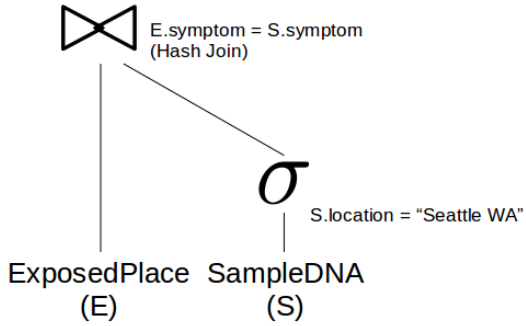
Worksheet 8

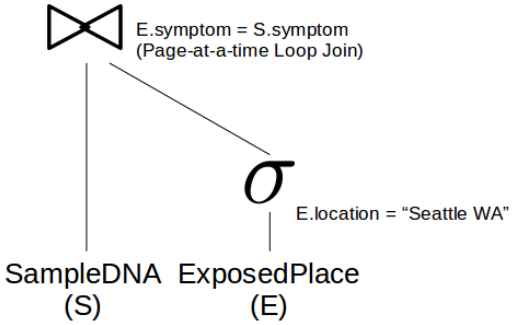
Part 1: Physical Plan Analysis

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:

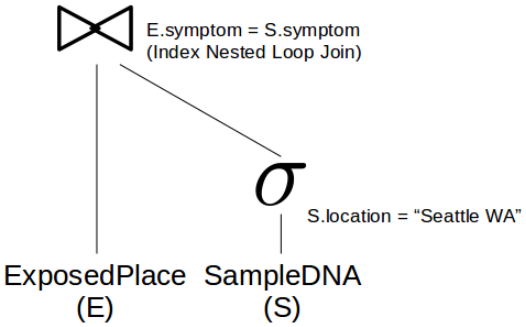
Places(<u>location</u>) Symptoms(<u>symptom</u>) SampleDNA(<u>sid</u>, symptom, sequence, located) -- located is a foreign key to Places -- symptom is a foreign key to Symptoms ExposedPlace(toxin, symptom, located) -- located is a foreign key to Places -- symptom is a foreign key to Symptoms	M = 100 Unclustered index (located) for SampleDNA Unclustered index (toxin) for ExposedPlace Unclustered index (symptom) for ExposedPlace	
B(SampleDNA) = 10^5 T(SampleDNA) = 10^6	B(ExposedPlace) = 10^4 T(ExposedPlace) = 10^5	B(Places) = 10^3 T(Places) = 10^6 B(Symptoms) = 10^1 T(Symptoms) = 10^4



Index scan on S
Cost = 1 + B(E)



Sequential Scan on E
Cost = B(E) + B(S)



Index scan on S
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.

SampleDNA			
sid	symptomatic	sequence	located
49396937	T	ATTTCGATGCGCGTAAA...	Seattle WA
68478053	F	ATTCCGATGCGCGAAAA...	Boston MA
...	...	...	

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 ¯\_(ツ)_/¯
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));
```

```
// t should be original rows
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;
}

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

// pair should be (nucleotide, distribution)
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;
}

// pair should be (index, Iterable<(nucleotide, count)>)
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;
}
```