Geo Hash-Partitioner

Motto:

Introduce a partitioning scheme in Apache Spark that partitions data-sets considering Geo-spatial location such that:

- Communication in distributed environment is expensive – minimize network traffic by allowing spark program to control RDD partitioning
- Spatial Locality is retained: nearby points are probably in the same partition / cluster
- Minimize data points shuffling during Geo-spatial queries
- Optimized for Search related operation on Geo-points

Default partitioning in Spark:

Spark has two partitioning schemes ***Range Partitioner*** and ***Hash Partitioner*** operating on Key value pairs. As it is obvious that range partitioning is not suitable for retaining spatial locality in storage, the option is Hash Partitioner.

Built in Hash partitioner could not be used directly for the following reasons:

- Hashing technique on keys(geo-points) is based on Java hashCode() which does not help in retaining locality
- Also there are criticisms on using Java hashCode() on distributed environment, esp., on virtual machines because the implementation considers physical memory location

We need a special partitioning technique that is aware of spatial locality of data points

Indexing and Encoding Schemes:

Several indexing schemes are being used in today's spatial databases to store Geo-spatial information.

- Scheme 1
- Scheme 2...

[Benefits of choosing an Encoding scheme is described in detail in other document]

Geo-Hash encoding:

We are encoding the coordinates based on GeoHashing technique to use them as key for partitioning. *[Benefits and Limitations of using GeoHashing]*

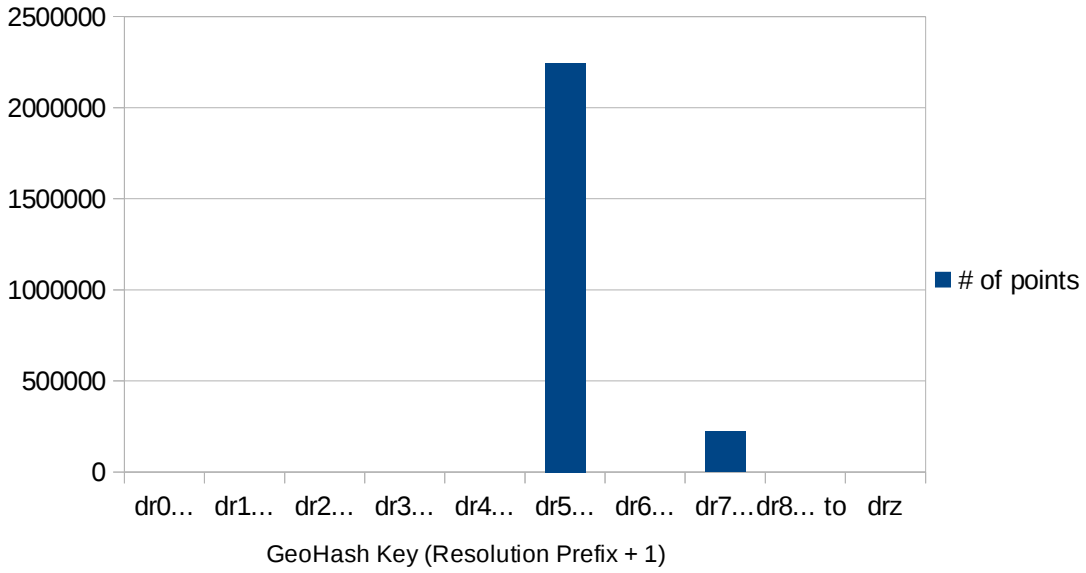
This encoding scheme based on z-curve helps in retaining Spatial locality and is used in recent No-SQLs like MongoDB to support spatial features.

