COMP1702: BIG DATA

Course Leader: Hai Huang

DATE: 08-04-2024

NAME: NAVED SALIM

ID: 001364756

Table of Contents

TASK A: Hive Data Warehouse Design

In this task, we design a data warehouse in Hive using original data which contains 3 tables and at least 50 records. Subsequently, we implement 10 different queries using adequate variety and complexity on the created data.

Data Warehouse Design:

We create a database based on an e-commerce platform with 3 tables:

- Customers
- Orders
- Products

The data in each table is created in .csv files, which are then uploaded to the Hadoop distributed file system (HDFS). Hence, we create 3 .csv files as shown below:

- Customer.csv
- Order.csv
- Product.csv

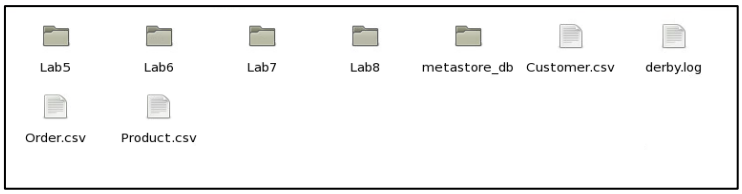


Fig 1: The 3 .csv files created for E-commerce database.

Subsequently, we log into the Terminal and launch Hadoop:

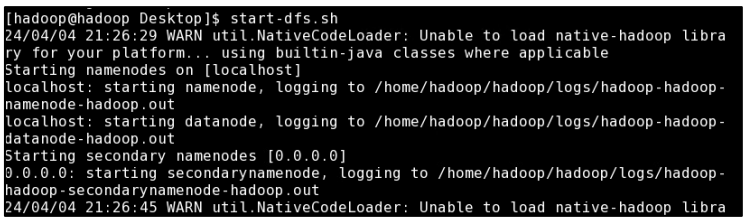


Fig 2: Screenshot launching Hadoop in Terminal.

The next step involves uploading the three .csv files we created into the HDFS. To begin, we navigate to the directory where the .csv files are stored (in this case Datasets folder):

```
[hadoop@hadoop Desktop]$ pwd
/home/hadoop/Desktop
[hadoop@hadoop Desktop]$ cd Datasets
[hadoop@hadoop Datasets]$ pwd
/home/hadoop/Desktop/Datasets
```

```
[hadoop@hadoop Datasets]$ ls
Customer.csv  Lab5  Lab6  Lab7  Lab8  Order.csv  Product.csv
```

Fig 3: Navigating to the directory for uploading the .csv files.

The next step involves uploading the files into HDFS as shown below:

```
[hadoop@hadoop Datasets]$ hdfs dfs -put Customer.csv
24/04/04 21:34:30 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using
 builtin-java classes where applicable
```

```
[hadoop@hadoop Datasets]$ hdfs dfs -put Order.csv
^[[A24/04/04 21:34:41 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... u
sing builtin-java classes where applicable
```

```
[hadoop@hadoop Datasets]$ hdfs dfs -put Product.csv
24/04/04 21:34:50 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using
 builtin-java classes where applicable
```

Fig 4: Uploading the .csv files into HDFS.

The three .csv files have been uploaded into the HDFS:

```
[hadoop@hadoop Datasets]$ hdfs dfs -ls
24/04/04 21:36:02 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using
 builtin-java classes where applicable
Found 10 items
-rw-r--r--   1 hadoop supergroup    1428841 2024-01-29 17:08 5000.txt
-rw-r--r--   1 hadoop supergroup        730 2024-04-04 21:34 Customer.csv
drwxr-xr-x   - hadoop supergroup          0 2024-02-19 16:36 Lab99
-rw-r--r--   1 hadoop supergroup        503 2024-04-04 21:34 Order.csv
-rw-r--r--   1 hadoop supergroup        301 2024-04-04 21:34 Product.csv
drwxr-xr-x   - hadoop supergroup          0 2024-01-29 17:10 Results
-rw-r--r--   1 hadoop supergroup       1854 2024-02-05 16:43 WordCount.java
drwxr-xr-x   - hadoop supergroup          0 2014-07-17 10:43 id.out
-rw-r--r--   1 hadoop supergroup         56 2024-02-05 17:33 test.txt
drwxr-xr-x   - hadoop supergroup          0 2024-02-12 16:25 wcoutput
```

Fig 5: Confirming the csv files have been uploaded into HDFS.

We use the ‘hive’ command to log into the hive environment:

```
[hadoop@hadoop Desktop]$ hive

Logging initialized using configuration in jar:file:/home/hadoop/hadoop/hive/lib
/hive-common-0.13.1.jar!/hive-log4j.properties
```

Fig 6: Logging into the Hive environment.

