# CSP 554 – Big Data Technologies

## *Fall 2023 – All Sections*

### *Midterm Exam - Part 1*

### Ankan Mazumdar- A20541357

**Part I** – Short Answer (Show Points/Results) – 5 points each, 30 points total

1. Given the following steps of a Map-Reduce job, order them in correct sequence that they are executed in: Partition, Combine, Map, Shuffle, Reduce. Which steps are guaranteed to execute? Which are not?

Answer- The steps follow this specific sequence:

1. Map Phase: This initial step is the starting point for a Map-Reduce task. During the Map phase, each input record is processed autonomously and transformed into key-value pairs. The Map phase organizes and sorts the generated data based on these keys.

2. Combine Phase (Optional): The Combine step, while not mandatory, is an additional phase that comes after the Map phase. Sometimes referred to as the "mini-reduce," it serves to locally aggregate data within each Mapper node before transmitting it to the Reducer. The purpose of the Combine phase is to minimize the amount of data transferred during the Shuffle and sort phase, which can enhance overall job performance.

3. Shuffle and Sort Phase: Following the Map and, if applicable, Combine phases, the Shuffle and Sort phase comes into play. This step is instrumental in redistributing data from the Mapper nodes to the Reducers. During this phase, the data is rearranged and arranged according to the keys, ensuring that each Reducer obtains the pertinent key-value pairs from the Mappers.

4. Partition Phase: Subsequent to the Shuffle and Sort phase, the Partition step takes effect. It figures out which Reducer should be assigned each key-value pair and guarantees that all key-value pairs with identical keys are routed to the same Reducer.

5. Reduce Phase: The final step is the Reduce phase, where each Reducer processes the grouped and ordered key-value pairs corresponding to its assigned keys. Within this phase, a user-defined reduce function is applied to the data, consolidating and processing the values associated with each key.

Certain to Execute:

- The Map and Reduce phases are integral and are executed in every Map-Reduce job, forming the core elements of the Map-Reduce framework. Shuffle and partition are integral part of Map reduce.

Not Necessarily Guaranteed to Execute:

- The Combine step is an optional component and may or may not run, contingent on the configuration settings of the job. It is utilized primarily for performance optimization but is not obligatory.

1. If a Map-Reduce job produces output in the Map stage across M maps nodes, and a total of N reduce nodes are available, describe a function to evenly distribute all Key-Value Pairs <K,V> from the Map job across all N reducers. Does this change if M<N?

Answer- In a Map-Reduce job, it's crucial to evenly distribute Key-Value pairs <K, V> from the Map stage across N reducers. To achieve this, we can use a partitioning function that assigns each key to a specific reducer. This helps ensure a balanced workload across all reducers.

Here's a definition of the partitioning function with all the necessary steps:

function partition(K, N):

# Step 1: Apply a hash function to the key K to get a numerical value

hash\_value = hash(K)

# Step 2: Perform a modulo operation to assign a reducer number

reducer\_number = hash\_value % N

# Step 3: Return the assigned reducer number

return reducer\_number

The partitioning function can be used during the Map stage for each Key-Value pair. It takes the key K and the total number of reducers N as inputs, and it returns the reducer number to which that key should be assigned.

The steps are as follows:

1. Hash the Key: The partitioning function applies a hash function to the key K, which converts the key into a numerical value. The hash function should be deterministic, meaning that for the same key K, it should always produce the same numerical result.

2. Perform Modulo Operation: The hash value obtained from step 1 is then subjected to a modulo operation with the total number of reducers, N. This operation ensures that the key is assigned to one of the N reducers. The result is the reducer number to which the Key-Value pair will be sent.

3. Assign to Reducer: Finally, the partitioning function returns the reducer number. This indicates the destination reducer for the Key-Value pair. All Key-Value pairs that produce the same reducer number will be sent to the same reducer for further processing.

This partitioning function helps distribute data evenly across all the available reducers, ensuring efficient parallel processing. If the number of Map tasks (M) is less than the number of reducers (N), we can still use this partitioning function, but we should configure the system to handle situations where some reducers may not receive data from certain Map tasks, possibly resulting in idle reducers.

