar	
Giraph	Giraph on YARN
Storm	Storm on YARN
Hoya (HBase on YARN)	
Malhar	

3.7.18 Libraries / Frameworks

Table 3.18 : Libraries / Frameworks

Tool	Remarks
Kiji	Build Real-time Big Data Applications on Apache HBase
Elephant Bird	Compression codes and serializers for Hadoop
Summing Bird	MapReduce on Storm / Scalding
Apache Crunch	Simple, efficient MapReduce pipelines for Hadoop and Spark
Apache DataFu	Pig UDFs that provide cool functionality
Continuity	Build apps on Hbase easily

3.7.19 Data Management

Table 3.19 : Data Management

Tool	Remarks
Apache Falcon	Data management, data lineage

3.7.20 Security

Table 3.20 : Security

Tool	Remarks
Apache Sentry	
Apache Knox	

3.7.21 Testing Frameworks

Table 3.21 : Testing Frameworks

Tool	Remarks
MRUnit	Unit Testing frameworks for Java MapReduce
PigUnit	For testing Pig scripts

3.7.22 Miscellaneous

Table 3.22 : Miscellaneous Stuff

Tool	Remarks
Spark	In memory analytics engine developed by Berkeley AMP labs.
Shark (Hive on Spark)	Hive compatible data warehouse system developed at Berkeley. Claims to be much faster than Hive.

EXERCISE

1. What is streaming? Mention the importance of Big Data platforms.
2. Write short note on: Big Data streaming platforms for fast data.
3. Describe any three Big Data streaming platforms for fast data.
4. What is streaming of the data? Give the typical examples of the streaming data.
5. Draw and explain the streaming data architecture.
6. Draw and explain the modern streaming architecture. Also state the benefits of a modern streaming architecture.
7. Give any three typical use cases of streaming data.
8. What is the need of big data pipeline?
9. Draw and explain the Lambda architecture of real time Big Data pipeline.
10. Draw and explain the Kappa architecture of real time Big Data pipeline.
11. Describe the features of Big Data pipeline.
12. State and explain the components of Big Data Pipeline.
13. Write short note on Spark Streaming.
14. Describe the four major aspects of spark streaming and working details.
15. What is Kafka? Give the advantages of Kafka.
16. Describe the need of Kafka and any 2 uses cases of the same.
17. Describe the following terms:
    a. Topics b. Partition
    c. Broker d. Kafka Cluster
    e. Producer f. Consumer
    g. leader h. follower
18. Mention the advantages and disadvantages of Kafka.
19. Describe in brief the concept of streaming ecosystem.

UNIT IV

BIG DATA APPLICATIONS

4.1 INTRODUCTION TO BIG DATA APPLICATIONS

- The quantities, characters, or symbols on which operations are performed by a computer, which may be stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.
- Big Data is also **data** but with a **huge size**. Big Data is a term used to describe a collection of data that is huge in volume and yet growing exponentially with time. In short such data is so large and complex that none of the traditional data management tools are able to store it or process it efficiently.

Following are some the examples of Big Data-

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

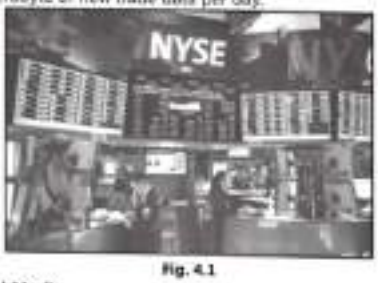


Fig. 4.1

Social Media

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

Fig. 4.2

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

Fig. 4.3

Types of Big Data

BigData could be found in three forms:

1. Structured
2. Unstructured
3. Semi-structured

1. Structured

- Any data that can be stored, accessed and processed in the form of fixed format is termed as a 'structured' data. Over the period of time, talent in computer science has achieved greater success in developing techniques for working with such kind of data (where the format is well known in advance) and also deriving value out of it. However, nowadays, we are foreseeing issues when a size of such data grows to a huge extent, typical sizes are being in the rage of multiple zettabytes.

Note : 1021 bytes equal to 1 zettabyte or one billion terabytes forms a zettabyte.

- Looking at these figures one can easily understand why the name Big Data is given and imagine the challenges involved in its storage and processing.

Note : Data stored in a relational database management system is one example of a 'structured' data.

Example of Structured Data

- An 'Employee' Table 4.1 in a database is an example of Structured Data

Table 4.1

Employee_ID	Employee_Name	Gender	Department	Salary_In lacs
2365	Rajesh Kulkarni	Male	Finance	650000
3398	Pratibha Joshi	Female	Admin	650000
7465	Shushil Roy	Male	Admin	500000
7500	Shubhojit Das	Male	Finance	500000
7699	Priya Sane	Female	Finance	550000

2. Unstructured

- Any data with unknown form or the structure is classified as unstructured data. In addition to the size being huge, un-structured data poses multiple challenges in terms of its processing for deriving value out of it. A typical example of unstructured data is a heterogeneous data source containing a combination of simple text files, images, videos etc. Now day organizations have wealth of data available with them but unfortunately, they don't know how to derive value out of it since this data is in its raw form or unstructured format.

Example of Unstructured Data

The output returned by 'Google Search'

3. Semi-Structured

- Semi-structured data can contain both the forms of data. We can see semi-structured data as a structured in form but it is actually not defined with e.g. a table definition in relational DBMS. Example of semi-structured data is a data represented in an XML file.

Example of Semi-Structured Data

Personal data stored in an XML file-

```
<rec><name>Prashant
Rao</name><sex>Male</sex><age>35</age></rec>
<rec><name>Seema
R.</name><sex>Female</sex><age>41</age></rec>
<rec><name>Satish
Mane</name><sex>Male</sex><age>29</age></rec>
<rec><name>Subrata
Roy</name><sex>Male</sex><age>26</age></rec>
<rec><name>Jeremiah
J.</name><sex>Male</sex><age>35</age></rec>
```

Data Growth over the Years

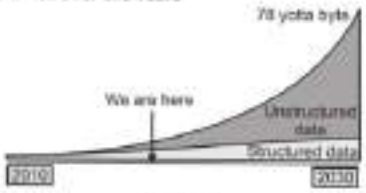


Fig. 4.4

- Please note that web application data, which is unstructured, consists of log files, transaction history files etc. OLTP systems are built to work with structured data wherein data is stored in relations (tables).

Characteristics of Big Data

1. **Volume :** The name Big Data itself is related to a size which is enormous. Size of data plays a very crucial role in determining value out of data. Also, whether a particular data can actually be considered as a Big Data or not, is dependent upon the volume of data. Hence, 'Volume' is one characteristic which needs to be considered while dealing with Big Data.
2. **Variety :** The next aspect of Big Data is its variety. Variety refers to heterogeneous sources and the nature of data, both structured and unstructured. During earlier days, spreadsheets and databases were the only sources of data considered by most of the applications. Nowadays, data in the form of emails, photos, videos, monitoring devices, PDFs, audio, etc. are also being considered in the analysis applications. This variety of unstructured data poses certain issues for storage, mining and analyzing data.
3. **Velocity :** The term 'velocity' refers to the speed of generation of data. How fast the data is generated and processed to meet the demands, determines real potential in the data.

   Big Data Velocity deals with the speed at which data flows in from sources like business processes, application logs, networks, and social media sites, sensors, Mobile devices, etc. The flow of data is massive and continuous.
4. **Variability :** This refers to the inconsistency which can be shown by the data at times, thus hampering the process of being able to handle and manage the data effectively.

Benefits of Big Data Processing

- Ability to process Big Data brings in multiple benefits, such as-
  - Businesses can utilize outside intelligence while taking decisions
- Access to social data from search engines and sites like facebook, twitter are enabling organizations to fine tune their business strategies.
  - Improved customer service
- Traditional customer feedback systems are getting replaced by new systems designed with Big Data technologies. In these new systems, Big Data and natural language processing technologies are being used to read and evaluate consumer responses.
  - Early identification of risk to the product/services, if any
  - Better operational efficiency
- Big Data technologies can be used for creating a staging area or landing zone for new data before identifying what data should be moved to the data warehouse. In addition, such integration of Big Data technologies and data warehouse helps an organization to offload infrequently accessed data.

4.2 OVERVIEW OF BIG DATA MACHINE LEARNING

- In the past few years, more data has been produced than in the millennia of human history before. This data represents a gold mine in terms of commercial value and also important reference material for policy makers. But much of this value will stay untapped or, worse, be misinterpreted as long as the tools necessary for processing the staggering amount of information remain unavailable.
- Data consists of numbers, words, measurements and observations formatted in ways computers can process. Big data refers to vast sets of that data, either structured or unstructured.
- The digital era presents a challenge for traditional data-processing software: information becomes available in such volume, velocity and variety that it ends up outpacing human-centered computation. And we can describe big data using these three "V"s: volume, velocity and variety. Volume refers to the scale of available data; velocity is the speed with which data is accumulated; variety refers to the different sources it comes from.
- Two other Vs are often added to the aforementioned three: Veracity refers to the consistency and certainty (or lack thereof) in the sourced data, while value measures the usefulness of the data that's been extracted from the data received.
- Good data analysis requires someone with business acumen, programming knowledge and a comprehensive skill set of math and analytic techniques. But how can a professional armed with traditional techniques sort through millions of credit card scores, or billions of social media interactions? That's where machine learning comes in.
- Machine-learning algorithms become more effective as the size of training datasets grows. So when combining big data with machine learning, we benefit twice: the algorithms help us keep up with the continuous influx of data, while the volume and variety of the same data feeds the algorithms and helps them grow.

Let's look at how this integration process might work:

- By feeding big data to a machine-learning algorithm, we might expect to see defined and analyzed results, like hidden patterns and analytics, that can assist in predictive modeling.
- For some companies, these algorithms might automate processes that were previously human-centered. But more often than not, a company will review the algorithm's findings and search them for valuable insights that might guide business operations.
- Here's where people come back into the picture. While AI and data analytics run on computers that outperform humans by a vast margin, they lack certain decision-making abilities. Computers have yet to replicate many characteristics inherent to humans, such as critical thinking, intention and the ability to use holistic approaches. Without an expert to provide the right data, the value of algorithm-generated results diminishes, and without an expert to interpret its output, suggestions made by an algorithm may compromise company decisions.
- Let's look at some real-life examples that demonstrate how big data and machine learning can work together.

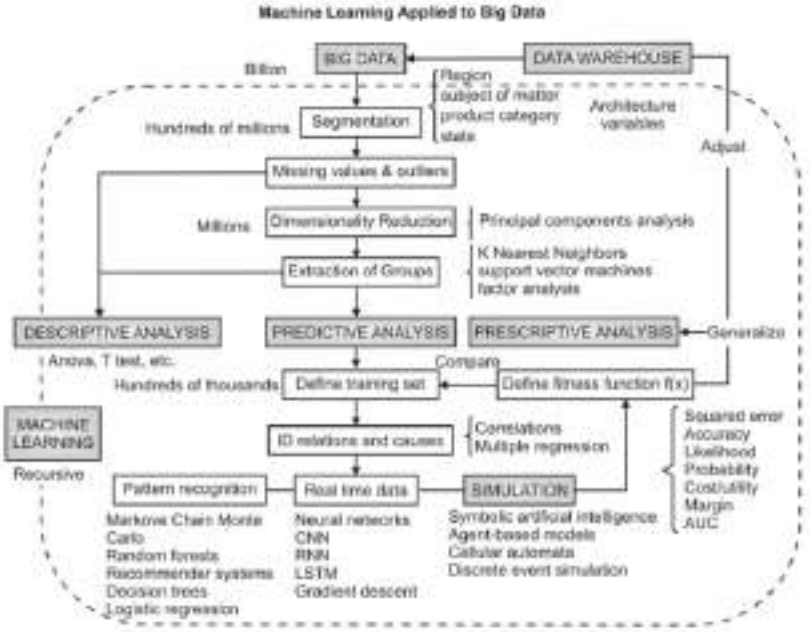


Fig. 4.5

Cloud Networks

- A research firm has a large amount of medical data it wants to study, but in order to do so on-premises it needs servers, online storage, networking and security assets, all of which adds up to an unreasonable expense. Instead, the firm decides to invest in Amazon EMR, a cloud service that offers data-analysis models within a managed framework.
- Machine-learning models of this sort include GPU-accelerated image recognition and text classification. These algorithms don't learn once they are deployed, so they can be distributed and supported by a Content Delivery Network (CDN). Check out LiveRamp's detailed outline describing the migration of a big-data environment to the cloud.

Web Scraping

- Let's imagine that a manufacturer of kitchen appliances learns about market tendencies and customer-satisfaction trends from a retailer's quarterly reports. In their desire to find out what the reports might have left out, the manufacturer decides to web-scrape the enormous amount of existing data that pertains to online customer feedback and product reviews. By aggregating this data and feeding it to a deep-learning model, the manufacturer learns how to improve and better describe its products, resulting in increased sales.
- While web scraping generates a huge amount of data, it's worthwhile to note that choosing the sources for this data is the most important part of the process. Check out this IT Svit guid for some best data-mining practices.

Mixed-Initiative Systems

- The recommendation system that suggests titles on your Netflix homepage employs collaborative filtering. It uses big data to track your history (and everyone else's) and machine-learning algorithms to decide what it should recommend next. This example demonstrates how big data and machine learning intersect in the arena of mixed-initiative systems, or

human-computer interactions, whose results come from humans and/or machines taking initiative.

- Similarly, smart-car manufacturers implement big data and machine learning in the predictive-analytics systems that run their products. Tesla cars, for example, communicate with their drivers and respond to external stimuli by using data to make algorithm-based decisions.
- Achieving accurate results from machine learning has a few prerequisites. Apart from a well-built learning algorithm, you need clean data, scalable tools and a clear idea of what you want to achieve. While some might see these requirements as obstacles preventing their business from reaping the benefits of using big data with machine learning, in fact any business wishing to correctly implement this technology should invest in them.

Data Hygiene

- Just as training for a sport can become dangerous for injury-prone athletes, learning from unsanitized or incorrect data can get expensive. Incorrectly trained algorithms produce results that will incur costs for a company and not save on them, as discussed in the article Towards Data Science. Because mislabeled, missing or irrelevant data can impact the accuracy of your algorithm, you must be able to attest to the quality and completeness of your data sets as well as their sources.

Practicing with Real Data

- Suppose you want to create a machine-learning algorithm but lack the massive amount of data required to train it. You hear somewhere that *derived computed data* could be substituted for real data you generated. But beware: Because an ideal algorithm should solve a specific problem, it needs a specific type of data to learn from. Derived data rarely mimics the real data the algorithm needs to solve the problem, so using it almost guarantees that the trained algorithm will not fulfill its potential. Experimenting with real data offers the safest path.

Knowing What You Want to Achieve

- Don't let the hype around integrating machine learning with big data end up catapulting you into a poor understanding of the problem you want to solve. If you've pinpointed a complex problem but don't know how to use your data to solve it, you could wind up feeding inappropriate data to your algorithm or using correct data in inaccurate ways. To harness the power of big data, we recommend taking the time needed to create your own data before diving into an algorithm. That way you can educate yourself about your data, so when the time comes, you can use (and train) an algorithm appropriate to your problem.

Scaling Tools

- Big data gives us access to more information, and machine learning increases our problem-solving capacity. Put together, the two present opportunities to scale entire businesses. To take advantage of this, we should also prepare our other tools (in the realms of finance, communication, etc.) for scaling.
- The core of machine learning consists of self-learning algorithms that evolve by continuously improving at their assigned task. When structured correctly and fed proper data, these algorithms eventually produce results in the contexts of pattern recognition and predictive modeling.
- For machine-learning algorithms, data is like exercise: the more the better. Algorithms fine-tune themselves with the data they train on in the same way Olympic athletes hone their bodies and skills by training every day.
- Many programming languages work with machine learning, including Python, R, Java, JavaScript and Scala. Python is the preferred choice for many developers because of its TensorFlow library, which offers a comprehensive ecosystem of machine-learning tools. If you'd like to practice coding on an actual algorithm, check out our article on machine learning with Python.

4.3 MAHOUT

- We are living in a day and age where information is available in abundance. The information overload has scaled to such heights that sometimes it becomes difficult to manage our little mailboxes! Imagine the volume of data and records some of the popular websites (the likes of Facebook, Twitter, and Youtube) have to collect and manage on a daily basis. It is not uncommon even for lesser known websites to receive huge amounts of information in bulk.

- Normally we fall back on data mining algorithms to analyze bulk data to identify trends and draw conclusions. However, no data mining algorithm can be efficient enough to process very large datasets and provide outcomes in quick time, unless the computational tasks are run on multiple machines distributed over the cloud.
- We now have new frameworks that allow us to break down a computation task into multiple segments and run each segment on a different machine. Mahout is such a data mining framework that normally runs coupled with the Hadoop infrastructure at its background to manage huge volumes of data.

What is Apache Mahout?

- A mahout is one who drives an elephant as its master. The name comes from its close association with Apache Hadoop which uses an elephant as its logo.
- Hadoop is an open-source framework from Apache that allows to store and process big data in a distributed environment across clusters of computers using simple programming models.
- Apache Mahout is an open source project that is primarily used for creating scalable machine learning algorithms. It implements popular machine learning techniques such as:
  - ➢ Recommendation
  - ➢ Classification
  - ➢ Clustering
- Apache Mahout started as a sub-project of Apache's Lucene in 2008. In 2010, Mahout became a top level project of Apache.

Features of Mahout

The primitive features of Apache Mahout are listed as follows.

- The algorithms of Mahout are written on top of Hadoop, so it works well in distributed environment. Mahout uses the Apache Hadoop library to scale effectively in the cloud.
- Mahout offers the coder a ready-to-use framework for doing data mining tasks on large volumes of data.
- Mahout lets applications to analyze large sets of data effectively and in quick time.
- Includes several MapReduce enabled clustering implementations such as k-means, fuzzy k-means, Canopy, Dirichlet, and Mean-Shift.
- Supports Distributed Naive Bayes and Complementary Naive Bayes classification implementations.
- Comes with distributed fitness function capabilities for evolutionary programming.
- Includes matrix and vector libraries.

Applications of Mahout

- Companies such as Adobe, Facebook, LinkedIn, Foursquare, Twitter, and Yahoo use Mahout internally.
- Foursquare helps you in finding out places, food, and entertainment available in a particular area. It uses the recommender engine of Mahout.
- Twitter uses Mahout for user interest modelling.
- Yahoo! uses Mahout for pattern mining.
- Apache Mahout is a highly scalable machine learning library that enables developers to use optimized algorithms. Mahout implements popular machine learning techniques such as recommendation, classification, and clustering. Therefore, it is prudent to have a brief section on machine learning before we move further.
- Machine learning is a branch of science that deals with programming the systems in such a way that they automatically learn and improve with experience. Here, learning means recognizing and understanding the input data and making wise decisions based on the supplied data.
- It is very difficult to cater to all the decisions based on all possible inputs. To tackle this problem, algorithms are developed. These algorithms build knowledge from specific data and past experience with the principles of statistics, probability theory, logic, combinatorial optimization, search, reinforcement learning, and control theory.
- The developed algorithms form the basis of various applications such as:
  - ➢ Vision processing
  - ➢ Language processing
  - ➢ Forecasting (e.g., stock market trends)
  - ➢ Pattern recognition
  - ➢ Games
  - ➢ Data mining
  - ➢ Expert systems
  - ➢ Robotics

- Machine learning is a vast area and it is quite beyond the scope of this tutorial to cover all its features. There are several ways to implement machine learning techniques, however the most commonly used ones are supervised and unsupervised learning.

Supervised Learning

- Supervised learning deals with learning a function from available training data. A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. Common examples of supervised learning include:
  - Classifying e-mails as spam,
  - Labeling webpages based on their content, and
  - Voice recognition.
- There are many supervised learning algorithms such as neural networks, Support Vector Machines (SVMs), and Naive Bayes classifiers. Mahout implements Naive Bayes classifier.

Unsupervised Learning

- Unsupervised learning makes sense of unlabeled data without having any predefined dataset for its training. Unsupervised learning is an extremely powerful tool for analyzing available data and look for patterns and trends. It is most commonly used for clustering similar input into logical groups. Common approaches to unsupervised learning include:
  - k-means
  - self-organizing maps, and
  - hierarchical clustering

Recommendation

- Recommendation is a popular technique that provides close recommendations based on user information such as previous purchases, clicks, and ratings.
- Amazon uses this technique to display a list of recommended items that you might be interested in, drawing information from your past actions. There are recommender engines that work behind Amazon to capture user behavior and recommend selected items based on your earlier actions.
- Facebook uses the recommender technique to identify and recommend the "people you may know list".

Fig. 4.6

Classification

- Classification, also known as categorization, is a machine learning technique that uses known data to determine how the new data should be classified into a set of existing categories. Classification is a form of supervised learning.
- Mail service providers such as Yahoo! and Gmail use this technique to decide whether a new mail should be classified as a spam. The categorization algorithm trains itself by analyzing user habits of marking certain mails as spams. Based on that, the classifier decides whether a future mail should be deposited in your inbox or in the spams folder.
- iTunes application uses classification to prepare playlists.

Fig. 4.7

Clustering

- Clustering is used to form groups or clusters of similar data based on common characteristics. Clustering is a form of unsupervised learning.

- Search engines such as Google and Yahoo! use clustering techniques to group data with similar characteristics.
- Newsgroups use clustering techniques to group various articles based on related topics.
- The clustering engine goes through the input data completely and based on the characteristics of the data, it will decide under which cluster it should be grouped. Take a look at the following example.

Installing Mahout

- Java and Hadoop are the prerequisites of mahout. Following are the steps to download and install Java, Hadoop, and Mahout.

Pre-Installation Setup

- Before installing Hadoop into Linux environment, we need to set up Linux using ssh (Secure Shell). Follow the steps mentioned below for setting up the Linux environment.

Creating a User

- It is recommended to create a separate user for Hadoop to isolate the Hadoop file system from the Unix file system. Follow the steps given below to create a user.
- Open root using the command "su".
- Create a user from the root account using the command "useradd username".
- Now you can open an existing user account using the command "su username".
- Open the Linux terminal and type the following commands to create a user.

```
$ su
password:
# useradd hadoop
# passwd hadoop
New passwd:
Retype new passwd
```

SSH Setup and Key Generation

- SSH setup is required to perform different operations on a cluster such as starting, stopping, and distributed daemon shell operations. To authenticate different users of Hadoop, it is required to provide public/private key pair for a Hadoop user and share it with different users.
- The following commands are used to generate a key value pair using SSH, copy the public keys form id_rsa.pub to authorized_keys, and provide owner, read and write permissions to authorized_keys file respectively.

```
$ ssh-keygen -t rsa
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys
Verifying ssh
ssh localhost
```

Installing Java

- Java is the main prerequisite for Hadoop and HBase. First of all, you should verify the existence of Java in your system using "java -version". The syntax of Java version command is given below.

```
$ java -version
```

It should produce the following output.

```
java version "1.7.0_71"
Java(TM) SE Runtime Environment (build 1.7.0_71-b13)
Java HotSpot(TM) Client VM (build 25.0-b02, mixed mode)
```

- If you don't have Java installed in your system, then follow the steps given below for installing Java.

Step 1

Download java (JDK <latest version> - X64.tar.gz) by visiting the following link: Oracle

Then jdk-7u71-linux-x64.tar.gz is downloaded onto your system.

Step 2

Generally, you find the downloaded Java file in the Downloads folder. Verify it and extract the jdk-7u71-linux-x64.gz file using the following commands.

```
$ cd Downloads/
$ ls
jdk-7u71-linux-x64.gz
$ tar zxf jdk-7u71-linux-x64.gz
$ ls
jdk1.7.0_71 jdk-7u71-linux-x64.gz
```

Step 3

To make Java available to all the users, you need to move it to the location "/usr/local/". Open root, and type the following commands.

```
$ su
password:
# mv jdk1.7.0_71 /usr/local/
# exit
```

Step 4

For setting up PATH and JAVA_HOME variables, add the following commands to ~/.bashrc file.

```
export JAVA_HOME=/usr/local/jdk1.7.0_71
export PATH= $PATH:$JAVA_HOME/bin
```

Now, verify the java -version command from terminal as explained above.

Downloading Hadoop

After installing Java, you need to install Hadoop initially. Verify the existence of Hadoop using "Hadoop version" command as shown below.

```
hadoop version
```

It should produce the following output:

```
Hadoop 2.6.0
Compiled by jenkins on 2014-11-13T21:10Z
Compiled with protoc 2.5.0
From source with checksum
18e43357c8f927c0695f1e9522859d6a
This command was run using
/home/hadoop/hadoop/share/hadoop/common/hadoopc
ommon-2.6.0.jar
```

- If your system is unable to locate Hadoop, then download Hadoop and have it installed on your system. Follow the commands given below to do so.
- Download and extract hadoop-2.6.0 from apache software foundation using the following commands.

```
$ su
password:
# cd /usr/local
# wget
http://mirrors.advancedhosters.com/apache/hadoop/co
mmon/hadoop-2.6.0/hadoop-2.6.0-src.tar.gz
# tar xzf hadoop-2.6.0-src.tar.gz
# mv hadoop-2.6.0/* hadoop/
# exit
```

Installing Hadoop

- Install Hadoop in any of the required modes. Here, we are demonstrating HBase functionalities in pseudo-distributed mode, therefore install Hadoop in pseudo-distributed mode.

Follow the steps given below to install Hadoop 2.4.1 on your system.

Step 1: Setting up Hadoop

You can set Hadoop environment variables by appending the following commands to ~/.bashrc file.

```
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export
HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_
HOME/lib/native
export
PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HO
ME/bin
export HADOOP_INSTALL=$HADOOP_HOME
```

- Now, apply all changes into the currently running system.

```
$ source ~/.bashrc
```

Step 2: Hadoop Configuration

You can find all the Hadoop configuration files at the location "$HADOOP_HOME/etc/hadoop". It is required to make changes in those configuration files according to your Hadoop infrastructure.

```
$ cd $HADOOP_HOME/etc/hadoop
```

- In order to develop Hadoop programs in Java, you need to reset the Java environment variables in hadoop-env.sh file by replacing JAVA_HOME value with the location of Java in your system.

```
export JAVA_HOME=/usr/local/jdk1.7.0_71
```

- Given below are the list of files which you have to edit to configure Hadoop.

core-site.xml

- The core-site.xml file contains information such as the port number used for Hadoop instance, memory allocated for file system, memory limit for storing data, and the size of Read/Write buffers.
- Open core-site.xml and add the following property in between the <configuration>, </configuration> tags:

```
<configuration>
 <property>
  <name>fs.default.name</name>
```

```
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

hdfs-site.xml

- The hdfs-site.xml file contains information such as the value of replication data, namenode path, and datanode paths of your local file systems. It means the place where you want to store the Hadoop infrastructure.
- Let us assume the following data:

```
dfs.replication (data replication value) = 1
```

- (In the below given path /hadoop/ is the user name.
- hadoopinfra/hdfs/namenode is the directory created by hdfs file system.)

```
namenode path =
//home/hadoop/hadoopinfra/hdfs/namenode
(hadoopinfra/hdfs/datanode is the directory created by hdfs file system.)
datanode path =
//home/hadoop/hadoopinfra/hdfs/datanode
```

- Open this file and add the following properties in between the <configuration>, </configuration> tags in this file.

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>

  <property>
    <name>dfs.name.dir</name>
<value>file:///home/hadoop/hadoopinfra/hdfs/namenode
</value>
  </property>

  <property>
    <name>dfs.data.dir</name>

<value>file:///home/hadoop/hadoopinfra/hdfs/datanode
</value>
  </property>
</configuration>
```

Note: In the hdfs-site.xml file, all the property values are user defined. You can make changes according to your Hadoop infrastructure.

mapred-site.xml

- This file is used to configure yarn into Hadoop. Open mapred-site.xml file and add the following property in between the <configuration>, </configuration> tags in this file.

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

mapred-site.xml

- This file is used to specify which MapReduce framework we are using. By default, Hadoop contains a template of mapred-site.xml. First of all, it is required to copy the file from mapred-site.xml.template to mapred-site.xml file using the following command.

```
$ cp mapred-site.xml.template mapred-site.xml
```

- Open mapred-site.xml file and add the following properties in between the <configuration>, </configuration> tags in this file.

```
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

Verifying Hadoop Installation

- The following steps are used to verify the Hadoop installation.

Step 1: Name Node Setup

Set up the namenode using the command "hdfs namenode -format" as follows:

```
$ cd ~
$ hdfs namenode -format
```

The expected result is as follows:

```
10/24/14 21:30:55 INFO namenode.NameNode:
STARTUP_MSG:
/************************************************
************
```

```
STARTUP_MSG: Starting NameNode
STARTUP_MSG: host = localhost/192.168.1.11
STARTUP_MSG: args = [-format]
STARTUP_MSG: version = 2.4.1
...
...
10/24/14 21:30:56 INFO common.Storage: Storage
directory
/home/hadoop/hadoopinfra/hdfs/namenode has been
successfully formatted.
10/24/14 21:30:56 INFO
namenode.NNStorageRetentionManager: Going to
retain
1 images with txid >= 0
10/24/14 21:30:56 INFO util.ExitUtil: Exiting with
status 0
10/24/14 21:30:56 INFO namenode.NameNode:
SHUTDOWN_MSG:
/************************************************
************
SHUTDOWN_MSG: Shutting down NameNode at
localhost/192.168.1.11
************************************************
***********/
```

Step 2: Verifying Hadoop dfs

The following command is used to start dfs. This command starts your Hadoop file system.

```
$ start-dfs.sh
```

The expected output is as follows:

```
10/24/14 21:37:56
Starting namenodes on [localhost]
localhost: starting namenode, logging to
/home/hadoop/hadoop-
2.4.1/logs/hadoop-hadoop-namenode-localhost.out
localhost: starting datanode, logging to
/home/hadoop/hadoop-
2.4.1/logs/hadoop-hadoop-datanode-localhost.out
Starting secondary namenodes [0.0.0.0]
```

Step 3: Verifying Yarn Script

The following command is used to start yarn script. Executing this command will start your yarn demons.

```
$ start-yarn.sh
```

The expected output is as follows:

```
starting yarn daemons
starting resource manager, logging to
/home/hadoop/hadoop-2.4.1/logs/yarn-
hadoop-resourcemanager-localhost.out
localhost: starting node manager, logging to
/home/hadoop/hadoop-
2.4.1/logs/yarn-hadoop-nodemanager-localhost.out
```

Step 4: Accessing Hadoop on Browser

The default port number to access hadoop is 50070. Use the following URL to get Hadoop services on your browser.

```
http://localhost:50070/
```

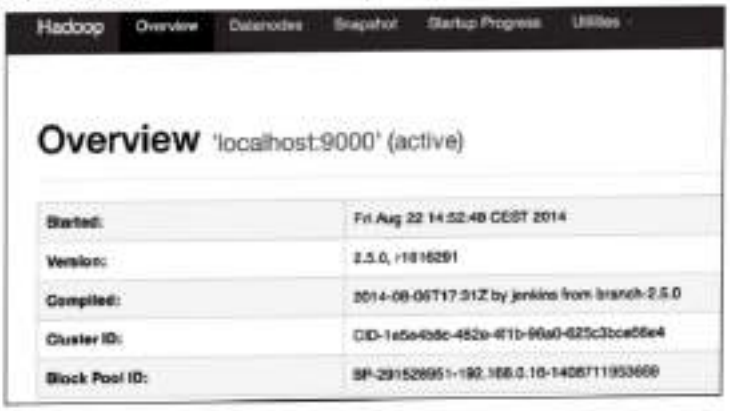


Fig. 4.8

Step 5: Verify All Applications for Cluster

The default port number to access all application of cluster is 8088. Use the following URL to visit this service.

```
http://localhost:8088/
```

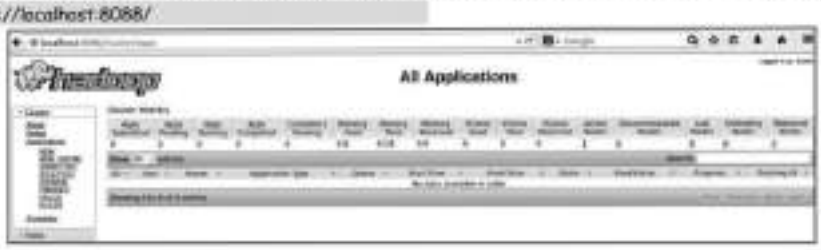


Fig. 4.9

Downloading Mahout

- Mahout is available in the website Mahout. Download Mahout from the link provided in the website. Here is the screenshot of the website.

Fig. 4.10

Step 1

Download Apache mahout from the link http://mirror.nexcess.net/apache/mahout/ using the following command.

```
[Hadoop@localhost ~]$ wget
http://mirror.nexcess.net/apache/mahout/0.9/mahout-distribution-0.9.tar.gz
```

Then mahout-distribution-0.9.tar.gz will be downloaded in your system.

Step 2

Browse through the folder where mahout-distribution-0.9.tar.gz is stored and extract the downloaded jar file as shown below.

```
[Hadoop@localhost ~]$ tar zxvf mahout-distribution-0.9.tar.gz
```

Maven Repository

Given below is the pom.xml to build Apache Mahout using Eclipse.

