Analysing Yelp Dataset using Spark and Comparative Study with different methodologies

SanVinoth Pacham
Concordia Unviersity
40198906

Revanth Velagandula
Concordia University
40229629

Visnunathan Chidambaranathan
Concordia University
40230157

Sharanyu Pillai
Concordia University
40227794

Abstract

The Yelp data set[1] is a subset of Yelp's businesses, reviews, and user information. This dataset contains information about firms in eight metropolitan areas in the United States and Canada. This dataset contains five JSON files namely such as business, review, user, tip and checkin. As a result, the goal of this project is to use the spark distributed system to analyze various data insights across all business elements. We examine the distributed computing elements of spark in this study, followed by a discussion of the queried results utilizing the same data system and comparative study of spark programming with other distributed programming methodologies.

1 Introduction

Apache Spark is a data processing framework that can quickly analyze large datasets and distribute data processing tasks across multiple computers. Its fully managed framework offers resourceful, reliable, and user-friendly querying capabilities to extract valuable insights from data. In this project, Spark is used to evaluate the Yelp Data dataset with regard to various test cases. The report also covers Spark's distributed and parallel computing properties, as well as its architecture, features, and results. The conclusion and future prospects are also discussed.

2 System : Apache Spark Architecture

Apache Spark Architecture[9] is a framework-based open-source component used to handle huge amounts of unstructured, semi structured, and structured data for analytics for big data processing, Spark Architecture is considered an alternative to Hadoop and MapReduce architecture. For data storage and processing, spark architecture is coupled with Resilient Distributed Datasets (RDD)and Directed Acyclic Graph (DAG). It also contains four architecture components, including a spark driver, executors, cluster managers, and worker nodes. Spark's key data storage components are datasets and data frames, which aid in the optimization of the Spark process and large data computing. Apache Spark's architecture is made up of loosely linked components. Spark's architecture takes into account the master/worker process, and all tasks run on top of the Hadoop-distributed file system. Hadoop is used by Apache Spark for data processing and storage procedures. They are an in-memory data processing engine that allows their applications to execute on Hadoop clusters faster than a memory. In Memory processing avoids disc I/O from failing. Spark enables several jobs to operate with the same data. Spark divides data into divisions, the size of which is determined by the data source. The following are the two main Apache Spark Architecture implementations: 1. Resilient Distributed Datasets (RDD) It is responsible for providing API to handle caching and partitioning. It is an important set of tools for data computation. It aids in the re-computing of items in the event of a failure, is considered immutable data, and serves as an interface. RDD performs two operations: transformations and actions.

2. Directed Acyclic Graph (DAG) It connects one node to another in a sequence. For each job, the driver turns the program into DAG. API core, Spark SQL, Streaming and real-time processing, MLIB, and Graph X are all components of the Apache Spark Eco-system. Some terms to learn here are Spark shell, which aids in reading huge amounts of data, Spark context cancel, run a job, task (a work), job (computation)

