Geospatial Operation using Apache Spark

Akash Nishar
School of Computing, Informatics, and Decision Systems Engineering, Arizona State University
Tempe, USA

Aritra Kumar Lahiri
School of Computing, Informatics, and Decision Systems Engineering, Arizona State University
Tempe, USA

Arun Subramanian
School of Computing, Informatics, and Decision Systems Engineering, Arizona State University
Tempe, USA

Milind Jindal
School of Computing, Informatics, and Decision Systems Engineering, Arizona State University
Tempe, USA

Rohit Singh
School of Computing, Informatics, and Decision Systems Engineering, Arizona State University
Tempe, USA

Venkata Sai Girish Konda
School of Computing, Informatics, and Decision Systems Engineering, Arizona State University
Tempe, USA

ABSTRACT

The rapid increase in volume of data in terms of size as well as in terms of location specific applications has led to the emergence of developing the spatial operations for performance and scalability purposes. In this project, seven of the most important spatial operations were designed and implemented, namely polygon union, convex hull, farthest pair of points, closest pair of points, spatial join, spatial range and spatial aggregation. The algorithm is built on Apache Spark framework interfacing with the Hadoop Distributed File System for faster in memory computations and efficient processing of very large datasets across multiple nodes. The following sections describe the implementation algorithms and experimental analysis along with an example of Tweet Heat Map, a real world application which is based on spatial aggregation.

Categories and Subject Descriptors

Spatial Operations, Spatial Join, Spatial Range, Spatial Union, Spatial Farthest Point, Spatial Closest Pair of Points, Convex Hull, Spatial Aggregation (Heat Map).

General Terms

Distributed Systems, Apache Hadoop, Apache Spark, Spatial Queries, Resilient Distributed Dataset, Hadoop Distributed File System.

Keywords

Scala, Java, RDD, HDFS, broadcast variable.

1. INTRODUCTION

This paper contains a detailed implementation plan and experimental evaluation analysis when the above mentioned Geo spatial operations were carried out in a distributed system. Resilient Distributed Datasets (RDDs) were used as because the operations were iterative in nature.

The paper is divided into five sections. The second section throws light on the architecture of Spark and Apache Hadoop as it is very important for our readers to know a bit about Spark and the HDFS file system to evaluate our paper better. The section following it gives enunciates the implementation of each of the Geo-spatial method in great details. And finally the experimental setup and details section provides thorough technical analysis of the commodity hardware used and the out efficiency of each of the above mentioned Geo-spatial operations. An appendix section enunciating the actual values of memory utilization, CPU utilization and network cost have also been added for readers to evaluate our approach better.

2. ARCHITECTURE

2.1 Apache Spark

Apache Spark is an open-source cluster computing framework originally developed in the AMPLab at UC Berkeley. [1] Spark requires a cluster manager and a distributed storage system. For cluster management, Spark supports standalone (native Spark cluster), Hadoop YARN, or Apache MESOS. For distributed storage, Spark can interface with a wide variety, including Hadoop Distributed File System (HDFS), Cassandra, OpenStack Swift, and Amazon S3. Spark also supports a pseudo-distributed local mode, usually

used only for development or testing purposes, where distributed storage is not required and the local file system can be used instead; in this scenario, Spark is running on a single machine with one executor per CPU core.

The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures.

RDDs support two types of operations: **transformations**, which create a new dataset from an existing one, and **actions**, which return a value to the driver program after running a computation on the dataset.

All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently

A second abstraction in Spark is shared variables that can be used in parallel operations. Spark supports two types of shared variables: broadcast variables, which can be used to cache a value in memory on all nodes, and accumulators, which are variables that are only "added" to, such as counters and sums.

2.2 Apache Hadoop

Apache Hadoop is an open source software framework written in Java for processing and storage of very large distributed data over clusters built from commodity hardware. The Apache Hadoop can be broadly differentiated into two parts - storage part, which is the Hadoop Distributed File System (HDFS) and the processing part, which is the Map-Reduce. Hadoop splits the files into a specific sized blocks and distributes them among the nodes present in the cluster. Thus this phenomenon helps data to be processed much faster and efficiently ensuring parallel operation on the large sized data.

