VIETNAM NATIONAL UNIVERSITY

HO CHI MINH UNIVERSITY OF SCIENCE

Course: CSC14118 - INTRODUCTION TO BIG DATA

Lab 02: Document Clustering with Hadoop MapReduce

Group: 21KHDL08

Members: Hồ Đinh Duy Lực - 21127351

Tô Khánh Linh - 21127638

Đoàn Thị Yến Nhi - 21127660

Lê Mỹ Khánh Quỳnh - 21127681

Class: 21KHDL

Instructors: Mr. Lê Ngọc Thành

Ms. Nguyễn Ngọc Thảo

Mr. Bùi Huỳnh Trung Nam

Mr. Đỗ Trọng Lễ

August 2024, Ho Chi Minh City

TABLE OF CONTENT

1. Group information

Student ID	Fullname	Group ID
21127638	Tô Khánh Linh	KHDL- 08
21127660	Đoàn Thị Yến Nhi	
21127351	Hồ Đinh Duy Lực	
21127681	Lê Mỹ Khánh Quỳnh	

2. Tasks Evaluation

Task		Completed
Part 1: Data Preprocessing	1.1. Text Cleaning and Term Frequency	100%
	1.2. Low-Frequency Term Elimination	100%
	1.3. Top 10 Most Frequent Words	100%
	1.4. TF-IDF	100%
	1.5. Highest average TF-IDF	100%
Part 2: K-Means Algorithm	2.1. K-Means on 2D Data	100%
	2.2. K-Means on Preprocessed Data	100%
	2.3. Scalable K-Means++ Initialization	100%

3. Part 1: Data Preprocessing

3.1. Task 1.1 Text Cleaning and Term Frequency

a. Data Description: In this section, we use the following data files:

- The input consists of files organized in five folders within the unprocessed bbc_dataset directory. These folders are named business, entertainment, politics, sport, and tech. Each folder contains multiple .txt files, sequentially labeled from 001.txt to N.txt.
- There is also a stopwords file (stopwords.txt) used to filter out stop words from the text files.
- Additionally, the processing code requires reading two files: bbc.terms and bbc.docs, for the following purposes:
    - bbc.terms: This file is used to assign an ID to each term. After removing stop words, each term is given an ID based on its line number.
    - bbc.docs: This file is used to assign an ID to each text file within a specific folder, with each file's ID corresponding to its line number.

b. MapReduce Job Implementation

In this task, we use **1 Mapper** phase and **1 Reducer** phase. Specifically, the execution steps are as follows:

- **In Mapper phase:**
    - This mapper processes each line of a text file, cleans and filters the words (remove stop words and regex), and then outputs key-value pairs where the key is a combination of the term ID and the document ID, and the value is always 1.

Step 1: Before map (set up):

- During the map phase, before mapping the data, the code will read and process auxiliary files as follows:

- For the stopwords file, it will extract words from the stopwords file and add them to a string set.
- For the bbc.terms, it will extract the words from the file and read it into a HashMap<String, Interger> which String indicates the word in bbc.term, Interger indicates the line of that word.
- For the bbc.docs, it will extract the documents id from the file into to into a HashMap<String, Interger> String is the name of the file and Interger indicates the line of that file.

Step 2: Mapping:

- **File Identification**: The mapper retrieves the file path of the current input split using context.getInputSplit(). Then it extracts the file name (fileTail) and its parent directory name (fileItem[len-2]), combining them to form a fileId (e.g folder.filename).
- **Skipping README Files**: If the file name is "README", the mapper skips by returning early.
- **Tokenizing and Processing the Text**: Each word is cleaned by removing special characters and converting it to lowercase using the ***CleanTheText*** method. The cleaned word is then checked against a list of stopwords and its length, if it's not a stopword and has a length greater than zero, it's can be process further
- **Handling the Word**: The cleaned word is passed to ***HandleWord***, which checks if it exists in the termId dictionary (a mapping of words to term IDs). If the word exists in termId, its corresponding term ID (line of that word) is returned. If it doesn’t exist, 0 is returned.
- **Emitting Key-Value Pairs:** If the word has a valid term ID (checkTermNumber != 0), the mapper creates a key by combining the term ID and fileId.
  - **Key:** checkTermNumber (the ID of the word) + " " + fileId (the ID of the file which is folder.filename (e.g bussiness.001))

- **Value:** 1
- Example output: [K,V] [1 business.001, 1]

```java
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

      String PreClean;
      String fileInput = ((FileSplit) context.getInputSplit()).getPath().toString();
      String[] fileItem = fileInput.split("/");
      int len = fileItem.length;
      String fileTail = fileItem[len-1].split("\\.")[0];

            if(fileTail.equals("README")) { return;}

      String fileId = fileItem[len-2]+"."+fileTail;
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        PreClean = itr.nextToken();
        String Cleantext = CleanTheText(PreClean);
        if (!stopWords.contains(Cleantext) && Cleantext.length() > 0){
            int checkTermNumber = HandleWord(Cleantext);
            if(checkTermNumber !=0){
              word.set(checkTermNumber + " " + fileId);
              context.write(word, one);
            }
        }
      }
    }
  }
```

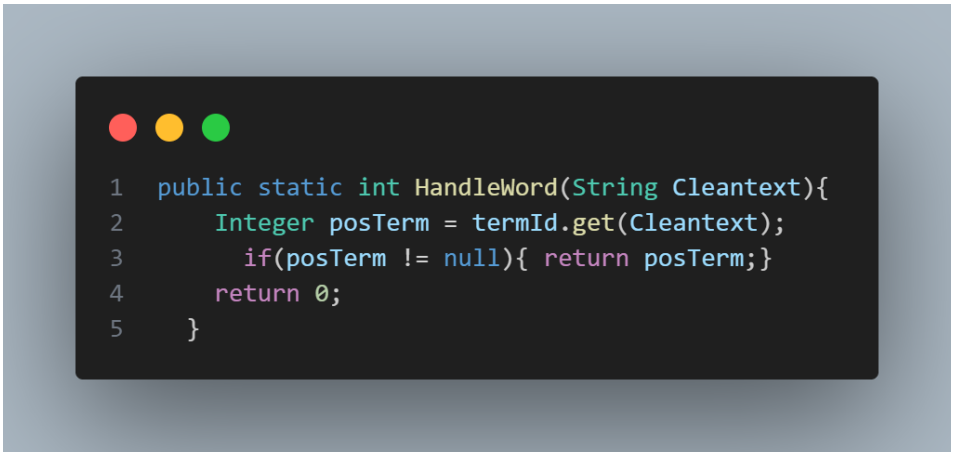
Figure 1: Map code

- CleanTheText:

```java
public static String CleanTheText(String PreClean){
    String specialChar = "~!@#$%^&*()\\-+\\[\\]\"':.,<>";
    Pattern Regex = Pattern.compile("[" + Pattern.quote(specialChar) + "]");
    Matcher Detectmatch = Regex.matcher(PreClean);
    String Cleantext = (Detectmatch.replaceAll("")).toLowerCase();
    return Cleantext;
  }
```

Figure 2: CleanTheText method

- HandleWord:



```java
public static int HandleWord(String Cleantext){
    Integer posTerm = termId.get(Cleantext);
      if(posTerm != null){ return posTerm;}
    return 0;
  }
```

Figure 3: HandleWord method

- **In Reducer phase:**

  Step 3: Reducing:

  - During the reduce phase, the process is straightforward. When key-value pairs with the same key are encountered, their values are summed up to generate the final value. This step is essential because certain words may appear multiple times within a single file, making aggregation necessary.
  - Once the reduce phase is completed, the result will be a key-value pair where the key remains the same as in the map phase, and the value reflects the total count of a word's occurrences within a file. Thus, the key-value format will be:
    - **Key**: checkTermNumber (the ID of the word) + " " + fileId (the ID of the file which is folder.filename (e.g bussiness.001))
    - **Value:** total number of occurrences in the file

```java
public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();
  public void reduce(Text key, Iterable<IntWritable> values,Context context) throws IOException, InterruptedException {
    int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
    result.set(sum);
    context.write(key, result);
  }
}
```

Figure 4: Reduce code

- **MTX converter phase:** This method essentially converts and aggregates the output of a MapReduce job into a single, structured Matrix Market file

  Step 4: Output formatting:

  - After reducing, we will continue converting and reformatting the output file to MTX file by calling the MTXConverter method in the ConvertMTXFile by passing in the output directory location.
  - **Configuration Setup**: The method starts by creating a Hadoop Configuration object and obtaining a reference to the file system (FileSystem Filesys) using that configuration.
  - **File Path Initialization**: It then defines the path for the output .mtx file, MTXFile, which will store the final converted data. The file will be named OutputTask1_1.mtx and stored in the specified directory (MTXDir).
  - **Reading Input Files**: The method searches for all files in the MTXDir directory that match the pattern part-r-*. These files typically represent the output of the reduce tasks in a MapReduce job. For each matching file, it opens the file using FSDataInputStream and reads its content line by line using a BufferedReader.
  - **Processing and Formatting Data**: Each line of the input files is split into components which is term ID, document ID, and frequency.The document ID is mapped using a docId mapping (business.001 have docId = 1), and the components are then formatted into a string of three integers which are termId and docId, frequency. These formatted strings are added to the MTXContent list.
  - **Writing the .mtx File**: After reading all the input files and storing the formatted data in MTXContent, the list is sorted.The method then creates the output .mtx file and writes the header information. Finally, it writes each formatted line from MTXContent to the .mtx file.

- When finishing the Reducer phase, the program will then run a function to reformat the structure of the output and return an output file with 3 columns of data(TermId DocId Frequency), namely:
    - TermId: the ID of the term
    - DocId: the ID of the file
    - Frequency: the number of occurrences of the term in a file

```java
public static class ConvertMTXFile {
    public static void MTXConverter(String MTXDir) throws IOException {
        Configuration conf = new Configuration();
        FileSystem Filesys = FileSystem.get(conf);
        Path MTXFile = new Path(MTXDir + "/OutputTask1_1.mtx");

        List<String> MTXContent = new ArrayList<>();
        FileStatus[] Filestate = Filesys.globStatus(new Path(MTXDir + "/part-r-*"));
          for (FileStatus state : Filestate) {
              Path statePath = state.getPath();
              try (FSDataInputStream InStream = Filesys.open(statePath);
                   BufferedReader Buffread = new BufferedReader(new InputStreamReader(InStream))) {
                  String line;
                  while ((line = Buffread.readLine()) != null) {
                      String[] cont = line.split("\\s+");
                      MTXContent.add(String.format("%d %d %d", Integer.parseInt(cont[0]), docId.get(cont[1]), Integer.parseInt(cont[2])));
                  }
              }
          }

        try (FSDataOutputStream OutStream = Filesys.create(MTXFile);
             BufferedWriter Buffwrite = new BufferedWriter(new OutputStreamWriter(OutStream))) {
            MTXContent.sort(Comparator.naturalOrder());
            Buffwrite.write("%%MatrixMarket matrix coordinate real general\n");
            Buffwrite.write(String.format("%d %d %d%n", termId.size(), docId.size(), MTXContent.size()));
            for (String x : MTXContent) {
                Buffwrite.write(x + "\n");
            }
        } finally {
            Filesys.close();
        }
    }
}
```

Figure 5: Output converter code

c. Task Flow:

- **The task run with this flow:**

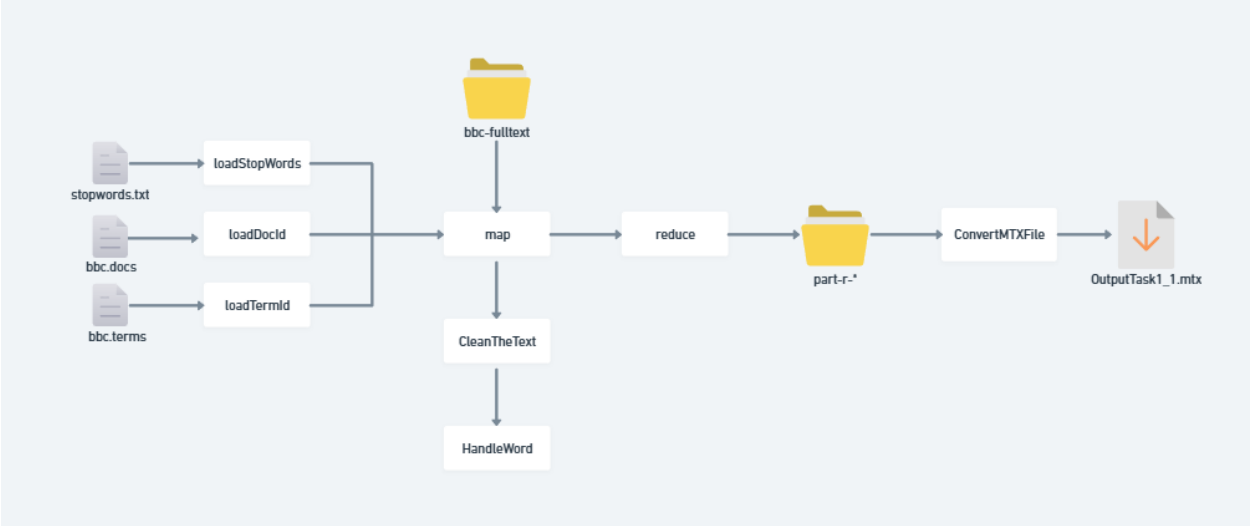


Figure 6: Figures describe task 1.1 flow

- **An example for this task:**
  - This output file will be named "task_1_2.mtx"
  - For example, we have a file name 001.txt, and it put in folder Business, and the content in file is: “The Ad Sales boost Time ad”.
    - We can see that the word “The ” is stop word and if in file terms, we have that:
    - ad in line 1
    - sales in line 2
    - boost in line 3
    - time in line 4
  - In file docs, we have that Business.001 is in the line 1
  - We will describe an example for this task as the below figure:

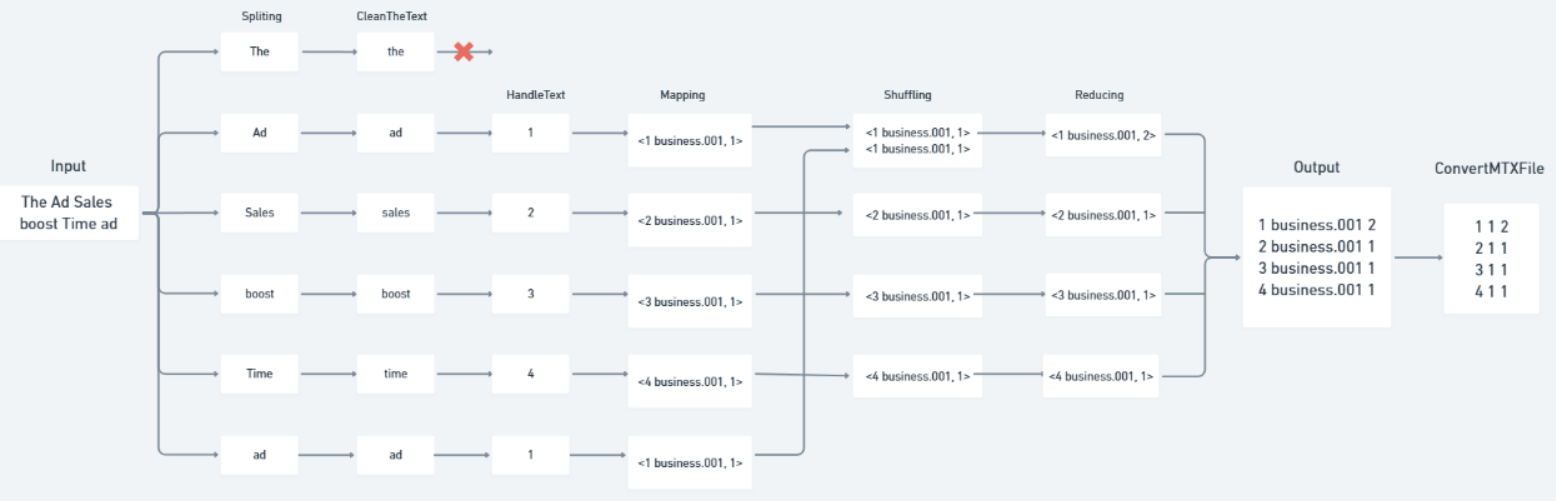
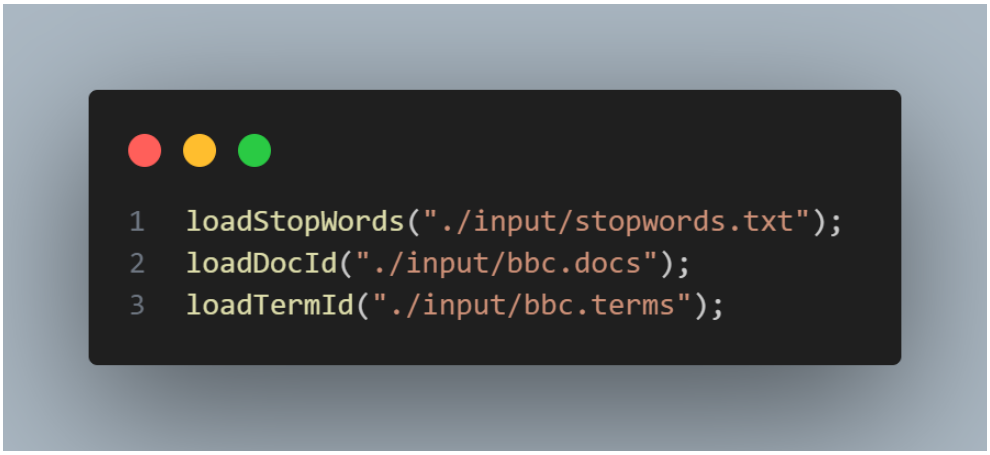


