Comput Syst Sci & Eng (2017) 5: 405–415

International Journal of
Computer Systems
Science & Engineering

A mapreduce-based efficient H-bucket PMR quadtree spatial index

Hari Singh[1] **and Seema Bawa**[2]

[1] *PhD Scholar, Computer Science & Engineering Department, Thapar University, Patiala-147001, Punjab, India. E-mail: hsrawat2016@gmail.com*
[2] *Professor, Computer Science & Engineering Department, Thapar University, Patiala-147001, Punjab, India. E-mail: seema@thapar.edu*

Majority of the MapReduce-Hadoop based spatial indexes are based on either non-disjoint decomposition or the data-dependent disjoint decomposition of space. Quadtree index based regular disjoint decomposition in MapReduce takes different forms of spatial data as point data. Lines, curves, polygons and other higher dimensional data are transformed to point data through a mapping process. Though, the mapping makes index-building quite easy, but it is not suitable for answering search queries. This paper proposes H-bucket PMR Quadtree, a parallel implementation of the existing bucket-PMR Quadtree to handle curvilinear or polygonal map data, in MapReduce. The proposed index uses a two-level of indexing: a global index that indexes the decomposed dataset among cluster nodes to support parallel index building and a local bucket-PMR Quadtree index maintained by each participating cluster node. The proposed index is compared with the state-of-the-art MapReduce based R+-tree indexing and the default key-value storage (non-indexed) Hadoop towards index build-time and spatial queries, such as line search and range search queries. The experimental results demonstrate the effectiveness of the proposed index in MapReduce environment

Keywords: Hadoop, MapReduce, Quadtree, Spatial, Index

1. INTRODUCTION

It has become very important to design high performance queries on spatial data, as the data is voluminous by its internal characteristics, high computational complexity involved in processing and considerable time taken by complex spatial queries. In earlier times, parallelization was limited to storage of data on parallel disks rather than parallel computing using multiple processors. There has been a lot of research on data-parallel construction of spatial index algorithms on different parallel platforms. The hardware architecture containing single processor and multiple disks is used (Kamel & Faloutsos 1992). It made query search fast, but it did not parallelize bulk-loading of indexes. In (Papadopoulos & Manolopoulos 2003), authors enhanced the architecture of (Kamel & Faloutsos 1992) by considering parallel bulk-loading with multiple processors, in addition to the existing multiple disks. Other multiple processor variants optimized partitioning of input data space for bulk-loading using algorithms, such as Hilbert packing. (Schnitzer & Leutenegger 1999). The SAM (Scan-And-Monotonic-mapping) architecture of parallel computation used a linearly ordered set of processors for performing element-wise and scan-wise operations in parallel (Bestul 1992).

MapReduce has been proven very efficient for solving problems involving huge datasets of semi-structured characteristic. However, a lot of researches have been proposed for optimizing the MapReduce Hadoop. The performance of the Hadoop cluster is optimized through improving fault tolerance. It is accomplished through predicting failures using the Markov process in the MapReduce programming framework (Zheng et al. 2016). The MapReduce-Hadoop framework has shown optimized query performance when input dataset is indexed (Dean & Ghemawat 2013; Singh & Bawa 2017; McCreadie et al. 2012). It is only the last few years that MapReduce model has been exploited in the field of spatial data to parallelize query execution over huge dataset for improving query efficiency. An effective indexing and searching with dimensionality reduction on high-dimensional space is achieved through a two step transformation.

Firstly, the high dimensonal data is represented as vectors and secondly, mapping feature vectors in multi-dimensional space into vectors in low-dimensional space. Finally, vecotrs in low-dimensional space are indexed for information retrieval (Jeong et al. 2016). The accuracy of outputs generated for processed spatial data in the Hadoop is similar to that generated using GIS software ArcGIS Singh & Bawa 2016). Until recently, most of the MapReduce oriented researches on spatial indexes are based on non-disjoint decomposition or irregular disjoint decomposition, such as R-tree and variants (Xun & Wenfeng 2013; Eldawy & Mokbel 2013; Wang & Weng 2010; Achakeev et al. 2012; Cary et al. 2010; Liu et al. 2009; Zhang et al. 2012; Tan et al. 2000; Liao et al. 2010). The major drawback of non-disjoint decomposition methods is that object may be spatially contained in more than one bounding rectangle, yet it is only associated with one bounding rectangle. A spatial query may require searching many bounding rectangles in search of an object. In the worst case, whole database or all bounding rectangles would need to be searched. The non-disjoint decomposition data structures suffer from a high load time of structure, poor tree structure and poor search time, but these provide good storage utilization. On the other hand disjoint decomposition methods require more storage, but these reduce search time considerably. The performance of quadtree and variant indexes, for spatial index building and query processing, is well established (Bestul 1992; Hoel & Samet 1994b; Hoel & Samet 2003; Hoel & Samet 1994a; Hoel & Samet 1992; Jun et al. 2014). The disjoint decomposition causes sub-objects obtained from main object to be associated with different cells. This causes a little problem when the area covered by an object is calculated, because it forces the area calculation for all the cells to which the object is associated. R+-Tree (Wang & Weng 2010), the Quadtree (Samet 1990) and Grid file (Nievergelt et al. 1984) are examples of data structures that follow disjoint decomposition approach. R+-tree is a variant of R-tree that allows disjoint decomposition of space by not allowing overlapping among intermediate nodes. It leads to a good improvement in search time but requires more storage as compared to R-tree and R∗-tree. The drawback of R+-tree is that decomposition is data-dependent, which makes it difficult to perform some tasks that require composition of different operations and data sets.

In contrast, Grid file index and quadtree data structures have a greater degree of data-independence. Grid file uniformly decomposes grid space into blocks of uniform size with use of a multidimensional hash algorithm (Nievergelt et al. 1984). The spatial dataset is mapped onto a grid of uniform sized cells and accessed through an indexed grid file. A grid file consisted of one dimensional array that provides a mapping from spatial location to grid directory. Each directory within n-dimensional grid directory points to a specific dataset. Though, this method has a simple structure, but it also has a high disk I/O time. Two disk accesses are required for data retrieval. First disk access obtains directory entry and the second accesses the required data. Besides a high disk I/O time, the grid file is more suitable for representing uniformly distributed data. In general, the spatial data is not uniformly distributed, so, a more flexible quadtree data structure that suits to non-uniformly distributed data is proposed in this paper. A specific quadtree index for curvilinear data, the bucket-PMR Quadtree index, has been considered for parallelizing in MapReduce.

The rest of the paper is organized as follows. Section 2 describes related work and motivation. Section 3 describes background of data structures used in the paper in MapReduce. Section 4 describes the design of proposed H-bucket PMR Quadtree index and search queries in MapReduce. Section 5 describes the implementation and discusses experimental results obtained. Section 6 presents conclusions and future scope of the presented work.

2. RELATED WORK

The regular disjoint decomposition using quadtree have more potential for inter-processor communication or in other words for parallelism. However, not much work is available that uses quadtree-based spatial data structure in MapReduce. One such approach uses quadtree-based global index and Hilbert-curve based local index (Zhong et al. 2012). The global index searches data blocks and Hilbert index locates spatial objects for efficient data retrieval. The use of quadtree based indexes for building a local index in MapReduce is proposed (Jun et al. 2014). The authors proposed a HQ-Tree index in Hadoop that considers PR-Quadtree index for spatial point objects. It solves issues of order of data insertion and space overlap in non-disjoint decomposition approaches. It improved execution time performance for index creation and, point and range query, for parameters data size, node size and number of query target points. However, the HQ-Tree approach is limited to spatial point objects.

