# Big Data Systems: Assignment1

Deadline 11 07, 2024 at 23:59 PM

*LEE CHIH-PIN NDHU CSIE*

## Problem 1

In Hadoop cluster, we will implement the sorting algorithm. The input is a list of string

### Pseudo code

```
import sys
for line in sys.stdin do
    line ← line.strip()                    ▷ Remove leading and trailing whitespace
    words ← line.split()                   ▷ Split line into words
    for word in words do
        print (word, 1)                    ▷ Output word with count 1, tab-separated
    end for
end for
```

Algorithm 1: Sort Mapper

```
import sys
word_count ← {}                            ▷ Initialize an empty dictionary for word counts
for line in sys.stdin do
    line ← line.strip()                    ▷ Remove leading and trailing whitespace
    parts ← line.split("\t", 1)            ▷ Split line by tab into two parts
    if len(parts) < 2 then
        continue                           ▷ Skip incomplete lines
    end if
    word, count ← parts                    ▷ Extract word and count
    try:
    count ← int(count)                     ▷ Convert count to integer
    except ValueError:
    continue                               ▷ Skip lines with invalid counts
    if word in word_count then
        word_count[word] ← word_count[word] + count
    else
        word_count[word] ← count           ▷ Add word with initial count
    end if
end for
for word in sorted(word_count) do
    print (word, word_count[word])         ▷ Output word and its total count, tab-separated
end for
```

Algorithm 2: Sort Reducer

### Explanation

In this section, we will explain the pseudo code of the sorting algorithm using MapReduce.

The algorithm consists of two main components: the Mapper and the Reducer. The Mapper reads the input data and emits key-value pairs, while the Reducer aggregates the key-value pairs and produces the final output.

Problem 1 continued on next page. . .

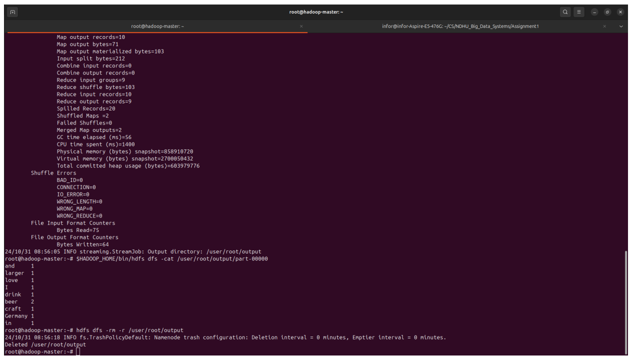
The Mapper reads the input data line by line and splits each line into words. For each word, the Mapper outputs a key-value pair where the key is the word and the value is 1. This allows the Reducer to count the occurrences of each word.

The Reducer reads the key-value pairs emitted by the Mapper and aggregates the counts for each word. It uses a dictionary to store the word counts and updates the count for each word as it processes the input. Finally, the Reducer sorts the words in lexicographical order and outputs the word and its total count.

The sorting algorithm using MapReduce is an example of the MapReduce programming model, which is commonly used for processing large datasets in parallel. By dividing the input data into smaller chunks and processing them in parallel, MapReduce allows for efficient processing of large datasets on distributed systems.

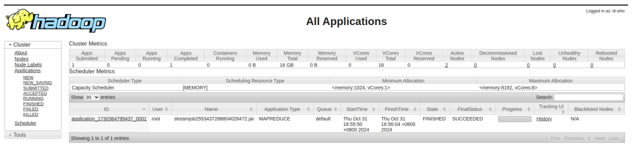
## Result

The result of sorting in terminal and Hadoop website is shown in Figure 1 and Figure 2 respectively.


