A Survey of Traditional and MapReduce-Based Spatial Query Processing Approaches

Hari Singh[*]
Computer Science and Engineering Department
N.C. College of Engineering
Israna, Panipat, Haryana, India
harirawat@rediffmail.com

Seema Bawa
Computer Science and Engineering Department
Thapar University
Patiala, Punjab, India
seema@thapar.edu

ABSTRACT
Various indexing methods of spatial data have come out after rigorous efforts put by many researchers for fast processing of spatial queries. Parallelizing spatial index building and query processing have become very popular for improving efficiency. The MapReduce framework provides a modern way of parallel processing. A MapReduce-based works for spatial queries consider the existing traditional spatial indexing for building spatial indexes in parallel. The majority of the spatial indexes implemented in MapReduce use R-Tree and its variants. Therefore, R-Tree and its variant-based traditional spatial indexes are thoroughly surveyed in the paper. The objective is to search for still less explored spatial indexing approaches, having the potential for parallelism in MapReduce. The review work also provides a detailed survey of MapReduce-based spatial query processing approaches - hierarchical indexed and packed key-value storage based spatial dataset. Both approaches use different data partitioning strategies for distributing data among cluster nodes and managing the partitioned dataset through different indexing. Finally, a number of parameters are selected for comparison and analysis of all the existing approaches in the literature.

Keywords
Spatial, Index, MapReduce, R-Tree

1. INTRODUCTION AND MOTIVATION
Support of high performance queries on spatial data has become important due to the large volume, high computational complexity of spatial data, and considerable time taken by complex spatial queries [70]. The representation of semi-structured spatial data in Well-Known-Text (WKT) and Well-Known-Binary (WKB) spatial data storage format, specified by the Open Geospatial Consortium (OGC) [4], makes it interoperable. Distributed spatial database systems offer a variety of spatial query functions and indexes for fast data retrieval but lacks in scalability [28]. Limitation of spatial databases for fixed schema and strict database norms does not make these suitable for handling big spatial data [63].

The distributed computing technology has witnessed high scalability and an excellent performance through its computational power. It has encouraged the evolution of modern parallel processing frameworks, such as the MapReduce-based Hadoop [1], HBase [2, 71], Cassandra [43] and BigTable [19]. The pros and cons of these frameworks are discussed in [9, 32, 37, 45]. These frameworks provide an excellent scope for scalability and high performance computational power over traditional stand-alone systems for processing a large amount of data [3, 30, 31, 56]. Recently, MapReduce parallel frameworks have been extensively used for dealing with semi-structured spatial data and related queries efficiently [8, 66, 72]. CG_Hadoop contains a suite of MapReduce algorithms for various computational geometry problems for dealing with large scale spatial data [21]. However, the key-value storage based techniques do not process spatial queries efficiently, as these require exhaustive searching. These techniques are able to scale, but cannot handle multi-dimensional spatial data [68]. Due to limitations of spatial databases and the key-value storage based distributed systems, integration of well known spatial indexing methods on MapReduce has evolved for improving the data access.

The research in the field of spatial index construction and spatial query has always been inspired by minimizing index construction and query execution time. Top-down or bottom-up approaches for well-known datasets, also known as batch-oriented methods [12, 39, 40, 44, 46, 59], over the slow and incremental methods [11, 14, 15, 29, 35, 41, 62] are the result of such motivation. Parallel processing techniques for the bulk loading spatial index and spatial query execution has continued this research trend. A single processor-multiple disk system [35], multiple processors-multiple disk system [55] and, now, a MapReduce-based systems [7, 8, 17, 18, 27, 47, 49, 65, 66, 68, 69, 70, 72] are the results of such researches.

Recently, a lot of work has been done for indexing spatial data and implementing spatial queries for fast data retrieval. Many algorithms, discussed in Section 2, are available for the same. The MapReduce framework for parallel processing is proven handy for operations requiring intense computing and improved execution time to a considerable extent. Through this survey, it has been found that in the last few years the MapReduce framework has been exploited in the field of spatial data. The Section 3 discusses research works, for parallelizing existing traditional spatial indexes, for speeding up spatial query execution. Probably, the most relevant review of the present survey work

[*]Hari Singh has been presently working as a faculty in Computer Science & Engineering Department at Panipat Institute of Engineering & Technology, Panipat, Haryana, India.

is the survey of a large-scale analytical query processing in MapReduce [20]. It has provided a very good classification of existing approaches for optimizing the performance of MapReduce and analyzed join queries in MapReduce. However, in the present paper, spatial data-oriented data access methods have been surveyed to 1) analyze the existing non-disjoint decomposition methods for bulk-loading spatial indexes, which forms an integral part for processing spatial queries based on spatial index methodology, and 2) analyze the work done so far in the domain of spatial query processing on MapReduce.

The rest of the survey is organized as follows. Section 2 reviews the existing traditional indexing approaches for spatial data. Section 3 presents classification and details of recent spatial query processing approaches, implemented in MapReduce, into two categories. The first category discusses a hierarchical indexed approaches on spatial dataset and the second category discusses key-value storage based indexed approaches. The two categories differ in the way spatial index is implemented on the partitioned dataset. The approaches in both categories mainly differ in the spatial data partitioning strategies on cluster nodes. Section 4 presents summary of the paper.

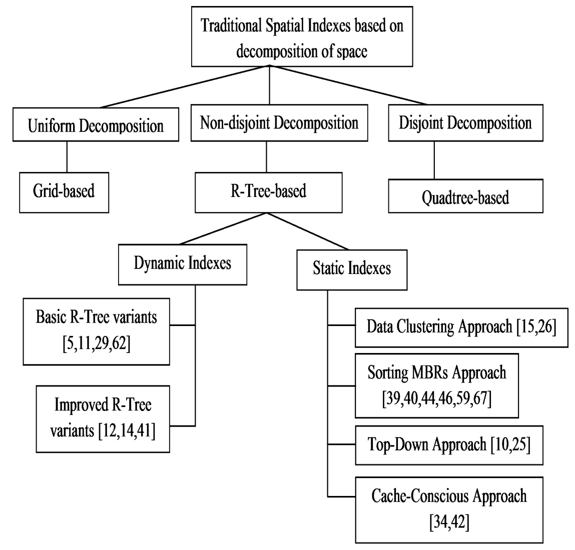
Traditional Spatial Indexes based on decomposition of space
Uniform Decomposition
Non-disjoint Decomposition
Disjoint Decomposition
Grid-based
R-Tree-based
Quadtree-based
Dynamic Indexes
Static Indexes
Basic R-Tree variants [5,11,29,62]
Improved R-Tree variants [12,14,41]
Data Clustering Approach [15,26]
Sorting MBRs Approach [39,40,44,46,59,67]
Top-Down Approach [10,25]
Cache-Conscious Approach [34,42]

Figure 1: Traditional Spatial Indexing Approaches

2. TRADITIONAL SPATIAL INDEXING APPROACHES

In this section, existing spatial indexes, for non-disjoint decomposition, in the serial programming environment, are discussed. These are categorized according to the approach used for building an R-Tree and its variants, as shown in Fig. 1. The intent is to identify a good spatial index, having the potential for parallelization. Various dynamic and static indexing techniques, in a serial programmed environment are discussed in Section 2.1 and 2.2, respectively. The R-Tree and its variant indexes have been explored thoroughly with regard to parameters, such as space utilization, insertion cost, spatial query performance for uniformly and non-uniformly distributed data, number of nodes to be searched for spatial query, applicability in high dimensions and worst-case performance. A summary of traditional spatial indexes describing support for various functionalities is presented in Table 1.

2.1 Dynamic Indexes

The dynamic index is built at run-time for dynamic data. The techniques mentioned in this section for spatial index structures, mainly work on one or a combination of more factors, such as coverage, margin and overlap, for creating index structure. Firstly, we discuss the basic R-Tree [5, 29], R*-Tree [11], and R+-Tree[62] and secondly, various improvements over these basic tree structures for optimizing the parameters and its effect on query execution [12, 14, 41].

2.1.1 The Basic R-Tree Variants

The index of a dynamic R-Tree provides a high load time, sub-optimal space utilization, and a poor R-Tree structure [5, 29]. The index takes a large search time due to high overlapping of rectangles. The R-Tree optimization metrics require enclosing rectangle to be of larger size and contains the maximum number of data rectangles as per the node capacity. This causes assignment of a large number of entries to a node, and consequently, a high overlap among nodes. The query performance of R-Tree deteriorates with skewed data, as it causes increased overlapping. The directory rectangles from the early-inserted data rectangles may not efficiently represent the current data. Various node splitting and re-insertion methods provide solutions to the problem that distinguishes dynamic variants of the R-Tree.

R*-Tree optimizes coverage, margin and overlap of enclosing rectangles in internal nodes for data insertion and node splitting [11]. It is due to the split and forced reinsert algorithms of the R*-Tree that preserves the proximity of smaller rectangles in a node. The R*-Tree has a better retrieval performance due to a better tree structure. A better structure of R*-Tree than the R-Tree causes the insertion cost of R*-Tree comparable to R-Tree for uniformly distributed data, but much better for skewed data. The execution time of the spatial-join queries improves for R*-Tree on processor time and Input/Output (I/O) [16]. The R+-Tree provides a zero overlap among intermediate nodes through a disjoint decomposition [62]. The search performance of R+-Tree, in terms of disk accesses, is more than 50% than R-Tree for point queries, but space consumption of R+-Tree structure is more than the R*-Tree due to disjoint search space.

2.1.2 The Improved R-Tree Variants

The improved R-Tree variants work towards a better node-split for minimizing overlap among partitioned MBRs. Node splitting algorithm in RR*-Tree considers, a degree of the balance of a split and the perimeter based strategy, apart from the criteria, coverage, margin and overlap considered in R*-Tree [12]. The RR*-Tree shows a better performance than R-Tree variants due to its overlap optimization at all directory levels. It becomes better for high dimensional space due to a good balance maintaining splitting algorithm and perimeter based optimization. However, the overlap and perimeter based optimization is more compute-intensive for insertions. The WeR-Tree achieves better space utilization

Table 1: A Summary of Traditional Spatial Indexes in Serial Programming Environment

Approach	Types	Index	A	B	C1	C2	C3	C4	C5	D	E	F	G
Dynamic indexes	Basic R-Tree variants	R-Tree [5, 29]	✓	✓	✓	✓	✓	✓	X	✓	✓	✓	X
		R*-Tree [11]	✓	✓	✓	✓	✓	✓	X	✓	✓	✓	X
		R+-Tree [62]	✓	✓	✓	✓	X	X	X	✓	✓	✓	X
	Improved R-Tree variants	Revised R*-Tree [12]	✓	✓	✓	X	X	X	X	✓	✓	✓	X
		WeR-Tree [14]	✓	✓	✓	✓	X	X	✓	✓	✓	✓	X
		X-Tree [41]	✓	X	✓	✓	X	X	X	X	✓	✓	X
Static indexes	Data clustering approach	cR-Tree using k-means clustering [15]	✓	✓	✓	X	X	✓	✓	✓	✓	✓	X
		R-Tree through iterative optimization [26]	✓	✓	✓	X	X	X	X	✓	✓	✓	X
	Sorting MBRs approach	Hilbert R-Tree [39, 40]	✓	✓	✓	✓	X	X	X	✓	✓	✓	X
		STR R-Tree [46]	✓	X	✓	✓	X	X	X	✓	✓	✓	X
		Lowx R-Tree [59]	✓	X	✓	✓	X	X	X	✓	✓	✓	X
		Hilbert-curve on a tree structure [44]	X	✓	X	✓	X	X	X	X	✓	✓	X
		MR-Tree [67]	✓	X	✓	✓	X	X	X	✓	✓	✓	X
	Top-down approach	TGS R-Tree [25]	✓	✓	✓	✓	X	X	X	✓	✓	✓	✓
		Priority R-Tree [10]	✓	✓	✓	✓	X	X	X	✓	✓	✓	✓
	Cache-conscious approach	CR-Tree [42]	✓	✓	✓	✓	X	X	X	✓	✓	✓	X
		CR-Tree variant [34]	✓	✓	✓	✓	X	X	X	✓	✓	✓	X

✓- Support for functionality exists and X - Support for functionality does not exist
A-Efficient storage utilization, B-Reducing insertion cost, Spatial query performance for uniformly distributed data: C1-Query Rectangle/ Enclosure query, C2-Point query, C3-Intersection query, C4-Spatial-join query, C5-Nearest-neighbor query, D-Effect of data skewness-a kind of non-uniform data distribution, E-Number of nodes to be searched for spatial query, F-Applicability in high dimension, and G–Worst-case performance

and search performance than the R*-Tree [14]. It uses a packing technique to organize its structure better than R*-Tree, however, it takes a significant amount of time to bulk-load and reconstructing a sub-tree of unbalanced node.

The packing causes data points to be stored uniformly in leaf nodes that lead to fewer activated paths for queries. The insertion strategy searches for an unbalanced node location to insert a new entry in the existing R-Tree and partially builds a sub-tree there by keeping nodes in balance. The R*-Tree splits nodes for minimizing the volume of the resulting MBRs and thus causes more overlap in high dimensions and reduces the efficiency of an index structure. The X-Tree introduced overlap-free split policy and high page capacity nodes, named Super-nodes [41]. The overlap-free node split along a particular axis uses split history for data insertion. The Super-nodes handle unbalancing of node-fill caused due to overlap-free split and store more entries as compared to simple nodes to provide more storage utilization.

2.2 Static Indexes

Index building with dynamic insertion algorithms provide a significant dead space in nodes and results in bad performance. R-Tree variants do not exploit known dataset during insertion. However, if R-Tree is built statically, then, space utilization improves, as heuristic pack the input data space. This section categorizes different static indexes for spatial data on the basis of heuristic packing used for building index and discusses the effect of packing spatial datasets for constructing R-Tree and variants [10, 15, 25, 26, 39, 40, 44, 46, 59, 67], as well as spatial query performance.

2.2.1 Data Clustering Approach

The clustering technique splits spatial objects in the nodes on the basis of spatial proximity according to some parameter to minimize data access time. The K-means clustering technique is used for constructing the cR-Tree [15]. The k-means algorithm is order independent, unlike linear split heuristic of R-Tree and, time and space complexity of K-means is analogous to linear split algorithm of R-Tree. It uses multi-way split procedure than the traditional two-way split procedure for realizing an efficient R-Tree. The low index building time of cR-Tree is due to significant time saving on following a simple insertion algorithm as compared to the one used in R*-Tree. In another D-dimensional and batch oriented packing, the dimensional sort curve builds R-Tree by partitioning the D-dimensional data space into K partitions such that the volume of all the enclosing rectangles is minimized [26]. Though the linear packing methods are fast, but the D-dimensional approach better packs the data. It takes into account positions and spatial extents of objects in all dimensions that are achieved by a linear method. The batch oriented methods follow a bottom-up approach level by level, and consequently, achieves a high degree of parallelism. The method is poor, as it uses a large number of disk accesses and the efficiency deteriorates with data skew and dimensionality. The clustering method for constructing R-Tree in high dimensions is compute intensive as compared to R*-Tree and Hilbert R-Tree, but it performs better than the two on query execution time. It is because the latter two assign rectangles from different clusters to the same R-Tree node [15, 26].

2.2.2 *Sorting MBRs Approach*

One class of R-Tree indexes is bulk-loaded by sorting the MBRs either along one dimension or both dimensions in a two dimensional space. The tree leaves are filled-up first and, then, the rest of the index is built step-by-step in a bottom-up manner [39, 40, 44, 46, 59, 67]. In one such approach, the correspondence between points on space-filling curve, Hilbert-curve, and their sequence numbers are expressed as a tree structure [44]. It provides an overlap free tree node structure for the Hilbert-curve of a particular order. However, it is impractical to store the mapping of space filling curves to a tree representation explicitly and the traversal from the root to a leaf takes excessive node accesses.

In another approach, the Lowx R-Tree, a packed R-Tree for static environment, provides a simple method of packing spatial data using a dimension sort curve [59]. It sorts the rectangles with their x or y coordinates of one of the corners of the rectangle. It provides thin, long bounding rectangles along one dimension that results in nodes having less area but large perimeter. It performs well for point queries, but not so well for larger queries, such as region queries. The performance of queries decreases for skewed data. The solution to the problem was obtained by applying sorting and partitioning step for each of the dimensions [46]. The authors presented a Sort-Tile-Recursive (STR) packing algorithm to improve load time, space utilization and data retrieval efficiency of R-Tree. In another two-tier index MR-Tree, a combination of grid index and STR R-Tree index, two disk accesses take the search to a local STR R-Tree [67]. It reduces the search space and the number of node accesses in the MR index. However, the MR index is inferior to STR index in terms of spatial efficiency of the index. It is due to the low spatial efficiency of the grid index. The I/O cost of MR-Tree is lower as compared to STR-Tree for similar reason.

The Hilbert-curve based packing shows higher performance for uniformly and skewed data by minimizing area and perimeter of R-Tree leaf nodes [39, 40]. A slight variation of it sort MBRs on the basis of the Hilbert value of the center of rectangles for constructing R-Tree [40]. The nodes of the tree, put similar MBRs together and minimize the area and perimeter of MBRs under one node and achieve high space utilization. This approach brings proximity to the data objects in R-Tree nodes, and consequently, provides more space utilization by reducing the perimeter and area of the nodes. The Hilbert R-Tree performs better for all types of data than R*-Tree in terms of the number of node accesses. It achieves a high space utilization but the insertion time is comparable to R*-Tree due to ordering of data according to Hilbert-curve. The STR and Lowx R-Tree are better than Hilbert-curve based R-Tree for uniformly distributed points and region data [46, 59]. It is because the indexing methods based on space-filling curves (SFC) for R-Tree construction do not preserve spatial locality well and produce approximate results. For the same reason, STR-based R-Tree is much better than Hilbert-curve based R-Tree for skewed data for point and region queries. However, Lowx-based R-Tree performs poorly because of poor packing of data.

2.2.3 *Top-Down Approach*

One class of R-Tree indexes is bulk-loaded in a top-down manner. The R-Tree index is constructed in two steps: firstly, a good partition of the data is generated recursively, and secondly, the index is built from root to leaf. A Top-down Greedy Split (TGS) algorithm divides input dataset into two subsets through a recursive split procedure and constructs R-Tree in a top-down manner [25]. The split applies heuristic, such that the cost of some objective function on MBRs of each split subset is minimized and each subset has sufficient number of rectangles, so that resulting sub-trees are packed. The bulk-loading in TGS R-Tree requires more I/Os as compared to other R-Tree variants, since it scans all the rectangles to make the partition decision. However, TGS R-Tree performs comparable to Hilbert R-Tree and STR R-Tree on uniformly distributed data, and outperforms the latter two for large rectangles and skewed data, for point and range query. In another top-down approach, a Priority R-Tree is built from the priority leaves, that contain extreme rectangles along each dimension of the dataset, and the rest of the rectangle is further divided into two subsets of approximately equal size, and pseudo PR-Tree is constructed recursively [10]. The PR-Tree bulk-loading algorithm executes a window query in the optimal number of O($(N/B)^{1-1/d}$+T/B) I/Os in the worst case as compared to other R-Tree bulk loading methods, where N is the total number of dataset rectangles in the R-Tree, B is the block size of the disk, and T is the output size. The PR-Tree outperforms all the others for window query on highly skewed data. The bulk-loading time of PR-Tree is more than the TGS R-Tree. However, the window query performance of PR-Tree is slightly better than the TGS R-Tree.

2.2.4 *Cache-Conscious Approach*

One class of R-Tree index has focused on cache-conscious indexes, similar to the cache-conscious B+-Tree [58], to optimize R-Trees [34, 42]. A cache-conscious version of R-Tree, CR-Tree, uses compressed MBR keys as indexed keys to obtain a wider and smaller R-Tree [42]. The compression is done with a Quantized Relative Minimum Bounding Rectangle (QRMBR) technique and the output is quantized. The QRMBR compresses the MBR keys by representing a child MBR relatively to its parent MBR. The compression and quantization technique used has the drawback that the false hits increase. However, selecting a proper quantization level, false hits are reduced. The authors found that in two, three and four dimensions, the number of node accesses for CR-Tree is smaller than R-Tree and the performance of CR-Tree improves with increasing node size. The number of cache misses is also smaller for CR-Tree in comparison to R-Tree in all dimensions. However, the cache miss graph initially decreases for a rise in node size in certain node size, and thereafter, the graph declines with a rise in node size. The cause of such a shape is the increased overhead due to a large node size that costs more than the gain obtained due to wider and smaller R-Tree. The solution to the problem was proposed by reducing the amount of L2 cache misses in the cache-conscious QRMBR R-Tree variants for better memory utilization and improved query performance [34]. The authors introduced Optimistic Latch Free Index Traversal (OLFIT) technique to overcome the cache miss problem of conventional index concurrency control by using a version and a latch in each node.

3. SPATIAL QUERY PROCESSING APPROACHES IN MAPREDUCE

In the past, parallelization of bulk-loading spatial indexes and spatial querying is achieved through one processor in communication with multiple disk architecture [35] and a shared-nothing architecture [55]. The MapReduce programming model offers a new distributed environment for parallel processing that provides high efficiency for executing tasks [8, 66, 72]. However, the MapReduce framework incurs high data transfer overhead, which need to be dealt carefully [8, 47, 49, 53]. A lot of work is found in literature that considers different methods of spatial query processing from the serial programming model and revising these in distributed environments, especially in MapReduce environment, for parallelization. The performance of spatial queries is found better over the indexed dataset as compared to the default hashing-based key-value storage in the Hadoop [2, 16, 50, 60]. In this section, spatial query processing approaches in MapReduce, based on a hierarchically indexed (Section 3.1) and packed key-value storage based (Section 3.2) approaches, have been surveyed. Both approaches use different spatial data partitioning methods, as shown in Figure 2, and then organizing spatial index on the partitioned dataset. The hierarchically indexed dataset uses uniform data partitioning [6, 22, 23, 24], random-sampling-based data partitioning [6, 22, 23, 24], clustering-based data partitioning [17], space-filling-curve based data partitioning such as Hilbert-curve based [47, 49, 66, 68], STR packing based [47], Z-curve based [17, 18] and X-mean algorithm based [17], Quadtree-based spatial data partitioning, such as Quadtree-based recursive tile partitioning for R*-Tree indexing [7, 8], a quadtree partitioning and Hilbert-curve based local indexing [72], quadtree-based data partitioning for implementing the PR-Quadtree based local index in MapReduce[38]. The key-value storage basde approaches are based on uniform data partitioning [27], SFC-based space partitioning [65, 69, 70] and spatio-temporal partitioning [52].

A survey of the two approaches for various functionalities is presented in Table 2. It describes whether the research works under the approaches provides support for functionalities. The functionalities considered under the survey are spatial proximity of distributed spatial data on cluster nodes, index build-time, efficiency of query execution, load balancing, data transmission overhead through network, applicability in high dimensions, latency for random access for large number of concurrent reads, latency for sequential access for large number of concurrent reads, effect of cluster scaling on query execution, effect of index-node size, effect of packet-data size and performance for non-uniformly distributes dataset.

3.1 Spatial Query Processing on Hierarchically Indexed Spatial Dataset

MapReduce speeds-up bulk-loading of spatial index and query execution. The benefits of building spatial index, such as R-Tree in MapReduce framework is that MapReduce abstracts data load-balancing, process scheduling and fault tolerance from the application logic, and manages transparently [18]. Otherwise, a lot of difficulty was involved in managing these distributed computing aspects earlier [55, 61].

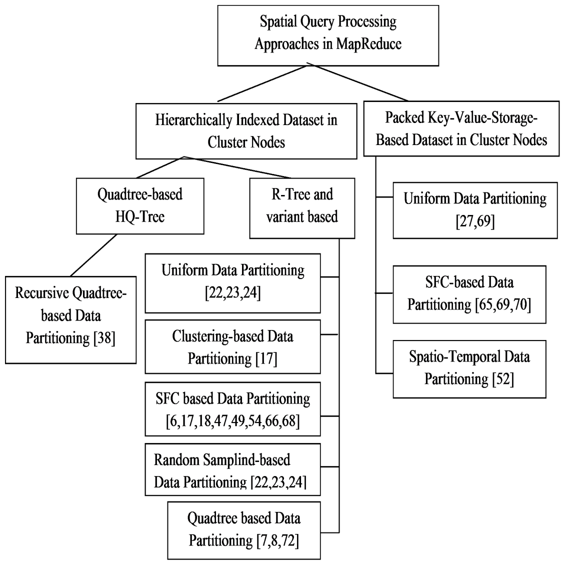
Spatial Query Processing Approaches in MapReduce
Hierarchically Indexed Dataset in Cluster Nodes
Packed Key-Value-Storage-Based Dataset in Cluster Nodes
Quadtree-based HQ-Tree
R-Tree and variant based
Recursive Quadtree-based Data Partitioning [38]
Uniform Data Partitioning [22,23,24]
Clustering-based Data Partitioning [17]
SFC based Data Partitioning [6,17,18,47,49,54,66,68]
Random Samplind-based Data Partitioning [22,23,24]
Quadtree based Data Partitioning [7,8,72]
Uniform Data Partitioning [27,69]
SFC-based Data Partitioning [65,69,70]
Spatio-Temporal Data Partitioning [52]

Figure 2: Spatial Query Processing Approaches in MapReduce

The MapReduce framework is also enhanced due to use of spatial indexes on MapReduce, as these improve latency for random reads [72].

This section discusses spatial query processing on the basis of various hierarchical indexes on spatial dataset. The hierarchical indexes mainly differ in data partitioning strategies and building spatial indexes on the partitioned dataset.

3.1.1 *Uniform Data Partitioning*

The method divides input spatial space into equal sized rectangles depending on number of mappers in MapReduce. The spatial data is partitioned into n rectangles, and the data that overlap in rectangles is redundantly assigned to overlapping rectangles. The number and size of rectangles are decided by the number of partitions required for input data. Each partitioned data are taken by a cluster node and processed there. For a grid index, the input space is partitioned in $\sqrt{n}$ x $\sqrt{n}$ rectangles of uniform size and the number of partitions (n) is calculated by dividing the input data size with HDFS block size [22, 23, 24].

Then, each slave node builds a local index in memory and writes it to disk. Lastly, a global index is built by master node. The index building time is very small due to the simple process and computations involved. The uniform or rectilinear space partitioning approach is easy to implement, but it causes non-uniform data distribution among cluster nodes for processing in MapReduce, especially for non-uniformly distributed and skewed dataset. This affects load balancing and hence efficiency of queries. Some of the nodes complete their task early and sit idle, waiting for other heavily loaded nodes to complete their tasks. The efficiency of spatial queries is not very good, as the data are unorganized and it takes a lot of time to search query data. The method works fine for uniformly distributed data, but performs poorly for non-uniformly distributed and skewed data.

Table 2: A Summary of Hierarchically Indexed and Packed Key-Value Storage Based Spatial Dataset in MapReduce

Approach	Data partitioning	Index realized on MapReduce	A	B	C1	C2	C3	C4	D	E	F	G	H	I	J	K	L
Hierarchically indexed	Uniform and Random-sampling-based [22, 23, 24]	Grid index	X	X	X	X	✓	X	X	✓	X	X	X	✓	X	X	X
	Clustering-based (x-mean) [17]	R-Tree	✓	✓	X	X	X	X	X	X	X	X	X	X	X	X	✓
	SFC-based (Z-curve) [17]	R-Tree	✓	✓	X	X	✓	X	X	X	X	X	X	X	X	X	X
	SFC-based (Z-curve) [18]	R-Tree	X	✓	X	X	X	X	X	X	X	X	X	✓	X	X	X
	SFC-based (Hilbert-curve) [47]	R-Tree on STR packing	✓	X	✓	X	✓	X	X	✓	✓	✓	✓	X	✓	X	X
	SFC-based (Hilbert-curve) [49]	R-Tree	✓	X	✓	X	X	X	✓	✓	X	X	X	X	X	X	X
	SFC-based (Hilbert-curve) [66]	R-Tree	✓	✓	✓	X	X	X	X	X	X	X	X	X	X	X	X
	SFC-based (Hilbert-curve) [68]	R-Tree	✓	✓	X	X	X	X	X	X	X	X	X	X	X	X	X
	Quadtree-based recursive tile partitioning [7, 8]	R*-Tree	X	✓	✓	✓	✓	X	✓	✓	X	✓	X	✓	X	X	X
	Quadtree-based [72]	R-Tree	✓	✓	✓	✓	✓	X	X	X	✓	✓	✓	✓	X	X	X
	Quadtree-based [38]	PR-Quadtree	X	✓	✓	X	X	X	✓	✓	X	✓	X	✓	✓	X	✓
Based on packed key-value storage	Uniform [27] (uses Controlled-Replicate approach)	Default key-value pair	X	X	X	X	✓	X	✓	✓	X	✓	X	X	X	X	X
	Uniform [69] (uses H-BRJ and H-BNLJ) and SFC-based (uses H-zkNNJ)	Default key-value pair	X	X	X	X	✓	X	X	✓	X	✓	X	✓	X	X	X
	SFC-based [65] (uses PBSM)	Default key-value pair	✓	X	X	X	✓	X	✓	✓	X	✓	X	✓	X	X	X
	SFC-based [70] (SJMR)	Default key-value pair	X	X	X	X	✓	X	✓	X	X	✓	X	✓	X	X	X
	Hybrid: spatio-temporal [52]	PMI- and OMI-based key-value pair	✓	✓	✓	✓	✓	✓	✓	X	X	X	X	✓	X	X	X

✓- Support for functionality exists and X - Support for functionality does not exist
A-Evaluating spatial proximity of distributed spatial data on cluster nodes, B-Index build-time, Efficiency of query execution: C1-Spatial Selection(Point, Line, Window and Range search query), C2-Spatial Aggregation, C3-Spatial-join, C4-Spatio-temporal, D-Load balancing: data distribution among cluster nodes, E-Data transmission overhead through network, F-Applicability in high dimensions, G-Latency of random access for large number of concurrent reads, H-Latency of sequential access for large number of concurrent reads, I-Effect of cluster scaling on query execution, J-Effect of index-node size, K-Effect of packet-data size, and L-Performance for non-uniformly distributed dataset

3.1.2 Clustering-Based Data Partitioning

The method partitions spatial objects into groups according to their spatial clustering. A comparison between the Z-curve based data partitioning and the X-means clustering-based data partitioning has been done for parallel R-Tree construction [17]. It is observed that the Z-curve has a linear complexity of the mappers input and generates almost equal sized partitions, but the spatial locality is not always well preserved. The X-means based iterative clustering algorithm uses Bayesian Information Criteria (BIC) score to rank clusters according to the Gaussian distribution. In this scheme, though the number of partitions is estimated but the size of partition varies considerably and iterations cause expensive computations. For nearest-neighbor queries, the X-means better approximates spatial data distribution and reduces overlapping as compared to the Z-curve which directly relates to data retrieval efficiency. But, the X-means takes almost double time for R-Tree index creation than the Z-order and significant time is elapsed in the clustering phase.

3.1.3 Random-Sampling-Based Data Partitioning

In this approach, random sampling of spatial data is distributed among clustered nodes. The SpatialHadoop framework uses it for implementing R-Tree and R+-Tree [23, 24]. The bulk-loading of indexes is done using an STR packing technique. Partitioning the data of input file is guided by the boundaries of the leaf node. The index building time of R-Tree and R+-Tree is more as compared to the grid index due to the complexity of index building process and computations involved [22]. However, the efficiency of spatial query is very good, as data is indexed and query data search time is low. The method works fine for uniformly distributed data, but does not work that well for non-uniformly distributed and skewed data.

3.1.4 SFC-Based Data Partitioning

The space-filling curve is used to transform multi-dimensional location information into one-dimensional space. A Z-curve based uniform data partitioning is used for data partitioning during the map-phase [17, 18, 49]. In MD-HBase, a scalable multi-dimensional data store on HBase, a similar approach is used to distribute the data on cluster nodes. Multidimensional index structures, K-d Tree and Quadtree, are implemented on the partitioned dataset for demonstrating the scalability and efficiency of range and kNN queries [54].

A similar SFC-based approach for bulk-loading R-Tree on MapReduce uses the Hilbert-curve [66, 68]. The partitioning function puts objects in same partition to keep spatial proximity by using the sorted MBR values of object nodes from the Hilbert-curve and transforms to a standard and proven multi-dimensional index structure, R-Tree, through parallelizarion in MapReduce. In another SFC-based approach, parallel-gopt (p-gopt), a parallel R-Tree is built on the SFC and gopt-partitioned dataset [6]. The leaf nodes of R-Tree are filled-up in order to minimize a cost function named gopt-loading, rather than filling-up nodes to the maximum. Initially, the input dataset is sorted, in parallel, according to a space-filling curve. The sorted sequence is partitioned into sub-sequences according to gopt-partitioning method. The method makes sub-sequences of sizes between b (lower limit) and B (upper limit) according to a cost function, where each sub-sequence corresponds to a leaf node. The bulk-loading time of R-Tree using the p-gopt partitioning is more than the other parallel R-Tree construction approaches in MapReduce. However, the method outperforms other parallel R-Trees for average spatial queries in terms of node accesses.

The packing algorithms, such as STR and Hilbert packing guarantee the proximity of spatial data in R-Tree leaf nodes to reduce query response time and data transfer overhead, through network [47]. The buffer management and R-Tree node size further improves query efficiency. The buffer management speeds-up data access by keeping less space occupying internal nodes in the buffer to 1) minimize the disk access 2) avoid the bottleneck caused in case of concurrent access. A limited number of leaf nodes are permitted in the buffer, depending on space availability to further reduce the disk access time. A large index node size reduces data transfer overhead for two reasons. 1) High I/O costs of loading data from HDFS than local storage device 2) High cost of random reads than sequential reads in HDFS. The cost paid for low data transfer overhead and improved I/O is increased CPU effort for filtering more data objects. However, the space-filling curve based data partitioning approaches lose on preserving spatial locality due to a mapping from the higher dimensions to one-dimensional space, but it works better in high dimensions.

3.1.5 Quadtree-Based Data Partitioning

Quadtree-based data partitioning preserves spatial locality of objects and provides a uniform recursive decomposition of space into partitions until the number of objects in a partition are not more than a defined limit. The approach is highly suitable for parallel processing, but it is difficult to apply in high dimensions. Though, the performance of Quadtree-based indexes for index building and query processing is well established [13, 33, 36, 48]. It is due to the regular disjoint decomposition approach of Quadtree-based indexes which takes less index building and query processing time, as compared to non-disjoint and irregular disjoint decomposition approach, in R-Tree and variants. However, Quadtree-based approaches incur high data transfer and I/O costs [64].

One class of MapReduce-based approaches, for constructing R-Tree, uses a Quadtree-based space partitioning [7, 8, 72]. The VegaGiStore consists of a Quadtree-based global index and Hilbert-curve local index [72]. The former index finds data blocks and the latter locates spatial objects for efficient data retrieval with low latency access. It was observed that Quadtree-based regular disjoint decomposition technique, for spatial data partitioning, gives a stable performance for increasing k in kNN queries as compared to the other key-value storage systems, such as Hadoop, Cassandra, HBase, and the traditional spatial databases such as PostGIS, Oracle Spatial, etc. In another two-tier indexed approach, a global partition indexing for regions and local spatial indexing for objects in tiles, is used [7, 8]. The bulk-loading of spatial index is performed on each dataset by using the R*-Tree. It uses a recursive partitioning approach and multiple-assignment approach for load-balancing and boundary object problems, respectively. The indexing improves latency time of random read queries. The performance of spatial queries improve with the scalability of cluster, but it causes a high intermediate data transfer overhead. However, the proposed approaches have not considered the effect of index-node size and data-packet size.

A different technique, HQ-Tree, uses a recursive regular quadtree partitioning for handling point data [38]. It is a MapReduce implementation of PR-Quadtree index. It is free from order of data insertion and space overlap due to disjoint decomposition spatial occupancy approach. The efficiency of index creation in MapReduce environment is found better for both uniform and non-uniform data than over the standalone machine. It is good at dealing with skewed data of point objects for search queries. However, the storage of index is high due to disjoint storage of objects. The HQ-Tree approach has not been compared with other MapReduce-based non-disjoint decomposition approaches, such as R-Tree and variants. The approach is limited to spatial point objects and can be extended to other spatial data, such as lines, rectangles, and polygons in spatial data. The authors found an increase in read-time with increasing index-node size due to the rise of communication overhead with increasing node size. The read-time increases drastically when the size of index node becomes greater than the size of HDFS data packet i.e. 64 KB.

3.2 Spatial Query Processing on Packed Key-Value Storage Based Spatial Dataset

Packed key-value storage based indexes in MapReduce do not build a hierarchical index on partitioned dataset. These uses key-value pair on a partitioned dataset in the MapReduce framework as an index for spatial query processing. The hierarchical tree structures, such as R-Tree and its variants, are good for queries that access only a particular part of the dataset, such as range and region search. However, for complex spatial queries that require reading the dataset in a linear fashion, the packed key-value storage data performs better under conditions, such as the characteristic of data distribution. Various approaches that deal with complex spatial queries use different clustering methods, such as uniform data partitioning [27, 69], space-filling curve [65, 69, 70] and spatio-temporal [52], for packing input spatial objects.

3.2.1 *Uniform Data Partitioning*

Similar to the uniform data partitioning approach of hierarchical indexed spatial dataset, there are many techniques in the domain of packed key-value storage index which are based on uniform data partitioning. The Hadoop-Block R-Tree Join (H-BRJ) builds a parallel R-Tree index on one of the dataset for executing kNN query. It uses a uniform sized partitioning for distributing input data over the cluster nodes and builds an R-Tree index there. Similarly, a Hadoop-Block Nested Loop Join (H-BNLJ) approach does not use indexing on any dataset and use a nested loop for kNN join. Here also, the uniform data partitioning shows similar characteristics, such as ease of implementation and non-uniform distribution among cluster nodes that subsequently leads to poor load balancing and efficiency of queries [69].

In an advancement over the uniform data partitioning, the efficiency of spatial-join query is observed to improve drastically when undesired data, duplicate data and data that does not form a part of the query space, are eliminated [27]. It uses a Controlled-Replicate framework for running multi-way spatial-join, that controls the replication of rectangles and, avoids unnecessary replication and processing, and hence, reduces both communication I/O costs.

3.2.2 *SFC-Based Data Partitioning*

The space-filling curve based data partitioning approach in MapReduce arranges original input spatial dataset according to a SFC and partitions input spatial dataset into blocks of uniform size. A key-value storage index is applied to the packed partitioned dataset for spatial queries. A Z-value based SFC is used in MapReduce for handling kNN query (H-zkNNJ) [69]. The method reduces excessive communication and computation cost incurred by H-BNLJ and H-BRJ. The Z-curve based partitioning approximates the solution and requires only linear number of reducers. The advantage of this method is a linear communication and computation cost as compared to quadratic costs involved with baseline methods H-BNLJ and H-BRJ that use a quadratic number of reducers for kNN-join. The cost paid for lower computation and communication in H-zkNNJ is in terms of accuracy of query results, as Z-order based SFCs do not well preserve the spatial locality. The Z-order based H-zkNNJ performs better than R-Tree based H-BRJ for index building and querying with increasing number of reducers. It is because the size of data blocks decreases and a large number of smaller R-Trees are constructed in parallel that consequently increase building costs.

A double-transformation technique, PBSM [56], is implemented in a parallel programming environment in MapReduce [65, 70]. A pending file structure and redundant partition method are used to reduce communication overhead and to deal with the boundary objects problem, in MapReduce [65]. The authors observed that the quantity of buckets and tiles, tile coding method, and tile-to-bucket mapping strategy affect performance. Therefore, two SFCs, Z-curve and Hilbert-curve were used. The Z-curve used for tile coding provides weak position consistency, but the convenience of implementation. The Hilbert-curve provides a better position consistency, but needed intense computation. The two-dimensional plane sweeping technique lowers computation cost in the absence of an index, to accelerate computations. The approach performs better for the ANN query for parallel spatial databases, such as Oracle Spatial, and query performance improves with scalability.

The SJMR technique partitions dataset with disjoint partitions evenly at map function with a Z-curve tile coding method and a round-robin tile to partition mapping method [70]. The SJMR approach uses a duplication avoidance strategy, named reference tile method, to avoid replication overhead increased by spatial objects. It is present in tiles from multiple partitions by replicating these in all partitions. The Z-curve tile coding method in combination with a round-robin mapping scheme works as a spatial partitioning function. The reference tile method returns result pair for common smallest tile falling inside current partition and strip of two records. A strip-based plane sweeping method produces a superset of spatial-join result through overlapped MBRs. The performance of SJMR increases with an increase in the number of strips in the plane sweeping algorithm and with a number of nodes in the cluster. The SJMR performs better than the Parallel PBSM [65]. The SJMR carries out partitioning of the dataset and, then, elimination of duplicates in the map phase before a spatial-join is performed by a reduce task. A single MapReduce task carries out spatial-join, while the Parallel-PBSM uses two MapReduce tasks for executing spatial-join. Firstly, a map task performs data partitioning and a reduce task computes spatial-join. Secondly, the next MapReduce task eliminates duplication. The performance of both increases with increase in reduce task number up to a level, however, beyond it, the performance of both methods deteriorates as reduce task is not able to complete in one cycle.

3.2.3 *Spatio-Temporal Data Partitioning*

The spatio-temporal data represent spatial objects with respect to time, such as the trajectory of a moving object. It is represented as (x,y,t), where x and y are coordinates of an object and t represents a timestamp of an object at a specified position. A framework is described for query processing in sequential trajectory data of moving objects based on MapReduce [52]. The MapReduce framework is not suitable for handling continuously changing trajectory data as frequent updates are inefficient and costs too much in a cluster. A data partitioning strategy is also not applicable for maintaining continuity of trajectories.

The main problems are management of frequent updates to a trajectory data due to mobility of objects, data partitioning of skewed data and online query processing. The first problem is solved by maintaining new updated data in main memory at each node and writing to disk in batches when a particular size of the data is accumulated. The second problem of data partitioning is solved with a hybrid partitioning method. Some static and dynamic spatio-temporal space partitioning strategies are suitable for uniformly distributed and skewed data respectively, generated by a small number of moving objects. However, for massive moving objects, a highly skewed trajectory data consists of historic- static data and the new updated data. A hybrid method provides a solution by using individual static partitioning strategy for each time period. The key-value store in MapReduce is rearranged over partitioned dataset on cluster nodes to optimize query processing of range queries and trajectory based queries through Partition based Multilevel Index (PMI) and Object Inverted Index (OII). A good load balance, scalability of data importing, an index creation and query processing are achieved with a partitioning strategy with increasing number of computing nodes. However, the hierarchical tree structures avoid the exhaustive search over a provided input dataset for point query, range query and nearest-neighbor query.

4. SUMMARY

After comparing the surveyed approaches by means of classification criteria, some peculiar issues have been revealed which are thought to be relevant with respect to efficient spatial query processing. In the paper, these issues have been discussed with an intention to reflect its potential for further research. Many R-Tree variant spatial indexes for efficient spatial data handling exist, but not all have been used in the MapReduce framework. Basic R-Tree and its variant spatial indexes have been implemented extensively in MapReduce, as can be seen from the Table 2. However, many other existing spatial indexing techniques in a sequential programming environment, surveyed in Section 2 and summarized in Table 1 which perform better than the basic R-Tree variants, have not been implemented in MapReduce. It is learnt from Section 2 that approaches such as improved R-Tree variants, data clustering, sorting MBRs, top-down and cache-conscious approaches are superior to basic R-Tree variants. However, a lot of research work in implementing spatial indexes in MapReduce, presented in Section 3, has implemented basic R-Tree variants. Spatial indexes from other approaches have been rarely implemented. Presenting all spatial indexing approaches in detail in Section 2, motivates for their implementation in MapReduce.

The improved R-Tree variants are better than basic R-Tree variants [12, 14, 41]. The storage utilization, insertion cost and window query time of Revised R*-Tree is better than R*-Tree [12]. The applicability of Revised R*-Tree in high dimensions is more than the basic R-Tree variants [12]. Similarly, the storage utilization, insertion cost, performance of spatial queries such as window query, point query and nearest-neighbor query, of WeR-Tree is better than R*-Tree [14]. The applicability in high dimensions and spatial query performance for skewed data is also better than R*-Tree [14]. In a similar way, the X-Tree has proven better than R*-Tree for storage utilization, spatial queries such as window query and point query, and applicability in high dimensions [41].

The data clustering approach are better than basic R-Tree variants [15, 26]. The storage utilization, insertion cost, spatial query performance of queries, such as window query, point query and nearest-neighbor query of the cR-Tree, and performance of spatial queries for skewed data are better than R*-Tree [15]. However, the applicability of the index is low and comparable to the basic R-Tree variants [15]. The insertion cost and window query efficiency of another clustering approach, R-Tree using iterative optimization, is higher than R*-Tree [26]. However, the performance of the R-Tree using iterative optimization for window query is low and comparable to basic R-Tree variants when spatial dataset is skewed or is in high dimensions [26].