Among many quadtree based data structures for indexing and querying, the PMR Quadtree is found to be the best for representing the curvilinear data (Samet 1990). In the domain of traditional serial programming models, PMR Quadtree index based data structure has been proven better for spatial join queries (Hoel & Samet 1995a) and line segment queries (Hoel & Samet 1992) over the R-Tree (Antonin Guttman 1984), R∗-tree (Beckman et al. 1990) and R+-tree (Sellis et al. 1987). However, motive of the concerned researches has always been on reducing build-time and query execution time. Efforts had been made in the past for implementing existing serial spatial indexes on parallel platforms. One such effort has used hypercube architecture, Scan-And-Monotonic-mapping (SAM) model of parallel computation (Bestul 1992), for spatial data structure indexing and querying (Hoel & Samet 1994b; Hoel & Samet 2003; Hoel & Samet 1994a). The hypercube architecture is a tightly coupled architecture that is characterized by 2^n processors interconnected as n-dimensional binary cube (Hayes & Mudge 1989). A scan operation (Blelloch & Little 1994), comprising of primitive operations (Hoel & Samet 1995b)-element-wise, permutation and scan, operates on long vectors of data. In one such comparative study on the SAM model, PMR Quadtree is found to be slightly faster than R-tree and much better than R+-tree for similar tree node capacities (the capacity to hold the number of spatial objects by a node of tree) for spatial join queries (Hoel & Samet 1994b; Hoel & Samet 1994a) and polygonization queries (Hoel & Samet 2003).

However, MapReduce based parallel programming, the Hadoop (Apache 2015), provides a loosely coupled architecture. The computing nodes can be added on the fly to provide scalability, which is not possible in the SAM model. The pro-

gramming by using Map and Reduce functions is quite easy in MapReduce, as compared to the scan operations that operates on long vectors of data. Other important things that favor use of MapReduce model is its simplicity, ease of use and ability to easily handle large datasets. Besides all these factors, fault tolerance in MapReduce model and the property of abstracting parallelization from user are reasons that have motivated researchers to use it. It provides good performance through scaling out to computing clusters.

The ability of bucket-PMR quadtree index to efficiently handle the curvilinear data for indexing and querying, and advantages of the Hadoop parallel processing framework have motivated us to carry out this work. This paper proposes and implements H-bucket PMR Quadtree index, a data-parallel version of the bucket-PMR quadtree index in the Hadoop, where the letter "H" stands for the Hadoop.

3. BACKGROUND: SPATIAL INDEXES USED IN MAPREDUCE

The Hadoop is an open source implementation of MapReduce parallel programming model. It programs logic through map-and-reduce functions in a Master-Slave Hadoop architecture that handles input-output in key-value pair. The model parallelizes processing by partitioning input dataset on the distributed file system of the cluster of computing nodes. Each node processes part of its task and returns result to master node. In this section, a brief description of two existing disjoint decomposition data structures: bucket-PMR quadtree and R+-tree is presented, that follows its implementation in MapReduce. The purpose is to compare build-time and query execution time of state-of-the-art MapReduce-based R+-tree and H-bucket PMR Quadtree.

3.1 Overview of bucket-PMR Quadtree

The detailed process of inserting line segments in PMR (Polygonal Map Random) quadtree is described in (Nelson & Samet 1986). The line segments are inserted into PMR quadtree, starting with inserting first line segment into an empty block. A block is split into four quadrants of equal sizes until object count in each block reaches a threshold value. The splitting threshold is a permissible number of line segments in a block. A new line segment is inserted into a block, if it intersects a block, and the block is checked for the splitting threshold. A block is split into sub-blocks only once, as it contains a few very close lines. It is important to keep particular value of splitting threshold, as too small value causes many subdivisions that lead to many empty sub-blocks and hence increased storage requirement. A large value of splitting threshold causes a decreased construction time and storage requirement but at the same time it increases the time for performing operations on it. The order of inserted line segments decides shape of the resulting PMR quadtree.

However, in parallel construction of PMR quadtree, insertion ordering of line segments is not known, as lines are inserted simultaneously in parallel and therefore a slight modification to PMR quadtree, known as a bucket-PMR quadtree is used. The tree data structures generally access data from primary memory and pointers are used to access data stored in pages of secondary memory, however, it gives rise to page faults. The bucketing method overcomes this problem by collecting data objects into sets called buckets, and providing access to buckets through appropriate address computation mechanism. In bucket-PMR quadtree, a block or bucket is split repeatedly until a splitting threshold is reached, unlike in the PMR quadtree, which allows splitting a block only once. The splitting threshold of a bucket is also represented as node capacity of a tree node. The node capacity of a data structure tree node is the data holding capacity of a node in terms of size of data in a tree. It decides the number of objects in a bounding rectangle. The objects are grouped according to their proximity for better results.

3.2 Conceptual view of the H-bucket PMR Quadtree in MapReduce

The conceptual idea of creating a bucket-PMR Quadtree index in MapReduce is presented in Figure 1. The Figure 1(a) shows the input polygonal map dataset of 11 line objects. The dataset is partitioned as per the block size over Hadoop Distributed File System (HDFS), and required number of mappers use Algorithm-1 to split line objects into different quadrants. The line objects from same quadrant are output to a reduce function, as shown in the Figure 1(b) that starts the index building process as per Algorithm-2.

A bucket size of three is assumed here. It means only three line objects can be accommodated in the proposed tree node. Each Reduce node checks the number of line objects it has received from the mapper. If the count is less than or equal to three then all line objects are assigned to the node and the node gets the status of a leaf node (L). Otherwise, the node is vacated with status changed to internal node (IN) and all entries of the node are given to four newly created child nodes as per the MBR intersect condition of Algorithm-2. This process is repeated until all nodes are filled-up with at most three line objects or there are no more objects for insertion. The process on completion creates an H-bucket PMR Quadtree, as shown in Figure 1(c), and its logical counterpart is shown in Figure 1(d).