The next step involves the creation of three tables to load data from the .csv files into our database as shown below:

```
hive> create table Customers (Customer_ID INT, name STRING, email STRING, age IN
T) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;
OK
```

```
hive> create table Orders(Order_ID INT, Customer_ID INT, Order_Date DATE, Total_
Amount FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;
OK
```

```
hive> CREATE TABLE products ( product_id INT, name STRING, price FLOAT, category
 STRING ) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;
OK
```

Fig 7: Creation of 3 tables for e-commerce database.

After creating the tables, we load the data from .csv files into the tables. This is done using the command line query as shown below:

```
hive> LOAD DATA LOCAL INPATH 'Customer.csv' OVERWRITE INTO TABLE Customers;
Copying data from file:/home/hadoop/Desktop/Customer.csv
Copying file: file:/home/hadoop/Desktop/Customer.csv
Loading data to table default.customers
rmr: DEPRECATED: Please use 'rm -r' instead.
Deleted hdfs://localhost:9000/user/hive/warehouse/customers
Table default.customers stats: [numFiles=1, numRows=0, totalSize=730, rawDataSiz
e=0]
OK
Time taken: 0.838 seconds
```

Fig 8: Copying data from customer.csv to Customers table.

```
hive> LOAD DATA LOCAL INPATH 'Order.csv' OVERWRITE INTO TABLE Orders;
Copying data from file:/home/hadoop/Desktop/Order.csv
Copying file: file:/home/hadoop/Desktop/Order.csv
Loading data to table default.orders
rmr: DEPRECATED: Please use 'rm -r' instead.
Deleted hdfs://localhost:9000/user/hive/warehouse/orders
Table default.orders stats: [numFiles=1, numRows=0, totalSize=503, rawDataSize=0
]
OK
Time taken: 0.405 seconds
```

Fig 9: Copying data from Order.csv to Orders table.

```
hive> LOAD DATA LOCAL INPATH 'Product.csv' OVERWRITE INTO TABLE products;
Copying data from file:/home/hadoop/Desktop/Product.csv
Copying file: file:/home/hadoop/Desktop/Product.csv
Loading data to table default.products
rmr: DEPRECATED: Please use 'rm -r' instead.
Deleted hdfs://localhost:9000/user/hive/warehouse/products
Table default.products stats: [numFiles=1, numRows=0, totalSize=301, rawDataSize
=0]
OK
Time taken: 0.326 seconds
```

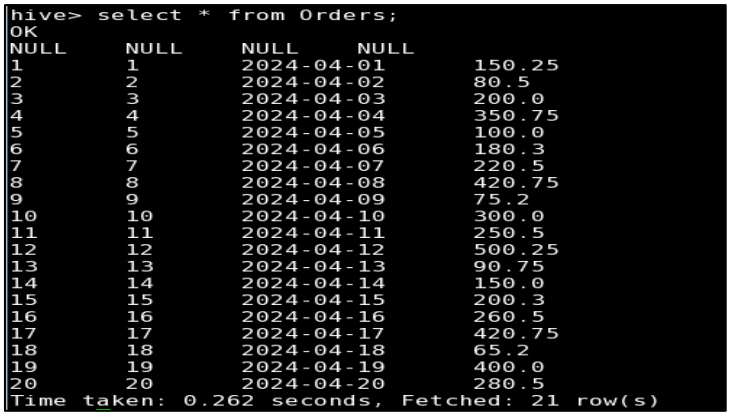
Fig 10: Copying data from Products.csv to the products table.

The tables have now been loaded with the data from their respective .csv files.

We use the ‘select’ query to view all the tables and data as depicted:

```
hive> select * from Customers;
OK
NULL    Name    Email   NULL
1       Alice Smith     alice@example.com       30
2       Bob Johnson     bob@example.com 25
3       Charlie Brown   charlie@example.com     40
4       Diana Ross      diana@example.com       35
5       Eve Johnson     eve@example.com 28
6       Frank White     frank@example.com       45
7       Gina Davis      gina@example.com        32
8       Henry Ford      henry@example.com       50
9       Ivy Lee ivy@example.com 27
10      Jack Black      jack@example.com        38
11      Kate Winslet    kate@example.com        33
12      Luke Skywalker  luke@example.com        60
13      Mary Johnson    mary@example.com        22
14      Nick Carter     nick@example.com        29
15      Olivia Newton   olivia@example.com      42
16      Peter Parker    peter@example.com       31
17      Quincy Jones    quincy@example.com      55
18      Rachel Green    rachel@example.com      26
19      Sam Adams       sam@example.com 48
20      Tina Turner     tina@example.com        36
Time taken: 0.061 seconds, Fetched: 21 row(s)
```