root@hadoop-master: ~
root@hadoop-master: ~
infor@infor-Aspire-E5-476G: ~/CS/NDHU_Big_Data_Systems/Assignment1

```
                Map output records=10
                Map output bytes=71
                Map output materialized bytes=103
                Input split bytes=212
                Combine input records=0
                Combine output records=0
                Reduce input groups=9
                Reduce shuffle bytes=103
                Reduce input records=10
                Reduce output records=9
                Spilled Records=20
                Shuffled Maps =2
                Failed Shuffles=0
                Merged Map outputs=2
                GC time elapsed (ms)=56
                CPU time spent (ms)=1400
                Physical memory (bytes) snapshot=858910720
                Virtual memory (bytes) snapshot=2700050432
                Total committed heap usage (bytes)=603979776
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=75
        File Output Format Counters
                Bytes Written=64
24/10/31 08:56:05 INFO streaming.StreamJob: Output directory: /user/root/output
root@hadoop-master:~# $HADOOP_HOME/bin/hdfs dfs -cat /user/root/output/part-00000
and     1
larger  1
love    1
I       1
drink   1
beer    2
craft   1
Germany 1
in      1
root@hadoop-master:~# hdfs dfs -rm -r /user/root/output
24/10/31 08:56:18 INFO fs.TrashPolicyDefault: Namenode trash configuration: Deletion interval = 0 minutes, Emptier interval = 0 minutes.
Deleted /user/root/output
root@hadoop-master:~# 
```

Figure 1: The result of sorting in terminal



All Applications

Cluster Metrics

| Apps Submitted | Apps Pending | Apps Running | Apps Completed | Containers Running | Memory Used | Memory Total | Memory Reserved | VCores Used | VCores Total | VCores Reserved | Active Nodes | Decommissioned Nodes | Lost Nodes | Unhealthy Nodes | Rebooted Nodes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 B | 16 GB | 0 B | 0 | 16 | 0 | 2 | 0 | 0 | 0 | 0 |

Scheduler Metrics

| Scheduler Type | Scheduling Resource Type | Minimum Allocation | Maximum Allocation |
|---|---|---|---|
| Capacity Scheduler | [MEMORY] | <memory:1024, vCores:1> | <memory:8192, vCores:8> |

Show 20 entries

| ID | User | Name | Application Type | Queue | StartTime | FinishTime | State | FinalStatus | Progress | Tracking UI | Blacklisted Nodes |
|---|---|---|---|---|---|---|---|---|---|---|---|
| application_1730364799437_0001 | root | streamjob2553437288804028472.jar | MAPREDUCE | default | Thu Oct 31 16:55:50 +0800 2024 | Thu Oct 31 16:56:04 +0800 2024 | FINISHED | SUCCEEDED | | History | N/A |

Showing 1 to 1 of 1 entries

Figure 2: The result of sorting is successful status in Website

## Problem 2

Now, we will implement the Searching algorithms. The input is a list of string and the output is the word that we want to search.

### Pseudo code

```
import sys, os
search_word ← os.getenv('SEARCH_WORD', 'default_word')   ▷ Get search word from environment variable
for line in sys.stdin do
    filepath ← os.environ.get('map_input_file', 'unknown_file')
    filename ← os.path.basename(filepath)   ▷ Get filename from file path
    line ← line.strip()   ▷ Remove leading and trailing whitespace
    line_num ← 0   ▷ Initialize line number (for demonstration purposes)
    positions ← [ ]   ▷ Initialize empty list to store positions of search word
    start ← 0
    while True do
        start ← line.find(search_word, start)   ▷ Find position of search word
        if start = -1 then
            break   ▷ Exit loop if word not found
        end if
        positions.append(start)   ▷ Add position to list
        start ← start + len(search_word)   ▷ Move start to next position after found word
    end while
    if positions is not empty then
        print (filename, "_line_", line_num, line, positions)   ▷ Output filename, line number, line content, and positions of word
    end if
end for
```

Algorithm 3: Search Mapper

### Explanation

In this section, we will explain the pseudo code of the searching algorithm using MapReduce.

The algorithm consists of two main components: the Mapper and the Reducer. The Mapper reads the input data and emits key-value pairs, while the Reducer aggregates the key-value pairs and produces the final output.

The Mapper reads the input data line by line and searches for the search word in each line. If the search word is found, the Mapper outputs a key-value pair where the key is the filename and the value is a tuple containing the line content and the positions of the search word in the line.