```
<dependency>
  <groupId>org.apache.mahout</groupId>
  <artifactId>mahout-core</artifactId>
  <version>0.9</version>
</dependency>

<dependency>
  <groupId>org.apache.mahout</groupId>
  <artifactId>mahout-math</artifactId>
  <version>${mahout.version}</version>
</dependency>

<dependency>
  <groupId>org.apache.mahout</groupId>
  <artifactId>mahout-integration</artifactId>
  <version>${mahout.version}</version>
</dependency>
```

library of tutorials contains topics on various ects. When we receive a new tutorial, it gets essed by a clustering engine that decides, based content, where it should be grouped.

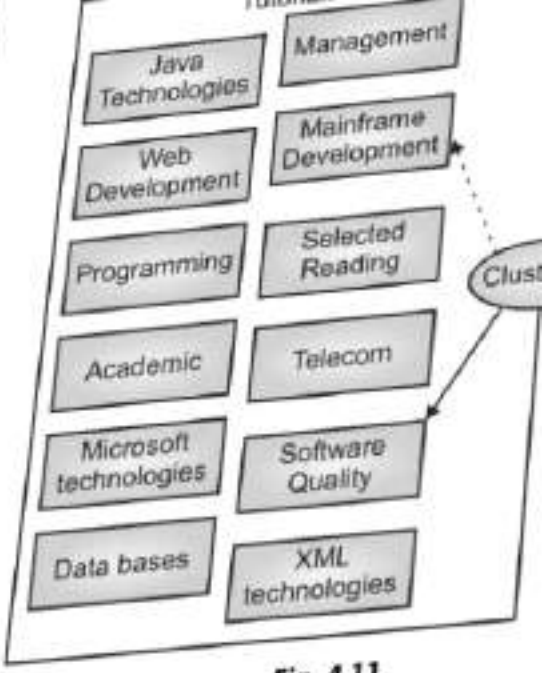
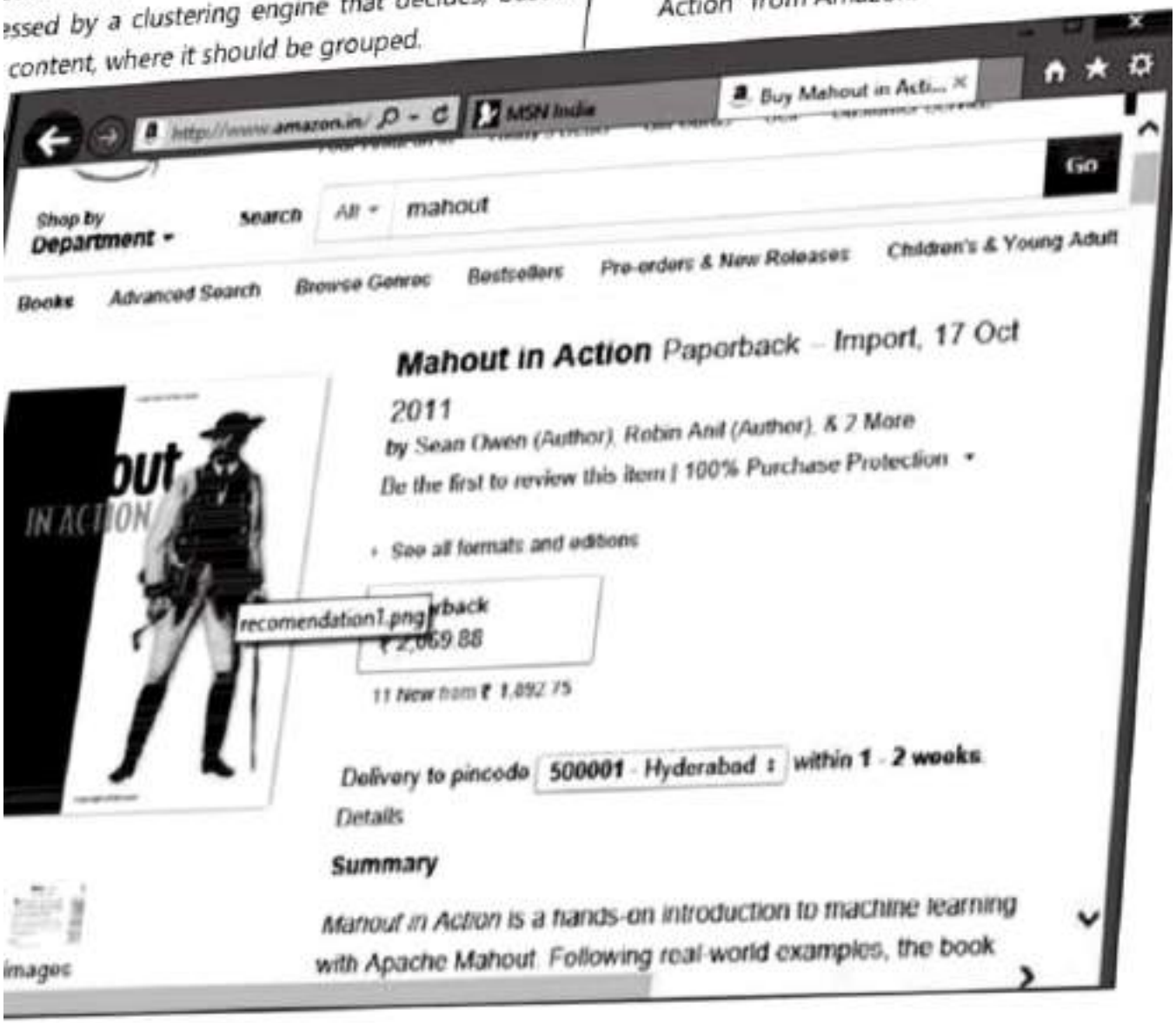


Fig. 4.11

Recommendation

- Ever wondered how Amazon comes up recommended items to draw your att particular product that you might be intere
- Suppose you want to purchase the book Action" from Amazon:



- Along with the selected product, Amazon also displays a list of related recommended items, as shown in Fig. 4.13.

Fig. 4.13

- Such recommendation lists are produced with the help of recommender engines. Mahout provides recommender engines of several types such as:

  user-based recommenders,

  item-based recommenders, and

  several other algorithms.

Mahout Recommender Engine

- Mahout has a non-distributed, non-Hadoop-based recommender engine. You should pass a text document having user preferences for items. And the output of this engine would be the estimated preferences of a particular user for other items.

Example

- Consider a website that sells consumer goods such as mobiles, gadgets, and their accessories. If we want to implement the features of Mahout in such a site, then we can build a recommender engine. This engine analyzes past purchase data of the users and recommends new products based on that.
- The components provided by Mahout to build a recommender engine are as follows:

  DataModel

  UserSimilarity

  ItemSimilarity

  UserNeighborhood

  Recommender

- From the data store, the data model is prepared and is passed as an input to the recommender engine. The Recommender engine generates the recommendations for a particular user. Given below is the architecture of recommender engine.

Architecture of Recommender Engine

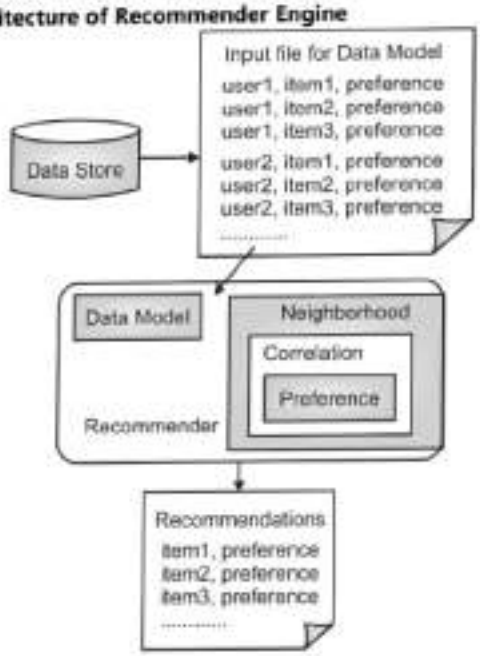


Fig. 4.14

Building a Recommender using Mahout

Here are the steps to develop a simple recommender:

Step 1: Create DataModel Object

The constructor of PearsonCorrelationSimilarity class requires a data model object, which holds a file that contains the Users, Items, and Preferences details of a product. Here is the sample data model file:

```
1,00,1.0
1,01,2.0
1,02,5.0
1,03,5.0
1,04,5.0

2,00,1.0
2,01,2.0
2,05,5.0
2,06,4.5
2,02,5.0

3,01,2.5
3,02,5.0
3,03,4.0
3,04,3.0
4,00,5.0
4,01,5.0
4,02,5.0
4,03,0.0
```

- The DataModel object requires the file object, which contains the path of the input file. Create the DataModel object as shown below.

```
DataModel datamodel = new FileDataModel(new
File("input file"));
```

Step 2: Create UserSimilarity Object

- Create UserSimilarity object using PearsonCorrelationSimilarity class as shown below:

```
UserSimilarity similarity = new
PearsonCorrelationSimilarity(datamodel);
```

Step 3: Create UserNeighborhood object

- This object computes a "neighborhood" of users like a given user. There are two types of neighborhoods:
  (i) NearestNUserNeighborhood - This class computes a neighborhood consisting of the nearest n users to a given user. "Nearest" is defined by the given UserSimilarity.
  (ii) ThresholdUserNeighborhood - This class computes a neighborhood consisting of all the users whose similarity to the given user meets or exceeds a certain threshold. Similarity is defined by the given UserSimilarity.
- Here we are using ThresholdUserNeighborhood and set the limit of preference to 3.0.

```
UserNeighborhood neighborhood = new
ThresholdUserNeighborhood(3.0, similarity, model);
```

Step 4: Create Recommender Object

Create UserbasedRecomender object. Pass all the above created objects to its constructor as shown below.

```
UserBasedRecommender recommender = new
GenericUserBasedRecommender(model,
neighborhood, similarity);
```

Step 5: Recommend Items to a User

- Recommend products to a user using the recommend() method of Recommender interface. This method requires two parameters. The first represents the user id of the user to whom we need to send the recommendations, and the second represents the number of recommendations to be sent. Here is the usage of recommender() method:

```
List<RecommendedItem> recommendations =
recommender.recommend(2, 3);
for (RecommendedItem recommendation
    :recommendations)
{
 System.out.println(recommendation);
}
```

Example Program

- Given below is an example program to set recommendation. Prepare the recommendations for the user with user id 2.

```
import java.io.File;
import java.util.List;
import
org.apache.mahout.cf.taste.impl.model.file.FileDataMod
el;
import
org.apache.mahout.cf.taste.impl.neighborhood.Threshol
dUserNeighborhood;
import
org.apache.mahout.cf.taste.impl.recommender.GenericU
serBasedRecommender;
import
org.apache.mahout.cf.taste.impl.similarity.PearsonCorrel
ationSimilarity;

import org.apache.mahout.cf.taste.model.DataModel;
import
org.apache.mahout.cf.taste.neighborhood.UserNeighbor
hood;

import
org.apache.mahout.cf.taste.recommender.Recommended
Item;
import
org.apache.mahout.cf.taste.recommender.UserBasedRec
ommender;

import
org.apache.mahout.cf.taste.similarity.UserSimilarity;
public class Recommender {
  public static void main(String args[]){
    try{
                                  //Creating data model
      DataModel datamodel = new FileDataModel(new
                                            //data
       File("data"));

        //Creating UserSimilarity object.
      UserSimilarity usersimilarity = new
PearsonCorrelationSimilarity(datamodel);

      //Creating UserNeighbourHHood object.
  UserNeighborhood userneighborhood = new
ThresholdUserNeighborhood(3.0, usersimilarity,
datamodel);

      //Create UserRecomender
      UserBasedRecommender recommender = new
GenericUserBasedRecommender(datamodel,
userneighborhood, usersimilarity);

      List<RecommendedItem> recommendations =
recommender.recommend(2, 3);

      for (RecommendedItem recommendation :
recommendations) {
         System.out.println(recommendation);
      }

    }
catch(Exception e){}

  }
}
```

Compile the program using the following commands:

```
javac Recommender.java
java Recommender
```

It should produce the following output:

```
RecommendedItem [item:3, value:4.5]
RecommendedItem [item:4, value:4.0]
```

4.4 BIG DATA MACHINE LEARNING ALGORITHMS IN MAHOUT

4.4.1 Clustering

- Clustering is the procedure to organize elements or items of a given collection into groups based on the similarity between the items. For example, the applications related to online news publishing group their news articles using clustering.

Applications of Clustering

- Clustering is broadly used in many applications such as market research, pattern recognition, data analysis, and image processing.
- Clustering can help marketers discover distinct groups in their customer basis. And they can characterize their customer groups based on purchasing patterns.
- In the field of biology, it can be used to derive plant and animal taxonomies, categorize genes with simila

functionality and gain insight into structures inherent in populations.

- Clustering helps in identification of areas of similar land use in an earth observation database.
- Clustering also helps in classifying documents on the web for information discovery.
- Clustering is used in outlier detection applications such as detection of credit card fraud.
- As a data mining function, Cluster Analysis serves as a tool to gain insight into the distribution of data to observe characteristics of each cluster.

Using Mahout, we can cluster a given set of data. The steps required are as follows:

- **Algorithm** : You need to select a suitable clustering algorithm to group the elements of a cluster.
- **Similarity and Dissimilarity :** You need to have a rule in place to verify the similarity between the newly encountered elements and the elements in the groups.
- **Stopping Condition :** A stopping condition is required to define the point where no clustering is required.

Procedure of Clustering

To cluster the given data you need to :

- Start the Hadoop server. Create required directories for storing files in Hadoop File System. (Create directories for input file, sequence file, and clustered output in case of canopy).
- Copy the input file to the Hadoop File system from Unix file system.
- Prepare the sequence file from the input data.
- Run any of the available clustering algorithms.
- Get the clustered data.

Starting Hadoop

- Mahout works with Hadoop, hence make sure that the Hadoop server is up and running.

```
$ cd HADOOP_HOME/bin
$ start-all.sh
```

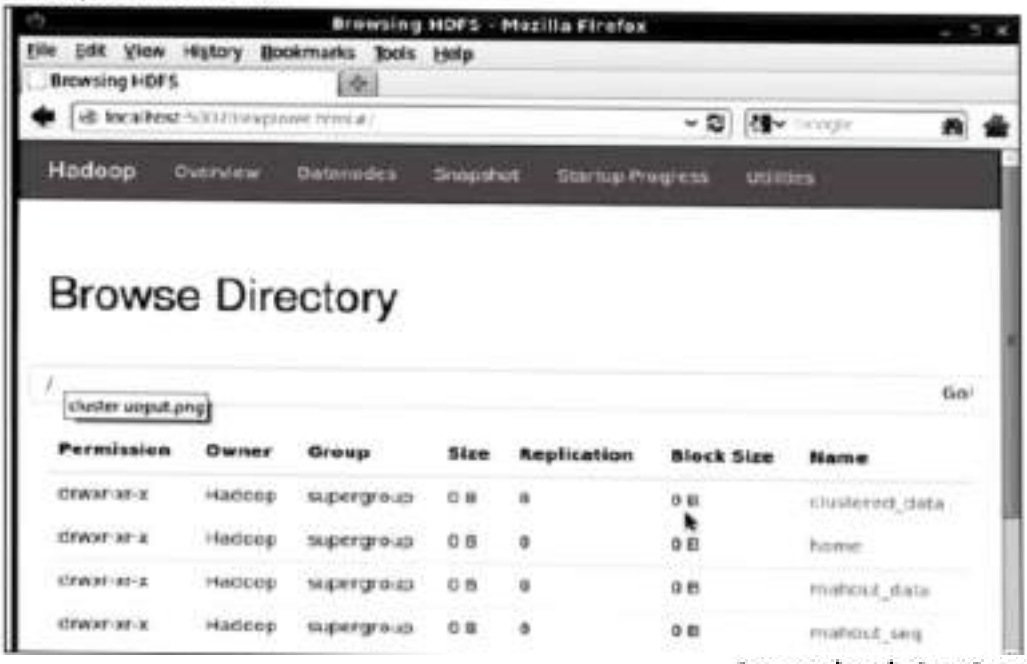
Preparing Input File Directories

- Create directories in the Hadoop file system to store the input file, sequence files, and clustered data using the following command:

```
$ hadoop fs -p mkdir /mahout_data
$ hadoop fs -p mkdir /clustered_data
$ hadoop fs -p mkdir /mahout_seq
```

- You can verify whether the directory is created using the hadoop web interface in the following URL - **http://localhost:50070/**

It gives you the output as shown below:



Copying Input File to HDFS

- Now, copy the input data file from the Linux file system to mahout_data directory in the Hadoop File System as shown below. Assume your input file is mydata.txt and it is in the /home/Hadoop/data/ directory.

```
$ hadoop fs -put /home/Hadoop/data/mydata.txt
/mahout_data/
```

Preparing the Sequence File

- Mahout provides you a utility to convert the given input file in to a sequence file format. This utility requires two parameters.
  1. The input file directory where the original data resides.
  2. The output file directory where the clustered data is to be stored.
- Given below is the help prompt of mahout **seqdirectory** utility.

Step 1: Browse to the Mahout home directory. You can get help of the utility as shown below:

```
[Hadoop@localhost bin]$ ./mahout seqdirectory --help
```

Job-Specific Options:

--input (-i) input Path to job input directory.

--output (-o) output The directory pathname for output.

--overwrite (-ow) If present, overwrite the output directory

Generate the sequence file using the utility using the following syntax:

```
mahout seqdirectory -i <input file path> -o <output
directory>
```

Example

mahout seqdirectory

```
-i hdfs://localhost:9000/mahout_seq/
-o hdfs://localhost:9000/clustered_data/
```

Clustering Algorithms

Mahout supports two main algorithms for clustering namely:

1. Canopy clustering
2. K-means clustering

1. Canopy Clustering

- Canopy clustering is a simple and fast technique used by Mahout for clustering purpose. The objects will be treated as points in a plain space. This technique is often used as an initial step in other clustering techniques such as k-means clustering. You can run a Canopy job using the following syntax:

```
mahout canopy -i <input vectors directory>
-o <output directory>
-t1 <threshold value 1>
-t2 <threshold value 2>
```

- Canopy job requires an input file directory with the sequence file and an output directory where the clustered data is to be stored.

Example

```
mahout canopy -i
hdfs://localhost:9000/mahout_seq/mydata.seq
-o hdfs://localhost:9000/clustered_data
-t1 20
-t2 30
```

You will get the clustered data generated in the given output directory.

2. K-Means Clustering

- K-means clustering is an important clustering algorithm. The k in k-means clustering algorithm represents the number of clusters the data is to be divided into. For example, the k value specified to this algorithm is selected as 3, the algorithm is going to divide the data into 3 clusters.
- Each object will be represented as vector in space. Initially k points will be chosen by the algorithm randomly and treated as centers, every object closest to each center are clustered. There are several algorithms for the distance measure and the user should choose the required one.

Creating Vector Files

- Unlike Canopy algorithm, the k-means algorithm requires vector files as input, therefore you have to create vector files.
- To generate vector files from sequence file format, Mahout provides the **seq2parse** utility.
- Given below are some of the options of **seq2parse** utility. Create vector files using these options.

```
$MAHOUT_HOME/bin/mahout seq2sparse
--analyzerName (-a) analyzerName
                    The class name of the analyzer
--chunkSize (-chunk) chunkSize
                    The chunkSize in MegaBytes.
```

--output (-o) output	The directory pathname for o/p
--input (-i) input	Path to job input directory.

After creating vectors, proceed with k-means algorithm. The syntax to run k-means job is as follows:

```
mahout kmeans -i <input vectors directory>
-c <input clusters directory>
-o <output working directory>
-dm <Distance Measure technique>
-x <maximum number of iterations>
-k <number of initial clusters>
```

- K-means clustering job requires input vector directory, output clusters directory, distance measure, maximum number of iterations to be carried out, and an integer value representing the number of clusters the input data is to be divided into.

4.4.2 Classification

- Classification is a machine learning technique that uses known data to determine how the new data should be classified into a set of existing categories. For example,
  - iTunes application uses classification to prepare playlists.
  - Mail service providers such as Yahoo! and Gmail use this technique to decide whether a new mail should be classified as a spam. The categorization algorithm trains itself by analyzing user habits of marking certain mails as spams. Based on that, the classifier decides whether a future mail should be deposited in your inbox or in the spams folder.

How Classification Works

- While classifying a given set of data, the classifier system performs the following actions:
  - Initially a new data model is prepared using any of the learning algorithms.
  - Then the prepared data model is tested.
  - Thereafter, this data model is used to evaluate the new data and to determine its class.

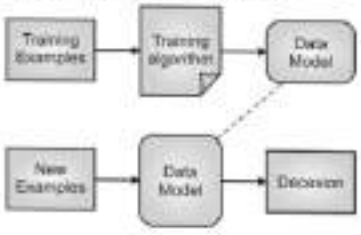


Fig. 4.16

Applications of Classification

- **Credit Card Fraud Detection** : The Classification mechanism is used to predict credit card frauds. Using historical information of previous frauds, the classifier can predict which future transactions may turn into frauds.
- **Spam E-Mails** : Depending on the characteristics of previous spam mails, the classifier determines whether a newly encountered e-mail should be sent to the spam folder.

Naive Bayes Classifier

- Mahout uses the Naive Bayes classifier algorithm. It uses two implementations:
  1. Distributed Naive Bayes classification
  2. Complementary Naive Bayes classification
- Naive Bayes is a simple technique for constructing classifiers. It is not a single algorithm for training such classifiers, but a family of algorithms. A Bayes classifier constructs models to classify problem instances. These classifications are made using the available data.
- An advantage of naive Bayes is that it only requires a small amount of training data to estimate the parameters necessary for classification.
- For some types of probability models, naive Bayes classifiers can be trained very efficiently in a supervised learning setting.
- Despite its oversimplified assumptions, naive Bayes classifiers have worked quite well in many complex real-world situations.

Procedure of Classification

The following steps are to be followed to implement Classification:

1. Generate example data
2. Create sequence files from data
3. Convert sequence files to vectors
4. Train the vectors
5. Test the vectors

Step 1: Generate Example Data

Generate or download the data to be classified. For example, you can get the **20 newsgroups** example data from the following link: http://people.csail.mit.edu/jrennie/20Newsgroups/20news-bydate.tar.gz

Create a directory for storing input data. Download the example as shown below.

```
$ mkdir classification_example
$ cd classification_example
$tar xzvf 20news-bydate.tar.gz
Wget
http://people.csail.mit.edu/jrennie/20Newsgroups/20news-bydate.tar.gz
```

Step 2: Create Sequence Files

Create sequence file from the example using **seqdirectory** utility. The syntax to generate sequence is given below:

```
mahout seqdirectory -i <input file path> -o <output directory>
```

Step 3: Convert Sequence Files to Vectors

Create vector files from sequence files using **seq2parse** utility. The options of **seq2parse** utility are given below:

```
$MAHOUT_HOME/bin/mahout seq2sparse
--analyzerName (-a) analyzerName
                    The class name of the analyzer
--chunkSize (-chunk) chunkSize
                    The chunkSize in MegaBytes
--output (-o) output    The directory pathname for o/p
--input (-i) input      Path to job input directory.
```

Step 4: Train the Vectors

Train the generated vectors using the **trainnb** utility. The options to use **trainnb** utility are given below:

```
mahout trainnb
-i ${PATH_TO_TFIDF_VECTORS}
-el
-o ${PATH_TO_MODEL}/model
-li ${PATH_TO_MODEL}/labelindex
-ow
-c
```

Step 5: Test the Vectors

Test the vectors using **testnb** utility. The options to use **testnb** utility are given below:

```
mahout testnb
-i ${PATH_TO_TFIDF_TEST_VECTORS}
-m ${PATH_TO_MODEL}/model
-l ${PATH_TO_MODEL}/labelindex
-ow
-o ${PATH_TO_OUTPUT}
-c
-seq
```

4.5 MACHINE LEARNING WITH SPARK

- Apache Spark is known as a fast, easy-to-use and general engine for big data processing that has built-in modules for streaming, SQL, Machine Learning (ML) and graph processing. This technology is an in-demand skill for data engineers, but also data scientists can benefit from learning Spark when doing Exploratory Data Analysis (EDA), feature extraction and, of course, ML.
- In this tutorial, you'll interface Spark with Python through PySpark, the Spark Python API that exposes the Spark programming model to Python. More concretely, you'll focus on
  - Installing PySpark locally on your personal computer and setting it up so that you can work with the interactive Spark shell to do some quick, interactive analyses on your data. You'll see how to do this with pip, Homebrew and via the Spark download page.
  - Learning how to work with the basics of Spark: you'll see how you can create RDDs and perform basic operations on them.
  - Getting started with PySpark in Jupyter Notebook and loading in a real-life data set.
  - Exploring and preprocessing the data that you loaded in at the first step the help of DataFrames, which demands that you make use of Spark SQL, which allows you to query structured data inside Spark programs.
  - Creating a Linear Regression model with Spark ML to feed the data to it, after which you'll be able to make predictions. And, lastly,
  - Evaluating the machine learning model that you made.

Installing Apache Spark

- Installing Spark and getting it to work can be a challenge. In this section, you'll cover some steps that will show you how to get it installed on your pc.
- First thing that you want to do is checking whether you meet the prerequisites. Spark is written in Scala

Programming Language and runs on Java Virtual Machine (JVM) environment. That's why you need to check if you have a Java Development Kit (JDK) installed. You do this because the JDK will provide you with one or more implementations of the JVM. Preferably, you want to pick the latest one, which, at the time of writing is the JDK8.

- Next, you're reading to download Spark!

Downloading pyspark with pip

- Then, you can download and install PySpark it with the help of pip. This is fairly easy and much like installing any other package. You just run the usual command and the heavy lifting gets done for you:

```
$ pip install pyspark
```

- Alternatively, you can also go to the Spark download page. Keep the default options in the first three steps and you'll find a downloadable link in step 4. Click on that link to download it. For this tutorial, you'll download the 2.2.0 Spark Release and the "Pre-built for Apache Hadoop 2.7 and later" package type.

Note that the download can take some time to finish!

Downloading Spark with Homebrew

- You can also install Spark with the Homebrew, a free and open-source package manager. This is especially handy if you're working with macOS.
- Simply run the following commands to search for Spark, to get more information and to finally install it on your personal computer:

```
# Search for spark
$ brew search spark

# Get more information on apache-spark
$ brew info apache-spark

# Install apache-spark
$ brew install apache-spark
```

Download and Set Up Spark

- Next, make sure that you untar the directory that appears in your Downloads folder. This can happen automatically for you, by double clicking the spark-2.2.0-bin-hadoop2.7.tgz archive or by opening up your Terminal and running the following command:

```
$ tar xvf spark-2.2.0-bin-hadoop2.7.tgz
```

- Next, move the untarred folder to /usr/local/spark by running the following line:

```
$ mv spark-2.1.0-bin-hadoop2.7 /usr/local/spark
```

Note that if you get an error that says that the permission is denied to move this folder to the new location, you should add sudo in front of this command. The line above will then become:

```
$ sudo mv spark-2.1.0-bin-hadoop2.7 /usr/local/spark
```

You'll be prompted to give your password, which is usually the one that you also use to unlock your pc when you start it up :)

Now that you're all set to go, open the README file in the file path /usr/local/spark. You can do this by executing:

```
$ cd /usr/local/spark
```

- This will brings you to the folder that you need to be. Then, you can start inspecting the folder and reading the README file that is included in it.
- First, use $ ls to get a list of the files and folders that are in this spark folder. You'll see that there's a README.md file in there. You can open it by executing one of the following commands:

```
# Open and edit the file
$ nano README.md

# Just read the file
$ cat README.md
```

- **Tip** use the tab button on your keyboard to auto complete as you're typing the file name :) This will save you some time.
- You'll see that this README provides you with some general information about Spark, online documentation, building Spark, the Interactive Scala and Python shells, example programs and much more.
- The thing that could interest you most here is the section on how to build Spark but note that this will only be particularly relevant if you haven't downloaded a pre-built version. For this tutorial, however, you downloaded a pre-built version. You can press CTRL + X to exit the README, which brings you back to the spark folder.
- In case you selected a version that hasn't been built yet, make sure you run the command that is listed in the README file. At the time of writing, this is the following:

```
$ build/mvn -DskipTests clean package run
```

Note that this command can take a while to run.

PySpark Basics: RDDs

- Now that you've successfully installed Spark and PySpark, let's first start off by exploring the interactive Spark Shell and by nailing down some of the basics that you will need when you want to get started. In the rest of this tutorial, however, you'll work with PySpark in a Jupyter notebook.

Spark Applications Versus Spark Shell

- The interactive shell is an example of a Read-Evaluate)-Print-Loop (REPL) environment: That means that whatever you type in is read, evaluated and printed out to you so that you can continue your analysis. This might remind you of IPython, which is a powerful interactive Python shell that you might know from working with Jupyter. If you want to know more, consider reading DataCamp's IPython or Jupyter blog post.
- This means that you can use the shell, which is available for Python as well as Scala, for all interactive work that you need to do.
- Besides this shell, you can also write and deploy Spark applications. In contrast to writing Spark applications, the SparkSession has already been created for you so that you can just start working and not waste valuable time on creating one.
- Now you might wonder: what is the SparkSession?
- Well, it's the main entry point for Spark functionality: it represents the connection to a Spark cluster and you can use it to create RDDs and to broadcast variables on that cluster. When you're working with Spark, everything starts and ends with this SparkSession.

Note that before Spark 2.0.0, the three main connection objects were SparkContext, SqlContext and HiveContext.

- You'll see more on this later on. For now, let's just focus on the shell.

The Python Spark Shell

From within the spark folder located at /usr/local/spark, you can run

```
$ ./bin/pyspark
```

- At first, you'll see some text appearing. And then, you'll see "Spark" appearing, just like this:

```
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016,
12:39:47)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
Using Spark's default log4j profile:
org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
For SparkR, use setLogLevel(newLevel).
17/07/26 11:41:26 WARN NativeCodeLoader: Unable to
load native-hadoop library for your platform... using
builtin-java classes where applicable
17/07/26 11:41:47 WARN ObjectStore: Failed to get
database global_temp, returning
NoSuchObjectException
Welcome to

Using Python version 2.7.13 (v2.7.13:a06454b1afa1, Dec
17 2016 12:39:47)
SparkSession available as 'spark'.
>>>
```

- When you see this, you know that you're ready to start experimenting within the interactive shell!

Tip: If you prefer using the IPython shell instead of the Spark shell, you can do this by setting the following environment variable:

```
export
PYSPARK_DRIVER_PYTHON="/usr/local/ipython/bin/i
python"
```

Creating RDDs

- Now, let's start small and make an RDD, which is the most basic building block of Spark. An RDD simply represents data but it's not one object, a collection of records, a result set or a data set. That is because it's intended for data that resides on multiple computers: a single RDD could be spread over thousands of Java Virtual Machines (JVMs), because Spark automatically partitions the data under the hood to get this parallelism. Of course, you can adjust the parallelism to get more partitions. That's why an RDD is actually a collection of partitions.

- You can easily create a simple RDD by using the parallelize() function and by simply passing some data (an iterable, like a list, or a collection) to it:

```
>>> rdd1 =
spark.sparkContext.parallelize([('a',7),('a',2),('b',2)])
>>> rdd2 =
spark.sparkContext.parallelize([("a",["x","y","z"]),
("b",["p", "r"])])
>>> rdd3 = spark.sparkContext.parallelize(range(100))
```

Note that the SparkSession object has the SparkContext object, which you can access with spark.sparkContext. For backwards compatibility reasons, it's also still possible to call the SparkContext with sc, as in rdd1 = sc.parallelize([('a',7),('a',2),('b',2)]).

RDD Operations

- Now that you have created the RDDs, you can use the distributed data in rdd1 and rdd2 to operate on in parallel. You have two types of operations: transformations and actions.
- Now, to intuitively get the difference between these two, consider some of the most common transformations are map(), filter(), flatMap(), sample(), randomSplit(), coalesce() and repartition() and some of the most common actions are reduce(), collect(), first(), take(), count(), saveAsHadoopFile().
- Transformations are lazy operations on a RDD that create one or many new RDDs, while actions produce non-RDD values: they return a result set, a number, a file, ...
- You can, for example, aggregate all the elements of rdd1 using the following, simple lambda function and return the results to the driver program:

```
>>> rdd1.reduce(lambda a,b: a+b)
```

- Executing this line of code will give you the following result: ('a', 7, 'a', 2, 'b', 2). Another example of a transformation is flat MapValues(), which you run on key-value pair RDDs, such as rdd2. In this case, you pass each value in the key-value pair RDD rdd2 through a flat Map function without changing the keys, which is the lambda function defined below and you perform an action after that by collecting hte results with collect().

```
>>> rdd2.flatMapValues(lambda x: x).collect()
[('a', 'x'), ('a', 'y'), ('a', 'z'), ('b', 'p'), ('b', 'r')]
```

The Data

- Now that you have covered some basics with the interactive shell, it's time to get started with some real data. For this tutorial, you'll make use of the California Housing data set. Note, of course, that this is actually 'small' data and that using Spark in this context might be overkill. This tutorial is for educational purposes only and is meant to give you an idea of how you can use PySpark to build a machine learning model.

Loading And Exploring Your Data

- Even though you know a bit more about your data, you should take the time to go ahead and explore it more thoroughly. Before you do this, however, you will set up your Jupyter Notebook with Spark and you'll take some first steps to defining the SparkContext.

PySpark in Jupyter Notebook

- For this part of the tutorial, you won't use the ishell but you'll build your own application. You'll do this in a Jupyter Notebook. You already have all the things that you need installed, so you don't need to do much to get PySpark to work in Jupyter.
- You can just launch the notebook application the same way like you always do, by running $ jupyter notebook. Then, you make a new notebook and you simply import the findspark library and use the init() function. In this case, you're going to supply the path /usr/local/spark to init() because you're certain that this is the path where you installed Spark.

```
# Import findspark
import findspark

# Initialize and provide path
findspark.init("/usr/local/spark")

# Or use this alternative
#findspark.init()
```

Tip: If you have no idea whether your path is set correctly or where you have installed Spark on your pc, you can always use findspark.find() to automatically detect the location of where Spark is installed.

- If you're looking for alternative ways to work with Spark in Jupyter, consult our Apache Spark in Python: Beginner's Guide.

- Now that you have got all of that settled, you can finally start by creating your first Spark program!

Creating Your First Spark Program

- What you first want to be doing is importing the SparkContext from the pyspark package and initializing it. **Remember** that you didn't have to do this before because the interactive Spark shell automatically created and *initialized* it for you! Here, you'll need to do a little bit more work yourself ;)
- Import the SparkSession module from pyspark.sql and build a SparkSession with the builder() method. Afterwards, you can set the master URL to connect to, the application name, add some additional configuration like the executor memory and then lastly, use getOrCreate() to either get the current Spark session or to create one if there is none running.

```
# Import SparkSession
from pyspark.sql import SparkSession

# Build the SparkSession
spark = SparkSession.builder \
  .master("local") \
  .appName("Linear Regression Model") \
  .config("spark.executor.memory", "1gb") \
  .getOrCreate()

sc = spark.sparkContext
```

Note that if you get an error where there's a FileNotFoundError similar to this one: "No such file or directory: '/Users/YourName/Downloads/spark-2.1.0-bin-hadoop2.7//bin/spark-submit'", you know that you have to reset your Spark PATH. Go to your home directory by executing $ cd and then edit the .bash_profile file by running $ nano .bash_profile.

Add something like the following to the bottom of the file:

```
export SPARK_HOME="/usr/local/spark/"
```

- Use CTRL + X to exit the file but make sure to save your adjustments by also entering Y to confirm the changes. Next, don't forget to set the changes in motion by running source .bash_profile.

Tip: you can also set additional environment variables if you want. You probably don't need them, but it's definitely good to know that you can set them if desired. Consider the following examples:

```
# Set a fixed value for the hash seed secret
export PYTHONHASHSEED=0

# Set an alternate Python executable
export
PYSPARK_PYTHON=/usr/local/ipython/bin/ipython

# Augment the default search path for shared
libraries
export
LD_LIBRARY_PATH=/usr/local/ipython/bin/ipython

# Augment the default search path for private
libraries
export
PYTHONPATH=$SPARK_HOME/python/lib/py4j-*-
src.zip:$PYTHONPATH:$SPARK_HOME/python/
```

Note also that now you have initialized a default SparkSession. However, in most cases, you'll want to configure this further. You'll see that this will be really needed when you start working with big data. If you want to know more about it, check this page.

Loading in Your Data

- This tutorial makes use of the California Housing data set. It appeared in a 1997 paper titled Sparse Spatial Autoregressions, written by Pace, R. Kelley and Ronald Barry and published in the Statistics and Probability Letters journal. The researchers built this data set by using the 1990 California census data.
- The data contains one row per census block group. A block group is the smallest geographical unit for which the U.S. Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). In this sample a block group on average includes 1425.5 individuals living in a geographically compact area. You'll gather this information from this web page or by reading the paper which was mentioned above and which you can find here.

- These spatial data contain 20,640 observations on housing prices with nine economic variables:
  1. **Longitude** refers to the angular distance of a geographic place north or south of the earth's equator for each block group;
  2. **Latitude** refers to the angular distance of a geographic place east or west of the earth's equator for each block group;
  3. **Housing Median Age** is the median age of the people that belong to a block group.

     Note that the median is the value that lies at the midpoint of a frequency distribution of observed values;
  4. **Total rooms** is the total number of rooms in the houses per block group;
  5. **Total bedrooms** is the total number of bedrooms in the houses per block group;
  6. **Population** is the number of inhabitants of a block group;
  7. **Households** refers to units of houses and their occupants per block group;
  8. **Median Income** is used to register the median income of people that belong to a block group;
  9. **Median House Value** is the dependent variable and refers to the median house value per block group.
- What's more, you also learn that all the block groups have zero entries for the independent and dependent variables have been excluded from the data.
- The Median house value is the dependent variable and will be assigned the role of the target variable in your ML model.
- You can download the data here. Look for the houses.zip folder, download and untar it so that you can access the data folders.
- Next, you'll use the textFile() method to read in the data from the folder that you downloaded it to RDDs. This method takes an URI for the file, which is in this case the local path of your machine, and reads it as a collection of lines. For all convenience, you'll not only read in the .data file, but also the .domain file that contains the header. This will allow you to double check the order of the variables.