➢ Rough Partitioning:

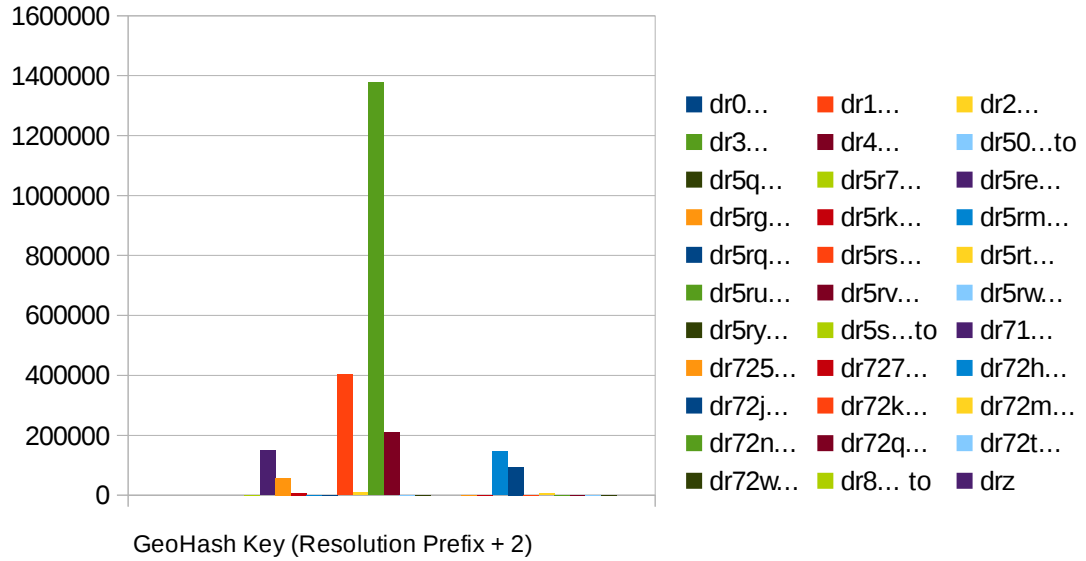
The implicit side effect of Hashing techniques is the improper distribution of load. Suppose the partitioning is made based on GeoHashing by altering resolutions the final data distribution results in load imbalance.

Consider a small data-set for points in Manhattan region:

- All the points have a common prefix of ***dr.....***
- The next code has 32 probabilities[Geohash base32 code 0-9 b~z], based on which we create 32 partitions.



- Going one level deeper into resolution



- This imbalance continues to exist until we ignore the load distribution

Load Aware Geo-Hash partitioning:

We consider the load distribution + take advantage of benefits from GeoHashing technique to make balanced, location aware partitions. By doing, this we add the benefit of Range partitioning to Hash Partitioning(GeoHash). We do this by,

- Determining a threshold parameter
- Stop partitioning on areas which has fewer points (or) combining neighbors with less data points together
- Drilling deeper into resolutions for areas which has higher number of data points
- Iterative process that stops when there are no more areas with more points than the threshold
- This varied resolution in areas is stored in general tree data structure

Algorithm:

Determine the Threshold:

We determine threshold in terms of number of records each partition could handle within a default partition size. We choose the number of records as the base because it might vary depending upon number and type of column values in each record.

Average Record in each partition = Total number of records / Number of default partitions

= 2459106 / 17 = 144654 records

Total number of records counting is done alone with any operation that involves overall scan

Encode the Coordinates:

- Map coordinates in data-set into GeoHash String (base32) and create an RDD with just the key string(reduce load during partition computation)

Prefix Finding:

The first step is finding out the common prefix for all the data points. This could be done in two ways:

- Bounding Box: Measure the MIN(lat, long), MAX(lat, long) from coordinates and determine the bounding area. Find their GeoHash code and Calculate the longest common prefix
- From the Key String RDD determine the prefix by map reduce on first character of each key string. Continue with next String and proceed as long as the number of distinct entries in Map is more than one