Figure 7: Figures describe map-reduce job

- From the input file, it's clear that the word "the" is a stopword and will be removed. The remaining words will be paired with the file-folder ID to create key-value pairs. During the reduction process, these pairs will be aggregated into a key in the format (termId foldername.fileid), where the value represents the total number of occurrences.

d. How to run:

- **Configure this section**:
  - We need to update the file paths according to your browser. Passing the stopwords file path to the loadStopWords, bbc.docs file path to the loadDocId , and bbc.terms file path to the loadTermId.



```
1  loadStopWords("./input/stopwords.txt");
2  loadDocId("./input/bbc.docs");
3  loadTermId("./input/bbc.terms");
```

Figure 8 : Configure Path

- **Command to run the program:** After completing the program, the results will return to the task_1_1.mtx file:

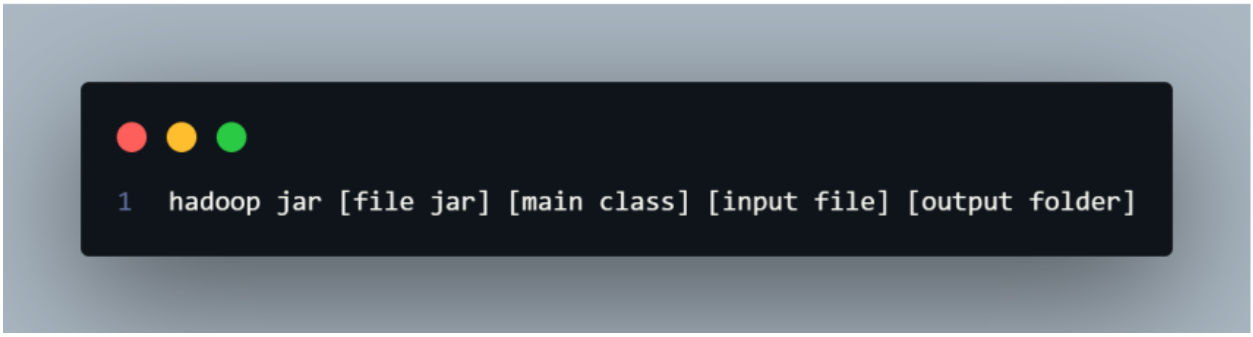


Figure 9: Command

e. Result:

- Sample of file task_1_1.mtx:

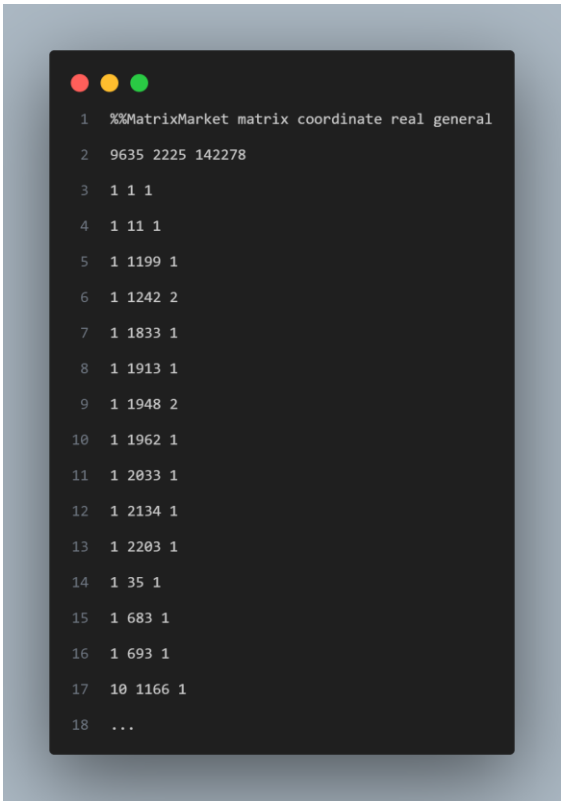


Figure 10: Sample of output task 1.1

- From the result file, we see that a file that meets the requirements is returned, including 3 columns: terms, docs and frequency.

3.2. Task 1.2 Low-Frequency Term Elimination

a. Data Description:

- MTX file generated from Task 1.1 named.
- Additionally, the processing code requires reading two files: bbc.terms and bbc.docs, for the following purposes:
    - bbc.terms: This file is used to assign an ID to each term. After removing stop words, each term is given an ID based on its line number.
    - bbc.docs: This file is used to assign an ID to each text file within a specific folder, with each file's ID corresponding to its line number.

b. MapReduce Job Implementation:

- **In Mapper phase:**

This mapper reads lines of text input, skips the first and second row which is the header of the MTX file, it splits the line into tokens. It then emits the first token as the key (termId) and the combination of the second and third tokens as the value (docId and frequency).

Step 1: Before map (set up):

- During the map phase, before mapping the data, the code will read and process auxiliary files as follows:
    - For the bbc.terms, it will extract the words from the file and read it into a HashMap<String, Interger> which String indicates the word in bbc.term, Interger indicates the line of the word.
    - For the bbc.docs, it will extract the documents id from the file into to into a HashMap<String, Interger> String is the name of the file and Interger indicates the line of the file.

      Step 2: Mapping:
- **Skipping the header of the MTX file**: The method first checks if row is less than 2. If it is, it increments the row counter and skips processing the current line. This effectively skips the first row of the input data.
- **Processing the Line**: It split into tokens using whitespace as the delimiter (split("\\s+")). The first token (cont[0]) is set as the termId. The second and

third tokens (cont[1] and cont[2]) which are docId and frequency are combined into a single string separated by a space and set as pairVal.

- **Emitting Key-Value Pairs**:The method then emits a key-value pair using context.write(termId, pairVal). Here, termId is the key, and pairVal is the value:
    - **Key:** termId (the ID of the word)
    - **Value:** pairVal (the ID of the doc +" "+ frequency)

```java
public static class TokenizerMapper extends Mapper<Object, Text, Text, Text>{
    private Text termId = new Text();
    private Text pairVal = new Text();
    private int row=1;

    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        if(row<2) {
            row++;
        }
        else {
        String [] cont = value.toString().split("\\s+");
        termId.set(cont[0]);
        pairVal.set(cont[1]+" "+cont[2]);
        context.write(termId,pairVal);
        }
    }
  }
```

Figure 11: Map code

- **In Reducer phase:** This reducer takes a group of values for each key and adds up the numbers of the frequency of the termId from those values. If the total is 3 or more, it writes out each value again with the same key.
    - **Reduce Method**: Iteration Over Values:
        - The method iterates over each Text value in the values iterable which is the docId and frequency. Each value is converted to a string and added to the PairVal list.
        - The string value is then split into its components using whitespace as the delimiter (split("\\s+")). So it can take the second (index 1) component which is the frequency of the termId to add it up.

- The second component (index 1 which is the **frequency** of the **termId** in the **docId**) of the split string is parsed into an integer (Valadd) and added to sum.

- **Conditional**: After iterating through all the values, the reducer checks if the add up sum (**frequency** of the **termId**) is greater than or equal to 3. If the condition is met, the reducer iterates over the PairVal list and writes each value back out as a key-value pair, with the original key and the value from PairVal.

```
public static class IntSumReducer
       extends Reducer<Text,Text,Text,Text> {
    private Text result = new Text();

    public void reduce(Text key, Iterable<Text> values, Context context )
    throws IOException, InterruptedException {

        ArrayList<String> PairVal = new ArrayList<>();
        Integer sum = 0;
      for (Text val : values) {
        String strVal = val.toString();
        PairVal.add(strVal);
        String ValCon[] = strVal.split("\\s+");
        Integer Valadd = Integer.parseInt(ValCon[1]);
        sum = sum + Valadd;
      }
      if (sum>=3){
        for (String val : PairVal) {
              result.set(val);
              context.write(key, result);
          }
      }
    }
  }
```

Figure 12: Reduce code

- **MTX converter phase:** This method essentially converts and aggregates the output of a MapReduce job into a single, structured Matrix Market file

<u>Step 3:</u> Output formatting:

- It has exactly the same code flow with task 1.1 but the output file would be OutputTask1_2.mtx.

c. Task Flow:

- **The task run with this flow:**

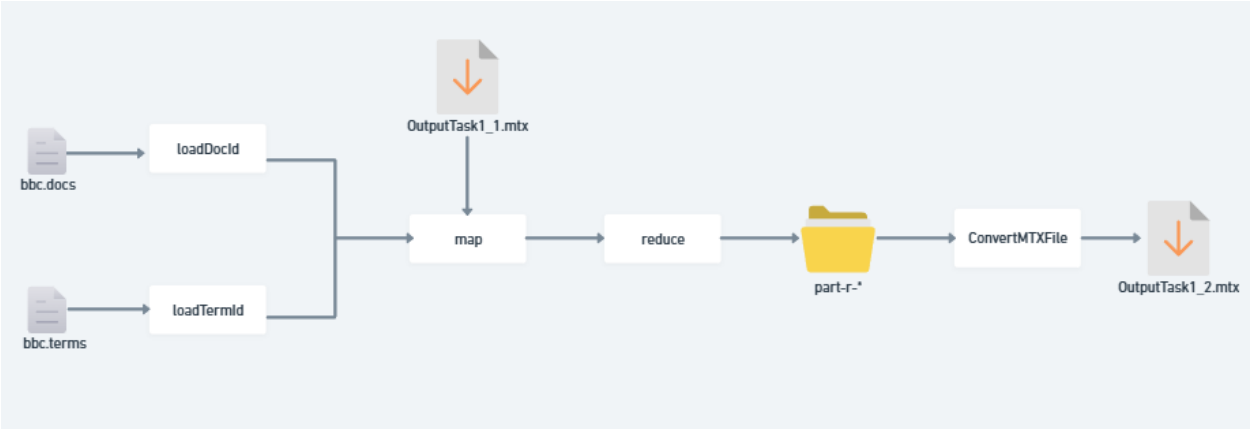


Figure 13: Figures describe task 1.2 flow

- **An example for this task:** We will describe an example for this task as the below figure:
  - For example, we have an input file as follows:

Figure 14: Example input

  - We can see that in the above file contains 3 columns show:
    - Word has termId = 1 appear in file has docId is 1 and 2 with frequency is 3 and 3
    - Word has termId = 2 appear in file has docId is 1 and frequency is 1 o
    - Word has termId = 3 appear in file has docId is 1 and 2 with frequency is 2 and 1

- The description of the map reduce will be show in the figures below:

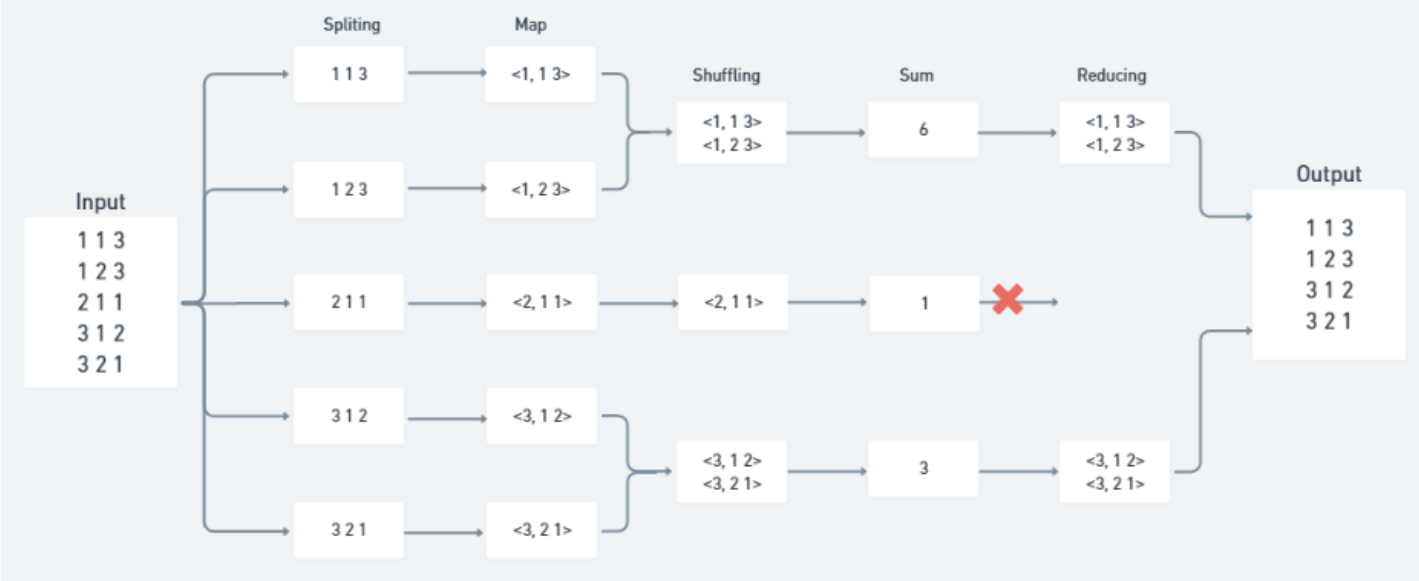


Figure 15: Figures describe map-reduce job

- Given the input, the word with termId = 2 does not meet the requirement of having a total occurrence count of at least 3. Consequently, after the map-reduce process, only the words with termId = 1 and termId = 3 satisfy the condition and remain.

d. How to run:

- **Configure this section**:
  - We need to update the file paths according to your browser. Passing bbc.docs file path to the loadDocId , and bbc.terms file path to the loadTermId.

Figure 16: Configure Path

- **Command to run the program:** After completing the program, the results will return to the OutputTask1_1.mtx file:

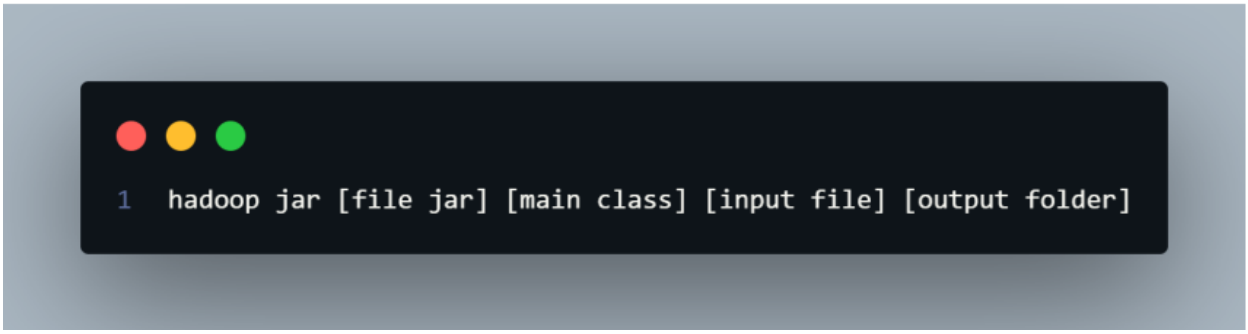


Figure 17: Command

e. Results:

- Sample of file task_1_2.mtx:

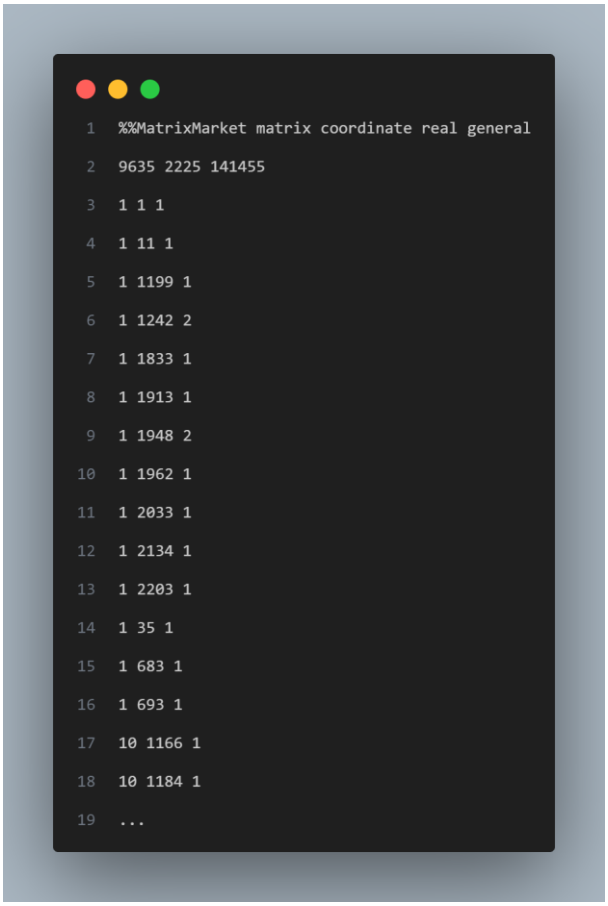


Figure 18: Sample of output task 1.2

- From the result file, the MTX file size has been subtracted 828 lines.

