DAS 839: NoSQL Systems
Assignment-2

MArch 24, 2025

Submitted by:

Group 15

1. Siddharth Ayathu (IMT2022517)
2. Shreyank G Bhat (IMT2022516)
3. Aayush Bhargav (IMT2022089)
4. Praveen Peter Jay (IMT2022064)

Problem 1

1. Do you think the M-Counter state-based object (SBO) qualifies as a CRDT? Justify your answer by considering the CRDT properties. (3 marks)

A. To determine if the M-Counter qualifies as a CRDT, we need to compare it against the key CRDT properties:

(a) The M-Counter's merge function works by taking the maximum value from each corresponding element in the array. This ensures that values never decrease over time, which is essential for maintaining a monotonic state. This behavior forms a join-semilattice, a key requirement for state-based CRDTs. Therefore, as updates are merged, the system moves toward a consistent state without ever rolling back or losing progress.

(b) The merge operation in the M-Counter is:

- Commutative: For any two state-based objects x and y, the merge function satisfies:

$$\text{merge}(x, y) = \text{merge}(y, x)$$

  This holds because the pairwise maximum operation is commutative (i.e., $\max(a, b) = \max(b, a)$). Thus, M-Counter satisfies commutativity.

- Associative: the merge operation works by taking the maximum value for each corresponding element in the state arrays, and the maximum operation itself is associative. Given three state-based objects x, y, and z:

$$\text{merge}(\text{merge}(x, y), z) = \text{merge}(x, \text{merge}(y, z))$$

  Since the maximum operation is associative (i.e., $\max(\max(a, b), c) = \max(a, \max(b, c))$), M-Counter satisfies associativity.

- Idempotent: For any state-based object x, if you merge it with itself, nothing changes. The result is just the original state:

$$\text{merge}(x, x) = x$$

  Since the maximum of any value with itself remains unchanged ($\max(a, a) = a$), the merge operation is idempotent, satisfying this property.

- Increasing Updates:The update operation add(x) only increases a specific index in the internal array. Let x be the current state and y be the state after an update:

$$\text{merge}(x, y) = y$$

  Since merge takes the maximum values element-wise, and updates only increment specific elements, the update method is increasing.

Since the M-Counter satisfies associativity, commutativity, idempotency, and increasing updates, it meets all the necessary conditions to be classified as a state-based CRDT (SBO CRDT).

(c) Eventual Consistency: Because every replica eventually merges updates from others using the merge function, all replicas will converge to the same state over time. This guarantees that the system will achieve consistency without requiring immediate synchronization.

Thus, the M-Counter satisfies CRDT properties and qualifies as a state-based (SBO) grow-only counter (G-Counter) CRDT.

2. Consider the following state diagram and populate the state table with the fields: state, query, and history. (Refer to the lecture slides on LMS for an explanation of the notations used in the diagram.) (3 marks)

A. Regarding the notation of the given state diagram:

- Each node represents the state of the object at a particular server at a specific instance of time.
- The state of an object is given by three variables: i, n and xs.
- a and b are two servers having a copy of the object, and their indices indicate how recently this object at that server has been changed (higher the index, newer it is).
- *add*() and *merge*() operations occur as defined in the question.
- On querying from a server at a particular state, the server returns the value of the sum of elements in its version of xs, as defined.
- History at a particular node keeps track of the identifiers of the add operations that have contributed to its current state.

We can now populate the state table as follows:

State	State Representation (i, n, xs)	query()	History
a_0	i: 0, n: 2, xs = [0, 0]	0	$\{\}$
a_1	i: 0, n: 2, xs = [1, 0]	1	$\{0\}$
a_2	i: 0, n: 2, xs = [1, 4]	5	$\{0, 1\}$
b_0	i: 1, n: 2, xs = [0, 0]	0	$\{\}$
b_1	i: 1, n: 2, xs = [0, 2]	2	$\{1\}$
b_2	i: 1, n: 2, xs = [0, 6]	6	$\{1, 2\}$
b_3	i: 1, n: 2, xs = [1, 6]	7	$\{0, 1, 2\}$

Table 1: State table for the given state diagram

3. Should CRDTs guarantee formal correctness (e.g., convergence and consistency) or ensure that their behavior aligns with intuitive application semantics? Discuss the trade-offs between correctness and application semantics. (2 marks)

A. When designing CRDTs, two key considerations come into play:

- Formal Correctness (Convergence & Consistency): This ensures that all replicas eventually reach the same state, which is critical for distributed systems. Without this, the system could become inconsistent or unpredictable.
- Application Semantics: This ensures that the CRDT behaves in a way that aligns with real-world expectations. For example, a counter should accurately reflect the number of increments performed, even in a distributed environment.
- **Trade-offs**:
  - Prioritizing formal correctness: While this guarantees convergence, it might lead to behaviors that seem counter-intuitive in real-world applications. For instance, certain edge cases might produce results that don't align with user expectations.
  - Prioritizing application semantics: This might make the system more intuitive for users, but it could introduce inconsistencies or make it harder to ensure convergence.

In practice, a balance is necessary. Strong guarantees on convergence are essential, but the application logic should also be designed to handle potential anomalies or edge cases in a way that makes sense for users.

Problem 2

Q. In this problem, you are required to develop a MapReduce program to process Wikipedia articles and simulate a collaborative editing scenario.

The program should read each article from the Wikipedia dump available on the LMS: **Wikipedia-EN-2010601-ARTICLES.tar.gz**. For each word in the article, the mapper should emit a key-value pair, where the key represents the document ID, and the value is a pair containing the index position of the word in the document and the word itself.

The reducer will group the incoming key-value pairs based on the document ID and emit key-value pairs in the form:

(index, (doc-ID, word))

You should consider using at least three reducers in your deployment.

If you observe the output of the reducer, it simulates a collaborative editing environment. If each document is thought of as a different version or snapshot of the same document, with the document ID acting as the snapshot ID, then each key-value pair emitted by the reducer would indicate the index position and the respective changes that occurred at that index across different timestamps.

To develop the code, you may use the smaller dataset **Wikipedia-50-ARTICLES.tar** shared on the LMS. Be mindful that processing the full Wikipedia dump will require significant time; plan accordingly to ensure efficient execution of the MapReduce job.

A. Kindly refer the file `Problem2_re.java`. Given below is an explanation of the code, and how it solves the given problem. Note: To set up Hadoop, refer the appendix at the end of this document.

1. TokenizerMapper (Mapper Class)

Description:

- Reads input files and tokenizes each line into words.
- Extracts the document ID from the filename and assigns an index to each word in the document, based on the position at which it is read.
- Utilizes the custom *map*() function defined in it to achieve intended results.

Functionality:

- Retrieves the filename from the Hadoop input split and extracts the document ID (`doc_id`).
- Splits the text when a series of tab and/or new line characters are encountered.
- In `Problem2.java`, we originally split the text into tokens using a regex, which removes all non-word characters except apostrophes (word characters include letters, numeric digits and underscores). We had to modify this approach as the output of this code was resulting in incorrect results for our solution for Problem 3, which is described later.
- Iterates over the tokens (words) while incrementing an index and creates a `PairWritable` object containing `(index, word)`.
- Emits `(doc_id, (index, word))` as key-value pairs.

2. IntSumReducer (Reducer Class)

Description:

- Processes grouped key-value pairs received from the mapper and reorganizes them by word index, essentially swapping it with `doc_id`.
- Utilizes the custom *reduce*() function defined in it to achieve intended results.

Functionality:

- Iterates over all words and their positions within the same document.
- Extracts the index and word from each `PairWritable` object.
- Emits a key-value pair where index is the key and `(doc_id, word)` is the value.
- Results in words sorted by their positions across documents.

3. PairWritable (Custom Writable Class)

Description:

- A custom Hadoop Writable class used for mapper output, to store a word and its index.

Functionality:

- Uses a constructor to generate an object containing `IntWritable` (index) and a `Text` (word).

4. PairWritableReducer (Custom Writable Class)

Description:

- A custom Hadoop Writable class used for reducer output, to store a word and its doc_id.

Functionality:

- Uses a constructor to generate an object containing `Text` (doc_id) and a `Text` (word).

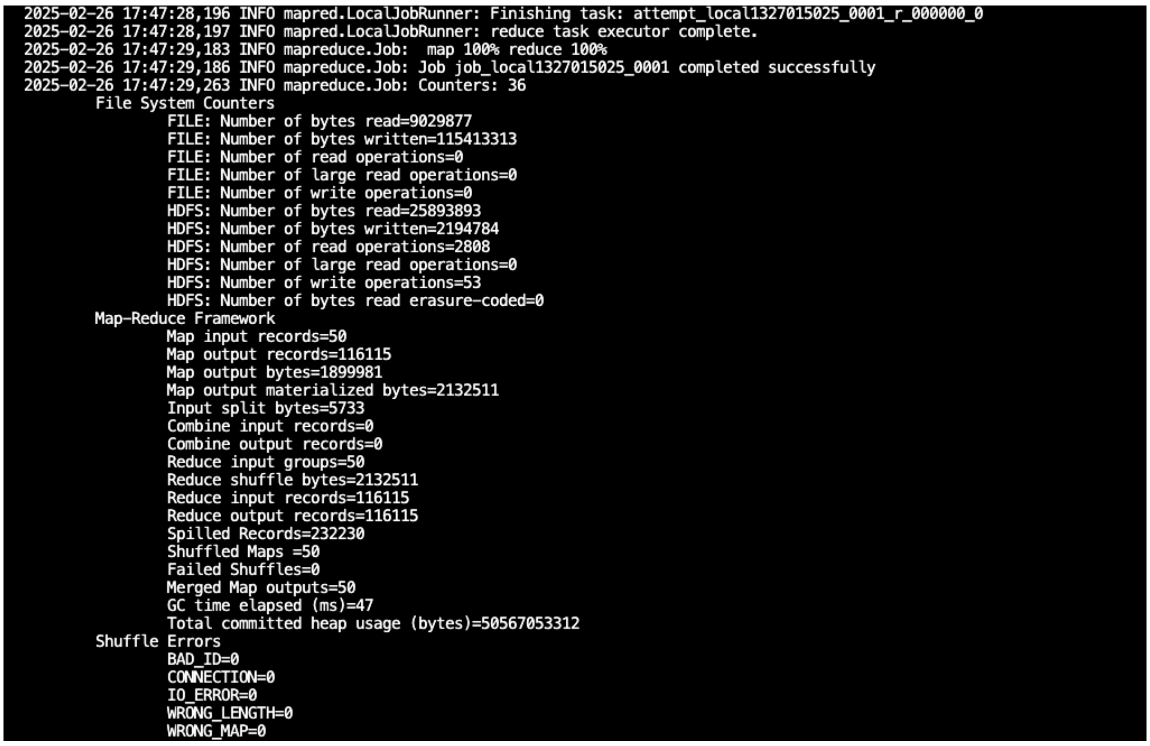
5. main() (Driver Program)

Description:

- Configures and runs the MapReduce job.

