Data Engineering with Apache Spark

Thesis submitted in partial fulfillment of requirements
For the degree of
Master of Computer Application
of
Computer Science and Engineering Department
of
Jadavpur University

by

Subhankar Karmakar
Registration No. - 149864 of 2019-2020
Exam Roll No. - MCA226001

under the supervision of

Mridul Sankar Barik
Assistant Professor

Department of Computer Science and Engineering
Jadavpur University
Kolkata, West Bengal, India
2022

Certificate from the Supervisor

This is to certify that the work embodied in this thesis entitled **"Data Engineering with Apache Spark"** has been satisfactorily completed by **Subhankar Karmakar** (Registration Number 149864 of 2019-2020; Class Roll No. 001910503001; Examination Roll No. MCA226001). It is a bonafide piece of work carried out under my supervision and guidance at Jadavpur University, Kolkata for partial fulfilment of the requirements for the awarding of the **Master of Computer Application** degree of the Department of Computer Science and Engineering, Faculty of Engineering & Technology, Jadavpur University, during the academic year $2021 - 22$.

Mridul Sankar Barik,
Assistant Professor,
Department of Computer Science and Engineering,
Jadavpur University.
(Supervisor)

Forwarded By:

Prof. Anupam Sinha,
Head,
Department of Computer Science and Engineering,
Jadavpur University.

Prof. Chandan Mazumdar,
Dean,
Faculty of Engineering & Technology,
Jadavpur University.

Department of Computer Science and Engineering
Faculty of Engineering And Technology
Jadavpur University, Kolkata - 700 032

Certificate of Approval

This is to certify that the thesis entitled **"Data Engineering with Apache Spark"** is a bonafide record of work carried out by **Subhankar Karmakar** (Registration Number 149864 of 2019-2020; Class Roll No. 001910503001; Examination Roll No. MCA226001) in partial fulfillment of the requirements for the award of the degree of **Master of Computer Application** in the **Department of Computer Science and Engineering**, during the period of January 2021 to June 2022. It is understood that by this approval, the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the thesis only for the purpose of which it has been submitted.

Examiners:

(Signature of The Examiner)

(Signature of The Supervisor)

Department of Computer Science and Engineering
Faculty of Engineering & Technology
Jadavpur University, Kolkata - 700 032

Declaration of Originality and Compliance of Academic Ethics

I hereby declare that the thesis entitled **"Data Engineering with Apache Spark"** contains literature survey and original research work by the undersigned candidate, as a part of his degree of **Master of Computer Application** in the **Department of Computer Science and Engineering, Jadavpur University**. All information have been obtained and presented in accordance with academic rules and ethical conduct.

I also declare that, as required by these rules and conduct, I have fully cited and referenced all materials and results that are not original to this work.

Name: Subhankar Karmakar

Examination Roll No.: MCA226001

Registration No.: 149864 of 2019-2020

Thesis Title: Data Engineering with Apache Spark

Signature of the Candidate:

ACKNOWLEDGEMENT

I am pleased to express my gratitude and regards towards my Project Guide **Shri Mridul Sankar Barik**, Assistant Professor, Department of Computer Science and Engineering, Jadavpur University, without whose valuable guidance, inspiration and attention towards me, pursuing my project would have been impossible.

Last but not the least, I express my regards towards my friends and family for bearing with me and for being a source of constant motivation during the entire term of the work.

Subhankar Karmakar
MCA Final Year
Exam Roll No. - MCA226001
Regn. No. - 149864 of 2019-2020
Department of Computer Science and Engineering,
Jadavpur University.

Contents

List of Figures

Abstract

In this era of information, a large amount of data is easily available on hands of scientists and decision makers but the problem is that, they come not only in large volume but also in high variety, velocity, and as soon as they are available then veracity and value of the available data needs to be analyzed to make some decisions on that. So, before analyzing it for taking decisions, another big challenge is to manage this high volume of data, extracting data from various resources, transforming and cleaning the data in a structured format, and loading it to data warehouses to make them available to Data Scientists, Data Analysts. This process is known as ETL which stands for Extract, Transform and Load. Here comes the role of Data Engineers, as a separate category of experts in the world of data science. Over the years, a large number of data tools and products has been evolved and among all of them Apache Spark has been evolved as a best friend of data engineers for a few years. Apache Spark is one of the most widely used open source processing framework for big data, it allows one to process large datasets in parallel using a large number of compute nodes which make Spark as a unified general-purpose distributed data processing engine. This thesis aims to give a brief introduction to how spark works internally and some examples which will help to depict an idea how data engineers manages big data. In this thesis DataBricks, a cloud solution leveraging Spark processing engine, is used with PySpark, Sparks's python APIs.

Chapter 1

Introduction

1.1 What is Big Data

The term ‘Big Data’ has been in use since the early 1990s. Although it is not exactly known who first used the term, but most people give credit to **John R. Mashey** (who at the time worked at **Silicon Graphics**) for making the term popular.

In its true essence, Big Data is not something that is completely new or only of the last two decades. Over the course of centuries, people have been trying to use data analysis and analytics techniques to support their decision-making process. The ancient *Egyptians* around $300BC$ already tried to capture all existing ‘data’ in the library of *Alexandria*. Moreover, the Roman Empire used to carefully analyze statistics of their military to determine the optimal distribution for their armies.

Big Data[1] refers to data that is so large, fast or complex that it’s difficult or impossible to process using traditional methods. The act of accessing and storing large amounts of information for analytics has been around for a long time. But the concept of big data gained momentum in the early 2000s, when industry analyst Doug Laney articulated the now-mainstream definition of big data as the three V’s:

1.2 The Five Vs of Big Data

Volume. One of the key concepts to get to know when it comes to big data is volume.There are a lot of companies and businesses that consume so much data, maybe because they have a lot of users or because they use AI to feed this data. This includes smart devices in our homes that are constantly learning and taking in their surroundings or something like Uber, which has millions of customers at any time – and this is all adding lots of data to the mix.

To better explain this, let’s take a look at some real-life examples, such as Facebook, that stores photographs.

Overall, it is thought Facebook stores a whopping 250 billion images in total. This isn’t even taking into account things such as Facebook posts – where there is thought to be approximately 2.5 trillion (and that is only from 2016 onwards).

Organizations collect data from a variety of sources, including transactions, smart (IoT) devices, industrial equipment, videos, images, audio, social media and more. In the past, storing all that data would have been too costly – but cheaper storage using data lakes, Hadoop, and the cloud have eased the burden.

Velocity. With big data, velocity refers to how quickly data is coming in. With the increasing growth of the Internet of Things(IoT), data streams into businesses at an unprecedented speed and must be handled in a timely manner. RFID tags, sensors and smart meters are driving the need to deal with these torrents of data in near-real time.

Using the Facebook example from earlier, whilst the social media giant stores 250 billion images, approximately 900 million photos are uploaded by its users each and every day.

With this colossal amount of data coming in every day, this has to be processed, filed and retrieved.

Another example of velocity is sensor data. With the Internet of Things taking off at a dramatic rate, we will be seeing more and more connected sensors. This will effectively mean more data being transmitted almost all the time.

Variety. The third sector of big data is variety. When discussing variety in reference to big data, it means that the data can be very different from one application to another, much of it being unstructured data too.

All the data does not necessarily fit into one neat database application as it may have in the past.Data comes in all types of formats – from structured, numeric data in traditional databases to unstructured text documents, emails, videos, audio, stock ticker data and financial transactions.

Veracity. The fourth and another important aspect of big data is Veracity. This refers to the quality and the origin of big data and its conformity to facts and accuracy. Some of the key attributes that come up while talking about the veracity of big data are consistency, completeness, integrity, and ambiguity. The drivers include cost and the need for traceability.

Value. The fifth and finale aspect of the big data can be focused upon the value of the data. Value refers to our need and ability to turn data into value. Value is not only connected with the profitability comes from the data, but also It might have medical or social benefits as well as customer, employee, or personal satisfaction. This last V is one of the main reasons why people invest time into big data. They are looking to derive value from it.

1.3 Types of Big Data

Data usually comes in various formats, but they mainly fall under the following three broad categories.

1.3.1 Structured Data

Data that can be stored, accessed and processed in the form of fixed-format is termed as 'structured data.' Since this data comes in a similar format, businesses get the maximum out of it by performing analysis. Various advanced technologies are also invented to extract data-driven decisions from structured data. However, the world is going towards an extent where the creation of structured data is ballooning too much, as it has already reached the zettabytes mark.

1.3.2 Unstructured Data

Any data that comes in an unknown form or structure falls under unstructured data. Processing unstructured data and analyzing them to get data-driven answers is a challenging task as they are from different categories and outing them together will only make things worse. A heterogeneous data source containing a combination of simple text files, images, videos, etc. is an example of unstructured data.

1.3.3 Semi-structured Data

Semi-structured data has both structured and unstructured data in it. We can see semi-structured data as structured in form, but it is actually not defined with table definition in relational DBMS. Web application data is an example of semi-structured data. It has unstructured data like log files, transaction history files, etc. OLTP systems are built to work with structured data, wherein data is stored in relations.

1.4 Evolution of Big Data

To illustrate the development in the field of big data over time, the evolution of Big Data can roughly be subdivided into three main phases. Each phase has its own characteristics and capabilities.

1.4.1 Phase 1.0

Data analysis, data analytics and Big Data originate from the long-standing domain of database management. It relies heavily on the storage, extraction, and optimization techniques that are common in data that is stored in Relational Database Management Systems (RDBMS).

Database management and data warehousing are considered the core components of Big Data Phase 1. It provides the foundation of modern data analysis as we know it today, using well-known techniques such as database queries, online analytical processing and standard reporting tools.

1.4.2 Phase 2.0

Since the early 2000s, the Internet and the Web began to offer unique data collections and data analysis opportunities. With the expansion of web traffic and online stores, companies such as Yahoo, Amazon, and eBay started to analyze customer behavior by analyzing click-rates, IP-specific location data and search logs. This opened a whole new world of possibilities.

From a data analysis, data analytics, and Big Data point of view, HTTP-based web traffic introduced a massive increase in semi-structured and unstructured data. Besides the standard structured data types, organizations now needed to find new approaches and storage solutions to deal with these new data types in order to analyze them effectively. The arrival and growth of social media data greatly aggravated the need for tools, technologies, and analytics techniques that were able to extract meaningful information out of this unstructured data.

1.4.3 Phase 3.0

Although web-based unstructured content is still the main focus for many organizations in data analysis, data analytics, and big data, the current possibilities to retrieve valuable information are emerging out of mobile devices.

Mobile devices not only give the possibility to analyze behavioral data (such as clicks and search queries), but also give the possibility to store and analyze location-based data (GPS-data). With the advancement of these mobile devices, it is possible to track movement, analyze physical behavior and even health-related data (number of steps you take per day). This data provides a whole new range of opportunities, from transportation, to city design and health care.

Simultaneously, the rise of sensor-based internet-enabled devices is increasing the data generation like never before. Famously coined as the 'Internet of Things' (IoT), millions of TVs, thermostats, wearables, and even refrigerators are now generating zettabytes of data every day. And the race to extract meaningful and valuable information out of these new data sources has only just begun.

1.5 Thesis Objective

The objective of the thesis is to explore and experiment with one of the most trending big data tools now a days, that are currently available in the market, naming Apache Spark(a unified general purpose distributed high performant parallel processing engine) and Data Bricks(a cloud platform which actually owned by the creators of Apache Spark, leveraging the all the Spark Functionalities under the same hood) through a Data Engineer's point of view.

1.6 Thesis Contribution

As Apache Spark is pretty new in the Big Data world and continuously evolving with new features, I learned and explored a lot and read relevant journals and have gone through all the official documentations of Data Bricks and PySpark for implementation purpose. I talked to some of the data engineers to get an idea about the work they used to do in their day to day routine. I have used that knowledge to depict some data engineering task in this thesis as one can go further by reading this by summing up all relevant knowledge under the same hood from different sources I learnt and gathered.

1.7 Thesis Outline

- In the **Chapter 2**, headed as **Big Data Analytics** of this thesis, the relation between Big Data and Data Analytics is depicted briefly, by describing
  1. The need of Big Data analytics in current days explaining how it helps large organizations.
  2. Different types of point of views to see Big Data Analytics like descriptive analytics, diagnostic analytics, predictive analytics and perspective analytics
  3. How a Data Analytics task is performed step by step
  4. Various tools and technologies that are being widely used in current days
  5. what are some challenges to handle and perform analytics tasks with big data.
- In the **Chapter 3**, headed as **Apache Spark** of this thesis, what is spark and why it is gaining so much popularity, these two main questions are depicted by defining spark, describing its inter working, its design philosophy, some of very important terminologies used frequently while talking about spark, various components of spark ecosystem etc in brief. Basically, chapter 3 is the theoretical overview of one of the most trending big data tool, Apache Spark.

- In the **Chapter 4**, headed as **An overview of Databricks Cloud Platform to use Apache Spark functionalities**, giving a brief introduction to what actually Databricks is, it is described well enough to get started with the platform to test and experiment with data engineering tasks mentioned in **Chapter 5**.

- In the **Chapter 5**, headed as **Sample Data Engineering Tasks with the help of Data Bricks and PySpark**, some sample data engineering task are described along with the description of the data set used and implementation codes, as it can intuitively be inferred from the name of the chapter.

- **Chapter 6** is the last chapter for drawing conclusion and describing future possibilities that can be worked upon having the knowledge of this thesis.

Chapter 2

Big Data Analytics

2.1 Introduction

Big data analytics[6] describes the process of uncovering trends, patterns, and correlations in large amounts of raw data to help make data-informed decisions. Analysis of big data allows analysts, researchers, and business users to make better and faster decisions using data that was previously inaccessible or unusable. Businesses can use advanced analytics techniques such as text analytics, machine learning, predictive analytics, data mining, statistics and natural language processing to gain new insights from previously untapped data sources independently or together with existing enterprise data. With the explosion of data, early innovation projects like Hadoop, Spark, and NoSQL databases were created for the storage and processing of big data. This field continues to evolve as data engineers look for ways to integrate the vast amounts of complex information created by sensors, networks, transactions, smart devices, web usage, and more. Even now, big data analytics methods are being used with emerging technologies, like machine learning, to discover and scale more complex insights.

2.2 Need of Big Data Analytics

Big data analytics is a combination of multiple advanced technologies that work together to help business organizations use the best set of technologies to get the best value out of their data. Some of these technologies are machine learning, data mining, data management, Hadoop, Spark, NoSQL Databases etc.

Some core strengths of Big Data Analytics are discussed below.

2.2.1 Cost Reduction

Big data analytics offers data-driven insights for the business stakeholders and they can take better strategic decisions, streamline and optimize the operational processes and understand their customers better. All these helps in cost-cutting and adds efficiency to the business model. Big data analytics also streamline the supply chains to reduce time, effort, and resource consumption.

2.2.2 Reliable and Continuous Data

As big data analytics allows business enterprises to make use of organizational data, they don't have to rely upon third-party market research or tools for the same. Further, as the organizational data expands continually, having a reliable and robust big data analytics platform ensures reliable and continuous data streams.

2.2.3 New Products and Services

Because of the availability of a set of diverse and advanced technologies in the form of big data analytics, you can take better decisions related to developing new products and services. Also, using big data analysis, one always have the best market and customer or end-user insights to steer the development processes in the right direction.

Hence, big data analytics also facilitates faster decision-making stemming from data-driven actionable insights.

2.2.4 Improved Efficiency

Big data analytics improves accuracy, efficiency, and overall decision-making in business organizations. You can analyze the customer behavior via the shopping data and leverage the power of predictive analytics to make certain calculations, such as checkout wait times, etc. Stats reveal that 38% of companies use big data for organizational efficiency.

2.2.5 Better Monitoring and Tracking

Big data analytics also empowers organizations with real-time monitoring and tracking functionalities and amplifies the results by suggesting the appropriate actions or strategizing nudges stemming from predictive data analytics.

These tracking and monitoring capabilities are of extreme importance in:

- Security posture management
- Mitigating cybersecurity attacks and minimizing the damage
- Database backup
- IT infrastructure management

2.2.6 Better Remote Resource Management

Be it hiring or remote team management and monitoring, big data analytics offers a wide range of capabilities to enterprises. Big data analytics can empower business owners with core insights to make better decisions regarding employee tracking, employee hiring, performance management, etc. This remote-resource-management capability works well for IT infrastructure management as well.

2.2.7 Taking Right Organizational Decisions

Have a look at the following hierarchy that shows how big data analytics can help companies take better and data-driven organizational decisions.

- Level 4: Optimized Talent Acquisition
    - ∗ Strategic Enabler of the Business
    - ∗ Mitigating cybersecurity attacks and minimizing the damage
    - ∗ Database backup
    - ∗ IT infrastructure management
- Level 3: Strategic Analytics
- Level 2: Proactive Advanced Reporting
- Level 1: Reactive Operational Reporting

2.3 Types of Big Data Analytics

Types[4] of Big Data Analytics can be classified into four major parts, as discussed below.

2.3.1 Descriptive Analytics (What Happened and When)

Descriptive Analytics is considered a useful technique for uncovering patterns within a certain segment of customers. It simplifies the data and summarizes past data into a readable form. Descriptive analytics provide insights into what has occurred in the past and with the trends to dig into for more detail. This helps in creating reports like a company's revenue, profits, sales, and so on.

Use Case: The Dow Chemical Company analyzed its past data to increase facility utilization across its office and lab space. Using descriptive analytics, Dow was able to identify underutilized space. This space consolidation helped the company save nearly US $4 million annually.

2.3.2 Diagnostic Analytics (Where and How it Happened)

Diagnostic Analytics, as the name suggests, gives a diagnosis to a problem. It gives a detailed and in-depth insight into the root cause of a problem. Data scientists turn to this analytics craving for the reason behind a particular happening. Techniques like drill-down, data mining, and data recovery, churn reason analysis, and customer health score analysis are all examples of diagnostic analytics. In business terms, diagnostic analytics is useful when you are researching the reasons leading churn indicators and usage trends among your most loyal customers.

Use Case: An e-commerce company's report shows that their sales have gone down, although customers are adding products to their carts. This can be due to various reasons like the form didn't load correctly, the shipping fee is too high, or there are not enough payment options available. This is where you can use diagnostic analytics to find the reason.

2.3.3 Predictive analytics (What Will Happen and How)

This type of analytics looks into the historical and present data to make predictions of the future. Predictive analytics uses data mining, AI, and machine learning to analyze current data and make predictions about the future. It works on predicting customer trends, market trends, and so on.

Use Case: PayPal determines what kind of precautions they have to take to protect their clients against fraudulent transactions. Using predictive analytics, the company uses all the historical payment data and user-behavior data and builds an algorithm that predicts fraudulent activities.

2.3.4 Prescriptive Analytics (What Should We Do)

Prescriptive analytics is the most valuable yet underused form of analytics. It is the next step in predictive analytics. The prescriptive analysis explores several possible actions and suggests actions depending on the results of descriptive and predictive analytics of a given dataset.

Prescriptive analytics is a combination of data and various business rules. The data of prescriptive analytics can be both internal (organizational inputs) and external (social media insights).

Prescriptive analytics allows businesses to determine the best possible solution to a problem. When combined with predictive analytics, it adds the benefit of manipulating a future occurrence like mitigate future risk.

Use Case: Prescriptive analytics can be used to maximize an airline's profit. This type of analytics is used to build an algorithm that will automatically adjust the flight fares based on numerous factors, including customer demand, weather, destination, holiday seasons, and oil prices.

2.4 The Lifecycle of Big Data Analytics

Stage 1: Business Case Evaluation - The Big Data Analytics Lifecycle begins with a business case, which defines the reason and goal behind the analysis.

↓

Stage 2: Identification of data - Here, a broad variety of data sources are identified.

↓

Stage 3: Data Filtering - All the identified data from the previous stage is filtered here to remove corrupt data.

↓

Stage 4: Data Extraction - Data that is not compatible with the tool is extracted and then transformed into a compatible form.

↓

Stage 5: Data Aggregation - In this stage, data with the same fields across different datasets are integrated.

↓

Stage 6: Data Analysis - Data is evaluated using analytical and statistical tools to discover useful information.

↓

Stage 7: Visualization of Data - With tools like Tableau, Power BI, and QlikView, Big Data analysts can produce graphic visualizations of the analysis.

2.5 Tools and Techniques Used in Big Data Analytics

- **Apache Hadoop** is a software framework employed for clustered file system and handling of big data. It processes datasets of big data by means of the Map-Reduce programming model.Hadoop is an open-source framework that is written in Java, and it provides cross-platform support. It allows distributed processing of large data sets across clusters of computers. It is one of the best big data tools designed to scale up from single servers to thousands of machines.

- **Map Reduce** is an essential component to the Hadoop framework, serving two functions. The first is mapping, which filters data to various nodes within the cluster. The second is reducing, which organizes and reduces the results from each node to answer a query.

