Balanced Geohash Partitioning and Efficient Retrieval of Geospatial Big Data on Distributed and Parallel Platforms (Apache Spark)

Master-Thesis von Hariharan Gandhi aus Darmstadt
Tag der Einreichung:

1. Gutachten: Prof. Alejandro Buchmann, Robert Rehner M.Sc [TU Darmstadt]
2. Gutachten: Dr. Gregor Moehler, Dr. Raghu Kiran Ganti [IBM Research & Development]

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik DVS

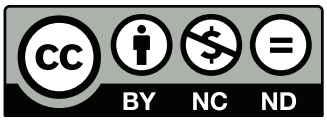
Databases and Distributed Systems
Department of Computer Science
Technical University of Darmstadt

Balanced Geohash Partitioning and Efficient Retrieval of Geospatial Big Data on Distributed and Parallel Platforms (Apache Spark)

Vorgelegte Master-Thesis von Hariharan Gandhi aus Darmstadt

1. Gutachten: Prof. Alejandro Buchmann, Robert Rehner M.Sc [TU Darmstadt]
2. Gutachten: Dr. Gregor Moehler, Dr. Raghu Kiran Ganti [IBM Research & Development]

Tag der Einreichung:



Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 17th May 2016

_____

(Hariharan Gandhi)

Abstract

Amongst several big data disciplines, spatial data is one of the most exponentially proliferating data type. This could be attributed to the explosive increase in the number of users of mobile devices with sophisticated GPS location sensors, emergence of social media and advancement in weather and navigation systems. More and more applications and businesses are providing location based services, personalised suggestions and most disruptive apps of recent times, such as Uber for taxi services, AirBnB for resource sharing, Google Maps, geo-tagged Tweets, Instagram photos, drones, logistics, weather apps, all provide location aware service to their users.

Exploiting this Geospatial dimension of information for the benefits of analytics is interesting and challenging. To store and process such huge volumes of data, we harness the efficiency of emerging distributed and parallel computing platforms, such as Apache Spark, Hadoop Mapreduce, Hadoop Distributed Files Systems. However, conventional distributed storage and processing systems, do not provide native support for handling and efficiently storing Geospatial data types. The challenge is further made intricate by the necessity to preserve data points' locality in order to minimize network cost involved due to shuffling of intermittent results between the computing nodes. In this thesis work, we use the large scale data processing engine, Apache Spark, to process Geospatial datasets. Spark provides *resilient distributed dataset* (RDD), which is a collection of dataset partitioned across the nodes of a cluster for parallel operation [1]. Geospatial operations such as *contains, within, overlaps* etc involves a shuffling between partitions when there is a scan for data points in a region. This means that geographically closer data points should be preserved in the same partitions within a Spark RDD for reduced shuffling, minimal network cost, and efficient scans and retrieval. Spark's default partitioning are Range-based and Hash-based (Java Hashcode) both of which are not suitable for achieving spatial locality within partitions. It emphasizes the need for developing a custom Geospatial partitioning and retrieval methodology tailored to store and retrieve the overwhelming amount of Geospatial big data.

This thesis works aims at addressing this issue by proposing and providing a geographically load balanced partitioning mechanism, for Apache Spark, tailored for Geospatial dataset and further by providing an optimized querying layer for efficient retrieval of records on spatial queries. Experimental results, using New York Taxi dataset, show improvement in data points' locality for minimized shuffling and efficiency with scanning and retrieving results for spatial queries in terms of response time and number of records scanned. [2]

Contents

List of Figures

A Introduction

UNDER PROOF READING

A.1 Problem Statement and Motivation

A.2 Proposed Approach in this Thesis

A.3 Dissertation Road map

B Background and Related Work

UNDER PROOF READING

B.1 Apache Spark

B.2 Spatial Indexing techniques

B.3 Need for (Spatial)Partitioning: Shuffling

B.4 Space Filling Curves

B.5 Geohash

B.6 Related Work

B.7 Summary

C GeoHash Partitioner: Design and Implementation

In this chapter, we look into the design and implementation of a custom spark partitioner that provides Geospatial partitioning of spatial big data

C.1 Partitioning

As we realize, communication is very expensive in a distributed environment, it is necessary to distribute data in a such a way that minimizes communication. This would result in performance gain and greatly improves speed of execution for key based transformations. In Geospatial application, this means, laying out RDD partitioning in such a way that retains spatial locality of the data points.

However, (Geospatial) Partitioning is not helpful (or not beneficial) in all applications and scenarios. The gain is considerable when: [2]

- The dataset is scanned multiple times in key oriented operations like *joins*.
- Considerable dataset size wherein benefits of partitioning outweighs the effort involved partitioning.
- And clearly when the application involves a relatively more spatial queries.

