Distributed Computing Tasks II

To handle data skew in one of the geographical locations in Dataset A, we can use a technique called **salting**. Salting involves adding an extra field to the keys before performing a join operation to distribute the data more evenly across partitions.

Method: Add a Salt to the Keys:

- Modify the keys in Dataset A by adding a random salt to distribute skewed data more evenly

```
def addSalt(row: Row, numSalts: Int): Seq[(Long, Row)] = {
  val locationId = row.getAs[Long]("geographical_location_oid")
  (0 until numSalts).map(salt => (locationId * numSalts + salt, row))
}
```

- Perform the join operation using the salted keys.

```
def combine2RDDs(rddAMapped: RDD[(Long, Row)], rddBMapped: RDD[(Long,
String)], numSalts: Int, applySalt: Boolean): RDD[Row] = {
  val rddASalted = if (applySalt) {
    rddAMapped.flatMap { case (key, row) => addSalt(row, numSalts) }
  } else {
    rddAMapped.map { case (key, row) => (key, row) }
  }

  val rddBSalted = if (applySalt) {
    rddBMapped.flatMap { case (key, value) => (0 until numSalts).map(salt
=> (key * numSalts + salt, value)) }
  } else {
    rddBMapped.map { case (key, value) => (key, value) }
  }

  val rddAMappedDistinct = rddASalted.distinct()

  val matchedPairs = rddAMappedDistinct
    .join(rddBSalted)
    .map { case (_, (rowA, geoLocation)) =>
      Row(
        rowA.getAs[Long]("geographical_location_oid"),
        rowA.getAs[Long]("video_camera_oid"),
        rowA.getAs[Long]("detection_oid"),
        rowA.getAs[String]("item_name"),
        rowA.getAs[Long]("timestamp_detected"),
        geoLocation
      )
    }

  val unmatchedPairs = rddAMappedDistinct
    .leftOuterJoin(rddBSalted)
    .filter { case (_, (_, geoLocation)) => geoLocation.isEmpty }
    .map { case (_, (rowA, _)) =>
      Row(
        rowA.getAs[Long]("geographical_location_oid"),
        rowA.getAs[Long]("video_camera_oid"),
        rowA.getAs[Long]("detection_oid"),
```

```
        rowA.getAs[String]("item_name"),
        rowA.getAs[Long]("timestamp_detected"),
        null
      )
    }

  matchedPairs.union(unmatchedPairs)
}
```

The different sorting strategies in Spark are:

1. Range Partitioning
- Partitions data based on ranges of the sort key.
- Balances data across partitions.
- Suitable for ordered data.

2. Hash Partitioning
- Partitions data based on the hash of the sort key.
- Simple and fast.
- Can result in imbalanced partitions if the hash function is not well-suited to the data distribution.

3. Sort-merge Join
- Sorts both datasets on the join key and then merges them.
- Efficient for large datasets.
- Requires both datasets to be sorted.
- Minimizes shuffle stages.

4. Broadcast Hash Join
- Broadcasts the smaller dataset to all executors.
- Allows a fast hash join with the larger dataset.
- Efficient for small datasets.
- Can result in memory issues if the smaller dataset is too large.

Recommended Strategy: Since both datasets are large (> 1milliion rows), the Sort-merge join is recommended to minimize shuffle stage and hence improve efficiency