- **Apache Hive** is a data warehouse framework for querying and analysis of data stored in HDFS. It is developed on top of Hadoop. Hive is an open-source software to analyze large data sets on Hadoop. It provides an SQL-like declarative language, called HiveQL, to express queries. Using HiveQL, users associated with SQL can perform data analysis very easily.

- **NoSQL Databases** like Apache Cassandra or MongoDB are non-relational data management systems that do not require a fixed scheme, making them a great option for big, raw, unstructured data. NoSQL stands for "not only SQL," and these databases can handle a variety of data models.

- **YARN** stands for "Yet Another Resource Negotiator." It is another component of second-generation Hadoop. The cluster-management technology helps with job scheduling and resource management in the cluster.

- **Apache HBase** is a distributed column-oriented database that is run at the top of the HDFS file system. It is nothing but a NoSQL Data Storage System, and it is similar to a database management system, but it provides quick random access to a huge amount of structured data.

- **Apache Storm** Apache Storm is an open-source distributed real-time computation system. It is used wherever to generate a lot of data streaming. Twitter uses it for real-time data analysis.

- **Apache Spark** is a unified engine for large-scale data analytics.This is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.It was developed by people from the University of California and written in Scala. The performance of Apache Spark is fast because it has in-memory processing. It does real-time data processing as well as batch processing with a huge amount of data and requires a lot of memory, but it can deal with standard speed and amount of disk.

- **Data Bricks** is a web-based platform for working with Spark, that provides automated cluster management and IPython-style notebooks.This provides a unified, open platform for all need for data. It empowers data scientists, data engineers and data analysts with a simple collaborative environment to run interactive and scheduled data analysis workloads. One can easily collaborate with one of the most widely used cloud platforms like Azure or GCP or AWS.

- **Tableau** is an excellent data visualization and business intelligence tool used for reporting and analyzing vast volumes of data. Some use cases of Tableau are Business Intelligence, Data Visualization, Data Blending, Data Collaboration, query translation into visualization, To create no-code data queries, Real-time data analysis, To manage large size metadata, To import large size of data. Tableau is compatible to almost all kind of data-storage system to extract data from that.

- **Power BI** is a collection of software services, apps, and connectors that work together to turn your unrelated sources of data into coherent, visually immersive, and interactive insights.It is developed by Microsoft with primary focus on business intelligence. It is part of the Microsoft Power Platform.

2.6 Big challenges in Big Data Analytics

Big data brings big benefits, but it also brings big challenges[1] such new privacy and security concerns, accessibility for business users, and choosing the right solutions for your business needs. To capitalize on incoming data, organizations will have to address the following:

- **Making Big Data Accessible:** Collecting and processing data becomes more difficult as the amount of data grows. Organizations must make data easy and convenient for data owners of all skill levels to use.

- **Maintaining Quality Data:** With so much data to maintain, organizations are spending more time than ever before scrubbing for duplicates, errors, absences, conflicts, and inconsistencies.

- **Keeping Data Secure:** As the amount of data grows, so do privacy and security concerns. Organizations will need to strive for compliance and put tight data processes in place before they take advantage of big data.

- **Finding The Right Tools & Platforms:** New technologies for processing and analyzing big data are developed all the time. Organizations must find the right technology to work within their established ecosystems and address their particular needs. Often, the right solution is also a flexible solution that can accommodate future infrastructure changes.

- **Making The Ecosystem Scalable:** In data analytics, the management, and analysis of massive volumes of data is a challenge. Storage systems are not adequately capable of storing rapidly increasing data sets. Though, by improving processor speed, such problems can be reduced. Therefore, needed to develop a processing system that will also maintain the necessity of the future.

Chapter 3

Apache Spark

3.1 History

Researchers at UC Berkeley who had previously worked on Hadoop MapReduce took on this challenge with a project they called Spark. They acknowledged that MR was inefficient (or intractable) for interactive or iterative computing jobs and a complex framework to learn, so from the onset they embraced the idea of making Spark simpler, faster, and easier. This endeavor started in 2009 at the RAD Lab, which later became the AMP Lab (and now is known as the RISE Lab).

Early papers published on Spark demonstrated that it was 10 to 20 times faster than Hadoop MapReduce for certain jobs. Today, it's many orders of magnitude faster. The central thrust of the Spark project was to bring in ideas borrowed from Hadoop MapReduce, but to enhance the system: make it highly fault-tolerant and embarrassingly parallel, support in-memory storage for intermediate results between iterative and interactive map and reduce computations, offer easy and composable APIs in multiple languages as a programming model, and support other workloads in a unified manner. We'll come back to this idea of unification shortly, as it's an important theme in Spark.

By 2013 Spark had gained widespread use, and some of its original creators and researchers—Matei Zaharia, Ali Ghodsi, Reynold Xin, Patrick Wendell, Ion Stoica, and Andy Konwinski—donated the Spark project to the ASF and formed a company called Databricks.

Databricks and the community of open source developers worked to release Apache Spark 1.0 in May 2014, under the governance of the ASF. This first major release established the momentum for frequent future releases and contributions of notable features to Apache Spark from Databricks and over 100 commercial vendors.

As a top-level Apache Software Foundation project, Spark has more than 400 individual contributors and committers from companies such as Facebook, Yahoo!, Intel, Netflix, Databricks, and others.

3.2 Defining Spark

If someone googles that, *"What is Spark[12]?"* then he will get the most common definition of spark as *"Spark is a unified general-purpose distributed data processing engine"*. Those who don't have any prior knowledge of spark will get the definition quite hard to digest. So, if we break it down, then it will look like the following

- **Distributed Data/Distributed Computing:** Apache Spark operates in a world that is slightly different from then usual computer science. When datasets get too big, or when new data comes in too fast, it can become too much for a single computer to handle. This is where distributed computing comes in. Instead of trying to process a huge dataset or run super computationally-expensive programs on one computer, these tasks can be divided between multiple computers that communicate with each other to produce an output. This technology has some serious benefits, but allocating processing tasks across multiple computers has its own set of challenges and can't be structured the same way as normal processing. When Spark says it has to do with distributed data, this means that it is designed to deal with very large datasets and to process them on a distributed computing system with the help of master and worker node.

  On a side note, in a distributed computing system, each individual computer is called a **node** and the collection of all of them is called a **cluster**.

- **Processing Engine/Processing Framework:** A processing engine, sometimes called a processing framework, is responsible for performing data processing tasks. A comparison is probably the best way to understand this. Apache Hadoop is an open source software platform that also deals with "Big Data" and Distributed Computing. Hadoop has a processing engine, distinct from Spark, called MapReduce. MapReduce has its own particular way of optimizing tasks to be processed on multiple nodes, and Spark has a different way. Apache Spark is a next generation batch processing framework with stream processing capabilities. Built using many of the same principles of Hadoop's MapReduce engine, Spark focuses primarily on speeding up batch processing workloads by offering full in-memory computation and processing optimization.

  Spark can be deployed as a standalone cluster (if paired with a capable storage layer) or can hook into Hadoop as an alternative to the MapReduce engine. Following are the two processing models spark uses.

  Batch Processing Model - Unlike MapReduce, Spark processes all data in-memory, only interacting with the storage layer to initially load the data into memory and at the end to persist the final results. All intermediate results are managed in memory.

  While in-memory processing contributes substantially to speed, Spark is also faster on disk-related tasks because of holistic optimization that can be achieved by analyzing the complete set of tasks ahead of time. It achieves this by creating Directed Acyclic Graphs, or DAGs which represent all the operations that must be performed, the data to be operated on, as well as the relationships between them, giving the processor a greater ability to intelligently coordinate work.

  To implement in-memory batch computation, Spark uses a model called Resilient Distributed Datasets, or RDDs, to work with data. These are immutable structures that exist within memory that represent collections of data. Operations on RDDs produce new RDDs. Each RDD can trace its lineage back through its parent RDDs and ultimately to the data on disk. Essentially, RDDs are a way for Spark to maintain fault tolerance without needing to write back to disk after each operation.

Stream Processing Model - Stream processing capabilities are supplied by Spark Streaming. Spark itself is designed with batch-oriented workloads in mind. To deal with the disparity between the engine design and the characteristics of streaming workloads, Spark implements a concept called micro-batches*. This strategy is designed to treat streams of data as a series of very small batches that can be handled using the native semantics of the batch engine.

Spark Streaming works by buffering the stream in sub-second increments. These are sent as small fixed datasets for batch processing. In practice, this works fairly well, but it does lead to a different performance profile than true stream processing frameworks.

- **General Purpose:** One of the main advantages of Spark is how flexible it is, and how many application domains it has. It supports Scala, Python, Java, R, and SQL. It has a dedicated SQL module, it is able to process streamed data in real-time, and it has both a machine learning library and graph computation engine built on top of it. All these reasons contribute to why Spark has become one of the most popular processing engines in the realm of Big Data.
- **Unified:** The concept of unification is not unique to Spark, but this is the core philosophy and design principle of evolution of Spark. In November 2016, the Association for Computing Machinery (ACM) recognized Apache Spark and conferred upon its original creators the prestigious ACM Award for their paper describing Apache Spark as a "Unified Engine for Big Data Processing." The award-winning paper notes that Spark replaces all the separate batch processing, graph, stream, and query engines like Storm, Impala, Dremel, Pregel, etc. with a unified stack of components that addresses diverse workloads under a single distributed fast engine.

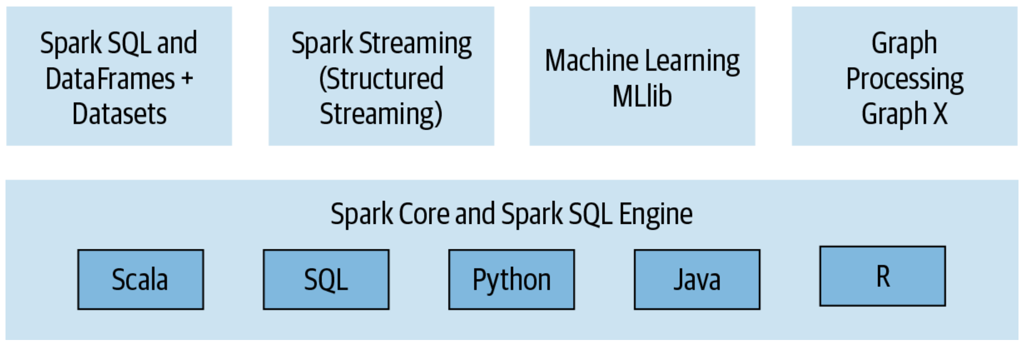

Spark SQL and DataFrames + Datasets
Spark Streaming (Structured Streaming)
Machine Learning MLlib
Graph Processing Graph X
Spark Core and Spark SQL Engine
Scala
SQL
Python
Java
R

Figure 3.1: Apache Spark components and API stack

As shown in Figure 3.1, Spark offers four distinct components as libraries for diverse workloads: Spark SQL, Spark MLlib, Spark Structured Streaming, and GraphX. Each of these components is separate from Spark's core fault-tolerant engine, in that APIs are used to write various Spark application and Spark converts this into a DAG that is executed by the core engine. So whether one write the Spark code using the provided Structured APIs in whatever languages may be Java, R, Scala, SQL, or Python, the underlying code is decomposed into highly compact byte code that is executed in the workers' JVMs across the cluster.

3.3 Internal Design Philosophy of Spark

Spark internals[10] are mainly built on top of four key characteristics, as follows.

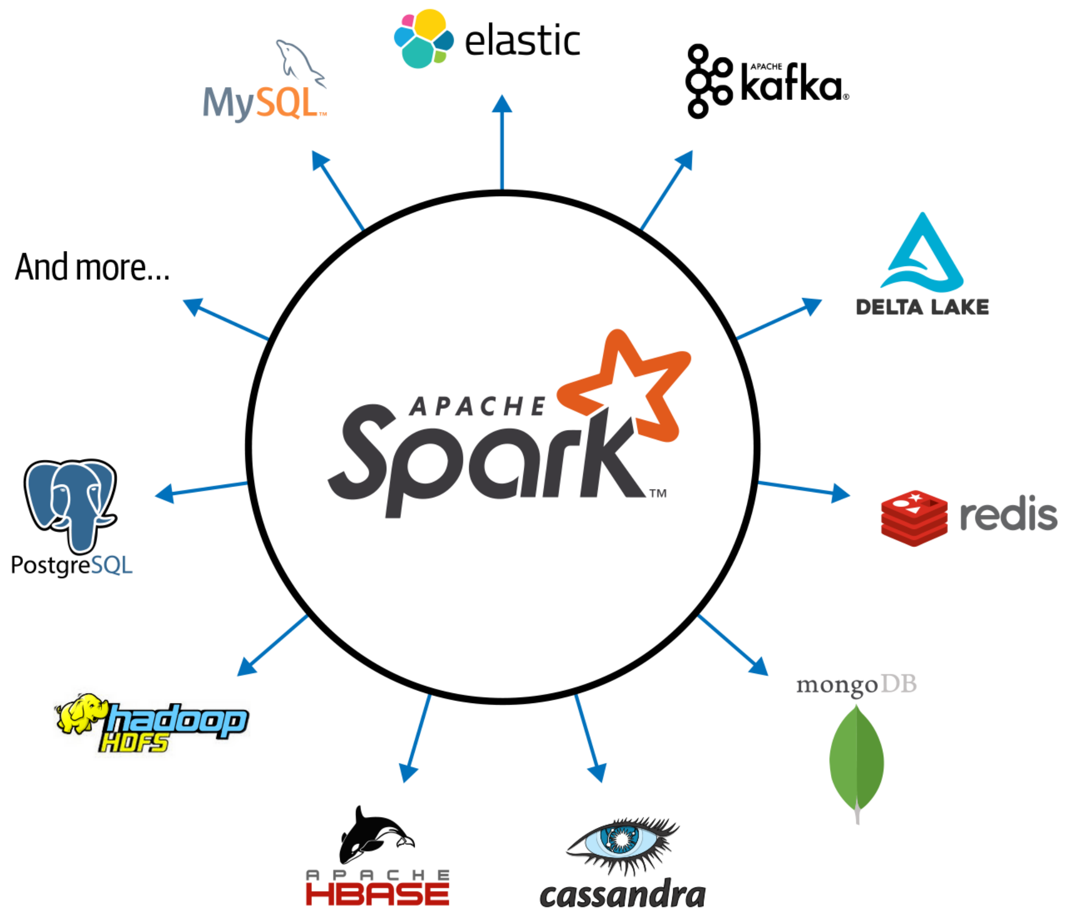


Figure 3.2: Apache Spark's ecosystem of connectors

1. **Speed of Execution:** Spark has pursued the goal of high speed execution mainly in three different ways.

   Firstly, Spark's internal implementation benefits immensely from the hardware industry's recent huge strides in improving the price and performance of CPUs and memory. Today's commodity servers come cheap, with hundreds of gigabytes of memory, multiple cores, and the underlying Unix-based operating system taking advantage of efficient multithreading and parallel processing. The framework is optimized to take advantage of all of these factors.

   Secondly, Spark builds its query computations as a *Directed Acyclic Graph (DAG)*; its DAG scheduler and query optimizer construct an efficient computational graph that can usually be decomposed into tasks that are executed in parallel across workers on the cluster.

Thirdly, Spark's physical execution engine, Tungsten, uses whole-stage code generation to generate compact code for execution

As Spark retains all the intermediate results in memory and its limited disk I/O, this gives it a huge performance boost.

2. **Ease of Use:** All the high level data abstraction such as *Data Frame*(s) or *Data Set*(s) are built on top a fundamental abstraction of a simple logical data structure called a *Resilient Distributed Dataset (RDD).* By providing a set of *transformations* and *actions* as operations, Spark offers a simple programming model that can be used to build big data applications in familiar languages.

3. **Modularity:** Spark operations can be applied across many types of workloads and expressed in any of the supported programming languages: Scala, Java, Python, SQL, and R. Spark offers unified libraries with well-documented APIs that include the following modules as core components: Spark SQL, Spark Structured Streaming, Spark MLlib, and GraphX, combining all the workloads running under one engine.

   One can write a single Spark application that can do it all, no need for distinct engines for diverse workloads, no need to learn separate APIs. With Spark, we get a unified processing engine for every workload.

4. **Scalability:** Unlike Hadoop, Spark focuses only on the lightning fast computation rather than the storage. Spark supports a large variety of data sources, i.e., Spark can be used to read data from Apache Hadoop, Apache Cassandra, Apache HBase, MongoDB, Apache Hive, RDBMS(s) like Ms SQL Server, My SQL, Oracle, PostgreSQL etc. and a lot more and can process the data in memory. Spark's *DataFramekReader*(s) and *DataFrameWriter*(s) can also be extended to read data from other sources, such as Apache Kafka, Kinesis, Azure Storage, and Amazon S3, into its logical data abstraction, on which it can operate.

   The community of Spark developers maintains a list of third-party Spark packages as part of the growing ecosystem (see Figure 3.2). This rich ecosystem of packages includes Spark connectors for a variety of external data sources, performance monitors, and more.

3.4 Various Components of Spark

As depicted in the figure 3.1 Spark Core is the main building block of Spark and other components[9] like Spark SQL, MLlib, Spark Streaming, and GraphX.

- **Spark Core:** Spark Core is the heart of the Apache Spark framework. Spark Core provides the execution engine for the Spark platform which is required and used by other components which are built on top of Spark Core as per the requirement. Spark Core provides the in-built memory computing and referencing datasets stored in external storage systems. It is Spark's core responsibility to perform all the basic I/O functions, scheduling, monitoring, etc. Also, fault recovery and effective memory management are Spark Core's other important functions.

Spark Core uses a very special data structure called the RDD. Data sharing in distributed processing systems like MapReduce need the data in intermediate steps to be stored and then retrieved from permanent storage like HDFS or S3 which makes it very slow due to the serialization and de-serialization of I/O steps. RDDs overcome this as these data structures are in-memory and fault-tolerant and can be shared across different tasks within the same Spark process. The RDDs can be any immutable and partitioned collections and can contain any type of objects; Python, Scala, Java or some user-defined class objects. RDDs can be created either by Transformations of an existing RDD or loading from external sources like HDFS or HBase etc. We will look into RDD and its transformations in-depth in later sections in the tutorial.

- **Spark SQL:** Spark SQL is built on top of Shark, which was the first interactive SQL on the Hadoop system. Shark was built on top of Hive codebase and achieved performance improvement by swapping out the physical execution-engine part of the Hive. But due to the limitations of Hive, Shark was not able to achieve the performance it was supposed to. So the Shark project was stopped, and Spark SQL was built with the knowledge of Shark on top of Spark Core Engine to leverage the power of Spark.

  Spark SQL is named like this because it works with the data similarly to SQL. In fact, it there is a mention that Spark SQL's aim is to meet SQL 92 standards. But the gist is that it allows developers to write declarative code, letting the engine use as much of the data and stored structure (RDDs) as it can to optimize the resultant distributed query behind the scenes. The goal is to allow the user to not have to worry about the distributed nature as much and focus on the business use case. Users can perform extract, transform and load functions on data from a variety of sources in different formats like JSON, Parquet or Hive and then execute ad-hoc queries using Spark SQL.

  DataFrame constitutes the main abstraction for Spark SQL. A distributed collection of data ordered into named columns is known as a DataFrame in Spark. In the earlier versions of Spark SQL, DataFrames were referred to as SchemaRDDs. DataFrame API in Spark integrates with the Spark procedural code to render tight integration between procedural and relational processing. DataFrame API evaluates operations in a lazy manner to provide support for relational optimizations and optimize the overall data processing workflow. All relational functionalities in Spark can be encapsulated using the SparkSQL context or HiveContext.

  Catalyst, an extensible optimizer, is at the core functioning of Spark SQL, which is an optimization framework embedded in Scala to help developers improve their productivity and performance of the queries that they write. Using Catalyst, Spark developers can briefly specify complex relational optimizations and query transformations in a few lines of code by making the best use of Scala's powerful programming constructs like pattern matching and runtime meta-programming. Catalyst eases the process of adding optimization rules, data sources and data types for machine learning domains.

- **Spark Streaming:** This is a very popular Spark library, as it takes Spark's big data processing power and cranks up the speed. Spark Streaming has the ability to Stream gigabytes per second. This capability of big and fast data has a lot of potentials. Spark Streaming is used for analyzing a continuous stream of data in real time. A common example is processing log data from a website or server.

Spark streaming is not really streaming technically. What it really does is it breaks down the data into individual chunks that it processes together as small RDDs. So it actually does not process data as bytes at a time as it comes in, but it processes data every second or two seconds or some fixed interval of time. So strictly speaking, Spark streaming is not real-time but near real-time or micro batching, but it suffices for a vast majority of applications.

