International Journal of Digital Earth

ISSN: 1753-8947 (Print) 1753-8955 (Online) Journal homepage: https://www.tandfonline.com/loi/tjde20

A hierarchical indexing strategy for optimizing Apache Spark with HDFS to efficiently query big geospatial raster data

Fei Hu, Chaowei Yang, Yongyao Jiang, Yun Li, Weiwei Song, Daniel Q. Duffy, John L. Schnase & Tsengdar Lee

To cite this article: Fei Hu, Chaowei Yang, Yongyao Jiang, Yun Li, Weiwei Song, Daniel Q. Duffy, John L. Schnase & Tsengdar Lee (2020) A hierarchical indexing strategy for optimizing Apache Spark with HDFS to efficiently query big geospatial raster data, International Journal of Digital Earth, 13:3, 410-428, DOI: 10.1080/17538947.2018.1523957

To link to this article: https://doi.org/10.1080/17538947.2018.1523957

INTERNATIONAL JOURNAL OF DIGITAL EARTH
2020, VOL. 13, NO. 3, 410–428
https://doi.org/10.1080/17538947.2018.1523957

A hierarchical indexing strategy for optimizing Apache Spark with HDFS to efficiently query big geospatial raster data

Fei Hu[a,b], Chaowei Yang [a], Yongyao Jiang[a], Yun Li[a], Weiwei Song[a], Daniel Q. Duffy[c], John L. Schnase[d] and Tsengdar Lee[e]

[a]NSF Spatiotemporal Innovation Center and Dept. of Geography and GeoInformation Sciences, George Mason University, Fairfax, VA, USA; [b]Center for Open-Source Data and AI Technologies, IBM, San Francisco, CA, USA; [c]Office of Computational and Information Sciences and Technology, NASA Goddard Space Flight Center, Greenbelt, MD, USA; [d]NASA Center for Climate Simulation, Goddard Space Flight Center, Greenbelt, MD, USA; [e]Earth Science Division, NASA Headquarters, Washington, DC, USA

ABSTRACT

Earth observations and model simulations are generating big multidimensional array-based raster data. However, it is difficult to efficiently query these big raster data due to the inconsistency among the geospatial raster data model, distributed physical data storage model, and the data pipeline in distributed computing frameworks. To efficiently process big geospatial data, this paper proposes a three-layer hierarchical indexing strategy to optimize Apache Spark with Hadoop Distributed File System (HDFS) from the following aspects: (1) improve I/O efficiency by adopting the chunking data structure; (2) keep the workload balance and high data locality by building the global index (k-d tree); (3) enable Spark and HDFS to natively support geospatial raster data formats (e.g., HDF4, NetCDF4, GeoTiff) by building the local index (hash table); (4) index the in-memory data to further improve geospatial data queries; (5) develop a data repartition strategy to tune the query parallelism while keeping high data locality. The above strategies are implemented by developing the customized RDDs, and evaluated by comparing the performance with that of Spark SQL and SciSpark. The proposed indexing strategy can be applied to other distributed frameworks or cloud-based computing systems to natively support big geospatial data query with high efficiency.

ARTICLE HISTORY
Received 17 May 2018
Accepted 10 September 2018

KEYWORDS
Big data; hierarchical indexing; multi-dimensional; Apache Spark; HDFS; distributed computing; GIS

1. Introduction

In The era of 'Big Data', characterized by the 5 Vs, geospatial data are produced at an unprecedented rate by fast measurements of physical conditions, environmental observation, high-precision simulations of geophysical phenomena (Lynch 2008; Demchenko et al. 2013; Ma et al. 2015; Li et al. 2016a; Yang et al. 2017a). The geospatial raster data are generally managed in accordance with their spatiotemporal dimensions and attributes. Therefore, the multi-dimensional array-based data model is used to represent geospatial raster data in both industry and scientific communities (Rusu and Cheng 2013; Li et al. 2017a; Yang et al. 2017a). For example, an increasing amount of Earth observation and simulation data for long time recording of atmosphere, climate and oceans have been archived as multi-dimensional arrays in different data formats (e.g. HDF4/5, HDF-EOS, NetCDF3/4, GeoTiff) (Jiang, Sun, and Yang 2016). However, there still lacks an off-the-shelf

CONTACT Chaowei Yang cyang3@gmu.edu NSF Spatiotemporal Innovation Center and Dept. of Geography and GeoInformation Sciences, George Mason University, Fairfax, VA 22030, USA

solution for efficiently querying big geospatial raster data in terms of data pre-processing, uploading, locating, reading, and computing (Li et al. 2017a; Yang et al. 2017a; Hu et al. 2018a).

The traditional database systems (e.g. PostgreSQL and MySQL), although they have been widely applied for data management, are unable to handle big geospatial raster data. The difference between arrays and relations requires the complex data pre-processing. The relational database requires a relational schema for the input data with a formal structure. It means that geospatial raster data must be decomposed into points with attributes and loose spatiotemporal constraints, which results in the lacking of ordering functions to identify the specified data with dimensional indices. Several extra columns must be allocated to record the dimensional index for each row, and then the data size will be much larger than the original datasets (Hu et al. 2018b). Therefore, it is quite time-, computing- and disk- consuming for traditional database systems to query big geospatial raster data due to the complex data pre-processing (Cheng 2016; Hu et al. 2018b).

Recently, MapReduce-like high performance computing frameworks (e.g. MapReduce, Spark) coupled with distributed file systems (e.g. HDFS, Cassandra) have been adapted to deal with big data in a wide range of domains, including climate, biomedicine, finance, astronomy and social science, due to their parallel processing ability and scalability (Buck et al. 2011; Palamuttam et al. 2015; Wang et al. 2015; Li et al. 2016; Li et al. 2017a, 2017b; Yang et al. 2017a). Apache Spark, an advanced DAG (Directed Acyclic Graph) execution engine that supports cyclic data flow and in-memory computing, has become one of the most popular frameworks for big data processing as it powers a stack of libraries including Spark SQL, Spark MLlib for machine learning, GraphX for network analysis, and Spark Streaming. These libraries provide users with various high-level operators to build fast, parallel, in-memory computing applications. In the benchmarking of data containers for big geospatial data, the Spark-based frameworks have better performance than MongoDB, Rasdaman, SciDB, and Hive when processing large volumes of geospatial raster data (Hu et al. 2018b).

However, there still remain several issues in Apache Spark when querying large volume of geospatial data. First, it does not natively support geospatial data formats yet, but it is very time- and disk- consuming to convert big geospatial raster data to other formats (Hu et al. 2018b); Second, it does not support any indexing methods for fast locating the queried geospatial raster data. Without the index, it needs to traverse all input files to retrieve the target data. To improve the efficiency of Apache Spark on processing big geospatial data, a hierarchical indexing strategy for Apache Spark with HDFS is proposed with the following features: (1) improving I/O efficiency and computing parallelism of Spark on HDFS by adopting the chunking data structure; (2) keeping the workload balance and high data locality at the data-node level by building the global index (k-d tree); (3) enabling Spark and HDFS to natively support geospatial raster data formats (e.g. HDF4, NetCDF4, GeoTiff) without data pre-processing by building the local index (hash table) to index all chunks with detailed logical and physical data model information at the byte-, file-, and data-node levels; (4) caching and indexing the in-memory data to further improve the efficiency of geospatial data query; (5) developing a data repartition strategy to tune the query parallelism while keeping high data locality. The above strategies are implemented by developing the customized Resilient Distributed Datasets (e.g. ChunkID RDD, ChunkMeta RDD, Chunk RDD, IndexedChunk RDD). A proof-of-concept prototype demonstrates the feasibility and efficiency of the proposed hierarchical indexing strategy for handling big geospatial raster data.

Section 2 reviews the related index techniques and distributed systems for querying big geospatial raster data; Section 3 introduces the details of the proposed hierarchical index architecture and its implementation; Section 4 evaluates the performance of the proposed strategies by the comparing Spark SQL with SciSpark; Section 5 summarizes this research and discusses the limitations and future work.

2. Related work

2.1. Indexes for geospatial raster data

Index is an auxiliary data structure for the file system to quickly locate and reduce the disk-read data when answering range queries. For a multi-dimensional geospatial raster data query, the most common indices are the hash-table- and tree-like approaches.

In hash-table-like approaches, the key point is a function of the dimensions/attributes. One of the hash-table-like approaches, 'grid file', partitions the space of points along each dimension by a certain stripe into a grid, and points in a grid are sorted along that dimension. The region represented by a grid can be treated as a bucket of a hash table, and each point in the grid has its record along each dimension of the grid to locate the position (Hinrichs and Nievergelt 1983; Nievergelt, Hinterberger, and Sevcik 1984). The hash-table-like approach works best on range, partial-match and nearest-neighbour queries where data are uniformly distributed; however, its number of buckets increases exponentially with increases in dimension. Another hash-table-like method, partitioned hash functions, divides the bits of the key into n partitions representing the values for n dimensions/attributes, and each partition has its own hash function to generate the representative value. Such hash functions can be optimized for certain queries, including nearest neighbour and partial match queries (Gionis, Indyk, and Motwani 1999; Li et al. 2003).

