Scalable Spatio-Textual Similarity Joins (22marks)

Background: In modern data science, joining datasets based on similarity is a critical task. This challenge is particularly complex and relevant in the domain of **Geographic Information Systems (GIS)** and location-based services. Real-world entities, such as restaurants, landmarks, or social media posts, are often described by both their physical location and a rich set of textual attributes. The ability to connect records from different data sources—for instance, matching Foursquare venues with Google Maps listings, or linking tweets to news articles about the same event—is crucial for data enrichment, entity resolution, and contextual analysis.

This project tackles the problem of performing a **spatio-textual similarity join**. You will develop a scalable system to identify pairs of records from two distinct datasets that are not only **spatially close** but also **textually similar**. This requires moving beyond simple database joins and implementing advanced, two-dimensional filtering techniques on the Apache Spark parallel computing framework.

Task: Given two sets of records, **Dataset A** and **Dataset B**, where each record R = (L, S) has two attributes: a spatial location in 2D space (i.e., R.L is a coordinate (x, y)) and a textual description (i.e., R.S is a set of terms/words).

The task is to find all pairs of records, (R_A, R_B), where RA is in Dataset A and RB is in Dataset B, which satisfy **both** of the following conditions simultaneously:

- Their **Euclidean distance** is less than or equal to a given threshold ***d***.
- Their **Jaccard Similarity** on the set of terms is greater than or equal to a given threshold ***s***.

Their **Euclidean distance** is less than or equal to a given threshold ***d***:

$$\sqrt{(R_A.L.x - R_B.L.x)^2 + (R_A.L.y - R_B.L.y)^2} \leq d$$

Their **Jaccard Similarity** is larger than or equal to a given threshold ***s***, that is:

$$\frac{|R_A.S \cap R_B.S|}{|R_A.S \cup R_B.S|} \geq s$$

Sample Data:

Given the following two sample files, `A.txt` and `B.txt`, and the thresholds `d = 2.0` and `s = 0.5`:
File: `A.txt`

```
A0#(1,1)#apple banana orange
A1#(10,10)#grape kiwi pear
A2#(2,1)#apple banana kiwi
```

File: `B.txt`

```
B0#(1,2)#apple banana grape
B1#(11,11)#kiwi pear mango
B2#(8,8)#apple lemon
```

Output Format:

The output file contains all qualifying pairs along with their calculated distance and similarity. The output format is `(recordA.id,recordB.id):distance_value,jaccard_similarity_value`.

- Each pair must consist of one record from Dataset A and one from Dataset B.
- For the distance and similarity values, round the results to 6 decimal places.
- There should be no duplicate pairs in the output.
- **Sorting Rule:** The output pairs must be strictly sorted according to the following two-level rule. The sorting must be based on the **numerical value** of the IDs, not their string representation (e.g., A10 should come after A2).
  - **Primary Key:** Sort in ascending order based on the numerical value of the ID from Dataset A (e.g., for `A2` and `A10`, sort by `2` and `10`).
  - **Secondary Key:** For pairs with the same recordA.id, sort in ascending order based on the numerical value of the ID from Dataset B.

Given the sample datasets above with the threshold `d = 2.0` and `s = 0.5`, the output result should be:

```
(A0,B0):1.0,0.5
(A1,B1):1.414214,0.5
(A2,B0):1.414214,0.5
```

Code and Execution:

The code template has been provided below. Your code should take four parameters: the path to the first input file, the path to the second input file, the output folder, and the distance threshold `d`, and the similarity threshold `s`. You must use the following command to run

your code:

```
$ spark-submit project3.py <input_A_path> <input_B_path> <output_path> <d> <s>
```

Some notes

- You must design an exact approach to finding similar records (*Please revisit Week 8 slides for more tips*).
- You need to use a grid index to perform the distance join.
- For similarity join, check the paper mentioned in the slides Efficient Parallel Set-Similarity Joins Using MapReduce. SIGMOD'10
- **You cannot compute the pairwise similarities directly (this method has O(N^2) complexity)!!!**
- Regular Python programming is not permitted (must use RDD or DataFrame, no third-party libraries).
- When testing the correctness and efficiency of submissions, all the code will be run with *two local threads* using the default setting of Spark. Please be careful with your runtime and memory usage.
- **Input File Format & Assumptions**: You can assume the input file structure is consistent and well-formed. Each line will strictly follow the `record_id#(x,y)#term1 term2 ...` format. The main components are always separated by `#`, and terms are always separated by single spaces. You are **not** required to handle malformed lines. You can assume that all `record_id`s from the first input file (Dataset A) will begin with the prefix 'A', and all `record_id`s from the second input file (Dataset B) will begin with the prefix 'B'. The character following the prefix will be the start of a unique numerical identifier.

Marking Criteria

Your source code will be inspected and marked based on readability, correctness, and efficiency.

- Submission can be compiled and run on Spark => +6
- Correctness (no unexpected pairs, no missing pairs, correct order, correct distance and similarity values, correct format) => +4
- Technical Implementations => +8
  - Frequency Sorting (2 marks): string to number conversion, broadcasting
  - Correct Prefix Filtering Implementation (3 marks): calculating prefix, filtering based on prefix
  - Correct Grid Index Implementation (3 marks): partitioning the space, filtering based on the grid
- Efficiency => +4
  - This score is based on the rank of your program's runtime on the largest test case (e.g., the top 25% got all 4 marks)

- To be eligible for efficiency marks, your submission must produce the correct result and not be a pairwise method

Solution

```
import sys
from pyspark import SparkConf, SparkContext
from pyspark.sql.session import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
import math
import builtins

class project3:
    def run(self, inputpathA, inputpathB, outputpath, d, s):
        # You can use either RDD or DataFrame APIs
        d = float(d)
        s = float(s)
        #Spark
        conf = SparkConf().setAppName("project3")
        sc = SparkContext(conf= conf)

        def parse_line(line):
            parts = line.strip().split('#')
            rid = parts[0]
            coord = parts[1].strip('()').split(',')
            x ,y = float(coord[0]), float(coord[1])
            terms = parts[2].split()
            num = int(rid[1:]) if len(rid) > 1 and rid[1:].isdigit() else 0
            return (rid, num, (x, y), set(terms))

        rddA = sc.textFile(inputpathA).map(parse_line).cache()
        rddB = sc.textFile(inputpathB).map(parse_line).cache()

        terms_count = rddA.flatMap(lambda rec:
rec[3]).union(rddB.flatMap(lambda rec: rec[3])).map(lambda t:
(t,1)).reduceByKey(lambda a,b :a+b)
        freq_map = dict(terms_count.collect())
        bc_freq = sc.broadcast(freq_map)

        def assign_grid(record):
            rid, num, (x,y), terms = record
            grid_x = int(math.floor(x/d))
            grid_y = int(math.floor(x/d))
            for dx in (-1, 0, 1):
                for dy in (-1, 0, 1):
                    yield ((grid_x + dx, grid_y + dy), record)
        gridA = rddA.flatMap(assign_grid)
```

```
        gridB = rddB.flatMap(assign_grid)
        spatial_candidate = gridA.join(gridB).map(lambda kv: kv[1])

        def euclidean(p, q):
            return math.hypot(p[0] - q[0], p[1] - q[1])

        spatial_filter = spatial_candidate.map(lambda pair: (pair[0],
pair[1], euclidean(pair[0][2], pair[1][2]))).filter(lambda x: x[2] <= d)

        def prefix_jaccard(rec):
            recA, recB, dist = rec
            termsA, termsB = recA[3], recB[3]
            lenA, lenB = len(termsA), len(termsB)
            t = math.ceil(s * (lenA + lenB) / (1.0+s))

            prA = builtins.max(lenA-t+1,0)
            prB = builtins.max(lenB-t+1,0)

            sortedA = sorted(termsA, key = lambda w: bc_freq.value.get(w,
0))
            sortedB = sorted(termsB, key = lambda w: bc_freq.value.get(w,
0))
            prefixA, prefixB = sortedA[:prA], sortedB[:prB]

            if not set(prefixA).intersection(prefixB):
                return  None

            inter = len(termsA & termsB)
            union = len(termsA | termsB)
            js = float(inter) / union if union > 0 else 0.0
            if js >= s:
                return (recA[1],recB[1], recA[0], recB[0], dist, js)
            return None

        def trim(x):
            s = format(x, '.6f')
            s = s.rstrip('0').rstrip('.')
            if '.' not in s:
                s += '.0'
            return s

        filtered = spatial_filter.map(prefix_jaccard).filter(lambda x: x is
not None)

        results = filtered.distinct().sortBy(lambda x: (x[0], x[1]))
        output = results.map(lambda r: f"({r[2]},{r[3]}):{trim(r[4])},
{trim(r[5])}").coalesce(1)
        output.saveAsTextFile(outputpath)
        sc.stop()
```

```
if __name__ == '__main__':
    if len(sys.argv) != 6:
        print("Wrong arguments")
        sys.exit(-1)
    project3().run(sys.argv[1], sys.argv[2], sys.argv[3], sys.argv[4],
sys.argv[5])
```