Spark streaming can be configured to talk to a variety of data sources. So we can just listen to a port that has a bunch of data being thrown at it, or we can connect to data sources like Amazon Kinesis, Kafka, Flume, etc. There are connectors available to connect Spark to these sources. The good thing about Spark streaming is it is reliable. It has a concept called "check pointing" to store state to the disk periodically and depending on what kind of data sources or receiver we are using, it can pick up data from the point of failure. It is a very robust mechanism to handle all kinds of failures like disk failure or node failure etc.

Just like how Spark SQL has the concept of Dataframe/Dataset built on top of RDD, Spark streaming has something called Dstream. This is a collection of RDDs that consists the entire stream data. The good thing about Dstream is that we can apply most of the built-in functions on RDDs also on the DStream like flatMap, map, etc. Also, the Dstream can be broken into individual RDDs and can be processed one chunk at a time. Spark developers can reuse the same code for stream and batch processing, and can also integrate the streaming data with historical data.

- **MLib:** Today many companies focus on building customer-centric data products and services which need machine learning to build predictive insights, recommendations, and personalized results. Data scientists can solve these problems using popular languages like Python and R, but they spend a lot of time in building and supporting infrastructure for these languages. Spark has built-in support for doing machine learning and data science at a massive scale using the clusters. It's called MLLib which stands for Machine Learning Library.

  MLlib is a low-level machine learning library. It can be called from Java, Scala, and Python programming languages. It is simple to use, scalable and can be easily integrated with other tools and frameworks. MLlib eases the deployment and development of scalable machine learning pipelines. Machine learning in itself is a subject, and it may not be possible to get into details here. But these are some important features and capabilities Spark MLLib offers are

  * Linear regression, logistic regression
  * Support Vector Machines
  * Naive Bayes classifier
  * K-Means clustering
  * Decision trees
  * Recommendations using Alternating Least Squares
  * Basic statistics
  * Chi-squared test, Pearson's or Spearman correlation, min, max, mean, variance
  * Feature Extraction
  * Term Frequency/ Inverse Document Frequency useful for search

- **GraphX:** For graphs and graph-parallel processing Apache Spark provides another API called GraphX. The graph here does not mean charts, lines or bar graphs, but these are graphs in computer sciences like social networks which consist of vertices where each vertex consists of an individual user in the social network and there are many users connected to each other by edges. These edges represent the relationship between the users in the network.

  GraphX is useful in giving overall information about the graph network, like it can tell how many triangles appear in the graph and apply the PageRank algorithm to it. It can measure things like "connectedness", degree distribution, average path length and other high-level measures of a graph. It can also join graphs together and transform graphs quickly. It also supports the Pregel API for traversing a graph. Spark GraphX provides Resilient Distributed Graph (RDG- an abstraction of Spark RDD's). RDG's API is used by data scientists to perform several graph operations through various computational primitives. Similar to RDDs basic operations like map, filter, property graphs also consist of basic operators. Those operators take UDFs (user-defined functions) and produce new graphs. Moreover, these are produced with transformed properties and structure.

3.5 Internal Working[3] of Spark:

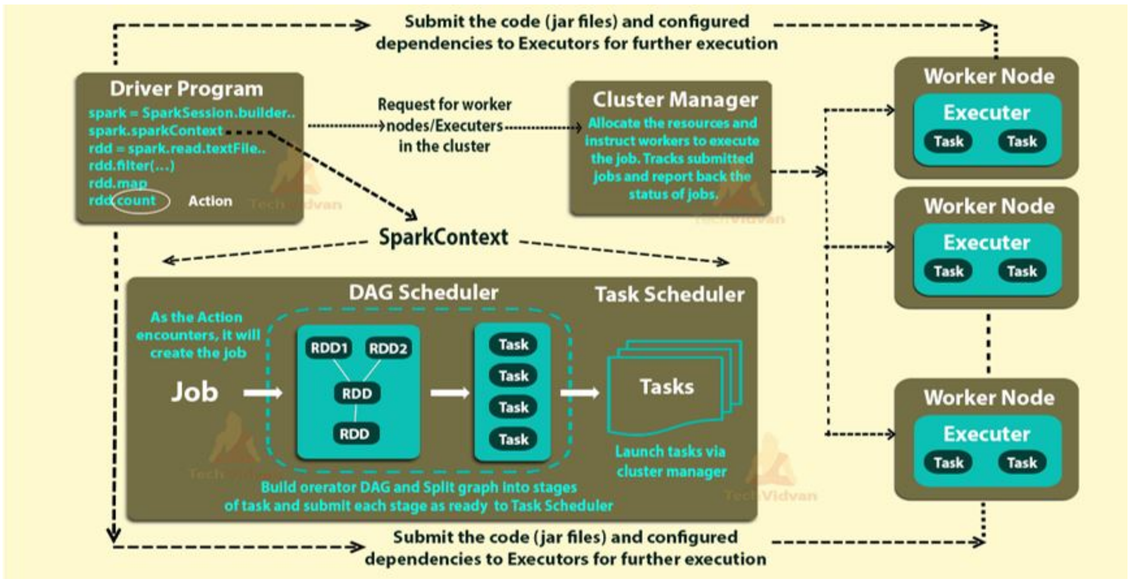

Submit the code (jar files) and configured dependencies to Executors for further execution
Driver Program
spark = SparkSession.builder..
spark.sparkContext
rdd = spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
Action
Request for worker nodes/Executers in the cluster
Cluster Manager
Allocate the resources and instruct workers to execute the job. Tracks submitted jobs and report back the status of jobs.
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
SparkContext
DAG Scheduler
Task Scheduler
As the Action encounters, it will create the job
Job
RDD1
RDD2
RDD
RDD
Task
Task
Task
Task
Tasks
Launch tasks via cluster manager
Build orerator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler
Submit the code (jar files) and configured dependencies to Executors for further execution

Figure 3.3: Internal Working of Spark

The first block seen in the figure 3.3 is the driver program. Once the *Spark Submit* is done, a driver program is launched and this requests for resources to the cluster manager and at the same time the main program of the user function of the user processing program is initiated by the driver program.

Based on that, the execution logic is processed and parallelly, Spark context is also created. Using the Spark context, the different transformations and actions are processed. So, till the time the action is not encountered, all the transformations will go into the Spark context in the form of DAG that will create RDD lineage.

Once the action is called, the job is created. Job is the collection of different task stages. Once these tasks are created, they are launched by the cluster manager on the worker nodes, and this is done with the help of a class called task scheduler.

The conversion of RDD lineage into tasks is done by the DAG scheduler. Here DAG is created based on the different transformations in the program and once the action is called these are split into different stages of tasks and submitted to the task scheduler as tasks become ready.

Then these are launched to the different executors in the worker node through the cluster manager. The entire resource allocation and the tracking of the jobs and tasks are performed by the cluster manager.

As soon as a *Spark Submit* is done, the user program and other configuration mentioned are copied onto all the available nodes in the cluster. So that, the program becomes the local read on all the worker nodes. Hence, the parallel executors running on the different worker nodes do not have to do any kind of network routing.

3.6 Some Spark Terminologies[12]

3.6.1 Distributed Computing:

There are a few terms related to the distributed computing, I will frequently refer while discussing various spark operations.

- **Partitioned Data:** When working with large amount of Data and a cluster of computers, we can't just throw a vanilla DataFrame to the spark engine and expect it to know what to do. Because, the processing tasks will be divided across multiple nodes in a distributed computing environment, the data also has to be able to be divided across multiple nodes. Here the concept of *Partitioned Data* has come into the picture. *Partitioned Data* refers to data that has been optimized to be able to be processed on multiple nodes.

- **Fault Tolerance:** In short, fault tolerance refers to a distributed system's ability to continue working properly even when a failure occurs. A failure could be a node bursting into flames, for example, or just a communication breakdown between nodes. Fault tolerance in Spark revolves around Spark's RDDs. Basically, the way data storage is handled in Spark allows Spark programs to function properly despite occurrences of failure.

- **Lazy Evaluation:** Lazy Evaluation, or Lazy Computing, is all about how code should be compiled. When a compiler that is not lazy (which is called strict evaluation) compiles code, it sequentially evaluates each expression it comes across. A lazy compiler, on the other hand, doesn't continually evaluate expressions, but rather, waits until it is actually told to generate a result(in Spark, until any *Action* is called on RDD), and then performs all the evaluation all at once. So as it compiles code, it keeps track of everything it will eventually have to evaluate (in Spark this kind of evaluation log is called a *Lineage Graph*), and then whenever it is prompted to return something, it performs evaluations according to what it has in its evaluation log(Lineage Graph). This is useful because it makes programs more efficient and optimized, as the compiler doesn't have to evaluate anything that isn't actually used.

3.6.2 Spark Context:

SparkContext has been available since Spark 1.x versions, and it's an entry point to Spark when we wanted to program and use Spark RDD. Most of the operations/methods or functions we use in Spark are comes from SparkContext for example accumulators, broadcast variables, parallelize and more.

3.6.3 Spark Session:

With Spark 2.0 a new class called SparkSession has been introduced to use which is a combined class for all different contexts we used to have prior to 2.0 (SQLContext and HiveContext e.t.c) release hence SparkSession can be used in replace with SQLContext and HiveContext.

SparkSession is an entry point to Spark and creating a SparkSession instance would be the first statement we would write to program with RDD, DataFrame and Dataset and SparkSession will be created using SparkSession.builder() builder patterns.

Spark Session also includes all the APIs available in different contexts –

∗ Spark Context

∗ SQL Context

∗ Streaming Context

∗ Hive Context

3.6.4 RDD(Resilient Distributed Dataset):

RDDs are data structures that are the core building blocks of Spark. A RDD is an immutable, partitioned collection of records, which means that it can hold values, tuples, or other objects, these records are partitioned so as to be processed on a distributed system, and that once an RDD has been made, it is impossible to alter it. That basically sums up its acronym: they are resilient due to their immutability and lineage graphs (which will be discussed shortly), they can be distributed due to their partitions, and they are datasets because, well, they hold data.

A crucial thing to note is that RDDs do not have a schema, which means that they do not have a columnar structure. Records are just recorded row-by-row, and are displayed similar to a list.

3.6.5 Data Frame:

A DataFrame is a distributed collection of data organized into named columns. It is conceptually equal to a table in a relational database. Spark DataFrame have all the features of RDDs but also have a schema. This will make them our data structure of choice for getting started with PySpark.

3.6.6 Data Set:

Dataset is a data structure in SparkSQL which is strongly typed and is a map to a relational schema. It represents structured queries with encoders. It is an extension to DataFrame API. Spark Dataset provides both type safety and object-oriented programming interface. The concept of Dataset has been introduced in Spark 1.6 version.

3.6.7 Type of Spark Operations:

There are mainly two type of spark operations we can do in Spark as follows:

- **Transformation:** Transformations are one of the things that can be done to an RDD in Spark. They are lazy operations that create one or more new RDDs. It's important to note that Transformations create new RDDs because, RDDs are immutable, so they can't be altered in any way once they've been created. So, in essence, Transformations take an RDD as an input and perform some function on them based on what Transformation is being called, and outputs one or more RDDs. Due to the lazy evaluation, as spark engine comes across each *Transformation*, it doesn't actually build any new RDDs, but rather constructs a chain of hypothetical RDDs that would result from those Transformations which will only be evaluated once an *Action* is called. This chain of hypothetical, or *Child*, RDDs, all connected logically back to the original *Parent* RDD, is what a lineage graph is.

- **Action:** Actions are another type of spark operations on RDDs that does not produce a new RDD. Some examples of common *Action*(s) are doing a count of the data, or finding the max or min, or returning the first element of an RDD, etc. So, ultimately, to sum up, the *Action* is the signal to the compiler to evaluate the lineage graph and return the value specified by the Action.

3.6.8 Lineage Graph:

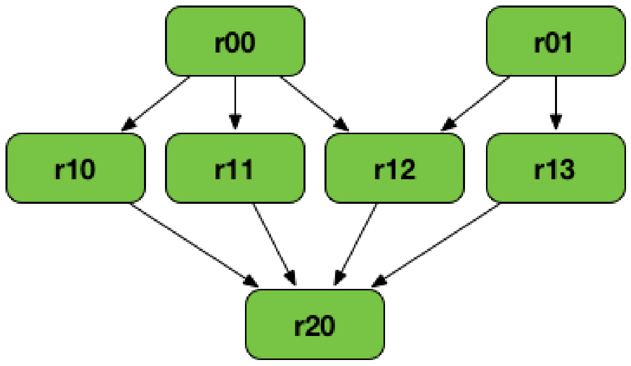


Figure 3.4: Visualization of example lineage graph; r00, r01 are parent RDDs, r20 is final RDD

A *Lineage Graph* outlines what is called a *"Logical Execution Plan"* which means the compiler begins with the earliest RDDs that aren't dependent on any other RDDs, and follows a logical chain of Transformations until it ends with the RDD that an Action is called on. This feature is primarily what drives Spark's fault tolerance. If a node fails for some reason, all the information about what that node was supposed to be doing is stored in the lineage graph, which can be replicated elsewhere.

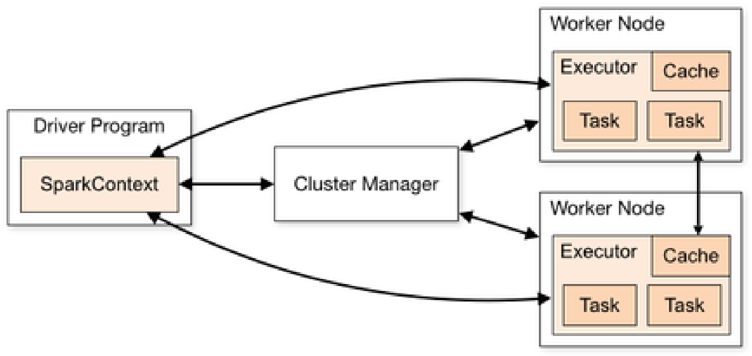


Figure 3.5: Visualizing the Spark Architecture

3.6.9 Spark Application & Spark Job:

In Spark, when an item of processing has to be done, there is a "driver" process that is in charge of taking the user's code and converting it into a set of multiple tasks. There are also "executor" processes, each operating on a separate node in the cluster, that are in charge of running the tasks, as delegated by the driver. Each driver process has a set of executors that it has access to in order to run tasks. A *Spark Application* is a user built program that consists of a driver and that driver's associated executors. A *Spark Job* is a task or set of tasks to be executed with executor processes, as directed by the driver. A *Spark Job* is triggered by the calling of an RDD Action.

Chapter 4

An overview of Databricks Cloud Platform to use Apache Spark functionalities

4.1 Introduction to DataBricks

Databricks[5], developed by the creators of Apache Spark, is a Web-based platform, which is also a one-stop product for all Data requirements, like Storage and Analysis. It can derive insights using SparkSQL, provide active connections to visualization tools such as Power BI, Qlikview, and Tableau, and build Predictive Models using SparkML. Databricks also can create interactive displays, text, and code tangibly. Databricks is an alternative to the MapReduce system.

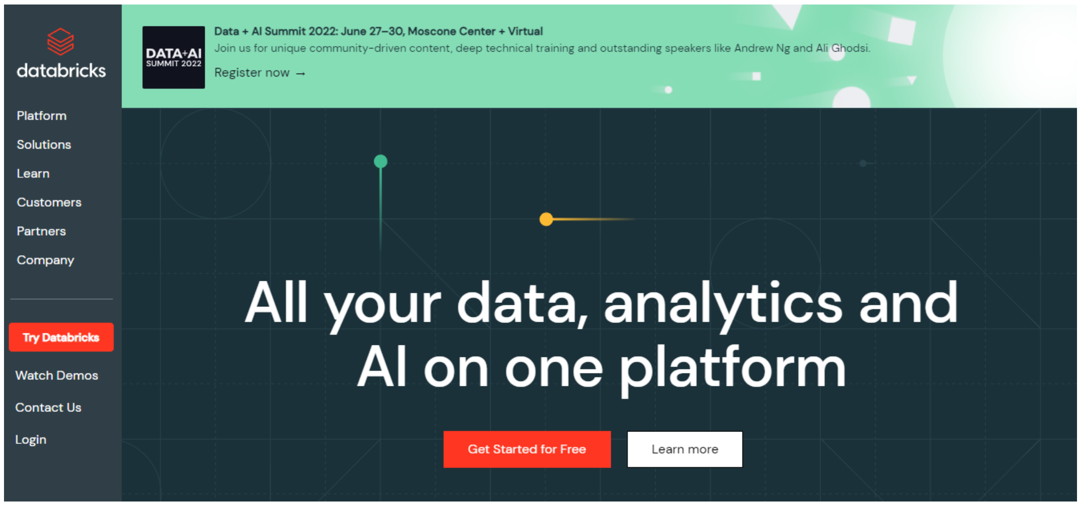
Databricks is integrated with Microsoft Azure, Amazon Web Services, and Google Cloud Platform, making it easy for businesses to manage a colossal amount of data and carry out Machine Learning tasks.

It deciphers the complexities of processing data for data scientists and engineers, which allows them to develop ML applications using R, Scala, Python, or SQL interfaces in Apache Spark[9]. Organizations collect large amounts of data, either in data warehouses or data lakes. According to requirements, data is often moved between them at a high frequency which is complicated, expensive, and non-collaborative.

However, Databricks simplifies Big Data Analytics by incorporating a LakeHouse architecture that provides data warehousing capabilities to a data lake. As a result, it eliminates unwanted data silos created while pushing data into data lakes or multiple data warehouses. It also provides data teams with a single source of the data by leveraging LakeHouse architecture.

4.2 Signing Up to the DataBricks Community Edition

1. Go to the website of DataBricks which is https://databricks.com/. The page will look something like figure 4.1

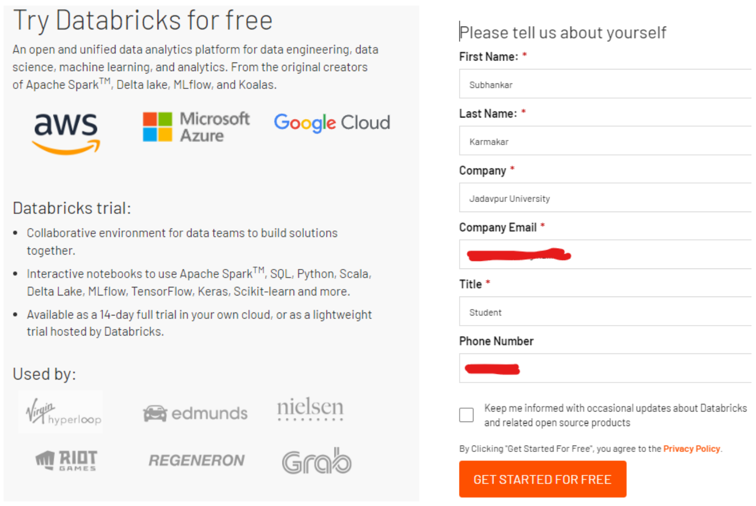

Data + AI Summit 2022: June 27–30, Moscone Center + Virtual

Join us for unique community-driven content, deep technical training and outstanding speakers like Andrew Ng and Ali Ghodsi.

All your data, analytics and AI on one platform

Figure 4.1: Home Page of Databricks Website

2. Then click on **Get Started for Free** . The SignUp page will look like figure 4.1. Then click on he *Get Stated for Free* at the bottom of the SignUp form.


Try Databricks for free

An open and unified data analytics platform for data engineering, data science, machine learning, and analytics. From the original creators of Apache Spark^{TM}, Delta lake, MLflow, and Koalas.

Databricks trial:

- Collaborative environment for data teams to build solutions together.
- Interactive notebooks to use Apache Spark^{TM}, SQL, Python, Scala, Delta Lake, MLflow, TensorFlow, Keras, Scikit-learn and more.
- Available as a 14-day full trial in your own cloud, or as a lightweight trial hosted by Databricks.

Used by:

Please tell us about yourself

First Name: * Subhankar

Last Name: * Karmakar

Company * Jadavpur University

Company Email * [redacted]

Title * Student

Phone Number [redacted]

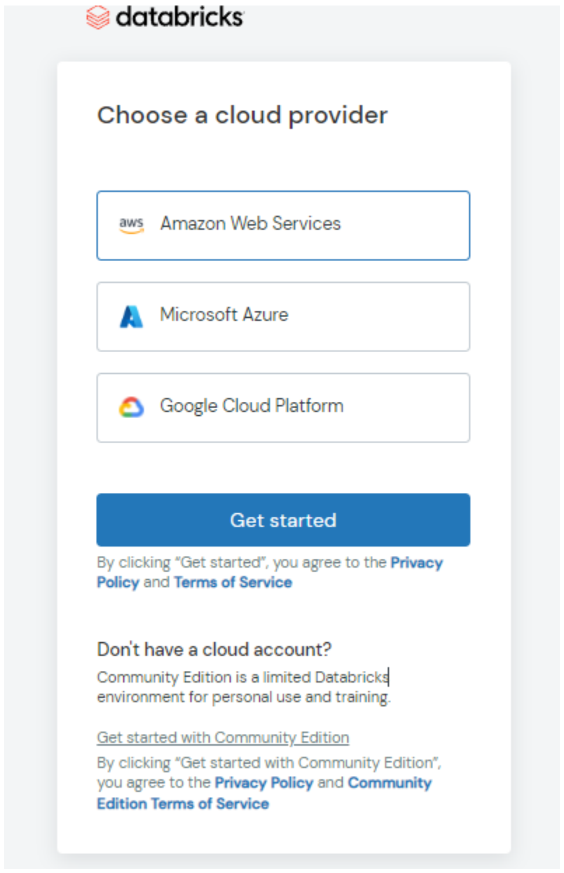
☐ Keep me informed with occasional updates about Databricks and related open source products

By Clicking "Get Started For Free", you agree to the Privacy Policy.