For tree-like approaches, there are four common structures: multiple-key index, k-d tree, quad-tree and r-tree. The multiple-key index is an index of indexes, where nodes at each level are indexes for an attribute. A k-d tree (k-dimensional tree) is a main-memory binary search tree in which interior nodes use their attribute values to split their child nodes into two parts, and nodes at different tree levels have different attributes (Ooi, McDonell, and Sacks-Davis 1987; Zhou et al. 2008). A k-d tree is best at partial-match, range and nearest-neighbour queries (Robinson 1981). A Quad tree divides a space into a square two-dimensional region, and each interior node splits its region into four quadrants (Finkel and Bentley 1974; Eppstein, Goodrich, and Sun 2005). The r-tree is similar to b-tree, but its internal nodes represent the region in any shape (Beckmann et al. 1990). It is efficient for point location and overlapped region query but is complicated to insert/delete data.

By comparing the hash-table-like and tree-like approach, the hash-table index is more easily maintained with better scalability than the tree-like index when inserting/deleting datasets, but the data size of the hash-table index increases exponentially with the increase of data dimension. Considering the above characteristics, the k-d tree is chosen as the global index hosted by the master node to be as small as possible; the hash table is chosen as the local index for each worker node for easy maintenance. Meanwhile, these traditional spatial indices are adapted in the proposed hierarchical index because they do not support the search of where the queried data are stored in the distribute environment, which is the key point for the distributed query system. Besides, these above spatial data indexing strategies cannot be directly applied to Spark with HDFS due to the inconsistency among the raster data models, distributed physical data storage model in HDFS, and the complex distributed data pipeline in Spark.

2.2. Distributed systems for multi-dimensional geospatial raster data processing

As the volume of geospatial data is usually beyond a single computer's capability, the distributed systems with the capabilities of scalable storage and computing are required for handling big geospatial raster data. SciDB and Rasdaman are two of the most popular array-based distributed systems for geospatial raster data. SciDB implements regular chunking with arbitrary chunk size, stores chunks that logically belong together, and builds an index for chunks to accelerate query speed (Brown 2010). However, geospatial data need to be converted to the SciDB-specific binary format for data uploading, which is time- and disk- consuming (Hu et al. 2018b). Rasdaman supports several geospatial data formats (e.g. NetCDF, GeoTiff) and various tiling structures (e.g. regular, aligned directional tiling) with specific indices to provide quick data access (Baumann 2001). Nevertheless,

geospatial data need to be manually uploaded to each data node, and the projection coordinate information is missing after the data are imported (Hu et al. 2018b).

Besides the array-based distributed systems, Hadoop-based frameworks (e.g. Spark, Hive, MapReduce, HDFS) have been widely used to solve the large computational problems in big geospatial data (Li et al. 2016b; Yang et al. 2017b; Hu et al. 2018a). For example, SpatialHadoop integrates the spatial index and parallel computing to analyse big spatial data by building a two-level spatial index for vector data, and quad-trees for satellite data to improve the efficiency of MapReduce on spatial query and analytics (Eldawy and Mokbel 2013; Eldawy et al. 2015). These index trees, however, require the input data to be decompressed and reorganized in HDFS, so the data size will be much bigger than the original data. Li et al. (2017a) proposed a Hive-based framework to query big array-based climate data. The MySQL database is used to store the spatiotemporal index, but it will be the bottleneck when querying big geospatial data because large volumes of indexing data need to be transferred from the MySQL database to the Hive cluster. As an evolved version of MapReduce, Spark has also been applied in big geospatial data processing with faster execution engine and more expressive API (Hu et al. 2018a). For example, SciSpark extends Spark to natively support multi-dimensional data structure in RDD, and develops data mining algorithms (e.g. detecting climate extremes) for scientific datasets (Palamuttam et al. 2015). ClimateSpark proposes a spatiotemporal index for the chunked climate datasets to provide an interactive data analytical platform with high parallel I/O efficiency (Hu et al. 2018b).

The above indexing techniques and frameworks provide initial success for efficient geospatial raster data query and processing. However, the management of big geospatial raster data can be further improved by avoiding the data pre-processing and enhancing the parallel query capability with high scalability. Apache Spark has become one of the most popular big data platforms, but limited work exists to extend Apache Spark to utilize its powerful in-memory parallel data processing for multi-dimensional geospatial raster data. This paper presents a hierarchical indexing technology for Spark with HDFS to efficiently query big geospatial raster data and in so doing (1) enables Spark and HDFS to natively support geospatial raster formats (e.g. HDF4, NetCDF4, GeoTiff); (2) avoids the overhead of transferring large volumes of indexing data from the database to the Spark cluster; (3) efficiently queries big geospatial raster data with high parallelism, I/O efficiency, and data locality, where high data locality means moving computation tasks/programmes close to data rather than moving data towards computation; and (4) further accelerate the in-memory geospatial raster data query and process.

3. Methodology

3.1. Geospatial raster data model and layout in HDFS

The geospatial raster data consists of a two-dimensional matrix or multi-dimensional arrays. The multi-dimensional array-based data model (Figure 1) is used to represent geospatial raster data (Li et al. 2017a; Hu et al. 2018a;). It assembles a collection of cells that can be randomly accessed by an index tuple computed from a mathematical formula (Black 2006). This logical mapping saves storage capacity for coordinate information. An array is created by specifying dimensions and attributes. For example, in an n-dimensional array with dimensions $d_1, d_2, \ldots, d_n$, each dimension's size is the number of ordered values in that dimension. The combination of dimension values identifies a cell of the array, which holds the data value called attribute. To improve the computational and I/O efficiency for big geospatial raster data, a data partitioning technology (chunking/tiling) is usually adopted to partition multi-dimensional arrays into sub-arrays (chunks/tiles).

Because the chunk size is much smaller than the block size (e.g. 64 MB/128 MB) in HDFS, the chunk is treated as an atomic data unit and delivered to several data nodes when storing the chunked geospatial raster data on HDFS (Li et al. 2017a; Hu et al. 2018a) (Figure 1). The data locality is one of the essential factors for efficient parallel processing. It avoids the large data transferring *via* network by moving computation tasks to the data nodes where data are stored rather than moving data towards computation. The data location, the physical data nodes for each chunk, can be retrieved

by querying the block metadata of the distributed file system with the chunk layout information as follows:

$$hosts_{chunk_i} = f(\, filePath_{chunk_i},\ byteOffset_{chunk_i},\ byteLength_{chunk_i}) \tag{1}$$

According to $filePath_{chunk_i}$, the physical locations of the blocks that the input file contains can be derived from the metadata of HDFS; meanwhile, the combination of $byteOffset_{chunk_i}$ and $byteLength_{chunk_i}$ could identify which blocks are overlapped with the queried chunk. Then, the data host of the queried chunk ($hosts_{chunk_i}$) can be derived as the queried chunk has the same physical hosts with the identified blocks. A programme is developed to extract the chunk layout information from the headers of geospatial raster data files (e.g. HDF4, NetCDF4) using HDFS I/O API.

According to the physical location of each data chunk, the query tasks can be delivered to their corresponding data nodes to achieve high data locality. However, there are too many chunks in a collection of big geospatial data to quickly find the specified chunks or keep the workload balance among the worker nodes. Using MERRA-2 (a climate simulation geospatial raster data) as an example, one collection of daily MERRA-2 production has ~ 0.5 billion chunks varying in content and dimensions (e.g. 2-, 3-, higher dimension), and the size of chunk metadata is about 10GB. The MERRA-2 dataset has about 42 data collections, so the total chunk metadata size is more than 400GB. A standalone database could not efficiently manage these large volumes of information when the queries are frequently submitted. It is also time-consuming to transfer these metadata from the external database to the computing cluster. In addition, there are few spatial indices considering the data locality for spatial queries.

3.2. The Architecture of the hierarchical index

To enable Spark with HDFS to efficiently query big geospatial raster data, a three-layer hierarchical index (Figure 2) is proposed: (1) the global index: the k-d tree is constructed at the master node to globally overview all the chunks' location across the cluster; (2) the local index: the hash table is built for each worker node to index all the chunks stored in the local worker node and provide the chunk

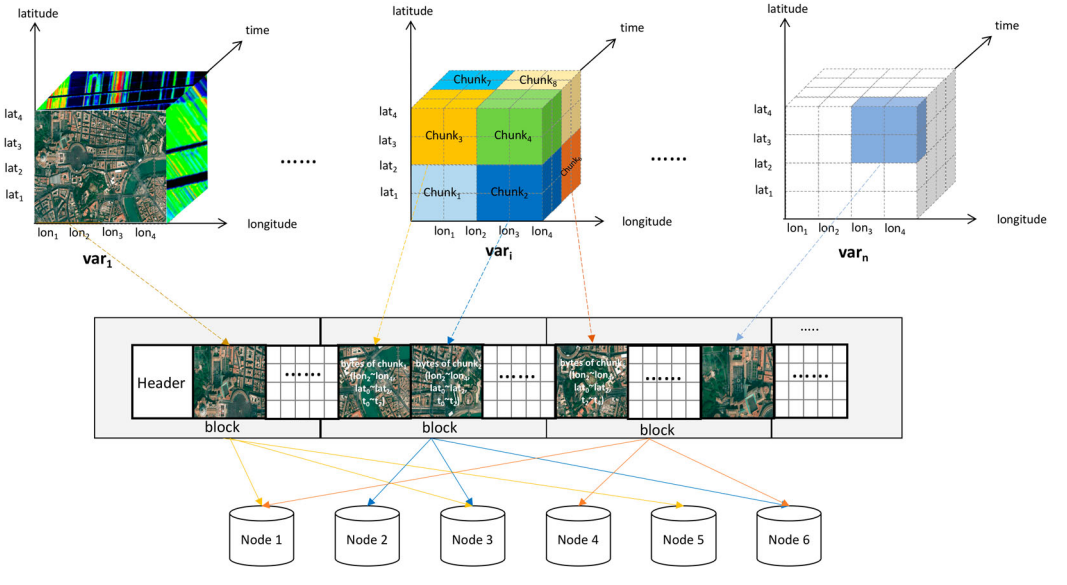

Figure 1. Array-based data model with chunked data structure in HDFS.

layout information at the byte-, block-, and file- levels; (3) the RDD index: the k-d tree is built to persist and index all the in-memory chunks in each RDD partition to accelerate spatial queries of in-memory data. The hierarchical index is stored in HDFS to avoid the transferring of indexing data from the external data sources. Meanwhile, the division of the global and local index reduces the size of the index on each node to be bulk loaded in memory, and avoids chunk metadata transferring between the master and worker nodes. The hierarchical index provides the information of chunk layout in HDFS at the byte-, chunk-, block-, file-, memory-, and node- level for the schedule of query tasks. More details are introduced in the following sections.