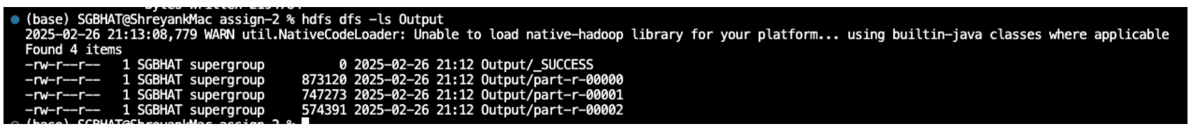
Functionality:

- Sets up a Hadoop configuration and creates a new MapReduce job named `"wordcount"`.
- Specifies the Mapper (`TokenizerMapper`) and Reducer (`IntSumReducer`) classes.
- Defines key-value output types:
  - **Mapper**
    - **Input:** `(Object, Text)`
    - **Output:** `(Text, PairWritable)` (Key: Document ID, Value: (Index, Word))
  - **Reducer**
    - **Input:** `(Text, Iterable<PairWritable>)`
    - **Output:** `(IntWritable, PairWritableReducer)` (Key: Index, Value: (Document ID, Word))
- Sets the number of reducers to 3 (`job.setNumReduceTasks(3)`).
- Specifies input and output paths using `FileInputFormat` and `FileOutputFormat`.
- Submits the job to the Hadoop cluster and waits for its completion.



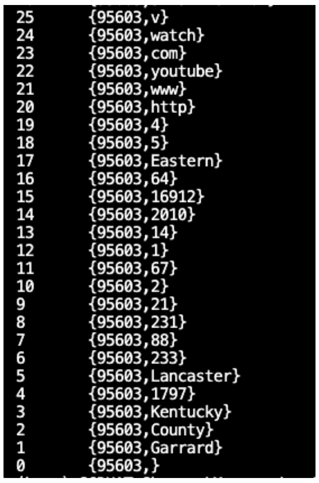
```
2025-02-26 17:47:28,196 INFO mapred.LocalJobRunner: Finishing task: attempt_local1327015025_0001_r_000000_0
2025-02-26 17:47:28,197 INFO mapred.LocalJobRunner: reduce task executor complete.
2025-02-26 17:47:29,183 INFO mapreduce.Job:  map 100% reduce 100%
2025-02-26 17:47:29,186 INFO mapreduce.Job: Job job_local1327015025_0001 completed successfully
2025-02-26 17:47:29,263 INFO mapreduce.Job: Counters: 36
        File System Counters
                FILE: Number of bytes read=9029877
                FILE: Number of bytes written=115413313
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=25893893
                HDFS: Number of bytes written=2194784
                HDFS: Number of read operations=2808
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=53
                HDFS: Number of bytes read erasure-coded=0
        Map-Reduce Framework
                Map input records=50
                Map output records=116115
                Map output bytes=1899981
                Map output materialized bytes=2132511
                Input split bytes=5733
                Combine input records=0
                Combine output records=0
                Reduce input groups=50
                Reduce shuffle bytes=2132511
                Reduce input records=116115
                Reduce output records=116115
                Spilled Records=232230
                Shuffled Maps =50
                Failed Shuffles=0
                Merged Map outputs=50
                GC time elapsed (ms)=47
                Total committed heap usage (bytes)=50567053312
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
```

Figure 1: Success message obtained on running `Problem2_re.java`



```
● (base) SGBHAT@ShreyankMac assign-2 % hdfs dfs -ls Output
  2025-02-26 21:13:08,779 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
  Found 4 items
  -rw-r--r--   1 SGBHAT supergroup          0 2025-02-26 21:12 Output/_SUCCESS
  -rw-r--r--   1 SGBHAT supergroup     873120 2025-02-26 21:12 Output/part-r-00000
  -rw-r--r--   1 SGBHAT supergroup     747273 2025-02-26 21:12 Output/part-r-00001
  -rw-r--r--   1 SGBHAT supergroup     574391 2025-02-26 21:12 Output/part-r-00002
```

Figure 2: Files produced by the outputs of each of the three reducers



```
25      {95603,v}
24      {95603,watch}
23      {95603,com}
22      {95603,youtube}
21      {95603,www}
20      {95603,http}
19      {95603,4}
18      {95603,5}
17      {95603,Eastern}
16      {95603,64}
15      {95603,16912}
14      {95603,2010}
13      {95603,14}
12      {95603,1}
11      {95603,67}
10      {95603,2}
9       {95603,21}
8       {95603,231}
7       {95603,88}
6       {95603,233}
5       {95603,Lancaster}
4       {95603,1797}
3       {95603,Kentucky}
2       {95603,County}
1       {95603,Garrard}
0       {95603,}
```

Figure 3: Key-Value pair output of reducer 1

Problem 3

Q. In this problem, you will write another MapReduce program that processes the output file generated by your previous program.

Write a mapper that reads from the file containing the index, document ID, and word, and emits key-value pairs of the form::

(index, (timestamp, word))

Here, index serves as the key, while the value is a pair containing the document ID, timestamp, and word. The timestamp should be derived from the document ID and converted into the standard Java timestamp format.

Write a reducer that groups the key-value pairs based on the index and emits the word associated with the maximum timestamp. Compare the output of the reducer with the actual Wikipedia article that has the highest document ID (i.e., the most recent version of the file).

If the generated file does not match the actual Wikipedia article, identify the reasons for the discrepancies. Implement another MapReduce function with the necessary modifications to address these issues. Finally, discuss how the modified MapReduce job produced the correct output.

Note: For Problem-3, submit both the MapReduce programs and their respective output files. Additionally, you may write a simple program in your preferred programming language to compare the output with the Wikipedia article that has the maximum timestamp value in the filename—submit that program as well.

A. Kindly refer the file `Prob3.java`, which was written to perform MapReduce for this part. Given below is an explanation of the code, and how it solves the given problem.

1. TimestampWordPair (Custom WritableComparable Class)

Description:

- A custom Hadoop WritableComparable class used to store a timestamp and a word for the mapper output, allowing for sorting based on the timestamp.

Functionality:

- Stores a Unix timestamp as a `long` and a word as a `String`.
- Implements the `WritableComparable` interface to support Hadoop serialization and comparison.
- Provides read and write functions to enable Hadoop to process the data efficiently.
- Overrides `compareTo()` to sort instances based on timestamp.

2. TimestampMapper (Mapper Class)

Description:

- Reads input lines, extracts document IDs, and converts them into timestamps while keeping associated words.

Functionality:

- Parses each line into an index and a `(doc_id, word)` pair.
- Converts the document ID into a Unix timestamp format (`docID * 1000`).
- Creates a `TimestampWordPair` object storing `(timestamp, word)`.
- Emits key-value pairs `(index, (timestamp, word))`.

3. MaxTimestampReducer (Reducer Class)

Description:

- Processes grouped key-value pairs received from the mapper and selects the latest word based on timestamp.

Functionality:

- Iterates over values for the same index and finds the word with the highest timestamp.
- Ensures the latest word is written to output `(index, word)`.

4. main() (Driver Program)

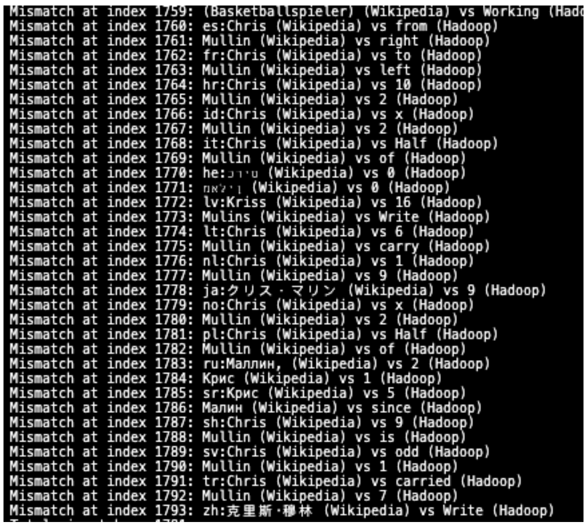
Purpose:

- Configures and runs the MapReduce job.

Functionality:

- Sets up a Hadoop configuration and creates a new MapReduce job named `"Find Latest Word by Index"`.
- Specifies the Mapper (`TimestampMapper`) and Reducer (`MaxTimestampReducer`) classes.
- Defines key-value output types:
  - **Mapper**
    - **Input:** `(LongWritable, Text)`
    - **Output:** `(IntWritable, TimestampWordPair)` (Key: Index, Value: (Timestamp, Word))
  - **Reducer**
    - **Input:** `(IntWritable, Iterable<TimestampWordPair>)`
    - **Output:** `(IntWritable, Text)` (Key: Index, Value: Latest Word)
- Specifies input and output paths using `FileInputFormat` and `FileOutputFormat`.
- Submits the job to the Hadoop cluster and waits for its completion.