1. Explain the difference between the traditional relational database concept of Schema-On-Write versus the Hive paradigm of Schema-On-Read. How is this related to the architecture of Hive?

Answer- Schema-on-write is a traditional relational database concept where the schema of the data must be defined before the data is loaded into the database. This ensures that the data is consistent and that all queries can be answered efficiently. However, schema-on-write can be inflexible and time-consuming, especially for data warehouses that need to handle a variety of different data sources and formats.

Schema-on-read is a Hive paradigm where the schema of the data is defined when the data is read, not when it is loaded. This makes Hive more flexible and scalable, as it can handle a variety of different data sources and formats without the need to pre-define a schema. However, schema-on-read can be less efficient for queries that require complex joins or aggregations.

Hive architecture is designed to support schema-on-read. Hive uses a metastore to store the schema of the data. When a Hive query is executed, the metastore is used to determine the schema of the data and to generate the appropriate MapReduce job.

Here is a table that summarizes the key differences between schema-on-write and schema-on-read:

|  |  |  |
| --- | --- | --- |
| Feature | Schema-on-write | Schema-on-read |
| Schema definition | Before data is loaded | When data is read |
| Flexibility | Less flexible | More flexible |
| Scalability | Less scalable | More scalable |
| Query performance | More efficient for complex joins and aggregations | Less efficient for complex joins and aggregations |

Hive's schema-on-read architecture makes it well-suited for data warehousing and big data analytics. Hive can be used to process a variety of different data sources and formats, including structured, semi-structured, and unstructured data. This makes Hive a powerful tool for exploring and analyzing large datasets.

Here are some examples of how Hive's schema-on-read architecture can be used:

* Loading data from a variety of sources: Hive can be used to load data from a variety of sources, including relational databases, NoSQL databases, and file systems. This makes it easy to combine data from different sources into a single data warehouse.
* Processing semi-structured and unstructured data: Hive can be used to process semi-structured and unstructured data, such as JSON, XML, and text files. This makes it easy to analyze data from a variety of sources, including social media data, website logs, and customer reviews.
* Exploring data: Hive can be used to explore data using a variety of SQL-like queries. This makes it easy to get insights from large datasets without having to write custom code.
* Data Storage: Data in Hive is often stored in raw, flexible formats like text files or columnar storage (e.g., Parquet). This allows for efficient storage and flexibility in handling different data types and structures.
* Metadata and Schemas: Hive maintains metadata about the data, including the schema, in a Metastore. The schema is stored independently of the data and is applied dynamically when querying. This separation of metadata and data is a key feature of Hive's architecture, allowing for Schema-On-Read flexibility.
* Query Processing: When queries are executed in Hive, the schema is applied as part of the query processing. This approach enables users to perform complex queries on data with different structures without the need for extensive upfront data transformation.

Overall, Hive's schema-on-read architecture makes it a powerful and flexible tool for data warehousing and big data analytics.

The key difference between the Schema-On-Write and Schema-On-Read paradigms lies in when and how the schema is applied to the data. Hive follows the Schema-On-Read approach, where the schema is enforced at the time of querying, providing more flexibility and adaptability to handle diverse data sources and formats.

1. For a Pig script operating on a Relation that has the following schema: {UserId, UserAge, UserIncome}, what transform would allow us to obtain all records with UserAge >= 18? How would we then obtain average UserIncome by UserAge?

Answer-

To obtain all records with UserAge greater than or equal to 18 from a Pig script operating on a relation with the schema {UserId, UserAge, UserIncome}, we can use the FILTER transformation. Here's how we can do it:

filtered\_relation = FILTER relation BY UserAge >= 18;

In the above script, FilteredData will contain all the records where UserAge is greater than or equal to 18.

To obtain the average UserIncome by UserAge from the FilteredData, we can use the GROUP BY and FOREACH transformations along with the AVG function. Here's how to do it

To obtain average UserIncome by UserAge, we can use the following Pig transform:

grouped\_relation = GROUP filtered\_relation BY UserAge;

average\_income\_by\_age = FOREACH grouped\_relation GENERATE UserAge, AVG(UserIncome);

In the above script, AverageIncomeByAge will give we the average UserIncome for each unique UserAge group from the FilteredData. This is done by grouping the data by UserAge and then using the AVG function to calculate the average UserIncome for each group.

