CSE 512 Distributed Database Systems

FALL 2015

PHASE – 1

Design Document

Submitted By – Group 20

Vageesh Bhasin (1207683382)

Jon Lammers (1000024020)

Nagarjuna Myla (1207540161)

Bernard Ngabonziza (1207993380)

Anoop Sahoo (1207736344)

Madhu Meghana Talasila (1207740881)

1. Geometry Union

Definition: The union of a set S of polygons is the set of all points that lie in at least one of the polygons in S, where only the perimeter of all points is kept while inner segments are removed.

Given Information:

InputLocation – Location of the input in HDFS
OutputLocation – Location of the output in HDFS

Input Dataset Schema – x1, y1, x2, y2
Every row is a pair of points (longitude, latitude) which defines a polygon.

Output Dataset Schema – x, y
Every row is a point (longitude, latitude). The result polygon is formed by these points.

Algorithm:

The algorithm, PolygonUnion[1], is based from a research paper by Avraham Margalit and Gary D. Knott. The initial algorithm works on oriented polygons, i.e., polygons having edges with orientation. However, our input data does not fit this criteria, and hence we will apply the generalized algorithm. We start by finding the intersection points between the polygons.

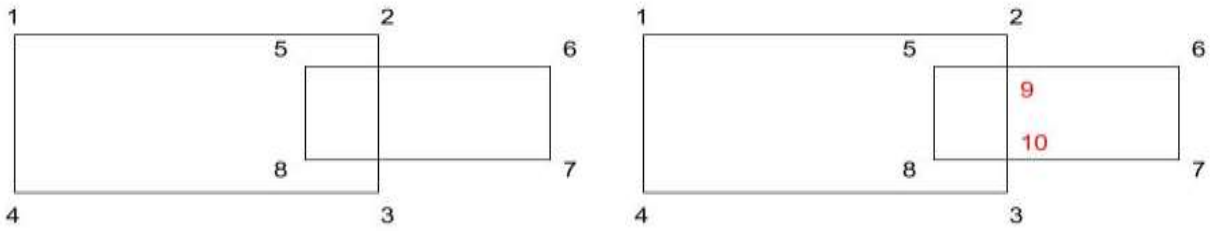

Figure 1 (a) Initial Overlapping polygons, (b) Intersection points

We then classify the edge fragments to be either inside, outside or on the boundary of the polygons. An edge fragment is classified *inside* if at least one of its endpoints is an inside vertex, or the two endpoints are boundary vertices and all the other points on the edge are inside points. An edge fragment is classified *outside* if at least one of its endpoints is an outside vertex, or the two endpoints are boundary vertices and all the other points on the edge are outside points. After classification of the edge fragments, we select the fragments based on the operation. In case of Geometry Union, we select all the edge fragments that have been classified as outside. Finally, we construct the result polygon using the selected edge fragments.

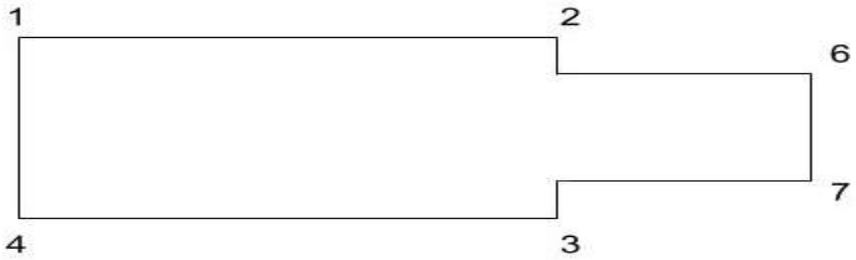

Figure 2 Final resultant polygon

The above algorithm works well for small sets of polygons, however using this on a larger set would not be an optimal solution. Another algorithm, called CascadingUnion[2], is a modification of this algorithm for faster processing. It breaks the input into smaller, manageable sets of polygon, and distributes the union computing across these subsets. It then takes the result of this operation, and recursively performs the union operation on intermediate results until it finds the result polygon.