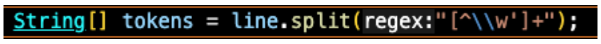
Furthermore, we wrote the program `compare_output.py` to do an index by index comparison between the output of `Prob3.java` and the most recent text file. On running it, it gaves an *output mismatch* indicating our result was incorrect.



```
Mismatch at index 1759: (Basketballspieler) (Wikipedia) vs Working (Hado
Mismatch at index 1760: es:Chris (Wikipedia) vs from (Hadoop)
Mismatch at index 1761: Mullin (Wikipedia) vs right (Hadoop)
Mismatch at index 1762: fr:Chris (Wikipedia) vs to (Hadoop)
Mismatch at index 1763: Mullin (Wikipedia) vs left (Hadoop)
Mismatch at index 1764: hr:Chris (Wikipedia) vs 10 (Hadoop)
Mismatch at index 1765: Mullin (Wikipedia) vs 2 (Hadoop)
Mismatch at index 1766: id:Chris (Wikipedia) vs x (Hadoop)
Mismatch at index 1767: Mullin (Wikipedia) vs 2 (Hadoop)
Mismatch at index 1768: it:Chris (Wikipedia) vs Half (Hadoop)
Mismatch at index 1769: Mullin (Wikipedia) vs of (Hadoop)
Mismatch at index 1770: he:כריס (Wikipedia) vs 0 (Hadoop)
Mismatch at index 1771: מאלין (Wikipedia) vs 0 (Hadoop)
Mismatch at index 1772: lv:Kriss (Wikipedia) vs 16 (Hadoop)
Mismatch at index 1773: Mulins (Wikipedia) vs Write (Hadoop)
Mismatch at index 1774: lt:Chris (Wikipedia) vs 6 (Hadoop)
Mismatch at index 1775: Mullin (Wikipedia) vs carry (Hadoop)
Mismatch at index 1776: nl:Chris (Wikipedia) vs 1 (Hadoop)
Mismatch at index 1777: Mullin (Wikipedia) vs 9 (Hadoop)
Mismatch at index 1778: ja:クリス・マリン (Wikipedia) vs 9 (Hadoop)
Mismatch at index 1779: no:Chris (Wikipedia) vs x (Hadoop)
Mismatch at index 1780: Mullin (Wikipedia) vs 2 (Hadoop)
Mismatch at index 1781: pl:Chris (Wikipedia) vs Half (Hadoop)
Mismatch at index 1782: Mullin (Wikipedia) vs of (Hadoop)
Mismatch at index 1783: ru:Маллин, (Wikipedia) vs 2 (Hadoop)
Mismatch at index 1784: Крис (Wikipedia) vs 1 (Hadoop)
Mismatch at index 1785: sr:Крис (Wikipedia) vs 5 (Hadoop)
Mismatch at index 1786: Малин (Wikipedia) vs since (Hadoop)
Mismatch at index 1787: sh:Chris (Wikipedia) vs 9 (Hadoop)
Mismatch at index 1788: Mullin (Wikipedia) vs is (Hadoop)
Mismatch at index 1789: sv:Chris (Wikipedia) vs odd (Hadoop)
Mismatch at index 1790: Mullin (Wikipedia) vs 1 (Hadoop)
Mismatch at index 1791: tr:Chris (Wikipedia) vs carried (Hadoop)
Mismatch at index 1792: Mullin (Wikipedia) vs 7 (Hadoop)
Mismatch at index 1793: zh:克里斯·穆林 (Wikipedia) vs Write (Hadoop)
```

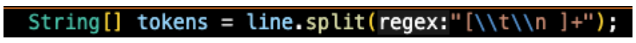
Figure 4: Output mismatch between `Prob3.java` and the most recent text file

On inspection, we narrowed down one error to be caused due to the split condition used in `Problem2.java` for tokenization. As mentioned earlier, we rectified it to `Problem2_re.java` and reproduced results from that code, to feed as input to `Prob3.java`.



```
String[] tokens = line.split(regex:"[^\\w']+");
```

Figure 5: Incorrect split condition in `Program2.java`



```
String[] tokens = line.split(regex:"[\\t\\n ]+");
```

Figure 6: Correct split condition in `Program2_re.java`

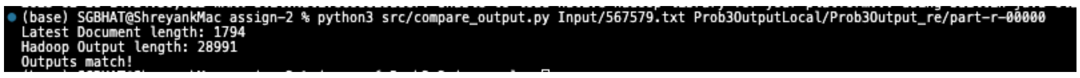
Also, the generated file did not match the latest Wikipedia article (567579.txt), since it may be possible that there exist files with names having a lesser numeric value, but are of greater length (greater number of words).

Because of this, we can have key-value pairs where the key is greater than the length of the latest text file. Thus, pairs with such indices will correspond to those generated from older versions of the text file.

One way we can handle this situation is to trim all the text files to the length of the most recent file, so that all the files either share the length of 567579.txt or are lesser than it. Alternatively, we can ignore all key-value pairs generated in the end which have a key greater than the length of 567579.txt.

The resultant output of this modified MapReduce - `Prob3_re` - yields an output which matches with the latest text file.

Finally, we ran `compare_output.py` once again, and obtained a perfect match with the most recent text file.



```
(base) SGBHAT@ShreyankMac assign-2 % python3 src/compare_output.py Input/567579.txt Prob3OutputLocal/Prob3Output_re/part-r-00000
Latest Document length: 1794
Hadoop Output length: 28991
Outputs match!
```

Figure 7: `Prob3.java` matching with the most recent text file after fixing the split condition

Problem 4

a) Develop a MapReduce program aimed at identifying the top 50 most frequently occurring words from the provided Wikipedia dump located on LMS (Wikipedia-EN-20120601 ARTICLES.tar.gz). Utilize the pairs approach for this task. Exclude non-stop words while computing the frequency of occurrence. Utilize the distributed caching option within Hadoop, demonstrated in WordCount2.java, to eliminate all stop-words in the mapper. (3 marks)

A. 1.) WordCount2 (MapReduce Program)

Description

The WordCount2 program implements a Hadoop MapReduce job to count word occurrences in a given dataset while filtering out stopwords. The program then selects and outputs the top 50 most frequently occurring words.

Functionality

This program performs the following steps:

- Reads the input text files.
- Tokenizes words while removing stopwords.
- Counts occurrences of each word using the Mapper and Reducer.
- Maintains a priority queue to track the top 50 words.
- Outputs the 50 most frequently occurring words.

2.) TokenizerMapper (Mapper Class)

Description

The TokenizerMapper class processes the input text, tokenizes words, removes stopwords, and emits (word, 1) key-value pairs.

Functionality

- Loads stopwords from a distributed cache.
- Converts input text to lowercase (unless case-sensitive mode is enabled).
- Tokenizes the text and removes non-alphanumeric characters.
- Skips stopwords while counting occurrences of each valid word.
- Emits (word, 1) for each valid token.

3.) IntSumReducer (Reducer Class)

Description

The IntSumReducer class aggregates word counts from the Mapper output and keeps track of the top 50 most frequently occurring words.

Functionality

- Accumulates word counts by summing the values for each word.
- Uses a min heap to maintain the top 50 words by frequency.
- Pops the minimum word when size of the min heap is greater than 50.
- Fig. 8 Outputs the 50 most frequently occurring words.

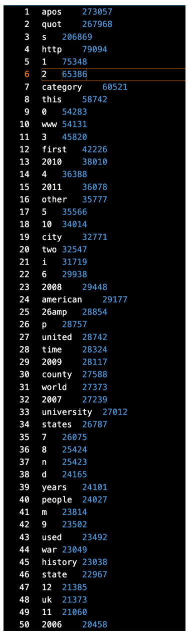
4.) Main Function

Description

The main function configures and runs the Hadoop MapReduce job.

Functionality

- Sets up the Hadoop job configuration.
- Allows optional arguments for skipping stopwords and enabling case sensitivity.
- Defines the Mapper and Reducer classes.
- Specifies input and output paths.
- Runs the job and waits for completion.



```
apos    273057
quot    267968
s   206869
http    79094
1   75348
2   65386
category    60521
this    58742
0   54283
www 54131
3   45820
first   42226
2010    38010
4   36388
2011    36078
other   35777
5   35566
10  34014
city    32771
two 32547
i   31719
6   29938
2008    29448
american    29177
26amp   28854
p   28757
united  28742
time    28324
2009    28117
county  27588
world   27373
2007    27239
university  27012
states  26787
7   26075
8   25424
n   25423
d   24165
years   24101
people  24027
m   23814
9   23502
used    23492
war 23049
history 23038
state   22967
12  21385
uk  21373
11  21060
2006    20458
```

Figure 8: wordcount output file

b) Write another MapReduce program, employing the pairs approach, to construct the co-occurring word matrix based solely on the frequent words identified in the previous question. Consider a word distance of d when determining co-occurring pairs. Report the runtime for d = 1, 2, 3, 4. (3 marks)

A. 1.) Question4b (MapReduce Program)

Description

The Question4b program implements a Hadoop MapReduce job to compute word co-occurrences in a text dataset. It extracts word pairs based on a specified distance and counts the number of times each pair appears within the top 50 most frequent words.

Functionality

This program performs the following steps:

- Reads the input text files.
- Identifies word pairs appearing within a given distance.
- Filters out words that are not in the top 50 most frequent words.
- Counts occurrences of each word pair using the Mapper and Reducer.
- Outputs the word pairs along with their frequencies.

2.) WordPair (Custom Writable Class)

Description

The WordPair class represents a pair of words and implements the WritableComparable interface for serialization and sorting in Hadoop.

Functionality

- Stores two words as Text objects.
- Implements the compareTo method to define sorting order.
- Implements readFields and write methods for serialization.
- Provides a toString method for output formatting.

2.) pairMapper (Mapper Class)

Description

The pairMapper class processes input text files, tokenizes words, filters out unwanted words, and emits word pairs that appear within a specified distance.

Functionality

- Loads the top 50 most frequent words from a distributed cache.
- Tokenizes the text into words.
- Identifies word pairs that appear within the specified distance.
- Emits (word pair, 1) for valid co-occurrences.

3.) CoReducer (Reducer Class)

Description

The CoReducer class aggregates word pair counts from the Mapper output.

Functionality

- Accumulates counts for each word pair.
- Emits (word pair, count) as the final output.
- Fig 13 shows the output of the pairs algorithm.

4.) Main Function

Description

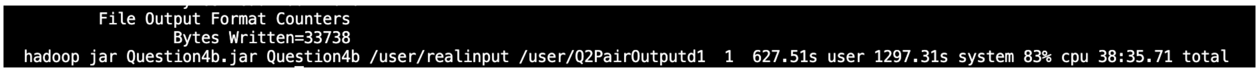
The main function configures and runs the Hadoop MapReduce job.

Functionality

- Configures the Hadoop job settings.
- Defines the Mapper and Reducer classes.
- Sets the co-occurrence distance from command-line arguments.
- Specifies input and output paths.
- Runs the job and waits for completion.

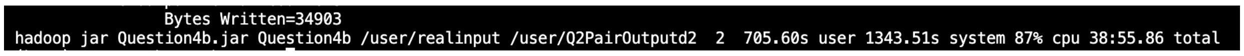
Run-time

- Total CPU Time = user time + system time.
- In Fig 9 cpu run-time is ((627.51+1297.31)/60=)32:04 mins, total run-time is 38:35.71 mins.
- In Fig 10 cpu run-time is ((705.60+1343.51)/60=)34:09 mins, total run-time is 38:55.86 mins.
- In Fig 11 cpu run-time is ((635.58+1301.72)/60=)32:17 mins, total run-time is 38:11.50 mins.
- In Fig 12 cpu run-time is ((636.04+1283.23)/60=)31:59 mins, total run-time is 37:31.01 mins.