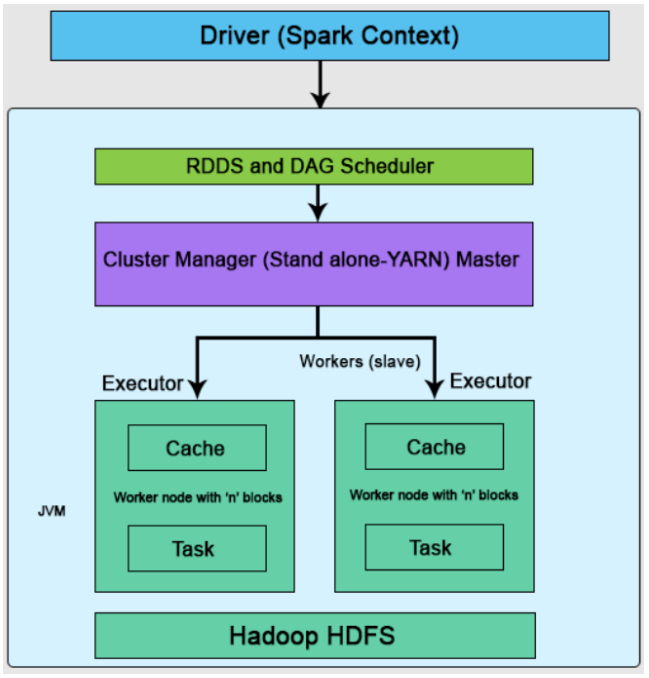


Figure 1: Spark Architecture

2.1 Components of Apache Spark Architecture

The Four main components of Spark[10] are given below and it is necessary to understand them for the complete framework. 1. Spark Driver 2. Executors 3. Cluster manager 4. Worker Nodes

2.1.1 Spark driver It supports the management of clusters with a single master and a number of slaves. There are two types of cluster managers: YARN and standalone, which are both handled by the Resource Manager and the Node. Standalone cluster work necessitates the roles of Spark Master and worker node. The cluster manager is in charge of allocating resources and carrying out tasks. The driver is responsible for coordinating duties and workers for management. It is an Application JVM process and is a master node. A driver divides the spark into tasks and schedules them to run on cluster executors. The driver programs in the diagram invoke the main application and generate a spark context (which functions as a gateway), monitor the job operating within the provided cluster, and connect to a Spark cluster. The spark context is used for all functionalities and instructions. Each session has its own Spark context. To run jobs in clusters, the Spark driver has more components. Spark clusters are linked to various types of cluster managers, and the context obtains worker nodes to execute and store data. . In the cluster, when we execute the process their job is subdivided into stages with gain stages into scheduled tasks.

2.1.2 Executor It is responsible for the job execution and data storage in a cache. Executors register with the drivers from the very beginning. This executor has a set of time slots available for running the application concurrently.

Executors carry out read/write operations on external sources. When the executor has loaded data, it runs the job, and the data is removed in idle mode. Dynamic allocation enables the executor, and they are constantly added and excluded depending on the duration. A driver software monitors the executors while they are performing their duties. In a Java process, executors carry out the tasks of the users.

2.1.3 Cluster manager It supports the management of clusters with a single master and a number of slaves. There are two types of cluster managers: YARN and standalone, which are both handled by the Resource Manager and the Node. Standalone cluster work necessitates the roles of Spark Master and worker node. The cluster manager is in charge of allocating resources and carrying out tasks.

2.1.4 Worker node They are the slave nodes; their primary function is to conduct tasks and deliver the results to the spark context. They convey the availability of resources to the master node. The Spark content runs it and sends it to the worker nodes. One spark worker is allocated to each worker node for monitoring. They simplify the computation by increasing the number of worker nodes (1 to n) such that all tasks are completed in parallel by partitioning the job over different computers. The other element task is regarded as a unit of work and is assigned to one executor; spark conducts one task for each partition.

3 Features

3.1 Fault Tolerance

Fault tolerance[7] in distributed systems refers to the ability of the system to maintain functionality and achieve its goals even when some of its components fail or experience problems. This can be achieved through various strategies such as replication, backup components, and error detection and correction. Ensuring fault tolerance is important in distributed systems because it helps to ensure the reliability and availability of the system, especially in mission-critical or safety-critical applications.

Apache Spark is a distributed data processing platform that is designed to be fault-tolerant, meaning it can recover from failures and continue processing data in the event of node failures, network issues, or other disruptions. Spark achieves fault tolerance through data replication and the ability to re-execute failed tasks. Data is replicated across multiple nodes in the cluster to ensure it is still accessible in the event of a node failure, and the driver system tracks task progress and coordinates recovery in the event of a failure by re-executing failed tasks. These mechanisms help to ensure the reliability and availability of Spark for mission-critical applications.

