VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION AND TECHNOLOGY

BIG DATA

Document Clustering with Hadoop MapReduce

Class 21KHMT

Group 7

21127021 - Trương Văn Chí
21127430 - Nguyễn Huy Thành
21127166 - Nguyễn Tuấn Thanh
21127290 - Nguyễn Gia Hưng

Supervisors:

Nguyễn Ngọc Thảo
Đỗ Trọng Lễ
Bùi Huỳnh Trung Nam

Ho Chi Minh - 3/2024

CONTENT

LIST OF FIGURE

LIST OF TABLE

I. Role and assignment of each member:

Table I.1 Table of role

	Name	Student ID	Role
KHMT01	Trương Văn Chí	21127021	Leader
	Nguyễn Tuấn Thanh	21127166	Member
	Nguyễn Gia Hưng	21127290	Member
	Nguyễn Huy Thành	21127430	Member

Table I.2 Checklist and Task assignment

Task	Name	Student ID	Score	Finished
Task 1.1: Text Cleaning and Term Frequency	Nguyễn Gia Hưng	21127290	1	100%
Task 1.2: Low-Frequency Term Elimination	Nguyễn Huy Thành	21127430	1	100%
Task 1.3: Top 10 Most Frequent Words	Nguyễn Tuấn Thanh	21127166	1	100%
Task 1.4: TF-IDF	Nguyễn Gia Hưng	21127290	1	100%
Task 1.5: Highest average tfidf	Nguyễn Huy Thành	21127430	1	100%
Task 2.1: K-Means on 2D Data	Nguyễn Tuấn Thanh	21127166	1.5	100%
Task 2.2: K-Means on Preprocessed Data	Trương Văn Chí	21127021	1.5	100%
Task 2.3: Scalable K-Means++ Initialization	Trương Văn Chí	21127021	1	100%
Report	Trương Văn Chí	21127021	1	100%
	Nguyễn Tuấn Thanh	21127166		
	Nguyễn Gia Hưng	21127290		
	Nguyễn Huy Thành	21127430		
Total			10	100%

II. TF-IDF (Term Frequency-Inverse Document Frequency)

1. Data Description:

In this lab, our text data is stored in the **BBC** folder, which contains five subfolders: business, sports, tech, entertainment, and politics.

- **Business** folder has 510 documents named: doc_id.txt
- **Tech** folder has 401 documents named: doc_id.txt
- **Entertainment** folder has 386 documents name: doc_id.txt
- **Politics** folder has 417 documents name: doc_id.txt
- **Sport** folder has 511 documents name: doc_id.txt

Additionally, we possess a document concerning stopwords that is utilized for preprocessing our raw text data.

2. MapReduce Job Implementation

2.1. Overall design of MapReduce job

In this section, I will discuss the setup of each MapReduce task and its corresponding job. I'll provide details on the number of mappers and reducers utilized, as well as the inputs and outputs involved.

- **Task 1.1:** This task involves cleaning the text by replacing words in documents with corresponding **term_id** and **docs_id**. It also calculates term frequency and presents it in the **.mtx** file format. This task has one input (folder **bbc**) and one output (file **Task_1_1.mtx**). It employs one mapper, one reducer, and one driver to execute the MapReduce job.

- **Task 1.2:** Here, the goal is to filter terms where the frequency is less than 3 across all documents. The input is the file **Task_1_1.mtx**, and the output is the file **Task_1_2.mtx**. Similar to Task 1.1, this task employs one mapper, one reducer, and one driver.

- **Task 1.3:** This task calculates the total frequency of each term across all documents and sorts the terms in descending order based on their total frequency. It takes **Task_1_2.mtx** as input and has two outputs: one in the **tmp** folder containing **part-r-00000**, and another as **Task_1_3.txt**. It utilizes two mappers, two reducers, and one driver for the MapReduce job.

- **Task 1.4:** Task 1.4 involves two subtasks: one for calculating **Term Frequency (TF)** - $tf(t,d)$ based on frequent terms in each document - $f(t,d)$, and another for calculating **Inverse Document Frequency (IDF)** - $idf(t,D)$ using the result from the first job to compute TFIDF by multiplying TF and IDF. It takes **Task_1_2.mtx** as input and has two outputs: one

in the **tmp** folder containing **part-r-00000**, and another as **Task_1_4.mtx**. Similar to Task 1.3, this task employs two mappers, two reducers, and one driver for the MapReduce job.

- **Task 1.5:** This task computes, for each term, the average TFIDF over each class C_i (documents in class i). The input is **Task_1_4.mtx**, and the output is **Task_1_5.txt**. Like Task 1.3 and Task 1.4, this task utilizes two mappers, two reducers, and one driver for the MapReduce job.

The information on resource allocation (mappers, reducers, drivers) and data flow (inputs and outputs) facilitates the efficient execution and monitoring of the text classification pipeline.

2.2. In-Depth Analysis of a MapReduce Job:

Task 1.1:

How to run MapReduce job:

```
hadoop jar <path to jar file> <input file path> <output directory path>
```

Logic of Map function:

Tokenization: The input value is tokenized into individual words using a StringTokenizer.

File Information Extraction: The function extracts information about the input file, such as the parent directory name and the file name. This information is crucial for determining the document associated with each term.

Document Identification: The function constructs a unique identifier for the current document being processed. It concatenates the parent directory name with a modified version of the file name (excluding the file extension) to create this identifier.

Processing Tokens: For each token in the input, the function performs the following steps:

- Converts the token to lowercase for case-insensitive matching.

- Checks if the token is a stop word. If it is, the token is skipped, and the function moves to the next token.

- Retrieves the corresponding term ID from the terms_id mapping. If the token does not have a corresponding term ID, it is skipped.

- Constructs a key-value pair where the key consists of the term ID and the document ID, separated by a space, and the value is set to 1 (one).

- Writes the key-value pair to the Context object, which manages the intermediate out-

put of the Map phase.

Logic of Reducer function:

Reduce Operation: For each unique key received, the function iterates over the associated values. In this case, the values are of type FloatWritable, denoting the occurrences or frequencies of the key.

Aggregation: Within the loop, the function accumulates the values by adding them together. This step aggregates the occurrences or frequencies of the key across all occurrences in the input.

Setting Result: Once all values associated with the key have been processed, the aggregated result is set to the result variable.

Emitting Output: Finally, the function writes the key-value pair to the output context. The key remains unchanged, while the value is set to the aggregated result obtained in the previous step.

Format of Keys and Values at each stage: In the input map task, a pair of <Object, Text> will be passed. The output of the map task will be an intermediate pair of <Text, FloatWritable>. Subsequently, a pair of <Text, Iterable<FloatWritable> > will be passed to the reduce task, which will then return a pair of <Text, FloatWritable>.

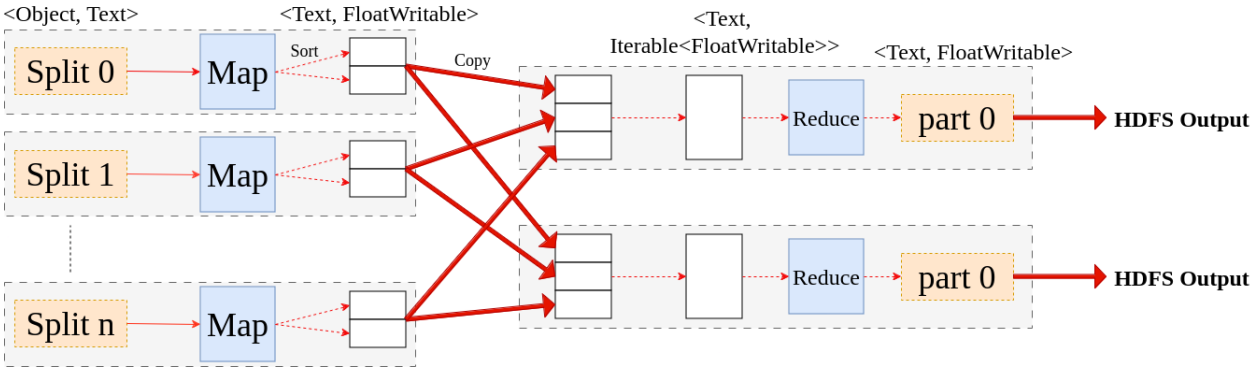

Fig II.1 Format of Keys and Values at each stage Task_1_1

Results following the execution of the MapReduce job:

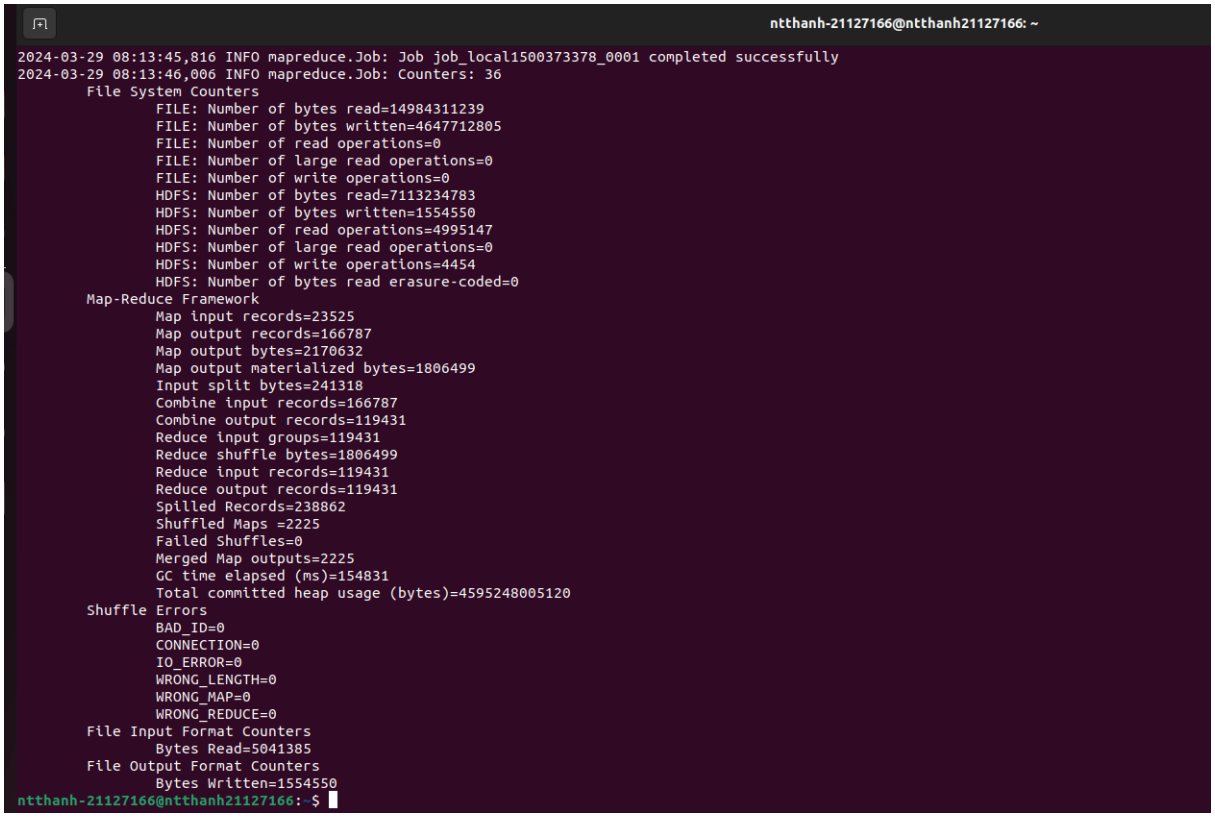

Fig II.2 MapRecuce Job of Task_1_1: Text Cleaning and Term Frequency

Task 1.2:

How to run MapReduce job:

```
hadoop jar <path to jar file> <input file path> <output directory path>
```

Logic of Map function:

Tokenization and Processing: The while loop iterates over each tokenized word in the input value. For each token:

- The token is extracted using *itr.nextToken()*.

- The extracted token is set as the value of the word Text object.

- The entire original input value (as a string) is set as the value of the number Text object.

- The loop breaks after processing the first token, ensuring that only the first word of each line is processed.

Logic of Reduce function:

Processing Intermediate Values: The function iterates over each intermediate value associated with the key. For each value:

- A StringTokenizer named *itr* is created to tokenize the value.

- A loop iterates over each tokenized element.

- The index variable is used to track the position of the token. If the index is 2 (indicating a specific position in the value), the token is parsed as a float and added to the sum.

- Regardless of the position, the token is added to thislist.

Format of Keys and Values at each stage: In the map task, an <Object, Text> pair will be passed as input. The map task will then produce an intermediate <Text, Text> pair. Subsequently, a <Text, Iterable<Text» pair will be passed to the reduce task, which will return a final <Text, Text> pair.