3.2.1. Global index for high data locality and workload balance

To efficiently query a multi-dimensional bounding box in Spark, the first step is to identify which chunks are overlapped with the input bounding box and find their physical host nodes to read the data. The global index (Figure 3) is designed to index all chunks with their physical locations to get the global overview of chunk locations across the cluster. Since the global index is located at a single master node, it is optimized to be as small as possible. The k-d tree index structure, which is an efficient data structure for range and nearest neighbour searches in multi-dimensional spaces, is adjusted to implement the global index. It is known that the k-d tree indexes the points rather than cubes in a k-dimensional space. When applied to chunks, a unique vertex of the chunk (e.g. the upper-left corner) is identified to represent its location in the k-d tree (Figure 4). Different from the traditional k-d tree, the physical host addresses of each chunk are added into the k-d tree, helping keep the workload balance and data locality for the query tasks.

In details, each node in the tree is a chunk's representative point containing its physical host address (Figure 3). At each level of the global index, the children nodes are split along a hyperplane that is constructed perpendicular to the corresponding dimension. All children of the root node are split by the first dimension, so that for a children node of the root whose first-dimension coordinate is less than the root, it is in the left-sub tree; otherwise, it is in the right sub-tree. For each level down

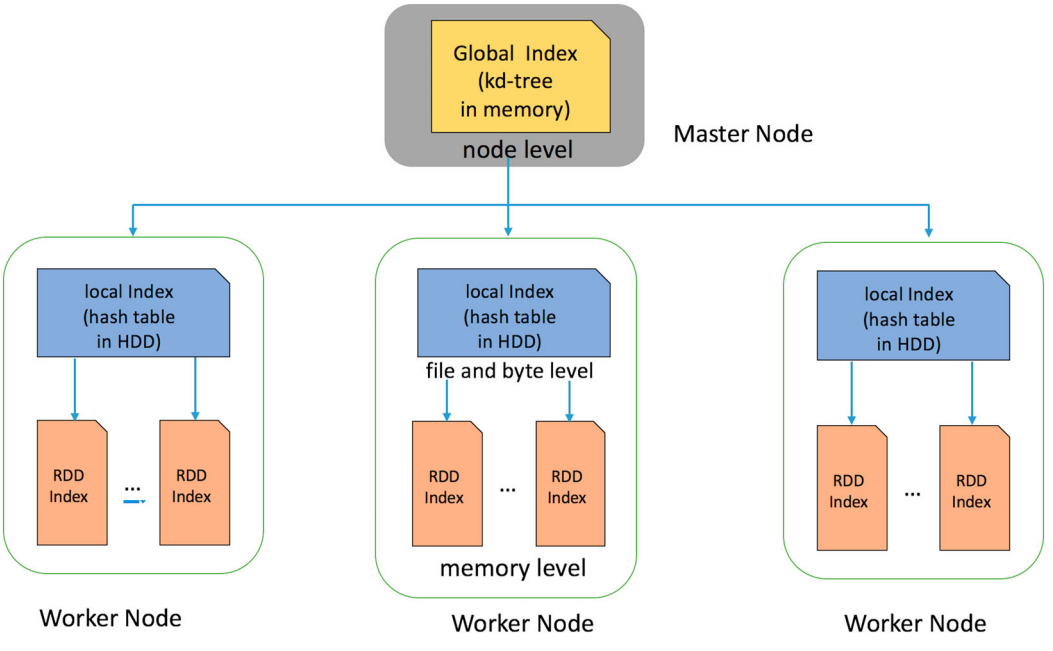

Figure 2. The three-layer hierarchical index.

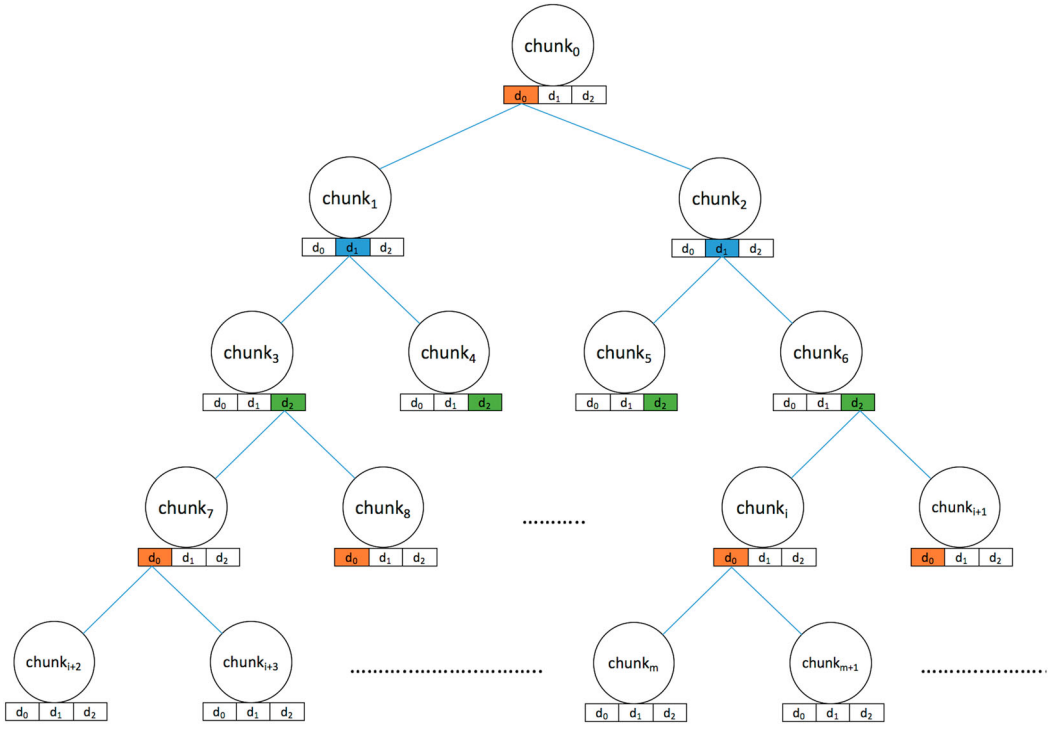

Figure 3. The architecture of the global K-D tree index.

in the tree, it is split by the next dimension, and if all other dimensions have been exhausted, the split dimension returns to the first dimension. To keep the balance of the k-d tree, the chunk in the centre of the big cube is placed at the root, and every node with a smaller dimensional value is placed to the left and larger to the right. This procedure is repeated on both the left and right sub-trees until the last tree has only one element. The global index supports the point, range and nearest neighbour queries.

Once the global index is achieved, the next steps are to retrieve the specified chunks and equally partition them as the sub-query tasks for data nodes. In order to get the high data locality for each task and keep the workload balance among the data nodes, the following algorithm is proposed (Algorithm-1):

(1) Go through the global index with the input geospatial bounding box and retrieve the collection of overlapped chunks S.

$$S = \{chunk_i \mid \forall chunk_j \cap boundingbox\}$$

$$chunk_i = [vertex_i, (dataNode_i, \ldots\ldots, dataNode_j)]$$

(2) Specify which nodes handle which chunks. First, the data nodes are counted and recorded as $N = \{dataNode_i, dataNode_j, \ldots\ldots, dataNode_m\}$; When assigning a retrieved chunk, it is delivered to the data node that physically stores it but with the smaller number of assigned chunks, resulting that each data node is assigned with a similar number of chunks and only process the chunks that are locally stored; and

(3) To reduce the data transferring from the master to data nodes, chunks are represented by an ID, a combination of the vertex's coordination as $id_{chunk_i} = vertex_i(x_i, y_i, z_i, \ldots\ldots)$.