C.1.1 Default Partitioning in Spark

Spark allows programs to take control over their RDD's partitioning and this configurable partitioning is provided only for RDDs of key-value pairs, for instance JavaPairRDD [3], since special distributed "*shuffle*" operations, such as grouping or aggregating the elements by a key.

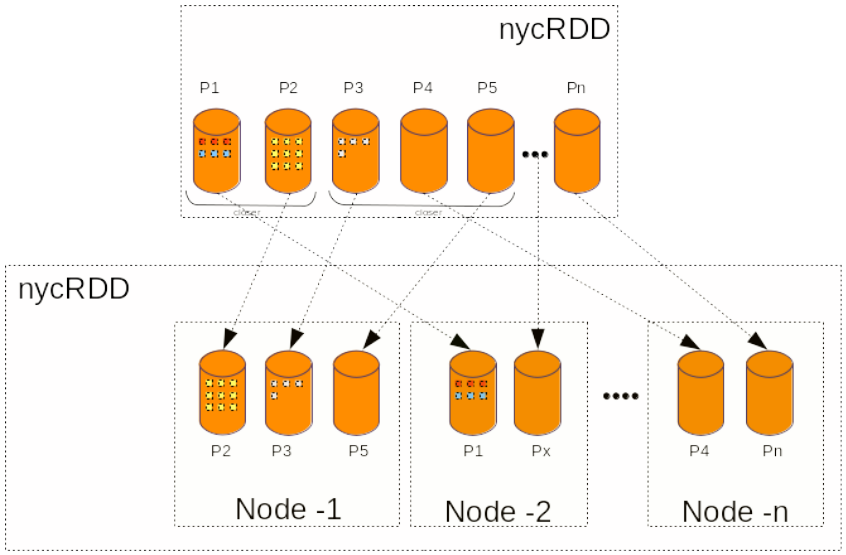


Figure 1: Hash key based Partitioning scenario in Spark

Spark provides two default partitioning schemes:

HashPartitioner: partitions records based on their key's hash using Java's Object.hashCode method. The partition is obtained as,

$$Partition = key.hashCode() / numPartitions$$

RangePartitioner: partitions sortable records by range into roughly equal ranges. The content of the RDD passed in are sampled to determine the range [4]. This partitioning scheme is useful when keys have natural ordering and are non-negative.

Spark's `HashPartitioner` and `RangePartitioner` do not involve **domain specific knowledge** during partitioning. The absence of domain knowledge tends to hash the locations *L1(40.881142 -73.907021)* and *L2(40.88129 -73.90693)* into two different partitions even though these two locations are physically closer and are intended to stay within a single partition. However, Spark provides **Custom Partitioner object** to leverage domain specific(Geospatial) knowledge to further cut down communication cost. Rest of this segment demonstrates the design of a Geospatial aware custom Partitioner.

It is important to note that during partitioning based on key, spark does not provide explicit control of which physical worker node each key goes to. This is partly because of the system design to support node failures. [2] However, it ensure that the set of keys resides in the same partition. For instance, keys that produce same hashcode stick together in the same partition. So, in our aim to achieve spatial locality with the data points we could achieve partition level locality, not node level i.e., the data points corresponding to the neighboring region, at a particular level of resolution, might physically reside in different nodes of the cluster. However all data points belonging to a 'same' region(containing same geohash prefix) reside within a single partition and in turn in the same physical node. So, we could summarize the properties of partitioning in Spark as follows: [5]

1. Each node in a spark cluster can contain one or more partitions
2. Partitions never span across multiple node. Records in a single partition are guaranteed to be in the same node.
3. The number of partitions in the RDD, **`numPartitions`** parameter, could be configured. Default value: Total number of cores on all executor nodes.

Contract to implement a Custom Partitioner:

The implementation of Custom Partitioner should extend org.apache.spark.Partitioner class and implement the following three methods: [2]

- **`numPartitions: Int`** - returns the number of partitions that will be created.
- **`getPartition(key: Any): Int`** - returns the partition ID (0 to numPartitions-1) for a given key.

- **`equals()`** - the standard Java equality method. When two RDDs are aggregated or operated on, Spark uses this method to verify the Partitioner object against other instance of Partitioner if both RDDs are partitioned the same way i.e., using the same custom Partitioner.

C.2 Geohash as Partitioning Key

In the background section, the structure and usage of Geohash technique was discussed. Here we use Geohash as the key for creating our Key-Value RDD. We generate this Geohash key by encoding the Latitude and Longitude values from each record.