Figure 4.2: Databricks SignUp Page

3. Now if any one has access to any of the clod service provider like AWS,GCP or Azure then choose for that as shown in the figure 4.3 which will initially give you 14 days of trial period to use Databricks components for free, otherwise to opt for the community edition click on the GET STARTED WITH COMMUNITY EDITION at the bottom.

4. Then verify with captcha and an email would be sent for the password creation. Once you


databricks

Choose a cloud provider

Amazon Web Services

Microsoft Azure

Google Cloud Platform

Get started

By clicking "Get started", you agree to the Privacy Policy and Terms of Service

Don't have a cloud account?

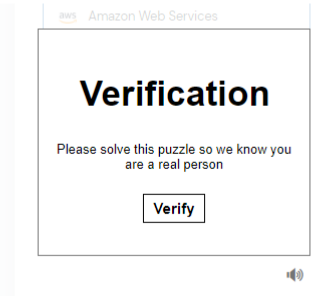
Community Edition is a limited Databricks environment for personal use and training.

Get started with Community Edition

By clicking "Get started with Community Edition", you agree to the Privacy Policy and Community Edition Terms of Service

Figure 4.3: Choosing Cloud Provider or Opt for Community Edition

do that then you are successfully done with Account Creation using Community Edition of DataBricks. The community edition uses the DBFS(Databricks File System) to store the data in the Databricks itself which is not dependent on any of the Cloud Solution Providers.

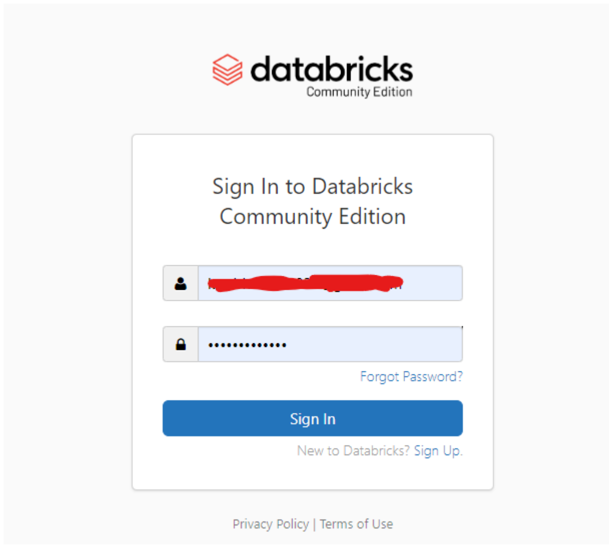

Verification

Please solve this puzzle so we know you are a real person

Verify

Figure 4.4: Verify with Captcha

5. Now to login to the community edition as and when required you need to got the Databricks Community Cloud which is https://community.cloud.databricks.com/. Then by providing username and password as shown in the figure 4.5 one will be able to login to the *Databricks Community Cloud*.


databricks
Community Edition
Sign In to Databricks Community Edition

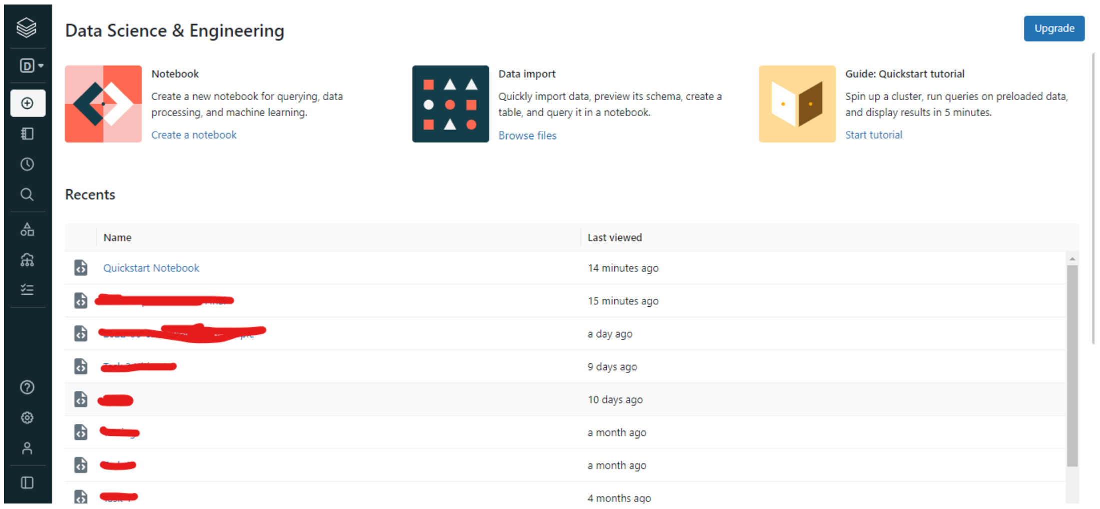
[redacted]

•••••••••••••

Forgot Password?
Sign In
New to Databricks? Sign Up.
Privacy Policy | Terms of Use

Figure 4.5: Login form for Community Edition

6. After logging into the system the home page like the figure 4.6 would be visible.


Data Science & Engineering
Upgrade
Notebook
Create a new notebook for querying, data processing, and machine learning.
Create a notebook
Data import
Quickly import data, preview its schema, create a table, and query it in a notebook.
Browse files
Guide: Quickstart tutorial
Spin up a cluster, run queries on preloaded data, and display results in 5 minutes.
Start tutorial
Recents

Name	Last viewed
Quickstart Notebook	14 minutes ago
[redacted]	15 minutes ago
[redacted]	a day ago
[redacted]	9 days ago
[redacted]	10 days ago
[redacted]	a month ago
[redacted]	a month ago
[redacted]	4 months ago

Figure 4.6: Home Page for Community Edition

4.3 Exploring Databricks Community Edition

4.3.1 Side Navigation Panel

Through this side navigation panel one can directly jump into the desired section like *Workspace* , *Recents*, *Data* , *Compute* , *Worlflows*. Refer to the figure 4.7 .

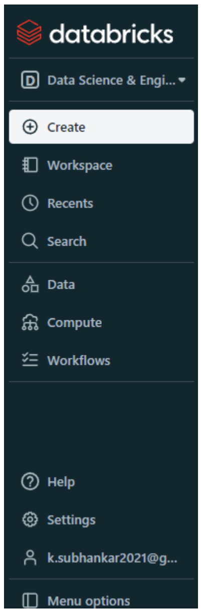


Figure 4.7: Side Navigation Panel

4.3.2 Workspace

This section is responsible for displaying all the notebooks we have created so far. It is like a repository of notebooks. Here one can easily navigate to different notebooks and edit them, rename them, export them, import new notebooks and a lot more.Refer to the figure 4.8.

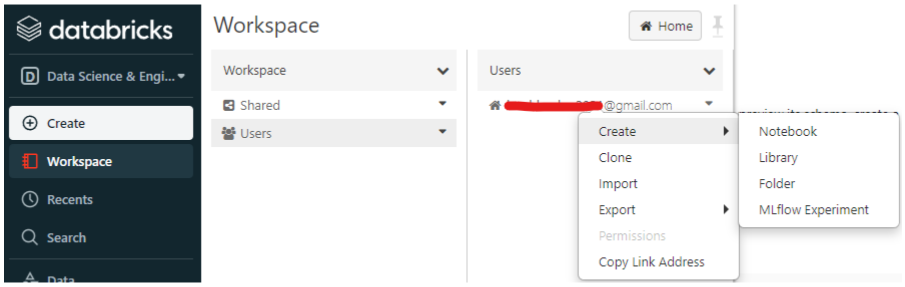
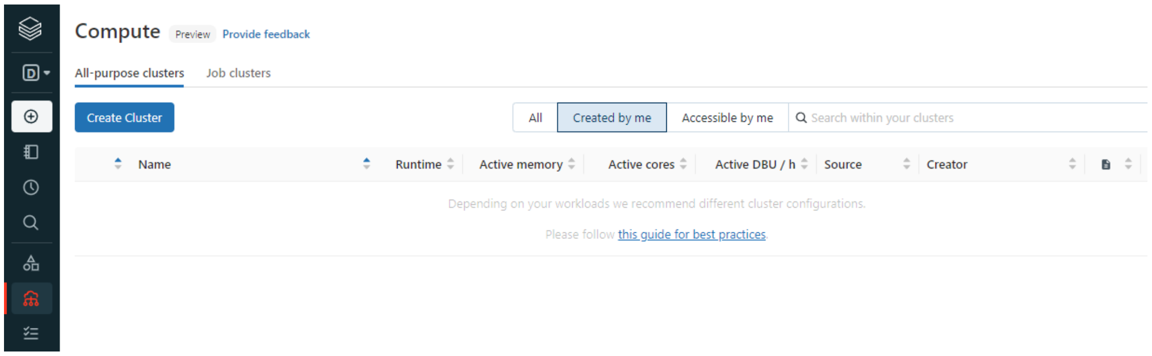


Figure 4.8: Workspace View

4.3.3 Compute

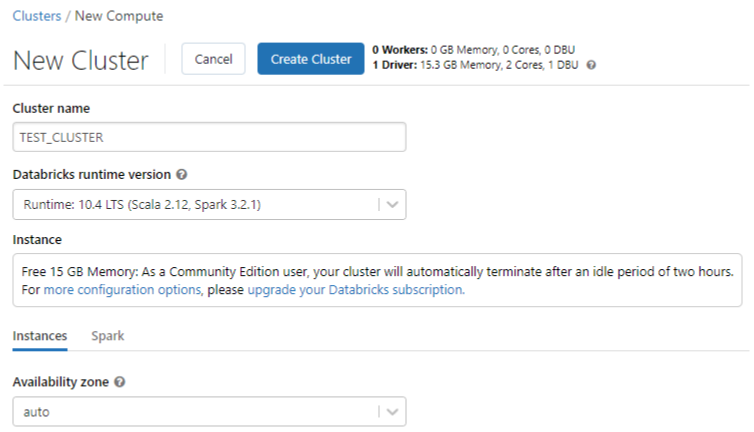
This section is very much important one because here we can create , configure, terminate, restart our clusters. But terminate and restart facility is not available in the community edition. We can only create and delete clusters and all the created cluster would be terminated automatically after 2 hours of inactivity.



Depending on your workloads we recommend different cluster configurations.

Please follow this guide for best practices.

Figure 4.9: To view Created Clusters



0 Workers: 0 GB Memory, 0 Cores, 0 DBU
1 Driver: 15.3 GB Memory, 2 Cores, 1 DBU

Cluster name: TEST_CLUSTER

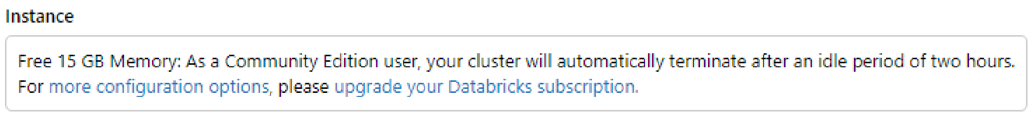
Databricks runtime version: Runtime: 10.4 LTS (Scala 2.12, Spark 3.2.1)

Instance

Free 15 GB Memory: As a Community Edition user, your cluster will automatically terminate after an idle period of two hours. For more configuration options, please upgrade your Databricks subscription.

Availability zone: auto

Figure 4.10: Creating Clusters



Instance

Free 15 GB Memory: As a Community Edition user, your cluster will automatically terminate after an idle period of two hours. For more configuration options, please upgrade your Databricks subscription.

Figure 4.11: Restriction for Community Edition

4.3.4 Data

This section is responsible for showing up the files like csv , json etc. that are uploaded to the DBFS(Databricks File System).

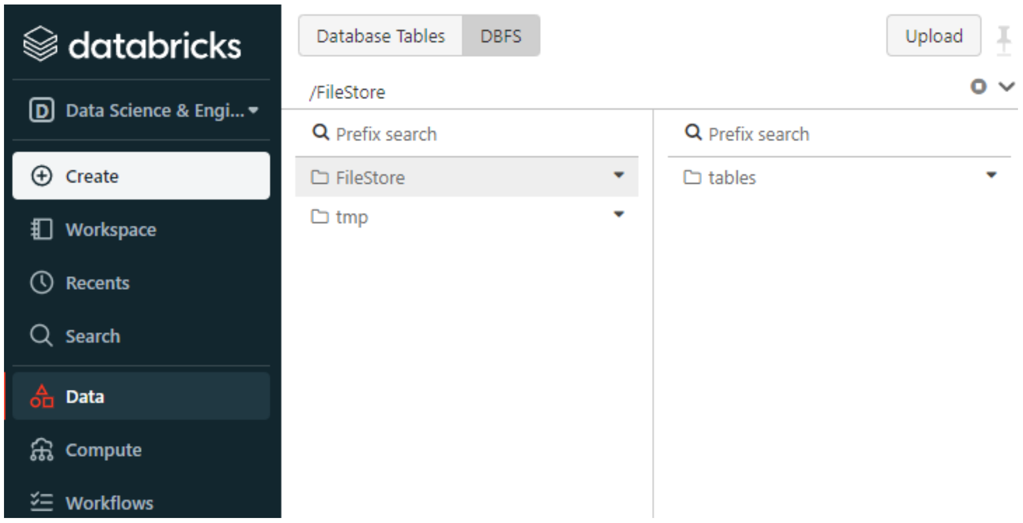


Figure 4.12: DBFS UI

4.4 Create and Manage Interactive Clusters

A Databricks cluster is a set of computation resources and configurations on which you run data engineering, data science, and data analytics workloads, such as production ETL pipelines, streaming analytics, ad-hoc analytics, and machine learning. You run these workloads as a set of commands in a notebook or as an automated job.

Databricks makes a distinction between all-purpose clusters and job clusters.

- **All-Purpose Clusters** to analyze data collaboratively using interactive notebooks.
- **Job Clusters** to run fast and robust automated jobs.

4.4.1 Create Cluster

Depending on the workspace in which you're currently working, you may or may not have cluster creation privileges.

Instructions in this section assume that you **do** have cluster creation privileges, and that you need to deploy a new cluster to execute the lessons in this course.

NOTE: Check with your instructor or a platform admin to confirm whether or not you should create a new cluster or connect to a cluster that has already been deployed. Cluster policies may impact your options for cluster configuration.

Steps:

1. Use the left sidebar to navigate to the **Compute** page
2. Click the blue **Create Cluster** button
3. For the **Cluster name**, use your name so that you can find it easily and the instructor can easily identify it if you have problems
4. Set the **Cluster mode** to **Single Node** (this mode is required to run this course)
5. Use the recommended **Databricks runtime version** for this course
6. Leave boxes checked for the default settings under the **Autopilot Options**
7. Click the blue **Create Cluster** button

NOTE: Clusters can take several minutes to deploy. Once you have finished deploying a cluster, feel free to continue to explore the cluster creation UI.

4.4.2 Restart, Terminate, and Delete

Note that while **Restart**, **Terminate**, and **Delete** have different effects, they all start with a cluster termination event. (Clusters will also terminate automatically due to inactivity assuming this setting is used.)

When a cluster terminates, all cloud resources currently in use are deleted. This means:
- Associated VMs and operational memory will be purged
- Attached volume storage will be deleted
- Network connections between nodes will be removed

In short, all resources previously associated with the compute environment will be completely removed. This means that any results that need to be persisted should be saved to a permanent location. Note that you will not lose your code, nor will you lose data files that you've saved out appropriately.

The **Restart** button will allow us to manually restart our cluster. This can be useful if we need to completely clear out the cache on the cluster or wish to completely reset our compute environment.

The **Terminate** button allows us to stop our cluster. We maintain our cluster configuration setting, and can use the **Restart** button to deploy a new set of cloud resources using the same configuration.

The **Delete** button will stop our cluster and remove the cluster configuration.

4.5 Using Notebook

4.5.1 Running a Cell

Notebooks provide cell-by-cell execution of code. Multiple languages can be mixed in a notebook. Users can add plots, images, and markdown text to enhance their code.

- Run the cell below using one of the following options:
- **CTRL+ENTER** or **CTRL+RETURN**

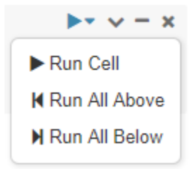
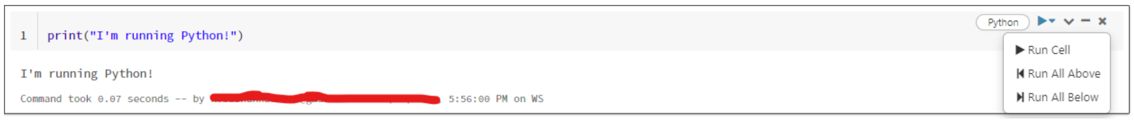


Figure 4.13: Running a Cell



```
print("I'm running Python!")
```

```
I'm running Python!
Command took 0.07 seconds -- by [redacted] 5:56:00 PM on WS
```

Figure 4.14: Running a Cell

- **SHIFT+ENTER** or **SHIFT+RETURN** to run the cell and move to the next one
- Using **Run Cell**, **Run All Above** or **Run All Below**

4.5.2 Setting the Default Notebook Language

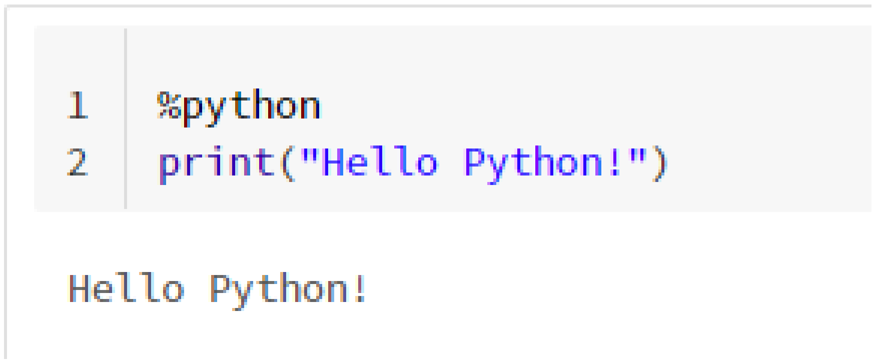
Databricks notebooks support Python, SQL, Scala, and R. A language can be selected when a notebook is created, but this can be changed at any time.

The default language appears directly to the right of the notebook title at the top of the page. We'll change the default language for this notebook to SQL. Steps:

- Click on the **Python** next to the notebook title at the top of the screen
- In the UI that pops up, select **SQL** from the drop down list

4.5.3 Magic Commands

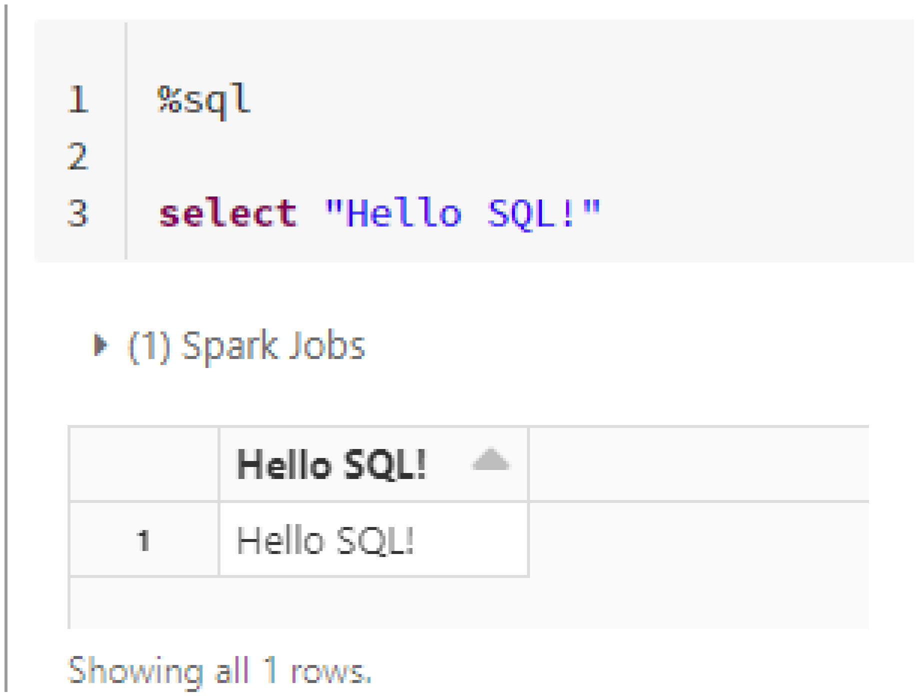
- Magic commands are specific to the Databricks notebooks
- They are very similar to magic commands found in comparable notebook products
- These are built-in commands that provide the same outcome regardless of the notebook's language
- A single percent (%) symbol at the start of a cell identifies a magic command
  - One can only have one magic command per cell
  - A magic command must be the first thing in a cell
- **Language Magics** Language magic commands allow for the execution of code in languages other than the notebook's default. In this course, we'll see the following language magics:
  - **%python**
  - **%sql**



```
%python
print("Hello Python!")
```

```
Hello Python!
```

Figure 4.15: Using Python Magic



```
%sql

select "Hello SQL!"
```

	Hello SQL!
1	Hello SQL!