3.2 Data Partitioning

Data partitioning is a technique used to divide a large dataset into smaller, more manageable pieces called partitions that can be processed in parallel by different nodes in a distributed system. It can improve the scalability and performance of the system, as well as its fault tolerance, by allowing the nodes to continue functioning even if one or more nodes fail. Data partitioning can be implemented using different methods, such as dividing the data based on a key or using a hash function to evenly distribute it across the nodes. In Apache Spark, data partitioning refers to the process of dividing a large dataset into smaller chunks, called partitions, which can be processed independently in parallel. This is a key feature of Spark that enables it to scale out and perform distributed processing on a large number of machines. Data partitioning is a crucial part of the Spark architecture, as it allows the system to distribute the data and computation across multiple nodes in a cluster, thereby improving the performance and efficiency of the system. Data partitioning can be done in various ways, such as by hash, range, or custom logic, depending on the requirements of the application.

3.3 Lazy Evaluation

Lazy evaluation is a programming technique that delays the evaluation of an expression until it is needed, which can improve the performance of a program by avoiding unnecessary computation. It is commonly used in functional programming languages and can be implemented using various methods, such as lazy data structures, thunks, or call-by-need evaluation. In Apache Spark, lazy evaluation refers to the delay of data processing until it is actually needed. This means that transformations on data are not executed until an action is called, at which point the data is materialized and the transformations are applied. Lazy evaluation is a key feature of Spark that allows it to optimize the execution of data pipelines, minimize the amount of data that needs to be shuffled between nodes in the cluster, and build a directed acyclic graph (DAG) of the data transformations for more efficient execution. Overall, lazy evaluation helps Spark scale out and perform distributed processing on large datasets efficiently.

3.4 Persistence

Persistence refers to the ability of a distributed system to store and manage data over a long period of time, even when the system is inactive or offline. This is important for maintaining data integrity and availability, as well as for recovering from failures. Apache Spark is a distributed computing platform that allows for the persistence of data across multiple jobs or stages, enabling the reuse of previously computed data rather than recomputing it. This is achieved through the use of caching and storage methods such as cache(), persist(), and save(). Persistence can improve the performance of Spark applications by reducing the need to read from slower storage systems and providing faster access to data, but it is important to consider the available resources when using these features.

3.5 Parallelism

Parallelism in distributed data systems allows multiple tasks to be performed concurrently on multiple compute nodes, improving performance and scalability. There are three types of parallelism: data parallelism involves dividing data into smaller chunks and distributing them among nodes, task parallelism involves dividing a large task into smaller sub-tasks that can be processed by different nodes, and pipeline parallelism involves breaking down a task into smaller tasks that can be processed in parallel with the output of one serving as the input to the next. Spark supports both data parallelism and task parallelism. In data parallelism, Spark divides the data in an RDD into smaller chunks called partitions and assigns them to different nodes in the cluster for processing in parallel. In task parallelism, Spark divides a larger task into smaller sub-tasks, which can be processed in parallel by different nodes in the cluster.

3.6 Data Lineage

Data lineage refers to the history and movement of data within a distributed system, including its origins, transformations, and destinations. This information is useful for understanding the provenance of data and can be helpful for debugging, compliance, and data governance. Tracking data lineage in a distributed system can be challenging due to the complexity of data movement and transformation between different nodes and systems. In Apache Spark, data lineage can be traced through features such as the DAG execution model, structured streaming, and the lineage API, which allow developers to understand how data has been transformed and where it came from. Data lineage is an important aspect of data management in Spark and is useful for a variety of purposes.

4 Demo Scenario - Methodologies

4.1 Databricks and Azure for data Analytics

4.1.1 Azure Account creation Open The Azure portal[2] and open opt for a free subscription. Log in with your Microsoft account-outlook.com, live.com, hotmail.com, or any email associated with Microsoft. Provide your details and open the standard subscription which is free for a limited time.

4.1.2 Setting up storage account and loading data Once the Azure standard subscription is enabled, Open the resource group section in the Azure portal[3] and create a resource by giving it a name. Assign the resource to a nearby region (such as East US). Once the resource group is created, Open the storage accounts section and create a storage account and assign the resource group to the storage account. Loading Data in Azure can be accomplished in multiple ways. We can either use the upload option in the Azure portal or the Azure storage explorer or an AzCopy command from your local system to azure blob using a SAS token.