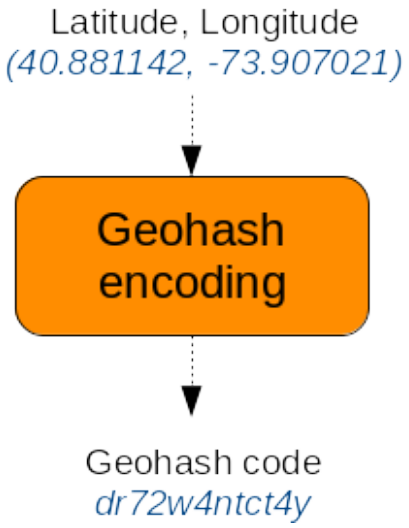


Figure 2: Geohash encoder to encode spatial points to Geohash Key

In our approach, we use Geohash of Latitude, Longitude of each record as its key. We create a Key-Value RDD with the generated Geohash code as Key and the record itself as the value, as shown in table.

$$Geohash\ Key = GeoHashEncode.withCharacterPrecision(Lat, Long, 12^{1}).toBase32()$$

dr5ru7c02wnv	-	8EB9CE00A5AD729095BC542FC	VTS	2013-01-01 10:31:00	840	4.67	***-73.991516***	***40.75798***	CSH	16.5	0	0.5
dr5rugbmh6ym	-	EAF2F96369A07DB6305A02DB37	CMT	2013-01-02 16:18:43	932	6.1	***-73.970413***	***40.758778***	CRD	19.5	1	0.5
dr5rsrjjfgg3	-	01C3BF069A4CB23CEE0389534	CMT	2013-01-01 03:41:39	867	2.9	***-73.985931***	***40.732819***	CRD	12.5	0.5	0.5
dr5ruk393jx9	-	5540EDF731F24EAF0A762A2AC	CMT	2013-01-02 23:02:03	836	4.5	***-73.990807***	***40.760895***	CRD	16	0.5	0.5
dr5rusvvxq1d	-	49FB6E186D4E1784996B3A88A8	VTS	2013-01-02 15:42:00	150	5.72	***-73.973724***	***40.764374***	CSH	22	0	0.5

Figure 3: Sample records of Key-Value RDD with Geohash code as key

[1] 12 - the maximum precision of 12 characters for 64bits Geohash

As discussed in the background section of Geohash technique, this encoding scheme based on Z-curves, could be used in retaining spatial locality. All data points with same Geohash prefix belong to the same geographical region. This encoding scheme also helps in identifying neighbour along the path of the z traversal. So, the next step is create custom partitioner based on the Geohash code.

C.3 Custom Geohash Partitioner

The custom extension of Partitioner object should be able to read a recordâĂŹs key and assign a partition number specific to that particular key. Since we need to create spatial awareness to partition allocation we cannot utilize Java Hashcode method on the keys(Geohash) to determine its destined partition. Hence, our custom Geohash partitioner requires a mapping from desired Geohash grid to a physical partition number of the RDD. So, whenever there is an input record's key, the custom partitioner refers to mapping meta-data and returns destined partition number. The design and construction of the *GeohashPartitionMapper* is explained in detail in the following sections. The approach is to leverage the inherent property of Geohashes to denote grids on the earth surface, to create the partitions. For instance, all of New York region falls under the grid 'd'. So, when we partition data from entire world and need all data points in New York and neighboring region to reside in the same partition then we map all Geohash key with prefix of length 1 as 'd' to the same partition number.

Key '*dr5rugbmh6ym*', at geohash grid resolution 1 has the prefix 'd', goes to partition 'x' Now Key '*dr5rusvvxq1d*', at geohash grid resolution 1 has the prefix 'd', also goes to partition 'x' Whereas the key '*sr72w4ntct4y*' with 's' as grid resolution 1 prefix, goes to partition 'n', because physically the code represents a point in Europe, not NYC. The Geohash grid for the world with resolution size '1' is shown in Figure 4

