Part 1: Physical Plan Analysis

Compute the cost of each physical plan below. For each select, choose the operation that would minimize the cost the most. You have the following data:

<table>
<thead>
<tr>
<th>Table / Index</th>
<th>Cost Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Places(location)</td>
<td>M = 100 Unclustered index (located) for SampleDNA</td>
</tr>
<tr>
<td>Symptoms(symptom)</td>
<td>Unclustered index (toxin) for ExposedPlace</td>
</tr>
<tr>
<td>SampleDNA(sid, symptom, sequence, located)</td>
<td>Unclustered index (symptom) for ExposedPlace</td>
</tr>
<tr>
<td>ExposedPlace(toxin, symptom, located)</td>
<td></td>
</tr>
</tbody>
</table>

- **Index scan on S**
  - Cost = 1 + $B(E)$
  - Index scan on S
  - Cost = $B(E) + B(S)$
  - Sequential scan on E
  - Cost = 1 + 1
Part 2: Distributed Computing Problem Solving

Assume for these problems that you have a relation SampleDNA(sid, symptomatic, sequence, located)
Sequences are of, at most, length 2000. All sequences only contain the nucleotides A, T, G, and C.

<table>
<thead>
<tr>
<th>SampleDNA</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>symptomatic</td>
<td>sequence</td>
<td>located</td>
</tr>
<tr>
<td>49396937</td>
<td>T</td>
<td>ATTTGCATGCACGTAAA...</td>
<td>Seattle WA</td>
</tr>
<tr>
<td>68478053</td>
<td>F</td>
<td>ATCCGCATGCGCGAAAA...</td>
<td>Boston MA</td>
</tr>
</tbody>
</table>

For each of the problems write MapReduce pseudo-code and a Spark function to compute the result.
For MapReduce, assume you are given sid as a key and the remaining attributes as a tuple value.
For Spark, assume you are given an RDD r. Compute the information (don’t worry about outputting).

1. Count number of symptomatic samples that were found in each location. Get the max count (you don’t need the associated location).

Map(sid, values):
   if values.symptomatic then:
       emitIntermediate(values.located, sid)
       # could also be emitIntermediate(values.located, 1)

Reduce(location, list):
   emit(list.length)

JavaRDD<> x = r.filter(t -> t.get(SYMPTOMATIC))
              .mapToPair(t -> new Tuple2<>(t.get(LOCATED), 1))
              .reduceByKey((v1, v2) -> v1 + v2)
              .max(new Dummy())

static class Dummy implements Serializable, Comparator<Row> {
    @Override
    public int compare(Row o1, Row o2) {
        return Integer.compare(o1.get(1), o2.get(1));
    }
}


2. For symptomatic samples find the distribution of the nucleotides (i.e. count the number of times nucleotide X appears in index n), for each nucleotide. Using that information determine which indexes have an occurrence of nucleotide T greater than 95%. (HW is not nearly this complicated)

Map(sid, values):
    if values.symptomatic then:
        for k from 0 to values.sequence.length:
            emitIntermediate(values.sequence[k], k)

Reduce(nucleotide, index_list):
    distribution = [0] * 2000
    for each k in index_list:
        distribution[k]++
        emit(nucleotide, distribution)

Map(nucleotide, distribution):
    for k from 0 to 2000:
        emitIntermediate(k, (nucleotide, distribution[k]))

Reduce(index, labeled_count_list):
    total = sum(labeled_count_list)
    if (labeled_count_list[“T”] > 0.95 * total):
        emit(index)

    // This Java is not syntactically correct "\(\_\_\_\_\_\(\_\)\_\)\_\_\_\_"
    JavaPairRDD<> dist = r.filter(t -> t.get(1))
        .flatMapToPair(t -> generateCount(t))
        .groupByKey()
        .mapToPair(pair -> generateDist(pair));

    JavaRDD<Row> idx = x.flatMapToPair(pair -> generateIndexCount(pair))
        .groupByKey()
        .filter(pair -> majorityCheck(pair))
        .map(pair -> RowFactory.create(pair._1));
private <T> Iterator<Tuple2<?>> generateCount(T t) {
    List<Tuple2<?>> out = new List<Tuple2<?>>();
    for (int k = 0; k < t.get(SEQUENCE).length(); k++) {
        out.add(new Tuple2<>(t.get(SEQUENCE).charAt(k), k));
    }
    return out;
}

private <T> Iterator<Tuple2<?>> generateDist(T pair) {
    int[] out = new int[2000];
    for (int k : pair._2) {
        out[k]++;
    }
    return Tuple2<>(pair._1, out);
}

private <T> Iterator<Tuple2<?>> generateIndexCount(T pair) {
    List<Tuple2<?>> out = new List<Tuple2<?>>();
    for (int k = 0; k < 2000; k++) {
        out.add(new Tuple2<>(k, new Tuple2<>(pair._1, pair._2[k])));
    }
    return out;
}

private <T> boolean majorityCheck(T pair) {
    int total = 0;
    int tCount = 0;
    for (Tuple2<> val : pair._2) {
        total += val._2;
        if (val._1 == 'T') tCount = val._2;
    }
    return tCount > 0.95 * total;
}