CHAPTER 3

3. SPATIAL DATA PLACEMENT IN HADOOP DISTRIBUTED FILE SYSTEM

3.1. INTRODUCTION

In the previous chapter, we have presented the necessary background and surveyed related work. In this chapter, we present Co-SpatialHadoop, our main thesis contribution. The chapter is organized as follows: In Section 3.2, we give overview to our work, and present the system architecture of Co-SpatialHadoop. In Section 3.3, we discuss a use case to explain the goals and challenges of colocation. In Section 3.4, we discuss how we colocate files based on their spatial attribute. In Section 3.5, we discuss some problems that challenge colocation enhancement, and we explain how we addressed these challenges in our proposed Co-SpatialHadoop. In Section 3.6, we present the implementation of Co-SpatialHadoop algorithms. In Section 3.7, we explain the index layer enhancement using inverted indexes. Finally, Section 3.8 concludes this chapter.

3.2. OVERVIEW

In the previous chapter, we discussed how the various Hadoop-based spatial systems proposed in the literature have used spatial indexes to enhance the performance of the execution of spatial queries. It was noted that all systems have not changed HDFS and therefore it uses a default placement policy to store the data blocks. They are the same blocks indexed by the spatial indexes maintained by these systems. The HDFS default block placement policy guarantees load balancing and data availability over Hadoop cluster without considering spatial data characteristics. Co-SpatialHadoop is inspired by Co-Hadoop [2] approach, and extends HDFS by the *new spatial block placement policy* to colocate blocks of spatial files that are accessed together by queries. Blocks colocation eliminates data shuffling in MapReduce tasks, and it decreases network overhead because data processed by any map task will be located on the node executing this map task.

Co-SpatialHadoop uses SpatialHadoop [1] as a base for this new introduced work. SpatialHadoop has addressed several spatial challenges but it has not changed HDFS. Besides, it is a well-implemented open source system.

SpatialHadoop indexes spatial records using the spatial attribute only. If the user needs to query data by any other non-spatial attribute, all file blocks need to be scanned. Co-SpatialHadoop is inspired by the approach proposed in [24], and it builds non-spatial indexes using inverted indexes to enhance the performance of non-spatial queries. Additionally, it implements non-spatial operations to test the inverted indexes.

By comparing the SpatialHadoop architecture that is illustrated in Figure 2.10 and the architecture of Co-SpatialHadoop that is illustrated in Figure 3.1, we can notice that Co-SpatialHadoop's implementation is based on SpatialHadoop. The red items are Co-SpatialHadoop enhancements: (1) Spatial Block placement policy: a new policy extended to HDFS to colocate blocks of spatial files that are accessed together by queries. (2) Locator Table: a file used by the new policy to colocate blocks, and it is saved with Hadoop config files on the Master node. (3) Inverted file index: is non-spatial index operation that is added to SpatialHadoop index layer to index files based on non-spatial attributes. (4) Non-spatial operation: a non-spatial operation that is added to SpatialHadoop operations layer to test inverted file indexes.

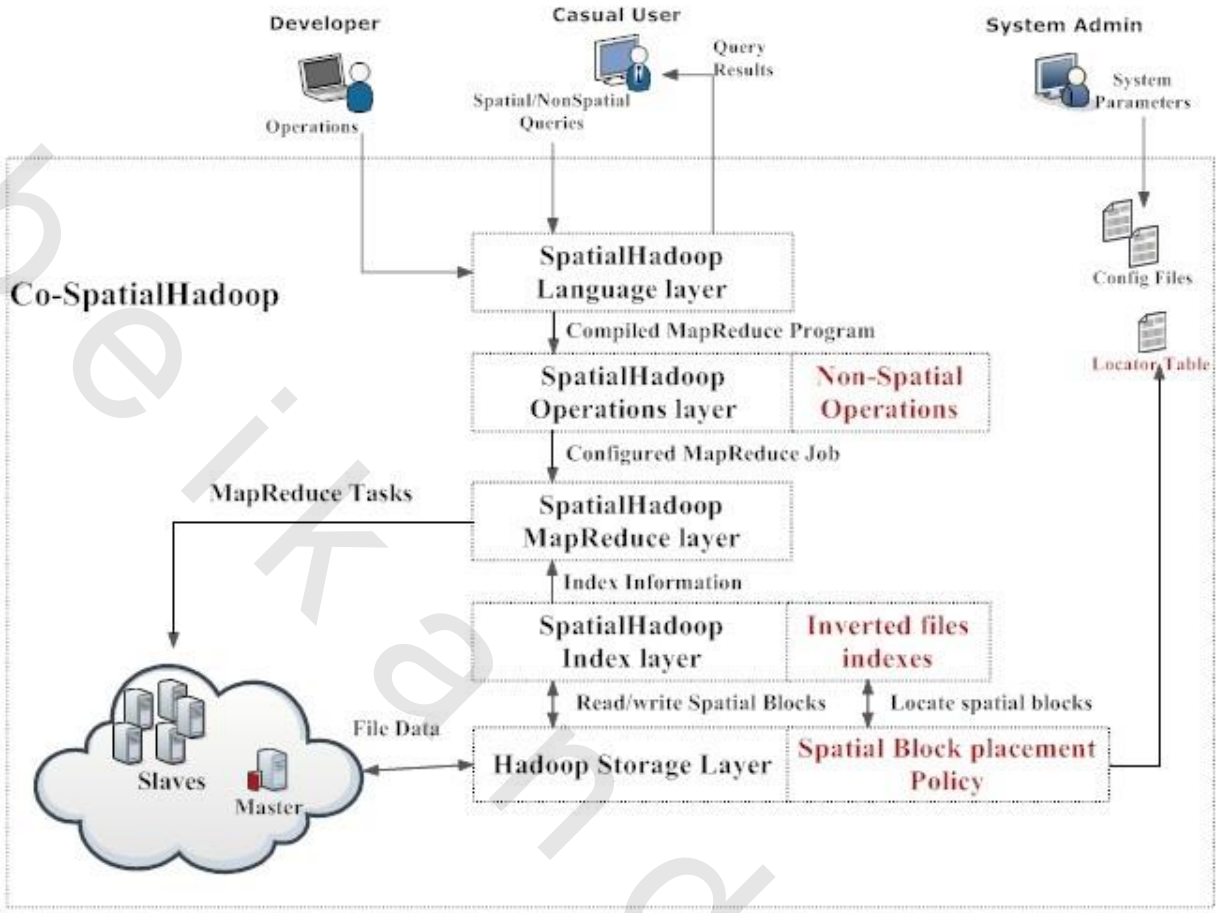

Figure 3.1: Co-Spatial Hadoop Architecture

3.3. SPATIAL JOIN USE CASE

In this section, we shall discuss a spatial join use case to illustrate the main goals of colocating spatial data files.

