Code Documentation:

SDK: Amazon Coretto version 1.8, scala SDK 2.13.13

Build Tools: SBT

Project Structure:

- */src/main/scala/*: Contains the Scala source files for the application.
- *src/main/scala/createParquet*: Contains scripts to generate sample Parquet files (SampleA.scala and SampleB.scala).
- *src/main/scala/processing*: Contains the data processing objects (TopItemsProcessor.scala and TopItemsProcessorSkewed.scala).
- *Main.scala*: The main entry point of the application that orchestrates the data processing flow.
- */src/main/resources*: Holds all the Parquet files that the application reads from and writes to (sampleA.parquet, sampleB.parquet, and output_top_items.parquet).
- */src/test/scala/:* Contains test cases for the application, including integration tests (IntegrationTest.scala).

To run the application, you will execute the **Main** object within the Scala environment.

Code Documentation and workings:

In the TopItemsProcessor object, it processes datasets to identify the top N items by count within each geographical location.

1. Read Input Data

Detections Data: Reads the detections dataset from a Parquet file. Each row in the dataset contains a *geographical_location_oid* and an *item_name*. The data is transformed into an RDD of tuples (Long, String) using the .map() and row.getAs() methods, where each tuple represents a detection event with a location ID and item name.

Locations Data: Reads the locations dataset from a Parquet file. Similar to detections, each row has a geographical_location_oid and a geographical_location (name). The data is transformed into an RDD and then converted to a Map using collectAsMap(), mapping each location ID to its name.

- **Time Complexity**: O(N) for both detections and locations data, where N is the number of rows in each Parquet file. Reading data is linearly proportional to the size of the dataset.
- **Space Complexity**: O(N) for storing the data in memory as RDDs or Maps.

2. Broadcast Locations Map

The locations map is broadcasted to all nodes in the Spark cluster using *spark.sparkContext.broadcast(locationsRDD).* Broadcasting prepares and optimizes the join (broadcast hash join) process by ensuring that each node has a local copy of the locations dataset, reducing data shuffling across the network. I have broadcasted the locationsRDD instead of detectionsRDD because locationsRDD is much smaller, and it is used for lookup operations against large datasets like detectionsRDD.

- **Time Complexity**: O(L), where L is the size of the locations dataset. Broadcasting involves distributing the data across all nodes in the cluster.
- **Space Complexity**: O(L) on each node, but since the locations dataset is assumed to be significantly smaller than detections, it's generally manageable.

3. Perform Broadcast Hash Join

A broadcast hash join is performed between the detectionsRDD and the broadcasted locations map. For each detection event, it tries to find a matching location ID from the detectionsRDD with the one in the broadcasted map. If a match is found, it constructs a new tuple with the format ((location, itemName), 1), where location is the geographical location name from the map, itemName is the item name, and 1 represents a single occurrence. Using "1" helps for counting the occurrences in the future. I did not explicitly use .join() operation, but rather, use .get() to compare and match the location ID from both RDDs.

- **Time Complexity**: O(D), where D is the number of detection events.
- **Space Complexity**: O(D) for the resulting RDD after the join.

4. Count Occurrences

The occurrences of each (location, itemName) pair are counted using reduceByKey(_ + _), which aggregates the counts for each unique key (location and item name combination) across the dataset.

- **Time Complexity**: O(D) for the reduce operation.
- **Space Complexity**: O(U) where U is the number of unique **(location, itemName)** pairs.

5. Prepare for Ranking

The structure of the data is transformed to prepare for ranking. It changes from ((location, itemName), count) to (location, (itemName, count)), making location the key for subsequent operations.

- **Time Complexity**: O(D), as it involves mapping over each element of the RDD.
- **Space Complexity**: Same as before, O(U), since the transformation doesn't change the amount of data.

6. Group by Location and Rank Items

The data is grouped by location using groupByKey(), and within each group, items are sorted by their counts in descending order. The top N items (topX) are selected from each group, and a rank (starting from 1) is assigned based on their position in the sorted list.

- **Time Complexity**: O(U log T) for each group, where T is the size of items per location, due to sorting. The total time complexity can vary based on the distribution of items across locations.
- **Space Complexity**: O(V), where V is the number of top items selected.

7. Convert to DataFrame and Define Schema

The resulting RDD, which contains tuples of (location, itemName, rank), is converted to an RDD of Row objects. A schema is defined for the final DataFrame, specifying the data types of the geographical_location, item_name, and item_rank columns.

