# A NOVEL APPROACH OF INDEXING AND RETRIEVING SPATIAL POLYGONS FOR EFFICIENT SPATIAL REGION QUERIES

J.H. Zhao [a,b], X.Z. Wang [a, *], F.Y. Wang [a,b], Z.H. Shen [a], Y.C. Zhou [a] , Y.L. Wang [c]

[a] Computer Network Information Center, Chinese Academy of Sciences, Beijing, China 100190 - (zjh, wxz, wfy, bluejoe zyc )@cnic.cn
[b] University of Chinese Academy of Sciences, Beijing 100049
[c] Lawrence University, Appleton, Wisconsin, United States 54911 - wangyo@lawrence.edu

**KEY WORDS:** Spatial Region Query, DKD-Tree, Spatial Polygon Indexing, Spark, Retrieval Efficiency

**ABSTRACT:**

Spatial region queries are more and more widely used in web-based applications. Mechanisms to provide efficient query processing over geospatial data are essential. However, due to the massive geospatial data volume, heavy geometric computation, and high access concurrency, it is difficult to get response in real time. Spatial indexes are usually used in this situation. In this paper, based on k-d tree, we introduce a distributed KD-Tree (DKD-Tree) suitbable for polygon data, and a two-step query algorithm. The spatial index construction is recursive and iterative, and the query is an in memory process. Both the index and query methods can be processed in parallel, and are implemented based on HDFS, Spark and Redis. Experiments on a large volume of Remote Sensing images metadata have been carried out, and the advantages of our method are investigated by comparing with spatial region queries executed on PostgreSQL and PostGIS. Results show that our approach not only greatly improves the efficiency of spatial region query, but also has good scalability, Moreover, the two-step spatial range query algorithm can also save cluster resources to support a large number of concurrent queries. Therefore, this method is very useful when building large geographic information systems.

## 1. INTRODUCTION

Spatial data are generated via specialized sensors with GPS devices recording geographical coordinates. They are usually divided into two types: raster data (Remote Sensing images) and vector data (points, lines, polygons). By linking attributes data with spatial extents, spatial data are of extreme value in many fields (Giuliani, Ray, & Lehmann, 2011), such as mining land-use patterns, studying the impacts of the environment (e.g., weather, climate, biological diversity), and traffic management, and so on(Shekhar, Evans, Gunturi, & Yang, 2012). With the great development of Remote Sensing techniques and wide spread of mobile devices equipped with GPS, spatial data are no longer generated only by official land surveying offices and commercial companies, but also by citizens. Nearly continuous streams of all kinds of spatial data are generated every day(Ma, Wang, Liu, & Ranjan, 2015). NASA generates about 5 TB of data per day, and Google generates 25PB of data per day, of which spatial data account for a significant portion (Vatsavai et al., 2012). Plus the Web 2.0 technology enabling citizens to geo-tag all information and disseminate them, a big spatial data era has already come.

Big spatial data provide us with more detailed information in terms of high spatial and temporal resolution(Shekhar et al., 2012), and open up many new applications, such as fine-grained classification, damage assessments, and near real-time environment monitoring. However, the explosive growth in volume, velocity and variety of spatial data make it hard to manage and process with reasonable effort. In order to explore the data effectively, efficient spatial query is necessary (Yu, Jinxuan, & Mohamed, 2015).

### 1.1 Spatial Region Query

Spatial query is one of the fundamental operations in spatial data applications. There are three types of traditional spatial queries. The first is point query, which finds all objects that contain a given point. The second type is a geometric join query. The third type is region query, which finds out objects that intersect or covered by a given region (Samet, 1990). The common way of region query is to firstly define a polygon in the coordinate space, and then inquiries about certain spatial objects that are contained in, or overlap the query polygon (e.g., Return all lakes in Qinghai-Tibetan Plateau) (Larson & Frontiera, 2004). In order to handle such spatial queries, the records of all spatial element need to be checked by geometric computations. As each spatial polygon contains hundreds of points, common computational geometry algorithms used for verifying intersection of polygon pairs are polynomial complexity (Aji & Wang, 2012). Moreover, the increase in the amount of spatial polygons increases the region query processing time. Therefore, spatial region queries are compute-intensive and time consuming, especially with large datasets. And optimization of region query becomes an important issue (Lupa & Pi, 2014).

### 1.2 Spatial Index for Polygons

The speed of spatial region queries is closely correlated with the data structure to organize the polygon data. As the exact boundaries of spatial polygons bring heavy geometric computations, they are only accessed when there is a need for precise calculation. In region query filed, the general processing method is a filter-and-refine approach. It reduces unnecessary computation and I/O cost by employing approximations of the polygons, such as the minimum boundary rectangles (MBRs). The general processing steps are as follows. First, spatial objects are filtered with MBR to eliminate those whose MBRs

* Corresponding author

are not intersect with the query polygon or not covered by it. Next, remaining candidate polygons from the filtering step are further refined with accurate geometry computation. Finally, objects which satisfy the query conditions are selected out for further processing such as calculating areas and perimeters. In addition, spatial index is a key component for enabling efficient range queries and rapid response for users. Many common spatial indexes have been adopted, such as such as grid file, KD-Tree (Bentley, 1975), R-Tree (Guttman, 1984), Quad-Tree (Samet, 1984), and their variants (Procopiuc, Agarwal, Arge, & Vitter, 2003). The basic idea is to partition data in all kinds of ways. For example, in quadtree, the two-dimensional space is recursively divided into four quadrants; In R-Tree, nearby spatial objects are grouped together and are put in different levels of the tree.