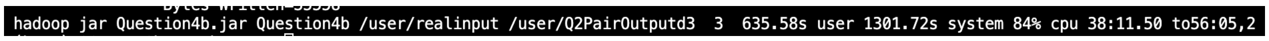
```
        File Output Format Counters
                Bytes Written=33738
 hadoop jar Question4b.jar Question4b /user/realinput /user/Q2PairOutputd1  1  627.51s user 1297.31s system 83% cpu 38:35.71 total
```

Figure 9: run time analysis with d=1



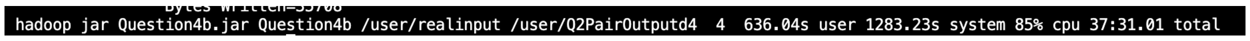
```
                Bytes Written=34903
 hadoop jar Question4b.jar Question4b /user/realinput /user/Q2PairOutputd2  2  705.60s user 1343.51s system 87% cpu 38:55.86 total
```

Figure 10: run time analysis with d=2



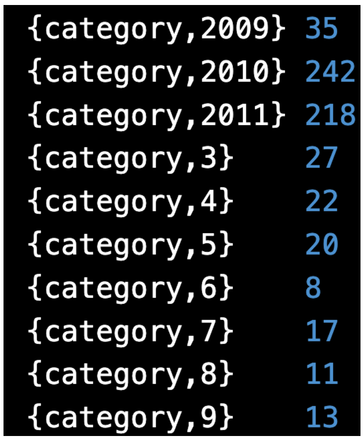
```
 hadoop jar Question4b.jar Question4b /user/realinput /user/Q2PairOutputd3  3  635.58s user 1301.72s system 84% cpu 38:11.50 to56:05,2
```

Figure 11: run time analysis with d=3



```
 hadoop jar Question4b.jar Question4b /user/realinput /user/Q2PairOutputd4  4  636.04s user 1283.23s system 85% cpu 37:31.01 total
```

Figure 12: run time analysis with d=4



```
{category,2009} 35
{category,2010} 242
{category,2011} 218
{category,3}    27
{category,4}    22
{category,5}    20
{category,6}    8
{category,7}    17
{category,8}    11
{category,9}    13
```

Figure 13: output for pairs algorithm

c) Develop a stripe algorithm to create the co-occurring word matrix. Again, report the runtime for d = 1, 2, 3, 4. (3 marks)

A. 1.) Question4c (MapReduce Program)

Description

The Question4c program implements a Hadoop MapReduce job to compute word co-occurrences using the stripe approach. It constructs a co-occurrence matrix where each word is associated with a map of its neighboring words and their counts within a specified distance.

Functionality

This program performs the following steps:

- Reads the input text files.
- Identifies word co-occurrences within a given distance.
- Filters out words that are not in the top 50 most frequent words.
- Constructs a stripe (map) for each word, storing its co-occurrence counts.
- Aggregates co-occurrence counts in the reducer.
- Outputs each word with its associated co-occurring words and their frequencies.

2.) StripeMapper (Mapper Class)

Description

The StripeMapper class processes input text, identifies word co-occurrences within a specified distance, and emits each word with a map containing its co-occurring words and their counts.

Functionality

- Loads the top 50 most frequent words from a distributed cache.
- Tokenizes the text into words.
- Uses a sliding window approach to identify co-occurring words.
- Constructs a map (stripe) for each word with its neighboring words.
- Emits (word, stripe) pairs.

3.) StripeReducer (Reducer Class)

Description

The StripeReducer class aggregates co-occurrence maps from the Mapper output and merges the counts for each word pair.

Functionality

- Iterates through the stripes for a given word.
- Merges co-occurrence counts across different stripes.
- Outputs each word along with its co-occurring words and their final counts.

4.) Main Function

Description

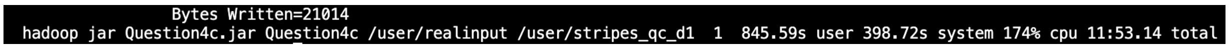
The main function configures and runs the Hadoop MapReduce job.

Functionality

- Configures the Hadoop job settings.
- Defines the Mapper and Reducer classes.
- Sets the co-occurrence distance from command-line arguments.
- Specifies input and output paths.
- Runs the job and waits for completion.

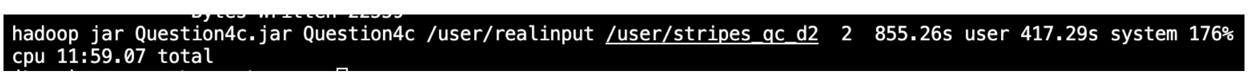
Run-time

- Total CPU Time = user time + system time.
- In Fig 14 cpu run-time is ((845.59+398.72)/60=)20:44 mins, total run-time is 11:53.14 mins.
- In Fig 15 cpu run-time is ((855.26+417.29)/60=)21:12 mins, total run-time is 11:59.07 mins.
- In Fig 16 cpu run-time is ((750.83+384.95)/60=)18:54 mins, total run-time is 11:36.53 mins.
- In Fig 17 cpu run-time is ((775.20+393.85)/60=)19:28 mins, total run-time is 11:33.42 mins.



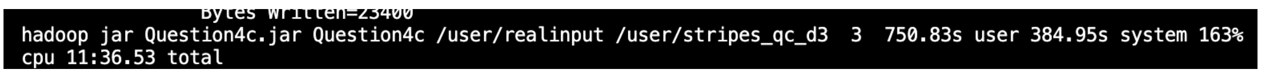
```
                Bytes Written=21014
 hadoop jar Question4c.jar Question4c /user/realinput /user/stripes_qc_d1  1  845.59s user 398.72s system 174% cpu 11:53.14 total
```

Figure 14: run time analysis for stripes algorithm for d=1



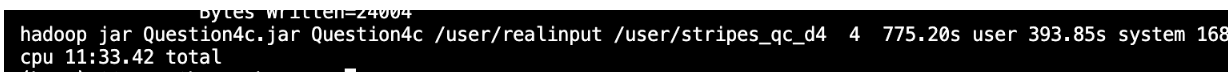
```
hadoop jar Question4c.jar Question4c /user/realinput /user/stripes_qc_d2  2  855.26s user 417.29s system 176%
cpu 11:59.07 total
```

Figure 15: run time analysis for stripes algorithm for d=2



```
                Bytes Written=23400
hadoop jar Question4c.jar Question4c /user/realinput /user/stripes_qc_d3  3  750.83s user 384.95s system 163%
cpu 11:36.53 total
```

Figure 16: run time analysis for stripes algorithm for d=3



```
                Bytes Written=24004
hadoop jar Question4c.jar Question4c /user/realinput /user/stripes_qc_d4  4  775.20s user 393.85s system 168
cpu 11:33.42 total
```

Figure 17: run time analysis for stripes algorithm for d=4

e) Additionally, implement local aggregations separately at the Map-class level and the Map-function level, and compare their performance. Conduct this comparison for parts (b) and (c). Once again, the runtime must be reported for d = 1, 2, 3, 4. (5 marks)

A. 1.) Question4e (MapReduce Program)

Description

The Question4e program implements a Hadoop MapReduce job to compute word co-occurrences using both the pairs and stripes approaches. It constructs a co-occurrence matrix where each word is associated with its neighboring words and their frequencies, using either of the two approaches.

Functionality

This program performs the following steps:

- Reads the input text files.
- Identifies word co-occurrences within a given distance.
- Filters out words that are not in the top 50 most frequent words.
- Constructs word co-occurrence representations using the pairs or stripes approach.
- Aggregates co-occurrence counts using reducers.
- Outputs word pairs with their co-occurrence counts.

2.) WordPair (Custom Writable Class)

Description

The WordPair class represents a pair of words and implements the WritableComparable interface for serialization and sorting in Hadoop.

Functionality

- Stores two words as Text objects.
- Implements the compareTo method to define sorting order.
- Implements readFields and write methods for serialization.
- Provides a toString method for output formatting.
- Ensures proper hashing and equality checking.

3.) PairsClassLevelAggregationMapper (Mapper Class)

Description

The PairsClassLevelAggregationMapper class processes input text, identifies word co-occurrences within a specified distance, and maintains state within the Mapper instance for class-level aggregation.

Functionality

- Loads the top 50 most frequent words from a distributed cache.
- Tokenizes the text into words.
- Uses a class-level map to accumulate word pair counts before emitting them.
- Emits aggregated word pair counts at the end of the Mapper's lifecycle.

4.) PairsFunctionLevelAggregationMapper (Mapper Class)

Description

The PairsFunctionLevelAggregationMapper class processes input text, identifies word co-occurrences within a specified distance, and aggregates counts within each map function call.

Functionality

- Loads the top 50 most frequent words from a distributed cache.
- Tokenizes the text into words.
- Uses a function-level map to accumulate word pair counts.
- Emits aggregated word pair counts at the end of processing each input split.

5.) StripesClassLevelAggregationMapper (Mapper Class)

Description

The StripesClassLevelAggregationMapper class processes input text, identifies word co-occurrences, and maintains a co-occurrence matrix using a stripes approach with class-level aggregation.

Functionality

- Loads the top 50 most frequent words from a distributed cache.
- Tokenizes the text into words.
- Maintains a map of co-occurrence stripes for words.
- Emits aggregated word co-occurrence stripes at the end of the Mapper's lifecycle.

6.) StripesFunctionLevelAggregationMapper (Mapper Class)

Description

The StripesFunctionLevelAggregationMapper class processes input text, identifies word co-occurrences, and aggregates co-occurrence counts within each map function.

Functionality

- Loads the top 50 most frequent words from a distributed cache.
- Tokenizes the text into words.
- Maintains a local map of co-occurrence stripes.
- Emits aggregated word co-occurrence stripes at the end of processing each input split.

7.)PairsReducer (Reducer Class)

Description

The PairsReducer class aggregates word pair counts from the Mapper output.

Functionality

- Iterates through word pair counts.
- Accumulates counts for each word pair.
- Emits word pairs with their co-occurrence counts.

8.) StripesReducer (Reducer Class)

Description

The StripesReducer class aggregates co-occurrence maps from the Mapper output and merges the counts for each word.

Functionality

- Iterates through word co-occurrence maps.
- Merges co-occurrence counts across different stripes.
- Outputs each word along with its co-occurring words and their final counts.

