# CSE 512
# Distributed Database Systems
# Project Phase 1
# Implementation Plan

**Group 2:**
Audrey Wong
Boxin Du
Lei Chen
Rongyu Lin
Yihan Lu

# Table of Contents

# Project Plan

## Time Schedule

After a quick evaluation, we broke the project down into the following steps. Each step is followed by its estimated duration.

1. Install and configure Hadoop and Spark: Weeks 1-4
2. Get familiar with Spark and try demo applications: Week 5
3. Plan how to implement the 6 functions: Weeks 6-7
4. Implement the 6 functions and related unit tests: Weeks 8-9
5. Plan the system prototype demo: Week 10
6. Complete system integration: Week 11
7. Plan how to evaluate and analyze the system: Week 12
8. Perform experimental evaluation and analysis of the system: Week 13
9. Write the final report and create the poster presentation: Weeks 14-15

## Responsibilities

In order to let every team member have an opportunity to hone their teamwork skills, as well as their leadership skills, we decided to take turns being the leader of the team. More specifically, the following are the responsibilities of each member:

1. **Audrey Wong:** Geometry Convex Hull
2. **Boxin Du:** Geometry Closest Pair, evaluation and analysis
3. **Lei Chen:** Geometry Union, installation of Hadoop and Spark
4. **Rongyu Lin:** Spatial Range Query and Spatial Join Query, system integration
5. **Yihan Lu:** Geometry Farthest Pair, test and demo planning

## Platform

At present, we are running three Ubuntu 15.04 LTS instances using VMWare Player on each team member's personal computer. This is more flexible and convenient for development purpose. Since the three instances are running on one host machine, the advantage of parallelism cannot be seen. Later on, when doing the performance analysis, we may move to separated machines to achieve more accurate readings from the performance point of view. In the future, we also may deploy and test our functions on remote servers, such as Amazon AWS.

# Functions Plan

## 1. Geometry Union

We will be focusing on finding an efficient way to split the input polygons into partitions and convert them into a distributed form that can be processed using Spark. Since the geometric union function is a widely used function and there exists several implementations that can be utilized, we decided to use Java Topology Suite (JTS) to handle the non-distributed part. We will have multiple implementations with comparisons and analyses. The following is the pseudocode:

```
// 0. copy local file to HDFS
Utils.copyFile(localFilePath, hdfsFilePath);

// 1. read the lines from the input file in HDFS
JavaSparkContext sc = new JavaSparkContext(conf);
JavaRDD<String> lines = sc.textFile(hdfsFilePath);

// 2. Geometric Union
// 2.1 map: convert each line of string in the input file into an JTS polygon
JavaRDD<Geometry> polygons = lines.map(new Function<String, Geometry>() {
    public Geometry call(String s) {return convertToPolygonFromLeftTopAndRightBottom(s);}
});

// 2.2 reduce: combine every 2 polygons
Geometry finalPolygon = polygons.reduce(new Function2<Geometry, Geometry, Geometry>() {
    @Override
    public Geometry call(Geometry g1, Geometry g2) throws Exception {return g1.union(g2);}
});

// 3. output to an HDFS file
writeToOutputFile(finalPolygon)
```

From the pseudocode, we can see that the core part is in “2. Geometric Union”, more specifically “2.2 reduce”. The reduce code above is more like a brute-force solution. It simply unions every two polygons into one until it reaches the final polygon as shown in Figure 1. A similar idea used in JTS is called the cascaded union [1]. In the real implementation, we will try multiple ways to improve. For example, the first optimization would be trying to combine polygons that overlap. To find polygons that overlap, a simple way is to horizontally partition the input polygons by the x-coordinate of their leftmost vertex. Although this cannot guarantee the partitioned polygons to overlap, as shown in Figure 2, it will likely provide some degree of speedup in cases where polygons are more uniformly distributed. Again, this is just one simple example; we will try to come up with more sophisticated and more effective ways of partitioning.