The Hadoop architecture consists of the Hadoop common package which provides files and OS level abstractions, the Map Reduce processing engine and the HDFS. The Hadoop Common Package contains the necessary JAR files needed to start the Hadoop.

A small Hadoop cluster contains a single master and multiple worker nodes. Master has the following parts - Job Tracker, Task Tracker, Name Node and Data Node. The worker node does the job of a Data Node and Task Tracker. In a large cluster the HDFS is managed through a Name Node server and a secondary Name Node which has the ability to generate selected portions of the Name Node server. This helps in prevention of loss or damage of data.

The HDFS is a distributed, scalable and portable file system written in Java for Hadoop network. The advantages of HDFS are - a) Ability to store very large data over distributed system across multiple machines. b) High reliability by replication of data over multiple nodes. c) High availability by having backup to the main Name Node Server. d) Data Awareness among the Job Tracker and Task Tracker.

Map Reduce is the processing engine that contains one of the Job Tracker which keeps track of the jobs submitted by client applications. The Job Tracker then delegates the job across the nodes in the cluster through Task Tracker according to their availability.

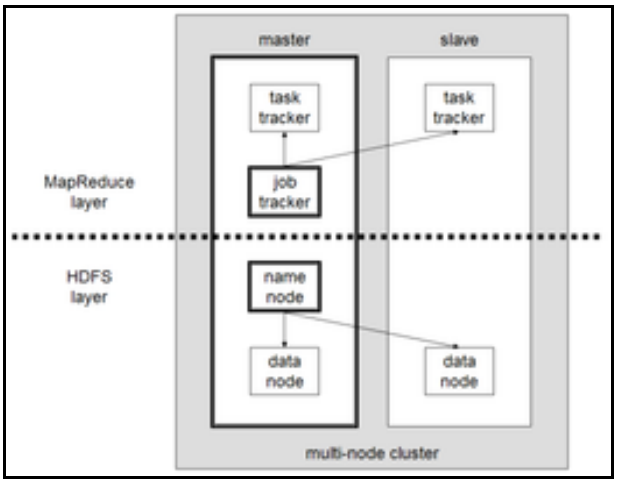


Fig 1 Depicts the Hadoop architecture.

2.3 Spatial Spark

The architecture of Apache Hadoop and Spark were utilized in this project. The data layer was comprised of HDFS and the Native file system of Ubuntu "ext4". The application layer was implemented in the Spark framework using the Java language API. The whole system comprised of 4 machines. The details of those machine is mentioned in the implementation details.

The HDFS was run over the four nodes, with one node running the master i.e. the Name Node daemon. The other three nodes designated as the worker nodes were made the data nodes for this project. The SPARK master was placed on the same machine as the Name Node to reduce any complexity of setting up the architecture. The other three nodes were made the workers for spark for this project.

3. IMPLEMENTATION DETAILS

3.1 Farthest Point

In farthest pair operation, the input provided is set of n points in the form of (x, y) in a rectangle coordinate system and output of the operation is a pair of points which are having the highest distance. The implementation approach for this operation is first find the point lying on convex hull because the farthest pair points all lies on convex hull. This step can be done in O (nlogn) time. Now once points on convex hull are obtained, apply Rotating Calipers algorithm over this points to find the farthest points. This step can be done in O (n) time. So the overall operation can be done in O (nlogn) time. The farthest points operation is an iterative operation and can be done on Apache Spark using Map Partition and Reduce function. Here to find the farthest points, the computation is dependent on all other points, so we will use broadcast variable for storing the information of all points on each partition.

3.2 Closest Pair

Closest Pair operation finds a closest pair of points given a set of points. Input to this operation is a set of points in the Cartesian coordinate system and output is a pair of closest points in the same format. The approach used for this operation is recursive divide and conquer. In this approach, we will first read the input points as tuples into an RDD. The RDD is then sorted using the sortBy function in a distributed fashion. The RDD is then partitioned into left and right partition RDDs based on the median x-coordinate using the filter function. Closest pair in each partition is found recursively using the divide and conquer approach using the map and reduce functions. Closest pair across the partitions is found and compared with the other closest pairs found from each partition and minimum of all the three is returned as an RDD and stored in HDFS. The time complexity of the sorting phase is O (nlogn).There are a total of O (logn) divides and merge takes O (n). So overall time complexity will become O (logn).