3.3. Task 1.3 Top 10 Most Frequent Words

a. Data Description: Output mtx file from task 1.2, this is a Matrix Market data file containing termID, docID and frequency. Each column represents the id of the term, the id of the document containing that term and the frequency of the term appearing in that document. The two first rows contain the header and the matrix size.

b. MapReduce Job Implementation:

The MapReduce job for this task is structured into 2 phases: the Map phase and the Reduce phase.

- **Map Phase:**

```java
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
        public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            String[] tokens = value.toString().split("\\s+");
            if (tokens.length == 3) {
                String termId = tokens[0];
                int freq = Integer.parseInt(tokens[2]);
                context.write(new Text(termId), new IntWritable(freq));
            }
        }
    }
```

Figure 19: Map code

- Input: Each line of the input MTX file represents a term - document pair and its corresponding frequency.
- Process: The Mapper Processes each line by splitting it into tokens to extract the termid and frequency. It then emits a key-value pair where the key is the termid, and the value is the frequency. This allows us to group all frequencies of a term across different documents.
- Output: The output of the Mapper is a list of key-value pairs, where each key is a termid, and each value is the frequency of that term in a specific document.

- **Reduce Phase:**

```java
public static class TopWordsReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        private Map<String, Integer> termFrequencyMap = new HashMap<>();
        private static final int TOP_N = 10;

        @Override
        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
            int totalFrequency = 0;
            for (IntWritable value : values) {
                totalFrequency += value.get();
            }
            termFrequencyMap.put(key.toString(), totalFrequency);
        }

        @Override
        protected void cleanup(Context context) throws IOException, InterruptedException {
            // Sắp xếp và ghi kết quả
            List<Map.Entry<String, Integer>> sortedEntries = new ArrayList<>(termFrequencyMap.entrySet());
            sortedEntries.sort(Map.Entry.comparingByValue(Comparator.reverseOrder()));

            int count = 0;
            for (Map.Entry<String, Integer> entry : sortedEntries) {
                if (count < TOP_N) {
                    context.write(new Text(entry.getKey()), new IntWritable(entry.getValue()));
                    count++;
                } else {
                    break;
                }
            }
        }
```

Figure 20: Reduce code

- Input: The Reducer receives key-value pairs from the Mapper, where each key is a termid, and the corresponding values are the list of frequencies of that term across all documents.
- Process: The Reducer sums these frequencies to obtain the total frequency for each term across all documents. It stores these totals in a map for further processing.
- Sorting and Selection: After processing all the terms, the Reducer sorts the terms by their total frequency in descending order and selects the top 10 most frequent terms.
- Output: The output of the Reducer is the top 10 most frequent terms, written to the file **task_1_3.txt** in the format [termid] [frequency].

- The description of the map-reduce will be show in the figures below (example finding top 2):

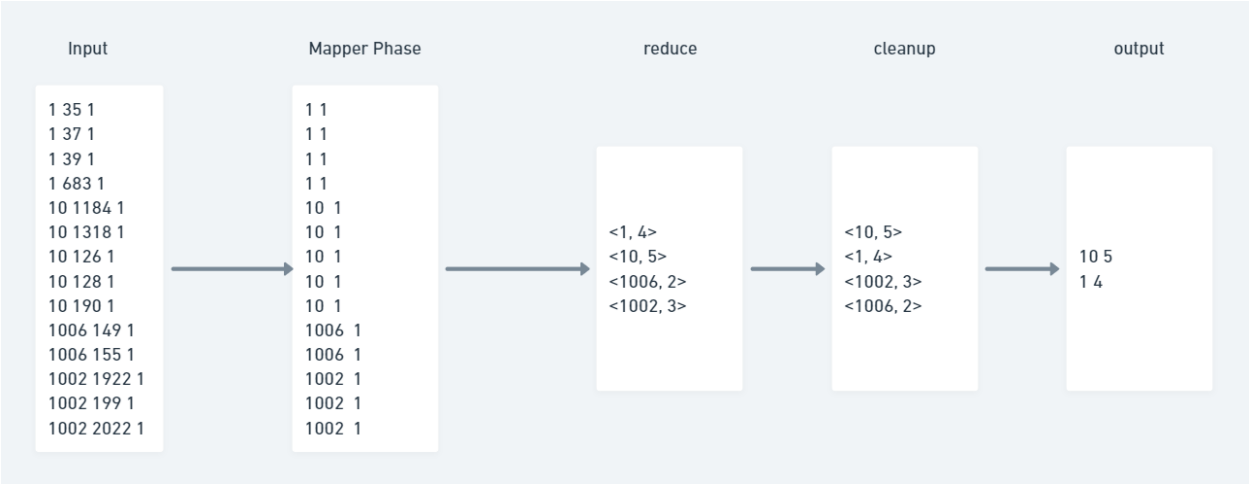


Figure 21: describe map reduce flow

c. Task Flow:

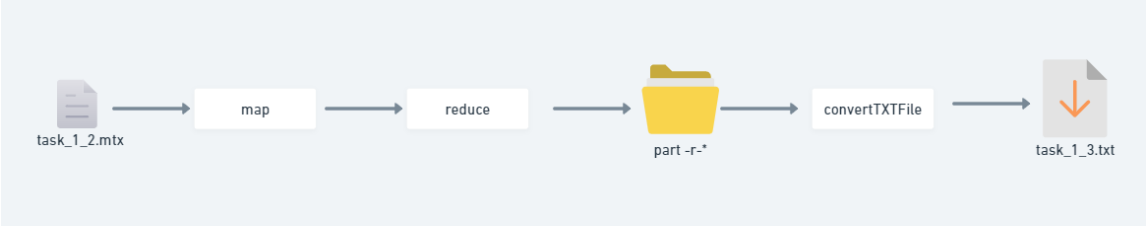


Figure 22: describe task 1.3 flow

- Step 1: Data Preparation
  - The MTX file generated from Task 1.2 is used as input.
- Step 2: Map
  - Each term-document pair is processed to emit term frequency key-value pairs.
- Step 3: Reduce
  - The term frequencies are summed, and the top 10 terms are selected based on their total frequency.
- Step 4: Sorting and Output
  - The terms are sorted in descending order, and the top 10 are written to the output file task_1_3.txt.

d. How to run:

- Command to run the program:

```
hadoop jar [file jar] [main class] [input file] [output folder]
```

Figure 23: Command

e. Result:

- File named task_1_3.txt, which contains the top 10 most frequent terms across all documents. Each line of this file is formatted as [termid] [frequency], where termid is the unique identifier of the term, and frequency is the total number of occurrences of that term across all documents.
- Sample of task_1_3.txt

```
213 2309
601 1355
4   1322
522 1282
461 1201
470 1115
865 974
166 971
529 957
318 911
```

Figure 24: Output (task_1_3.txt)

3.4. Task 1.4 TF-IDF

a. Data Description: Output mtx file from task 1.2, this is a Matrix Market data file containing termID, docID and frequency. Each column represents the id of the

term, the id of the document containing that term and the frequency of the term appearing in that document. The two first rows contain the header and the matrix size.

b. MapReduce Job Implementation:

In this task we divide into **2 Map phases** and **2 Reduce phases** for the MapReduce process into **2 distinct Jobs:**

- **Job 1** calculates the Term Frequency (TF) value for each term in each document class.
- **Job 2** calculates and outputs the TF-IDF score for each term in each document class.

Before going into the map reduce work, we will have to save the **total document count value (totalDocs)** in the dataset to use for calculating the TF-IDF score. We used the ***saveToTalDocumenttoConfig*** function to save the total document value into the configuration. The **totalDocs** value is read from the input file in the second line, where the size values are contained.

1. Job 1: calculates the Term Frequency (TF)

- **Map Phase**

- Input data: An input data file in data description (after skipping the first 2 lines) containing 3 parts: *termID*, *docID*, and *frequency*.
- The map() function in TermFrequencyMapper will read each line, split it into 3 parts based on the space, then return the key-value pairs with:
  - **Key**: docID (document index).
  - **Value:** Combine termID and frequency with tab (termID + "\t" + frequency).

```java
public static class TermFrequencyMapper extends Mapper<LongWritable, Text, Text, Text> {
    private int row = 0;

    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        // skip the first two lines
        if (row < 2) {
            row++;
        } else {
            String[] parts = value.toString().split("\\s+");
            // key: docID, value: termID+frequency
            context.write(new Text(parts[1]), new Text(parts[0] + "\t" + parts[2]));
        }
    }
}
```

Figure 25: Job 1 Map function

- **Reduce Phase:**

- The purpose of reduce phase is to calculate the term frequency value for each term in each document, according to the following formula:

$$tf(t,d) = \frac{f(t,d)}{sum\{f(w,d) : (w \in d\}} = \frac{frequency\ of\ term}{total\ number\ of\ terms\ in\ the\ document}$$

- Input data: The output of Mapper with key as docID and value as termID combined with frequency.
- First, the reducer extracts the termID and frequency from each value and calculates the total number of terms in that document by summing up all the frequency values.
- After determining the total number of terms in the document, the reducer calculates the TF for each termID in that document by dividing the frequency of the term by the total number of terms in the document.
- The output will be key-value pairs where
    - **Key:** docID + “\t” + termID
    - **Value:** the computed TF value.
    - The output form: [docID] [termID] [TF]

```java
public static class TermFrequencyReducer extends Reducer<Text, Text, Text, Text> {
        public void reduce(Text key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {

            double total = 0;
            Map<String, Integer> termFrequencyMap = new HashMap<>();

            for (Text value : values) { // compute total words per document
                String[] parts = value.toString().split("\\s+");
                int frequency = Integer.parseInt(parts[1]);
                total += frequency;
                termFrequencyMap.put(parts[0], frequency);
            }

            // compute TF
            for (Map.Entry<String, Integer> entry : termFrequencyMap.entrySet()) {
                String termID = entry.getKey();
                int termFrequency = entry.getValue();
                double tf = termFrequency / total;
                //key: docID+termID, value: TF
                context.write(new Text(key.toString() + "\t" + termID), new Text(String.format("%.6f", tf)));
            }
        }
    }

```

Figure 26: Job 1 Reduce function

2. Job 2

- Map Phase:

- The input for the map phase is the output from the reduce phase of Job 1, where each line consists of a docID, termID, and TF value.
- The mapper's job is to process this data and split each line into key-value pairs where:
    - **Key**: termID
    - **Value**: docID+ “\t” + TF value

```java
public static class TFIDFMapper extends Mapper<LongWritable, Text, Text, Text> {
        @Override
        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
            String[] parts = value.toString().split("\\s+");

            // key: termID, value: docID+TF
            context.write(new Text(parts[0]), new Text(parts[1] + "\t" + parts[2]));
        }
    }
```

Figure 27: Job 2 Map function

- **Reduce Phase:**

- The purpose of the Reduce phase in Job 2 is to calculate the TF-IDF score for each term in each document.
- Input data: the output from the Map phase above with the key: termID and the value: docID and TF.
- Before going into the Reduce phase, we will need to read the value of the total number of documents that I introduced at the beginning, this value is saved in the configuration and is used to calculate the IDF score below.
- First, we will calculate Inverse Document Frequency (IDF) for each term in the document. It is calculated as the logarithm of the ratio of the total number of documents to the number of documents containing the term.

$$idf(t, D) = log(\frac{|D|}{|\{d \in D : t \in d\}|})$$

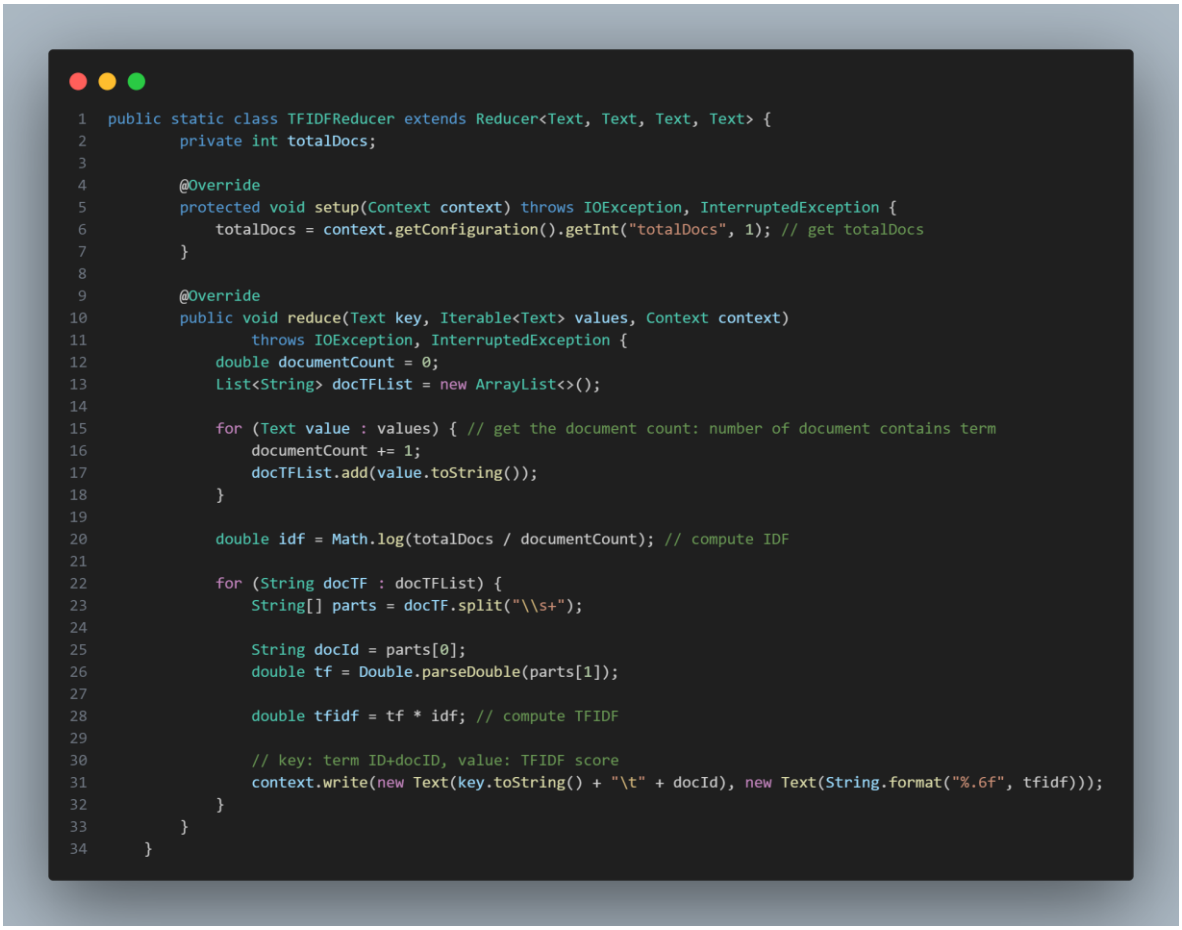
$$= log(\frac{total\ number\ of\ documents}{number\ of\ document\ contains\ the\ term})$$

- Then, for each pair of docID and TF in the list, we will calculate the TF-IDF score value by multiplying TF by IDF:

$$tfidf(t, d, D) = tf(t, d) \times idf(t, D)$$

- The final result will be returned as key value pairs in the following format:
    - **Key:** termID + "\t" + docID

- **Value**: TF-IDF score
- The output form: [termID] [docID] [TFIDF]



```java
public static class TFIDFReducer extends Reducer<Text, Text, Text, Text> {
        private int totalDocs;

        @Override
        protected void setup(Context context) throws IOException, InterruptedException {
            totalDocs = context.getConfiguration().getInt("totalDocs", 1); // get totalDocs
        }

        @Override
        public void reduce(Text key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {
            double documentCount = 0;
            List<String> docTFList = new ArrayList<>();

            for (Text value : values) { // get the document count: number of document contains term
                documentCount += 1;
                docTFList.add(value.toString());
            }

            double idf = Math.log(totalDocs / documentCount); // compute IDF

            for (String docTF : docTFList) {
                String[] parts = docTF.split("\\s+");

                String docId = parts[0];
                double tf = Double.parseDouble(parts[1]);

                double tfidf = tf * idf; // compute TFIDF

                // key: term ID+docID, value: TFIDF score
                context.write(new Text(key.toString() + "\t" + docId), new Text(String.format("%.6f", tfidf)));
            }
        }
    }
```