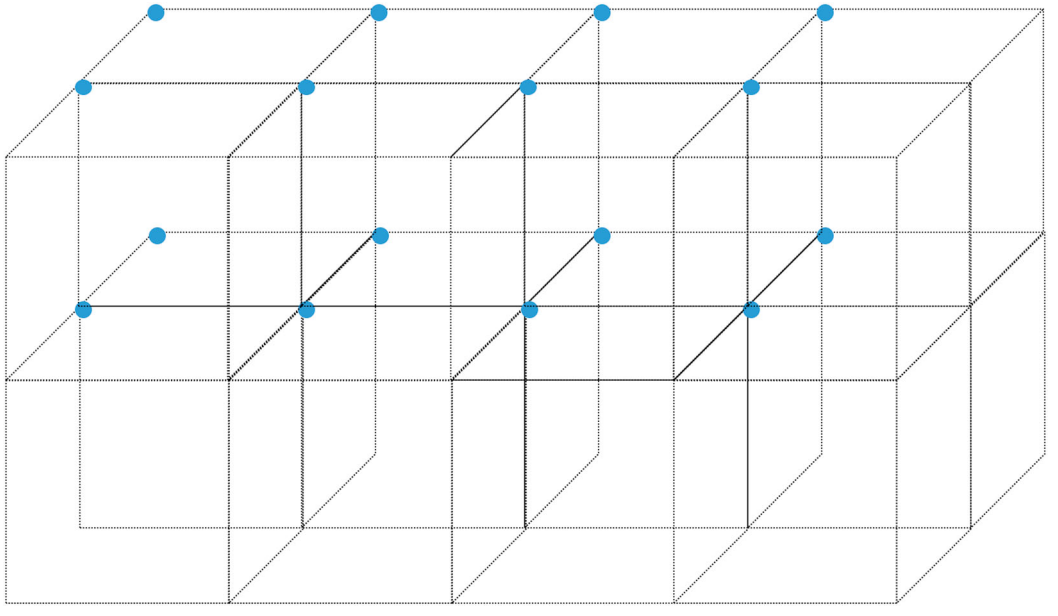

Figure 4. The chunked cube with the representative points. The blue point represents its corresponding chunk.

Therefore, the outputs after querying the global index are represented as follows:

$$list_{partioned} = \begin{cases} (dataNode_m, & l_{nodeID}\{a,\ b,\ c,\ d,\ \ldots\ldots\}) \\ & \ldots \\ (dataNode_n, & l_{nodeID}\{e,\ f,\ g,\ h,\ \ldots\ldots\}) \end{cases} \tag{2}$$

Algorithm-1 ChunkID partition

1. g ← global index
2. b ← bounding box
3. *targetNum* ← the predefined number of chunks assigned for each data node
4. *f* (*g*, *b*) ← function which returns the *chunkIDs* that overlapped with input bounding box
5. Set of queried *chunkIDs S* ← *f* (*g*, *b*)
6. Set of involved data nodes N ← {}
7. ***for*** each *chunkID* in *S* ***do***
8. N ← N ∪ {*datanode* | *dataNode* ∈ *ChunkID but* ∉ *N*}
9. Set of repartitioned *chunkIDs* for each involved data node $list_{repartitioned} \leftarrow \{l_1,\ l_1,\ \ldots\ldots,\ l_n\}$
10. ***for*** each *chunkID* in *S* ***do***
11. *targetNode* ← null
12. ***for*** each data node $node_i$ in *chunkID* ***do***
13. if the number of *chunkIDs* assigned to $node_i < targetNum$
14. $targetNode = node_i$
15. $l_{targetNodeID}$ ← assign *chunkID* to *targetNode*
16. return $list_{partitioned}$

3.2.2. Local index for data chunk metadata information retrieval

After the worker/data node receives the assigned sub-query tasks, which contains the assigned query chunk IDs, the chunk metadata information is fetched from the local index (Figure 5). The metadata information for each chunk falls into two categories: (1) logical data information (*chunk_corner*, *chunk_shape*, *variable_name*, *data_type*, *file_name*, *unit*, *missing_value*, and *valid_value_range*); and (2) physical data information (*byte_offset*, *byte_length*, and *compression_type)*, where the *byte_offset* refers to the byte offset of the chunk in the original image. Each chunk ID refers to a

pair of logical and physical metadata information, so the local index is implemented by a hash table as follows (Figure 5):

On each worker node, a local index is built for each variable and stored in separate files, so the local index files are loaded on demand when querying multiple variables. After the local index is loaded into memory, the chunk metadata is retrieved according to the input chunk IDs.

3.2.3. RDD index for in-memory data query

Based on the chunk metadata, the byte streams for each chunk can be retrieved from HDFS and decompressed into arrays, and these arrays in different worker nodes will be organized together as an RDD. but these arrays are not ordered yet. That means, for each query, the arrays need to be linearly scanned even for querying a small number of points. Therefore, an RDD index (k-d tree) is built for each RDD partition to improve the query efficiency. The k-d tree structure in the RDD index is the same with that in the global index, but the content of each node is the combination of the chunk's dimensions and their array values rather than chunk IDs.

3.2.4. The workflow of Apache Spark with the hierarchical index

The workflow of Apache Spark working with the hierarchical index to retrieve the chunks (Figure 6) has several distinct steps as follows:

(1) Input the querying range on the master node and proceed through the global index to retrieve the specified chunk IDs and their physical locations;
(2) Equally partition these chunk IDs by their physical hosts and deliver the grouped chunkID lists to their corresponding worker/data node. This step keeps the workload balance among the worker nodes, while achieving high data locality for the assigned tasks in the next step.
(3) Each executor on the worker node launches a task to retrieve the chunk metadata from the local index according to the assigned chunk IDs; and
(4) Equally split each chunk metadata list into several sub-partitions to keep workload balance among threads, and for each partition, launch a task to read the data specified by the metadata as array.
(5) Once the real arrays are read out, they will be organized as an RDD and the RDD index will be built for each RDD partition.

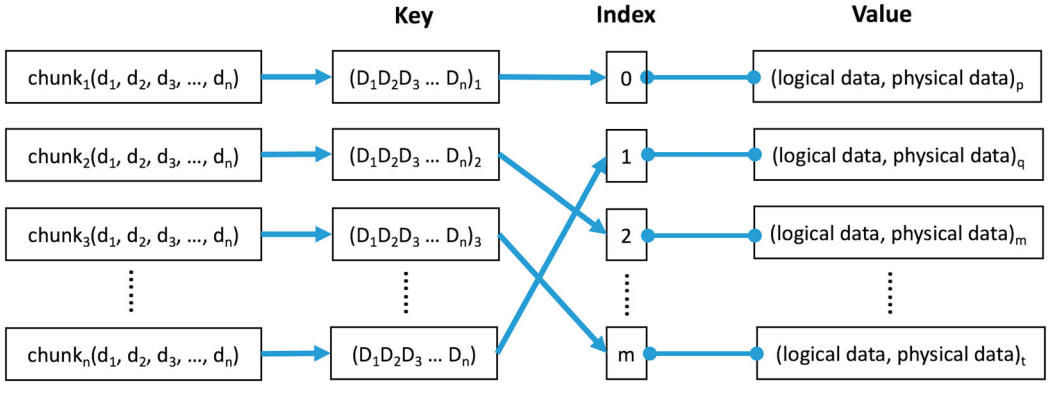

Figure 5. The architecture of the local index.

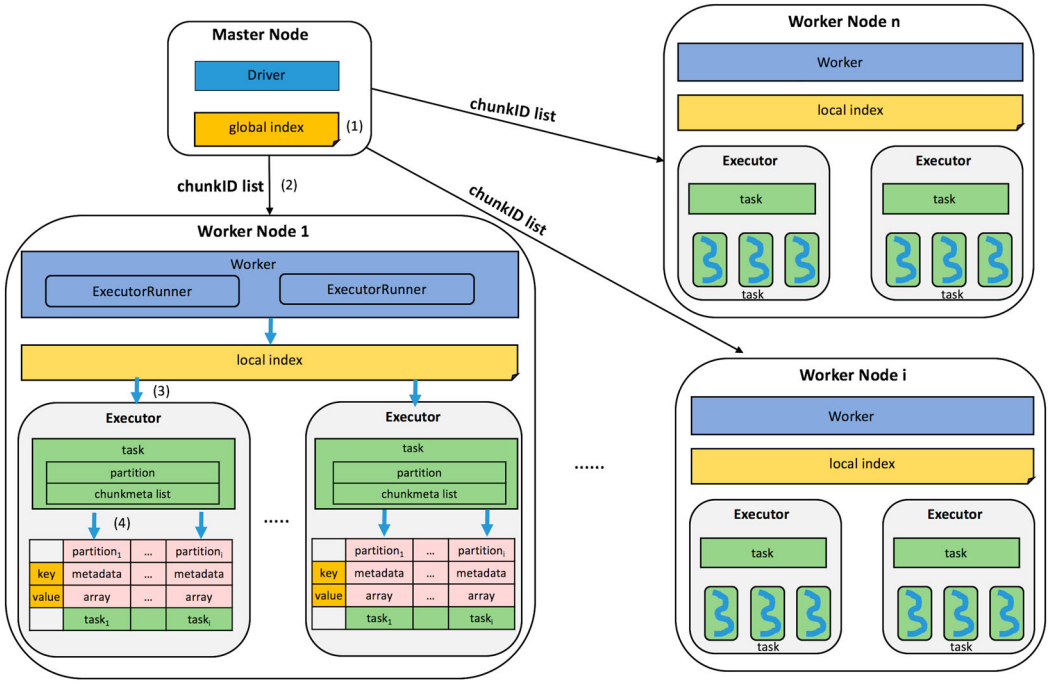

Figure 6. Workflow of Apache Spark working with hierarchical index.

3.3. Customized RDD for the hierarchical index

Resilient Distributed Dataset (RDD) is the basic data abstraction in Spark, which represents an immutable partitioned collection of elements processed in parallel. However, the default RDDs do not support the array-based geospatial data with the hierarchical index yet, so four kinds of customized RDDs are proposed: ChunkID RDD, ChunkMeta RDD, Chunk RDD, and IndexedChunk RDD (Figure 7).