3.3 Convex Hull

The Conex Hull was implemented using the Graham Scan's algorithm. The input of co-ordinates were given in the form of a file, which were read into RDDs in the form of tuples per line. The tuple containing the lowest y co-ordinate was filtered out .The obtained RDD was further transformed into another RDD comprising of tuples in form of (x1,y1,r,θ); where x1,y1 are the rectangular co-ordinates and r,θ are the polar co-ordinates of the corresponding rectangular co-ordinate. The theta angle was computed with the point having the lowest y co-ordinate. Graham Scan [9] algorithm was then applied to RDD set. Thus the Convex Hull algorithm was run in a distributed manner and the "local" convex hull was at each node was computed. The results (of this intermediate stage) were gathered into another RDD. Thus points were filtered out and only the points that may constitute the "global" convex hull were kept. Graham Scan was applied again on the reduced set and the final convex hull was obtained. The obtained result was stored again into HDFS in the form of a text file were each line constituted a tuple.

3.4 Polygon Union

In polygon union, the input provided is a set of n polygons in the form of (x1, y1, x2, y2) where (x1, y1) and (x2, y2) are the opposite diagonal points of a rectangle. The output of the operation should be a set of boundary points that forms the union of all n polygons. To implement these operation, we will be using a JTS Topology Suite Java library by Vivid Solutions Inc. The union operation is an iterative operation and can be done on Apache Spark using a Reduce function. Also the union operation is dependent on input of two polygon, so it is easily implemented on a distributed environment.

3.5 Spatial Range Query

In Spatial Range, the input provided is a set of points(x1, y1) and a set of query windows(x1, y1, x2, y2). The values of these variables represent the geometric locations of the polygons present in a geometric space. The windows are taken as the input in an RDD and then stored as a list, which is treated as a broadcast variable. Then with this list, a final filter method is applied where if the point falls in any of the window then that point is kept and "true" Boolean is returned, if the point does not occur in any of the windows then the point is not considered and is discarded by giving a false value as output. The output of the code is written as a text file with all the points that fall in the windows in a text file in HDFS.

3.6 Spatial Join Query

In Spatial Join , the input provided is a set of points (x1,y1,x2,y2) in a window and a set of polygons (px1, px2, py1, py2) where (x1,y1) , (px1,py1) and (x2,y2) and (px2, py2) are the x and y coordinates denoting the latitude and longitude of the window points and the set of polygons. To implement this operation, at first a JavaRDD is taken to get each of the polygons windows. Then a broadcast list variable is created, much the same way as for the spatial range query. This broadcast variable is fed to a pair function. For every polygon present in the JavaRDD for the points, the windows are iterated over and if the polygon lies inside the window, the window is added to another list which is paired with that point and the value is returned in an RDD. The JavaRDD is then stored to a simple text file in HDFS

3.7 Join Aggregation

In spatial aggregation, the input is a set of window polygons (x1, y1, x2, y2) and a set of points (x1, y1). The expected output is for all the windows, a count of the number of points that lie inside that window should be given. For the implementation of this code, first of all the window points were taken as input in an RDD as a broadcast variable, then a flat map was created, which mapped each of the points to give out a mapping of a window and an integer value 1, which indicated that this window has encountered this point once. Once this flat map was created, the output of this had to be created into a simple key value pair. This was achieved through reading the output RDD and creating a string variable as the key, which included all the points of the window polygon appended, separated by underscores. Once these key-value mappings were created, these had to be reduced into simple form through adding the integer values for all the respective keys or polygons. This was done by making a "reduce-by-key" function which only aggregated the integer values for the keys present in the RDD. The output of this code came out as a String key with the value as the number of points that fall inside that particular window. This output is written into a flat file in HDFS.

4. EXPERIMENTAL SETUP

Experiment was perform on a LAN with a capacity of 100Mbps transfer rate. Experiment uses 3 nodes as a worker and 1 node as a master. The hardware configuration on each node is provided below:

Table 1

Node (Master / Worker)	Memory	Processor
Node 1 (M & W)	8 GB	4th Gen, Intel i5
Node 2 (W)	4GB	4th Gen, Intel i5
Node 3 (W)	8GB	4th Gen, Intel i5
Node 4 (W)	8GB	4th Gen, Intel i5

A HDFS cluster is also formed consists of all above nodes for storing of input and output files. A replication factor of 2 is used for partition tolerance. While consistency and availability was handle by HDFS. The experiment uses 4 datasets of 5MB, 50MB, 100MB, and 200MB respectively to perform regression testing on each functions.