Figure 28: Job 2 Reduce Function

- The description of the map-reduce will be show in the figures below (illustrative data):

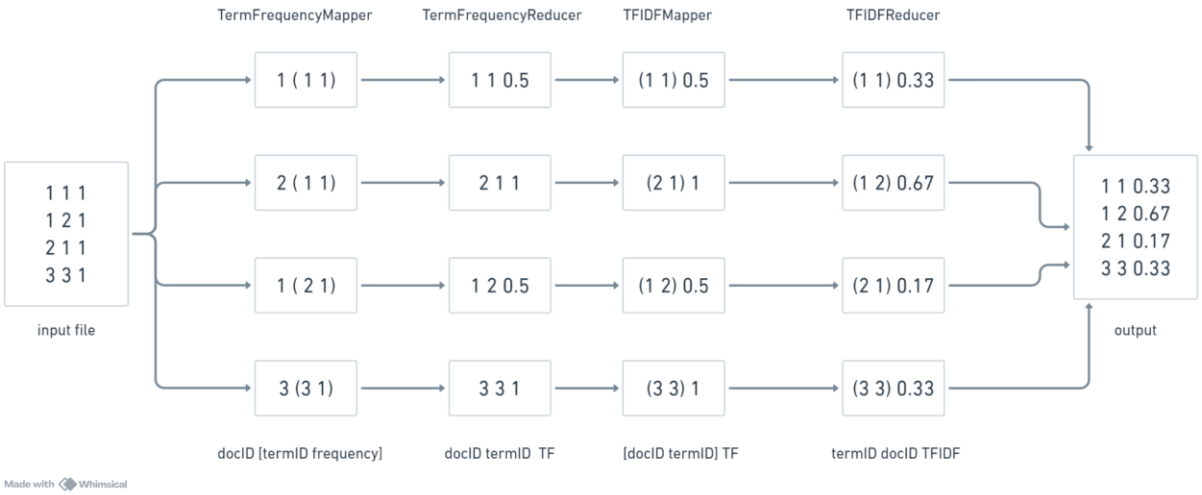


Figure 29: Description of the map-reduce flow

c. Task Flow:

- Step 1: Read data file to save total document quantity value into config via ***saveTotalDocumenttoConfig*** function.
- Step 2: Perform Job 1 to calculate the **TF** for each term
  - TermFrequencyMapper:
    - Read input data file and return key-value pairs corresponding to [docID] and [termID frequency]
  - TermFrequencyReducer:
    - Calculate TF value for each term in each document, return key-value pairs corresponding to [docID termID] and [TF]
- Step 3: Save the output from Job 1 to be the input for Job 2
- Step 4: Perform Job 2 to calculate the **TF-IDF** score for each term
  - TFIDFMapper:
    - Read values from input file which is output of job 1, return key value pairs corresponding to [docID termID] and [TF]
  - TFIDFReducer:
    - Using the previously set documentCount and TotalDocs, calculates the IDF value.

- Compute the TF-IDF score and generate the key value pairs corresponding to [termID docID] [TF]

- Step 5: Save the final output and convert it to an MTX file.

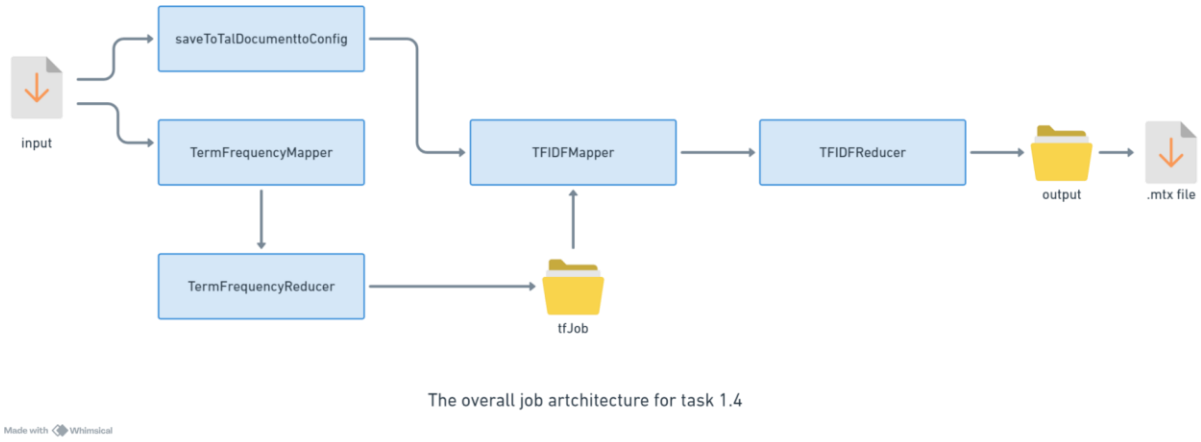


Figure 30: Overall task flow of task 1.4

d. How to run:

- **Configure this section**: We have to define the output file path for Job 1 - TF Job, this will also be the input path for the Mapper in Job 2.

```
1  FileOutputFormat.setOutputPath(tfJob, new Path("/tfJob"));
```

Figure 31: Config output path for Job 1 (Term Frequency Job)

```
1  FileInputFormat.addInputPath(tfidfJob, new Path("/tfJob"));
```

Figure 32: Config input path for Job 2 (TFIDF Job)

- **Command to run the program:**

```
1  hadoop jar [file jar] [main class] [input file] [output folder]
```

Figure 33: Command

e. Result:

- Sample of file task_1_4.mtx

```
1  1 21 0.021386
2  1 23 0.085548
3  1 25 0.042776
4  1 26 0.021386
5  1 27 0.064162
6  1 28 0.085548
7  1 29 0.042776
8  1 132 0.021386
9  1 135 0.042776
```

Figure 34: Sample output of task 1.4

3.5. Task 1.5 Highest average TF-IDF

- **Formular:**

$$avg_{tfidf_{t,C_i,D}} = \frac{1}{|C_i|} \sum_{d \in C_i} tdidf_{t,d,D}$$

where: $|C_i|$ is total of files in class C_i

- To complete Task 1.5, we divided the MapReduce process into two distinct jobs:
  - **Job 1** calculates the average TF-IDF value for each term in each document class.
  - **Job 2** then identifies and outputs the top 5 terms with the highest average TF-IDF values for each class

a. Data Description:

- **Job 1:** Calculate Average TF-IDF for Each Term in Each Class
  - The input for Job 1 is an MTX file from Task 1.4. This file follows the Matrix Market format and contains:
    - *termID:* Identifier for the term (word).
    - *docID:* Identifier for the document containing the term.
    - *TF-IDF:* TF-IDF value of the term in the document.
- **Job 2:** Identify Top 5 Terms with Highest Average TF-IDF for Each Class
  - The input for Job 2 is the output from Job 1, which contains termID, class, and average TF-IDF values.

b. MapReduce Job Implementation:

1. Job 1: Calculate Average TF-IDF for Each Term in Each Class

- **Map Phase:**

```
public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            String[] tokens = value.toString().split("\\s+");
            String doc = getDoc(Integer.valueOf(tokens[1]));
            String type = doc.substring(0, doc.length() - 4);
            context.write(new Text(type),
                    new Text(tokens[0] + "\t" + doc.substring(doc.length() - 4) + "\t" + tokens[2]));
        }
```

Figure 35: Map code (job 1)

  - Input: Each line of the MTX file represents a term-document pair with a TF-IDF value.

- Processing:
    - Step 1: Read the **bbc.docs** file**:** Read the list of documents from the **bbc.docs** file and store them in a list for reference.
    - Step 2: Extract classID from docID: For each docID, split it to obtain the first part as classID.
    - Step 3: Emit key-value pairs:
        - **Key:** docID (class name corresponding to the docID).
        - **Value:** termID + "\t" + docID + "\t" + TF-IDF.
- Output: A list of key-value pairs where the key is classID and the value is a combination of termID, docID, and the TF-IDF value.

- **Reduce Phase:**

```java
public static class ReduceJob1 extends Reducer<Text, Text, Text, Text> {
        public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
            HashSet<String> doc = new HashSet<>();
            HashMap<String, Double> data = new HashMap<>();

            for (Text val : values) {
                String[] tokens = val.toString().split("\\s+");
                doc.add(tokens[1]);
                if (!data.containsKey(tokens[0]))
                    data.put(tokens[0], Double.valueOf(tokens[2]));
                else
                    data.put(tokens[0], data.get(tokens[0]) + Double.valueOf(tokens[2]));
            }

            double count = doc.size();
            data.forEach((keyMap, value) -> {
                double result = (double) value / count;
                try {
                    context.write(new Text(keyMap + "\t" + key.toString()), new Text(Double.toString(result)));
                } catch (InterruptedException | IOException e) {
                    e.printStackTrace();
                }
            });
        }
```

Figure 36: Reduce code (job 1)

- Input: The Reducer receives key-value pairs with classID as the key and termID + docID + TF-IDF as the value.
- Processing:
    - Step 1: Create a HashSet to store the documents: A HashSet helps count the number of documents in each class (classID).
    - Step 2: Use a HashMap to store cumulative TF-IDF values: The HashMap stores the total TF-IDF for each termID within a class.

- Step 3: Calculate the average TF-IDF: For each *termID* in a class, calculate the average TF-IDF by dividing the total TF-IDF by the number of documents in the class.
- Step 4: Emit the average TF-IDF with termID and classID as the key.

Key: termID + "\t" + classID.

Value: The average TF-IDF value.

- Output: Key-value pairs with:
  - Key: *TermID + classID*
  - Value: The average TF-IDF of the termID in the classID.

2. Job 2: Identify Top 5 Terms with Highest Average TF-IDF for Each Class

- **Map Phase:**

```
public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            String[] tokens = value.toString().split("\\s+");
            context.write(new Text(tokens[1]), new Text(tokens[0] + "\t" + tokens[2]));
        }
```

Figure 37: Map code (job 2)

- Input: Each key-value pair from Job 1 with *termID* and *average TF-IDF value*.
- Processing: Create key-value pairs:
  - Key: classID (class name).
  - Value: termID + “\t” + average TF-IDF value.
- Output: Key-value pairs where the key is classID, and the value is termID with the average TF-IDF value.

- **Reduce Phase:**

```java
public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
    List<Pair<String, Double>> data = new ArrayList<>();

    for (Text val : values) {
        String[] tokens = val.toString().split("\\s+");
        String term = getTerm(Integer.valueOf(tokens[0]));
        data.add(new Pair<>(term, Double.valueOf(tokens[1])));
    }

    data.sort(new Comparator<Pair<String, Double>>() {
        @Override
        public int compare(Pair<String, Double> o1, Pair<String, Double> o2) {
            return o2.getValue().compareTo(o1.getValue());
        }
    });

    int count = 0;
    DecimalFormat df = new DecimalFormat("#.##");
    String result = "";
    for (Pair<String, Double> pair : data) {
        if (count >= 5)
            break;
        if (count != 0)
            result += ", ";
        count++;
        String roundedString = df.format(pair.getValue());
        result += pair.getKey() + ":" + roundedString;
    }
    String output = key.toString().substring(0, 1).toUpperCase() + key.toString().substring(1) + ": " + result;
    context.write(new Text(output), new Text(" "));
}
```

Figure 38: Reduce code (job 2)

- Input: The Reducer receives key-value pairs where the key is classID, and the values are termID + average TF-IDF.
- Processing:
    - Step 1: Create a Pair class to map termID to actual term names: Use the bbc.terms file to map termID to the actual term name.
    - Step 2: Sort terms by their average TF-IDF values: Sort the terms by their average TF-IDF values in descending order.
    - Step 3: Select the top 5 terms with the highest average TF-IDF: Choose the top 5 terms with the highest average TF-IDF values for each class.
    - Step 4: Emit the top 5 terms with their average TF-IDF for each class.
        - **Key:** The name of the class (e.g., “Business,” “Tech”).
        - **Value:** A string containing the top 5 terms formatted as: term:average TF-IDF, with terms sorted by their average TF-IDF values in descending order.

- Output:
    - Key: The name of the class.
    - Value: A string containing the top 5 terms and their average TF-IDF for each class.

- The description of the map reduce will be show in the figures below (illustrative data):

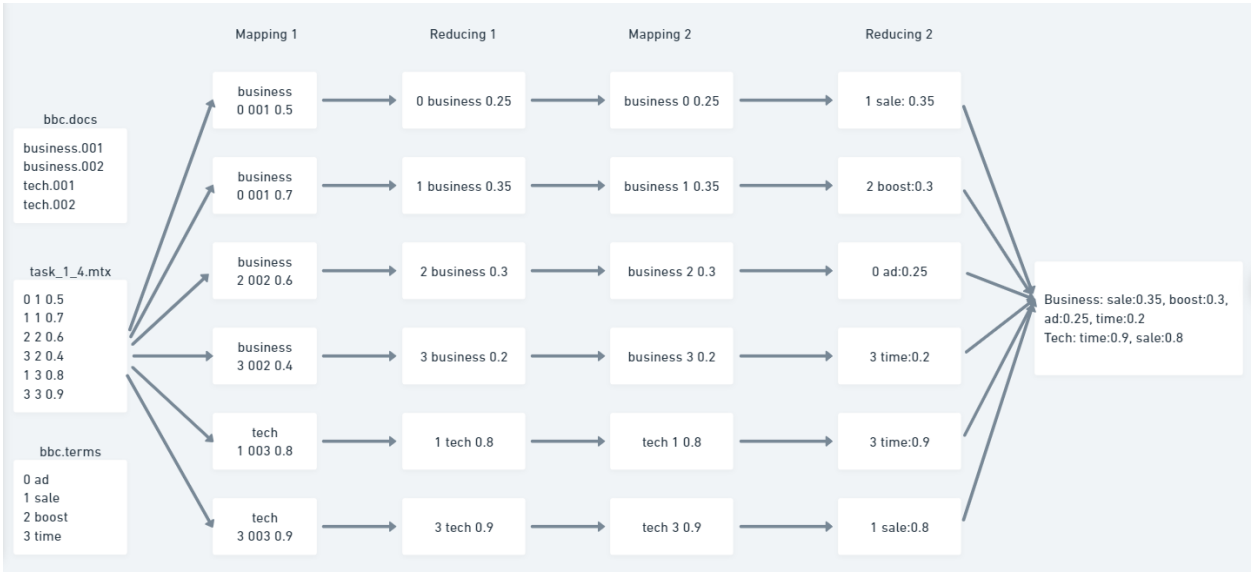


Figure 39: Description of the map reduce flow

c. Task Flow:

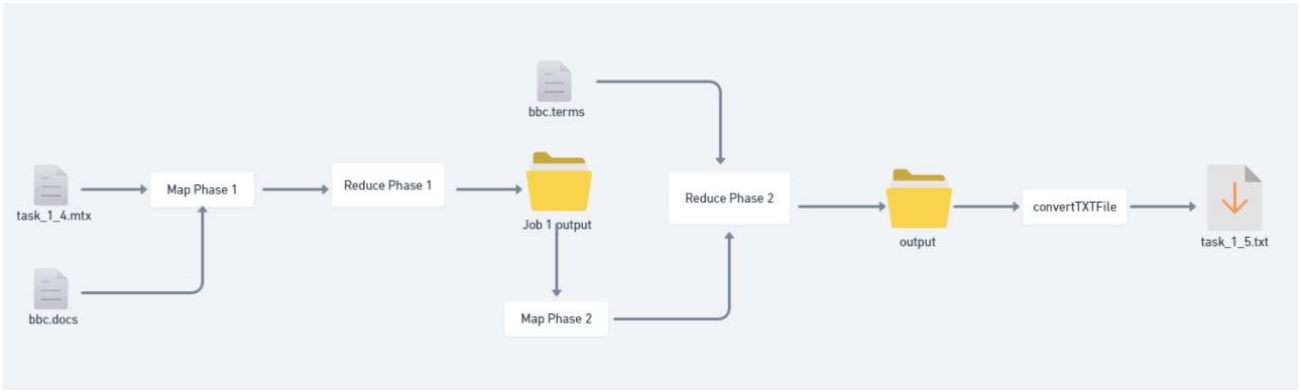


Figure 40: Description of task 1.5 flow

- ***Job 1:*** Calculate Average TF-IDF for Each Term in Each Class
    - **Data Preparation:** Use the MTX file from Task 1.4.

- **Map Phase:** Process term-document pairs to emit termID and TF-IDF values with docID as the key.
- **Reduce Phase:** Compute average TF-IDF values for each termID within each class and emit results.

