Spring 5-22-2019

R*-Tree index in Cassandra for Geospatial Processing

Avinashilingam Nanjappan
San Jose State University

R*-Tree index in Cassandra for Geospatial Processing

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Avinashilingam Nanjappan

May 2019

The Designated Project Committee Approves the Project Titled

R*-Tree index in Cassandra for Geospatial Processing

by

Avinashilingam Nanjappan

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2019

ABSTRACT

R*-Tree index in Cassandra for Geospatial Processing

by Avinashilingam Nanjappan

Geospatial data has garnered enough attention in recent times that it is being used everywhere right from simple applications such as booking a taxi ride to complex applications such as autonomous driving. Though the attention towards geospatial processing is something new, substantial research has been going on for years. With the evolution of NoSQL databases in recent times, geospatial processing has attained a new dimension concerning its applications and capability. The most popular NoSQL database to be used for geospatial processing is the MongoDB followed by Cassandra. It is the indexing process that is important concerning the data at hand irrespective of the type of the database. Some of the most common indexes used for the geospatial processing are R-tree, R*-tree, B-tree, Z-curve. R*-tree is the area of our study as it is one among the widely used indexes for geospatial querying. The database of our interest is Cassandra as it is one among the widely used NoSQL database that does not have native support for geospatial query processing. To support geospatial workload, Cassandra should interact with external libraries such as GeoMesa and Solr. In particular, we are interested in the working of the GeoMesa as it uses the Z-curve as the indexing mechanism for the geospatial processing. R*-tree is a dynamic structure capable of representing multi-dimensional data whereas Z-curves are capable of representing multi-dimensional data in a single dimension. In this study, we compare and contrast the performance of R*-tree and Z-curve for various geospatial operations in Cassandra.

ACKNOWLEDGMENTS

I would like to take this opportunity to thank my project advisor Dr. Suneuy Kim, for her ceaseless support and motivation over the months without which this project would have not reached this stage. I would also like to thank my committee members Dr. Christopher Pollett and Dr. Robert Chun for their valuable time and input which enhanced the overall outcome of the project.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Modern day computing involves processing vast volumes of structured and unstructured data from a variety of data sources. Such data is often referred to as "Big Data". Geospatial data are data that include geographical components and frequently appear in big data. With the advent of NoSQL databases and the rapidly increasing number of geospatial applications in recent times, the interest and necessity for research within the geospatial domain have increased [12]. Geospatial processing has found its usage in areas that include disaster management, crime statistics, civic planning[13]. Relational databases face performance limitation to house geospatial data due to its rigid schema and restricted scalability. On the other hand, NoSQL databases can store unstructured data efficiently and can scale up depending on the quantity of the data stored. The aim of this project is to compare the performance of R*-tree index system with Z-Curve index system for geospatial operations.

The base for an efficient geospatial processing system is to store and retrieve vast volumes of data rapidly and efficiently. Indexes play a critical role in efficient retrieval primarily when the size of the data set is enormous. The index in general is a data structure that contains a copy of index columns from the data set which helps in quick look up and retrieval of data from the entire data set. Indexes for geospatial processing should be capable of holding multidimensional data. The choice of the underlying data structure for the index depends on the use case. Some commonly used data structures for geospatial indexes are the R-trees, R*-trees, Quadtrees, Geohash, Z-curve. Each of these structures has its trade-offs and advantages facilitating its usage over a wide range of use cases within the geospatial domain[14].

Famous NoSQL databases such as MongoDB, Couchbase, DynamoDB have native support for geospatial querying and processing, and each of the data stores has its custom indexing mechanism. For example, MongoDB uses a 2d sphere as the indexing mechanism for geospatial data whereas DynamoDB uses Geohash as its two-dimensional index [15]. While Cassandra is one of the prominent NoSQL data store [16], it does not have any native support for geospatial processing. In this study, we choose Cassandra as the underlying database to conduct a performance study on the geospatial index mechanisms that enable and facilitate geospatial functionality of this highly scalable NoSQL system. Currently, Cassandra cannot process geospatial data on its own, an external library called GeoMesa adds this capability by building an index based on the Z-curve [11].

In this project, we developed two geospatial index systems for Cassandra, one using the R*-tree index and the other using GeoMesa which uses Z-curve index. R*-Trees are widely accepted as efficient index data structures especially when they come to multidimensional data such as geospatial data [17]. This project aims to make a performance comparison between the R*-tree index and the Z-curve index for geospatial operations in Cassandra. Representative geospatial operations such as insertion, deletion, overlaps, intersects, and k-nearest neighbors are chosen and their processing times are measured in both the systems for comparison purpose. We conducted experiments to study the impact of various performance factors and analyzed their results.

The organization of the report is as follows: Chapter 2 overviews fundamental concepts and the background techniques this project entails and also presents a research work that implements a strategy to use R-tree index in MongoDB. Chapter 3 describes Cassandra and GeoMesa. Chapter 4 presents the proposed work, the rationale behind the work's architecture and system design. Experiments and our analysis of the results are presented in Chapter 5. Finally, chapter 6 presents conclusions and future work.

CHAPTER 2

Background and Related work

2.1 Geospatial Data

A diverse set of sources generate geospatial data in today's world. Some sources include satellites, transportation systems, urban housing, and planning systems. All of these sources generate data at a high pace and in vast quantity. A closer look at these data shows that it is unstructured, meaning, it does not have a default data model. Multiple attributes or features are recorded as a part of the data collection process depending on the source. When data contains any location related information such as longitude-latitude, coordinates, a GPS recording, then the data can be termed as geospatial data. Usually, geospatial data are stored along with their metadata. Metadata provides additional information such as appropriateness, source of the geospatial data which helps in understanding the context of the data. In this project, we use a geospatial dataset for performing query operations.

Raster data and Vector data are two classifications within the geospatial data [13]. Raster data are cell-based. This concept is similar to the case where an image is split into pixels, and each pixel is assigned a value denoting the color of itself in the entire image. The value that a cell represents can be anything that helps to position the cell in the map such as the soil type, average rainfall, elevation. Some of the commonly used raster data formats include GeoTIFF files, .dem files, .png or .jpeg format along with its associated georeferencing files[18].

An example representation of raster data is given below as Fig.1. The image on the left side is a general picture of a land area segmented as cells. One area within the raster image is zoomed to show the individual cells. A single cell record attributes such as height and width of the cell.

Vector data, on the other hand, represent the earth as a set of features. Each feature has a geometric shape associated with it. Here the shapes could be points, lines, polygons or any combination of these three shapes. Some of the popular formats include shapefiles, GeoJSON, and Geometry Markup Language [18].

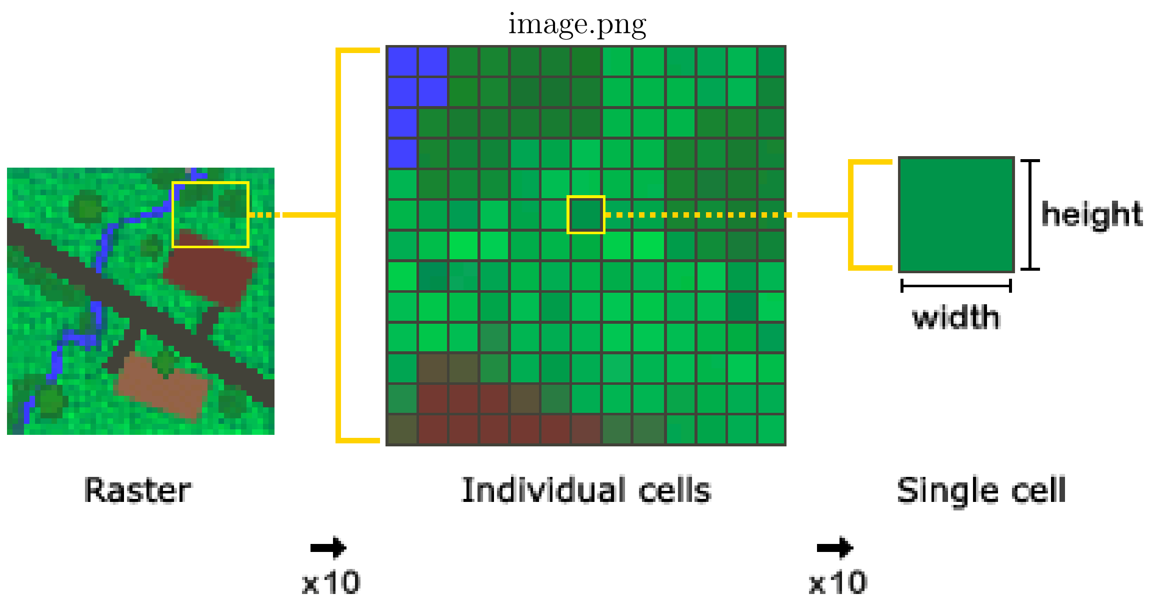


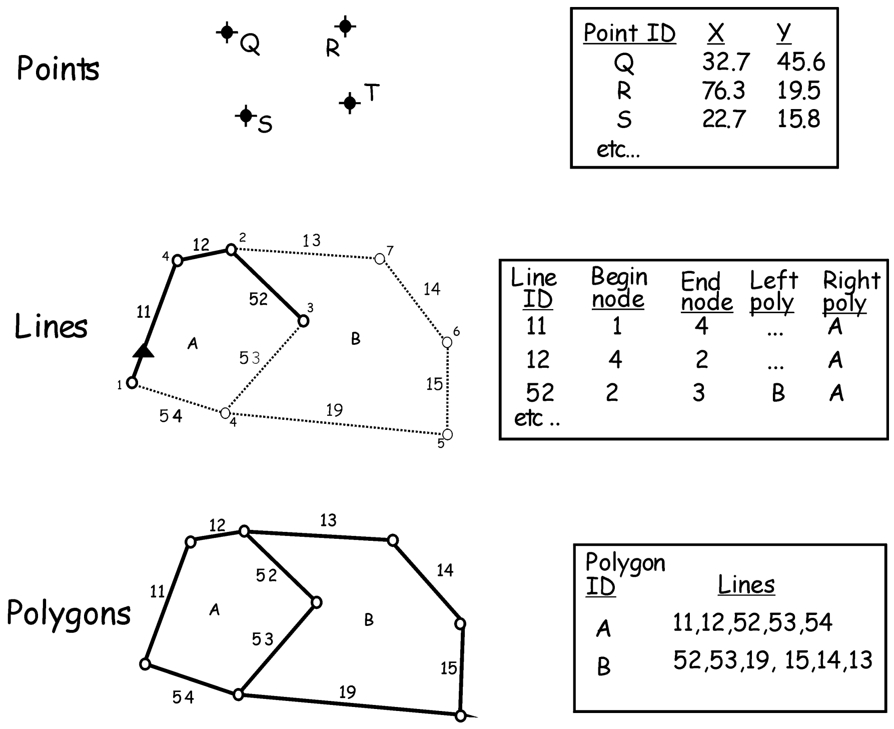
Figure 1: Example of Raster data - [1]

Figure 2 shows examples of geometry shapes. The table on the right side shows a sample representation of data and the left side figure shows the shape corresponding to the data.

a) Points: Point type represents an object as a set of x and y values. Usually, longitude and latitude are used to represent a point. Point data is commonly used to represent discrete data points and non-adjacent features. Point dataset cannot measure area or length as it does not have any dimensions. Point data can have other temporal and qualitative attributes associated with it.

b) Lines: A set of points connected forms a line. Lines can represent connections or motion. Similar to point data, line data can also have other attributes associated with it.

c) Polygons: When lines connect a set of vertices, and if the starting and the ending vertex is the same then it forms a polygon. A polygon can be used to represent complex figures, and it is usually used to represent boundaries. Similar to the other types, polygons can also be associated with other attributes.



Point ID	X	Y
Q	32.7	45.6
R	76.3	19.5
S	22.7	15.8
etc...		

Line ID	Begin node	End node	Left poly	Right poly
11	1	4	...	A
12	4	2	...	A
52	2	3	B	A
etc ..				