5. EXPERIMENTAL DETAILS

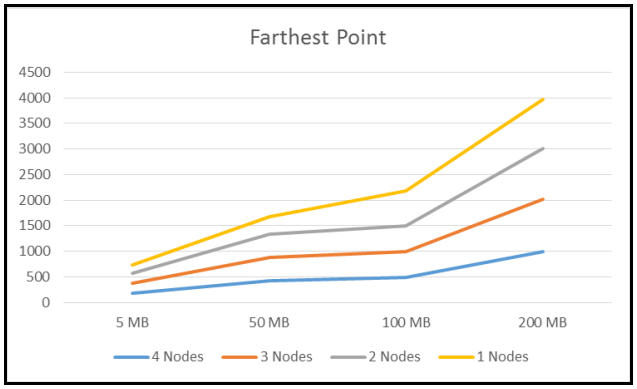
Experiments were conducted using experimental setup parameters and following results are obtained. The following are the time (Millisecond) vs Dataset size (Megabyte) graph for each operation.

The variations done were the number of nodes in the cluster, the size of the data used and the different operations being performed. The time taken for all of the operations was noted and graphs were made out of them.

5.1 Farthest Point

The experimental result for the farthest points operation suggests that as the dataset size increases, the time taken for the operation first increases in O (nlogn) time but at later stage with large dataset it increases in O (n^2) time. Also as we increases number of nodes (i.e. scale out), the time

taken for operation is almost increasing linearly with dataset size.



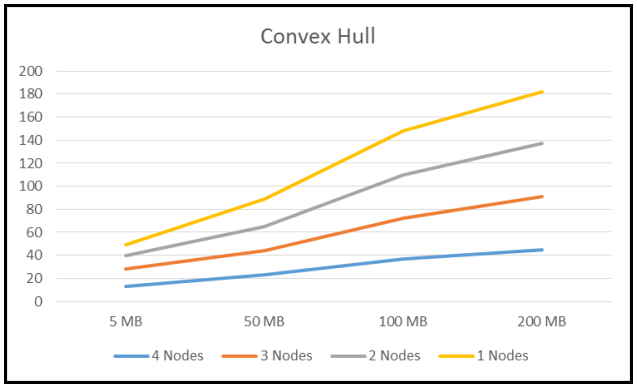
The experimental result for the farthest points operation suggests that as the dataset size increases, the time taken for the operation first increases in O (nlogn) time but at later stage with large dataset it increases in O (n^2) time. Also as we increases number of nodes (i.e. scale out), the time taken for operation is almost increasing linearly with dataset size.

5.2 Closest Pair

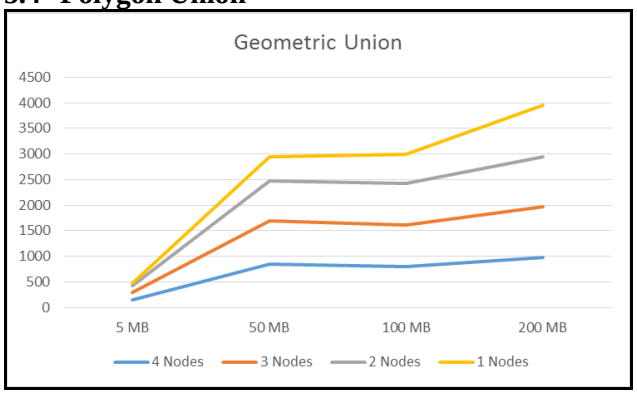
The plot shows that the time taken for closest pair improved with increase in number of nodes. The graph appears impeccable as with increase in number of nodes data gets partitioned across various nodes and operation is parallelized and as operation returns only a pair of points at each recursion, network overhead is not significant.