However, as the volume of available spatial data increases tremendously, combined with the complexity of spatial queries, there are great challenges on effective region queries. Firstly, the distribution of the whole dataset is difficult to obtain. With various kinds of distribution of spatial polygons, one index method is not suitable. Secondly, with huge datasets, depth of the index tree will be large, or the index file can be large. Moreover, calculating all intersected spatial polygons in a spatial region query is not only time-consuming, but also useless. In this paper, based on k-d tree, we introduce a distributed KD-Tree (DKD-Tree) which is suitbable for polygon data, and a two-step query algorithm. The spatial index construction is recursive and iterative, and the query is an in memory process. Both the index and query methods can be processed in parallel. At last, they are implemented based on HDFS, Spark parallel computing framework and Redis memory cache.

The rest of this paper is organized as follows. After a description of related work in Section 2, we give a detailed illustration of our method including the construction and parallelization implementation of the proposed index data structure, and the two-step query algorithm in Section 3. Then in the following section, we present our experiments and give an analysis and a discussion of our experiment results. In Section 5, we conclude our work and provide an outlook of the future work.

## 2. RELATED WORK

### 2.1 Parallel and Distributed Spatial Data Indexing

Spatial oriented queries involve heavy geometric computations for spatial filtering and measurements, and require high performance computations to support fast response of queries (Aji et al., 2012). Query parallelism is a significant issue of query processing. Parallel R-tree (Kamel & Faloutsos, 1992) is designed for shared-disk environments. However, the spatial indexes only improve data retrieval efficiency, and they are regardless of I/O throughput and spatial computation capability. Thus, they cannot achieve efficient spatial query processing that involves massive spatial data and concurrent users. There are also some approaches that have been designed for parallel databases and cluster systems. Apostolos studied the index parallel batch loading algorithm in parallel spatial database (Papadopoulos & Manolopoulos, 2003). The Paradise project of Wisconsin University runs on a parallel database (Akdogan, Demiryurek, Banaei-Kashani, & Shahabi, 2010). Guo Jing proposed H2R-tree, which was built on spatial grid and Hilbert curve (Jing, Guangjun, Xurong, & Lei, 2006). However, these are only suitable to the systems which have limited parallel processing capability.

Hadoop is a series of technology for distributed storage and processing of big data. The core of Hadoop consists of two parts, a storage part Hadoop Distributed File System (HDFS) and a processing part MapReduce (Dean & Ghemawat, 2004). HDFS enhances the I/O performance of data storage and ensures that the distributed storage system is scalable, fault tolerant. MapReduce is a programming model and an associated implementation for processing massive amount of data sets. Hadoop has achieved an unprecedented success in implementing many real-world distributed tasks. There are some researches focus on spatial data storage and query under Hadoop framework, such as SpatialHadoop (Eldawy & Mokbel, 2015), Hadoop-GIS (Aji et al., 2013), SciHive (Geng, Huang, Zhu, Ruan, & Yang, 2013), GeoMesa (Fox, Eichelberger, Hughes, & Lyon, 2013), Terry Fly (Cary, Yesha, Adjouadi, & Rishe, 2010), and GeoBase (Li, Hu, Schnase, & Duffy, 2016), and so on. However, the coupling among MapReduce model and HDFS is high. And it often needs a large number of reads and writes from HDFS. Thus, it is not suitable for applications with iterative computation, and it is hard to achieve real-time query response.

Spark is an in memory based computing framework, which can effectively avoid high frequency of read and write operations from HDFS. It does not need to save the intermediate results to HDFS. Meanwhile it retains the excellent properties of MapReduce. A Resilient Distributed Dataset (RDD) (Zaharia, Chowdhury, Das, & Dave, 2012) is a basic abstraction in Spark. It is a logical collection of data partitioned across machines and can be rebuilt if a partition is lost. An RDD can be explicitly cached in memory across machines and reused for later MapReduce-like parallel operations. For the algorithms with iterative computation, the intermediate RDD data sets do not need to read and to write from the hard disk at each iterative manner. This is one of the major reasons why Spark works faster. Besides, Spark has an advanced DAG execution engine that supports in memory computing. So Spark is a fast and general engine for large scale data processing which is suited for iterative computation. Considering the above advantages of Spark, we combine Spark in our method and employ Spark for implementation in the paper.

### 2.2 Tree Index for Spatial Region Query

Quadtree is one of the earliest structure used to index spatial polygons. Because quadtree nodes that intersect the query polygon may contain many polygons which do not intersect with the query polygon, a lot of intersection computation are needed on each polygon. As a result, the efficiency of quadtree is not high. The most common improvement method is using a more compact representation of each polygon. So a number variants of quedtree for polygon index appears, among which the most popular one is the R-Tree. In R-Tree data structure, each node in the tree represents the smallest rectangle that encloses its son nodes. As there are no definite node splitting method, when applying R-Tree index, the polygon data has to be understood first.

