Big Data Systems: Assignment2

Deadline 11 28, 2024 at 23:59 PM

LEE CHIH-PIN NDHU CSIE

Problem 1

In Spark cluster, we will implement PageRank algorithm to calculate the page rank of each node in the graph.

Pseudo code

Input: Graph as a text file $PR\_data.txt$, Number of iterations N
Output: PageRank values for each node, saved to $output\_ranks.txt$

1: Initialize SparkContext with configuration
2: Load the data from $PR\_data.txt$
3: Parse the input data to create links: $links \leftarrow \{(node, list\_of\_neighbors)\}$
4: Initialize ranks: $ranks \leftarrow \{(node, 1.0)\}$ **for** $i = 1$ *to* N **do**
5:
end
Compute contributions for each node:

$$contributions \leftarrow \bigcup_{(node,neighbors)\in links} \{(neighbor, rank(node)/|neighbors|) \ \forall \ neighbor \in neighbors\}$$

6: Aggregate contributions and compute new ranks:

$$rank(node) \leftarrow 0.15 + 0.85 \cdot \sum contributions(node)$$

Algorithm 1: PageRank using PySpark

Explanation

The PageRank algorithm implemented in PySpark computes the rank of nodes in a graph based on their connectivity. Below is an explanation of each step in the algorithm:

- **Initialization:** A SparkContext is created to set up the distributed computing environment. The input graph data is read from a text file and parsed into a key-value structure, where each key represents a node and the value is a list of its neighbors. Initially, all nodes are assigned an equal rank of 1.0.
- **Iterative Computation:** The algorithm iteratively computes the PageRank values for all nodes over N iterations:
  - Each node distributes its current rank equally among its neighbors. The contribution to each neighbor is computed as:
    $$\text{contribution} = \frac{\text{rank of the node}}{\text{number of neighbors}}$$
  - These contributions are aggregated for each node to calculate its new rank using the formula:
    $$\text{new rank} = 0.15 + 0.85 \cdot \sum (\text{contributions from all neighbors})$$
    Here, 0.15 represents the damping factor $(1 - d)$, and 0.85 is the teleportation probability d.
- **Output:** After N iterations, the final PageRank values are collected and written to an output text file, where each line contains a node and its corresponding rank.
- **Termination:** The SparkContext is stopped to release resources.

Result

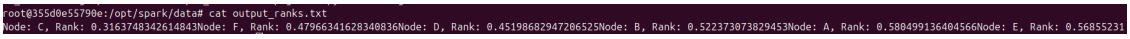


Figure 1: The result of Pagerank in terminal

Problem 2

Now we will compute the GPA of each student, the total number students of each course, the average grade among all students of each course and the failure rate of each course

Algorithm: GPA Computation Using PySpark

1: Initialize SparkContext and SparkSession.
2: Load the grades data from `/opt/spark/data/grades.txt`.
3: Load the academic credit data from `/opt/spark/data/academic_credit.txt`.
4: Parse course credit data:

$$\text{course\_credits} \leftarrow \text{credits.map(parse\_credits).filter(non\_null).collectAsMap()}$$

5: Define the GPA scoring function `score_to_gpa(score)` based on score ranges.
6: Define a function `compute_gpa(student_record)` to compute GPA for a student:

- Parse student grades into a list of course IDs and scores.
- For each course, compute weighted GPA using:

$$\text{total\_points} \leftarrow \text{GPA} \times \text{course\_credit}$$

- Compute the overall GPA:

$$\text{GPA} \leftarrow \frac{\text{total\_points}}{\text{total\_credits}}$$

7: Compute student GPAs:

$$\text{student\_gpas} \leftarrow \text{grades.map(compute\_gpa)}$$

8: Compute course statistics:

- Parse grades into course-wise GPAs:

$$\text{(course, GPA)} \leftarrow \text{grades.flatMap(parse\_grades)}$$

- Reduce to compute mean GPA and count of students per course:

$$\text{course\_stats} \leftarrow \text{reduceByKey(sum\_and\_count).mapValues(mean\_GPA)}$$

9: Compute failure rates for each course:

- Parse grades to identify failing scores:

$$\text{(course, fail\_count)} \leftarrow \text{grades.flatMap(failure\_flags)}$$

- Compute failure rates as:

$$\text{failure\_rate} \leftarrow \frac{\text{fail\_count}}{\text{total\_students}}$$

10: Stop SparkContext.