The Reducer reads the key-value pairs emitted by the Mapper and aggregates the matches by filename. It uses a dictionary to store the matches for each file and updates the matches as it processes the input. Finally, the Reducer outputs the filename and all matches for that file.

Problem 2 continued on next page...

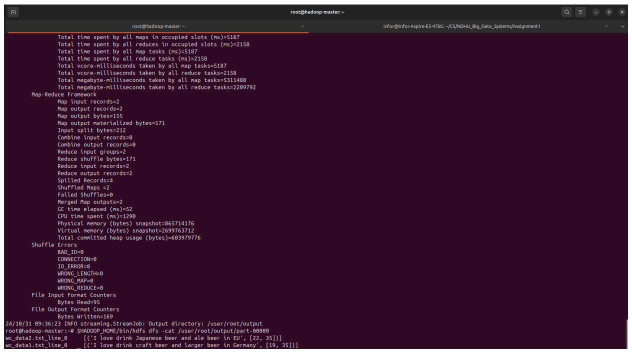
```
import sys
from collections import defaultdict
import ast
file_matches ← defaultdict(list)    ▷ Initialize a dictionary to store matches by filename
for line in sys.stdin do
    filename, content, positions ← line.strip().split("\t", 2)    ▷ Split input line into filename, content, and positions
    positions ← ast.literal_eval(positions)    ▷ Convert positions from string to list
    file_matches[filename].append((content, positions))    ▷ Add content and positions to the file's entry
end for
for current_file, matches in file_matches.items() do
    print (current_file, matches)    ▷ Output filename and all matches
end for
```

Algorithm 4: Search Reducer

The searching algorithm using MapReduce is an example of the MapReduce programming model, which is commonly used for processing large datasets in parallel. By dividing the input data into smaller chunks and processing them in parallel, MapReduce allows for efficient processing of large datasets on distributed systems.

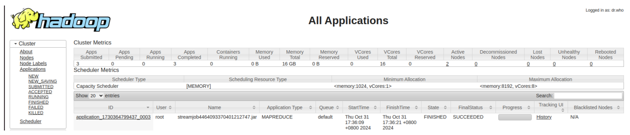
## Result

The result of searching in terminal and Hadoop website is shown in Figure 3 and Figure 4 respectively.


root@hadoop-master: ~
root@hadoop-master: ~
infor@infor-Aspire-E5-476G: ~/CS/NDHU_Big_Data_Systems/Assignment1

```
                Total time spent by all maps in occupied slots (ms)=5187
                Total time spent by all reduces in occupied slots (ms)=2158
                Total time spent by all map tasks (ms)=5187
                Total time spent by all reduce tasks (ms)=2158
                Total vcore-milliseconds taken by all map tasks=5187
                Total vcore-milliseconds taken by all reduce tasks=2158
                Total megabyte-milliseconds taken by all map tasks=5311488
                Total megabyte-milliseconds taken by all reduce tasks=2209792
        Map-Reduce Framework
                Map input records=2
                Map output records=2
                Map output bytes=155
                Map output materialized bytes=171
                Input split bytes=212
                Combine input records=0
                Combine output records=0
                Reduce input groups=2
                Reduce shuffle bytes=171
                Reduce input records=2
                Reduce output records=2
                Spilled Records=4
                Shuffled Maps =2
                Failed Shuffles=0
                Merged Map outputs=2
                GC time elapsed (ms)=52
                CPU time spent (ms)=1290
                Physical memory (bytes) snapshot=865714176
                Virtual memory (bytes) snapshot=2699763712
                Total committed heap usage (bytes)=603979776
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=95
        File Output Format Counters
                Bytes Written=169
24/10/31 09:36:23 INFO streaming.StreamJob: Output directory: /user/root/output
root@hadoop-master:~# $HADOOP_HOME/bin/hdfs dfs -cat /user/root/output/part-00000
wc_data2.txt_line_0     [('I love drink Japanese beer and ale beer in EU', [22, 35])]
wc_data1.txt_line_0     [('I love drink craft beer and larger beer in Germany', [19, 35])]
```