Fig 11: Screenshot displaying the Customers table.



```
hive> select * from Orders;
OK
NULL    NULL    NULL    NULL
1       1       2024-04-01      150.25
2       2       2024-04-02      80.5
3       3       2024-04-03      200.0
4       4       2024-04-04      350.75
5       5       2024-04-05      100.0
6       6       2024-04-06      180.3
7       7       2024-04-07      220.5
8       8       2024-04-08      420.75
9       9       2024-04-09      75.2
10      10      2024-04-10      300.0
11      11      2024-04-11      250.5
12      12      2024-04-12      500.25
13      13      2024-04-13      90.75
14      14      2024-04-14      150.0
15      15      2024-04-15      200.3
16      16      2024-04-16      260.5
17      17      2024-04-17      420.75
18      18      2024-04-18      65.2
19      19      2024-04-19      400.0
20      20      2024-04-20      280.5
Time taken: 0.262 seconds, Fetched: 21 row(s)
```

Fig 12: Screenshot displaying the Orders table.

```
hive> select * from products;
OK
NULL    Name    NULL    Category
1       Laptop  800.0   Electronics
2       Headphones      50.0    Electronics
3       Smartphone      600.0   Electronics
4       Backpack        40.0    Accessories
5       T-Shirt 20.0    Clothing
6       Jeans   50.0    Clothing
7       Watch   100.0   Accessories
8       Sneakers        80.0    Shoes
9       Perfume 30.0    Beauty
10      Skincare Set    70.0    Beauty
Time taken: 1.301 seconds, Fetched: 11 row(s)
```

Fig 13: Screenshot displaying the products table.

List of queries implemented:

1. List of all customers along with their orders:

The first query implements a list of all customers along with their orders. We use the '**SELECT'** query for this operation to retrieve data from the 'Customers' table and perform a '**JOIN'** operation with corresponding data from the 'Orders' table.

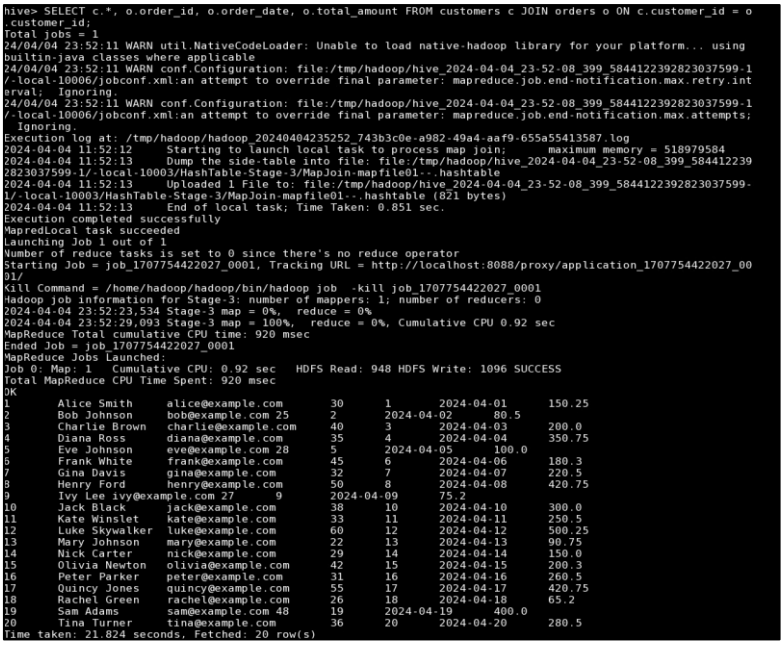


Fig 14: Screenshot displaying query and output.

2. Query to calculate the total sales amount

The second query calculates the total sales amount. We perform this operation using the '**SUM'** command which performs an addition of the values in the 'total_amount' column from the 'Orders' table to return the output as: 4697.0.

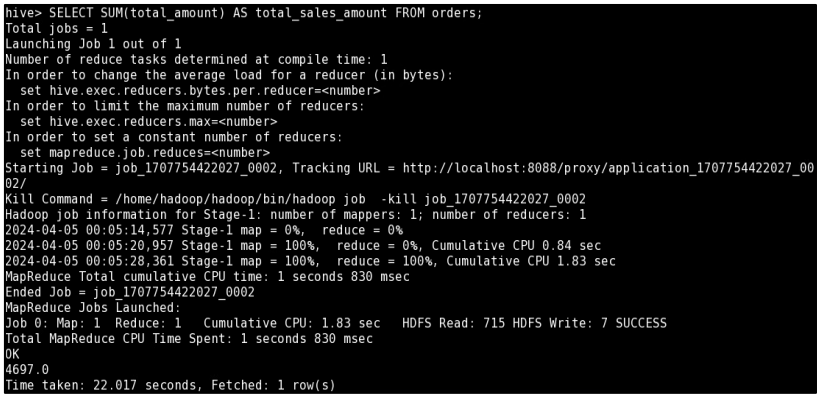


Fig 15: Screenshot displaying query and output.

3. Finding the top 5 customers who spent the most:

This query is a further implementation of the previous one where we calculate the top 5 customers who have spent the most. We utilize a SELECT query to compute the sum of 'total_amount' from the 'customers' table, followed by a JOIN operation with the 'orders' table. Subsequently, we employ the GROUP BY clause to aggregate the results based on customer identifiers and names. Finally, we sort the output in descending order to obtain the top 5 customers who have spent the most.

Fig 16: Screenshot displaying query and output.

4. List of all products in the 'Electronics' category:

In this query, we retrieve the list of all products which fall under the 'Electronics' category. This is a simple operation that can be performed using the '**SELECT'** query.

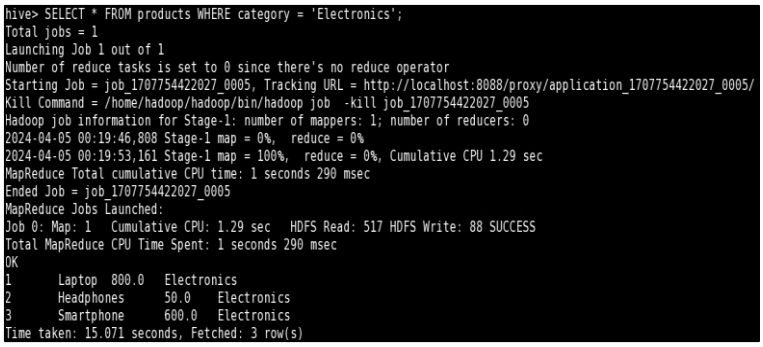


Fig 17: Screenshot displaying query and output.

5. Finding the average age of customers:

We display the average age of customers using the '**SELECT'** query which is performed using the '**AVG**' function applied on the 'age' column in the 'customers' table.

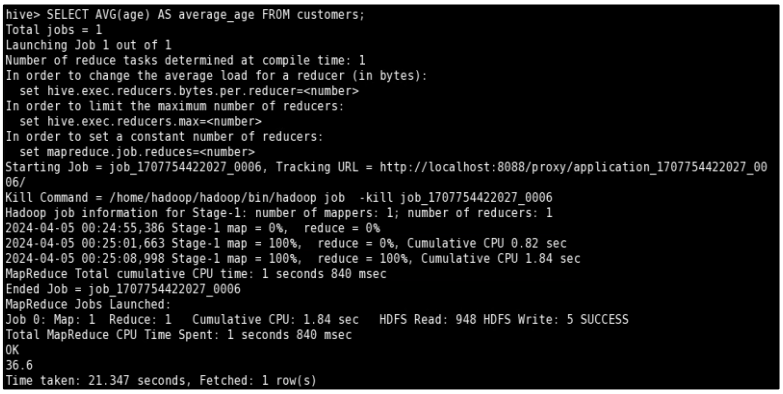


Fig 18: Screenshot displaying query and output.

6. Query to count the number of orders placed in a certain month in 2024:

This is a query to retrieve the total number of orders placed in the month of April, 2024. To do this, we use the '**COUNT'** function to retrieve total number of orders from the 'orders' table and use the '**WHERE'** command to apply a range between '2024-04-01' and '2024-04-30'.

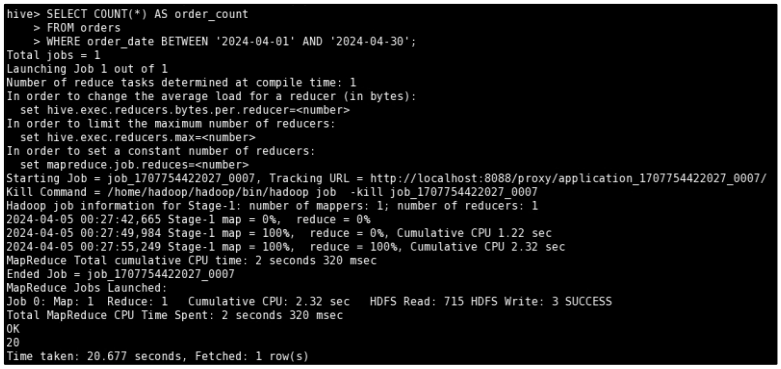


Fig 19: Screenshot displaying query and output.

7. List of customers who have not placed any orders:

This query operates to display a list of customers who have not places any orders. This is performed using the SELECT query on 'customers' table and apply a '**LEFT JOIN'** with the 'orders' table where the 'order_id' represents a '**NULL'** value.

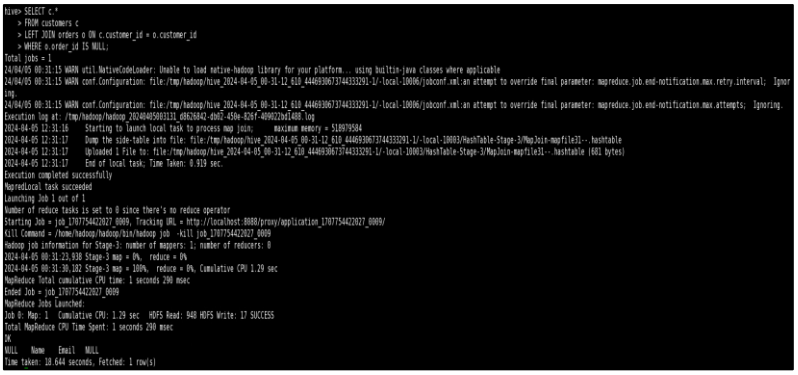


Fig 20: Screenshot displaying query and output.