9.) Main Function

Description

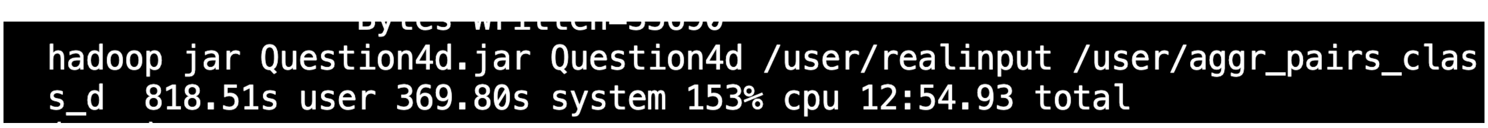
The main function configures and runs the Hadoop MapReduce job.

Functionality

- Configures the Hadoop job settings.
- Selects the appropriate Mapper and Reducer based on user input (pairs or stripes).
- Defines whether aggregation should occur at the class or function level.
- Specifies input and output paths.
- Runs the job and waits for completion.

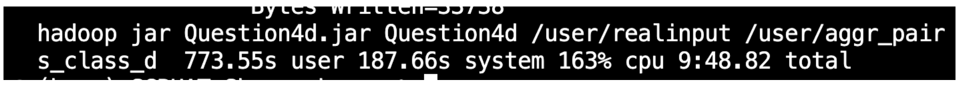
Run-time

- Total CPU Time = user time + system time.
- In Fig 18 cpu run-time is ((818.51+369.80)/60=)19:48 mins, total run-time is 12:54.93 mins.
- In Fig 19 cpu run-time is ((773.55+187.66)/60=) 16:01 mins, total run-time is 9:48.82 mins.
- In Fig 20 cpu run-time is ((818.51+369.80)/60=)19:48 mins, total run-time is 12:54.93 mins.
- In Fig 21 cpu run-time is ((773.55+187.66)/60=) 16:01 mins, total run-time is 9:48.82 mins.
- In Fig 22 cpu run-time is ((733.69+198.04)/60=)15:32 mins, total run-time is 9:57.40 mins.
- In Fig 23 cpu run-time is ((662.14+188.19)/60=)14:10 mins, total run-time is 9:49.54 mins.
- In Fig 24 cpu run-time is ((685.49+187.53)/60=)14:33 mins, total run-time is 9:34.51 mins.
- In Fig 25 cpu run-time is ((679.75+183.18)/60=)14:28 mins, total run-time is 9:21.42 mins.
- In Fig 26 cpu run-time is ((774.16+206.15)/60=)16:20 mins, total run-time is 9:55:09 mins.
- In Fig 28 cpu run-time is ((778.78+193.89)/60=)16:11 mins, total run-time is 9:32.65 mins.
- In Fig 28 cpu run-time is ((787.39+197.38)/60=)16:25 mins, total run-time is 9:48.27 mins.
- In Fig 29 cpu run-time is ((739.44+194.61)/60=)15:34 mins, total run-time is 9:33.79 mins.
- In Fig 30 cpu run-time is ((716.26+213.45)/60=)15:30 mins, total run-time is 9:19.87 mins.
- In Fig 31 cpu run-time is ((699.10+216.67)/60=)15:16 mins, total run-time is 9:35.29 mins.
- In Fig 32 cpu run-time is ((676.87+205.59)/60=)14:42 mins, total run-time is 9:26.03 mins.
- In Fig 33 cpu run-time is ((719.67+206.46)/60=)15:26 mins, total run-time is 9:31.32 mins.



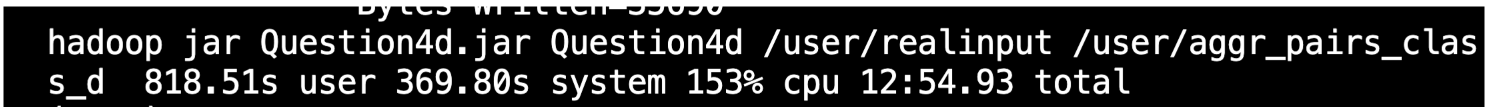
```
hadoop jar Question4d.jar Question4d /user/realinput /user/aggr_pairs_clas
s_d  818.51s user 369.80s system 153% cpu 12:54.93 total
```

Figure 18: run time analysis pairs algorithm for Map-class level aggregation, d=1



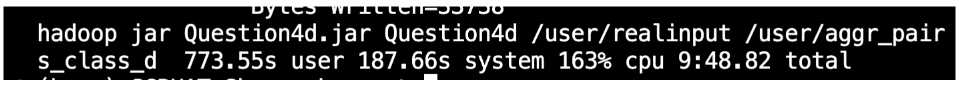
```
 hadoop jar Question4d.jar Question4d /user/realinput /user/aggr_pair
 s_class_d  773.55s user 187.66s system 163% cpu 9:48.82 total
```

Figure 19: run time analysis pairs algorithm for Map-class level aggregation, d=2



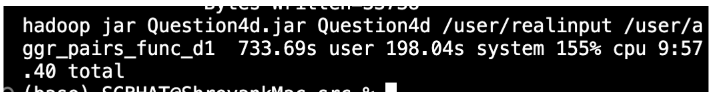
```
 hadoop jar Question4d.jar Question4d /user/realinput /user/aggr_pairs_clas
 s_d  818.51s user 369.80s system 153% cpu 12:54.93 total
```

Figure 20: run time analysis pairs algorithm for Map-class level aggregation, d=3



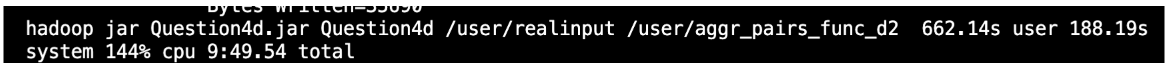
```
 hadoop jar Question4d.jar Question4d /user/realinput /user/aggr_pair
 s_class_d  773.55s user 187.66s system 163% cpu 9:48.82 total
```

Figure 21: run time analysis pairs algorithm for Map-class level aggregation, d=4



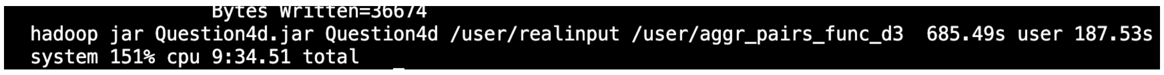
```
 hadoop jar Question4d.jar Question4d /user/realinput /user/a
 ggr_pairs_func_d1  733.69s user 198.04s system 155% cpu 9:57
 .40 total
```

Figure 22: run time analysis pairs algorithm for Map-function level aggregation, d=1



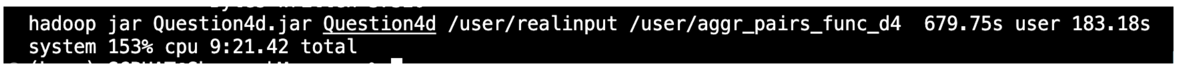
```
 hadoop jar Question4d.jar Question4d /user/realinput /user/aggr_pairs_func_d2  662.14s user 188.19s
 system 144% cpu 9:49.54 total
```

Figure 23: run time analysis pairs algorithm for Map-function level aggregation, d=2



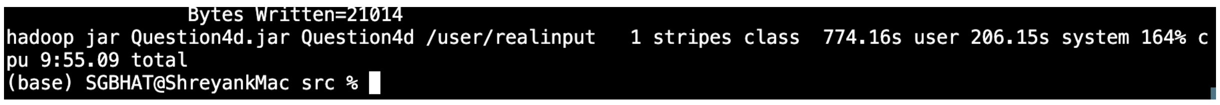
```
              Bytes Written=36674
 hadoop jar Question4d.jar Question4d /user/realinput /user/aggr_pairs_func_d3  685.49s user 187.53s
 system 151% cpu 9:34.51 total
```

Figure 24: run time analysis pairs algorithm for Map-function level aggregation, d=3



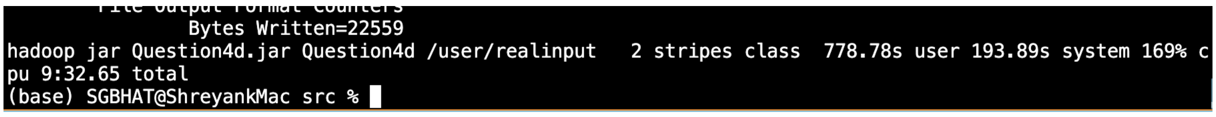
```
 hadoop jar Question4d.jar Question4d /user/realinput /user/aggr_pairs_func_d4  679.75s user 183.18s
 system 153% cpu 9:21.42 total
```

Figure 25: run time analysis pairs algorithm for Map-function level aggregation, d=4



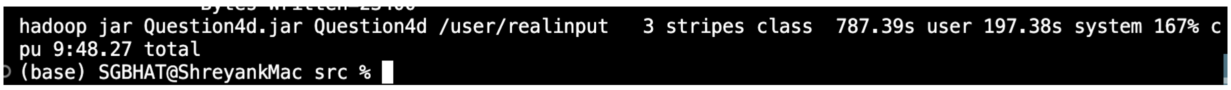
```
               Bytes Written=21014
hadoop jar Question4d.jar Question4d /user/realinput   1 stripes class  774.16s user 206.15s system 164% c
pu 9:55.09 total
(base) SGBHAT@ShreyankMac src %
```

Figure 26: run time analysis stripes algorithm for Map-class level aggregation, d=1



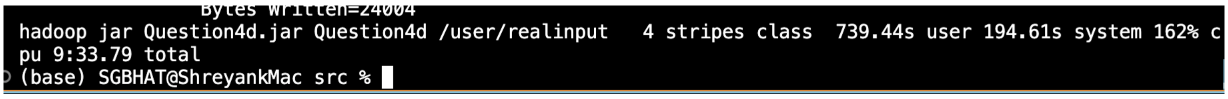
```
               Bytes Written=22559
hadoop jar Question4d.jar Question4d /user/realinput   2 stripes class  778.78s user 193.89s system 169% c
pu 9:32.65 total
(base) SGBHAT@ShreyankMac src %
```

Figure 27: run time analysis stripes algorithm for Map-class level aggregation, d=2



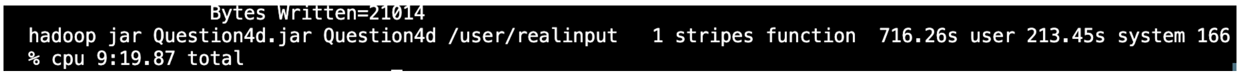
```
 hadoop jar Question4d.jar Question4d /user/realinput   3 stripes class  787.39s user 197.38s system 167% c
 pu 9:48.27 total
 (base) SGBHAT@ShreyankMac src %
```

Figure 28: run time analysis stripes algorithm for Map-class level aggregation, d=3