```
# Load in the data
rdd =
sc.textFile('/Users/yourName/Downloads/CaliforniaHo
using/cal_housing.data')

# Load in the header
header =
sc.textFile('/Users/yourName/Downloads/CaliforniaHo
using/cal_housing.domain')
```

Data Exploration

- You already gathered a lot of information by just looking at the web page where you found the data set, but it's always better to get hands-on and inspect your data with the help of Spark with Python, in this case.
- Important to understand here is that, because Spark's execution is "lazy" execution, nothing has been executed yet. Your data hasn't been actually read in. The rdd and header variables are actually just concepts in your mind. You have to push Spark to work for you, so let's use the collect() method to look at the header:

```
header.collect()
```

- The collect() method brings the entire RDD to a single machine, and you'll get to see the following result:

```
[u'longitude: continuous.', u'latitude: continuous.',
u'housingMedianAge: continuous. ', u'totalRooms:
continuous. ', u'totalBedrooms: continuous. ',
u'population: continuous. ', u'households: continuous
. ', u'medianIncome: continuous. ',
u'medianHouseValue: continuous. ']
```

Tip: be careful when using collect()! Running this line of code can possibly cause the driver to run out of memory. That's why the following approach with the take() method is a safer approach if you want to just print a few elements of the RDD. In general, it's a good principle to limit your result set whenever possible, just like when you're using SQL.

- You learn that the order of the variables is the same as the one that you saw above in the presentation of the data set, and you also learn that all columns should have continuous values. Let's force Spark to do some more work and take a look at the California housing data to confirm this.

Call the take() method on your RDD:

```
rdd.take(2)
```

- By executing the previous line of code, you take the first 2 elements of the RDD. The result is as you expected: because you read in the files with the textFile() function, the lines are just all read in together. The entries are separated by a single comma and the rows themselves are also separated by a comma:

```
[u'-122.230000,37.880000,41.000000,880.000000,129.000000,322.000000,126.000000,8.325200,452600.000000', u'-122.220000,37.860000,21.000000,7099.000000,1106.000000,2401.000000,1138.000000,8.301400,358500.000000']
```

- You definitely need to solve this. Now, you don't need to split the entries, but you definitely need to make sure that the rows of your data are separate elements. To solve this, you'll use the map() function to which you pass a lambda function to split the line at the comma. Then, check your result by running the same line with the take() method, just like you did before.
- **Remember** that lambda functions are anonymous functions which are created at runtime.

```
# Split lines on commas
rdd = rdd.map(lambda line: line.split(","))
# Inspect the first 2 lines
rdd.take(2)
```

You'll get the following result:

```
[[u'-122.230000', u'37.880000', u'41.000000', u'880.000000', u'129.000000', u'322.000000', u'126.000000', u'8.325200', u'452600.000000'], [u'-122.220000', u'37.860000', u'21.000000', u'7099.000000', u'1106.000000', u'2401.000000', u'1138.000000', u'8.301400', u'358500.000000']]
```

Alternatively, you can also use the following functions to inspect your data:

```
# Inspect the first line
rdd.first()

# Take top elements
rdd.top(2)
```

- If you're used to working with Pandas or data frames in R, you'll have probably also expected to see a header, but there is none. To make your life easier, you will move on from the RDD and convert it to a DataFrame. Dataframes are preferred over RDDs whenever you can use them. Especially when you're working with Python, the performance of DataFrames is better than RDDs.
- But what is the difference between the two?
- You can use RDDs when you want to perform low-level transformations and actions on your unstructured data. This means that you don't care about imposing a schema while processing or accessing the attributes by name or column. Tying in to what was said before about performance, by using RDDs, you don't necessarily want the performance benefits that DataFrames can offer for (semi-) structured data. Use RDDs when you want to manipulate the data with functional programming constructs rather than domain specific expressions.
- To recapitulate, you'll switch to DataFrames now to use high-level expressions, to perform SQL queries to explore your data further and to gain columnar access.
- So let's do this.
- The first step is to make a SchemaRDD or an RDD of Row objects with a schema. This is normal, because just like a DataFrame, you eventually want to come to a situation where you have rows and columns. Each entry is linked to a row and a certain column and columns have data types.
- You'll use the map() function again and another lambda function in which you'll map each entry to a field in a Row. To make this more visual, consider this first line:

```
[u'-122.230000', u'37.880000', u'41.000000', u'880.000000', u'129.000000', u'322.000000', u'126.000000', u'8.325200', u'452600.000000']
```

- The lambda function says that you're going to construct a row in a SchemaRDD and that the element at index 0 will have the name "longitude", and so on.
- With this SchemaRDD in place, you can easily convert the RDD to a DataFrame with the toDF() method.

```
# Import the necessary modules
from pyspark.sql import Row

# Map the RDD to a DF
df = rdd.map(lambda line: Row(longitude=line[0],
        latitude=line[1],
        housingMedianAge=line[2],
        totalRooms=line[3],
        totalBedRooms=line[4],
        population=line[5],
        households=line[6],
        medianIncome=line[7],
        medianHouseValue=line[8])).toDF()
```

- Now that you have your DataFrame df, you can inspect it with the methods that you have also used before, namely first() and take(), but also with head() and show():

```
# Show the top 20 rows
df.show()
```

- You'll immediately see that this looks much different from the RDD that you were working with before:

households	housingMedianAge	Latitude	longitude	medianHouseValue	medianIncome	population	totalBedRooms	totalRooms
126.000000	41.000000	37.880000	-122.230000	452600.000000	8.325200	322.000000	129.000000	880.000000
1138.000000	21.000000	37.860000	-122.220000	358500.000000	8.301400	2401.000000	1106.000000	7099.000000
177.000000	52.000000	37.850000	-122.240000	352100.000000	7.257400	496.000000	190.000000	1467.000000
219.000000	52.000000	37.850000	-122.250000	341300.000000	5.643100	558.000000	235.000000	1274.000000
259.000000	52.000000	37.850000	-122.250000	342200.000000	3.846200	565.000000	280.000000	1627.000000
193.000000	52.000000	37.850000	-122.250000	269700.000000	4.036800	413.000000	213.000000	919.000000
514.000000	52.000000	37.840000	-122.250000	299200.000000	3.659100	1094.000000	489.000000	2535.000000
647.000000	52.000000	37.840000	-122.250000	241400.000000	3.120000	1157.000000	587.000000	3104.000000
595.000000	42.000000	37.840000	-122.260000	226700.000000	2.080400	1206.000000	665.000000	2655.000000
714.000000	52.000000	37.840000	-122.250000	261100.000000	3.691200	1551.000000	707.000000	3549.000000
402.000000	52.000000	37.850000	-122.260000	281500.000000	3.203100	910.000000	434.000000	2202.000000
734.000000	52.000000	37.850000	-122.260000	241800.000000	3.270500	1504.000000	752.000000	3503.000000
468.000000	52.000000	37.850000	-122.260000	213500.000000	3.075000	1098.000000	474.000000	2491.000000
174.000000	52.000000	37.840000	-122.260000	191300.000000	2.673600	345.000000	191.000000	696.000000
620.000000	52.000000	37.850000	-122.260000	159200.000000	1.916700	1212.000000	626.000000	2643.000000
264.000000	50.000000	37.850000	-122.260000	140000.000000	2.125000	697.000000	283.000000	1120.000000
331.000000	52.000000	37.850000	-122.270000	152500.000000	2.775000	793.000000	347.000000	1966.000000
303.000000	52.000000	37.850000	-122.270000	155600.000000	2.120200	648.000000	293.000000	1228.000000
419.000000	50.000000	37.840000	-122.260000	158700.000000	1.561100	990.000000	455.000000	2239.000000
275.000000	52.000000	37.840000	-122.270000	162900.000000	2.663500	686.000000	298.000000	1503.000000

Only showing top 20 rows

Tip: use df.columns to return the columns of your DataFrame.

- The data seems all nicely ordered into columns, but what about the data types? By reading in your data, Spark will try to infer a schema, but has this been successful here? Use either df.dtypes or df.printSchema() to get to know more about the data types that are contained within your DataFrame.

```
# Print the data types of all `df` columns
# df.dtypes

# Print the schema of `df`
df.printSchema()
```

- Because you don't execute the first line of code, you will only get back the following result:

```
root
 |-- households: string (nullable = true)
 |-- housingMedianAge: string (nullable = true)
 |-- latitude: string (nullable = true)
 |-- longitude: string (nullable = true)
 |-- medianHouseValue: string (nullable = true)
 |-- medianIncome: string (nullable = true)
 |-- population: string (nullable = true)
 |-- totalBedRooms: string (nullable = true)
 |-- totalRooms: string (nullable = true)
```

- All columns are still of data type string... That's disappointing!
- If you want to continue with this DataFrame, you'll need to rectify this situation and assign "better" or more accurate data types to all columns. Your performance will also benefit from this. Intuitively, you could go for a solution like the following, where you declare that each column of the DataFrame df should be cast to a FloatType():

```
from pyspark.sql.types import *
df = df.withColumn("longitude",
df["longitude"].cast(FloatType())) \
  .withColumn("latitude",
df["latitude"].cast(FloatType())) \
  .withColumn("housingMedianAge",
df["housingMedianAge"].cast(FloatType())) \
  .withColumn("totalRooms",
df["totalRooms"].cast(FloatType())) \
  .withColumn("totalBedRooms",
df["totalBedRooms"].cast(FloatType())) \
  .withColumn("population",
df["population"].cast(FloatType())) \
  .withColumn("households",
df["households"].cast(FloatType())) \
  .withColumn("medianIncome",
df["medianIncome"].cast(FloatType())) \
  .withColumn("medianHouseValue",
df["medianHouseValue"].cast(FloatType()))
```

- But these repeated calls are quite obscure, error-proof and don't really look nice. Why don't you write a function that can do all of this for you in a more clean way?
- The following User-Defined Function (UDF) takes a DataFrame, column names, and the new data type that you want the have the columns to have. You say that for every column name, you take the column and you cast it to a new data type. Then, you return the DataFrame:

```
# Import all from `sql.types`
from pyspark.sql.types import *

# Write a custom function to convert the data type of
DataFrame columns
def convertColumn(df, names, newType):
  for name in names:
    df = df.withColumn(name, df[name].cast(newType))
  return df
# Assign all column names to `columns`
columns = ['households', 'housingMedianAge',
'latitude', 'longitude', 'medianHouseValue',
'medianIncome', 'population', 'totalBedRooms',
'totalRooms']


# Convert the `df` columns to `FloatType()`
df = convertColumn(df, columns, FloatType())
```

- That already looks much better! You can quickly inspect the data types of df with the printSchema() method, just like you have done before.
- Now that you've got that all sorted out, it's time to really get started on the data exploration. You have seen that columnar access and SQL queries were two advantages of using DataFrames. Well, now it's time to dig a little bit further into that. Let's start small and just select two columns from df of which you only want to see 10 rows:

```
df.select('population','totalBedRooms').show(10)
```

This query gives you the following result:

```
+----------+-------------+
|population|totalBedRooms|
+----------+-------------+
|   322.0|     129.0|
|  2401.0|    1106.0|
|   496.0|     190.0|
|   558.0|     235.0|
|   565.0|     280.0|
|   413.0|     213.0|
|  1094.0|     489.0|
|  1157.0|     687.0|
|  1206.0|     665.0|
|  1551.0|     707.0|
+----------+-------------+
only showing top 10 rows
```

- You can also make your queries more complex, as you see in the following example:

```
df.groupBy("housingMedianAge").count().sort("housingMedianAge",ascending=False).show()
```

Which gives you the following result:

```
+----------------+-----+
|housingMedianAge|count|
+----------------+-----+
|            52.0| 1273|
|            51.0|   48|
|            50.0|  136|
|            49.0|  134|
|            48.0|  177|
|            47.0|  198|
|            46.0|  245|
|            45.0|  294|
|            44.0|  356|
|            43.0|  353|
|            42.0|  368|
|            41.0|  296|
|            40.0|  304|
|            39.0|  369|
|            38.0|  394|
|            37.0|  537|
|            36.0|  862|
|            35.0|  824|
|            34.0|  689|
|            33.0|  615|
+----------------+-----+
only showing top 20 rows
```

Besides querying, you can also choose to describe your data and get some summary statistics. This will most definitely help you after!

```
df.describe().show()
```

Summary	households	housingMedianAge	latitude	longitude	medianHouseValue	medianIncome	population	totalBedRooms	totalRooms
Count	20640	20640	20640	20640	20640	20640	20640	20640	20640
Mean	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Stddev	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Min	1.0	1.0	32.54	-124.35	14999.0	0.4999	3.0	1.0	2.0
Max	6082.0	52.0	41.95	-114.31	500001.0	15.0001	35682.0	6445.0	39320.0

- Look at the minimum and maximum values of all the (numerical) attributes. You see that multiple attributes have a wide range of values: you will need to normalize your dataset.

Data Preprocessing

- With all this information that you gathered from your small exploratory data analysis, you know enough to preprocess your data to feed it to the model.
  - ➢ You shouldn't care about missing values; all zero values have been excluded from the data set.
  - ➢ You should probably standardize your data, as you have seen that the range of minimum and maximum values is quite big.
- There are possibbly some additional attributes that you could add, such as a feature that registers the number of bedrooms per room or the rooms per household.
  - ➢ Your dependent variable is also quite big: To make your life easier, you'll have to adjust the values slightly.

Preprocessing The Target Values

First, let's start with the median HouseValue, your dependent variable. To facilitate your working with the target values, you will express the house values in units of 100,000. That means that a target such as 452600.000000 should become 4.526:

```
# Import all from `sql.functions`
from pyspark.sql.functions import *

# Adjust the values of `medianHouseValue`
df = df.withColumn("medianHouseValue",
col("medianHouseValue")/100000)

# Show the first 2 lines of `df`
df.take(2)
```

- You can clearly see that the values have been adjusted correctly when you look at the result of the `take()` method:

```
[Row(households=126.0, housingMedianAge=41.0,
latitude=37.880001068115234, longitude=-
122.23000335693336, medianHouseValue=4.526,
medianIncome=8.325200080871582, population=322.0,
totalBedRooms=129.0, totalRooms=880.0),
Row(households=1138.0, housingMedianAge=21.0,
latitude=37.86000061035156, longitude=-
122.22000122070312, medianHouseValue=3.585,
medianIncome=8.301400184631348, population=2401.0,
totalBedRooms=1106.0, totalRooms=7099.0)]
```

Feature Engineering

- Now that you have adjusted the values in medianHouseValue, you can also add the additional variables that you read about above. You're going to add the following columns to the data set:
  - **Rooms Per Household** which refers to the number of rooms in households per block group;
  - **Population Per Household**, which basically gives you an indication of how many people live in households per block group; And
  - **Bedrooms Per Room** which will give you an idea about how many rooms are bedrooms per block group;
- As you're working with DataFrames, you can best use the select() method to select the columns that you're going to be working with, namely total Rooms, households, and population. Additionally, you have to indicate that you're working with columns by adding the col() function to your code. Otherwise, you won't be able to do element-wise operations like the division that you have in mind for these three variables:

```
# Import all from `sql.functions` if you haven't yet
from pyspark.sql.functions import *

# Divide `totalRooms` by `households`
roomsPerHousehold =
df.select(col("totalRooms")/col("households"))

# Divide `population` by `households`
populationPerHousehold =
df.select(col("population")/col("households"))

# Divide `totalBedRooms` by `totalRooms`
bedroomsPerRoom =
df.select(col("totalBedRooms")/col("totalRooms"))

# Add the new columns to `df`
df = df.withColumn("roomsPerHousehold",
col("totalRooms")/col("households")) \
  .withColumn("populationPerHousehold",
col("population")/col("households")) \
  .withColumn("bedroomsPerRoom",
col("totalBedRooms")/col("totalRooms"))

# Inspect the result
df.first()
```

- You see that, for the first row, there are about 6.98 rooms per household, the households in the block group consist of about 2.5 people and the amount of bedrooms is quite low with 0.14:

```
Row(households=126.0, housingMedianAge=41.0,
latitude=37.880001068115234, longitude=-
122.23000335693336, medianHouseValue=4.526,
medianIncome=8.325200080871582, population=322.0,
totalBedRooms=129.0, totalRooms=880.0,
roomsPerHousehold=6.984126984126984,
populationPerHousehold=2.555555555555556,
bedroomsPerRoom=0.14659090909090908)
```

- Next, -and this is already forseeing an issue that you might have when you'll standardize the values in your data set- you'll also re-order the values. Since you don't want to necessarily standardize your target values, you'll want to make sure to isolate those in your data set.
- In this case, you'll need to do this by using the select() method and passing the column names in the order that is more appropriate. In this case, the target variable median HouseValue is put first, so that it won't be affected by the standardization.

Note also that this is the time to leave out variables that you might not want to consider in your analysis. In this case, let's leave out variables such as longitude, latitude, housing MedianAge and total Rooms.

```
# Re-order and select columns
df = df.select("medianHouseValue",
               "totalBedRooms",
               "population",
               "households",
               "medianIncome",
               "roomsPerHousehold",
               "populationPerHousehold",
               "bedroomsPerRoom")
```

Standardization

- Now that you have re-ordered the data, you're ready to normalize the data. Or almost, at least. There is just one more step that you need to go through: separating the features from the target variable. In essence, this boils down to isolating the first column in your DataFrame from the rest of the columns.
- In this case, you'll use the map() function that you use with RDDs to perform this action. You also see that you make use of the DenseVector() function. A dense vector is a local vector that is backed by a double array that represents its entry values. In other words, it's used to store arrays of values for use in PySpark.
- Next, you go back to making a DataFrame out of the input_data and you re-label the columns by passing a list as a second argument. This list consists of the column names "label" and "features":

```
# Import `DenseVector`
from pyspark.ml.linalg import DenseVector

# Define the `input_data`
input_data = df.rdd.map(lambda x: (x[0],
DenseVector(x[1:])))

# Replace `df` with the new DataFrame
df = spark.createDataFrame(input_data, ["label",
"features"])
```

- Next, you can finally scale the data. You can use Spark ML to do this: this library will make machine learning on big data scalable and easy. You'll find tools such as ML algorithms and everything you need to build practical ML pipelines. In this case, you don't need to do that much preprocessing so a pipeline would maybe be overkill, but if you want to look into it, definitely consider visiting the this page.
- The input columns are the features, and the output column with the rescaled that will be included in the scaled_df will be named "features_scaled":

```
# Import `StandardScaler`
from pyspark.ml.feature import StandardScaler

# Initialize the `standardScaler`
standardScaler = StandardScaler(inputCol="features",
outputCol="features_scaled")

# Fit the DataFrame to the scaler
scaler = standardScaler.fit(df)

# Transform the data in `df` with the scaler
scaled_df = scaler.transform(df)

# Inspect the result
scaled_df.take(2)
```

- Let's take a look at your DataFrame and the result. You see that, indeed, a third column features_scaled was added to your DataFrame, which you can use to compare with features:

```
[Row(label=4.526, features=DenseVector([129.0, 322.0,
126.0, 8.3252, 6.9841, 2.5556, 0.1466]),
features_scaled=DenseVector([0.3062, 0.2843,
```

```
0.3296, 4.3821, 2.8228, 0.2461, 2.5264])),
Row(label=3.585, features=DenseVector([1106.0,
2401.0, 1138.0, 8.3014, 6.2381, 2.1098, 0.1558]),
features_scaled=DenseVector([2.6255, 2.1202, 2.9765,
4.3696, 2.5213, 0.2031, 2.6851]))]
```

Note that these lines of code are very similar to what you would be doing in Scikit-Learn.

Building A Machine Learning Model With Spark ML

- With all the preprocessing done, it's finally time to start building your Linear Regression model! Just like always, you first need to split the data into training and test sets. Luckily, this is no issue with the randomSplit() method:

```
# Split the data into train and test sets
train_data, test_data =
scaled_df.randomSplit([.8,.2],seed=1234)
```

- You pass in a list with two numbers that represent the size that you want your training and test sets to have and a seed, which is needed for reproducibility reasons. If you want to know more about this, consider DataCamp's Python Machine Learning Tutorial.
- Then, without further ado, you can make your model!

Note that the argument elasticNetParam corresponds to α or the vertical intercept and that the regParam or the regularization parameter corresponds to λ. Go here for more information.

```
# Import `LinearRegression`
from pyspark.ml.regression import LinearRegression

# Initialize `lr`
lr = LinearRegression(labelCol="label", maxIter=10,
regParam=0.3, elasticNetParam=0.8)

# Fit the data to the model
linearModel = lr.fit(train_data)
```

- With your model in place, you can generate predictions for your test data: use the transform() method to predict the labels for your test_data. Then, you can use RDD operations to extract the predictions as well as the true labels from the DataFrame and zip these two values together in a list called predictionAndLabel.
- Lastly, you can then inspect the predicted and real values by simply accessing the list with square brackets []:

```
# Generate predictions
predicted = linearModel.transform(test_data)

# Extract the predictions and the "known" correct
labels
predictions =
predicted.select("prediction").rdd.map(lambda x: x[0])
labels = predicted.select("label").rdd.map(lambda x:
x[0])

# Zip `predictions` and `labels` into a list
predictionAndLabel = predictions.zip(labels).collect()

# Print out first 5 instances of `predictionAndLabel`
predictionAndLabel[:5]
```

You'll see the following real and predicted values (in that order):

```
[(1.4491508524918457, 0.14999), (1.5705029404692372,
0.14999), (2.148727956912464, 0.14999),
(1.5831547768979277, 0.344), (1.5182107797955968,
0.398)]
```

Evaluating the Model

- Looking at predicted values is one thing, but another and better thing is looking at some metrics to get a better idea of how good your model actually is. You can first start by printing out the coefficients and the intercept of the model:

```
# Coefficients for the model
linearModel.coefficients

# Intercept for the model
linearModel.intercept
```

Which gives you the following result:

```
# The coefficients
[0.0,0.0,0.0,0.0,0.276239709215,0.0,0.0,0.0,0.0]

# The intercept
0.990399577462
```

Next, you can also use the summary attribute to pull up the rootMeanSquaredError and the r2.

```
# Get the RMSE
linearModel.summary.rootMeanSquaredError

# Get the R2
linearModel.summary.r2
```

- The RMSE measures how much error there is between two datasets comparing a predicted value and an observed or known value. The smaller an RMSE value, the closer predicted and observed values are.
- The R2 ("R squared") or the coefficient of determination is a measure that shows how close the data are to the fitted regression line. This score will always be between 0 and a 100% (or 0 to 1 in this case), where 0% indicates that the model explains none of the variability of the response data around its mean, and 100% indicates the opposite: it explains all the variability. That means that, in general, the higher the R-squared, the better the model fits your data.

You'll get back the following result:

```
# RMSE
0.8692118678997669
# R2
0.42408952872183 79
```

4.6 MACHINE LEARNING ALGORITHMS IN SPARK

4.6.1 Spark MLlib

- MLlib is Spark's scalable machine learning library consisting of common machine learning algorithms in spark. For example, basic statistics, classification, regression, clustering, collaborative filtering.

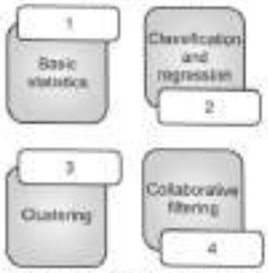


Fig. 4.17 : Introduction of Machine Learning algorithm in Apache Spark

Machine Learning Algorithm (MLlib)

- MLlib is nothing but a Machine Learning (ML) library of **Apache Spark**. Basically, it helps to make practical machine learning scalable and easy. Moreover, it provides the following ML Algorithms:
  - Basic statistics
  - Classification and Regression
  - Clustering
  - Collaborative filtering
- Furthermore, let's start discussing each Machine Learning algorithm one by one.

Spark Machine Learning Algorithm Statistics

- This Machine Learning algorithm in spark consists of several algorithms, such as:
  - Summary statistics
  - Correlations
  - Stratified sampling
  - Hypothesis testing
  - Random data generation

1. Spark Machine Learning Algorithm – Summary Statistics

- Basically, for RDD[Vector] we offer column summary statistics. Moreover, it is possible through the function colStats, available in statistics.
- In addition, function colStats() returns an instance of Multivariate Statistical Summary. That contains the column-wise max, min, mean, variance, and the number of nonzeros, as well as the total count.

```
import org.apache.spark.mllib.linalg.Vector
import
org.apache.spark.mllib.stat.{MultivariateStatisticalSum
mary, Statistics}
val observations: RDD[Vector] = ...
  // an RDD of Vectors
// Compute column summary statistics.
val summary: MultivariateStatisticalSummary =
Statistics.colStats(observations)
println(summary.mean)
// a dense vector containing the mean value for each
column
println(summary.variance) // column-wise variance
println(summary.numNonzeros)
                    // number of nonzeros in each column
```

2. Spark Machine Learning Algorithm – Correlations

- In statistics, calculating the correlation of two series is a common operation. Although, MLlib offers some flexibility. Basically, it helps to calculate pairwise correlations among many series. Moreover, it currently supports two correlation methods. For example, Pearson's and Spearman's correlation.
- In addition, statistics offers methods to calculate correlations between series. However, it depends on the type of input either two RDD[Double]s or an RDD[Vector]. Therefore, the output will be a double or the correlation matrix respectively.

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.stat.Statistics
val sc: SparkContext = ...
val seriesX: RDD[Double] = ... // a series
val seriesY: RDD[Double] = ... // must have the same number of partitions and cardinality as seriesX
// compute the correlation using Pearson's method. Enter "spearman" for Spearman's method. If a
// method is not specified, Pearson's method will be used by default.
val correlation: Double = Statistics.corr(seriesX, seriesY, "pearson")
val data: RDD[Vector] = ... // note that each Vector is a row and not a column
// calculate the correlation matrix using Pearson's method. Use "spearman" for Spearman's method.
// If a method is not specified, Pearson's method will be used by default.
val correlMatrix: Matrix = Statistics.corr(data, "pearson")
```

3. Spark Machine Learning Algorithm – Stratified Sampling

- Basically, stratified sampling methods, sampleByKey and sampleByKeyExact, can be performed on RDD's of key-value pairs. However, for stratified sampling, we can consider keys as a label and the value as a specific attribute.
- Moreover, let's understand this by an example, consider the key as a man or woman, or document ids. Whereas respective values can be the list of ages of the people in the population or the list of words in the documents. However, to decide whether an observation will be sampled or not, the sampleByKey method will flip a coin. Therefore requires one pass over the data. Also provides an expected sample size.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.rdd.PairRDDFunctions
val sc: SparkContext = ...
val data = ... // an RDD[(K, V)] of any key value pairs
val fractions: Map[K, Double] = ... // specify the exact fraction desired from each key
// Get an exact sample from each stratum
val approxSample = data.sampleByKey(withReplacement = false, fractions)
val exactSample =
data.sampleByKeyExact(withReplacement = false, fractions)
```

4. Spark Machine Learning Algorithm – Hypothesis Testing

- In statistics, we can determine whether a result is statistically significant, whether this result occurred by chance or not. Hence, to determine that a powerful tool is Hypothesis testing. Basically, MLlib currently supports Pearson's chi-squared (χ^2). Although it tests for goodness of fit and independence. Moreover, the input data types determine whether the goodness of fit or the independence test is conducted. Also, The goodness of fit test requires an input type of vector, whereas the independence test requires a matrix as input.
- In addition, MLlib also supports the input type RDD[LabeledPoint]. Moreover, it helps to enable feature selection via chred independence tests.
- Moreover, statistics offers methods to run Pearson's chi-squared tests. Furthermore, the example below demonstrates how to run and interpret hypothesis tests.

For Example

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.stat.Statistics._
val sc: SparkContext = ...
```

```
val vec: Vector = ... // a vector composed of the
frequencies of events
// compute the goodness of fit. If a second vector to
test against is not supplied as a parameter,
// the test runs against a uniform distribution.
val goodnessOfFitTestResult =
Statistics.chiSqTest(vec)
println(goodnessOfFitTestResult) // summary of the
test including the p-value, degrees of freedom,
// test statistic, the method used, and the null
hypothesis.
val mat: Matrix = ... // a contingency matrix
// conduct Pearson's independence test on the input
contingency matrix
val independenceTestResult =
Statistics.chiSqTest(mat)
println(independenceTestResult) // summary of the
test including the p-value, degrees of freedom...
val obs: RDD[LabeledPoint] = ... // (feature, label) pairs.
// The contingency table is constructed from the raw
(feature, label) pairs and used to conduct
// the independence test. Returns an array containing
the ChiSquaredTestResult for every feature
// against the label.
val featureTestResults: Array[ChiSqTestResult] =
Statistics.chiSqTest(obs)
var i = 1
featureTestResults.foreach { result =>
println(s"Column $i:\n$result")
i += 1
} // summary of the test
```

5. Spark Machine Learning Algorithm – Random Data Generation

- Basically, for randomized algorithms, prototyping, and performance testing, we use Random data generation. Moreover, MLlib also supports generating random RDDs with i.i.d. Values. However, drawn from a given distribution, either uniform, standard normal, or Poisson.
- In addition, to generate random double RDDs or vector RDDs, RandomRDDs offers factory methods. Let's understand this in the following example. Moreover, it generates a random double RDD, whose values follow the standard normal distribu... Also map it to N(1, 4).

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.random.RandomRDDs._
val sc1: SparkContext = ...
// Generate a random double RDD that contains 1
million i.i.d. values drawn from the
// standard normal distribution `N(0, 1)`, evenly
distributed in 10 partitions.
val u1 = normalRDD(sc, 1000000L, 10)
// Apply a transform to get a random double RDD
following `N(1, 4)`.
val v1 = u.map(x => 1.0 + 2.0 * x)
```

6. Spark Machine Learning Algorithm – Classification and Regression

(a) Classification in Spark Machine Learning Algorithm

(i) Logistic Regression

- To predict a categorical response, logistic regression is a popular method. Basically, it is a special case of Generalized Linear models. Also helps to predict the probability of the outcomes. Moreover, to predict a binary outcome by using binomial logistic regression, we can use logistic regression in spark.ml. Also, we can use it to predict a multiclass outcome by using multinomial logistic regression.

(ii) Decision Tree Classifier

- Basically, A popular family of classification and regression methods is decision trees.

For Example

- In the below examples load a dataset in LibSVM format, split it into training and test sets, train on the first dataset, and then evaluate the held-out test set. Although, we use two feature transformers to prepare the data. Basically, these help index categories for the label and categorical features. Also, helps for adding metadata to the DataFrame which the decision tree algorithm can recognize.

```
import org.apache.spark.ml.Pipeline
import
org.apache.spark.ml.classification.DecisionTreeClassificationModel
import
org.apache.spark.ml.classification.DecisionTreeClassifier
```

```
import
org.apache.spark.ml.evaluation.MulticlassClassificationE
valuator
import org.apache.spark.ml.feature.{IndexToString,
StringIndexer, VectorIndexer}
// Load the data stored in LIBSVM format as a
DataFrame.
val data =
spark.read.format("libsvm").load("data/mllib/sample_lib
svm_data.txt")
// Index labels, adding metadata to the label column.
// Fit on whole dataset to include all labels in index.
val labelIndexer = new StringIndexer()
.setInputCol("label")
.setOutputCol("indexedLabel")
.fit(data)
// Automatically identify categorical features, and
index them.
val featureIndexer = new VectorIndexer()
.setInputCol("features")
.setOutputCol("indexedFeatures")
.setMaxCategories(4) // features with > 4 distinct
values are treated as continuous.
.fit(data)
// Split the data into training and test sets (30% held
out for testing).
val Array(trainingData, testData) =
data.randomSplit(Array(0.7, 0.3))
// Train a DecisionTree model.
val dt = new DecisionTreeClassifier()
.setLabelCol("indexedLabel")
.setFeaturesCol("indexedFeatures")
// Convert indexed labels back to original labels.
val labelConverter = new IndexToString()
.setInputCol("prediction")
.setOutputCol("predictedLabel")
.setLabels(labelIndexer.labels)
// Chain indexers and tree in a Pipeline.
val pipeline = new Pipeline()
.setStages(Array(labelIndexer, featureIndexer, dt,
labelConverter))
// Train model. This also runs the indexers.
val model = pipeline.fit(trainingData)
// Make predictions.
val predictions = model.transform(testData)
// Select example rows to display.
predictions.select("predictedLabel", "label",
"features").show(5)
// Select (prediction, true label) and compute test
error.
val evaluator = new MulticlassClassificationEvaluator()
.setLabelCol("indexedLabel")
.setPredictionCol("prediction")
.setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println("Test Error = " + (1.0 - accuracy))
val treeModel =
model.stages(2).asInstanceOf[DecisionTreeClassificati
onModel]
println("Learned classification tree model:\n" +
treeModel.toDebugString)
```

(b) Regression in Spark Machine Learning Algorithm

(i) Linear Regression

- Basically, for working with linear regression models and model summaries, the interface is similar to the logistic regression case.

  Example for Regression in Machine Learning algorithm

For Example

- Moreover, Below example shows training an elastic net regularized linear regression model and extracting model summary statistics.

```
import org.apache.spark.ml.regression.LinearRegression
// Load training data
val training = spark.read.format("libsvm")
.load("data/mllib/sample_linear_regression_data.txt")
val lr = new LinearRegression()
.setMaxIter(10)
.setRegParam(0.3)
.setElasticNetParam(0.8)
// Fit the model
val lrModel = lr.fit(training)
// Print the coefficients and intercept for linear
regression
println(s"Coefficients: ${lrModel.coefficients}
Intercept: ${lrModel.intercept}")
```

```
// Summarize the model over the training set and print
out some metrics
val trainingSummary = lrModel.summary
println(s"numIterations:
${trainingSummary.totalIterations}")
println(s"objectiveHistory:
[${trainingSummary.objectiveHistory.mkString(",")}]")
trainingSummary.residuals.show()
println(s"RMSE:
${trainingSummary.rootMeanSquaredError}")
println(s"r2: ${trainingSummary.r2}")
```

7. Collaborative Filtering in Spark Machine Learning Algorithm

- For recommender systems, we commonly use collaborative filtering. Also, to fill in the missing entries of a user-item association matrix, we use these techniques. Although, spark.ml currently supports model-based collaborative filtering.
- In addition, these filtering users and products are described by a small set of latent factors. Basically, those we can use to predict missing entries. Also, spark.ml uses the alternating least squares (ALS) algorithm. Moreover, it helps to learn these latent factors. There are following parameters of the implementation in MLlib:
  - Here, numBlocks is the number of blocks. It is used to parallelize computation (set to -1 to auto-configure).
  - Moreover, rank is the number of latent factors in the model.
  - And, iterations is the number of iterations to run.
  - In ALS, lambda specifies the regularization parameter.
  - Moreover, implicitPrefs specifies whether to use the explicit feedback ALS variant or one adapted for implicit feedback data.
  - Basically, alpha is a parameter applicable to the implicit feedback variant of ALS. Also, helps to govern the baseline confidence in preference observations.

(i) Explicit vs. Implicit Feedback

- Basically, the standard approach to matrix factorization-based collaborative filtering treats the entries in the user-item matrix as explicit preferences given by the user to the item.
- Although, in many real-world use cases, it is common to only have access to implicit feedback. For example, views, clicks, purchases, likes, shares and many more. Moreover, to deal with such data in MLlib, the approach used is taken from collaborative filtering for implicit feedback datasets.

(ii) Scaling of the Regularization Parameter

- By the number of ratings the user generated in updating user factors, or the number of ratings the product received in updating product factors, we scale the regularization parameter lambda. Basically, this approach is named "ALS-WR". Moreover, it makes lambda less dependent on the scale of the dataset. Hence, we can apply the best parameter learned from a sampled subset to the full dataset.

(iii) Example for Collaborative Filtering in Machine Learning Algorithm in Spark

For Example

- In the example below, we load rating data. Basically, each row consists of a user, a product, and a rating. Moreover, we will use the default ALS.train() method, that assumes ratings are explicit. Also, by measuring the Mean Squared Error of rating prediction we will evaluate the recommendation model.