5.3 Convex Hull

The graph suggests that the time taken for the output of the code for Convex Hull increases with increase in dataset(as expected) but shows a degradation of time taken when the number of nodes is increased (keeping the data size same). The efficiency is non-linear as for smaller datasets the network overhead trumps the computational rescources added. We can thus see that maxium gain has been obtained when the dataset of around 200 MB was considered and was run on cluster of 4 nodes.



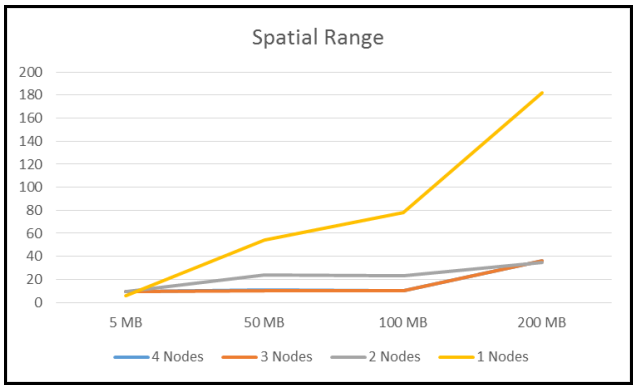
5.4 Polygon Union



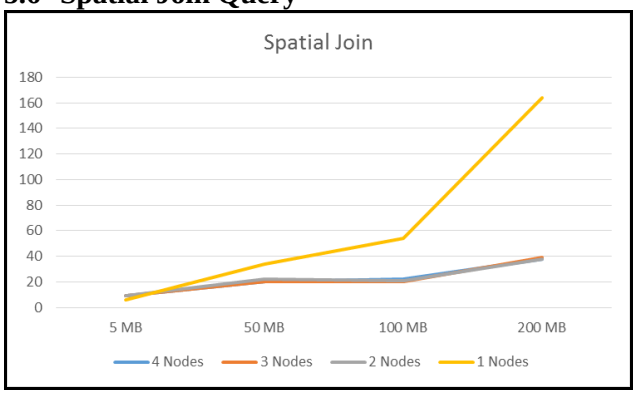
The graph suggests that the time taken for getting the output of union operation increases with increase in the size of the dataset. The time taken for small dataset of 5MB is almost same even when run on multiple nodes as the netwrok cost and communication cost is also added. But with the increase in number of nodes , the time taken for union decreases as the dataset size is increased.We can also observe that as the number of nodes are increased the running time decreses for the given size of dataset.

5.5 Spatial Range Query

The spatial range query is shown to take higher time for single node, which is as expected. The abnormality is when the nodes are increased from 3 to 4. The graphs for 3 nodes and 4 nodes overlap almost exactly on each other. This is because when increasing the number of nodes, the parallelism increases, but the network cost and the communication cost in general also increases, so the system has to spend more time in communication.



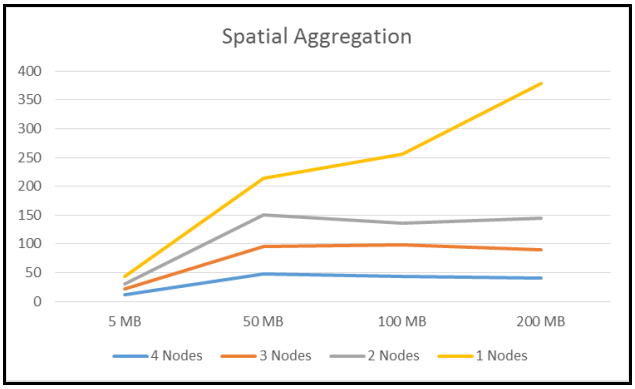
5.6 Spatial Join Query



The graph suggests that the time taken for the output of the code for spatial Join query was similar when tested with 2 , 3 or 4 nodes, this is because of the case that even when the files were copied to the cluster of hadoop, the files got stored only at 2 machines and not other machines, as the replication factor was kept as 2. The graph rightly shows an increased time taken when there was only one node present in the Hadoop cluster, which is quite logical as the amount of distribution to be managed by a single node is much higher with a replication factor of 2. And all the memory utilizations were to be managed by a single machine and not a cluster of machines.