This transform will group the records in the filtered relation by UserAge and then calculate the average UserIncome for each group. The result is a new relation that contains one row for each unique UserAge value, with the average UserIncome for that age group.

Here is an example of a complete Pig script that obtains all records with UserAge >= 18 and then calculates the average UserIncome by UserAge:

relation = LOAD '/path/to/relation/file';

filtered\_relation = FILTER relation BY UserAge >= 18;

average\_income\_by\_age = GROUP filtered\_relation BY UserAge;

average\_income\_by\_age = FOREACH average\_income\_by\_age GENERATE AVG(UserIncome);

STORE average\_income\_by\_age INTO '/path/to/output/file';

This script will load the relation file from the /path/to/relation/file path, filter the relation to only include records where the UserAge column is greater than or equal to 18, group the filtered relation by UserAge, calculate the average UserIncome for each group, and then store the resulting relation to the /path/to/output/file path.

1. Given a Hive table containing the following fields: CustomerId, CustomerName, CustomerCity, CustomerState - what parts of a HQL query would allow for pruning of partitions if the table was partitioned by: CustomerCity, CustomerState.

Answer -

To prune partitions when querying a Hive table that is partitioned by CustomerCity and CustomerState, we can use the following parts of a HQL query:

* WHERE clause: The WHERE clause can be used to filter the data based on the partition columns. For example, the following WHERE clause would filter the data to only include records where the CustomerCity column is equal to 'San Francisco' and the CustomerState column is equal to 'CA':

SQL

WHERE CustomerCity = 'San Francisco' AND CustomerState = 'CA'

* IN clause: The IN clause can also be used to filter the data based on the partition columns. For example, the following WHERE clause would filter the data to only include records where the CustomerCity column is in the list of values ['San Francisco', 'Los Angeles', 'New York']:

SQL

WHERE CustomerCity IN ('San Francisco', 'Los Angeles', 'New York')

* JOIN clause: The JOIN clause can be used to join the table with another table that is partitioned by the same columns. For example, the following JOIN clause would join the Customer table with a Product table that is partitioned by ProductCity and ProductState:

SQL

JOIN Product ON Customer.CustomerCity = Product.ProductCity AND Customer.CustomerState = Product.ProductState

When the Hive query optimizer detects that the WHERE clause, IN clause, or JOIN clause can be used to prune partitions, it will generate a more efficient query plan that only scans the relevant partitions.

Example of a HQL query that uses the WHERE clause to prune partitions:

SQL

SELECT \*

FROM Customer

WHERE CustomerCity = 'San Francisco' AND CustomerState = 'CA';

This query will only scan the partitions of the Customer table where the CustomerCity column is equal to 'San Francisco' and the CustomerState column is equal to 'CA'.

Partition pruning can significantly improve the performance of Hive queries, especially when the table is partitioned on columns that are frequently used in filters and joins.

1. For a HDFS filesystem configured for a replication factor of 3, what are the maximum number of data node failures that can be tolerated before data loss occurs? If a total of 12PB of data is stored on the filesystem, how much physical storage is required given this replication factor?

Answer- For a HDFS filesystem configured for a replication factor of 3, the maximum number of data node failures that can be tolerated before data loss occurs is 2. This is because each block of data is replicated to 3 data nodes, so if 2 data nodes fail, there will still be one copy of the data remaining.

If a total of 12PB of data is stored on the filesystem, then the total physical storage required given a replication factor of 3 is 36PB. This is because each block of data is replicated to 3 data nodes, so the total physical storage required is 3 times the amount of data stored on the filesystem.

Here is the calculation to show how this works:

Physical storage required = Replication factor \* Data stored

Physical storage required = 3 \* 12PB

Physical storage required = 36PB

It is important to note that the replication factor is a trade-off between durability and performance. A higher replication factor will make the filesystem more durable, but it will also require more physical storage and reduce the overall performance of the filesystem.

**Part II** – Long Answer (Show Reasoning/Calculations) – 10 points each, 20 points total

1. Within a distributed system, explain the difference between a Fault, an Error, and a Failure. Define the types of Faults which can occur - does a Fault always lead to a Failure? Define the types of Failures which can result - what recovery mechanisms exist to address Failure modes?