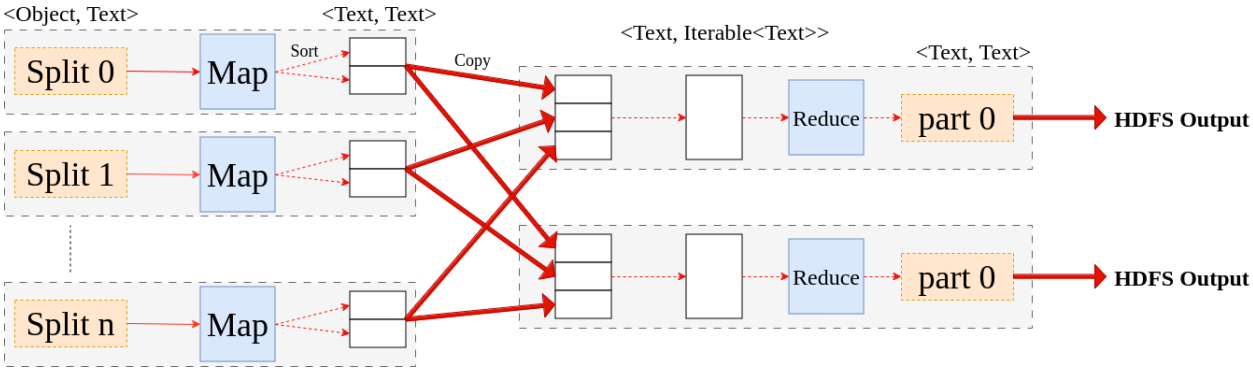

Fig II.3 Format of Keys and Values at each stage Task_1_1

Results following the execution of the MapReduce job:

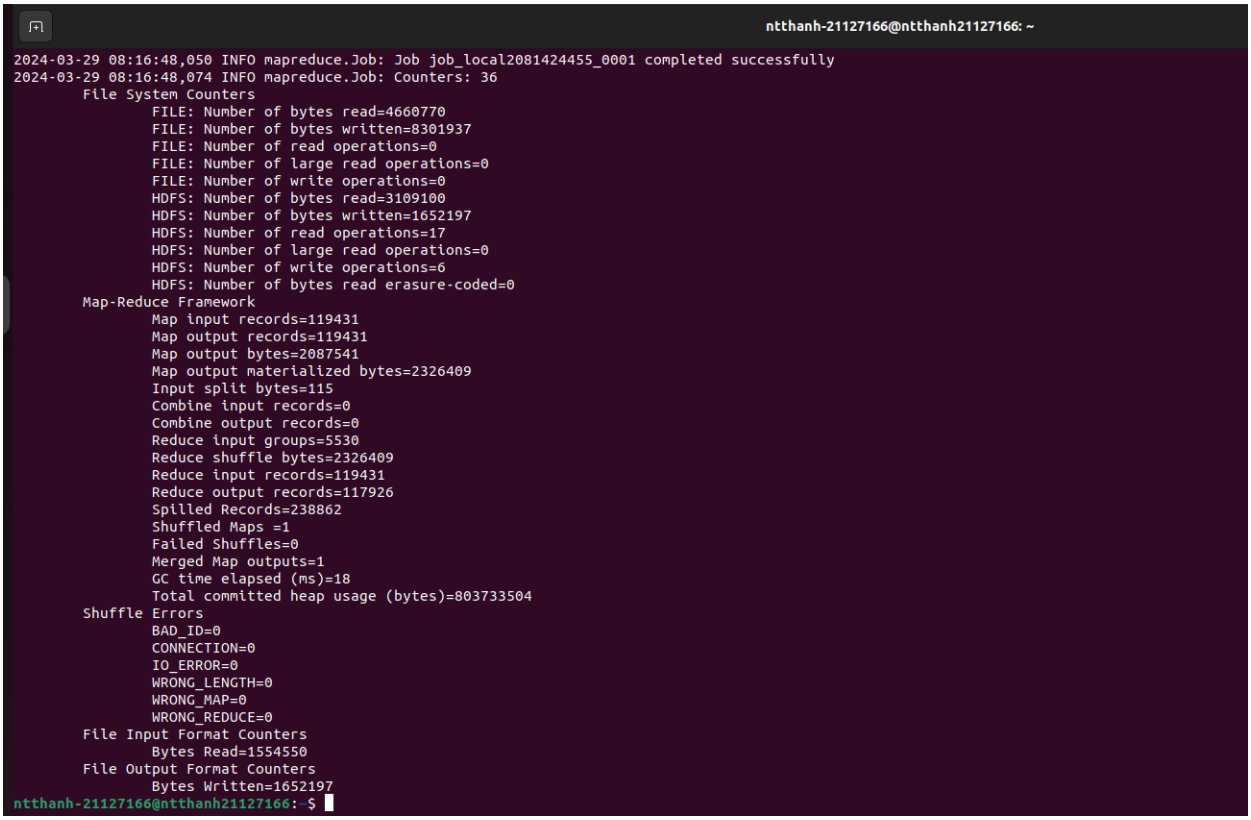

Fig II.4 MapRecuce Job of Task_1_2: Low-Frequency Term Elimination

Task 1.3:

How to run MapReduce job:

```
hadoop jar <path to jar file> <input file path> <output temp path>
<output directory path>
```

Logic of first Map function:

Mapping Operation: The map function receives input key-value pairs, where the key is of type Object and the value is of type Text. It overrides the default map method provided by the Mapper class.

Tokenization: Within the function, the input value is tokenized into individual strings using a StringTokenizer. This tokenizer splits the text based on whitespace by default.

Key-Value Pair Formation: For each tokenized string, the function performs the following steps:

- Sets the *pairKey* to the current token, assuming it represents a term or identifier.

- Advances the tokenizer to the next token (skipping one token), assuming it represents

the frequency or numerical value associated with the term.

- Retrieves the frequency value from the next token and converts it to a Float type.

- Sets the *pairValue* to the obtained frequency value.

- Emits the intermediate key-value pair (*pairKey*, *pairValue*) using the context.write method.

Logic of first Reducer function:

Reduce Operation: For each unique key received, the function iterates over the associated values provided by the Iterable<FloatWritable> values. Within the loop, it accumulates the values by adding them together. This aggregation step calculates the total sum of values associated with the key across all occurrences in the input.

Aggregation: The loop iterates through all values associated with the current key, summing them up to obtain the total sum.

Setting Result: Once all values associated with the key have been processed and aggregated, the total sum is set to the result variable.

Emitting Output: Finally, the function emits the key-value pair to the output context using context.write(key, result). The key remains unchanged, while the value is set to the aggregated sum obtained in the previous step.

Logic of second Map function:

Tokenization: The input value is tokenized using a StringTokenizer. This tokenizer splits the input text into tokens, separated by whitespace.

Pair Formation: Within the while loop, the mapper iterates through the tokens obtained from the input text. For each iteration, it forms a key-value pair where the key remains empty (*pairKey*) and the value (*pairValue*) is constructed by concatenating the current token with the next token obtained from the tokenizer. This concatenation is performed to create pairs of adjacent tokens.

Emitting Key-Value Pairs: After forming the key-value pair, the mapper writes this pair to the output context using the context.write() method. The key remains constant (*pairKey*), while the value is set to the concatenated pair obtained earlier.

Logic of second Reducer function:

Processing Values: For each unique key received, the reducer iterates over the associ-

ated values. Each value is tokenized to extract the term and its frequency. These key-value pairs are stored in the maps HashMap.

Sorting Frequencies: After processing all values, the reducer sorts the frequencies stored in the list ArrayList in descending order. This sorting step is crucial for identifying the top terms based on their frequencies.

Output Generation: The reducer then iterates over the sorted frequency list. For each frequency value, it searches the maps HashMap to find the corresponding term. The top 10 terms, along with their frequencies, are emitted as output key-value pairs.

Format of Keys and Values at each stage:

The input map of the first job will receive a pair of <Object, Text>. The output map of the first job will produce an intermediate pair of <Text, FloatWritable>. Subsequently, a pair of <Text, Iterable<FloatWritable» will be passed to the reducer of the first job, which will then return a pair of <Text, Text>.

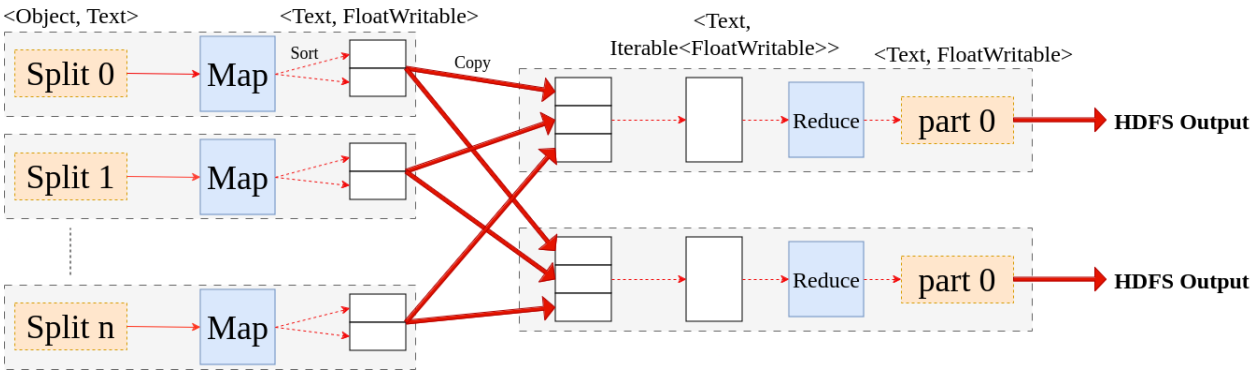

Fig II.5 Format of Keys and Values at each stage of the second MapReduce Job in Task_1_3

In the second job, the input map will receive a pair of <Object, Text>. The map's output will produce intermediate <Text, Text> pairs. Subsequently, <Text, Iterable<Text» pairs will be passed to the reducer of the second job, which will then return <Text, Text> pairs.

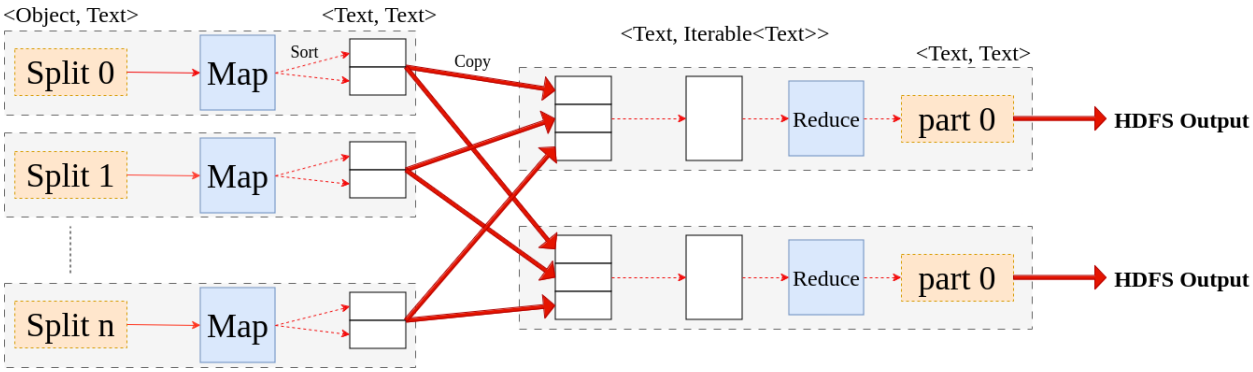

Fig II.6 Format of Keys and Values at each stage of the second MapReduce Job in Task_1_3

Results following the execution of the MapReduce job:

```
ntthanh-21127166@ntthanh21127166: ~
2024-03-30 21:23:53,415 INFO mapred.LocalJobRunner: Finishing task: attempt_local1786597941_0001_r_000000_0
2024-03-30 21:23:53,415 INFO mapred.LocalJobRunner: reduce task executor complete.
2024-03-30 21:23:54,034 INFO mapreduce.Job:  map 100% reduce 100%
2024-03-30 21:23:54,034 INFO mapreduce.Job: Job job_local1786597941_0001 completed successfully
2024-03-30 21:23:54,062 INFO mapreduce.Job: Counters: 36
        File System Counters
                FILE: Number of bytes read=112432
                FILE: Number of bytes written=1477115
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=3304394
                HDFS: Number of bytes written=42610
                HDFS: Number of read operations=19
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=6
                HDFS: Number of bytes read erasure-coded=0
        Map-Reduce Framework
                Map input records=117926
                Map output records=117926
                Map output bytes=997224
                Map output materialized bytes=48967
                Input split bytes=115
                Combine input records=117926
                Combine output records=4513
                Reduce input groups=4513
                Reduce shuffle bytes=48967
                Reduce input records=4513
                Reduce output records=4513
                Spilled Records=9026
                Shuffled Maps =1
                Failed Shuffles=0
                Merged Map outputs=1
                GC time elapsed (ms)=80
                Total committed heap usage (bytes)=779091968
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=1652197
        File Output Format Counters
                Bytes Written=42610
```

Fig II.7 MapRecuce Job of Task_1_3: Calculating the sum of each term across all documents.

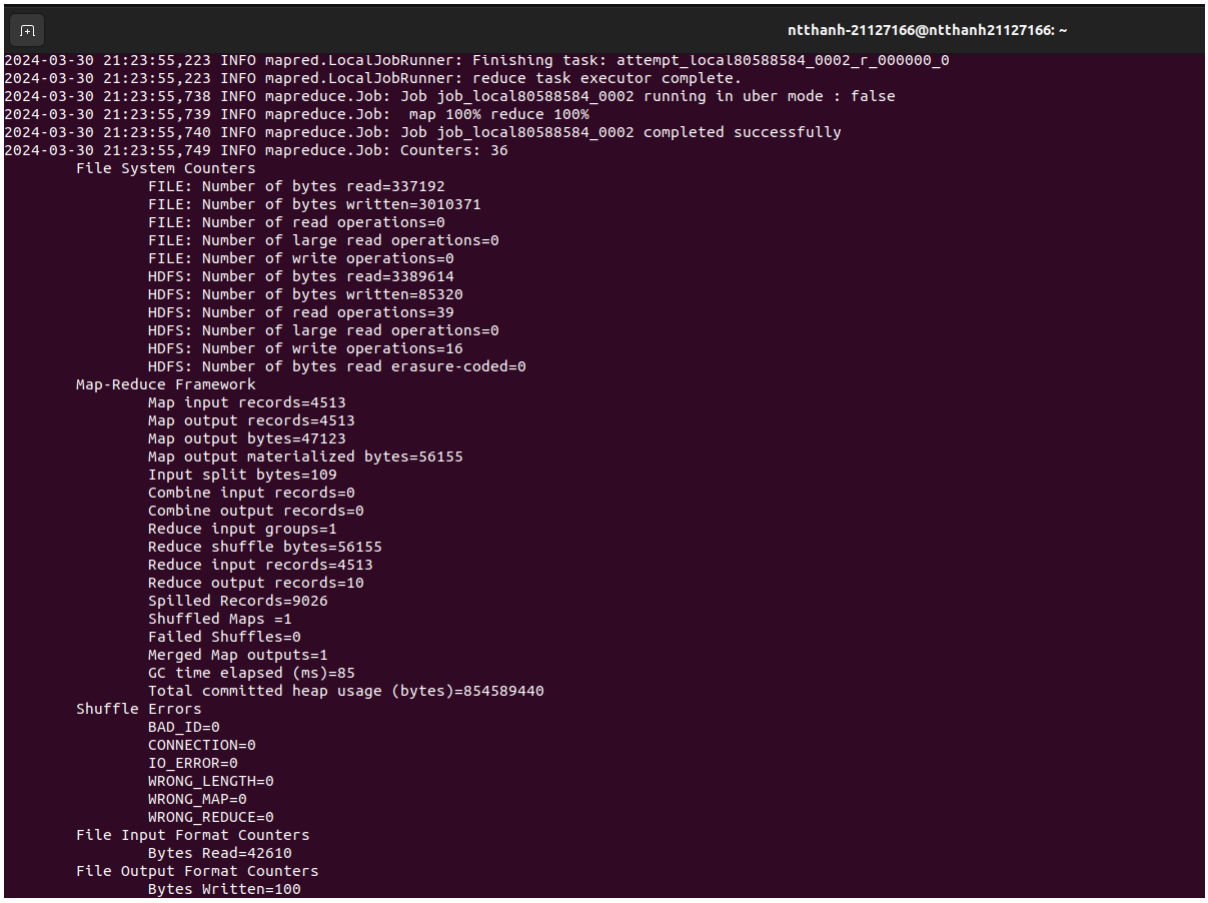

Fig II.8 MapRecuce Job of Task_1_3: Sorting and retrieving the top 10 terms with the highest term frequency.