- ChunkID RDD

Querying the global index generates a list of chunk IDs specified by the input bounding box. ChunkID RDD is designed to represent these queried chunk IDs. The ChunkID RDD is split into several partitions according to the chunk's physical data hosts (Section 3.2.1). That is, each partition of the ChunkID RDD is a list of chunk IDs for a certain physical data node, and the number of partitions is the same as the number of nodes.

- ChunkMeta RDD

In the local index each chunk ID maps to the referred chunk's metadata (ChunkMeta), including its logical information and physical information used to read the chunk data out from HDFS. Passing each partition of the ChunkID RDD into its corresponding local index generates a partition of chunk metadata, the collection of which is a ChunkMeta RDD. Accordingly, the ChunkMeta RDD has the same number of partitions with ChunkID RDD, and their corresponding partitions are stored at the same node.

The data specified for each chunk is read by using the logical and physical information in ChunkMeta RDD. However, the current ChunkMeta RDD has a small number of partitions, the same as the number of data nodes. That means there is only one task launched on each data node. However, the data node with multiple cores should be able to run multiple tasks at the same time. Therefore,

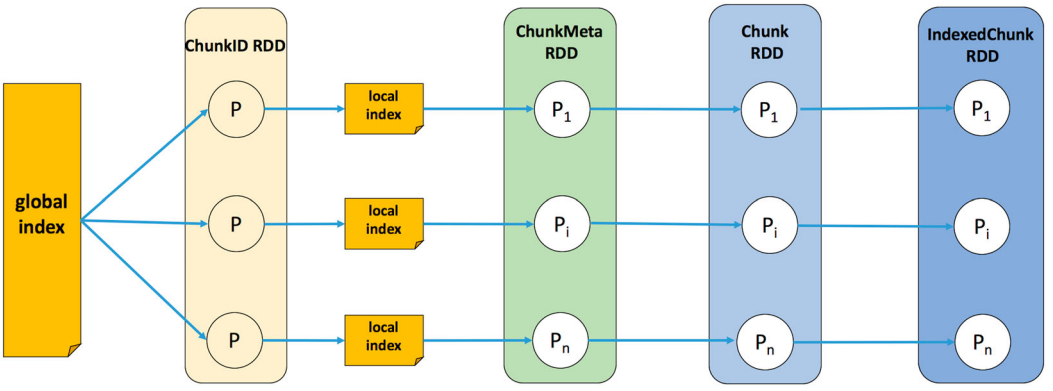

Figure 7. Workflow for customized RDD transformation.

the strategy for tuning parallelism is designed in Section 3.5 to better leverage the available computing resources.

- Chunk RDD and IndexedChunk RDD for in-memory query

Given the meta data information in ChunkMeta RDD, the specified data are retrieved *via* HDFS I/O API and stored as Chunk RDD with high data locality, but the chunks in each ChunkRDD partition is still not ordered. Therefore, the RDD index (k-d tree) is built for each partition of ChunkRDD to generate the IndexedChunk RDD, which can efficiently query the target chunk without visiting all chunks.

3.4. Tuning parallelism

Each partition of an RDD launches a task, so the parallelism can be adjusted by changing the partition number of RDDs. Although the default RDD in Spark supports the repartition of RDDs, it does not keep the partition's original data location, resulting in large volumes of data shuffling and consuming a large partition of network and memory resources.

Instead, a new repartition strategy (Algorithm-2) is proposed, changing the partitions of customized RDD to control the parallelism level while avoiding the shuffling to achieve high data locality. The repartition strategy has two parts - increasing parallelism and decreasing parallelism (Figure 8). With increasing parallelism, the partitions of ChunkMeta RDD are equally split into several small partitions. For keeping data locality, new sub-partitions are at the same node with its parent partition. The increasing number of partitions enables the ChunkMeta RDD to launch more tasks to read data in parallel. With decreasing parallelism, the ChunkMeta RDD combines small partitions to get bigger partitions, each of which has larger number of chunks. If the current parallelism takes the best of the available resources, ChunkMeta RDD keeps the same number of partitions to maintain the current parallelism.

Algorithm-2 Tuning Parallelism

1. $g \leftarrow$ global index
2. $l \leftarrow$ local index
3. $b \leftarrow$ input bounding box
4. $f\ (g,\ b) \leftarrow$ function which returns the *chunkIDs* that overlapped with input bounding box
5. $h\ (\ chunkID,\ l) \leftarrow$ function which returns the chunk metadata for the specified chunk
6. $p \leftarrow$ the folds of increasing the number of partition
7. $d \leftarrow$ the folds of decreasing the number of partition
8.
9. set of chunkIDs $S \leftarrow f\ (g,\ b)$

(*Continued*)

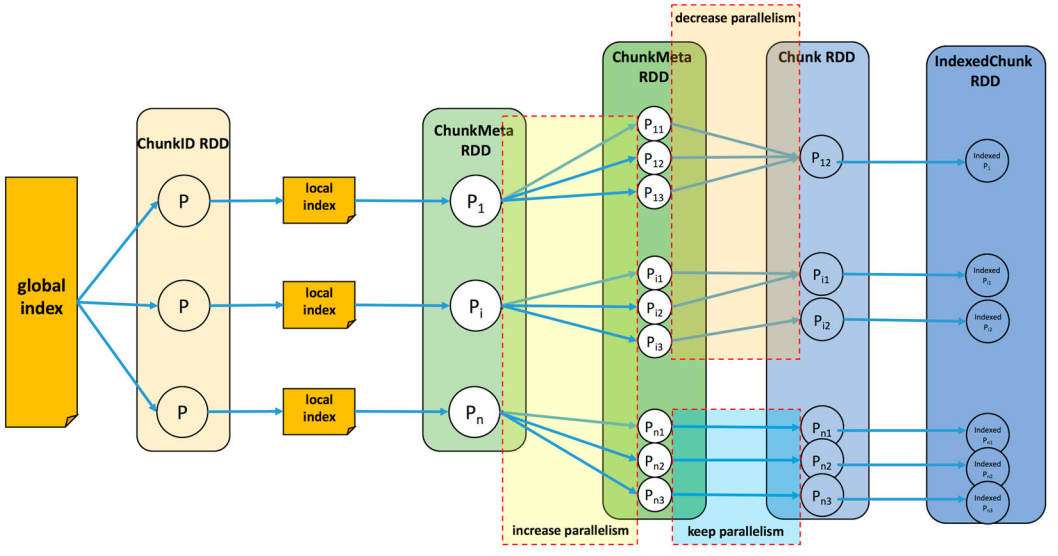

Figure 8. Dynamic parallelism change.

Continued.

Algorithm-2 Tuning Parallelism

10. set of dataNodes $N \leftarrow$ the data nodes that store the queried chunks
11. set of chunk metadata $M \leftarrow \{m_1, m_2, \ldots\ldots, m_n\}$
12. $n_i \leftarrow$ the number of chunks assigned for the node N_i
13. ***for*** each data node $N_i \in N$ ***do***
14. $S_i \leftarrow$ Assign n_i *ChunkIDs* from S to N_i where the chunks are physically stored
15. *chunkID_RDD* $\leftarrow$ parallelize $\{S_1, S_2, \ldots\ldots, S_n\}$
16. ***for*** each partition p_i in *chunkID_RDD* ***do***
17. ***for*** each *chunkID* in p_i ***do***
18. $m_i \leftarrow m_i \cup^h (chunkID, l_i)$
19. *SplitPartition*$(M, p) \leftarrow$ split each partition m_i into p sub-partitions
20. *CombinePartition*$(M, d) \leftarrow$ combine d sub-partitions on the same node into one partition
21. *chunkmeta_RDD* $\leftarrow$ derived from the repartitioned $\{m_1, m_2, \ldots\ldots, m_n\}$
22. *Chunk_RDD* $\leftarrow$ utilize HDFS IO API to read the chunks specified by *chunkmeta_RDD*
23. ***for*** each partition p_i in *chunk_RDD* ***do***
24. $indexedPartition_i \leftarrow$ build k-d tree for p_i
25. *IndexedChunk_RDD* $\leftarrow$ {*indexedParition*}

4. Performance evaluation

4.1. Experimental design

The hierarchical index is implemented and integrated with Apache Spark to efficiently query big geospatial raster data. To evaluate the proposed hierarchical indexing technology, the performance of Spark with the proposed hierarchical index is compared with that of Spark SQL and SciSpark. The experiment data are MERRA-2 tavg1_2d_int_Nx data (M2T1NXINT) in NetCDF4 data format and mixed with 2-, 3-, and 4-dimensional variables. The total data size approaches 7.35 TB from 01/01/1980 to 12/31/1995. The spatial resolution is 0.625 * 0.5 degree, and the temporal resolution is hourly. Inside the dataset, each grid is split into 16 chunks with the resolution of 91 by 144 points, and there are ~ 494.5 millions of chunks in the 16-year data.

The Spark cluster (v1.5.0 + Hadoop v2.6.0) consists of one master node and 19 worker nodes, and the cluster resource is managed by Yarn.[1] Each node is configured with 24 CPU cores (2.35 GHz) and 24 GB RAM on CentOS 7.2 and connected with 20 GB Ethernet (GPS).