Another spatial index is the k-d tree (or multidimensional binary search tree) where k is the dimensionality of the search space. It was firstly described in 1975 by Bentley in a theoretical paper (Bentley, 1975). It is a data structure to store data points to be retrieved by associative searches. A significant

advantage of this structure is that a single data structure can handle many types of queries very efficiently. If data are represented as a k-d tree, then each record is stored as a node in the tree, and each node is associated with a discriminator. For k keys, there are k discriminators, which are defined as DISC[0], DISC[1], … , DISC[K-1] respectively. All nodes on any given level of the tree have the same discriminator. The root node has discriminator DISC[0], its two sons have discriminator DISC[1], and so on to the kth level on which the discriminator is DISC[k-1]; the (k+1)th level has discriminator DISC[0], and the cycle repeats.

The construction of k-d tree is as follows: Start with a large rectangular region that contains all the data points to be put into the tree. This large rectangle represents the root of the k-d tree (Rosenberg, 1985). Then by comparing with discriminators, the multi-dimensional space is split recursively into subspaces. All these subspaces are organized in a systematic manner as a search tree. And it is terminated at the maximum depth or when the data associated with each node is sufficiently small.

K-d tree is a superior serial data structure. However, it is unsuitable for indexing non-zero size objects such as lines and polygons. Moreover, k-d tree is hard to be implemented in parallel (Samet, 1990). So in this paper, the DKD-tree which not only provides efficient retrieval of polygons but also runs in distributed environment are presented.

## 3. METHODOLOGIES

### 3.1 DKD-Tree Index

**3.1.1 DKD-Tree construction principle**: DKD tree is substantially a two-dimensional search tree. Branching on odd levels is done with respect to the first key, and branching on even levels is done with respect to the second key. The root is arbitrarily chosen to be an odd level. Spatial polygon data has two dimensions which are longitude and latitude. Here we use X and Y to represent longitude and latitude coordinates. And X and Y are the two keys to split the DKD tree.

The principle of constructing DKD Tree is as follows:

1) Construct the root node.

The root node corresponds to the MBR containing all polygons

2) Chose the first key to split the root node.

Calculate the minimum longitude (minX), minimum latitude (minY), maximum longitude (maxX) and maximum latitude (maxY) for each polygon, and the total number N of all polygons. Then, calculate variance for maxX and maxY ($Var_{maxX}$, $Var_{maxY}$) according to Equations 1 to Equation 4. The dimension with larger variance values is chosen as the first key to split the root node, and the other dimension is the second key. In this paper, we assume that X is the dimension with larger variance value of MaxX.

$$Mean_{max\ Y} = \frac{\sum_{i}^{N} maxY}{N} \quad (1)$$

$$Mean_{max\ X} = \frac{\sum_{i}^{N} maxX}{N} \quad (2)$$

$$Var_{max\ X} = \frac{\sum_{i}^{N} (maxX - Mean_{max\ X})^2}{N} \quad (3)$$

$$Var_{max\ Y} = \frac{\sum_{i}^{N} (maxY - Mean_{maxY})^2}{N} \quad (4)$$

$$Max_{dim} = \max(Var_{max\ X}, Var_{max\ Y}) \quad (5)$$

3) Split the tree into three sub trees by the split point.

In order to build a balanced tree, compute the medium value of selected dimension as the split point. Use the medium value to divide the node to three sub-nodes. Suppose the selected dimension is X. The three sub-nodes are calculated as following:

Left sub-node consist of polygons whose maxX are less than the calculated medium.

Right sub-node consist of polygons whose minX is greater than the calculated medium.

Medium sub-node consist of polygons that are neither contained in the Left sub-node and the Right sub-node.

4) Split the tree recursively by cycling through the two keys at each level.

In this case, Y is used as the split key for the next level of the tree, and then X again.

5) Stop splitting the tree until the number of polygons contained in the current node is less than a given threshold or the depth of the tree is larger than a given threshold. At this time the current node is a leaf node.

The partitioning process is described in Figure1 and Figure 2 is the corresponding tree structure.

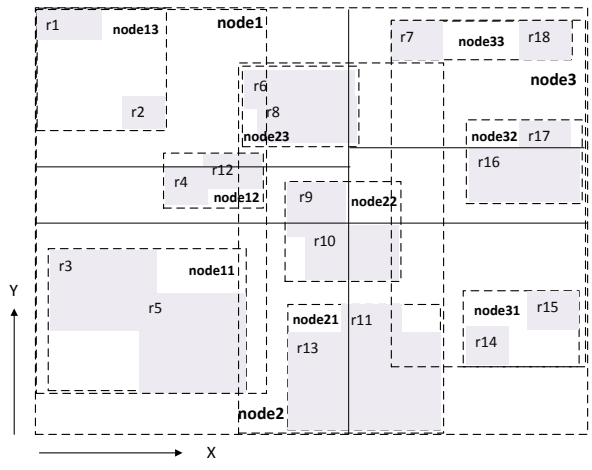


Figure 1. Spatial polygons partition process

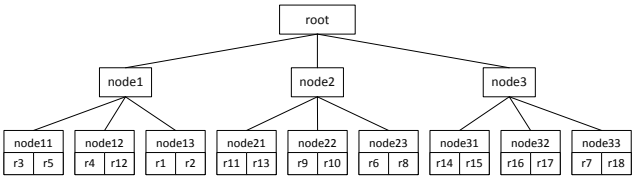


Figure 2. the DKD Tree and the records it represents

As it is shown in figure 1, rectangles marked from r1 to r18 with grey background color are MBRs of 18 spatial polygons. MBR is an expression of the maximum extents of a 2-dimensional spatial object, and is frequently used as an indication of the general position of spatial object (Zhong et al., 2012). The