5.7 Spatial Aggregation

The experimental results for the spatial aggregation show that the code takes much less time for a multinode cluster as compared to that for a 1 node cluster. The drop in time is sizeable for a shift from a single node to 2 nodes in a cluster, but the time dips only slightly for movement from 2 nodes to 3 nodes and 3 nodes to 4 nodes. Also it can be observed from the graph that the time remains nearly constant even after increasing the data size frm 5 MB to 200 MB. This shows that the context required to maintain a data size of 5 MB is much more as compared to the data itself, but as the data size increases the context becomes less relevant and is correctly reflected in the graph.



6. CONCLUSION

The spatial operations that were written in spark are complex operations to be written in a distributed manner. The operations if done in a single node system can be easily written. But the overall time complexity of the distributed code becomes relevant only when the data size becomes huge. The data sizes provided for the current project were of 200MB. Which can be easily accommodated in memory in single node code structure. Overall, writing the code for the spatial operations in Spark is much more complex as compared to that written in a single system architecture. Spark is an evolving technology and much work needs to be done over it to make it more robust and flexible for making durable code. The in-memory architecture is a very big advantage in case of the distributed architectures as the hardware is getting inexpensive by the day. Other distributed frameworks such as Map-reduce are much more flexible but have a number of constraints which Spark has tried to overcome. Map-Reduce in particular has seen a lot of enhancements and architecture level upgrades, which has made it more flexible to code and extend.

7. REFERENCES

[1] Wikimedia Foundation, Inc. Apache Spark 2014. http://en.wikipedia.org/wiki/Apache_Spark

[2] The Apache Software Foundation 2015. Spark, Lightning fast cluster computing. https://spark.apache.org/

[3] Databricks: Intro to Apache Spark 2014. http://stanford.edu/~rezab/sparkclass/slides/itas_workshop.pdf

[4] The Apache Software Foundation 2014. Hadoop, Welcome to Apache Hadoop. https://hadoop.apache.org/

[5] Wikimedia Foundation, Inc. 2014. Computational Geometry. http://en.wikipedia.org/wiki/Computational_geometry

[6] Wikimedia Foundation, Inc. 2014. Convex Hull. http://en.wikipedia.org/wiki/Convex_hull

[7] Knoll, Michael G. 2014-2015. Running Hadoop on Ubuntu Linux (Single-Node Cluster). http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/

[8] Knoll, Michael G. 2014-2015. Running Hadoop on Ubuntu Linux (Multi-Node Cluster). http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/

[9] Graham, R.L. (1972). An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. Information Processing Letters 1, 132-133

APPENDIX A:

1. Farthest Point

Number of nodes	**CPU utilization (%)**			**Memory utilization (% of total RAM)**			**Communication Cost (KBps)**			**Runtime (second)**	**Data Size (MB)**
1	24			5.1			10			180	5
1	25			5.2			11			632	50
1	36			6.3			13			1190	100
1	53			7.4			14			3132	200
2	20	25		6.25	12.5		100	110		200	5
2	23	29		7.30	14.5		105	105		410	50
2	33	39		6.85	13.8		105	105		817	100
2	52	60		8.05	19.30		105	105		1417	200
3	20	25	20	6.25	12.5	6.25	105	105	105	150	5
3	23	29	23	7.30	14.5	7.30	105	105	105	390	50
3	33	39	33	6.85	13.8	6.85	105	105	105	689	100
3	52	60	52	8.05	19.30	8.05	105	105	105	945	200

2. Closest Pair

Number of nodes	CPU utilization (%)			Memory utilization (% of total RAM)			Communication Cost (KBps)			Runtime (second)	Data Size (MB)
1	24			5.1			10			190	5
1	25			5.2			11			642	50
1	36			6.3			13			1195	100
1	53			7.4			14			3150	200
2	25	29		4.7	9.2		105	110		205	5
2	27	31		6.1	11.3		105	105		454	50
2	32	37		7.2	13.1		105	105		793	100
2	45	52		9.2	18.4		105	105		1022	200
3	20	25	20	6.25	12.5	6.25	105	105	105	212	5
3	23	29	23	7.30	14.5	7.30	105	105	105	403	50
3	33	39	33	6.85	13.8	6.85	105	105	105	650	100
3	52	60	52	8.05	19.30	8.05	105	105	105	967	200