Algorithm 2: GPA Computation Using PySpark

Explanation

The algorithm computes the GPA of each student, the total number of students in each course, the average grade among all students of each course, and the failure rate of each course. Below is an explanation of each step in the algorithm:

- **Initialization:** A SparkContext and SparkSession are created to set up the distributed computing environment. The grades data and academic credit data are loaded from text files.
- **Parsing Course Credits:** The academic credit data is parsed to create a mapping of course IDs to their respective credits. This information is used to compute the weighted GPA for each course.
- **GPA Scoring Function:** A scoring function is defined to map scores to GPA values based on predefined score ranges.
- **GPA Computation:** A function `compute_gpa` is defined to compute the GPA for each student. The function parses the student's grades, computes the weighted GPA for each course, and calculates the overall GPA based on the total credits.
- **Student GPAs:** The algorithm computes the GPA for each student by applying the `compute_gpa` function to the grades data.
- **Course Statistics:** The algorithm computes the mean GPA and the count of students for each course. It parses the grades data to extract course-wise GPAs, reduces the data to compute the mean GPA and count of students per course, and stores the results in `course_stats`.
- **Failure Rates:** The algorithm computes the failure rate for each course by identifying failing scores in the grades data. It calculates the failure rate as the ratio of the number of failing students to the total number of students in the course.
- **Termination:** The SparkContext is stopped to release resources.

Result

```
root@355d0e55790e:/opt/spark/data# cat /opt/spark/data/output_course_stats/part-00000
('GC01', (1.59, 20))
('GC08', (1.594, 18))
('GC04', (1.216, 19))
('GC09', (1.559, 17))
('GC10', (1.639, 18))
('GC06', (1.377, 26))
('GC03', (1.461, 18))
('GC02', (0.882, 17))
('GC05', (1.614, 21))
('GC07', (1.825, 20))
```

Figure 2: The result of course stats in terminal

```
root@355d0e55790e:/opt/spark/data# cat /opt/spark/data/output_failure_rates/part-00000
('GC01', 0.6)
('GC08', 0.611)
('GC04', 0.632)
('GC09', 0.529)
('GC10', 0.556)
('GC06', 0.654)
('GC03', 0.556)
('GC02', 0.765)
('GC05', 0.571)
('GC07', 0.55)
```

Figure 3: The result of failure rates in terminal

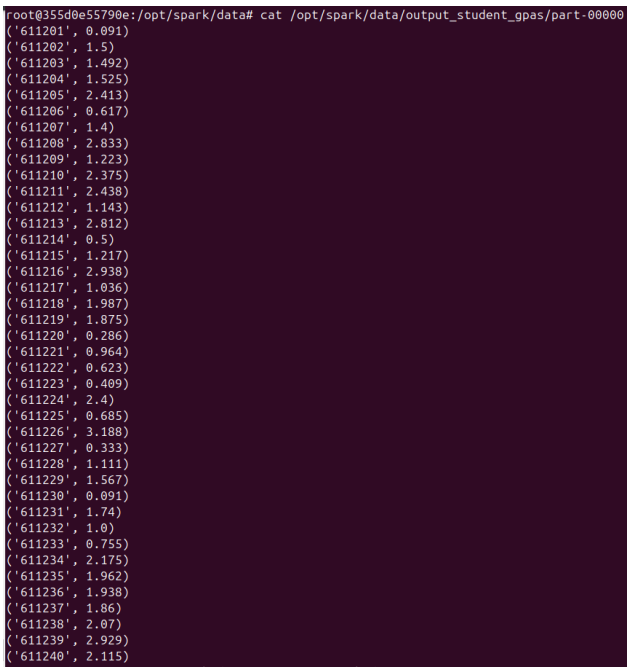


Figure 4: The result of student gpas in terminal

Problem 3

Write a Spark program to compute the inverted index, frequency and total counts of keywords on a set of documents

Algorithm: Inverted Index Using PySpark

1: Initialize SparkContext and SparkSession.
2: Load the text file containing the documents: `/opt/spark/data/article.txt`.
3: Define the function `parse_document(line)` to parse each document:

- Split the line into two parts: document ID and text.
- Use the Jieba library to tokenize the text into words.
- Return a tuple: $(\text{doc\_id}, \text{list\_of\_words})$.

4: Parse all documents:

$$\text{parsed\_documents} \leftarrow \text{documents.map(parse\_document)}$$

5: Define the function `word_count(doc)` to prepare word-level data:

- For each word in the document, emit:

$$(\text{word}, (\text{doc\_id}, 1))$$

6: Compute word counts:

$$\text{word\_counts} \leftarrow \text{parsed\_documents.flatMap(word\_count)}$$

7: Define the function `combine_counts(values)` to aggregate counts per document:

- Iterate through all $(\text{doc\_id}, \text{count})$ pairs for a word.
- Sum counts for each document.
- Return a list of $(\text{doc\_id}, \text{total\_count})$ pairs.