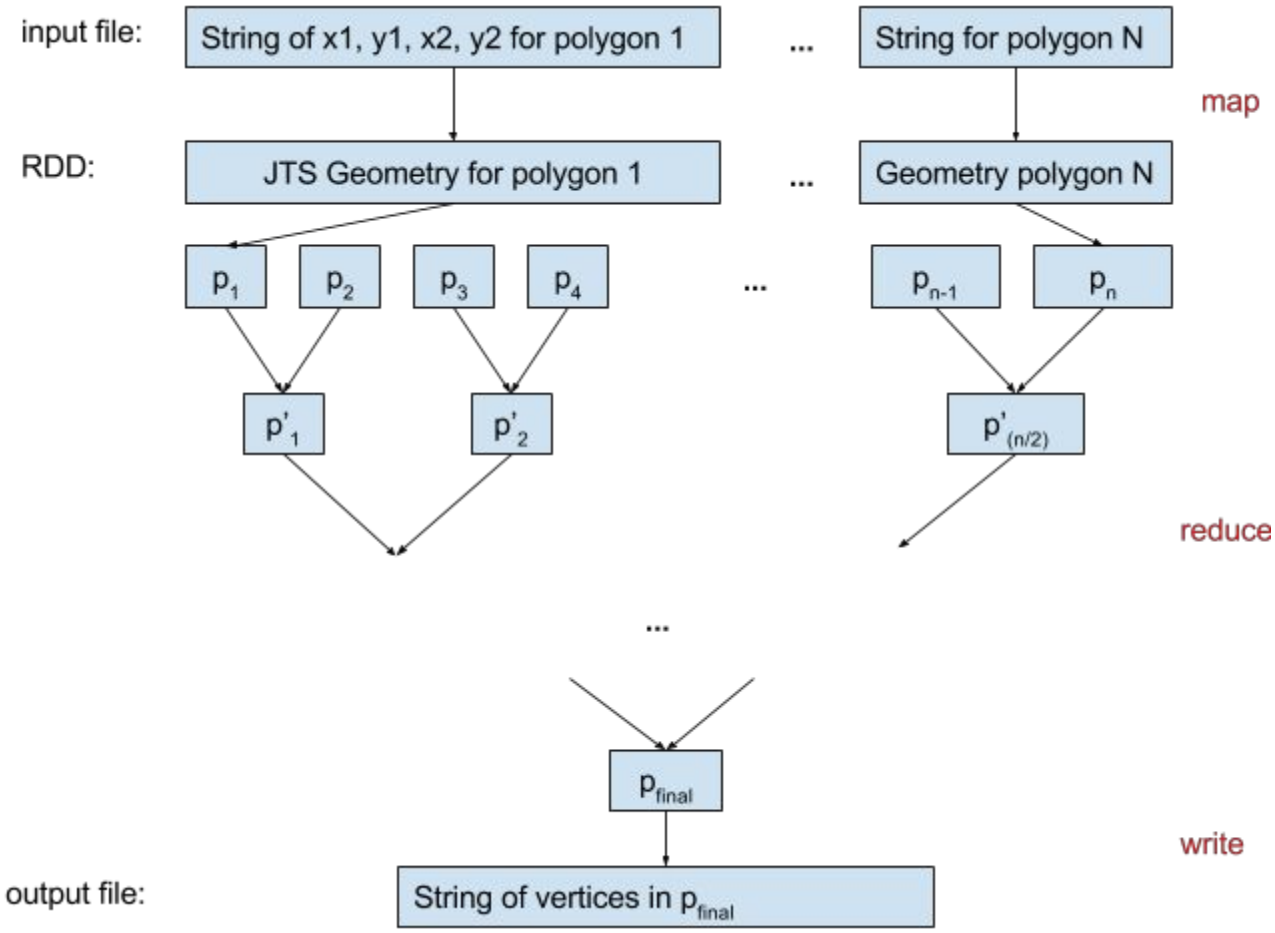


**Figure 1:** Data flow of the Geometry Union function

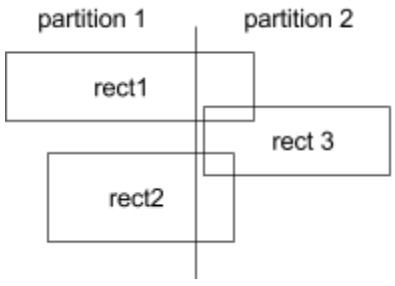


**Figure 2:** The case where the partitioned polygons, rect1 and rect2, do not overlap

## 2. Geometry Convex Hull

Pseudocode for the algorithm is provided below:

```
1   lines = read lines from inputFile
2   coordinates = map lines to coordinates:
3           c = array of coordinates
4           for each line in lines:
5                   x = extract x-coordinate from line
6                   y = extract y-coordinate from line
7                   coordinate = create coordinate from x, y
8                   add coordinate to c
9           return c
10  finalCoordinates = reduce coordinates to convex hull coordinates:
11          convexHull = create convex hull from coordinates
12          chGeometry = get convex hull geometry from convexHull
13          finalCoordinates = get array of coordinates from chGeometry
14          return finalCoordinates
15  if finalCoordinates is successful:
16          for each finalCoordinate in finalCoordinates:
17                  write finalCoordinate x, y to outputFile
18          return true
19  else:
20          return false
```

Line 1: Create a text file RDD (lines) from the input file (inputFile) using the location of the input in HDFS. Each row of the input file is a point with x- and y-coordinates.

Line 2: Use a map transformation to create an RDD of Java Topology Suite (JTS) Coordinates (coordinates) by converting each line of the input file into a Coordinate object. Lines 3-9 describe the map function used.