```
import org.apache.spark.mllib.recommendation.ALS
import org.apache.spark.mllib.recommendation.Rating
// Load and parse the data
val data = sc.textFile("data/mllib/als/test.data")
val ratings = data.map(_.split(',') match { case
Array(user, item, rate) =>
Rating(user.toInt, item.toInt, rate.toDouble)
})
// Build the recommendation model using ALS
val rank = 10
val numIterations = 20
val model = ALS.train(ratings, rank, numIterations,
0.01)
// Evaluate the model on rating data
val usersProducts = ratings.map { case Rating(user,
product, rate) =>(user, product)
}
val predictions =
model.predict(usersProducts).map { case Rating(user,
product, rate) => ((user, product), rate)
}
```

```
val ratesAndPreds = ratings.map { case Rating(user,
product, rate) => ((user, product), rate)
}.join(predictions)
val MSE = ratesAndPreds.map { case ((user, product),
(r1, r2)) => val err = (r1 - r2)
err * err
}.mean()
println("Mean Squared Error = " + MSE)
```

In addition, we can use the trainImplicit method to get better results, if the rating matrix is derived from another source of information.

```
val alpha = 0.01
val model = ALS.trainImplicit(ratings, rank,
numIterations, alpha)
```

8. Clustering in Machine Learning algorithm in Spark

- Basically, it is an unsupervised learning problem. Here we aim to group subsets of entities with one another on the basis of the notion of similarity. Moreover, we use clustering for exploratory analysis. Also, use it as a component of a hierarchical supervised learning pipeline. However, in that distinct classifiers or regression models are trained for each cluster.
- In addition, MLlib supports k-means clustering. However, it is the most commonly used clustering algorithms. Basically, it clusters the data points into the predefined number of clusters. Moreover, here MLlib implementation includes a parallelized variant of the k-means++ method called kmeans. Moreover, there are following implementation parameters in MLlib:
  - Here, k is the number of desired clusters.
  - Although, maxIterations is the maximum number of iterations to run.
  - Moreover, initialization Mode specifies either random initialization or initialization via k-means.
  - Basically, runs is the number of times to run the k-means algorithm.
  - Further, to determines the number of steps in the k-means algorithm, we use initializationSteps.
  - Moreover, to determines the distance threshold within which we consider k-means to have converged we use epsilon.

In spark-shell, it is possible to execute the following code snippets.

Example for Clustering in Machine Learning Algorithm in Spark

For Example

- Basically, after loading and parsing data, we use the KMeans object in the example to cluster the data into two clusters. Moreover, the number of desired clusters is passed to the algorithm. Afterwards, we compute Within Set Sum of Squared Error (WSSSE). Also, can reduce this error measure by increasing k.

```
import org.apache.spark.mllib.clustering.KMeans
import org.apache.spark.mllib.linalg.Vectors
// Load and parse the data
val data1 = sc.textFile("data1/mllib/kmeans_data.txt")
val parsedData1 = data.map(s => Vectors.dense(s.split('
').map(_.toDouble)))
// Cluster the data into two classes using KMeans
val numClusters1 = 2
val numIterations1 = 20
val clusters1 = KMeans.train(parsedData1, numClusters1,
numIterations1)
// Evaluate clustering by computing Within Set Sum of
Squared Errors
val WSSSE1 = clusters1.computeCost(parsedData)
println("Within Set Sum of Squared Errors = " +
WSSSE1)
```

4.7 DEEP LEARNING FOR BIG DATA

- Big Data Analytics and Deep Learning are two high-focus of data science. Big Data has become important as many organizations both public and private have been collecting massive amounts of domain-specific information, which can contain useful information about problems such as national intelligence, cyber security, fraud detection, marketing, and medical informatics. Companies such as Google and Microsoft are analyzing large volumes of data for business analysis and decisions, impacting existing and future technology.
- Deep Learning algorithms extract high-level, complex abstractions as data representations through a hierarchical learning process. Complex abstractions are learnt at a given level based on relatively simpler abstractions formulated in the preceding level in the hierarchy. A key benefit of Deep Learning is the analysis and learning of massive amounts of

unsupervised data, making it a valuable tool for Big Data Analytics where raw data is largely unlabeled and un-categorized. In the present study, we explore how Deep Learning can be utilized for addressing some important problems in Big Data Analytics, including extracting complex patterns from massive volumes of data, semantic indexing, data tagging, fast information retrieval, and simplifying discriminative tasks.

- The general focus of machine learning is the representation of the input data and generalization of the learnt patterns for use on future unseen data. The goodness of the data representation has a large impact on the performance of machine learners on the data: a poor data representation is likely to reduce the performance of even an advanced, complex machine learner, while a good data representation can lead to high performance for a relatively simpler machine learner. Thus, feature engineering, which focuses on constructing features and data representations from raw data, is an important element of machine learning.
- Deep Learning algorithms are one promising avenue of research into the automated extraction of complex data representations (features) at high levels of abstraction. Such algorithms develop a layered, hierarchical architecture of learning and representing data, where higher-level (more abstract) features are defined in terms of lower-level (less abstract) features.
- The hierarchical learning architecture of Deep Learning algorithms is motivated by artificial intelligence emulating the deep, layered learning process of the primary sensorial areas of the neocortex in the human brain, which automatically extracts features and abstractions from the underlying data. Deep Learning algorithms are quite beneficial when dealing with learning from large amounts of unsupervised data, and typically learn data representations in a greedy layer-wise
- Big Data represents the general realm of problems and techniques used for application domains that collect and maintain massive volumes of raw data for domain-specific data analysis. Modern data-intensive technologies as well as increased computational and data storage resources have contributed heavily to the development of Big Data science.
- Technology based companies such as Google, Yahoo, Microsoft, and Amazon have collected and maintained data that is measured in exabyte proportions or larger. Moreover, social media organizations such as Facebook, YouTube, and Twitter have billions of users that constantly generate a very large quantity of data. Various organizations have invested in developing products using Big Data Analytics to addressing their monitoring, experimentation, data analysis, simulations, and other knowledge and business needs, making it a central topic in data science research.
- Mining and extracting meaningful patterns from massive input data for decision-making, prediction, and other inferencing is at the core of Big Data Analytics. In addition to analyzing massive volumes of data, Big Data Analytics poses other unique challenges for machine learning and data analysis, including format variation of the raw data, fast-moving streaming data, trustworthiness of the data analysis, highly distributed input sources, noisy and poor quality data, high dimensionality, scalability of algorithms, imbalanced input data, unsupervised and un-categorized data, limited supervised/labeled data, etc. Adequate data storage, data indexing/tagging, and fast information retrieval are other key problems in Big Data Analytics. Consequently, innovative data analysis and data management solutions are warranted when working with Big Data.
- The knowledge learnt from (and made available by) Deep Learning algorithms has been largely untapped in the context of Big Data Analytics. Certain Big Data domains, such as computer vision and speech recognition, have seen the application of Deep Learning largely to improve classification modeling results. The ability of Deep Learning to extract high-level, complex abstractions and data representations from large volumes of data, especially unsupervised data, makes it attractive as a valuable tool for Big Data Analytics. More specifically, Big Data problems such as semantic indexing, data tagging, fast information retrieval, and discriminative modeling can be better addressed with the aid of Deep Learning. More traditional machine learning and feature engineering algorithms are not efficient enough to extract the complex and non-linear patterns generally observed in Big Data. By extracting such features, Deep Learning enables the use of relatively simpler linear models for Big Data analysis tasks, such as classification and

prediction, which is important when developing models to deal with the scale of Big Data.

- The main concept in deep learning algorithms is automating the extraction of representations (abstractions) from the data. Deep learning algorithms use a huge amount of unsupervised data to automatically extract complex representation. These algorithms are largely motivated by the field of artificial intelligence, which has the general goal of emulating the human brain's ability to observe, analyze, learn, and make decisions, especially for extremely complex problems.
- Work pertaining to these complex challenges has been a key motivation behind Deep Learning algorithms which strive to emulate the hierarchical learning approach of the human brain. Models based on shallow learning architectures such as decision trees, support vector machines, and case-based reasoning may fall short when attempting to extract useful information from complex structures and relationships in the input corpus. In contrast, Deep Learning architectures have the capability to generalize in non-local and global ways, generating learning patterns and relationships beyond immediate neighbors in the data.
- Deep learning is in fact an important step toward artificial intelligence. It not only provides complex representations of data which are suitable for AI tasks but also makes the machines independent of human knowledge which is the ultimate goal of AI. It extracts representations directly from unsupervised data without human interference.
- A key concept underlying Deep Learning methods is distributed representations of the data, in which a large number of possible configurations of the abstract features of the input data are feasible, allowing for a compact representation of each sample and leading to a richer generalization.
- The number of possible configurations is exponentially related to the number of extracted abstract features. Noting that the observed data was generated through interactions of several known/unknown factors, and thus when a data pattern is obtained through some configurations of learnt factors, additional (unseen) data patterns can likely be described through new configurations of the learnt factors and patterns. Compared to learning based on local generalizations, the number of patterns that can be obtained using a distributed representation scales quickly with the number of learnt factors.
- Deep learning algorithms lead to abstract representations because more abstract representations are often constructed based on less abstract ones. An important advantage of more abstract representations is that they can be invariant to the local changes in the input data. Learning such invariant features is an ongoing major goal in pattern recognition (for example learning features that are invariant to the face orientation in a face recognition task). Beyond being invariant such representations can also disentangle the factors of variation in data.
- The real data used in AI-related tasks mostly arise from complicated interactions of many sources. For example an image is composed of different sources of variations such a light, object shapes, and object materials. The abstract representations provided by deep learning algorithms can separate the different sources of variations in data.
- Deep learning algorithms are actually Deep architectures of consecutive layers. Each layer applies a nonlinear transformation on its input and provides a representation in its output. The objective is to learn a complicated and abstract representation of the data in a hierarchical manner by passing the data through multiple transformation layers. The sensory data (for example pixels in an image) is fed to the first layer. Consequently the output of each layer is provided as input to its next layer.

Big Data Analytics

- Big Data generally refers to data that exceeds the typical storage, processing, and computing capacity of conventional databases and data analysis techniques. As a resource, Big Data requires tools and methods that can be applied to analyze and extract patterns from large-scale data. The rise of Big Data has been caused by increased data storage capabilities, increased computational processing power, and availability of increased volumes of data, which give organization more data than they have computing resources and technologies to process. In addition to

the obvious great volumes of data, Big Data is also associated with other specific complexities, often referred to as the four Vs: Volume, Variety, Velocity, and Veracity.

- The unmanageable large Volume of data poses an immediate challenge to conventional computing environments and requires scalable storage and a distributed strategy to data querying and analysis. However, this large Volume of data is also a major positive feature of Big Data. Many companies, such as Facebook, Yahoo, Google, already have large amounts of data and have recently begun tapping into its benefits.
- A general theme in Big Data systems is that the raw data is increasingly diverse and complex, consisting of largely un-categorized/unsupervised data along with perhaps a small quantity of categorized/supervised data. Working with the Variety among different data representations in a given repository poses unique challenges with Big Data, which requires Big Data preprocessing of unstructured data in order to extract structured/ordered representations of the data for human and/or downstream consumption. In today's data-intensive technology era, data Velocity the increasing rate at which data is collected and obtained – is just as important as the Volume and Variety characteristics of Big Data.
- While the possibility of data loss exists with streaming data if it is generally not immediately processed and analyzed, there is the option to save fast-moving data into bulk storage for batch processing at a later time. However, the practical importance of dealing with Velocity associated with Big Data is the quickness of the feedback loop, that is, process of translating data input into useable information.
- This is especially important in the case of time-sensitive information processing. Some companies such as Twitter, Yahoo, and IBM have developed products that address the analysis of streaming data. Veracity in Big Data deals with the trustworthiness or usefulness of results obtained from data analysis, and brings to light the old adage "Garbage-In-Garbage-Out" for decision making based on Big Data Analytics. As the number of data sources and types increases, sustaining trust in Big Data Analytics presents a practical challenge.
- Big Data Analytics faces a number of challenges beyond those implied by the four Vs. While not meant to be an exhaustive list, some key problem areas include: data quality and validation, data cleansing, feature engineering, high-dimensionality and data reduction, data representations and distributed data sources, data sampling, scalability of algorithms, data visualization, parallel and distributed data processing, real-time analysis and decision making, crowd sourcing and semantic input for improved data analysis, tracing and analyzing data provenance, data discovery and integration, parallel and distributed computing, exploratory data analysis and interpretation, integrating heterogenous data, and developing new models for massive data computation.

Four Applications of Deep Learning for Big Data

Deep learning, with artificial neural networks at its core, is a new and powerful tool that can be used to derive value from big data. Most of the data today is unstructured, and deep learning algorithms are very effective at learning from, and generating predictions for, wildly unstructured data. Following are several ways deep learning is being applied to big data sets.

1. **Text Classification and Automatic Tagging**

- Deep learning architectures including Recurrent Neural Networks and Convolutional Neural Networks are used to process free text, perform sentiment analysis and identify which categories or types it belongs to. This can help search through, organize and make use of huge unstructured datasets.

2. **Automatic Image Caption Generation**

- Deep learning is used to identify the contents of an image and automatically generate descriptive text, turning images from unstructured to structured and searchable content. This involves the use of Convolutional Neural Networks, in particular very large networks like ResNet, to perform object detection, and then using Recurrent Neural networks to write coherent sentences based on object classification.

3. **Deep Learning in Finance**

- Today, most financial transactions are electronic. Stock markets generate huge volumes of data reflecting buy and sell actions, and the resulting financial metrics such as stock prices. Deep learning can ingest these huge data volumes, understand the current market

position and create an accurate model of the probabilities of future price movements.

- However deep learning is mainly used for analyzing macro trends or making one-time decisions such as analyzing the possibility of company bankruptcy; it is still limited in its ability to drive real-time buying decisions.

4. Deep Learning in Healthcare

- Deep learning, particularly computer vision algorithms, are used to help diagnose and treat patients. Deep learning algorithms analyze blood samples, track glucose levels in diabetic patients, detect heart problems, analyze images to detect tumors, can diagnose cancer, and are able to detect osteoarthritis from an MRI scan before damage is caused to bone structures.

Key Challenges of Deep Learning and Big Data

- While deep learning has tremendous potential to help derive more value from big data, is still in its infancy, and there are significant challenges facing researchers and practitioners. Some of these challenges are:
  - ➢ **Deep Learning Needs Enough Quality Data :** As a general rule, neural networks need more data to make more powerful abstractions. While big data scenarios have abundant data, the data is not always correct or of sufficiently high quality to enable training. Small variations or unexpected features of the input data can completely throw off neural network models.
  - ➢ **Deep Learning has Difficulty with Changing Context :** A neural network model trained on a certain problem will find it difficult to answer very similar problems, presented in a different context. For example, deep learning systems which can effectively detect a set of images can be stumped when presented by the same images, rotated or with different characteristics (grayscale vs. color, different resolution, etc).
  - ➢ **Security Concerns :** Deep learning needs to train and retrain on massive, realistic datasets, and during the process of developing an algorithm, that data needs to be transferred, stored, and handled securely. When a deep learning algorithm is deployed in a mission-critical environment, attackers can affect the output of the neural network by making small, malicious changes to inputs. This could change financial outcomes, result in wrong patient diagnosis, or crash a self-driving car.
  - ➢ **Real-Time Decisions :** Much of the world's big data is streamed in real time, and real time data analytics is growing in importance. Deep learning is difficult to use for real time data analysis, because it is very computationally intensive. For example, computer vision algorithms went through several generations over the course of two decades, until they became fast enough to detect objects in a live video stream.
  - ➢ **Neural Networks are Black Boxes :** Organizations who deal with big data need more than just good answers. They need to justify those answers and understand why they are correct. Deep learning algorithms rely on millions of parameters to reach decisions, and it is often impossible to explain "why" the neural network selected one label over another. This opacity will limit the ability to use deep learning for critical decisions such as patient treatment in healthcare or large financial investments.

4.8 GRAPH PROCESSING

- Graph processing is useful for many applications from social networks to advertisements. Inside a big data scenario, we need a tool to distribute that processing load.
- Graph analytics, which is an analytics alternative that uses an abstraction called a graph model. The simplicity of this model allows for rapidly absorbing and connecting large volumes of data from many sources in ways that finesse limitations of the source structures (or lack thereof, of course). Graph analytics is an alternative to the traditional data warehouse model as a framework for absorbing both structured and unstructured data from various sources to enable analysts to probe the data in an undirected manner.
- Big data analytics systems should enable a platform that can support different analytics techniques that can be adapted in ways that help solve a variety of challenging problems. This suggests that these systems are high performance, elastic distributed data environments that enable the use of creative

algorithms to exploit variant modes of data management in ways that differ from the traditional batch-oriented approach of traditional approaches to data warehousing.

1. Introduction Graph Analytics

- It is worth delving somewhat into the graph model and the methods used for managing and manipulating graphs.

What Constitutes Graph Analytics?

- Types of problems that are suited to graph analytics.
- Types of questions that are addressed using graph analytics.
- Types of graphs that are commonly encountered.
- The degree of prevalence within big data analytics problems.
- This motivates an understanding of its utility and flexibility for discovery-style analysis in relation to specific types of business problems, how common those types of problems are, and why they are nicely abstracted to the graph model. In addition, we will discuss the challenges of attempting to execute graph analytics on conventional hardware and consider aspects of specialty platforms that can help in achieving the right level of scalability and performance.

The Simplicity of the Graph Model

- Graph analytics is based on a model of representing individual entities and numerous kinds of relationships that connect those entities. More precisely, it employs the graph abstraction for representing connectivity, consisting of a collection of vertices (which are also referred to as nodes or points) that represent the modeled entities, connected by edges (which are also referred to as links, connections, or relationships) that capture the way that two entities are related.
- The flexibility of the model is based on its simplicity. A simple unlabeled undirected graph, in which the edges between vertices neither reflect the nature of the relationship nor indicate their direction, has limited utility.
- Among other enhancements, these can enrich the meaning of the nodes and edges represented in the graph model:
  - ➢ Vertices can be labeled to indicate the types of entities that are related.
  - ➢ Edges can be labeled with the nature of the relationship.
  - ➢ Edges can be directed to indicate the "flow" of the relationship.
  - ➢ Weights can be added to the relationships represented by the edges.
  - ➢ Additional properties can be attributed to both edges and vertices.
  - ➢ Multiple edges can reflect multiple relationships between pairs of vertices.

Representation as Triples

- In essence, these enhancements help in building a semantic graph a directed graph that can be represented using a triple format consisting of a subject (the source point of the relationship), an object (the target), and a predicate (that models the type of the relationship).
- A collection of these triples is called a semantic database, and this kind of database can capture additional properties of each triple relationship as attributes of the triple. Almost any type of entity and relationship can be represented in a graph model, which means two key things: the process of adding new entities and relationships is not impeded when new datasets are to be included, with new types of entities and connections to be incorporated, and the model is particularly suited to types of discovery analytics that seek out new patterns embedded within the graph that are of critical business interest.

Graphs and Network Organization

- The concept of the social network has been around for many years, and in recent times has been materialized as communities in which individual entities create online personas and connect and interact with others within the community. Yet the idea of the social network extends beyond specific online implementations, and encompasses a wide variety of example environments that directly map to the graph model. That being said, one of the benefits of the graph model is the ability to detect patterns or organization that are inherent within the represented network, such as:
  - ➢ **Embedded Micronetworks** : Looking for small collections of entities that form embedded "microcommunities." Some examples include

determining the originating sources for a hot new purchasing trend, identifying a terrorist cell based on patterns of communication across a broad swath of call detail records, or sets of individuals within a particular tiny geographic region with similar political views.

- **Communication Models:** Modeling communication across a community triggered by a specific event, such as monitoring the "buzz" across a social media channel associated with the rumored release of a new product, evaluating best methods for communicating news releases, or correlation between travel delays and increased mobile telephony activity.
- **Collaborative Communities:** Isolating groups of individuals that share similar interests, such as groups of health care professionals working in the same area of specialty, purchasers with similar product tastes, or individuals with a precise set of employment skills.

  Influence modeling: Looking for entities holding influential positions within a network for intermittent periods of time, such as computer nodes that have been hijacked and put to work as proxies for distributed denial of service attacks or for emerging cybersecurity threats, or individuals that are recognized as authorities within a particular area.
- **Distance Modeling:** Analyzing the distances between sets of entities, such as looking for strong correlations between occurrences of sets of statistically improbable phrases among large sets of search engines queries, or the amount of effort necessary to propagate a message among a set of different communities.

  Each of these example applications is a discovery analysis that looks for patterns that are not known in advance. As a result, these are less suited to pattern searches form relational database systems, such as a data warehouse or data mart, and are better suited to a more dynamic representation like the graph model.

2. Choosing Graph Analytics

- Deciding the appropriateness of an analytics application to a graph analytics solution instead of the other big data alternatives can be based on these characteristics and factors of business problems:
  - **Connectivity:** The solution to the business problem requires the analysis of relationships and connectivity between a variety of different types of entities.
  - **Undirected Discovery:** Solving the business problem involves iterative undirected analysis to seek out as-of-yet unidentified patterns.
  - **Absence of Structure:** Multiple datasets to be subjected to the analysis are provided without any inherent imposed structure.
  - **Flexible Semantics:** The business problem exhibits dependence on contextual semantics that can be attributed to the connections and corresponding relationships.
  - **Extensibility:** Because additional data can add to the knowledge embedded within the graph, there is a need for the ability to quickly add in new data sources or streaming data as needed for further interactive analysis.
  - **Knowledge is Embedded in the Network:** Solving the business problem involves the ability to exploit critical features of the embedded relationships that can be inferred from the provided data.
  - **Ad Hoc Nature of the Analysis:** There is a need to run ad hoc queries to follow lines of reasoning.
  - **Predictable Interactive Performance:** The ad hoc nature of the analysis creates a need for high performance because discovery in big data is a collaborative man/machine undertaking, and predictability is critical when the results are used for operational decision making.

3. Graph Analytics Algorithms and Solution Approaches

- The graph model is inherently suited to enable a broad range of analyses that are generally unavailable to users of a standard data warehouse framework. As suggested by these examples, instead of just providing reports or enabling online analytical processing (OLAP) systems, graph analytics applications employ algorithms that traverse or analyze graphs to detect and potentially identify interesting patterns that are sentinels for business opportunities for increasing revenue, identifying security risks, detecting fraud,

waste, or abuse, financial trading signals, or even looking for optimal individualized health care treatments. Some of the types of analytics algorithmic approaches include:

- Community and network analysis, in which the graph structures are traversed in search of groups of entities connected in particularly "close" ways. One example is a collection of entities that are completely connected (i.e., each member of the set is connected to all other members of the set).
  - Path analysis, which analyze the shapes and distances of the different paths that connect entities within the graph.
  - Clustering, which examines the properties of the vertices and edges to identify characteristics of entities that can be used to group them together.
  - Pattern detection and pattern analysis, or methods for identifying anomalous or unexpected patterns requiring further investigation.
  - Probabilistic graphical models such as Bayesian networks or Markov networks for various application such as medical diagnosis, protein structure prediction, speech recognition, or assessment of default risk for credit applications.
  - Graph metrics that are applied to measurements associated with the network itself, including the degree of the vertices (i.e., the number of edges in and out of the vertex), or centrality and distance (including the degree to which particular vertices are "centrally located" in the graph, or how close vertices are to each other based on the length of the paths between them).
- These graph analytic algorithms can yield interesting patterns that might go undetected in a data warehouse model, and these patterns them-selves can become the templates or models for new searches. In other words, the graph analytics approach can satisfy both the discovery and the use of patterns typically used for analysis and reporting.

4. Dedicated Appliances for Graph Analytics

- There are different emerging methods of incorporating graph analytics into the enterprise. One class is purely a software approach, providing an ability to create, populate, and query graphs. This approach enables the necessary functionality and may provide the ease-of-implementation and deployment. Most, if not all, software implementations will use industry standards, such as RDF and SPARQL, and may even leverage complementary tools for inferencing and deduction. However, the performance of a software-only implementation is limited by its use of the available hardware, and even using commodity servers cannot necessarily enable it to natively exploit performance and optimization.
- Another class is the use of a dedicated appliance for graph analytics. From an algorithmic perspective, this approach is equally capable as one that solely relies on software. However, from a performance perspective, there is no doubt that a dedicated platform will take advantage of high-performance I/O, high-bandwidth net-working, in-memory computation, and native multithreading to provide the optimal performance for creating, growing, and analyzing graphs that are built from multiple high-volume data streams. Software approaches may be satisfactory for smaller graph analytics problems, but as data volumes and network complexity grow, the most effective means for return on investment may necessitate the transition to a dedicated graph analytics appliance.

4.8.1 Pregel

- Many problems by nature form a graph such as the web, transport and social media. However, a common large scale distributed processing pipeline, MapReduce is ill suited for such tasks. MapReduce requires the data chunks to be processed independently. In other words, every job needs to know all the information to compute. This independence is certainly not the case with graphs and jobs might require previous computation or information from neighbouring jobs in order to calculate. To address this issue, Google's Pregel architecture employs a message passing system creating a "large-scale graph processing" framework.
- Another problem is the fact that MapReduce moves data around, for example by shuffling, in order to process it. This approach entails graph partitions moving around machines incurring high network overhead. A similar problem arises when MapReduce jobs are chained to implement iterative graph algorithms. Therefore, "MapReduce requires passing the entire state of the graph from one stage to the

next - in general requiring much more communication and associated serialization overhead."

- Finally, "coordinating the steps of a chained MapReduce adds programming complexity that is avoided by Pregel's iteration over supersteps". MapReduce is not iterative, it can handle single iteration and requires the user to handle the iteration, i.e. the chaining of multiple MapReduce jobs. This situation can get complicated really quick for real world graph algorithms running at scale not only to implement but also to debug when things go wrong.

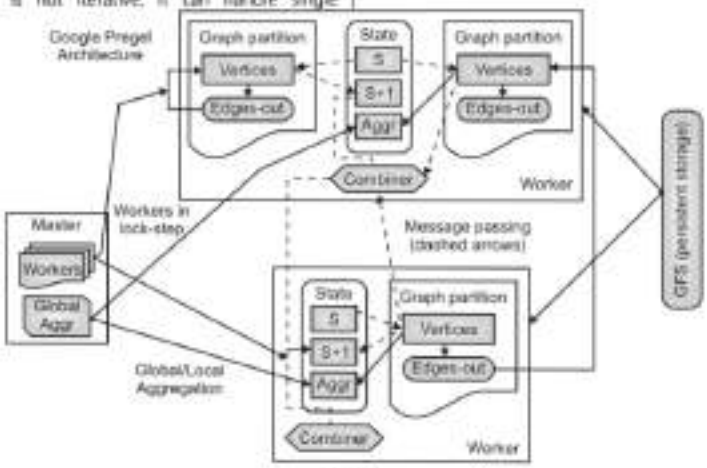


Fig. 4.18

- The user algorithm is distributed to many machines one of which becomes the **Master**. The Master partitions the graph into multiple *graph partitions* and assigns them to workers. A worker can have multiple graph partitions which contain a set of vertices and their corresponding outgoing edges which might point to any vertex in the graph, possibly stored on another worker.
- Once the setup is complete, the master tells each worker to execute a **superstep**. At the beginning of a superstep, workers save their current state into persistent storage for fault tolerance. Workers then `Compute()` the user function for active vertices in their graph partitions sending and receiving messages asynchronously. When the worker is done computing, it tells the master it has finished and how many vertices will be active in the next superstep.
- For **message passing**, two queues are maintained for superstep `S` and `S+1` to simultaneously receive messages while computing the current superstep. Any message sent at superstep `S` is received at superstep `S+1` to avoid locking at message level. There is no guarantee on the ordering of received messages. While computing, a worker can send messages to other workers including itself depending on where the vertex is stored. Messages are sent in a single network message when a certain message buffer size threshold is reached in order to reduce network traffic. If an optional *combiner* is specified by the user, messages can be combined when either outbound or inbound reducing network usage and storage respectively. For example, several messages can be combined into one if the user is interested in the sum of their value.
- Aggregators can be specified by the user to collect results at each superstep. Workers provide values to an aggregator instance to produce a single local value `Aggr`. At the end of the superstep, workers combine the partially reduced aggregator local values into a global value using a tree-based reduction and deliver it to the master.
- When all workers complete a superstep, the master initiates the execution of the next superstep. This

process repeats until there are no active vertices. The vertices become inactive after they vote to halt and there are no incoming messages to wake them up again. When all vertices are inactive, the execution of the algorithm terminates.

Alternative to Message Passing

- An alternative to message passing is a state passing model in which the sent messages are stored locally and only exchanged at the end of a superstep. In other words, the model latches onto the synchronisation barrier between supersteps to exchange the state of message queues rather than during computation. As an immediate advantage, workers don't incur any network traffic during computation. At the end of a superstep, messages can be sent in bulk, larger than the current buffer size, compensating for the handshake overhead of opening and closing multiple connections to different machines. In this case, all messages destined to a another worker will be sent in one network connection.
- A disadvantage would be a longer pause time in between supersteps. State passing approach would be worse whenever computation is less than information passing. For example, if nodes only compute the average of neighbours, getting neighbour values outweighs computing an average causing the state to be transferred at the end of the first superstep. Perhaps, the user can specify depending on the algorithm when messages should be exchanged prioritising computation or communication. In this manner, the message passing of the Pregel execution can be fine tuned to optimise the performance of the user's algorithm and dataset.

Scalability Limitation

- A common limitation with large-scale distributed frameworks is the network traffic message passing creates. For dense graphs, the network could pose as a bottleneck when a lot of messages are passed during computation. Similarly if there is a node with a high degree, whenever it sends or receives messages a single worker will have to deal with all the network traffic causing that worker to slow down and increase wait time on superstep barrier synchronisation.
- Another limitation is the superstep barrier itself when every worker waits for other workers to complete to proceed to the next superstep. When scaled horizontally, the increased probability of failing machines might mean that the computation can wait a long time until all workers have completed and are ready for the next superstep. For example, while a worker is being recovered another can fail causing another recovery during which the overall computation is stagnating. Unlike MapReduce in which other sub branches of the computation such as reduction continue, Pregel needs to wait for the barrier to be passed.

4.8.2 Giraph

- Apache Giraph is an iterative graph processing framework, built on top of Apache Hadoop.
- The input to a Giraph computation is a graph composed of vertices and directed edges, see Fig. 4.19. For example vertices can represent people, and edges friend requests. Each vertex stores a value, so does each edge. The input, thus, not only determines the graph topology, but also the initial values of vertices and edges.
- As an example, consider a computation that finds the distance from a predetermined source person s to any person in the social graph. In this computation, the value of an edge E is a floating point number denoting distance between adjacent people. The value of a vertex V is also a floating point number, representing an upper bound on the distance along a shortest path from the predetermined vertex s to v. The initial value of the predetermined source vertex s is 0, and the initial value for any other vertex is infinity.

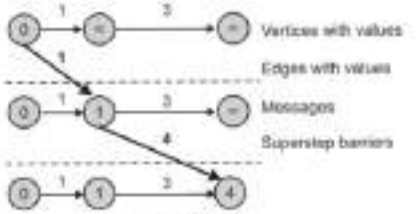


Fig. 4.19

- Fig. 4.19: An illustration of an execution of a single source shortest paths algorithm in Giraph. The input is a chain graph with three vertices (black) and two edges (thin lines). The values of edges are 1 and 3 respectively. The algorithm computes distances from

the leftmost vertex. The initial values of vertices are 0, ∞ and ∞ (top row). Distance upper bounds are sent as messages (dark lines), resulting in updates to vertex values (successive rows going down). The execution lasts three supersteps (separated by dotted lines).

- Computation proceeds as a sequence of iterations, called supersteps in BSP. Initially, every vertex is active. In each superstep each active vertex invokes the Compute method provided by the user. The method implements the graph algorithm that will be executed on the input graph. Intuitively, you should think like a vertex when designing a Giraph algorithm, it is vertex oriented. The Compute method:
  - ➢ Receives messages sent to the vertex in the previous superstep,
  - ➢ Computes using the messages, and the vertex and outgoing edge values, which may result in modifications to the values, and
  - ➢ May send messages to other vertices.
- The Compute method does not have direct access to the values of other vertices and their outgoing edges. Inter-vertex communication occurs by sending messages.
- In our single-source shortest paths example, a Compute method will:
  1. Find the minimum value arriving on any message,
  2. If that value is less than the current value of the vertex, then
  3. The minimum will be adopted as the vertex value, a
  4. The value plus the edge value will be sent along every outgoing edge.