Polygon ID	Lines
A	11,12,52,53,54
B	52,53,19, 15,14,13

Figure 2: Example of Vector data - [2]

The difference between the raster and the vector representations for the same area within the map is given below as figure 3. In our project, we used geospatial data in vector format.

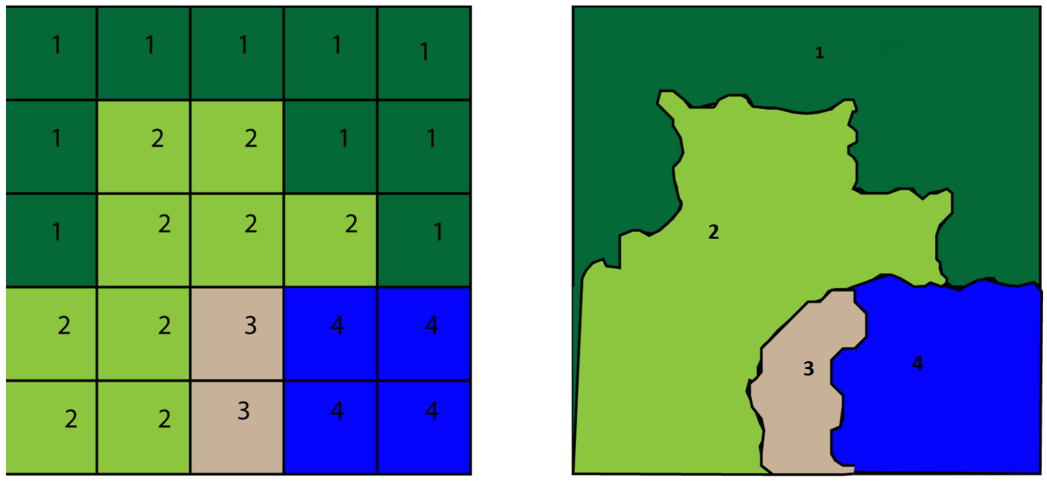


Figure 3: Raster data vs Vector data - [3]

2.2 Database Indexes

An Index in a database enables a query to retrieve data efficiently. An index is a set of key-value pairs with the keys being the columns from the table, and the values being pointers to the location of the data in the disk. These pointers help retrieve the data directly after a lookup operation over the keys. Index creation is a vital process as it determines the overall retrieval efficiency of the system. The decision about selecting the right column for indexing is taken based on the data access pattern and workload characteristics of the applications. As we design geospatial systems in this project to perform query operations, we study about the available geospatial indexes.

Most commonly used index data structure is the B-tree. B-tree has a key-value pair at each level of the tree. The leaf nodes contain pointers to the actual data while the internal nodes contain pointers to the child nodes[19].

While B-tree index facilitates searching value for a given index key, it does not efficiently support searching a value within multiple dimensions [5]. Typical geospatial queries are finding a range of points within a particular area, finding k number of locations near a given single point, and verifying if two objects are intersecting each other [5]. B-tree indexes do not efficiently support such queries involving multi-dimensional geospatial data.

Active research on designing an index data structure that can be used to accommodate multi-dimensional data led the way to the creation of R-trees [5]. R-tree is a tree data structure which can be used to index multi-dimensional data. Various enhancements were applied on top of R-tree leading to R+-tree[20], R*-tree[6], Hilbert R-tree[21]. Z-order curves[22], Quadtree [23], and Geohash are other data structures that can index two-dimensional data. The following section describes the R-tree, R*-tree, and Z-order curve index this project entails.

2.3 R-Trees and R*-Trees

As a variant of R-trees, R*-trees show most appealing performance and are often used in comparison studies for indexes [5]. We first describe the mechanism of R-trees and then specify R*-tree focusing on their significant improvement over R-trees.

2.3.1 R-Trees

R-trees are dynamic self-balancing trees based on B+-trees [5]. R-trees have internal nodes and leaf nodes. Nodes are placed within the R-tree based on the Minimum Bounding Rectangles (MBRs). An MBR is a bounding box or rectangle that contains all the geospatial points within a node. These MBRs are constructed in a way that there is little dead space apart from the space occupied by points within it, hence the name minimum bounding rectangles [5]. Each internal node of R-tree corresponds to an MBR that bounds the MBRs of all the lower nodes under it. Each leaf node has a pointer to the geospatial tuple in the database and an MBR.

One important property that impacts the performance is the number of entries that can fit in a single node of the tree. This parameter determines the tree depth and the frequency of the node splitting operation. Node splitting is a sequence of steps which is triggered when the number of entries within a node exceeds the allowed number of entries. Assuming M be the maximum number of geospatial entries that a node can contain and M/2 >= m be a factor denoting a minimum number of geospatial entries that has to be in a node, the properties are:

a) Each leaf node and non-leaf node should contain geospatial entries between m and M except for the root node.

b) For all the nodes, the minimum bounding rectangle should be the smallest possible rectangle containing all the tuples under it.

c) Root index node should have at least two children unless the child node is a leaf node.

d) All the leaf nodes should be at the same level [5].

Figure 4 presents an example of R-tree index structure. In this example, the root node has two entries T and U denoting the parent MBRs. Leaf node A has an MBR constructed in a way that all the points within leaf A is contained in the MBR. Also, there are two leaf nodes H and G present in both the boxes T and U. Node G is not completely within U's region and overlaps with T's region. Same is the case for node H. As we progress higher towards the root, constructing MBRs over the child MBRs, we include the child MBR into the parent MBR only when the parent MBR completely contains the child MBR. This rule is followed for constructing the MBRs of the internal nodes until we reach the root node.

Search operation starts from the root node and descends towards the leaf node, similar to that of the B-tree's process. A geospatial search query specifies a query rectangle denoting the target search area. The database system serves this query by returning all the points within this query rectangle, and the R-tree index facilitates the search process. At a given point in time, either a leaf node or a non-leaf node is consulted. For a non-leaf node, the query rectangle is compared with that of the bounding rectangle stored in the node. If any entry overlaps, the same process is recursively done with the children of the current node. For a leaf node, all the entries of the leaf node are compared with the query rectangle. If any entry overlaps, that entry is a qualifying record, and the entry is added into the result set of the query. This approach makes sure that irrelevant regions of the indexed space are eliminated and only entries that overlap with the search area are examined. A

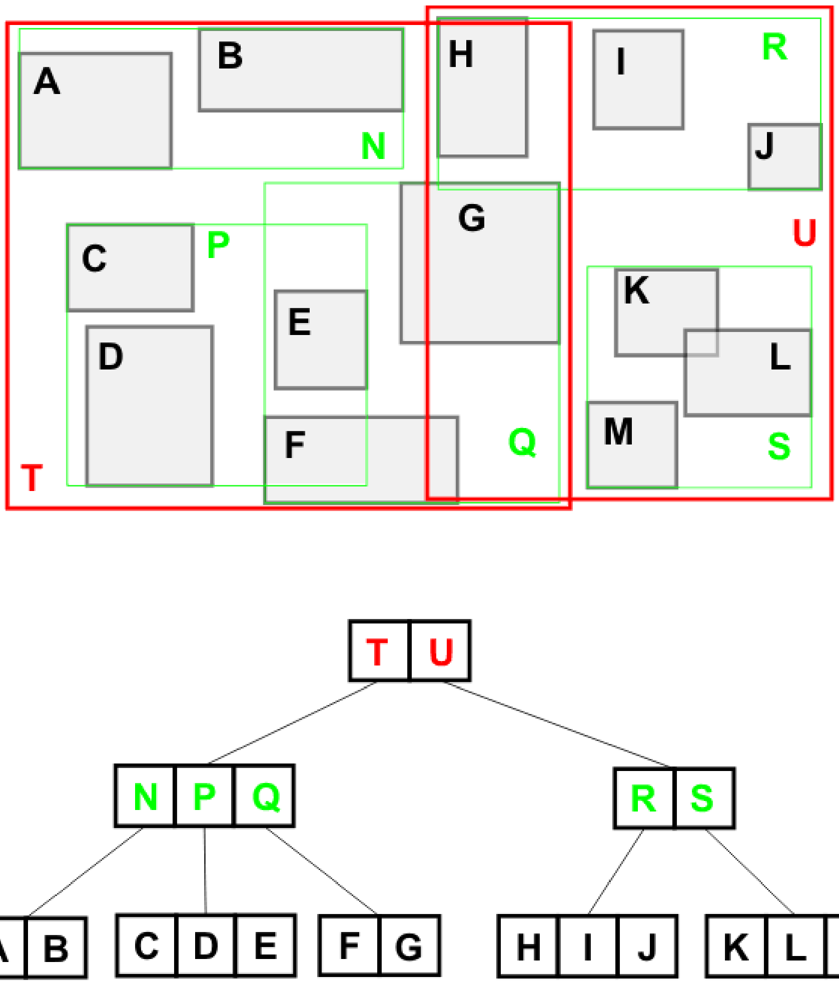


Figure 4: Example of an R-tree - [4]

Insertion in an R-tree is achieved through a series of steps. The insertion process is similar to that of a B-tree wherein new index records are added to the leaves. If the number of entries within a node exceeds the predefined value, the node is considered to be overflowing, and the split process starts. The overflowing node is split, and the change is propagated upwards until the tree is balanced.

When a geospatial entry is added to the index, the R-tree algorithm finds the leaf node that will house the new entry. The leaf node is determined based on the leaf node's bounding rectangle. Precisely, the leaf node's MBR that needs the least enlargement to accommodate the new entry is selected. If two nodes have the same least enlargement to accommodate the new entry, then the leaf node's rectangle having the smallest resultant area is selected to house the new entry. Having selected a node to add the new entry, the insertion algorithm checks if the selected node will overflow after adding the new entry and performs node splitting if the node will overflow. Splitting process constructs two nodes. Parent of the overflowing node before the split is taken and its bounding rectangle is adjusted in a way that it tightly encloses all the entries of the first new node. A new entry is created with a pointer to the second new node and a bounding rectangle that tightly encloses all the values in the second new node. Add this entry to the parent node if there is space, if not split this node as well. Repeat this process iteratively until the tree is adjusted and the iteration process reaches the root node.

Node splitting has to be done in a way that it becomes unlikely that two split nodes have to be examined to serve a subsequent search. An example of a good split and a bad split are presented in figure 5. The case with the bad split is that there is much dead space in the bounding rectangle that is formed whereas, in the case of a good split, the rectangle is as tightly bound as possible.

Representative splitting algorithms are proposed in [5]. The goal of these node splitting algorithms is to split a node containing M+1 entries into two nodes, where M is the maximum number of entries a node can have. The splitting algorithm proposed in [5] comes with quadratic running time. It selects two entries from M+1 entries in a way that the selected pair, when grouped into an MBR, would create much dead space in the MBR. These two points should be assigned to separate nodes, namely node 1 and node 2. For each of the remaining entries, calculate the increment in the area of the node 1's MBR and node 2's MBR. The splitting algorithm chooses the entry that has the highest difference and adds the entry to the node that requires the least enlargement in the MBR area. Continue this process for all the entries and assign them to the corresponding node.