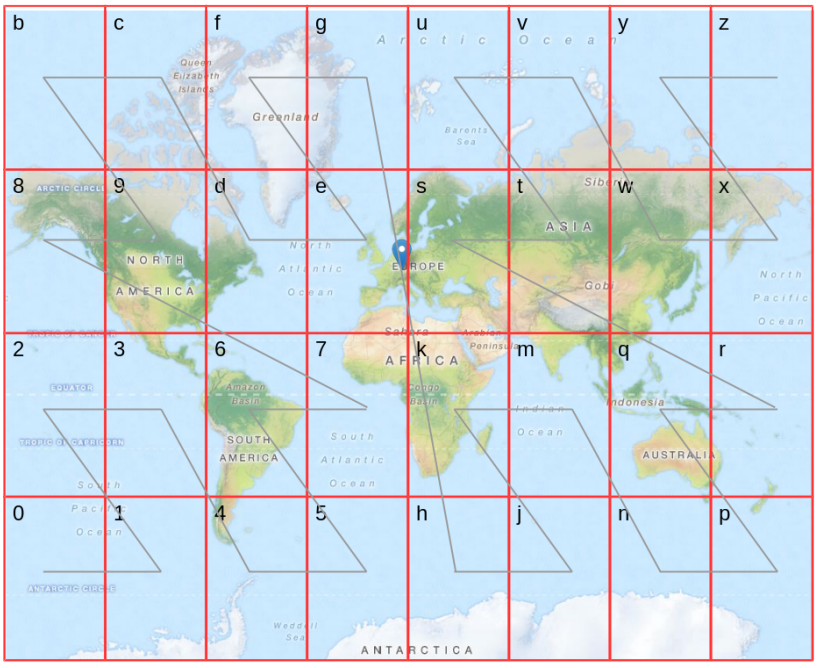


Figure 4: Geohash grid of resolution '1'

To proceed further deeper and to denote smaller region as a partition size, we could simply increase the size of Geohash prefix considered as partition. This means drilling into Geohash grid resolution 2, wherein each of the 32 grids are in turn sampled into another 8x4 grids(32 grids). This would result in a permutation of 32 x 32 = 1024, distinct partitions, Figure 5. As the work proceeds with this approach, we are hit by the explicit side effect of Hashing techniques - imbalance in load distribution as seen in Graph 6. This is also prominent with spatial datasets with data point concentrated at hotspots, major cities and metropolitan areas.

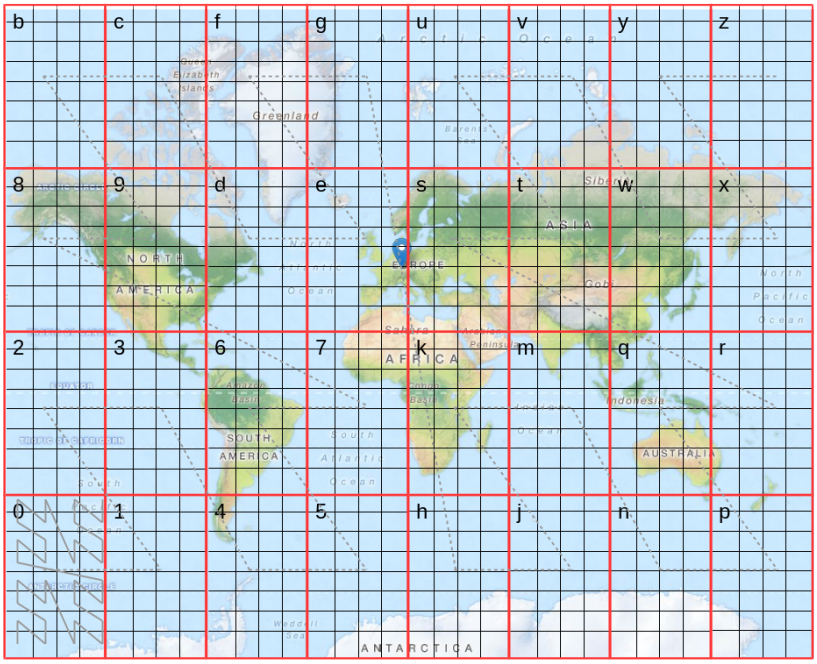


Figure 5: Geohash grid of resolution '2'

At a particular Geohash grid resolution, amongst the 32 cells of a grid, if one grid represents a major city (hotspot), then the remaining 31 cells/partitions have very few data points and in turn result in idle processing time. Lets look into a smaller data set for points in Manhattan region:

- All data points have a common Geohash key prefix - '*dr*. . . . . ' i.e., the complete Manhattan dataset is enclosed with Geohash grid resolution 2.
- So, in order to create Geohash key to Partitions mapping, we drill deep into grid resolution 3. It results in 32 possibilities[Geohash base32 code 0-9 b z] from '*dr0*. . . . . ' to '*drz*. . . . . ' and hence 32 partitions. However, when we look at Figure 6, we realize that around 95% resides in a single partition, 5% in another, leaving out 30 empty partitions
- An attempt to balance load, by drilling deeper into one more resolution, would result in further increase starving partitions as shown in Figure 7