Figure 3: The result of search in terminal



All Applications

Cluster Metrics

| Apps Submitted | Apps Pending | Apps Running | Apps Completed | Containers Running | Memory Used | Memory Total | Memory Reserved | VCores Used | VCores Total | VCores Reserved | Active Nodes | Decommissioned Nodes | Lost Nodes | Unhealthy Nodes | Rebooted Nodes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 0 | 0 | 3 | 0 | 0 B | 16 GB | 0 B | 0 | 16 | 0 | 2 | 0 | 0 | 0 | 0 |

Scheduler Metrics

| Scheduler Type | Scheduling Resource Type | Minimum Allocation | Maximum Allocation |
|---|---|---|---|
| Capacity Scheduler | [MEMORY] | <memory:1024, vCores:1> | <memory:8192, vCores:8> |

Show 20 entries  Search:

| ID | User | Name | Application Type | Queue | StartTime | FinishTime | State | FinalStatus | Progress | Tracking UI | Blacklisted Nodes |
|---|---|---|---|---|---|---|---|---|---|---|---|
| application_1730364799437_0003 | root | streamjob4464093370401212747.jar | MAPREDUCE | default | Thu Oct 31 17:36:09 +0800 2024 | Thu Oct 31 17:36:21 +0800 2024 | FINISHED | SUCCEEDED | | History | N/A |

Figure 4: The result of search is successful in website

## Problem 3

Futhermore, we will implement the TF-IDF computation algorithm. The input is a list of string and the output is the TF-IDF value of each word.

### Definition

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure used to evaluate the importance of a word in a document relative to a collection of documents. It is commonly used in information retrieval and text mining to identify the most relevant words in a document.

The TF-IDF value of a word is calculated based on two components: Term Frequency (TF) and Inverse Document Frequency (IDF). The TF component measures the frequency of a word in a document, while the IDF component measures the rarity of the word in a collection of documents.

The TF-IDF algorithm computes the TF-IDF value of each word in a document by combining the TF and IDF components. The TF component is calculated as the frequency of the word in the document divided by the total number of words in the document. The IDF component is calculated as the logarithm of the total number of documents divided by the number of documents containing the word.

1. **Input:** Corpus of documents $D = \{d_1, d_2, \ldots, d_n\}$ and terms $T = \{t_1, t_2, \ldots, t_m\}$
2. **Output:** TF-IDF score for each term in each document.
3. **For each document** $d \in D$:
   (a) **For each term** $t \in T$:
      i. Calculate Term Frequency (TF) of $t$ in $d$:
      $$TF(t,d) = \frac{\text{Number of times } t \text{ appears in } d}{\text{Total number of terms in } d}$$
      ii. Calculate Document Frequency (DF) of $t$ in $D$:
      $$DF(t) = \text{Number of documents containing } t$$
      iii. Calculate Inverse Document Frequency (IDF) of $t$:
      $$IDF(t,D) = \log\left(\frac{|D|}{DF(t)}\right)$$
      iv. Calculate TF-IDF score for $t$ in $d$:
      $$TF\text{-}IDF(t,d,D) = TF(t,d) \times IDF(t,D)$$
4. **End For**

### Pseudo code

### Explanation

In this section, we will explain the pseudo code of the TF-IDF computation algorithm using MapReduce.

Problem 3 continued on next page...

```
import sys, os, re
from collections import defaultdict
file_name ← os.getenv('mapreduce_map_input_file', 'unknown_file').split('/')[-1]    ▷ Get filename from environment variable
word_counts ← defaultdict(int)    ▷ Initialize a dictionary for word counts
total_words ← 0
for line in sys.stdin do
    line ← line.strip().lower()    ▷ Remove leading/trailing whitespace and convert to lowercase
    words ← re.findall(r'\b\w+\b', line)    ▷ Extract words using regular expressions
    for word in words do
        word_counts[word] ← word_counts[word] + 1
        total_words ← total_words + 1    ▷ Increment total word count
    end for
end for
for word, count in word_counts.items() do
    tf ← count / total_words    ▷ Calculate term frequency
    print (word, file_name, tf)    ▷ Output word, filename, and term frequency
end for
```