To delete an index entry from an R-tree, the leaf node that contains the target entry is located first. To find the target node, the deletion algorithm starts with the root node and compares the MBR of the root node with the MBR of the entry to be

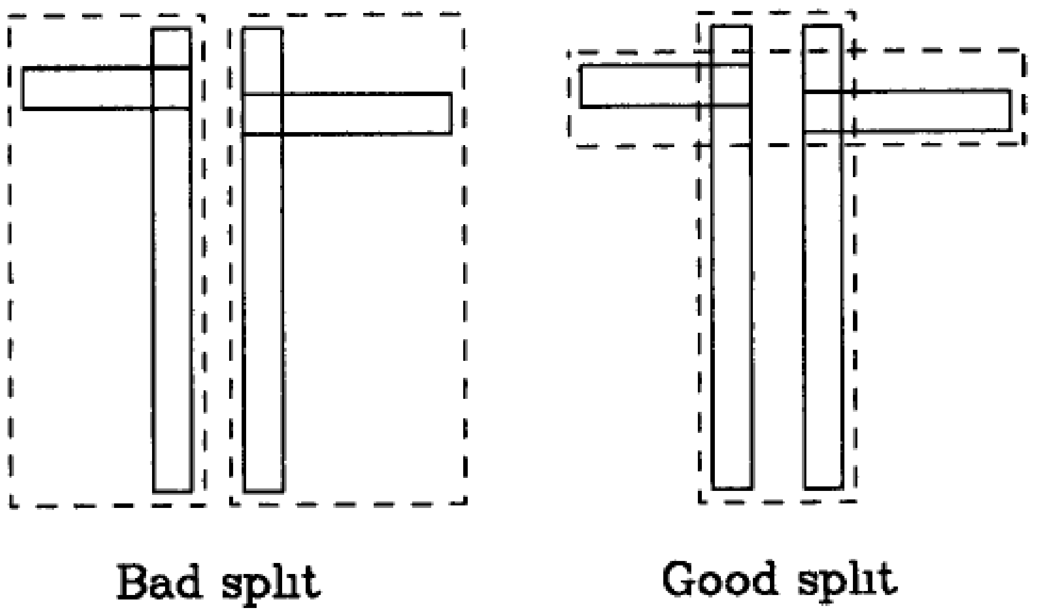


Figure 5: Node Split cases - [5]

deleted. If a leaf node is reached, each entry within the leaf node is checked to find a match. A matched entry is deleted from the leaf node. If the number of entries on the leaf node does not meet the required minimum after the deletion, the remaining entries of the leaf node are added into a set Q. If the leaf node has enough entries, then the possibility to reduce the MBR area is checked. Recursively move towards the root node from the leaf node adding the node entries that do not meet the minimum threshold. After reaching the root node, all the values in the set Q are reinserted into the tree similar to the insertion operation.

2.3.2 R*-Trees

R*-tree was proposed in [6] and have been studied as an efficient index structure for multi-dimensional data in many pieces of literature. [6] performs a heuristic based approach to identify the parameters that are essential for effective retrieval performance. The four parameters as concluded by [6] which are crucial for effective retrieval performance are - the area covered by the MBR should be minimal, the overlap between the MBR should be minimal, the margins (lengths of the edges of the MBR added together) of the MBR should be minimal, and the storage utilization should be optimal.

In a tree multi-dimensional index, its insertion algorithm has to be examined to identify the optimization approaches that have been used for effective retrieval. Insertion algorithm in R-tree calls two algorithms in which the decision for retrieval performance is made. One is the choose subtree procedure, and the other is the split algorithm. From [5], it is clear that choose subtree procedure optimizes the area of the MBRs. For the node splitting algorithm, [5] proposed linear and quadratic approaches. In [6], the linear and quadratic splitting algorithms were evaluated for their performance, and the quadratic splitting algorithm was found to be better performing than the linear algorithm in large. Hence the quadratic algorithm was examined for optimization parameters in [6]. From [5], the node splitting algorithm considers the area of the MBRs and degree of overlap between the MBRs. Out of the four optimization parameters identified by [6], R-tree considered only two parameters.

Because of the way these algorithms were designed, R-tree had few disadvantages. The first issue is because of the local reorganization within the MBR during a split process. Local reorganization does not support a restructuring of the MBRs. As R-tree is non-deterministic in choosing the nodes for an entry to be inserted, meaning tree structure differs if the sequence of entries inserted is changed, MBRs that were built using the initial set of entries may no longer suit the entries that are inserted later. This problem is worsened when a node becomes underfilled during a deletion operation after which the entries of the underfilled node are merged with the old parent. The second disadvantage is with the quadratic split algorithm. During the split process, when two entries are selected and evaluated based on the MBR area, the possibility of having an MBR with a small area but large distance is high. This process results in creating a needle-like MBR which is a bad split. Another problem with this split algorithm is that when splitting entries between two new nodes, the number of entries within one node may reach the limit. In this case, all the remaining entries are directly assigned to the second node without considering any geometric or area properties. This process results in a bad split. Figure 6 shows the issues with the splitting process in an R-tree. An overfilled node and possible results after the split are shown. First split results in uneven distribution of the entries between the nodes and the second split results in a higher overlap between the nodes for the same entries as the minimum number of entries in a node is set to 30

R*-tree deduced a revised node splitting algorithm and forced reinsertion technique during node overflow. These algorithms were designed to minimize node overlap and coverage. When a node overflows, a part of its entries is removed and inserted into the

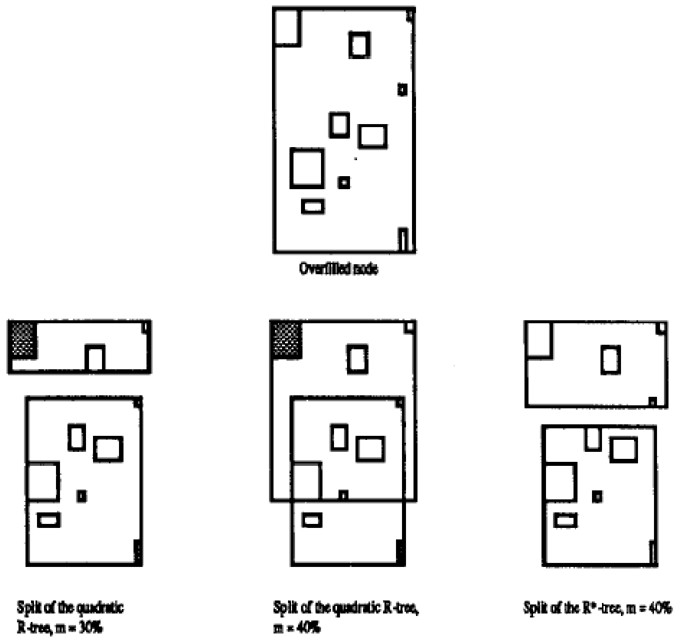

Overfilled node
Split of the quadratic R-tree, m = 30%
Split of the quadratic R-tree, m = 40%
Split of the R*-tree, m = 40%

Figure 6: Comparison - Node Split cases - [6]

tree again. This reinsertion helps in producing a tree that is well-clustered, reducing the dead space and builds a better tree incrementally. Also, R*-tree postpones the node splitting until necessary increasing the average node occupancy. Better node occupancy at every level improves the storage utilization which is the fourth parameter to improve the retrieval efficiency [6].

For insertion, [6] tested multiple combinations of area, margin and overlap parameters to determine the optimal combination. The steps in the insertion process are as follows. Choose subtree algorithm identifies the node to add the new entry. Starting from the root node, the R-tree is traversed until the right leaf node is identified comparing the MBRs of the node at each level with the new entry's MBR. If the node compared has its child to be a leaf node, choose the entry in the current node that needs the least overlap enlargement to include the new entry's MBR. If not, choose the entry in the current node that needs the least area enlargement (similar to R-tree) to include the new entry's MBR. The overlap condition is given in figure 7. Here E_1 , , E_p are the list of entries in the current node [6].

$$\text{overlap}(E_k) = \sum_{i=1, i \neq k}^{p} \text{area}(E_k.\text{Rectangle} \cap E_i.\text{Rectangle}) \quad , 1 \leq k \leq p$$

Figure 7: Overlap condition - [6]

The revised split algorithm of R*-tree first sort all the entries within the overflowing node based on the lower value of the bounding rectangle and then sort again by its upper value. For each sort, (M-2m+2) distributions of the M+1 entries are split into two groups. For the k-th distribution (k takes values from 1 to (M-2m+2)), the first node contains the first (m-1) + k entries and the second node contains the remaining entries. Here m and M represents the minimum and the maximum number of entries that should be present in a node respectively. For each distribution, the goodness value is computed. [6] devises a method to determine the goodness value. Now based on the selected goodness value, an axis is chosen as the split axis. Now that split axis is determined, choose the entries that fall within each group using the minimum overlap value.

Given below is figure 8 showing an example case on how split occurs in R*-tree. The resultant nodes after the split have minimum coverage and minimum overlap. The resultant shape of the MBRs is more quadratic (like a square) and uniform. This shape promotes better and compact enclosing MBRs over the split nodes.

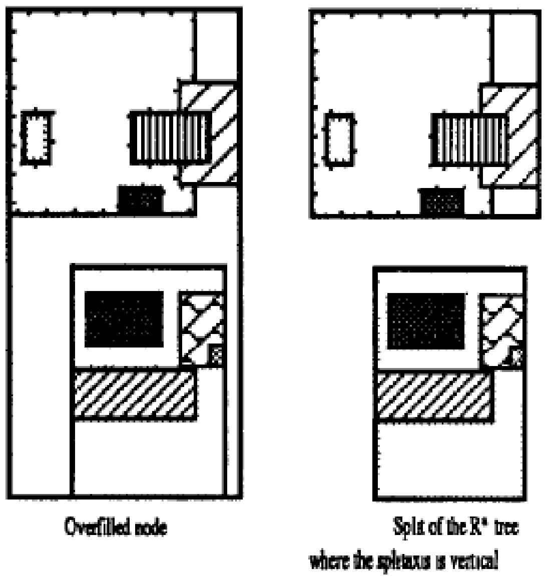


Figure 8: R*-Tree Split case - [6]

2.4 Z-Curves

Z-curves are space-filling curves that map multidimensional data into a single dimensional output. Z-curves work based on the Z values generated from the given data in any dimension. Usually, Z-curves are used for indexing purpose as it presents multidimensional data in a linear way. Searching in a linear set of data is always easier and hence its usage in indexing huge data sets.

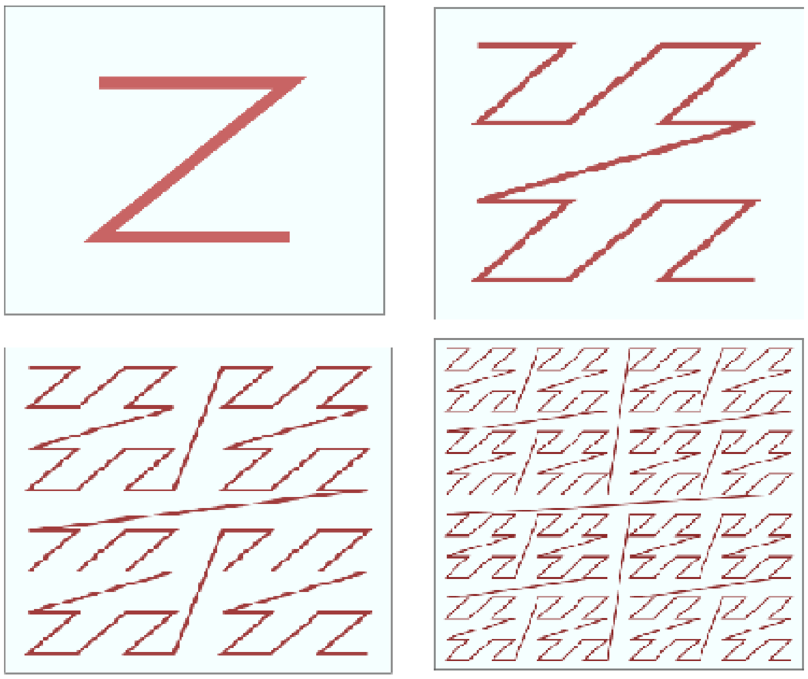


Figure 9: Z-Curve - [7]

Figure 9 shows the generated Z-curves for 1 to 4 iterations. A curve like structure is formed by connecting the Z values in the order of their precedence. Z values for the input are generated based on a process called Binary interleaving. In this process, the coordinate values within the input data are converted to their binary form, and these binary values are interleaved to form the corresponding Z value. Interleaving is a process wherein the binary representation of the n-dimensional input are combined into a single value with the binary bits at the alternating positions. The resultant single value is called the Z value of the n-dimensional input. This process is performed for all the values of the input. Now the entries are all sorted based on the Z values. The property of this Z ordering is that it preserves the natural ordering as in an n-dimensional plane.