Showing all 1 rows.

Figure 4.16: Using SQL Magic

Chapter 5

Sample Data Engineering Tasks with the help of Data Bricks and PySpark

5.1 NYC Taxi Dataset Analysis

Load NYC taxi data to DBFS and extract the data through dataframe in the notebook. Perform Following Queries using PySpark.

5.1.1 Task Description

- **Query 1.** Add a column named as ""Revenue"" into dataframe which is the sum of the below columns 'Fare_amount','Extra','MTA_tax','Improvement_surcharge','Tip_amount', 'Tolls_amount','Total_amount'
- **Query 2.** Increasing count of total passengers in New York City by area
- **Query 3.** Realtime Average fare/total earning amount earned by 2 vendors
- **Query 4.** Moving Count of payments made by each payment mode
- **Query 5.** Highest two gaining vendor's on a particular date with no of passenger and total distance by cab
- **Query 5.** Most no of passenger between a route of two location.
- **Query 6.** Get top pickup locations with most passengers in last 5/10 seconds.

The data dictionary of the dataset used for the task is given in the next page where one can get a brief idea about the data set. To know more about the data set refer to the link New York Taxi Data User Guide[7].

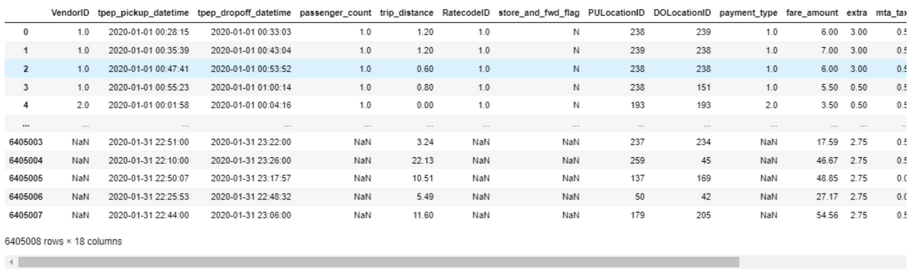
Goal The goal of the task is to get used to with the PySpark i.e. the python APIs to deal with SparkInternals by answering few real life queries that add business values.

Field Name	Description
VendorID	A code indicating the TPEP provider that provided the record. **1= Creative Mobile Technologies, LLC; 2= VeriFone Inc.**
tpep_pickup_datetime	The date and time when the meter was engaged.
tpep_dropoff_datetime	The date and time when the meter was disengaged.
Passenger_count	The number of passengers in the vehicle. This is a driver-entered value.
Trip_distance	The elapsed trip distance in miles reported by the taximeter.
PULocationID	TLC Taxi Zone in which the taximeter was engaged
DOLocationID	TLC Taxi Zone in which the taximeter was disengaged
RateCodeID	The final rate code in effect at the end of the trip. **1= Standard rate** **2=JFK** **3=Newark** **4=Nassau or Westchester** **5=Negotiated fare** **6=Group ride**
Store_and_fwd_flag	This flag indicates whether the trip record was held in vehicle memory before sending to the vendor, aka "store and forward," because the vehicle did not have a connection to the server. **Y= store and forward trip** **N= not a store and forward trip**
Payment_type	A numeric code signifying how the passenger paid for the trip. **1= Credit card** **2= Cash** **3= No charge** **4= Dispute** **5= Unknown** **6= Voided trip**
Fare_amount	The time-and-distance fare calculated by the meter.
Extra	Miscellaneous extras and surcharges. Currently, this only includes the $0.50 and $1 rush hour and overnight charges.
MTA_tax	$0.50 MTA tax that is automatically triggered based on the metered rate in use.
Improvement_surcharge	$0.30 improvement surcharge assessed trips at the flag drop. The improvement surcharge began being levied in 2015.
Tip_amount	Tip amount – This field is automatically populated for credit card tips. Cash tips are not included.
Tolls_amount	Total amount of all tolls paid in trip.
Total_amount	The total amount charged to passengers. Does not include cash tips.
Congestion_Surcharge	Total amount collected in trip for NYS congestion surcharge.
Airport_fee	$1.25 for pick up only at LaGuardia and John F. Kennedy Airports

5.1.2 Implementation

Loading data to Pandas Dataframe directly from AWS S3 Bucket.

```
import pandas as pd
pandas_df = pd.read_csv("https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2020
    -01.csv",low_memory=False)
pandas_df
```



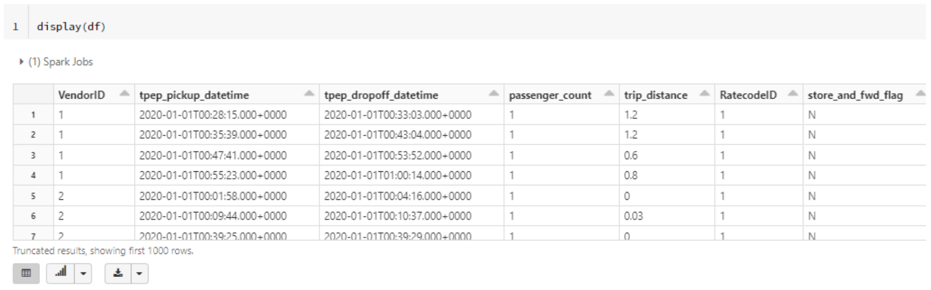
	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	RatecodeID	store_and_fwd_flag	PULocationID	DOLocationID	payment_type	fare_amount	extra	mta_ta
0	1.0	2020-01-01 00:28:15	2020-01-01 00:33:03	1.0	1.20	1.0	N	238	239	1.0	6.00	3.00	0.5
1	1.0	2020-01-01 00:35:39	2020-01-01 00:43:04	1.0	1.20	1.0	N	239	238	1.0	7.00	3.00	0.5
2	1.0	2020-01-01 00:47:41	2020-01-01 00:53:52	1.0	0.60	1.0	N	238	238	1.0	6.00	3.00	0.5
3	1.0	2020-01-01 00:55:23	2020-01-01 01:00:14	1.0	0.80	1.0	N	238	151	1.0	5.50	0.50	0.5
4	2.0	2020-01-01 00:01:58	2020-01-01 00:04:16	1.0	0.00	1.0	N	193	193	2.0	3.50	0.50	0.5
...	...	...	...	...	...	...	...	...	...	...	...	...	..
6405003	NaN	2020-01-31 22:51:00	2020-01-31 23:22:00	NaN	3.24	NaN	NaN	237	234	NaN	17.59	2.75	0.5
6405004	NaN	2020-01-31 22:10:00	2020-01-31 23:26:00	NaN	22.13	NaN	NaN	259	45	NaN	46.67	2.75	0.5
6405005	NaN	2020-01-31 22:50:07	2020-01-31 23:17:57	NaN	10.51	NaN	NaN	137	169	NaN	48.85	2.75	0.0
6405006	NaN	2020-01-31 22:25:53	2020-01-31 22:48:32	NaN	5.49	NaN	NaN	50	42	NaN	27.17	2.75	0.0
6405007	NaN	2020-01-31 22:44:00	2020-01-31 23:06:00	NaN	11.60	NaN	NaN	179	205	NaN	54.56	2.75	0.5

6405008 rows × 18 columns

Figure 5.1: Overview of part of the NYC Taxi Dataset

Creating Spark Dataframe from Pandas Dataframe

```
df= spark.createDataFrame(pandas_df)
display(df)
```



```
display(df)
```

(1) Spark Jobs

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	RatecodeID	store_and_fwd_flag
1	1	2020-01-01T00:28:15.000+0000	2020-01-01T00:33:03.000+0000	1	1.2	1	N
2	1	2020-01-01T00:35:39.000+0000	2020-01-01T00:43:04.000+0000	1	1.2	1	N
3	1	2020-01-01T00:47:41.000+0000	2020-01-01T00:53:52.000+0000	1	0.6	1	N
4	1	2020-01-01T00:55:23.000+0000	2020-01-01T01:00:14.000+0000	1	0.8	1	N
5	2	2020-01-01T00:01:58.000+0000	2020-01-01T00:04:16.000+0000	1	0	1	N
6	2	2020-01-01T00:09:44.000+0000	2020-01-01T00:10:37.000+0000	1	0.03	1	N
7	2	2020-01-01T00:39:25.000+0000	2020-01-01T00:39:29.000+0000	1	0	1	N

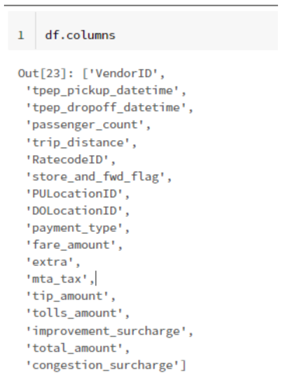
Truncated results, showing first 1000 rows.

Figure 5.2: Displaying Spark Dataframe created from the Pandas Dataframe

Printing the available columns in the Dataframe

Casting the existing data types of the columns to suitable data types that can help us for computation and querying the data

```
from pyspark.sql.functions import *
df = df.withColumn("tpep_pickup_datetime",to_timestamp(df.tpep_pickup_datetime,"yyyy-MM-
    dd HH:mm:ss"))\
```


```
df.columns
```

```
Out[23]: ['VendorID',
 'tpep_pickup_datetime',
 'tpep_dropoff_datetime',
 'passenger_count',
 'trip_distance',
 'RatecodeID',
 'store_and_fwd_flag',
 'PULocationID',
 'DOLocationID',
 'payment_type',
 'fare_amount',
 'extra',
 'mta_tax',
 'tip_amount',
 'tolls_amount',
 'improvement_surcharge',
 'total_amount',
 'congestion_surcharge']
```

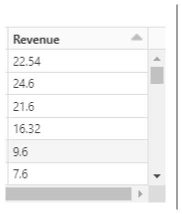
Figure 5.3: List of all Columns present in the NYC Taxi Data Set

```
.withColumn("tpep_dropoff_datetime",to_timestamp(df.tpep_dropoff_datetime,"yyyy-MM-dd HH:
    mm:ss"))\
.withColumn("VendorID",df.VendorID.cast("int"))
df.printSchema()



"""
OUTPUT:

root
 |-- VendorID: integer (nullable = true)
 |-- tpep_pickup_datetime: timestamp (nullable = true)
 |-- tpep_dropoff_datetime: timestamp (nullable = true)
 |-- passenger_count: double (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- RatecodeID: double (nullable = true)
 |-- store_and_fwd_flag: string (nullable = true)
 |-- PULocationID: long (nullable = true)
 |-- DOLocationID: long (nullable = true)
 |-- payment_type: double (nullable = true)
 |-- fare_amount: double (nullable = true)
 |-- extra: double (nullable = true)
 |-- mta_tax: double (nullable = true)
 |-- tip_amount: double (nullable = true)
 |-- tolls_amount: double (nullable = true)
 |-- improvement_surcharge: double (nullable = true)
 |-- total_amount: double (nullable = true)
 |-- congestion_surcharge: double (nullable = true)
"""
```

Querying Data

- Query 1

```
df = df.withColumn("Revenue",df.fare_amount+\
                      df.extra+\
                      df.mta_tax+\
                      df.tip_amount+\
                      df.tolls_amount+\
                      df.improvement_surcharge+\
                      df.total_amount)
```

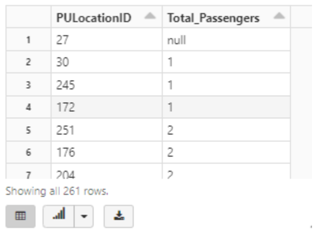


Revenue
22.54
24.6
21.6
16.32
9.6
7.6

Figure 5.4: Displaying the Calculated Revenue Column

- Query 2

```
location_wise_total_passengers = df.groupBy("PULocationID").sum("passenger_count
    ").withColumnRenamed("sum(passenger_count)","Total_Passengers")

display(location_wise_total_passengers.orderBy(location_wise_total_passengers["
    Total_Passengers"].asc()))
```



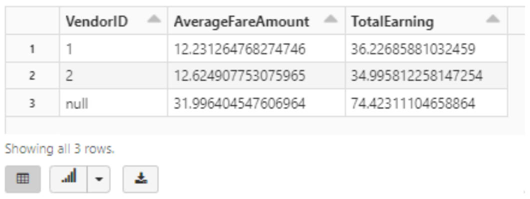
	PULocationID	Total_Passengers
1	27	null
2	30	1
3	245	1
4	172	1
5	251	2
6	176	2
7	204	2

Showing all 261 rows.

Figure 5.5: Pickup Location wise total number of passengers

- Query 3

```
display(df.groupBy("VendorID").agg(avg("fare_amount").alias("AverageFareAmount"),
    avg("Revenue").alias("TotalEarning")))
```



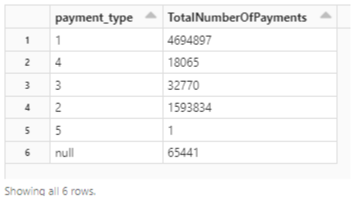
	VendorID	AverageFareAmount	TotalEarning
1	1	12.231264768274746	36.22685881032459
2	2	12.624907753075965	34.995812258147254
3	null	31.996404547606964	74.42311104658864

Showing all 3 rows.

Figure 5.6: Vendor wise average fare amount and total earning

- Query 4

```
display(df.groupBy("payment_type").agg(count("*").alias("TotalNumberOfPayments"))
    )
```



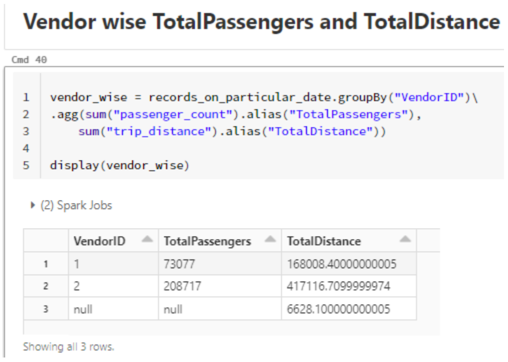
	payment_type	TotalNumberOfPayments
1	1	4694897
2	4	18065
3	3	32770
4	2	1593834
5	5	1
6	null	65441

Showing all 6 rows.

Figure 5.7: Total Number of payments for each payment mode

- Query 5

```
records_on_particular_date = df.filter((df.tpep_pickup_datetime > to_timestamp(
    lit("2020-01-01 00:00:00")))\
     &\
     (df.tpep_pickup_datetime < to_timestamp(lit("2020-01-01 23:59:59"))))

display(records_on_particular_date)
```



Vendor wise TotalPassengers and TotalDistance

Cmd 40

```
vendor_wise = records_on_particular_date.groupBy("VendorID")\
.agg(sum("passenger_count").alias("TotalPassengers"),
    sum("trip_distance").alias("TotalDistance"))

display(vendor_wise)
```

(2) Spark Jobs

	VendorID	TotalPassengers	TotalDistance
1	1	73077	168008.40000000005
2	2	208717	417116.7099999974
3	null	null	6628.100000000005

Showing all 3 rows.

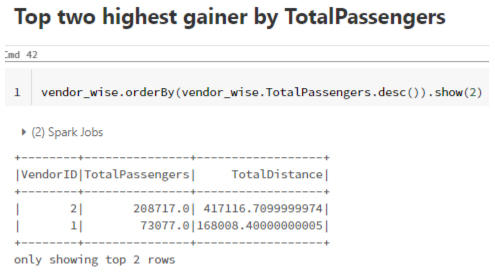
Figure 5.8: Vendorwise TotalPassengers and TotalDistance

- Query 6

```
route_wise_TotalPassengers = df.groupBy("PULocationID","DOLocationID")
      .agg(sum("passenger_count").alias("TotalPassengers"))

route_wise_TotalPassengers_descending = route_wise_TotalPassengers
      .orderBy(route_wise_TotalPassengers.TotalPassengers.desc())
display(route_wise_TotalPassengers_descending)
```

- Query 7

```
df_last_120_seconds = df.filter((df.tpep_pickup_datetime > (to_timestamp(lit
    ("2020-01-01 05:30:00"))-expr('INTERVAL 120 SECONDS'))) & (df.
    tpep_pickup_datetime < (to_timestamp(lit("2020-01-01 05:30:00")))))
```


Top two highest gainer by TotalPassengers

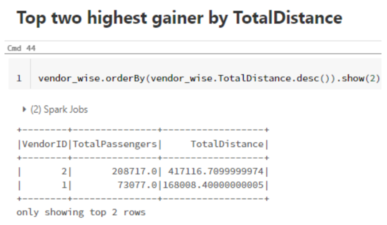
Cmd 42

```
vendor_wise.orderBy(vendor_wise.TotalPassengers.desc()).show(2)
```

▸ (2) Spark Jobs

```
+--------+---------------+------------------+
|VendorID|TotalPassengers|     TotalDistance|
+--------+---------------+------------------+
|       2|       208717.0| 417116.7099999974|
|       1|        73077.0|168008.40000000005|
+--------+---------------+------------------+
only showing top 2 rows
```

Figure 5.9: Top two highest gainer by total passengers


Top two highest gainer by TotalDistance

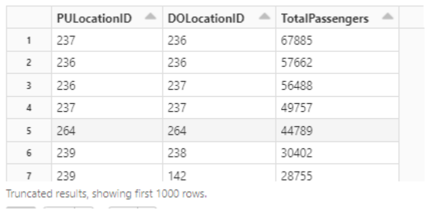
Cmd 44

```
vendor_wise.orderBy(vendor_wise.TotalDistance.desc()).show(2)
```

▸ (2) Spark Jobs

```
+--------+---------------+------------------+
|VendorID|TotalPassengers|     TotalDistance|
+--------+---------------+------------------+
|       2|       208717.0| 417116.7099999974|
|       1|        73077.0|168008.40000000005|
+--------+---------------+------------------+
only showing top 2 rows
```

Figure 5.10: Top two highest gainer by total distance covered

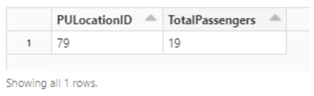


	PULocationID	DOLocationID	TotalPassengers
1	237	236	67885
2	236	236	57662
3	236	237	56488
4	237	237	49757
5	264	264	44789
6	239	238	30402
7	239	142	28755

Truncated results, showing first 1000 rows.

Figure 5.11: Route wise total passengers travelled

```
df_picup_locwise_totPassengers = df_last_120_seconds.groupBy("PULocationID").agg(
    sum("passenger_count").alias("TotalPassengers"))


display(df_picup_locwise_totPassengers.orderBy(desc("TotalPassengers")).limit(1))
```



	PULocationID	TotalPassengers
1	79	19

Showing all 1 rows.

Figure 5.12: Top pickup location with most passengers in last 2 minutes

5.2 Flatten nested JSON Column fields

JSON data is one of the examples of semi-structured data. This example can help to understand the way how these type of data is handled. Flattening JSON means taking the key value pairs of the json and put the keys as the name of the columns and values would be the field values of the columns.

example:

```
{
   "L1K0": 0,

   "L1K1": {
      "L2K1": "1",
      "L2K2": "2",
   },
   "L1K2": [
       {
         "L2K1": "10",
         "L2K2": "20",
      },
      {
         "L2K1": "100",
         "L2K2": "200",
      }
   ],
}
```

The above JSON field should be converted to the table below.