- ***Job 2:*** Identify Top 5 Terms with Highest Average TF-IDF for Each Class
  - **Data Preparation:** Use the output from Job 1.
  - **Map Phase:** Emit class and termID with average TF-IDF values.
  - **Reduce Phase:** Sort terms by average TF-IDF, select the top 5 terms for each class, and emit the final results.

d. Result:

- **Command to Run the Program:**

```
hadoop jar [file jar] [main class] [input file] [output folder]
```

Figure 41: command

- The final output is written to *task_1_5.txt*. This file contains the top 5 terms with the highest average TF-IDF values for each class.
- Each line in the file is formatted as:
  - Class: term: average IF - IDF value
- After completing the program, the results will return to the task_1_3.txt file. The resule of the task_1_3.txt file are as follows:

```
Business: bank:0.02, growth:0.02, oil:0.02, market:0.02, firm:0.02
Entertainment: film:0.05, best:0.02, show:0.02, music:0.02, band:0.02
Politics: labour:0.04, blair:0.03, election:0.03, brown:0.02, howard:0.02
Sport: game:0.02, england:0.02, cup:0.02, match:0.02, win:0.02
Tech: microsoft:0.02, net:0.02, phone:0.02, virus:0.02, music:0.02
```

Figure 42: Output (task_1_5.txt)

4. Part 2: K-Means Algorithm

4.1. Task 2.1 K-Means on 2D Data

a. Data Description: A sample file containing 2D data points (one point per line, separated by spaces) with cluster, datapoint x, datapoint y.

b. MapReduce Job Implementation:

First, we will randomly initialize k centroids based on selecting k random points from the given dataset. This will be used on the ***initializeCentroids*** function.

- This function will read the data file, then randomly select k data points from the file to be centroids.

```java
public static List<Point> initializeCentroids(int k, FileSystem fs, String inputPath) throws IOException {
    List<Point> allPoints = new ArrayList<>();
    List<Point> centroids = new ArrayList<>();

    FileStatus[] files = fs.listStatus(new Path(inputPath));
    for (FileStatus file : files) {
        try (BufferedReader br = new BufferedReader(new InputStreamReader(fs.open(file.getPath())))) {
            String line;
            while ((line = br.readLine()) != null) {
                try {
                    String[] parts = line.trim().split(",");
                    double x = Double.parseDouble(parts[1]);
                    double y = Double.parseDouble(parts[2]);
                    allPoints.add(new Point(x, y));
                } catch (NumberFormatException e) {
                    continue;
                }
            }
        }
    }

    // shuffle the list to get random centroids
    Collections.shuffle(allPoints);

    for (int i = 0; i < k; i++) {
        centroids.add(allPoints.get(i));
    }

    return centroids;
}
```

Figure 43: initializeCentroids function

We have added a parameter called **THRESHOLD** with the value of **0.00001** to set the **early stopping** mechanism for the algorithm. If the distance between the newly generated centroids and the old centroids is not greater than this threshold, the algorithm will stop.

In this task, we have **1 Map phase** and **1 Reduce phase.** Specifically, the execution steps are as follows:

- **In Map Phase:**
    - Before the mapping phase begins, the Mapper loads the centroids from the previous iteration (or from the initial centroids if it's the first iteration) using the ***setup*** method. These centroids are stored in the Hadoop configuration via ***saveCentroidsToConfig*** and ***readCentroidsFromConfig*** functions (check it via our source code).
    - The input data for the Mapper consists of data points in the format “cluster, datapoint_x, datapoint_y”. We will parse this string to extract the x and y coordinates of each data point, which are essential to determine the location of each point on the 2D plane. We intentionally skip over any lines that start with the letter "c" (these lines typically store centroids in the format "centroids:cluster, centroid_x, centroid_y", we will explain in detail in the Reduce Phase).
    - Then, we will find the nearest cluster for the datapoint by using the ***getNearestCluster*** function.
    - Finally, it will generates pairs of key-value corresponding to the following format:
        - **Key:** the nearest cluster of the datapoint
        - **Value:** datapoint (x,y)
        - The output form: “[cluster],[datapoint_x],[datapoint_y]”.

```java
public static class ClusterMapper extends Mapper<Object, Text, IntWritable, Text> {
        private List<Point> centroids;
        private IntWritable clusterKey = new IntWritable();

        @Override
        protected void setup(Context context) {
            centroids = readCentroidsFromConfig(context.getConfiguration());
        }

        @Override
        protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            String[] parts;
            Point point;
            try {
                parts = value.toString().trim().split(",");
                double x, y;
                x = Double.parseDouble(parts[1]);
                y = Double.parseDouble(parts[2]);
                if (parts[0].charAt(0) == 'c') { // skip if it is centroids
                    return;
                }
                point = new Point(x, y);
            } catch (Exception e) {
                return;
            }

            int nearestCluster = getNearestCluster(point);
            clusterKey.set(nearestCluster);

            // key: cluster, value: datapoint
            context.write(clusterKey, new Text(point.toString()));
        }

        private int getNearestCluster(Point point) {
            int nearestCluster = 0;
            double minDistance = Double.MAX_VALUE;

            for (int i = 0; i < centroids.size(); i++) {
                double distance = point.distanceTo(centroids.get(i));
                if (distance < minDistance) {
                    minDistance = distance;
                    nearestCluster = i;
                }
            }

            return nearestCluster;
        }
    }
```

Figure 44: Map function

- **In Reduce Phase:**
    - After the "shuffle and sort" stage, Reducer receives as input the cluster index (key) and the list of data points belonging to that cluster (values). Then it will convert the input text datapoint values into lists of Points.

- Next, it will calculate to create new centroids values for the cluster based on averaging the points in the cluster.
- Because we need to save the datapoint values for future iterations, we will save the data points and the cluster they belong to as key-value pairs of the form:
    - **Key:** cluster
    - **Value:** datapoint (x,y)
    - The output form: “[cluster],[datapoint_x],[datapoint_y]”.
- Finally we will save the newly calculated centroids values, however to mark and distinguish them from the data points saved above we will save them as: "centroids:”[cluster],[centroid_x],[centroid_y]. That's why there are data lines starting with the char “c” that we need to ignore in the Map phase above. Therefore, the Reducer output key-value pairs will be of the form:
    - **Key: “**centroids:”cluster
    - **Value:** centroid (x,y)
    - The output form: "centroids:[cluster],[centroid_x],[centroid_y]”

```java
public static class ClusterReducer extends Reducer<IntWritable, Text, Text, Text> {
        @Override
        protected void reduce(IntWritable key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {
            List<Point> points = new ArrayList<>();
            for (Text value : values) {
                points.add(Point.fromString(value.toString()));
            }

            Point newCentroid = Point.average(points);

            for (Point point : points) {
                // save the cluster & datapoints: key: cluster, value: datapoint
                context.write(new Text(key.toString() + ","), new Text(point.toString()));

            }

            // save the centroids, key: "centroids:"cluster, value: centroid
            context.write(new Text("centroids:" + key.toString() + ","), new Text(newCentroid.toString()));
        }
    }
```

Figure 45: Reduce Function

- The description of the map reduce will be show in the figures below (illustrative data):

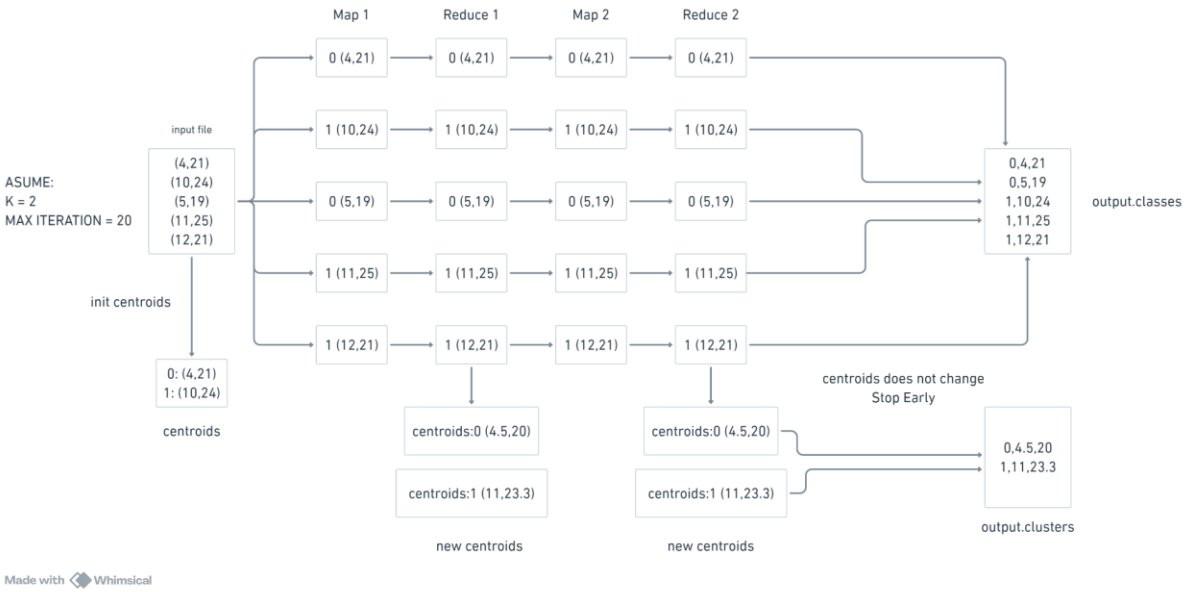


Figure 46: The description of the map reduce

c. Task Flow:

- Step 1: Init randomly k centroids by ***initializeCentroids***. Save it to the configuration by ***saveCentroidsToConfig*** function.
- Step 2: With K = 3 and Iterations = 20, for each iteration:
    - Start Map Phase:
        - Load the centroids from configuration via ***readCentroidsFromConfig***.
        - Find the nearest cluster for the datapoint.
        - Return the key-value pairs of cluster and datapoint.
    - Start Reduce Phase
        - Load the output of Map Phase.
        - Compute new centroids by averaging all data points in the cluster.
        - Return the key-value pairs of cluster and centroids and key-value pairs of cluster and data points.
        - Save the Reduce Phase output.
    - Check if the distance between new centroids and old centroids does not larger than THRESHOLD -> stop early
    - Read the output Reduce Phase from HDFS to get the new centroid via ***readCentroidsFromFile*** function and then save it to configuration using ***saveCentroidsToConfig.***
- Step 3: Save the last iteration output into **task_2_1.classes** and **task_2_1.cluster.**

```java
public static void runKmeans(Configuration conf, String inputPath, String outputPath) throws Exception {
        FileSystem fs = FileSystem.get(conf);

        // init randomly k centroids
        List<Point> centroids = initializeCentroids(K, fs, inputPath);
        saveCentroidsToConfig(conf, centroids);
        Path centroidOutputPath = new Path(outputPath);

        for (int iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
            Job job = Job.getInstance(conf, "KMeans Iteration " + iteration);
            job.setJarByClass(task2_1.class);

            job.setMapperClass(ClusterMapper.class);
            job.setReducerClass(ClusterReducer.class);

            job.setMapOutputKeyClass(IntWritable.class);
            job.setMapOutputValueClass(Text.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Text.class);

            FileInputFormat.addInputPath(job, new Path(inputPath));
            FileOutputFormat.setOutputPath(job, new Path(outputPath + "/iter_" + iteration));

            job.waitForCompletion(true);

            // update centroids
            centroidOutputPath = new Path(outputPath + "/iter_" + iteration + "/part-r-00000");
            List<Point> newCentroids = readCentroidsFromFile(fs, centroidOutputPath);

            // check if can stop early
            double maxDistance = 0;
            for (int j = 0; j < centroids.size(); j++) {
                double distance = centroids.get(j).distanceTo(newCentroids.get(j));
                if (distance >= maxDistance) {
                    maxDistance = distance;
                }
            }

            if (maxDistance <= THRESHOLD) {
                break;
            }

            centroids = newCentroids;
            saveCentroidsToConfig(conf, centroids);
        }

        // save the output into task_2_1.cluster & task_2_1.classes
        writeOutput(fs, K, centroidOutputPath, outputPath + "/task_2_1.clusters", outputPath + "/task_2_1.classes");
    }
```

Figure 47: Function to set up flow for run KMeans program

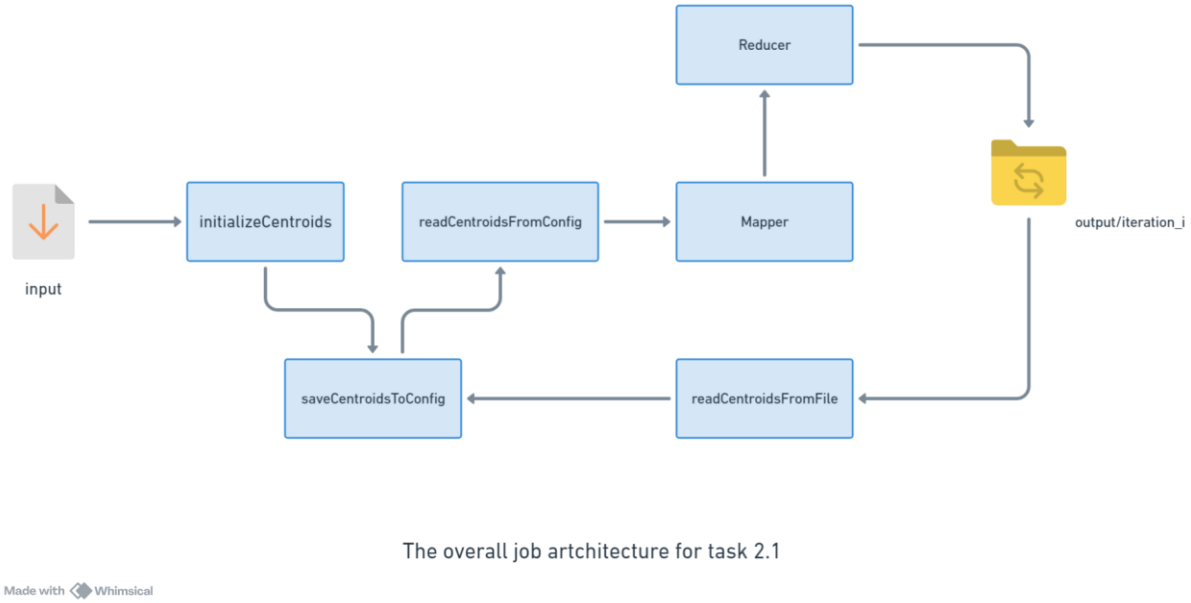


Figure 48: Overall task flow of task 2.1

d. How to run:

- Command to run the program:

```
hadoop jar [file jar] [main class] [input file] [output folder]
```

Figure 49: Command

e. Result:

- File task_2_1.cluster

Figure 50: File task_2_1.clusters

- Part of file task_1_2.classes

Figure 51: Sample of task_2_1.classes

4.2. Task 2.2 K-Means on Preprocessed Data

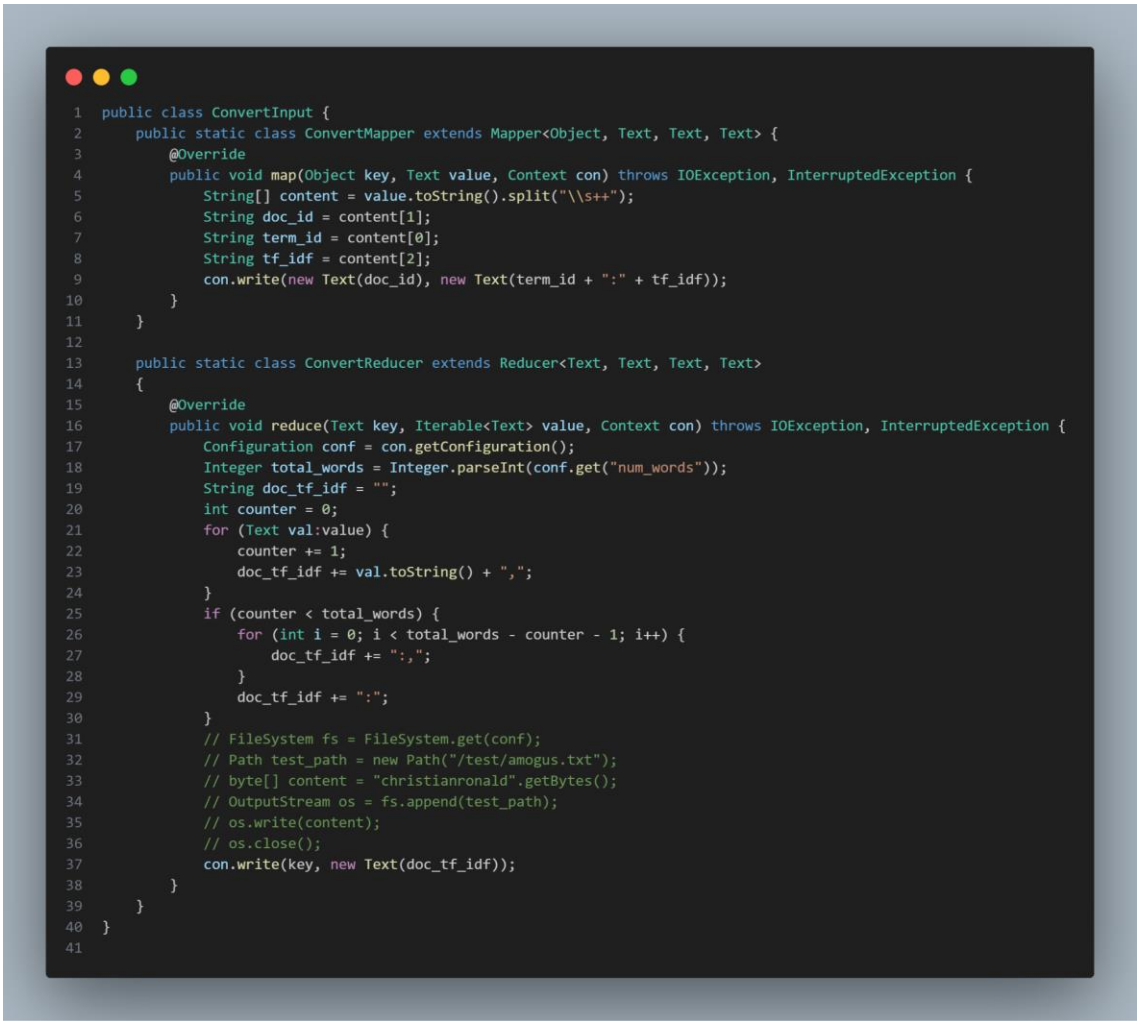
a. Data Description:

- The input data from Task 1.4 consists of a .mtx file containing tf-idf scores for words, which have been filtered by frequency and organized line by line. Due to the complexity of applying MapReduce logic directly to this format, a reformatting stage is necessary to convert the file into a format more suitable for efficient MapReduce processing.
- This reformatting stage involves two sequential MapReduce jobs:

- **Counting Unique Words:** The first MapReduce job processes each line of the input file from Task 1.4 to count the number of unique words. This step is essential as it determines the dimensionality of the vector representing each document in the dataset. The map phase operates similarly to the standard word count example, using the term_id as the key. During the reduce phase, identical term_id values are combined and a global counter variable is incremented. This counter represents the vector dimension, which will be used in subsequent calculations.
- **Conversion to the Desired Format:** The second MapReduce job focuses on converting the input file into a specified format as outlined in the project guidelines. During the map phase, the document_id is used as the key, and the term_id
  pair as the value. The reduce phase then aggregates all term_id
  pairs into a single string, with each pair separated by a comma. The structure of the output file will adhere to the following format:

  ```
  doc_id term_id_1:tf_idf_1,term_id_2:tf_idf_2,...
  ```

- The outputs of these two MapReduce jobs will be stored in a /tmp subdirectory within the output directory. The structure and details of this directory will be elaborated upon in the following section of the report.



```java
public class ConvertInput {
    public static class ConvertMapper extends Mapper<Object, Text, Text, Text> {
        @Override
        public void map(Object key, Text value, Context con) throws IOException, InterruptedException {
            String[] content = value.toString().split("\\s++");
            String doc_id = content[1];
            String term_id = content[0];
            String tf_idf = content[2];
            con.write(new Text(doc_id), new Text(term_id + ":" + tf_idf));
        }
    }

    public static class ConvertReducer extends Reducer<Text, Text, Text, Text>
    {
        @Override
        public void reduce(Text key, Iterable<Text> value, Context con) throws IOException, InterruptedException {
            Configuration conf = con.getConfiguration();
            Integer total_words = Integer.parseInt(conf.get("num_words"));
            String doc_tf_idf = "";
            int counter = 0;
            for (Text val:value) {
                counter += 1;
                doc_tf_idf += val.toString() + ",";
            }
            if (counter < total_words) {
                for (int i = 0; i < total_words - counter - 1; i++) {
                    doc_tf_idf += ":,";
                }
                doc_tf_idf += ":";
            }
            // FileSystem fs = FileSystem.get(conf);
            // Path test_path = new Path("/test/amogus.txt");
            // byte[] content = "christianronald".getBytes();
            // OutputStream os = fs.append(test_path);
            // os.write(content);
            // os.close();
            con.write(key, new Text(doc_tf_idf));
        }
    }
}
```

Figure 52: ConvertInput function

b. MapReduce Job Implementation:

- **In Map Phase:**

- Takes in an object key, Text value, and Context con.
- Splits the input text (value) by whitespace into an array.
- Extracts doc_id, term_id, and tf_idf (term frequency-inverse document frequency) from the array.
- Writes the key-value pair as <Text(doc_id), Text(term_id + ":" + tf_idf)> to the context.

- **In Reduce Phase:**

- Retrieves the total number of words (total_words) from the configuration.
- Iterates over the values associated with a given key (Text type), constructing a string doc_tf_idf by concatenating the values with ";" as a separator.
- If there are fewer values than total_words, it appends additional ";" to ensure the string has the correct format.
- Writes the resulting doc_tf_idf string back to the context associated with the key.

- **KMeans Implementation:**

- The implementation of the K-Means algorithm will follow a structured sequence of steps:
    - **Initialization of Centroids:** The process begins by randomly initializing k centroids.
    - **Assignment of Data Points:** Each data point (document) in the dataset is then assigned to the cluster with the highest cosine similarity between the data point and the centroid.
    - **Centroid Update:** For each cluster, the centroids are recalculated based on the mean position of all data points assigned to that cluster.
    - **Iteration:** Steps 2 and 3 are repeated until the algorithm converges, meaning the centroids no longer change significantly.
- While these steps outline the basic workflow of the K-Means algorithm, understanding its implementation using MapReduce requires further exploration, which is illustrated in the following graphs

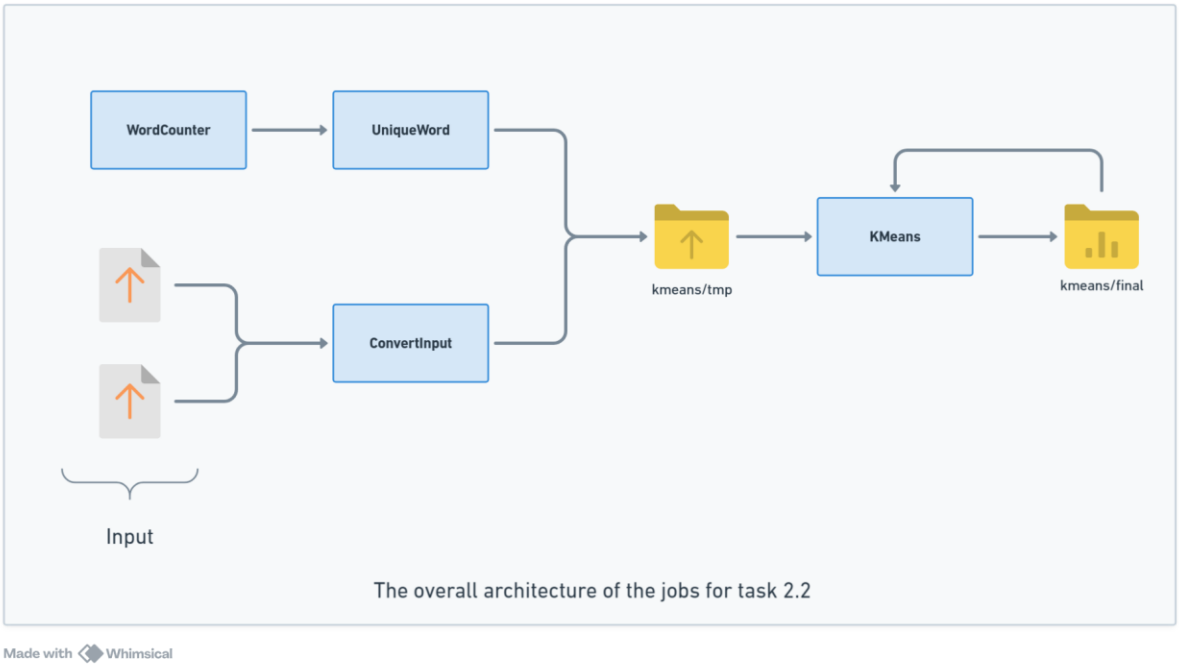


Figure 53: The overall architecture

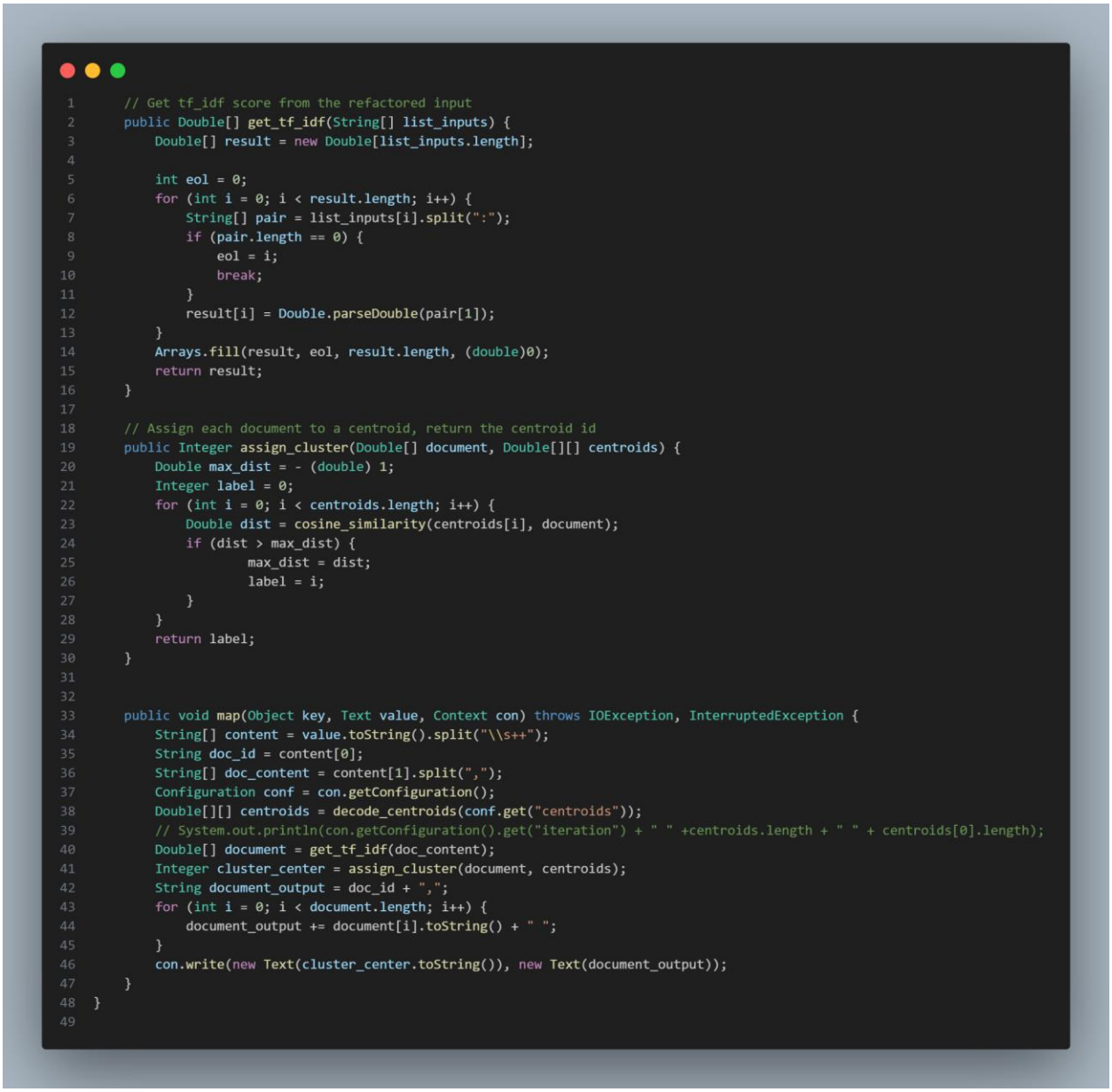
- The diagram illustrates the architecture of the job sequence for Task 2.2, which involves the implementation of the K-Means algorithm using MapReduce. The process begins with three distinct preprocessing jobs: ***UniqueWord***, ***WordCounter***, and ***ConvertInput***. Each of these jobs performs a specific function to prepare the input data.
    - ***UniqueWord*** is responsible for identifying and isolating unique words within the dataset.
    - ***WordCounter*** counts the occurrences of words or documents, providing essential frequency data.
    - ***ConvertInput*** converts the data into a format suitable for subsequent processing by the K-Means algorithm.
- The outputs from these preprocessing jobs are directed into a temporary storage location, represented by the cylinder labeled */kmeans/tmp*. This intermediary

storage acts as a staging area where the preprocessed data is consolidated and made ready for the core K-Means processing.

- The KMeans job then processes the data from */kmeans/tmp*. This job involves the iterative steps of the K-Means algorithm, where data points are clustered, and centroids are recalculated until the algorithm converges.
- Finally, the results of the K-Means processing are stored in the final output location, labeled */kmeans/final*. The diagram also highlights the iterative nature of the K-Means algorithm, as indicated by the feedback loop from the KMeans job back to */kmeans/tmp*, allowing for continuous refinement of the clustering results until convergence is achieved.

```java
public class KMeansMapper extends Mapper<Object, Text, Text, Text>{
    // Returns the 2D array representation of the centroids
    public Double[][] decode_centroids(String centroid_string) {
        String[] initial_centroids = centroid_string.split(",");
        Integer k = initial_centroids.length;
        String[] sample_centroid = initial_centroids[0].split(" ");
        Double[][] centroids = new Double[k][sample_centroid.length];
        for (int i = 0; i < k; i++) {
            String[] ith_centroid = initial_centroids[i].split(" ");
            for (int j = 0; j < sample_centroid.length; j++) {
                centroids[i][j] = Double.parseDouble(ith_centroid[j]);
            }
        }
        return centroids;
    }

    // Utility function for cosine similarity
    public Double vector_length(Double[] vector) {
        Double squared = (Double) 0.0;
        for (int i = 0; i < vector.length; i++) {
            squared += vector[i] * vector[i];
        }

        return (Double) Math.sqrt((double)squared);
    }

    // Calculate cosine similarity between centroid and sample
    public Double cosine_similarity(Double[] vector_a, Double[] vector_b) {
        if (vector_a.length != vector_b.length) {
            return - (Double) 9999.0;
        }
        if (vector_a.equals(vector_b)) {
            return (Double) 1.0;
        }

        Double dot_product = (Double) 0.0;
        Double sum_vector_length = (Double) 0.0;
        for (int i = 0; i < vector_a.length; i++) {
            dot_product += vector_a[i] * vector_b[i];
        }
        sum_vector_length = vector_length(vector_a) + vector_length(vector_b);
        if (sum_vector_length == 0.0)
            return - (Double) 9999.0;
        return (Double) dot_product / sum_vector_length;
    }
```

Figure 54: Map Phase



```java
    // Get tf_idf score from the refactored input
    public Double[] get_tf_idf(String[] list_inputs) {
        Double[] result = new Double[list_inputs.length];

        int eol = 0;
        for (int i = 0; i < result.length; i++) {
            String[] pair = list_inputs[i].split(":");
            if (pair.length == 0) {
                eol = i;
                break;
            }
            result[i] = Double.parseDouble(pair[1]);
        }
        Arrays.fill(result, eol, result.length, (double)0);
        return result;
    }

    // Assign each document to a centroid, return the centroid id
    public Integer assign_cluster(Double[] document, Double[][] centroids) {
        Double max_dist = - (double) 1;
        Integer label = 0;
        for (int i = 0; i < centroids.length; i++) {
            Double dist = cosine_similarity(centroids[i], document);
            if (dist > max_dist) {
                    max_dist = dist;
                    label = i;
            }
        }
        return label;
    }


    public void map(Object key, Text value, Context con) throws IOException, InterruptedException {
        String[] content = value.toString().split("\\s++");
        String doc_id = content[0];
        String[] doc_content = content[1].split(",");
        Configuration conf = con.getConfiguration();
        Double[][] centroids = decode_centroids(conf.get("centroids"));
        // System.out.println(con.getConfiguration().get("iteration") + " " +centroids.length + " " + centroids[0].length);
        Double[] document = get_tf_idf(doc_content);
        Integer cluster_center = assign_cluster(document, centroids);
        String document_output = doc_id + ",";
        for (int i = 0; i < document.length; i++) {
            document_output += document[i].toString() + " ";
        }
        con.write(new Text(cluster_center.toString()), new Text(document_output));
    }
}
```

Figure 55: Map Phase

Define decode_centroids function:

Input: A string representing the centroids, separated by commas.

Output: A 2D array of Double representing the centroids.

Steps:

- Split the centroid string by commas to get individual centroids.
- Determine the number of centroids (k) and the dimensionality of each centroid (from the first centroid).