Binary interleaving is explained with an example below. Consider a random coordinate point (X, Y) as (79, 412). The first step here is to convert the X and Y values into their binary representation. After conversion, 79 becomes 001001111 and 214 becomes 110011100. Now the resultant Z value is obtained by combining both of these values with each bit from the corresponding byte at alternating positions. So Z becomes 010110000111111010 which when converted into decimal form yields 90618. This resultant number is the Z value for the point (79, 412).

The suitability of Z-curves for representing the geospatial data is given as figure 10. The z value is also called as Morton code after the name of the person who formalized this concept [22].

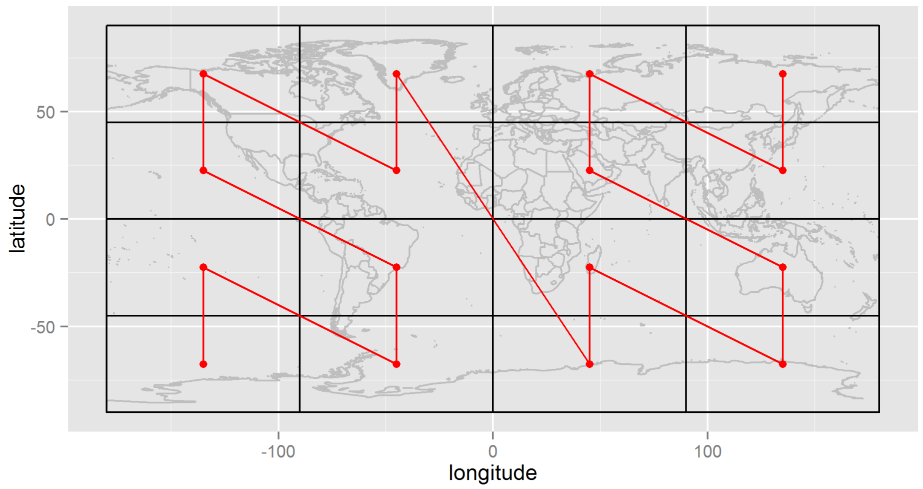
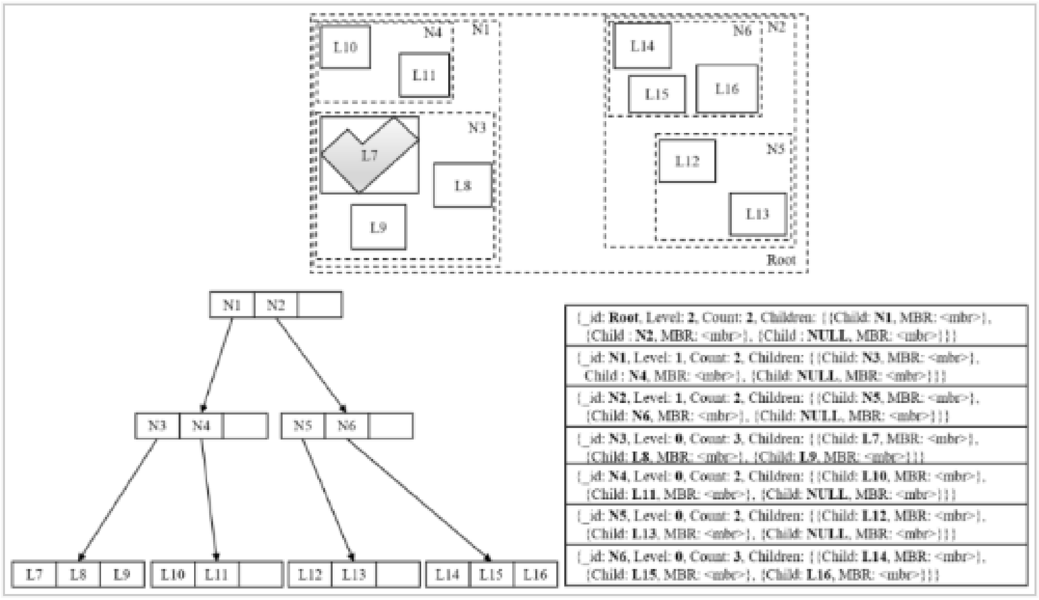


Figure 10: Z-Curve representation on a World Map - [8]

2.5 Indexing spatial data using Flattened R-Tree

To design a new indexing component that can function along with a NoSQL database and to plugin the indexing scheme into the functioning of the NoSQL database, we study an R-tree based flattening method that is integrated into MongoDB. According to [9], in MongoDB, all of the geospatial processing is performed using 2dsphere index which supports only input data in the format of GeoJSON. [9] suggests an idea of using R-trees for indexing geospatial data in MongoDB. However, MongoDB is a document-oriented data, and all the data are stored as documents. According to [5], R-trees are tree structures which represent hierarchy and node traversing concepts. Though R-tree is a widely accepted and used geospatial indexing scheme, because of the nature of the MongoDB, it cannot be directly used in MongoDB. So, [9] suggests a method wherein the resultant R-tree obtained after indexing the geospatial data is flattened into a MongoDB document.



```
{_id: Root, Level: 2, Count: 2, Children: {{Child: N1, MBR: <mbr>},
  {Child : N2, MBR: <mbr>}, {Child : NULL, MBR: <mbr>}}}
{_id: N1, Level: 1, Count: 2, Children: {{Child: N3, MBR: <mbr>},
  Child : N4, MBR: <mbr>}, {Child: NULL, MBR: <mbr>}}}
{_id: N2, Level: 1, Count: 2, Children: {{Child: N5, MBR: <mbr>},
  {Child: N6, MBR: <mbr>}, {Child: NULL, MBR: <mbr>}}}
{_id: N3, Level: 0, Count: 3, Children: {{Child: L7, MBR: <mbr>},
  {Child: L8, MBR: <mbr>}, {Child: L9, MBR: <mbr>}}}
{_id: N4, Level: 0, Count: 2, Children: {{Child: L10, MBR: <mbr>},
  {Child: L11, MBR: <mbr>}, {Child: NULL, MBR: <mbr>}}}
{_id: N5, Level: 0, Count: 2, Children: {{Child: L12, MBR: <mbr>},
  {Child: L13, MBR: <mbr>}, {Child: NULL, MBR: <mbr>}}}
{_id: N6, Level: 0, Count: 3, Children: {{Child: L14, MBR: <mbr>},
  {Child: L15, MBR: <mbr>}, {Child: L16, MBR: <mbr>}}}
```

Figure 11: Flattening an R-Tree into a Mongo collection - [9]

An example transformation is given above in Fig.11.The traversal in R-tree is based on pointers to the child node which is now transformed to identifier based document traversal when flattened into MongoDB. When the Fig.11 is observed closely, each document has 4 key-value pairs. They are node identifier, node level counting, number of index entries that are put to use, index entry information. Other than this, each index entry has two fixed key-value pairs. The value "child" is used for document referencing, and MBR is used to store the MBR values. If the node is a leaf, then the corresponding "child" node points to the actual spatial object. Four different collections are maintained to store the R-tree related information. They are R-tree collection (RC), Spatial collection (SC), R-tree metadata collection (RMC) and Spatial metadata collection (SMC). Based on the real-time situation, the two metadata collections can have only one, and the other two can have any number of instances. Also for each R-tree collection, there is a Spatial collection. Coming to the details of these collections, as the naming implies, SC contains key-value pairs of GeoJSON data. RC has a structure that was explained above with the same name as that of the SC that it indexed, suffixed with "-RTree".

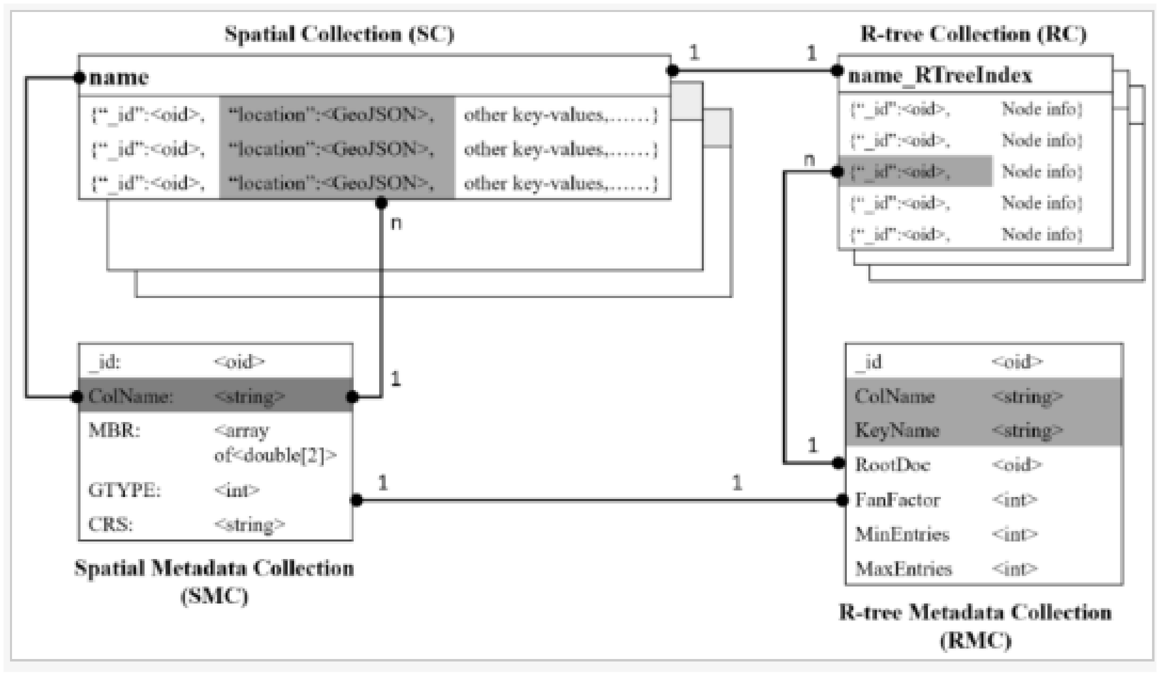

Spatial Collection (SC)
name
{"_id":<oid>, "location":<GeoJSON>, other key-values,......}
{"_id":<oid>, "location":<GeoJSON>, other key-values,......}
{"_id":<oid>, "location":<GeoJSON>, other key-values,......}
R-tree Collection (RC)
name_RTreeIndex
{"_id":<oid>, Node info}
1
1
n
n
1
1
1
1
_id: <oid>
ColName: <string>
MBR: <array of<double[2]>
GTYPE: <int>
CRS: <string>
Spatial Metadata Collection (SMC)
_id <oid>
ColName <string>
KeyName <string>
RootDoc <oid>
FanFactor <int>
MinEntries <int>
MaxEntries <int>
R-tree Metadata Collection (RMC)

Figure 12: The flattened R-Tree schema - [9]

SMC contains the metadata of the planar spatial data and a corresponding RMC denoting SMC is indexed. All of the above definitions are visualized as in Fig. 12. Now that the schema is ready, this implementation is then plugged into the MongoDB architecture. A new module based out of R-tree is built and is added into the route server. The existing message dispatching module is now altered. Whenever the user fires a geospatial query, the message dispatching module identifies the type of query and invokes the R-tree module. For example, let us assume that the user fires a search overlap command on a specific geospatial field. Now the message dispatcher after receiving this query consults with the config server to determine if there is a flattened R-tree that is already available for the target collection. If found, then the R-tree module is called up, and it executes the necessary query that is mapped. For the query execution, the RC and the SC are now queried using the native mongo commands. The results of these queries are sent to the R-tree module. Now if necessary, the results are refined and sent back to the user. This execution flow is performed for all types of queries. Depending on the type of query, the necessary module is invoked, and query execution is p, and the results are sent to the user.

CHAPTER 3

Cassandra and GeoMesa

3.1 Cassandra

Cassandra is an open source, Apache licensed, distributed wide column NoSQL database system. It is optimized to run on a cluster providing high scalability [24]. Cassandra achieves high availability and partition tolerance (denoting no single point of failure) while loosening the consistency factor in terms of the CAP theorem [25]. Cassandra follows peer to peer distribution model to replicate data across multiple nodes in a cluster. With the peer-to-peer model, all nodes in a cluster take the same role without any designated primary node.

A Cassandra cluster can be viewed as a ring because it partitions data across a cluster through the consistent hashing strategy [25]. Figure 13 depicts a ring layout of the Cassandra cluster. Cassandra provides tunable consistency, allowing different consistency level to be specified for a read and write operation in balance with availability. A consistency level can be defined per operation, keyspace or cluster. Cassandra supports replication to achieve high availability. A replication factor specifies how many nodes house a copy of the same partition and a consistency level is defined in terms of the replication factor. In a Cassandra cluster, all nodes are identical in function and are aware of the way data are distributed and replicated. In this way, adding and removing nodes can be done without much downtime, and a failed node can be recovered quickly by getting data from other nodes in the cluster.

Cassandra houses data in a table of which rows can be skinny rows or wide rows. The primary key defined in the table schema determines the type of rows in the table. The first component of the primary key serves as a partition key which is used to partition tables in a cluster. Skinny rows are similar to rows in relational databases except for that they do not save null for non-existing values. Wide rows are identified by their partition keys and stores value ordered by the clustering key, which is the second component of the primary key.

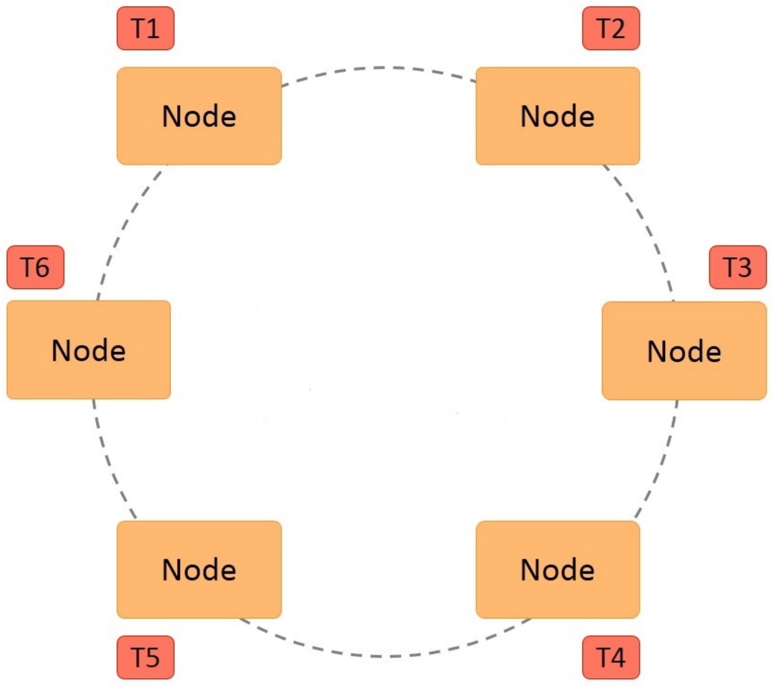


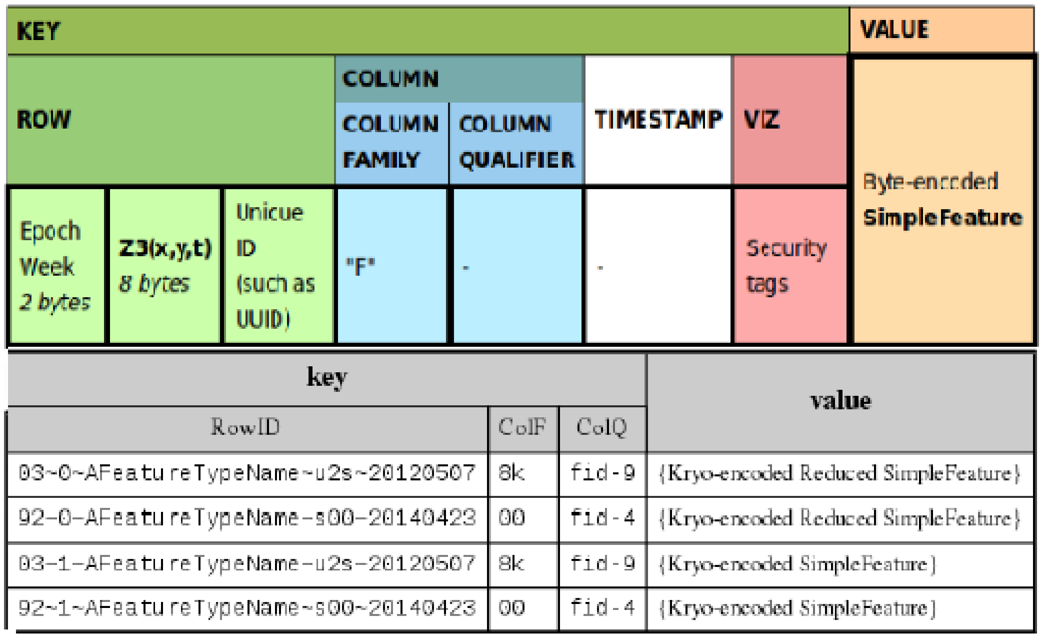
Figure 13: Architecture of Cassandra - [10]

CQL is a query language for Cassandra which provides a Data Definition Language (DDL) and a Data Modification Language(DML) similar to SQL. Currently, CQL does not support any geospatial queries and Cassandra serves geospatial queries by interacting with GeoMesa.

3.2 GeoMesa for Cassandra

GeoMesa is an open source library or tool capable of performing geospatial querying in a large scale distributed system. It is used along with Cassandra for geospatial processing. GeoMesa provides spatio-temporal indexing of geodata which enables high-speed geospatial querying on point data. GeoMesa is a part of the LocationTech group of the eclipse foundation and is licensed under Apache License 2.0. GeoMesa library can be incorporated on top of any key-value database. In this study, we use the GeoMesa datastore library available for Cassandra. GeoMesa uses Z values as the core indexing mechanism for the geospatial input data. When a given area in a map is formalized as a group of cells, Z-curve visits each cell exactly once establishing a unique ordering between the cells. Ordering here is based on the Z value that is generated for the given set of geospatial points.

As Cassandra is a key-value store, geoMesa formulates an indexing scheme wherein the resultant Z value is kept as the key, and the data entries form the value for that key. The index types within GeoMesa are Z2, Z3, XZ2, XZ3. Number 2 denotes two-dimensional data, i.e. longitude and latitude and Number 3 denote three-dimensional data which has a temporal attribute included. XZ is a type of space-filling curve and is an extension of the Z-curve.



KEY							VALUE
ROW			COLUMN		TIMESTAMP	VIZ	
			COLUMN FAMILY	COLUMN QUALIFIER			
Epoch Week *2 bytes*	**Z3(x,y,t)** *8 bytes*	Unicue ID (such as UUID)	"F"	-	-	Security tags	Byte-encoded **SimpleFeature**

key			value
RowID	ColF	ColQ	
03~0~AFeatureTypeName~u2s~20120507	8k	fid-9	{Kryo-encoded Reduced SimpleFeature}
92-0-AFeatureTypeName-s00-20140423	00	fid-4	{Kryo-encoded Reduced SimpleFeature}
03-1-AFeatureTypeName-u2s-20120507	8k	fid-9	{Kryo-encoded SimpleFeature}
92~1~AFeatureTypeName~s00~20140423	00	fid-4	{Kryo-encoded SimpleFeature}

Figure 14: Indexing in GeoMesa - - [11]

In general, indexes are classified as data-specific and space-specific. A detailed overview of the types of indexes are given in [26]. A data specific index constructs indexes whose sub-divisions are determined by the specific records that they accept. The category that the corresponding value gets into cannot be identified without knowing where the previous values were indexed. On the other hand, space-specific indexes always index a value only based on the space that is being indexed. No information regarding the previous records is necessary to compute the index of current value. GeoMesa is based on Z-curve which is in turn based on the space-specific index. Figure 14 shows the indexing scheme followed by GeoMesa. The first table gives the schema whereas the table below gives a list of sample index values. Presence of 0 or 1 after the first two characters in the RowID denote whether the corresponding entry is an index or data entry.

GeoMesa uses Contextual Query Language (CQL) which is a library under the geotools package. GeoMesa's query planner takes in a CQL query and converts it into corresponding Cassandra Query Language. The Contextual Query Language supports filters which are equivalent to the where clauses in Cassandra Query Language. Filters within a query can be classified into primary and secondary CQL filter. The primary CQL filter is used to determine the scan range, and secondary CQL is used to identify matching rows. GeoMesa has few indices built in the initialization phase. So the query planner selects the index type that scans the least number of rows. For this decision, GeoMesa uses two methods : cost-based and heuristic-based method. Each method has a process of identifying the best index using a series of steps [11]. After identifying the type of index to consult, query planner executes the query and retrieves the results from the Cassandra.

CHAPTER 4

R*-Tree index and Z-Curve index systems

To perform a comparison between the R*-tree and Z-curve, we designed and implemented two index systems, R*-tree index and Z-curve index systems, for geospatial operations in Cassandra. For each index system, we developed an application consisting of a user interface and a back-end component. The back-end component comes with a driver that establishes the connection between the application and the rest of the system and also a query planner that enables geospatial operations in Cassandra and facilitates them using the given index type.

The R*-tree index system follows the NodeJS framework and uses JavaScript as the implementation language. The Z-curve index system is written in Java based out of Spring MVC (Model View Controller) framework and uses GeoMesa for indexing the geospatial data. This chapter describes the design principles and implementation of these two systems. These two systems are used to compare the performance of R*-tree and Z-curve index for the representative geospatial operations.

4.1 R*-Tree index system

The architecture of the R*-tree index system is given below as figure 15. For this system, we developed a NodeJS application consisting of a user interface and a query planner. User can submit the geospatial operations through the user interface. This page has provision to enter necessary input and perform desired geospatial operations. The back-end component of the application processes all the requests from the user interface. This component also has a query planner module integrated into it. For a given geospatial operation, the query planner interacts with the R*-tree index. Post this interaction, query planner generates the CQL query, and Cassandra executes this query.

While deploying the application, the back-end component establishes a connection with Cassandra using the NodeJS driver. The driver attempts to connect to the port that Cassandra is running during initialization. Read and write paths are the two significant workflows within the application. All geospatial operations fall within either of the workflows.

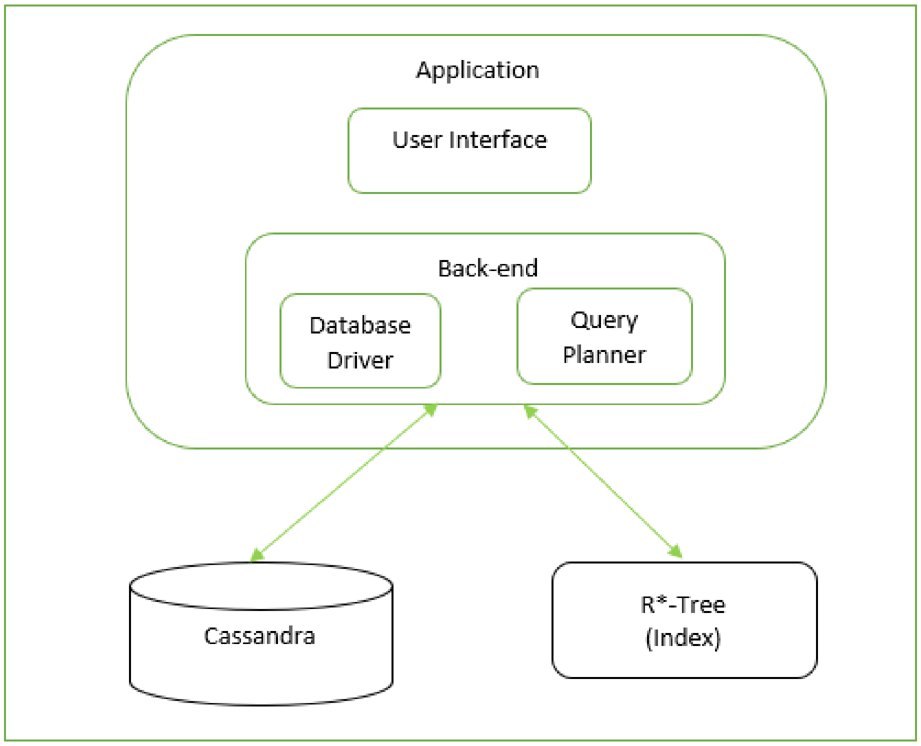


Figure 15: R*-Tree index system

4.1.1 Write path

This subsection provides an in-depth description of the sequence of steps performed during the write path. The execution of insert and delete operations follow the write path. Our system supports both single point data insertion and bulk insertion using a CSV file. The figure 16 depicts the write path of the R*-tree index system. When a user provides input for insertion (step-1), the query planner first extracts the longitude, and latitude from the user defined values and inserts it into the in-memory R*-tree index (step-2). The query planner then generates CQL prepare statement to perform the insertion operation in the Cassandra and invokes the database driver (step-3). Database driver performs insertion into the Cassandra (step-4). Post insertion in Cassandra, the user is notified about the successful insertion.

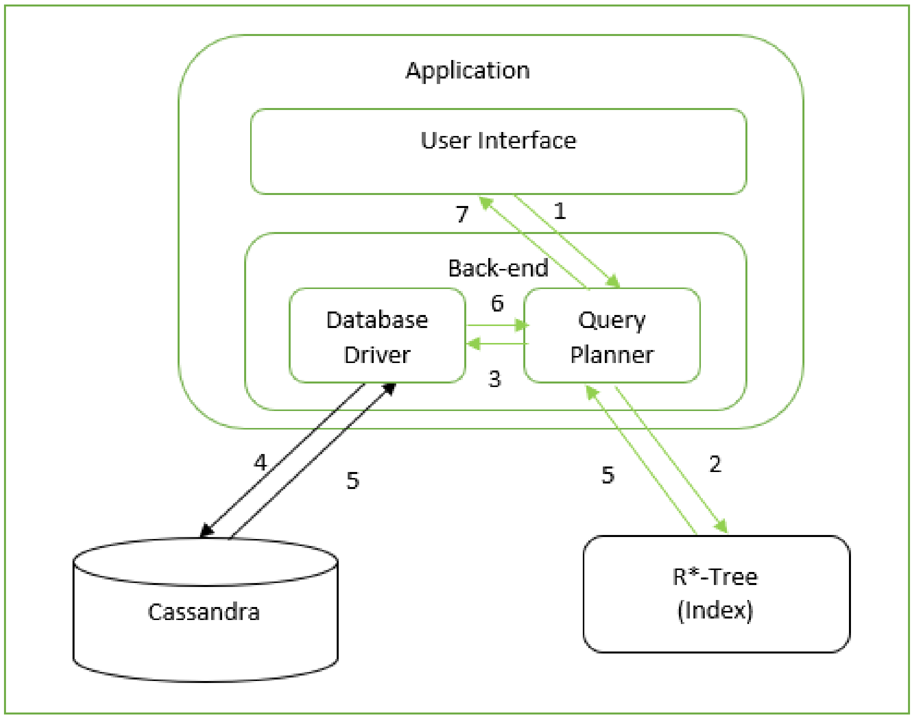


Figure 16: Write path - R*-Tree index system

4.1.2 Read path

Figure 17 presents the sequence of steps followed in a read path. When a user provides input for the search query (step-1), the query planner first consults the R*-tree index for lookup (step-2). Input for the read path operations is usually the coordinates of the bounding box. For a given operation, the R*-tree index returns a list of results containing the longitude and latitude pairs (step-3). The query planner takes these results and constructs a corresponding CQL filter query to retrieve all the necessary columns from Cassandra (step-4). The query planner invokes database driver and submits the CQL query. Database driver executes this query in Cassandra (step-5) and Cassandra returns the result set to the driver (step-6) which is then passed to the query planner module (step-7). Finally, the results are passed to the user interface (step-8).

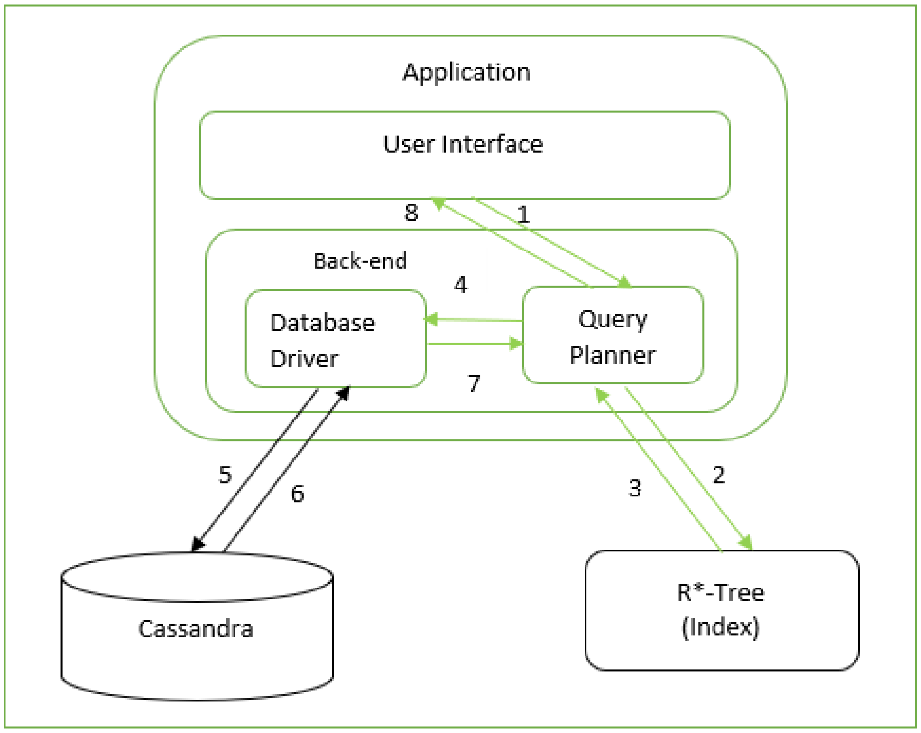


Figure 17: Read path - R*-Tree index system

4.2 Z-Curve index system

We developed an application consisting of an user interface, and a query planner for the Z-curve index system.The application uses the GeoMesa library as the abstraction layer on top of Cassandra. As discussed in the previous section, GeoMesa does not use a multi-dimensional index to handle and process geospatial data but converts the multi-dimensional geospatial data into a single dimensional value based on the Z-curve instead. The configuration of Cassandra is set to be precisely the same as the one set for the R*-tree index system.

To enclose the GeoMesa library into an interaction mechanism, we create a Spring MVC application and design a user interface with geospatial operations. When a user enters input to perform a spatial process, the query planner module is invoked. Query planner, depending on the type of geospatial operation, generates a query based on Contextual Query Language (CQL) with the user input and passes it to the GeoMesa module which interacts with the Cassandra and retrieves the results. The figure 18 below depicts the architecture of Z-curve index system.

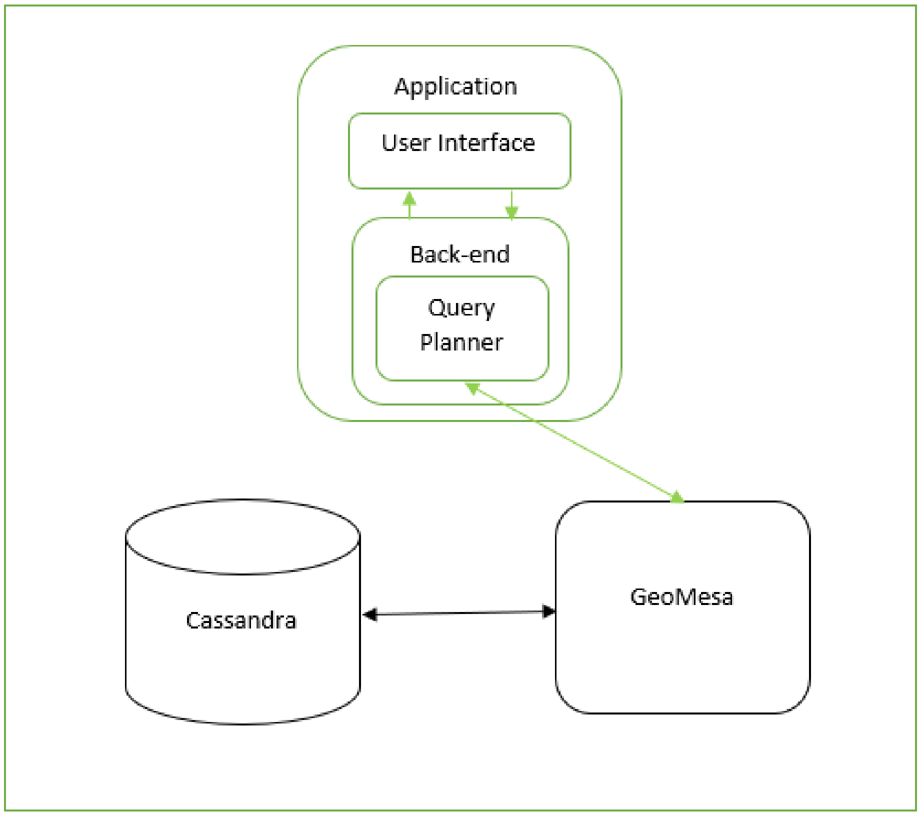


Figure 18: Z-Curve index system

4.2.1 Write and read path

Write and read paths for our GeoMesa application follow the same sequence of steps. For a write path, when a user enters an input in the interface, the query planner receives the request (step-1). It extracts the values and creates a list of SimpleFeature objects. SimpleFeature is a package present in the OpenGIS library used to represent the geospatial data [27]. Query planner module constructs the Contextual Query Language insertion query with the SimpleFeature object list. Query planner also groups the geospatial attributes into a geometry feature and marks the feature with a "*" field to denote the default geometry. Now, the query planner submits the CQL query to GeoMesa (step-2). GeoMesa inserts the input data into Cassandra (step-3) and builds multiple indexes on the data for efficient retrieval. One among them is a Z2 index which GeoMesa constructs on the generated Z-values for the geometry feature. In our implementation, Z2 is the primary index for all the geospatial operations. Cassandra acknowledges the insertion (step-4). This acknowledgement is passed to the query planner (step-5) and then to the user (step-6). Figure 19 below shows the sequence of steps in a write and read path.

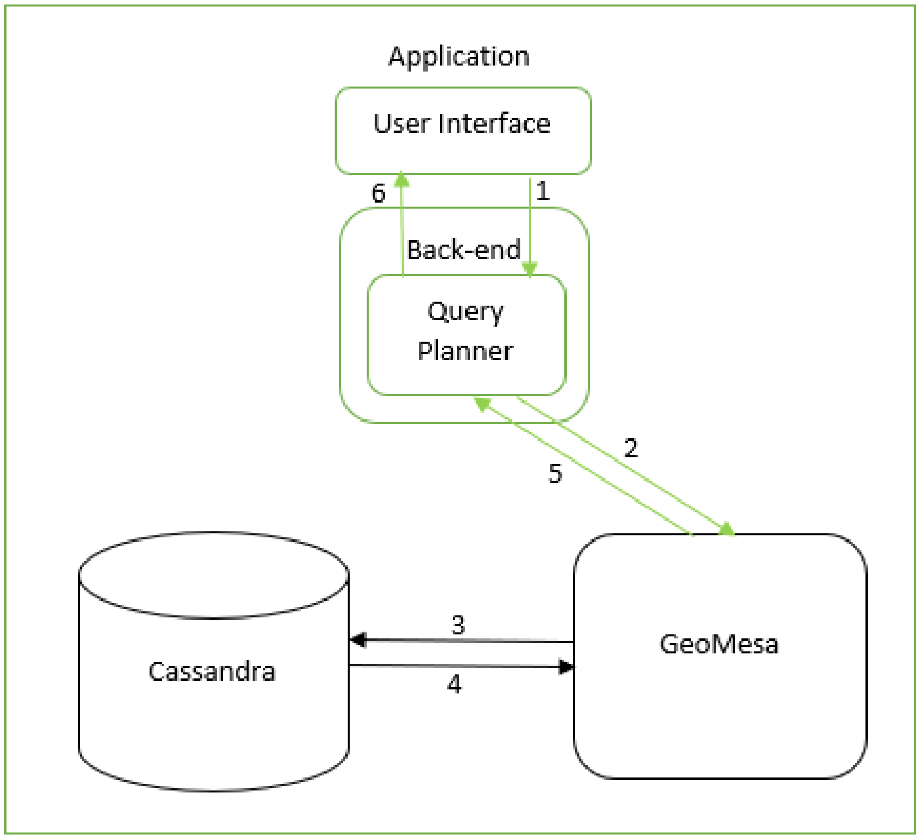


Figure 19: Write and Read path - Z-Curve based GeoMesa application

During a read operation, after the user enters the input, the query planner receives the request (step-1). It then creates a Contextual Query Language query for the corresponding geospatial operation along with a filter component containing the coordinates of the query bounding box and submits it to GeoMesa (step-2). GeoMesa identifies the type of operation and consults the corresponding index for efficient lookup (step-3). For example, in case of a geospatial search operation, GeoMesa computes the Z-values for the user entered bounding box coordinates and performs a search based on Z-values using the Z2 index. Finally, Cassandra returns the results for the query (step-4). The result set is then passed to the query planner (step-5) and then displayed in the user interface (step-6).