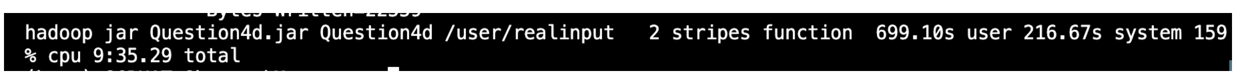
```
                    Bytes Written=24004
hadoop jar Question4d.jar Question4d /user/realinput   4 stripes class  739.44s user 194.61s system 162% c
pu 9:33.79 total
(base) SGBHAT@ShreyankMac src %
```

Figure 29: run time analysis stripes algorithm for Map-class level aggregation, d=4



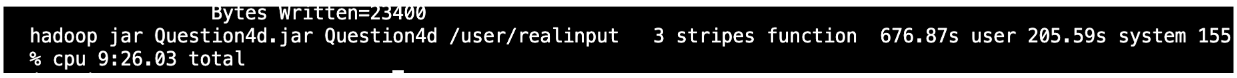
```
                    Bytes Written=21014
hadoop jar Question4d.jar Question4d /user/realinput   1 stripes function  716.26s user 213.45s system 166
% cpu 9:19.87 total
```

Figure 30: run time analysis stripes algorithm for Map-function level aggregation, d=1



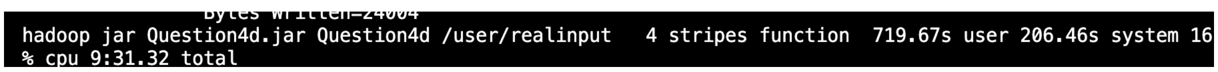
```
hadoop jar Question4d.jar Question4d /user/realinput   2 stripes function  699.10s user 216.67s system 159
% cpu 9:35.29 total
```

Figure 31: run time analysis stripes algorithm for Map-function level aggregation, d=2



```
                    Bytes Written=23400
hadoop jar Question4d.jar Question4d /user/realinput   3 stripes function  676.87s user 205.59s system 155
% cpu 9:26.03 total
```

Figure 32: run time analysis stripes algorithm for Map-function level aggregation, d=3



```
                    Bytes Written=24004
hadoop jar Question4d.jar Question4d /user/realinput   4 stripes function  719.67s user 206.46s system 16
% cpu 9:31.32 total
```

Figure 33: run time analysis stripes algorithm for Map-function level aggregation, d=4

Problem 5

a) Implement a pair of a Map and a Reduce function which, for each distinct term that occurs in any of the text documents in Wikipedia-EN-20120601 ARTICLES.tar.gz, counts the number of distinct documents in which the term appears. We will call this value the Document Frequency (DF) of that term in the entire set of Wikipedia articles. Store the resulting DF values of all terms in a single TSV file with the following schema:
TERM <tab> DF
While generating the output in the above format, consider filtering out all terms that belong to the stopwords.txt file shared on LMS. (You may perform this filter operation in your map method.) Identify the top 100 terms with a high document frequency. Use those terms alone for the next sub problem. (6 marks)

A. 1.) DFMapper (Mapper Class)

Description

The DFMapper class processes Wikipedia text documents and extracts terms that appear in each document. It filters out stopwords, applies Porter Stemming, and emits (term, document ID) pairs to be used in the Reduce phase for computing Document Frequency (DF).

Functionality

- Loads a stopword list from HDFS.
- Extracts the document ID from the filename.
- Tokenizes the text and filters stopwords.
- Applies stemming using the Porter Stemmer.
- Emits unique (word, document ID) pairs.

Setup Function

Description

The setup function is executed once per Mapper task before processing the input data. It loads stopwords from HDFS (or distributed cache) and extracts the document ID from the filename.

Functionality

- Reads the stopwords file path from the Hadoop configuration.
- Loads stopwords into a HashSet for fast lookup.
- Extracts the document ID from the filename by removing the file extension.

Map Function

Description

The map function processes the content of a text document. It tokenizes the text, removes stopwords, applies stemming, and emits unique (term, document ID) pairs.

Functionality

- Prepares the text for processing by converting it to lowercase and removing special characters.
- Tokenizes the text into words.
- Filters out stopwords and applies stemming.
- Emits (term, document ID) pairs.

2.) DFReducer (Reducer Class)

Description

The DFReducer class processes the intermediate key-value pairs generated by the DFMapper. It aggregates document occurrences for each term and computes the Document Frequency (DF), which is the number of unique documents in which a term appears.

Functionality

This Reducer performs the following steps:

- Collects document IDs associated with each term.
- Stores them in a HashSet to ensure uniqueness.
- Computes the document frequency (DF) by counting unique document IDs.
- Emits (term, DF) pairs.

Reduce Function

Description

The reduce function aggregates document IDs for each term and calculates the number of distinct documents in which the term appears.

Functionality

- Initializes a HashSet to store unique document IDs.
- Iterates over the list of document IDs for the given term.
- Adds each document ID to the HashSet to remove duplicates.
- Sets the count of unique document IDs as the document frequency.
- Emits (term, DF) as the final output.

3.) DFDriver (Driver Class)

Description

The DFDriver class is responsible for configuring and running the Hadoop MapReduce job for Document Frequency (DF) calculation. It sets up the job with the Mapper and Reducer classes, specifies input and output paths, and ensures that the stopwords file is passed to the Mapper.

Functionality

This Driver performs the following steps:

- Reads input, output, and stopwords file paths from command-line arguments.
- Configures the Hadoop job with the required Mapper and Reducer.
- Sets the stopwords file path as a configuration parameter.
- Specifies key-value data types for intermediate and final output.
- Adds the OpenNLP library to the classpath.
- Runs the MapReduce job and waits for completion.

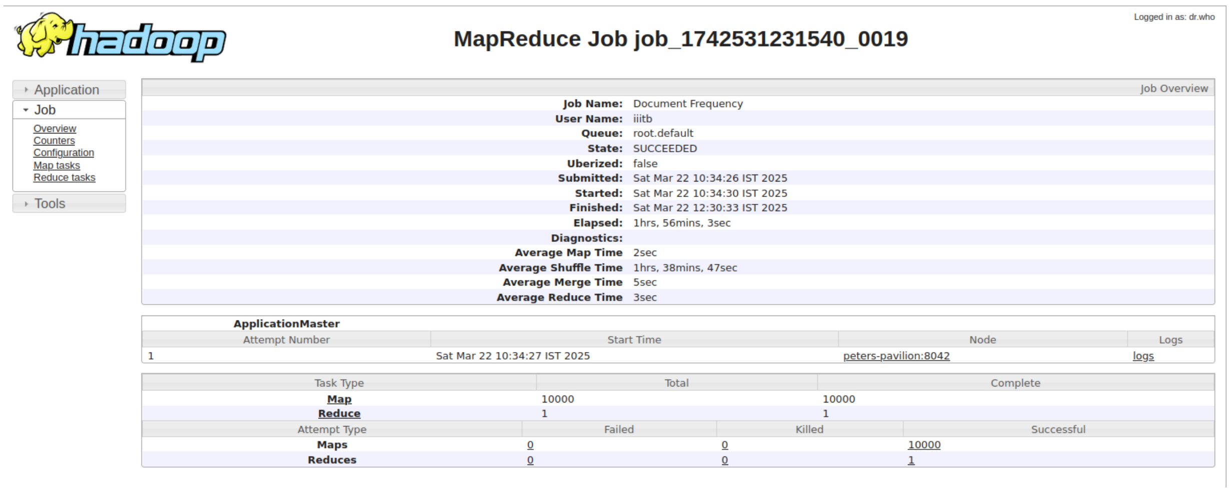
Main Function

Description

The main function is the entry point for running the Document Frequency job. It validates input arguments, configures the Hadoop job, and initiates execution.

Functionality

- Checks if the correct number of command-line arguments are provided.
- Extracts the input file path, output file path, and stopwords file path.
- Creates a Hadoop Configuration object and sets the stopwords file path.
- Initializes a new Hadoop Job instance with the job name.
- Sets the Mapper class (DFMapper) and Reducer class (DFReducer).
- Specifies key-value data types for output and intermediate results.
- Configures input and output paths for the job.
- Adds the OpenNLP library to the classpath to enable stemming.
- Runs the job and waits for completion.



MapReduce Job job_1742531231540_0019

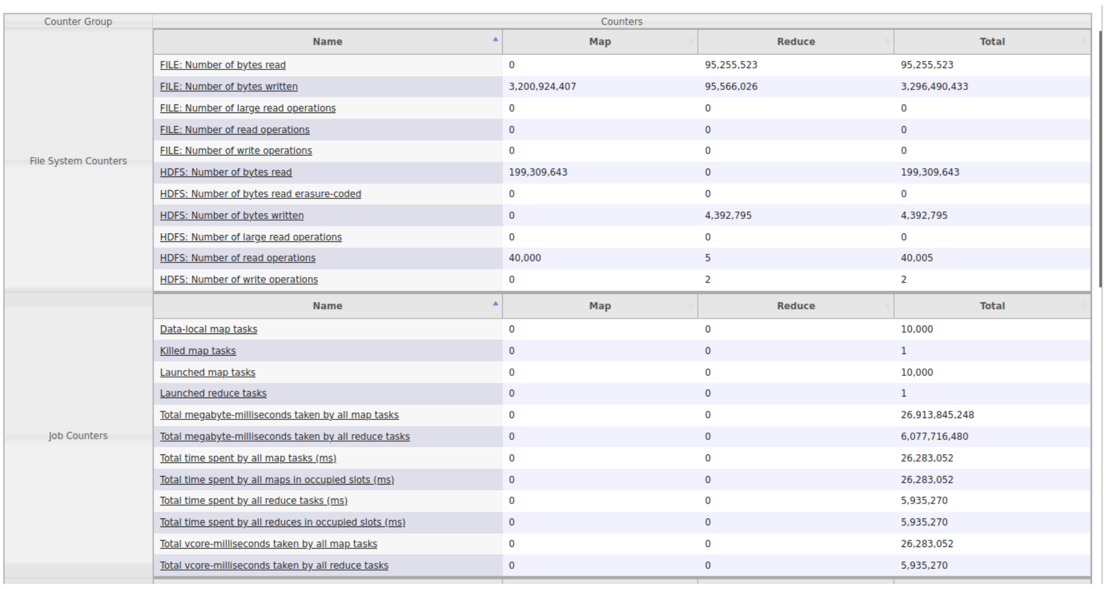
	Job Overview
Job Name:	Document Frequency
User Name:	iiitb
Queue:	root.default
State:	SUCCEEDED
Uberized:	false
Submitted:	Sat Mar 22 10:34:26 IST 2025
Started:	Sat Mar 22 10:34:30 IST 2025
Finished:	Sat Mar 22 12:30:33 IST 2025
Elapsed:	1hrs, 56mins, 3sec
Diagnostics:	
Average Map Time	2sec
Average Shuffle Time	1hrs, 38mins, 47sec
Average Merge Time	5sec
Average Reduce Time	3sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Sat Mar 22 10:34:27 IST 2025	peters-pavilion:8042	logs