4.1.3 Creating a data bricks workspace Open data bricks[4] in Azure Portal and open data bricks dashboard. Choose the create Create workspace and provide the necessary details. Once the deployment is completed, Open the workspace by clicking launch workspace.

4.1.4 Authentication between azure BLOB and databricks To connect data bricks with Azure BLOB we need to create a SAS (shared access signatures) token in the storage account and use the data bricks mount function to connect with the azure BLOB storage.

4.1.5 Developing PySpark usecases Databricks allows us to create IPython notebooks with an extension of .dbc which is the native format for data bricks. It supports Python, Scala, SQL and R. We can create PySpark queries using data bricks as it is built on top of Apache spark. Databricks standard subscription provides a Single node cluster to execute the queries. We can execute the created use cases and save them as delta tables which are built on top of parquet files. This table can be queried to create reports and visualizations.

4.1.6 Visualisation using Tableau Once The use case tables are generated. We can use the data bricks partner connect option to download the necessary reporting tool's ODBC connector to establish the connection. For authentication, it takes in the server hostname, HTTP path, OAuth / Azure AD connection, and OAuth endpoint. For data, the query utilizes the data bricks single-node cluster. Once the connection has been established we can create meaningful insights and dashboards.

4.2 Multinode cluster setup

4.2.1 Spark installation In each of the systems, Download and Extract the archive spark bundle. Save it under any specific folder. Note the path where you have saved the folder. Make sure to have JDK 8 or 11 (Java Development kit) installed in all the systems.

4.2.2 Setting up SPARK_HOME location Set the *SPARK_HOME* environment variable on each system to point to the directory where you installed Spark in the $bash\_profile$ and export it to the PATH variable.

4.2.3 Network settings Configure the firewall settings in all computers involved. The default port for the Spark master is 7077, and the default port for the Spark worker is 8888.

4.2.4 Connecting the workers Start the master node and start the worker nodes by using the master nodes IP address and 7077 port.

4.2.5 Submitting jobs Once the workers are connected and a cluster is formed. We can submit spark jobs using the spark-submit command and mention the python file which we want to execute.

4.2.6 Analyze the SparkUI and logs Open the master node URL and analyze the job execution and logs for the output and results.

4.3 Comparative analysis

Spark execution is compared with different sequential and parallel programming techniques like MPI(Message passing interface), Multi-processing, Multi-threading, and Serial execution. Scripts have been developed to compare based on the use cases to prove the power of spark programming with its counterparts.

5 Results and Conclusion

5.1 Usecase Dashboard

Reports have been created using the curated data coming from delta tables created in Databricks. Tableau tool has been used to create dashboard and present the different insights we gathered from data.