To join two *non-indexed* spatial files, each record from the first file needs a full scan of the second file to merge it with the records that intersect with it. The complexity of the join operation is NxM, where N is the number of records in the first file and M is number of records in the second file. To enhance the performance of this join operation, we may index these files using spatial attributes to avoid repeatedly scanning the file being joined. Each input file is contained spatial records of a map layer (refer to Section 2.1.2), where each record represents an object in the map. We use R-Tree spatial index

to partition each file into multiple tiles, each tile consists of several records and it is saved as a separate block on the cluster. Therefore, the minimum bounding rectangle (MBR) of a tile is a 2-dimentional rectangle that encloses all objects records of this tile. Figure 3.2 shows the partitioning of two files A and B. The figure represents the R-Tree partitioning for each file, and the MBRs of each of these partitions on X and Y axes of the map.[1]

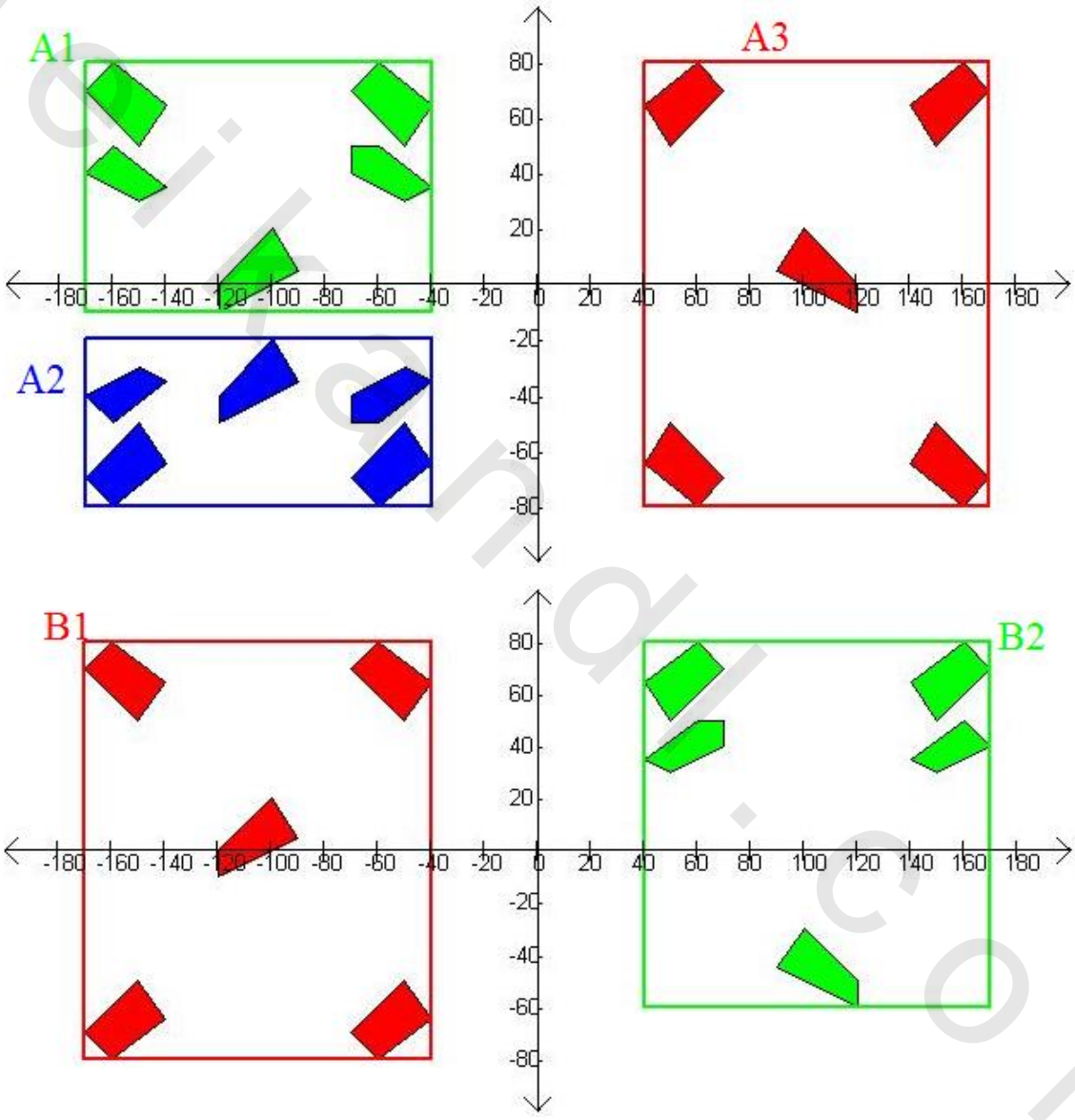

Figure 3.2: Indexed Files "A" and "B"

[1] We used SpatialHadoop [1] R-Tree implementation to partition and index files A and B.

In this example, we are referring the same two files A and B, and their partitioning as shown in Figure 3.2. The advantage of using an index is that blocks with overlapping MBR will only be joined, and therefore to join files A and B using spatial join operation implemented by SpatialHadoop [1], three map tasks will be created to join these partitions: (A1,B1), (A2, B1) and (A3, B2). There are not reduce functions here because we do not need to collect the output record; map functions guarantee that every record of the first file is joined with all intersected records of the other file. We note that there shall be no attempt to join pairs such as (A1, B2) together because of their different MBR. The use of indexes reduces the execution time of the join operation.

As discussed in Section 2.3.2, HDFS replicates each block and places the replicas on the cluster nodes using default block placement policy. The main objective of HDFS placement policy is to maintain data availability, load balance, and fault-tolerance. However, this policy does not account for the spatial properties of the data and therefore, it is not guaranteed that blocks from different files but have the same MBR will be placed on the same nodes. This means that a map task that reads these two blocks as input might need to read one or both of them from remote machines. Next, we show the details of this in the context of the running example.