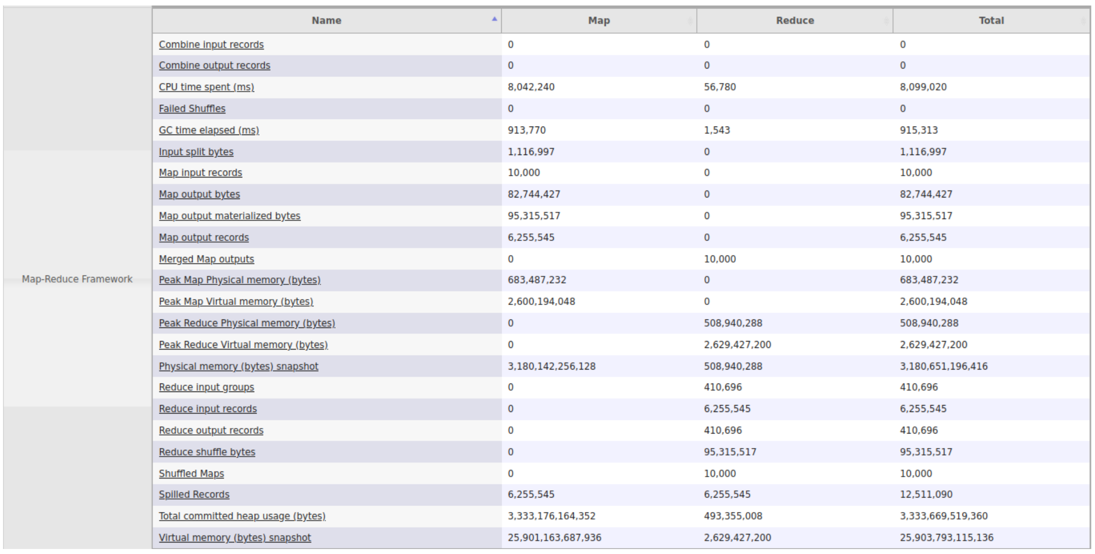
Task Type	Total	Complete	
Map	10000	10000	
Reduce	1	1	
Attempt Type	Failed	Killed	Successful
Maps	0	0	10000
Reduces	0	0	1

Figure 34: MR overview for 5a



Counter Group	Name	Map	Reduce	Total
File System Counters	FILE: Number of bytes read	0	95,255,523	95,255,523
	FILE: Number of bytes written	3,200,924,407	95,566,026	3,296,490,433
	FILE: Number of large read operations	0	0	0
	FILE: Number of read operations	0	0	0
	FILE: Number of write operations	0	0	0
	HDFS: Number of bytes read	199,309,643	0	199,309,643
	HDFS: Number of bytes read erasure-coded	0	0	0
	HDFS: Number of bytes written	0	4,392,795	4,392,795
	HDFS: Number of large read operations	0	0	0
	HDFS: Number of read operations	40,000	5	40,005
	HDFS: Number of write operations	0	2	2
Job Counters	Data-local map tasks	0	0	10,000
	Killed map tasks	0	0	1
	Launched map tasks	0	0	10,000
	Launched reduce tasks	0	0	1
	Total megabyte-milliseconds taken by all map tasks	0	0	26,913,845,248
	Total megabyte-milliseconds taken by all reduce tasks	0	0	6,077,716,480
	Total time spent by all map tasks (ms)	0	0	26,283,052
	Total time spent by all maps in occupied slots (ms)	0	0	26,283,052
	Total time spent by all reduce tasks (ms)	0	0	5,935,270
	Total time spent by all reduces in occupied slots (ms)	0	0	5,935,270
	Total vcore-milliseconds taken by all map tasks	0	0	26,283,052
	Total vcore-milliseconds taken by all reduce tasks	0	0	5,935,270

Figure 35: Some statistics for 5a



	Name	Map	Reduce	Total
Map-Reduce Framework	Combine input records	0	0	0
	Combine output records	0	0	0
	CPU time spent (ms)	8,042,240	56,780	8,099,020
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	913,770	1,543	915,313
	Input split bytes	1,116,997	0	1,116,997
	Map input records	10,000	0	10,000
	Map output bytes	82,744,427	0	82,744,427
	Map output materialized bytes	95,315,517	0	95,315,517
	Map output records	6,255,545	0	6,255,545
	Merged Map outputs	0	10,000	10,000
	Peak Map Physical memory (bytes)	683,487,232	0	683,487,232
	Peak Map Virtual memory (bytes)	2,600,194,048	0	2,600,194,048
	Peak Reduce Physical memory (bytes)	0	508,940,288	508,940,288
	Peak Reduce Virtual memory (bytes)	0	2,629,427,200	2,629,427,200
	Physical memory (bytes) snapshot	3,180,142,256,128	508,940,288	3,180,651,196,416
	Reduce input groups	0	410,696	410,696
	Reduce input records	0	6,255,545	6,255,545
	Reduce output records	0	410,696	410,696
	Reduce shuffle bytes	0	95,315,517	95,315,517
	Shuffled Maps	0	10,000	10,000
	Spilled Records	6,255,545	6,255,545	12,511,090
	Total committed heap usage (bytes)	3,333,176,164,352	493,355,008	3,333,669,519,360
	Virtual memory (bytes) snapshot	25,901,163,687,936	2,629,427,200	25,903,793,115,136

Figure 36: Some more statistics for 5a

Top 100 words from the output of 5a with the highest frequency and length greater than two are used in 5b.

b) Implement another pair of a Map and a Reduce function which, for each document in WikipediaEN- 20120601 ARTICLES.tar.gz, first counts the number of occurrences of each distinct term within the given document. We will call this value the Term Frequency (TF) of that term in the given document. Implement a stripes algorithm here as we are dealing with just 100 terms. In a second step, your combined MapReduce function should multiply the TF value of each such term with the inverse of the logarithm of the normalized DF value calculated by the previous MapReduce function, i.e., SCORE = TF×log(10000/DF+1) for each combination of a term and a document. You may cache the former TSV file with the DF values by adding it via Job.addCacheFile(¡path-to-DF-file¿) in the driver function of your MapReduce program. The Map class should then load this file upon initialization into an appropriate main-memory data structure. The result of this MapReduce function should be a single TSV file that has the same schema as the file we used in the previous exercise sheets: (6 Points)
ID<tab>TERM<tab>SCORE

A. 1.) TFIDFMapper (Mapper Class)

Description

The TFIDFMapper class processes Wikipedia text documents and calculates Term Frequency (TF) for a predefined set of the top 100 terms. It filters out stopwords, applies Porter Stemming, and emits key-value pairs of the form (term, documentID=TF).

Functionality

This Mapper performs the following steps:

- Loads stopwords and the top 100 most frequent terms from the distributed cache.
- Extracts the document ID from the filename.
- Tokenizes the text and removes stopwords.
- Applies stemming using the Porter Stemmer.
- Counts occurrences of each of the top 100 terms in the document.
- Emits key-value pairs (term, documentID=TF).

Setup Function

Description

The setup function is executed once per Mapper task before processing the input data. It loads stopwords and the top 100 most frequent terms from the distributed cache and extracts the document ID from the filename.

Functionality

- Retrieves the stopwords and top 100 words file paths from the distributed cache.
- Reads and stores stopwords in a HashSet for quick lookup.
- Reads and stores the top 100 words in a HashSet to ensure only relevant terms are processed.
- Extracts the document ID from the filename by removing the file extension.

Map Function

Description

The map function processes the content of a text document. It tokenizes the text, removes stopwords, applies stemming, counts occurrences of the top 100 terms, and emits (term, documentID=TF) pairs.

Functionality

- Converts text to lowercase and removes special characters.
- Tokenizes the text into words.
- Filters out stopwords and applies stemming.
- Counts occurrences of each top 100 term in the document.
- Ensures all 100 terms are emitted, even if absent (TF = 0).
- Emits (term, documentID=TF) pairs.

2.) TFIDFReducer (Reducer Class)

Description

The TFIDFReducer class processes the intermediate key-value pairs generated by the TFIDFMapper. It calculates the Term Frequency-Inverse Document Frequency (TF-IDF) score for each term in a document. The TF-IDF score measures the importance of a term within a document relative to the entire corpus.

Functionality

This Reducer performs the following steps:

- Retrieves the total number of documents from the configuration.
- Aggregates term frequencies for each document.
- Computes document frequencies (DF) to determine how many documents contain each term.
- Calculates the TF-IDF score using the formula:

$$TFIDF = TF \times \log_{10}\left(\frac{N}{DF+1}\right)$$

  where N is the total number of documents.
- Emits key-value pairs in the format (DOC_ID, TERM, SCORE).

Setup Function

Description

The setup function runs once per Reducer task before processing input data. It retrieves the total number of documents from the Hadoop configuration.

Functionality

- Reads the total document count from the Hadoop configuration.
- Uses a default value of 10,000 if the total count is not provided.

Reduce Function

Description

The reduce function aggregates term frequencies and document frequencies, then calculates the TF-IDF score for each term in a document.

Functionality

- Iterates over input values and extracts term frequency (TF) pairs.
- Accumulates document frequencies (DF) by counting how many documents contain each term.
- Computes TF-IDF using the log-scaled formula.
- Emits (document ID, term, TF-IDF score).

3.) TFIDFDriver (Driver Class)

Description

The TFIDFDriver class is responsible for configuring and running the Hadoop MapReduce job for TF-IDF computation. It sets up the job with the appropriate Mapper and Reducer, specifies input and output paths, and ensures that stopwords and the top 100 terms are available through the distributed cache.

Functionality

This Driver performs the following steps:

- Reads input, output, top 100 terms, and stopwords file paths from command-line arguments.
- Configures the Hadoop job with the required Mapper and Reducer classes.
- Ensures that the output directory is deleted if it already exists.
- Adds stopwords and top 100 words to the distributed cache for efficient access.
- Specifies key-value data types for intermediate and final outputs.
- Runs the MapReduce job and waits for completion.

