NoSQL Assignment 2 Part B

Vishruth Vijay IMT2022507
Siddharth Menon IMT2022001
Sathvik S Rao IMT2022082
Shreyas Arun Saggere IMT2022006

March 24, 2025

Problem 4

(a)

The goal of this question was to identify the top 50 most frequently occurring words from a Wikipedia dump while eliminating stop words. The solution was built upon the WordCount2 implementation, incorporating distributed caching to filter out stop words during the mapping phase.

Mapper (TokenizerMapper)

The Mapper class processes input text to extract meaningful words while filtering out stop words.

- **Stop Words Filtering:** The Distributed Cache is used to load stop words from an external file. These are stored in a HashSet for fast lookup.
- **Tokenization:** The input text is split using the regex pattern `[^\w']+` to isolate valid words. This regex pattern was already part of the WordCount2 code and hasn't been changed.

Mapping Process

The Mapper performs the following steps:

1. Reads each line from the input.
2. Converts text to lowercase.
3. Splits the text into tokens and removes stop words.
4. Emits each valid word with a count of 1 in the form: `(word, 1)`

Reducer (IntSumReducer)

The Reducer class aggregates the word counts and extracts the top 50 most frequent words.

- **Counting Words:** Each word's frequency is aggregated using a HashMap.
- **Sorting Logic:** A Priority Queue is used to efficiently track the top 50 words.
- **Final Output:** The top 50 words are written to the output.

Reduction Process

The Reducer executes the following steps:

1. Aggregates word counts for each key.
2. Stores results in a HashMap.
3. Inserts all word-frequency pairs into a Priority Queue.
4. Extracts and writes the top 50 most frequent words.

Driver Program

The Driver program initializes and configures the Hadoop job. The following key settings are configured:

- Input and output paths are specified.
- The stop words file is added to the Distributed Cache using:

```
job.addCacheFile(new Path(args[i]).toUri());
```

- Optional case sensitivity is enabled using the argument `-casesensitive`. This was part of the WordCount2 java code, while running our hadoop task, we do not pass an argument for this, and hence runs on default non-case sensitivity.

(b)

The objective of this question was to construct a **co-occurrence word matrix** based on the **frequent words** identified in the previous question. The program employs the **Pairs Approach** to compute co-occurrences within a specified **window distance** d. The solution filters out less frequent words using a precomputed stop-word list, ensuring that only significant word pairs are considered.

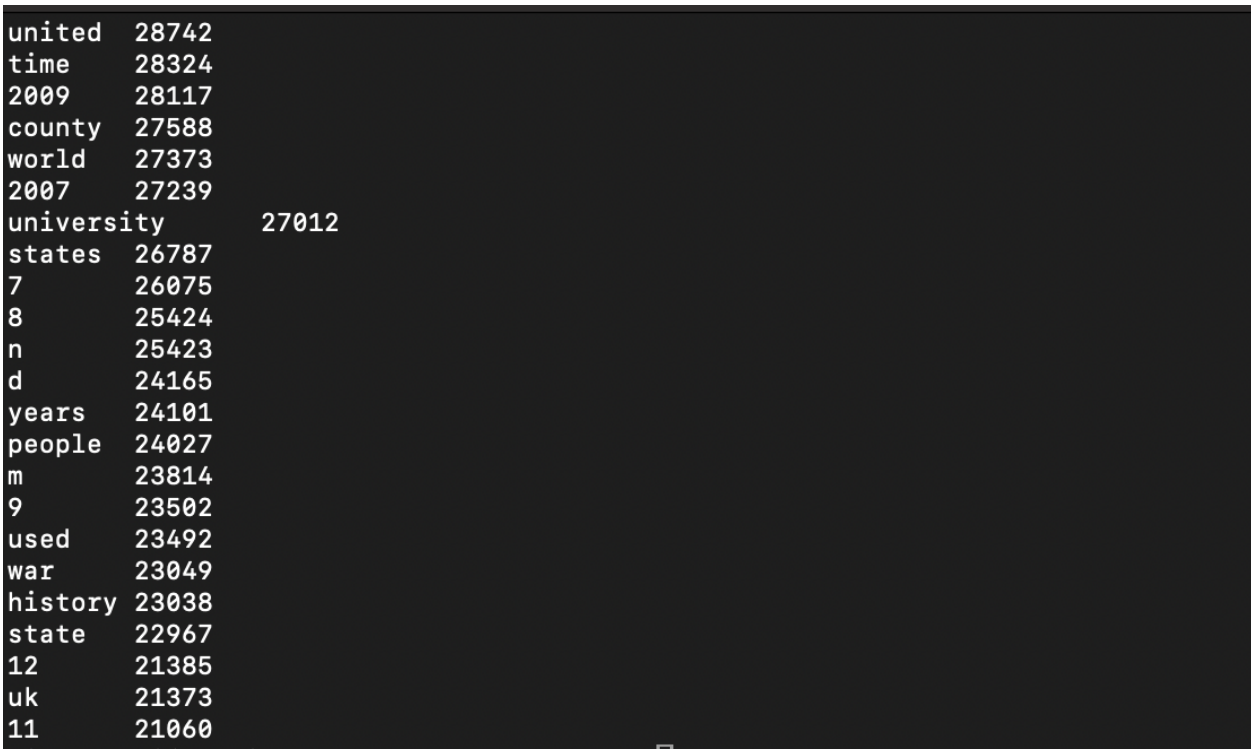

Figure 1: Top 50 words based on frequency of occurrance

Mapper (PairsMapper)

The Mapper class reads input text and constructs co-occurrence pairs.

- **Frequent Words Filtering:** A Distributed Cache file is used to load frequent words, ensuring only significant words are included in co-occurrence calculations.
- **Adjustable Window Size:** The parameter d determines the maximum distance within which two words are considered co-occurring.
- **Pairs Generation:** Word pairs are emitted in the format `(word1, word2, 1)`.

Mapping Process

1. The input line is tokenized into words.
2. Each word is cleaned (punctuation removal, lowercase conversion).
3. If the word is in the **frequent words set**, a co-occurrence window is created around it.
4. Co-occurring word pairs within the window distance d are emitted.

Reducer (PairsReducer)

The Reducer class aggregates the frequency of each word pair.

- **Summing Pair Counts:** For each word pair, the frequency is summed across all input splits.
- **Final Output:** The total count of each pair is written to the output.

Reduction Process

1. The Reducer receives key-value pairs of the form $(word1, word2), count$.
2. The total frequency for each pair is computed.
3. The final count is emitted as output.

Driver Program

The Driver program configures the Hadoop job with the following settings:

- The input and output paths are specified.
- The frequent words file is added to the **Distributed Cache**.
- The window distance d is set using:

```
conf.setInt("windowDistance", windowDistance);
```

- The Mapper and Reducer classes are registered.
- The job is submitted to Hadoop for execution.

Runtime Comparison

Window Distance (d)	**Execution Time (seconds)**
1	173.536
2	177.309
3	179.853
4	187.254

Table 1: Runtime Analysis for Different Window Distances

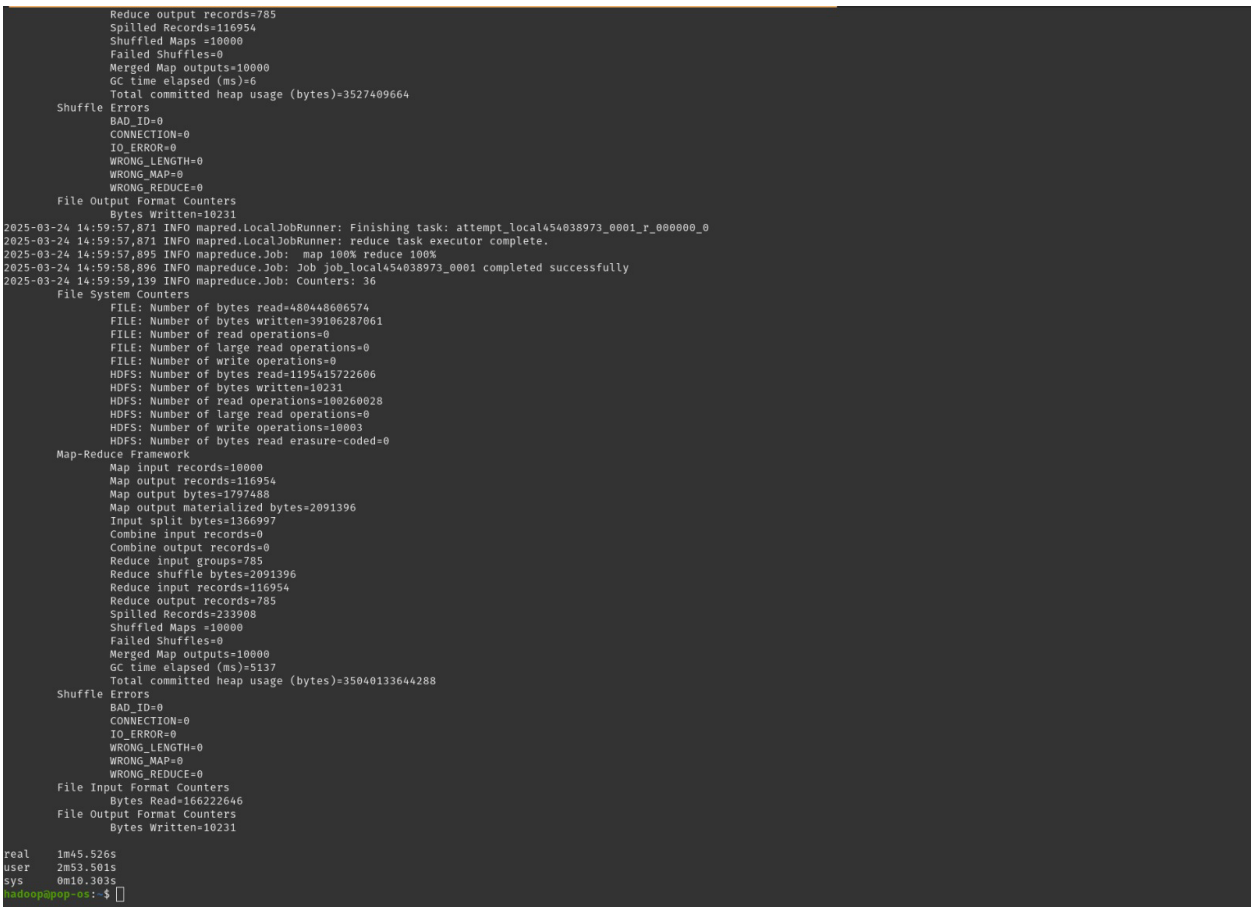

Figure 2: The figure represents the output for pairs approach (d=1)