Lines 3-9: Create an array (c) to hold the Coordinate objects. For each line of the input file, extract the x- and y-coordinates and create a JTS Coordinate using these as parameters; then add the Coordinate to the array. Return the array when completed.

Line 10: Use a reduce action to create an RDD of JTS Coordinates (finalCoordinates) by combining the coordinates from Line 2 into the final coordinates that make up the convex hull. This process could involve partitioning the coordinates into smaller sets of coordinates (which can be formed by taking every *n* coordinates in the array either sequentially or randomly), finding the convex hull coordinates of each set, and

combining sets until they form the final set of coordinates that make up the vertices of the convex hull as shown in Figure 3. Lines 11-14 describe the reduce function used.

Lines 11-14: Create a JTS ConvexHull (convexHull) using the coordinates from Line 2. Get the JTS Geometry (chGeometry) from the convex hull by calling the JTS function getConvexHull(). Then get the array of JTS Coordinates (finalCoordinates) that make up the convex hull geometry by calling the JTS function getCoordinates(). Return the array when completed.

Lines 15-20: If the final coordinates that make up the convex hull were successfully obtained (e.g., they are not null and consist of at least two coordinates), then for each coordinate, write its x-y components to the output file (outputFile) using the location of the output in HDFS, and return true. Otherwise, return false.