The sorting MBRs approach is better than basic R-Tree variants [39, 40, 44, 46, 59, 67]. The insertion cost, point query efficiency and applicability of expressing Hilbert-curve and their sequence numbers in a tree structure is better as compared to the R-Tree [44]. The storage utilization and, efficiency of the window and point query of uniformly distributed and skewed data, of Hilbert R-Tree is significantly higher as compared to R*-Tree. However, the insertion cost is comparable to R*-Tree and the applicability in high dimensions is low as compared to R*-Tree [39, 40]. The storage utilization, query efficiency and applicability in high dimensions, of Lowx R-Tree is better than R*-Tree but low as compared to Hilbert R-Tree [59]. The storage utilization and, performance of window and point query, of STR R-Tree is even better than Hilbert R-Tree. However, the query performance of skewed data and high dimensional data is comparable and lower than Hilbert R-Tree, respectively [46]. The storage utilization and insertion cost of MR-Tree is better than Hilbert R-Tree, but lower than STR R-Tree. The window and point query efficiency of MR R-Tree is better than STR R-Tree, however, the query performance of MR R-Tree for skewed data remains comparable to Hilbert R-Tree and the applicability in high dimensions is even lower than Hilbert R-Tree [67].

The performance of top-down approach, TGS R-Tree and Priority R-Tree, is better than all other approaches for all parameters [10, 25]. The insertion cost of the TGS R-Tree is even better than Priority R-Tree, however, the query performance of Priority R-Tree becomes better for skewed data. Both approaches have very good applicability in high dimensions and best worst-case performance. The cache-conscious approach shows better performance for all parameters as compared to basic R-Tree variants, comparable performance as compared to improved R-Tree variants, data clustering approach and sorting MBRs approach, and lower performance than top-down approaches [34, 42].

A significantly better bulk-loading and query execution time for all MapReduce-based spatial indexing approaches than traditional serial programming environment is strongly evident from the survey work carried out in Section 3. A comparison of the existing hierarchical indexed and packed key-value storage spatial index implementations in MapReduce, as presented in Table 2, for parameters, spatial proximity of distributed data on cluster nodes, index-build time, efficiency of query execution, load balancing, data transmission overhead through the network, and applicability in high dimensions, latency for random and sequential access for a large number of concurrent reads, effect of cluster scaling on query execution, effect of index-node size, effect of packet-data size and performance for non-uniformly distributed spatial dataset have been done. The former approach is better for random access spatial queries, such as search queries, while the latter approach is better for sequential access spatial queries, such as spatial-join queries. The spatio-temporal, spatial index is useful for queries, such as tracking mobile objects with respect to time. Among the hierarchical indexed dataset, the uniform data partitioning based grid index shows poor performance on all parameters, however, it is quite strong towards applicability in high dimensions.

In-depth analysis of spatial indexes in MapReduce, presented in Section 3, is not available as compared to the traditional serial programming environment, presented in Section 2. The comparison of spatial index implemented in MapReduce has been done with parallel spatial databases only, however, comparison with other spatial indexes implemented in MapReduce is rarely available. The performance of disjoint decomposition based indexes, Quadtree-based indexes, for index building and query processing is well established [13, 33, 36, 48]. The Quadtree-based approaches provide better spatial proximity and data distribution, efficiency for search queries and low network transfer overhead as compared to other data partitioning approaches, however, their storage requirement is more which is evident from their high index building time [7, 8, 72]. It has been analyzed from the survey that not much work has been done on implementing Quadtree indexes and Quadtree-based data partitioning in MapReduce. The quadtree partitioning based spatial indexes are the best, but these are very poor for the index building time and applicability in high dimensions [7, 8, 72].

In the survey, it has been seen that query processing is highly dependent on the size and nature of the dataset, and indexes show varying performance with different type of dataset. Besides it, the explored indexes in MapReduce have not been deeply analyzed for uniformly distributed, non-uniformly distributed and skewed data of varying sizes [6, 7, 8, 17, 18, 47, 49, 54, 66, 68, 72]. It has been observed that some of the existing implementations on MapReduce have not considered the effect of index-node size [7, 8, 17, 18, 22, 23, 24, 47, 49, 54, 66, 68, 72] and communication overhead [6, 17, 18, 22, 23, 24, 49, 54, 66, 68, 72]. The high communication overhead in MapReduce is due to its run-time scheduling scheme and the pull model that interfere the efficiency for queries [37, 57]. One method to reduce the communication overhead for the intermediate data generated during query processing in the Hadoop system is implemented in [51]. The authors used an independent distributed file system Parallel Secondo File System (PSFS) that avoids the transform and transfer of intermediate data through HDFS, and transfers data among database engines directly.

5. REFERENCES

[1] Hadoop. In *http://hadoop.apache.org*.

[2] HBase. In *http://hbase.apache.org*.

[3] OGC. In *http://www.opengis.orgltechno*.

[4] Performance Measurement of a Hadoop Cluster. In *http://www.acma.com/acma/pdfs /AMAX Emulex Hadoop Whitepaper.pdf*.

[5] R-Tree. In *http://en.wikipedia.org/wiki/R-tree*.

[6] D. Achakeev, M. Seidemann, M. Schmidt, and B. Seeger. Sort-Based Parallel Loading of R-Trees. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 62–70, 2012.

[7] A. Aji and F. Wang. High Performance Spatial Query Processing for Large Scale Scientific Data. In *Proceedings of the SIGMODPODS PhD Symposium*, pages 9–14, 2012.

[8] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, 2013.

[9] R. M. Arasanal and D. U. Rumani. Improving MapReduce Performance through Complexity and Performance Based Data Placement in Heterogeneous Hadoop Clusters. In *Proceedings of the International Conference Distributed Computing and Internet Technology*, pages 115–125, 2013.

[10] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree. *ACM Transactions on Algorithms*, 4(1), 2008.

[11] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *ACM SIGMOD Record*, 19(2):322–331, 1990.

[12] N. Beckmann and B. Seegar. A Revised R*-Tree in Comparison with Related Index Structures. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 799–812, 2009.

[13] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 975–986, 2010.

[14] P. Bozanis and P. Foteinos. WeR-Trees. *Data and Knowledge Engineering*, 63(2):397–413, 2007.

[15] S. Brakatsoulas, D. Pfoser, and Y. Theodoridis. Revisiting R-Tree Construction Principles. In *Proceedings of the 6th Springer East European Conference on Advances in Databases and Information System*, pages 149–162, 2002.

[16] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. *ACM SIGMOD Record*, 22(2):237–246, 1993.

[17] A. Cary, Y. Yesha, M. Adjouadi, and N. Rishe. Leveraging Cloud Computing in Geodatabase Management. In *Proceedings of the IEEE International Conference on Granular Computing*, pages 73–78, 2010.

[18] A. Cary, Zhengguo, V. Hristidis, and N. Rishe. Experiences on Processing Spatial Data with MapReduce. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management*, pages 302–319, 2009.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable-A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[20] C. Doulkeridis and K. Norvag. A Survey of Large-Scale Analytical Query Processing in MapReduce. *VLDB Journal*, 23(3):355–380, 2013.

[21] A. Eldawy, Y. Li, M. F. Mokbel, and R. Janardan. CG_Hadoop: Computational Geometry in MapReduce. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 294–303, 2013.

[22] A. Eldawy and M. F. Mokbel. SpatialHadoop. In *http://spatialhadoop.cs.umn.edu/*.

[23] A. Eldawy and M. F. Mokbel. A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data. *VLDB Journal*, 6(12):1230–1233, 2013.

[24] A. Eldawy and M. F. Mokbel. The Ecosystem of SpatialHadoop. *SIGSPATIAL Special*, 6(3):3–10, 2014.

[25] Y. J. Garcia, M. A. Lopez, and S. T. Leutenegger. A Greedy Algorithm for Bulk Loading R-Trees. In *Proceddings of the 6th ACM international symposium on Advances in geographic information system*, pages 163–164, 1998.