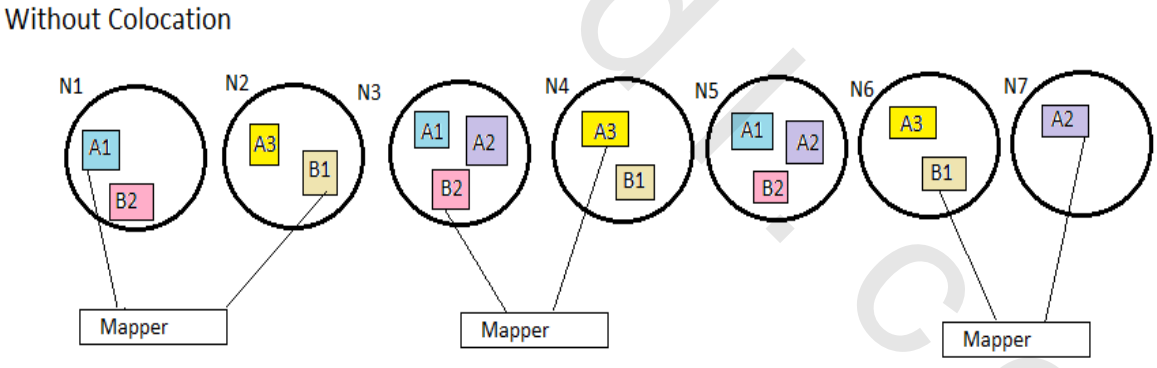

Figure 3.3: Spatial block distribution among nodes without colocation

Figure 3.3 shows random distributions of the blocks and their replicas of files A and B on a cluster of seven nodes. The figure shows that all replicas of A1 and B1 are placed on different nodes. Hadoop JobTracker by default does a best effort to assign a map task to a node, which has all input split blocks. However, if it fails to do so, it assigns it to node that has at least one block (or tile) of the input data and the second block is read

from the remote node through the network. Therefore, map functions might need blocks transmissions and cause network overhead to read input blocks.

Figure 3.4 shows that colocating replicas of A1, A2 and B3 reduces the network overhead caused by reading some of these blocks through the network by map tasks.

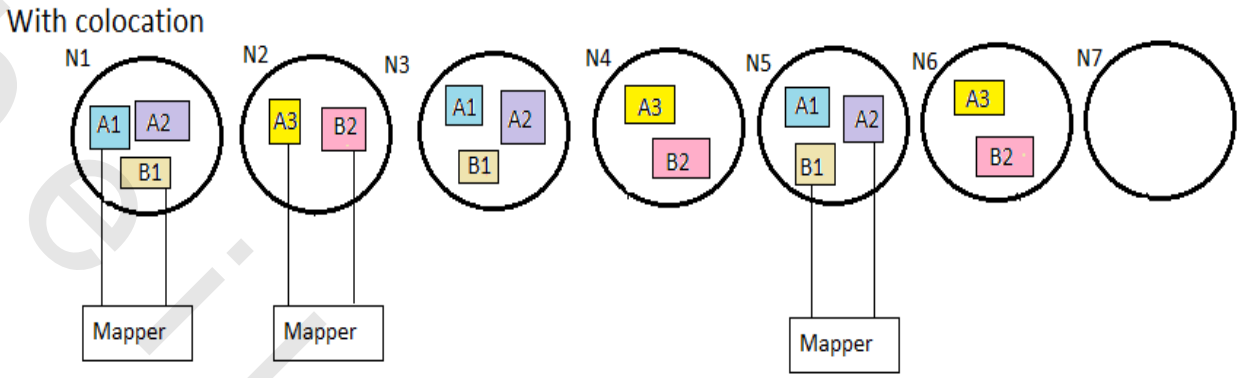

Figure 3.4: Spatial block distribution among nodes with colocation

3.4. COLOCATING FILES BASED ON THEIR SPATIAL PROPERTIES

As discussed in Section 2.3.2, HDFS replicates each block and places the replicas on the cluster nodes using the default block placement policy. If the default replicas number of the cluster is three, the default block placement policy places the first replica to a random node if the client is outside the cluster, if not the policy places the replica in the same client node. The second node is placed in a different rack, and the last replica is placed in another node in the same rack as the second replica. The main objective of HDFS block placement policy is to maintain data availability, load balance, and fault-tolerance without consider spatial data characteristics.

In this section, we introduce a new block placement policy that is based on the spatial properties of the data. We use this placement policy to colocate spatial data files that are accessed together in the same queries using an approach that is similar to that introduced in Co-Hadoop [2]. This approach is based on using a structure called "Locator Table" to keep track of the locations where partitions of file blocks are placed. Next, we

describe how we use the locator table to colocate files based on their spatial characteristics.

3.4.1. Locator Table

The Locator Table structure is introduced by Co-Hadoop [2] to control the placement of data blocks in HDFS. Each file stored in HDFS, is assigned to a locator item, and is added to the locator table. Files that we would like to colocate together (i.e. the partitions of their blocks are placed on the same set of nodes of the cluster). Before saving blocks of a file, the new placement policy browses the locator table for the file's "Locator Item". If the locator item has an entry in the table, the nodes that save blocks of the files that are assigned to this locator item are retrieved, and the blocks of the new file are placed on these set of nodes. If the locator item is not found in the table, a new entry with this locator item is created and the file is added to this entry. The default placement policy is now used to store this file. There is an N:1 relationship between files and locators items: Each file in HDFS is assigned to at most one locator item and many files can be assigned to the same locator item. Experiments in [2] show that load balance is guaranteed using this placement policy.

Next, we describe how we use a similar approach to colocate spatial data files stored in HDFS.

3.4.2. Spatial Locator Table

In Co-Hadoop, the objective is to colocate files, and therefore, a locator item is assigned to each file, and files with the same locator items are to be colocated together. However, for spatial data, our main objective is to colocate blocks that have the same MBR and therefore we assign similar locator items to them. To colocate spatial blocks based on the spatial attribute, the locator table items must be spatial too, therefore, instead of using integer ID values to identify locator items, we use MBRs. Each locator item represents a geographic rectangle area identified by its 2-diemntional coordinates to represent an MBR. A spatial block is assigned to a spatial locator item that represents the MBR that contains it.