- Initialize a 2D array centroids of size k x dimensionality.
- For each centroid:
    - Split the centroid string by spaces to get its components.
    - Parse each component to Double and store it in the 2D array.
- Return the 2D array centroids.

Define vector_length function:

Input: A vector of Double.

Output: The length (magnitude) of the vector.

Steps:

- Initialize squared as 0.
- For each component in the vector:
- Add the square of the component to squared.
- Return the square root of squared.

Define cosine_similarity function:

Input: Two vectors of Double, vector_a and vector_b.

Output: The cosine similarity between the two vectors.

Steps:

- If the lengths of the two vectors are not equal, return an error value (-9999.0).
- If the two vectors are identical, return a similarity of 1.0.
- Initialize dot_product as 0.
- For each pair of components in the two vectors:
- Multiply the components and add the result to dot_product.
- Calculate the sum of the lengths of the two vectors.
- If the sum of the lengths is 0, return an error value (-9999.0).
- Return the ratio of dot_product to the sum of the vector lengths as the cosine similarity.

Define get_tf_idf function:

Input: An array of strings, each representing a term and its tf-idf value separated by a colon.

Output: An array of Double representing the tf-idf values.

Steps:

- Initialize an array result of Double of the same length as the input.
- For each string in the input:
- Split the string by the colon to separate the term and the tf-idf value.
- If splitting fails, break the loop and fill the rest of the result array with 0.
- Parse the tf-idf value to Double and store it in the result array.
- Return the result array.

Define assign_cluster function:

Input: A Double array representing a document, and a 2D array of Double representing the centroids.

Output: An integer representing the index of the closest centroid.

Steps:

- Initialize max_dist as -1 and label as 0.
- For each centroid:
- Calculate the cosine similarity between the centroid and the document.
- If the similarity is greater than max_dist, update max_dist and set label to the current centroid index.
- Return the label.

Define map function:

Input: A key, a text value (representing a document), and a context object.

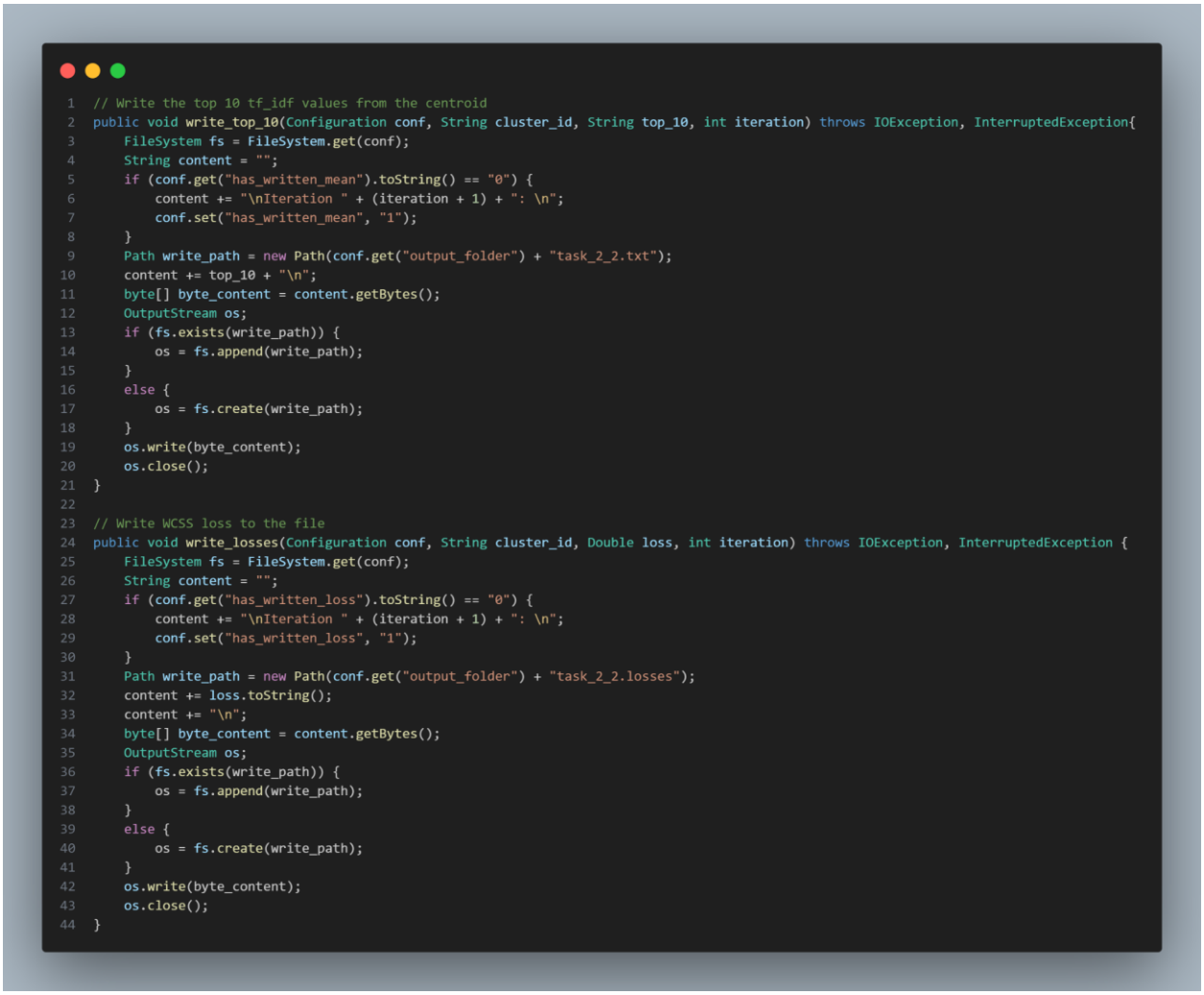
Steps:

- Split the value into the document ID and document content.
- Get the centroids from the configuration using decode_centroids.
- Convert the document content to an array of tf-idf values using get_tf_idf.
- Assign the document to the closest centroid using assign_cluster.

- Prepare the document output as a string with the document ID followed by its tf-idf values.
- Write the centroid index and document output to the context.

```java
public Double[] convert_string(String document_vector) {
    String[] values = document_vector.split(" ");
    Double[] new_vec = new Double[values.length];
    for (int i = 0; i < new_vec.length; i++) {
        new_vec[i] = Double.parseDouble(values[i]);
    }
    return new_vec;
}

// Write the centroids to an output file, deleting the old iterations every new iteration
public void write_centroids(Configuration conf, String cluster_id, String centroids, int iteration) throws IOException, InterruptedException {
    FileSystem fs = FileSystem.get(conf);
    Path old_path = new Path(conf.get("output_folder") + "task_2_2_iter" + (iteration - 1) + ".clusters");
    if (fs.exists(old_path)) {
        fs.delete(old_path, true);
    }
    Path write_path;
    if (iteration == (Integer.parseInt(conf.get("max_iteration"))) - 1) {
        write_path = new Path(conf.get("output_folder") + "task_2_2.clusters");
    }
    else {
        write_path = new Path(conf.get("output_folder") + "task_2_2_iter" + iteration + ".clusters");
    }
    String content = cluster_id + "\t" + centroids + "\n";
    byte[] byte_content = content.getBytes();
    OutputStream os;
    if (fs.exists(write_path)) {
        os = fs.append(write_path);
    }
    else {
        os = fs.create(write_path);
    }
    os.write(byte_content);
    os.close();
}

// Write the document id and their assigned cluster id to the file, deleting the old iterations every new iteration
public void write_document_clustering(Configuration conf, String cluster_id, String doc_id, int iteration) throws IOException, InterruptedException {
    FileSystem fs = FileSystem.get(conf);
    Path old_path = new Path(conf.get("output_folder") + "task_2_2_iter" + (iteration - 1) + ".classes");
    if (fs.exists(old_path)) {
        fs.delete(old_path, true);
    }
    Path write_path;
    if (iteration == (Integer.parseInt(conf.get("max_iteration")) - 1)) {
        write_path = new Path(conf.get("output_folder") + "task_2_2.classes");
    }
    else {
        write_path = new Path(conf.get("output_folder") + "task_2_2_iter" + iteration + ".classes");
    }
    String content = "";
    String[] list_docs = doc_id.split(" ");
    for (int i = 0; i < list_docs.length; i++) {
        content += list_docs[i] + " " + cluster_id + "\n";
    }
    byte[] byte_content = content.getBytes();
    OutputStream os;
    if (fs.exists(write_path)) {
        os = fs.append(write_path);
    }
    else {
        os = fs.create(write_path);
    }
    os.write(byte_content);
    os.close();
}
```

Figure 56: Reduce Phase



```java
// Write the top 10 tf_idf values from the centroid
public void write_top_10(Configuration conf, String cluster_id, String top_10, int iteration) throws IOException, InterruptedException{
    FileSystem fs = FileSystem.get(conf);
    String content = "";
    if (conf.get("has_written_mean").toString() == "0") {
        content += "\nIteration " + (iteration + 1) + ": \n";
        conf.set("has_written_mean", "1");
    }
    Path write_path = new Path(conf.get("output_folder") + "task_2_2.txt");
    content += top_10 + "\n";
    byte[] byte_content = content.getBytes();
    OutputStream os;
    if (fs.exists(write_path)) {
        os = fs.append(write_path);
    }
    else {
        os = fs.create(write_path);
    }
    os.write(byte_content);
    os.close();
}

// Write WCSS loss to the file
public void write_losses(Configuration conf, String cluster_id, Double loss, int iteration) throws IOException, InterruptedException {
    FileSystem fs = FileSystem.get(conf);
    String content = "";
    if (conf.get("has_written_loss").toString() == "0") {
        content += "\nIteration " + (iteration + 1) + ": \n";
        conf.set("has_written_loss", "1");
    }
    Path write_path = new Path(conf.get("output_folder") + "task_2_2.losses");
    content += loss.toString();
    content += "\n";
    byte[] byte_content = content.getBytes();
    OutputStream os;
    if (fs.exists(write_path)) {
        os = fs.append(write_path);
    }
    else {
        os = fs.create(write_path);
    }
    os.write(byte_content);
    os.close();
}
```

Figure 57: Reduce Phase

```java
public Double calculate_loss(Double[] data, Double[] centroid) {
    Double wcss = (double) 0;
    for (int i = 0; i < data.length; i++) {
        wcss += Math.pow(centroid[i] - data[i], 2);
    }
    return wcss;
}

public String get_top_10(Double[] centroid) {
    TreeMap<Double, Integer> top_10 = new TreeMap<Double, Integer>();
    for (int i = 0; i < centroid.length; i++) {
        top_10.put(centroid[i], i);
    }
    String top_10_mean = "";
    NavigableMap<Double, Integer> descending_top_10 = top_10.descendingMap();
    Iterator tmp = descending_top_10.entrySet().iterator();
    for (int i = 0; i < 10; i++) {
        top_10_mean += tmp.next().toString() + ", ";
    }
    return top_10_mean;
}


public void reduce(Text key, Iterable<Text> value, Context con) throws IOException, InterruptedException {
    Configuration conf = con.getConfiguration();
    Integer iteration = Integer.parseInt(conf.get("iteration"));
    String old_centroids = conf.get("centroids");
    String[] num_centroids = old_centroids.split(",");
    int elements = num_centroids[0].split(" ").length;
    Double[] new_centroid = new Double[elements];
    Arrays.fill(new_centroid, Double.valueOf(0));
    int counter = 0;
    String list_docs = "";
    String cluster_id = key.toString();
    Double[] data = new Double[elements];
    Double wcss = (double) 0;
    for (Text v: value) {
        String[] true_val = v.toString().split(",");
        String doc_id = true_val[0];
        list_docs += doc_id + " ";
        String doc_content = true_val[1];
        Double[] tmp = convert_string(doc_content);
        for (int i = 0; i < tmp.length; i++) {
            data[i] = tmp[i];
        }
        counter += 1;
        for (int i = 0; i < new_centroid.length; i++) {
            new_centroid[i] += tmp[i];
        }
        wcss += calculate_loss(data, new_centroid);
    }
    String new_centroid_str = "";
    for (int i = 0; i < new_centroid.length; i++) {
        new_centroid[i] /= (counter + 1);
        new_centroid_str += new_centroid[i].toString() + " ";
    }
    String top_10 = get_top_10(new_centroid);
    write_centroids(conf, cluster_id, new_centroid_str, iteration);
    write_document_clustering(conf, cluster_id, list_docs, iteration);
    write_losses(conf, cluster_id, wcss, iteration);
    write_top_10(conf, cluster_id, top_10, iteration);
    con.write(key, new Text(new_centroid_str));
}
```

Figure 58: Reduce Phase

Define convert_string function:

Input: A string representing a document vector with values separated by spaces.

Output: An array of Double representing the document vector.

Steps:

- Split the input string by spaces to get individual values.
- Initialize a Double array of the same length as the values.
- Convert each value to Double and store it in the array.
- Return the Double array.

Define write_centroids function:

Input: Configuration object, cluster ID, centroid string, and iteration number.

Output: Writes centroids to an output file.

Steps:

- Initialize the file system using the configuration.
- Delete the old centroid file if it exists (from the previous iteration).
- Determine the write path based on the current iteration:
- If it's the last iteration, use the final cluster file path.
- Otherwise, use the iteration-specific file path.
- Convert the centroid data to bytes.
- Write the centroid data to the determined path, either appending to the file if it exists or creating a new file.
- Close the file output stream.

Define write_document_clustering function:

Input: Configuration object, cluster ID, document ID, and iteration number.

Output: Writes the document ID and their assigned cluster ID to a file.

Steps:

- Initialize the file system using the configuration.
- Delete the old document clustering file if it exists (from the previous iteration).
- Determine the write path based on the current iteration:

- If it's the last iteration, use the final document cluster file path.
- Otherwise, use the iteration-specific file path.
- For each document ID in the list, prepare content by appending the document ID and cluster ID.
- Convert the content to bytes.
- Write the document clustering data to the determined path, either appending to the file if it exists or creating a new file.
- Close the file output stream.

Define write_top_10 function:

Input: Configuration object, cluster ID, top 10 TF-IDF values string, and iteration number.

Output: Writes the top 10 TF-IDF values for centroids to a file.

Steps:

- Initialize the file system using the configuration.
- If not already noted, write the iteration header to the file (only once per iteration).
- Determine the file path for writing top 10 values.
- Convert the top 10 data to bytes.
- Write the top 10 data to the determined path, either appending to the file if it exists or creating a new file.
- Close the file output stream.

Define write_losses function:

Input: Configuration object, cluster ID, loss value, and iteration number.

Output: Writes the Within-Cluster Sum of Squares (WCSS) loss to a file.

Steps:

- Initialize the file system using the configuration.
- If not already noted, write the iteration header to the file (only once per iteration).
- Determine the file path for writing loss values.
- Convert the loss data to bytes.

- Write the loss data to the determined path, either appending to the file if it exists or creating a new file.
- Close the file output stream.

Define calculate_loss function:

Input: Document vector and centroid vector as arrays of Double.

Output: The WCSS loss (a Double value).

Steps:

- Initialize wcss as 0.
- For each component in the document vector and corresponding component in the centroid vector:
    - Calculate the squared difference between the centroid and document components and add it to wcss.
- Return the wcss value.

Define get_top_10 function:

Input: Centroid vector as an array of Double.

Output: A string representing the top 10 TF-IDF values from the centroid.

Steps:

- Initialize a TreeMap to store centroid values and their indices.
- For each value in the centroid vector, add it to the TreeMap.
- Get a descending order of the TreeMap and iterate through it.
- Extract the top 10 values from the map and convert them to a string format.
- Return the formatted string of the top 10 values.

Define reduce function:

Input: Key (Text), Iterable of values (Text), and context object.

Output: Writes the updated centroid information and other metrics.

Steps:

- Initialize the configuration and retrieve the current iteration.
- Get the old centroids from the configuration.
- Determine the number of elements in each centroid.