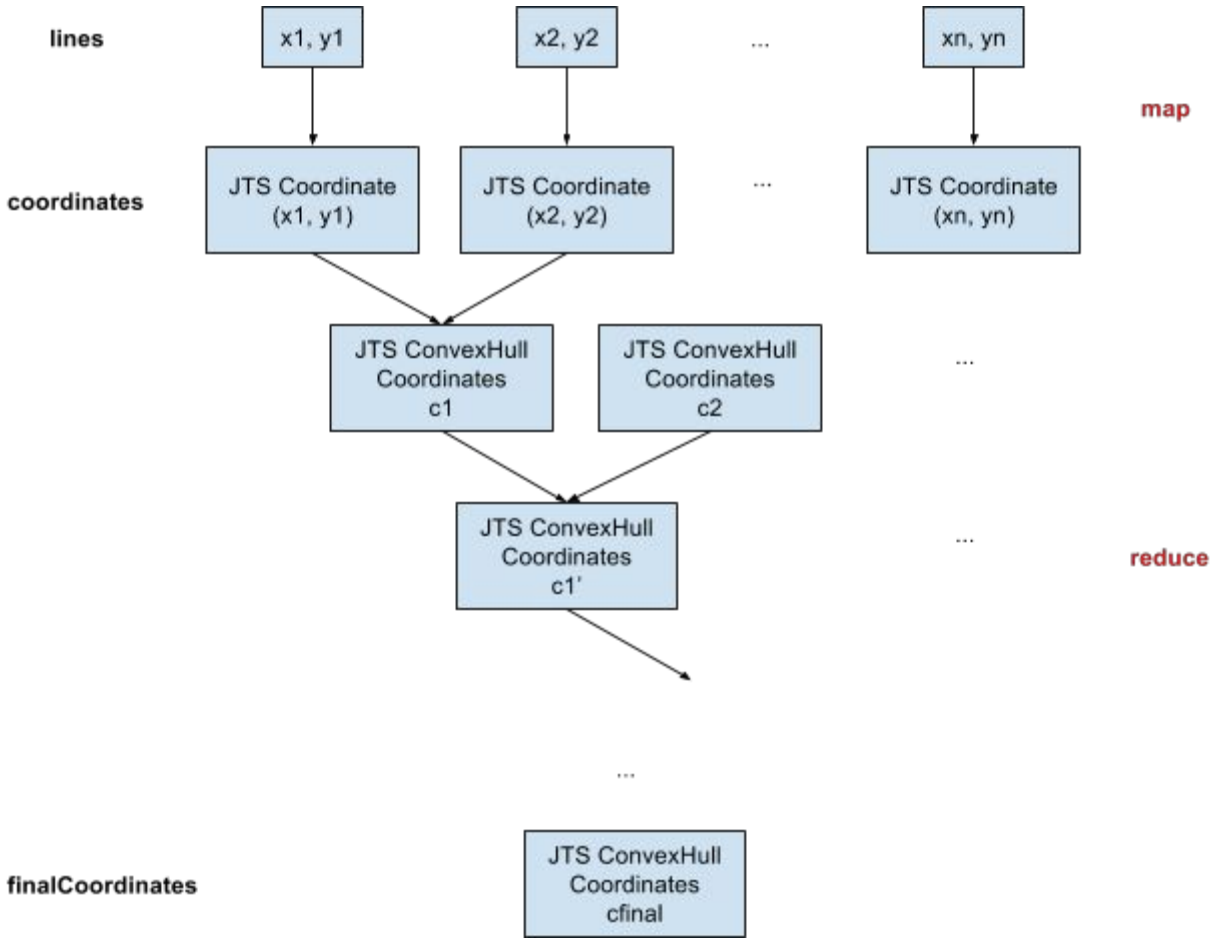


**Figure 3:** Data flow of the Geometry Convex Hull function

## 3. Geometry Farthest Pair

The input of the farthest pair is a set of points on the x-y plane. The output is a pair of points that have the farthest distance among all the input points. Based on an observation that the farthest pair is always from the points on the convex hull, we should be able to use the previous convex hull function to significantly reduce the input size. From the vertices on the convex hull, we can create the cartesian product and reduce to the farthest pair. The following is the pseudocode:

```
// 0. copy local file to HDFS
Utils.copyToHDFS(INPUT_FILE_1, HDFS_ROOT_PATH, LOCAL_PATH);

// 1. read the lines from the input files in HDFS
sc = new JavaSparkContext(conf);
JavaRDD<String> lines = sc.textFile(HDFS_ROOT_PATH + INPUT_FILE_1);

// 2. convert lines to JTS geometries to get points
// String --map--> Geometry

// 3. use Convex hull to reduce the size of the points
points = ConvexHull(points).getPoints();

// 4. cartesian: get the cartesian product of the points
JavaPairRDD<Geometry, Geometry> cartesian = points.cartesian(points);

// 5. calculate the distance and save it by mapping to a pair
// <Geometry, Geometry> -- map --> <<Geometry, Geometry>, Double>

// 6. reduce: to get the resultPair that has the largest distance
// <<Geometry, Geometry>, Double> -- reduce --> <<Geometry, Geometry>, Double>
// where the function return the key-value pair which has the larger value

// 7. output to an HDFS file
Utils.writeToOutputFile(resultPair)
```

The data flow is shown in Figure 4:

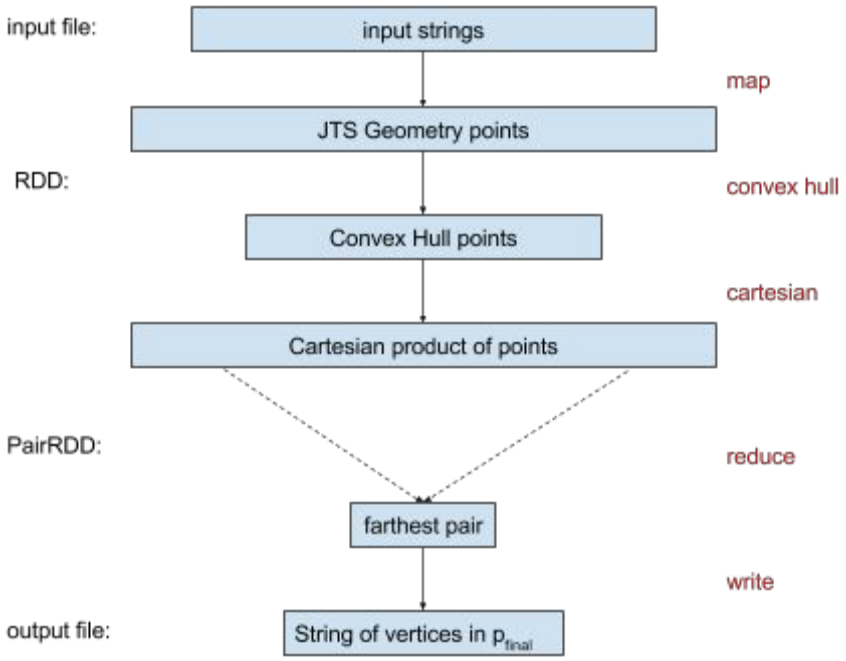


**Figure 4:** Data flow of the Geometry Farthest Pair function

## 4. Geometry Closest Pair

The input of the Closest Pair is a set of points on the x-y plane. The output is a pair of points that has the smallest distance among all the input points. Our focus is to find an efficient way of implementing closest pair of points in the Apache Spark environment. Assume the total number of points is n. The brute-force algorithm runs in $O(n^2)$ time, the more advanced divide-and-conquer algorithm runs in O(nlogn) [2]. We will propose an approach similar to the divide-and-conquer algorithm but in the distributed environment using Spark.

We assume there exists an implementation of the divide-and-conquer algorithm to solve the closest pair problem for the non-distributed, called CP(List<Point>). It takes a set of points and returns the closest pair in O(nlogn) time. The main idea is to divide the input into small pieces that can be fed to CP(). We will define a constant integer CP_ENTRY_SIZE to determine when the input is small enough to feed CP(). The following is the pseudo code:

```
// 1. define a helper function that is recursive
Tuple2<Point, Point> ClosestPairRecursive(Set<Point> points) {
        // 1.1 check if the size of the input is small enough
        if (points.size() < CP_ENTRY_SIZE) return CP(points);

        // 1.2 otherwise divide the input into two parts and get each one's closest pair
        // 1.3 first we need to have the bounding box of the input points
        //  Point -- map --> Point  -- reduce --> BoundingBox

        // 1.4 cut the input in half by x, using rangePartitioner, < BoundBox.width/2
        // points-- range partition --> points1  and points2

        // 1.5 call ClosestPairRecursive on points1 and points2, get the smaller one as cp
        cp1 = ClosestPairRecursive(points1); cp2 = ClosestPairRecursive(points2)
        cp = min(cp1, cp2)

        // 1.6 create a "belt area" along the cut line, similar to the divide-and-conquer
        // any point that is not in this belt area can not form the closest pair
        // points1 -- filter --> pointsInBelt1
        // points2 -- filter --> pointsInBelt2

        // 1.7 call ClosestPairRecursive on the belt points
        cp = min (cp, ClosestPairRecursive(pointsInBelt)

        return cp
}
```

```
// the driver program that uses the recursive function
// 0. copy local file to HDFS
Utils.copyToHDFS(INPUT_FILE_1, HDFS_ROOT_PATH, LOCAL_PATH);

// 1. read the lines from the input files in HDFS
sc = new JavaSparkContext(conf);
JavaRDD<String> lines = sc.textFile(HDFS_ROOT_PATH + INPUT_FILE_1);

// 2. convert lines to JTS geometries to get points
// String --map--> Geometry

// 3. call the ClosestPairRecursive
Tuple2<Point, Point> = ClosestPairRecursive(points)

// 4. output to an HDFS file
Utils.writeToOutputFile(resultPair)
```

At step 1.7, where we make a recursive call on the belt points, we probably need to toggle the orientation of the current cut. The reason is that in the case shown in Figure 5, the belt cannot exclude any point; otherwise, it may cause an infinite loop. The solution is to add a parameter to indicate the current orientation, and toggle it at step 1.7.