If we apply the above rules for creating a spatial locator table on the example presented in Section 3.3, the spatial locator table contains two spatial locator items: (1) a locator item for the negative x-axis area, and (2) a locator item for the positive area. So A1, A2 and B1 are assigned to the first locator item. A3 and B2 are assigned to the second locator item. Table 3.1 shows the spatial locator table corresponding to that use case. The spatial locator table has two columns: Locator column and Blocks column. Locator column contains different locator items, and the blocks column contains the list of blocks assigned to each locator item. This table is stored on the disk with Hadoop's configuration files and each time the NameNode is restarted, it is loaded from the disk to the memory in a Hash Map.

locator	Blocks
L1:MBR[-170,-80,-40,80]	A1, A2, B1
L2:MBR[40,-80,170,80]	A3, B2

Table 3.1: use case Locator Table

3.5. CHALLENGES OF COLOCATING SPATIAL DATA

There are some challenges that affect the query execution performance enhancement obtained from colocating spatial files accessed by query. In this section, we discuss two problems: balancing data assigned to locator items and overlapping between locator items.

3.5.1. Balancing Data among Locator Items

The main goal of our proposed method for spatial data colocation is to colocate related spatial blocks together and place them on the same node to reduce data transmissions between nodes in different spatial operations. This means that we need to allocate data blocks that represent objects on the map in nearby locations to the same cluster nodes.

As mentioned in Section 3.3, the spatial join operation is implemented as a map only function that has an input as two blocks, one from each file, which has overlapping MBR. If the two blocks read by the map function have a replica on the same node, Hadoop assigns the map task to this node with its best effort to save remote over network. At the same time, the block placement policy needs to maintain load balance among all nodes. In HDFS, this is done through assigning new blocks to the nodes with the smallest data size.

Therefore, the main objectives of our proposed placement policy are: (1) locating blocks that will be accessed together by the same map tasks on the same nodes and (2) load balancing of data on a cluster of nodes. The former objective is achieved by using the spatial locator table to keep track of node placements. To achieve the second objective, we need to balance blocks among all nodes, which can be done by choosing the spatial locator items with the right MBRs to guarantee balanced data densities.

Accordingly, we use the R-Tree partitioning of files as hints to how we should choose the right MBRs as locator table items. R-Tree cells info of several files sizes help to choose the right locator items number and the MBR of each locator item. The constraints that we have are as follows. The number of locator items number must not be small to distribute data blocks among them between all nodes, and it must not be big to allow overlapping blocks to be assigned to the same locator items. If the number of locator items is big, the MBR are of each item will be small, and they cannot contain the tiles of small files because they have larger MBRs.

Experiments prove that it is better to build the locator table using big file after choosing locator items number suitable to file size, because its data density balances more locator items.

3.5.2. Overlapping Between MBRs of Locator Items

The objective of colocation algorithm is to place partitions of data blocks that have overlapping MBRs on the same nodes, hence colocating them. However, data colocation is not perfect and some blocks with overlapping MBRs may be assigned to two different locator items. This assignment affects the number of blocks that are read remote-

ly during execution. It is important to carefully select the MBRs of the different locator items to reduce the number of data blocks in the overlapping area between them.

Figure 3.5 illustrates an example of blocks overlapping with the MBR of multiple locator items. The dotted red blocks L1 and L2 are two locator items. B1 to B8 are blocks. B1, B2, B3 and B4 are assigned to L1, and B5, B6, B7 and B8 are assigned to L2. In the figure, the MBRs of B1 and B6 are overlapping. However, each one of them is assigned to a different locator item. If B1 and B6 are now accessed together by a mapper executing a query, at least one of them will be read remotely. The same observation holds for B3 and B8.

The style used to choose the MBRs of the locator items affects the number of blocks that overlaps between multiple locator items. This problem cannot be completely avoided but it can be reduced. From our experience, its effect is minimal, and therefore, we leave the exploration of more techniques to avoid this problem to future work.

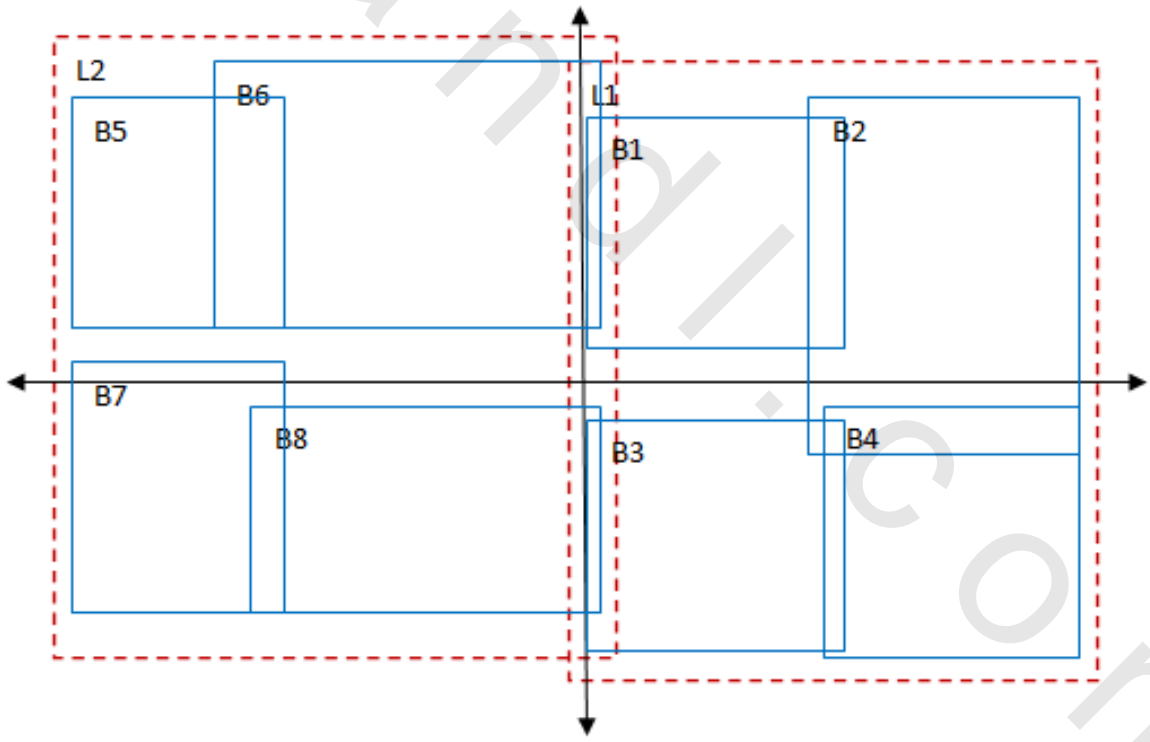

Figure 3.5: An example demonstrating the overlapping between data blocks and locator items

3.6. COLOCATION ALGORITHMS

In this section, we introduce the algorithms used by Co-SpatialHadoop to build the locator table while addressing all challenges that we have discussed. Besides, we introduce the new spatial block placement policy, and discuss how it uses the spatial locator table to assign suitable locator item to each indexed block.