Pseudocode:

1. Load file InputLocation and read data into RDD
2. Map the data into different partitions, using *mapPartitions()*
    a. Pass in the *PolygonUnion()* function to perform the Union operation on the partitioned set of polygons
    b. Save the results into an intermediate RDD
3. Gather the intermediate result polygons from the RDD using *repartition()*
4. Map the data again and perform a final union
    a. Pass the *PolygonUnion()* function again to get the union of the intermediates polygons, to form the final result polygon
    b. Save the result polygon into a file, at *OutputLocation*, on the HDFS
    c. Return true
5. Handle exceptions, by logging error and returning false

Testing Procedure:

1. Create custom dataset to cover test cases including:
    a. Island polygons (having no edges in common)
    b. Polygons with holes
    c. Overlapping polygons
    d. Bad/ Incorrect data
2. Create test program to utilize the test dataset and run the union function for verifying the correctness.
3. Quantify the performance of the algorithm for each type of data sets.

2. Geometry Convex Hull

Definition: The convex hull of a set of points P is the smallest convex polygon that contains all points in P. The output of the convex hull operation is the points forming the convex hull ordered in a clockwise direction.

Given Information:

InputLocation – Location of the input in HDFS
OutputLocation – Location of the output in HDFS

Input Dataset Schema – x1, y1
Every row is a point (longitude, latitude)

Output Dataset Schema – x, y
Every row is a point (longitude, latitude). The polygon defined by the points is the convex hull of the input dataset.

Algorithm:

The algorithm is based on Andrew's Monotone Chain[3] convex hull algorithm. This algorithm takes an input sorted by the x-value. The first point is on the convex hull by definition. The algorithm iterates over the dataset looking at each point in order. If the previous edge is a "left turn" (i.e. the slope is greater for top hull) from the next edge, then this point is removed from the convex hull. The point list is iterated front to back, looking for the upper surface of the convex hull, and then in reverse order looking for the lower surface.

The time complexity of algorithm is O(n log(n)). The initial sorting of the data, and the recursive search are both of this complexity. This algorithm will also provide an alternate interface to take an RDD and return an RDD. This is to enable the building of other Spark-based functions on top of this function without incurring the penalty of File I/O when not required.

Additional possibilities: We may attempt to improve this algorithm using the following:

- Step 3 & 4 of the Pseudo Code separately process the entire RDD. While this can be done in parallel, more performance may be gained by combining the calculation of the slope with the removal of non-hull points.
- According to Devroye and Toussaint[4], the algorithm can be improved by selecting a four vertex polygon consisting of the points with the highest and lowest X and Y coordinates, and then removing all points from contention that are within this polygon. The "Spatial Range Query" in this project may be inserted after the sort to greatly lower the number of points processed. This will not lower the complexity, as the sort will still drive complexity to O(n log n), but may still significantly improve performance.

Pseudocode:

1. Read input dataset into initial RDD
2. Use sortBy() to sort the input by the value of the x-coordinate.
    a. This step creates a sorted RDD by X so that we can process left to right. The leftmost and rightmost points are by definition in the convex hull.

3. Use Map to create an RDD with the slope to the next point from this point.
    a. The map calculates the slope of edge to the next point and adds to the point's tuple.
4. Use Filter to remove any points that have a "left turn" i.e. slope P-1,P < slope P,P+1
    a. The filter removes this point when the previous and next slope indicates a concave set of edges. Track the number of points removed across the cluster with an accumulator.
5. Recurse back to step 3, unless no points removed, then exit the recursion.
    a. If the accumulator indicates that some points were removed in step 4, then we recurse back to step 3 through 5 to check if any more points can be removed.
6. Reverse logic in steps 3-5, to add lower hull points, in parallel with the forward logic.
7. Take the union of upper and lower hulls and write to output file.

Testing Procedure:

1. Create custom dataset to cover edge cases including:
    a. 0, 1, 2, 3 and 4 point data sets
    b. Co-linear data
    c. Three-point convex hull
    d. Four-point convex hull
    e. All points on a circle (worst case, all points on hull)
2. Generate random data sets to verify input data in various sizes.
3. Create test program to compare all points in input data set to convex hull to verify correctness.
4. Quantify performance of the algorithm at each type of data set.
5. Use visualization tools for verification of correct output.