Since Apache Spark does not natively support the binary data in NetCDF data format, the experiment raster dataset is converted to points and saved in the relation table in Parquet format.[2] Subsequently, Spark SQL is applied to query/process the data, which is the default query fashion in Spark. SciSpark and the Spark with the proposed hierarchical index can directly query and compute the data without data pre-processing, since they both natively support the experiment data format. For the hierarchical index, the global index is built for the master node, whereas the local indexes are built for each data node and stored on the corresponding nodes. The experiments on temporal queries, spatial queries, and in-memory queries investigate the proposed hierarchical index from three perspectives: (1) how it handles different volumes of data; (2) how it searches geospatial raster data with high efficiency; and (3) how it accelerates multi-dimensional data query in memory. Each experiment is replicated ten times and from which a mean is calculated.

4.2. Temporal query

The temporal query subsets the data in a specified time range, and the data volume increases at the same speed as the query time range increases. For each query, the data specified by the time range in Colorado area are extracted from the data pool, and the monthly spatial average is calculated. The queried time range increases from 1 to 10 years in a one-year step, and the corresponding queried data sizes increase from 0.45TB to 4.59TB. The hierarchical index has the best performance, increasing by a factor of 5 from 7.4 to 36.9 seconds. The runtime for Spark SQL and SciSpark increases by a factor of ∼ 2.6 (20.2 to 51.7 seconds) and ∼14 (108 to 1512 seconds) respectively (Figure 9).

Spark SQL provides an SQL-style fashion for users to easily implement and run Spark jobs. The high-level SQL query script can be parsed to the logical plan and then optimized by Spark's computing optimiser (Catalyst) at multiple levels to choose the best physical plan for high data locality and efficient task execution (Yadav 2015). Then, according to the optimized physical plan, the query script can be automatically translated to Spark RDD APIs. This kind of cost-based optimization algorithms for the computational directed acyclic graph makes sure the query run as fast as possible in the distributed environment. Besides, in the Parquet files, all the columns are stored separately, and each column is split into several groups with metadata to describe its data layout. Spark SQL natively support the structure of Parquet files with high I/O efficiency and data locality. When reading the input Parquet data, Spark SQL first reads the queried columns in the SQL script with high data locality to figure out which rows of the table are queried, and then only read the identified rows. This data pipeline improves the I/O efficiency by reducing the reading of unnecessary data. However, irrespective of the input time range, Spark SQL still needs to read and traverse the entire columns that are specified in the query script due to the lack of column indexing. That is why the query time for Spark SQL increases much slower with the enlarged temporal query range. Meanwhile, the conversion of the raster data to a relational table causes repeating coordinate information in the table, and then more data reading and memory allocation when running the query tasks. That means the relational table-like data structure is not an efficient way to store geospatial raster data (Hu et al. 2018b). The low efficiency of identifying the queried rows and the redundant data in the tables slows down the performance of Spark SQL.

The proposed hierarchical index enables Spark to query the global k-d tree index, about 0.003% of the input data size, to quickly identify the queried chunks and subsequently launch tasks on the local physical machine with high data locality to read and compute the data. At the same time, although the optimiser in Spark SQL could not work for RDD-based programmes, the customized RDDs make sure that the sub-query tasks are executed as expected. Besides, the array data model is used in the cumulative data pipeline to organize the data in both files and memory, which is more disk-, network- and memory-saving than the relational data model in Spark SQL. In terms of run time increasing speed, the hierarchical index is faster than Spark SQL. That is because

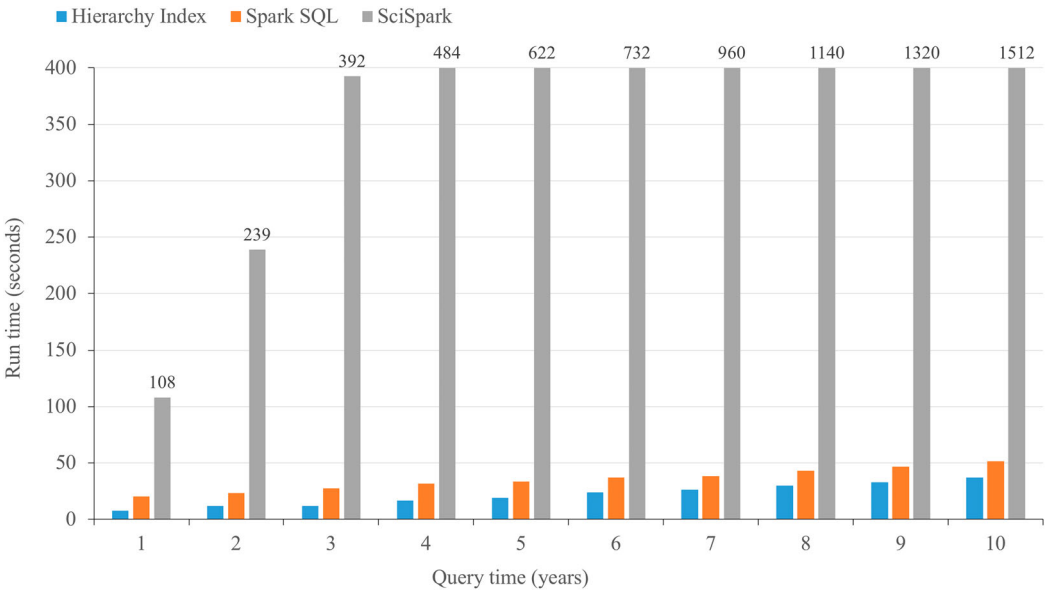

Figure 9. Time for computing daily mean with varied query time.

Spark with the hierarchical index only reads and decompresses the data specified by the query bounding box, so the processed data volume increases with query time range; But Spark SQL always reads the whole queried columns even for the different temporal query range, so the run time increases slowly but is inefficient.

SciSpark, although natively supporting the experiment data, does not consider the data locality or chunking data structure in the experiment data. Basically, it randomly allocates the query tasks to worker nodes. Each worker node reads the assigned files using HDFS I/O APIs and then interpret the byte streams using NetCDF Java library.[3] Consequently, large volumes of data serialization and transferring via network put heavy pressure on the network and memory resources of the Spark cluster; at the same time, when the memory usage reaches the limitation, the java's 'stop-the-world' garbage collection leads to a significant latency in SciSpark (Palamuttam et al. 2015).

4.3. Spatial query

In the spatial queries, the eight different bounding boxes enlarge at a factor of two (Figure 10) and the involved data sizes increases from 0.28TB to 2.30TB. Each query has the same time range from 1:00 am to 2:00 am from 01/01/1986 to 12/31/1990, and the temporal average is computed for each point in the queried time series. The run time for the hierarchical index, Spark SQL, and SciSpark are 10.8 to 14.9 seconds, 36.2 to 48.5 seconds, and 734 to 782 seconds (Figure 10). In each query the hierarchical index encompasses less time (~ 2.5 fold less) than the Spark SQL and ~40 times faster than SciSpark. The performance of SciSpark gets worse than the temporal query. It is caused by the data aggregation difference between the spatial and temporal queries. The temporal average in the spatial query aggregates each point's values along the time dimension rather than aggregating all into a single point as the temporal query. That means much more data need to be transferred via network. It brings serious network overhead while putting heavy pressure on the Java virtual machine since large volume of data need to be serialized and deserialized. However, Spark SQL can automatically optimize the query execution to minimize the data transferring by the Catalyst optimizer; the data pipeline in the proposed hierarchical indexing approach is optimized by the customized RDDs to

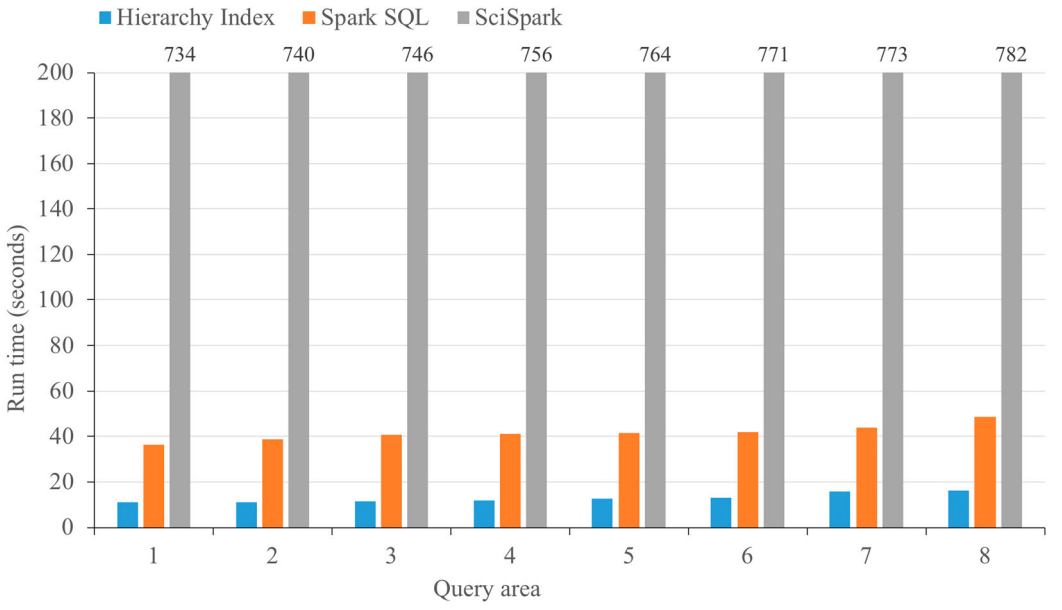

Figure 10. Time for computing temporal mean with varied query area. The bounding box for the eight queries are as follows: (1) (−45°, −90°) ~ (0°, 90°); (2) (−45°, −90°) ~ (45°, 90°); (3) (−45°, −90°) ~ (90°, 90°); (4) (−90°, −90°) ~ (90°, 90°); (5) (−90°, −180°) ~ (45°, 120°); (6) (−90°, −180°) ~ (45°, 180°); (7) (−90°, −180°) ~ (90°, 120°); and (8) (−90°, −180°) ~ (90°, 180°).

move the computing programmes to the data as much as possible. The spatial query results further prove the importance of data locality and I/O efficiency for locating data and reducing data transferring when answering range queries in the distributed big data system.