3.6.1. Default Spatial Indexing Algorithm

Here, we discuss the algorithm used by the R-Tree index operation of SpatialHadoop [1] because it is the base of our algorithm. Algorithm 1 outlines the R-Tree indexing operation introduced by SpatialHadoop. The steps used by this algorithm is performed as follows: (1) R-Tree reads random sample from the spatial file to prepare tiles MBRs, which will be used later by Co-SpatialHadoop to build the locator table. (2) R-Tree launches MapReduce job to partition the spatial file. The map functions assign each record to the suitable tile and the reduce functions collect each tile records. (3) Then, each reduce function writes its tile using the default write operation that was discussed in Section 2.3.2 and illustrated by Figure 2.7.

Algorithm 1: default placement for new spatial block

```
SpatialHadoop "RTree" reads sample records of the file to partition it to some cells
"RTree" Map functions assigns each record to the suitable cell
"RTree" Reduce functions combines all records assigned to each cell
Foreach Reduce function
    "RTree" creates "FSDataOutputStream" to write the reduced cell to HDFS
    "RTree" calls "DFSClient" of the current node
        "DFSClient" calls "NameNode" to locate dataNodes for block (cell) replicas
            "NameNode" calls "FSNameSystem" to book dataNodes
                "FSNameSystem" calls "DefaultBlockPlacementPolicy" to choose nodes
                    choose available nodes which save load balance
                    Return DataNodes names list
                "FSNameSystem" assigns the returned list to the block
                "FSNameSystem" returns block
            "NameNode" returns block
        "DFSClient" uses "FSDataOutputStream" to write block to the assigned nodes
    Close "FSDataOutputStream"
EndForeach
```

This algorithm needs some updates to be able to use the new block placement policy introduced by Co-SpatialHadoop to colocate spatial blocks, instead of using the default block placement policy of HDFS write procedure.

3.6.2. Building Spatial Locator Table That Guarantee Load Balancing

To use new block placement policy introduced by our work, we need to build the spatial locator table. As we mentioned in Section 3.5.1, to build a spatial locator table that guarantees load balance, we can use the R-Tree partitioning method of files as hints to choose the right MBRs as locator table items. In the first block assignment of first index operation, the locator table is not ready yet. The new block placement policy uses cells info of R-Tree to build the locator table then assigns suitable item to this block. The next paragraph gives more information about R-Tree cells info and how new block placement policy uses it to build the table.

R-Tree partitions the map vertically and horizontally to multiple cells, each cell area depends on data density of that area on the map (i.e. the number of objects, and hence, records in the spatial data files that have coordinates enclosed in this cell). The R-Tree balances the amount of data in all cells. Co-SpatialHadoop uses the partitioning of the R-Tree to the data file as a seed when building the spatial locator table. Each locator item is assigned to a map region which consists of some adjacent R-Tree cells of the indexed spatial data file.

Algorithm 2: Build Locator Table

```
K is number of locator items
H is number of horizontal locator items
if (PartitioningStyle is vertical)
    partition CellsInfo.columns to K groups, each group contains adjacent columns
    assign each group to LocatorItem
    LocatorItemMBR = assigned group MBR
else if (PartitioningStyle is vertical & horizontal)
    partition CellsInfo.columns to K-H groups, each group contains adjacent columns
    partition top CellsInfo.rows to H groups, each group contains adjacent columns
    number of cells in each group must be equal
```

Co-SpatialHadoop uses two styles to build the spatial locator table: (1) Vertical partitioning style, it partitions the spatial locator table vertically to K regions, and (2) combined partitioning style (Vertical and Horizontal), it partitions the spatial locator table vertically and horizontally to K regions. Figure 3.6 and Figure 3.8 illustrate the difference between the two styles; these locator tables have five regions. Algorithm 2 illustrates the building algorithm used by Co-SpatialHadoop. If cluster administrator wants to partition the locator table vertically to K items, the algorithm partitions the columns

of the R-Tree cells info to K groups, each group contains adjacent columns, and each column consists of a group of cells. If the cluster administrator wants to use the combined style to K-H vertical regions and H horizontal regions, the algorithm partitions the columns of the R-Tree cells info to K-H groups, and it partitions the top rows of R-Tree cells to H regions vertically. We chose top rows because this area on the map has high data density. Locator table items regions must contain equal number of R-Tree cells to balance the locator table. Figure 3.6 illustrates locator table partitioned vertically and horizontally into five regions: four vertical regions and one horizontal region. Figure 3.7 is partitioned vertically and horizontally also into six regions: four vertical regions and two horizontal regions. Figure 3.8 is partitioned vertically into five regions.

The number of locator items is chosen by the database administrators based on their experience. For example, our experiments use a cluster of 10 nodes and spatial data files up to 20 GB, therefore, the experiments prove that locator table with K=5 items has better results than locator table with K=6. Moreover, the experiments show the effect of the partitioning style on the performance of the system, and combined portioning style has better results. We can use an automated technique to choose k items and effective partitioning style, but we leave such procedures for future work.