biggest rectangle is the MBR of the root node. Here we are assuming that the latitude (X axis) has greater variance. So the split point is the medium of the longitude of these 18 MBRs. Therefore, the splitting plane is determined by the solid line which passes through the right margin of r9 and perpendicular to X axis. The splitting plane divides the root to three sub-node. Node1 consists of polygons which are allocated at the left of the splitting plane, including r1, r2, r3, r4, r5 and r12. Node3 consists of polygons on the right of the splitting plane, which are r7, r14, r15, r16, r17, r18. Node2 consists of the polygons which intersect with the splitting plane. They are r6, r8, r9, r10, r11, r13. The partition process for node1, node2, node3 is the same with the root node, except that the splitting plane is the medium of the latitude and it is perpendicular to Y axis. The final data structure for the 18 polygons is shown in figure 2. If the search reaches a node with coordinate values (e, f) and DKD_COMPARE(query polygon, (e, f))=”RIGHT”, then there is no need to examine any nodes in the left subtree of (e, f). By avoiding examining many nodes, the DKD-Tree data structure serves as a pruning device on the amount of search required.

**3.1.2 Building DKD Tree with Spark**: The detailed DKD-Tree construction is as following:

1) Construct root node of DKD-Tree. The features of the root node include the number of all polygons N, MBR of each polygon, $Var_{maxX}$ and $Var_{maxY}$.
2) Save the root node to a queue named as rootQueue. Then pop a node from the top of rootQueue as currentRootNode.
3) Save currentRootNode to a queue named as treeQueue and construct a subtree for currentRootNode.
4) Pop a node from the top of treeQueue as currentNode. There are three cases for the processing of the node：
   a) If the number of polygons of the current node is less than the pre-defined minimum threshold, stop to divide the node.
   b) If the depth of the current tree is greater than the pre-defined maximum threshold, stop to divide the node. Serialize the current tree to HDFS. And add the node into *rootQueue*.
   c) Otherwise, split the node into three sub-nodes according to the method described in section 3.1. And add these sub-nodes into *treeQueue*.
5) Save the data of all the leaf nodes to HDFS as a document and allocate a unique regionID as the name of the document. HDFS creates one block for each file, and file blocks are distributed across cluster nodes for load balancing (Zhong et al., 2012).
6) Repeat step 4 and 5 until the treeQueue is empty.

The pseudo-code for the parallel construction of DKD-Tree on Spark is given in algorithm1.

**Algorithm 1 the Parallel Construction of DKD-Tree on Spark**

**Input**：(sc:SparkContext, inPath:String, outputParts:String, outputTree:String, maxCount:Long, maxDepth:Int)
**Output**：(dkdtree,partitions)

```
// build of the dkdtree
1：function buildDKDTree (sc, inPath, outputParts, outputTree, maxCount, maxDepth)
2：    linesRDD= sc.textFile(inPath) // read all the files from hdfs
3：          .map({
4：              parseLine() // parse each row of data
5：           }).cache // cache data to memory
6：    rootNode = buildNode(linesRDD)
7:     Initialize the queue rootQueue
8：    rootQueue.enqueue(rootNode)
9：    while rootQueue.size != 0 do
10：       rootNode = rootQueue.dequeue()
11：       buildSubTree (rootNode, rootQueue, maxCount, maxDepth)
12：       savePartitions(outputParts) // save partitions data into HDFS
13：       saveSubTree(outputTree) // save sub-tree data into HDFS
14：   end while
15：  end function

// build of the sub-tree
1：function buildSubTree(rootNode, rootQueue, maxCount, maxDepth)
2：    Initialize the queue currentTreeQueue
3：    currentTreeQueue.enqueue(rootNode)
4：    while currentTreeQueue.size != 0 do
5：       node = currentTreeQueue.dequeue()
6：       if node.count > maxCount then
7:           The current node is split into three children nodes, save to the set
8:           for childrenNode in set do
9:              if childrenNode.depth == maxDepth then
10:                rootQueue.enqueue(childrenNode)
11:             else
12:                currentTreeQueue.enqueue(childrenNode)
13:             end if
14:          end for
15：      end if
16：   end while
17：  end function
```

### 3.2 Region Query Method with DKD-Tree

The region query algorithm based on DKD tree we propose consists of two phases. The first phase is named as Counting Query, during which the number of all the polygons intersecting with the given region and the distribution of the results are computed. In the second phase, only several polygons are returned first. The polygons returned are determined by the whole results set and the number of pages displaying query results. This phase is called Paging Query.

**3.2.1 Counting Query:** In this subsection, we will give a detailed process of Counting Query.

1) Read all the subtrees using the strategy of inorder traversal from HDFS and deserialize them to all the cluster nodes.
2) Perform the query in parallel on different nodes.
   a) If the given polygon for the query is completely covered by the MBR of a tree node, it will not

need to perform intersect calculations. For a leaf node, the (regionID, count) of the node is returned. And for a none-leaf node, (regionID, count) pairs of all leaf nodes belonging to the current none-leaf node are returned.

b) If the given query polygon is partly intersect with the MBR of a tree node, perform intersect calculation. For a leaf node, read the data files with corresponding name regionID from HDFS and calculate the number of polygons intersecting the query polygon. For a none-leaf node perform intersect estimation recursively for all its sub-nodes.