3.3 Conceptual view of R+-Tree in MapReduce

R+-tree is a variant of R-tree that allows disjoint decomposition of space by not allowing overlapping among intermediate nodes. It leads to a good improvement in search time, but requires more storage as compared to R-tree and R∗-tree. The drawback of R+-tree is that the decomposition is data-dependent, which makes it difficult to perform some tasks that require composition of different operations and datasets. A recursive top-down approach is used to insert a line segment into an R+-tree that places it into every leaf node that it intersects (Hoel & Samet 1992). If a leaf node in which a line segment is inserted overflows, then the leaf node is split. The split approach minimizes the total number of resulting portions of line segments. To fulfill this requirement, all possible vertical and horizontal split lines are considered, and for each split line, the number of intersected line segments is counted. Finally, a split line having a minimum number of

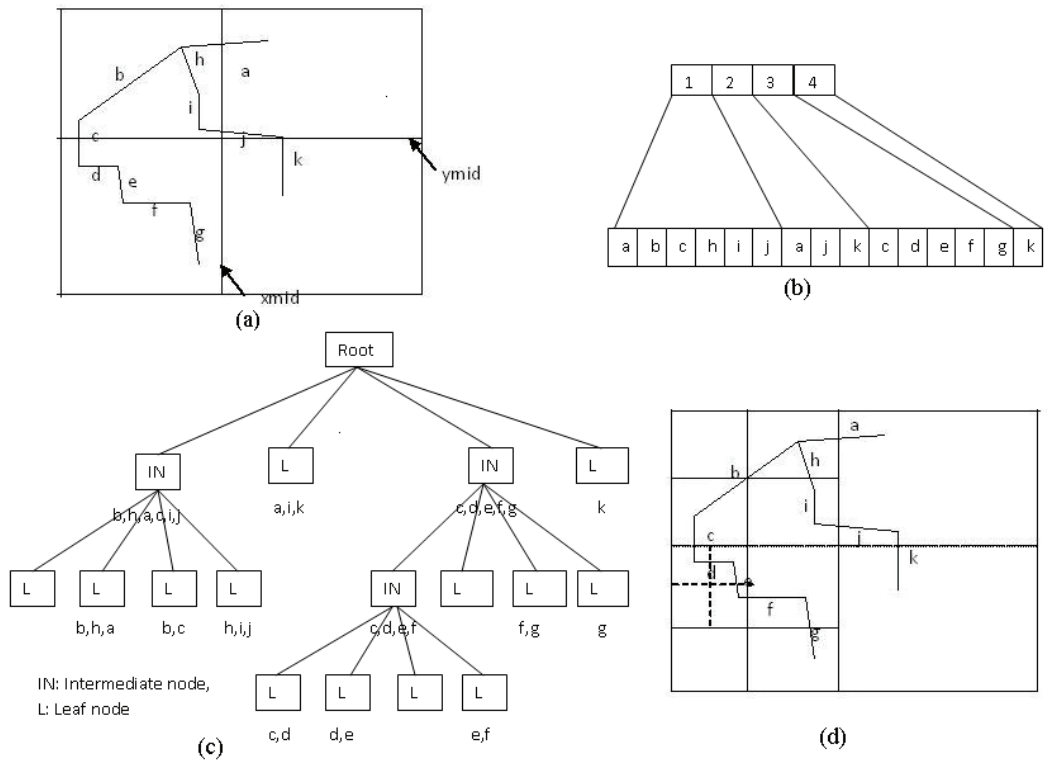


Figure 1 The Conceptual View of the H-bucket PMR Quadtree in MapReduce (a) Input spatial line data to MapReduce (b) Map function in MapReduce assigns the decomposed input spatial line objects into four quadrants (c) Data in the quadrants recursively filled-up in a bucket-PMR Quadtree nodes for a bucket size=3 (d) Logical structure of the MapReduce constructed H-bucket PMR Quadtree for bucket size=3.

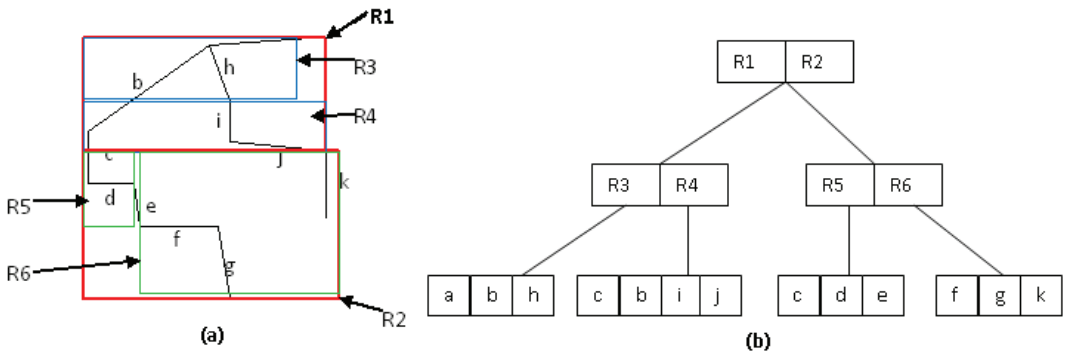


Figure 2 Conceptual View of R+-Tree Index in MapReduce (a) The spatial area representative of the bounding rectangle (b) Corresponding R+-Tree for the collection of line segments.

intersections is taken, and the splits are propagated up the tree.

For an input spatial line dataset shown in Figure 1(a), there can be many spatial representations of the extents of bounding rectangles. One such representation of bounding rectangles is presented in Figure 2(a) and the corresponding R+-Tree is shown in Figure 2(b). The structure of R+-Tree of order (m, M) is such that all intermediate nodes and leaf nodes contain between $m <= (M/2)$ and M entries. However, it is not guaranteed until a complicated record insertion and deletion procedure is followed. The root node has at least two entries, if it is not a leaf node.

The procedure for building R+-tree of the Figure 2(a) is described here. Initially, root node consists of R1 and R2 rectangles. R1 contains line segments a,b,h,i,j,c and R2 contains line segments c,d,e,f,g,k. Notice that the line segment c is replicated in both the rectangles. The number of entries is much greater than permissible range, so rectangles R1 and R2 are further divided into rectangles R3, R4 and R5, R6 respectively. Now, rectangle R3 contains entries a,b,h and rectangle R4 contains entries c,b,i,j. The line segment b is replicated in both sub-rectangles. Similarly, rectangle R5 contains entries c,d,e and rectangle R6 contains entries f,g,k. R+-tree has been investigated over many parallel models such as SAM (Hoel & Samet 1994b; Hoel & Samet 1994a; Sellis et al. 1987) and MapReduce (Zhong et al. 2012). R+-tree implementation in MapReduce (Zhong et al. 2012) is considered here for studying index build-time and spatial query execution time.

4. DESIGN OF H-BUCKET PMR QUADTREE IN MAPREDUCE

This section presents H-bucket PMR quadtree in MapReduce. Firstly, data partitioning into four quadrants in MapReduce is carried out, for parallel construction of proposed index. Secondly, tree node structure and bulk-loading of the proposed index is discussed. Thirdly, design of two spatial queries, line and range query, is described on the proposed index.

4.1 Proposed Parallel H-bucket PMR Quadtree Index in MapReduce

The split of spatial dataset depends on the size of the data block in HDFS. If the block size is b and spatial dataset is of size s, then $n = s/b$ splits are created and each split is taken care of by one mapper. Each Mapper runs Algorithm-1 and segregate line objects in one of the quadrants. For each row of spatial data in the input (Comma Separated Value) CSV file, each map function checks in parallel both ends of line coordinates for insertion in four quadrants. For deciding the space of four quadrants, two variables xmid and ymid are computed on the basis of the minimum and maximum values for x and y coordinates from the whole spatial data. For each line having start and end coordinates ((lx-min, ly-min),(lx-mx, ly-max)), Each mapper function puts the line in one of the four quadrants through finding out lx-min and lx-max against the x-mid, and ly-min and ly-max against the y-mid. Two cases arise when lines are compared against xmid and ymid values: (a) lines do not intersect xmid and ymid coordinates and completely falls into one of the four quadrants. (b) lines intersect xmid and ymid coordinates and are put in both the quadrants. The output of Map function for a particular line is (id of the quadrant, line (line-id and MBR)). All lines common to a quadrant become input to the reducer with (id of the quadrant, list(line)). Now, the H-bucket PMR Quadtree index is created for all the lines in a particular quadrant. The generated H-bucket PMR Quadtree index from each reducer is merged outside the reduce function in a serial way to get the complete H-bucket PMR Quadtree index.

