Analyzing Large Scale Transactional Data: A Distributed FP-Growth Implementation on Spark

Madhan.S-22BDS036, Bharath L-22BDS013, Gnanesh-22BDS023, Gopal-22BDS025, Suhaas-22BDS056

Abstract-This project, we investigate the utilization of Hadoop and Spark, two leading technologies in the realm of Big Data processing, for executing the FP-Growth algorithm on a distributed system. With a dataset comprising 79 lakhs entries, we distribute the data across multiple nodes using Hadoop's HDFS and employ Spark for parallel processing. The FP-Growth algorithm, renowned for its efficiency in mining frequent itemsets from transactional databases, serves as the focal point of our study. By varying the number of nodes in the cluster, we examine the algorithm's performance and analyze the impact of scalability on execution time. Through this endeavor, we aim to explore the efficacy of distributed systems in handling large-scale data processing tasks and provide insights into optimizing performance for similar data analysis endeavors.

I. INTRODUCTION

The ever-growing volume, variety, and velocity of data pose a challenge for traditional data processing techniques. This data has given rise to the concept of Big Data. Big Data refers to datasets that are so large and complex that traditional data processing applications are inadequate to handle them efficiently. This necessitates the use of specialized tools and technologies capable of processing and analyzing massive amounts of data.

Two prominent technologies in the Big Data landscape are Hadoop and Spark. Hadoop is an open-source framework designed for distributed storage and processing of large datasets across clusters of commodity hardware. Hadoop's core strength lies in its Distributed File System (HDFS), which fragments and stores data across multiple nodes, ensuring scalability and fault tolerance. Additionally, Hadoop's MapReduce programming model breaks down complex tasks into smaller, manageable units that can be executed in parallel across the cluster, significantly accelerating processing.

While Hadoop excels at large-scale batch processing, there's a growing need for real-time analytics and interactive exploration of data. This is where Apache Spark steps in. Spark is another open-source framework built on top of Hadoop that offers lightning-fast processing capabilities. Spark's in-memory processing engine allows for significantly faster computations compared to traditional disk-based approaches in Hadoop. Moreover, Spark's versatility extends beyond batch processing, supporting real-time data streams, machine learning algorithms, and interactive data exploration.

By leveraging the scalability and parallel processing capabilities of Hadoop and Spark, organizations can efficiently process, analyze, and derive insights from large volumes of data. In this report, we explore the application of these technologies in executing the FP-Growth algorithm, a popular method for mining frequent item sets in transactional databases, on a distributed system. We evaluate the performance of the algorithm across different numbers of nodes in the cluster and analyze the impact of scalability on execution time.

II. METHODOLOGY

A. Distributing data over HDFS

We utilized Hadoop Distributed File System (HDFS) as the foundation for distributing and storing our large-scale dataset across multiple nodes in the Hadoop cluster. HDFS is a distributed file system designed to run on commodity hardware. It follows a master-slave architecture, consisting of the following two key components:

1) **NameNode:** The NameNode is the master node in the HDFS cluster and manages the metadata for all the files and directories stored in the file system. It keeps track of the file system namespace, file permissions, and the mapping of blocks to DataNodes i.e. tracks the location of all data blocks across the cluster and directs clients on how to access them.
2) **DataNode:** DataNodes are the slave or worker nodes in the HDFS cluster and are responsible for storing and managing the actual data blocks. They receive instructions from the NameNode regarding data storage and retrieval operations.

B. Data storage in HDFS:

1) **Data splitting:** HDFS breaks down the large dataset into smaller manageable chunks called data blocks. By default, HDFS uses a block size of 128 MB, but this can be configured based on the specific requirements of the dataset.
2) **Replication:** For fault tolerance, HDFS replicates each data block across multiple DataNodes in the cluster. The number of replicas can be configured but typically defaults to 3. This ensures that even if one DataNode fails, there are copies of the data block available on other nodes, preventing data loss.
3) **Data distribution:** Once the data blocks are split and replicated, HDFS distributes them across the available DataNodes in the cluster. The NameNode maintains the mapping of each file to its corresponding data blocks and their locations on the DataNodes.
4) **Block Management:** The NameNode meticulously tracks the locations of all data blocks and the corresponding DataNodes that store them. It maintains a map of the entire filesystem.
5) **Fault Tolerance:** HDFS ensures fault tolerance by automatically detecting and recovering from DataNode failures. If a DataNode becomes unavailable or fails, the NameNode redirects data requests to other replicas of the affected data blocks stored on different DataNodes.