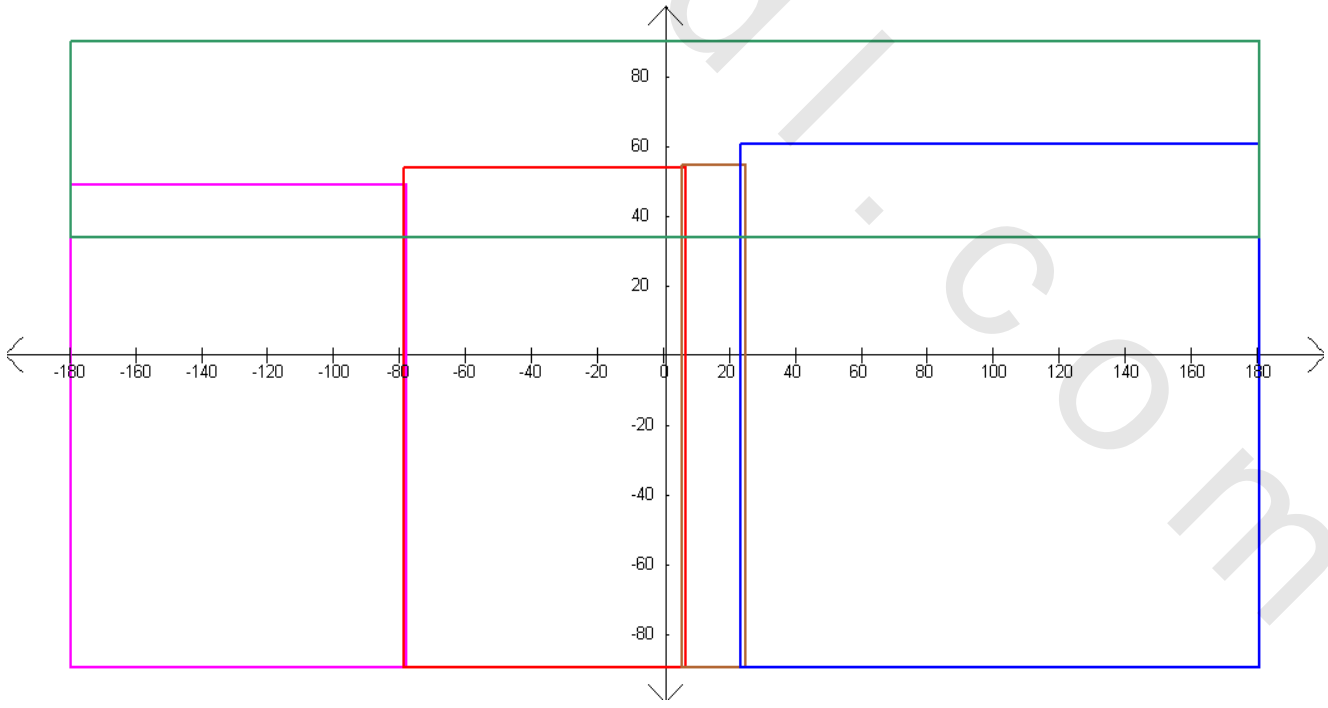

Figure 3.6: Locator Table is partitioned to five partitions

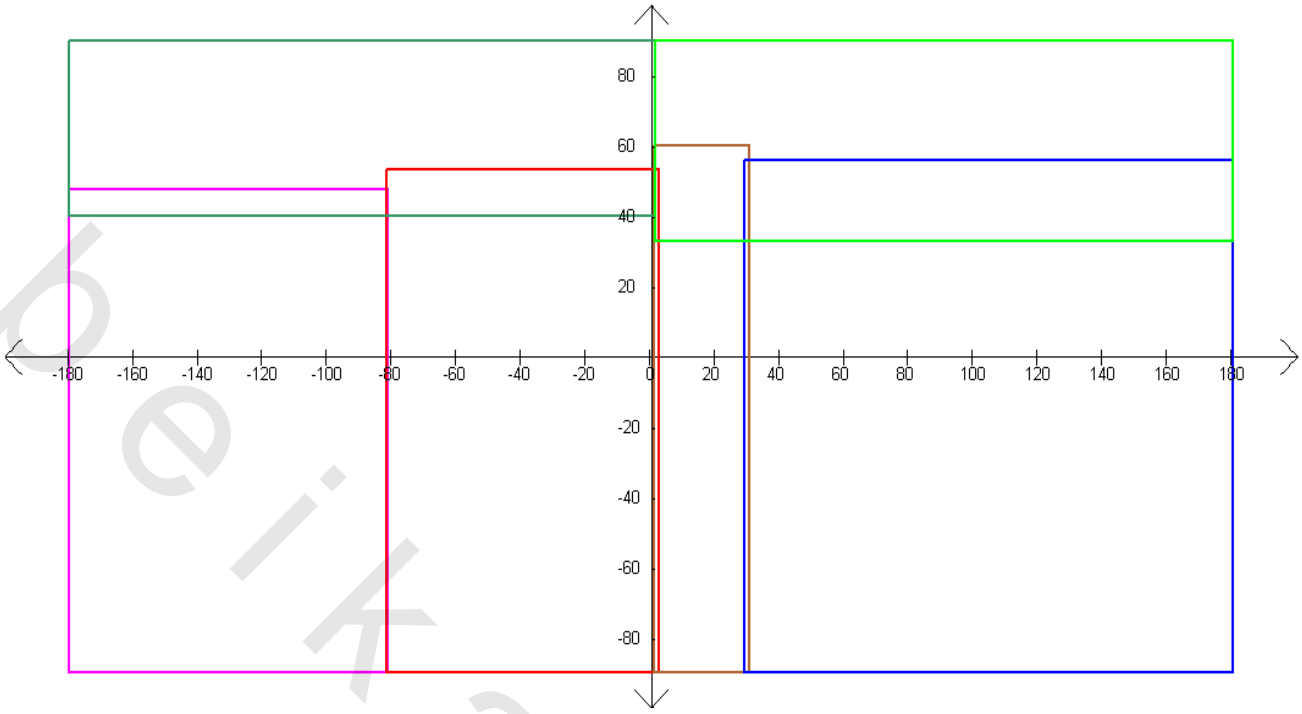

Figure 3.7: Locator Table is partitioned to six partitions

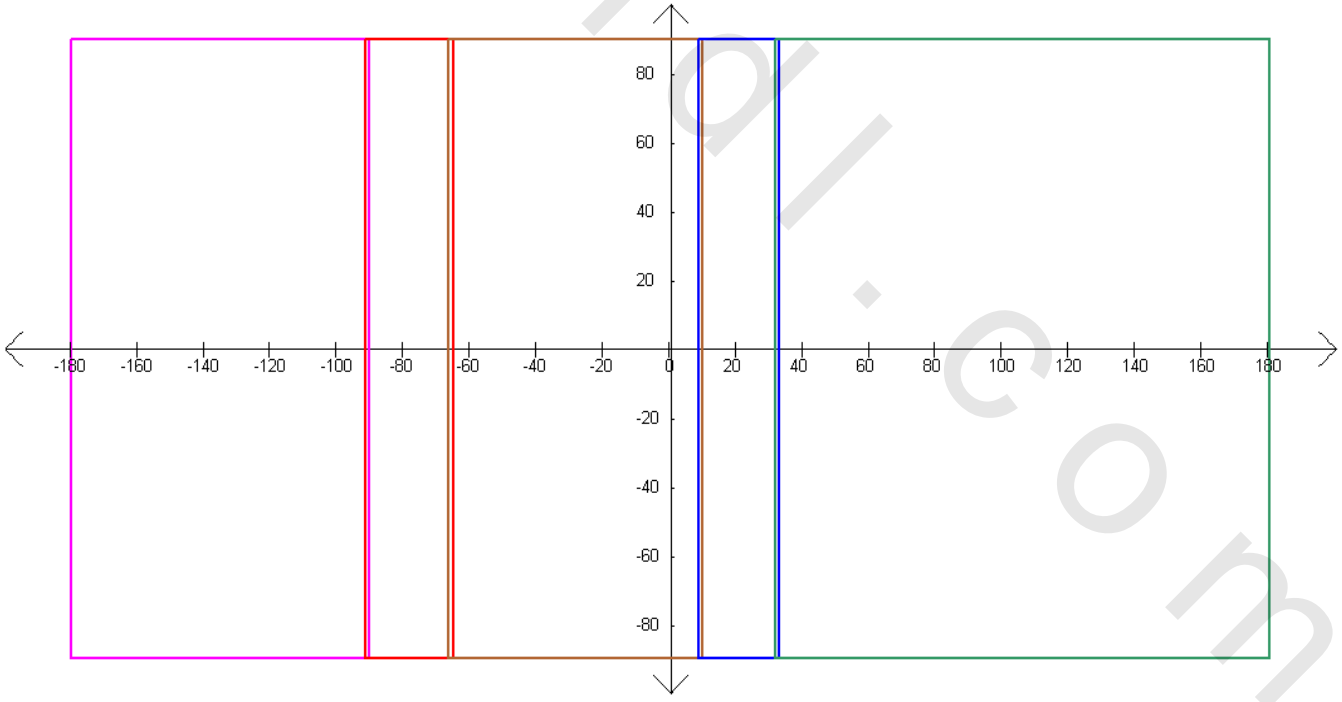

Figure 3.8: Use vertical partitions only

3.6.3. Choosing Suitable Locator Item

Now that we have populated the spatial locator table with a set of locator items, the spatial block placement policy can assign a suitable locator item for each block in the spatial files. Algorithm 3 outlines the assignment method that is used by the spatial block placement policy.

Algorithm 3: Assign locator item to new block and choose suitable DataNodes

```
1  Foreach LocatorItem of LocatorTable
2      If (LocatorItem intersect wih Block and IntesectedArea/BlockArea >= 0.8)
3          ChosenLocators.add (LocatorItem)
4  EndForeach
5  choose one from ChosenLocators array which has less blocks number
6  If (ChoosenLocatorItem has Blocks assigned to it)
7      Foreach Block assigned to ChoosenLocatorItem
8          browse FileSystem to get DataNodes of Block replicas
9          add DataNodes to DataNodesList
10     EndForeach
11     Foreach DataNode of DataNodesList
12         if(DataNode maintains data distribution property)
13             ChoosenDataNodes.add(DataNode)
14     EndForeach
15     if (ChoosenDataNodes.length < Replicas.length)
16         use DefaultPlacmentPolicy to choose reaming DataNodes
17 else
18     use DefaultPlacmentPolicy to choose suitable DataNodes
```

(1) For each block in the file, we select all locator items that have an MBR that overlaps with the block MBR. The MBR of the selected locator items must have at least a percentage of 80% overlap with the MBR of the file block (Algorithm 3, line 2). The experiments show that 80% is a good overlap threshold. (2) Next, we select one of these locator items. For load balancing purpose, we select the locator item with the fewest number of blocks (Algorithm 3, line 5). (3) The next step is to retrieve the nodes that have replicas of the blocks assigned to the chosen locator item. If this is the first block assignment to this locator item, the spatial policy uses the default placement policy by HDFS to choose the nodes to place the replica of this block on (Algorithm 3, line 18). Otherwise, we query the NameNode for the node IDs that have replicas of all the blocks assigned to this locator item (Algorithm 3, lines 7-10). We then place replicas of the block on these nodes while maintaining the same load balancing property of HDFS. If the number of retrieved nodes is not sufficient or does not have enough space for stor-