3. Geometry farthest pair

Definition: Considering a 2D Euclidean space , the geometrical farthest pair refers to two points *p1(x1,y1)* and *p2(x2,y2)* among a set of P points which are maximum apart from each other, which means whose Euclidean distance between these two points is maximum when compared to other points.

Given Information

Method Name: GeometryFarthestPair

Parameters:
String inputPath - The path of input file in HDFS to read data from.
String outputPath - The path of output file in HDFS to be write the results.

Method outcome:

1. A return value of type Boolean to tell about the success or failure. In case of success returns true else returns false.
2. A file is written to given outputPath in case of success. The output file will have two lines, each line will have two numeric values separated by comma that represents a point p(x,y) in 2D Euclidean space.

Input Data Requirements:
The input will be read from a file identified using given inputPath. Each line in the file should have two numeric values separated by a comma.

Ex: 1,2
3,4
-1,2.0

Algorithm

Bruteforce: To compute the farthest points we can use brute-force approach by calculating the pairwise Euclidean distance between all points in P let say there are n such points. This will give a time complexity of $O(n^2)$. As distance between (x_1, y_1) and (x_2, y_2) is same as (x_2, y_2) and (x_1, y_1) calculating it once is sufficient so we can reduce the number of loops by half. Even then the time complexity stills remains same because $O(n^2/2)$ reduces to $O(n^2)$.

Instead of calculating the pairwise distances between all the points we can reduce the problem space to number of points that lie on the convex hull because the farthest points should always lie on the convex. We can compute convex hull in O(nlogn) using Andrew's Monotone Chain[5] convex hull algorithm which is discussed above. We can apply brute force approach on these points which will be faster.

Pseudocode

1. Load the input data set using *SparkContext.textFile()* and store in inputRDD
2. Pass inputRDD to Convex Hull algorithm which is explained above and store the results in convexHullPointsRDD
3. Create a cartesian product using sparks cartesian method of all the points on convex Hull RDD.

4. Using map() on the cartesian product points compute distances between points.
5. Now we have distance along with points as key value pair.
6. Combine all the pairs obtained using *reduce()* method and get the most distant pair.
7. Save the output points into a outputpath file in HDFS using *saveAsTextFilePath()*.

Additional details on optimal algorithm: From the convex hull we will get upper and lower hulls as intermediate data. We use these upper and lower hulls and finds out all the antipodal pairs of the convex hull using rotating calipers [5][6] method in O(n). Hence overall time complexity will be O(nlogn). The obtained farthest points which are of the diameter endpoints of the convex polygon are saved into file in HDFS.

Pseudocode

1. Load the input data file using sparkcontext.textFile
2. Using mapPartitions split the input RDD into multiple RDDs
3. For each RDD call the convexHull algorithm and get the points on convex hull.
4. For each convexhull RDD find the antipodal pairs using rotating callipers.
5. rotatingCallipers method tries to find an initial antipodal pair by locating the vertex opposite to current vertex. A pair is called antipodal if they can admit parallel support lines that does not go through polygon.
6. Once initial antipodal pair is found its goes around the polygon by rotating the parallel lines until one line touches other edge of polygon, the new antipodal pair is found, similarly it finds all possible antipodal pairs.
7. we have duplicate pairs which will be eliminated during RDD *union()*
8. Combine all the pairs using *reduce()* method
9. The pairs which has longest distance are found using *max()* by key.
10. Save the output points into a outputpath file in HDFS using *saveAsTextFilePath()*.

Testing Procedure

1. Create custom dataset to cover edge cases including:
    a. 0 or 1 point
    b. 2 or more distinct points
    c. 2 points which are duplicate
    d. 2 or more collinear points
    e. Points on a circle
    f. Vertices of a regular polygon like square or hexagon
2. Generate random data sets to verify input data in various sizes.
3. Create test program to compare all points in input data set to geometry farthest pair to verify correctness.
4. Quantify performance of the algorithm at each type of data set.
5. Use visualization tools for verification of correct output.