4.4. In-memory data query

One of the biggest advantages in Spark is to accelerate the iterative computing by caching data. In Section 3.4.3, the design of *IndexedChunk RDD* improves the data accessibility in memory. To evaluate the advantages of *IndexedChunk RDD*, the performance of *IndexedChunk RDD*, Spark SQL, and SciSpark for querying the cached data is compared by using single point, range, and nearest neighbour queries (top 8 nearest neighbours) (Figure 11). In the single point and range queries, Spark with the hierarchical index and SciSpark have similar performance but are faster than Spark SQL. Spark SQL needs to compare the input boundary with each point, whereas *IndexedChunk RDD* can filter out the unspecified chunks according to the k-d tree, and extract the points using array indices; SciSpark designs the tensor data structure to combine neighbour chunks as arrays in memory, so the queried points can be fast identified by array indices. For the nearest neighbour search, Spark SQL first computes each point's distance to the query point and sorts the points by distance to get the top 8 nearest neighbours; this is a time-consuming task for processing ~ 5.5 billion points. For *IndexedChunk RDD* each partition of its chunks is indexed by the k-d trees and cached. When querying the nearest neighbours, the first step goes through the k-d tree to identify the nearest chunk and subsequently finds the nearest points in the nearest chunk. This avoids sorting the points by their distance to the target point, which is necessary for Spark SQL, by leveraging the spatial chunk-level topology information in the k-d tree. The tensor in SciSpark contains the queried neighbour points in the same array, so SciSpark could efficiently locate the neighbour points using array indices, which illustrates the efficiency of array-based data model in querying the in-memory geospatial raster data.

4.5. Increasing and decreasing parallelism level

The parallelism level is an important parameter for the performance of Spark. The proposed method (Section 3.4) optimizes the parallelism by adjusting the number of partitions in customized RDDs to better utilize the available computing resources. To evaluate the feasibility of the proposed method, we design a group of experiments, each of which computes the global monthly mean value from 01/01/1986 to 12/31/1990 but launches different numbers of tasks (Figure 12). By increasing the number of tasks, the run time initially decreases but begins to increase after the number of tasks exceeds 3652. This result has two explanations. First, when the task number is small, each task processes more data sequentially and more memory is consumed. This places more pressure on garbage collection, requiring more data being split to disk, which in turn causes disk I/O and memory wasted. Second, when the task number is large, each task processes less data but consumes more time to allocate/release the task resources. As the number of task increases, more repeated operations in each task are required.

5. Conclusion and discussion

Big geospatial raster data require scalable data management systems and efficient computing frameworks for scientists to discover the hidden knowledges (Buck et al. 2011; Li et al. 2017b; Yang et al. 2017b). As an advanced distributed computing framework, Apache Spark has been utilized to process and analyse big geospatial data, but a gap remains between Apache Spark and multi-dimensional array-based geospatial raster data. This paper designs and implements a hierarchical index strategy for Apache Spark with HDFS to efficiently query and process multi-dimensional data without data pre-processing. The combination of a global k-d tree index for the master node and the local hash table for each data node provides a scalable indexing strategy to search big geospatial raster data in a distributed environment. Moreover, the index structure is adjusted to provide the indexing information at node, file, block, and byte level while indexing the data in the memory to further improve the performance of Spark on in-memory data query. The

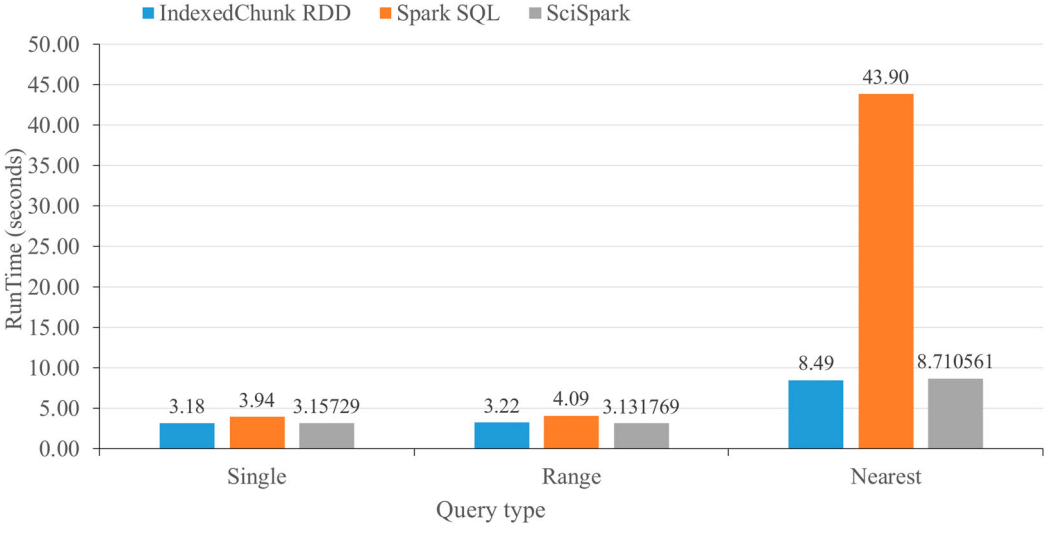

Figure 11. Comparison of the query performance of IndexedDataChunkRDD and Spark SQL for cached data: (1) single point query: query the point at (80.5°, − 174.5°) at 2:00 am 01/01/1987; (2) range query: query the points at (80.5°, − 174.5°) during 1:00 am to 2:00 am from 01/01/1987 to 01/31/1987; (3) nearest neighbour query: find the 8 nearest neighbours for the point at (80.5°, − 174.5°) at 2:00am 1987/01/01.

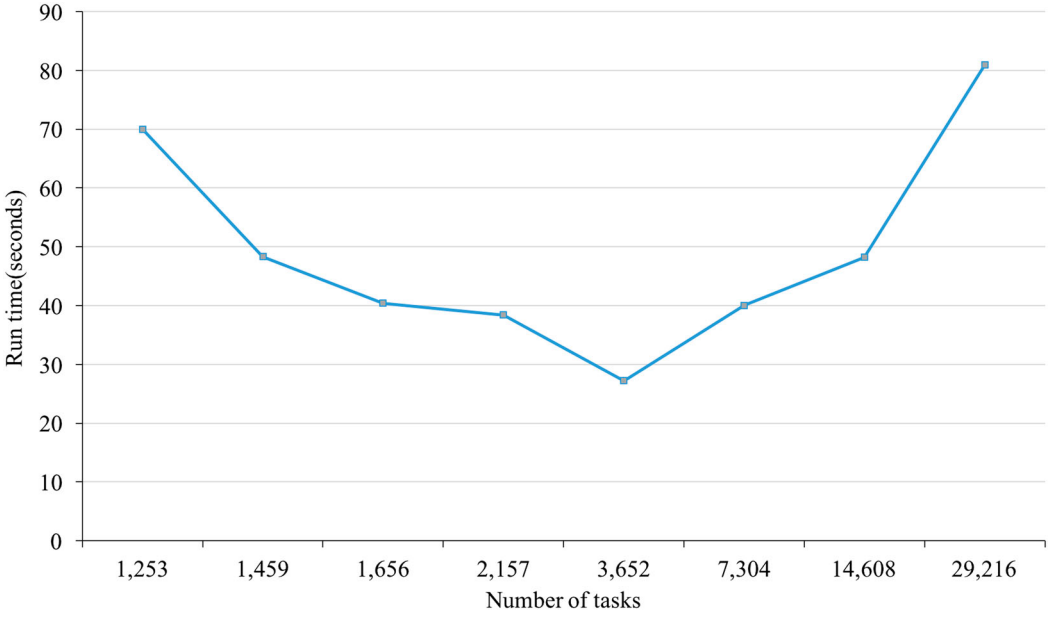

Figure 12. Run time for different number of tasks to compute the global mean from 01/01/1986 to 12/31/1990.

hierarchical index is integrated with Spark by developing customized RDDs (i.e. *ChunkID*, *ChunkMeta*, *Chunk*, *IndexedChunk*). By experimentally comparing the performance of the proposed hierarchical index method, Spark SQL, and SciSpark, the hierarchical index strategy accomplishes the following: (1) efficiently queries multi-dimensional geospatial raster data without unnecessary data reading by utilizing the logical and physical data information provided by the hierarchical index; (2) reads data with high data locality to avoid the network bottleneck; (3) indexes the cached multi-dimensional data to improve the spatial query efficiency; and (4) tunes the parallelism by repartitioning the customized RDDs with high workload balance and data locality.