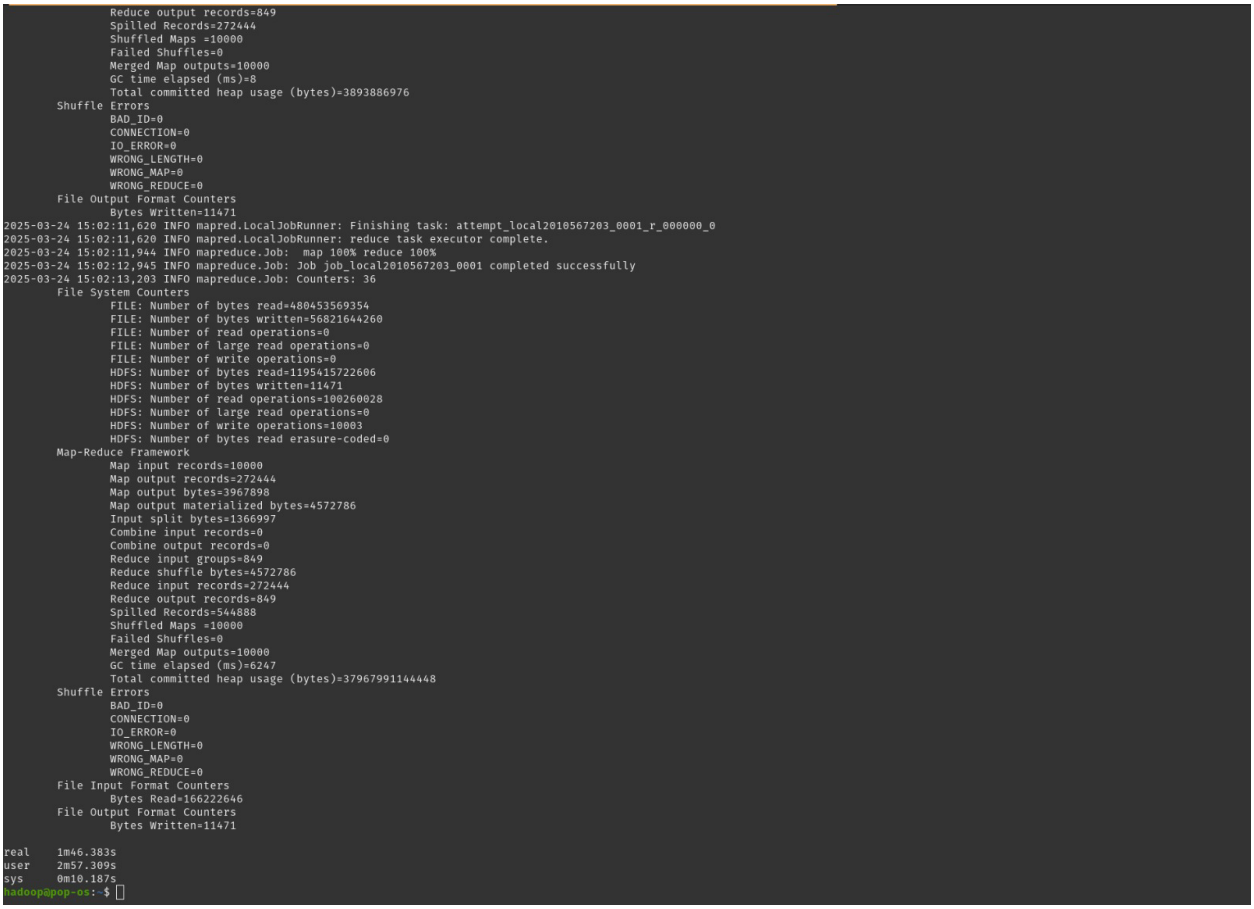

Figure 3: The figure represents the output for pairs approach (d=2)

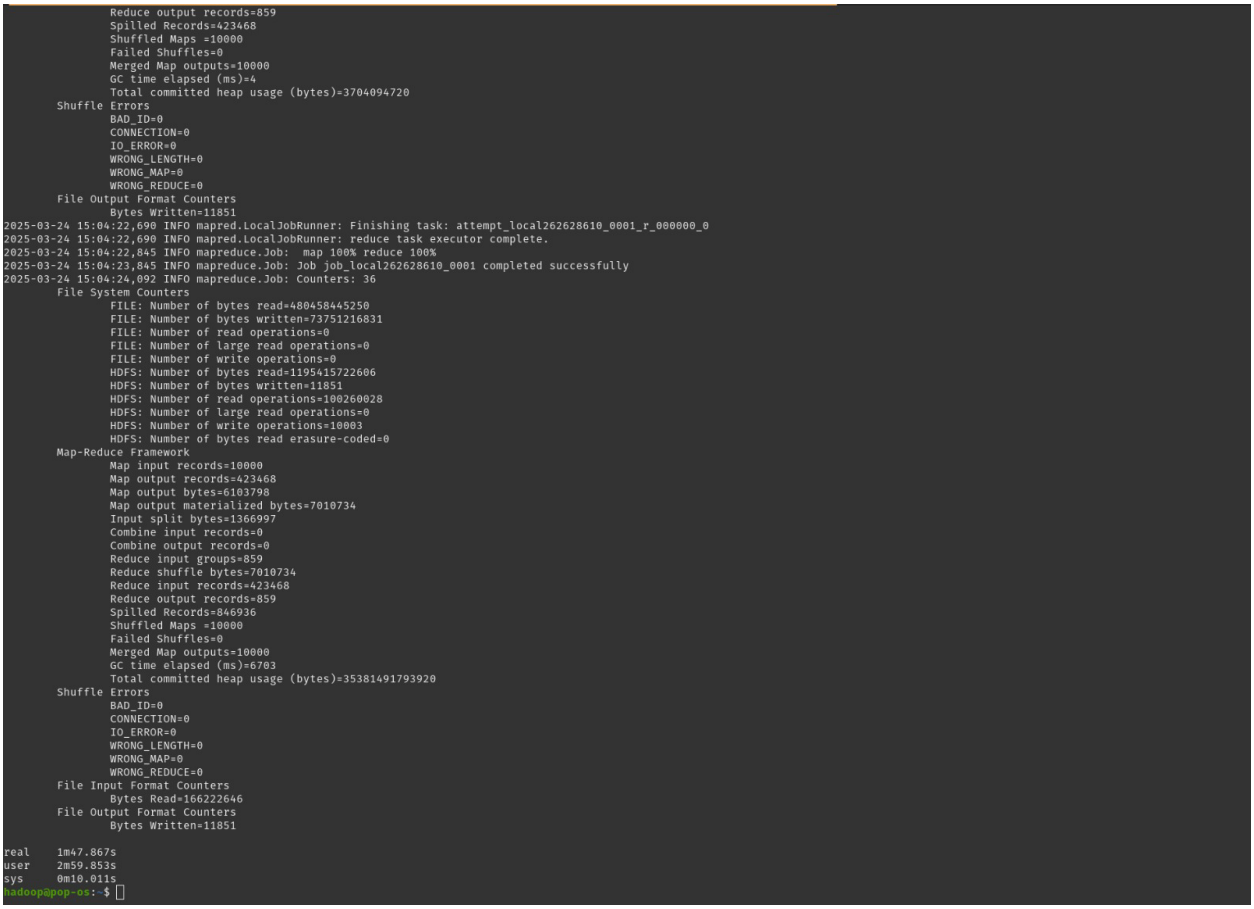

Figure 4: The figure represents the output for pairs approach (d=3)

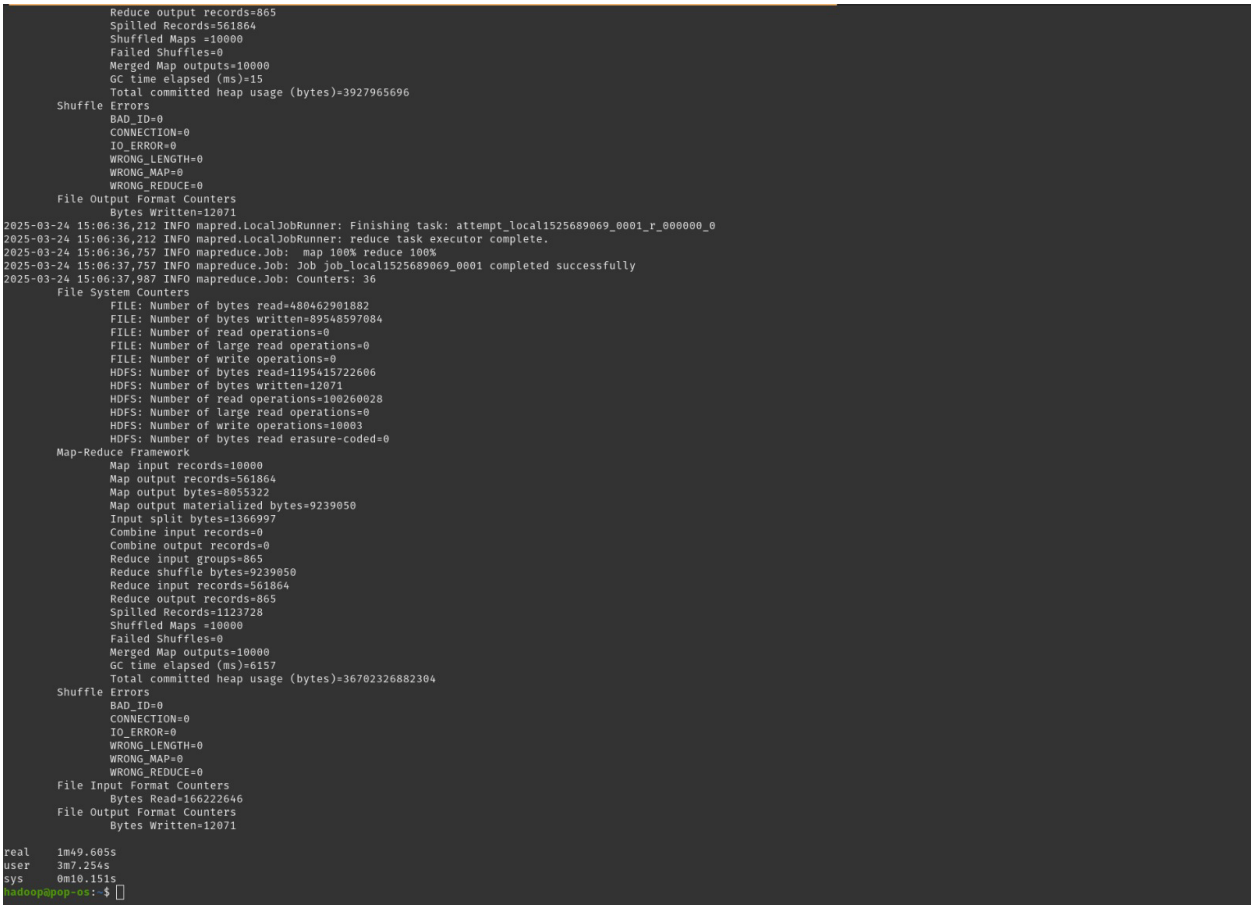

Figure 5: The figure represents the output for pairs approach (d=4)

(c)

The goal of the question was to construct a **co-occurrence word matrix** using the **Stripe Algorithm**. The implementation considers only the **frequent words** identified in the subpart (a) and computes co-occurring word pairs within a specified **window distance** d. This approach leverages a more efficient data structure compared to the Pairs Approach, reducing the number of intermediate key-value pairs transmitted.

Mapper (StripesMapper)

The Mapper processes the input text to identify co-occurring words within a given window distance.

- **Frequent Words Filtering:** A Distributed Cache file is loaded to retain only important words.
- **Window-Based Co-occurrence:** Each word forms a local co-occurrence matrix (stripe) with its neighbors.
- **Efficient Data Representation:** Instead of emitting individual word pairs, a single stripe (hashmap) is emitted per word.

Mapping Process

1. The input line is tokenized into words.
2. Words are cleaned (punctuation removal, lowercase conversion).
3. If the word is frequent, a **co-occurrence stripe** is created.
4. Co-occurrence counts are updated in the stripe.
5. The word and its stripe are emitted in the format:

```
(word1, {word2: count, word3: count, ...})
```

Reducer (StripesReducer)

The Reducer merges stripes from multiple mappers, ensuring that final co-occurrence counts are computed.

- **Stripe Merging:** Partial counts from multiple stripes are aggregated.
- **Final Output Format:** The final output contains each word and its corresponding co-occurrence matrix.

Reduction Process

1. Receives key-value pairs where each key is a word and each value is a stripe.
2. Merges all stripes for the word.
3. Outputs the final co-occurrence matrix for each word.

Driver Program

The Driver program initializes the Hadoop job with the following configurations:

- Input and output paths are specified.
- The frequent words file is added to the **Distributed Cache**.
- The window distance d is set dynamically using:

```
conf.setInt("windowDistance", windowDistance);
```

- The Mapper and Reducer classes are registered.
- The job is submitted to Hadoop for execution.