```
public void compute(Iterable<DoubleWritable>
messages) {
  double minDist = Double.MAX_VALUE;
  for (DoubleWritable message : messages) {
   minDist = Math.min(minDist, message.get());
  }
  if (minDist < getValue().get()) {
   setValue(new DoubleWritable(minDist));
   for (Edge<LongWritable, FloatWritable> edge :
getEdges()) {
    double distance = minDist + edge.getValue().get();
    sendMessage(edge.getTargetVertexId(), new
DoubleWritable(distance));
   }
  }
  voteToHalt();
}
```

- There is a barrier between consecutive supersteps. By this we mean that:
  1. The messages sent in any current superstep get delivered to the destination vertices only in the next superstep,
  2. Vertices start computing the next superstep after every vertex has completed computing the current superstep.
- The graph can be mutated during computation by adding or removing vertices or edges. Our example shortest paths algorithm does not mutate the graph.
- Values are retained across barriers. That is, the value of any vertex or edge at the beginning of a superstep is equal to the corresponding value at the end of the previous superstep, when graph topology is not mutated. For example, when a vertex has set the distance upper bound to D, then at the beginning of the next superstep the distance upper bound will still be equal D. Of course the vertex can modify the value of the vertex and of the outgoing edges during any superstep.
- Any vertex can stop computing after any superstep. The vertex simply declares that it does not want to be active anymore. However, any incoming message will make the vertex active again.
- The computation halts after the vertices have voted to halt and there are no messages in flight. Each vertex outputs some local information, which usually amounts to the final vertex value.
- Giraph is a scalable and fault-tolerant implementation of graphic algorithms in Hadoop clusters for thousands of computing nodes.
- Used on a large scale by LinkedIn, Twitter and Facebook for social analytics, representing billions or trillions of connections in datasets to identify popularity, importance, location, and interests among users.
- Sets of nodes or objects compose a graph, connect to each other by edges, represented in matrix form and processed by computers.
- It utilizes them for modeling things such as road networks, social networks, goods flow, among other math applications.

- One example is a web HTML page. The pages are vertices connected by edges representing such links.
- A web graph has several billion vertices and several billion edges.

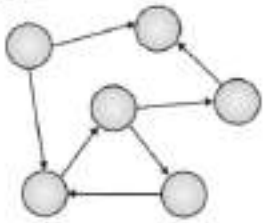


Fig. 4.20

- ➢ Pregel is a framework and graphics processing architecture developed by Google and referred to in an article in 2010. Giraph originated from Pregel.
- ➢ Giraph runs in memory and can be a task in MapReduce. Giraph uses Zookeeper for synchronization through the network nodes.
- ➢ Graph theory is a branch of mathematics that studies the relationships between objects in a dataset.

4.8.3 Spark GraphX

- Today, big graphs exist in various important applications, be it web, advertising, or social networks. Examples of a few of such graphs are presented on the sections.

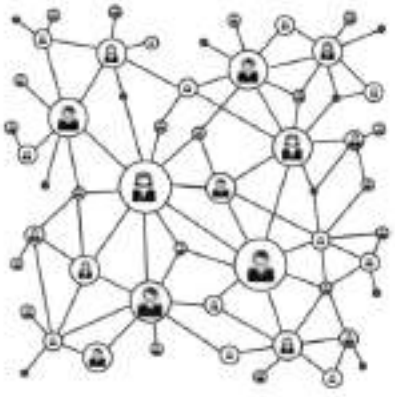


Fig. 4.21

Fig. 4.22

- These graphs allow the users to perform tasks such as target advertising, identify communities, and decipher the meaning of documents. This is possible by modeling the relations between products, users, and ideas.
- As the size and significance of graph data is growing, various new large-scale distributed graph-parallel frameworks, such as GraphLab, Giraph, and PowerGraph, have been developed. With each framework, a new programming abstraction is available. These abstractions allow the users to explain graph algorithms in a compact manner.
- They also explain the related runtime engine that can execute these algorithms efficiently on distributed and multi core systems. Additionally, these frameworks abstract the issues of the large-scale distributed system design.
- Therefore, they are capable of simplifying the design, application, and implementation of the new sophisticated graph algorithms to large-scale real-world graph problems.

Let's look at a few limitations of the Graph-parallel system.

Limitations of Graph-Parallel System

- Firstly, although the current graph-parallel system frameworks have various common properties, each of them presents different graph computations. These computations are custom-made for a specific graph applications and algorithms family or the original domain.
- Secondly, all these frameworks depend on different runtime. Therefore, it is tricky to create these programming abstractions.

- And finally, these frameworks cannot resolve the data Extract, Transform, Load or ETL issues and issues related to the process of deciphering and applying the computation results.
- The new graph-parallel system frameworks, however, have built-in support available for interactive graph computation.
- Spark GraphX is a graph computation system running in the framework of the data parallel system which focuses on distributing the data across different nodes that operate on the data in parallel.
  - ➢ It addresses the limitations posed by the graph parallel system.
  - ➢ Spark GraphX is more of a real-time processing framework for the data that can be represented in a graph form.
  - ➢ Spark GraphX extends the Resilient Distributed Dataset or RDD abstraction and hence, introduces a new feature called Resilient Distributed Graph or RDG.
  - ➢ Spark GraphX simplifies the graph ETL and analysis process substantially by providing new operations for viewing, filtering, and transforming graphs.

Features of Spark GraphX

Spark GraphX has many features like:

- Spark GraphX combines the benefits of graph parallel and data parallel systems as it efficiently expresses graph computations within the framework of the data parallel system.
- Spark GraphX distributes graphs efficiently as tabular data structures by leveraging new ideas in their representations.
  - ➢ It uses in-memory computation and fault tolerance by leveraging the improvements of the data flow systems.
  - ➢ Spark GraphX also simplifies the graph construction and transformation process by providing powerful new operations.
- With the use of these features, you can see that Spark is well suited for graph-parallel algorithm.

EXERCISE

1. Name few applications of Big Data.
2. What are the benefits of big data processing?
3. Explain the role of machine learning techniques in big data processing.
4. What is Mahout? Describe its features.
5. Name the applications of Mahout.
6. Explain the following Big Data Machine Learning algorithms in Mahout
   a. K-means clustering
   b. Naïve Bayes classification
7. State and explain the machine learning algorithms in Apache Spark.
8. Write short note on: MLlib
9. Write short note on : Deep Learning for Big Data
10. Describe any four applications of Deep Learning for Big Data.
11. Mention the key challenges of Deep Learning and Big Data.
12. What is graph processing? Mention the real time applications of graph processing.
13. Write short note on:
    a. Pregel
    b. Giraph
    c. Apache GraphX

❖ ❖ ❖

UNIT V

DATABASE FOR THE MODERN WEB

5.1 INTRODUCTION TO THE DATABASES FOR THE MODERN WEB

- If you've built web applications in recent years, you've probably used a relational database as the primary data store, and it probably performed acceptably. Most developers are familiar with SQL, and most of us can appreciate the beauty of a well-normalized data model, the necessity of transactions, and the assurances provided by a durable storage engine. And even if we don't like working with relational databases directly, a host of tools, from administrative consoles to object-relational mappers, helps alleviate any unwieldy complexity.
- Simply put, the relational database is mature and well known. So when a small but vocal cadre of developers starts advocating alternative data stores, questions about the viability and utility of these new technologies arise. Are these new data stores replacements for relational database systems? Who's using them in production, and why? What are the trade-offs involved in moving to a non relational database? The answers to those questions rest on the answer to this one: why are developers interested in MongoDB?
- It turns out that MongoDB is immediately attractive, not because of its scaling strategy, but rather because of its intuitive data model. Given that a document-based data model can represent rich, hierarchical data structures, it's often possible to do without the complicated multi-table joins imposed by relational databases. For example, suppose you're modeling products for an e-commerce site.
- With a fully normalized relational data model, the information for any one product might be divided among dozens of tables. If you want to get a product representation from the database shell, we'll need to write a complicated SQL query full of joins. As a consequence, most developers will need to rely on a secondary piece of software to assemble the data into something meaningful.
- With a document model, by contrast, most of a product's information can be represented within a single document. When you open the MongoDB JavaScript shell, you can easily get a comprehensible representation of your product with all its information hierarchically organized in a JSON-like structure. You can also query for it and manipulate it. MongoDB's query capabilities are designed specifically for manipulating structured documents, so users switching from relational databases experience a similar level of query power. In addition, most developers now work with object-oriented languages, and they want a data store that better maps to objects.
- With MongoDB, the object defined in the programming language can be persisted "as is," removing some of the complexity of object mappers. of a much more ambitious project. In mid-2007, a startup called 10gen began work on a software platform-as-a-service, composed of an application server and a database, that would host web applications and scale them as needed. Like Google's AppEngine, 10gen's platform was designed to handle the scaling and management of hardware and software infrastructure automatically, freeing developers to focus solely on their application code. 10gen ultimately discovered that most developers didn't feel comfortable giving up so much control over their technology stacks, but users *did* want 10gen's new database technology. This led 10gen to concentrate its efforts solely on the database that became MongoDB.
- With MongoDB's increasing adoption and production deployments large and small, 10gen continues to sponsor the database's development as an open source project. The code is publicly available and free to modify and use, subject to the terms of its license. And the community at large is encouraged to file bug reports and submit patches. Still, all of MongoDB's core developers are either founders or employees of 10gen, and the project's roadmap continues to be

determined by the needs of its user community and the overarching goal of creating a database that combines the best features of relational databases and distributed key-value stores.

- Thus, 10gen's business model is not unlike that of other well-known open source companies: support the development of an open source product and provide subscription services to end users.

5.2 MONGODB KEY FEATURES

- A database is defined in large part by its data model. In this section, we'll look at the document data model, and then we'll see the features of MongoDB that allow us to operate effectively on that model. We'll also look at operations, focusing on MongoDB's flavor of replication and on its strategy for scaling horizontally.

5.2.1 The Document Data Model

- MongoDB's data model is document-oriented. If you're not familiar with documents in the context of databases, the concept can be most easily demonstrated by an example. A JSON document needs double quotes everywhere except for numeric values. The following listing shows the JavaScript version of a JSON document where double quotes aren't necessary.

Listing 1.1 : A Document Representing an Entry on a Social News Site

```
{ _id: ObjectID('4bd9e8e17cefd644108961bb'),
  title: 'Adventures in Databases',
  url: 'http://example.com/databases.txt',
  author: 'msmith',
  vote_count: 20,
  tags: ['databases', 'mongodb', 'indexing'],
  image: {
    url: 'http://example.com/db.jpg',
    caption: '',
    type: 'jpg',
    size: 75381,
    data: "Binary"
  },
  comments: [
    { user: 'bjones',
      text: 'Interesting article!'
    },
    { user: 'blogger',
      text: 'Another related article is at http://example.com/db/db.txt'
    }
  ]
}
```

_id field is primary key

(1) Tags stored as array of strings

Attribute points to (2) another document

(3) Comments stored as array of comment objects

Fig. 5.1

- This listing shows a JSON document representing an article on a social news site (think Reddit or Twitter). As you can see, a document is essentially a set of property names and their values. The values can be simple data types, such as strings, numbers, and dates. But these values can also be arrays and even other JSON documents. These latter constructs permit documents to represent a variety of rich data structures. You'll see that the sample document has a property, tags 8, which stores the article's tags in an array. But even more interesting is the comments property d, which is an array of comment documents. Internally, MongoDB stores documents in a format called Binary JSON, or BSON. BSON has a similar structure but is intended for storing many documents.
- When you query MongoDB and get results back, these will be translated into an easy-to-read data structure. The MongoDB shell uses JavaScript and gets documents in JSON, which is what we'll use for most of our examples. We'll discuss the BSON format

extensively in later chapters. Where relational databases have tables, MongoDB has collections. In other words, MySQL (a popular relational database) keeps its data in tables of rows, while MongoDB keeps its data in collections of documents, which you can think of as a group of documents. Collections are an important concept in MongoDB.

- The data in a collection is stored to disk, and most queries require you to specify which collection you'd like to target. Let's take a moment to compare MongoDB collections to a standard relational database representation of the same data. Fig. 5.2 shows a likely relational analog. Because tables are essentially flat, representing the various one-to-many relationships in your post document requires multiple tables. You start with a posts table containing the core information for each post. Then you create three other tables, each of which includes a field, post_id, referencing the original post.
- The technique of separating an object's data into multiple tables like this is known as normalization. A normalized data set, among other things, ensures that each unit of data is represented in one place only. But strict normalization isn't without its costs. Notably, some assembly is required. To display the post you just referenced, you'll need to perform a join between the post and comments tables. Ultimately, the question of whether strict normalization is required depends on the kind of data you're modeling. What's important to note here is that a document-oriented data model naturally represents data in an aggregate form, allowing you to work with an object holistically: all the data representing a post, from comments to tags, can be fitted into a single database object.
- Fig. 5.2 A basic relational data model for entries on a social news site. The line terminator that looks like a cross represents a one-to-one relationship, so there's only one row from the images table associated with a row from the posts table. The line terminator that branches apart represents a one-to-many relationship, so there can be many rows in the comments table associated with a row from the posts table.

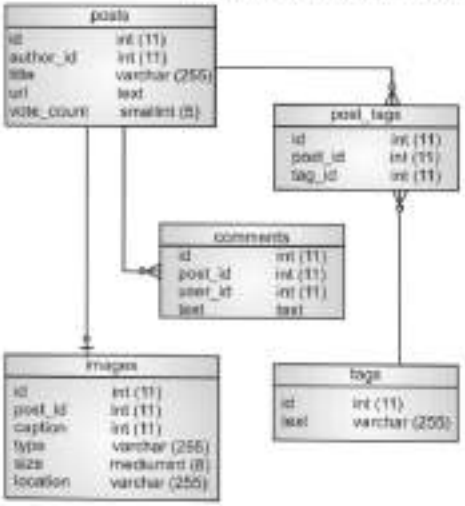


Fig. 5.2

- You've probably noticed that in addition to providing a richness of structure, documents needn't conform to a pre-specified schema. With a relational database, you store rows in a table. Each table has a strictly defined schema specifying which columns and types are permitted. If any row in a table needs an extra field, you have to alter the table explicitly.
- MongoDB groups documents into collections, containers that don't impose any sort of schema. In theory, each document in a collection can have a completely different structure; in practice, a collection's document will be relatively uniform. For instance, every document in the posts collection will have fields for the title, tags, comments, and so forth.
- **Schema-Less Model Advantages** : This lack of imposed schema confers some advantages. First, your application code, and not the database, enforces the data's structure. This can speed up initial application development when the schema is changing frequently. Second, and more significantly, a schema-less model allows you to represent data with truly variable properties. For example, imagine you're building an e-commerce product catalog.
- There's no way of knowing in advance what attributes a product will have, so the application will need to account for that variability. The traditional way of

handling this in a fixed-schema database is to use the entity-attribute-value pattern shown in Fig. 5.3.

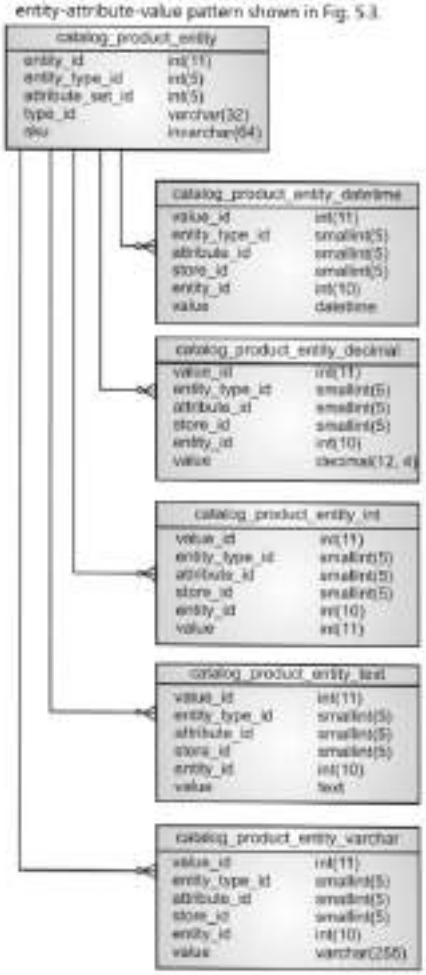


Fig. 5.3 : A portion of the schema for an e-commerce application. These tables facilitate dynamic attribute creation for products.

- What you're seeing is one section of the data model for an e-commerce framework. Note the series of tables that are all essentially the same, except for a single attribute, value, that varies only by data type. This structure allows an administrator to define additional product types and their attributes, but the result is significant complexity. Think about firing up the MySQL shell to examine or update a product modeled in this way; the SQL joins required to assemble the product would be enormously complex. Modeled as a document, no join is required, and new attributes can be added dynamically. Not all relational models are this complex, but the point is that when you're developing a MongoDB application you don't need to worry as much about what data fields you'll need in the future.

5.2.2 Ad Hoc Queries

- To say that a system supports ad hoc queries is to say that it isn't necessary to define in advance what sorts of queries the system will accept. Relational databases have this property; they'll faithfully execute any well-formed SQL query with any number of conditions. Ad hoc queries are easy to take for granted if the only databases you've ever used have been relational. But not all databases support dynamic queries. For instance, key-value stores are queryable on one axis only: the value's key. Like many other systems, key-value stores sacrifice rich query power in exchange for a simple scalability model.
- One of MongoDB's design goals is to preserve most of the query power that's been so fundamental to the relational database world. To see how MongoDB's query language works, let's take a simple example involving posts and comments. Suppose you want to find all posts tagged with the term politics having more than 10 votes. A SQL query would look like this:

```
SELECT * FROM posts
INNER JOIN posts_tags ON posts.id =
posts_tags.post_id
INNER JOIN tags ON posts_tags.tag_id == tags.id
WHERE tags.text = 'politics' AND posts.vote_count >
10;
```

- The equivalent query in MongoDB is specified using a document as a matcher. The special $gt key indicates the greater-than condition:

```
db.posts.find({'tags': 'politics', 'vote_count': {'$gt':
10}});
```

Note that the two queries assume a different data model. The SQL query relies on a strictly normalized model, where posts and tags are stored in distinct tables, whereas the

MongoDB query assumes that tags are stored within each post document. But both queries demonstrate an ability to query on arbitrary combinations of attributes, which is the essence of ad hoc query ability.

5.2.3 Indexes

- A critical element of ad hoc queries is that they search for values that you don't know when you create the database. As you add more and more documents to your database, searching for a value becomes increasingly expensive; it's a needle in an ever-expanding haystack. Thus, you need a way to efficiently search through your data.
- The solution to this is an index. The best way to understand database indexes is by analogy: many books have indexes matching keywords to page numbers. Suppose you have a cookbook and want to find all recipes calling for pears (maybe you have a lot of pears and don't want them to go bad).
- The time-consuming approach would be to page through every recipe, checking each ingredient list for pears. Most people would prefer to check the book's index for the pears entry, which would give a list of all the recipes containing pears. Database indexes are data structures that provide this same service. Indexes in MongoDB are implemented as a B-tree data structure. B-tree indexes, also used in many relational databases, are optimized for a variety of queries, including range scans and queries with sort clauses. But WiredTiger has support for log structured merge-trees (LSM) that's expected to be available in the MongoDB 3.2 production release. Most databases give each document or row a primary key, a unique identifier for that datum.
- The primary key is generally indexed automatically so that each datum can be efficiently accessed using its unique key, and MongoDB is no different. But not every database allows you to also index the data inside that row or document. These are called secondary indexes. Many NoSQL databases, such as Hbase, are considered keyvalue stores because they don't allow any secondary indexes.
- This is a significant feature in MongoDB; by permitting multiple secondary indexes MongoDB allows users to optimize for a wide variety of queries. With MongoDB, you can create up to 64 indexes per collection. The kinds of indexes supported include all the ones you'd find in an RDMBS: ascending, descending, unique, compound-key, hashed, text, and even geospatial indexes are supported. Because MongoDB and most RDBMSs use the same data structure for their indexes, advice for managing indexes in both of these systems is similar.

5.2.4 Replication

- MongoDB provides database replication via a topology known as a replica set. Replica sets distribute data across two or more machines for redundancy and automate failover in the event of server and network outages. Additionally, replication is used to scale database reads. If you have a read-intensive application, as is commonly the case on the web, it's possible to spread database reads across machines in the replica set cluster.
- Replica sets consist of many MongoDB servers, usually with each server on a separate physical machine; we'll call these nodes. At any given time, one node serves as the replica set primary node and one or more nodes serve as secondary's. Like the master-slave replication that you may be familiar with from other databases, a replica set's primary node can accept both reads and writes, but the secondary nodes are read only. What makes replica sets unique is their support for automated failover: if the primary node fails, the cluster will pick a secondary node and automatically promote it to primary. When the former primary comes back online, it'll do so as a secondary. An illustration of this process is provided in Fig. 5.4.

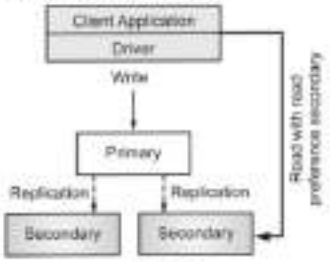


Fig. 5.4

5.3 REPLICATION

5.3.1 Speed and Durability

- To understand MongoDB's approach to durability, it pays to consider a few ideas first. In the realm of database systems there exists an inverse relationship between write speed and durability. Write speed can be understood as the volume of inserts, updates, and deletes that a database can process in a given time frame.
- Durability refers to level of assurance that these write operations have been made permanent. For instance, suppose you write 100 records of 50 KB each to a database and then immediately cut the power on the server. Will those records be recoverable when you bring the machine back online? The answer depends on your database system, its configuration, and the hardware hosting it. Most databases enable good durability by default, so you're safe if this happens. For some applications, like storing log lines, it might make more sense to have faster writes, even if you risk data loss.
- The problem is that writing to a magnetic hard drive is orders of magnitude slower than writing to RAM. Certain databases, such as Memcached, write exclusively to RAM, which makes them extremely fast but completely volatile. On the other hand, few databases write exclusively to disk because the low performance of such an operation is unacceptable.
- Therefore, database designers often need to make compromises to provide the best balance of speed and durability. In MongoDB's case, users control the speed and durability trade-off by choosing write semantics and deciding whether to enable journaling. Journaling is enabled by default since MongoDB v2.0. In the drivers released after November 2012 MongoDB safely guarantees that a write has been written to RAM before returning to the user, though this characteristic is configurable.
- You can configure MongoDB to fire-and-forget, sending off a write to the server without waiting for an acknowledgment. You can also configure MongoDB to guarantee that a write has gone to multiple replicas before considering it committed. For high-volume, low-value data (like click streams and logs), fire-and-forget-style writes can be ideal. For important data, a safe mode setting is necessary. It's important to know that in MongoDB versions older than 2.0, the unsafe fire-and-forget strategy was set as the default, because when 10gen started the development of MongoDB, it was focusing solely on that data tier and it was believed that the application tier would handle such errors. But as MongoDB was used for more and more use cases and not solely for the web tier, it was deemed that it was too unsafe for any data you didn't want to lose.
- Since MongoDB v2.0, journaling is enabled by default. With journaling, every write is flushed to the journal file every 100 ms. If the server is ever shut down uncleanly (say, in a power outage), the journal will be used to ensure that MongoDB's data files are restored to a consistent state when you restart the server. This is the safest way to run MongoDB. It's possible to run the server without journaling as a way of increasing performance for some write loads.
- The downside is that the data files may be corrupted after an unclean shutdown. As a consequence, anyone planning to disable journaling should run with replication, preferably to a second datacenter, to increase the likelihood that a pristine copy of the data will still exist even if there's a failure. MongoDB was designed to give you options in the speed-durability tradeoff, but we highly recommend safe settings for essential data.

5.3.2 Scaling

- The easiest way to scale most databases is to upgrade the hardware. If your application is running on a single node, it's usually possible to add some combination of faster disks, more memory, and a beefier CPU to ease any database bottlenecks.
- The technique of augmenting a single node's hardware for scale is known as vertical scaling, or scaling up. Vertical scaling has the advantages of being simple, reliable, and cost-effective up to a certain point, but eventually you reach a point where it's no longer feasible to move to a better machine. It then makes sense to consider scaling horizontally, or scaling out (see Fig. 5.5). Instead of beefing up a single node, its better scaling horizontally means distributing the database across multiple machines.

- A horizontally scaled architecture can run on many smaller, less expensive machines, often reducing your hosting costs. What's more, the distribution of data across machines mitigates the consequences of failure. Machines will unavoidably fail from time to time. If you've scaled vertically and the machine fails, then you need to deal with the failure of a machine on which most of your system depends.

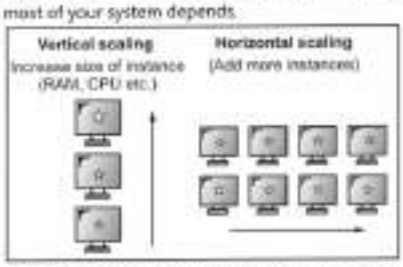


Fig. 5.5 : MongoDB: Horizontal and Vertical Scaling

- This may not be an issue if a copy of the data exists on a replicated slave, but it's still the case that only a single server need fail to bring down the entire system. Contrast that with failure inside a horizontally scaled architecture. This may be less catastrophic because a single machine represents a much smaller percentage of the system as a whole. MongoDB was designed to make horizontal scaling manageable. It does so via a range-based partitioning mechanism, known as sharding, which automatically manages the distribution of data across nodes.
- There's also a hash- and tag-based sharding mechanism, but it's just another form of the range-based sharding mechanism. The sharding system handles the addition of shard nodes, and it also facilitates automatic failover. Individual shards are made up of a replica set consisting of at least two nodes, ensuring automatic recovery with no single point of failure. All this means that no application code has to handle these logistics; your application code communicates with a sharded cluster just as it speaks to a single node.

5.4 MONGODB'S CORE SERVER AND TOOLS

- MongoDB is written in C++ and actively developed by MongoDB, Inc. The project compiles on all major operating systems, including Mac OS X, Windows, Solaris, and most flavors of Linux. Precompiled binaries are available for each of these platforms at http://mongodb.org. MongoDB is open source and licensed under the GNU-Affero General Public License (AGPL).
- The source code is freely available on GitHub, and contributions from the community are frequently accepted. But the project is guided by the MongoDB, Inc. core server team, and the overwhelming majority of commits come from this group. MongoDB v1.0 was released in November 2009. Major releases appear approximately once every three months, with even point numbers for stable branches and odd numbers for development. As of this writing, the latest stable release is v3.0. What follows is an overview of the components that ship with MongoDB along with a high-level description of the tools and language drivers for developing applications with the database.

5.4.1 Core Server

- The core database server runs via an executable called mongod (mongodb.exe on Windows). The mongod server process receives commands over a network socket using a custom binary protocol. All the data files for a mongod process are stored by default in /data/db on Unix-like systems and in c:\data\db on Windows. Some of the examples in this text may be more Linux-oriented.
- Most of our MongoDB production servers are run on Linux because of its reliability, wide adoption, and excellent tools. mongod can be run in several modes, such as a standalone server or a member of a replica set. Replication is recommended when you're running MongoDB in production, and you generally see replica set configurations consisting of two replicas plus a mongod running in arbiter mode. When you use MongoDB's sharding feature, you'll also run mongod in config server mode.
- Finally, a separate routing server exists called mongos, which is used to send requests to the appropriate shard in this kind of setup. Configuring a mongod process is relatively simple; it can be accomplished both with command-line arguments and with a text configuration file. Some common configurations to change are setting the port that mongod listens on and setting the directory where it stores its data. To see these configurations, you can run mongod --help.

5.4.2 JavaScript Shell

- The MongoDB command shell is a JavaScript -based tool for administering the database and manipulating data. The mongo executable loads the shell and connects to a specified mongod process, or one running locally by default. The shell was developed to be similar to the MySQL shell; the biggest differences are that it's based on JavaScript and SQL isn't used. For instance, you can pick your database and then insert a simple document into the users collection like this:

```
> use my_database
> db.users.insert({name: "Kyle"})
```

- The first command, indicating which database you want to use, will be familiar to users of MySQL. The second command is a JavaScript expression that inserts a simple document. To see the results of your insert, you can issue a simple query:

```
> db.users.find()
{ _id: ObjectId("4ba667b0a90578631c9caea0"), name:
"Kyle" }
```

- The find method returns the inserted document, with an object ID added. All documents require a primary key stored in the _id field. You're allowed to enter a custom _id as long as you can guarantee its uniqueness. But if you omit the _id altogether, a MongoDB object ID will be inserted automatically. In addition to allowing you to insert and query for data, the shell permits you to run administrative commands.
- Some examples include viewing the current database operation, checking the status of replication to a secondary node, and configuring a collection for sharding. As you'll see, the MongoDB shell is indeed a powerful tool that's worth getting to know well. All that said, the bulk of your work with MongoDB will be done through an application written in a given programming language. To see how that's done, we must say a few things about MongoDB's language drivers.

5.4.3 Database Drivers

- If the notion of a database driver conjures up nightmares of low-level device hacking, don't fret; the MongoDB drivers are easy to use. The driver is the code used in an application to communicate with a MongoDB server. All drivers have functionality to query, retrieve results, write data, and run database commands. Every effort has been made to provide an API that matches the idioms of the given language while also maintaining relatively uniform interfaces across languages.
- For instance, all of the drivers implement similar methods for saving a document to a collection, but the representation of the document itself will usually be whatever is most natural to each language. In Ruby, that means using a Ruby hash. In Python, a dictionary is appropriate. And in Java, which lacks any analogous language primitive, you usually represent documents as a Map object or something similar. Some developers like using an object relational mapper to help manage representing their data this way, but in practice, the MongoDB drivers are complete enough that this isn't required.

5.4.4 Command-Line Tools

MongoDB is bundled with several command-line utilities:

- **Mongodump and Mongorestore** : Standard utilities for backing up and restoring a database. mongodump saves the database's data in its native BSON format and thus is best used for backups only; this tool has the advantage of being usable for hot backups, which can easily be restored with mongorestore.
- **Mongoexport and Mongoimport** : Export and import JSON, CSV, and TSV7 data; this is useful if you need your data in widely supported formats. mongoimport can also be good for initial imports of large data sets, although before importing, it's often desirable to adjust the data model to take best advantage of MongoDB. In such cases, it's easier to import the data through one of the drivers using a custom script.
- **Mongosniff** : A wire-sniffing tool for viewing operations sent to the database. It essentially translates the BSON going over the wire to human-readable shell statements.
- **Mongostat** : Similar to iostat, this utility constantly polls MongoDB and the system to provide helpful stats, including the number of operations per second (inserts, queries, updates, deletes, and so on), the amount of virtual memory allocated, and the number of connections to the server.

- **Mongotop :** Similar to top, this utility polls MongoDB and shows the amount of time it spends reading and writing data in each collection.
- **Mongoperf :** Helps you understand the disk operations happening in a running MongoDB instance.
- **Mongooplog :** Shows what's happening in the MongoDB oplog.
- **Bsondump :** Converts BSON files into human-readable formats including JSON.

5.5 MONGODB THROUGH THE JAVASCRIPT'S SHELL

- MongoDB's JavaScript shell makes it easy to play with data and get a tangible sense of documents, collections, and the database's particular query language. Think of the following walkthrough as a practical introduction to MongoDB. You'll begin by getting the shell up and running. Then you'll see how JavaScript represents documents, and you'll learn how to insert these documents into a MongoDB collection. To verify these inserts, you'll practice querying the collection. Then it's on to updates. Finally, we'll finish out the CRUD operations by learning to remove data and drop collections.

5.5.1 Starting The Shell

- Follow the instructions in appendix A and you should quickly have a working MongoDB installation on your computer, as well as a running mongod instance. Once you do, start the MongoDB shell by running the mongo executable: mongo If the shell program starts successfully, your screen will look like Fig. 5.6. The shell heading displays the version of MongoDB you're running, along with some additional information about the currently selected database.

```
10:25 $ mongo
MongoDB shell version: 3.0.4
connecting to: test
>
```

Fig. 5.6

- If you know some JavaScript, you can start entering code and exploring the shell right away. In either case, read on to see how to run your first operations against MongoDB.

5.5.2 Databases, Collections, and Documents

- As you probably know by now, MongoDB stores its information in documents, which can be printed out in JSON (JavaScript Object Notation) format. You'd probably like to store different types of documents, like users and orders, in separate places.
- This means that MongoDB needs a way to group documents, similar to a table in an RDBMS. In MongoDB, this is called a collection. MongoDB divides collections into separate databases. Unlike the usual overhead that databases produce in the SQL world, databases in MongoDB are just namespaces to distinguish between collections. To query MongoDB, you'll need to know the database (or namespace) and collection you want to query for documents. If no other database is specified on startup, the shell selects a default database called test. As a way of keeping all the subsequent tutorial exercises under the same namespace, let's start by switching to the tutorial database:

```
> use tutorial
Switched to db tutorial
```

- You'll see a message verifying that you've switched databases. Why does MongoDB have both databases and collections? The answer lies in how MongoDB writes its data out to disk. All collections in a database are grouped in the same files, so it makes sense, from a memory perspective, to keep related collections in the same database. You might also want to have different applications access the same collections (multitenancy) and, it's also useful to keep your data organized so you're prepared for future requirements.
- It's time to create your first document. Because you're using a JavaScript shell, your documents will be specified in JSON. For instance, a simple document describing a user might look like this: {username: "smith"}
- The document contains a single key and value for storing Smith's username.

5.5.3 Inserts and Queries

- To save this document, you need to choose a collection to save it to. Appropriately enough, you'll save it to the users collection. Here's how:

```
> db.users.insert({username: "smith"})
WriteResult({ "nInserted" : 1 })
```

Note: In our examples, we'll preface MongoDB shell commands with a > so that you can tell the difference between the command and its output. You may notice a slight delay after entering this code. At this point, neither the tutorial database nor the users collection has been created on disk. The delay is caused by the allocation of the initial data files for both. If the insert succeeds, you've just saved your first document. In the default MongoDB configuration, this data is now guaranteed to be inserted even if you kill the shell or suddenly restart your machine. You can issue a query to see the new document:

```
> db.users.find()
```

Since the data is now part of the users collection, reopening the shell and running the query will show the same result. The response will look something like this:

```
{ "_id" : ObjectId("552e458158cd52bcb257c324"),
"username" : "smith" } _ID FIELDS IN MONGODB
```

Note that an _id field has been added to the document. You can think of the _id value as the document's primary key. Every MongoDB document requires an _id, and if one isn't present when the document is created, a special MongoDB ObjectID will be generated and added to the document at that time. The ObjectID that appears in your console won't be the same as the one in the code listing, but it will be unique among all _id values in the collection, which is the only requirement for the field. You can set your own _id by setting it in the document you insert, the ObjectID is just MongoDB's default. We'll have more to say about ObjectIDs in the next chapter. Let's continue for now by adding a second user to the collection:

```
> db.users.insert({username: "jones"})
WriteResult({ "nInserted" : 1 })
```

- There should now be two documents in the collection. Go ahead and verify this by running the count command:

```
>db.users.count()
```

- Pass a query predicate Now that you have more than one document in the collection, let's look at some slightly more sophisticated queries. As before, you can still query for all the documents in the collection:

```
> db.users.find()
{ "_id" : ObjectId("552e458158cd52bcb257c324"),
"username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"),
"username" : "jones" }
```

- You can also pass a simple query selector to the find method. A query selector is a document that's used to match against all documents in the collection. To query for all documents where the username is jones, you pass a simple document that acts as your query selector like this:

```
> db.users.find({username: "jones"})
{ "_id" : ObjectId("552e542a58cd52bcb257c325"),
"username" : "jones" }
```

- The query predicate {username: "jones"} returns all documents where the username is jones it literally matches against the existing documents. Note that calling the find method without any argument is equivalent to passing in an empty predicate; db.users.find() is the same as db.users.find({}). You can also specify multiple fields in the query predicate, which creates an implicit AND among the fields. For example, you query with the following selector:

```
> db.users.find({ ... _id:
ObjectId("552e458158cd52bcb257c324"), ...
username: "smith" ... })
{ "_id" : ObjectId("552e458158cd52bcb257c324"),
"username" : "smith" }
```

- The three dots after the first line of the query are added by the MongoDB shell to indicate that the command takes more than one line. The query predicate is identical to the returned document. The predicate ANDs the fields, so this query searches for a document that matches on both the _id and username fields. You can also use MongoDB's $and operator explicitly. The previous query is identical to

```
>db.users.find({ $and: [ ... { _id:
ObjectId("552e458158cd52bcb257c324") }, ... {
username: "smith" } ... ] })
{"_id" : ObjectId("552e458158cd52bcb257c324"),
"username" : "smith" }
```

- Selecting documents with an OR is similar: just use the $or operator. Consider the following query:

```
> db.users.find({ $or: [ ... { username: "smith" },
... { username: "jones" } ... ]})
{ "_id" : ObjectId("552e458158cd52bcb257c324"),
"username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"),
"username" : "jones" }
```

- The query returns both the smith and jones documents, because we asked for either a username of smith or a username of jones. This example is different than previous ones, because it doesn't just insert or search for a specific document. Rather, the query itself is a document. The idea of representing commands as documents is used often in MongoDB and may come as a surprise if you're used to relational databases. One advantage of this interface is that it's easier to build queries programmatically in your application because they're documents rather than a long SQL string. We've presented the basics of creating and reading data. Now it's time to look at how to update that data.

5.5.4 Updating Documents

- All updates require at least two arguments. The first specifies which documents to update, and the second defines how the selected documents should be modified. The first few examples demonstrate modifying a single document, but the same operations can be applied to many documents, even an entire collection, as we show at the end of this section. But keep in mind that by default the update() method updates a single document. There are two general types of updates, with different properties and use cases. One type of update involves applying modification operations to a document or documents, and the other type involves replacing the old document with a new one. For the following examples, we'll look at this sample document:

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"),
"username" : "smith" }
```

Operator Update

- The first type of update involves passing a document with some kind of operator description as the second argument to the update function. In this section, you'll see an example of how to use the $set operator, which sets a single field to the specified value. Suppose that user Smith decides to add her country of residence. You can record this with the following update:

```
> db.users.update({username: "smith"},
{$set: {country: "Canada"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0,
"nModified" : 1 })
```

- This update tells MongoDB to find a document where the username is smith, and then to set the value of the country property to Canada. You see the change reflected in the message that gets sent back by the server. If you now issue a query, you'll see that the document has been updated accordingly:

```
>db.users.find({username: "smith"}) { "_id" :
ObjectId("552e458158cd52bcb257c324"), "username"
: "smith", "country" : "Canada" }
```

Replacement Update

- Another way to update a document is to replace it rather than just set a field. This is sometimes mistakenly used when an operator update with a $set was intended. Consider a slightly different update command:

```
>db.users.update({username: "smith"}, {country:
"Canada"}) WriteResult({ "nMatched" : 1, "nUpserted" :
0, "nModified" : 1 })
```

- In this case, the document is replaced with one that only contains the country field, and the username field is removed because the first document is used only for matching and the second document is used for replacing the document that was previously matched. You should be careful when you use this kind of update. A query for the document yields the following:

```
> db.users.find({country: "Canada"}) { "_id" :
ObjectId("552e458158cd52bcb257c324"), "country" :
"Canada" }
```

- The _id is the same, yet data has been replaced in the update. Be sure to use the $set operator if you intend to add or set fields rather than to replace the entire document. Add the username back to the record:

```
> db.users.update({country: "Canada"},
{$set: {username: "smith"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0,
"nModified" : 1 })
> db.users.find({country: "Canada"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"),
"country" : "Canada", "username" : "smith" }
```

- If you later decide that the country stored in the profile is no longer needed, the value can be removed as easily using the $unset operator:

```
> db.users.update({username: "smith"}, {$unset:
{country: 1}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0,
"nModified" : 1 })
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"),
"username" : "smith" }
```

Updating Complex Data

- Let's suppose that, in addition to storing profile information, your users can store lists of their favorite things. A good document representation might look something like this:

```
{ username: "smith",
favorites: {
cities: ["Chicago", "Cheyenne"],
movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
}
}
```

- The favorites key points to an object containing two other keys, which point to lists of favorite cities and movies. Given what you know already, can you think of a way to modify the original smith document to look like this? The $set operator should come to mind:

```
> db.users.update( {username: "smith"},
... {
... $set: {
... favorites: {
... cities: ["Chicago", "Cheyenne"],
... movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
... }
... }
... })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

- Please note that the use of spacing for indenting isn't mandatory, but it helps avoid errors as the document is more readable this way. Let's modify jones similarly, but in this case you'll only add a couple of favorite movies:

```
> db.users.update( {username: "jones"},
... {
... $set: {
... favorites: {
... movies: ["Casablanca", "Rocky"]
... }
... }
... })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

- If you make a typo, you can use the up arrow key to recall the last shell statement. Now query the users collection to make sure that both updates succeeded:

```
> > db.users.find().pretty()
{ "_id" : ObjectId("552e458158cd52bcb257c324"),
"username" : "smith",
"favorites" :
{ "cities" : [ "Chicago", "Cheyenne" ],
"movies" : [ "Casablanca", "For a Few Dollars More",
"The Sting" ]
}
}
{ "_id" : ObjectId("552e542a58cd52bcb257c325"),
"username" : "jones",
"favorites" : { "movies" : [ "Casablanca", "Rocky" ]
}
}
```

- Strictly speaking, the find() command returns a cursor to the returning documents. Therefore, to access the documents you'll need to iterate the cursor. The find() command automatically returns 20 documents if they're available after iterating the cursor 20 times. With a couple of example documents at your fingertips, you can now begin to see the power of MongoDB's query language. In particular, the query engine's ability to reach into nested inner objects and match against array elements proves useful in this situation.
- Notice how we appended the pretty operation to the find operation to get nicely formatted results returned by the server. Strictly speaking, pretty() is actually cursor.pretty(), which is a way of configuring a cursor to display results in an easy-to-read format. You can see an example of both of these concepts demonstrated in this query to find all users who like the movie Casablanca:

```
> db.users.find({"favorites.movies": "Casablanca"})
```

- The dot between favorites and movies instructs the query engine to look for a key named favorites that points to an object with an inner key named movies and then to match the value of the inner key. Thus, this query will return both user documents because queries on arrays will match if any element in the array matches the original query. To see a more involved

example, suppose you know that any user who likes Casablanca also likes The Maltese Falcon and that you want to update your database to reflect this fact. How would you represent this as a MongoDB update?