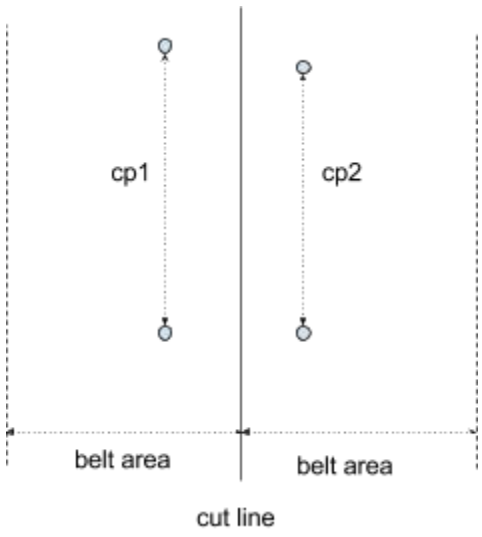


**Figure 5**. The case where the belt area cannot exclude any point

Another potential challenge is that since we have not implemented any recursive function in Spark, it may be difficult or even impossible to achieve our proposed recursive function. A backup plan is to use cartesian product of each pair of points and then reduce it. It is similar to the method used in Farthest Pair. The data flow is shown in the following Figure 6.

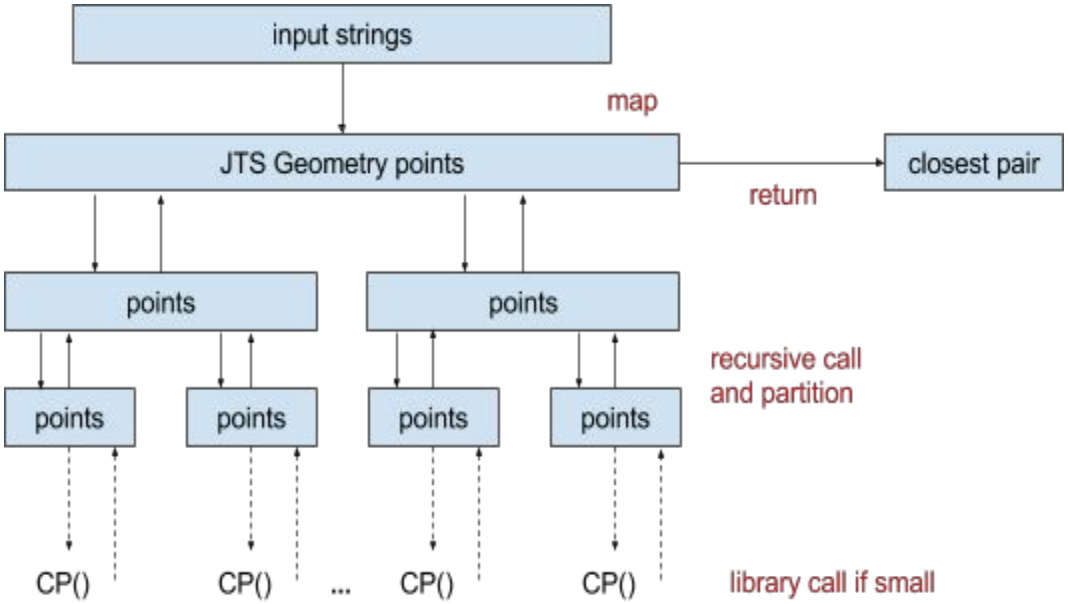


**Figure 6**. Data flow of the Geometry Closest Pair function

## 5. Spatial Range Query

The range query takes two inputs: one is a set of objects and the other one is the query window. It returns a subset of input objects that are fully contained by the query window. The idea to make this process parallelable is to divide the input objects and apply the query on each one of them. The following is the pseudocode and its data flow.

```
// 0. copy local file to HDFS
Utils.copyToHDFS(INPUT_FILE_1, INPUT_FILE_2, HDFS_ROOT_PATH, LOCAL_PATH);

// 1. read the lines from the input file in HDFS
sc = new JavaSparkContext(conf);
JavaRDD<String> lines = sc.textFile(HDFS_ROOT_PATH + INPUT_FILE_1);

// 2. read the query window
final Geometry window = Utils.readQueryWindowFromFile(HDFS_ROOT_PATH + INPUT_FILE_2);

// 3. spatial range query
// 3.1 map: convert each line of string to a polygon
// String -- map --> Geometry

// 3.2 filter: filter to allow the ones only in the query window
// Geometry -- filter --> Geometry

// 4. output to an HDFS file
Utils.writeToOutputFile(rangeResults.collect())
```