For the future research, several lines of investigation are suggested as follows:

(1) The proposed hierarchical index is implemented by the low-level RDD APIs, which are not used as easily as Spark SQL or DataFrame APIs. The User Defined Functions (UDFs) provided by Spark SQL will be utilized to wrap the customized RDDs and add the query functions into the Spark SQL language. Then, users just need to write SQL-like scripts to query large volumes of geospatial raster data.
(2) The number of tasks need to be changed by the method proposed in Section 3.4, but an intelligent scheduling algorithm needs to be found for the most appropriate number of tasks in light of the available computing resources.
(3) As the latest version of Spark (released on Feb. 2018) starts to support Kubernetes, the proposed hierarchical index strategies could be further revised to leverage the scalability provided by the Kubernetes cluster, such as dynamically adding more nodes to improve the parallelism.
(4) The hierarchical indexing method can be extended to support other MapReduce-style computing frameworks (e.g. Hive and MapReduce) to improve the geospatial raster data accessibility and query efficiency. Besides, the proposed index strategies can be applied to index the layout of geospatial raster data in the cloud object storage systems (e.g. Seagate Kinetic Open Storage Platform, AWS S3, and OpenStack Swift) as well to avoid large volumes of data transferring and iterations.

(5) The proposed global index may not work efficiently with the cases that frequently update data, such as the applications with real-time data sources. The trade-off between the query speed and the efficiency of index maintenance needs to be further investigated.

Notes

1. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
2. https://parquet.apache.org/.
3. https://www.unidata.ucar.edu/software/thredds/current/netcdf-java/.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This research is funded by NASA (National Aeronautics and Space Administration) NCCS and AIST (NNX15AM85G) and NSF I/UCRC, CSSI, and EarthCube Programs (1338925 and 1835507).

ORCID

Chaowei Yang http://orcid.org/0000-0001-7768-4066

References

Baumann, P. 2001. "Web-Enabled Raster Gis Services for Large Image and Map Databases." Proceedings. 12th international workshop on database and expert systems applications, 870–874. IEEE.

Beckmann, N., H. P. Kriegel, R. Schneider, and B. Seeger. 1990, May. "The R*-Tree: an Efficient and Robust Access Method for Points and Rectangles." ACM sigmod record (Vol. 19, No. 2, 322–331). ACM.

Black, P. E. 2006. *Dictionary of Algorithms and Data Structures, US National Institute of Standards and Technology*. New York: Wiley.

Brown, P. G. 2010, June. "Overview of SciDB: Large Scale Array Storage, Processing and Analysis." Proceedings of the 2010 ACM SIGMOD international conference on management of data, 963–968. ACM.

Buck, J. B., N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. 2011, November. "SciHadoop: Array-Based Query Processing in Hadoop." 2011 international conference for high performance computing, networking, storage and analysis (SC), 1–11. IEEE.

Cheng, Y. 2016. "In-situ Data Processing Over Raw File." (*Thesis (PhD)*). University of California, Merced.

Demchenko, Y., P. Grosso, C. De Laat, and P. Membrey. 2013, May. "Addressing big Data Issues in Scientific Data Infrastructure." 2013 international conference on collaboration technologies and systems (CTS), 48–55. IEEE.

Eldawy, A., and M. F. Mokbel. 2013. "A Demonstration of Spatialhadoop: An Efficient Mapreduce Framework for Spatial Data." *Proceedings of the VLDB Endowment* 6 (12): 1230–1233.

Eldawy, A., M. F. Mokbel, S. Alharthi, A. Alzaidy, K. Tarek, and S. Ghani. 2015, April. "Shahed: A Mapreduce-Based System for Querying and Visualizing Spatio-Temporal Satellite Data." 2015 IEEE 31st international conference on data engineering (ICDE), 1585–1596. IEEE.

Eppstein, D., M. T. Goodrich, and J. Z. Sun. 2005, June. "The Skip Quadtree: A Simple Dynamic Data Structure for Multidimensional Data." Proceedings of the twenty-first annual symposium on computational geometry, 296–305. ACM.

Finkel, R. A., and J. L. Bentley. 1974. "Quad Trees a Data Structure for Retrieval on Composite Keys." *Acta Informatica* 4 (1): 1–9.

Gionis, A., P. Indyk, and R. Motwani. 1999, September. "Similarity Search in High Dimensions via Hashing." *In VLDB* 99 (6): 518–529.

Hinrichs, K., and J. Nievergelt. 1983. The Grid File: A Data Structure Designed to Support Proximity Queries on Spatial Objects. Institut Fuer Informatik Zurich (SWITZERLAND).

Hu, F., M. Xu, J. Yang, Y. Liang, K. Cui, M. M. Little, C. S. Lynnes, D. Q. Duffy, and C. Yang. 2018b. "Evaluating the Open Source Data Containers for Handling Big Geospatial Raster Data." *ISPRS International Journal of Geo-Information* 7 (4): 144.

Hu, F., C. Yang, J. Schnase, D. Duffy, M. Xu, M. Bowen, and T. Lee. 2018a. "ClimateSpark: An In-Memory Distributed Computing Framework for Big Climate Data Analytics." *Computers & Geosciences* 15: 154–166.

Jiang, Y., M. Sun, and C. Yang. 2016. "A Generic Framework for Using Multi-Dimensional Earth Observation Data in GIS." *Remote Sensing* 8 (5): 382.

Li, S., S. Dragicevic, F. A. Castro, M. Sester, S. Winter, A. Coltekin, C. Pettit, et al. 2016a. "Geospatial Big Data Handling Theory and Methods: A Review and Research Challenges." *ISPRS Journal of Photogrammetry and Remote Sensing* 115: 119–133.

Li, Z., C. Yang, K. Liu, F. Hu, and B. Jin. 2016b. "Automatic Scaling Hadoop in the Cloud for Efficient Process of Big Geospatial Data." *ISPRS International Journal of Geo-Information* 5 (10): 173.

Li, Z., F. Hu, J. L. Schnase, D. Q. Duffy, T. Lee, M. K. Bowen, and C. Yang. 2017a. "A Spatiotemporal Indexing Approach for Efficient Processing of Big Array-Based Climate Data with MapReduce." *International Journal of Geographical Information Science* 31 (1): 17–35.

Li, Z., Q. Huang, G. J. Carbone, and F. Hu. 2017b. "A High Performance Query Analytical Framework for Supporting Data-Intensive Climate Studies." *Computers, Environment and Urban Systems* 62: 210–221.

Li, Y., Y. Jiang, F. Hu, C. Yang, T. Huang, D. Moroni, and C. Fench. 2016, September. "Leveraging Cloud Computing to Speedup User Access Log Mining." OCEANS 2016 MTS/IEEE monterey, 1–6. IEEE.

Li, X., Y. J. Kim, R. Govindan, and W. Hong. 2003, November. "Multi-dimensional Range Queries in Sensor Networks." Proceedings of the 1st international conference on embedded networked sensor systems, 63–75. ACM.

Lynch, C. 2008. "Big Data: How do Your Data Grow?" *Nature* 455 (7209): 28.

Ma, Y., H. Wu, L. Wang, B. Huang, R. Ranjan, A. Zomaya, and W. Jie. 2015. "Remote Sensing big Data Computing: Challenges and Opportunities." *Future Generation Computer Systems* 51: 47–60.

Nievergelt, J., H. Hinterberger, and K. C. Sevcik. 1984. "The Grid File: An Adaptable, Symmetric Multikey File Structure." *ACM Transactions on Database Systems (TODS)* 9 (1): 38–71.

Ooi, B. C., K. J. McDonell, and R. Sacks-Davis. 1987, October. "Spatial kd-Tree: An Indexing Mechanism for Spatial Databases." IEEE COMPSAC 87, 85.

Palamuttam, R., R. M. Mogrovejo, C. Mattmann, B. Wilson, K. Whitehall, R. Verma, L. McGibbney, and P. Ramirez. 2015, October. "SciSpark: Applying in-Memory Distributed Computing to Weather Event Detection and Tracking." 2015 IEEE international conference on Big data (Big data), 2020–2026. IEEE.

Robinson, J. T. 1981, April. "The KDB-Tree: A Search Structure for Large Multidimensional Dynamic Indexes." In proceedings of the 1981 ACM SIGMOD international conference on management of data, 10–18. ACM.

Rusu, F., and Y. Cheng. 2013. "A Survey on Array Storage, Query Languages, and Systems." arXiv preprint arXiv:1302.0103.

Wang, C., F. Hu, X. Hu, S. Zhao, W. Wen, and C. Yang. 2015. "A Hadoop-Based Distributed Framework for Efficient Managing and Processing Big Remote Sensing Images." *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 2 (4): 63.

Yadav, R. 2015. *Spark Cookbook*. Packt Publishing.

Yang, C., Q. Huang, Z. Li, K. Liu, and F. Hu. 2017a. "Big Data and Cloud Computing: Innovation Opportunities and Challenges." *International Journal of Digital Earth* 10 (1): 13–53.

Yang, C., M. Yu, F. Hu, Y. Jiang, and Y. Li. 2017b. "Utilizing Cloud Computing to Address big Geospatial Data Challenges." *Computers, Environment and Urban Systems* 61: 120–128.

Zhou, K., Q. Hou, R. Wang, and B. Guo. 2008. "Real-time kd-Tree Construction on Graphics Hardware." *ACM Transactions on Graphics (TOG)* 27 (5): 126.