3. Convex Hull

Number of nodes	CPU utilization (%)			Memory utilization (% of total RAM)			Communication Cost (KBps)			Runtime (second)	Data Size (MB)
1	15			4.3			10			134	5
1	25			7.1			10			502	50
1	29			8.2			11			980	100
1	37			9.1			11			1603	200
2	19	36		4.3	9.9		90	85		156	5
2	23	39		7.6	8.3		1200	790		360	50
2	24	45		8.3	8.2		3867	968		750	100
2	27	54		9.1	9.3		4722	1512		1350	200
3	19	32	22	5.1	4.9	9.7	100	110	180	145	5
3	25	32	22	7.1	15.9	7.2	2500	620	500	280	100
3	28	35	25	8.2	17.8	8.3	3400	2700	2560	560	50
3	31	37	29	9.1	19.5	8.5	3800	2900	2800	985	200

4. Polygon Union

Number of nodes	CPU utilization (%)			Memory utilization (% of total RAM)			Communication Cost (KBps)			Runtime (second)	Data Size (MB)
1	24			5.1			10			180	5
1	25			5.2			14			632	50
1	36			6.3			12			1190	100
1	53			7.4			11			3132	200
2	25	228		5.0	9.2		1052	1010		187	5
2	27	33		6.5	11.3		1045	1005		520	50
2	32	39		7.7	13.1		1100	1050		989	100
2	45	52		9.9	18.4		1120	1070		2096	200
3	20	25	20	6.25	10.5	6.25	1052	1010	1048	192	5
3	23	29	23	7.30	13.5	7.30	1045	1005	1040	455	50
3	33	39	33	6.85	13.8	6.85	1100	1050	1090	876	100
3	52	60	52	8.05	19.30	8.05	1120	1070	1120	1654	200

5. Spatial Range Query

Number of nodes	CPU utilization (%)			Memory utilization (% of total RAM)			Communication Cost (KBps)			Runtime (second)	Data Size (MB)
1	24			5.1			10			50	5
1	29			5.2			11			240	50
1	36			6.3			13			560	100
1	45			7.4			14			1250	200
2	25	29		4.7	9.2		105	110		45	5
2	27	31		6.1	11.3		105	105		225	50
2	32	37		7.2	13.1		105	105		480	100
2	45	52		9.2	18.4		105	105		996	200
3	20	25	24	5.25	11.5	5.25	105	105	105	50	5
3	23	29	22	6.60	11.5	6.60	105	105	105	230	50
3	33	39	35	6.85	13.8	6.85	105	105	105	390	100
3	52	60	52	8.05	19.30	8.05	105	105	105	680	200

6. Spatial Join Query

Number of nodes	CPU utilization (%)			Memory utilization (% of total RAM)			Communication Cost (KBps)			Runtime (second)	Data Size (MB)
1	12			5.5			10			65	5
1	24			6.4			11			280	50
1	29			7.8			13			650	100
1	35			8.1			14			1420	200
2	18	20		5.4	4.5		105	110		55	5
2	12	15		6.6	7.0		150	150		250	50
2	27	25		7.5	8.0		150	150		590	100
2	40	38		8.7	9.6		150	150		1350	200
3	20	25	20	5.0	12.5	6.25	150	150	150	60	5
3	23	29	23	7.3	14.5	7.30	150	150	150	220	50
3	33	39	33	8.4	13.8	6.85	150	150	150	420	100
3	50	55	48	8.6	19.30	8.05	150	150	150	940	200

7. Spatial Aggregation

Number of nodes	CPU utilization (%)			Memory utilization (% of total RAM)			Communication Cost (KBps)			Runtime (second)	Data Size (MB)
1	24			5.1			10			80	5
1	25			5.2			11			280	50
1	36			6.3			13			650	100
1	53			7.4			14			1420	200
2	25	29		4.7	5.5		150	150		85	5
2	27	31		6.1	5.0		150	150		250	50
2	32	37		7.2	6.9		150	150		590	100
2	45	52		9.2	10.1		150	150		1350	200
3	20	25	20	60	12.5	6.25	150	150	150	75	5
3	23	29	23	220	14.5	7.30	150	150	150	220	50
3	33	39	33	420	13.8	6.85	150	150	150	420	100
3	52	60	52	940	19.30	8.05	150	150	150	940	200