Task 1.4:

How to run MapReduce job:

```
hadoop jar <path to jar file> <input file path> <output temp path>
<output directory path>
```

Logic of first Map function:

Tokenization and Initialization:

- The function starts by tokenizing the input value using a StringTokenizer, converting it into a sequence of tokens for processing.
- An integer variable index is initialized to keep track of the position of tokens in the input line.
- A string variable res is initialized to store the intermediate result as tokens are processed.

Processing Tokens: Within the loop, each token from the input line is iterated over.

At the start of the loop, the index is checked to determine the type of token:

- If the index is 0, it indicates the first token in the line, typically representing a term-document pair. This token is appended to the string res followed by an "@" symbol as a delimiter.

- If the index is 1, it signifies the second token in the line, usually representing the term. This token is set as the key (*pairKey*) for the emitted key-value pair.

- For tokens beyond index 1, they are concatenated to the string res, representing additional information associated with the term-document pair.

Setting Key-Value Pair: Once all tokens are processed, the value of res is set as the value (*pairValue*) for the emitted key-value pair.

Emitting Output: Finally, the function writes the key-value pair to the output context. The key (*pairKey*) represents the term extracted from the input line, while the value (*pairValue*) holds additional information associated with the term-document pair.

Logic of first Reducer function:

Processing Values: Within the loop, each value is processed. The values are split based on the delimiter "@" to extract the term ID and its corresponding frequency from the input string.

Aggregation: The reducer accumulates the total frequency of the term across all documents by adding up the individual frequencies.

Normalization: After aggregating the frequencies, the reducer calculates the term frequency for each document. It divides the frequency of the term in the current document by the total count of occurrences of the term across all documents.

Logic of second Map function:

Tokenization: Each line of text from the input is tokenized using a StringTokenizer. The StringTokenizer breaks the text into individual tokens delimited by whitespace.

Key-Value Pair Formation: Within the loop, the function iterates through the tokens extracted from the input text. It sets the first token encountered (at index 0) as the *pairKey*, which represents a term or document identifier.

Value Aggregation: The function concatenates the remaining tokens (starting from index 1) into a string, representing the values associated with the key. These values may

include term frequency or other relevant information.

Setting PairValue: Once all relevant tokens have been processed, the concatenated string representing the values is set as the *pairValue*.

Emitting Output: Finally, the function emits a key-value pair using the Context object. The *pairKey* serves as the key, and the *pairValue* serves as the corresponding value for that key.

Logic of second Reducer function:

Aggregation of Values: For each unique key, the function iterates over the associated values. These values are the term-frequency pairs generated by the previous map phase. Within this loop, the function increments term_docs to count the number of documents containing the current term. It also adds each value to *thislist*.

Calculation of TF-IDF:

- After aggregating the values, the function iterates over the list of term-frequency pairs stored in *thislist*.

- For each pair, it splits the string to extract the document ID and the term frequency.

- It calculates the inverse document frequency (IDF) using the formula: $idf = log(totalDocument$
where *totalDocuments* is the total number of documents in the corpus and term_docs is the number of documents containing the term.

- Then, it calculates the TF-IDF score for the term in the current document using the formula: $tfidf = tf * idf$, where **tf** is the term frequency.

- The result, along with the term and document ID, is stored in the result variable.

Emitting Output: For each term-document pair, the function writes a key-value pair to the output context. The key consists of the term, document ID, and TF-IDF score, while the value is set to an empty string.

Format of Keys and Values at each stage:

In the input map task, an <Object, Text> pair will be passed. The map task will then produce an intermediate <Text, Text> pair. Subsequently, a <Text, Iterable<Text» pair will be passed to the reduce task, which will return a final <Text, Text> pair.

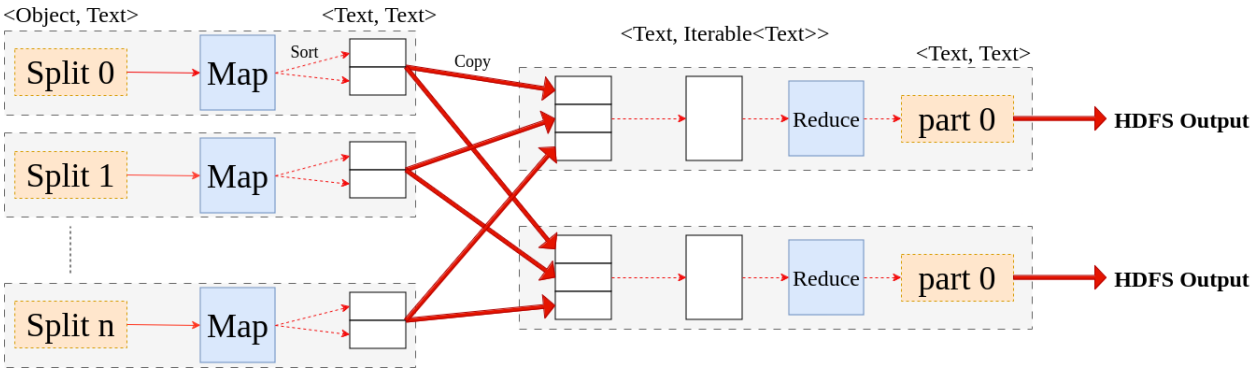

Fig II.9 Format of Keys and Values at each stage of first MapReduce Job in Task_1_4

In the input map, the second job will receive an <Object, Text> pair. The output of the map phase for the second job will yield an intermediate <Text, Text> pair. Subsequently, a <Text, Iterable<Text» pair will be passed to the reducer of the second job, which will then return a <Text, Text> pair.

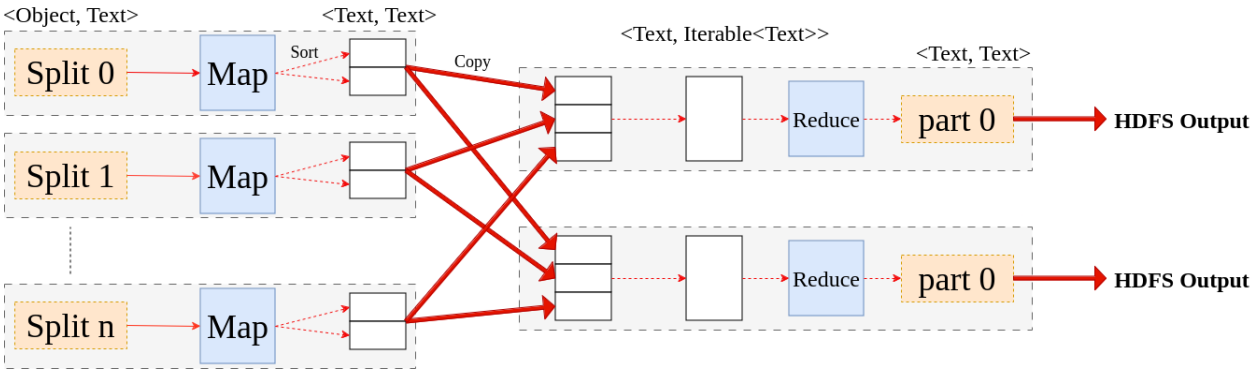

Fig II.10 Format of Keys and Values at each stage of second MapReduce Job in Task_1_4

Results following the execution of the MapReduce job:

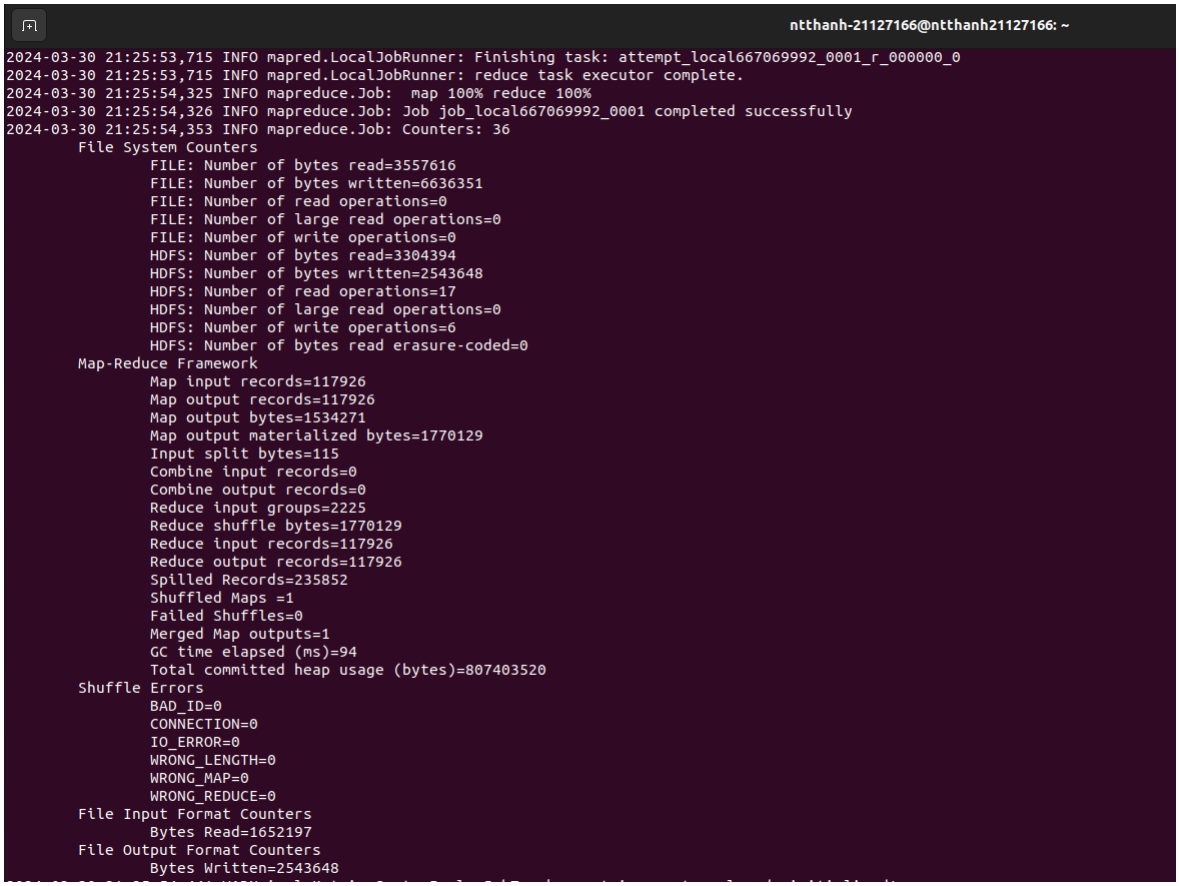

Fig II.11 MapReduce job in Task 1.4: Calculating Term Frequency of each term.

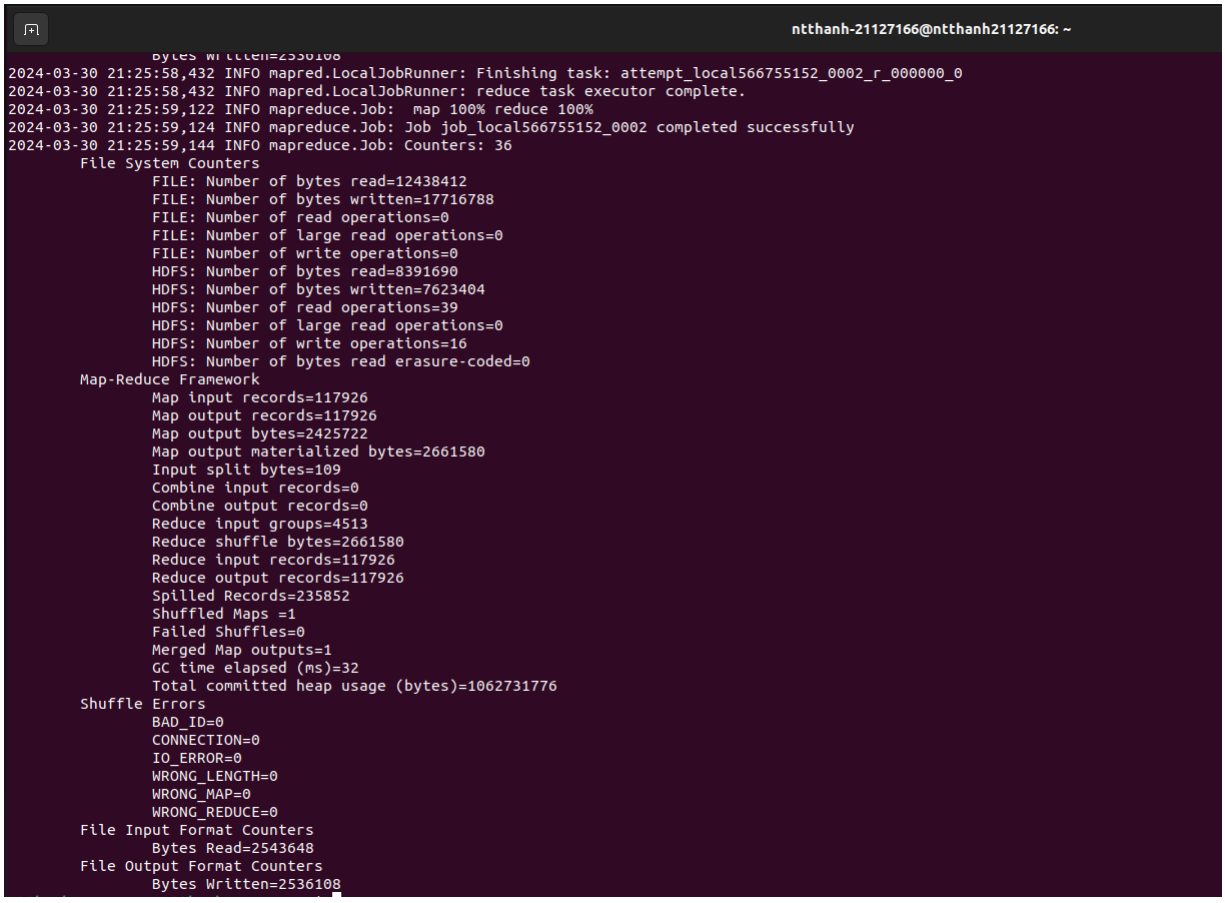

Fig II.12 MapReduce job in Task 1.4: Calculating Inverse Document Frequency and TFIDF of each term.