3) Collect the results obtained in (2), and sort the results by regionID. Save the results as a resultList: [(regionID1,count1),(regionID2,count2),……]. Then cache the results to Redis in the form of (query polygon, resultList).

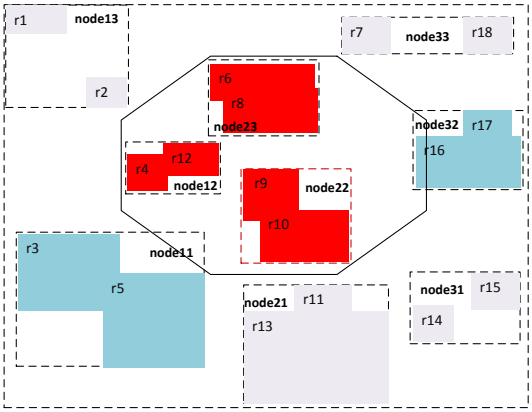


Figure 3. Illustration of the Counting Query process

Figure 3 is an illustration of the Counting Query process. In Fig. 3, the polygon of black solid line is the given query polygon. Polygons r1 to Polygon r18 are to be retrieved. First, execute query operation on all subtrees and find out polygons intersecting the query region. MBRs of node12, node22 and node23 are completely covered by the query polygon. Therefore, there is no need to perform intersect calculation for these three nodes. MBRs of Node11 and node32 intersect with the query polygon. So the intersect calculation should be performed for these two nodes. Say the naming rules of regionID for each node is "id_"+nodename, in this case the results cached to Redis is (query polygon, [(id_node12,2), (id_node22,2),(id_node23,2),(id_node32,2)].

**3.2.2 Paging Query:** When the volume of data to be retrieved is huge, the general method to display the results is via paging. Thus, after the total number of intersected polygons and the regionIDs of leaf nodes containing intersected polygons are calculated, the Paging Query process is performed.

1) Read the cached results (query polygon, resultList) from Redis.

2) Calculate the total number of retrieved polygons by formula 6.

$$totalNum = \sum_{i=1}^{N} counti \qquad (6)$$

Where N is the length of the resultList.

3) Give the pageNum (a page number) and pageSize (the number of records that can be displayed in one page). The pageNum should be smaller that [totalNum/pageSize]+1.

4) Compute the range of regionIDs of leaf nodes where the data to be displayed in the pageNum-th page are stored in. The formulas are as follows:

$$\text{regionID}_{Min} = mim(P), P\ satisfy\ \frac{count_1 + count_2 + \cdots + count_P}{pageSize} \geq pageNum - 1 \qquad (7)$$

$$\text{regionID}_{Max} = mim(Q), Q\ satisfy\ \frac{count_1 + count_2 + \cdots + count_Q}{pageSize} \geq pageNum \qquad (8)$$

Where $(P, P+1, \ldots, Q-1, Q), P \leq Q$ are the regionIDs of the leaf nodes where data to be displayed in the current pager are stored in.

5) Calculate polygons intersecting query polygon in leaf nodes of $(P, P+1, \ldots, Q-1, Q)$ in parallel.

6) Sort all polygons intersecting query polygon in leaf nodes of $(P, P+1, \ldots, Q-1, Q)$.

7) For nodes P and Q at both ends of the regionIDs range, only some of polygons that intersect the query polygon are returned. For leaf node P, return the last Pr retrieved polygons. And for leaf node Q, return the first Qt retrieved polygons. Formulas to calculate Pr and Qt are as follows:

$$Pr = (count_1 + count_2 + \cdots + count_P) - pageSize \times (pageNum - 1) \qquad (9)$$

$$Qt = pageSize \times pageNum - (count_1 + count_2 + \cdots + count_{Q-2} + count_{Q-1}) \qquad (10)$$

8) All retrieved polygons in leaf nodes of $(P+1, P+2, \ldots, Q-2, Q-1)$ are displayed in pageNum-th page directly.

## 4. PERFORMANCE EVALUATION

### 4.1 Experiment Design

**4.1.1 Experiment data:** Experiments are executed on metadata of Remote Sensing images provided by Geospatial Data Cloud (GSCloud, http://www.gscloud.cn), an open cloud platform which provides diverse and huge volume of geospatial data for the public. In GSCloud, each metadata is a polygon, representing the spatial extent of corresponding satellite image. Users can search Remote Sensing images covering certain geographic regions by drawing polygons on the map. The interface of data retrieval of GSCloud is shown in figure 4. By building index on the metadata, the efficiency of retrieving satellite images can be greatly improved. The volume of the metadata used in our experiments is 8.65 million records.