More Advanced Updates

- You could conceivably use the $set operator again, but doing so would require you to rewrite and send the entire array of movies. Because all you want to do is to add an element to the list, you're better off using either $push or $addToSet. Both operators add an item to an array, but the second does so uniquely, preventing a duplicate addition. This is the update you're looking for:

```
> db.users.update( {"favorites.movies": "Casablanca"},
... {$addToSet: {"favorites.movies": "The Maltese
Falcon"} },
... false,
... true )
WriteResult({ "nMatched" : 2, "nUpserted" : 0,
"nModified" : 2 })
```

- Most of this should be decipherable by now. The first argument is a query predicate that matches against users who have Casablanca in their movies list. The second argument adds The Maltese Falcon to that list using the $addToSet operator. The third argument, false, controls whether an update is allowed. This tells the update operation whether it should insert a document if it doesn't already exist, which has different behavior depending on whether the update is an operator update or a replacement update. The fourth argument, true, indicates that this is a multi-update. By default, a MongoDB update operation will apply only to the first document matched by the query selector. If you want the operation to apply to all documents matched, you must be explicit about that. You want your update to apply to both smith and jones, so the multi-update is necessary. We'll cover updates in more detail later, but try these examples before moving on.

5.5.5 Deleting Data

- Now you know the basics of creating, reading, and updating data through the MongoDB shell. We've saved the simplest operation, removing data, for last. If given no parameters, a remove operation will clear a collection of all its documents. To get rid of, say, a foo collection's contents, you enter:

```
> db.foo.remove()
```

- You often need to remove only a certain subset of a collection's documents, and for that, you can pass a query selector to the remove() method. If you want to remove all users whose favorite city is Cheyenne, the expression is straightforward:

```
> db.users.remove({"favorites.cities": "Cheyenne"})
WriteResult({ "nRemoved" : 1 })
```

Note that the remove() operation doesn't actually delete the collection; it merely removes documents from a collection. You can think of it as being analogous to SQL's DELETE command. If your intent is to delete the collection along with all of its indexes, use the drop() method:

```
> db.users.drop()
```

- Creating, reading, updating, and deleting are the basic operations of any database. If you've followed along, you should be in a position to continue practicing basic CRUD operations in MongoDB. In the next section, you'll learn how to enhance your queries, updates, and deletes by taking a brief look at secondary indexes.

5.5.6 Other Shell Features

- You may have noticed this already, but the shell does a lot of things to make working with MongoDB easier. You can revisit earlier commands by using the up and down arrows, and use autocomplete for certain inputs, like collection names. The autocomplete feature uses the tab key to autocomplete or to list the completion possibilities.
- You can also discover more information in the shell by typing this: > help A lot of functions print pretty help messages that explain them as well. Try it out:

```
> db.help()
```

DB Methods:

```
db.adminCommand(nameOrDocument) - switches to
'admin' db, and runs
command [ just calls db.runCommand(...) ]
db.auth(username, password)
db.cloneDatabase(fromhost)
db.commandHelp(name) returns the help for the
command
db.copyDatabase(fromdb, todb, fromhost)
```

- Help on queries is provided through a different function called explain, which we'll investigate in later sections. There are also a number of options you can

use when starting the MongoDB shell. To display a list of these, add the help flag when you start the MongoDB shell:

```
$ mongo --help
```

- You don't need to worry about all these features, and we're not done working with the shell yet, but it's worth knowing where you can find more information when you need it.

5.6 CREATING AND QUERYING WITH INDEXES

- It's common to create indexes to enhance query performance. Fortunately, MongoDB's indexes can be created easily from the shell. If you're new to database indexes, this section should make the need for them clear; if you already have indexing experience, you'll see how easy it is to create indexes and then profile queries against them using the explain() method.

5.6.1 Creating a Large Collection

- An indexing example makes sense only if you have a collection with many documents. So you'll add 20,000 simple documents to a numbers collection. Because the MongoDB shell is also a JavaScript interpreter, the code to accomplish this is simple:

```
> for(i = 0; i < 20000; i++)
{
db.numbers.save({num: i});
}
WriteResult({ "nInserted" : 1 })
```

- That's a lot of documents, so don't be surprised if the insert takes a few seconds to complete. Once it returns, you can run a couple of queries to verify that all the documents are present:

```
> db.numbers.count()
20000
> db.numbers.find()
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830a"),
"num": 0 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830b"),
"num": 1 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830c"),
"num": 2 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830d"),
"num": 3 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830e"),
"num": 4 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830f"),
"num": 5 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8310"),
"num": 6 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8311"),
"num": 7 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8312"),
"num": 8 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8313"),
"num": 9 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8314"),
"num": 10 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8315"),
"num": 11 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8316"),
"num": 12 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8317"),
"num": 13 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8318"),
"num": 14 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8319"),
"num": 15 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831a"),
"num": 16 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831b"),
"num": 17 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831c"),
"num": 18 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831d"),
"num": 19 }
Type "it" for more
```

- The count() command shows that you've inserted 20,000 documents. The subsequent query displays the first 20 results (this number may be different in your shell). You can display additional results with the it command:

```
> it
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831e"),
"num": 20 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831f"),
"num": 21 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8320"),
"num": 22 } ...
```

- The it command instructs the shell to return the next result set.2 With a sizable set of documents available, let's try a couple queries. Given what you know about MongoDB's query engine, a simple query matching a document on its num attribute makes sense:

```
> db.numbers.find({num: 500})
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac84fe"),
"num" : 500 }
```

Range Queries

- More interestingly, you can also issue range queries using the special $gt and $lt operators. They stand for greater than and less than, respectively. Here's how you query for all documents with a num value greater than 199,995:

```
> db.numbers.find( {num: {"$gt": 19995 }} )
{ "_id" : ObjectId("552e660b58cd52bcb2581142"),
"num" : 19996 }
{ "_id" : ObjectId("552e660b58cd52bcb2581143"),
"num" : 19997 }
{ "_id" : ObjectId("552e660b58cd52bcb2581144"),
"num" : 19998 }
{ "_id" : ObjectId("552e660b58cd52bcb2581145"),
"num" : 19999 }
```

- You can also combine the two operators to specify upper and lower boundaries:

```
> db.numbers.find( {num: {"$gt": 20, "$lt": 25 }} )
{ "_id" : ObjectId("552e660558cd52bcb257c33b"),
"num" : 21 }
{ "_id" : ObjectId("552e660558cd52bcb257c33c"),
"num" : 22 }
{ "_id" : ObjectId("552e660558cd52bcb257c33d"),
"num" : 23 }
{ "_id" : ObjectId("552e660558cd52bcb257c33e"),
"num" : 24 }
```

- You can see that by using a simple JSON document, you're able to specify a range query in much the same way you might in SQL. $gt and $lt are only two of a host of operators that comprise the MongoDB query language. Others include $gte for greater than or equal to, $lte for (you guessed it) less than or equal to, and $ne for not equal to. You'll see other operators and many more example queries in later chapters. Of course, queries like this are of little value unless they're also efficient.

5.6.2 Indexing and explain( )

- If you've spent time working with relational databases, you're probably familiar with SQL's EXPLAIN, an invaluable tool for debugging or optimizing a query. When any database receives a query, it must plan out how to execute it; this is called a query plan. EXPLAIN describes query paths and allows developers to diagnose slow operations by determining which indexes a query has used. Often a query can be executed in multiple ways, and sometimes this results in behavior you might not expect. EXPLAIN explains. MongoDB has its own version of EXPLAIN that provides the same service. To get an idea of how it works, let's apply it to one of the queries you just issued. Try running the following on your system:

```
> db.numbers.find({num: {"$gt":
19995}}).explain("executionStats")
```

- What this collection needs is an index. You can create an index for the num key within the documents using the createIndex() method. Try entering the following index creation code:

```
> db.numbers.createIndex({num: 1})
{ "createdCollectionAutomatically" : false,
"numIndexesBefore" : 1, "numIndexesAfter" : 2, "ok" :
1 }
```

- The createIndex() method replaces the ensureIndex() method in MongoDB 3.
- If you're using an older MongoDB version, you should use ensureIndex() instead of createIndex(). In MongoDB 3, ensureIndex() is still valid as it's an alias for createIndex(), but you should stop using it. As with other MongoDB operations, such as queries and updates, you pass a document to the createIndex() method to define the index's keys. In this case, the {num: 1} document indicates that an ascending index should be built on the num key for all documents in the numbers collection. You can verify that the index has been created by calling the getIndexes() method:

```
> db.numbers.getIndexes() [ { "v" : 1, "key" : { "_id" : 1 },
"name" : "_id_", "ns" : "tutorial.numbers" },
{ "v" : 1, "key" : { "num" : 1 }, "name" : "num_1", "ns" :
"tutorial.numbers" } ]
```

- The collection now has two indexes. The first is the standard _id index that's automatically built for every

collection; the second is the index you created on num. The indexes for those fields are called _id_ and num_1, respectively. If you don't provide a name, MongoDB sets hopefully meaningful names automatically. Indexes don't come free: they take up some space and can make your inserts slightly more expensive, but they're an essential tool for query optimization. If this example intrigues you, be sure to check out chapter 8, which is devoted to indexing and query optimization. Next you'll look at the basic administrative commands required to get information about your MongoDB instance. You'll also learn techniques for getting help from the shell, which will aid in mastering the various shell commands.

5.7 PRINCIPLES OF SCHEMA DESIGN

- Database schema design is the process of choosing the best representation for a data set, given the features of the database system, the nature of the data, and the application requirements. The principles of schema design for relational database systems are well established. With RDBMSs, you're encouraged to shoot for a normalized data model, which helps to ensure generic query ability and avoid updates to data that might result in inconsistencies.
- Moreover, the established patterns prevent developers from wondering how to model, say, one-to-many and many-to-many relationships. But schema design is never an exact science, even with relational databases. Application functionality and performance is the ultimate master in schema design, so every "rule" has exceptions. If you're coming from the RDBMS world, you may be troubled by MongoDB's lack of hard schema design rules.
- Good practices have emerged, but there's still usually more than one good way to model a given data set. The premise of this section is that principles can drive schema design, but the reality is that those principles are pliable. To get you thinking, here are a few questions you can bring to the table when modeling data with any database system:

What are your Application Access Patterns?

- You need to pin down the needs of your application, and this should inform not only your schema design but also which database you choose. Remember, MongoDB isn't right for every application. Understanding your application access patterns is by far the most important aspect of schema design. The idiosyncrasies of an application can easily demand a schema that goes against firmly held data modeling principles. The upshot is that you must ask numerous questions about the application before you can determine the ideal data model. What's the read/write ratio? Will queries be simple, such as looking up a key, or more complex? Will aggregations be necessary? How much data will be stored?

What's The Basic Unit of Data?

- In an RDBMS, you have tables with columns and rows. In a key-value store, you have keys pointing to amorphous values. In MongoDB, the basic unit of data is the BSON document.
- What are the capabilities of your database? Once you understand the basic data type, you need to know how to manipulate it. RDBMSs feature ad hoc queries and joins, usually written in SQL, while simple key-value stores permit fetching values only by a single key. MongoDB also allows ad hoc queries, but joins aren't supported.
- Databases also diverge in the kinds of updates they permit. With an RDBMS, you can update records in sophisticated ways using SQL and wrap multiple updates in a transaction to get atomicity and rollback. MongoDB doesn't support transactions in the traditional sense, but it does support a variety of atomic update operations that can work on the internal structures of a complex document. With simple key-value stores, you might be able to update a value, but every update will usually mean replacing the value completely.

What Makes a Good Unique id or Primary Key for a Record?

- There are exceptions, but many schemas, regardless of the database system, have some unique key for each record. Choosing this key carefully can make a big difference in how you access your data and how it's stored. If you're designing a user's collection, for example, should you use an arbitrary value, a legal name, a username, or a social security number as the primary key? It turns out that neither legal names nor social security numbers are unique or even applicable to all users within a given dataset.

- In MongoDB choosing a primary key means picking what should go in the _id field. The automatic object ids are good defaults, but not ideal in every case. This is particularly important if you shard your data across multiple machines because it determines where a certain record will go.
- The best schema designs are always the product of deep knowledge of the database you're using, good judgment about the requirements of the application at hand, and plain old experience. A good schema often requires experimentation and iteration, such as when an application scales and performance considerations change. Don't be afraid to alter your schema when you learn new things; only rarely is it possible to fully plan an application before its implementation. The examples in this chapter have been designed to help you develop a good sense of schema design in MongoDB. Having studied these examples, you'll be well-prepared to design the best schemas for your own applications.

5.8 CONSTRUCTING QUERIES

- In this section, we will learn how to query document from MongoDB collection.

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.niralipublications.com",
        tags: ["mongodb", "database", "NoSQL"],
        likes: 100
    },
    {
        title: "NoSQL Database",
    description: "NoSQL database doesn't have tables",
        by: "tutorials point",
        url: "http://www.niralipublications.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.niralipublications.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.niralipublications.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.niralipublications.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.niralipublications.com",
    "tags" : [
        "mongodb",
        "database",
        "NoSQL"
    ],
    "likes" : 20,
    "comments" : [
        {
            "user" : "user1",
            "message" : "My first comment",
            "dateCreated" : ISODate("2013-12-
T21:05:00Z"),
            "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.niralipublications.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:	where likes

Operation	Syntax	Example	RDBMS Equivalent
Values in an array	{<key>:{$in:[<value1>, <value2>,.....<value N>]}}	db.mycol.find({"name":{$in: ["Raj", "Ram", "Raghu"]}}).pretty()	Where name matches any of the value in :["Raj", "Ram", "Raghu"]
Values not in an array	{<key>:{$nin:<value>} }	db.mycol.find({"name":{$ni n:["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.niralipublications.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.

OR In 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.niralipublications.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

Using AND and OR Together

Example

- The following example will show the documents th have likes greater than 10 and whose title is eith 'MongoDB Overview' or by is 'tutorials poin Equivalent SQL where clause is **'where likes>10 AN (by = 'tutorials point' OR title = 'MongoD 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.niralipublications.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(
    {
        $not: [
            {key1: value1}, {key2:value2}
        ]
    }
)
```

Example

- 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"
}
```

5.9 COLLECTIONS AND DOCUMENTS

Databases

- A number of databases can be run on a single MongoDB server. Default database of MongoDB is 'db', which is stored within data folder.
- MongoDB can create databases on the fly. It is not required to create a database before you start working with it.

  "show dbs" command provides you with a list of all the databases.

```
D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
Connecting to: test
> show dbs
admin      <empty>
comedy     0.03125GB
local      <empty>
student    0.03125GB
test       0.03125GB
> _
```

- Run 'db' command to refer to the current database object or connection.

```
D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
Connecting to: test
> db
test
> _
```

- To connect to a particular database, run use command.

```
> db
test
> use student
switched to db student
>
```

- In the above command, 'student' is the database we want to select.
- w3resource MongoDB tutorial has a separate page dedicated to commands related to creation and management of the database.
- Database names can be almost any character in the ASCII range. But they can't contain an empty string, a dot (i.e. "") or " ".
- Since it is reserved, "system" can't be used as a database name.
- A database name can contain "$".

Documents

- The document is the unit of storing data in a MongoDB database.
- document use JSON (JavaScript Object Notation, is a lightweight, thoroughly explorable format used to interchange data between various applications) style for storing data.
- A simple example of a JSON document is as follows :

  { site : "w3resource.com" }

  Often, the term "object" is used to refer a document.

Data Types	Description
String	May be an empty string or a combination of characters.
Integer	Digits.
Boolean	Logical values True or False.
Double	A type of floating point number.
Null	Not zero, not empty.
Array	A list of values.
Object	An entity which can be used in programming. May be a value, variable, function, or data structure.
Timestamp	A 64 bit value referring to a time and unique on a single "mongod" instance. The first 32 bit of this value refers to seconds since the UTC January 1, 1970. And last 32 bits refer to the incrementing ordinal for operations within a given second.

Data Types	Description
Internationalized Strings	UTF-8 for strings.
Object IDs	Every MongoDB object or document must have an Object ID which is unique. This is a BSON(Binary JavaScript Object Notation, which is the binary interpretation of JSON) object id, a 12-byte binary value which has a very rare chance of getting duplicated. This id consists of a 4-byte timestamp (seconds since epoch), a 3-byte machine id, a 2-byte process id, and a 3-byte counter.

- Documents are analogous to the records of an RDBMS. Insert, update, and delete operations can be performed on a collection. The following table will help you to understand the concept more easily :

RDBMS	MongoDB
Table	Collection
Column	Key
Value	Value
Records / Rows	Document / Object

- The above table shows the various datatypes which may be used in MongoDB.

Collections

- A collection may store a number of documents. A collection is analogous to a table of an RDBMS.
- A collection may store documents those who are not same in structure. This is possible because MongoDB is a Schema-free database. In a relational database like MySQL, a schema defines the organization / structure of data in a database. MongoDB does not require such a set of formula defining structure of data. So, it is quite possible to store documents of varying structures in a collection. Practically, you don't need to define a column and it's datatype unlike in RDBMS, while working with MongoDB.
- In the following code, it is shown that two MongoDB documents, belongs to same collection, storing data of different structures.

```
{"tutorial" : "NoSQL"}
        {"topic_id" : 7}
```

Copy

- A collection is created, when the first document is inserted.

Pictorial Presentation : Collections and Documents

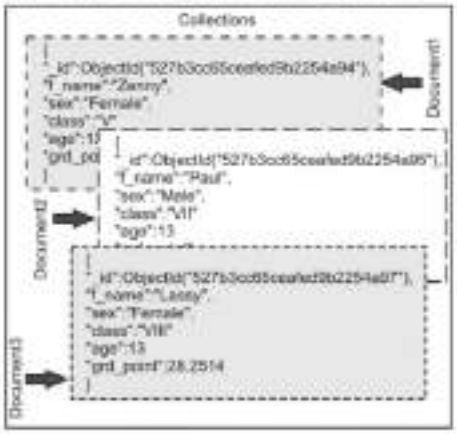


Fig. 5.7

Valid Collection Names

- Collection names must begin with letters or an underscore.
- A Collection name may contain numbers.
- You can't use "$" character within the name of a collection. "$" is reserved.
- A Collection name must not exceed 128 characters. It will be nice if you keep it within 80/90 characters.
- Using a "." (dot) notation, collections can be organized in named groups. For example, tutorials.php and tutorials.javascript both belong to tutorials. This mechanism is called as collection namespace which is for user primarily. Databases don't have much to do with it.
- Following is how to use it programmatically :

```
db.tutorials.php.findOne()
```

Capped Collections

- Imagine that you want to log the activities happening with an application, you want to store data in the same order it is inserted. MongoDB offers *Capped collections* for doing so.
- Capped collections are collections which can store data in the same order it is inserted.

- It is very fixed size, high-performance and "auto-FIFO age-Out". That is, when the allotted space is fully utilized, newly added objects (documents) will replace the older ones in the same order it is inserted.
- Since data is stored in the natural order, that is the order it is inserted, while retrieving data, no ordering is required, unless you want to reverse the order.
- New objects can be inserted into a capped collection.
- Existing objects can be updated.
- But you can't remove an individual object from the capped collection. Using drop command, you have to remove all the documents. After the drop, you have to recreate the capped collection.
- Presently, the maximum size for a capped collection is 1e9 (i.e. 1×10^9) for 32-bit machines. For 64 bit machines, there is no theoretical limit. Practically, it can be extended till your system resources permit.
- Capped collections can be used for logging, caching and auto archiving.

Use Number of Collections Instead of One

- This omits the requirement if creating index since you are not storing some repeating data on each object.
- If applied to a suitable situation, it can enhance the performance.

Metadata

- Information about a database is stored in certain collections. They are grouped in system namespace, as

dbname.system.*

Copy

The following table shows the collections and what they store

Collections with Namespace	Description
dbname.system.namespaces	list of all namespaces
dbname.system.indexes	list of all indexes
dbname.system.profile	stores database profiling information
dbname.system.users	list of users who may access the database
dbname.local.sources	stores replica slave configuration data and state
dbname.local.sources	stores replica slave configuration data and state

There are two more options to store metadata :

database.ns files stores additional namespace / index metadata if exists.

Information on the structure of a stored object is stored within the object itself.

In this section, we will see how to create a collections and documents using MongoDB.

The createCollection() Method

- MongoDB **db.createCollection(name, options)** is used to create collection.

Syntax

- Basic syntax of **createCollection()** command is as follows -

```
db.createCollection(name, options)
```

- In the command, **name** is name of collection to be created. **Options** is a document and is used to specify configuration of collection.

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Options parameter is optional, so you need to specify only the name of the collection. Following is the list of options you can use -

Field	Type	Description
CAPPED	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. **If you specify true, you need to specify size parameter also.**
AutoIndexId	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
Size	Number	(Optional) Specifies a maximum size in bytes for a capped collection. **If capped is true, then you need to specify this field also.**
Max	Number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

Examples

Basic syntax of **createCollection()** method without options is as follows –

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

You can check the created collection by using the command **show collections**.

```
>show collections
mycollection
system.indexes
```

The following example shows the syntax of createCollection() method with few important options –

```
> db.createCollection("mycol", { capped : true,
autoIndexID : true, size : 6142800, max : 10000 ){
"ok" : 0,
"errmsg" : "BSON field 'create.autoIndexID' is an
unknown field.",
"code" : 40415,
"codeName" : "Location40415"
}
>
```

- In MongoDB, you don't need to create collection. MongoDB creates collection automatically, when you insert some document.

```
>db.niralipublications.insert({"name" :
"niralipublications"})
WriteResult({ "nInserted" : 1 })
>show collections
mycol
mycollection
system.indexes
niralipublications
>
```

The drop() Method

- MongoDB's **db.collection.drop()** is used to drop a collection from the database.

Syntax

Basic syntax of **drop()** command is as follows –

```
db.COLLECTION_NAME.drop()
```

Example

- First, check the available collections into your database **mydb**.

```
>use mydb
switched to db mydb
>show collections
mycol
mycollection
system.indexes
niralipublications
>
```

Now drop the collection with the name **mycollection**.

```
>db.mycollection.drop()
true
>
```

Again check the list of collections into database.

```
>show collections
mycol
system.indexes
niralipublications
>
```

- drop() method will return true, if the selected collection is dropped successfully, otherwise it will return false.

5.10 MONGODB QUERY LANGUAGE

- 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.niralipublications.com',
 tags: ['mongodb', 'database', 'NoSQL'],
likes: 100,
comments: [
  {
    user:'user1',
    message: 'My first comment',
    dateCreated: new Date(2011,1,20,2,15),
    like: 0
 },
```

- 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 - Environment

- Let us now see how to install MongoDB on Windows.

Install MongoDB On Windows

- To install MongoDB on Windows, first download the latest release of MongoDB from https://www.mongodb.org/downloads. Make sure you get correct version of MongoDB depending upon your Windows version. To get your Windows version, open command prompt and execute the following command.

```
C:\wmic os get osarchitecture
OSArchitecture
64-bit
C:\>
```

- 32-bit versions of MongoDB only support databases smaller than 2GB and suitable only for testing and evaluation purposes.
- Now extract your downloaded file to c:\ drive or any other location. Make sure the name of the extracted folder is mongodb-win32-i386-[version] or mongodb-win32-x86_64-[version]. Here [version] is the version of MongoDB download.
- Next, open the command prompt and run the following command.

```
C:\>move mongodb-win64-* mongodb
  1 dir(s) moved.
C:\>
```

- In case you have extracted the MongoDB at different location, then go to that path by using command **cd FOLDER/DIR** and now run the above given process.
- MongoDB requires a data folder to store its files. The default location for the MongoDB data directory is c:\data\db. So you need to create this folder using the Command Prompt. Execute the following command sequence.

```
C:\>md data
C:\md data\db
```

- If you have to install the MongoDB at a differ location, then you need to specify an alternate path **\data\db** by setting the path **dbpath** in **mongod.e** For the same, issue the following commands.
- In the command prompt, navigate to the bin directo present in the MongoDB installation folder. Suppo my installation folder is **D:\set up\mongodb**

```
C:\Users\XYZ>d:
D:\>cd "set up"
D:\set up>cd mongodb
D:\set up\mongodb>cd bin
D:\set up\mongodb\bin>mongod.exe --dbpath "d:\set
up\mongodb\data"
```

- This will show **waiting for connections** message o the console output, which indicates that th mongod.exe process is running successfully.
- Now to run the MongoDB, you need to open anothe command prompt and issue the following command.

```
D:\set up\mongodb\bin>mongo.exe
MongoDB shell version: 2.4.6
connecting to: test
>db.test.save( { a: 1 } )
>db.test.find()
{ "_id" : ObjectId(5879b0f65a56a454), "a" : 1 }
>
```

- This will show that MongoDB is installed and run successfully. Next time when you run MongoDB, you need to issue only commands.

```
D:\set up\mongodb\bin>mongod.exe --dbpath "d:\set
up\mongodb\data"
D:\set up\mongodb\bin>mongo.exe
```

Install MongoDB on Ubuntu

- Run the following command to import the MongoDB public GPG key –

```
sudo apt-key adv --keyserver
hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
```

Create a /etc/apt/sources.list.d/mongodb.list file using the following command.

```
echo 'deb http://downloads-
distro.mongodb.org/repo/ubuntu-upstart dist 10gen'
| sudo tee /etc/apt/sources.list.d/mongodb.list
```

Now issue the following command to update the repository –

```
sudo apt-get update
```

- Next install the MongoDB by using the following command –

```
apt-get install mongodb-10gen = 2.2.3
```

- In the above installation, 2.2.3 is currently released MongoDB version. Make sure to install the latest version always. Now MongoDB is installed successfully.

Start MongoDB

```
sudo service mongodb start
```

Stop MongoDB

```
sudo service mongodb stop
```

Restart MongoDB

```
sudo service mongodb restart
```

To use MongoDB run the following command.

```
Mongo
```

This will connect you to running MongoDB instance.

MongoDB Help

- To get a list of commands, type **db.help()** in MongoDB client. This will give you a list of commands as shown in the following screenshot.

Fig. 5.8

MongoDB Statistics

- To get stats about MongoDB server, type the command **db.stats()** in MongoDB client. This will show the database name, number of collection and documents in the database. Output of the command is shown in the following screenshot.

Fig. 5.9

MongoDB - Data Modelling

- Data in MongoDB has a flexible schema.documents in the same collection. They 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.

Some Considerations While Designing Schema in MongoDB

Design your schema according to user requirements.

- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Example

- Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.
- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.
- In RDBMS schema, design for above requirements will have minimum three tables.

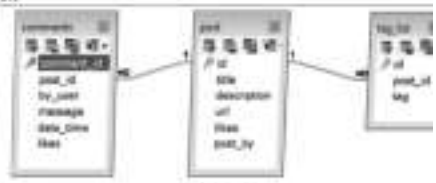


Fig. 5.10

While in MongoDB schema, design will have one collectio post and the following structure –

```
{
  _id: post_id
  title: title_of_post,
  description: post_description,
  by: post_by,
  url: url_of_post,
  tags: [tag1, tag2, tag3],
  likes: total_likes,
  comments: [
    {
      user:'comment_by',
      message: text,
      datecreated: date_time,
      like: likes
    },
    {
      user:'comment_by',
      message: text,
      datecreated: date_time,
      like: likes
    }
  ]
}
```

- So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

MongoDB - Create Database

- In this chapter, we will see how to create a database in MongoDB.

The Use Command

- MongoDB **use DATABASE_NAME** is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

Syntax

Basic syntax of **use DATABASE** statement is as follows –

```
use DATABASE_NAME
```

Example

- If you want to use a database with name **<mydb>**, then **use DATABASE** statement would be as follows –

```
>use mydb
switched to db mydb
```

- To check your currently selected database, use the command **db**

```
>db
mydb
```

- If you want to check your databases list, use the command **show dbs**.

```
>show dbs
local   0.78125GB
test    0.23012GB
```

- Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

```
>db.movie.insert({"name":"tutorials point"})
>show dbs
local       0.78125GB
mydb        0.23012GB
test        0.23012GB
```

- In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

MongoDB - Drop Database

- In this chapter, we will see how to drop a database using MongoDB command.

The dropDatabase() Method

- MongoDB **db.dropDatabase()** command is used to drop a existing database.

Syntax

- Basic syntax of **dropDatabase()** command is as follows

```
db.dropDatabase()
```

- This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

Example

- First, check the list of available databases by using the command, **show dbs**.

```
>show dbs
local       0.78125GB
mydb        0.23012GB
test        0.23012GB
>
```

- If you want to delete new database **<mydb>**, then **dropDatabase()** command would be as follows –

```
>use mydb
switched to db mydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

Now check list of databases.

```
>show dbs
local     0.78125GB
test      0.23012GB
>
```

MongoDB - Create Collection

- In this chapter, we will see how to create a collection using MongoDB.

The createCollection() Method

- MongoDB **db.createCollection(name, options)** is used to create collection.

Syntax

- Basic syntax of **createCollection()** command is as follows –

```
db.createCollection(name, options)
```

- In the command, **name** is name of collection to be created. **Options** is a document and is used to specify configuration of collection.

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

- Options parameter is optional, so you need to specify only the name of the collection. Following is the list of options you can use –

Field	Type	Description
Capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
autoIndexId	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
size	Number	(Optional) Specifies a maximum size in bytes for a capped collection. If capped is true, then you need to specify this field also.
max	Number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

Examples

Basic syntax of **createCollection()** method without options is as follows –

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

You can check the created collection by using the command **show collections**.

```
>show collections
mycollection
system.indexes
```

The following example shows the syntax of **createCollection()** method with few important options –

```
>db.createCollection("mycol", { capped : true,
autoIndexId : true, size :
   6142800, max : 10000 } )
{ "ok" : 1 }
>
```

- In MongoDB, you don't need to create collection. MongoDB creates collection automatically, when you insert some document.

```
>db.niralipublications.insert({"name" :
"niralipublications"})
>show collections
mycol
mycollection
system.indexes
niralipublications
>
```

MongoDB - Drop Collection

- In this chapter, we will see how to drop a collection using MongoDB.

The drop() Method

- MongoDB's **db.collection.drop()** is used to drop a collection from the database.

Syntax

- Basic syntax of **drop()** command is as follows –

```
db.COLLECTION_NAME.drop()
```

Example

First, check the available collections into your database **mydb**.

```
>use mydb
switched to db mydb
>show collections
mycol
mycollection
system.indexes
niralipublications
>
```

- Now drop the collection with the name **mycollection**.

```
>db.mycollection.drop()
true
>
```

Again check the list of collections into database.

```
>show collections
mycol
system.indexes
nirolipublications
>
```

- drop() method will return true, if the selected collection is dropped successfully, otherwise it will return false.

MongoDB - Datatypes

- 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.

MongoDB - Insert Document

- In this chapter, we will learn how to insert document in MongoDB collection.

The insert() Method

- To insert data into MongoDB collection, you need to use MongoDB's **insert()** or **save()** method.

Syntax

- The basic syntax of **insert()** command is as follows –

```
>db.COLLECTION_NAME.insert(document)
```

Example

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

- Here **mycol** is our collection name, as created in the previous chapter. If the collection doesn't exist in the database, then MongoDB will create this collection and then insert a document into it.
- In the inserted document, if we don't specify the _id parameter, then MongoDB assigns a unique ObjectId for this document.
  - _id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows –
  - _id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id, 3 bytes incrementer)
- To insert multiple documents in a single query, you can pass an array of documents in insert() command.

Example

```
>db.post.insert([
  {
    title: 'MongoDB Overview',
    description: 'MongoDB is no sql database',
    by: 'tutorials point',
    url: 'http://www.nirolipublications.com',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 100
```

```
  },
  {
    title: 'NoSQL Database',
    description: "NoSQL database doesn't have tables",
    by: 'tutorials point',
    url: 'http://www.niralipublications.com',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 20,
    comments: [
      {
        user:'user1',
        message: 'My first comment',
        dateCreated: new Date(2013,11,10,2,35),
        like: 0
      }
    ]
  }
])
```

- To insert the document you can use **db.post.save(document)** also. If you don't specify **_id** in the document then **save()** method will work same as **insert()** method. If you specify _id then it will replace whole data of document containing _id as specified in save() method.

MongoDB - Query Document

- In this chapter, we will learn how to query document from MongoDB collection.

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.

The pretty() Method

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

Syntax

```
db.mycol.find().pretty()
```

Example

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

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

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>:<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

AND In MongoDB

Syntax

In the **find()** method, if you pass multiple keys by separating them by ',' then MongoDB treats it as **AND** condition. Following is the basic syntax of **AND** –

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

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(7df78ad8902c),
  "title": "MongoDB Overview",
  "description": "MongoDB is no sql database",
  "by": "tutorials point",
  "url": "http://www.niralipublications.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.