ing the block replicas, the spatial placement policy calls the HDFS default placement policy to choose new nodes to place the replicas of the block on them (Algorithm 3, lines 11-16).

Note that even using this policy, there are still blocks that can be joined together and are assigned to different locator items. We have mentioned that in Section 3.4.2

3.6.4. Integrating New Colocation Algorithm with SpatialHadoop

Algorithm 1 has illustrated the default algorithm used by SpatialHadoop [1] to index spatial file using the R-Tree algorithm. Algorithm 2 has described the building procedure of spatial locator table used by new spatial placement policy. Algorithm 3 has described how the new spatial placement policy assigns locator items to indexed spatial blocks. In this section, we integrate all these three algorithms to Algorithm 4 to give a general view to the new indexing operation using SpatialHadoop and the new spatial placement policy.

Algorithm 4: Spatial placement for new spatial block

```
1  SpatialHadoop "RTree" reads sample records of the file to partition it to some cells
2  "RTree" Map functions assigns each record to the suitable cell
3  "RTree" Reduce functions combines all records assigned to each cell
4  Foreach Reduce function
5      "RTree" creates "FSDataOutputStream" to write the reduced cell to HDFS
6      "RTree" sets spatial flag to use spatial placement policy
7      "RTree" calls "DFSClient" of the current node with FileName, CellInfo and FileCellsInfo
8          "DFSClient" calls "NameNode" to locate dataNodes for block (cell) replicas
9              "NameNode" calls "FSNameSystem" to book dataNodes
10                 "FSNameSystem" calls "SpatialBlockPlacementPolicy" to choose nodes
11                     if(LocatorTable is ready)
12                         choose available nodes using CellInfo
13                     else
14                         build LocatorTable using FileCellsInfo
15                         choose available nodes using CellInfo
16                     Return DataNodes names list
17                 "FSNameSystem" assigns the returned list to the block
18                 "FSNameSystem" returns block
19             "NameNode" returns block
20         "DFSClient" uses "FSDataOutputStream" to write block to the assigned nodes
21     Close "FSDataOutputStream"
22     "RTree" clear spatial flag
23 EndForeach
```

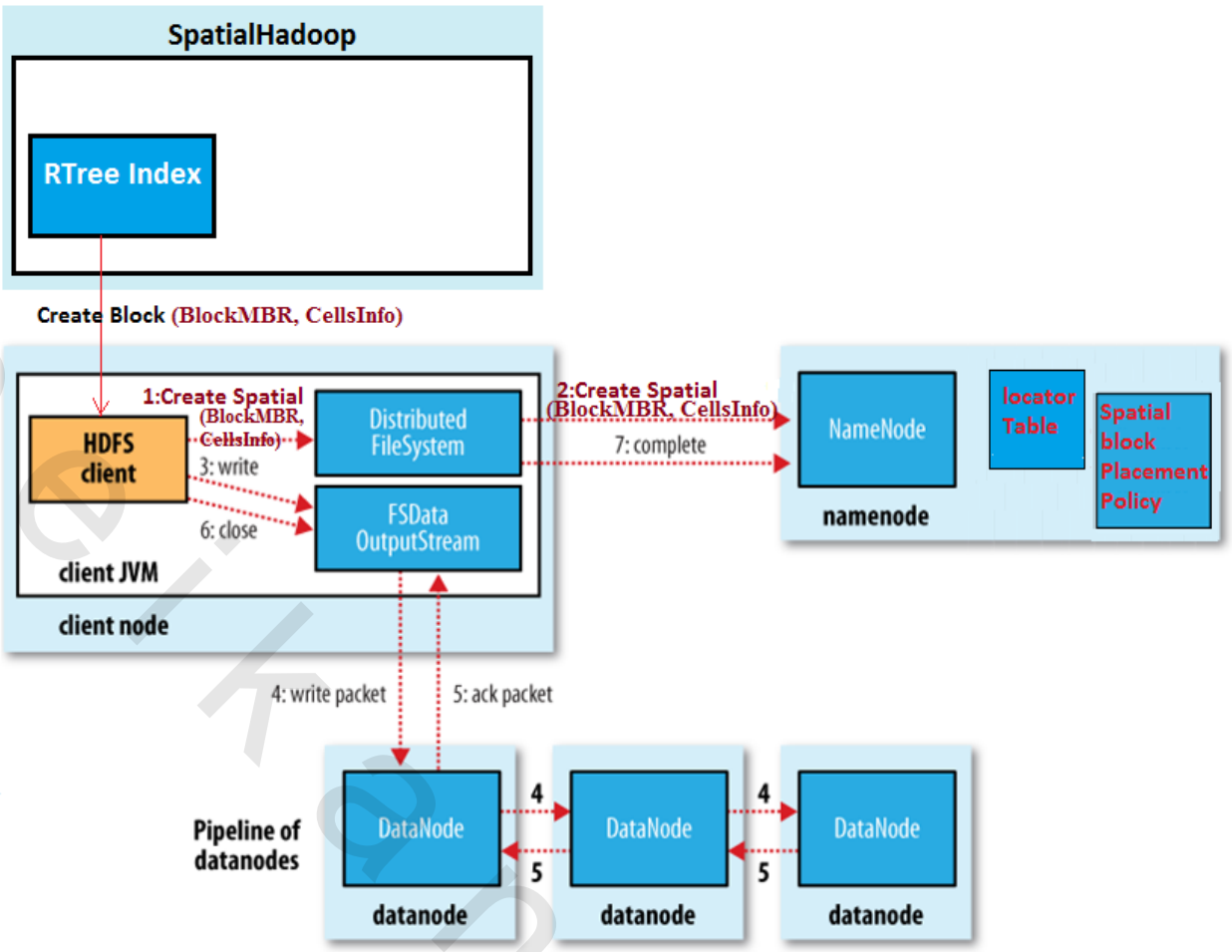

Figure 3.9: Physical architecture for new placement of a new block

Figure 3.9 shows the new updates added by Co-SpatialHadoop to the default HDFS write procedure that is illustrated by Figure 2.7. These new updates can be described as follows: (1) Co-SpatialHadoop updates reduce functions of SpatialHadoop [1] R-Tree to send the cells info of the file and the MBR of the reduced cell beside the block name to the HDFS client while writing the reduced tile. (2) It adds new spatial flag to HDFS client to force it to use new spatial methods. (3) It implements a new spatial "create" method, which sends block MBR and cells info to NameNode. (4) It implements new spatial placement policy to NameNode beside the HDFS default placement policy. (5) Finally, it saves spatial locator table to the NameNode with default configuration files.

The steps of Algorithm 4 can be described as follows: (1) SpatialHadoop R-Tree prepares cells info of the indexed file (Algorithm 4, line 1). (2) It creates a new MapReduce job to index the spatial file. The map functions assign each record to its suitable

cell, and reduce functions collect records of each cell and write it to the HDFS as a separate tile or block (Algorithm 4, lines 2-3). (4) To write the reduced tile, the reduce function sets the new spatial flag of the HDFS client by true, and it sends the cells info of the file and the MBR of the reduced cell beside the block name to the HDFS client (Algorithm 4, lines 5-7). (5) The HDFS client uses the spatial call method to send block MBR and file cells info to the NameNode (Algorithm 4, line 8). (6) The NameNode uses spatial call to the FSNameSystem to force it to use the new spatial placement policy instead of the HDFS default placement policy (Algorithm 4, line 9). (7) The FSNameSystem calls the new spatial block placement policy to choose nodes for this block replicas (Algorithm 4, line 10). (8) If this is the first indexing operation, the placement policy builds the locator table using Algorithm 2, then chooses a suitable locator item for the new block using Algorithm 3 (Algorithm 4, lines 11-15). (9) After choosing the locator, the placement policy returns the chosen DataNodes to the FSNameSystem. The FSNameSystem adds a new entry for this block to the list of blocks assigned to the chosen locator and returns the list of nodes to HDFS client (Algorithm 4, lines 16-19). (10) The client uses FSOutputStream to stream data of this new block into the chosen DataNodes client (Algorithm 4, line 20). (11) Finally, the reduce function closes the FSOutputStream and clears the spatial flag of HDFS client (Algorithm 4, lines 21- 22).

3.7. CONCLUSION

In this chapter, we have discussed the main contribution of this thesis: Co-SpatialHadoop. We have built Co-SpatialHadoop as an extension to SpatialHadoop [1]. We carefully studied the challenges of colocating spatial files that are accessed together by queries. We proposed new techniques for colocating spatial files and implemented them as an extension to the Hadoop Distributed File System (HDFS).