L1K0	L1K1-L2K1	L1K1-L2K2	L1K2-L2K1	L1K2-L2K2
0	1	2	10	20
0	1	2	100	200

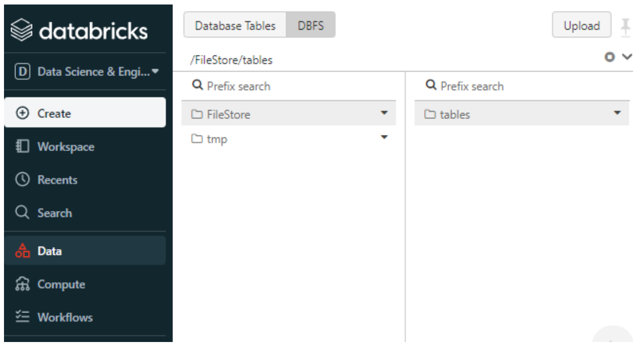
5.2.1 Task Description

- Load Flight data to DBFS.
- Flatten the two json fields(booksjon and travellerdetails).

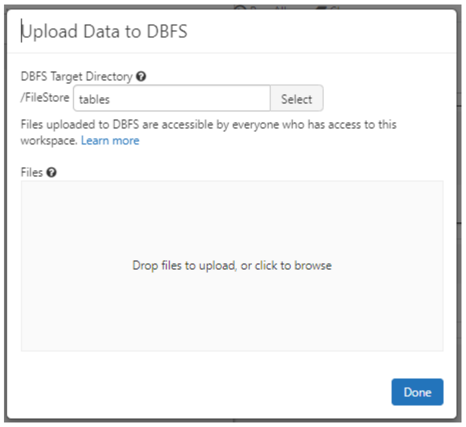
Goal: Flattening a JSON field is not a easy task to do in real life scenarios as in real life data this is quiet often that data may be corrupted and it is the data engineer's task to rescued the data from its corrupted form by analyzing it in depth.By corrupted it is meant that not easy to read in a structured format by default methods. Even after reading the data , converting it to a structured format that is also another challenge. Working with this data set will give good exposure to the formerly described problem and how to overcome that.

The key take away is the flatten() function that I have built that will take any dataframe, scans whether there exist a flattenable JSON field , automatically detects it, process it and flatten that programmatically with proper column alias as described previously.

Here in the given flight data set, only two columns named *bookjson* for storing the booking information and the *travellerdetails* for storing the traveller's information which hold JSON string. The main intention is to flatten those JSON fields.



(a) Data Tab View



Upload Data to DBFS

DBFS Target Directory
/FileStore tables Select

Files uploaded to DBFS are accessible by everyone who has access to this workspace. Learn more

Files

Drop files to upload, or click to browse

Done

(b) File Upload Dialogue Box

Figure 5.13: Opening File Upload Dialogue-box

5.2.2 Implementation

Uploading the necessary Data Set(CSV file)

1. Click on the **Data** tab from the Side Navigation Panel.Refer to the figure 5.13a.
2. Then click on the **Upload** button to to open the File Upload dialogue box. Refer to the figure 5.13b.
3. Now upload the file from the local disk to DBFS. File wil be automatically saved to the tables folder in the FileStore folder of DBFS as depicted in the figure 5.14a. One can find the files uploaded by directly navigating to the **/FileStore/tables/** folder in the DBFS as figure 5.14b.

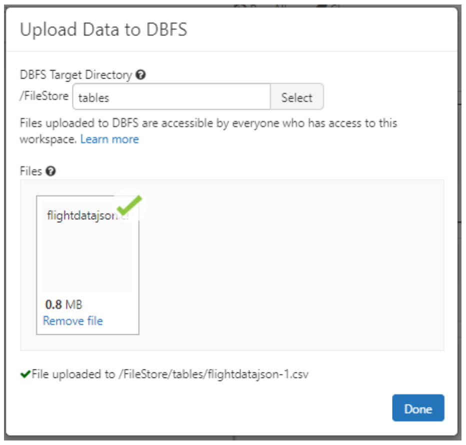
Importing Necessary Functions to work on

```
from pyspark.sql.functions import *
from pyspark.sql.types import *
```

Reading the Data from DBFS

The data used here is corrupted but this type of use case is very common in the real world. Data Engineers used to handle these type of use cases. By properly analyzing the whole data it is observed that we need to enable two spark reading options

- escaping the double quote(") by using **option('escape','\"')**. This is most important option to read this data.
- as this data is pipe(|) separated rather than comma separated so specifying the delimiter type is needed. So, it can be done by using **option("delimiter","|")**.


Upload Data to DBFS

DBFS Target Directory

/FileStore tables

Files uploaded to DBFS are accessible by everyone who has access to this workspace. Learn more

Files

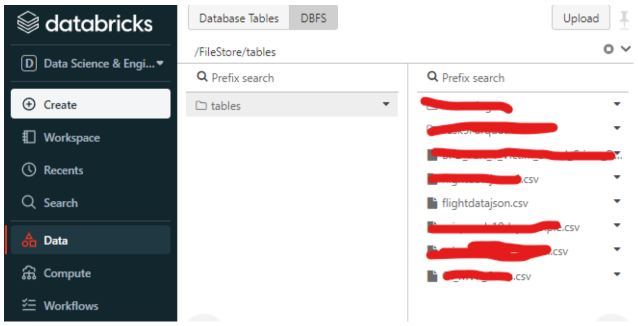
flightdatajson...

0.8 MB

Remove file

File uploaded to /FileStore/tables/flightdatajson-1.csv

(a) Uploading the CSV file



/FileStore/tables

[redacted]

[redacted]

[redacted]

[redacted].csv

flightdatajson.csv

[redacted].csv

[redacted].csv

[redacted].csv

(b) Finding the uploaded CSV file

Figure 5.14: Uploading to DBFS and Finding the Uploaded one

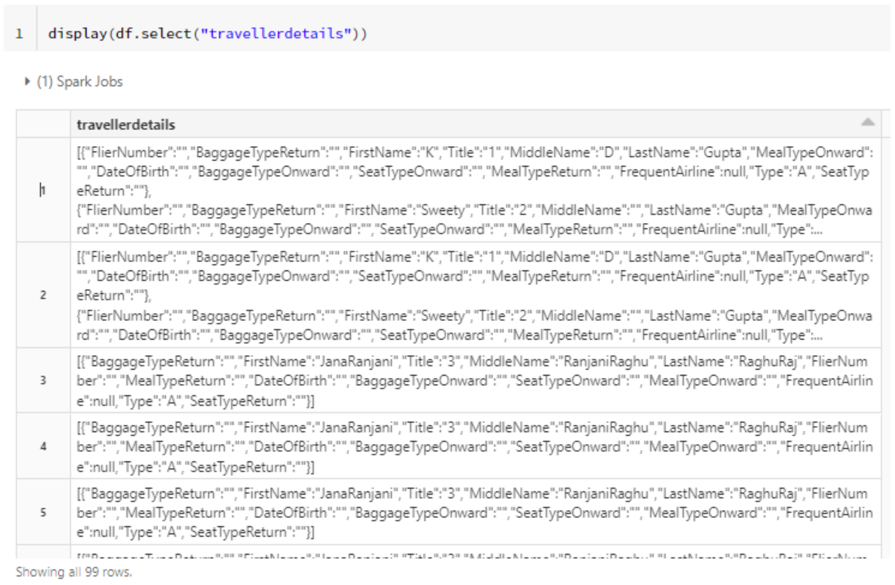
```
df = (spark.read.format("csv")
  .option("header",True)
  .option("inferSchema",True)
  .option("delimiter","|")
  .option("escape","\"")
  .load("dbfs:/FileStore/tables/flightdatajson.csv")
 )
```

Displaying the travellerdetails column

One Example field value of the travellerdetails column

This contains an array of JSON objects.

```
[
 {
  "FlierNumber": "",
  "BaggageTypeReturn": "",
  "FirstName": "Bipin",
  "Title": "1",
  "MiddleName": "",
  "LastName": "Kumar",
  "MealTypeOnward": "",
  "DateOfBirth": "",
  "BaggageTypeOnward": "",
  "SeatTypeOnward": "",
  "MealTypeReturn": "",
  "FrequentAirline": null,
  "Type": "A",
  "SeatTypeReturn": ""
 },
 {
  "FlierNumber": "",
  "BaggageTypeReturn": "",
```



```
display(df.select("travellerdetails"))
```

(1) Spark Jobs

	travellerdetails
1	[{"FlierNumber":"","BaggageTypeReturn":"","FirstName":"K","Title":"1","MiddleName":"D","LastName":"Gupta","MealTypeOnward":"","DateOfBirth":"","BaggageTypeOnward":"","SeatTypeOnward":"","MealTypeReturn":"","FrequentAirline":null,"Type":"A","SeatTypeReturn":""}, {"FlierNumber":"","BaggageTypeReturn":"","FirstName":"Sweety","Title":"2","MiddleName":"","LastName":"Gupta","MealTypeOnward":"","DateOfBirth":"","BaggageTypeOnward":"","SeatTypeOnward":"","MealTypeReturn":"","FrequentAirline":null,"Type":...
2	[{"FlierNumber":"","BaggageTypeReturn":"","FirstName":"K","Title":"1","MiddleName":"D","LastName":"Gupta","MealTypeOnward":"","DateOfBirth":"","BaggageTypeOnward":"","SeatTypeOnward":"","MealTypeReturn":"","FrequentAirline":null,"Type":"A","SeatTypeReturn":""}, {"FlierNumber":"","BaggageTypeReturn":"","FirstName":"Sweety","Title":"2","MiddleName":"","LastName":"Gupta","MealTypeOnward":"","DateOfBirth":"","BaggageTypeOnward":"","SeatTypeOnward":"","MealTypeReturn":"","FrequentAirline":null,"Type":...
3	[{"BaggageTypeReturn":"","FirstName":"JanaRanjani","Title":"3","MiddleName":"RanjaniRaghu","LastName":"RaghuRaj","FlierNumber":"","MealTypeReturn":"","DateOfBirth":"","BaggageTypeOnward":"","SeatTypeOnward":"","MealTypeOnward":"","FrequentAirline":null,"Type":"A","SeatTypeReturn":""}]
4	[{"BaggageTypeReturn":"","FirstName":"JanaRanjani","Title":"3","MiddleName":"RanjaniRaghu","LastName":"RaghuRaj","FlierNumber":"","MealTypeReturn":"","DateOfBirth":"","BaggageTypeOnward":"","SeatTypeOnward":"","MealTypeOnward":"","FrequentAirline":null,"Type":"A","SeatTypeReturn":""}]
5	[{"BaggageTypeReturn":"","FirstName":"JanaRanjani","Title":"3","MiddleName":"RanjaniRaghu","LastName":"RaghuRaj","FlierNumber":"","MealTypeReturn":"","DateOfBirth":"","BaggageTypeOnward":"","SeatTypeOnward":"","MealTypeOnward":"","FrequentAirline":null,"Type":"A","SeatTypeReturn":""}]

Showing all 99 rows.

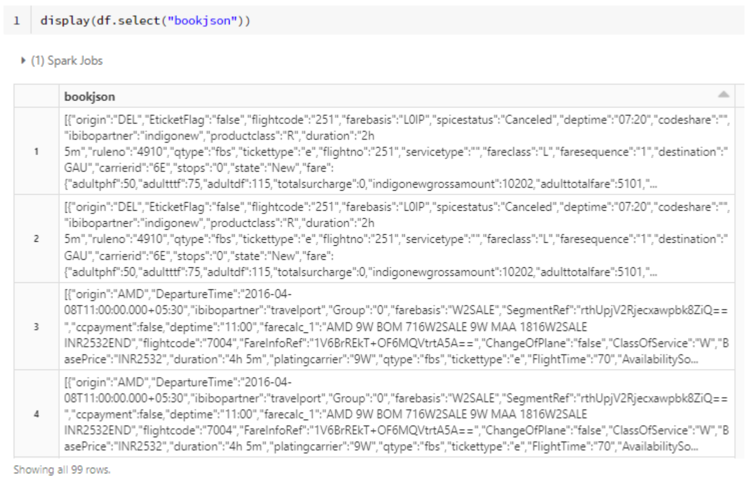
Figure 5.15: Traveller Details Column

```
    "FirstName": "Phani",
    "Title": "1",
    "MiddleName": "Kumar",
    "LastName": "Mandal",
    "MealTypeOnward": "",
    "DateOfBirth": "",
    "BaggageTypeOnward": "",
    "SeatTypeOnward": "",
    "MealTypeReturn": "",
    "FrequentAirline": null,
    "Type": "A",
    "SeatTypeReturn": ""
  },
  {
    "FlierNumber": "",
    "BaggageTypeReturn": "",
    "FirstName": "Mohit",
    "Title": "1",
    "MiddleName": "",
    "LastName": "Anand",
    "MealTypeOnward": "",
    "DateOfBirth": "",
    "BaggageTypeOnward": "",
    "SeatTypeOnward": "",
    "MealTypeReturn": "",
    "FrequentAirline": null,
    "Type": "A",
    "SeatTypeReturn": ""
  }
]
```

One Example field value of the bookjson column

```
[
 {
  "origin": "DEL",
  "EticketFlag": "false",
  "flightcode": "251",
  "farebasis": "L0IP",
  "spicestatus": "Canceled",
  "deptime": "07:20",
  "codeshare": "",
  "ibibopartner": "indigonew",
  "productclass": "R",
  "duration": "2h 5m",
  "ruleno": "4910",
  "qtype": "fbs",
  "tickettype": "e",
  "flightno": "251",
  "servicetype": "",
  "fareclass": "L",
  "faresequence": "1",
  "destination": "GAU",
  "carrierid": "6E",
  "stops": "0",
  "state": "New",
  "fare":
  {
   "adultphf": 50,
   "adulttttf": 75,
   "adultdf": 115,
   "totalsurcharge": 0,
   "indigonewgrossamount": 10202,
   "adulttotalfare": 5101,
   "totalcommission": 0,
   "adultbasefare": 4150,
   "totalpassengerhandlingfee": 0,
   "adultudf": 562,
   "adultpassengerservicefee": 149,
   "totalpassengerservicefee": 0,
   "totalothers": 0,
   "childtotalfare": 0,
   "totalbasefare": 8300,
   "totalfare": 10134,
   "discount": 68,
   "totaludf": 0,
   "transactionfee": 0,
   "totalservicetax": 0,
   "totaljn": 0,
   "totalchandlingfee": 0
  },
  "journeysellkey": "6E~ 251~ ~~DEL~01/29/2016 07:20~GAU~01/29/2016 09:25~",
  "fareapptype": "Route",
  "warnings": "Refundable",
  "farestatus": "Default",
  "signature": "57QjdgcZU1I=|ghbKTdJzLN3moAq/u+lQ+6
```

```
       tuDFs4vWkqFhGXb1JbtTlK77rn4S4vpRGz51mxxElGdT4Vhhyk24VwGPnnctDu1p2p+q//
       zLT4PU9ryJocDyXslwLhvPm8uqG6rVkj0GApHQ7d3dDCKICoEPbrRNm5gw==",
   "onwardflights": [],
   "seatingclass": "E",
   "dest_vid": "3518826103379012321",
   "src_vid": "2820046943342890302",
   "faresellkey": "0~L~~L0IP~4910~~1~X",
   "dotentative": "true",
   "ssrcode_baggage":
   [
    {
      "fare": 1000,
      "code": "XBPA",
      "desc": "Prepaid Excess Baggage - 5Kg"
    },
    {
      "fare": 1750,
      "code": "XBPB",
      "desc": "Prepaid Excess Baggage - 10Kg"
    },
    {
      "fare": 2250,
      "code": "XBPC",
      "desc": "Prepaid Excess Baggage - 15Kg"
    },
    {
      "fare": 3750,
      "code": "XBPD",
      "desc": "Prepaid Excess Baggage - 30Kg"
    }
   ],
   "splitduration": "2h 5m",
   "farerule": "",
   "inventorylegid": "0",
   "ssrcode_meal":
   [
    {
      "fare": 250,
      "code": "6ENVML",
      "desc": "Non Vegetarian Meal"
    },
    {
      "fare": 250,
      "code": "6EVGML",
      "desc": "Vegetarian Meal"
    }
   ],
   "airline": "IndiGo",
   "rowbody": " ",
   "depdate": "2016-01-29t0720",
   "arrtime": "09:25",
   "arrdate": "2016-01-29t0925"
 }
]
```



```
display(df.select("bookjson"))
```

(1) Spark Jobs

	bookjson
1	[{"origin":"DEL","EticketFlag":"false","flightcode":"251","farebasis":"L0IP","spicestatus":"Canceled","deptime":"07:20","codeshare":"", "ibibopartner":"indigonew","productclass":"R","duration":"2h 5m","ruleno":"4910","qtype":"fbs","tickettype":"e","flightno":"251","servicetype":"","fareclass":"L","faresequence":"1","destination":"GAU","carrierid":"6E","stops":"0","state":"New","fare": {"adultphf":50,"adulttf":75,"adultdf":115,"totalsurcharge":0,"indigonewgrossamount":10202,"adulttotalfare":5101,"...
2	[{"origin":"DEL","EticketFlag":"false","flightcode":"251","farebasis":"L0IP","spicestatus":"Canceled","deptime":"07:20","codeshare":"", "ibibopartner":"indigonew","productclass":"R","duration":"2h 5m","ruleno":"4910","qtype":"fbs","tickettype":"e","flightno":"251","servicetype":"","fareclass":"L","faresequence":"1","destination":"GAU","carrierid":"6E","stops":"0","state":"New","fare": {"adultphf":50,"adulttf":75,"adultdf":115,"totalsurcharge":0,"indigonewgrossamount":10202,"adulttotalfare":5101,"...
3	[{"origin":"AMD","DepartureTime":"2016-04-08T11:00:00.000+05:30","ibibopartner":"travelport","Group":"0","farebasis":"W2SALE","SegmentRef":"rthUpjV2Rjecxawpbk8ZiQ==","ccpayment":false,"deptime":"11:00","farecalc_1":"AMD 9W BOM 716W2SALE 9W MAA 1816W2SALE INR2532END","flightcode":"7004","FareInfoRef":"1V6BrREkT+OF6MQVtrtA5A==","ChangeOfPlane":"false","ClassOfService":"W","BasePrice":"INR2532","duration":"4h 5m","platingcarrier":"9W","qtype":"fbs","tickettype":"e","FlightTime":"70","AvailabilitySo...
4	[{"origin":"AMD","DepartureTime":"2016-04-08T11:00:00.000+05:30","ibibopartner":"travelport","Group":"0","farebasis":"W2SALE","SegmentRef":"rthUpjV2Rjecxawpbk8ZiQ==","ccpayment":false,"deptime":"11:00","farecalc_1":"AMD 9W BOM 716W2SALE 9W MAA 1816W2SALE INR2532END","flightcode":"7004","FareInfoRef":"1V6BrREkT+OF6MQVtrtA5A==","ChangeOfPlane":"false","ClassOfService":"W","BasePrice":"INR2532","duration":"4h 5m","platingcarrier":"9W","qtype":"fbs","tickettype":"e","FlightTime":"70","AvailabilitySo...

Showing all 99 rows.