8. Query to find customer who made the most expensive purchase:

This is a more complex query to find the customer who has made the most expensive purchase. We utilize the '**MAX'** function to retrieve the maximum value of 'total_amount' in the 'customers' table as the 'max_purchase_amount'. This is followed by a '**JOIN'** operation with the 'orders' table. The result is ordered by the 'max_purchase_amount' retrieved earlier in descending order and keeping a **LIMIT 1** to display the highest or most expensive purchase value.

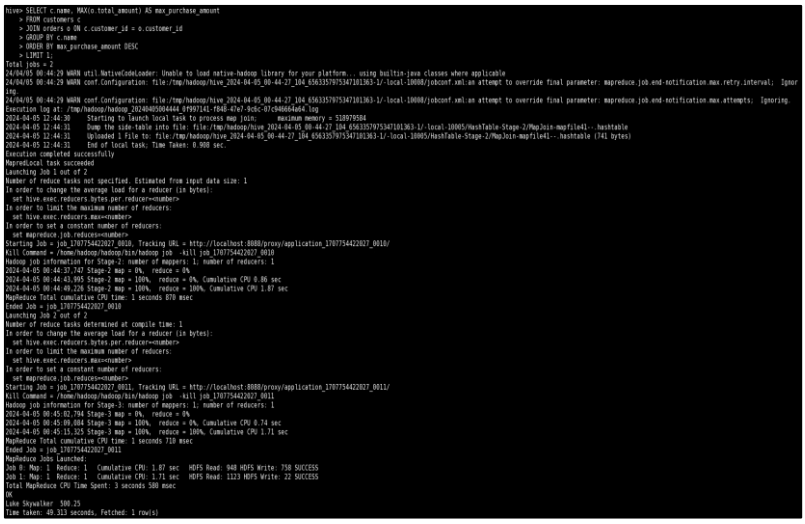


Fig 21: Screenshot displaying query and output.

9. Finding the month with the highest total sales amount:

In this query, we want to find the month when the highest total sales amount was achieved. We use the '**SELECT'** query to retrieve the year, month, and sum of 'total_sales' from the 'orders' table. We perform a '**GROUP BY'** operation using the 'YEAR' and 'MONTH' parameters and sort the result in descending order with '**LIMIT 1'** to display only the highest value or month with the total sales amount. In this case, the highest sales amount was achieved in the month of April with a value of 4697.0.

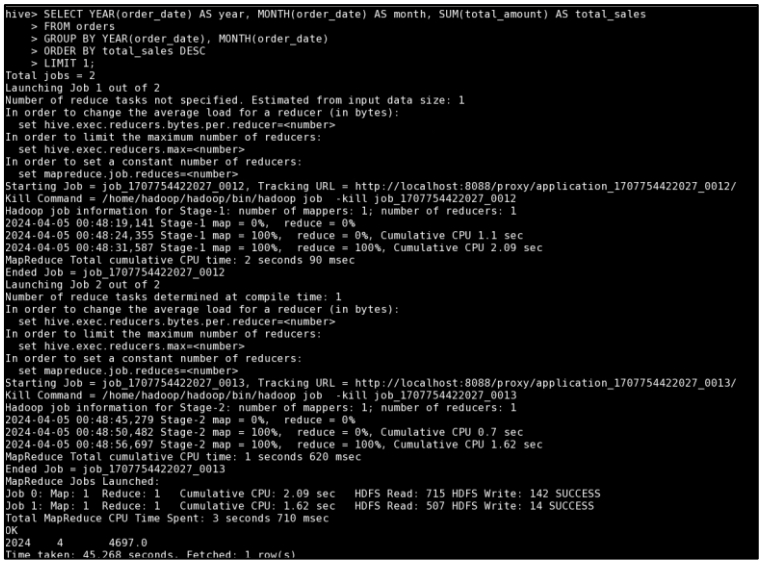


Fig 22: Screenshot displaying query and output.

10. Query for calculating the total revenue for each customer:

In this query, we calculate the total revenue for all the customers in the database. We use the '**SUM'** function to calculate the total amount from the 'customers' table and perform a '**JOIN'** operation with the 'orders' table which is then grouped by the customers' names.

Fig 23: Screenshot displaying query and output.

Task B: Map Reduce Programming

In this task we design a MapReduce algorithm using Java code to carry out certain calculations based on the Student ID number.