8: Compute the inverted index:

$$\text{inverted\_index} \leftarrow \text{word\_counts.groupByKey().mapValues(combine\_counts)}$$

9: Define the function `format_output(index)` to prepare output:

- For each word, compute the total occurrence across all documents:

$$\text{total} \leftarrow \sum (\text{counts per document})$$

- Return formatted output:

$$\text{"word: [(doc\_id1, count1), (doc\_id2, count2), ...], Total: total"}$$

10: Format the results:

$$\text{formatted\_results} \leftarrow \text{inverted\_index.map(format\_output)}$$

11: Stop SparkContext.

Algorithm 3: Inverted Index Using PySpark

Problem 3 continued on next page...

Explanation

The algorithm computes the inverted index, frequency, and total counts of keywords in a set of documents. Below is an explanation of each step in the algorithm:

- **Initialization:** A SparkContext and SparkSession are created to set up the distributed computing environment. The text file containing the documents is loaded.
- **Parsing Documents:** Each document is parsed to extract the document ID and tokenize the text into words using the Jieba library. The parsed documents are stored as tuples of document ID and a list of words.
- **Word Count:** The algorithm prepares word-level data by emitting a tuple for each word in the document. Each tuple contains the word and a pair $(\text{doc\_id}, 1)$ to represent the occurrence of the word in the document.
- **Word Counts:** The word counts are computed by applying the `word_count` function to the parsed documents. The result is a list of tuples containing the word and the document ID with a count of 1.
- **Combine Counts:** The algorithm aggregates counts per document by summing the counts for each document. The result is a list of tuples containing the document ID and the total count of the word in that document.
- **Inverted Index:** The inverted index is computed by grouping the word counts by the word and mapping the values to combine the counts per document. The result is a mapping of words to a list of tuples containing the document ID and the total count of the word in that document.
- **Format Output:** The algorithm formats the results by computing the total occurrence of each word across all documents. The output is formatted as a string containing the word, the list of document IDs with counts, and the total count of the word.
- **Termination:** The SparkContext is stopped to release resources.

Result

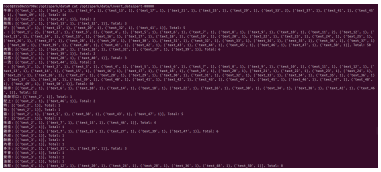


Figure 5: The result of Invert Index in terminal

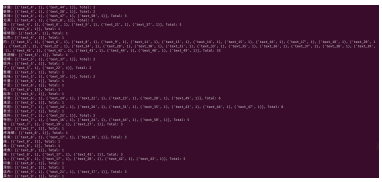


Figure 6: The result of Invert Index in terminal

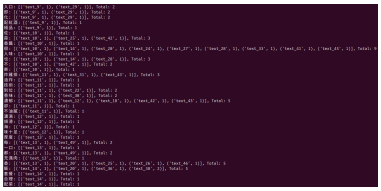


Figure 7: The result of Invert Index in terminal

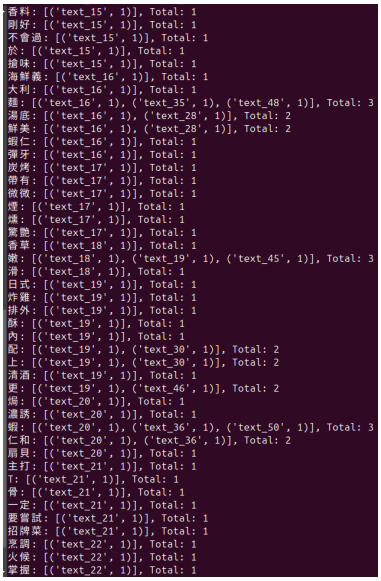


Figure 8: The result of Invert Index in terminal

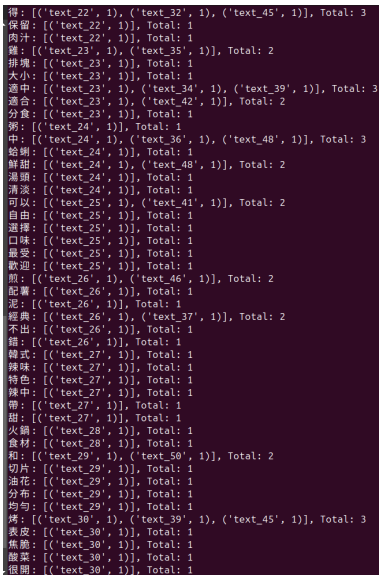
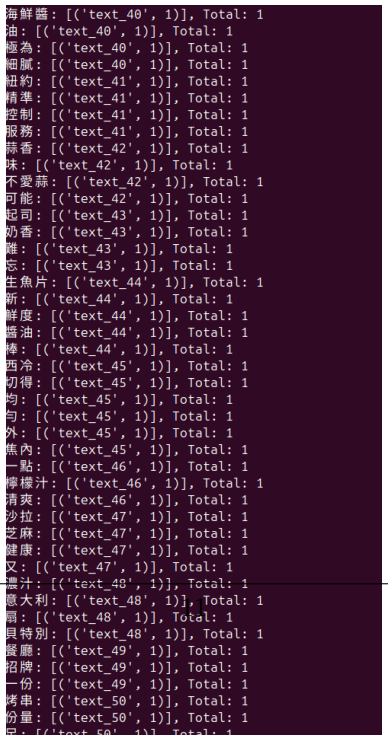