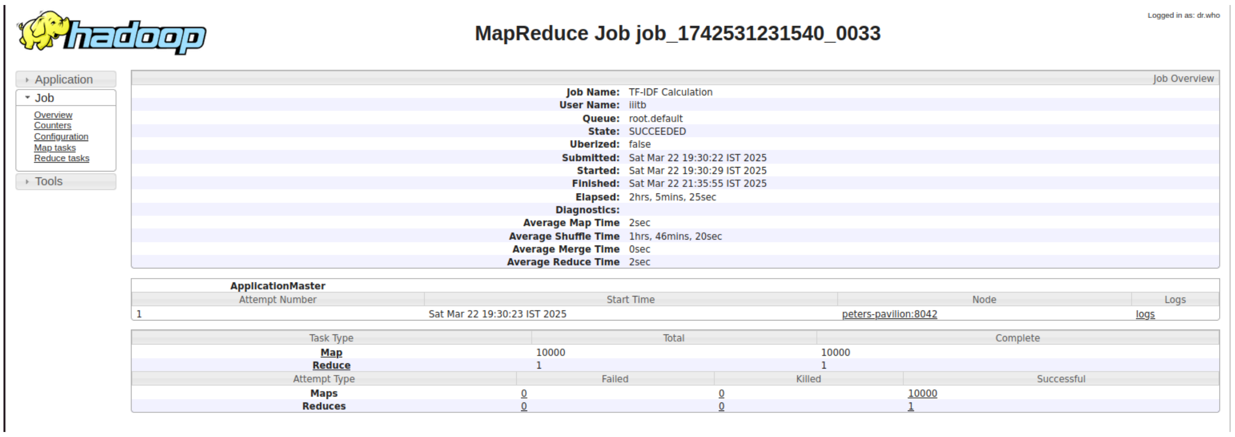
Main Function

Description

The main function is the entry point for running the TF-IDF job. It validates input arguments, configures the Hadoop job, and initiates execution.

Functionality

- Validates that four arguments (input path, output path, top 100 terms, and stopwords file) are provided.
- Creates a Hadoop Configuration object and initializes a new job instance.
- Sets up the Mapper (TFIDFMapper) and Reducer (TFIDFReducer) classes.
- Configures input and output paths for the job.
- Ensures that the output path does not already exist to prevent errors.
- Adds the top 100 terms and stopwords file to the distributed cache.
- Adds the OpenNLP library to the classpath to enable stemming.
- Runs the job and waits for completion.



MapReduce Job job_1742531231540_0033

Job Name:	TF-IDF Calculation
User Name:	iiitb
Queue:	root.default
State:	SUCCEEDED
Uberized:	false
Submitted:	Sat Mar 22 19:30:22 IST 2025
Started:	Sat Mar 22 19:30:29 IST 2025
Finished:	Sat Mar 22 21:35:55 IST 2025
Elapsed:	2hrs, 5mins, 25sec
Diagnostics:	
Average Map Time	2sec
Average Shuffle Time	1hrs, 46mins, 20sec
Average Merge Time	0sec
Average Reduce Time	2sec

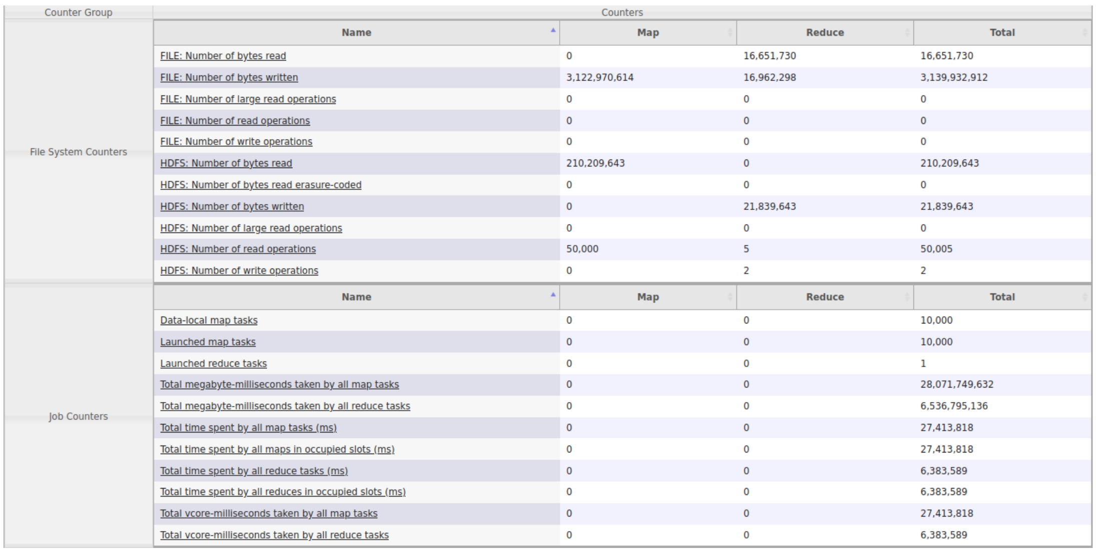
ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Sat Mar 22 19:30:23 IST 2025	peters-pavilion:8042	logs

Task Type	Total	Complete
Map	10000	10000
Reduce	1	1

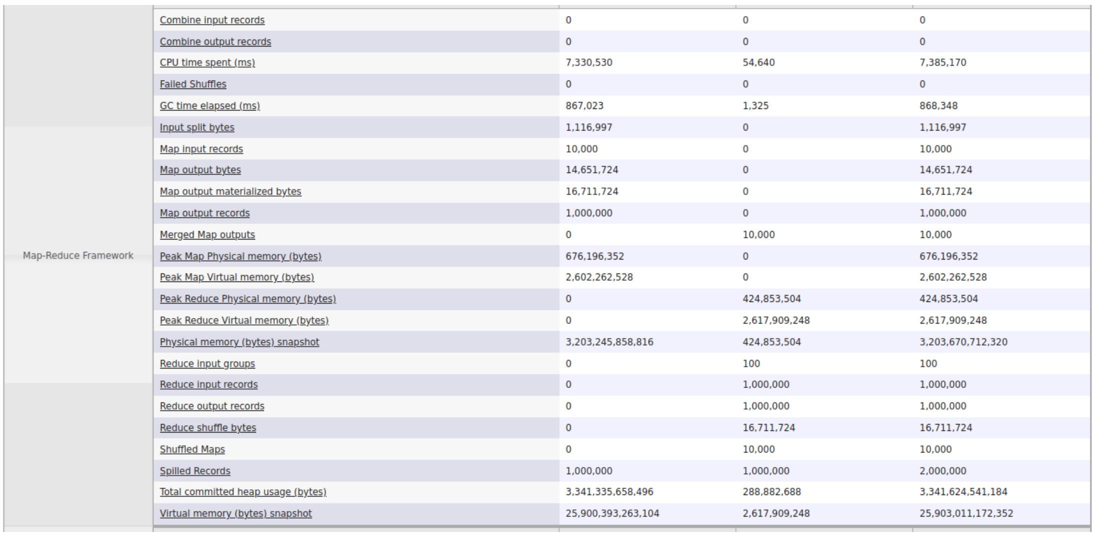
Attempt Type	Failed	Killed	Successful
Maps	0	0	10000
Reduces	0	0	1

Figure 37: MR overview for 5b



Counter Group	Name	Map	Reduce	Total
File System Counters	FILE: Number of bytes read	0	16,651,730	16,651,730
	FILE: Number of bytes written	3,122,970,614	16,962,298	3,139,932,912
	FILE: Number of large read operations	0	0	0
	FILE: Number of read operations	0	0	0
	FILE: Number of write operations	0	0	0
	HDFS: Number of bytes read	210,209,643	0	210,209,643
	HDFS: Number of bytes read erasure-coded	0	0	0
	HDFS: Number of bytes written	0	21,839,643	21,839,643
	HDFS: Number of large read operations	0	0	0
	HDFS: Number of read operations	50,000	5	50,005
	HDFS: Number of write operations	0	2	2
Job Counters	Data-local map tasks	0	0	10,000
	Launched map tasks	0	0	10,000
	Launched reduce tasks	0	0	1
	Total megabyte-milliseconds taken by all map tasks	0	0	28,071,749,632
	Total megabyte-milliseconds taken by all reduce tasks	0	0	6,536,795,136
	Total time spent by all map tasks (ms)	0	0	27,413,818
	Total time spent by all maps in occupied slots (ms)	0	0	27,413,818
	Total time spent by all reduce tasks (ms)	0	0	6,383,589
	Total time spent by all reduces in occupied slots (ms)	0	0	6,383,589
	Total vcore-milliseconds taken by all map tasks	0	0	27,413,818
	Total vcore-milliseconds taken by all reduce tasks	0	0	6,383,589

Figure 38: Some statistics for 5b



Map-Reduce Framework	Combine input records	0	0	0
	Combine output records	0	0	0
	CPU time spent (ms)	7,330,530	54,640	7,385,170
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	867,023	1,325	868,348
	Input split bytes	1,116,997	0	1,116,997
	Map input records	10,000	0	10,000
	Map output bytes	14,651,724	0	14,651,724
	Map output materialized bytes	16,711,724	0	16,711,724
	Map output records	1,000,000	0	1,000,000
	Merged Map outputs	0	10,000	10,000
	Peak Map Physical memory (bytes)	676,196,352	0	676,196,352
	Peak Map Virtual memory (bytes)	2,602,262,528	0	2,602,262,528
	Peak Reduce Physical memory (bytes)	0	424,853,504	424,853,504
	Peak Reduce Virtual memory (bytes)	0	2,617,909,248	2,617,909,248
	Physical memory (bytes) snapshot	3,203,245,858,816	424,853,504	3,203,670,712,320
	Reduce input groups	0	100	100
	Reduce input records	0	1,000,000	1,000,000
	Reduce output records	0	1,000,000	1,000,000
	Reduce shuffle bytes	0	16,711,724	16,711,724
	Shuffled Maps	0	10,000	10,000
	Spilled Records	1,000,000	1,000,000	2,000,000
	Total committed heap usage (bytes)	3,341,335,658,496	288,882,688	3,341,624,541,184
	Virtual memory (bytes) snapshot	25,900,393,263,104	2,617,909,248	25,903,011,172,352

Figure 39: Some more statistics for 5b

Appendix

This assignment was completed in its entirety using a MacOS system.

A.1 Installing Hadoop on MacOS (M1/M2)

- Install Hadoop with brew: $ brew install hadoop
- Refer the article to make necessary modifications to some files: https://medium.com/@MinatoNamikaze02/installing-hadoop-on-macos-m1-m2-2023-d963abeab38e
- To start the Hadoop server: $ start-all.sh
- To stop the Hadoop server: $ stop-all.sh