**Design considerations**

1) The input paths, output path, and number of items to be considered are all flexible and to be specified while running the script.
2) The use of RDD is as instructed.
3) Duplicates are handled by using 'reduceByKey' on the key 'detection_oid', which I assumed to be unique across all cameras and locations. The data is shuffled here.
4) Counts are generated by 'geographical_location_oid' and 'item_name', which fulfils the task requirement of ranking item popularity by geographical location. The data is shuffled here.
5) The item and count are mapped to the corresponding 'geographical_location_oid'.
6) Owing to the small size of dataset B and that only "=" joins are needed, the broadcast hash join was used to join the datasets, which avoids an expensive shuffle-based join. Also avoids using .join explicitly.
7) Grouping the final RDD by 'geographical_location_oid' allows all counts of items for a given location to be on the same partition, which is necessary for ranking.

**Proposed Spark configuration**

**spark.executor.memory**: 8g

**spark.executor.cores**: 4

**spark.driver.memory**: 4g

**spark.default.parallelism**: 1000

**spark.serializer: org.apache.spark.serializer.KryoSerializer**

**spark.kryoserializer.buffer.max**: 512m

Justification: The process does involve multi-threaded tasks which would benefit from multiple cores, as well as various operations which can be done on memory. The configuration above uses default numbers that generally work.

Kryo serialisation is faster and more compact than the default Java serialisation, especially because our data contain the type "varchar(5000)", which is potentially very large and would benefit from compact serialisation.