Algorithm-1: Parallel H-bucket PMR quadtree index in MapReduce

INPUT Lines: set of spatial lines.
OUTPUT: H-bucket PMR Quadtree index.
Step1: for each line object ((lx-min, ly-min),(lx-max, ly-max))
  if (((lx-min && lx-max)<x-mid) && ((ly-min && ly-max)>y-mid))
    then put line in 1^{st} quadrant.
  end if
  if (((lx-min && lx-max)>x-mid) && ((ly-min && ly-max)<y-mid))
    then put line in 2^{nd} quadrant.
  end if
  if (((lx-min && lx-max)<x-mid) && ((ly-min && ly-max)<y-mid))
    then put line in 3^{rd} quadrant.
  end if
  if (((lx-min && lx-max)>x-mid) && ((ly-min && ly-max)>y-mid))
    then put line in 4^{th} quadrant.
  end if
  if (lx-min<x-mid<lx-max)
    then include line in both the quadrants.
  end if
  if (ly-min<y-mid<ly-max)
    then include line in both the quadrants.
  end if
  emit (id of the quadrant, line (line-id and MBR))
 end for
Step2: H-bucket PMR Quadtree index is created by a reducer function with input (id of the quadrant, list (line)) for each quadrant according to Algorithm-2.
Step3: Set path for input and output directories, and then start up MapReduce.
Step4: A merge process in Hadoop combines all child indexes.

4.2 Data structure of H-bucket PMR Quadtree index node

The node structure of H-bucket PMR Quadree index is represented with a class BucketPMRQTreeNode that contains six elements: level-level of BucketPMRQuadTree node, the level describes stage of recursive decomposition; maxLevel-maximum permissible level defined for BucketPMRQuadTreeNode; BucketCapacity-maximum number of lines objects that can be kept in a tree node; Minimum Bounding Rectangle (MBR)-area of rectangular quadrant in which lines lie in the form of ((xmin, ymin),(xmax, ymax)), childNodes-an array of type BucketPMRQTreeNode; and SpatialData Collection-contains line objects implemented with Collection structure in Java that can store line objects but not more than BucketCapacity. A class Box is presented separately just to simply presentation of tree node structure. It specifies a rectangular region enclosed by MBR.

```
Class BucketPMRQTreeNode
{
   Int level; // level of this node
   Int maxLevel; // maximum number of levels of quadtree;
   Int BucketCapacity;
   Box MBR;
   BucketPMRQTreeNode childNodes[];
   SpatialDataCollection data;
}

Class Box
{
   Double xmin,ymin;
   Double xmax, ymax;
}

Public class SpatialDataCollection{
ArrayList<BucketPMRQTreeNode> items= new ArrayList<
BucketPMRQTreeNode >();}
```

4.3 Algorithm for data insertion in H-bucket PMR Quadtree Index

Input to the reducer as described in sub-section 4.1, (id of the quadrant, list(line, and the MBR to which it belongs)), creates a local H-bucket PMR Quadtree index for all line objects falling in this quadrant as per Algorithm-2.

Algorithm-2: Bulk-loading H-bucket PMR Quadtree index

Input: spatial data
Output: A local H-bucket PMR Quadtree:
Step1: Get spatial data to be inserted in H-bucket PMR Quadtree.
Step2: Check spatial data against quadrant MBR and perform step3 if it intersects the MBR.
Step3: Check BucketPMRQTreeNode for a leaf/non-leaf node of H-bucket PMR Quadtree index.
if Isleaf=true then
perform step 4 otherwise goto step 7.
end if
Step4: Add this spatial data item to ArrayList Collection.
Step5: After Inserting spatial data in ArrayList Collection, check for maximum level reached or the number of data items that exceed bucket capacity. If either one holds true, then split node into four child nodes and increase level by 1 and vacate ArrayList Collection into a child node's ArrayList Collection as per step 6.
Step6: Repeat steps 2, 3, and 4. Clear parent node's ArrayList Collection and set
Isleaf=false.
Step7: Repeat steps 2 to 6 for each child node in the H-bucket PMR Quadtree.

4.4 Spatial queries in H-bucket PMR Quadtree index

The efficiency of the proposed index is demonstrated through implementing and executing spatial queries, such as line search and range search, on the proposed index. The query algoritms in MapReduce are as follows:

4.4.1 Line search query

The line search query returns true for input line objects to be searched when search process finds a line object in the index. Algorithm-3 splits input lines spatial data and then passes the split data (line_id coodinates((xmin,ymin), xmax,ymax))) to map function. The map function carries out line search using conventional PMR Quadtree line search algorithm (Samet 1990). The reduce function counts the total number of line objects.

Algorithm-3: Line search query

Input: set of spatial lines
Output: the boolean value true for line which are found in the index.
Step1: A set of lines is partitioned into splits and input to the program.
Step2: In a Map function, the line search query is implemented. For a particular (line(xmin.ymin),(xmax,ymax)), the function returns 1 if found, otherwise returns -1.
Step3: In a reduce function, input from the map function is (Found,list(identifiers of existing lines). Count the number of lines that is found and output result onto HDFS.
Step4: Set path of input and output directories, and then start up MapReduce.

4.4.2 Range search query

Algorithm-4 finds out all line objects that intersects or lies in a particular input range or region. Firstly, it finds index space that intersects with or includes query range and then, Map function passes the index root node of the sub-tree for Reduce function. The Reduce function computes all line objects which overlap with query range and merge the result to form overall query result.

Algorithm-4: Range search query

Input: MBR of the query range.
Output: Line objects.
Step1: Split region into several parts.
Step2: For overlapped area of the split part of region and index space, Map function outputs (index, split part of the region).
Step3: Reduce function executes range query in index space provide by Map function as input to Reduce function.
Step4: Set path of input and output directories, and then start up MapReduce.
Step5: Merge outputs of split parts of the region, and return the whole result.

5. RESULTS AND DISCUSSIONS

This section starts with an introduction to experimental setup, configuration and dataset used in experimental analysis, and presents results for comparing H-bucket PMR Quadtree index, Haddop based R+-tree index and non-indexed-Hadoop with respect to various parameters. Non-indexed Hadoop is the default key-value storage provided by the Hadoop that does not use any built-in indexing mechanism. It simply takes input dataset on the HDFS and searches the dataset for queries.

5.1 Experimental setup, configuration and dataset used

A Hadoop cluster over 10 computing nodes consists of one computing node as Master node and the rest nine, act as slaves in master-slave cluster configuration. The configuration of all computing nodes is same except for a minor difference. The master node that is equipped with 2 GB RAM and the rest of computing nodes has following configuration: Dual Core @ 2.1 GHz processor, 1 GB RAM, Ubuntu 11.04 operating system, Java-6-openjdk, and Hadoop-0.21.0. The master node is configured with IP address 192.168.10.11 with a hostname Master and slave nodes have IP addresses in the range 192.168.10.12 to 192.168.10.20 with host names Slave-1 to Slave-9.