4. Geometry closest pair

Definition: Given a set of points P, the closest pair is the pair of points that have the smallest Euclidean distance between them. This pair of points should lie in the convex hull. The input of the farthest pair operation is a set of points and the output is a pair of points which have the closest distances between each other.

Given Information:

Function Name: GeometryClosestPair

Arguments:
String InputLocation: The location of the input in HDFS
String OutputLocation: The location of the output in HDFS

Return Value: Boolean: If the function works well, return True otherwise return False.

Algorithm

Brute force

1. Calculate the distance between all the points using Euclidean function.
2. Find the minimum distance/ closest pair of points among all the points by comparing Euclidean distances.

Divide and conquer

This algorithm is for a 2-D plane.

1. Sort pairs on x-axis
2. Divide pairs into two sets by the median on x: x-med
3. Recursively find the minimum distance between two pairs in each set : min(set1), min(set2)
4. Find the minimum distance between two pairs residing in different sets: min(pair(set1),pair(set2))
5. The closest pair : min(min(set1),min(set2),min(pair(set1),pair(set2)))[7]

Pseudocode

Brute force

1. Load the input data set using *SparkContext.textFile()* and store in inputRDD
2. Create a cartesian product of all the points in inputRDD.
3. Apply map() on the cartesian product points using cartesian method of spark to compute distances between points.
4. Now we have distance along with points as key value pair.
5. Combine all the pairs obtained using *reduce()* method and get the closest pair.
6. Save the output points into a outputpath file in HDFS using *saveAsTextFilePath()*.

Divide and conquer

1. Create a sparkcontext
2. Read the input file from Hadoop to Spark RDDs points
3. Apply map partitions on points. Save to RDD

4. Apply map on the mapped partitions and use the divide and conquer algorithm to return the closet pair as key value pair to resultRDD.
5. Apply coalesce(1) on the above resultRDD to get all the key value pairs into one partition.
6. Now apply flatmap on the obtained result and find the closest pair using divide and conquer algorithm and save the result as saveAsTextFilePath() to Hadoop.

Testing Procedure

1. Create custom dataset to cover edge cases including:
    a. 0 or 1 point
    b. 2 or more distinct points
    c. 2 points which are duplicate
    d. 2 or more collinear points
    e. Points on a circle
    f. Vertices of a regular polygon like square or hexagon
2. Generate random data sets to verify input data in various sizes.
3. Create test program to compare all points in input data set to geometry closest pair to verify correctness.
4. Quantify performance of the algorithm at each type of data set.
5. Use visualization tools for verification of correct output.

5. Spatial Range Queries

Definition: Spatial Range Queries are used to inquire about certain spatial objects which lie inside a certain query window.

Given Information

In this case we are interested in finding the objects which are within a rectangular window space. The input data set which we are provided with is stored in HDFS using the Spark API. Our work is to display only the rectangles that are falling inside this window. The data from HDFS is loaded into RDDS using SparkContext function textFile. This function takes in the filename as a parameter.

Arguments
1. String InputLocation1- This is the location of the input dataset1 in HDFS.
Input dataset1 schema is: id, x1, y1, x2, y2. Here every row represents a pair of points (longitude, latitude) which defines a polygon. So this dataset has a bunch of Polygons.

2. String InputLocation2- This is the location of the input dataset2 in HDFS.
Input dataset2 schema is: x1, y1, x2, y2. Here we have a pair of points (longitude, latitude) which defines a polygon. This dataset is our query window for the range query.

3. String Output Location- This is the location of the output in HDFS.
Function Name: SpatialRangeQuery (Inputfile1, Inputfile2, Outputfile)

Algorithm

The Algorithm will run on each partition of Spark Cluster. This Algorithm will be used inside the filter function of spark.