Task 1.5:

How to run MapReduce job:

```
hadoop jar <path to jar file> <input file path> <output directory path>
```

Logic of Map function:

Processing Input Values: The function tokenizes the input value using a StringTokenizer to extract relevant information. Each token represents an index or a value associated with the input data.

Index Retrieval and Value Assignment: The function retrieves the index for terms and documents from the tokenized input. It then retrieves the corresponding term and document values from the *termsid* and *docsid* maps, respectively.

Formatting Output: After obtaining the relevant term and document values, the func-

tion constructs a *pairKey* and a *pairValue*. The *pairKey* represents the document name, and the *pairValue* contains the term value, document index, and TF-IDF score concatenated together.

Logic of Reduce function:

Processing Intermediate Values: The function iterates over each intermediate value associated with the key. For each value:

- The value is split into tokens to extract the term, document, and TF-IDF score.
- The document identifier is added to the docSet.
- The TF-IDF score is aggregated into the termSum map, and the occurrence count is incremented in the termCount map.

Calculating Term Averages: After processing all intermediate values, the function calculates the average TF-IDF score for each term by dividing the sum by the count.

Sorting and Output Generation: The average TF-IDF scores are sorted in descending order. The top 5 terms are then selected based on these scores. The function constructs a string containing these top terms along with their average TF-IDF scores normalized by the total number of documents. This string is set as the value of *pairValue*, with the corresponding key being the input document's name.

Format of Keys and Values at each stage: The input map task receives a pair of <Object, Text>. The output of the map task is an intermediate pair of <Text, Text>. Subsequently, a pair of <Text, Iterable<Text» is passed to the reduce task, which then returns a pair of <Text, Text>.

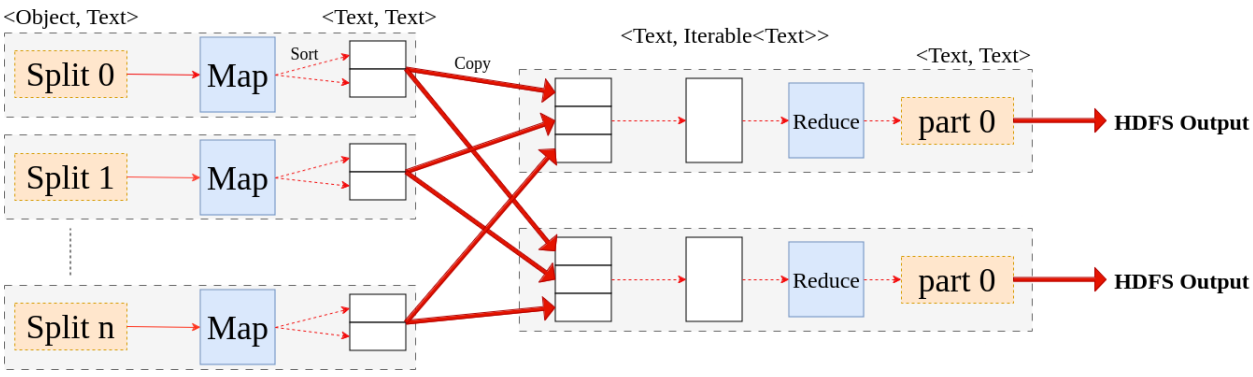

Fig II.13 Format of Keys and Values at each stage of MapReduce Job in Task_1_5

Results following the execution of the MapReduce job:

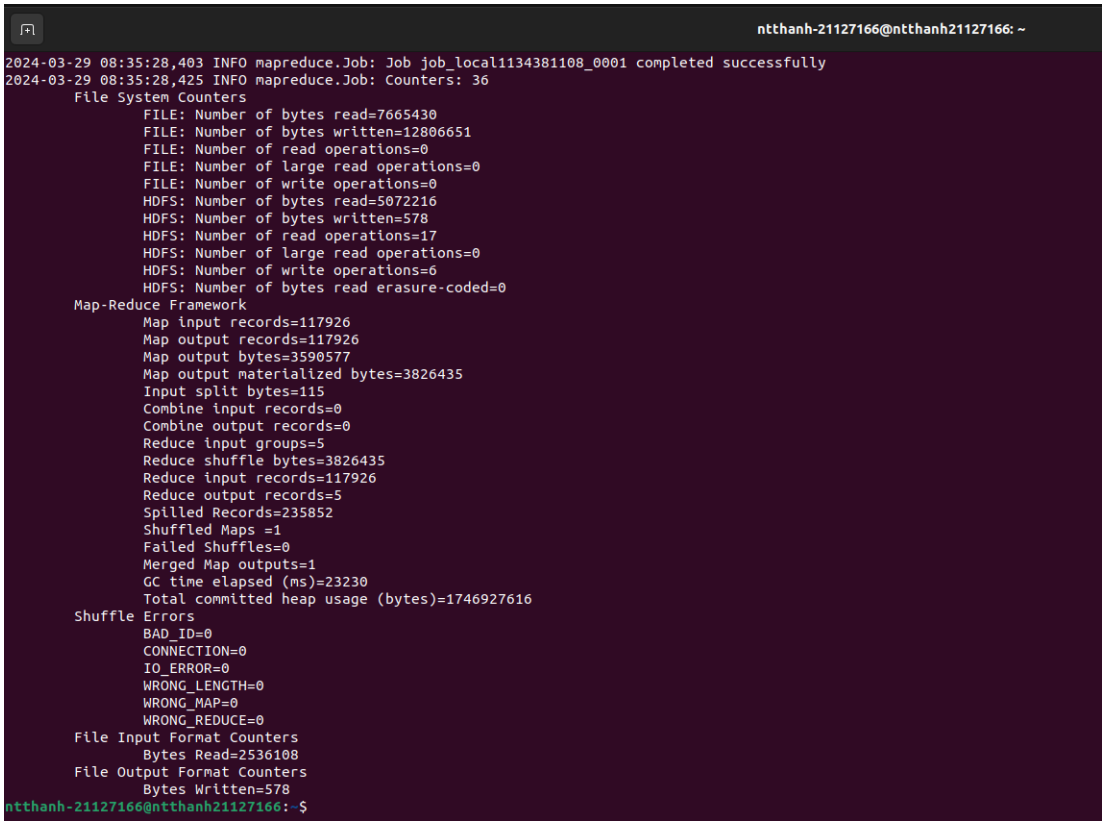

Fig II.14 MapReduce job in Task 1.5: Calculating the highest average tfidf

III. K-Means Algorithm

1. Data Description:

In this section, we are provided with a dataset named **2DPoints.csv**, which contains a set of points and their respective classes. Each point is defined by two real scalar values.

2. MapReduce Job Implementation

2.1. Overall design of MapReduce job

In this section, I will discuss the setup of each MapReduce task and its corresponding job. I'll provide details on the number of mappers and reducers utilized, as well as the inputs and outputs involved.

- **Task 2.1:** Implement the K-Means algorithm and run it with K=3. This task requires one input, the file **2DPoints.txt**, and produces two outputs: the **Task_2_1.clusters** folder and the **Task_2_1.classes** folder. It utilizes one mapper, two reducers, and one driver to carry out the MapReduce job.

- **Task 2.2:** Implement the K-Means algorithm, similar to Task 2.1, but use an appropriate distance metric for TF-IDF vectors, such as cosine similarity. The input should be the preprocessed file **Task_1_4.mtx**, and it should produce four outputs: the **Task_2_2.clusters** folder, the **Task_2_2.classes** folder, the **Task_2_2.txt** file, and the **Task_2_2.loss** file. This should be executed using three mappers, four reducers, and one driver to complete the MapReduce job.

- **Task 2.3:** Implement K-Means|| to initialize centroids for the K-Means algorithm and execute similarly to Task 2.2. The input should be the preprocessed file **Task_1_4.mtx**, and it should produce two outputs: the **Task_2_3.txt** file, and the **Task_2_3.loss** file. This should be executed using three mappers, four reducers, and one driver to complete the MapReduce job.

2.2. In-Depth Analysis of a MapReduce Job:

Task 2.1:

How to run MapReduce job:

```
hadoop jar <path to jar file> <input file path> <output clusters path>
<output classes path>
```

Logic of Map function:

getDistance() Function:

- This function calculates the Euclidean distance between two points given their coordinates (x1, y1) and (x2, y2).

- It utilizes the Euclidean distance formula: $\sqrt{(x2-x1)^2+(y2-y1)^2}$.

- The calculated distance is returned.

calculateCluster() Function:

- This function determines the closest cluster to a given point (x, y).

- It retrieves the maximum number of clusters from the configuration.

- It iterates over each cluster and calculates the distance between the point and the centroid of each cluster.

- The cluster with the shortest distance is selected as the nearest cluster for the point.

- The name of the nearest cluster is returned.

map() Function:

- This function is the mapper function that processes input data.

- It tokenizes each input line to extract the x and y coordinates of a point.

- It calls the calculateCluster() function to determine the closest cluster for the point.

- It emits a key-value pair, where the key is the name of the nearest cluster, and the value is the coordinates of the point (x, y).

Logic of Reduce function:

Aggregating Points:

- The reducer iterates over the values, which are the coordinates of points assigned to the cluster.

- It splits each value to extract the x and y coordinates of the point.

- It stores the x coordinates in the *xList ArrayList* and the y coordinates in the *yList ArrayList*.

Format of Keys and Values at each stage:

In the first job's input map, a pair of <Object, Text> will be received. This map will then generate an intermediate pair of <Text, Text>. Following this, a pair of <Text, Iterable<Text» will be passed to the reducer of the first job, which in turn will output a final pair of <Text, Text>.

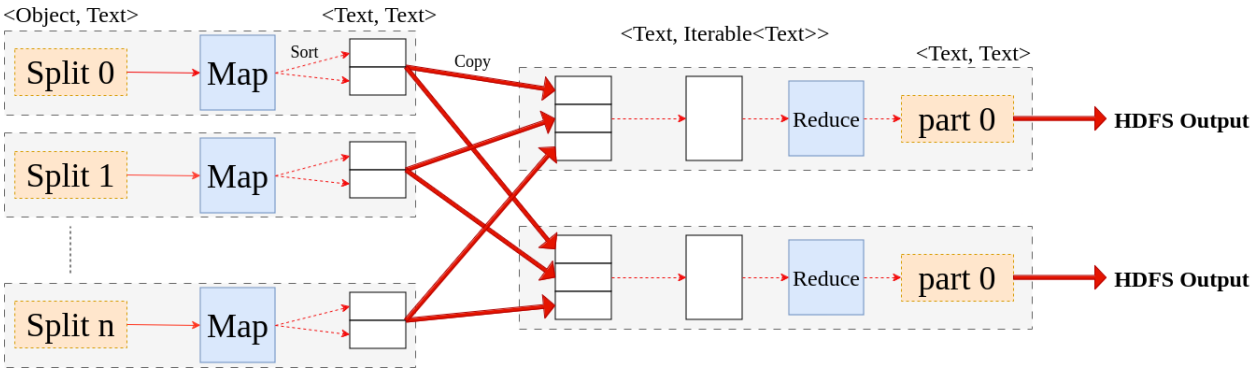

Fig III.1 Format of Keys and Values at each stage of first MapReduce Job in Task_2_1

In the input map, the second job receives a pair of <Object, Text>. The map phase's output for this job will be an intermediate pair of <Text, Text>. Following this, a pair of <Text, Iterable<Text» will be passed to the reducer of the second job, which will then produce a final pair of <Text, Text>.