The road intersection data of five counties of California state has been used from tiger/line dataset in the experimentation (ESRI 2016). The following counties of the California state have been considered: Fresno County, Sierra County, Santa Cruz County, Lake County, and Butte County. A CSV form of polygonal map data for each county is considered for building H-bucket PMR Quadtree and R+-tree index in MapReduce, as described earlier. Besides analyzing build-time, a number of test runs of line and range search queries are conducted on the two indexes and non-indexed-Hadoop for analyzing the execution time. The parameters are observed for a large number of test runs on each county dataset for obtaining better accuracy. The mean value from these test runs is plotted for presenting the analysis.

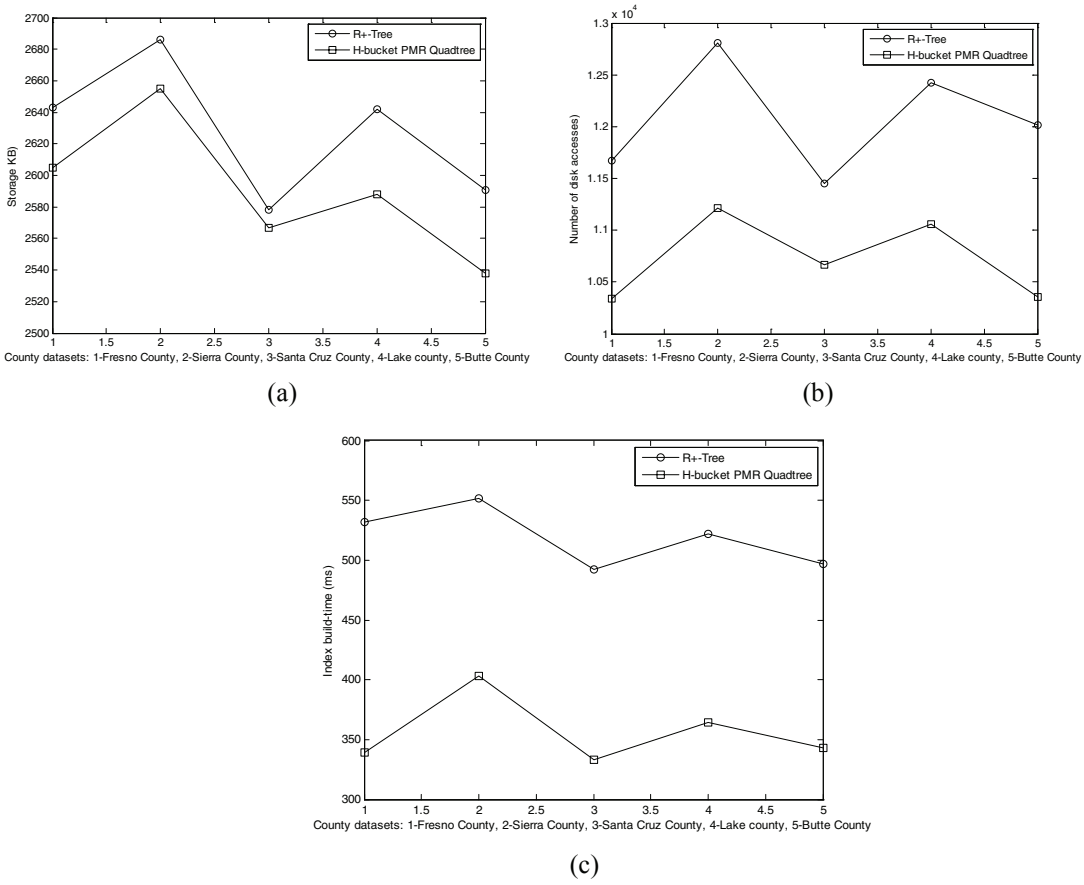


Figure 3 A comparison between MapReduce based R+-tree and H-bucket PMR Quadtree (a) Storage required (b) Number of disk accesses (c) Index build-time.

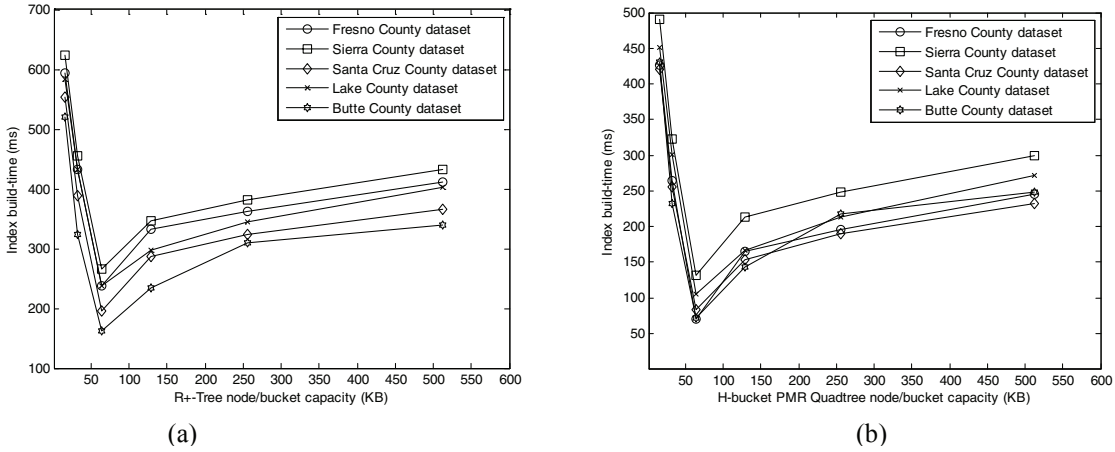


Figure 4 Index build-time in MapReduce in varying node capacity of indexes (a) R+-tree (b) H-bucket PMR Quadtree, in various county datasets.

5.2 Cost of building indexes with respect to storage, number of disk accesses and execution time

The costs of building indexes with respect to parameter storage, number of disk accesses and execution time are analyzed on a cluster of ten computing nodes. A set of 30 experiments was conducted for result analysis, three experiments for each index (R+-tree and H-bucket PMR Quadtree) and for each county dataset (five counties). Figure 3(a–c) shows the storage requirement of the MapReduce implemented R+-tree and H-bucket PMR Quadtree. Both are almost similar, but the latter is slightly better than the former. The disk access time of H-bucket PMR Quadtree is also almost similar to R+-tree, but the former takes a slightly less number of disk accesses as compared to the latter for all polygon maps. However, index building time of the former was found to be significantly smaller as compared to the latter, and the former takes approximately 20–50% less time than the latter. It is due to regular disjoint decomposition in case of bucket-PMR Quadtree, as the decision to split an overflowing node effectively requires only two candidate split axis/coordinate pairs. R+-tree requires testing a possibly large number of split