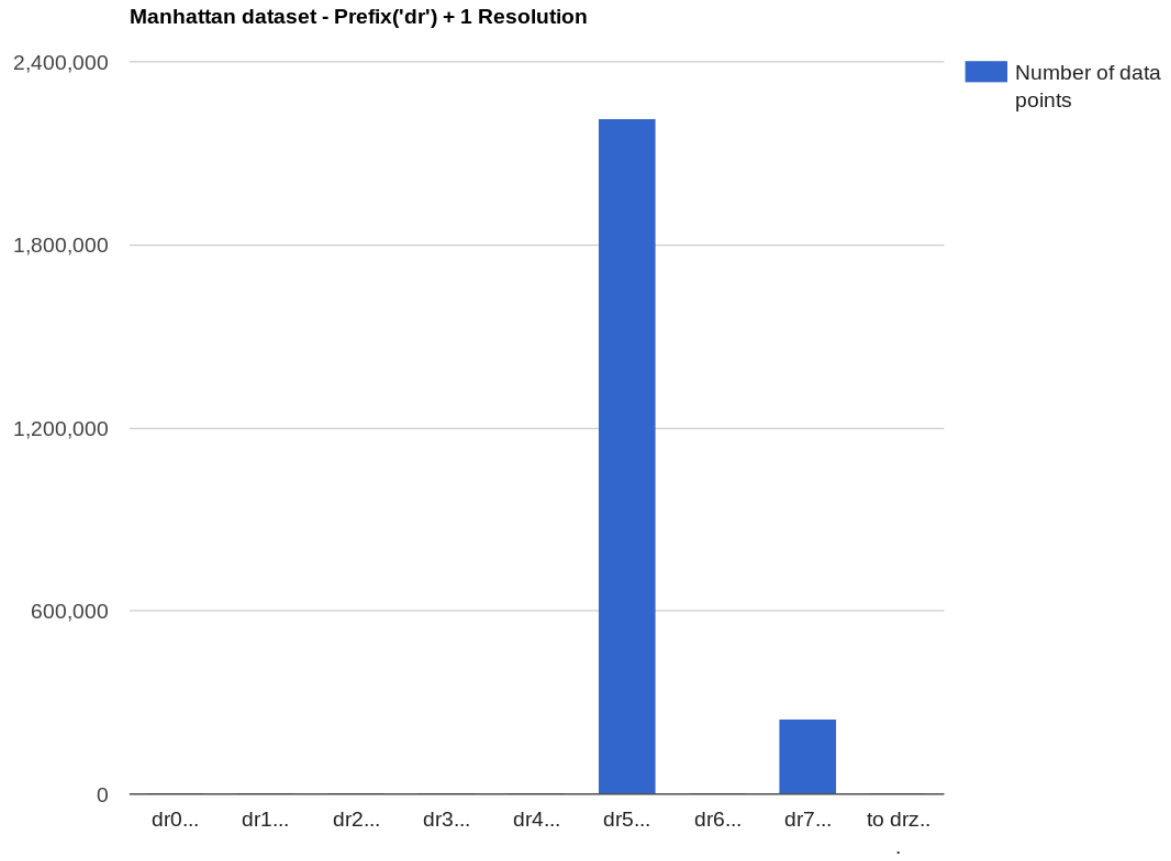


Figure 6: Imbalanced distribution of elements into partitions

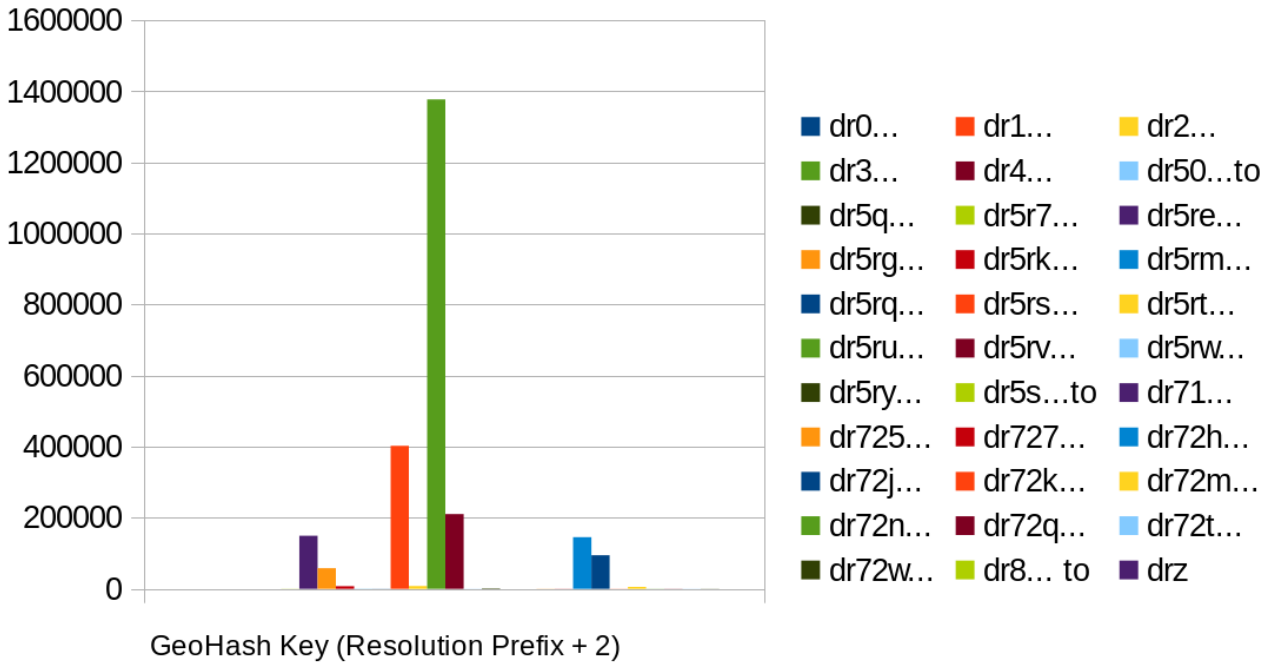


Figure 7: Increase in number of empty (or) imbalanced partitions

- This imbalance continues to exist until we ignore the load distribution and blindly increase the overall resolution of the Geohash grid
- Extremely varied size of partitions would result in other spark executors waiting idly for tasks or in *speculative execution* of tasks for slow running tasks. This is the behavior of spark to transfer Map task from slower nodes to the nodes that have already finished processing their allocation [6]. Or even worse hit time out on `spark.locality.wait` which would result in poor locality violating the core purpose of our partitioning. The effort to achieve *process local* fails and task steps through process-local, node-local, rack-local and then any. [7]. However both these parameters could be controlled by configuration settings.

C.3.1 Load Aware Geohash Key Generator

To avoid this phenomenon, our approach takes into consideration the geographical distribution of load and in addition leverages benefits of Geohashing to make balanced, location aware partitioning. The result would benefit both from (Geo)Hash based partitioning, to achieve locality, and from Range based partitioning, to attain maximum achievable (or) configured level of uniform load distribution. The custom partitioner achieves this by,

1. Determining a threshold parameter based on the size of dataset.
2. Drilling deeper into resolutions only for those grids which has number of data points greater than the calculated threshold.
3. Stop partitioning on grids which has fewer points which would result in varied levels of resolution in different Geohash grid depending on the geographical load distribution.

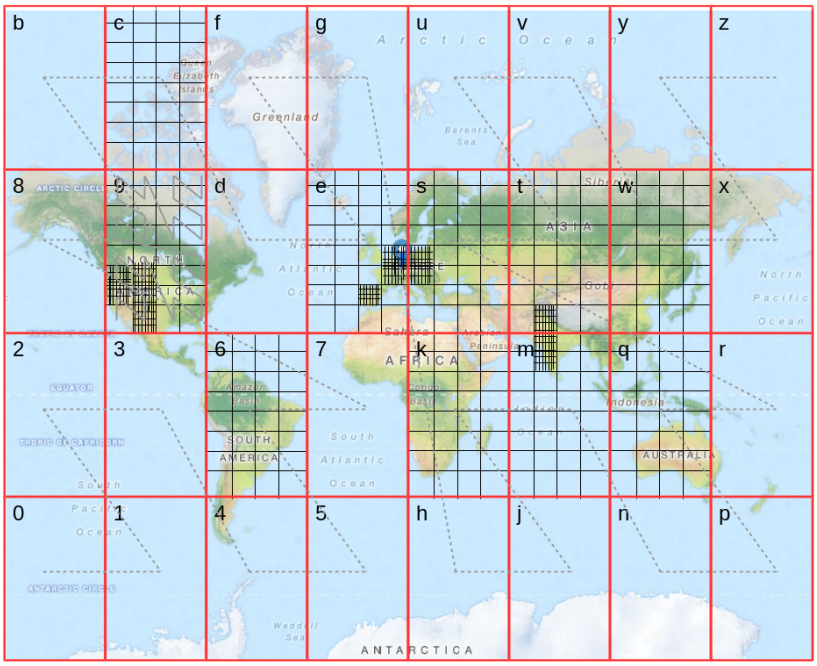


Figure 8: Geohash grids of varying resolution based on Load distribution

4. Iterative process stops at the resolution when there are no further grids with more points than the threshold (or) at a pre-configured level of resolution. For our evaluation, we configured a maximum grid resolution depth of 6 levels.
5. This varied resolution in areas is stored in a general (tree like or HashMap) data structure

C.3.2 Algorithm

Encode the Coordinates

- Create the key-value RDD, **JavaPairRDD***[String, Array[String]]*, with '*Geohash of the coordinates*' in data-set as keys. This is the base RDD which is to be partitioned.
- Create a secondary RDD, with just the key from previous RDD. *RDD[String]*. This reduced RDD, with only Geohash string, helps in reduced load during learning about load distribution and creating a Geohash grid to partition mapping.