OR In 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.niralipublications.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.niralipublications.com",
  "tags": ["mongodb", "database", "NoSQL"],
  "likes": "100"
}
>
```

MongoDB - Update Document

- MongoDB's **update()** and **save()** methods are used to update document into a collection. The update() method updates the values in the existing document while the save() method replaces the existing document with the document passed in save() method.

MongoDB Update() Method

- The update() method updates the values in the existing document.

Syntax

The basic syntax of **update()** method is as follows –

```
>db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)
```

Example

Consider the mycol collection has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5),
"title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6),
"title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7),
"title":"Tutorials Point Overview"}
```

- Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'.

```
>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
>
```

- By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

```
>db.mycol.update({'title':'MongoDB Overview'},
  {$set:{'title':'New MongoDB Tutorial'}},{multi:true})
```

MongoDB Save() Method

- The **save()** method replaces the existing document with the new document passed in the save() method.

Syntax

- The basic syntax of MongoDB **save()** method is shown below –

```
>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

Example

Following example will replace the document with the _id '5983548781331adf45ec5'.

```
>db.mycol.save(
  {
    "_id" : ObjectId(5983548781331adf45ec5),
"title":"Tutorials Point New Topic",
    "by":"Tutorials Point"
  }
)
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5),
"title":"Tutorials Point New Topic",
  "by":"Tutorials Point"}
{ "_id" : ObjectId(5983548781331adf45ec6),
{ "_id" : ObjectId(5983548781331adf45ec7),
"title":"Tutorials Point Overview"}
>
```

MongoDB - Delete Document

- In this chapter, we will learn how to delete a document using MongoDB.

The remove() Method

- MongoDB's **remove()** method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.
  - ➢ **deletion criteria** – (Optional) deletion criteria according to documents will be removed.
  - ➢ **justOne** – (Optional) if set to true or 1, then remove only one document.

Syntax

Basic syntax of **remove()** method is as follows –

```
>db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)
```

Example

Consider the mycol collection has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will remove all the documents whose title is 'MongoDB Overview'.

```
>db.mycol.remove({'title':'MongoDB Overview'})
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
>
```

Remove Only One

- If there are multiple records and you want to delete only the first record, then set **justOne** parameter in **remove()** method.

```
>db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)
```

Remove All Documents

If you don't specify deletion criteria, then MongoDB will delete whole documents from the collection. **This is Equivalent of SQL's Truncate Command.**

```
>db.mycol.remove({})
>db.mycol.find()
>
```

MongoDB - Projection

- In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

The find() Method

- MongoDB's **find()** method, explained in MongoDB Query Document accepts second optional parameter that is list of fields that you want to retrieve. In MongoDB, when you execute **find()** method, then it displays all fields of a document. To limit this, you need to set a list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the fields.

Syntax

- The basic syntax of **find()** method with projection is as follows –

```
>db.COLLECTION_NAME.find({},{KEY:1})
```

Example

Consider the collection mycol has the following data –

```
{ "_id" : ObjectId(5983548781331adf45ec5),
"title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6),
"title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7),
"title":"Tutorials Point Overview"}
```

- Following example will display the title of the document while querying the document.

```
>db.mycol.find({},{"title":1,_id:0})
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
{"title":"Tutorials Point Overview"}
>
```

- Please note **_id** field is always displayed while executing **find()** method, if you don't want this field, then you need to set it as 0.

MongoDB - Limit Records

- In this chapter, we will learn how to limit records using MongoDB.

The Limit() Method

- To limit the records in MongoDB, you need to use **limit()** method. The method accepts one number type argument, which is the number of documents that you want to be displayed.

Syntax

The basic syntax of **limit()** method is as follows –

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

Example

Consider the collection myycol has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5),
"title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6),
"title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7),
"title":"Tutorials Point Overview"}
```

- Following example will display only two documents while querying the document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(2)
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
>
```

- If you don't specify the number argument in **limit()** method then it will display all documents from the collection.

MongoDB Skip() Method

- Apart from limit() method, there is one more method **skip()** which also accepts number type argument and is used to skip the number of documents.

Syntax

The basic syntax of **skip()** method is as follows –

```
>db.COLLECTION_NAME.find().limit(NUMBER).skip(N
UMBER)
```

Example

Following example will display only the second document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(1).skip(1)
{"title":"NoSQL Overview"}
>
```

- Please note, the default value in **skip()** method is 0.

MongoDB - Sort Records

- In this chapter, we will learn how to sort records in MongoDB.

The sort() Method

- To sort documents in MongoDB, you need to use **sort()** method. The method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

Syntax

The basic syntax of **sort()** method is as follows –

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

Example

Consider the collection mycol has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5),
"title":"MongoDB Overview")
{ "_id" : ObjectId(5983548781331adf45ec6),
"title":"NoSQL Overview")
{ "_id" : ObjectId(5983548781331adf45ec7),
"title":"Tutorials Point Overview")
```

Following example will display the documents sorted by title in the descending order.

```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})
{"title":"Tutorials Point Overview")
{"title":"NoSQL Overview")
{"title":"MongoDB Overview")
>
```

- Please note, if you don't specify the sorting preference, then **sort()** method will display the documents in ascending order.

MongoDB - Indexing

- Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require MongoDB to process a large volume of data.
- Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

The ensureIndex() Method

- To create an index you need to use ensureIndex() method of MongoDB.

Syntax

- The basic syntax of **ensureIndex()** method is as follows().

```
>db.COLLECTION_NAME.ensureIndex({KEY:1})
```

- Here key is the name of the field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

Example

```
>db.mycol.ensureIndex({"title":1})
>
```

- In **ensureIndex()** method you can pass multiple fields, to create index on multiple fields.

```
>db.mycol.ensureIndex({"title":1,"description":-1})
>
```

- **ensureIndex()** method also accepts list of options (which are optional). Following is the list –

Parameter	Type	Description
background	Boolean	Builds the index in the background so that building an index does not block other database activities. Specify true to build in the background. The default value is false.
unique	Boolean	Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify true to create a unique index. The default value is false.
name	string	The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order.
DropDups	Boolean	Creates a unique index on a field that may have duplicates. MongoDB indexes only the first occurrence of a key and removes all documents from the collection that contain subsequent occurrences of that key. Specify true to create unique index. The default value is false.

Parameter	Type	Description
Sparse	Boolean	If true, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is false.
ExpireAfterSeconds	Integer	Specifies a value, in seconds, as a TTL to control how long MongoDB retains documents in this collection.
v	index version	The index version number. The default index version depends on the version of MongoDB running when creating the index.
weights	document	The weight is a number ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score.
default_language	String	For a text index, the language that determines the list of stop words and the rules for the stemmer and tokenizer. The default value is english.
language_override	String	For a text index, specify the name of the field in the document that contains, the language to override the default language. The default value is language.

MongoDB - Aggregation

- Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In SQL count(*) and with group by is an equivalent of mongodb aggregation.

The aggregate() Method

- For the aggregation in MongoDB, you should use **aggregate()** method.

Syntax

- Basic syntax of **aggregate()** method is as follows –

```
db.COLLECTION_NAME.aggregate(AGGREGATE_OPE
RATION)
```

Example

- In the collection you have the following data –

```
{
   _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
   description: 'MongoDB is no sql database',
   by_user: 'tutorials point',
   url: 'http://www.niralipublications.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100
},
{
   _id: ObjectId(7df78ad8902d)
   title: 'NoSQL Overview',
   description: 'No sql database is very fast',
   by_user: 'tutorials point',
   url: 'http://www.niralipublications.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 10
},
{
   _id: ObjectId(7df78ad8902e)
   title: 'Neo4j Overview',
   description: 'Neo4j is no sql database',
   by_user: 'Neo4j',
   url: 'http://www.neo4j.com',
   tags: ['neo4j', 'database', 'NoSQL'],
   likes: 750
},
```

- Now from the above collection, if you want to display a list stating how many tutorials are written by each user, then you will use the following **aggregate()** method –

```
> db.mycol.aggregate([{$group : {_id : "$by_user",
num_tutorial : {$sum : 1}}}])
{
  "result" : [
    {
      "_id" : "tutorials point",
      "num_tutorial" : 2
    },
    {
```

```
        "_id" : "Neo4j",
        "num_tutorial" : 1
    }
  ],
  "ok" : 1
}
>
```

- Sql equivalent query for the above use case will be **select by_user, count(*) from mycol group by by_user**.
- In the above example, we have grouped documents by field **by_user** and on each occurrence of by_user previous value of sum is incremented. Following is a list of available aggregation expressions.

Expression	Description	Example
$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([$group : {_id : "$by_user", num_tutorial : {$sum : "$likes"}}])
$avg	Calculates the average of all given values from all documents in the collection.	db.mycol.aggregate([$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}])
$min	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}])
$max	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([$group : {_id : "$by_user", num_tutorial : {$max : "$likes"}}])
$push	Inserts the value to an array in the resulting document.	db.mycol.aggregate([$group : {_id : "$by_user", url : {$push: "$url"}}])
$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([$group : {_id : "$by_user", url : {$addToSet : "$url"}}])
$first	Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "$sort"-stage.	db.mycol.aggregate([$group : {_id : "$by_user", first_url : {$first : "$url"}}])
$last	Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "$sort"-stage.	db.mycol.aggregate([$group : {_id : "$by_user", last_url : {$last : "$url"}}])

Pipeline Concept

- In UNIX command, shell pipeline means the possibility to execute an operation on some input and use the output as the input for the next command and so on. MongoDB also supports same concept in aggregation framework. There is a set of possible stages and each of those is taken as a set of documents as an input and produces a resulting set of documents (or the final resulting JSON document at the end of the pipeline). This can then in turn be used for the next stage and so on.

- Following are the possible stages in aggregation framework –
  - **$project** : Used to select some specific fields from a collection.
  - **$match** : This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.
  - **$group** : This does the actual aggregation as discussed above.
  - **$sort** : Sorts the documents.
  - **$skip** : With this, it is possible to skip forward in the list of documents for a given amount of documents.
  - **$limit** : This limits the amount of documents to look at, by the given number starting from the current positions.
  - **$unwind** : This is used to unwind document that are using arrays. When using an array, the data is kind of pre-joined and this operation will be undone with this to have individual documents again. Thus with this stage we will increase the amount of documents for the next stage.

5.10.1 MongoDB - Replication

- Replication is the process of synchronizing data across multiple servers. Replication provides redundancy and increases data availability with multiple copies of data on different database servers. Replication protects a database from the loss of a single server. Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

Why Replication?

- To keep your data safe
- High (24*7) availability of data
- Disaster recovery
- No downtime for maintenance (like backups, index rebuilds, compaction)
- Read scaling (extra copies to read from)
- Replica set is transparent to the application

How Replication Works in MongoDB

- MongoDB achieves replication by the use of replica set. A replica set is a group of **mongod** instances that host the same data set. In a replica, one node is primary node that receives all write operations. All other instances, such as secondary's, apply operations from the primary so that they have the same data set. Replica set can have only one primary node.
  - Replica set is a group of two or more nodes (generally minimum 3 nodes are required).
  - In a replica set, one node is primary node and remaining nodes are secondary.
  - All data replicates from primary to secondary node.
  - At the time of automatic failover or maintenance, election establishes for primary and a new primary node is elected.
  - After the recovery of failed node, it again join the replica set and works as a secondary node.
- A typical diagram of MongoDB replication is shown in which client application always interact with the primary node and the primary node then replicates the data to the secondary nodes.

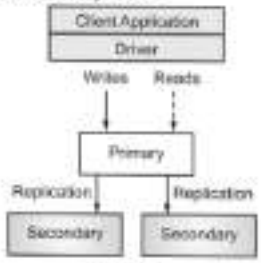


Fig. 5.11

Replica Set Features

- A cluster of N nodes
- Any one node can be primary
- All write operations go to primary
- Automatic failover
- Automatic recovery
- Consensus election of primary

Set Up a Replica Set

- In this tutorial, we will convert standalone MongoDB instance to a replica set. To convert to replica set, following are the steps –

  Shutdown already running MongoDB server.
  - Start the MongoDB server by specifying -- replSet option. Following is the basic syntax of --replSet –

```
mongod --port "PORT" --dbpath
"YOUR_DB_DATA_PATH" --replSet
"REPLICA_SET_INSTANCE_NAME"
```

Example

```
mongod --port 27017 --dbpath "D:\set
up\mongodb\data" --replSet rs0
```

- It will start a mongod instance with the name rs0, on port 27017.
- Now start the command prompt and connect to this mongod instance.
- In Mongo client, issue the command **rs.initiate()** to initiate a new replica set.
- To check the replica set configuration, issue the command **rs.conf()**. To check the status of replica set issue the command **rs.status()**.

Add Members to Replica Set

- To add members to replica set, start mongod instances on multiple machines. Now start a mongo client and issue a command **rs.add()**.

Syntax

The basic syntax of **rs.add()** command is as follows –

```
>rs.add(HOST_NAME:PORT)
```

Example

- Suppose your mongod instance name is **mongod1.net** and it is running on port **27017**. To add this instance to replica set, issue the command **rs.add()** in Mongo client.

```
>rs.add("mongod1.net:27017")
>
```

- You can add mongod instance to replica set only when you are connected to primary node. To check whether you are connected to primary or not, issue the command **db.isMaster()** in mongo client.

5.10.2 MongoDB - Sharding

- Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

Why Sharding?

- In replication, all writes go to master node
- Latency sensitive queries still go to master
- Single replica set has limitation of 12 nodes
- Memory can't be large enough when active dataset is big
- Local disk is not big enough
- Vertical scaling is too expensive

Sharding in MongoDB

- The following diagram shows the sharding in MongoDB using sharded cluster.

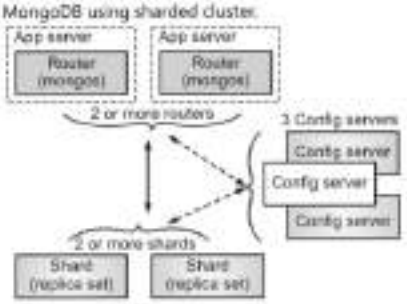


Fig. 5.12

- In the following diagram, there are three main components :
  1. **Shards** : Shards are used to store data. They provide high availability and data consistency. In production environment, each shard is a separate replica set.
  2. **Config Servers** : Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards. In production environment, sharded clusters have exactly 3 config servers.
  3. **Query Routers** : Query routers are basically mongo instances, interface with client applications and direct operations to the appropriate shard. The query router processes and targets the

operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Generally, a sharded cluster have many query routers.

5.10.3 MongoDB - Create Backup

- In this chapter, we will see how to create a backup in MongoDB.

Dump MongoDB Data

- To create backup of database in MongoDB, you should use **mongodump** command. This command will dump the entire data of your server into the dump directory. There are many options available by which you can limit the amount of data or create backup of your remote server.

Syntax

- The basic syntax of **mongodump** command is as follows –

```
>mongodump
```

Example

- Start your mongod server. Assuming that your mongod server is running on the localhost and port 27017, open a command prompt and go to the bin directory of your mongodb instance and type the command **mongodump**
- Consider the mycol collection has the following data.

```
>mongodump
```

- The command will connect to the server running at **127.0.0.1** and port **27017** and back all data of the server to directory **/bin/dump/**. Following is the output of the command –

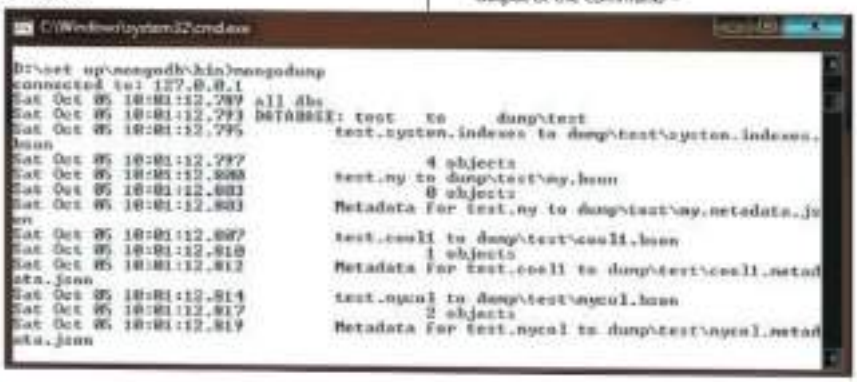


Fig. 5.13

- Following is a list of available options that can be used with the **mongodump** command.

Syntax	Description	Example
mongodump --host HOST_NAME --port PORT_NUMBER	This command will backup all databases of specified mongod instance.	mongodump --host nirajpublications.com --port 27017
mongodump --dbpath DB_PATH --out BACKUP_DIRECTORY	This command will backup only specified database at specified path.	mongodump --dbpath /data/db/ --out /data/backup/
mongodump --collection COLLECTION --db DB_NAME	This command will backup only specified collection of specified database.	mongodump --collection mycol --db test

Restore data

- To restore backup data MongoDB's **mongorestore** command is used. This command restores all of the data from the backup directory.

Syntax

The basic syntax of **mongorestore** command is –

```
mongorestore
```

Following is the output of the command –

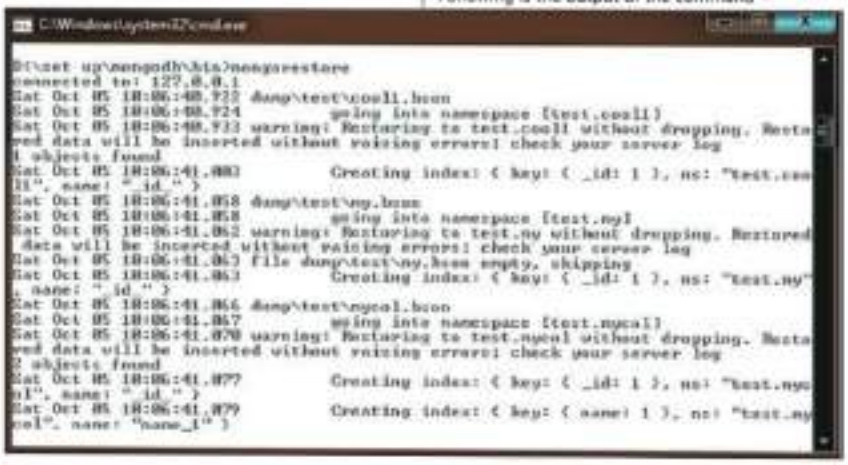


Fig. 5.14

5.10.4 MongoDB - Deployment

- When you are preparing a MongoDB deployment, you should try to understand how your application is going to hold up in production. It's a good idea to develop a consistent, repeatable approach to managing your deployment environment so that you can minimize any surprises once you're in production.
- The best approach incorporates prototyping your set up, conducting load testing, monitoring key metrics, and using that information to scale your set up. The key part of the approach is to proactively monitor your entire system - this will help you understand how your production system will hold up before deploying, and determine where you will need to add capacity.
- Having insight into potential spikes in your memory usage, for example, could help put out a write-lock fire before it starts.
- To monitor your deployment, MongoDB provides some of the following commands –

mongostat

- This command checks the status of all running mongod instances and return counters of database operations.
- These counters include inserts, queries, updates, deletes, and cursors.
- Command also shows when you're hitting page faults, and showcase your lock percentage.
- This means that you're running low on memory, hitting write capacity or have some performance issue.
- To run the command, start your mongod instance. In another command prompt, go to **bin** directory of your mongodb installation and type **mongostat**.

```
D:\set up\mongodb\bin>mongostat
```

Following is the output of the command –

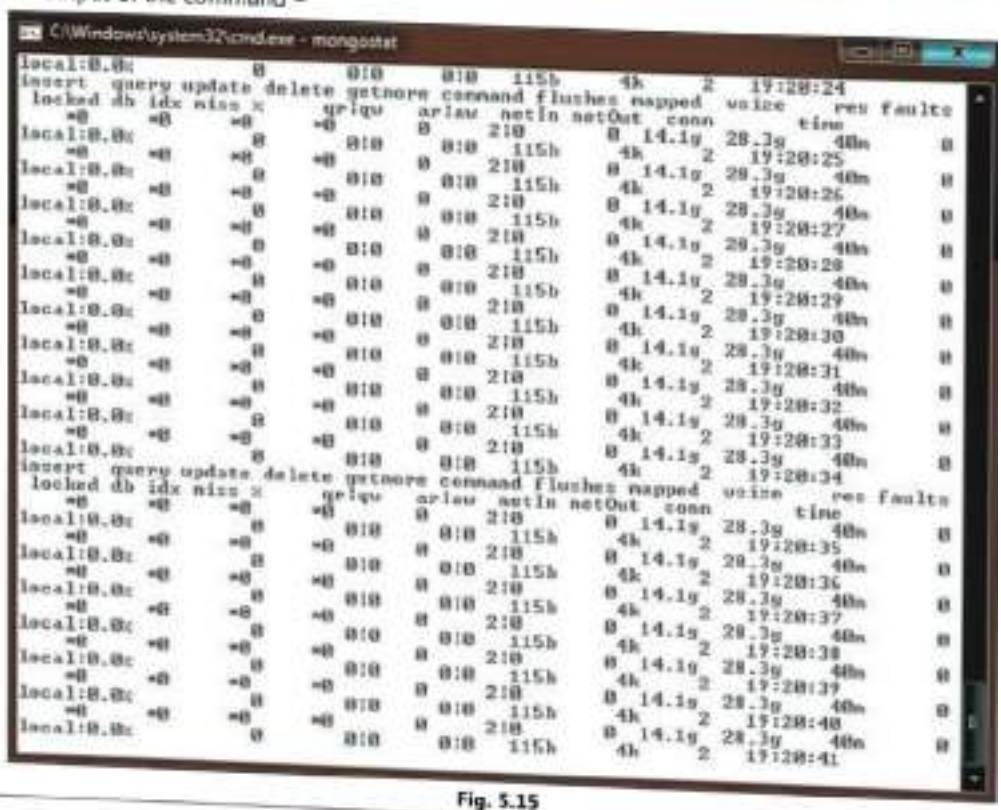


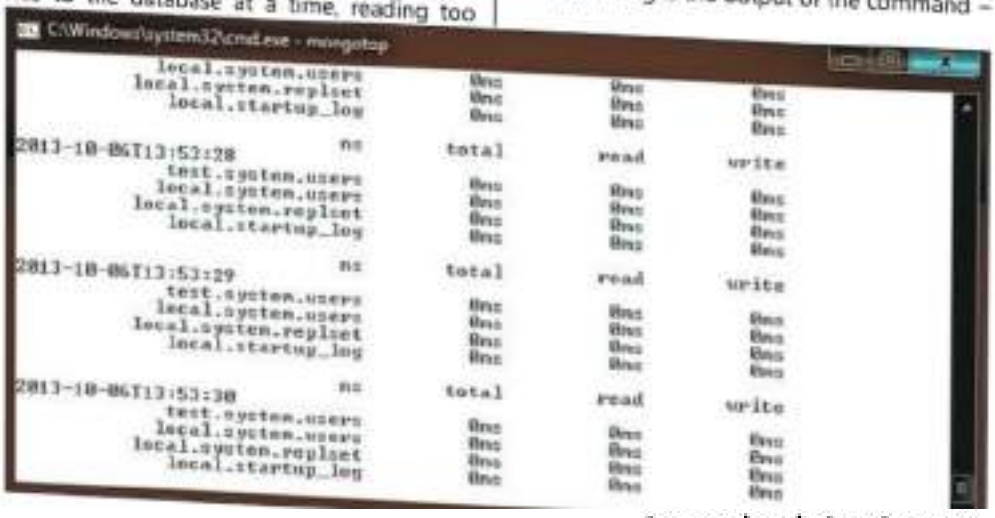
Fig. 5.15

mongotop

- This command tracks and reports the read and write activity of MongoDB instance on a collection basis. By default, **mongotop** returns information in each second, which you can change it accordingly. You should check that this read and write activity matches your application intention, and you're not firing too many writes to the database at a time, reading too frequently from a disk, or are exceeding y set size.
- To run the command, start your mongod another command prompt, go to **bin** dire mongodb installation and type **mongotop**

```
D:\set up\mongodb\bin>mongotop
```

- Following is the output of the command –



- To change **mongotop** command to return information less frequently, specify a specific number after the mongotop command.

```
D:\set up\mongodb\bin>mongotop 30
```

- The above example will return values every 30 seconds.
- Apart from the MongoDB tools, 10gen provides a free, hosted monitoring service, MongoDB Management Service (MMS), that provides a dashboard and gives you a view of the metrics from your entire cluster.

5.10.5 MongoDB - Java

- In this chapter, we will learn how to set up MongoDB JDBC driver.

Installation

- Before you start using MongoDB in your Java programs, you need to make sure that you have MongoDB JDBC driver and Java set up on the machine. You can check Java tutorial for Java installation on your machine. Now, let us check how to set up MongoDB JDBC driver.
- You need to download the jar from the path Download mongo.jar. Make sure to download the latest release of it.
- You need to include the mongo.jar into your classpath.

Connect to Database

- To connect database, you need to specify the database name, if the database doesn't exist then MongoDB creates it automatically.
- Following is the code snippet to connect to the database –

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class ConnectToDB {

  public static void main( String args[] ) {

    // Creating a Mongo client
    MongoClient mongo = new MongoClient( "localhost" ,
27017 );

    // Creating Credentials
    MongoCredential credential =
MongoCredential.createCredential("sampleUser",
"myDb", "password".toCharArray());
    System.out.println("Connected to the database
successfully");

    // Accessing the database
    MongoDatabase database =
mongo.getDatabase("myDb");
    System.out.println("Credentials ::"+ credential);
  }
}
```

- Now, let's compile and run the above program to create our database myDb as shown below.

```
$javac ConnectToDB.java
$java ConnectToDB
```

On executing, the above program gives you the following output.

```
Connected to the database successfully
Credentials ::MongoCredential{
  mechanism = null,
  userName = 'sampleUser',
  source = 'myDb',
  password = <hidden>,
  mechanismProperties = {}
}
```

Create a Collection

- To create a collection, **createCollection()** method of **com.mongodb.client.MongoDatabase** class is used.
- Following is the code snippet to create a collection –

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class CreatingCollection {

  public static void main( String args[] ) {

    // Creating a Mongo client
    MongoClient mongo = new MongoClient( "localhost"
27017 );
```

```
    // Creating Credentials
    MongoCredential credential;
    credential =
MongoCredential.createCredential("sampleUser",
"myDb", "password".toCharArray());
    System.out.println("Connected to the database
successfully");

    //Accessing the database
    MongoDatabase database =
mongo.getDatabase("myDb");

    //Creating a collection
    database.createCollection("sampleCollection");
    System.out.println("Collection created
successfully");
  }
}
```

On compiling, the above program gives you the following result –

```
Connected to the database successfully
Collection created successfully
```

Getting/Selecting a Collection

- To get/select a collection from the database, **getCollection()** method of **com.mongodb.client.MongoDatabase** class is used.

Following is the program to get/select a collection –

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class selectingCollection {

  public static void main( String args[] ) {

    // Creating a Mongo client
    MongoClient mongo = new MongoClient( "localhost" ,
27017 );
    // Creating Credentials
    MongoCredential credential;
    credential =
MongoCredential.createCredential("sampleUser",
"myDb", "password".toCharArray());
    System.out.println("Connected to the database
successfully");

    // Accessing the database
    MongoDatabase database =
mongo.getDatabase("myDb");

    // Creating a collection
    System.out.println("Collection created
successfully");

    // Retrieving a collection
    MongoCollection<Document> collection =
database.getCollection("myCollection");
    System.out.println("Collection myCollection selected
successfully");
  }
}
```

On compiling, the above program gives you the following result –

```
Connected to the database successfully
Collection created successfully
Collection myCollection selected successfully
```

Insert a Document

- o insert a document into MongoDB, **insert()** method of **com.mongodb.client.MongoCollection** class is used.

Following is the code snippet to insert a document –

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class InsertingDocument {
  public static void main( String args[] ) {
```

```
    // Creating a Mongo client
    MongoClient mongo = new MongoClient( "localhost" ,
27017 );

    // Creating Credentials
    MongoCredential credential;
    credential =
MongoCredential.createCredential("sampleUser",
"myDb", "password".toCharArray());
    System.out.println("Connected to the database
successfully");

    // Accessing the database
    MongoDatabase database =
mongo.getDatabase("myDb");

    // Retrieving a collection
    MongoCollection<Document> collection =
database.getCollection("sampleCollection");
    System.out.println("Collection sampleCollection
selected successfully");

    Document document = new Document("title",
"MongoDB")
    .append("id", 1)
    .append("description", "database")
    .append("likes", 100)
    .append("url",
"http://www.nirajpublications.com/mongodb/")
    .append("by", "tutorials point");
    collection.insertOne(document);
    System.out.println("Document inserted
successfully");
  }
}
```

On compiling, the above program gives you the following result −

```
Connected to the database successfully
Collection sampleCollection selected successfully
Document inserted successfully
```

Retrieve All Documents

- To select all documents from the collection, **find()** method of **com.mongodb.client.MongoCollection** class is used. This method returns a cursor, so you need to iterate this cursor.

Following is the program to select all documents −

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class RetrievingAllDocuments {

  public static void main( String args[] ) {

    // Creating a Mongo client
    MongoClient mongo = new MongoClient( "localhost" ,
27017 );

    // Creating Credentials
    MongoCredential credential;
    credential =
MongoCredential.createCredential("sampleUser",
"myDb",  "password".toCharArray());
    System.out.println("Connected to the database
successfully");

    // Accessing the database
    MongoDatabase database =
mongo.getDatabase("myDb");

    // Retrieving a collection
    MongoCollection<Document> collection =
database.getCollection("sampleCollection");
    System.out.println("Collection sampleCollection
selected successfully");
    // Getting the iterable object
```

```
    FindIterable<Document> iterDoc = collection.find();
    int i = 1;

    // Getting the iterator
    Iterator it = iterDoc.iterator();

    while (it.hasNext()) {
      System.out.println(it.next());
    i++;
    }
  }
}
```

On compiling, the above program gives you the following result –

```
Document{{
  _id = 5967745223993a32646baab8,
  title = MongoDB,
  id = 1,
  description = database,
  likes = 100,
  url = http://www.niralipublications.com/mongodb/, by
= tutorials point
}}
Document{{
  _id = 7452239959673a32646baab8,
  title = RethinkDB,
  id = 2,
  description = database,
  likes = 200,
  url = http://www.niralipublications.com/rethinkdb/,
by = tutorials point
}}
```

Update Document

- To update a document from the collection, **updateOne()** method of **com.mongodb.client.MongoCollection** class is used.

Following is the program to select the first document –

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Updates;

import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class UpdatingDocuments {

  public static void main( String args[] ) {

    // Creating a Mongo client
    MongoClient mongo = new MongoClient( "localhost" ,
27017 );

    // Creating Credentials
    MongoCredential credential;
    credential =
MongoCredential.createCredential("sampleUser",
"myDb", "password".toCharArray());
    System.out.println("Connected to the database
successfully");

    // Accessing the database
    MongoDatabase database =
mongo.getDatabase("myDb");

    // Retrieving a collection
    MongoCollection<Document> collection =
database.getCollection("sampleCollection");
    System.out.println("Collection myCollection selected
successfully");

    collection.updateOne(Filters.eq("id", 1),
Updates.set("likes", 150));
    System.out.println("Document update
successfully...");

    // Retrieving the documents after updation
    // Getting the iterable object
    FindIterable<Document> iterDoc = collection.find();
```

```
    int i = 1;

    // Getting the iterator
    Iterator it = iterDoc.iterator();

    while (it.hasNext()) {
      System.out.println(it.next());
      i++;
    }
  }
}
```

On compiling, the above program gives you the following result –

```
Document update successfully...
Document {{
  _id = 5967745223993a32646baab8,
  title = MongoDB,
  id = 1,
  description = database,
  likes = 150,
  url = http://www.niralipublications.com/mongodb/, by
= tutorials point
}}
```

Delete a Document

- To delete a document from the collection, you need to use the **deleteOne()** method of the **com.mongodb.client.MongoCollection** class.
- Following is the program to delete a document –

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;

import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class DeletingDocuments {
  public static void main( String args[] ) {
    MongoClient mongo = new MongoClient( "localhost" ,
27017 );

    // Creating Credentials
    MongoCredential credential;
    credential =
MongoCredential.createCredential("sampleUser",
"myDb", "password".toCharArray());
    System.out.println("Connected to the database
successfully");

    // Accessing the database
    MongoDatabase database =
mongo.getDatabase("myDb");

    // Retrieving a collection
    MongoCollection<Document> collection =
database.getCollection("sampleCollection");
    System.out.println("Collection sampleCollection
selected successfully");

    // Deleting the documents
    collection.deleteOne(Filters.eq("id", 1));
    System.out.println("Document deleted
successfully...");

    // Retrieving the documents after updation
    // Getting the iterable object
    FindIterable<Document> iterDoc = collection.find();
    int i = 1;

    // Getting the iterator
    Iterator it = iterDoc.iterator();

    while (it.hasNext()) {
      System.out.println("Inserted Document: "+i);
      System.out.println(it.next());
      i++;
    }
  }
}
```

On compiling, the above program gives you the following result –

```
Connected to the database successfully
Collection sampleCollection selected successfully
Document deleted successfully..
```

Dropping a Collection

- To drop a collection from a database, you need to use the **drop()** method of the **com.mongodb.client.MongoCollection** class.

Following is the program to delete a collection –

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class DropingCollection {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" ,
27017 );

      // Creating Credentials
      MongoCredential credential;
      credential =
MongoCredential.createCredential("sampleUser",
myDb", "password".toCharArray());
      System.out.println("Connected to the database
uccessfully");

      // Accessing the database
      MongoDatabase database =
ango.getDatabase("myDb");

      // Creating a collection
      System.out.println("Collections created
ccessfully");
      // Retrieving a collection
      MongoCollection<Document> collection =
database.getCollection("sampleCollection");

      // Dropping a Collection
      collection.drop();
      System.out.println("Collection dropped
successfully");
   }
}
```

On compiling, the above program gives you the following result –

```
Connected to the database successfully
Collection sampleCollection selected successfully
Collection dropped successfully
```

Listing All the Collections

To list all the collections in a database, you need to use the **listCollectionNames()** method of the **com.mongodb.client.MongoDatabase** class.

Following is the program to list all the collections of a database –

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class ListOfCollection {

   public static void main( String args[] ) {

      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" ,
27017 );

      // Creating Credentials
      MongoCredential credential;
      credential =
MongoCredential.createCredential("sampleUser",
"myDb", "password".toCharArray());

      System.out.println("Connected to the database
successfully");
      // Accessing the database
```

```
    MongoDatabase database =
mongo.getDatabase("myDb");
    System.out.println("Collection created
successfully");
    for (String name : database.listCollectionNames()) {
      System.out.println(name);
    }
  }
}
```

On compiling, the above program gives you the following result –

```
Connected to the database successfully
Collection created successfully
myCollection
myCollection1
myCollection5
```

- Remaining MongoDB methods **save(), limit(), skip(), sort()** etc. work same as explained in the subsequent tutorial.