1. For every polygon (rectangle) in InputDataset1 let say A.
2. Get all the polygons lying inside the InputDataset2 let say B (i.e. our Query Window) according to the logic:
    a. if((A.x1 <= abs(B.x1-B.x2)) && (A.x2 <= abs(B.x1-B.x2)) && (A.y1 <= abs(B.y1-B.y2)) && (A.y1 <= abs(B.y1-B.y2)))
    b. Save all the polygons/ rectangles of A that satisfy the above condition

Pseudocode

Steps to Code in Spark (Map, Filter, Reduce)

1. Create a sparkcontext Object.
2. Read the input files input1 and input2 from Hadoop to Spark RDDs polygonA and polygonB
3. Polygon A is to be partitioned into the spark cluster using MapPartitions
4. Polygon B is to be a read only copy across all spark clusters. Can be done using Broadcast
5. Apply filter on Broadcast variable (i.e. contains Polygon B RDD) and use the above algorithm to filter the entries of Map Partitioned Polygon A that lie inside Polygon B. The output is an RDD which contains list of IDs that passed the filter.
6. Combine the result of all partitions using spark coalesce(1) function.
7. Save the resultRDD, to Hadoop using saveAsTextFile.

Above algorithm returns Boolean value. True if algorithm works well otherwise false.

Testing Procedure

1. Create custom dataset to cover edge cases including:
    a. All data points lie inside the given query rectangle
    b. No data point lie inside the given query rectangle
2. Generate random data sets to verify input data in various sizes.
3. Create test program to compare all points in input data set to spatial range queries to verify correctness.
4. Quantify performance of the algorithm at each type of data set.
5. Use visualization tools for verification of correct output.

6. Spatial join query

Definition: Spatial join operation is used to combine two or more datasets with respect to a spatial predicate. Given a set of rivers and set of locations, spatial join returns all the rivers that lie inside or on the particular location.

Given Information

Input1 Dataset Schema: Aid, x1, y1, x2, y2
Every row is a pair of points (longitude, latitude) which defines a polygon. This set has a bunch of polygons.

Input2 Dataset Schema: Bid, x1, y1, x2, y2
Every row is a pair of points (longitude, latitude) which defines a polygon. This set has a bunch of polygons.

Output Dataset Schema: Aid, Bid1, Bid2, Bid3
Every row has one Aid and 0 - * Bid. Bids are the polygons that lie inside or on the polygon Aid

Function: SpatialJoinQuery (InputFile1, InputFile2, Outputfile)

Algorithms

Algorithms that run on each partition of spark cluster. Algorithms to be applied on Filter

Brute Force Approach

1. For each polygon/ rectangle in A.
2. Get all the polygons in B that lie inside A (filter). while(B.next !=null)
    a. if((B.x1<=abs(A.x1-A.x2)) && (B.x2<=abs(A.x1-A.x2)) && (B.y1<=abs(A.y1-A.y2)) && (B.y1<=abs(A.y1-A.y2))).
    b. Add all the polygons in B that satisfy the above condition to result[i] as key value pairs Aid, Bid1, Bid2.

Variant of R Tree: Sort-Tile-Recursive also known as STR

The concept of R Tree[8] is take the given input as leaf nodes and construct the tree basing on the minimum bounding regions also called as MBRs. Insert operation takes a lot of time in R tree. In our scenario, as we have static data, we can apply bulk uploads on R trees.

STRTree[9] is a simple and efficient algorithm for packing R Tree i.e, for bulk uploads. STRTree is a query-only R-tree created using STR algorithm. STRTree is simple to implement on two-dimensional spatial data, maximizes space utilization and more efficient for bulk inserts than R-tree. Tree has to be constructed in beginning. Once a query is made, insertions or deletions on the tree are not possible.

How STRTree Algorithm works?

1. Sort the rectangles by x-coordinate and partition them into S vertical slices.
2. Each slice contains a run of S*b rectangles.
3. Sort the rectangles in every slice by y-coordinate.

