HTX xData Assignment

Technical Design Document – Top-X Items per Geo

1. Problem Overview

For each geographical location, identify the top X items that have the most unique detections (`detection_oid`). Then, join these top items with the reference table of geographical names to produce the final output. For each geographical location, top X items

Column	Type	Description
geographical_location	string	Name of the geographical location
item_count	string	Rank of the item in the geo (1 = most popular)
item_name	string	Item name

2. Assumptions

- **Top X per geo** – Only the top X items per geographical location are considered.
- **Tie-breaking** – If multiple items have the same number of unique detections, rank alphabetically by item name.
- **Unique detection** – Only count **distinct** **`detection_oid`** per item to avoid duplicates.
- **Data completeness:** All events in `rddA` have valid `geographical_location_oid` and `item_name` .
- **Reference table** – `geoNames` dataset is static and fits in memory (can be broadcasted).
- **Output ordering** – Within each geo, items are sorted by rank (1 = most popular).

3. Key Design Decisions

3.1 Deduplication

- Deduplicate rows by `(geoId, itemName, detectionId)` so that each `detection_oid` only counts once.

```
def deduplicate(rdd: RDD[(Long, (String, Long))]): RDD[((Long, String), Long)] =
  rdd.map { case (geoId, (item, detectionId)) => ((geoId, item, detectionId), 1L) }
    .reduceByKey(_ + _)
    .map { case ((geoId, item, _), _) => ((geoId, item), 1L) }
```

- **Reasoning:** Ensures retries or duplicate sends do not inflate counts.
- **Shuffle:** Uses `reduceByKey` which performs map-side combine → lighter shuffle than `groupByKey`.

3.2 Aggregation per Geo

- Count total items per `(geoId, item)`:

```
def aggregateByGeo(rdd: RDD[((Long, String), Long)], aggFunc: AggFunc[Long]): RDD[((Long, String), Long)] =
  rdd.groupByKey().map { case ((geoId, item), counts) => ((geoId, item), aggFunc(counts)) }
```

- **Reasoning:** Flexible, can pass in `sum`, `max`, etc.
- **Drawback:** Uses `groupByKey`, which shuffles all values and may be memory-heavy. `reduceByKey` would be more efficient if only sums are needed.

3.3 Top-X per Geo

Steps:

1. Map `(geoId, item) → count`
2. `groupByKey` to collect all items per geo
3. Sort descending by count, take top-X, assign rank

```
def topXPerGeo(rdd: RDD[((Long, String), Long)], topX: Int): RDD[(Long, String, String)] = {
  val itemsByGeo = rdd.map { case ((geoId, item), count) => (geoId, (item, count)) }
    .groupByKey()

  itemsByGeo.flatMap { case (geoId, itemsIter) =>
    itemsIter.toList
      .sortBy { case (itemName, count) => (-count, itemName) }
      .take(topX)
      .zipWithIndex
      .map { case ((itemName, _), idx) => (geoId, (idx + 1).toString, itemName) }
  }
}
```

- **Advantages:** Easy to implement, deterministic ordering with tie-break by `itemName`.
- **Disadvantages:**

    - Brings **all items for a geo into memory** → risk if some geos are very large.
    - Shuffle-heavy since every item must go to the correct geo partition.
    - Can be optimized with a heap-based approach (`aggregateByKey` keeping only top-X locally).

3.4 Joining with Geo Names

- Join with static reference table (`geoNamesMap`) using broadcast:

```
val bcastB = topItems.sparkContext.broadcast(geoNamesMap)
topItems.map { case (geoId, rank, item) =>
  val geoName = bcastB.value.getOrElse(geoId, "UNKNOWN")
  (geoId, geoName, rank, item)
}
```

- **Why broadcast join:**
    - Reference table is small and fits in executor memory.
    - No shuffle, every executor can lookup locally.
    - Much faster than a full RDD join.

---

4. Shuffle and Complexity Analysis

Step	Shuffle	Time Complexity	Space Complexity
Deduplication (`reduceByKey`)	low	O(N)	O(unique keys per partition)
Aggregation (`groupByKey`)	high	O(N) shuffle + O(M) per geo	Must hold all counts per `(geo,item)` in memory
Top-X (`groupByKey` + sort)	high	O(N) shuffle + O(M log M) per geo	O(M) memory per geo for sorting
Broadcast Join	none	O(X × G)	O(G) memory (size of broadcast)

Where:

- **N** = total number of rows
- **M** = items per geo
- **X** = top items requested
- **G** = number of geo locations

---

5. Spark Configuration Recommendations

To run this job smoothly, we should tune a few Spark settings:

- **Executor memory**:

  Use `--executor-memory 4g` as a starting point. This gives each executor enough memory for shuffle data and top-X calculations without wasting resources. Increase if the job grows, decrease if the cluster is small

- **Number of executors**:

  Adjust to match the cluster. A good rule of thumb is to use most of the available cores, but avoid too many tiny executors (better to have a few bigger ones)

- **Shuffle partitions**:

  Spark defaults to 200 (`spark.sql.shuffle.partitions = 200`). For smaller datasets, you can lower this (20–50) to reduce overhead. For large data, increase it to spread the work more evenly

- **Serializer**:

  Set `spark.serializer=org.apache.spark.serializer.KryoSerializer`. Kryo is faster and uses less memory than Java serialization, which helps with shuffles.

- **Broadcast join threshold**:

  Leave as default (`spark.sql.autoBroadcastJoinThreshold = 10MB`). Our reference table is small enough, so Spark will broadcast it automatically and avoid a shuffle

- **Other useful options**:
  - `spark.speculation=true` → reruns slow tasks so they don't delay the job.
  - `spark.dynamicAllocation.enabled=true` → automatically adds/removes executors based on load.
  - Place shuffle spill directories on SSDs if possible, for faster performance.

---

6. Handling Data Skew

Problem:

Some `geoId`s in Dataset A may have significantly more items than others, causing **large partitions and slow tasks** when performing `groupByKey` or aggregation.

Solution (Salting):

1. Detect skewed geoIds based on count threshold or percentile.
2. For skewed geoIds, add a **random "salt"** to the key to split them across multiple partitions.
3. Aggregate locally per salted key.
4. Remove the salt to combine results for the final counts.

Illustrative code snippet:

```
val skewedGeoIds = geoCounts.filter(_._2 > skewThreshold).keys.collect().toSet
val broadcastSkewed = sc.broadcast(skewedGeoIds)

val geoItemCountsSalted = dedupedA.map { case ((geoId, item), _) =>
  if (broadcastSkewed.value.contains(geoId)) {
    ((geoId, Random.nextInt(10)), (item, 1L))
  } else {
    ((geoId, 0), (item, 1L))
  }
}
```

Benefit:

- Distributes skewed geoIds across partitions.
- Reduces memory pressure and task time.
- Improves overall pipeline performance.