Algorithm 5: TF-IDF Mapper

```
import sys, math
from collections import defaultdict
tf_values ← defaultdict(list)    ▷ Dictionary to store term frequencies by word
df_counts ← defaultdict(int)    ▷ Dictionary to store document frequency counts for each word
documents ← set()    ▷ Set to track unique document names
for line in sys.stdin do
    word, file_name, tf ← line.strip().split("\t")
    tf ← float(tf)    ▷ Convert tf to a floating-point number
    tf_values[word].append((file_name, tf))    ▷ Store tf value for each word and document
    df_counts[word] ← df_counts[word] + 1    ▷ Increment document frequency count for word
    documents.add(file_name)    ▷ Add document to the set
end for
total_docs ← len(documents)    ▷ Calculate the total number of documents
print("DEBUG: Total documents =", total_docs)
for word, file_tfs in tf_values.items() do
    print("DEBUG: DF count for word '", word, "':", df_counts[word])
    idf ← math.log(total_docs / df_counts[word]) if df_counts[word] > 0 else 0    ▷ Calculate inverse document frequency
    for file_name, tf in file_tfs do
        tf_idf ← tf * idf    ▷ Calculate TF-IDF
        print(word, file_name, ":.6f".format(tf_idf))    ▷ Output word, filename, and TF-IDF score
    end for
end for
```

Algorithm 6: TF-IDF Reducer

Problem 3 continued on next page. . .

The algorithm consists of two main components: the Mapper and the Reducer. The Mapper reads the input data and emits key-value pairs, while the Reducer aggregates the key-value pairs and produces the final output.

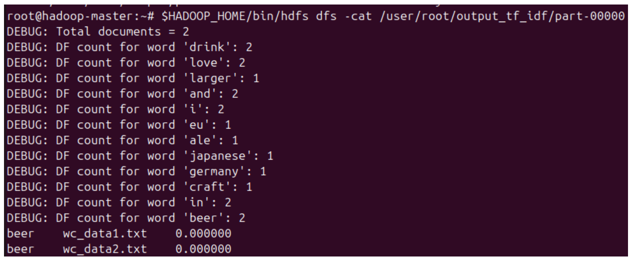
The Mapper reads the input data line by line and extracts words using regular expressions. For each word, the Mapper calculates the term frequency (TF) by counting the number of occurrences of the word in the document and dividing by the total number of words in the document. The Mapper outputs a key-value pair where the key is the word, the value is the filename, and the term frequency.

The Reducer reads the key-value pairs emitted by the Mapper and aggregates the TF values by word. It also calculates the document frequency (DF) for each word by counting the number of documents containing the word. The Reducer then calculates the inverse document frequency (IDF) for each word and combines it with the TF values to compute the TF-IDF score. Finally, the Reducer outputs the word, filename, and TF-IDF score for each word in each document.

The TF-IDF computation algorithm using MapReduce is an example of the MapReduce programming model, which is commonly used for processing large datasets in parallel. By dividing the input data into smaller chunks and processing them in parallel, MapReduce allows for efficient processing of large datasets on distributed systems.

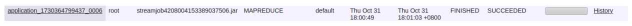
## Result

The result of TF-IDF in terminal and Hadoop website is shown in Figure 5 and Figure 6 respectively.



```
root@hadoop-master:~# $HADOOP_HOME/bin/hdfs dfs -cat /user/root/output_tf_idf/part-00000
DEBUG: Total documents = 2
DEBUG: DF count for word 'drink': 2
DEBUG: DF count for word 'love': 2
DEBUG: DF count for word 'larger': 1
DEBUG: DF count for word 'and': 2
DEBUG: DF count for word 'i': 2
DEBUG: DF count for word 'eu': 1
DEBUG: DF count for word 'ale': 1
DEBUG: DF count for word 'japanese': 1
DEBUG: DF count for word 'germany': 1
DEBUG: DF count for word 'craft': 1
DEBUG: DF count for word 'in': 2
DEBUG: DF count for word 'beer': 2
beer    wc_data1.txt    0.000000
beer    wc_data2.txt    0.000000
```

Figure 5: The result of TF IDF in terminal



```
application_1730364799437_0006  root  streamjob4208004153389037506.jar  MAPREDUCE  default  Thu Oct 31  Thu Oct 31         FINISHED  SUCCEEDED  History
                                                                                           18:00:49    18:01:03 +0800
```

Figure 6: The result of TF IDF is successful in website

## Problem 4

Finally, we will implement the Activity Mining algorithm. The input is a list of string and the output is the activity of each user.