- **Time Complexity**: O(V), as it involves converting each tuple to a row object and applying the schema.
- **Space Complexity**: O(V) for the resulting DataFrame.

8. Write Output Data

Finally, the processed data is converted to a DataFrame using the defined schema and written to a Parquet file at the location specified by outputPath. This Parquet file contains the top N items by count for each geographical location, along with their ranks.

- **Time Complexity**: O(V), writing out V entries to Parquet.
- **Space Complexity**: O(V) on the storage.

Design Considerations:

- The application is modular by structuring the data processing logic within the **TopItemsProcessor** object and the main application execution within the **Main** object. This separation enhances code readability and maintainability.
- RDDs and manual broadcast hash join techniques (withut explicitly writing the .join() technique) allow better control over data distribution, which may help processing and optimizing shuffle operations.
- The application has a flexible design by parameterizing key elements such as file paths and the topX value. This ensures the application's adaptability to different datasets and analytical requirements without needing code changes.

Flexibility and Reusability of code:

- *Parameters*: This allows better configurability for required parameters, promoting its reuse across different datasets and analytical needs. This is important in enabling the code to serve different data processing scenarios with ease.
- Extension to Different data manipulation techniques: The logic in **TopItemsProcessor**, focusing on ranking and aggregation through RDD transformations, may be a template for various data processing tasks beyond count-based aggregations, such as calculating averages or sums over different columns like timestamps.

Spark Configuration:

```
val spark = SparkSession.builder()
  .appName("Top X Detected Items")
  .master("local[*]")
  .getOrCreate()
```

- *Application Name: .appName("Top X Detected Items")* sets the name of the Spark application. This name will appear in the Spark UI and identifies the application in cluster management interfaces.

- *Master URL: .master("local[*]")* specifies that the Spark application should run locally with as many worker threads as logical cores on the machine (**[*]** denotes using all available cores). I use this configuration for development and testing purposes, which should be acceptable for this technical test.

Assumptions:

The actual Parquet files A and B are not available for development and testing. To proceed with the implementation and testing of the data processing logic, we assume the need to generate sample data that closely mimics the structure and nature of expected real-world data.

1. Data Schema and Content:
    - Dataset A: Represents streaming data from video camera sensors, with each row corresponding to an event. The schema includes fields such as **geographical_location_oid**, **video_camera_oid**, **detection_oid**, **item_name**, **timestamp_detected**.
    - Dataset B: Serves as a static reference table with a smaller number of rows. The schema includes **geographical_location_oid** and **geographical_location** to provide contextual information for the events in Dataset A.

2. Data Scale: For simplicity and ease of computation and testing, Dataset A includes 100 rows to represent a subset of streaming event data, while Dataset B contains 3 rows to cover basic reference information.
3. File Location: src/scala/createParquet/, named sampleA.scala and sampleB.scala respectively.

Testing:

The test cases help to validate the correctness and integrity of the data after processing:

1. **assert(outputDF.count() > 0, "Output DataFrame should not be empty")**:
    - **Purpose**: Ensures that the data processing results in a non-empty DataFrame.
    - **Rationale**: An empty DataFrame after processing could indicate a failure in the data flow, such as incorrect input data or a faulty join operation. This assertion verifies that the expected output contains data, confirming that the process has run and shown results.
2. **assert(outputDF.columns.contains("geographical_location"), "Output should contain 'geographical_location' column")**:
    - **Purpose**: Checks that the column **geographical_location** exists in the output DataFrame.
    - **Rationale**: The presence of this column is crucial because it indicates successful joining and projection operations. It also confirms that the output data includes the necessary context for location-based analysis.
3. **assert(outputDF.columns.contains("item_name"), "Output should contain 'item_name' column")**:
    - **Purpose**: Verifies that the **item_name** column exists in the output DataFrame.
    - **Rationale**: The **item_name** is an important piece of information expected to result from the data processing. Its presence assures that the item data has been correctly aggregated and is included in the output, enabling further analysis based on the items detected by video cameras.
4. **assert(outputDF.columns.contains("item_rank"), "Output should contain 'item_rank' column")**:
    - **Purpose**: Confirms that the **item_rank** column exists in the output DataFrame.
    - **Rationale**: Since the application aims to rank items, the existence of the **item_rank** column signifies that the ranking logic has been applied as intended. This column is important for understanding the popularity or frequency of items within each geographical location.