Runtime Comparison

Window Distance (d)	**Execution Time (seconds)**
1	172.412
2	182.862
3	192.073
4	199.560

Table 2: Runtime Analysis for Different Window Distances

(d)

The goal of this question is to provide a comparative analysis of local aggregation techniques applied to the **Pairs** and **Stripes Approaches** for constructing a **co-occurrence word matrix**. The primary objective is to optimize computation by reducing the number of intermediate key-value pairs generated during the Map phase. Two levels of local aggregation are implemented:

- **Map-Function Level Aggregation:** Aggregates data within a single call to the `map()` function.
- **Map-Class Level Aggregation:** Aggregates data across multiple calls to `map()` within the same Mapper instance.

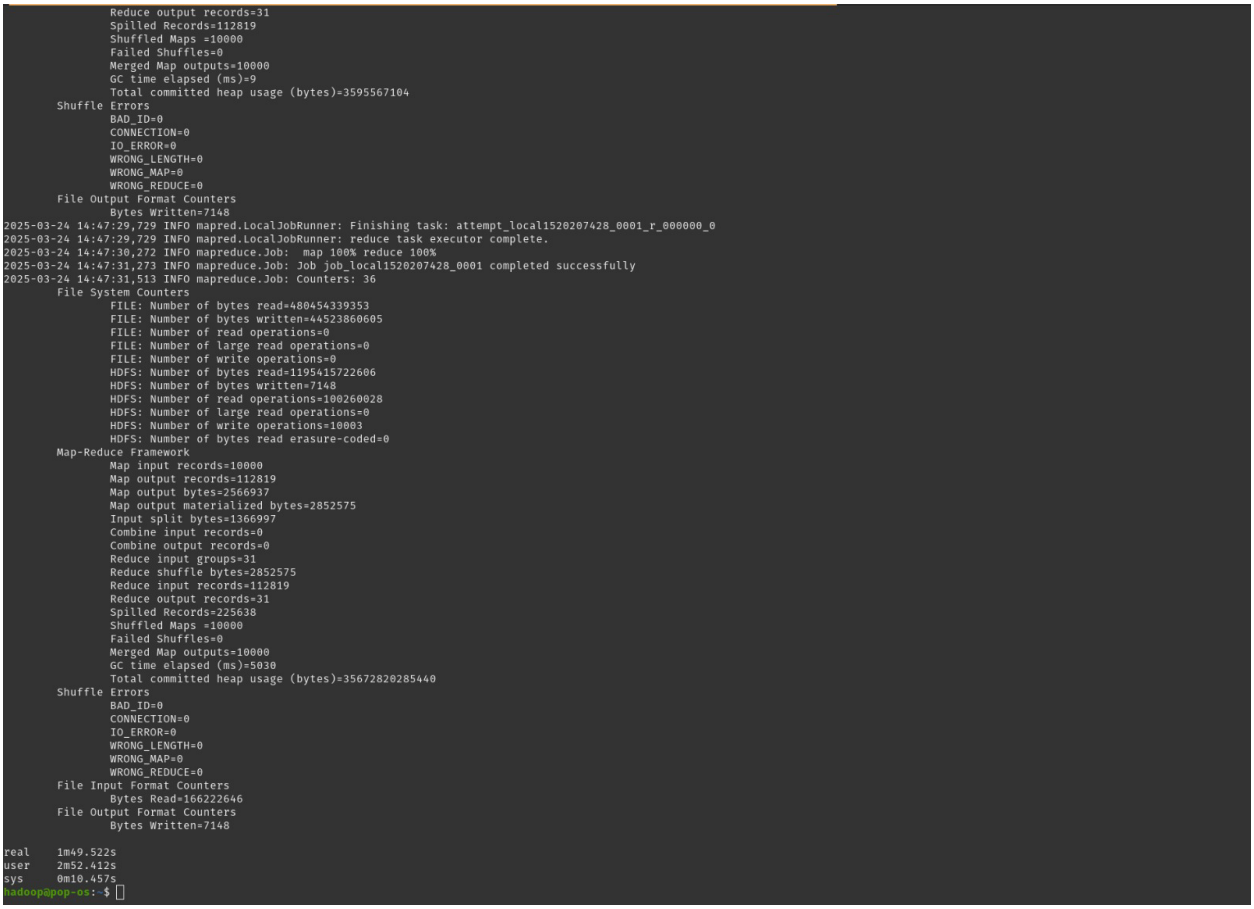

Figure 6: The figure represents the output for stripes approach (d=1)

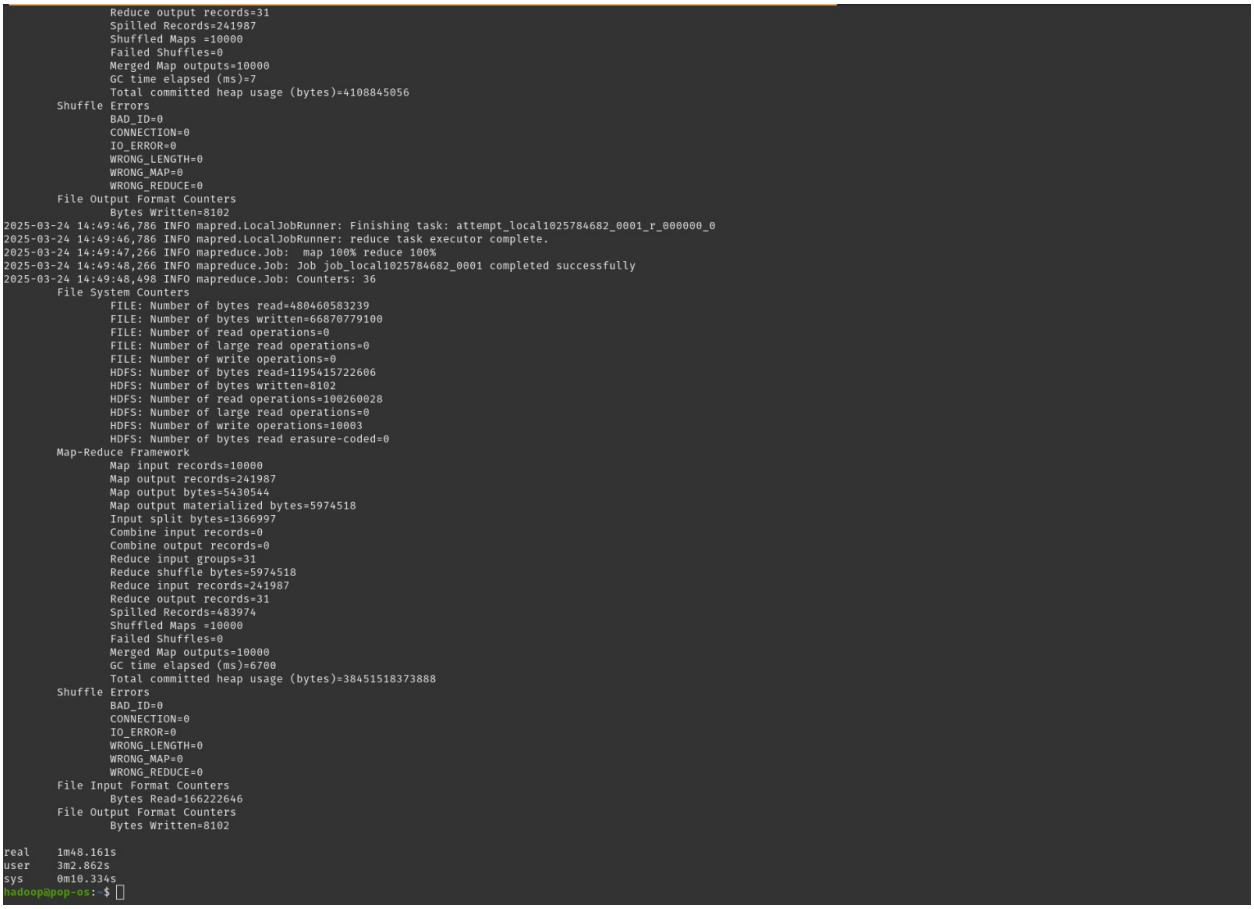

Figure 7: The figure represents the output for stripes approach (d=2)

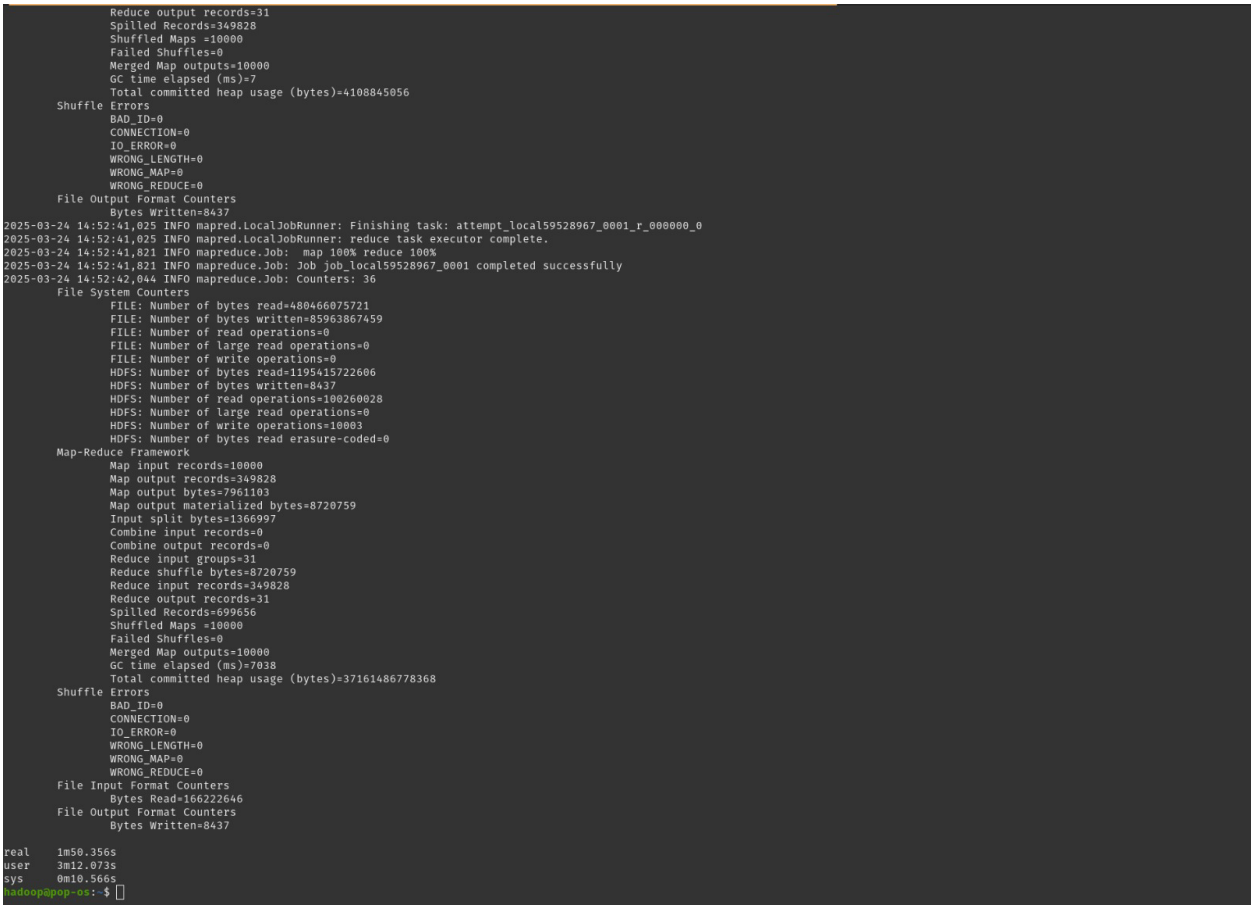

Figure 8: The figure represents the output for stripes approach (d=3)