### Definition

1. **Input:** Set of activity sequences $S = \{s_1, s_2, \ldots, s_n\}$, minimum support threshold $\theta$.
2. **Output:** Frequent activity patterns.
3. **Step 1: Preprocessing**
   (a) **For each sequence** $s \in S$:
      i. Tokenize each sequence $s$ into individual activities $A = \{a_1, a_2, \ldots, a_k\}$.
      ii. Store each unique activity in a list of candidate patterns $C$.
4. **Step 2: Pattern Generation and Counting**
   (a) Initialize an empty set $F$ for storing frequent patterns.
   (b) **For each candidate pattern** $c \in C$:
      i. **For each sequence** $s \in S$:
         A. **If** $c$ is found as a subsequence in $s$ **then**:
            - Increment the count of $c$.
      ii. **If** count of $c \geq \theta$:
         - Add $c$ to frequent patterns set $F$.
5. **Step 3: Pattern Expansion**
   (a) **For each pattern** $f \in F$:
      i. Generate expanded patterns by adding one additional activity to $f$.
      ii. Check support of each expanded pattern and retain patterns meeting $\theta$.
6. **Step 4: Output Results**
   (a) Return the set of frequent patterns $F$ as identified activity patterns.

### Pseudo code

## Explanation

In this section, we will explain the pseudo code of the Activity Mining algorithm using MapReduce.

The algorithm consists of two main components: the Mapper and the Reducer. The Mapper reads the input data and emits key-value pairs, while the Reducer aggregates the key-value pairs and produces the final output.

The Mapper reads the input data line by line and generates all possible subsets of activities of varying lengths. For each subset, the Mapper outputs a key-value pair where the key is the subset of activities and the value is 1. This allows the Reducer to count the occurrences of each subset.

Problem 4 continued on next page. . .

```
import sys
from itertools import combinations
MIN_LENGTH ← 2                                                    ▷ Minimum length for subsets
for line in sys.stdin do
    line ← line.strip().lower()          ▷ Remove leading/trailing whitespace and convert to lowercase
    words ← line.split()                                                  ▷ Split line into words
    for length in range(MIN_LENGTH, len(words) + 1) do
        for subset in combinations(words, length) do
            print("-¿".join(subset), 1)              ▷ Output the subset as a joined string with count 1
        end for
    end for
end for
```

Algorithm 7: Activity Mining Mapper

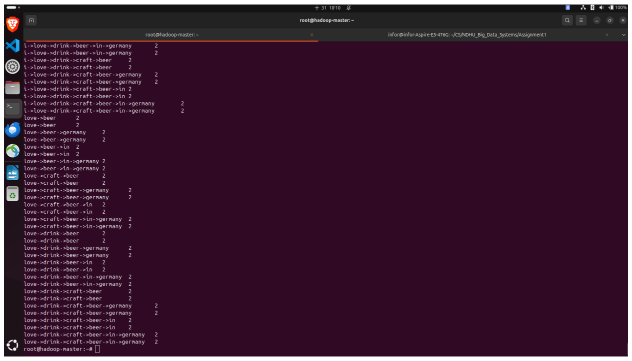
```
import sys
SUPPORT_THRESHOLD ← 2                                              ▷ Define the support threshold
current_combination ← None
current_count ← 0
for line in sys.stdin do
    line ← line.strip()
    if not line then
        continue                                                               ▷ Skip empty lines
    end if
    combination, count ← line.split("\t")
    count ← int(count)                                                ▷ Convert count to an integer
    if combination = current_combination then
        current_count ← current_count + count          ▷ Accumulate count for the same combination
    else
        if current_combination and current_count ≥ SUPPORT_THRESHOLD then
            print(current_combination, current_count)        ▷ Output if count meets support threshold
        end if
        current_combination ← combination
        current_count ← count
    end if
end for
if current_combination and current_count ≥ SUPPORT_THRESHOLD then
    print(current_combination, current_count)                  ▷ Final output if count meets threshold
end if
```

