Team members:

|  |  |
| --- | --- |
| **UBIT Name** | **Person Number** |
| Ankushad | 50291981 |
| Sandeshk | 50325421 |
| Shivalee | 50316979 |

Task 1: Word count

Mapper: Takes the text files as input and the output is the key-value pair where key is the word and the value are 1.

Logic: The text is read line by line from the standard input. In the preprocessing step, the text is type casted to lower case and all punctuation marks are removed. We have also listed a few stop words manually. Then each word in the line is considered, and checked if it’s a letter and number and a stop word, is yes, then the word is converted into a <key, value> = <word, 1> and printed on the terminal.

Reducer: Takes in the key-value pairs from the mapper in the sorted order as the input and the output is the aggregated counts of each word.

Logic: The input is read line by line and the key value pair is extracted from the text. Each word is added to a dictionary and if the same word appears, the count is incremented. Once all the words are counted, each word in the dictionary and its corresponding counts are printed as <key, value> = <word, count>.

Task 2: N-grams

To determine the 10 most occurred modified tri-gram for the given key words, we use 2 sets of map-reduce, one after the other. The first set consists of multiple reducers which results in that many output files each containing 10 tri-grams. The second set acts as global reducer, the outputs from the first reducers are aggregated and the overall result is obtained. The key words are “science”, “sea” and “fire”.

Set 1

Mapper: Takes the text files as input and the output is the key-value pair where key is the modified tri-gram and the value are 1.

Logic: The text is read line by line from the standard input. In the preprocessing step, the text is type casted to lower case and all punctuation marks are removed. To create a tri-gram, 3 words are maintained: previous, current and next corresponding to the trigram previous\_current\_next. From the nltk library, WordNetLemmatizer is used to determine the lemma of a word.

For each word in the line, corresponding lemma is considered. If previous and the next word in the line exists, then we check if the previous word is one of the keywords, if yes then emit trigram as $\_current\_next with a count value of 1. Similar check is done for current and next words and corresponding emits previous\_$\_next and previous\_current\_$ respectively with a count value of 1. In case the current word is the last word of the line, then the last 2 words of the line are retained to check if along with the first word of next sentence, we can get a trigram containing any keyword. Same checks are above are carried out and the emits are made for the key words.

Reducer: Takes in the key-value pairs from the mapper in the sorted order as the input and the output is the aggregated counts of 10 most occurred modified tri-gram locally.

Logic: The input is read line by line and the key value pair is extracted from the text. Each tri-gram is added to a dictionary and if the same tri-gram appears, the count is incremented. Once all the tri-grams are counted, the dictionary is sorted based on the value of counts in decreasing order. The 10 highest counts and their corresponding tri-grams are printed as <key, value> = < tri-gram, count>.

Set 2

Mapper: Takes the output of all the reducers and outputs the key value pairs to the global reducer.

Logic: The text is read line by line from the standard input. Each line consists of a key value pair. The text is processed by removing redundant spaces and the output is the passed onto the global reducer.

Reducer: Takes in the key-value pairs from the mapper in the sorted order as the input and the output is the aggregated counts of 10 most occurred modified tri-gram locally.

Logic: The input is read line by line and the key value pair is extracted from the text. Each tri-gram is added to a dictionary and if the same tri-gram appears, the count is incremented. Once all the tri-grams are counted, the dictionary is sorted based on the value of counts in decreasing order. The 10 highest counts and their corresponding tri-grams are printed as <key, value> = < tri-gram, count>.

Task 3: Inverted Index

Mapper: Takes the text files as input and the output is the key-value pair where key is the word and the value is the filename.

Logic: We use *os.getenv('mapreduce\_map\_input\_file')* to get the file name with path and use split to extract the file name. The text is read line by line from the standard input. In the preprocessing step, the text is type casted to lower case and all punctuation marks are removed. We have also listed a few stop words manually. Then each word in the line is considered, and checked if it’s a letter and not a number or stop-word. After checking word is converted into a <key, value> = <word, fileName> and printed on the terminal.

Reducer: Takes in the key-value pairs from the mapper in the sorted order as the input and the output is the list of file names of each word.

Logic: The input is read line by line and the key value pair is extracted from the text. Each word is added to a dictionary and corresponding files are added in a set. If same word appears, corresponding file is added to the set. Since set contains only unique items so we get the unique file names. Once all the words are counted, each word in the dictionary and its corresponding set of file names are printed as <key, value> = <word, set of file names>.

Task 4: Relational join

Mapper: Takes the csv files as input and the output is the tab separated five-string consisting of employee, name, salary, country & passcode.

Logic: The text is read line by line from the standard input. For each line, we store the first line element in employee and check its value to ignore headers. If length of line is 4 then we know that this line is from join1.csv and if length is 2 then this line is from ‘join 2.csv’. We store the required data in corresponding strings. We have kept default values as -1 for remaining fields.

Reducer: Takes in the five-string from the mapper in the sorted order as the input and the output is the joined five-string using employee ID as the primary key.

Logic: We create 2 dictionaries for join.csv data and ‘join 2.csv’ data. The input is read line by line and the read values are stored in 5 variables. We check the value of passcode to decide if this is data from join1 or join 2 as passcode will be -1 for join1 data. We store the data in corresponding dictionary using employee as the key. Once all the input is processed, we traverse through the two dictionaries and output all the data for the corresponding key.