CHAPTER 5

Experiments and Results

This section describes the experiments for performance evaluation and presents our analysis on the results. The representative geospatial operations (overlap, intersect, K Nearest Neighbor, insertion and deletion) are chosen for the experiments. First, the dataset that was used for these experiments is listed, and the features about the same are also presented.

5.1 Dataset

A railroad bridge dataset is chosen for the experiments. This dataset is available as an open source under the ArcGIS Hub [28], collected and published by U.S. Department of Homeland Security containing around 86,894 rows. The size of the dataset is around 60MB. This dataset lists all the railroad bridges in the United States and information associated with them such as the exact location, Id, width and height of the bridges. Exemplary use cases involving the dataset are locating the bridges nearby given a location in the event of a threat, identifying a list of bridges that would be impacted given a natural calamity, and swiftly identifying alternating routes given a blockage. In this dataset, geospatial information is represented as X and Y coordinates denoting longitude and latitude, ZIP code and the address. In this project, longitude and latitude values are used for indexing.

5.2 Experiment setup

All the experiments were performed on a PC with 1.8 GHz Intel Core i5 processor and 12GB RAM running Windows 10. Two systems were designed to perform the experiments and compare the results. In this chapter, the application running R*-tree index with Cassandra is abbreviated as A1 and the application running Z-curve index as A2. The application A1 is implemented in JavaScript following the NodeJS architecture. The implementation of the R*-tree is adopted from a JS Library [29]. The application A2 is implemented in Java 8 following the Spring MVC framework.

Two geospatial search (read) operations (overlap and intersect) and two write operations (insert and delete) are chosen for the experiments. Processing time is measured for a given operation on each index system. We also studied the impact of R*-tree on the performance of the KNN geospatial query which is not currently supported natively by GeoMesa.

Processing time consists of the time taken for index operations and the execution time of the given database operation. Specifically, the processing time is the interval that starts from the time the index is consulted for a search query or index construction for insertion query up until the time to return the result from the database in the case of a search query and persisting of data in the database in the case of the insertion operation. Processing time is measured 10 times for a given operation and the average processing time is presented.

5.2.1 Write (insert and delete) operations

In our experiments, insert and delete operations are considered to find the impact of the index on the performance of write operations.

- Insert: Through the user interface of each index system we developed, data can be inserted as a single row or from a file containing data denoting bulk insertion of data. Time taken to perform different use cases within insertion is explained below in the results section. Overhead while inserting new data into an already existing index structure is also experimented. Here, the input would be the data to be inserted, and during the process, data is inserted into the index and the database.
- Delete: Within deletion, deleting a single data entry from the database and deleting all the data from the database are considered. Time taken to perform these operations is also recorded provided varying the amount of data that is stored in the database. Depending on the type of operation, the input for this operation will either be a single point in case of single entry deletion or nothing in case of deleting all the contents from the database.

5.2.2 Read (search) operations

We perform the following three geospatial search operations in our experiments.

- Overlap: The overlap query is to find the data entries that fall within a given query bounding box and returns, an array of entries that satisfy this criterion. The coordinates of the query bounding box is an input to the query.
- Intersects: Given the coordinates of the query bounding box, this query returns a Boolean true or false denoting if any entries within the database intersect with the bounding box. Though the underlying principle for processing both intersects and overlaps are same, the subtle difference here is that the intersects query completes the processing as soon as it finds a single data entry within the bounding rectangle whereas overlaps have to find out all the points within the

box.

- KNN: Returning K nearest neighbors is the third type of search operation. Given a point and the number of neighbors to be returned, the output is the list of neighbors requested. For finding the k nearest neighbors, a depth-first traversal implementation using a priority queue is used. Also, the MINDIST property of a given point is used to fill the priority queue. The process is terminated after there are k elements in the priority queue. In our experiments, we are considering the impact of underlying data stored.

5.3 Results and Analysis

This section presents the experimental results and our analysis.

5.3.1 Insert operation

Figure 20 depicts the comparison between the insertion times of R*-tree and the Z-curve. As the number of entries inserted increases, the R*-tree system takes higher processing time compared to the Z-curve based system. This behavior is expected. With a larger volume of inserted data, R*-tree has to perform node splitting and forced-reinsertion more frequently to handle node overflowing while the Z-curve index inherently does not have the notion of node overflowing.

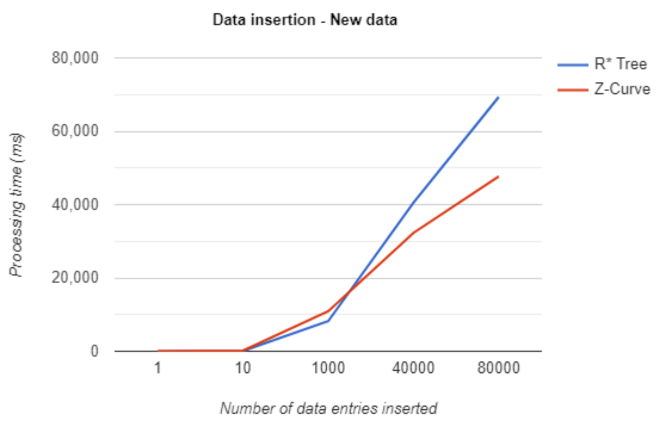


Figure 20: Data insertion - New data inserted into an empty database

5.3.2 Overlap operation

The query bounding box specified in an overlap query determines the number of data returned from the query. In this experiment, the processing time of an overlap operation is presented by the number of data the operation returns. Considering that the index size grows as the data size grows and thus searching can take longer time in a large sized database, we conducted experiments with two different database sizes, one with 40,000 and the other with 80,000 records.

Figure 21 presents the experimental results with the database size of 40,000 entries. The results show that the R*-tree index system outperforms the Z-curve index system as the number of results that a query has to find increases. R*-trees are primarily designed for faster retrieval from multi-dimensional data because the R*-tree node construction groups points that are close to each other into a bounding box. This structure is particularly useful in the case of range search as we have to return a set of points near a given query point.

Figure 22 presents the experimental results with database size of 80,000 records. The results mirror the results from the previous experiments depicted in Figure 21. The results also show that it takes more processing time as search area grows from 40,000 to 80,000 records for a given overlap operation.

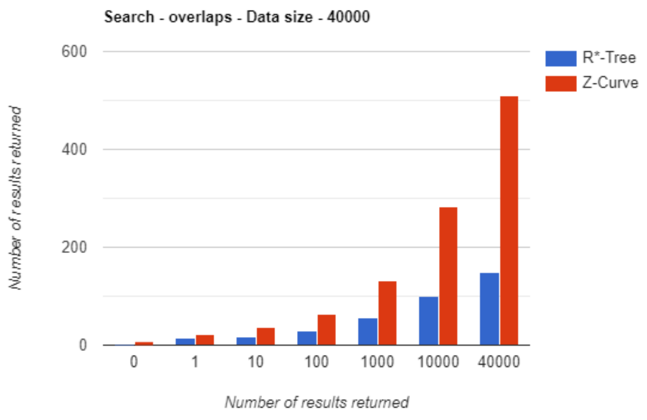


Figure 21: Overlap query (Database size = 40000 entries)

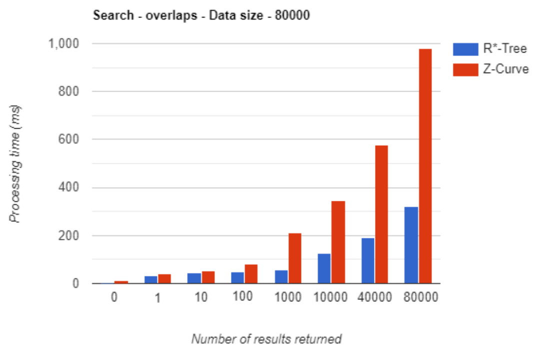


Figure 22: Overlap query(Database size = 80000 entries)

5.3.3 Intersect operation

Figure 23 and figure 24 show the performance of both the index systems given a intersect query. The first set of bars compares the processing times of an intersect operation from two systems when the intersect operation found a result that intersects with the given query bounding box. The second set of bars presents processing times when the search fails. Both index systems take longer processing time to determine there is no such target as compared to the time for a successful search. R*-tree index system performs better than the Z-Curve index system for both cases of successful search and search failure.

Since this query does not return all the values and returns a Boolean result as soon as it finds the first result, it is faster than the overlap operation. In the case of R*-trees, the query MBR and the node's MBR are compared once at each level. The performance gap between the cases of successful search and search failure in R*-tree index system is due to deeper tree travels that search failure entails. Z-curve index system underperforms than R-tree index for the cases of successful search and search failure because Z-curve scans the range of entries within the Z-values of the query MBR and eliminates false positives. This process takes additional time when compared to that of the R*-tree's mechanism.

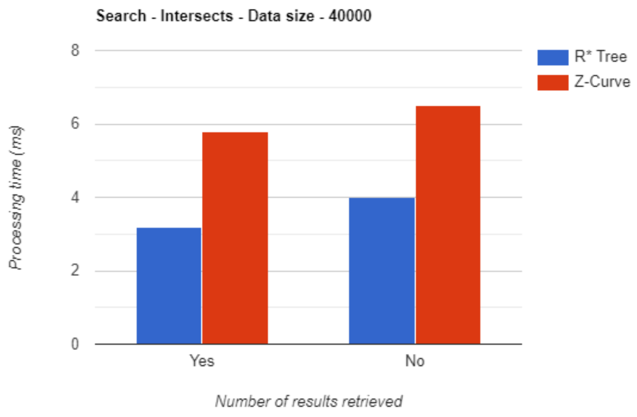


Figure 23: Search - Intersect query - Data entries = 40000

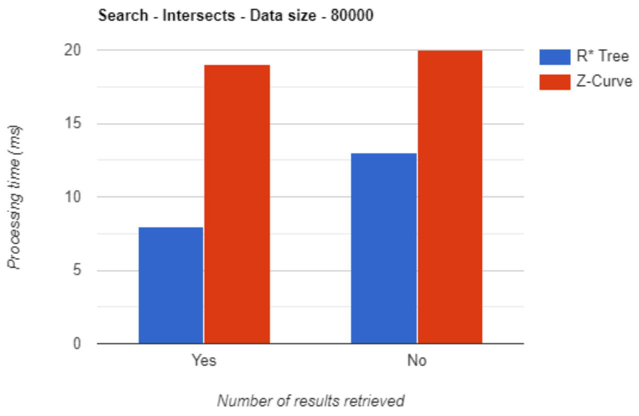


Figure 24: Search - Intersect query - Data entries = 80000

5.3.4 KNN (K Nearest Neighbor) operation

GeoMesa does not support KNN operation and thus cannot be executed in the Z-curve index system. Considering KNN operation as a prominent operation, we recorded the performance of R*-tree index to serve KNN operations in this experiment. Figures 25 and 26 present the experimental results with database sizes to be 40,000 and 80,000 records respectively.