Student ID: **001364756**
Last digit of student ID: **'6'**.
Based on the last digit of my student ID, the task assigned to me is '**Task b.4: Calculating the average number of authors per paper for each year'**.

We consider a hypothetical situation where a computer science bibliography file is stored on Hadoop. Each line of the file contains further information about a paper in different fields which are separated by the '|' character, while the names of the authors are separated by commas(','). The format of each line of information is given as:

authors|title|conference|years

Fig 24: Bibliography content format

We assume there are no duplicate records in the file and that each conference has a different name. The task we work on based on the last digit of my student ID is to design a MapReduce algorithm using Java to calculate the average number of authors per paper for each year. We create the Java code for the MapReduce algorithm along with explanations of how the input is mapped and how the output is processed in the reduce stage.

MapReduce algorithm:

```
class MAPPER:
    method MAP(key, value):
        // Converting the input value to string
        line = value.toString()
        // Splitting the line using "|"
        tokens = line.split("|")
        // Extracting authors, title, conference, and year
        authors = tokens[0]
        title = tokens[1]
        conference = tokens[2]
        year = tokens[3]
        // Splitting authors by ","
        authorList = authors.split(",")
        // Emitting (year, number of authors) as key-value pair
        EMIT(year, authorList.length)

class REDUCER:
    // Input as year, list of author counts
    method REDUCE(year, values):
        totalAuthors = 0
```

```
 paperCount = 0
// Iterating through all values (number of authors) for a given year
 for each val in values:
    totalAuthors += val // Sum of total number of authors
    paperCount++ // Incrementing number of papers counts

// Calculating average number of authors per paper for the year
 avgAuthors = totalAuthors / paperCount
// Emitting (year, avgAuthors) as key-value pair
 EMIT(year, avgAuthors)
```

The above pseudo code consists of two major classes following the MapReduce algorithm:

Mapper class:

- The Mapper method is applied to every input key-value pair, which processes each input record and then emits an arbitrary number of intermediate key-value pairs.
- The method inputs two parameters: **'key'**: line offset and **'value'**: each line of the input file for the corresponding key.
- Inside the Map method, the input value (**'value'**) is converted to a string value (**'line'**) to facilitate processing. The string value is then split into tokens using the delimiter **'|'**, which separates the authors, title, conference, and year and facilitates their extractions.
- Authors are further split by commas (**','**) to count the number of authors.
- Finally, this method emits key-value pairs as output where the key represents the year and the value represents the number of authors count.

Reducer class:

- The Reducer class aggregates values associated with each key (year) and generates the final output key-value pairs.
- This method also takes two parameters: **'key'** and **'values'**. In this case, 'key' represents the publication year, and 'values' represents the total list of author counts calculated in the mapper class.
- Inside the Reduce method, variables **'totalAuthors'** and **'paperCount'** are initialized to zero. The method iterates through all values (number of authors) associated with the input year and calculates the total number of authors (**'totalAuthors'**) and the total number of papers (**'paperCount'**).
- Subsequently, it calculates the total number of authors per year (**'avgAuthors'**) and emits the year and corresponding average number of authors per paper as the final output key-value pair.

This MapReduce algorithm can be implemented in Java with the two classes, Mapper and Reducer, to output the average number of authors per paper for each year. We should also include an In-Mapper combiner for increasing efficiency in the code:

In-Mapper Combiner:

```
Mapper Class: AuthorCountMapper
Setup:
   Initialize HashMap authorCounts

Map(key, value):
   line = value.toString()
   fields = line.split("|")
   year = fields[3].trim()
   authorCount = fields[0].split(",").length

   // Aggregating author counts within the mapper
   authorCounts.put(year, authorCounts.getOrDefault(year, 0) + authorCount)

Cleanup:
   For each entry in authorCounts:
      Emit (year, total author count) as key-value pair
```

The above pseudo code outlines the structure of the Mapper class and how the In-Mapper combiner aggregates author counts within the mapper before emitting them for further processing.

In-Mapper Combiner Setup:

- We initialize a HashMap names 'authorCounts', which will be used to aggregate the author counts within the Mapper.

Map Function:

- In this function, we process each input record (each line of the bibliography file).
- We extract the year and number of authors for each record.
- Instead of emitting key-value pairs immediately, we update the **'authorCounts'** HashMap to aggregate author counts for each year encountered in the input records.

In-Mapper Combiner:

- The In-Mapper combiner aggregates data within the mapper task itself before emitting key-value pairs.

- By aggregating data locally within each Mapper, we reduce the amount of data that needs to be transferred over the network during the shuffle phase.
- This local aggregation minimizes the volume of intermediate data that is passed between the mapper and reducer tasks, leading to reduce network traffic and improved overall efficiency.

Cleanup:

- In the cleanup phase, we iterate through the **'authorCounts'** HashMap.
- For each entry in the HashMap we emit the aggregated author count for the corresponding year as a key-value pair.