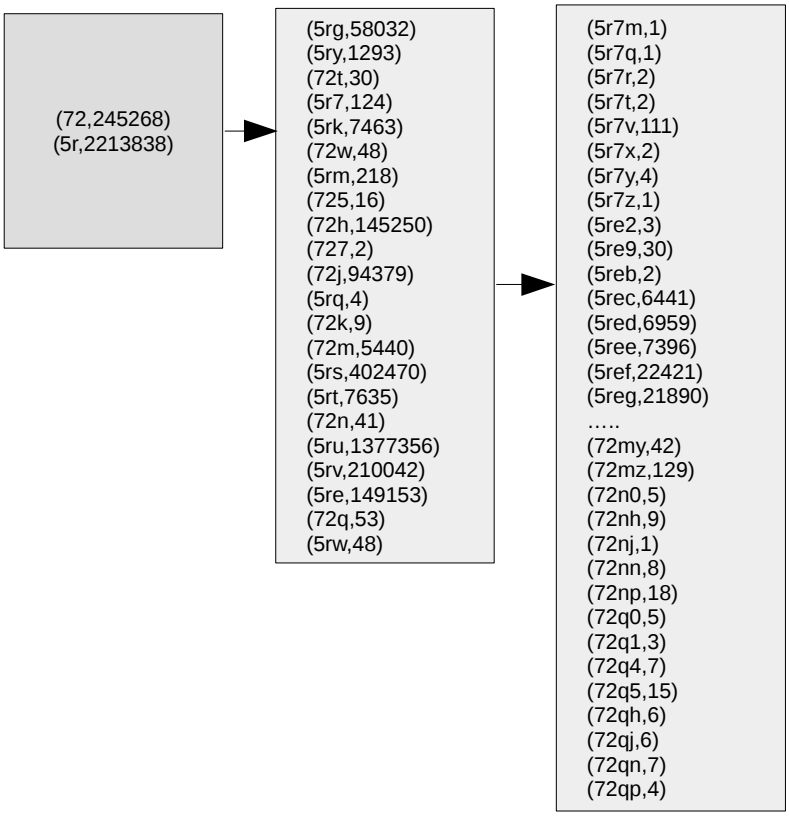
(d, 2459106) → (dr, 2459106) → (dr72,245268),(dr5r,2213838)

Deeper Resolution Tree building:

- Instantiate and maintain a pool of partition numbers.
- Repeat the count operation by increasing the length of the code by one character

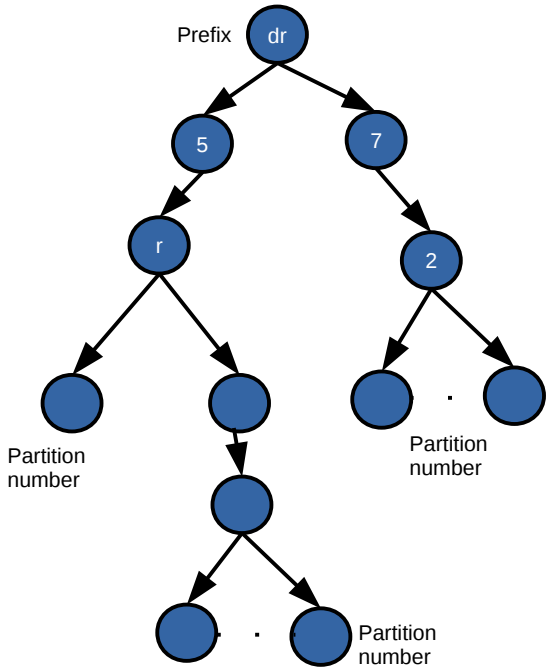
for every iteration until there are no segments that has more records than threshold.

- If segment(s) < threshold → add it to the tree and take the next available partition number from the counter pool and assign them as leaf node.
- If segment(s) > threshold → add it to the tree and continue the iteration(this node will get more children in the next iteration)



Key DS:

- The tree with details about the partitioning is maintained and passed to GeoHash partitioner.



Optimization:

- In real scenario, the process of starting from Prefix and increasing by one code for every iteration could be done in the reverse. This would be minimize the RDD operations {explain it}

Overview: