**Indexing Large Spatial Data using Hadoop, and another complementary Quad-Tree and R+ Tree implementation in Java**

Li Wang

**Abstract**

Spatial data is also known as geospatial data or geographic information. It is the data or information that identifies the geographic location of features and boundaries on Earth, such as natural or constructed features, oceans, and more. Spatial data is usually stored as coordinates and topology, and is data that can be mapped. Spatial data is often accessed, manipulated or analyzed through Geographic Information Systems (GIS).

Spatial data are usually very large in volume, spatial object data sets that are of hundreds of gigabytes of are not uncommon. We want to find an efficient way to index the spatial objects set and therefore process and analyze those data efficiently.

My exploration on spatial data can be divided into three parts:

1. Collaborated with Donghan Miao to implement Quad-Tree and translate it into MapReduce Framework
2. Compared the performance of R Tree and Grid File by using SpatialHadoop both on local machine and on cloud
3. Implement R+ Tree and get a closer look at the performance in materialized view

**1. Quad-Tree implementation in Hadoop MapReduce Framework**

**1.1 Hadoop, HDFS and MapReduce**

Designing a distributed system can be very hard. We use MapReduce computing model from the Hadoop to simply the work. The Apache Hadoop is an open-source project for reliable, scalable, distributed computing, which allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. It is designed to detect and handle failures at the application layer, so delivering a highly available service on top of a cluster of computers.

HDFS is short for Hadoop distributed file system. It is a distributed, scalable file system for the Hadoop framework. HDFS stores large files (typically in the range of gigabytes to terabytes) across multiple machines. It achieves reliability by replicating the data across multiple hosts. The file system is also fault tolerant – node failure is detected and handled in the application level. The file system uses the TCP/IP for communication between nodes.

MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. A MapReduce program is composed of a Map() procedure that performs filtering and sorting and a Reduce() procedure that performs a summary operation. The MapReduce System orchestrates by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing fault tolerance.

**1.2 Index Building and data persistence**

In general, database storage consists of persisted index structure and data storage. In big data environment, we leverage the advantage of data partitioning to achieve scalability. Indexing process partitions large data set into even-sized data chunks (the number of chunks is determined by the volume of spatial dataset). Each data chunk is a small but fully-fledged database storage.

Data

**Index**

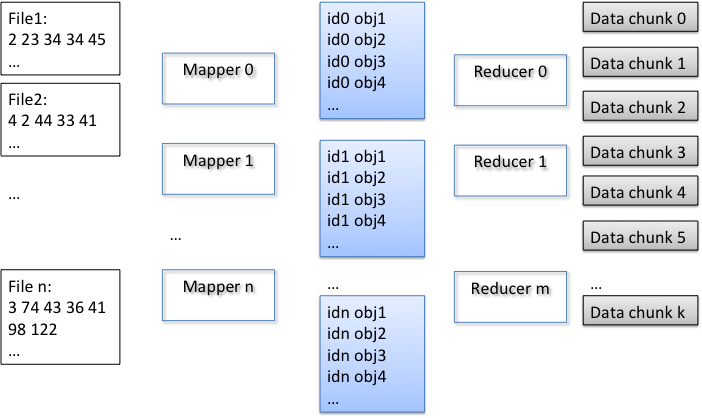
Data chunk, default size limit 64MB

The reason we are doing this is because, 1) by default, the block size of Hadoop distributed file system (HDFS for short hereafter) is 64MB. Files that are smaller than that will take up 64MB actual space. 2) We want to partition the data sets over a cluster of machines. The size of each file should not infinitely scale up. Therefore, one way to solve this issue is to define the default chunk size to a value that should be close to but no larger than the block size, so that I/O performance is not harmed.

The data chunk is organized as follows, it consists of two sections, and both are in binary format. The first section is spatial index structure, which may vary in user’s own choice. In the very beginning, both parts are constructed in memory. When the total size of the two sections grows close to the maximum size limit, we serialize the two parts into persisted file on HDFS.

We implement index building as a MapReduce job. In our case, input data are formatted texts where each line represents a polygon object. Mapper input takes <line number, line text string> as key-value pairs. MapReduce framework will split the files into smaller splits before submitting them to mappers, so that multiple mappers will work in parallel to process the input splits. The level of parallelism is determined by the volume of data set or can be customized by the user.

The following graph shows how indexing building was done is a MapReduce process.



Graph 1, MapReduce process for Tree Building

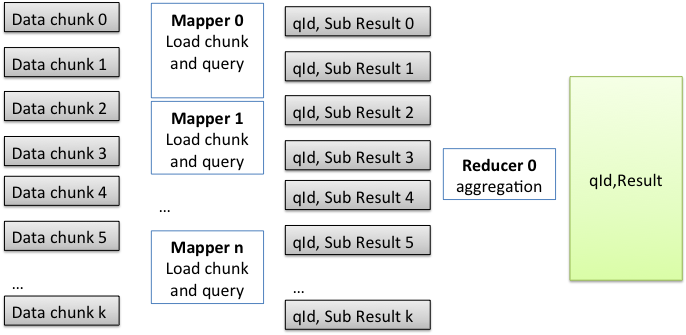
Mappers convert the text string into a spatial object and emits <mapperInstanceID, object> pairs for reducers. The emitted key-value pairs sharing the same key will be sent to same reducer. It is actually during reduce phase that partitioning is performed. The reducers will collect an appropriate number of objects (a number adjusted to make data chunk close to 64MB) and build spatial index for them. Upon finishing index building, the index and raw data will be merged as a single data chunk and emit as a single binary file to the HDFS file system.

**1.3 Executing a Query**

Executing a range query is also a MapReduce job, where input is a number of data chunks built from the first MapReduce job. In each mapper, a customized input format reader was implemented according to our own data format to load the data chunks into memory, where their original data structure are restored – the first section of file is restored as index structures, and the second section of file is simply loaded into a large byte block.

Query was executed in the mapper phase. Each mapper will conduct a search operation in each file scope. Index structure returns a set of object ids (id is a combination of file position and length as position in binary block) for any given range search. According to the object ids, the actual data of the object will be retrieved from the byte array.

The following graph shows how query was done is a MapReduce process.



Graph 2, Query In MapReduce

After that, a set of queried objects is sent to reducer with key being the queryID, value being a spatial object. In this way, results belong to the same query will be aggregated in the same reducer.

**1.4 Spatial Indexing Data Structures**

A Quad-Tree is a tree data structure in which each internal node has exactly four children. Quad-Trees partition a two-dimensional space by recursively subdividing it into four quadrants or regions.

Some modification was made to Quad-Tree for space efficiency. In our real data sets, there exist clusters of objects in which objects are very close to each other geographically. They may result in over division of leaf node, as the consequence, more space are used since the depth of tree increases.

In our design, more than one objects can be stored in a leaf node if the minimal bounding rectangle of an object has intersection with another object in the same node. In this way, objects clusters will could reside on the same leaf node, avoiding over division of the Quad-Tree.

During range query, we just need to judge each object in the node, to see if there is an intersection between each object and the query range.

**2. Performance Comparison between R Tree and Grid-File using SpatialHadoop**

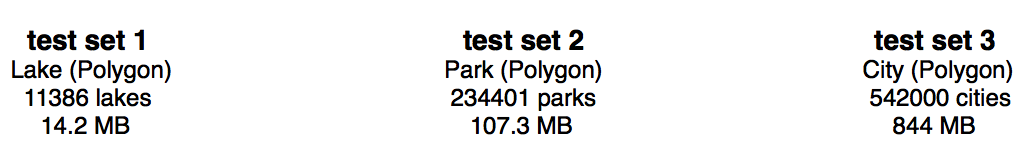
**2.1 SpatialHadoop**

SpatialHadoop is a project developed by University of Minnesota, a MapReduce application to Apache Hadoop, it is designed specially to work with spatial data. It can be used to analyze huge spatial datasets on a cluster of machines. This report dissects SpatialHadoop implementation details. SpatialHadoop supports R+ tree, R Tree, and grid indexing, lacking other popular spatial indexing structures like Quad-Tree, kd-tree, etc. )

**2.2 Objective**

This report mainly focuses on the efficiency of R-tree and Grid-File data structure when conducting indexing and query on spatial data. Also, it attempts to validate if cloud service could speed up the indexing and query response time. This experiment is conducted separately on Local Machine and Amazon Web Service. Both R-tree and Grid-File indexing method are borrowed from SpatialHadoop, which is a MapReduce framework for spatial data. This open source software is an extension on Apache Hadoop.

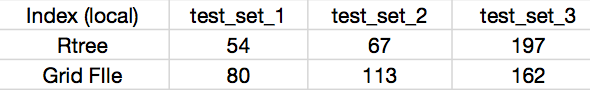
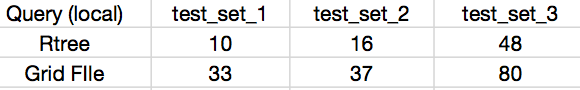
**2.3 Test Set**



\* Data on test set 2 and test set 3 are retrieved from <http://spatialhadoop.cs.umn.edu/datasets.html>

**2.4 Local Test**

Index Query

It appears that during the indexing process, R-tree performs up to around 2 times better than Grid File on small data set. However, on large data set, the time cost for R-tree indexing is larger than Grid File.

After indexing finished, we separately placed 3 rectangular range queries (1. large 2. medium 3. small) on each of the data sets. Queries are showed below:

test set 1 test set 2 test set 3

R Tree: **10s (1. 11s 2. 10s 3. 9s) 16s (1. 21s 2. 14s 3. 14s) 48s (1. 106s 2. 24s 3. 14s)**

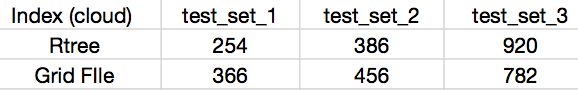
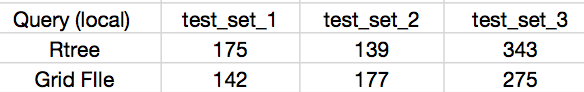
Grid-File: **33s (1. 29s 2. 44s 3. 27s) 37s (1. 42s 2. 33s 3. 35s) 80s (1. 95s 2. 74s 3. 71s)**

The result shows that queries on R-tree costs less time compared with Grid-File. Generally, queries on R-tree can enhance the performance 2 times.

**2.5 Cloud Test**

**2.5.1 3 NODE**

Index Query

On Amazon Web Service tests, it shows both R-tree and Grid-File take similar time to conduct indexing and queries. There is no obvious condition where one outperforms the other.

**2.5.2 3 NODE V.S. 10 NODE**

When we use more slave nodes (increase from 2 to 9), the performance is slightly better on small data set, whereas evidently enhances on large data set.

**2.5.3 Local Test V.S. Cloud Test**

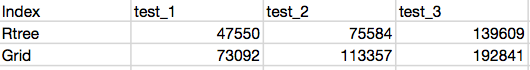
 

Generally, it takes longer time cost addressing spatial data on AWS instead of local machine, sometimes the inefficiency can be up to 5 times. Therefore, for fairly small set of data, it would be extremely diseconomy to run on the cloud.

**2.5.4 Poly-Line Test**

Test set: cut half of the edges in every polygon in Lake, Park, City data sets to make new poly-line data sets.

Index: Query:

**2.5.5 Summary**

1. R-tree performs better than Grid-File on local machine, but there is no obvious enhancement on the cloud. However, it should be mutually corresponded. The result might be satisfactory when using larger data set.

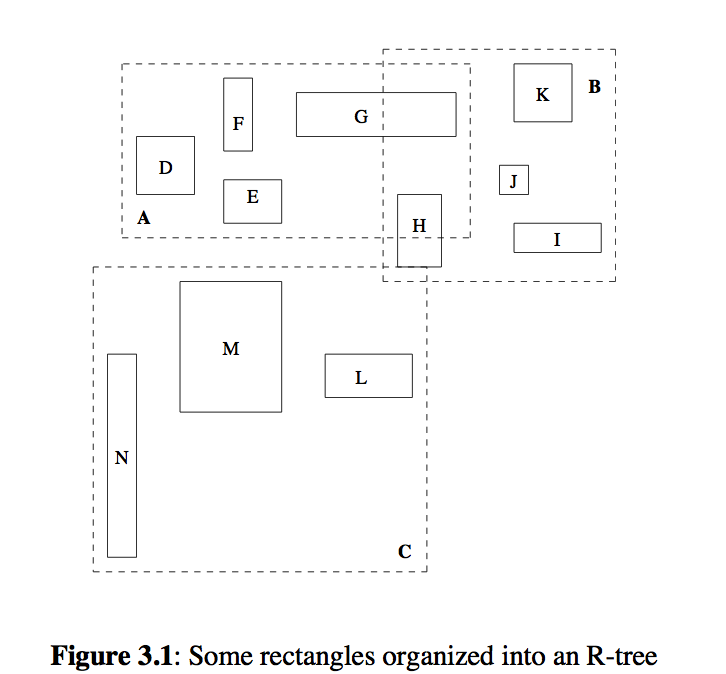
2. When we increase the number of working nodes on the cloud, the efficiency is fairly no difference on small data set, and would be more obvious when data set gets larger.

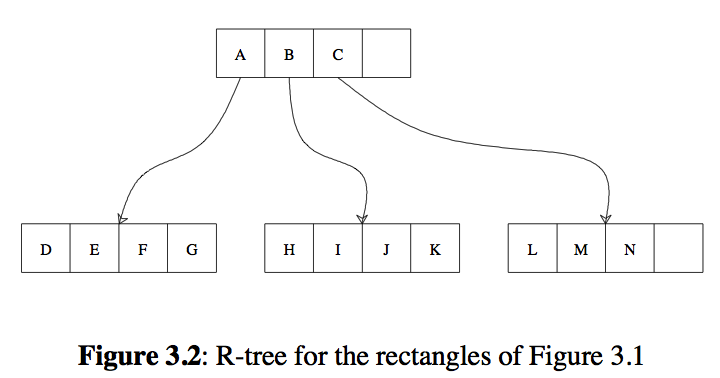
3. Addressing data set on the cloud would cost more time compared with running on local machine, thus it would be inefficient to use cloud computing when the data set is small. The extra time might consumed by the data transportation time between the nodes, the hardware condition of the node, or some features of the hadoop framework, like replica process. However, when data file grows to hundreds of GB or even TB, cloud computing would be good choice.

**3. Materialized Views of R+ Tree**

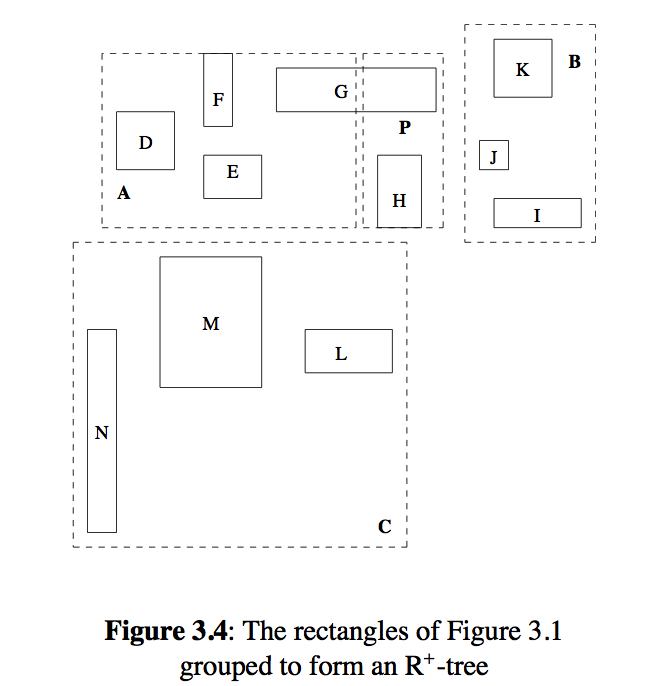
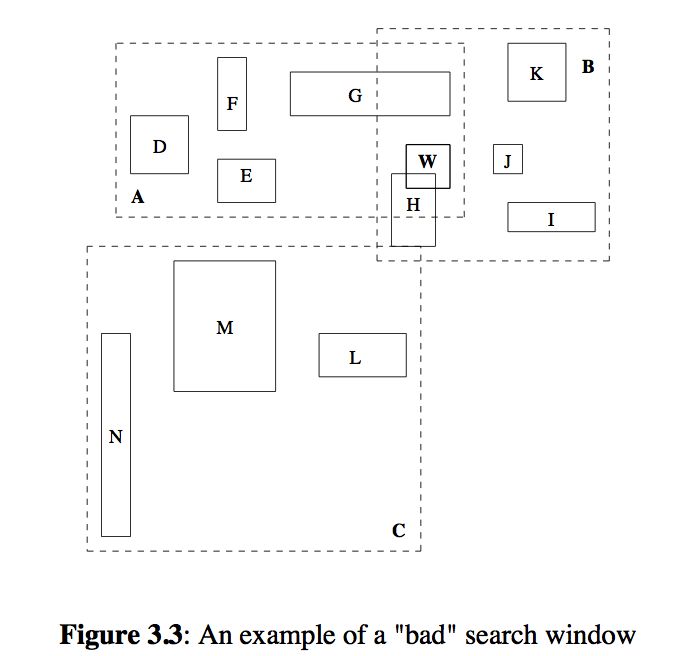
**3.1 R+ Tree and Implementation in Java**

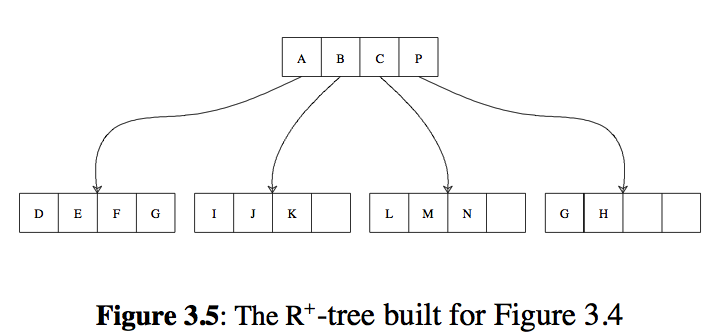
R-trees are tree data structures used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons. The R-tree was proposed by Antonin Guttman in 1984 and has found significant use in both theoretical and applied contexts. A common real-world usage for an R-tree might be to store spatial objects such as restaurant locations or the polygons that typical maps are made of: streets, buildings, outlines of lakes, coastlines, etc. and then find answers quickly to queries.





R+-Tree is a variation to Guttman’s R-Tree that avoids overlapping rectangles in intermediate nodes of the tree. R-Trees suffer in the case of few, large data objects, which force a lot of “forking” during the search. However, R+-Trees handle these cases easily, because they split these large data objects into smaller ones. The result of experimental comparison between R+ Tree and R Tree is shown by the paper “The R+ Tree: A Dynamic Index for Multi-Dimensional Objects”, which reflects the advantages of R+ Tree. The main advantage of R+-Trees compared to R-Trees is the improved search performance, especially in the case of point queries, where there can be even more than 50% savings in the disk accesses. Therefore, the R+-Tree can be used in a database system in order to index any kind of geometric data.





In conclusion, R+ Tree has its own advantages and disadvantages. The advantages are: 1. Because nodes are not overlapped with each other, point query performance benefits since all spatial regions are covered by at most one node 2. A single path is followed and fewer nodes are visited than with the R-tree. The disadvantages are: 1. Since rectangles are duplicated, an R+ tree can be larger than an R tree built on same data set 2. Construction and maintenance of R+ trees is more complex than the construction and maintenance of R trees and other variants of the R tree.

Since there is no R+ Tree code available on the web, I implemented R+ Tree in Java version by refer to the paper “The R+ Tree: A Dynamic Index for Multi-Dimensional Objects”, which gives the general algorithms for search, update and pack the R+ Tree data structure. The most important and difficult part to implement it is to choose the most appropriate position to split the nodes when its children exceed the maximum number. The paper did not write down the specific approach to solve it, instead, it suggest there are some ways can find the best partition place, like nearest neighbor, minimal coverage, minimal splits, etc. The evaluation part is tricky and importance because good partition could result in the optimized performance of R+ Tree.

I chose the most obviously approach to find the good partition place, which is first sort the coordinates both in x-axis and y-axis, and then find the place which could meet the fill factor of the first split node. By using nearest neighbor and minimal coverage algorithm, I evaluated the partition positions both in x-axis and y-axis to determine which one is better.

**3.2 Materialized View of Spatial Data Structure**

Traditional spatial data structure always stores the spatial objects in the tree leaves and all the intermediate nodes represent the maximum rectangular boundaries of their children nodes. The materialized view, which introduced by the paper “Materialized Views for Count Aggregates of Spatial Data”, is to store the objects information into some intermediate nodes, so that when a query region covers the intermediate node, there is no need to get further down to the leaves to figure out what these objects are.

This paper investigate the trade-offs that arise when materialized views of the count aggregate are maintained in a hierarchical index and propose two data structures that are based on the Quad-Tree indexes: Fully Materialized Views (FMV) and Partially Materialized Views (PMV). The experiments demonstrated that the both FMV and PMV approaches yield significant speedups, with reasonable overheads in terms of index maintenance and space.

My work is to compare the FMV and PMV R+ Tree to the traditional R+ Tree. However, there is some limitation of experimental data size, which is below 25k polygons. When I tried to use larger data size, the program get the exception that shows “Exception in thread "main" java.lang.OutOfMemoryError: Java heap space”. This might happened due to several reasons. One might be the memory of my laptop is limited, the other one might be the algorithm of my R+ tree is not optimized, which causes the redundant spaces or unnecessary recursions.

**3.3 Experimental Result**

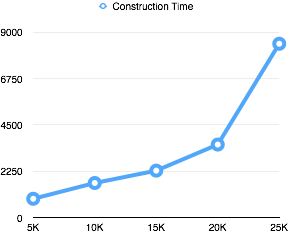
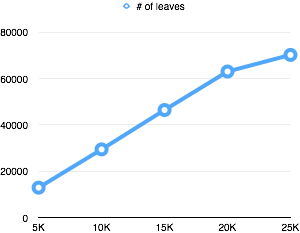
I first try to take a closer look at the properties and characteristics of indexed traditional R+ tree. The experimental data are grasped from Anan Yaagoub, who used the same data sets to conduct experiments on Quad-Tree. The experimental data size, which has a maximum of 25K due to the exception of the memory, is divided into 5 categories. Increased from 5K to 25K, I explored the tree construction time, tree levels, and total number of leaves of the R+ Trees built on the 5 data sets.

The maximum number of entries for each intermediate node in the R+ Tree is to be set as 10. Therefore, for the experimental data size, which has an upper bound to 25K, the tree size is no more than 6 levels. Experimental result is show as below. (Note that the construction time refers to the JVM time).

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Size** | **Construction Time** | **Tree Size (levels)** | **# of leaves** |
| **5K** | 910 | 5 | 12893 |
| **10K** | 1677 | 5 | 29424 |
| **15K** | 2278 | 6 | 46423 |
| **20K** | 3545 | 6 | 63049 |
| **25K** | 8443 | 6 | 70222 |

3.3.1 R+ Tree property comparison on each data size

The result can be saw as more intuitionistic when using charts, which is shown as below.

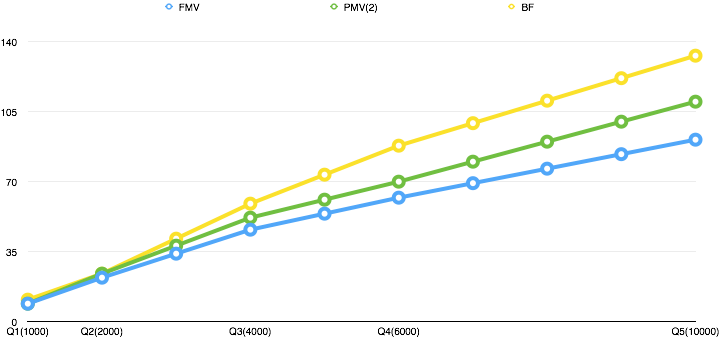
3.3.2 Construction time based on each data size 3.3.3 Number of leaves based on each data size

The two charts for R+ trees built on different data sizes shows that the construction time increases geometrically when the data size increases, whereas the number of leaves grows linearly. The result for the geometric growth on construction time might be that, when the number of objects increases, which causes the number of calls for the insertion function increases. In the insertion method, the recursion of searching for the proper position to insert does not grow linearly but geometrically. Therefore, the time for construction takes much longer when the data size increases.

In the next part, I conducted a experiment to compare the performance of answering queries on FMV, PMV, and traditional R+ Tree (which also can be called tree using Brute Force). The chosen data size is 25K which means the tree level is 6. Especially, for the PMV, I choose to put the objects information into the intermediate nodes each 2 levels. There is five queries, also provided by Anan Yaagoub, conducted on each tree. The difference between the five queries is the areas they covered. Larger area of range query results in more covered objects. Thus, query 1 covers 1000 objects, query 2 covers 2000 objects, query 3 covers 4000 objects, query 4 covers 6000 objects, query 5 covers 10000 objects. The result is shown as below. (Note the number represents the search time, which also refers to JVM time. Each number is the average for 5 experiments.)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Query Time** | **Q1(1000)** | **Q2(2000)** | **Q3(4000)** | **Q4(6000)** | **Q5(10000)** |
| **FMV** | 9 | 22 | 46 | 62 | 91 |
| **PMV(2)** | 9 | 24 | 52 | 70 | 110 |
| **BF** | 11 | 24 | 59 | 88 | 133 |

3.3.4 Query performance on both traditional tree and materialized tree

****

3.3.5 Query performance on both traditional tree and materialized tree

From the chart above, we can see obviously that FMV has the best performance for searching, whereas the traditional R+ tree has the worst performance on each kind of query. The better performance of FMV and PMV R+ Tree can be analyzed as, the object information is stored into intermediate nodes, thus when the query covers a rectangular range, there is no need to go down further to the leave, just return all the leave nodes information as result, which cuts off time cost for unnecessary deeper search. The FMV is the best because it stores all the objects information in every intermediate node, which can stop the unnecessary search as soon as possible. The PMV is also useful, since FMV has the largest space complexity, but PMV can reduce significantly. PMV is the compromise between time complexity and space complexity. If storage space is an issue, PMV should be considered first.

Another importance observation is that, as the range query become larger, the time saving on search is more apparent for both FMV and PMV R+ tree. For Q1, there is little difference between the efficiency of Materialized tree and traditional tree. However, for Q3, the saving time can be up to 22%, and for Q5, the saving time is up to 32%.

An assumption further is that when the tree grows larger, which means when the tree level increases, the efficiency of materialized R+ Tree can also become more obvious.

**4. Summary**

These works are done based on the EECS 590 project “Spatial Objects Indexing using Hadoop”. I appreciate the helps from Goce Trajcevski, Anan Yaagoub, Donghan Miao, who guided me to get deeper understanding about Cloud Computing, Hadoop, MapReduce Framework, SpatialHadoop and tree data structures that stores spatial objects. My work is not perfect, since the R+ Tree cannot addressing large amount of spatial data due to the memory exception, but I still get experiences to the power of cloud computing and materialized view of spatial tree data structure. I’ll keep learning and hope I could do related works in the future!