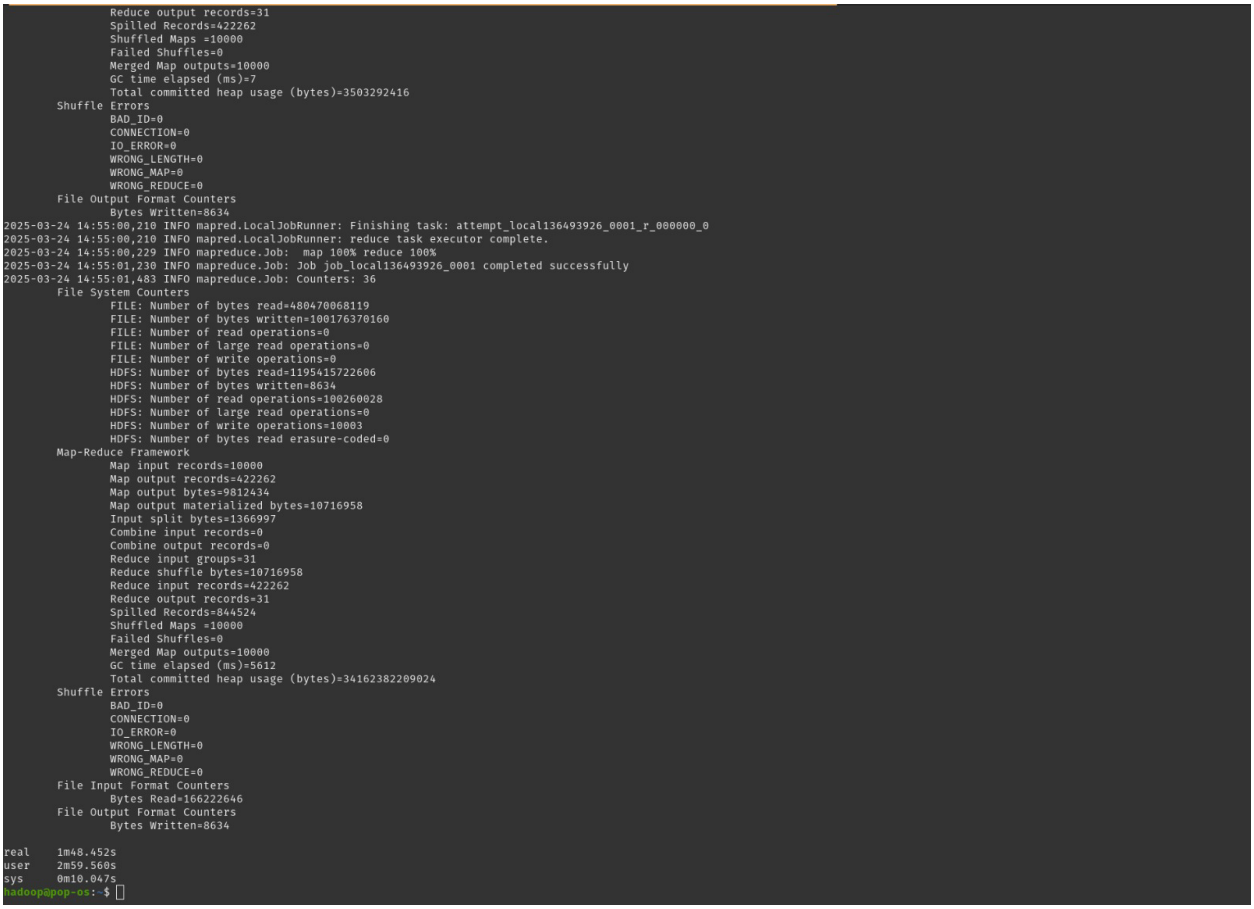

Figure 9: The figure represents the output for stripes approach (d=4)

The runtime is analyzed for different window distances $d = \{1, 2, 3, 4\}$. The implementation introduces an **aggregationLevel** parameter to toggle between different aggregation strategies.

Pairs Approach with Aggregation

The `PairsAggregation` program builds upon the traditional **Pairs Approach** that was implemented in (b), introducing local aggregation before emitting word pairs.

Mapper (PairsMapper)

- **Function-Level Aggregation:** Uses a `HashMap` within each call to `map()` to store word pair counts before emitting them.
- **Class-Level Aggregation:** Stores counts in a `HashMap` across multiple calls to `map()`, only emitting pairs during `cleanup()`.

Reducer (PairsReducer)

The Reducer sums all emitted word pair counts and outputs the final co-occurrence matrix.

Stripes Approach with Aggregation

The `StripesAggregation` program optimizes the **Stripes Approach** that was implemented in (c) by introducing local aggregation.

Mapper (StripesMapper)

- **Function-Level Aggregation:** A `MapWritable` (hashmap) is used within each call to `map()`.
- **Class-Level Aggregation:** A single `MapWritable` is maintained across multiple calls to `map()` and emitted in `cleanup()`.

Reducer (StripesReducer)

Receives and merges multiple stripes to generate the final co-occurrence matrix.

Window Distance (d)	Pairs (Function-Level)	Pairs (Class-Level)	Stripes (Function-Level)	Stripes (Class-Level)
1	8m:41s	8m:41s	6m:9s	6m:35s
2	6m:38sec	6m:50s	6m:39s	6m:37s
3	6m:42sec	6m:43s	6m:43s	6m:43s
4	6m:46sec	6m:51s	6m:59s	6m:57s

Table 3: Runtime Analysis for Different Window Distances and Aggregation Levels

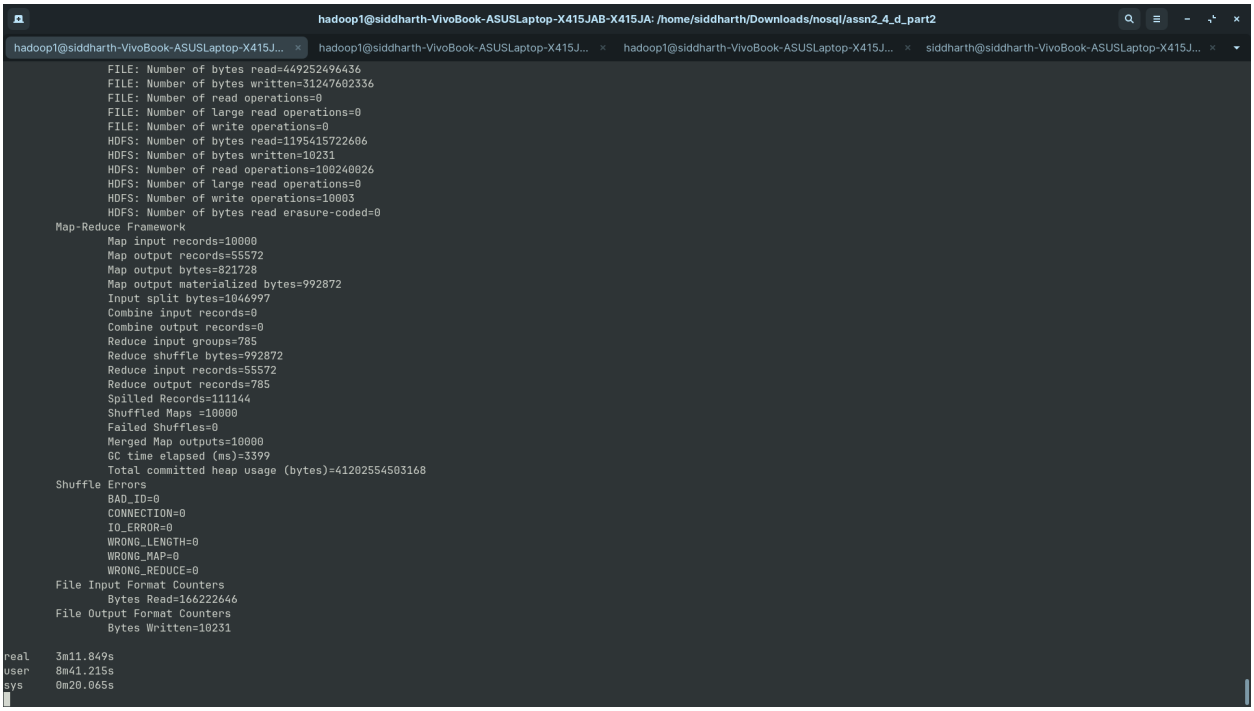

Figure 10: Pairs Approach with function level aggregation (d=1)

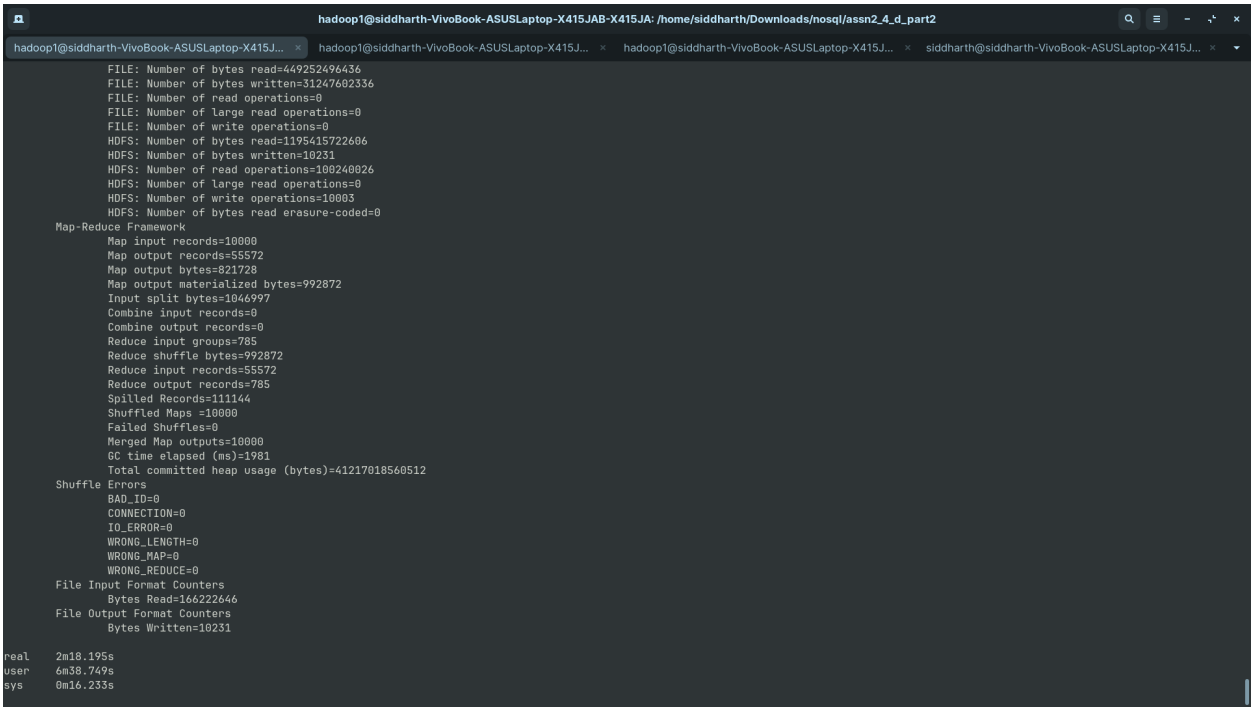

Figure 11: Pairs Approach with function level aggregation (d=2)