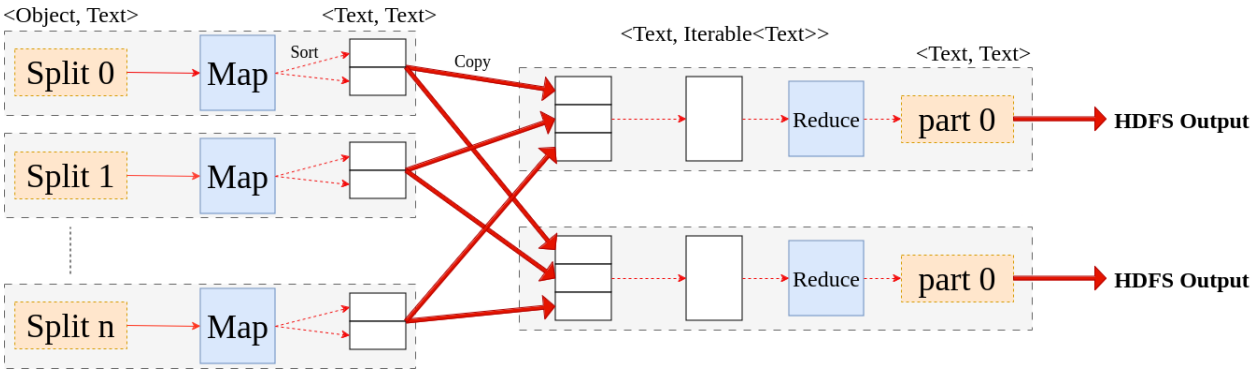

Fig III.2 Format of Keys and Values at each stage of second MapReduce Job in Task_2_1

Results following the execution of the MapReduce job

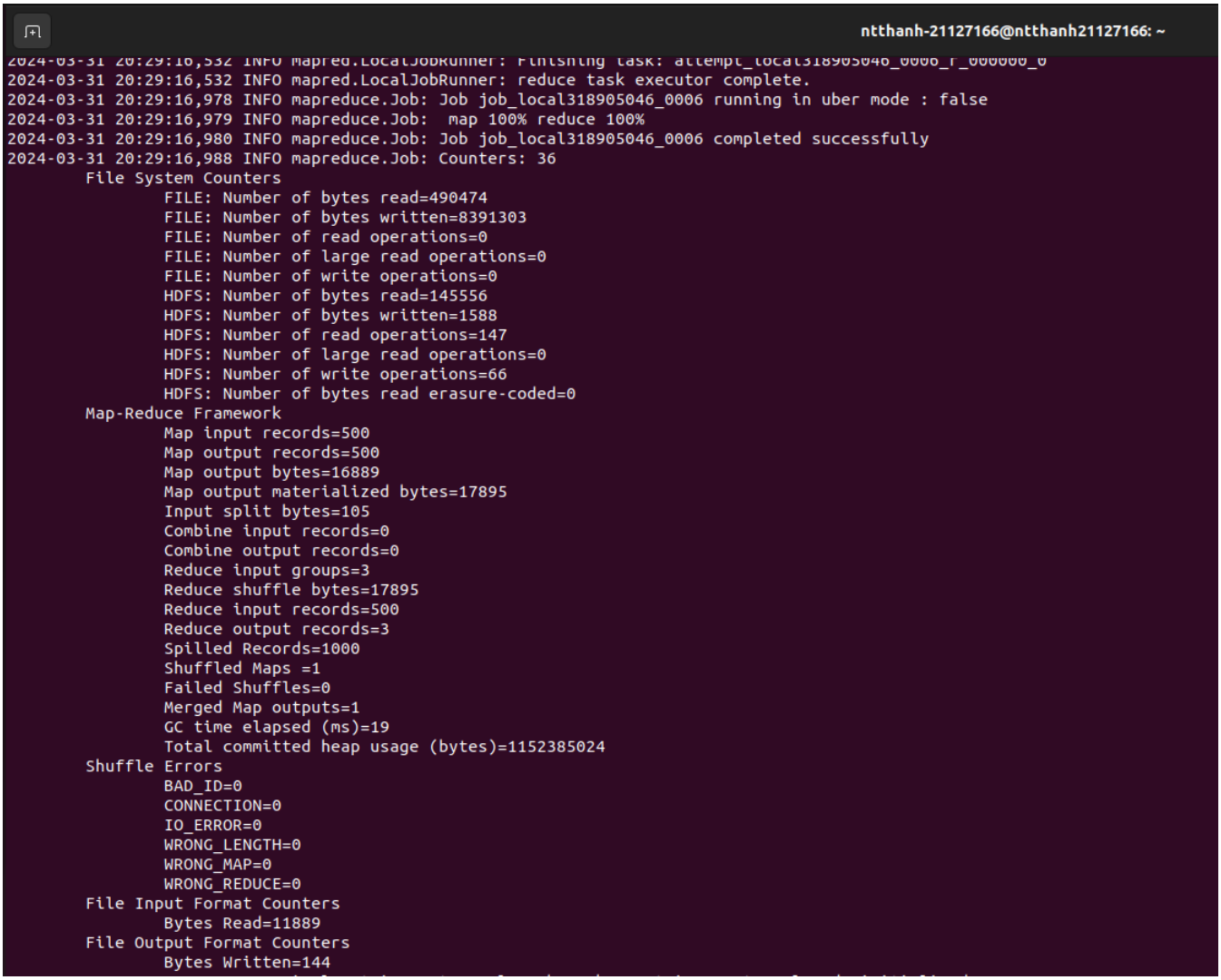

Fig III.3 MapRecuce Job of Task_2_1: Retrieving a list of clusters.

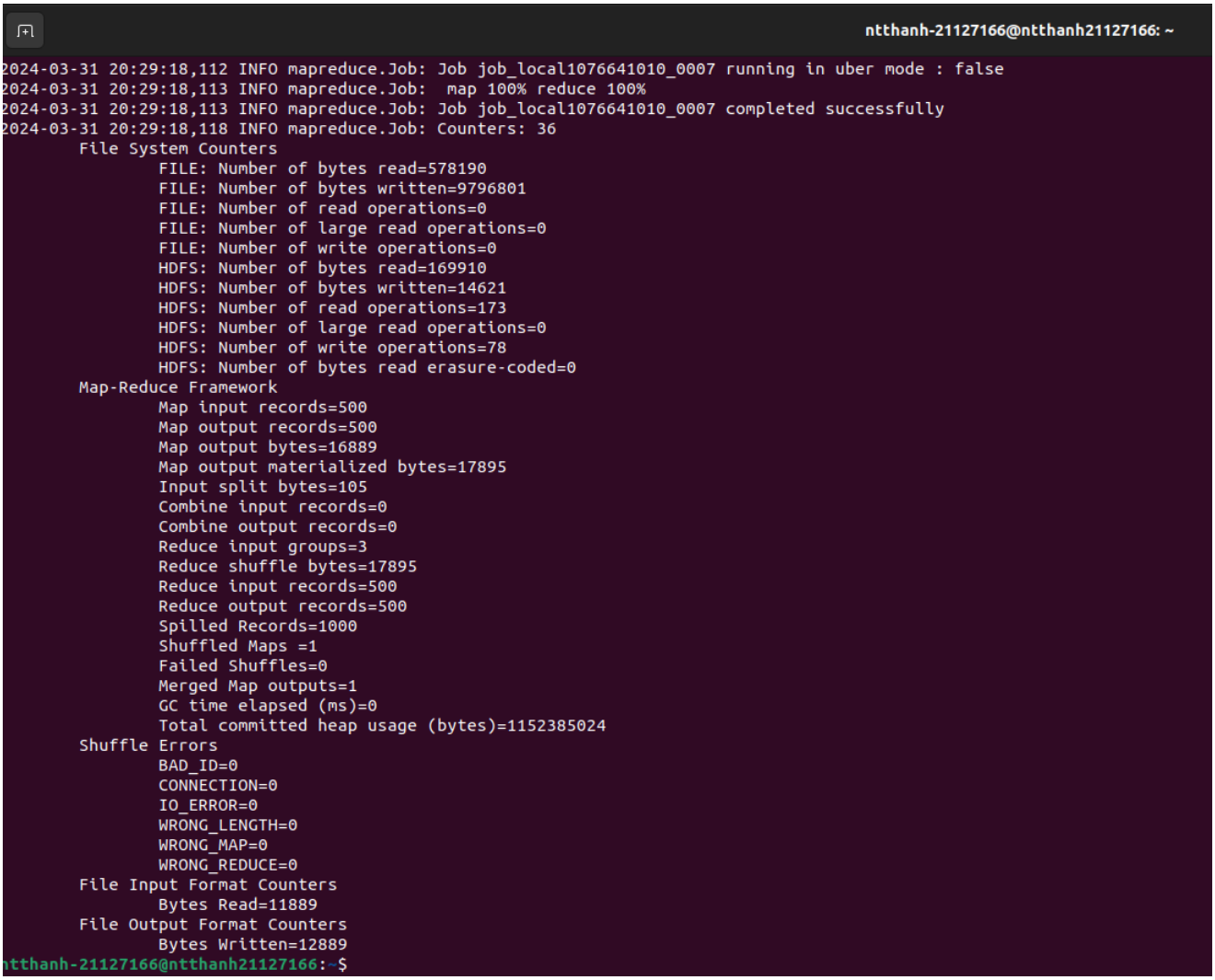

Fig III.4 MapRecuce Job of Task_2_1: Retrieving a list of data points following each cluster.

Task 2.2:

How to run MapReduce job:

```
hadoop jar <path to jar file> <input file path> <output clusters path>
<output classes path> <output loss path> <output sort path>
```

Logic of the first Map function:

Parsing Input: It parses the input value, splitting it into document ID and TF-IDF vector parts. The TF-IDF vector is stored as an array of doubles (tfidfVector), initialized with zeros.

Finding Closest Cluster:

- For each cluster, it retrieves the cluster center vector from the configuration and calculates the cosine similarity between the input point and the cluster center.

- The cluster with the minimum cosine distance (maximum similarity) is considered the closest.

Cosine Similarity Calculation: The *cosineSimilarity* method calculates the cosine similarity between two vectors using the dot product formula.The process iterates over each dimension of the vectors, computes the dot product, and calculates the vector norms. Finally, it returns the cosine similarity value.

Logic of the second Map function:

Parsing Input: It splits the input value into two parts: the cluster ID (clusterName) and the TF-IDF vector of the cluster centroid (tokens). The TF-IDF vector is parsed and stored as an array of doubles (tfidfVector).

Cluster Centroid Retrieval: It retrieves the cluster centroid vector from the job configuration based on the cluster name (*clusterName*). The centroid vector is split and stored in an array of doubles (centroid).

Squared Euclidean Distance Calculation:

- It iterates over each dimension of the TF-IDF vector and calculates the squared difference between the vector element and the corresponding centroid element.

- The squared differences are summed up to obtain the squared Euclidean distance (result).

Logic of the third Map function:

Parsing Input: The map function splits the input value into tokens based on whitespace.

Logic of the first Reduce function:

Input Processing: It iterates through the iterable values, counting the number of input points and storing their TF-IDF vectors in *thislist*.

Computing New Center:

- It creates a two-dimensional array named *points* to store the TF-IDF vectors for the input data points assigned to the cluster.

- The code parses each TF-IDF vector from *thislist*, converts it to a double array, and then stores it in *points*.

- It computes the mean of the TF-IDF vectors stored in points using the *computeMean* method.

computeMean() function: This method computes the mean of a list of vectors. It iterates over each dimension of the vectors, sums up the corresponding values, and divides by the total number of vectors to obtain the mean.

Logic of the second Reduce function:

Loss Calculation: The reducer iterates through the iterable values, summing up their numerical values (representing loss). It accumulates the sum of loss values into a variable (sum).

Logic of the third Reduce function:

Value Processing: The reducer iterates through the values associated with the key and adds them to a list of doubles (*valueList*). Each value is parsed from a comma-separated string into a double and added to the list.

Sorting: The values in *valueList* are sorted in descending order using *Collections.sort*.

Top N Selection: The reducer selects the top N values from the sorted list, where N is the minimum of 10 and the total number of values. It constructs a string (result) containing these top N values separated by commas.

Format of Keys and Values at each stage:

In the map task of the first job, an <Object, Text> input will be processed. The output from the map task of the first job will be an intermediate <Text, Text>. Subsequently, a <Text, Iterable<Text» will be passed to the reduce task of the first job, which will then return a <Text, Text>.

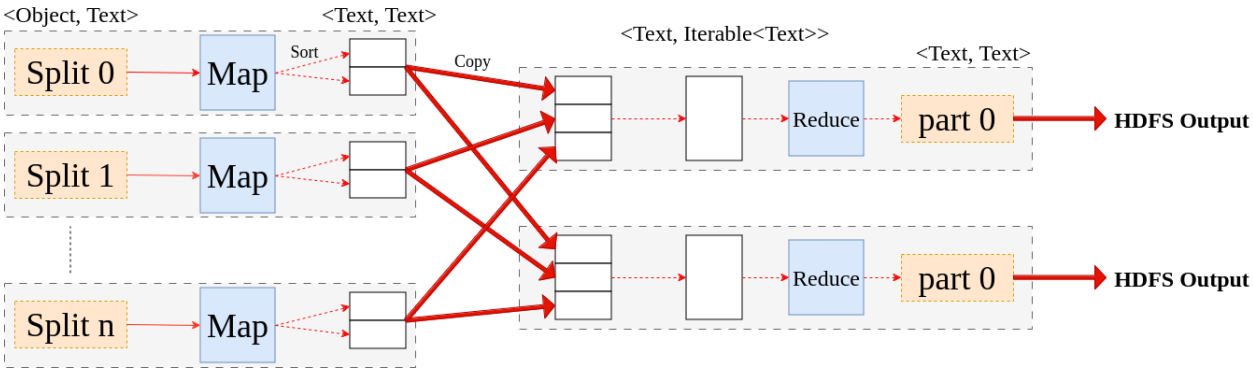

Fig III.5 Format of Keys and Values at each of the first MapReduce Job in Task_2_2

In the map task of the second job, an <Object, Text> input will be passed. The map task of the second job will then produce an intermediate <Text, Text> output. Subsequently, a <Text, Iterable<Text» will be passed to the reduce task of the second job, which will return a <Text, Text>.

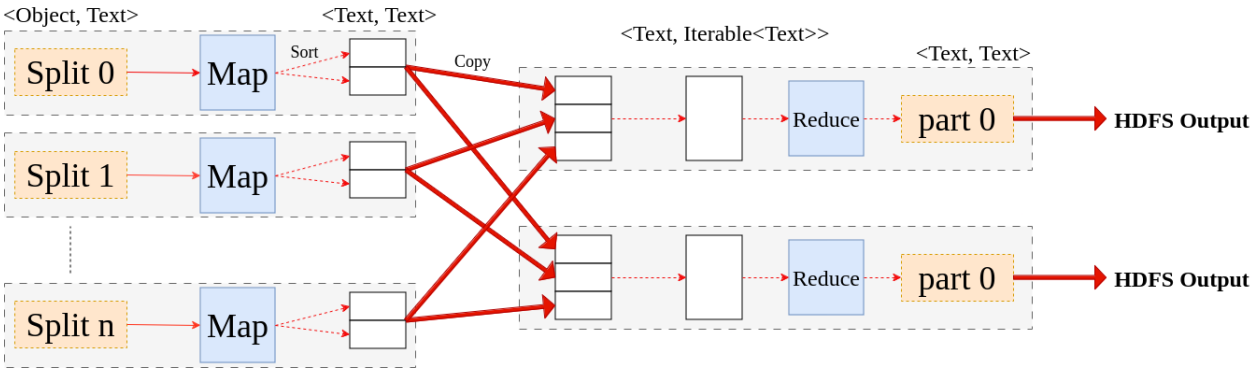

Fig III.6 Format of Keys and Values at each stage of the second MapReduce Job in Task_2_2

In the map task of the loss_job, an <Object, Text> input will be passed. The map task of the loss_job will then produce an intermediate <Text, Text> output. Subsequently, a <Text, Iterable<Text» will be passed to the reduce task of the loss_job, which will return a <Text, Text>.