Figure 5.16: Book Json Column

Displaying the bookjson column

Inferring the schema from complex nested json using the RDD api map()

```
travellerdetails_schema = ArrayType(
    spark.read.json(df.rdd.map(lambda row : row.travellerdetails)).schema
 )
bookjson_schema = ArrayType(
   spark.read.json(df.rdd.map(lambda row : row.bookjson)).schema
)
```

Displaying Travellerdetails schema in pretty format

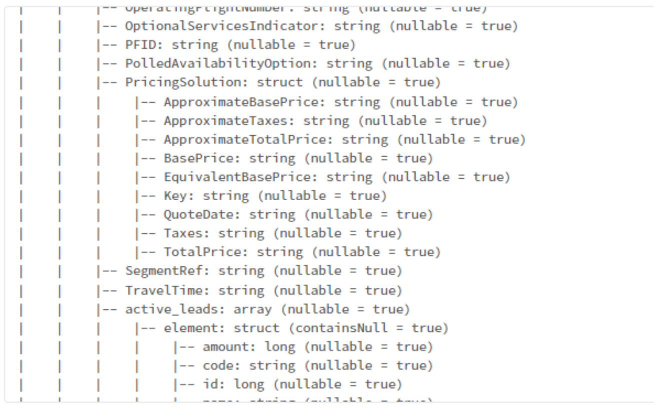
```
import json

print(json.dumps(travellerdetails_schema.jsonValue(), indent=4))
```

Displaying Bookjson schema in pretty format

```
print(json.dumps(bookjson_schema.jsonValue(), indent=4))
```

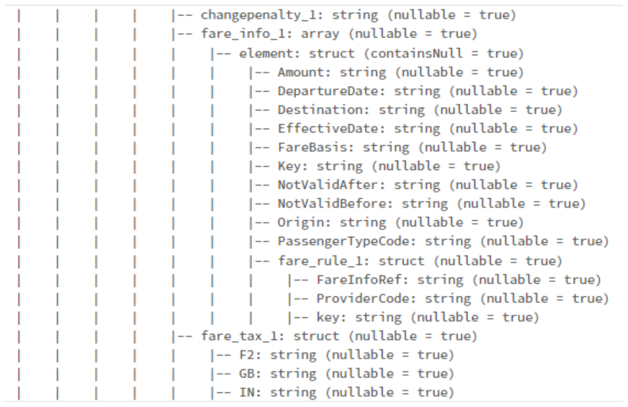
Reconstructing the Dataframe with the help of created JSON schema for travellerdetails column and bookjson column

```
structured_df = (
   df
   .withColumn('travellerdetails',from_json('travellerdetails',travellerdetails_schema
       ))
   .withColumn('bookjson',from_json('bookjson',bookjson_schema))
)
structured_df.printSchema()
```



```
|    |    |-- OperatingFlightNumber: string (nullable = true)
|    |    |-- OptionalServicesIndicator: string (nullable = true)
|    |    |-- PFID: string (nullable = true)
|    |    |-- PolledAvailabilityOption: string (nullable = true)
|    |    |-- PricingSolution: struct (nullable = true)
|    |    |    |-- ApproximateBasePrice: string (nullable = true)
|    |    |    |-- ApproximateTaxes: string (nullable = true)
|    |    |    |-- ApproximateTotalPrice: string (nullable = true)
|    |    |    |-- BasePrice: string (nullable = true)
|    |    |    |-- EquivalentBasePrice: string (nullable = true)
|    |    |    |-- Key: string (nullable = true)
|    |    |    |-- QuoteDate: string (nullable = true)
|    |    |    |-- Taxes: string (nullable = true)
|    |    |    |-- TotalPrice: string (nullable = true)
|    |    |-- SegmentRef: string (nullable = true)
|    |    |-- TravelTime: string (nullable = true)
|    |    |-- active_leads: array (nullable = true)
|    |    |    |-- element: struct (containsNull = true)
|    |    |    |    |-- amount: long (nullable = true)
|    |    |    |    |-- code: string (nullable = true)
|    |    |    |    |-- id: long (nullable = true)
|    |    |    |    |-- name: string (nullable = true)
```

Figure 5.17: Snapshot of converted schema



```
|    |    |    |    |-- changepenalty_1: string (nullable = true)
|    |    |    |    |-- fare_info_1: array (nullable = true)
|    |    |    |    |    |-- element: struct (containsNull = true)
|    |    |    |    |    |    |-- Amount: string (nullable = true)
|    |    |    |    |    |    |-- DepartureDate: string (nullable = true)
|    |    |    |    |    |    |-- Destination: string (nullable = true)
|    |    |    |    |    |    |-- EffectiveDate: string (nullable = true)
|    |    |    |    |    |    |-- FareBasis: string (nullable = true)
|    |    |    |    |    |    |-- Key: string (nullable = true)
|    |    |    |    |    |    |-- NotValidAfter: string (nullable = true)
|    |    |    |    |    |    |-- NotValidBefore: string (nullable = true)
|    |    |    |    |    |    |-- Origin: string (nullable = true)
|    |    |    |    |    |    |-- PassengerTypeCode: string (nullable = true)
|    |    |    |    |    |    |-- fare_rule_1: struct (nullable = true)
|    |    |    |    |    |    |    |-- FareInfoRef: string (nullable = true)
|    |    |    |    |    |    |    |-- ProviderCode: string (nullable = true)
|    |    |    |    |    |    |    |-- key: string (nullable = true)
|    |    |    |    |-- fare_tax_1: struct (nullable = true)
|    |    |    |    |    |-- F2: string (nullable = true)
|    |    |    |    |    |-- GB: string (nullable = true)
|    |    |    |    |    |-- IN: string (nullable = true)
```

Figure 5.18: Snapshot of converted schema

A function which takes a Dataframe as an argument, process that and return the flattened Dataframe

```
def flatten(df):
   # gathering the Complex Fields (ArrayTypes and StructTypes) in the Schema of the
       Dataframe
   complex_fields = {field.name: field.dataType
                    for field in df.schema.fields if type(field.dataType) == ArrayType
                         or type(field.dataType) == StructType}
   print(f"The list of complex fields is : \n {complex_fields.keys()}")
   while len(complex_fields)!=0:
      col_name=list(complex_fields.keys())[0]
      print(F"Processing the column: {col_name} of the Type : {str(type(complex_fields
           [col_name]))}")

      # if StructType then convert all sub element to columns.
```

```
        # i.e. flatten structs
        if (type(complex_fields[col_name]) == StructType):
           expanded = [col(col_name+'.'+k).alias(col_name+'_'+k) for k in [ n.name for n
                in complex_fields[col_name]]]
           # print(expanded,end="=============================================\n")
           df=df.select("*", *expanded).drop(col_name)

        # if ArrayType then add the Array Elements as Rows using the explode function
        # i.e. explode Arrays
        elif (type(complex_fields[col_name]) == ArrayType):
           df=df.withColumn(col_name,explode_outer(col_name))

        # recompute remaining Complex Fields in Schema
        complex_fields = dict([(field.name, field.dataType)
                        for field in df.schema.fields
                        if type(field.dataType) == ArrayType or type(field.dataType) ==
                            StructType])
     return df
```

Flattening the JSON fields

```
# Making the column name case sensitive
spark.sql("set spark.sql.caseSensitive=true")
# Flattening the Dataframe
flattened_df = flatten(structured_df)

"""
OUTPUT:

   The list of complex fields is :
    dict_keys(['travellerdetails', 'bookjson'])
   Processing the column: travellerdetails of the Type : <class 'pyspark.sql.types.
       ArrayType'>
   Processing the column: travellerdetails of the Type : <class 'pyspark.sql.types.
       StructType'>
   Processing the column: bookjson of the Type : <class 'pyspark.sql.types.ArrayType'>
   Processing the column: bookjson of the Type : <class 'pyspark.sql.types.StructType
       '>
   Processing the column: bookjson_PricingSolution of the Type : <class 'pyspark.sql.
       types.StructType'>
   Processing the column: bookjson_active_leads of the Type : <class 'pyspark.sql.
       types.ArrayType'>
   Processing the column: bookjson_active_leads of the Type : <class 'pyspark.sql.
       types.StructType'>
   Processing the column: bookjson_booking_info_1 of the Type : <class 'pyspark.sql.
       types.ArrayType'>
   Processing the column: bookjson_booking_info_1 of the Type : <class 'pyspark.sql.
       types.StructType'>
   Processing the column: bookjson_fare of the Type : <class 'pyspark.sql.types.
       StructType'>
   Processing the column: bookjson_fare_charges of the Type : <class 'pyspark.sql.
       types.ArrayType'>
   Processing the column: bookjson_fare_charges of the Type : <class 'pyspark.sql.
       types.StructType'>
   Processing the column: bookjson_fare_info_1 of the Type : <class 'pyspark.sql.types
       .ArrayType'>
```

```
Processing the column: bookjson_fare_info_1 of the Type : <class 'pyspark.sql.types
    .StructType'>
Processing the column: bookjson_fare_tax_1 of the Type : <class 'pyspark.sql.types.
    StructType'>
Processing the column: bookjson_farekey_dict of the Type : <class 'pyspark.sql.
    types.StructType'>
Processing the column: bookjson_gocash_breakup of the Type : <class 'pyspark.sql.
    types.ArrayType'>
Processing the column: bookjson_gocash_breakup of the Type : <class 'pyspark.sql.
    types.StructType'>
Processing the column: bookjson_intermediate of the Type : <class 'pyspark.sql.
    types.StructType'>
Processing the column: bookjson_intermediate0 of the Type : <class 'pyspark.sql.
    types.StructType'>
Processing the column: bookjson_leadlist of the Type : <class 'pyspark.sql.types.
    ArrayType'>
Processing the column: bookjson_leadlist of the Type : <class 'pyspark.sql.types.
    StructType'>
Processing the column: bookjson_onwardflights of the Type : <class 'pyspark.sql.
    types.ArrayType'>
Processing the column: bookjson_onwardflights of the Type : <class 'pyspark.sql.
    types.StructType'>
Processing the column: bookjson_seg of the Type : <class 'pyspark.sql.types.
    ArrayType'>
Processing the column: bookjson_seg of the Type : <class 'pyspark.sql.types.
    StructType'>
Processing the column: bookjson_sell_dict of the Type : <class 'pyspark.sql.types.
    StructType'>
Processing the column: bookjson_ssrcode_baggage of the Type : <class 'pyspark.sql.
    types.ArrayType'>
Processing the column: bookjson_ssrcode_baggage of the Type : <class 'pyspark.sql.
    types.StructType'>
Processing the column: bookjson_ssrcode_meal of the Type : <class 'pyspark.sql.
    types.ArrayType'>
Processing the column: bookjson_ssrcode_meal of the Type : <class 'pyspark.sql.
    types.StructType'>
Processing the column: bookjson_travelerref of the Type : <class 'pyspark.sql.types
    .StructType'>
Processing the column: bookjson_fare_charges_booking_info_1 of the Type : <class '
    pyspark.sql.types.ArrayType'>
Processing the column: bookjson_fare_charges_booking_info_1 of the Type : <class '
    pyspark.sql.types.StructType'>
Processing the column: bookjson_fare_charges_fare_info_1 of the Type : <class '
    pyspark.sql.types.ArrayType'>
Processing the column: bookjson_fare_charges_fare_info_1 of the Type : <class '
    pyspark.sql.types.StructType'>
Processing the column: bookjson_fare_charges_fare_tax_1 of the Type : <class '
    pyspark.sql.types.StructType'>
Processing the column: bookjson_fare_info_1_fare_rule_1 of the Type : <class '
    pyspark.sql.types.StructType'>
Processing the column: bookjson_intermediate_fare of the Type : <class 'pyspark.sql
    .types.StructType'>
Processing the column: bookjson_onwardflights_fare of the Type : <class 'pyspark.
    sql.types.StructType'>
Processing the column: bookjson_onwardflights_ssrcode_baggage of the Type : <class
    'pyspark.sql.types.ArrayType'>
Processing the column: bookjson_onwardflights_ssrcode_baggage of the Type : <class
    'pyspark.sql.types.StructType'>
    Processing the column: bookjson_travelerref_adult_ref of the Type : <class 'pyspark
        .sql.types.ArrayType'>
    Processing the column: bookjson_travelerref_child_ref of the Type : <class 'pyspark
        .sql.types.ArrayType'>
    Processing the column: bookjson_travelerref_infant_ref of the Type : <class '
        pyspark.sql.types.ArrayType'>
    Processing the column: bookjson_fare_charges_fare_info_1_fare_rule_1 of the Type :
        <class 'pyspark.sql.types.StructType'>

"""
```

Schema After Flattening

- **Part of Traveller-Details**

```
|-- travelamount: integer (nullable = true)
|-- refundamount: integer (nullable = true)
|-- airlinepnr: string (nullable = true)
|-- mihpayid: string (nullable = true)
|-- typeoftravel: string (nullable = true)
|-- booking_type: string (nullable = true)
|-- travellerdetails_Age: string (nullable = true)
|-- travellerdetails_BaggageTypeOnward: string (nullable = true)
|-- travellerdetails_BaggageTypeReturn: string (nullable = true)
|-- travellerdetails_DateOfBirth: string (nullable = true)
|-- travellerdetails_DateOfExpiry: string (nullable = true)
|-- travellerdetails_DateOfIssue: string (nullable = true)
|-- travellerdetails_Email: string (nullable = true)
|-- travellerdetails_FirstName: string (nullable = true)
|-- travellerdetails_FlierNumber: string (nullable = true)
|-- travellerdetails_FrequentAirline: string (nullable = true)
|-- travellerdetails_LastName: string (nullable = true)
|-- travellerdetails_MealTypeOnward: string (nullable = true)
|-- travellerdetails_MealTypeReturn: string (nullable = true)
|-- travellerdetails_MiddleName: string (nullable = true)
|-- travellerdetails_Mobile: string (nullable = true)
|-- travellerdetails_Nationality: string (nullable = true)
|-- travellerdetails_Passport: string (nullable = true)
|-- travellerdetails_PlaceOfIssue: string (nullable = true)
```

- **Part of Book-JSON**

```
|-- bookjson_seg_destination: string (nullable = true)
|-- bookjson_seg_flightno: string (nullable = true)
|-- bookjson_seg_journeytype: string (nullable = true)
|-- bookjson_seg_origin: string (nullable = true)
|-- bookjson_seg_rph: string (nullable = true)
|-- bookjson_sell_dict_C: long (nullable = true)
|-- bookjson_sell_dict_E: long (nullable = true)
|-- bookjson_sell_dict_F: long (nullable = true)
|-- bookjson_sell_dict_P2: long (nullable = true)
|-- bookjson_ssrcode_baggage_code: string (nullable = true)
|-- bookjson_ssrcode_baggage_desc: string (nullable = true)
|-- bookjson_ssrcode_baggage_fare: long (nullable = true)
|-- bookjson_ssrcode_baggage_originalfare: long (nullable = true)
|-- bookjson_ssrcode_meal_code: string (nullable = true)
|-- bookjson_ssrcode_meal_desc: string (nullable = true)
|-- bookjson_ssrcode_meal_fare: long (nullable = true)
|-- bookjson_travelerref_adult_ref: string (nullable = true)
|-- bookjson_travelerref_child_ref: string (nullable = true)
|-- bookjson_travelerref_infant_ref: string (nullable = true)
|-- bookjson_fare_charges_booking_info_1_BookingCode: string (nullable = true
    )
|-- bookjson_fare_charges_booking_info_1_CabinClass: string (nullable = true)
|-- bookjson_fare_charges_booking_info_1_FareInfoRef: string (nullable = true
    )
|-- bookjson_fare_charges_booking_info_1_SegmentRef: string (nullable = true)
|-- bookjson_fare_charges_booking_info_1_bookingclass: string (nullable =
    true)
|-- bookjson_fare_charges_booking_info_1_farebasis: string (nullable = true)
|-- bookjson_fare_charges_fare_info_1_Amount: string (nullable = true)
|-- bookjson_fare_charges_fare_info_1_DepartureDate: string (nullable = true)
|-- bookjson_fare_charges_fare_info_1_Destination: string (nullable = true)
|-- bookjson_fare_charges_fare_info_1_EffectiveDate: string (nullable = true)
|-- bookjson_fare_charges_fare_info_1_FareBasis: string (nullable = true)
|-- bookjson_fare_charges_fare_info_1_Key: string (nullable = true)
|-- bookjson_fare_charges_fare_info_1_NotValidAfter: string (nullable = true)
|-- bookjson_fare_charges_fare_info_1_NotValidBefore: string (nullable = true
    )
|-- bookjson_fare_charges_fare_info_1_Origin: string (nullable = true)
|-- bookjson_fare_charges_fare_info_1_PassengerTypeCode: string (nullable =
    true)
|-- bookjson_fare_charges_fare_tax_1_F2: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_GB: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_IN: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_JN: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_PZ: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_UB: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_US: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_WO: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_XA: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_XY: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_YC: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_YM: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_YQ: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_YR: string (nullable = true)
|-- bookjson_fare_charges_fare_tax_1_ZR: string (nullable = true)
|-- bookjson_fare_info_1_fare_rule_1_FareInfoRef: string (nullable = true)
|-- bookjson_fare_info_1_fare_rule_1_ProviderCode: string (nullable = true)
|-- bookjson_fare_info_1_fare_rule_1_key: string (nullable = true)
|-- bookjson_intermediate_fare_adultbasefare: long (nullable = true)
|-- bookjson_intermediate_fare_adultphf: long (nullable = true)
|-- bookjson_intermediate_fare_adultpsf: long (nullable = true)
|-- bookjson_intermediate_fare_adultsbct: long (nullable = true)
|-- bookjson_intermediate_fare_adultsvct: long (nullable = true)
|-- bookjson_intermediate_fare_adulttotalfare: long (nullable = true)
|-- bookjson_intermediate_fare_adulttttf: long (nullable = true)
|-- bookjson_intermediate_fare_adultudf: long (nullable = true)
|-- bookjson_intermediate_fare_adultudfa: long (nullable = true)
|-- bookjson_intermediate_fare_discount: long (nullable = true)
|-- bookjson_intermediate_fare_totalbasefare: long (nullable = true)
|-- bookjson_intermediate_fare_totalcommission: long (nullable = true)
|-- bookjson_intermediate_fare_totalfare: long (nullable = true)
|-- bookjson_intermediate_fare_totalsurcharge: long (nullable = true)
|-- bookjson_onwardflights_fare_adultF2: long (nullable = true)
|-- bookjson_onwardflights_fare_adultGB: long (nullable = true)
|-- bookjson_onwardflights_fare_adultPZ: long (nullable = true)
|-- bookjson_onwardflights_fare_adultUB: long (nullable = true)
|-- bookjson_onwardflights_fare_adultbasefare: long (nullable = true)
|-- bookjson_onwardflights_fare_adultcancelpenalty_1: long (nullable = true)
|-- bookjson_onwardflights_fare_adultchangepenalty_1: long (nullable = true)
|-- bookjson_onwardflights_fare_adultfuelsurcharge: long (nullable = true)
|-- bookjson_onwardflights_fare_adultpassengerservicefee: long (nullable =
    true)
|-- bookjson_onwardflights_fare_adultpsf: long (nullable = true)
|-- bookjson_onwardflights_fare_adultsbct: long (nullable = true)
|-- bookjson_onwardflights_fare_adultsvct: long (nullable = true)
|-- bookjson_onwardflights_fare_adulttaxes: long (nullable = true)
|-- bookjson_onwardflights_fare_adulttotalfare: long (nullable = true)
|-- bookjson_onwardflights_fare_adulttravelportservicetax: long (nullable =
    true)
|-- bookjson_onwardflights_fare_adulttttf: long (nullable = true)
|-- bookjson_onwardflights_fare_adultuserdevelopmentfee: long (nullable =
    true)
|-- bookjson_onwardflights_fare_airarabiagrossamount: long (nullable = true)
|-- bookjson_onwardflights_fare_childtotalfare: long (nullable = true)
|-- bookjson_onwardflights_fare_indigonewgrossamount: long (nullable = true)
|-- bookjson_onwardflights_fare_servicetax: long (nullable = true)
|-- bookjson_onwardflights_fare_totalbasefare: long (nullable = true)
|-- bookjson_onwardflights_fare_totalchandlingfee: long (nullable = true)
```

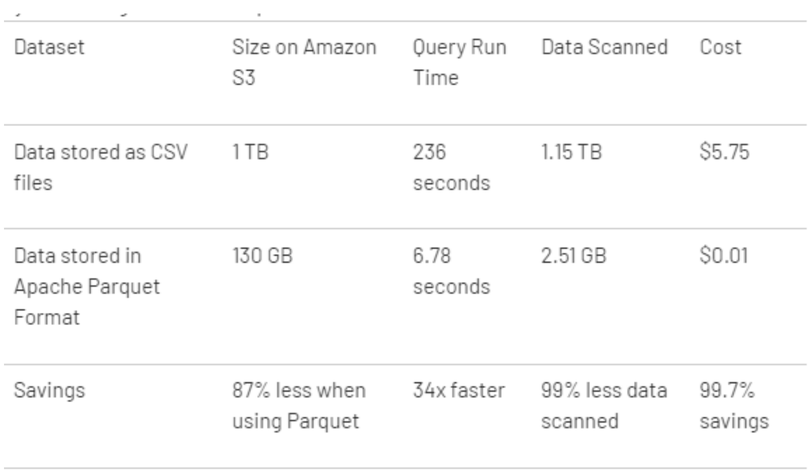
5.3 Handling Partitioned Data and Saving as Parquet File Format

5.3.1 A small background of Parquet File formats

Apache Parquet is an open source, column-oriented data file format designed for efficient data storage and retrieval. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk. Apache Parquet is designed to be a common interchange format for both batch and interactive workloads. It is similar to other columnar-storage file formats available in Hadoop, namely RCFile and ORC. For more details refer to the Apache Parquet Documentation[2]. For a brief comparison between using CSV and using Parquet in production level refer to the figure 5.19.

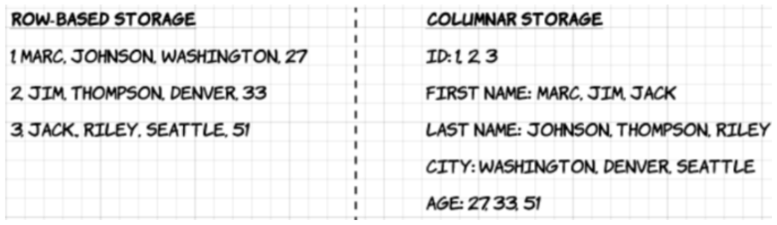
Three main key points to remember while describing about the parquet files are

- **Columnar:** Unlike row-based formats such as CSV or Avro, Apache Parquet is column-oriented – meaning the values of each table column are stored next to each other, rather than those of each record. Refer to the figure 5.20 for a brief comparison.
- **Open-Source:** Parquet is free to use and open source under the Apache Hadoop license, and is compatible with most Hadoop data processing frameworks. To quote the project website, "Apache Parquet is... available to any project... regardless of the choice of data processing framework, data model, or programming language."
- **Self-Describing:** In addition to data, a Parquet file contains metadata including schema and structure. Each file stores both the data and the standards used for accessing each record – making it easier to decouple services that write, store, and read Parquet files.