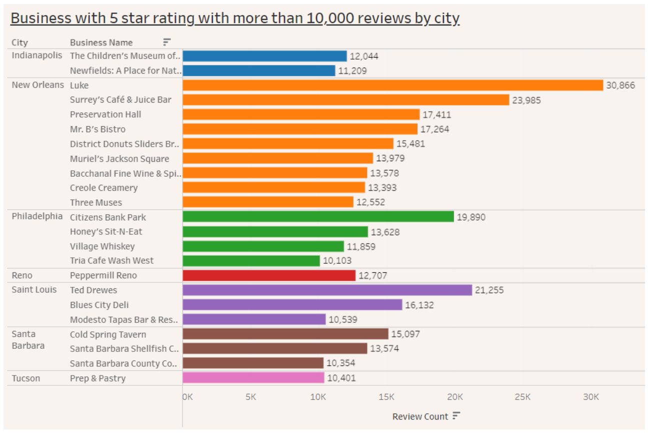


Figure 2: Business with 5 Star rating with more than 10000 reviews

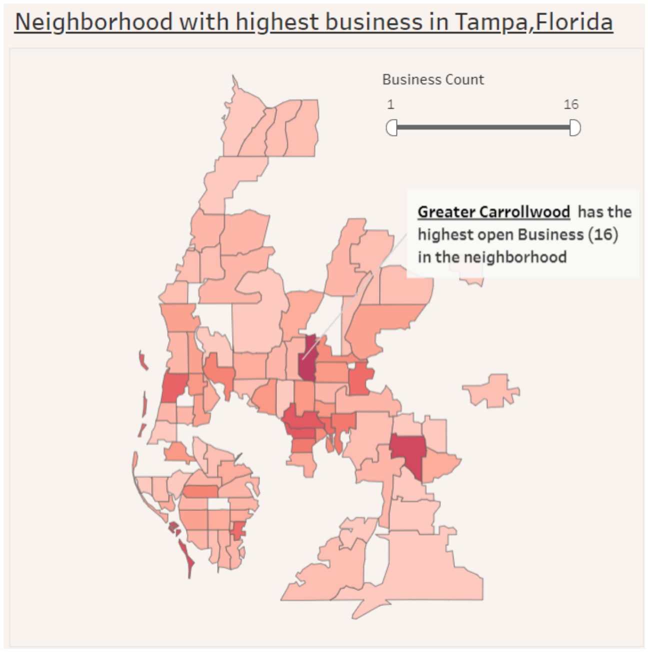
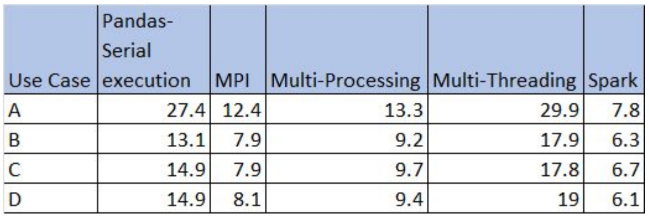

Neighborhood with highest business in Tampa,Florida
Business Count
1
16
Greater Carrollwood has the highest open Business (16) in the neighborhood

Figure 3: Neighbourhood with the highest business in Tampa, Florida

5.2 Comparative Analysis Stats



Use Case	Pandas-Serial execution	MPI	Multi-Processing	Multi-Threading	Spark
A	27.4	12.4	13.3	29.9	7.8
B	13.1	7.9	9.2	17.9	6.3
C	14.9	7.9	9.7	17.8	6.7
D	14.9	8.1	9.4	19	6.1

Figure 4: Comparative Analysis. Use cases vs time in seconds

From the figure 4 we can infer that spark programming is much efficient than other programming methodologies when it comes to distributed computing. MPI and Multiprocessing performed in almost same manner. MPI is generally faster than the multiprocessing module in Python because it is implemented in C and is designed specifically for use in parallel computing environments. It provides a wide range of features for efficiently communicating between processes. Multithreading performed poorly than rest of the methods. Mutlithreading is better choice for CPU bound executions.

6 References

[1]https://www.kaggle.com/datasets/yelp-dataset/yelp-dataset
[2]https://azure.microsoft.com/en-us/get-started/explore-azure

[3]https://learn.microsoft.com/en-us/azure/storage/blobs/

[4]https://learn.microsoft.com/en-us/azure/databricks/

[5]https://spark.apache.org/docs/latest/rdd-programming-guide.html

[6]https://data-flair.training/blogs/apache-spark-lazy-evaluation/

[7]https://data-flair.training/blogs/fault-tolerance-in-apache-spark/

[8]https://www.tableau.com/why-tableau/what-is-tableau

[9]https://www.interviewbit.com/blog/apache-spark-architecture/

[10]https://www.javatpoint.com/apache-spark-components

[11]https://paulx-cn.github.io/blog/6th_Blog/

[12]https://learn.microsoft.com/en-us/azure/?product=popular

[13]https://blog.knoldus.com/understanding-persistence-in-spark/

[14]https://www.geeksforgeeks.org/what-is-dfsdistributed-file-system/