Determine the Threshold

The next step is determine the *threshold*, which is considered as maximum number of records a region/grid could contain to stay in a single partition. Threshold is calculated in terms of average number of records each partition could handle within a default partition size. The approach chooses *number of records* as the threshold parameter because it might vary depending upon number and type of columns for different datasets.

Partitioning Threshold [2] *= Total number of records / Number of default partitions*

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

Deeper Resolution Tree building

The next step is to understand the load distribution w.r.t threshold. This is then used to build the data structure which contains mapping between Geohash Grid to partition number.

- Instantiate and maintain a new list to collect key prefix.
- From the KeyRDD, starting from the prefix size '1', count the number of elements in distinct keys (reduceByKey)

  val level1 = keysRDD.map(a => (a.substring(0,1), 1)).reduceByKey(_+_)
- From the result set, if value < threshold → add the current Geohash prefix to the list
- For all values > threshold → increase the prefix size to '2' and again count the number of elements in distinct key

  val level2 = keysRDD.map(a => (a.substring(0,2), 1)).reduceByKey(_+_) (for all 'a' not shortlisted in Level1)

[2] Average Record in each partition

- Repeat the count operation by increasing the length of the code by one character for every iteration until the predefined level of Geohash resolution or there are no segments that has more records than threshold.
- When the desired final level is reached, do not filter based on threshold, instead add all the left out Key prefix to the list.
- This would involve 'n' times complete scan of dataset. So, to optimize it, we could create a *reduceByKey* on the nth prefix length and calculating it in the reverse. This would involve only 1 complete scan of the data set. (n - the predefined prefix limit need to estimate load)

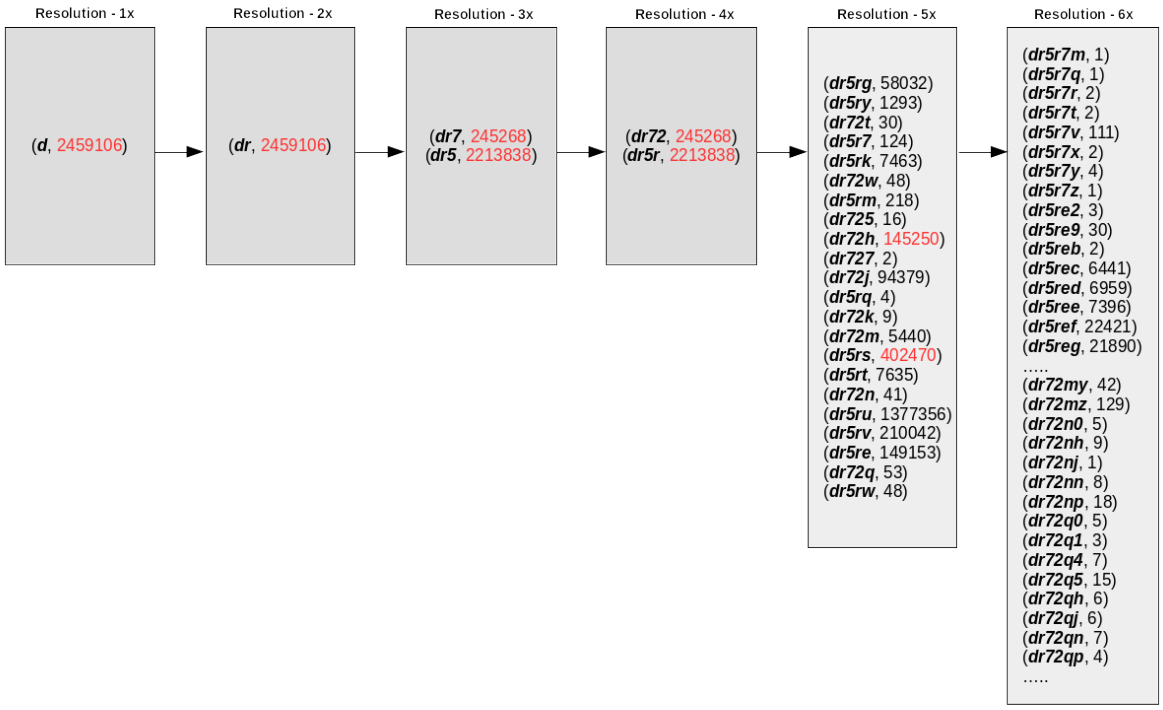


Figure 9: Levels of load distribution for increasing Geohash resolution

Variable length Key data structure

Finally the list would contain varying lengths of Geohash prefix. This variable length key could be stored in a tree data structure or a hashmap and each of them assigned a partition number. This would be the GeoHash partition mapper metadata and it is fed as input to our custom Geohash Partitioner for Spark.