By distributing our dataset over HDFS, we were able to leverage the fault tolerance, scalability, and parallel processing capabilities of Hadoop for efficient data storage and retrieval in our spark.

C. FP Growth algorithm

In the realm of big data analysis, frequent itemset mining is a crucial technique for uncovering frequently occurring item combinations within a dataset. Imagine a grocery store analyzing customer purchase patterns. FP-Growth (Frequent Pattern Growth) is a popular algorithm used for mining frequent itemsets in transactional databases. It efficiently discovers frequent patterns by recursively building a compact data structure called the FP-tree (Frequent Pattern tree) and then mining this tree to extract frequent itemsets.

1) Algorithm overview:

1) **Minimum Support Threshold:** You define a minimum support threshold, which represents the minimum number of times an itemset needs to appear in transactions to be considered frequent. This threshold helps filter out insignificant or rare item combinations.
2) **Frequent Item Identification:** FP-growth first identifies frequent single items based on their support count within the transactions. These items are then sorted in descending order of frequency.
3) **FP-Tree Construction:** The algorithm constructs a tree structure called the FP-tree. Each transaction is represented as a path in the tree, with frequent items appearing closer to the

root. The support count for each item is also stored along the path.

4) **Conditional Pattern Bases:** For each frequent item (except the most frequent one), a conditional pattern base is created. This base is essentially a subset of the original transactions containing only those transactions that include the specific frequent item.
5) **Recursive Mining:** The algorithm recursively mines frequent itemsets from these conditional pattern bases. It constructs smaller FP-trees for each base and repeats steps 2-4 until no more frequent itemsets are found.

2) Algorithm: FP-Growth Algorithm

function FP-GROWTH($Tree$, $HeaderTable$, $minSupport$, $prefix$)
  if $Tree$ is a single path tree **then**
    for all $prefix$ pattern in $Tree$ **do**
      output ($prefix$ pattern)
    end for
    return
  end if
  for all $item$ in $HeaderTable$ in ascending order of frequency **do**
    $pattern \leftarrow prefix + item$
    output ($pattern$)
    $conditional\_pattern\_base \leftarrow createConditionalPatternBase(HeaderTable[item])$
    $conditional\_tree \leftarrow createFP-Tree(conditional\_pattern\_base, minSupport)$
    if $conditional\_tree$ is not empty **then**
      FP-GROWTH($conditional\_tree$, $HeaderTable$, $minSupport$, $pattern$)
    end if
  end for
end function

FP-Growth is known for its efficiency in handling large-scale datasets and has become a cornerstone in association rule mining and market basket analysis due to its effectiveness in discovering frequent itemsets.

D. Job Execution in Spark

Spark is a distributed computing framework that provides an efficient and versatile platform for processing big-data workloads across clusters of computers. It offers several features and optimizations to maximize performance and scalability:

1) Spark Execution Model:

1) **Driver Program:** This program runs on the main node and coordinates the entire Spark application. It interacts with the SparkContext to submit jobs, manage resources, and schedule tasks on the cluster.
2) **SparkContext:** This is the central entry point for Spark applications. It connects to the cluster manager (YARN or standalone) to request resources and launch executors on worker nodes.
3) **Executors:** These are processes running on worker nodes in the cluster that are responsible for executing Spark tasks. Executors receive tasks from the SparkContext, fetch the required data, and perform the computations in parallel.

2) Spark Job Execution:

1) **Job Submission:** The driver program submits a Spark job through the SparkContext.
2) **Logical Plan Creation:** The Spark job defines a series of operations on a distributed dataset (RDD - Resilient Distributed Dataset). These operations create a logical plan representing the sequence of computations.
3) **Physical plan Optimization:** Spark analyses the logical plan and optimizes it into a physical plan that efficiently utilizes the cluster resources. This may involve optimizations like data shuffling between executors to perform specific operations.
4) **Task Breakdown:** The physical plan is broken down into smaller units of work called tasks. Each task operates on a specific partition of the data.
5) **Task Scheduling:** The SparkContext schedules these tasks onto available executors in the cluster. Spark considers factors like data locality (scheduling tasks closer to data for faster access) and resource availability for efficient scheduling.
6) **Task Execution:** Executors receive assigned tasks, fetch the required data, and execute the

operations defined in the physical plan.