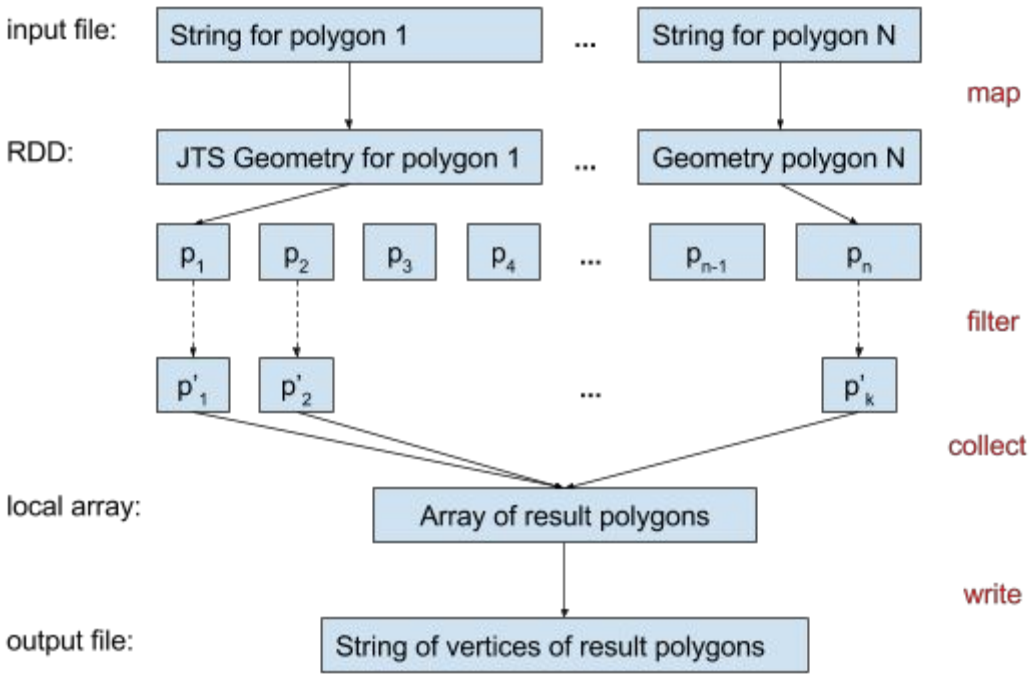


**Figure 7:** Data flow of the Spatial Range Query function

## 6. Spatial Join Query

The input of spatial join query is two sets of rectangles, A and B. Each line of the output starts with a rectangle in A followed by a list of rectangles in B that overlap with the first rectangle.

If we have a modified version of the spatial range query, SpatialRangeQuery2, which uses overlap rather than contains, then for this spatial join query, the first way to implement is to use it as a subroutine to iterate over the second set of rectangles and get the list of overlapped rectangles in the first set of rectangles. The second way is to create a cartesian product of the two sets of rectangles and filter it with an overlap checking function. This is more like a brute force method and likely to work for a relatively small input. The following pseudocode describes it:

```
// 0. copy local file to HDFS
Utils.copyToHDFS(INPUT_FILE_1, INPUT_FILE_2, HDFS_ROOT_PATH, LOCAL_PATH);

// 1. read the lines from the input files in HDFS
sc = new JavaSparkContext(conf);
JavaRDD<String> lines1 = sc.textFile(HDFS_ROOT_PATH + INPUT_FILE_1);
JavaRDD<String> lines2 = sc.textFile(HDFS_ROOT_PATH + INPUT_FILE_2);

// 2. convert lines to JTS geometries to get polygons1 and polygons2
// String --map--> Geometry

// 3. get the bounding boxes for each set of polygons, bb1 and bb2, and their intersections bb
// Geometry --map-> BoundingBox --reduce by merging--> BoundingBox

// 4. use the modified version of Spatial Range Query to reduce the size of input
polygons1 = SpatialRangeQuery2(polygons1, bb)
polygons2 = SpatialRangeQuery2(polygons2, bb)

// 5. cartesian: get the cartesian product of the two sets
JavaPairRDD<Geometry, Geometry> cartesian = polygons1.cartesian(polygons2);

// 6. filter: to get the pairs that overlaps
// <Geometry, Geometry> -- filter --> <Geometry, Geometry>

// 7. reduceByKey: to get the list of polygons for each key polygon
// <Geometry, Geometry> -- reduceByKey --> <Geometry, List<Geometry>>

// 8. output to an HDFS file
Utils.writeToOutputFile(joinResults.collect())
```