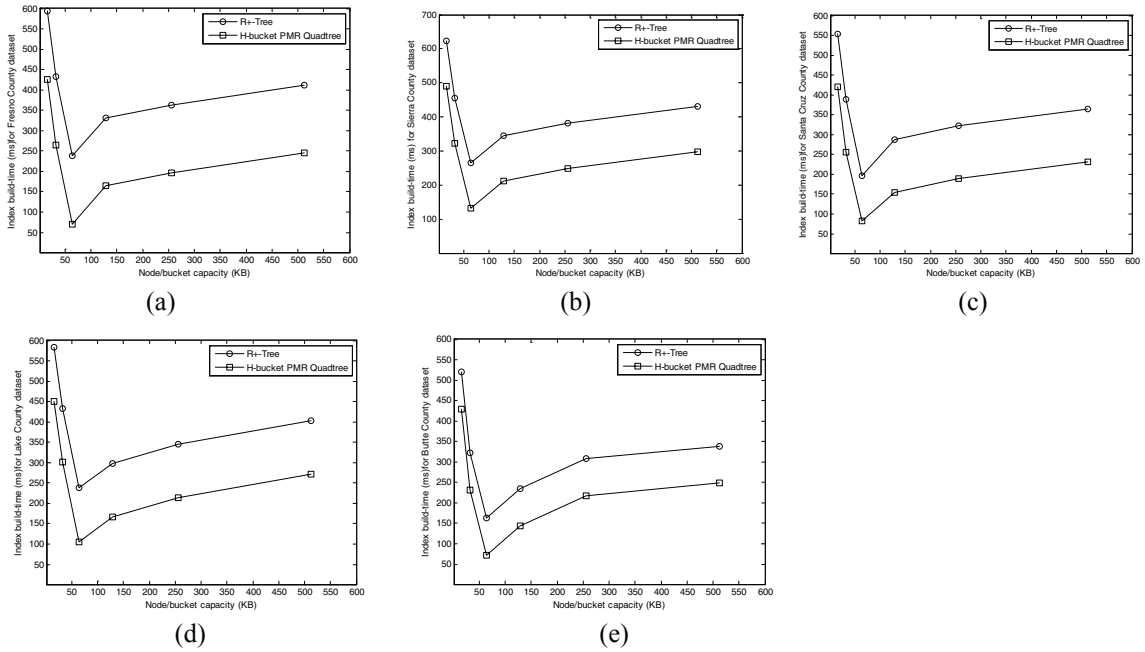


Figure 5 Index build-time in MapReduce in varying node capacity of indexes R+-tree and H-bucket PMR Quadtree Index (a) Fresno County (b) Sierra County (c) Santa Cruz County (d) Lake County (e) Butte County.

axis/coordinate pairs in determining a locally optimal node split. This node split is an iterative work that depends on population of objects in a tree node. A large number of clipping operations are required before determining which part of line is associated with the two nodes resulting from a split. While in case of bucket PMR Quadtree, clipping operations are constant due to regular disjoint decomposition of space.

5.3 Effect of tree node/bucket size on index-building time

The bucket size or node capacity is the data holding capacity of a tree node and is decided by the node size of an index tree. It significantly impacts performance of queries. A set of 180 experiments was conducted, three experiments for each node capacity (16KB, 32 KB, 64 KB, 128 KB, 256 KB and 512 KB), for each index and for each of the five counties.

It is observed that index building time of H-bucket PMR Quadtree and R+-tree index in MapReduce decreases with increasing bucket size up to 64 KB. However, results in Figure 4(a-b) show that index building time decreases sharply for both the indexes for a cluster size of 10 when the node size increases gradually up to a limit. It is due to a reduced computation required for testing split/axis coordinate pairs with an increase in node capacity, but this behavior persists till the size of data packets transfer in HDFS becomes 64 KB. When a tree node size increases further and becomes larger than 64 KB, the number of network transfers increases. For tree node sizes more than 128 KB, 256 KB and 512 KB, the number of network transfers continue to grow. So, node size calibration is important and setting it below 64 KB produces optimized query response performance. Further, a comparison between the two indexes in Figure 5(a-e) shows that index build time of H-bucket PMR Quadtree is faster than a R+-tree in MapReduce by a factor of 1.39–2.01, 1.27–2.00, 1.31–2.36, 1.29–2.24, 1.21–2.26 in counties Fresno, Sierra, Santa Cruz, Lake and Butte, respectively. In overall, H-bucket PMR Quadtree index build-time is faster than R+-tree index build time in MapReduce by an average factor of approximately 1.29–2.17.

5.4 Effect of cluster size on search queries execution time

A set of 450 experiments is conducted, three experiments for different sizes of cluster varying from 1 to 10 computing nodes and for each county dataset, for a tree-node/bucket size of 64 KB. The mean values are used to draw the graphs presented in Figure 6(a–e) and Figure 7(a–e) for line and range search queries, respectively. The query search behaves just like a search operation on a standalone system, when there is a single node in the cluster. But on the addition of a second node in the cluster, search time decreases slightly. On the addition of a third node, there is a sharp decrease in the execution time. With the addition of a fourth, fifth and sixth node, search time shows a continuous decreasing trend. However, on the addition of a seventh, eighth, ninth and tenth node, search time does not decrease further and it remains almost constant. The slight decrease in execution time when a second node is added is due to the increased shuffling of intermediate data and files, which overcomes the performance gain due to parallelism in the cluster. However, later on when more nodes (up to six in number) are added, the effect of computation due to cluster overweigh the shuffling of intermediate data. But, later on there does not seem any change in execution time when more nodes are added (from 7^{th} to 10^{th} node), it is due to the size of data that is being considered is enough for keeping busy six computing nodes. If, more nodes are added,

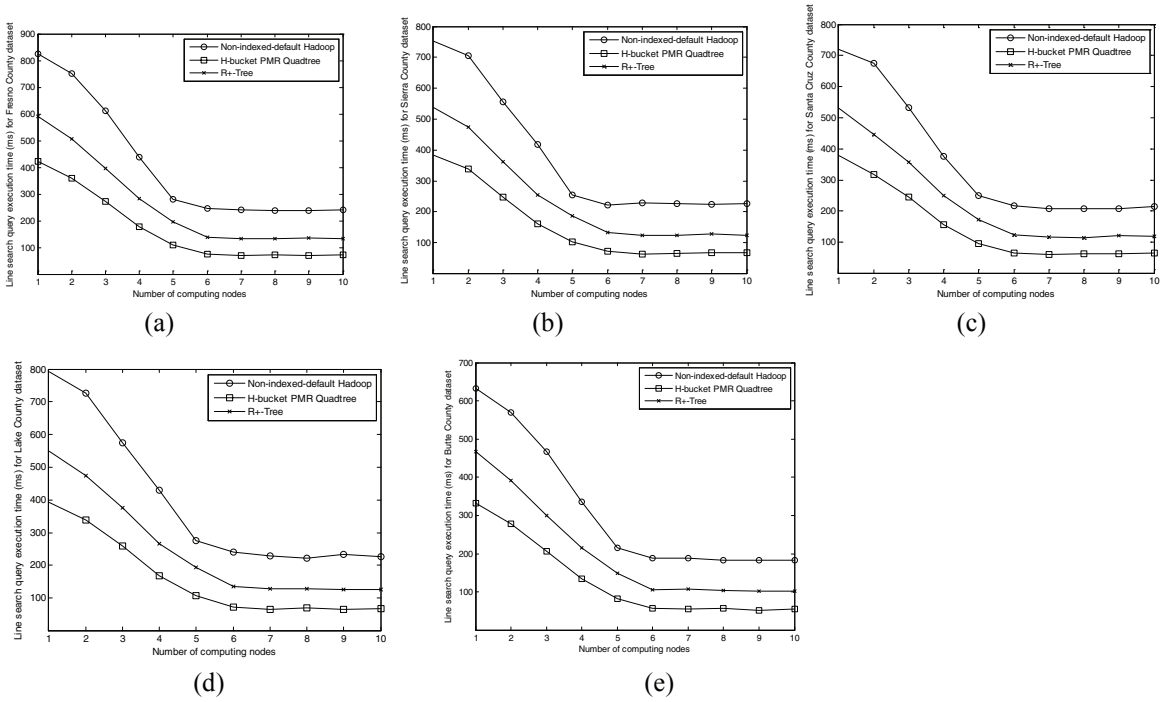


Figure 6 Line search query execution time of the two indexes and non-indexed Hadoop (a) Fresno County (b) Sierra County (c) Santa Cruz County (d) Lake County (e) Butte County.