Increasing the number of records in the database did not add up to the processing time in case of finding a smaller number of neighbors. While traversing down the R*-tree index to retrieve the neighbors, the depth-first search algorithm appends entries into the priority queue. Since the number of neighbors to be returned is less, given the total number of entries in the tree, the priority queue is filled up while reaching the depth of the tree index during the first pass. A massive increase in the processing time for retrieving 100 neighbors is because of the time associated with the depth-first search algorithm.

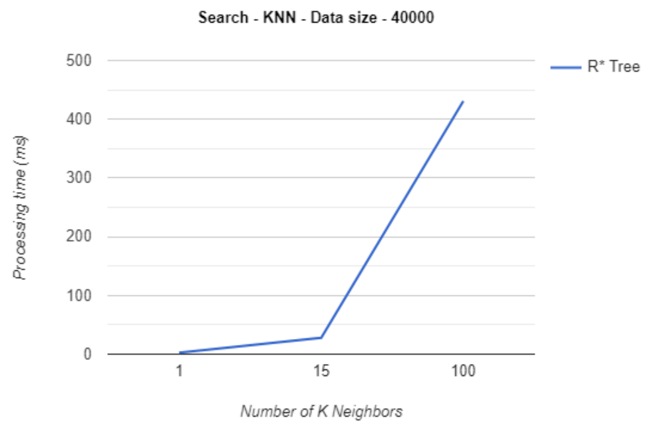


Figure 25: KNN (Database size = 40000 entries)

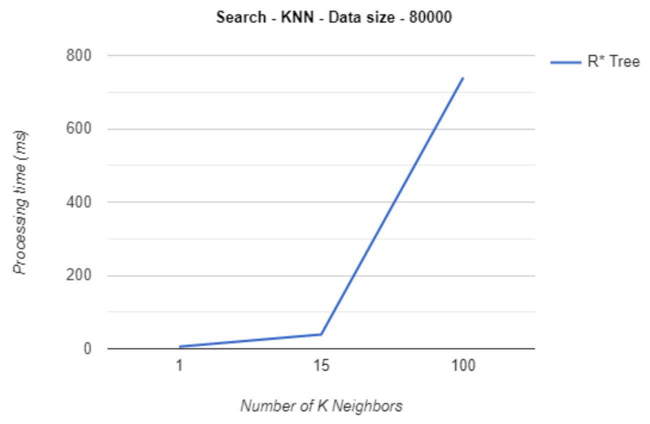


Figure 26: KNN (Database size = 80000 entries)

5.3.5 Delete operation

Delete operation is studied under two cases: deleting a single point and deleting all entries from the database. Figure 27 presents how the processing time of a given delete operation increases as the database sizes increase. For smaller data size, the

R*-tree system performs better compared to the Z-curve system. As the number of data entries stored in the database is increased, the Z-curve index performs better compared to the R*-tree system. This behavior is because the R*-tree system performs node reinsertion when the number of entries in the node falls below a threshold value. Hence it takes higher processing time when the data size increases and thus index size increases in the database. In the case of the Z-curve system, the processing time increases as the number of data entries stored in the databases increases.

Figure 28 shows the processing time to delete all entries from the database. As this operation does not involve any index modification, both the systems take nearly the same processing time regardless of the database size. The memory allocated to the index and the database is dropped as there are no filter conditions or lookup necessary.

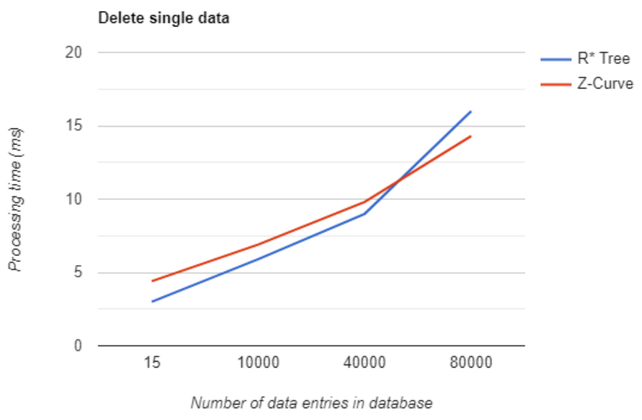


Figure 27: Delete single data

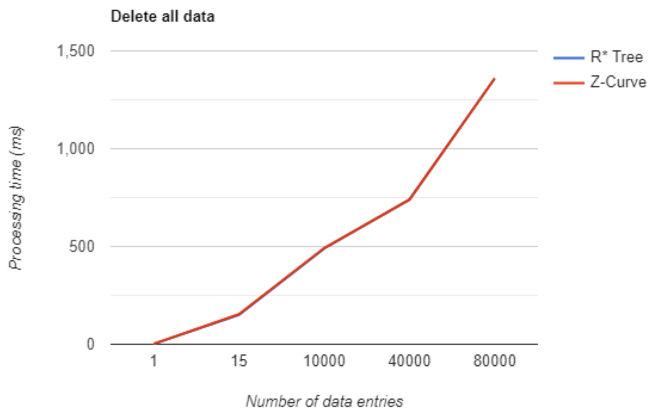


Figure 28: Delete all data

CHAPTER 6

Conclusions and Future Work

In this project, we developed an R*-tree index system and Z-curve index system to conduct a performance study on these two index mechanisms for geospatial processing in Cassandra. An R*-tree based indexing mechanism works based on the minimum bounding rectangles and is known to be efficient to index multidimensional data. Z-curve based indexing mechanism is used by GeoMesa which is an external library Cassandra currently is relying on to support geospatial operations. The processing times of representative geospatial operations are measured in both the index systems, and the experiment results are analyzed. The experiment results show that the R*-tree index system outperforms the Z-curve index system for search queries. We also found that the R*-tree index system underperforms compared to Z-curve as the database size increases because insertions in the R*-tree index system involve the overhead of spitting and forced-reinsertion to handle overflowing nodes. We also studied the performance of the K nearest neighbors operation, which is not currently supported by GeoMesa.

The experiments of this project were conducted in Cassandra deployed in a single server. High scalability within Cassandra is achieved by partitioning data across multiple nodes in a cluster. The research products and findings from this project can be extended to study distributed R*-tree index in Cassandra. Such distributed R*-tree indexes should maintain the index content consistent to new insertions and deletions across the cluster.

In this project, we set up experiments in a way that index always resides in memory. This setup was necessary to make an apple to apple comparison between index systems under test. In the future study, the performance of R*-tree can be examined under varying memory sizes to find the impact of disk access time on index performance. This work could also be extended to handle complex data types such as polygons and lines so that this indexing scheme can be used with a broader range of geospatial data types.

LIST OF REFERENCES

[1] ‘‘About raster data in spatial analyst.’’ [Online]. Available: https://pro.arcgis.com/en/pro-app/help/analysis/spatial-analyst/basics/about-raster-data-in-spatial-analyst.htm

[2] ‘‘G-FAQ - what is spatial topology in GIS Part I.’’ [Online]. Available: https://apollomapping.com/blog/g-faq-spatial-topology-gis-part

[3] ‘‘Did you know Raster vs. Vector.’’ [Online]. Available: https://blogs.lib.uconn.edu/outsidetheneatline/2009/08/17/did-you-know-6-raster-vs-vector/

[4] ‘‘R-tree how to calculate minimum bounding rectangles of non leaf nodes.’’ [Online]. Available: https://stackoverflow.com/questions/45535271/r-tree-how-to-calculate-minimum-bounding-rectangles-of-non-leaf-nodes

[5] A. Guttman, ‘‘R-trees. A Dynamic Index Structure For Spatial Searching,’’ vol. 14, no. 2, pp. 47--57, 1984. [Online]. Available: https://doi.org/10.1145/971697.602266

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, ‘‘The R*-tree: an efficient and robust access method for points and rectangles,’’ vol. 19, no. 2, pp. 322--331, 1990. [Online]. Available: https://doi.org/10.1145/93605.98741

[7] ‘‘Z-order curve.’’ [Online]. Available: https://en.wikipedia.org/wiki/Z-order_curve

[8] ‘‘Get to the point with big spatial data.’’ [Online]. Available: http://www.ccri.com/2015/07/29/get-to-the-point-with-big-spatial-data/

[9] L. Xiang, J. Huang, X. Shao, and D. Wang, ‘‘A mongodb-based management of planar spatial data with a flattened r-tree,’’ vol. 5, 2016. [Online]. Available: https://doi.org/10.3390/ijgi5070119

[10] ‘‘Apache cassandra architecture tutorial.’’ [Online]. Available: https://www.simplilearn.com/cassandra-architecture-tutorial-video

[11] J. Hughes, A. Annex, C. N. Eichelberger, A. Fox, and A. Hulbert, ‘‘Geomesa: a distributed architecture for spatio-temporal fusion,’’ vol. 9473, 2015. [Online]. Available: https://doi.org/10.1117/12.2177233

[12] P. Lobo, ‘‘A framework for the detection of utility conflicts using geo-spatial processing,’’ 2017.

[13] P. A. Longley, M. Goodchild, D. J. Maguire, and D. W. Rhind, *Geographic Information Systems and Science*, 3rd ed. Wiley Publishing, 2010.

[14] H. C. Karnatak and V. Kumar, ‘‘Performance study of various spatial indexes on 3d geo-data in geo-rdbms,’’ vol. 30, 2014. [Online]. Available: https://doi.org/10.1080/10106049.2014.952354

[15] X. Zhang, W. Song, and L. Liu, ‘‘An implementation approach to store gis spatial data on nosql database,’’ 2014. [Online]. Available: https://doi.org/10.1109/GEOINFORMATICS.2014.6950846

[16] ‘‘Db-engines ranking.’’ [Online]. Available: https://db-engines.com/en/ranking

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

[18] E. Westra, *Python Geospatial Analysis Essentials*. Packt Publishing, 2015.

[19] D. Comer, ‘‘Ubiquitous b-tree,’’ *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121--137, 1979. [Online]. Available: https://doi.org/10.1145/356770.356776

[20] T. Sellis, N. Roussopoulos, and C. Faloutsos, ‘‘The r+-tree: A dynamic index for multi-dimensional objects,’’ in *Acta Informatica*, 1987. [Online]. Available: https://doi.org/10.1007/BF00288933

[21] I. Kamel and C. Faloutsos, ‘‘Hilbert r-tree: An improved r-tree using fractals,’’ in *20th International Conference on Very Large Data Bases*, 1994.

[22] G. M. Morton, *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. IBM Ltd., 1966.

[23] R. Finkel and J. Bentley, ‘‘Quad trees: A data structure for retrieval on composite keys,’’ 1974. [Online]. Available: https://doi.org/10.1007/BF00288933

[24] A. Cockcroft and D. Sheahan, ‘‘Benchmarking Cassandra Scalability on AWS - Over a million writes per second,’’ https://medium.com/netflix-techblog/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e, 2011, [Online; accessed 25-April-2019].

[25] A. Lakshman and P. Malik, ‘‘Cassandra: a decentralized structured storage system,’’ vol. 44, pp. 35--40, 2010. [Online]. Available: https://doi.org/10.1145/1773912.1773922

[26] H. Samet, *Foundations of Multidimensional And Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2006.

[27] “Simplefeature (geotools module).” [Online]. Available: http://docs.geotools.org/stable/javadocs/org/opengis/feature/simple/SimpleFeature.html

[28] “Railroad bridges - arcgis hub.” [Online]. Available: https://hub.arcgis.com/datasets/72056d6f0b35445f893c642c033fede3_0

[29] V. Agafonkin, “Rbush - a high-performance javascript r-tree-based 2d spatial index for points and rectangles,” https://github.com/mourner/rbush, 2017.