Overall, the In-Mapper combiner optimizes the MapReduce process by performing local aggregation of data within each mapper task. This reduces the volume of data shuffle between mapper and reducer tasks, minimises network overhead, and improves the overall efficiency of the map reduce job. By aggregating data within the mapper itself we effectively combine the mapping and combining phases leading to faster processing and reduced resource utilization.

TASK C: Big Data Project Analysis

In the given scenario ABC Investment Bank Ltd wants to utilize various data sources such as social media, market data, online news, broker notes, and corporate data to produce efficient trading strategies and portfolio rebalancing decisions that would give them an edge in the competitive market. This involves dealing with extremely vast amounts of data and we are going to discuss the solutions to deal with this scenario.

Task C.1:

The business has a limited understanding of the IT world and wants us to build a data warehouse for this purpose. However, the traditional approach of using a data warehouse for this scenario may not be suitable due to the following reasons:

Data Variety: The data sources mentioned include social media, market data, online news, broker notes and corporate data, which includes a large amount of diverse and unstructured data. A data warehouse is primarily designed for structured data which may not be efficient for handling such large volumes and variety of unstructured data.

Scalability: The volume of data is expected to be in the 200's Petabytes scale, which may prove to be extremely overwhelming in a traditional data warehouse architecture. Data lakes, on the other hand, are designed to handle such large volumes of data in a wide variety of formats and provide scalability.

Flexibility: Data lakes permit storing raw, unprocessed, and unstructured data in its native format which also provides flexibility for future analysis. This proves to be an important aspect in sectors such as banking, where new data sources emerging over time may require analysis and insights compared to the old ones.

Cost-effectiveness: Data warehouse development involves storing structured data upfront which requires resources, and costs and is time-consuming. Comparatively, data lakes permit storing data in its raw form which also reduces the initial investment.

Data lakes are large pools of centralized repositories that can store structured, semi-structured, and unstructured data in their native format. Hence, for this scenario developing a data lake would be the most efficient and cost-effective (Stedman, 2021). We shall discuss the approach required for implementing a data lake:

Data Storage: To facilitate the storage of such large amounts of unstructured data, the investment company should utilize distributed file systems such as HDFS (Hadoop distributed file systems) or cloud-based storage such as Amazon S3. Attention should also be directed towards refraining from establishing and pre-defined schema for the data initially, thereby preserving its flexibility.

Data ingestion: refers to the process of importing large amounts of data from multiple data sources into a single, cloud-based storage. In this case, the utilization of technologies such as Apache Kafka or NiFi should be used to facilitate real-time data ingestion which will help create robust pipelines for collecting data from various sources such as online news, market vendors, and social media.

Metadata management: The vast amounts of data imported from various sources will contain further metadata, which will require efficient management. Tools like AWS Glue or Apache Atlas should be utilized to implement a catalog that can track and manage the metadata.

Data processing: Since it is a banking environment, the need to process such large amounts of data in real time would be imperative. Hence, the use of distributed processing networks such as Apache Spark or Flink should be implemented which will help process and support streaming and interactive processing of the data.

Data security and governance: Banking and investment are some of the most crucial and validated sectors that require the utmost importance given to security features. The use of encryption, access controls, audit trails, data backup, and network security must be prioritized for this purpose (Atlan, 2023). Attention should also be given to data governance and overall management of the data which should be done through maintaining data quality, stewardship, metadata management, and data privacy.

Data analysis and exploration: The utilization of efficient data analysis tools and algorithms to explore and analyze data should be done using tools such as machine learning and Apache Zeppelin which will build and improve efficient investment strategies (being the main goal of this project).

Monitoring and optimization: Once a robust system is built, effective measures must be undertaken to track and ensure the quality and performance of the data lake and its architecture.

By implementing a data lake, ABC Investment Bank can efficiently store, manage, and analyze large volumes of unstructured data. This will also enable them to derive valuable trading decisions and investment strategies for their clients, giving them an edge over their competitors.

Task C.2:

The investment bank also wants a system that implements near real-time performance which will enable them to act promptly once products are being discussed on social media. We shall discuss the use of parallel distributed processing on a cluster using MapReduce for this purpose and its pros, cons, and other approaches.

MapReduce is a distributed processing designed to process large-scale batches of data in parallel across a cluster of nodes (Pedamkar, 2023). While this is efficient for batch processing, it may not be an optimal choice for this scenario that requires near real-time processing due to the following reasons:

Batch processing: MapReduce is a batch processing framework, meaning it processes data in large chunks rather than continuously. This can lead to delays in processing social media and online data while acting on real-time trends.

Complexity: Implementing real-time processing using MapReduce requires additional infrastructure and complexity where developers would need to manage state, data partitioning, and ensure fault tolerance making the process complicated.

