Parallel and Distributed Computing
Assignment1
Grid Processing

The Close Pairs Problem

In processing geographic data, we may have points which need to interact with geographically nearby points but do not interact with points beyond a certain distance. The first step in solving this problem would be to generate a list of "close pair" points that are within the threshold distance of each other. Next we would run through the list and perform whatever calculation we wanted between those point pairs.

This is exactly what we will be doing for this assignment. You will be given files of point data. The input is a set of CSV (Comma-Separated Values) text files. Each line in each file contains the ID of a point, the floating-point X coordinate, and the floating-point Y coordinate, separated by commas. No additional white space is present on the line. Here is an example input line:

Pt04,4.84535,4.05086

There is a single dataset provided: Assignment1Data.zip. It contains several files that will need to be loaded into the DBFS. Note that you should not load in each CSV file separately into your Spark program but instead let Spark do it in parallel by simply reading in the directory with all the CSV files in it. Also note that the test data is quite small. This is simply to make your debugging easier. However, we will be designing our Spark program to be able to run on a huge set of data.

Also note that for this assignment, we will stop at the creation of a "list of close pair points" portion of the problem. That is, we will not then perform an additional calculations on those close points to solve some specific problem. The focus of this assignment will be on the generic structure of handing any sort of problem of this type.

Brute-force Approach

One approach to solve this problem is to check every pair of points to see which pairs are within the given threshold distance. Unfortunately, this is an O(n^2) approach and does not scale well. It will also be difficult to put this into Spark Workers as all the points need the full list of other points in order to process correctly.

However, we can use this approach for a small set of points to get "ground truth" data as to the close pairs. That way, we can check the work of our more complex, but efficient, parallel solution later. I highly suggest doing this, as I will not be providing "ground truth" results for this assignment.

Forming a Grid

A standard approach to this sort of problem is to do space partitioning. That is, a grid is overlaid on the problem space. Each point gets assigned to a single grid cell. Note this is a quick computation as you can just take the floor of dividing the x and y coordinates by the grid cell size, which is usually set to the threshold distance. This gridding is done as a pre-processing step before we even compute distances between points. It only takes O(n) to grid all the points.

Next when trying to find all the point pairs that are within the threshold distance, we only need to examine other points in our own cell and the immediate neighbor cells. We need to look at the neighbor cells because our sample point may be very close to the edge of its cell and thus, the threshold distance will include things in the next cell over. But only the immediate neighbors since we set the cell size to the threshold distance when we gridded in the pre-processing step. With a fairly evenly distributed set of points over space, this should greatly reduce the number of point pairs being compared and speed up the calculation significantly.

Note that we still need to do the distance calculations even with the gridding because 2 points could be in the same cell but at opposite corner and thus, be outside the threshold distance. That is, the point's cell and its neighbor cells contain all the points that are within the threshold distance (meaning we do not need to examine other points at all), but there are also points within the point's cell and its neighbor cells that are outside of the threshold distance.

Forming a Grid with Pre-Pushed Points

The gridding approach has been shown to be highly effective in areas such a graphics. However, graphics applications store all their point data into a single computer's memory. We would like to run our code on data sets that are too large to fit into a single computer. Thus, we need to use Spark and do it in parallel.

At first glance, gridding as described above may seem the perfect fit. Each grid cell could be distributed to a different worker and they could all run in parallel. And in fact, that is exactly what we will be doing. However, when a worker is trying to find all the close pairs for one of its points, that worker will have to ask neighboring workers for their sets of points in order to complete the calculation. And since each worker has lots of points it will be doing this for, it will end up asking for those neighboring point sets again and again. Because the neighboring workers are on different machines, it means sending that point data across the network again and again. That is not an efficient solution.

To solve this problem, we "pre-push" the neighboring points to the cells that will need them. That is, when we originally build the grid of cells, we not only place the points into their "home" cells, but also push copies of them into the neighboring cells. That way, when the distance calculation starts, each cell already has all the data it needs to perform the computation and nothing needs to be shuffled over the network between workers.

Avoiding Duplicates

Obviously, if we are duplicating points across several cells, we run the risk of creating duplicate pairs. That is, we might find point pair (p3, p5) in one cell and the same pair in another cell. We also want to treat the point pairs as unordered. That is, (p3, p5) is the same as (p5, p3).

What we do not want to do is to calculate tons and tons of duplicates and then use Python sets (really bad) or Spark distinct (somewhat bad) to remove them. We want to avoid putting duplicates in our lists from the start.

There are several places to watch out for duplicates.

Since the order of the points in the pair does not matter, you probably do not want to push a point to all 8 of its neighbors. For example, assume p3 and p4 are within the threshold distance. Further, assume p3 is in cell (5,8) and p4 is in cell (5,7). That is, p4 is in the cell above p3. If we push p3 into (5,7) when it is processed and we push p4 into (5,8) when it is processed, then both cells with have both points. This will result in duplicates as the workers for both cells with come up with pair (p3, p4). We can avoid this situation by only pushing points one way rather than to all 8 neighbors.

Once it comes time for doing distance calculations, you will also want to avoid duplicates such as calculating (p3,p4) and (p4,p3) as two separate close pairs. And you should watch out for pair duplicates involving two points that were pushed into a cell, as they might be counted in their original cells as well.

And for this assignment, I am not going to give you “ground truth”. Instead I have given you a very small dataset. I would suggest writing a simple brute force program (with perhaps only Spark code to read in the data, but the rest in pure python) that calculates all the close pairs. That way, when you are setting up your pre-pushed grid, you can compare your results for accuracy above and beyond duplicates. This is what you would normally have to do in the real-world with no ground truth available.

Implementation Notes

You should only use RDD for this assignment. That is, no DataFrames.

After reading in all the lines of data from the csv files, you will have to turn them into some form of tuple point representation of your choosing. You will also need to calculate the grid cell where each point is located. This includes the cells the point is pushed into. A simple tuple (xCell, yCell) is a good representation for the cell locations. You will eventually need to collect all the points together that belong into each cell. This is a somewhat complex, but classic map reduce problem.

Your grid cell size and threshold distance should be variables that are easy to change near the top of your code. But make sure you pass them properly to any function that needs them and do not access them as "global" variables from within your functions.

After you have all your cells with associated points constructed into an RDD, you can finally use distance calculations to compute all the closes pairs of points.

Use the "persist" method to avoid recomputing any RDDs that are used multiple times. Do not use "persist" on any single-use RDDs.

Output

The results of your Spark program should be a list of all the close pair point IDs. Please also list the threshold distance as well as the length of your close pair list. An example is given below. Note that this example is just made up to show formatting and is not actual results from the given dataset:

```
Dist:0.8

3 [('Pt03', 'Pt07'), ('Pt04', 'Pt11'), ('Pt01', 'Pt5')]
```

For the given dataset, please use 0.75 as both your threshold distance and grid cell size.

Submission

You are to upload 2 documents as your submission. The first is your Python source code (PY). Make sure to have your name in a comment at the top of your source code. Second is a document that has a cut-paste of your results from running your code on the dataset input files.