An improvement is possible in that we can partition the input into a grid first, and on each part of the grid, the above cartesian method can be performed. The number of rows and columns can be calculated based on the size of the input.

The data flow is shown in Figure 8:

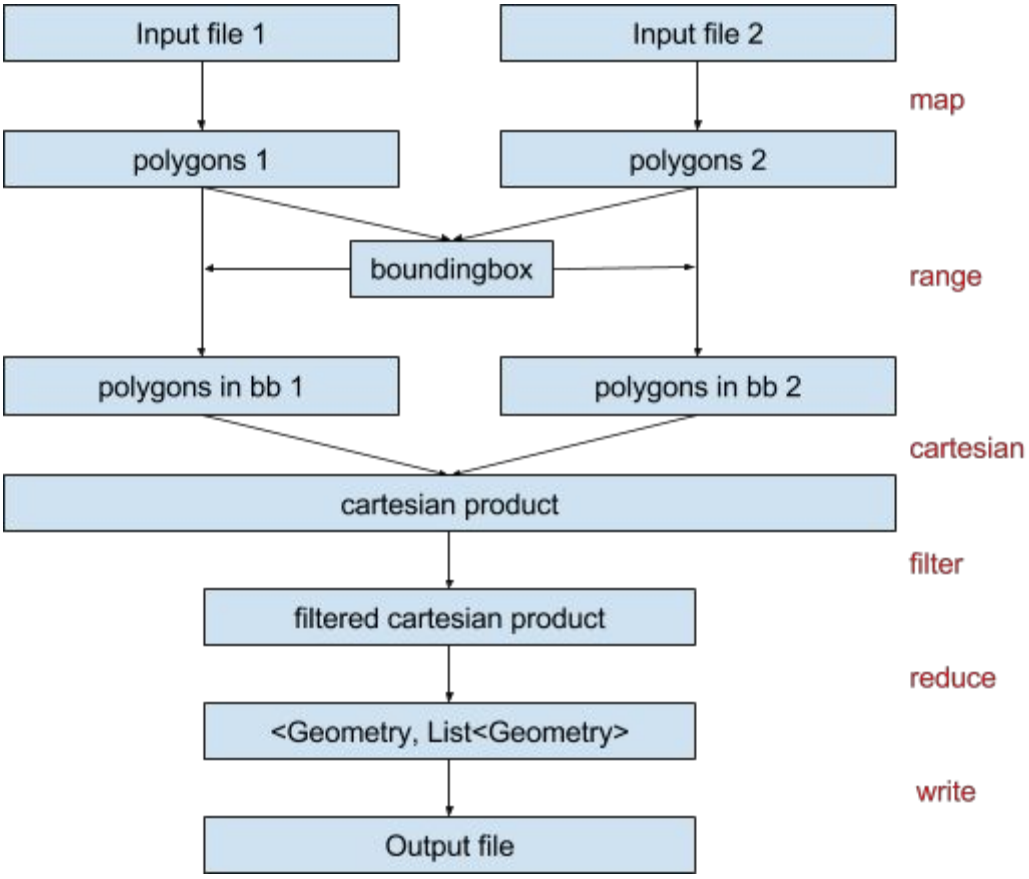


**Figure 8:** Data flow of the Spatial Join Query function

# References

[1] http://lin-ear-th-inking.blogspot.com/2007/11/fast-polygon-merging-in-jts-using.html
[2] http://www.cs.mcgill.ca/~cs251/ClosestPair/ClosestPairDQ.html