Latency: Near real-time applications require low-latency responses and interactive querying capabilities. MapReduce jobs require setting up tasks and shuffling data between nodes, which can latency in near real-time applications.

We shall now discuss alternative approaches to meet the near real-time performance requirements such as stream processing or in-memory processing.

Stream processing: Stream processing techniques provide low-latency processing, fault tolerance, and exactly-once semantics (each message or read-process-write sequence is delivered or completed exactly once) which make them ideal for near real-time applications. Technologies such as Apache Kafka Streams, Apache Flink, or Apache Spark streaming enable data streams in real time.

In-Memory processing: In-memory data processing techniques offer caching capabilities that offer low-latency data processing via storing data in memory. This offers low latency required for near real-time applications and can be implemented through techniques such as Apache Ignite or Apache Spark.

Considering the near real-time performance requirements of ABC Investment Bank, it is recommended to choose stream processing frameworks for this purpose such as Apache Flink or Apache Kafka streams. These frameworks provide the necessary features such as low latency, fault tolerance, automation, increased security, better decisions, and forecasting capabilities for processing real-time making these well-suited for the bank's investing and portfolio management applications.

Task C.3:

Eventually, the company wants us to devise a detailed hosting strategy for this Big Data project. To meet the scalability and high availability requirements for this project, it is essential to implement a comprehensive hosting strategy involving cloud-based solutions, distributed computing, and data replication (Jacinto, 2023).

Cloud Platform: The company should utilize cloud solutions such as Amazon Web Service (AWS), Microsoft Azure, or Google Cloud Platform (GCP) for hosting this Big Data project. These cloud solutions offer robust Big Data services like managing Hadoop clusters, Spark on Cloud, and real-time streaming services. These services provide a scalable infrastructure,

elasticity, and pay-as-you-go models also making them cost-effective for fluctuating data volumes.

Distributed Computing: Leveraging distributed computing frameworks such as Apache Hadoop and Spark for processing large volumes of data in parallel across a cluster of nodes provides features such as fault tolerance, data locality, scalability, and efficient data processing and analytics.

Regional deployment: Cloud solutions also provide the ability to deploy the big data project across multiple geographical regions, which ensures high availability and disaster recovery. Distributing the infrastructure across multiple regions mitigates the risk of data center outages against disasters or regional disruptions.

Data replication and backup: These public cloud providers also implement data replication and backup strategies to ensure data durability and availability. This also helps data consistency and resilience against hardware failures of network outages.

Load balancing and auto-scaling: Another advantage of utilizing cloud solutions includes load balancing and auto-scaling mechanisms which helps distribute incoming traffic evenly across multiple servers dynamically and adjust resource capacity based on data load fluctuations, ensuring optimal performance and cost-efficient resource utilization (Arunachalam, 2023).

High availability architecture: This involves using techniques such as cluster replication, and automatic failover to ensure that there is a continuous operation and availability even if there is a node failure. This will help implement redundancy within each region by deploying data and clusters in a highly available manner.

Security and compliance: Implementing robust and efficient security measures is imperative when dealing with such a large-scale finance and investing project. This can be implemented using encryptions, access controls, data backup, and network isolation and security. Ensuring compliance with industry regulations such as GDPR, PCI DSS, and HIPAA should also ensure data privacy and integrity requirements.

Monitoring and management: It is imperative to ensure the deployment of efficient monitoring and management tools for real-time monitoring of infrastructure performance, resource utilization, and overall application health. Implementing techniques such as logging, alerts, and dashboarding solutions will ensure proactively identifying and addressing potential issues.

The company should consider adopting such hosting strategies that leverage cloud-based solutions, distributed computing, multi-regional development, efficient security and network coverage, and robust data management practices through which they will be able to achieve a highly scalable, resilient, and available Big Data project for their banking and financial requirements.

References:

Arunachalam, Vijayakumar. (2023) *12 Benefits of Big Data on the Cloud.* Available at: https://payodatechnologyinc.medium.com/12-benefits-of-big-data-on-the-cloud-ca5ad1889448#:~:text=Effectively%20processing%20a%20large%20amount,for%20businesses%20of%20all%20sizes. (Accessed: 07 April 2024).

Atlan's *Data Governance vs Data Security (2023).* Available at: https://atlan.com/data-governance-vs-data-security/ (Accessed: 05 April 2024).

Jacinto, Andrea. (2023) *BIG DATA BEST PRACTICES FOR MAXIMIZING BUSINESS IMPACT.* Available at: https://www.startechup.com/blog/big-data-best-practices/ (Accessed: 06 April 2024).

Pedamkar, Priya. (2023) *What is MapReduce?* Available at: https://www.educba.com/what-is-mapreduce/ (Accessed: 07 April 2024).

Stegman, Craig. (2021) *DEFINITION data lake.* Available at: https://www.techtarget.com/searchdatamanagement/definition/data-lake (Accessed: 07-April-2024).