5.10.6 MongoDB - PHP

- To use MongoDB with PHP, you need to use MongoDB PHP driver. Download the driver from the url Download PHP Driver. Make sure to download the latest release of it. Now unzip the archive and put php_mongo.dll in your PHP extension directory ("ext" by default) and add the following line to your php.ini file –

```
extension = php_mongo.dll
```

Make a Connection and Select a Database

- To make a connection, you need to specify the database name, if the database doesn't exist then MongoDB creates it automatically.
- Following is the code snippet to connect to the database –

```
<?php
  // connect to mongodb
  $m = new MongoClient();

  echo "Connection to database successfully";
  // select a database
  $db = $m->mydb;

  echo "Database mydb selected";
?>
```

- When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
```

Create a Collection

Following is the code snippet to create a collection –

```
<?php
  // connect to mongodb
  $m = new MongoClient();
  echo "Connection to database successfully";

  // select a database
  $db = $m->mydb;
  echo "Database mydb selected";
  $collection = $db->createCollection("mycol");
  echo "Collection created succsessfully";
?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
Collection created succsessfully
```

Insert a Document

To insert a document into MongoDB, **insert()** method is used.

Following is the code snippet to insert a document –

```
<?php
  // connect to mongodb
  $m = new MongoClient();
  echo "Connection to database successfully";

  // select a database
  $db = $m->mydb;
  echo "Database mydb selected";
  $collection = $db->mycol;
  echo "Collection selected succsessfully";

  $document = array(
    "title" => "MongoDB",
    "description" => "database",
    "likes" => 100,
```

```
    "url" =>
"http://www.niralipublications.com/mongodb/",
    "by" => "tutorials point"
  );

  $collection->insert($document);
  echo "Document inserted successfully";
?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
Collection selected succesfully
Document inserted successfully
```

Find All Documents

To select all documents from the collection, find() method is used.

Following is the code snippet to select all documents –

```
<?php
  // connect to mongodb
  $m = new MongoClient();
  echo "Connection to database successfully";

  // select a database
  $db = $m->mydb;
  echo "Database mydb selected";
  $collection = $db->mycol;
  echo "Collection selected succesfully";

  $cursor = $collection->find();
  // iterate cursor to display title of documents

  foreach ($cursor as $document) {
    echo $document["title"] . "\n";
  }
?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
Collection selected succesfully (
  "title": "MongoDB"
)
```

Update a Document

- To update a document, you need to use the update() method.
- In the following example, we will update the title of inserted document to **MongoDB Tutorial**. Following is the code snippet to update a document –

```
<?php
  // connect to mongodb
  $m = new MongoClient();
  echo "Connection to database successfully";

  // select a database
  $db = $m->mydb;
  echo "Database mydb selected";
  $collection = $db->mycol;
  echo "Collection selected succesfully";

  // now update the document
  $collection->update(array("title"=>"MongoDB"),
    array('$set'=>array("title"=>"MongoDB Tutorial")));
  echo "Document updated successfully";

  // now display the updated document
  $cursor = $collection->find();

  // iterate cursor to display title of documents
  echo "Updated document";

  foreach ($cursor as $document) {
    echo $document["title"] . "\n";
  }
?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
Collection selected succesfully
Document updated successfully
Updated document (
  "title": "MongoDB Tutorial"
)
```

Delete a Document

- To delete a document, you need to use remove() method.
- In the following example, we will remove the documents that has the title **MongoDB Tutorial**. Following is the code snippet to delete a document –

```
<?php
  // connect to mongodb
  $m = new MongoClient();
  echo "Connection to database successfully";

  // select a database
  $db = $m->mydb;
  echo "Database mydb selected";
  $collection = $db->mycol;
  echo "Collection selected succesfully";

  // now remove the document
  $collection->remove(array("title"=>"MongoDB
Tutorial"),false);
  echo "Documents deleted successfully";

  // now display the available documents
  $cursor = $collection->find();

  // iterate cursor to display title of documents
  echo "Updated document";

  foreach ($cursor as $document) {
    echo $document["title"] . "\n";
  }
?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
Collection selected succesfully
Documents deleted successfully
```

- In the above example, the second parameter is boolean type and used for **justOne** field of **remove()** method.
- Remaining MongoDB methods **findOne()**, **save()**, **limit()**, **skip()**, **sort()** etc. works same as explained above.

5.10.7 MongoDB - Relationships

- Relationships in MongoDB represent how various documents are logically related to each other. Relationships can be modeled via **Embedded** and **Referenced** approaches. Such relationships can be either 1:1, 1:N, N:1 or N:N.
- Let us consider the case of storing addresses for users. So, one user can have multiple addresses making this a 1:N relationship.

Following is the sample document structure of **user** document –

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "name": "Tom Hanks",
  "contact": "987654321",
  "dob": "01-01-1991"
}
```

Following is the sample document structure of **address** document –

```
{
  "_id":ObjectId("52ffc4a5d85242602e000000"),
  "building": "22 A, Indiana Apt",
  "pincode": 123456,
  "city": "Los Angeles",
  "state": "California"
}
```

Modeling Embedded Relationships

- In the embedded approach, we will embed the address document inside the user document.

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address": [
    {
      "building": "22 A, Indiana Apt",
      "pincode": 123456,
      "city": "Los Angeles",
```

```
    "state": "California"
  },
  {
    "building": "170 A, Acropolis Apt",
    "pincode": 456789,
    "city": "Chicago",
    "state": "Illinois"
  }
 ]
}
```

- This approach maintains all the related data in a single document, which makes it easy to retrieve and maintain. The whole document can be retrieved in a single query such as –

```
>db.users.findOne({"name":"Tom Benzamin"},{"address":1})
```

- Note that in the above query, **db** and **users** are the database and collection respectively.
- The drawback is that if the embedded document keeps on growing too much in size, it can impact the read/write performance.

Modeling Referenced Relationships

- This is the approach of designing normalized relationship. In this approach, both the user and address documents will be maintained separately but the user document will contain a field that will reference the address document's **id** field.

```
{
 "_id":ObjectId("52ffc33cd85242f436000001"),
 "contact": "987654321",
 "dob": "01-01-1991",
 "name": "Tom Benzamin",
 "address_ids": [
   ObjectId("52ffc4a5d85242602e000000"),
   ObjectId("52ffc4a5d85242602e000001")
 ]
}
```

- As shown above, the user document contains the array field **address_ids** which contains ObjectIds of corresponding addresses. Using these ObjectIds, we can query the address documents and get address details from there. With this approach, we will need two queries: first to fetch the **address_ids** fields from **user** document and second to fetch these addresses from **address** collection.

```
>var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})
>var addresses =
db.address.find({"_id":{"$in":result["address_ids"]}})
```

5.10.8 MongoDB - Database References

- As seen in the last chapter of MongoDB relationships, to implement a normalized database structure in MongoDB, we use the concept of **Referenced Relationships** also referred to as **Manual References** in which we manually store the referenced document's id inside other document. However, in cases where a document contains references from different collections, we can use **MongoDB DBRefs**.

DBRefs V/s Manual References

- As an example scenario, where we would use DBRefs instead of manual references, consider a database where we are storing different types of addresses (home, office, mailing, etc.) in different collections (address_home, address_office, address_mailing, etc). Now, when a **user** collection's document references an address, it also needs to specify which collection to look into based on the address type. In such scenarios where a document references documents from many collections, we should use DBRefs.

Using DBRefs

There are three fields in DBRefs :

1. **$ref** : This field specifies the collection of the referenced document
2. **$id** : This field specifies the _id field of the referenced document
3. **$db** : This is an optional field and contains the name of the database in which the referenced document lies

- Consider a sample user document having DBRef field **address** as shown in the code snippet –

```
{
 "_id":ObjectId("53402597d852426020000002"),
 "address": {
 "$ref": "address_home",
 "$id": ObjectId("534009e4d852427820000002"),
 "$db": "tutorialspoint"},
```

```
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin"
}
```

- The **address** DBRef field here specifies that the referenced address document lies in **address_home** collection under **niralipublications** database and has an id of 534009e4d852427820000002.
- The following code dynamically looks in the collection specified by **$ref** parameter (**address_home** in our case) for a document with id as specified by **$id** parameter in DBRef.

```
>var user = db.users.findOne({"name":"Tom Benzamin"})
>var dbRef = user.address
>db[dbRef.$ref].findOne({"_id":(dbRef.$id)})
```

- The above code returns the following address document present in **address_home** collection –

```
{
  "_id" : ObjectId("534009e4d852427820000002"),
  "building" : "22 A, Indiana Apt",
  "pincode" : 123456,
  "city" : "Los Angeles",
  "state" : "California"
}
```

5.10.9 MongoDB - Covered Queries

- In this chapter, we will learn about covered queries.

What is a Covered Query?

- As per the official MongoDB documentation, a covered query is a query in which –
- All the fields in the query are part of an index.
- All the fields returned in the query are in the same index.
- Since all the fields present in the query are part of an index, MongoDB matches the query conditions and returns the result using the same index without actually looking inside the documents. Since indexes are present in RAM, fetching data from indexes is much faster as compared to fetching data by scanning documents.

Using Covered Queries

- To test covered queries, consider the following document in the **users** collection –

```
{
  "_id": ObjectId("53402597d852426020000002"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "gender": "M",
  "name": "Tom Benzamin",
  "user_name": "tombenzamin"
}
```

- We will first create a compound index for the **users** collection on the fields **gender** and **user_name** using the following query –

```
>db.users.ensureIndex({gender:1,user_name:1})
```

- Now, this index will cover the following query –

```
>db.users.find({gender:"M"},{user_name:1,_id:0})
```

- That is to say that for the above query, MongoDB would not go looking into database documents. Instead it would fetch the required data from indexed data which is very fast.
- Since our index does not include **_id** field, we have explicitly excluded it from result set of our query, as MongoDB by default returns _id field in every query. So the following query would not have been covered inside the index created above –

```
>db.users.find({gender:"M"},{user_name:1})
```

- Lastly, remember that an index cannot cover a query if
  - ➢ Any of the indexed fields is an array
  - ➢ Any of the indexed fields is a subdocument

5.10.10 MongoDB - Analyzing Queries

- Analyzing queries is a very important aspect of measuring how effective the database and indexing design is. We will learn about the frequently used **$explain** and **$hint** queries.

Using $explain

- The **$explain** operator provides information on the query, indexes used in a query and other statistics. It is very useful when analyzing how well your indexes are optimized.
- In the last chapter, we had already created an index for the **users** collection on fields **gender** and **user_name** using the following query –

```
>db.users.ensureIndex({gender:1,user_name:1})
```

- We will now use **$explain** on the following query –

```
>db.users.find({gender:"M"},{user_name:1,_id:0}).explain()
```

The above explain() query returns the following analyzed result –

```
{
  "cursor" : "BtreeCursor gender_1_user_name_1",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 0,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 0,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : true,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
    "gender" : [
      [
        "M",
        "M"
      ]
    ],
    "user_name" : [
      [
        {
          "$minElement" : 1
        },
        {
          "$maxElement" : 1
        }
      ]
    ]
  }
}
```

We will now look at the fields in this result set –

- The true value of **indexOnly** indicates that this query has used indexing.
- The **cursor** field specifies the type of cursor used. BTreeCursor type indicates that an index was used and also gives the name of the index used. BasicCursor indicates that a full scan was made without using any indexes.
- **n** indicates the number of documents matching returned.
- **nscannedObjects** indicates the total number of documents scanned.
- **nscanned** indicates the total number of documents or index entries scanned.

Using $hint

- The **$hint** operator forces the query optimizer to use the specified index to run a query. This is particularly useful when you want to test performance of a query with different indexes. For example, the following query specifies the index on fields **gender** and **user_name** to be used for this query –

```
>db.users.find({gender:"M"},{user_name:1,_id:0}).hint({gender:1,user_name:1})
```

- To analyze the above query using $explain –

```
>db.users.find({gender:"M"},{user_name:1,_id:0}).hint({gender:1,user_name:1}).explain()
```

5.10.11 MongoDB - Atomic Operations

Model Data for Atomic Operations

- The recommended approach to maintain atomicity would be to keep all the related information, which is frequently updated together in a single document using **embedded documents**. This would make sure that all the updates for a single document are atomic.
- Consider the following products document –

```
{
  "_id":1,
  "product_name": "Samsung S3",
  "category": "mobiles",
  "product_total": 5,
  "product_available": 3,
  "product_bought_by": [
    {
      "customer": "john",
      "date": "7-Jan-2014"
    },
    {
      "customer": "mark",
      "date": "8-Jan-2014"
    }
  ]
}
```

- In this document, we have embedded the information of the customer who buys the product in the **product_bought_by** field. Now, whenever a new customer buys the product, we will first check if the product is still available using **product_available** field. If available, we will reduce the value of product_available field as well as insert the new customer's embedded document in the product_bought_by field. We will use **findAndModify** command for this functionality because it searches and updates the document in the same go.

```
>db.products.findAndModify({
  query:{_id:2,product_available:{$gt:0}},
  update:{
    $inc:{product_available:-1},

$push:{product_bought_by:{customer:"rob",date:"9-
Jan-2014"}}
  }
})
```

- Our approach of embedded document and using findAndModify query makes sure that the product purchase information is updated only if it the product is available. And the whole of this transaction being in the same query, is atomic.
- In contrast to this, consider the scenario where we may have kept the product availability and the information on who has bought the product, separately. In this case, we will first check if the product is available using the first query. Then in the second query we will update the purchase information. However, it is possible that between the executions of these two queries, some other user has purchased the product and it is no more available. Without knowing this, our second query will update the purchase information based on the result of our first query. This will make the database inconsistent because we have sold a product which is not available.

5.10.12 MongoDB - Advanced Indexing

- Consider the following document of the **users** collection –

```
{
  "address": {
    "city": "Los Angeles",
    "state": "California",
    "pincode": "123"
  },
  "tags": [
    "music",
    "cricket",
    "blogs"
  ],
  "name": "Tom Benzamin"
}
```

- The above document contains an **address sub-document** and a **tags array**.

Indexing Array Fields

- Suppose we want to search user documents based on the user's tags. For this, we will create an index on tags array in the collection.
- Creating an index on array in turn creates separate index entries for each of its fields. So in our case when we create an index on tags array, separate indexes will be created for its values music, cricket and blogs.
- To create an index on tags array, use the following code –

```
>db.users.ensureIndex(["tags":1])
```

- After creating the index, we can search on the tags field of the collection like this –

```
>db.users.find({tags:"cricket"})
```

To verify that proper indexing is used, use the following **explain** command –

```
>db.users.find({tags:"cricket"}).explain()
```

- The above command resulted in "cursor" : "BtreeCursor tags_1" which confirms that proper indexing is used.

Indexing Sub-Document Fields

- Suppose that we want to search documents based on city, state and pincode fields. Since all these fields are part of address sub-document field, we will create an index on all the fields of the sub-document.
- For creating an index on all the three fields of the sub-document, use the following code –

```
>db.users.ensureIndex(["address.city":1,"address.st
ate":1,"address.pincode":1])
```

- Once the index is created, we can search for any of the sub-document fields utilizing this index as follows –

```
>db.users.find({"address.city":"Los Angeles"})
```

- Remember that the query expression has to follow the order of the index specified. So the index created above would support the following queries –

```
>db.users.find({"address.city":"Los
Angeles","address.state":"California"})
```

- It will also support the following query –

```
>db.users.find({"address.city":"LosAngeles","addres
s.state":"California",
"address.pincode":"123"})
```

MongoDB - Indexing Limitations

- In this chapter, we will learn about Indexing Limitations and its other components.

Extra Overhead

- Every index occupies some space as well as causes an overhead on each insert, update and delete. So if you rarely use your collection for read operations, it makes sense not to use indexes.

RAM Usage

- Since indexes are stored in RAM, you should make sure that the total size of the index does not exceed the RAM limit. If the total size increases the RAM size, it will start deleting some indexes, causing performance loss.

Query Limitations

- Indexing can't be used in queries which use –
  - Regular expressions or negation operators like $nin, $not, etc.
  - Arithmetic operators like $mod, etc.
  - $where clause
- Hence, it is always advisable to check the index usage for your queries.

Index Key Limits

- Starting from version 2.6, MongoDB will not create an index if the value of existing index field exceeds the index key limit.

Inserting Documents Exceeding Index Key Limit

- MongoDB will not insert any document into an indexed collection if the indexed field value of this document exceeds the index key limit. Same is the case with mongorestore and mongoimport utilities.

Maximum Ranges

- A collection cannot have more than 64 indexes.
- The length of the index name cannot be longer than 125 characters.
- A compound index can have maximum 31 fields indexed.

5.10.13 MongoDB - ObjectId

- We have been using MongoDB Object Id in all the previous chapters. In this chapter, we will understand the structure of ObjectId.
- An **ObjectId** is a 12-byte BSON type having the following structure –
  - The first 4 bytes representing the seconds since the unix epoch
  - The next 3 bytes are the machine identifier
  - The next 2 bytes consists of **process id**
  - The last 3 bytes are a random counter value
- MongoDB uses ObjectIds as the default value of _id field of each document, which is generated while the creation of any document. The complex combination of ObjectId makes all the _id fields unique.

Creating New ObjectId

To generate a new ObjectId use the following code –

```
>newObjectId = ObjectId()
```

- The above statement returned the following uniquely generated id –

```
ObjectId("5349b4ddd2781d08c09890f3")
```

- Instead of MongoDB generating the ObjectId, you can also provide a 12-byte id –

```
>myObjectId =
ObjectId("5349b4ddd2781d08c09890f4")
```

Creating Timestamp of a Document

- Since the _id ObjectId by default stores the 4-byte timestamp, in most cases you do not need to store the creation time of any document. You can fetch the creation time of a document using getTimestamp method –

```
>ObjectId("5349b4ddd2781d08c09890f4").getTimesta
mp()
```

- This will return the creation time of this document in ISO date format –

```
ISODate("2014-04-12T21:49:17Z")
```

Converting ObjectId to String

- In some cases, you may need the value of ObjectId in a string format. To convert the ObjectId in string, use the following code –

```
>newObjectId.str
```

- The above code will return the string format of the Guid –

```
5349b4ddd2781d08c09890f3
```

5.10.14 MongoDB - Map Reduce

- As per the MongoDB documentation, **Map-reduce** is a data processing paradigm for condensing large volumes of data into useful aggregated results. MongoDB uses **mapReduce** command for map-reduce operations. MapReduce is generally used for processing large data sets.

MapReduce Command

- Following is the syntax of the basic mapReduce command –

```
>db.collection.mapReduce(
  function() {emit(key,value);},  //map function
  function(key,values)
    {return reduceFunction},
    {   //reduce function
    out: collection,
    query: document,
    sort: document,
    limit: number
  }
)
```

- The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs, which is then reduced based on the keys that have multiple values.
- In the above syntax –
  - ➢ **map** is a javascript function that maps a value with a key and emits a key-value pair
  - ➢ **reduce** is a javascript function that reduces or groups all the documents having the same key
  - ➢ **out** specifies the location of the map-reduce query result
  - ➢ **query** specifies the optional selection criteria for selecting documents
  - ➢ **sort** specifies the optional sort criteria
  - ➢ **limit** specifies the optional maximum number of documents to be returned

Using MapReduce

- Consider the following document structure storing user posts. The document stores user_name of the user and the status of post.

```
{
  "post_text": "niralipublications is an awesome website for tutorials",
  "user_name": "mark",
  "status":"active"
}
```

- Now, we will use a mapReduce function on our **posts** collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user using the following code –

```
>db.posts.mapReduce(
  function() { emit(this.user_id,1); },
  function(key, values) {return Array.sum(values)}, {
    query:{status:"active"},
    out:"post_total"
  }
)
```

- The above mapReduce query outputs the following result –

```
{
  "result" : "post_total",
  "timeMillis" : 9,
  "counts" : {
    "input" : 4,
    "emit" : 4,
    "reduce" : 2,
    "output" : 2
  },
  "ok" : 1,
}
```

- The result shows that a total of 4 documents matched the query (status:"active"), the map function emitted 4 documents with key-value pairs and finally the reduce function grouped mapped documents having the same keys into 2.
- To see the result of this mapReduce query, use the find operator –

```
>db.posts.mapReduce(
  function() { emit(this.user_id,1); },
  function(key, values) {return Array.sum(values)}, {
    query:{status:"active"},
    out:"post_total"
  }
).find()
```

- The above query gives the following result which indicates that both users **tom** and **mark** have two posts in active states –

```
{ "_id" : "tom", "value" : 2 }
{ "_id" : "mark", "value" : 2 }
```

- In a similar manner, MapReduce queries can be used to construct large complex aggregation queries. The use of custom Javascript functions make use of MapReduce which is very flexible and powerful.

5.10.15 MongoDB - Text Search

- Starting from version 2.4, MongoDB started supporting text indexes to search inside string content. The **Text Search** uses stemming techniques to look for specified words in the string fields by dropping stemming stop words like **a, an, the,** etc. At present, MongoDB supports around 15 languages.

Enabling Text Search

- Initially, Text Search was an experimental feature but starting from version 2.6, the configuration is enabled by default. But if you are using the previous version of MongoDB, you have to enable text search with the following code –

```
>db.adminCommand({setParameter:true,textSearchEnabled:true})
```

Creating Text Index

- Consider the following document under **posts** collection containing the post text and its tags –

```
{
  "post_text": "enjoy the mongodb articles on
niralipublications",
  "tags": [
    "mongodb",
    "niralipublications"
  ]
}
```

- We will create a text index on post_text field so that we can search inside our posts' text –

```
>db.posts.ensureIndex({post_text:"text"})
```

Using Text Index

- Now that we have created the text index on post_text field, we will search for all the posts having the word **niralipublications** in their text.

```
>db.posts.find({$text:{$search:"niralipublications"}})
```

- The above command returned the following result documents having the word **niralipublications** in their post text –

```
{
  "_id" : ObjectId("53493d14d852429c10000002"),
  "post_text" : "enjoy the mongodb articles on
niralipublications",
  "tags" : [ "mongodb", "niralipublications" ]
}
{
  "_id" : ObjectId("53493d1fd852429c10000003"),
  "post_text" : "writing tutorials on mongodb",
  "tags" : [ "mongodb", "tutorial" ]
}
```

- If you are using old versions of MongoDB, you have to use the following command –

```
>db.posts.runCommand("text",{search:"
niralipublications "})
```

- Using Text Search highly improves the search efficiency as compared to normal search.

Deleting Text Index

- To delete an existing text index, first find the name of index using the following query –

```
>db.posts.getIndexes()
```

- After getting the name of your index from above query, run the following command. Here, **post_text_text** is the name of the index.

```
>db.posts.dropIndex("post_text_text")
```

5.10.16 MongoDB - Regular Expression

- Regular Expressions are frequently used in all languages to search for a pattern or word in any string. MongoDB also provides functionality of regular expression for string pattern matching using the **$regex** operator. MongoDB uses PCRE (Perl Compatible Regular Expression) as regular expression language.
- Unlike text search, we do not need to do any configuration or command to use regular expressions.
- Consider the following document structure under **posts** collection containing the post text and its tags –

```
{
  "post_text": "enjoy the mongodb articles on
niralipublications",
  "tags": [
    "mongodb",
    "niralipublications"
  ]
}
```

Using regex Expression

- The following regex query searches for all the posts containing string **niralipublications** in it –

```
>db.posts.find({post_text:{$regex:"niralipublications"}})
```

The same query can also be written as –

```
>db.posts.find({post_text:/niralipublications/})
```

Using regex Expression with Case Insensitive

- To make the search case insensitive, we use the **$options** parameter with value **$i**. The following command will look for strings having the word **niralipublications**, irrespective of smaller or capital case –

```
>db.posts.find({post_text:{$regex:"niralipublications",$
options:"$i"}})
```

- One of the results returned from this query is the following document which contains the word **niralipublications** in different cases –

```
{
  "_id" : ObjectId("53493d37d852429c10000004"),
  "post_text" : "hey! this is my post on
Niralipublications",
  "tags" : [ "niralipublications" ]
}
```

Using regex for Array Elements

- We can also use the concept of regex on array field. This is particularly very important when we implement the functionality of tags. So, if you want to search for all the posts having tags beginning from the word tutorial (either tutorial or tutorials or tutorialpoint or tutorialphp), you can use the following code –

```
>db.posts.find({tags:{$regex:"tutorial"}})
```

Optimizing Regular Expression Queries

- If the document fields are **indexed**, the query will use make use of indexed values to match the regular expression. This makes the search very fast as compared to the regular expression scanning the whole collection.
- If the regular expression is a **prefix expression**, all the matches are meant to start with a certain string characters. For e.g., if the regex expression is **^tut**, then the query has to search for only those strings that begin with **tut**.

Working with RockMongo

- RockMongo is a MongoDB administration tool using which you can manage your server, databases, collections, documents, indexes, and a lot more. It provides a very user-friendly way for reading, writing, and creating documents. It is similar to PHPMyAdmin tool for PHP and MySQL.

Downloading RockMongo

- You can download the latest version of RockMongo from here: https://github.com/iwind/rockmongo

Installing RockMongo

- Once downloaded, you can unzip the package in your server root folder and rename the extracted folder to **rockmongo**. Open any web browser and access the **index.php** page from the folder rockonongo. Enter admin/admin as username/password respectively.

Working with RockMongo

- We will now be looking at some basic operations that you can perform with RockMongo.

Creating New Database

- To create a new database, click **Databases** tab. Click **Create New Database**. On the next screen, provide the name of the new database and click on **Create**. You will see a new database getting added in the left panel.

Creating New Collection

- To create a new collection inside a database, click on that database from the left panel. Click on the **New Collection** link on top. Provide the required name of the collection. Do not worry about the other fields of Is Capped, Size and Max. Click on **Create**. A new collection will be created and you will be able to see it in the left panel.

Creating New Document

- To create a new document, click on the collection under which you want to add documents. When you click on a collection, you will be able to see all the documents within that collection listed there. To create a new document, click on the **Insert** link at the top. You can enter the document's data either in JSON or array format and click on **Save**.

Export/Import Data

- To import/export data of any collection, click on that collection and then click on **Export/Import** link on the top panel. Follow the next instructions to export your data in a zip format and then import the same zip file to import back data.

5.10.17 MongoDB - GridFS

- **GridFS** is the MongoDB specification for storing and retrieving large files such as images, audio files, video files, etc. It is kind of a file system to store files but its data is stored within MongoDB collections. GridFS has the capability to store files even greater than its document size limit of 16MB.
- GridFS divides a file into chunks and stores each chunk of data in a separate document, each of maximum size 255k.
- GridFS by default uses two collections **fs.files** and **fs.chunks** to store the file's metadata and the chunks. Each chunk is identified by its unique _id ObjectId field. The fs.files serves as a parent document. The **files_id** field in the fs.chunks document links the chunk to its parent.
- Following is a sample document of fs.files collection –

```
{
  "filename": "test.txt",
  "chunkSize": NumberInt(261120),
  "uploadDate": ISODate("2014-04-
13T11:32:33.557Z"),
  "md5": "7b762939321e146569b07f72c62cca4f",
  "length": NumberInt(646)
}
```

- The document specifies the file name, chunk size, uploaded date, and length.

Following is a sample document of fs.chunks document –

```
{
  "files_id": ObjectId("534a75d19f54bfec8a2fe44b"),
  "n": NumberInt(0),
  "data": "Mongo Binary Data"
}
```

Adding Files to GridFS

- Now, we will store an mp3 file using GridFS using the **put** command. For this, we will use the **mongofiles.exe** utility present in the bin folder of the MongoDB installation folder.
- Open your command prompt, navigate to the mongofiles.exe in the bin folder of MongoDB installation folder and type the following code –

```
>mongofiles.exe -d gridfs put song.mp3
```

- Here, **gridfs** is the name of the database in which the file will be stored. If the database is not present, MongoDB will automatically create a new document on the fly. Song.mp3 is the name of the file uploaded. To see the file's document in database, you can use find query –

```
>db.fs.files.find()
```

The above command returned the following document –

```
{
  _id: ObjectId('534a811bf8b4aa4d33fdf94d'),
  filename: "song.mp3",
  chunkSize: 261120,
  uploadDate: new Date(1397391643474), md5:
"e4f53379c909f7bed2e9d631e15c1c41",
  length: 10401959
}
```

- We can also see all the chunks present in fs.chunks collection related to the stored file with the following code, using the document id returned in the previous query –

```
>db.fs.chunks.find({files_id:ObjectId('534e811bf8b4aa4d33fdf94d')})
```

- In my case, the query returned 40 documents meaning that the whole mp3 document was divided in 40 chunks of data.

5.10.18 MongoDB - Capped Collections

- **Capped Collections** are fixed-size circular collections that follow the insertion order to support high performance for create, read, and delete operations. By circular, it means that when the fixed size allocated to the collection is exhausted, it will start deleting the oldest document in the collection without providing any explicit commands.
- Capped collections restrict updates to the documents if the update results in increased document size. Since capped collections store documents in the order of the disk storage, it ensures that the document size does not increase the size allocated on the disk. Capped collections are best for storing log information, cache data, or any other high volume data.

Creating Capped Collection

- To create a capped collection, we use the normal createCollection command but with **capped** option as **true** and specifying the maximum size of collection in bytes.

```
>db.createCollection("cappedLogCollection",{capped:true,size:10000})
```

- In addition to collection size, we can also limit the number of documents in the collection using the **max** parameter –

```
>db.createCollection("cappedLogCollection",{capped:true,size:10000,max:1000})
```

- If you want to check whether a collection is capped or not, use the following **is Capped** command –

```
>db.cappedLogCollection.isCapped()
```

- If there is an existing collection which you are planning to convert to capped, you can do it with the following code –

```
>db.runCommand({"convertToCapped":"posts",size:10000})
```

- This code would convert our existing collection **posts** to a capped collection.

Querying Capped Collection

- By default, a find query on a capped collection will display results in insertion order. But if you want the documents to be retrieved in reverse order, use the **sort** command as shown in the following code –

```
>db.cappedLogCollection.find().sort({$natural:-1})
```

- There are few other important points regarding capped collections worth knowing –
- We cannot delete documents from a capped collection.
- There are no default indexes present in a capped collection, not even on _id field.
- While inserting a new document, MongoDB does not have to actually look for a place to accommodate new document on the disk. It can blindly insert the new document at the tail of the collection. This makes insert operations in capped collections very fast.
- Similarly, while reading documents MongoDB returns the documents in the same order as present on disk. This makes the read operation very fast.

5.10.19 MongoDB - Auto-Increment Sequence

- MongoDB does not have out-of-the-box auto-increment functionality, like SQL databases. By default, it uses the 12-byte ObjectId for the **_id** field as the primary key to uniquely identify the documents. However, there may be scenarios where we may want the _id field to have some auto-incremented value other than the ObjectId.
- Since this is not a default feature in MongoDB, we will programmatically achieve this functionality by using a **counters** collection as suggested by the MongoDB documentation.

Using Counter Collection

- Consider the following **products** document. We want the _id field to be an **auto-incremented integer sequence** starting from 1,2,3,4 upto n.

```
{
 "_id":1,
 "product_name": "Apple iPhone",
 "category": "mobiles"
}
```

- For this, create a **counters** collection, which will keep track of the last sequence value for all the sequence fields.

```
>db.createCollection("counters")
```

- Now, we will insert the following document in the counters collection with **productid** as its key –

```
{
  "_id":"productid",
  "sequence_value": 0
}
```

- The field **sequence_value** keeps track of the last value of the sequence.
- Use the following code to insert this sequence document in the counters collection –

```
>db.counters.insert({_id:"productid",sequence_value:0})
```

Creating Javascript Function

- Now, we will create a function **getNextSequenceValue** which will take the sequence name as its input, increment the sequence number by 1 and return the updated sequence number. In our case, the sequence name is **productid**.

```
>function getNextSequenceValue(sequenceName){
   var sequenceDocument = db.counters.findAndModify({
      query:{_id: sequenceName },
      update: {$inc:{sequence_value:1}},
      new:true
   });
   return sequenceDocument.sequence_value;
}
```

Using the Javascript Function

- We will now use the function getNextSequenceValue while creating a new document and assigning the returned sequence value as document's _id field.
- Insert two sample documents using the following code –

```
>db.products.insert({
   "_id":getNextSequenceValue("productid"),
   "product_name":"Apple iPhone",
   "category":"mobiles"
})
```

```
>db.products.insert({
   "_id":getNextSequenceValue("productid"),
   "product_name":"Samsung S3",
   "category":"mobiles"
})
```

- As you can see, we have used the getNextSequenceValue function to set value for the _id field.
- To verify the functionality, let us fetch the documents using find command –

```
>db.products.find()
```

- The above query returned the following documents having the auto-incremented _id field –

```
{ "_id" : 1, "product_name" : "Apple iPhone", "category" : "mobiles" }
{ "_id" : 2, "product_name" : "Samsung S3", "category" : "mobiles" }
```

EXERCISE

1. Describe the key features of MongoDB.
2. How MongoDB provides database replication? Explain with suitable diagram.
3. Explain the horizontal and vertical scaling in MongoDB.
4. Describe in brief the core server tools.
5. Write short note on
   a. Insert
   b. Updating documents
   c. Operator update
   d. Updating complex data
   e. Deleting the data
6. Write short note on following in the context of MongoDB
   a. Creating a large collection
   b. Range Queries
7. Describe the principles of Schema Design
8. Explain the methods to query the document from MongoDB collection.

9. Write a short note on
   a. Documents in MongoDB
   b. Collections in MongoDB
10. Explain the following MongoDB queries with syntax and suitable examples
    a. create
    b. insert
    c. drop
    d. find
    e. conditional
    f. update
    g. limit
    h. sort
    i. aggregate

Model Question Papers for End-Semester Examination

PAPER I

Time : 3 Hours **Max. Marks : 60**

Instructions to the Candidates :

1. Each Question carries 12 Marks.
2. Attempt any five questions to the following.
3. Illustrate your answers with neat sketches, diagram etc., wherever necessary.
4. If some part or parameter is noticed to be missing, you may appropriately assume it and should mention it clearly.

Attempt Any Five Questions

1. Attempt any Two of the Following :

(a) Describe the characteristics of Big Data with suitable examples. **[6]**

(b) Explain in brief any three key technologies that enable Big Data for businesses. **[6]**

(c) What is YARN? Draw and explain the YARN architecture. **[6]**

2. Attempt any three of the following.

(a) State and explain various sources of big data. **[4]**

(b) What are the advantages of Big Data? **[4]**

(c) What are the goals of HDFS? **[4]**

(d) List and describe various HDFS operations. **[4]**

3. Attempt the following.

(a) How MapReduce is used for page ranking problem? **[6]**

(b) What is ZooKeeper? Explain the functionality of ZooKeeper in Hadoop ecosystem. **[6]**

4. Attempt any two of the following.

(a) Draw and explain the modern streaming architecture. Also state the benefits of a modern streaming architecture. **[6]**

(b) Draw and explain the Lambda architecture of real time Big Data pipeline. **[6]**

(c) Describe in brief the concept of streaming ecosystem. **[6]**

5. Attempt any two of the following.

(a) Describe any four applications of Deep Learning for Big Data. **[6]**

(b) Mention the key challenges of Deep Learning and Big Data. **[6]**

(c) What is graph processing? Mention the real time applications of graph processing. **[6]**

6. Attempt the following.

(a) How MongoDB provides database replication? Explain with suitable diagram. **[6]**

(b) Write short note on **[6]**

(i) Insert (ii) Updating documents (iii) Operator updated.

❖ ❖ ❖

PAPER II

Time : 3 Hours **Max. Marks : 60**

Instructions to the candidates :

1. Each Question carries 12 Marks.
2. Attempt any five questions to the following.
3. Illustrate your answers with neat sketches, diagram etc., wherever necessary.
4. If some part or parameter is noticed to be missing, you may appropriately assume it and should mention it clearly.

Attempt any Five Questions

1. Attempt any Two of the Following :

(a) Mention any six challenges of Big Data. **[6]**

(b) Mention various Big Data Distribution package. **[6]**

(c) Explain in brief any three key technologies that enable Big Data for businesses. **[6]**

2. Attempt any two of the following.

(a) Draw and explain the application workflow in HADOOP YARN. **[6]**

(b) What is MapReduce? Explain the components of MapReduce. **[6]**

(c) Draw and explain the MapReduce programming model with SPARK. **[6]**

3. Attempt the following.

(a) What is Paxos in Hadoop? How it can be used to solve the consensus problem? **[6]**

(b) What is Cassandra? Differentiate between NoSQL vs. Relational Database **[6]**

4. Attempt the following.

(a) What is streaming of the data? Give the typical examples of the streaming data. **[6]**

(b) Write short note on: Big Data streaming platforms for fast data. **[6]**

5. Attempt any two of the following.

(a) Explain the role of machine learning techniques in big data processing. **[6]**

(b) State and explain the machine learning algorithms in Apache Spark. **[6]**

(c) Explain the following Big Data Machine Learning algorithms in Mahout **[6]**

(i) K-means clustering

(ii) Naïve Bayes classification

6. Attempt the following.

(a) Explain the methods to query the document from MongoDB collection. **[6]**

(b) Write a short note on **[6]**

(i) Documents in MongoDB

(ii) Collections in MongoDB

♣ ♣ ♣