1. Fault:

- Definition: A fault is an anomaly or defect within a system that deviates from its expected behavior. It represents a departure from the norm but doesn't necessarily cause immediate system failure.

- Types of Faults:

- Transient Fault: A temporary fault that eventually resolves itself, like a temporary network packet loss.

- Intermittent Fault: A fault that occurs unpredictably and sporadically, with no fixed pattern.

- Permanent Fault: A fault that persists in the system and doesn't self-correct.

- Does a Fault always lead to a Failure?: No, a fault doesn't always result in a failure. Whether it leads to failure depends on the nature of the fault and the system's ability to tolerate or recover from it.

2. Error:

- Definition: An error is a discrepancy between the expected behavior of a system and its actual behavior. Errors often arise from faults or unexpected conditions and can be observed during system operation.

- Types of Errors:

- Software Error: A coding mistake or bug causing incorrect system behavior.

- Hardware Error: A malfunction or defect in hardware components.

- Configuration Error: Errors arising from incorrect system configuration.

- User Error: Errors caused by users interacting with the system.

3. Failure:

- Definition: A failure represents a significant deviation from the correct system behavior, rendering the system incapable of performing its intended functions or delivering expected results.

- Types of Failures:

- Crash Failure: Sudden system unresponsiveness or inoperability.

- Omission Failure: The system fails to respond to inputs or produce outputs.

- Byzantine Failure: The system produces arbitrary or incorrect outputs, often due to malicious intent.

- Performance Degradation Failure: The system's performance falls below acceptable levels without a complete crash.

-Timing failures: These are failures where the system produces correct results, but at the wrong time.

Recovery Mechanisms for Failures:

Replication: Replication is the process of storing multiple copies of data or components on different nodes in the system. This way, if one node fails, the system can continue to operate using the other nodes.

Error detection and correction: Error detection and correction (EDC) is the process of detecting and correcting errors in data. This can be done using a variety of techniques, such as checksums and parity bits.

Failover: Failover is the process of switching to a backup system when the primary system fails. This can be done manually or automatically.

Some other recovery mechanisms- Various mechanisms address failures, including redundancy, checkpoint and restart, replication, error detection and correction codes, monitoring and alerting, failover and load balancing, and graceful degradation.

In summary, a fault is a deviation from the expected behavior, an error is a discrepancy between the expected and actual behavior, and a failure is a significant deviation leading to system malfunction. Recovery mechanisms are employed to address failures, ensuring system resilience and reliability. Whether a fault leads to a failure depends on the system's robustness and fault tolerance measures.

1. We are provided with a document dataset consisting of a series of sentences (one per line). What would be the Map task which would yield each word in each sentence as a tuple? Define the keys and values. What subsequent Reduce task would allow us to obtain a total count for each word?

Answer-

Map Task:

In the Map task, we would split each sentence into words, emit each word as a key, and set the count (usually 1) as the corresponding value. The key-value pair emitted by the Map task would be of the form (word, 1).

Pseudo-code for the Map task:

```

map(line):

# Split the sentence into words

words = line.split()

for word in words:

# Emit each word as a key with a count of 1

emit(word, 1)

```

In this task:

- Keys: The keys in the Map task are the individual words in each sentence. Each word is considered a key.

- Values: The values in the Map task are typically set to 1. Each word is associated with a value of 1, indicating that the word has occurred once in the sentence.

Reduce Task:

In the Reduce task, we would receive key-value pairs from the Map task, group them by the word, and then sum up the counts for each word to get the total count.

Pseudo-code for the Reduce task:

```

reduce(word, counts):

# Initialize a variable to store the total count for the word

total\_count = 0

for count in counts:

# Sum up the counts for the same word

total\_count += count

# Emit the word and its total count

emit(word, total\_count)

```

In this task:

- Keys: The keys in the Reduce task are the unique words (as they are the result of grouping). Each unique word is considered a key.

- Values: The values in the Reduce task are counts associated with each word. These counts represent how many times each word has appeared in the dataset.

This MapReduce process efficiently yields the total count for each unique word in the document dataset, with each word acting as the key and the total count of that word as the value. This approach leverages parallel processing capabilities for word count analysis across a large dataset.