Dataset	Size on Amazon S3	Query Run Time	Data Scanned	Cost
Data stored as CSV files	1 TB	236 seconds	1.15 TB	$5.75
Data stored in Apache Parquet Format	130 GB	6.78 seconds	2.51 GB	$0.01
Savings	87% less when using Parquet	34x faster	99% less data scanned	99.7% savings

Figure 5.19: A brief Comparison b/w CSV and Parquet when used in Product Level



ROW-BASED STORAGE	COLUMNAR STORAGE
1 MARC, JOHNSON, WASHINGTON, 27	ID: 1, 2, 3
2 JIM, THOMPSON, DENVER, 33	FIRST NAME: MARC, JIM, JACK
3 JACK, RILEY, SEATTLE, 51	LAST NAME: JOHNSON, THOMPSON, RILEY
	CITY: WASHINGTON, DENVER, SEATTLE
	AGE: 27 33, 51

Figure 5.20: A visual comparison between Row-Based and Columnar Data

5.3.2 Task Description

Mixpanel is a business analytics service company. It tracks user interactions with web and mobile applications and provides tools for targeted communication with them. Data collected is used to build custom reports and measure user engagement and retention. Mixpanel works with web applications, in particular SaaS(SaaS stands for Software as a Service), but also supports mobile apps.

The data, I will be working with here, is sample analytic data for viewers of a web site of 10 consecutive days starting from $01 - 10 - 2020$ to $10 - 10 - 2020$ having three columns named

1. *distinct_id* for each unique viewers
2. *event* that contains two values that represent two type of user behaviour
   - *mediaReady* which means the video contained in that website is not watched but a viewer opened that website.
   - *videoWatched* which means the video contained in that website is watched.
3. *time* that stores the data when the website is opened or video is watched.

Below is the task description that depicts what to do with this data.

- Upload the mixpanel data to DBFS and read that data into spark dataframe.
- Calculate unique viewers for each day
- Calculate unique viewers for all possible date ranges
- Write result back to DBFS in date partitioned external table
- Load Mixpanel incremental data from DBFS , which contains the viewers information only for the date $11-10-2020$. By the term *incremental data*, which is a very important terminology in data engineering meaning *the data which contains only the new records or only those records which are updated from the previous set of records*
- Calculate unique viewers basis incremental data and update result table

Goal: Main goal for this experiment is to handle partitioned data i.e. here the whole data is partitioned into different dates , and all the records related to a particular date is saved into a particular folder with good nomenclature. Handling parquet data.

5.3.3 Implementation

Importing Necessary Functions to work on

```
from pyspark.sql.functions import *
from pyspark.sql.types import *
```

Defining Custom Schema for Better Performance

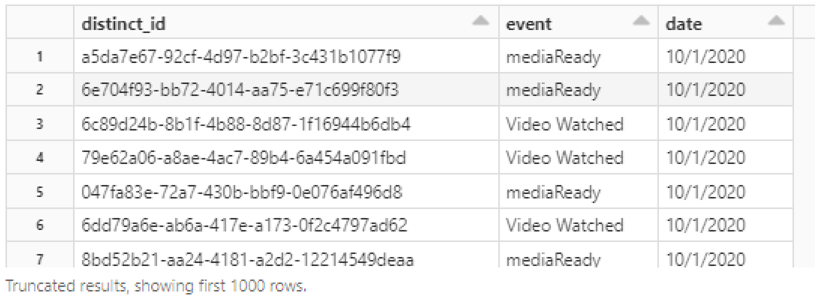
```
schemaDF = StructType([StructField("distinct_id",StringType(),True),StructField("event
    ",StringType(),True),StructField("date",StringType(),True)])
```

Reading the Data from DBFS

```
df = spark.read.csv("/FileStore/tables/mixpanel_10day_sample.csv",header=True,schema=
    schemaDF)
display(df)
```

Typecasting the date column in a proper Date format

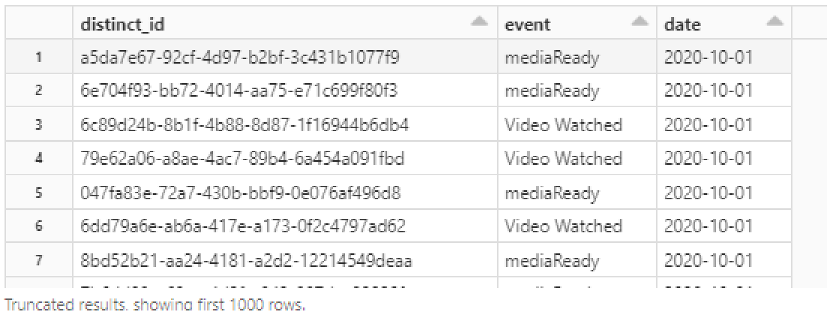
```
df = df.select("*").withColumn("date",to_date(col("date"),"M/d/yyyy"))
display(df)
```



	distinct_id	event	date
1	a5da7e67-92cf-4d97-b2bf-3c431b1077f9	mediaReady	10/1/2020
2	6e704f93-bb72-4014-aa75-e71c699f80f3	mediaReady	10/1/2020
3	6c89d24b-8b1f-4b88-8d87-1f16944b6db4	Video Watched	10/1/2020
4	79e62a06-a8ae-4ac7-89b4-6a454a091fbd	Video Watched	10/1/2020
5	047fa83e-72a7-430b-bbf9-0e076af496d8	mediaReady	10/1/2020
6	6dd79a6e-ab6a-417e-a173-0f2c4797ad62	Video Watched	10/1/2020
7	8bd52b21-aa24-4181-a2d2-12214549deaa	mediaReady	10/1/2020

Truncated results, showing first 1000 rows.

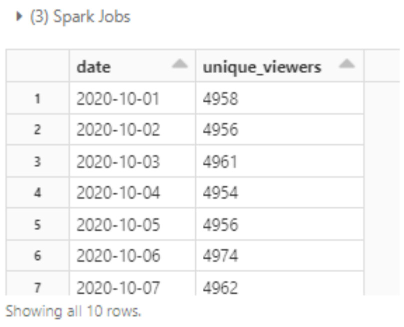
Figure 5.21: Overview of Mixpanel data



	distinct_id	event	date
1	a5da7e67-92cf-4d97-b2bf-3c431b1077f9	mediaReady	2020-10-01
2	6e704f93-bb72-4014-aa75-e71c699f80f3	mediaReady	2020-10-01
3	6c89d24b-8b1f-4b88-8d87-1f16944b6db4	Video Watched	2020-10-01
4	79e62a06-a8ae-4ac7-89b4-6a454a091fbd	Video Watched	2020-10-01
5	047fa83e-72a7-430b-bbf9-0e076af496d8	mediaReady	2020-10-01
6	6dd79a6e-ab6a-417e-a173-0f2c4797ad62	Video Watched	2020-10-01
7	8bd52b21-aa24-4181-a2d2-12214549deaa	mediaReady	2020-10-01

Truncated results, showing first 1000 rows.

Figure 5.22: The date column is typecasted in a proper Date format



(3) Spark Jobs

	date	unique_viewers
1	2020-10-01	4958
2	2020-10-02	4956
3	2020-10-03	4961
4	2020-10-04	4954
5	2020-10-05	4956
6	2020-10-06	4974
7	2020-10-07	4962

Showing all 10 rows.

Figure 5.23: Displaying number of unique viewers for each date

Unique Viewers for each date

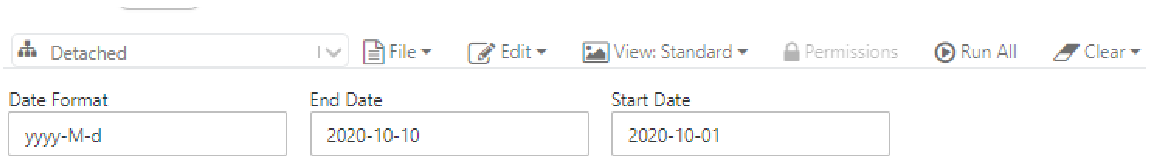
```
display(df.groupBy("date").agg(countDistinct("distinct_id").alias("unique_viewers")).
    orderBy("date"))
```

Creating a function that will return a dataframe with unique Viewers for any possible date range

```
def unique_viewers(data_frame,start_date,end_date,date_format):
return data_frame.where((to_date(col("date"),date_format)>= start_date) & (to_date(col
    ("date"),date_format)<= end_date)).groupBy("date").agg(countDistinct("distinct_id
    ").alias("unique_viewers")).orderBy("date")
```

Creating widgets with the help of dbutils apis to take suitable user input

```
dbutils.widgets.text("w1", "2020-10-01", label = "Start Date")
dbutils.widgets.text("w2", "2020-10-10", label = "End Date")
dbutils.widgets.text("w3", "yyyy-M-d", label = "Date Format")
```

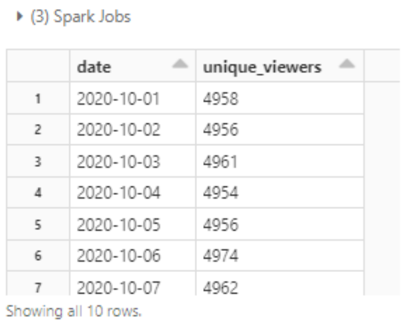


Date Format	End Date	Start Date
yyyy-M-d	2020-10-10	2020-10-01

Figure 5.24: Usefull widgets using *dbutils* for taking user input

Displaying Unique Viewers for each Date withing the given data range provided through the widgets

```
start_date = dbutils.widgets.get("w1")
end_date = dbutils.widgets.get("w2")
date_format = dbutils.widgets.get("w3")
unique_viewers_df = unique_viewers ( df,start_date,end_date,date_format)
display(unique_viewers_df)
```



	date	unique_viewers
1	2020-10-01	4958
2	2020-10-02	4956
3	2020-10-03	4961
4	2020-10-04	4954
5	2020-10-05	4956
6	2020-10-06	4974
7	2020-10-07	4962

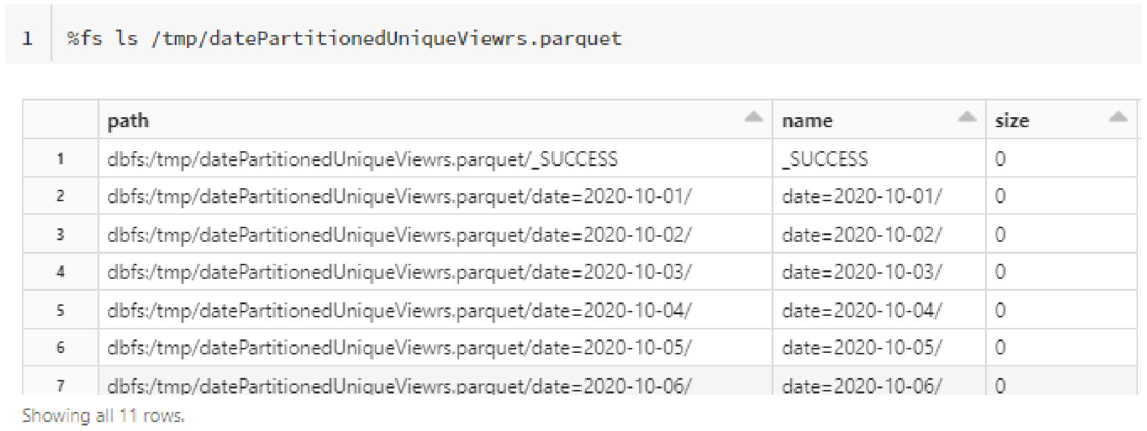
Showing all 10 rows.

Figure 5.25: Unique viewers for given date range

Partitioning the data for each date and save as a parquet file format

```
unique_viewers_df.write.partitionBy("date").mode("overwrite").parquet("/tmp/
    datePartitionedUniqueViewrs.parquet")
```

Displaying how the partitioned data is saved



```
%fs ls /tmp/datePartitionedUniqueViewrs.parquet
```

	path	name	size
1	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/_SUCCESS	_SUCCESS	0
2	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-01/	date=2020-10-01/	0
3	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-02/	date=2020-10-02/	0
4	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-03/	date=2020-10-03/	0
5	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-04/	date=2020-10-04/	0
6	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-05/	date=2020-10-05/	0
7	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-06/	date=2020-10-06/	0

Showing all 11 rows.

Figure 5.26: Displaying how the partitioned parquet files are stored using fs util

Loading Incremental Data containing information of 2020-10-11 and merging with the previous data

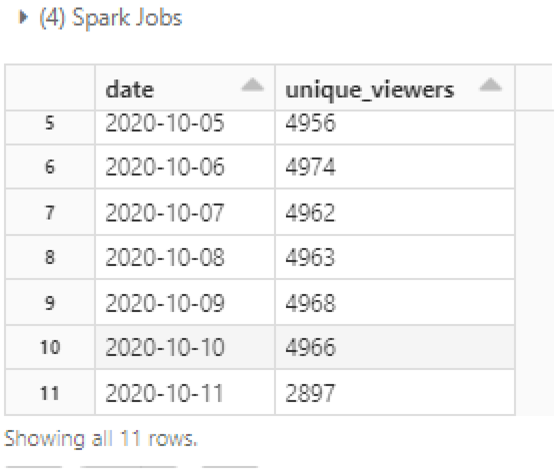
```
df_next = spark.read.csv("/FileStore/tables/mixpanel_incremental.csv",header=True,
    schema=schemaDF).withColumn("date",to_date(col("date"),"d/M/yyyy")).
    drop_duplicates(["distinct_id"]))

whole_df = df.union(df_next)
```

Displaying Unique Viewers for each Date withing the given data range provided through the widgets for merged dataframe

```
start_date = dbutils.widgets.get("w1")
end_date = dbutils.widgets.get("w2")
date_format = dbutils.widgets.get("w3")
unique_viewers_df = unique_viewers ( df,start_date,end_date,date_format)
display(unique_viewers_df)
```

Again Partitioning the data for each date and save as a parquet file format

```
whole_unique_viewers_df.write.partitionBy("date").mode("append").parquet("/tmp/
    datePartitionedUniqueViewrs.parquet")
```

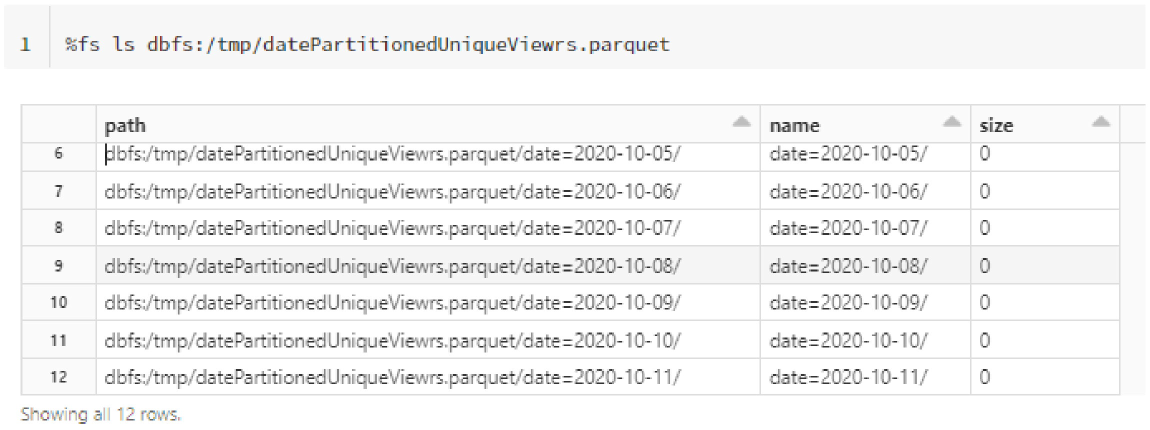

(4) Spark Jobs

	date	unique_viewers
5	2020-10-05	4956
6	2020-10-06	4974
7	2020-10-07	4962
8	2020-10-08	4963
9	2020-10-09	4968
10	2020-10-10	4966
11	2020-10-11	2897

Showing all 11 rows.

Figure 5.27: Unique viewers for merged data,notice the last row

Displaying how the partitioned data is saved



```
%fs ls dbfs:/tmp/datePartitionedUniqueViewrs.parquet
```

	path	name	size
6	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-05/	date=2020-10-05/	0
7	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-06/	date=2020-10-06/	0
8	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-07/	date=2020-10-07/	0
9	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-08/	date=2020-10-08/	0
10	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-09/	date=2020-10-09/	0
11	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-10/	date=2020-10-10/	0
12	dbfs:/tmp/datePartitionedUniqueViewrs.parquet/date=2020-10-11/	date=2020-10-11/	0

Showing all 12 rows.

Figure 5.28: Saved incrmental data in parquet format

More References

Apart from all the above examples, for more reference about Databricks[5], PysSark[8] kindly refer to the official documentations. I also took much help from the PySpark tutorial website SparkByExamples[11] for implementation.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In the era of Big Data, data engineering has become utmost important due to the requirement of managing large volume of data. But the big challenge for the data engineers is, they have to have a sound knowledge of a significant number of Big Data platforms, tools, and techniques, that exists and also the new technologies that will come in future. Gaining a good level of expertise on all these technologies is quite hard. But, in these so many challenges, Apache Spark has come out to be the rescuer of data engineers for its large unified ecosystem leveraging the high performant parallel processing. Spark, is the one ecosystem for various purposes like Data Engineering, Spark Streaming for real-time data analysis, Machine Learning, handling graph databases, and so much more under the same hood. That is why Spark is gaining popularity day by day and so as Data Bricks has the best support amongst all the three cloud providers (GCP, Azure, AWS) for data storage and retrieval. In this thesis, I have presented some data engineering works with PySpark, describing Spark's internal working briefly.

6.2 Future Work

Though initially Scala was used as the official language to work with Apache Spark but since PySpark, the Python APIs to deal with Apache Spark, has been released, PySpark has been evolved as the best suited language for Data Engineers and Data Scientists. However, as Python is a dynamically typed language that is why it fails to facilitate the use of DataSet, which is a newer data structure used in Spark 3.X.X giving the facility of intellisence. Consequently, the DataSet APIs are not available in PySpark, those are available in Scala APIs only. One can explore other facilities of Spark Ecosystems with GraphaX which provides a good set to APIs for processing graph data base. Machine Learning engineers and Data Scientist can use the MLLib which provides a good set of APIs to perform various Machine Learning techniques.

Spark can also be run on local machine, but Data Bricks has come out to be the best choice for leveraging the power of Spark's processing engine, mostly for Data Engineering kind of tasks. Almost all the spark features can be explored and tested freely through Data Bricks' community cloud and I have used Data Bricks with PySpark for the same reason. One can choose language of his choice among Scala (as spark is designed with Scala, so the creators of Data Bricks always keep the Scala APIs updated and all the new features would be available in Scala first), Python, R, Java.

Bibliography

[1] Acharjya, D. P., and P, K. A. A survey on big data analytics: Challenges, open research issues and tools. *International Journal of Advanced Computer Science and Applications 7*, 2 (2016).

[2] Parquet file format documentation. Available at https://parquet.apache.org/docs/.

[3] Aziz, K., Zaidouni, D., and Bellafkih, M. Leveraging resource management for efficient performance of apache spark. *Journal of Big Data 6*, 1 (Aug 2019), 78.

[4] Types of bigdata. Available at https://www.analyticssteps.com/blogs/what-big-data-analytics-definition-advantages-and-typesl.

[5] Databricks documentation. Available at https://docs.databricks.com/workspace-index.html.

[6] Elgendy, N., and Elragal, A. Big data analytics: A literature review paper. In *Advances in Data Mining. Applications and Theoretical Aspects* (Cham, 2014), P. Perner, Ed., Springer International Publishing, pp. 214–227.

[7] Tlc trip records user guide. Available at https://www1.nyc.gov/assets/tlc/downloads/pdf/trip_record_user_guide.pdf.

[8] Pyspark documentation. Available at https://spark.apache.org/docs/3.3.0/api/python/getting_started/index.html.

[9] Salloum, S., Dautov, R., Chen, X., Peng, P. X., and Huang, J. Z. Big data analytics on apache spark. *International Journal of Data Science and Analytics 1*, 3 (Nov 2016), 145–164.

[10] Shaikh, E., Mohiuddin, I., Alufaisan, Y., and Nahvi, I. Apache spark: A big data processing engine. In *2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)* (2019), pp. 1–6.

[11] Pyspark tutorial. Available at https://sparkbyexamples.com/pyspark-tutorial/.

[12] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. Apache spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (oct 2016), 56–65.