Algorithm 8: Activity Mining Reducer

The Reducer reads the key-value pairs emitted by the Mapper and aggregates the counts for each subset. It accumulates the counts for the same subset and outputs the subset and its count if the count meets the support threshold. The support threshold is a predefined value that determines the minimum number of occurrences required for a subset to be considered frequent.

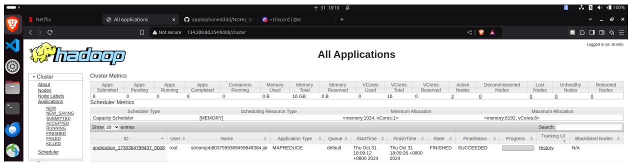
## Result

The result of Activity Mining in terminal and Hadoop website is shown in Figure 7 and Figure 8 respectively.



```
i->love->drink->beer->in->germany       2
i->love->drink->beer->in->germany       2
i->love->drink->craft->beer     2
i->love->drink->craft->beer     2
i->love->drink->craft->beer->germany    2
i->love->drink->craft->beer->germany    2
i->love->drink->craft->beer->in 2
i->love->drink->craft->beer->in 2
i->love->drink->craft->beer->in->germany        2
i->love->drink->craft->beer->in->germany        2
love->beer      2
love->beer      2
love->beer->germany     2
love->beer->germany     2
love->beer->in  2
love->beer->in  2
love->beer->in->germany 2
love->beer->in->germany 2
love->craft->beer       2
love->craft->beer       2
love->craft->beer->germany      2
love->craft->beer->germany      2
love->craft->beer->in   2
love->craft->beer->in   2
love->craft->beer->in->germany  2
love->craft->beer->in->germany  2
love->drink->beer       2
love->drink->beer       2
love->drink->beer->germany      2
love->drink->beer->germany      2
love->drink->beer->in   2
love->drink->beer->in   2
love->drink->beer->in->germany  2
love->drink->beer->in->germany  2
love->drink->craft->beer        2
love->drink->craft->beer        2
love->drink->craft->beer->germany       2
love->drink->craft->beer->germany       2
love->drink->craft->beer->in    2
love->drink->craft->beer->in    2
love->drink->craft->beer->in->germany   2
love->drink->craft->beer->in->germany   2
root@hadoop-master:~# 
```

Figure 7: The result of Activity Mining in terminal



All Applications

Logged in as: dr.who

Cluster Metrics

| Apps Submitted | Apps Pending | Apps Running | Apps Completed | Containers Running | Memory Used | Memory Total | Memory Reserved | VCores Used | VCores Total | VCores Reserved | Active Nodes | Decommissioned Nodes | Lost Nodes | Unhealthy Nodes | Rebooted Nodes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 0 | 0 | 8 | 0 | 0 B | 16 GB | 0 B | 0 | 16 | 0 | 2 | 0 | 0 | 0 | 0 |

Scheduler Metrics

| Scheduler Type | Scheduling Resource Type | Minimum Allocation | Maximum Allocation |
|---|---|---|---|
| Capacity Scheduler | [MEMORY] | <memory:1024, vCores:1> | <memory:8192, vCores:8> |

Show 20 entries Search:

| ID | User | Name | Application Type | Queue | StartTime | FinishTime | State | FinalStatus | Progress | Tracking UI | Blacklisted Nodes |
|---|---|---|---|---|---|---|---|---|---|---|---|
| application_1730364799437_0008 | root | streamjob8037559369459848384.jar | MAPREDUCE | default | Thu Oct 31 18:09:12 +0800 2024 | Thu Oct 31 18:09:26 +0800 2024 | FINISHED | SUCCEEDED | | History | N/A |

Figure 8: The result of Activity Mining is successful in website