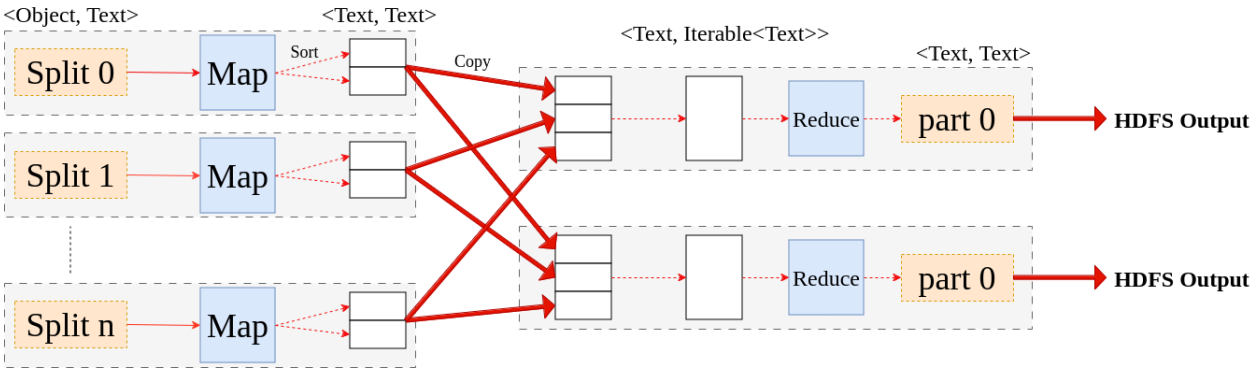

Fig III.7 Format of Keys and Values at each stage of the third MapReduce Job in Task_2_2

In the map task of the sort_job, an <Object, Text> input will be passed. The map task of the sort_job will then produce an intermediate <Text, Text> output. Subsequently, a <Text, Iterable<Text» will be passed to the reduce task of the sort_job, which will return a <Text, Text>.

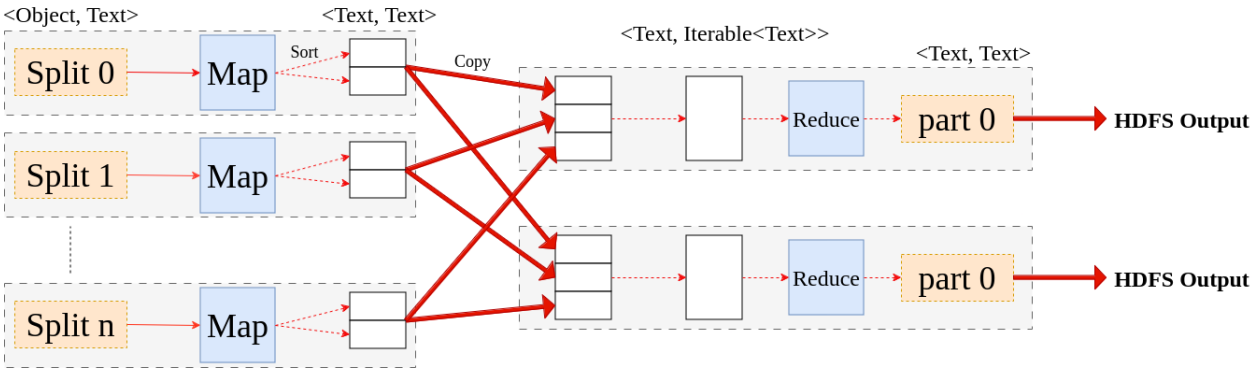

Fig III.8 Format of Keys and Values at each stage of the fourth MapReduce Job in Task_2_2

Results following the execution of the MapReduce job:

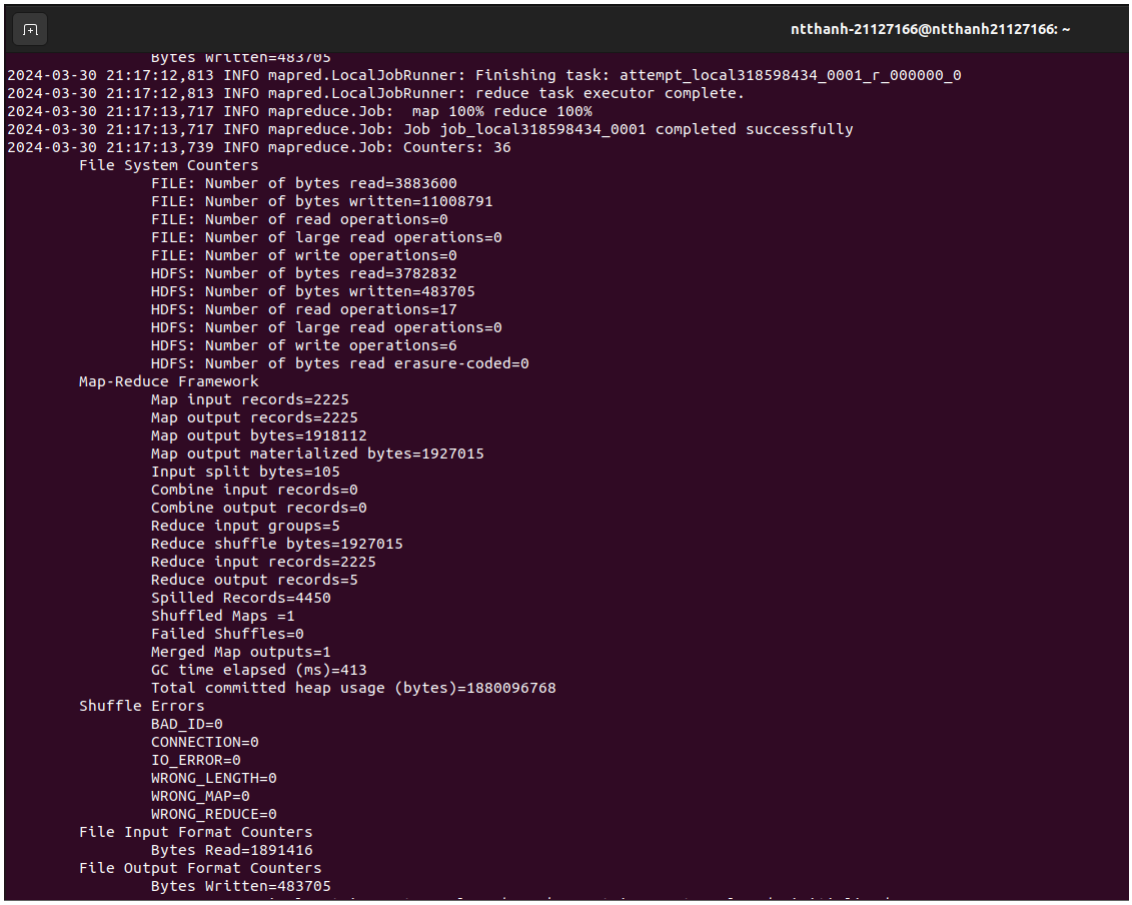

Fig III.9 MapRecuce Job of Task_2_2: Retrieving a list of clusters after using K-Means in TFIDF data.

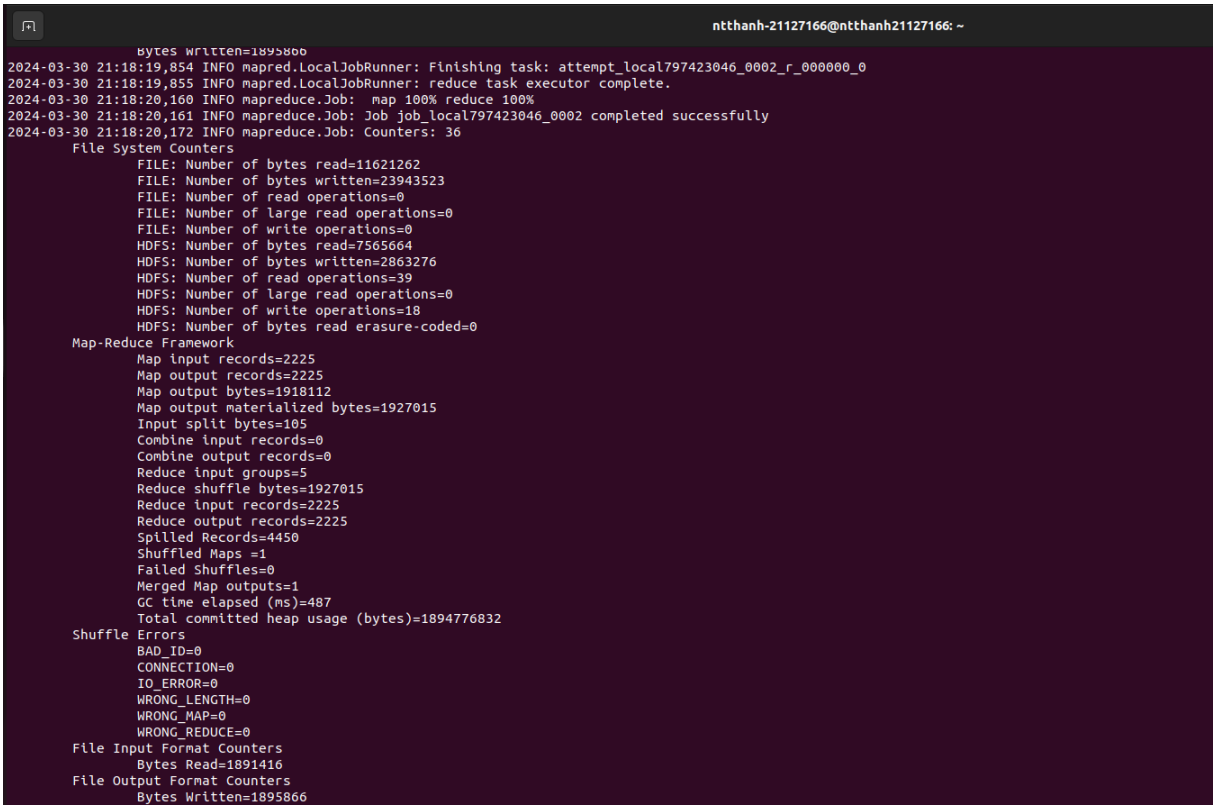

Fig III.10 MapRecuce Job of Task_2_2: Retrieve the data point subsequent to each cluster.

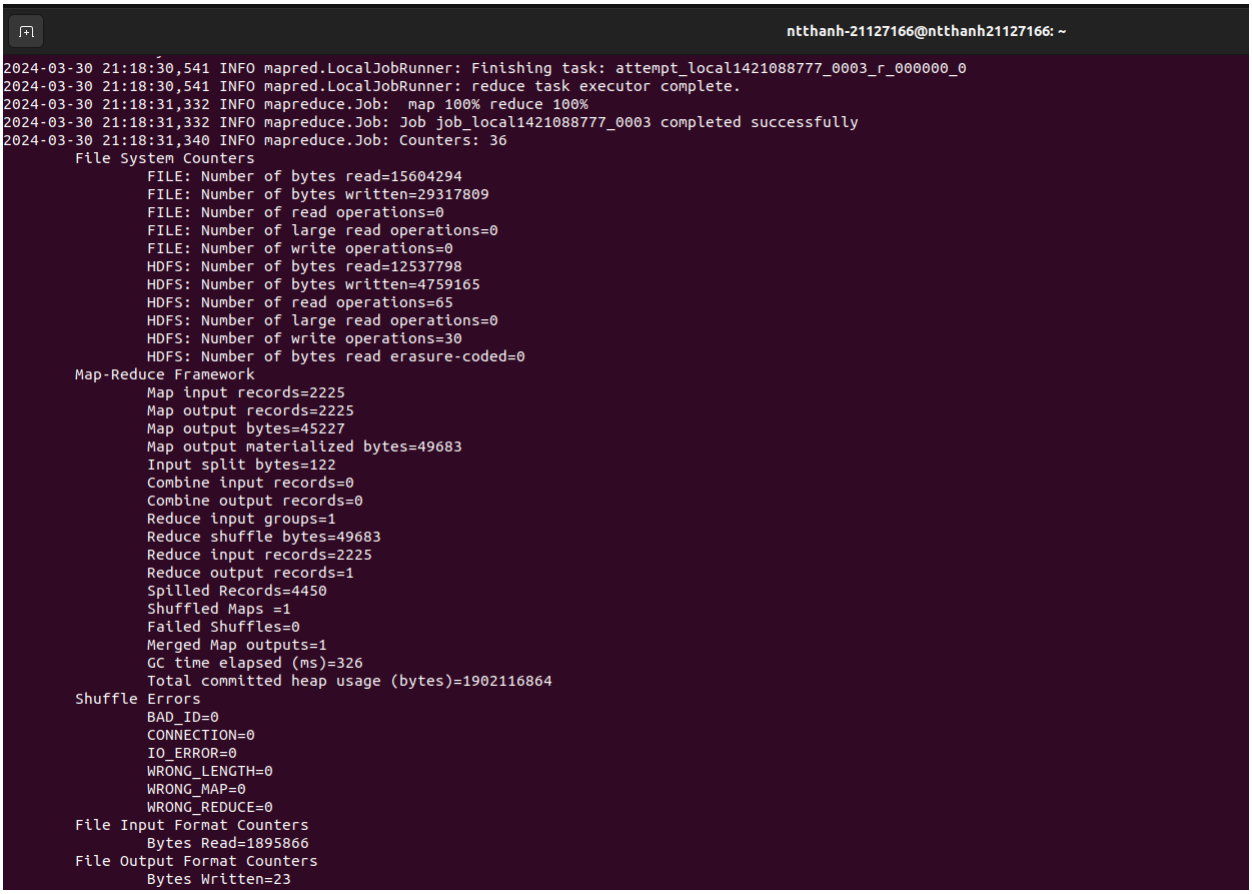

Fig III.11 MapRecuce Job of Task_2_2: Calculating the loss with each iteration of K-Means.

