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.

Time Complexity:

- The operation of reading Parquet files (**detectionsDF** and **locationsDF**) is I/O bound, which is O(N), where N is the number of records in the Parquet files.
- The .**groupBy** followed by .**agg** operation to calculate **item_count** involves shuffling data across the cluster to group records by the **geographical_location_oid** and **item_name**. This operation's complexity can be approximated as O(N log N) due to the shuffle and sort operations involved. The space complexity is O(M), where M <= N which is the number of unique combinations of **geographical_location_oid** and **item_name.**
- The .**join()** function is defaulted to Sort-Merge join if the methods are not specifically stated. This time complexity is O(N log N) + O(M log M), where N and M are the sizes of the 2 DataFrames. The space complexity is O(N + M).
- Applying a window function to rank items within each geographical location involves sorting and then ranking. The time complexity of sorting is O(N log N), and applying the ranking would be O(N), making the combined complexity O(N log N) + O(N) in terms of computational effort.
- Filtering the DataFrame to retain only the top X ranks and writing the output to a Parquet file is O(N), as it requires a single pass through the data. Writing to a Parquet file is again I/O bound.

Design Considerations:

- *Modular Design*: The separation of the application into different scala objects (**TopItemsProcessor** for processing logic and **Main** for application entry and running of other methods) allows modularity. This facilitates better maintenance and extension of the code by encapsulating functionalities within separate components.
- *Use of Spark DataFrame API*: The choice to utilize the DataFrame API for data manipulation and analysis leverages Spark's optimization capabilities for more efficient execution plans and resource management, making the application well-suited for processing large datasets.
- *Scalability and Performance*: By using transformations like **groupBy**, **agg**, and window functions, the code is designed to scale horizontally across a cluster. This scalability ensures that the application can handle increasing data volumes efficiently.
- *Parameterization*: The application is designed with flexibility in mind, allowing key parameters (e.g., input and output paths, **topX** value) to be passed as arguments. This parameterization makes the code adaptable to different datasets and requirements without modification.

Flexibility and Reusability of code:

- *Configurable Parameters*: By enabling the configuration of paths (**detectionPath**, **locationPath**, **outputPath**) and the **topX** parameter through command-line arguments or within the code, the application can easily be adapted to generate tables with different specifications or based on different datasets.
- *Extension to Different Metrics or Tables:* The core logic in **TopItemsProcessor** can serve as a template for processing other types of data or generating tables based on different metrics. For instance, with minimal changes, one could adapt the code to compute other requirements like the average duration of item detections instead of the count, or the implementation of **TopItemsProcessorSkewed** builds upon this and uses AQE for computation.
- *Incorporation of Additional Data Sources:* The modular design supports the easy incorporation of additional data sources. For example, if future requirements include correlating additional data (e.g. detection data with weather conditions), one could seamlessly integrate a new DataFrame for weather data and extend the join and aggregation logic accordingly.
- *Adaptability to Data Schema Changes:* The use of DataFrame API and column-based operations, compared to fixed schema RDDs, provides resilience against changes in data schema. New columns can be added to the input data without breaking existing processing logic, enhancing the code's longevity and adaptability.

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.