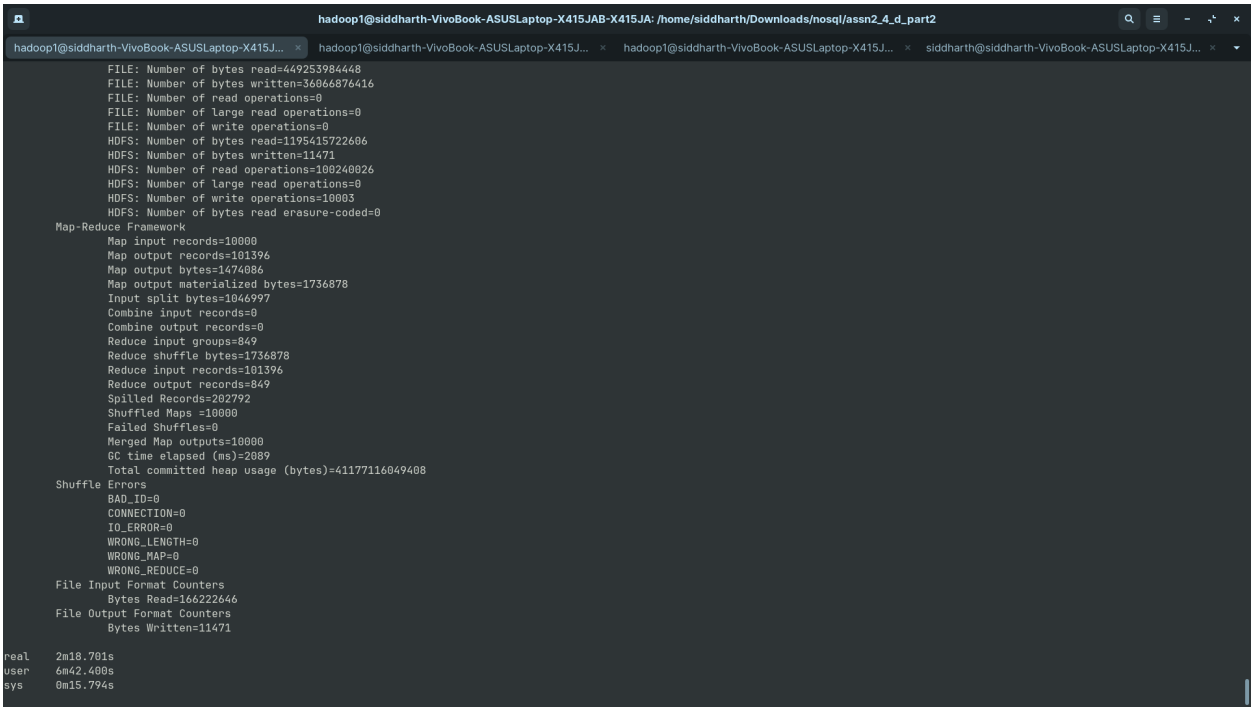

Figure 12: Pairs Approach with function level aggregation (d=3)

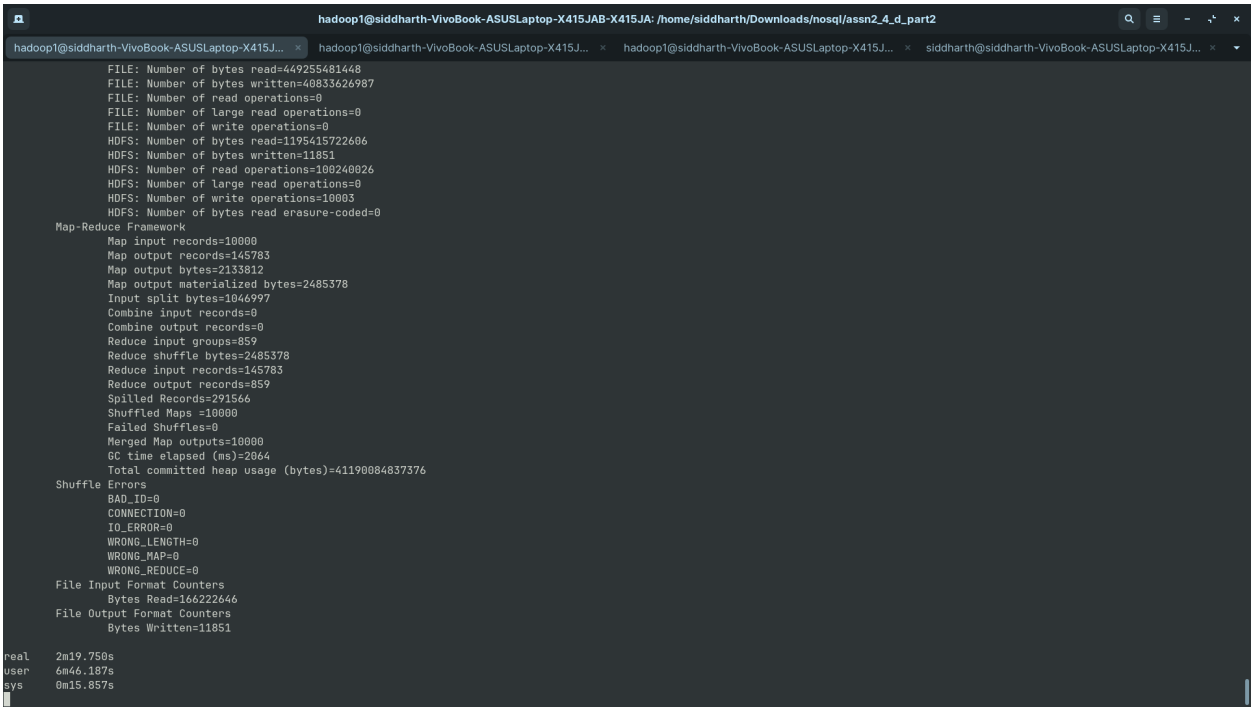

Figure 13: Pairs Approach with function level aggregation (d=4)

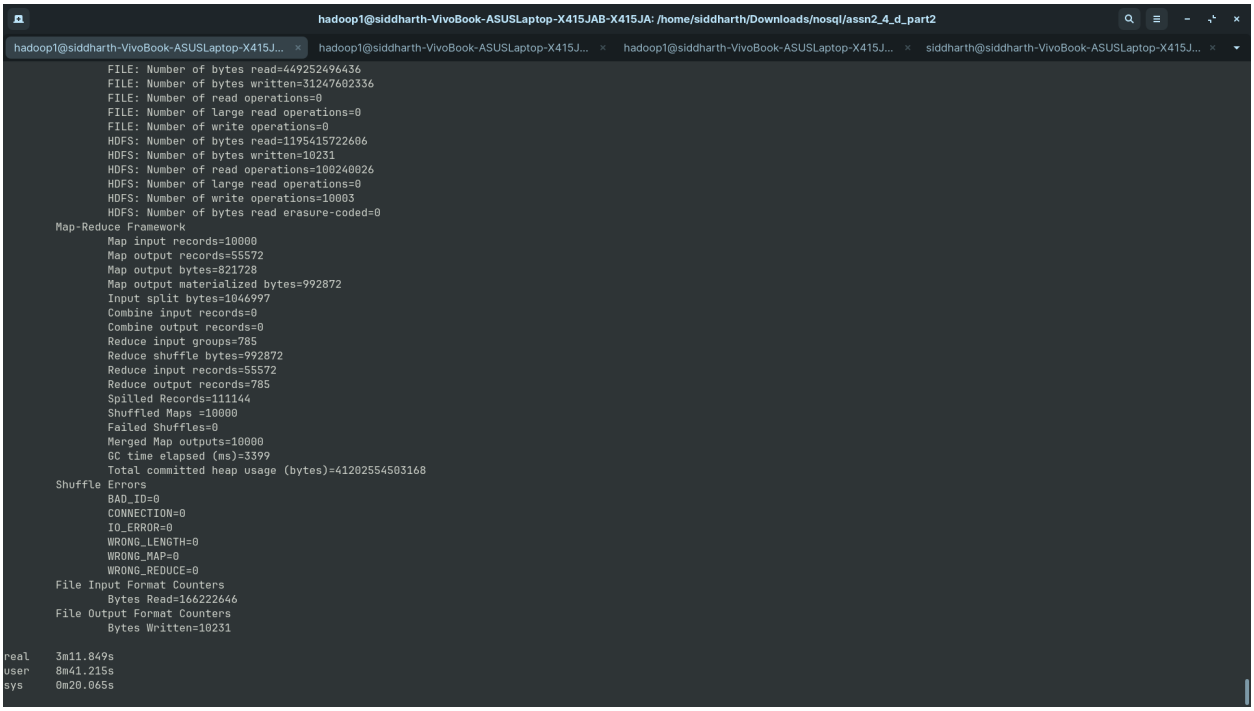

Figure 14: Pairs Approach with class level aggregation (d=1)

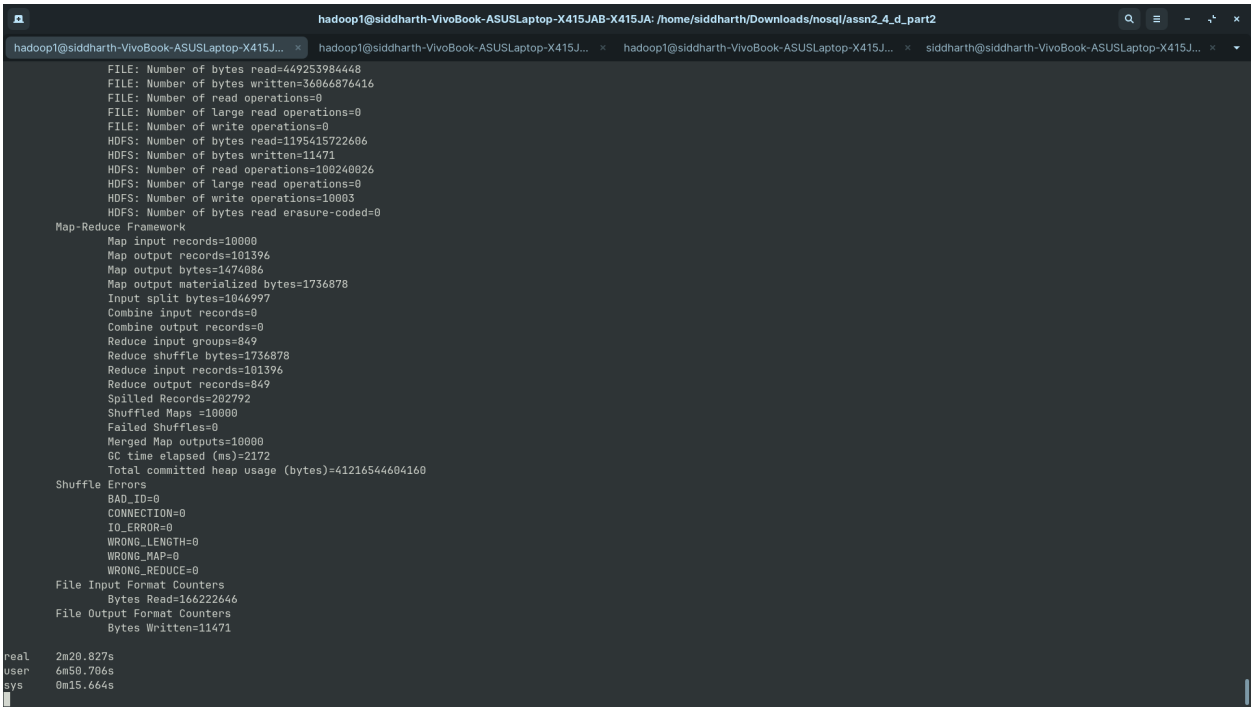

Figure 15: Pairs Approach with class level aggregation (d=2)