- Initialize a new centroid array and set all elements to 0.
- Initialize a counter and an empty string for storing document IDs.
- Initialize a WCSS value to 0.
- For each value in the input:
    - Parse the document ID and its content.
    - Convert the content string to a document vector.
    - Update the new centroid by adding the document vector components to it.
    - Increment the counter.
    - Calculate the WCSS loss and add it to the total WCSS.
- Calculate the final centroid by dividing the sum by the number of documents.
- Convert the centroid to a string format.
- Get the top 10 TF-IDF values from the centroid.
- Write the centroid, document clustering, WCSS loss, and top 10 TF-IDF values to the appropriate files.
- Write the new centroid information to the context.

c. Task Flow:

- The **map phase** operates as follows:

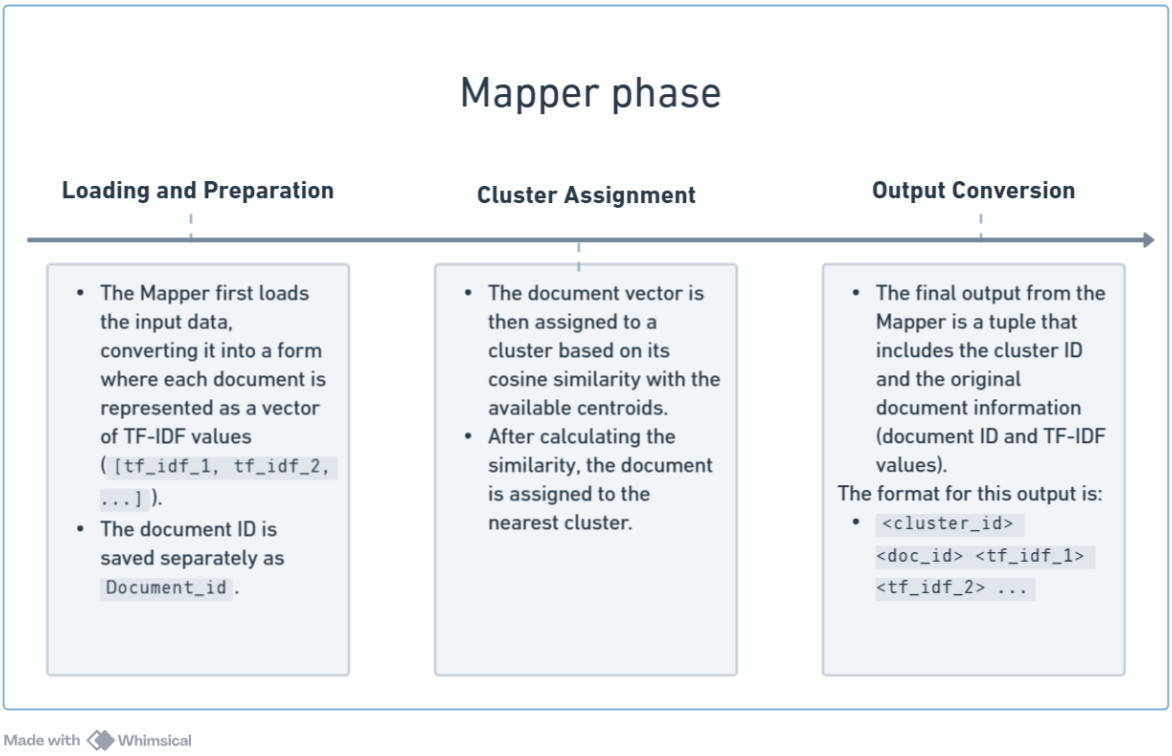


Figure 59: Maper phase for task 2.2

- In the initial iteration, the process begins by using randomly generated centroids. These centroids are serialized into a string and stored in the job configuration as a variable, enabling the job to keep track of centroid coordinates. After reading the input produced by the conversion job, the map function deserializes the string into an array of Doubles. It then calculates the similarity between each document and the centroids, assigning each document to a cluster. The cluster is used as the key for the reduce phase, and the value is formatted as: cluster_id doc_id tf_idf_1 tf_idf_2 ....

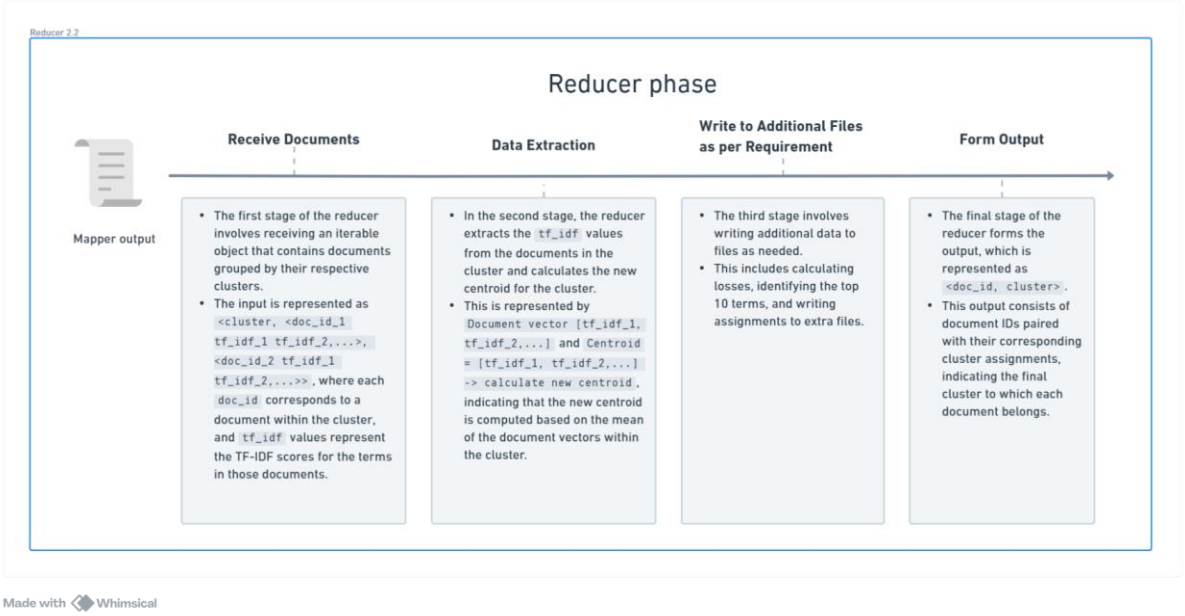


Figure 60: Reducer phase for task 2.2

- The reduce phase, which processes the output from the map phase, functions as follows: Upon receiving all documents belonging to the same cluster, the reduce phase iterates through them, adds them to a new centroid array, and keeps track of the document count. The new centroid array is then divided by this count, completing the centroid updating step described in the general plan. The losses are calculated as the sum of all squared Euclidean distances between each document in the cluster and the newly formed centroid. The top 10 terms are also identified, sorted, and written to a separate file. During this phase, all four required output files are generated without using the context.write function provided by MapReduce, in order to minimize follow-up tasks. However, a minor issue arises with the top 10 terms task, as no terms are transferred from the map phase to the reduce phase. This results in the output file *task_2_2.txt* only containing the index of words in the document vector, requiring an additional step to trace back to the actual term index.
- From the second iteration onward, the centroid configuration variable is updated using the *task_2_2.clusters* file. Each iteration has its own output directory,

allowing users to easily monitor the execution process. However, the final four output files can be found in the /kmeans/final/ directory as configured by the program.

d. How to run

Figure 61: Command

e. Result

Figure 62: task_2_2.cluster

Figure 63: task_2_2.loss

Figure 64: task_2_2.classes

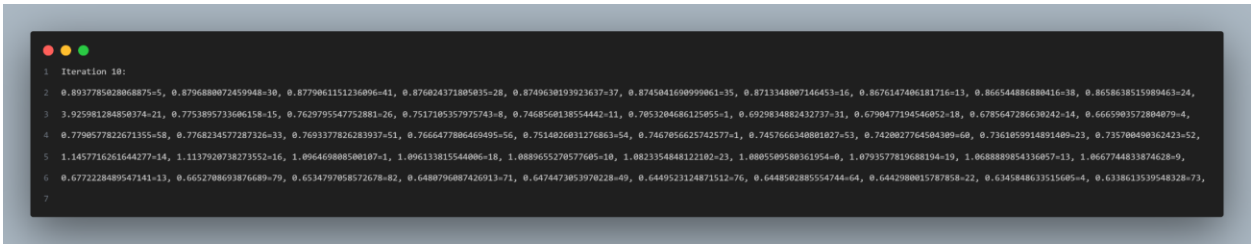


Figure 65: task_2_2.txt

4.3. Task 2.3 Scalable K-Means++ Initialization

a. Data description:

- In this task 2.3, we used the same method to process data as task 2.2

b. MapReduce Job Implementation:

- The K-Means++ initialization algorithm can be summarized as follows:
  - **Random Sampling:** A random sample point is selected from the entire dataset, referred to as X. This sample is added to a set C and subsequently removed from the dataset.

    Note: Each time a point is added to set C, it is removed from the dataset.
  - **Distance Calculation**: The distance between every point in X and each point in C is calculated. These distances are then summed and stored in a variable, such as ***sum_distances***.
  - **Iterative Sampling:** A for-loop is initiated, with the number of iterations set according to the value of ***log(sum_distances)***:
    a. Each point x_i in X is independently sampled with a probability defined by $\frac{l \times minDist(x_i, C)}{\sum distances}$.
    b. Then, the sampled points are added to set C.
  - **Weight Calculation:** An empty array w, matching the size of set C, is created. For each sample point in X, the closest centroid in C is identified, and the corresponding index (e.g., j) in w is incremented by 1.

- **Centroid Selection:** The weighted array w is returned, and to select k centroids, k elements are chosen from C, sorted in descending order.

c. Challenges and Solutions:

- A significant challenge arises when handling large datasets, as the algorithm requires calculating the distance between every point in X and the potential centroids in C. A simple solution might involve generating a random key and processing all data from the map phase on a single node during the reduce phase. However, this approach is unsuitable for larger datasets due to the risk of memory overflow.
- To mitigate this issue, an ensemble technique was implemented. The job configuration allows for specifying the number of nodes (e.g., 5 nodes) to work on this task. During the map phase, each line of data is assigned a random key, ranging from 0 to the number of configured nodes. This approach effectively splits dataset X into smaller subsets, enabling each node to process its mini-dataset independently. After completing their respective reduce tasks, which involve applying the K-Means|| algorithm to the smaller datasets, the output from each node is compiled into a single file. The results are then averaged to obtain the desired k centroids.
- A visual representation of the full MapReduce process for this task is provided in the accompanying graph.

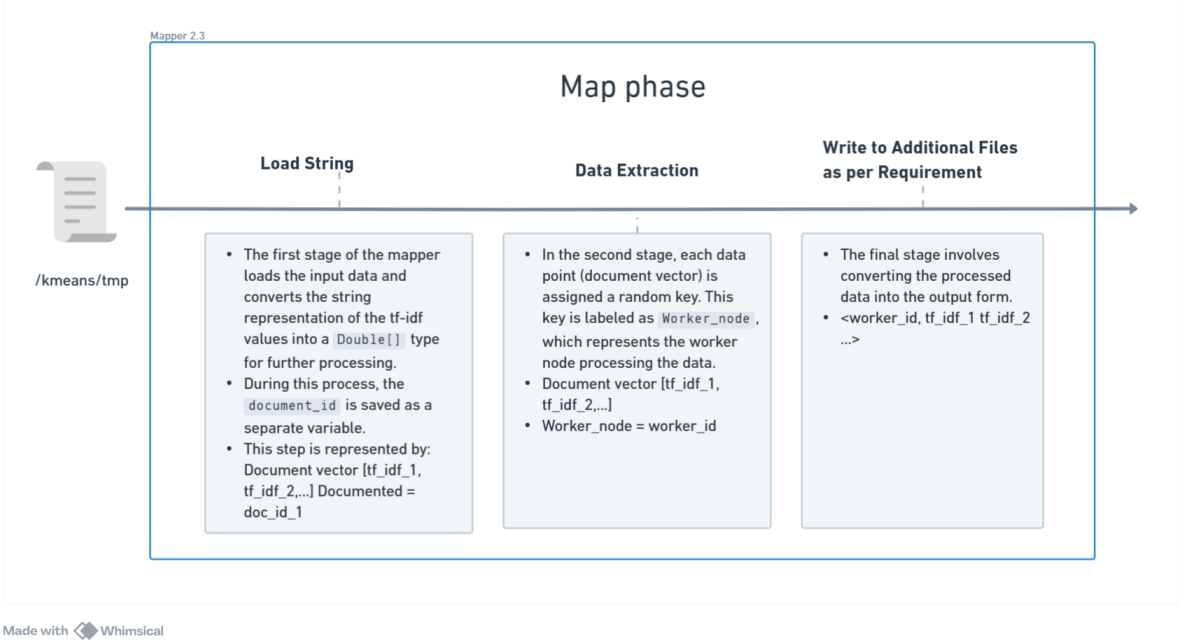


Figure 66: Map Phase for task 2.3

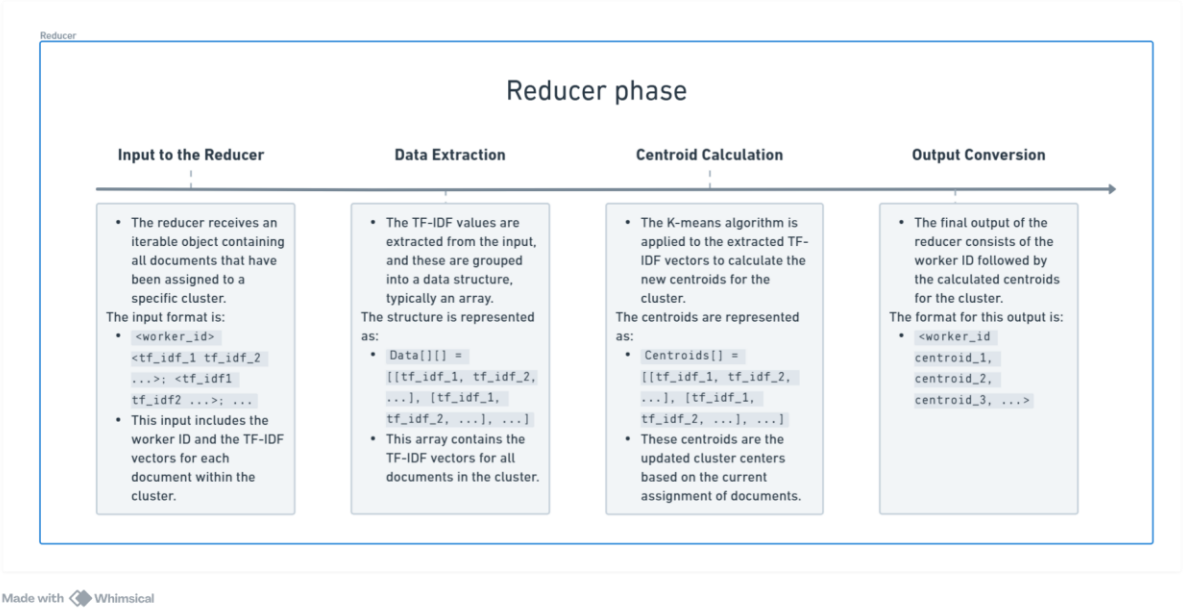


Figure 67: Reducer Phase for task 2.3

d. How to run:

```
$(sudo) hadoop jar /path/to/jar package.path.to.KMeans /path/to/input /path/to/output.
```

Figure 68: Command

e. Result:

```
Iteration 8:
5.901838850016779E9
2.230024966961907E10
1.2941278959564645E7
149116.60700060264
135761.18430515437

Iteration 9:
5.616679424753652E9
2.2341630846910957E10
1.6060629475042786E7
143602.26711959386
177725.7443308732

Iteration 10:
5.529911594810488E9
2.0789031648855812E10
3.706785827790922E7
124220.28765376497
199968.42493400522
```

Figure 9: task_2_3.loss

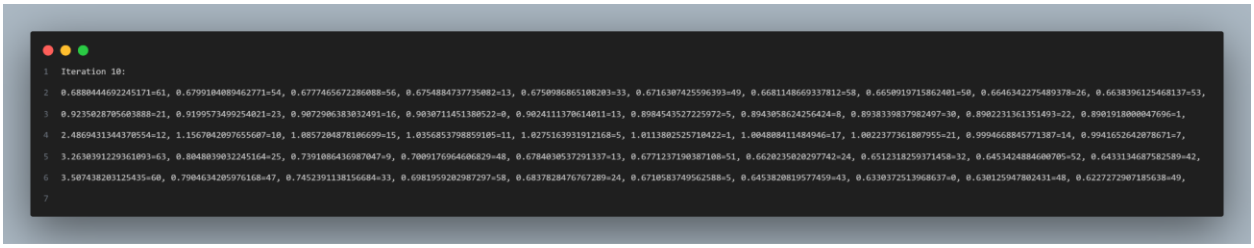


Figure 70: task_2_3.txt

5. Conclusion

Through this lab exercise, our group significantly deepened our understanding of the **MapReduce mechanism** and its **practical applications** in solving complex problems.

Initially, we encountered **challenges**, particularly with the programming aspects of MapReduce and the scarcity of related documentation. However, by engaging in **collaborative discussions** and **meticulously studying the MapReduce framework**, we were able to overcome these difficulties. As we became more familiar with the concepts, the implementation process became progressively smoother.

In summary, this lab was invaluable for our group. It not only enhanced our **comprehension of data processing techniques** but also provided us with hands-on experience in **implementing the K-means clustering algorithm using Hadoop MapReduce.** The skills and insights gained from this exercise will undoubtedly benefit us in future projects involving large-scale data processing.

6. References

[1] Apache Hadoop MapReduce Tutorial

[2] MapReduce Algorithms for k-means Clustering

[3] How to calculate TF-IDF with MapReduce - Slogix