III. Result

The analysis of our project revealed an unexpected trend: as the number of nodes increased, the execution time of the FP-Growth algorithm also increased instead of decreasing as anticipated. This counterintuitive result can be attributed to network overhead, which significantly impacted the performance of our distributed system.

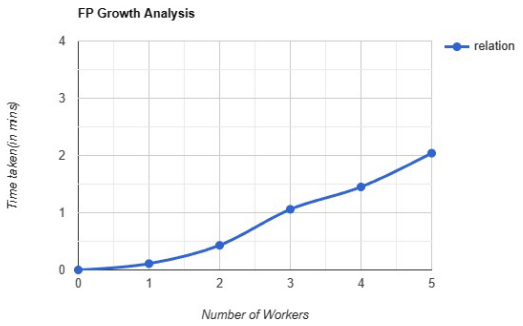
FP Growth Result

Worker(s)	Time taken (in mins)
1	3.23
2	7.66
3	9.89
4	12.07
5	14.36

A. Network overhead

Network overhead refers to the additional time, resources, and bandwidth consumed by communication between nodes in a distributed system. As we scale our system by adding more nodes, the amount of data transmission and coordination between nodes increases, leading to higher network overhead. This overhead can manifest in several ways:

1) **Serialization and Deserialization:** Data needs to be converted into a format suitable for transmission (serialization) and then back into its original form (deserialization) at the receiving node. This process takes time and adds to the overall overhead.
2) **Network Latency:** The physical limitations of the network introduce latency, which is the time it takes for data to travel between nodes. Factors like distance, network congestion, and hardware limitations can all impact latency.
3) **Data Shuffling:** In distributed processing, data might need to be shuffled between nodes for specific operations. For example, in FP-growth, conditional pattern base creation might involve sending relevant transactions to specific executors. This shuffling adds to the network overhead.
4) **Protocol Overhead:** Network protocols add additional control information to data packets for routing and error handling. This extra information consumes bandwidth and contributes to the overall overhead.



Network overhead becomes a significant factor as the number of nodes increases in a distributed system. With more nodes, the amount of data that needs to be transferred across the network also increases. This can lead to a situation where the time spent on data transfer outweighs the benefits of parallel processing.

IV. Conclusion

Our exploration of executing the FP-Growth algorithm on a distributed system using Hadoop and Spark has provided valuable insights into the challenges and opportunities of large-scale data processing. Through our methodology of distributing the dataset across multiple nodes and analyzing the execution time across different numbers of nodes, we have gained a deeper understanding of the impact of scalability and network overhead on algorithm performance.

Despite our initial expectation that increasing the number of nodes would lead to decreased execution time, our analysis revealed a contrary trend, highlighting the significance of network overhead in distributed systems. The observed increase in execution time underscores the importance of considering not only computational parallelism but also

network communication overhead when designing and optimizing distributed algorithms.

Our study emphasizes the need for holistic approaches to distributed computing, incorporating considerations of both computational and communication aspects. Strategies to mitigate network overhead, such as network topology optimization, data locality-aware scheduling, and efficient communication protocols, are essential for maximizing the performance and scalability of distributed systems.

In conclusion, our project highlights the complexities inherent in distributed computing environments and underscores the importance of holistic optimization strategies to achieve optimal performance. By leveraging the capabilities of Hadoop and Spark while addressing network overhead challenges, we can unlock the full potential of distributed systems for Big Data analytics and beyond.

REFERENCES

[1] C. Borgelt, "An implementation of the fp-growth algorithm," in *Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations*, ser. OSDM '05, Chicago, Illinois: Association for Computing Machinery, 2005, pp. 1–5, ISBN: 1595932100. DOI: 10.1145/1133905.1133907. [Online]. Available: https://doi.org/10.1145/1133905.1133907.

[2] X. Meng, J. Bradley, B. Yavuz, *et al.*, "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016. [Online]. Available: http://jmlr.org/papers/v17/15-237.html.

[3] J. G. Shanahan and L. Dai, "Large scale distributed data science using apache spark," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15, Sydney, NSW, Australia: Association for Computing Machinery, 2015, pp. 2323–2324, ISBN: 9781450336642. DOI: 10.1145/2783258.2789993. [Online]. Available: https://doi.org/10.1145/2783258.2789993.

[4] M. Zaharia, R. S. Xin, P. Wendell, *et al.*, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016, ISSN: 0001-0782. DOI: 10.1145/2934664. [Online]. Available: https://doi.org/10.1145/2934664.