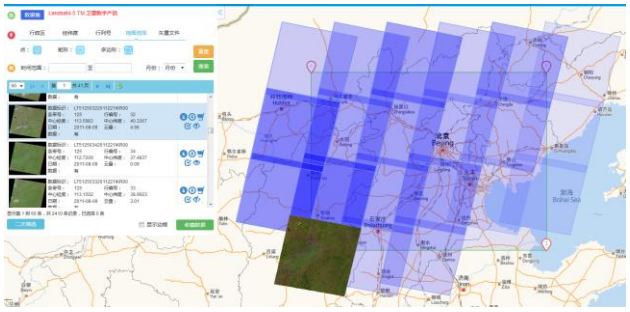

Figure 4. Illustration of experiment data

**4.1.2 Software Platform:** The experiments were conducted on a Spark cluster (version 1.5.2) consisting of three physical machines. Each machine was configured with Intel(R) Xeon(R) E5504 @ 2.00GHz 8-processor CPU, 32GB of RAM and 150GB of hard disk. The operating system is Fedora release 22 x86_64, and the Hadoop version is 2.6.0. All programs were implemented by Java1.8 and Scala2.10.4. Spatial database PostgreSQL was used as a comparison. The configuration of the database server is the same to that of a single machine in the Spark cluster. All the configurations of PostgreSQL are default.

**4.1.3 Strategies:**In order to achieve real-time performance, some optimization measures are taken first which are as follows:

1) In order to avoid the time cost for starting and stopping Spark tasks, we use RPC (Remote Procedure Call) to make Spark tasks resident in memory.
2) We deserialize and distribute all the subtrees to all the cluster nodes to achieve parallel query processing.
3) We cache all the data in memory to avoid reading data from HDFS. If the cluster do not have enough memory space to cache all the data, Spark will allocate 60% of the memory space to cache data and 40% of the memory space to perform calculations.
4) In the Counting Query, we use Redis to cache the results. The results are stored in the form of regionID-count pairs.

## 4.2 Results and Discussion

To evaluate the efficiency and scalability of the method, four experiments were carried out. The first two experiments evaluated the efficiency of Counting Query and Paging Query under the condition of different sizes of the query polygon respectively. Four cases were evaluated, which were PostgreSQL without GIST index, PostgreSQL with GIST index, Spark cluster without DKD-Tree index, and Spark cluster with DKD-Tree index. The third experiment evaluated the scalability of the method by comparing query time under Spark and DKD-Tree Spark. The last experiment evaluated the improvement of our method by avoiding a large number of polygon intersection calculation operations.

**4.2.1 Evaluation of Counting Query:** In this experiment, we evaluated the performance of our index in terms of Counting Query response time under the condition that the size of the query polygon varies. The area ratio of the query polygon to the MBR of the total experiment data set ranges from 0.05 to 1. The results are shown in Figure 5.

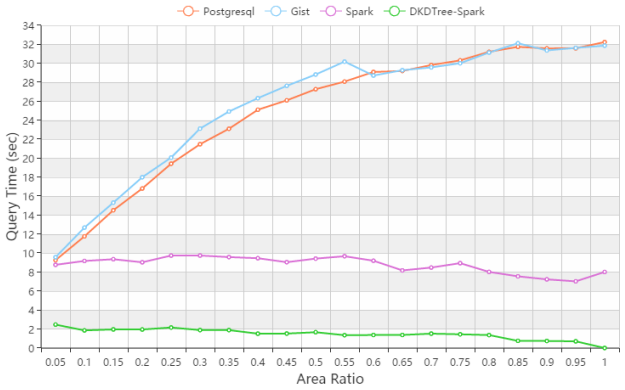


Figure 5. Counting Query in Different Sizes of the Query Polygon

The result in Figure 5 shows that the performance of Spark cluster with DKD-Tree is the best. The query response time of Spark cluster without DKD-Tree index is 6 times longer than our method, and the other two methods are about 5-60 times longer. This is because that polygon intersection operation is CPU-intensive. As Spark cluster is based on distributed parallel computation framework, it can use the CPU resources efficiently. While the computation of PostgreSQL is executed on a single core, so it is much slower. Besides, we can also see that the query time of PostgreSQL without GIST index is shorter than that of PostgreSQL with GIST index. That is because in case of large dataset, the depth of GIST index tree is deep, and the number of nodes is large, which results in a large index file. As a result, the query response time will be longer.

In addition, we can see that with the size of the query polygon increasing, the query response time of Spark cluster with DKD-Tree shows downward trend in the overall. The reason is that as the size of the query polygon increases, the MBR of the query polygon can fully cover more MBRs. When a MBR is fully covered by the MBR of the query polygon, it do not need to perform intersection calculation. Only a determination of whether the polygon is fully covered by the query polygon is enough. So the query response time is short.

**4.2.2 Evaluation of Paging Query:** In this experiment, we evaluated the performance of our index in terms of Paging Query response time under the condition that the size of the query polygon varies. For a given page number, we return 10 polygons that interest with the query polygon for display. The polygons are displayed in ID order. The results of this experiment is shown in Figure 6.

As shown in Figure 6, the query speed of Spark cluster with DKD-Tree is the fastest. That is because as the distribution of query results are obtained by Counting query phase, polygon intersection calculations are performed only on polygons that will be displayed in the specified current page. While for the other three cases, intersection calculations for all the polygons are needed. Also, the Paging Query must use ORDER BY SQL statement to get a global order when we use the SQL statement of LIMIT and OFFSET in database. And global ordering will take a long time. However, in with Spark cluster with DKD-Tree index, the number of polygons that need to be ordered is small, only polygons that will be displayed in the specified page. So the time spent in ordering will be greatly reduced. Moreover, as the sort process is stand-alone, it does not consume the resources of the cluster. However, in the case of PostgreSQL

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-IV-4-W2-131-2017 | © Authors 2017. CC BY 4.0 License. 

without GIST index, and PostgreSQL with GIST index, larger area contains more number of polygons and thus leads to longer processing time.