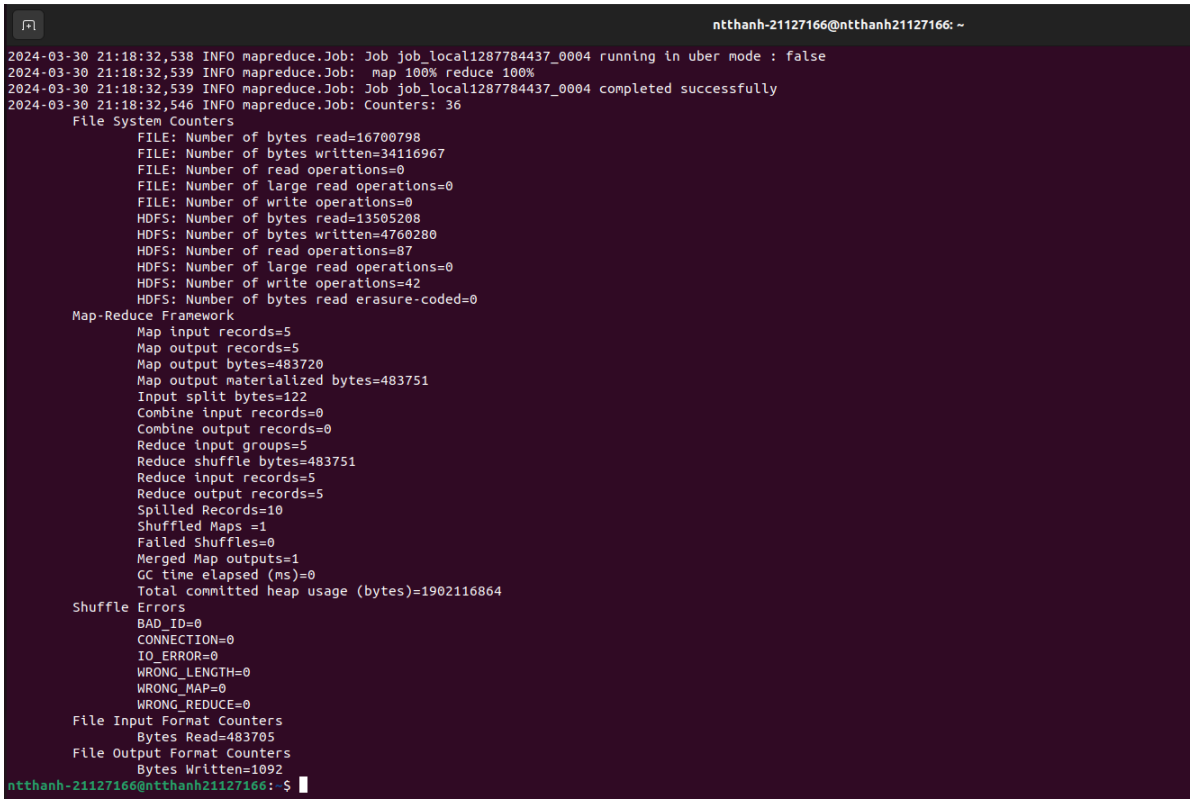

Fig III.12 MapRecuce Job of Task_2_2: Sorting and getting top 10 value in each cluster.

Task 2.3:

How to run MapReduce job:

```
hadoop jar <path to jar file> <input file path> <output clusters path>
<output classes path> <output loss path> <output sort path>
```

Logic of the first Map function

- The class declares two static variables *closestCenter* and point, each being an object of the Text class. *closestCenter* is used to store the name of the nearest cluster, while *point* is used to store the text value.

- In the map override function, input is being parse by split first, depending on the symbol | and comma symbol with each result above.

- From the processed data in the parse part, this function try to find the closest center for each point.

- The cosine similarity is count later in this class.

Logic of the second Map function TF-IDF Vector Construction:

- The TF-IDF values are parsed and stored in the tfidfVector array, which is initialized

to hold 10,000 elements, indicating the maximum possible number of terms.

- Each element of tfidfVector corresponds to a term index, and its value is the TF-IDF score for that term. If a term is not present in the input, its TF-IDF value is set to 0.

Cluster Centroid Retrieval:

- The cluster centroid values are obtained from the Hadoop configuration (conf) using the clusterName as the key.

- The centroid values are split into tokens and stored in the centroid array.

Similarity Calculation:

- The Euclidean distance between the TF-IDF vector and the centroid vector is calculated.

- For each term index, the squared difference between the TF-IDF score of the document and the corresponding centroid value is computed and accumulated into the result variable.

Emitting Intermediate Key-Value Pair: The computed distance (result) is written to the context as a key-value pair. An empty string is used as the key, and the distance value is converted to a string before being written.

Logic of the third Map funciton

- This function just simply split the input, using whitespace as the delimiter (including space, tab, newline, etc.).

Logic of the first Reduce function Centroid Calculation:

- The method initializes an empty list *thislist* to store the values (distances) associated with the current key (cluster).

- It iterates over the values, counting the number of documents (count) associated with the cluster and storing the distances in *thislist*.

Vector Construction:

- For each distance value in *thislist*, the method reconstructs the TF-IDF vector associated with the document.

- The TF-IDF values are extracted and stored in a 2D array points, where each row represents a document and each column represents a term's TF-IDF score.

Computing New Center:

- The method invokes the computeMean function to calculate the mean of the TF-IDF vectors associated with the documents in the cluster.

- The mean vector represents the new centroid for the cluster.

Emitting New Cluster Center:

- The newly computed centroid vector is converted to a string format and set as the value for the cluster key.

- The key-value pair representing the cluster identifier and its new centroid is written to the context.

Compute Mean Function: The computeMean function calculates the mean of a list of vectors (TF-IDF vectors in this case) by summing up the values for each dimension and dividing by the total number of vectors.

Logic of the second Reduce function Loss Calculation: The method initializes a variable sum to accumulate the total loss for the cluster represented by the current key. It iterates over the values, parsing each value as a double and adding it to the sum. These values represent the distances between documents and the cluster centroids, and summing them up provides a measure of the overall loss for the cluster.

Logic of the third Reduce function Value Aggregation: The method initializes an empty list valueList to store the numerical values associated with the current key. It iterates over the values, parsing each value as a double and adding it to the valueList.

Sorting: Once all values are collected, the method sorts them in descending order using Collections.sort() with Collections.reverseOrder() comparator.

Top N Selection: After sorting, the method selects the top 10 values (or less if there are fewer than 10 values) from the sorted list. The selected values are concatenated into a single string, separated by commas.

Emitting Result: Finally, the method writes a key-value pair to the context. The key remains unchanged, representing the category or cluster identifier. The value contains the top 10 (or less) sorted numerical values associated with the key, formatted as a comma-separated string.

Logic of the fourth Reduce function:

Cluster Extraction: Within the method, the cluster identifier is extracted from the input key. The key is converted to a string and split using the underscore character (_) as the delimiter. The cluster identifier is obtained from the second part of the split.

Output Generation: For each value in the values iterable, representing a document assigned to the cluster, a new key-value pair is emitted. The new key is set as the extracted

cluster identifier. The original value associated with the document is retained as the value in the emitted key-value pair.

Output Writing: After processing all values associated with the key, the method writes the generated key-value pairs to the context.

Format of Keys and Values at each stage:

In the map task of the first job, an <Object, Text> input will be processed. The output from the map task of the first job will be an intermediate <Text, Text>. Subsequently, a <Text, Iterable<Text» will be passed to the reduce task of the first job, which will then return a <Text, Text>.

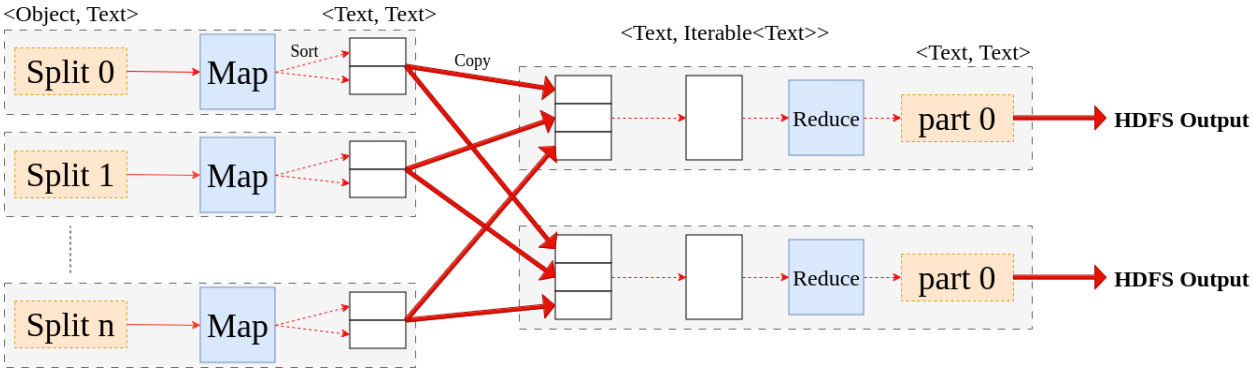

Fig III.13 Format of Keys and Values at each stage the first MapReduce Job in Task_2_3

In the map task of the second job, an <Object, Text> input will be passed. The map task of the second job will then produce an intermediate <Text, Text> output. Subsequently, a <Text, Iterable<Text» will be passed to the reduce task of the second job, which will return a <Text, Text>.

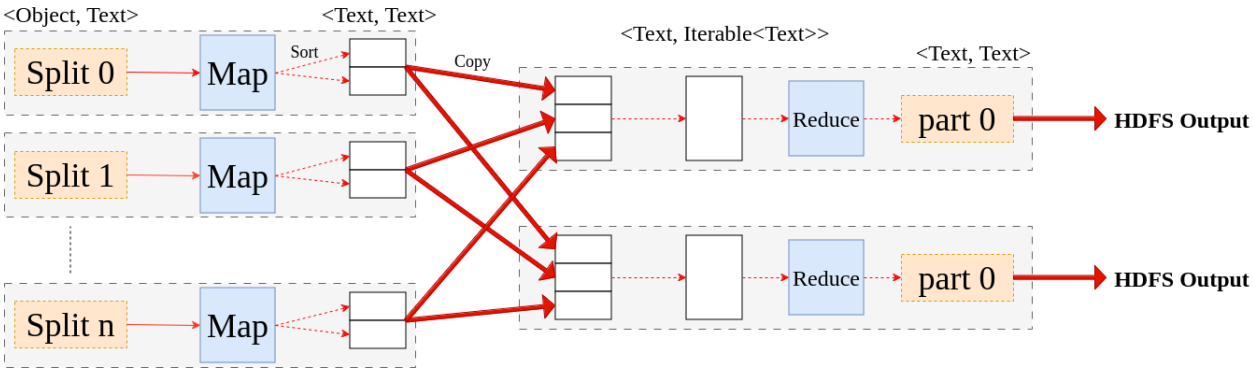

Fig III.14 Format of Keys and Values at each stage of the second MapReduce Job in Task_2_3

In the map task of the loss_job, an <Object, Text> input will be passed. The map task of the loss_job will then produce an intermediate <Text, Text> output. Subsequently, a <Text, Iterable<Text» will be passed to the reduce task of the loss_job, which will return a <Text, Text>.

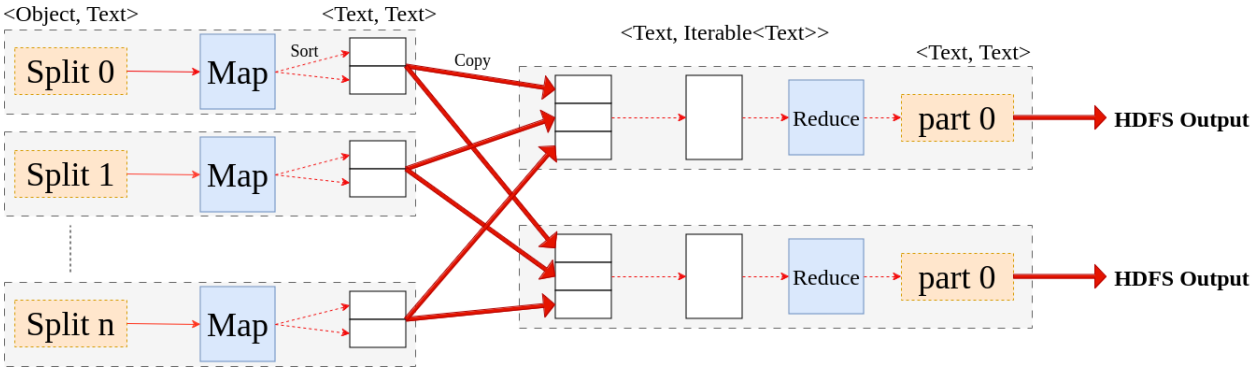

Fig III.15 Format of Keys and Values at each stage of the third MapReduce Job in Task_2_3

In the map task of the sort_job, an <Object, Text> input will be passed. The map task of the sort_job will then produce an intermediate <Text, Text> output. Subsequently, a <Text, Iterable<Text» will be passed to the reduce task of the sort_job, which will return a <Text, Text>.

