xData Technical Test for Nigel Mun

Distributed Computing Tasks II

1. Suppose there is data skew in one of the geographical locations in Dataset A. Please provide another code snippet on how you will be re-implement part of the program to speed up the computations (10 marks)

<code is shown in the TopItemsProcessorSkewed object>

In addressing data skew within my Spark application, I primarily utilized salting and broadcast hash join. My approach to mitigating skew involved introducing salting to diversify the keys associated with the data, ensuring a more uniform distribution across partitions. This is important in solving the issue of placing too much computation on any node due to unevenly distributed data, which is common when dealing with skewed datasets.

To complement the salting strategy, I used broadcast joins, whereby a smaller referrrence dataset is distributed to all nodes in the Spark cluster. This allows for efficient data join operations to be performed locally on each node. The broadcast join is effective in the context of salted joins, as it ensures that each node can independently match records from the skewed dataset to the reference data without additional network overhead. I did not explicitly use .join() operation, but rather, use .get() to compare and match the location ID from both RDDs.

I also used RDD operations for data transformation and aggregation, such as *flatMap*, *reduceByKey*, *map*, and *groupByKey*. They enable proper manipulation of data, allowing for the aggregation of counts and ranking of top items efficiently in a distributed environment.

In conclusion, salting addresses the root cause of skew by redistributing the data more evenly across partitions. The use of broadcast joins enhances the efficiency of joining operations. Finally, RDD transformations allow better data processing, ensuring that the application can scale effectively with increasing data volumes. This approach can lead to improved overall performance and scalability of the application.

2. Explain the different sorting strategies in Spark and which strategy you will be adopting when joining Parquet File 1 and 2 if you are implementing the code in Spark Dataframe.

Spark implements several sorting strategies to manage how data is organized and joined across its distributed architecture, such as including Sort Merge Join, Broadcast Hash Join, and Shuffle Hash Join.

Sort Merge Join is Spark's default join strategy for large datasets that cannot fit into memory. It sorts both datasets by the join keys and then merges them with minimal shuffling. This strategy is particularly effective for evenly distributed datasets but can become less efficient if there's significant data skew.

Broadcast Hash Join involves broadcasting the smaller of the two datasets to all nodes in the cluster, where it's joined in memory with the partitioned data from the larger dataset. This strategy is efficient for joins where one dataset is significantly smaller than the other, reducing the need for data shuffling.

Shuffle Hash Join is used when both datasets are too large to be broadcasted but small enough to fit in memory after being partitioned. This strategy hashes the join keys and shuffles the data to ensure that matching keys are co-located, facilitating the join operation.

In my code, I have implemented the Broadcast Hash Join strategy. This is done by broadcasting the smaller dataset (Parquet File 2, containing geographical locations) and joining it with the larger dataset (Parquet File 1, detection events). This approach was chosen for several reasons:

- **Efficiency**: Broadcasting a small dataset ensures that the join operation is computationally less intensive compared to shuffling of data across the cluster. This method reduces the overhead associated with data shuffling, leading to faster join operations.
- **Scalability**: Since the broadcast dataset is sent to all worker nodes ahead of the join operation, the larger dataset does not need to be shuffled across the network. This makes the join operation scale better as the size of the larger dataset increases, assuming the smaller dataset is still small enough to be efficiently broadcasted.
- **Cost-effective**: By minimizing data shuffling, there is a reduction to the overall resource consumption of the Spark job. This can lead to cost savings, especially in pay-per-use cloud-based environments.