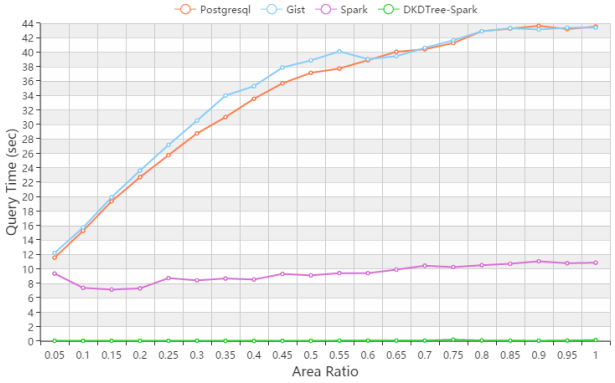


Figure 6. Paging Query in Different Sizes of the Query Polygon

**4.2.3 Counting Query in Different Number of CPU:** In this experiment, the scalability of our method is evaluated by comparing Counting Query response time under different number of CPUs are evaluated. The area ratio of query polygon to the MBR of the total experiment dataset is set to 0.5. Only two cases are tested in this experiment which are Spark cluster with DKD-Tree and Spark cluster without index. The results are shown in Figure 7.

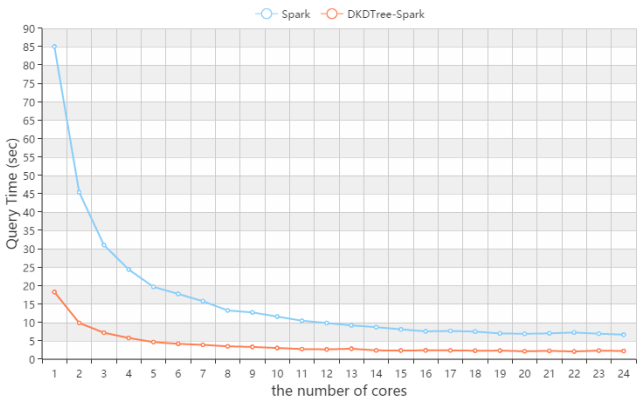


Figure 7. Counting Query with Different Number of Cores

As shown in Figure 7, the query time decreases with the number of cores increasing for both Spark cluster with DKD-Tree index and Spark cluster without DKD-Tree index. In Spark cluster node, the greater the number of cores, the more tasks can be executed in parallel. As the Spark cluster has good expandability, when the data volume grows, we can improve the query performance by adding the number of cluster cores.

**4.2.4 Evaluation of the Advantage of Avoiding Intersection Calculation for Fully Covered Regions:** The Spark with DKD-tree index can avoid intersection calculation when the query polygon fully cover the MBR of all polygons in a DKD-tree leaf node. In this experiment, we study the impact of varying the area of query polygon on average response time of our proposed DKD-Tree querying processing technique.

Twenty different query polygons are selected, the area ratio of which to the MBR of the total experiment data set ranges from 0.05 to 1. In the top half of Figure 8, the red columns stand for the number of DKD-tree nodes (partitions) intersect with the query polygon. The blue columns mean the number of DKD-tree nodes (partitions) that are fully covered by the query polygon. As illustrated in Figure 8, the number of fully covered partitions increases when the area of query polygon is larger. When the query polygon fully covers the MBR of the total experiment dataset, the number of partitions that intersects with the query polygon is 0. At this time, no polygon intersection calculation is needed.

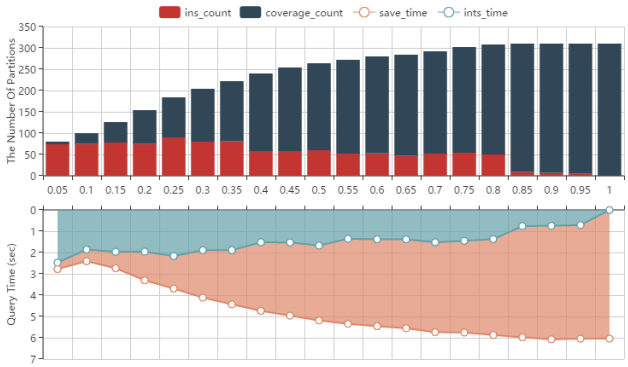


Figure 8. The advantage of Avoiding Intersection Calculation for Fully Covered Regions

In the bottom half of Figure 8, the blue curve shows the query time in the case of avoiding polygon intersection calculation for the polygons in partitions that are fully covered by query polygon. And the red curve shows the corresponding query time by performing polygon intersection calculation on all polygons. It clearly illustrates that our approach is very efficient by avoiding a large number of polygon intersection calculations.