[26] D. Gavrila. R-Tree Index Optimization. In *Proceedings of the 6th International Symposium on Spatial Data Handling*, pages 771–791, 1994.

[27] H. Gupta, B. Chawda, S. Negi, T. A. Faruquie, and L. Subramanium. Processing Multi-Way Spatial Joins on MapReduce. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 113–124, 2013.

[28] R. H. Guting. An Introduction to Spatial Database Systems. *VLDB Journal*, 3(4):357–399, 1994.

[29] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *ACM SIGMOD Record*, 14(2):47–57, 1984.

[30] B. Hedlund. Understanding Hadoop Clusters and the Network. In *http://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/*.

[31] D. A. Heger. Hadoop Design, Architecture and MapReduce Performance. In *http://www.datanubes.com/mediac/HadoopArchPerfDHT.pdf*.

[32] E. Hoel and H. Samet. A Qualitative Comparison Study of Data Structures for Large Line Segment Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 205–214, 1992.

[33] E. G. Hoel and H. Samet. Performance of Data-Parallel Spatial Operations. In *Proceedings of the 20th International Conference on very Large Data Bases*, pages 156–167, 1994.

[34] S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee. Performance Evaluation of Main-Memory R-Tree Variants. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases*, pages 10–27, 2003.

[35] C. F. Ibrahim Kamel. Parallel R-Trees. *ACM SIGMOD Record*, 21(2):195–204, 1992.

[36] Jens, Dittrich, Jorge-Arnulfo, Quiane-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, 2010.

[37] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The Performance of MapReduce: An In-depth Study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.

[38] F. Jun, T. Zhixian, W. Mian, and X. Liming. HQ-Tree: A Distributed Spatial Index Based on Hadoop. *China communications*, 11(7):128–141, 2014.

[39] I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the 2nd International Conference on Information and Knowledge Management*, pages 490–499, 1993.

[40] I. Kamel and C. Faloutsos. Hilbert R-Tree: An Improved R-tree Using Fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509, 1994.

[41] D. Keim, B. Bustos, S. Berchtold, and H.-P. Kreigel. *Indexing, X-tree*. 2008.

[42] K. Kim, S. K. Cha, and K. Kwon. Optimizing Multidimensional Index Trees for Main Memory Access. *ACM SIGMOD Record*, 30(2):139–150, 2001.

[43] A. Lakshman and P. Malik. Cassandra-A Decentralized Structured Storage System. In *ACM SIGOPS Operating Systems Review*, pages 35–40, 2010.

[44] J. Lawder and P. King. Using Space-Filling Curves for Multi-Dimensional Indexing. In *Proceedings of the 17th British National Conference on Databases: Advances in Databases*, pages 20–35, 2000.

[45] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel Data Processing with MapReduce: A Survey. *ACM SIGMOD Record*, 40(4):11–20, 2011.

[46] S. Leutenegger, M. Lopez, and J.Edgington. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *Proceedings of the 13th IEEE International Conference on Data Engineering*, pages 497–506, 1997.

[47] H. Liao, J. Han, and J. Fang. Multi-Dimensional Index on Hadoop Distributed File System. In *Proceedings of the 5th IEEE International Conference on Networking, Architecture, and Storage*, pages 240–249, 2010.

[48] X. Liu, J. Han, Y. Zhong, C. Han, and X. He. Implementing WebGIS on Hadoop: A Case Study of Improving Small File I/O Performance on HDFS. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, 2009.

[49] Y. Liu, N. Jing, L. Chen, and H. Chen. Parallel Bulk-Loading of Spatial Data with MapReduce: An R-Tree Case. *Wuhan University Journal of Natural Sciences*, 16(6):513–519, 2011.

[50] M.-L. Lo and C. V. Ravishankar. Spatial Joins Using Seeded Trees. *ACM SIGMOD Record*, 23(2):209–220, 1994.

[51] J. Lu and R. H. Guting. Parallel Secondo-Boosting Database Engines with Hadoop. In *Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems*, pages 738–743, 2012.

[52] Q. Ma, B. Yang, W. Qian, and A. Zhou. Query Processing of Massive Trajectory Data Based on MapReduce. In *Proceedings of the 1st International Workshop on Cloud Data Management*, pages 9–16, 2009.

[53] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications*. 2006.

[54] S. Nishimura, S. Das, D. Agarwal, and A. E. Abbadi. MD-HBase, Design and Implementation of An Elastic Data Infrastructure for Cloud-Based Location Services. *Distributed Parallel Databases*, 31:289–319, 2013.

[55] A. Papadopoulos and Y. Manolopoulos. Parallel Bulk-Loading of Satial Data. *Parallel Computing*, 29(10):1419–1444, 2013.

[56] J. M. Patel and D. J. DeWitt. Partition Based SpatialMerge Join. *ACM SIGMOD Record*, 25(2):259–270, 1996.

[57] A. Pavlo, E. Paulson, A.Rasin, D. abadi, D. DeWitt, S. Madden, , and M. S. braker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the 35th ACM SIGMOD International Conference on Management of Data*, pages 165–178, 2009.

[58] J. Rao and K. A. Ross. Making B+- Trees Cache Conscious in Main Memory. *ACM SIGMOD Record*, 29(2):475–486, 2000.

[59] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-Trees. *ACM SIGMOD Record*, 14(4):17–31, 1985.

[60] H. Samet. *The Design and Analysis of Spatial Data Structures*. 1990.

[61] B. Schnitzer and S. T. Leutenegger. Master-Client R-Trees: A New parallel R-Tree Architecture. In *Proceedings of the 11th IEEE International Conference on Scientific and Statistical Database Management*, pages 68–77, 1999.

[62] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 507–518, 1987.

[63] S. Shekhar and S. Chawla. *Spatial Databases-A Tour*. 2003.

[64] K.-L. Tan, B. C. Ooi, and D. J. Abel. Exploiting Spatial Indexes for Semijoin-Based Join Processing in Distributed Spatial Database. *IEEE Transactions on Knowledge and Data Engineering*, 12(6):920–937, 2000.

[65] K. Wang, J. Han, B. Tu, J. Dai, W. Zhou, and X. Song. Accelerating Spatial Data Processing with MapReduce. In *Proceedings of the 16th IEEE International Conference on Parallel and Distributed Systems*, pages 229 – 236, 2010.

[66] Y. Wang and S. Weng. Research and Implementation on Spatial Data Storage and Operation Based on Hadoop Platform. In *Proceedings of the 2nd IITA International Conference on Geoscience and Remote Sensing*, pages 275 – 278, 2010.

[67] X. Wu and C. Zang. A New Spatial Index Structure for GIS Data. In *Proceedings of the 3rd IEEE International Conference on Multimedia and Ubiquitous Engineering*, pages 471–476, 2009.

[68] L. Xun and Z. Wenfeng. Parallel Spatial Index Algorithm based on Hilbert Partition. In *Proceedings of the IEEE International Conference on Computational and Information Sciences*, pages 876–879, 2013.

[69] C. Zhang, F. Li, and J. Jestes. Efficient Parallel kNN Joins for Large Data in MapReduce. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 38–49, 2012.

[70] S. Zhang, J. Han, Z. Liu, K. Hwang, and Z. Xu. SJMR: Parallelizing Spatial Join with MapReduce on Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, 2009.

[71] K. Zheng and Y. Fu. Research on Vector Spatial Data Storage Schema Based on Hadoop Platform. *International Journal of Database Theory and Application*, 6(5):85–94, 2013.

[72] Y. Zhong, J. Han, T. Zhang, Z. Li, J. Fang, and G. Chen. Towards Parallel Spatial Query Processing for Big Spatial Data. In *Proceedings of the IEEE 26th International Conference on Parallel and Distributed Processing*, pages 2085 – 2094, 2012.