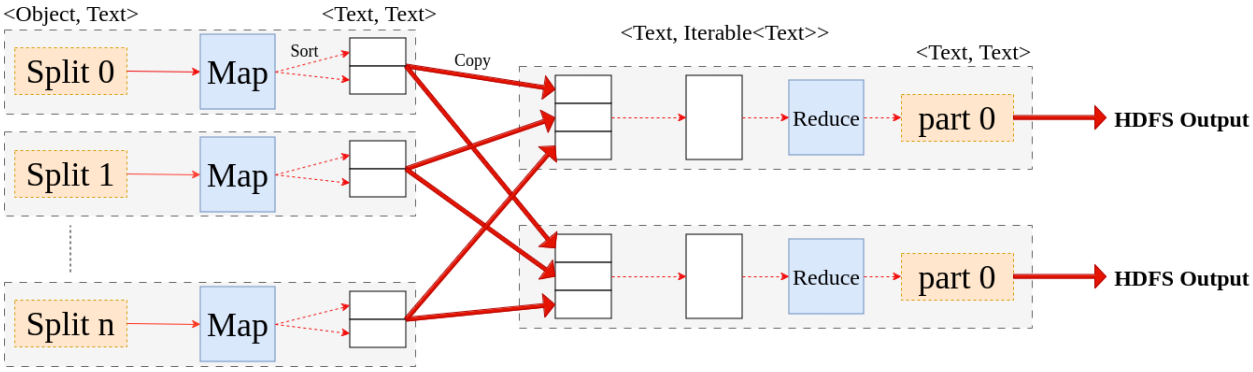

Fig III.16 Format of Keys and Values at each stage of the fourth MapReduce Job in Task_2_3

Results following the execution of the MapReduce job:

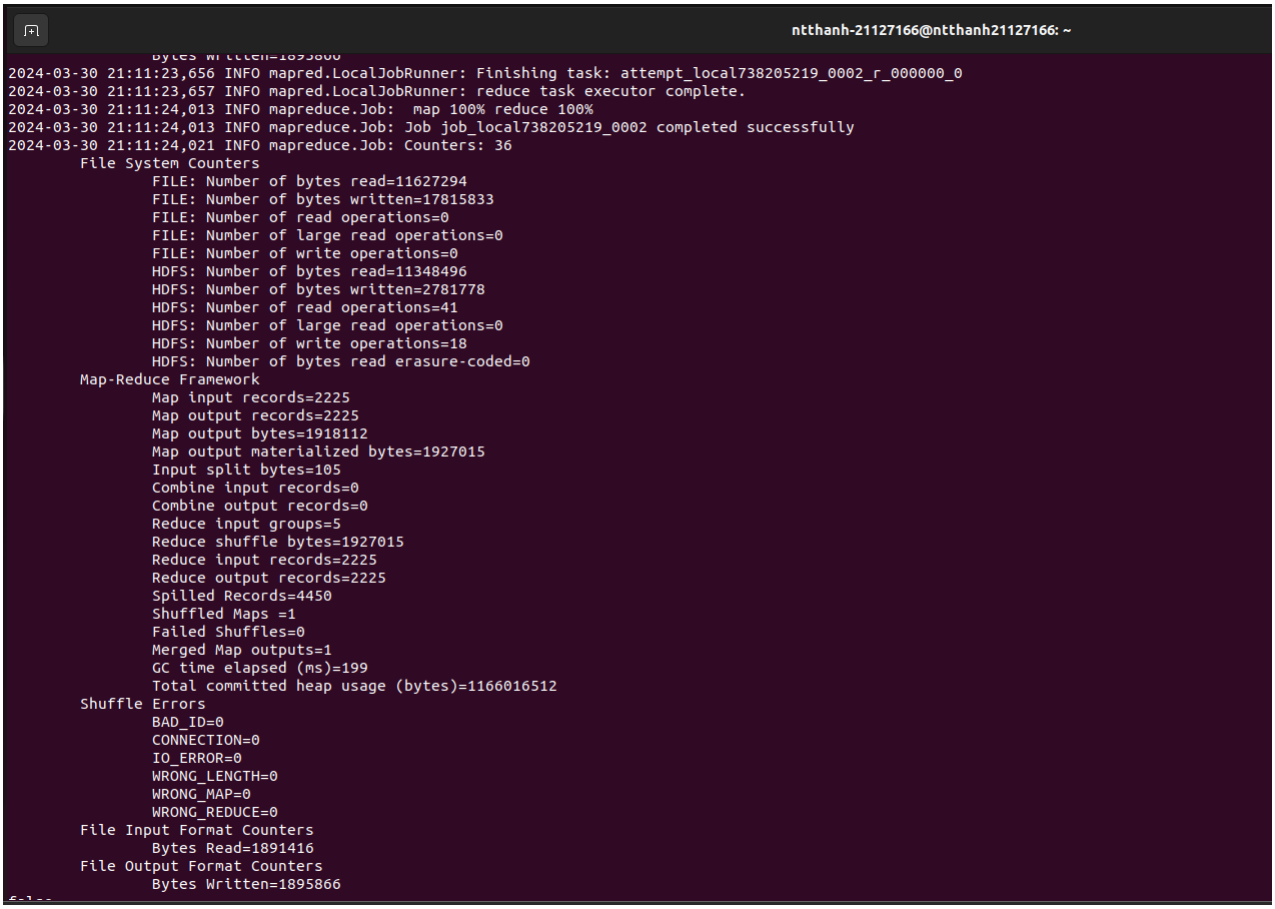

Fig III.17 MapRecuce Job of Task_2_3: Retrieving a list of clusters after using K-Means in TFIDF data.

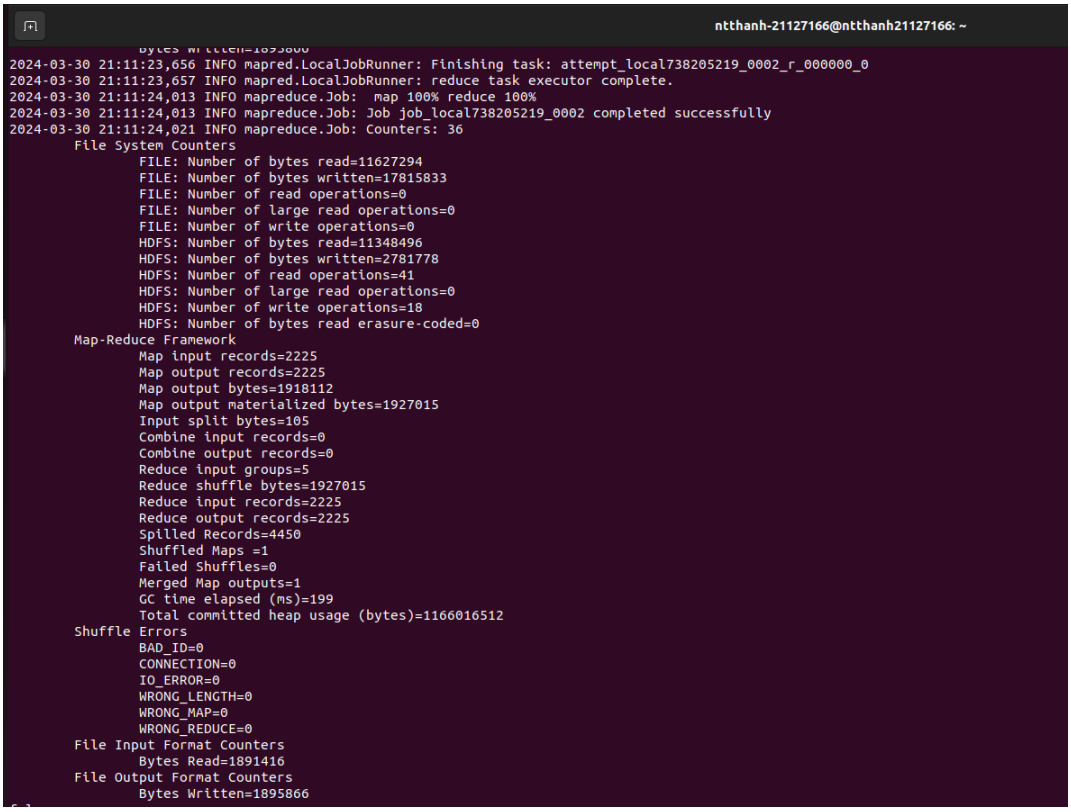

Fig III.18 MapRecuce Job of Task_2_3: Retrieve the data point subsequent to each cluster.

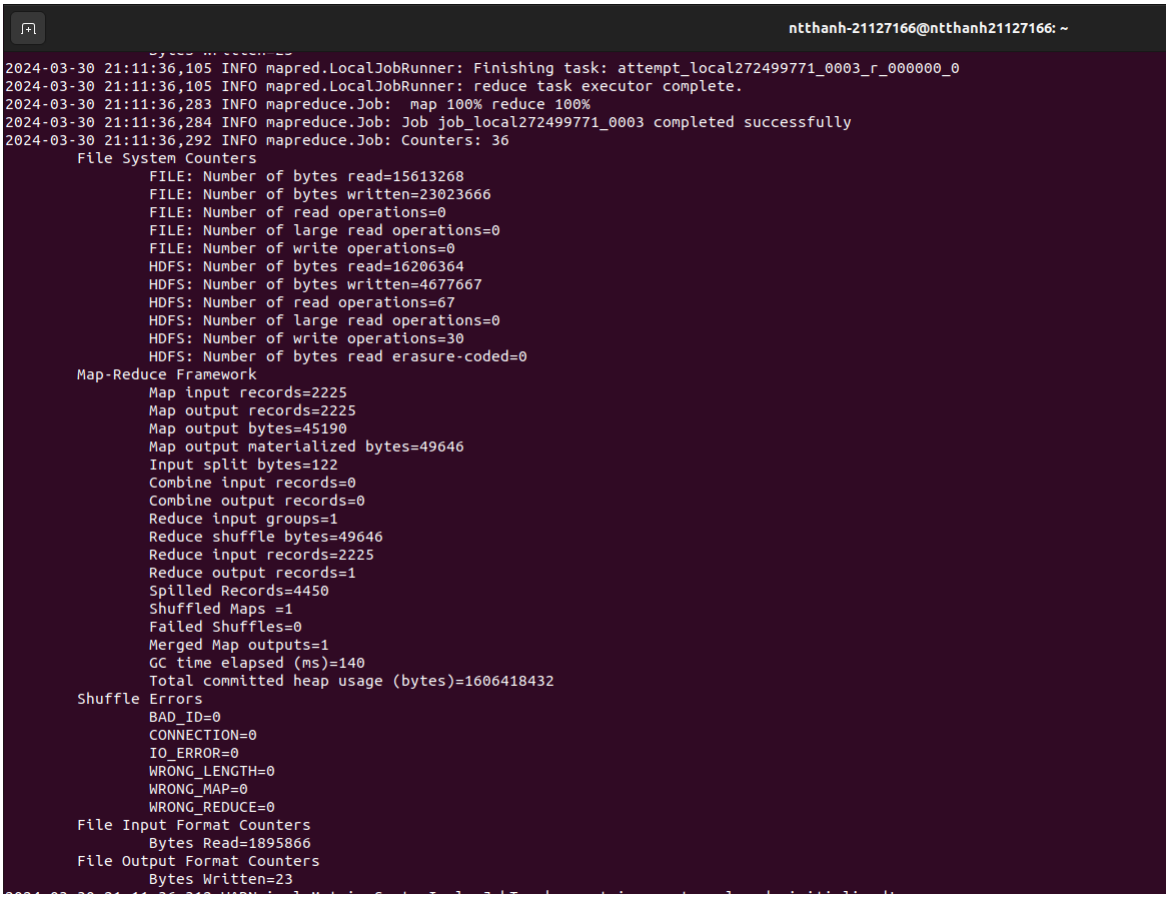

Fig III.19 MapRecuce Job of Task_2_3: Calculating the loss with each iteration of K-Means.

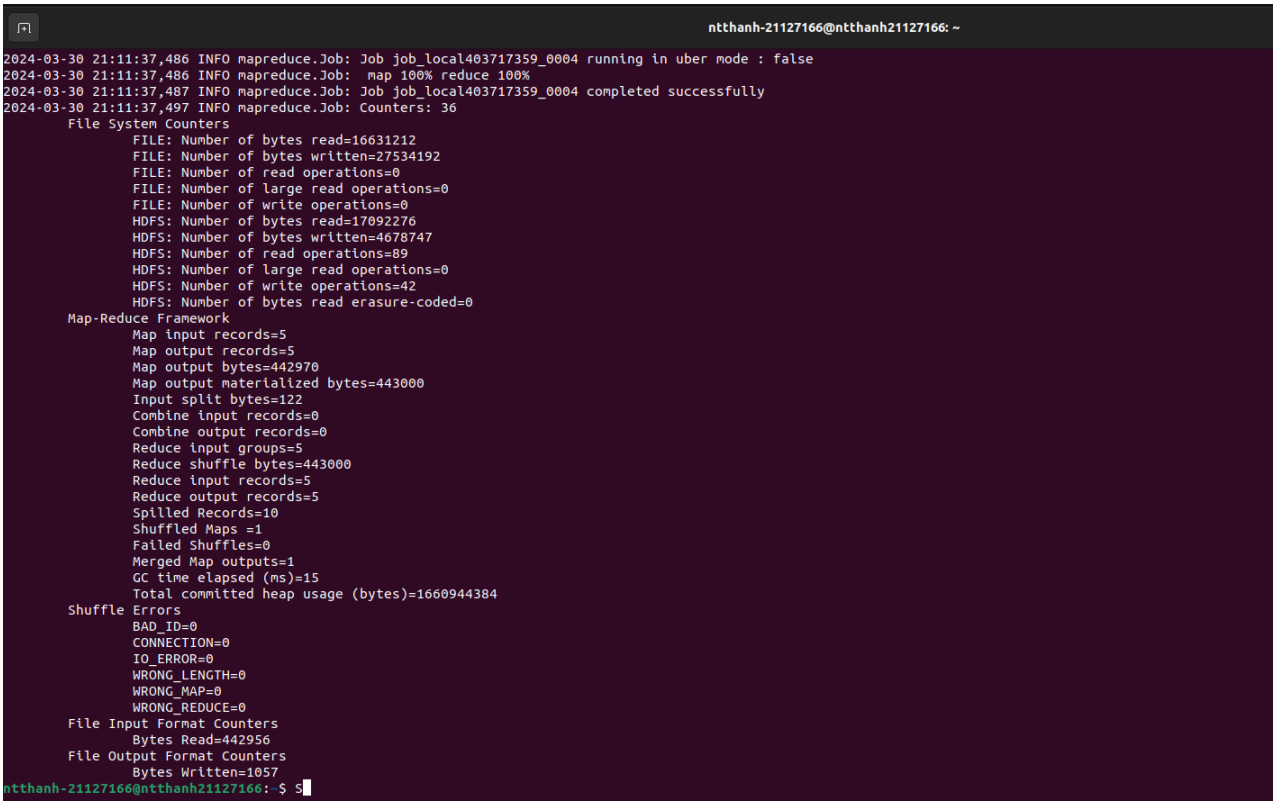

Fig III.20 MapRecuce Job of Task_2_3: Sorting and getting top 10 value in each cluster.