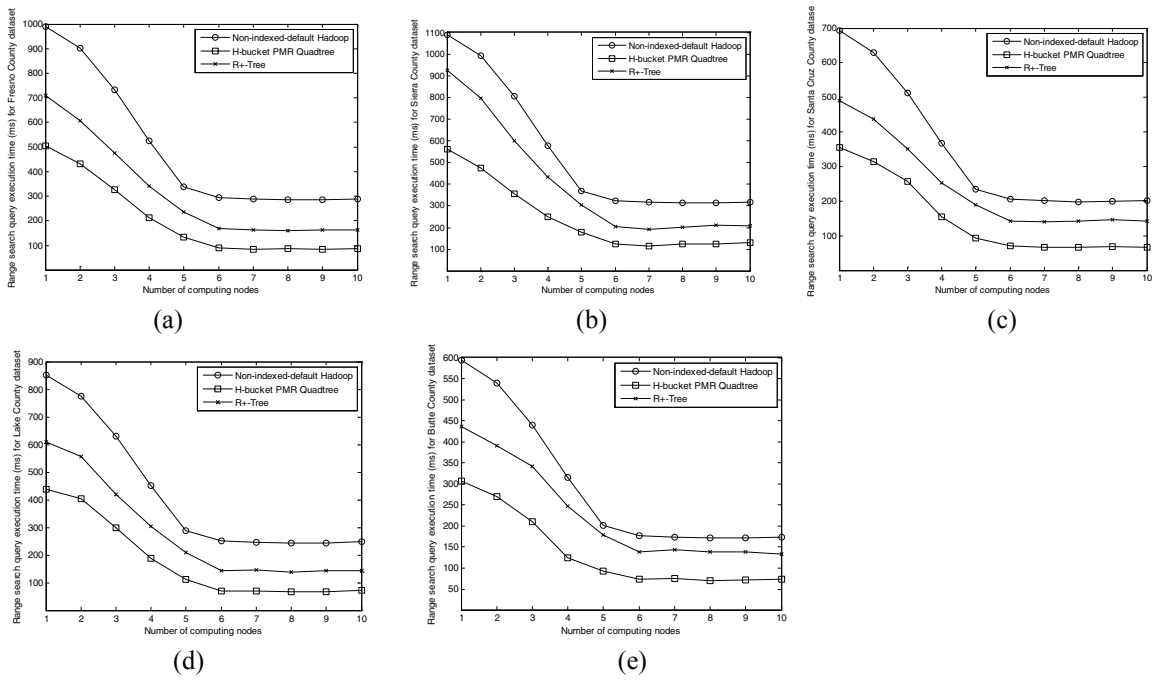


Figure 7 Range search query execution time of the two indexes and non-indexed Hadoop (a) Fresno County (b) Sierra County (c) Santa Cruz County (d) Lake County (e) Butte County.

then these nodes sit idle and do not contribute in reducing execution time. Figure 6(a–e) shows line search query execution time for non-indexed-Hadoop and the two indexes. A significant performance gain is observed for running queries once indexes are established. A similar kind of observations is noticed for range queries shown in Figure 7(a–e).

It is observed that H-bucket PMR Quadtree query execution time is better by 1-4-1.9 times than R+-tree and R+-tree query execution time is better by 1-4-1.8 times as compared to default non-indexed Hadoop. In overall, query performance becomes better for index-dataset and the proposed index is better than R+-tree index in MapReduce.

6. CONCLUSIONS AND FUTURE SCOPE

The proposed parallel index, H-bucket PMR Quadtree, is an implementation of existing bucket-PMR Quadtree in MapReduce. The index is inclined towards processing curvilinear spatial objects and ensures an improved efficiency for index build-time and search queries. It is compared with the MapReduced based state-of-the-art R+-tree implementation (Zhong et al. 2012). The experimental demonstration for line and range search queries proves superiority of the proposed index. The proposed index is built as a local index by all participating computing nodes in the cluster, and is accessed through a global index maintained by master node.

The proposed index strongly supports regular decomposition based approach in Quadtrees that requires less time to split an overflowing node for improving index building time over irregular disjoint decomposition based approach, such as R+–tree. The comparison of indexed approaches with non-indexed-Hadoop implementation shows that indexing improves data access in the Hadoop. Further, index building time and query execution time improve with increasing bucket size and cluster size. It is found through experimental analysis that for polygon map dataset for the five counties of Califonia state, the index build time of the proposed index is better by 20–50% than R+-tree in MapReduce. The index build time is best for tree node/bucket size of 64KB, which is the data-transfer size of packets in the Hadoop. Similarly, the proposed parallel index is more efficient than R+-tree in MapReduce for search queries and takes approximately 30–50% less time as compared to latter one. The proposed index is more efficient than default non-indexed Hadoop by 50–70% for search queries.

In future, we wish to test scalability of H-bucket PMR Quadtree for large datasets to further validate the implementation and search/develop a more efficient spatial index algorithm in MapReduce environment.