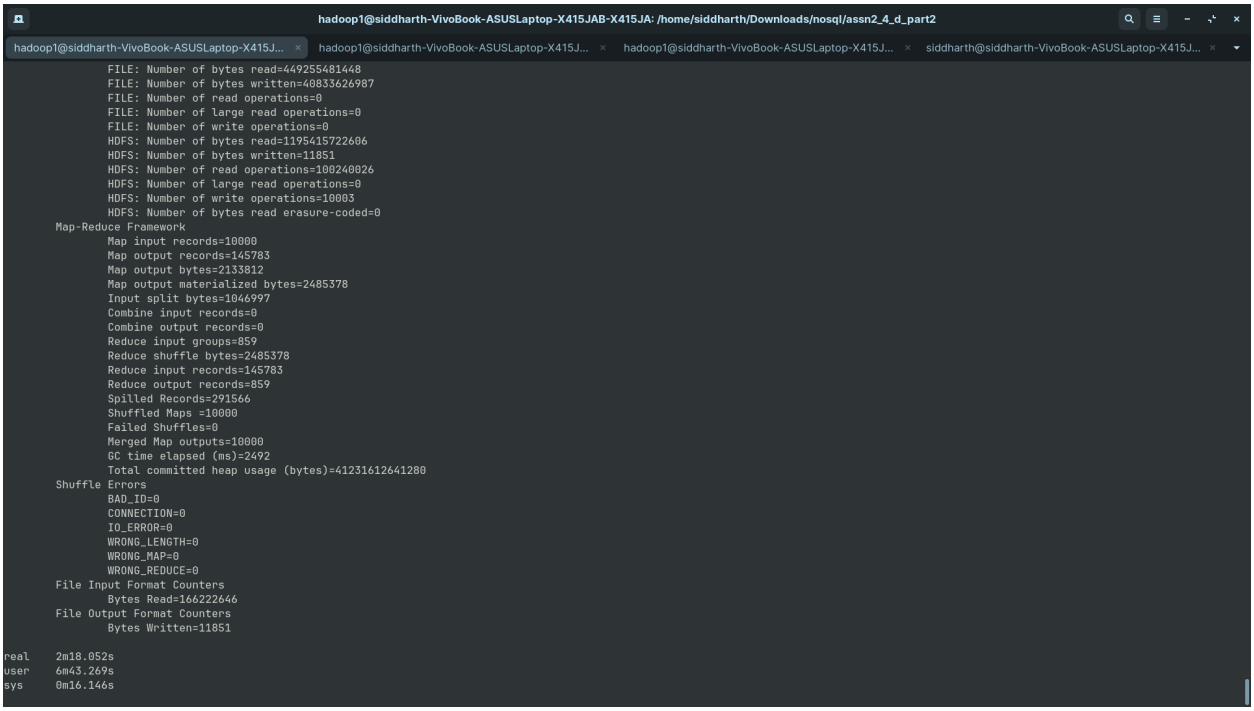

Figure 16: Pairs Approach with class level aggregation (d=3)

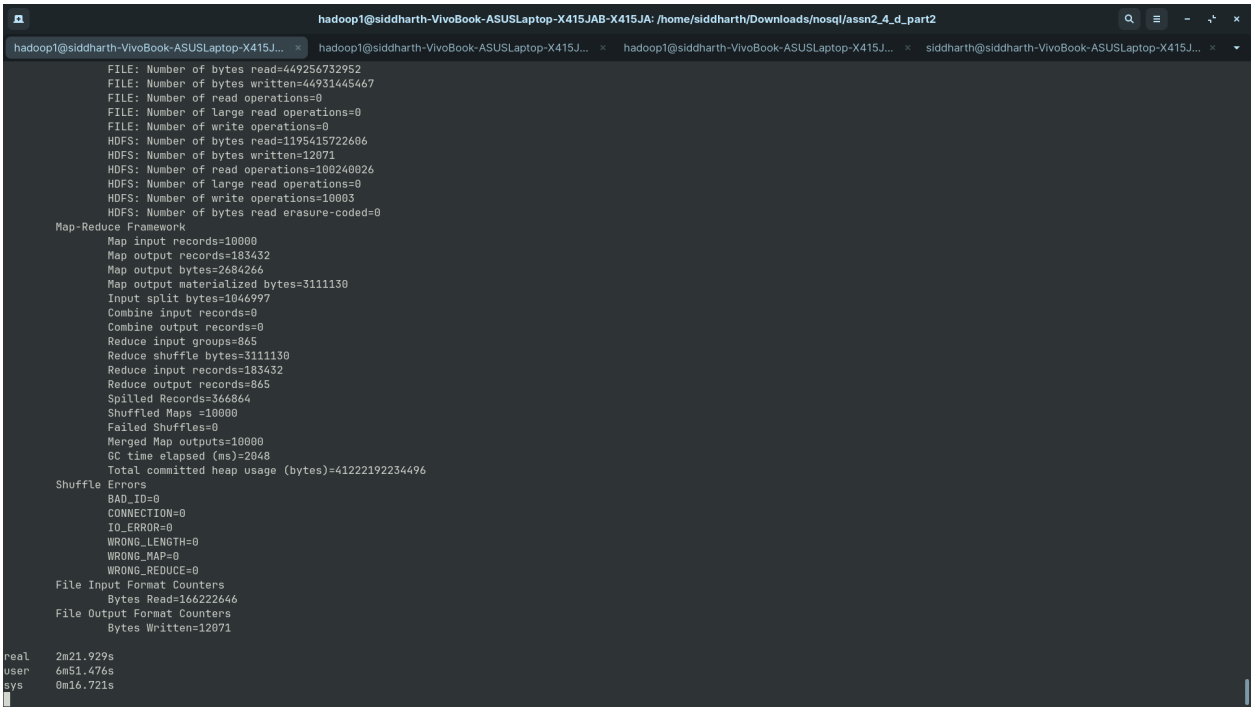

Figure 17: Pairs Approach with class level aggregation (d=4)

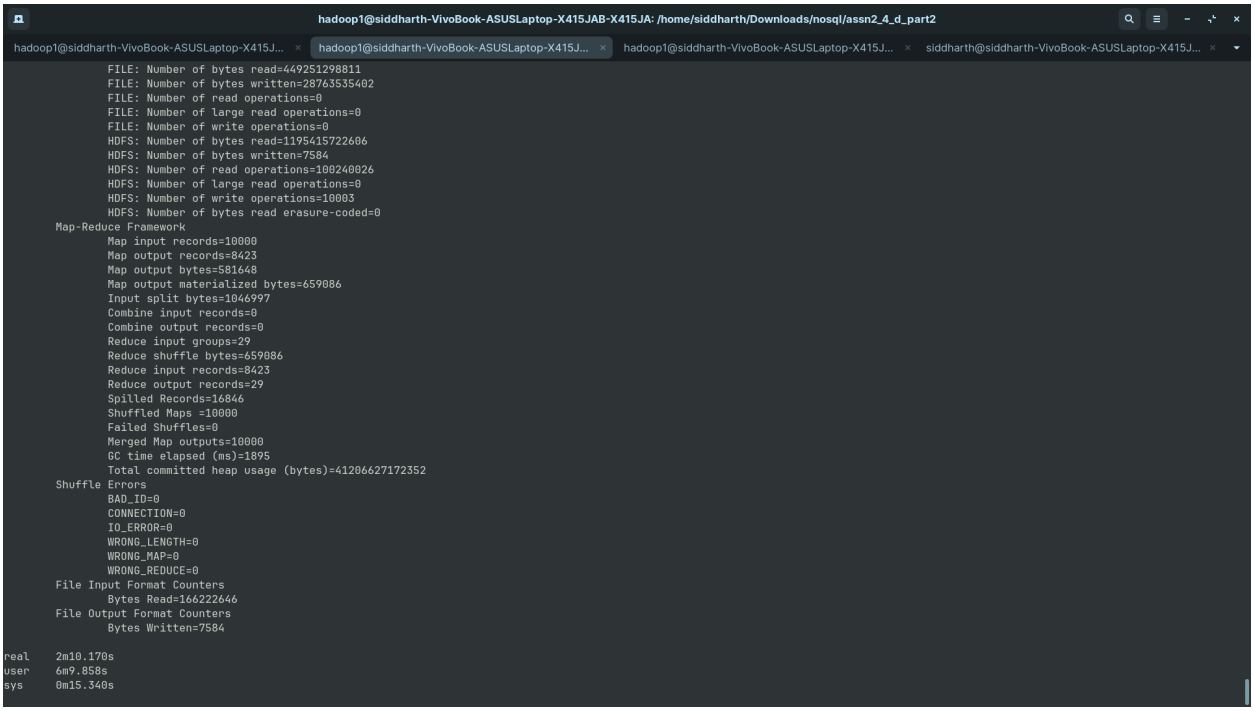

Figure 18: Stripes Approach with function level aggregation (d=1)

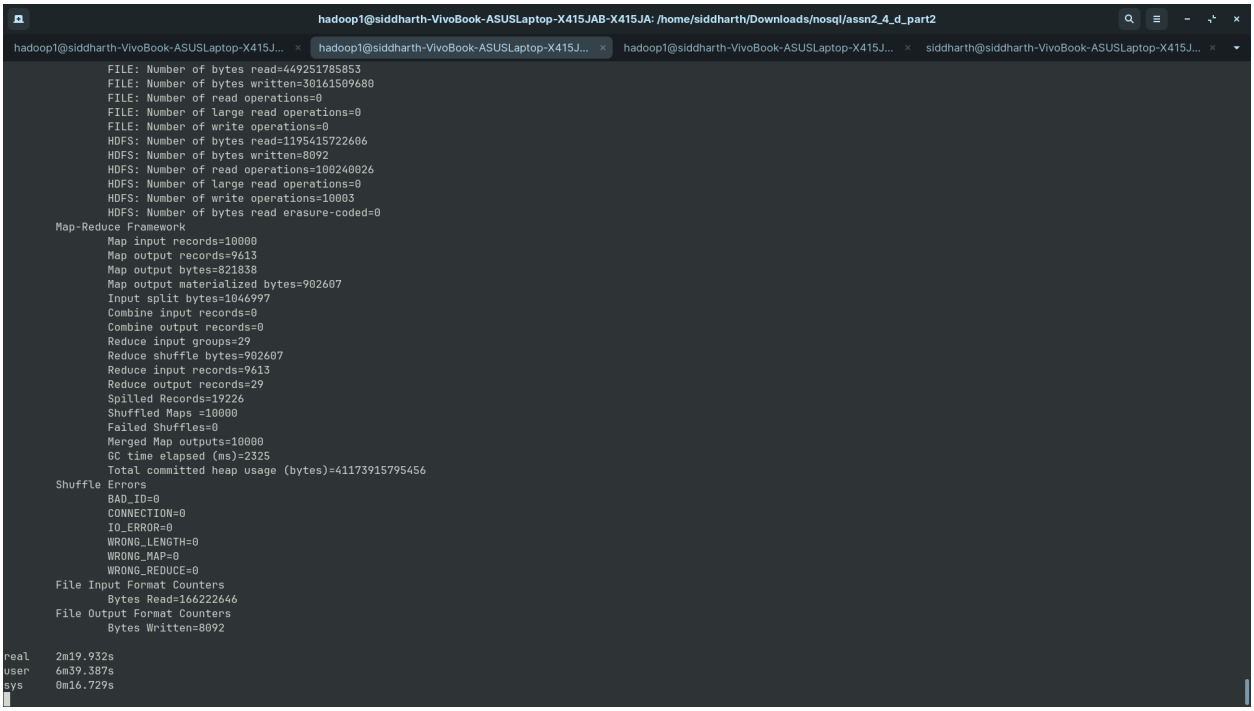

Figure 19: Stripes Approach with function level aggregation (d=2)

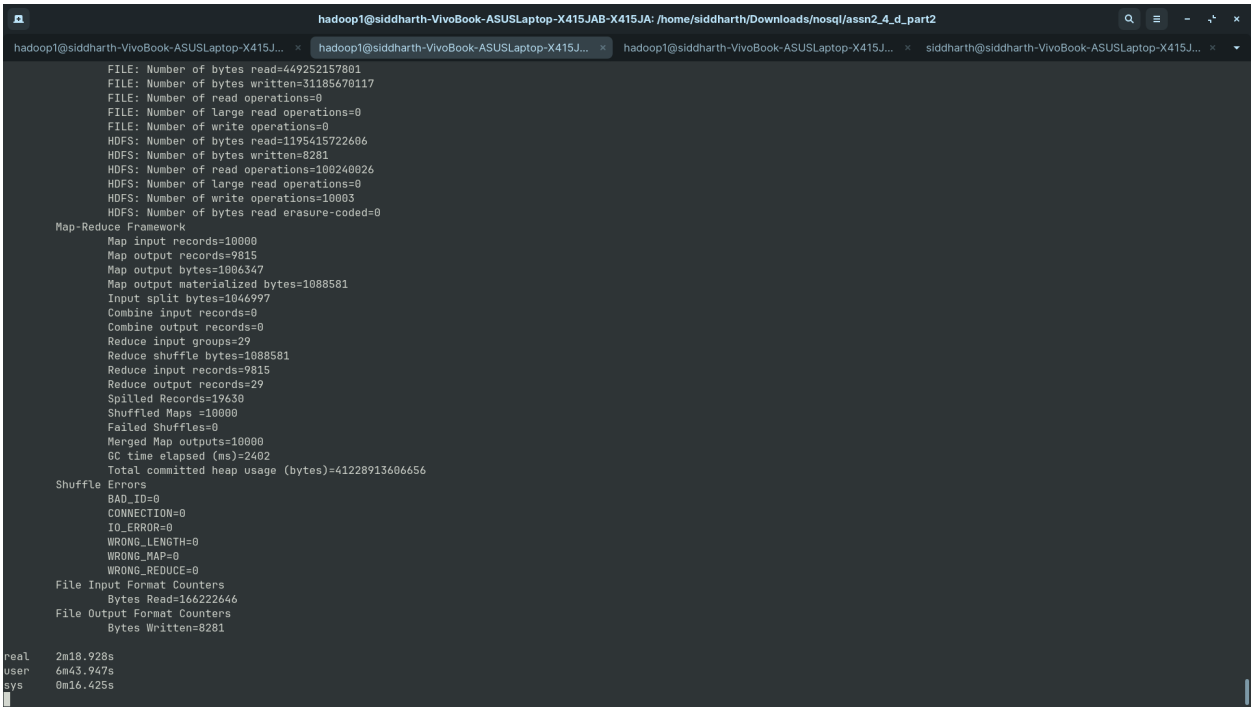

Figure 20: Stripes Approach with function level aggregation (d=3)

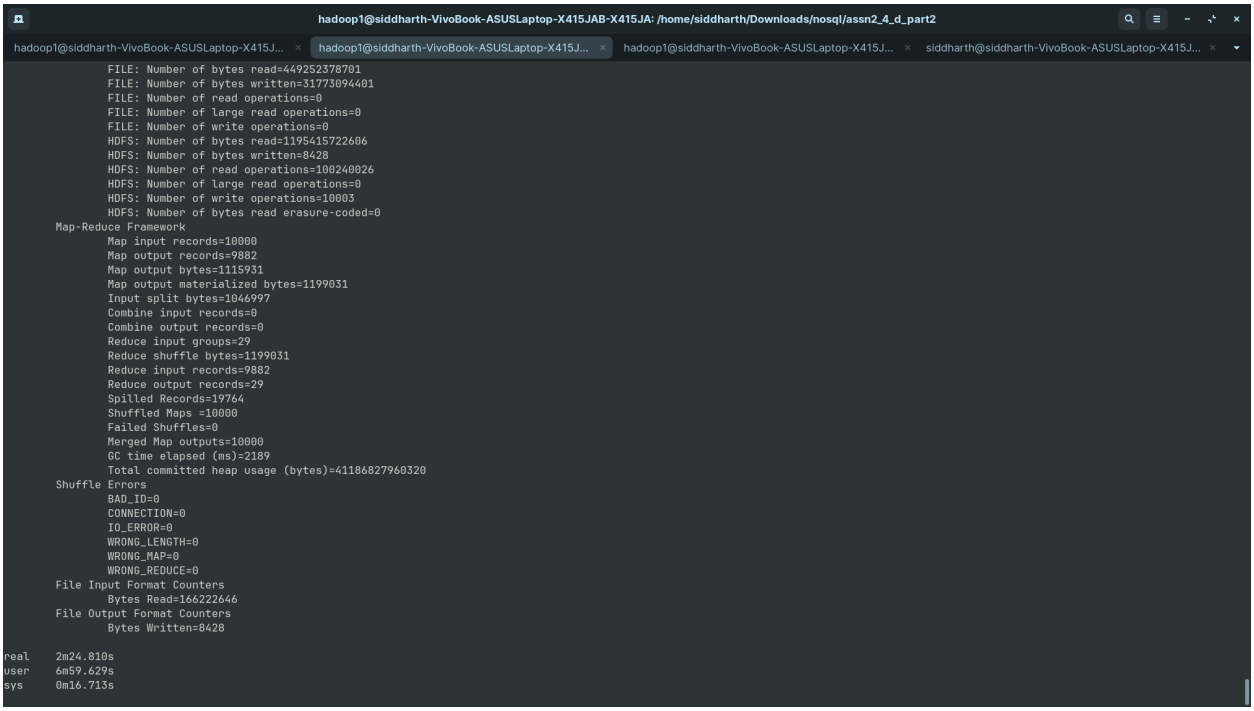

Figure 21: Stripes Approach with function level aggregation (d=4)

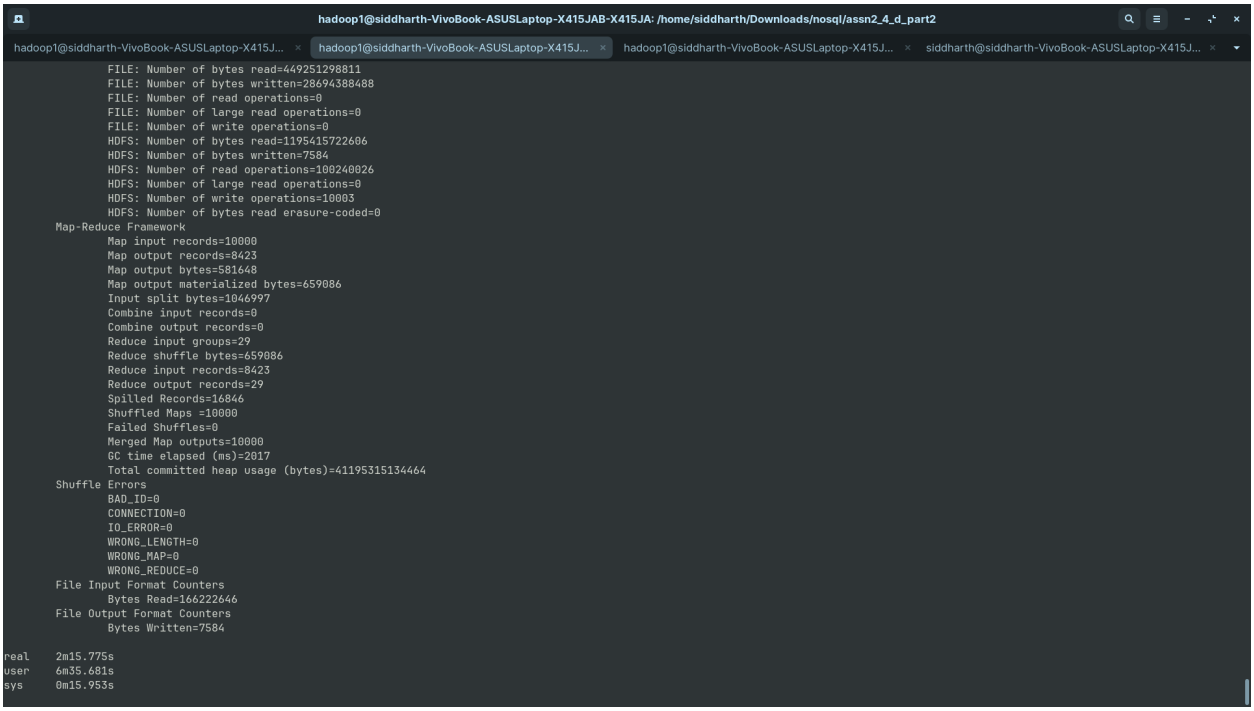

Figure 22: Stripes Approach with class level aggregation (d=1)

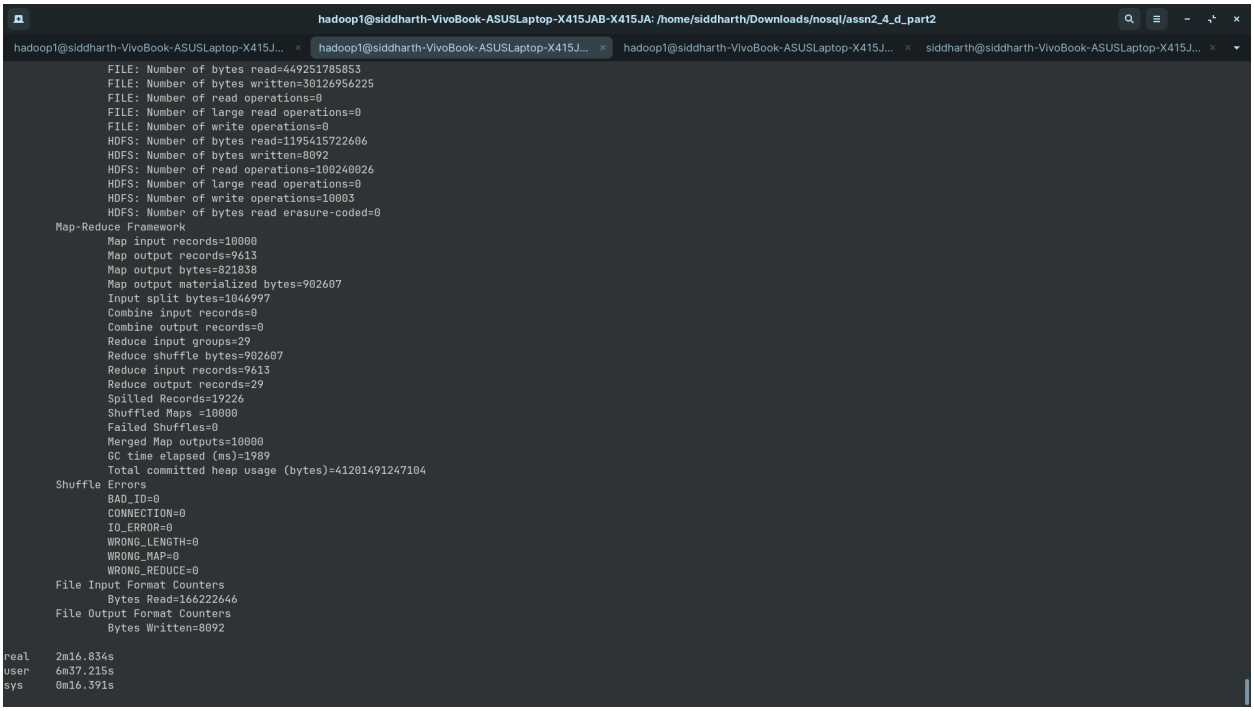

Figure 23: Stripes Approach with class level aggregation (d=2)

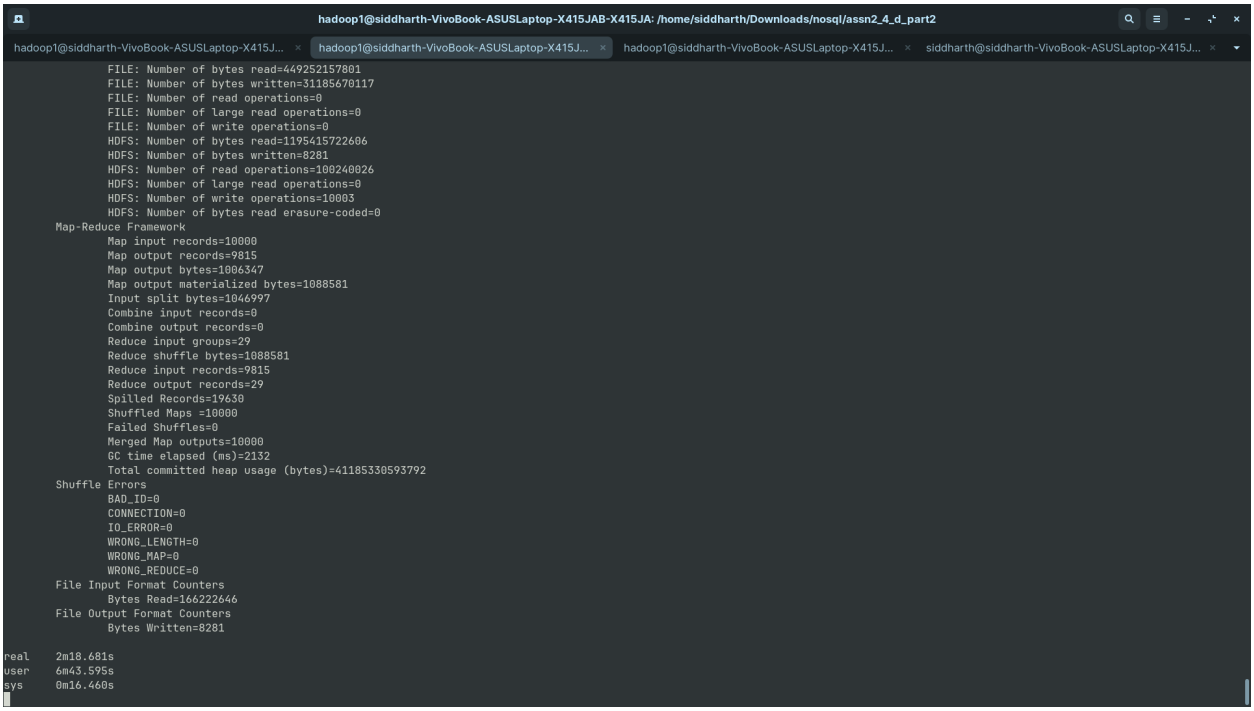

Figure 24: Stripes Approach with class level aggregation (d=3)

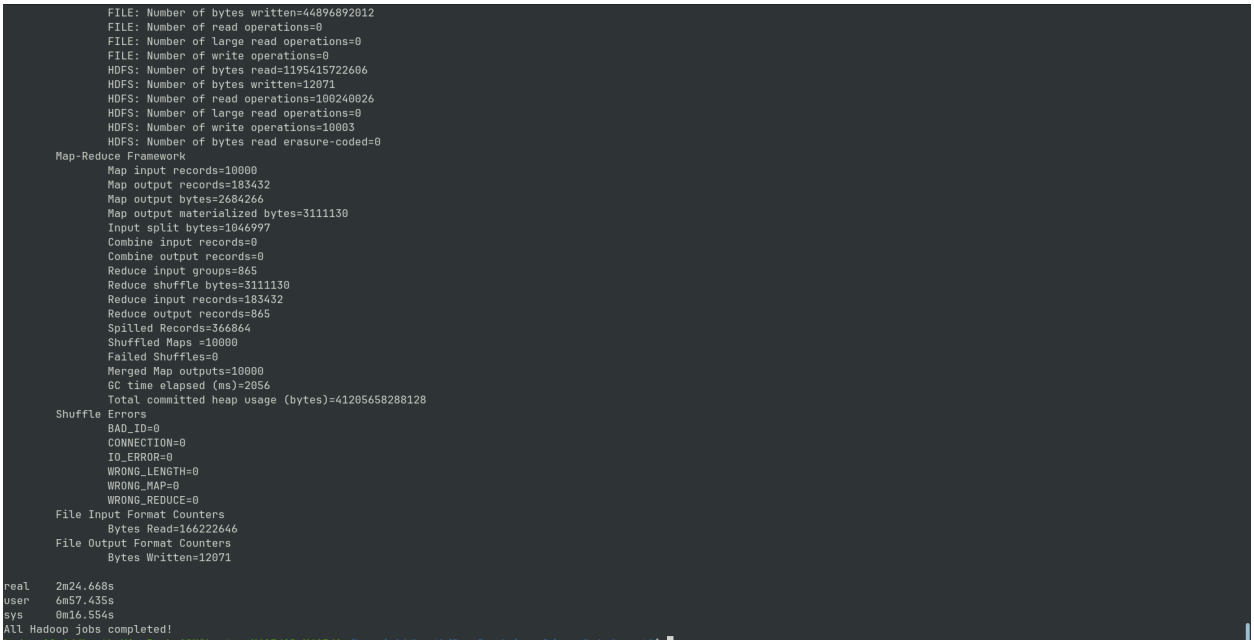

Figure 25: Stripes Approach with class level aggregation (d=4)

Problem 5

The objective of this problem is to compute the **Term Frequency-Inverse Document Frequency (TF-IDF)** scores for stemmed words across a collection of 10,000 Wikipedia articles.

(a) Document Frequency Computation

In this step, Map-Reduce is used to determine the number of documents in which each term appears.

Mapper

- Reads input data, converts it to lowercase, and tokenizes using whitespaces.
- Removes special characters and applies Porter Stemmer from OpenNLP.
- Excludes stopwords listed in *stopwords.txt*.
- Emits key-value pairs in the format `<term, docID>`.

Reducer

- Receives a key (term) and a list of document IDs where the term appears.
- Computes Document Frequency (DF) for each term.
- Outputs key-value pairs in the format `<term, DF>`.
- We store the top 100 words with the highest DF in HDFS.

(b) TF-IDF Computation

This step computes the Term Frequency-Inverse Document Frequency (TF-IDF) scores using Map-Reduce.

Setup

Before execution of the Map Reduce Job, the top 100 words and their document frequencies (DF) are loaded from a cached file(Distributed Cache) into a map structure for quick lookup.

Mapper

- Reads each document, converts it to lowercase, and tokenizes it using `\s+`.
- Removes special characters and applies Porter Stemmer for word normalization.
- Counts word occurrences only if they appear in the top 100 words list.
- Emits key-value pairs `<filename, word-frequency map>`.

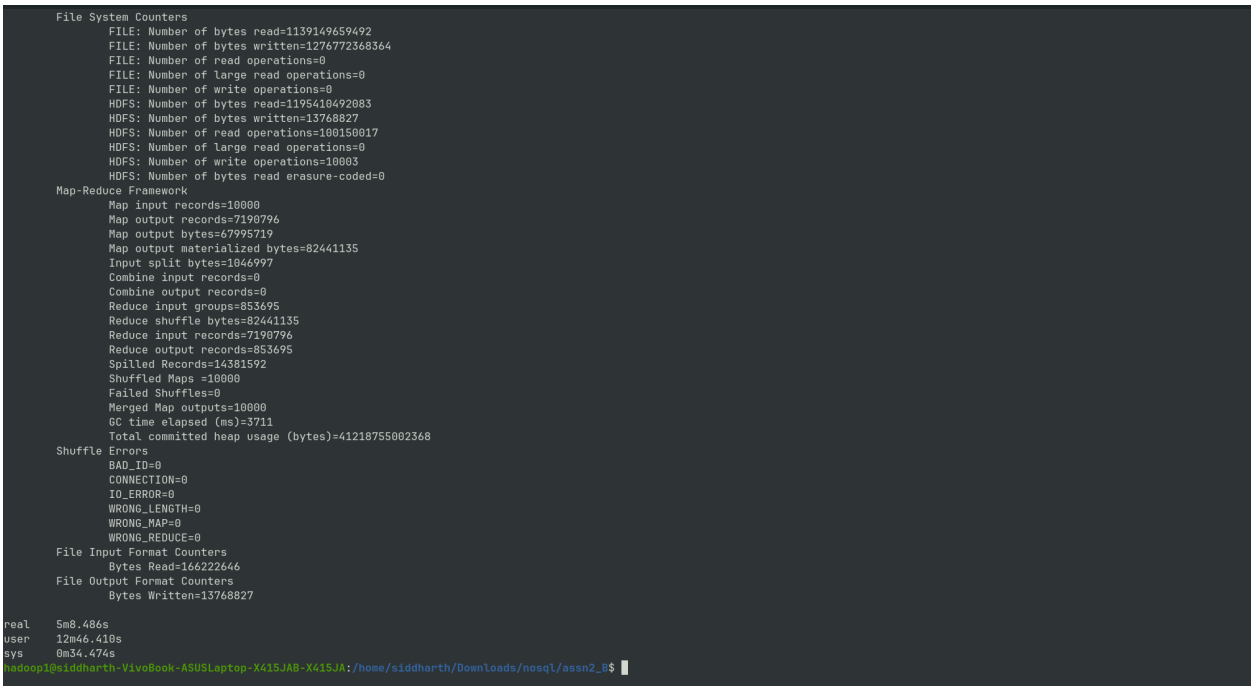

Figure 26: Run time for 5(a)

Reducer

- Aggregates term frequencies from all mappers.
- Retrieves the DF values for words from the cached file.
- Computes TF-IDF using the formula:

$$TFIDF = TF \times \log\left(\frac{TotalDocuments}{DF + 1}\right) \tag{1}$$

- Outputs `<filename, word-TFIDF>` pairs.

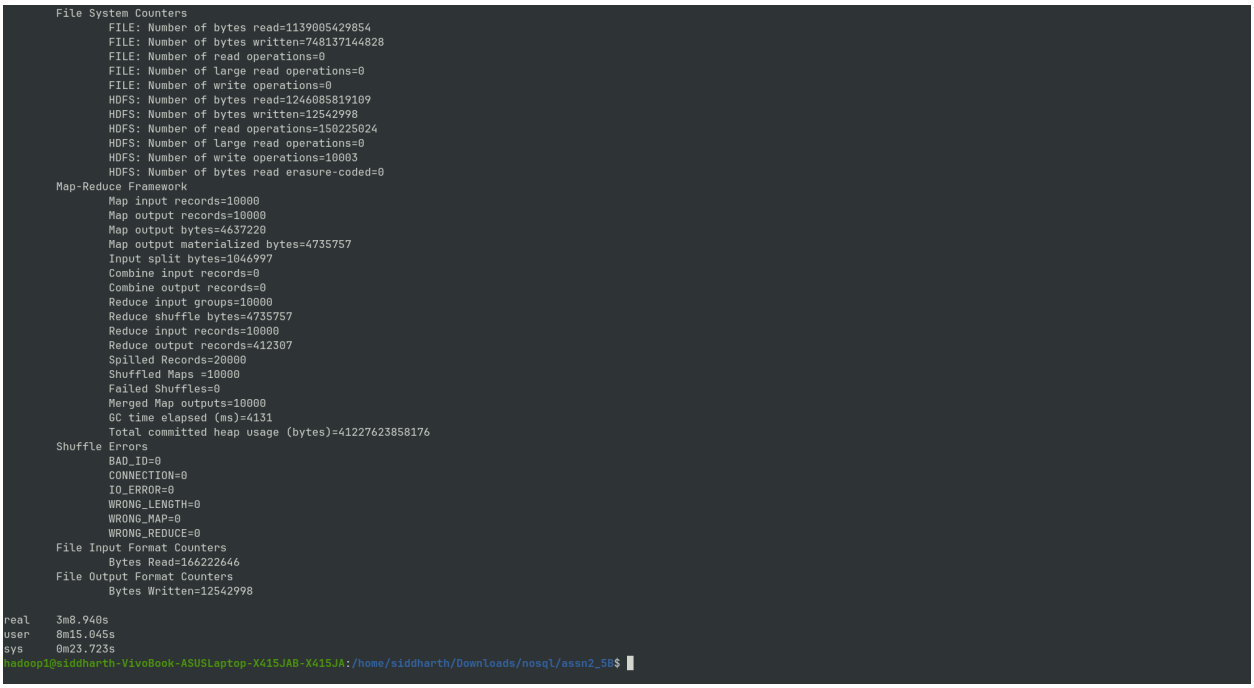

Figure 27: Run time for 5(b)