## 5. CONCLUSIONS

In this paper we investigated the problem of efficient region querying of massive spatial data in parallel. To this end, we have presented a DKD-tree index to facilitate the processing of region queries concerning spatial polygons. And we implement the parallel construction of DKD-Tree based on Spark. Besides, we provide a two-step region query algorithm to achieve real-time query. In the first step which is called Counting Query, we return the distribution of number and regionIDs of the query result. In the second step, which is Paging Query, the polygons to be displayed on the specified page are returned. The parallel query based on DKD-tree can avoid polygon intersection calculation for those that are fully covered by the query polygon. Meanwhile, it does not need to collect data from all nodes in the cluster. Several experiments on the proposed method are performed using a real polygon dataset which are metadata of satellite images. The experimental results show that our method can significantly speed up the region query processing. Moreover, our method has good scalability with more cores can be added. So it has a great advantage in the case of increased data volume.

The goal of this paper is to address the challenges for efficient and scalable region query. The next step of our research will mainly focus on improving our method to support other types of query.

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-IV-4-W2-131-2017 | © Authors 2017. CC BY 4.0 License. 

## REFERENCES

Aji, A., & Wang, F. (2012). High performance spatial query processing for large scale scientific data. Proceedings of the on SIGMOD/PODS 2012 PhD Symposium - PhD '12, 9.

Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., & Saltz, J. (2013). Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. Proceedings of the VLDB Endowment, 6(11), 1009.

Akdogan, A., Demiryurek, U., Banaei-Kashani, F., & Shahabi, C. (2010). Voronoi-based geospatial query processing with MapReduce. Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010, 9–16.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. Communications of the ACM, 18(9), 509–517.

Cary, A., Yesha, Y., Adjouadi, M., & Rishe, N. (2010). Leveraging cloud computing in geodatabase management. Proceedings - 2010 IEEE International Conference on Granular Computing, GrC 2010, 73–78.

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplied Data Processing on Large Clusters. Proceedings of 6th Symposium on Operating Systems Design and Implementation, 137–149.

Eldawy, A., & Mokbel, M. F. (2015). SpatialHadoop: A MapReduce framework for spatial data. In Proceedings - International Conference on Data Engineering (Vol. 2015-May, pp. 1352–1363).

Fox, A., Eichelberger, C., Hughes, J., & Lyon, S. (2013). Spatio-temporal Indexing in Non-relational Distributed Databases, 291–299.

Geng, Y., Huang, X., Zhu, M., Ruan, H., & Yang, G. (2013). SciHive: Array-based query processing with HiveQL. Proceedings - 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013, 887–894.

Giuliani, G., Ray, N., & Lehmann, A. (2011). Grid-enabled Spatial Data Infrastructure for environmental sciences: Challenges and opportunities. Future Generation Computer Systems, 27(3), 292–303.

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.

Jing, G., Guangjun, L., Xurong, D., & Lei, G. (2006). 2-level r-tree index based on spatial grids and Hilbert R-tree. Geo-Spatial Information Science, 9(2), 135–141.

Kamel, I., & Faloutsos, C. (1992). *Parallel R-trees* (Vol. 21, No. 2, pp. 195-204). ACM.

Larson, R. R., & Frontiera, P. (2004). Geographic information retrieval (GIR). In Proceedings of the 27th annual international conference on Research and development in information retrieval - SIGIR '04 (p. 600).

Li, Z., Hu, F., Schnase, J., & Duffy, D. (2016). A spatiotemporal indexing approach for efficient processing of big array-based climate data with MapReduce. International Journal …, 8816(February), 17–35.

Lupa, M., & Pi, A. (2014). Spatial Query Optimization Based on Transformation of Constraints Spatial Query Optimization Based on, 242(JANUARY), 0–9.

Ma, Y., Wang, L., Liu, P., & Ranjan, R. (2015). Towards building a data-intensive index for big data computing - A case study of Remote Sensing data processing. Information Sciences, 319, 171–188.

Papadopoulos, A., & Manolopoulos, Y. (2003). Parallel bulk-loading of spatial data. Parallel Computing, 29(10), 1419–1444. http://doi.org/10.1016/j.parco.2003.05.003

Procopiuc, O., Agarwal, P. K., Arge, L., & Vitter, J. S. (2003). Bkd-tree: A Dynamic Scalable kd-tree. Sstd, 2750(July 2003), 46–65.

Rosenberg, J. B. (1985). Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 4(1), 53–67.

Samet, H. (1984). The Quadtree and Related Hierarchical Data Structures. ACM Computing Surveys, 16(2), 187–260.

Samet, H. (1990). The design and analysis of spatial data structures. MA: Addison-Wesley (Vol. 199).

Shekhar, S., Evans, M. R., Gunturi, V., & Yang, K. (2012). Spatial big-data challenges intersecting mobility and cloud computing. MobiDE 2012 - Proceedings of the 11th ACM International Workshop on Data Engineering for Wireless and Mobile Access - In Conjunction with ACM SIGMOD / PODS 2012, 1, 1–6.

Vatsavai, R. R., Ganguly, A., Chandola, V., Stefanidis, A., Klasky, S., & Shekhar, S. (2012). Spatiotemporal data mining in the era of big spatial data. Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data - BigSpatial '12, 1–10.

Yu, J., Jinxuan, W., & Mohamed, S. (2015). GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data. In 23th International Conference on Advances in Geographic Information Systems (pp. 4–7).

Zaharia, M., Chowdhury, M., Das, T., & Dave, A. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Nsdi (pp. 2–2).