Figure 9: The result of Invert Index in terminal

Figure 10: The result of Invert Index in terminal



Problem 4

Write a Spark program to compute the frequent itemsets in a set of transactions

Algorithm: Frequent Itemsets Using PySpark

1: Initialize SparkContext and SparkSession.
2: Set the frequency threshold: `threshold = 10`.
3: Load the transaction data from `/opt/spark/data/purchase.txt`.
4: Parse the transaction data:
- Each transaction is represented as a set of items.
- Split each line of the file by spaces and convert to a set of items (ignoring the transaction ID).
- Result:

$$\text{parsed\_transactions} \leftarrow \text{transactions.map(parse\_line)}$$

5: Define the function `generate_combinations(transaction)` to create all possible combinations of items for a given transaction:
- For a transaction with n items, generate all subsets of size 1 to n.
- Return all subsets as sorted tuples to ensure consistency.

6: Generate all possible itemsets:

$$\text{itemsets} \leftarrow \text{parsed\_transactions.flatMap(generate\_combinations)}$$

7: Count the occurrences of each itemset:
- Emit key-value pairs: $(\text{itemset}, 1)$ for each itemset.
- Aggregate counts for each itemset using `reduceByKey`.

$$\text{itemset\_counts} \leftarrow \text{itemsets.map(key\_value).reduceByKey(sum\_counts)}$$

8: Filter the frequent itemsets:
- Keep only those itemsets whose count is greater than or equal to the threshold.

$$\text{frequent\_itemsets} \leftarrow \text{itemset\_counts.filter(counts ¿= threshold)}$$

9: Sort the frequent itemsets by their counts in descending order:

$$\text{sorted\_frequent\_itemsets} \leftarrow \text{frequent\_itemsets.sortBy(count, descending=True)}$$

10: Format the results for output:
- Convert each itemset to a formatted string:

$$\text{"Itemset: ¡items¿, Count: ¡count¿"}$$

11: Stop SparkContext.

Algorithm 4: Frequent Itemsets Using PySpark

Problem 4 continued on next page...

Explanation

The algorithm computes the frequent itemsets in a set of transactions. Below is an explanation of each step in the algorithm:

- **Initialization:** A SparkContext and SparkSession are created to set up the distributed computing environment. The frequency threshold is set to 10, and the transaction data is loaded from a text file.
- **Parsing Transactions:** Each transaction is parsed to extract the items. The transactions are split by spaces, and the transaction ID is ignored. The parsed transactions are stored as sets of items.
- **Generate Combinations:** The algorithm generates all possible combinations of items for a given transaction. It creates subsets of items of sizes 1 to n and returns them as sorted tuples to ensure consistency.
- **Generate Itemsets:** All possible itemsets are generated by applying the `generate_combinations` function to the parsed transactions. The result is a list of sorted tuples representing the itemsets.
- **Count Occurrences:** The algorithm counts the occurrences of each itemset by emitting key-value pairs $(\text{itemset}, 1)$ for each itemset and aggregating the counts using `reduceByKey`.
- **Filter Frequent Itemsets:** The frequent itemsets are filtered by keeping only those itemsets whose count is greater than or equal to the threshold.
- **Sort Frequent Itemsets:** The frequent itemsets are sorted by their counts in descending order to identify the most frequent itemsets.
- **Format Output:** The results are formatted for output by converting each itemset to a formatted string containing the items and their count.
- **Termination:** The SparkContext is stopped to release resources.

Result

```
root@355d0e55790e:/opt/spark/data# cat /opt/spark/data/sorted_frequent_data/part-00000
Itemset: C20, Count: 18
Itemset: C12, Count: 16
Itemset: C4, Count: 16
Itemset: C24, Count: 16
Itemset: C8, Count: 15
Itemset: C16, Count: 14
Itemset: C13, Count: 14
Itemset: C2, Count: 14
Itemset: C17, Count: 13
Itemset: C11, Count: 13
Itemset: C1, Count: 12
Itemset: C15, Count: 12
Itemset: C21, Count: 11
Itemset: C5, Count: 11
Itemset: C3, Count: 11
Itemset: C22, Count: 11
Itemset: C17, C20, Count: 10
Itemset: C23, Count: 10
Itemset: C14, Count: 10
Itemset: C25, Count: 10
```

Figure 12: The result of Sort Frequent in terminal