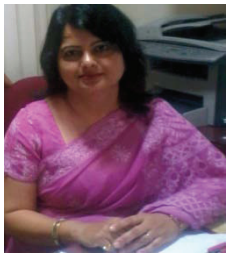
REFERENCES

1. Achakeev, D. et al., 2012. Sort-based parallel loading of R-trees. In *1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, ACM*. pp. 62–70.
2. Antonin Guttman, 1984. R-trees: A dynamic index structure for spatial searching. *ACM Sigmod Record*, 14(2), pp.47–57.
3. Apache, 2015. Hadoop. *http://Hadoop.apache.org, (Accessed on April 16, 2015)*.
4. Beckman, N. et al., 1990. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. pp. 322–331.
5. Bestul, T., 1992. *Parallel paradigms and practices for spatial data*. University of Maryland.
6. Blelloch, G.E. & Little, J.J., 1994. Parallel solutions to geometric problems on the scan model of computation. *Journal of Computer and System Sciences*, 48(1), pp.90–115.
7. Cary, A. et al., 2010. Leveraging Cloud Computing in Geodatabase Management. In *Proceeding of the 2010 IEEE International Conference on Granular Computing*. pp. 73–78.
8. Dean, J. & Ghemawat, S., 2013. MapReduce: SimplifiedDataProcessing on Large Clusters. *Communications of the ACM - 50th anniversary issue: 1958-2008*, 51(1), pp.107–113.
9. Eldawy, A. & Mokbel, M.F., 2013. A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data. *Proceedings of the VLDB Endowment*, 6(12), pp.1230–1233.
10. ESRI, 2016. Tiger Products - Geography U.S. Census Bureau. *www.esri.com/data/download/census2000-tigerline*.
11. Hayes, J.P. & Mudge, T.N., 1989. Hypercube Supercomputer. In *Proceedings of IEEE*. pp. 1829–1841.
12. Hoel, E.G. & Samet, H., 1992. A qualitative comparison study of data structures for large line segment databases. *SIGMOD Record*, 21(2), pp.205–214.
13. Hoel, E.G. & Samet, H., 1995a. Benchmarking spatial join operations with spatial output. In *Proceedings of the 21st International Conference on Very Large Databases*. pp. 606–618.
14. Hoel, E.G. & Samet, H., 2003. Data-parallel polygonization. *Parallel Computing*, 29(10), pp.1381–1401.
15. Hoel, E.G. & Samet, H., 1995b. Data-parallel primitives for spatial operations using PM Quadtrees. In *Proceedings of Computer Architectures for Machine Perception, doi:10.1109/CAMP.1995.521049*.
16. Hoel, E.G. & Samet, H., 1994a. Data-parallel spatial join algorithms. In *Proceedings of the 23rd International. Conference on Parallel Processing, doi:10.1109/ICPP.1994.82*.
17. Hoel, E.G. & Samet, H., 1994b. Performance of Data-Parallel Spatial Operations. In *Proceedings of the 20th International Conference on Very Large Data Bases*. pp. 156–167.
18. Jeong, S., Kim, S.-W. & Choi, B.-U., 2016. Effective indexing and searching with dimensionality reduction on high-dimensional space. *International Journal of Computer Systems Science and Engineering, CRL Publishing*, 31(4).
19. Jun, F. et al., 2014. HQ-Tree: A Distributed Spatial Index Based on Hadoop. *In China communications*, 11(7), pp.128–141.
20. Kamel, I. & Faloutsos, C., 1992. Parallel R-trees. *SIGMOD Record*, 21(2), pp.195–204.
21. Liao, H., Han, J. & Fang, J., 2010. Multi-dimensional Index on Hadoop Distributed File System. In *Fifth IEEE International Conference on Networking, Architecture, and Storage, IEEE Computer Society*. pp. 240–249.
22. Liu, Y. et al., 2009. Parallel bulk-loading of spatial data with MapReduce: An R-Tree case. *Wuhan University Journal of Natural Sciences*, 16(6), pp.513–519.
23. McCreadie, R., MacDonald, C. & Ounis, I., 2012. MapReduce indexing strategies: Studying scalability and efficiency. *Information Processing and Management*, 48(5), pp.873–888.
24. Nelson, R.C. & Samet, H., 1986. A consistent hierarchical representation for vector data. *Computer Graphics*, 20, pp.197–206.
25. Nievergelt, J., Hinterberger, H. & Sevcik, K., 1984. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)*, 9(1), pp.38–71.
26. Papadopoulos, A. & Manolopoulos, Y., 2003. Parallel bulk-loading of satial data. *Parallel computing*, 29(10), pp.1419–1444.
27. Samet, H., 1990. *The Design and Analysis of Spatial Data Structures*,
28. Schnitzer, B. & Leutenegger, S.T., 1999. Master-Client R-Trees: A new parallel R-Tree architecture. In *Proceedings of the 11th International Conference on Scientific and Statistical Database Management*. pp. 68–77.
29. Sellis, T., Roussopoulos, N. & Faloutsos, C., 1987. The R+-Tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th VLDB Conference*. pp. 507–518.
30. Singh, H. & Bawa, S., 2017. A MapReduce-based scalable discovery and indexing of structured big data. *Future Generation Computer Systems*, 73(August 2017), pp.32–43.
31. Singh, H. & Bawa, S., 2016. Spatial Data Analysis with ArcGIS and MapReduce. In *Proceedings of International conference on Conference Computing, Communication and Automation, IEEE*. p. doi:10.1109/CCAA.2016.7813687.

32. Tan, K.-L., Ooi, B.C. & Abel, D.J., 2000. Exploiting Spatial Indexes for Semijoin-Based Join Processing in Distributed Spatial Database. *IEEE Transactions on Knowledge and Data Engineering*, 12(6), pp.920–937.
33. Wang, Y. & Weng, S., 2010. Research and implementation on spatial data storage and operation based on Hadoop platform. In *Second IITA International Conference on Geoscience and Remote Sensing (IITA-GRS), IEEE, Vol. 2*. pp. 275–278.
34. Xun, L. & Wenfeng, Z., 2013. Parallel Spatial Index Algorithm Based on Hilbert Partition. In *International Conference on Computational and Information Sciences, IEEE*. pp. 876–879.
35. Zhang, C., Li, F. & Jestes, J., 2012. Efficient Parallel kNN Joins for Large Data in MapReduce. In *Proceedings of the 15th International Conference on Extending Database Technology, ACM*. pp. 38–49.
36. Zheng, X. et al., 2016. An optimization model of Hadoop cluster performance prediction based on Markov process. *International Journal of Computer Systems Science and Engineering, CRL Publishing*, 31(2).
37. Zhong, Y. et al., 2012. Towards Parallel Spatial Query Processing for Big Spatial Data. In *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), IEEE*. pp. 2085–2094.

Authors Profile

Hari Singh received his Bachelor of Engineering in Computer Science & Engineering (Honors) and Master of Technology in Computer Science & Engineering. He has been pursuing Ph.D. in Computer Science from Thapar University, Patiala, India. His areas of interest are Distributed Computing, Cloud Computing, Grid Computing, Big Data and, Programming and logic development. He has 15 years of teaching experience, including 6 years of research experience. He has been, presently working as Assistant Professor at Panipat Institute of Engineering & Technology, Panipat, Haryana, India. He is a member of IEEE, ISTE, and CSI.



Dr. Seema Bawa, holds M.Tech (Computer Science) degree from IIT Khargpur and Ph.D. from Thapar Institute of Engineering & Technology, Patiala. She is Professor, Computer Science and Engineering since 2004. Dr. Bawa headed the department for six and half years. The department has grown in all dimensions, including academics, research, manpower development and industry interaction. She has been Dean (Student Affairs) at Thapar University, Patiala for four years, from Sep 2010 to Sep 2014. As Dean (Student Affairs) she has made students excel in diverse skills and areas with determination and conviction. She has also been managing the entire network and IT infrastructure holding the additional role of Network Manager of Thapar University Campus for more than eleven years, since 1999 to 2011. Her areas of research interests include Parallel, Distributed, Grid and Cloud Computing, Energy aware computing and now Big Data Analytics and ensemble machine learning. Dr. Bawa has rich teaching, research and industry experience. She has also worked as a Software Engineer, Project Leader and Project Manager, in software industry for more than six years before joining the Thapar University. She has been Coordinator of three national level research & development projects sponsored by the Ministry of Information and Communication Technology. She is the author/co-author 125 research publications in SCI indexed journals and conferences of international repute. She has served as Advisor / Track chair for various national and international conferences. She has guided thirteen Ph.D.s, eight are ongoing. Understanding the importance of yoga and meditation in once life, Dr. Bawa is an active member and promoter of Sahaj Marg, meditation and Art of Living.