4. Pack sorted rectangles into nodes by grouping them in size of b.

$$\mathbf{P = [r/b] \quad S = \sqrt{P}}$$

Plan to implement STR Tree for spatial join

1. Create a STRTree using STR algorithm for Input set B using com.vividsolutions[10]
2. For each envelope (i.e., pair of coordinates/ each entry) in A
    a. Find all the list of data points in B that lie in the region of A using query(Envelope searchEnv) in com.vividsolutions
    b. Add all the polygons in B that satisfy the above condition to result[i] as key value pairs Aid, Bid1, Bid2.

Pseudocode

Steps to Code in Spark (Map, Filter, Reduce)

1. Create a sparkcontext[11]
2. Read the input files input1 and input2 from Hadoop to Spark RDDs polygonA and polygonB using JavaRDD = sparkcontext.textFile("")
3. Polygon B is to be partitioned into the spark cluster using MapPartitions[11] and stored in RDD
4. Polygon A is to be a read only copy across all spark clusters. This can be done using Broadcast and is stored in RDD.
5. Apply filter on Broadcast variable (contains Polygon A RDD) and use one of the above algorithm to filter (Use spark filter method: B.filter(A)) the entries of Map Partitioned Polygon B that lie inside Polygon A. The output is an mapRDD which contains list of values in the form of key value pairs (key: Aid, value: Bid1, Bid2..)
6. Apply spark coalesce(1) on mapRDD and save it as a outputRDD
7. Apply reduceByKey.collect on outputRDD and save result to resultRDD
8. Save the resultRDD to hadoop using saveAsTextFile.

The above algorithm returns boolean value. True if algorithm works well otherwise false.

Testing Procedure

1. Create custom dataset to cover edge cases including:
    a. All data points lie inside the given query rectangle
    b. No data point lie inside the given query rectangle
2. Generate random data sets to verify input data in various sizes.
3. Create test program to compare all points in input data set to spatial range queries to verify correctness.
4. Quantify performance of the algorithm at each type of data set.
5. Use visualization tools for verification of correct output.

Contributors

1) Geometry union – Vageesh Bhasin
2) Geometry convex hull – Jon Lammers
3) Geometry farthest pair – Nagarjuna Myla
4) Geometry closest pair – Bernard Ngabonziza
5) Spatial range query – Anoop Sahoo
6) Spatial join query – Madhu Meghana Talasila

References

1. Margalit A., Knott G.D., 1989, "An algorithm of computing the Union, Intersection or Difference of two polygons", http://www.cc.gatech.edu/~jarek/graphics/papers/04PolygonBooleansMargalit.pdf
2. Davis M., 2007, "Fast polygon merging in JTS using Cascaded Union", http://lin-ear-th-inking.blogspot.com/2007/11/fast-polygon-merging-in-jts-using.html
3. A. M. Andrew, Information Processing Letters, Volume 9, Issue 5, Pages 216-219 (1979), "Another Efficient Algorithm for Convex Hulls in Two Dimensions"
4. Luc Devroye and Godfried Toussaint, Computing, Vol. 26, 1981, pp. 361-366, "A note on linear expected time algorithms for finding convex hulls"
5. Shamos, M. (1978, May 15). (pp 77-82) Retrieved October 3, 2015, *http://euro.ecom.cmu.edu/people/faculty/mshamos/1978ShamosThesis.pdf*
6. Eppstein, David. "Convex hull and diameter of 2d point sets (Python recipe)".
7. Michiel Smid. Closest-point problems in computational geometry. In J.-R. Sack and J. Urutia, editors, Handbook of Computational Geometry, pages 877-935. Elsevier Science, Amsterdam, 2000.
8. Guttman, A. (1984). "R-Trees: A Dynamic Index Structure for Spatial Searching". Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84 (PDF). p. 47. ISBN 0897911288.
9. Bui Cong Giao, Duong Tuan Anh, The 2015 IEEE RIVF International Conference on Computing & Communication Technologies, "Improving Sort-Tile-Recusive Algorithm for R-tree Packing in Indexing Time Series"
10. http://www.vividsolutions.com/jts/javadoc/index.html
11. http://spark.apache.org/docs/latest/programming-guide.html