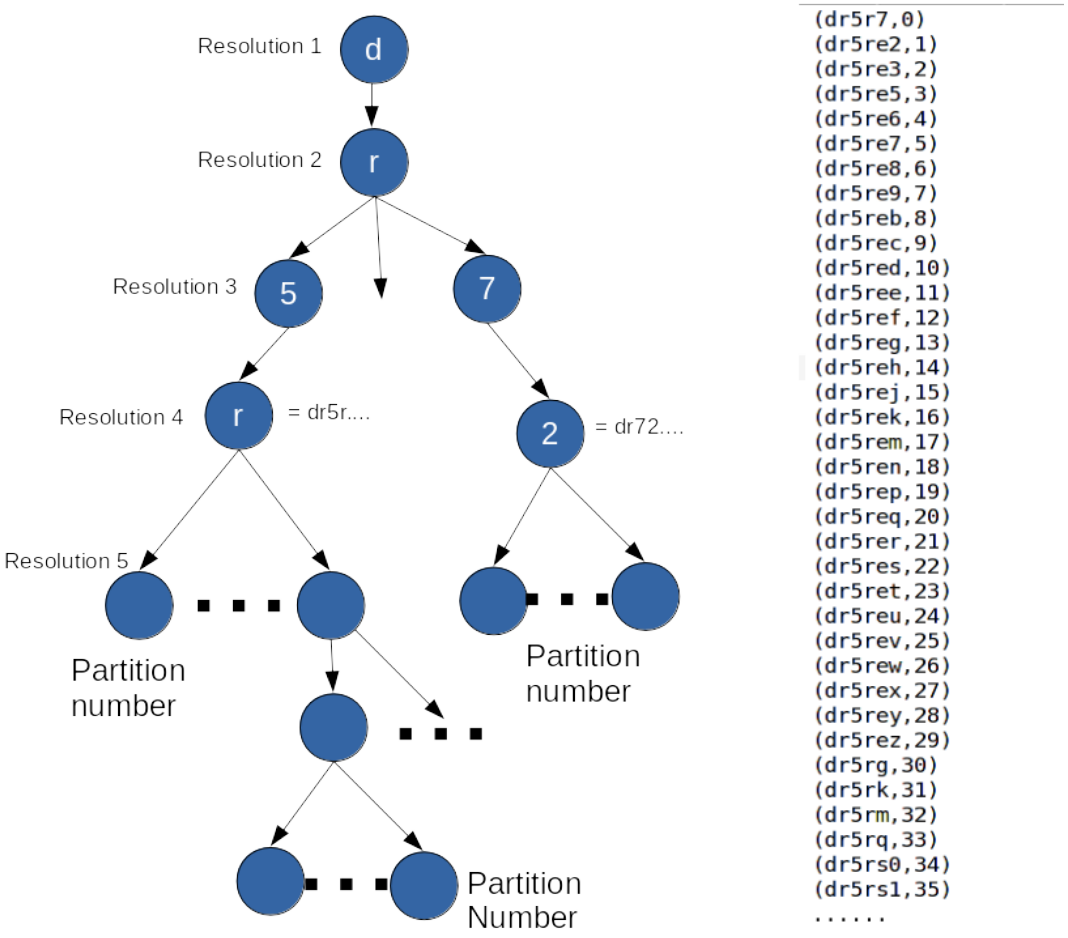


Figure 10: Geohash Partition mapper: variable length load aware Geohash key data structure

Optimized Partitions

The partitioner achieves good geo-load balancing by following above approach. However, there is scope for further optimization. Consider the scenario where we have two different Geohash Prefix of same length and number of elements in one Geohash Prefix exceeds the threshold. In this case, the procedure is to retain the prefix with less elements than threshold and drill deeper into the other prefix. This would result in 32 new partitions in the additional resolution. However if only one among these 32 contributed to more than 90% of its parent prefix's total number of elements, then this leaves other 31 under fed partitions which could have otherwise been combined into a single partition, see Figure 11, 12

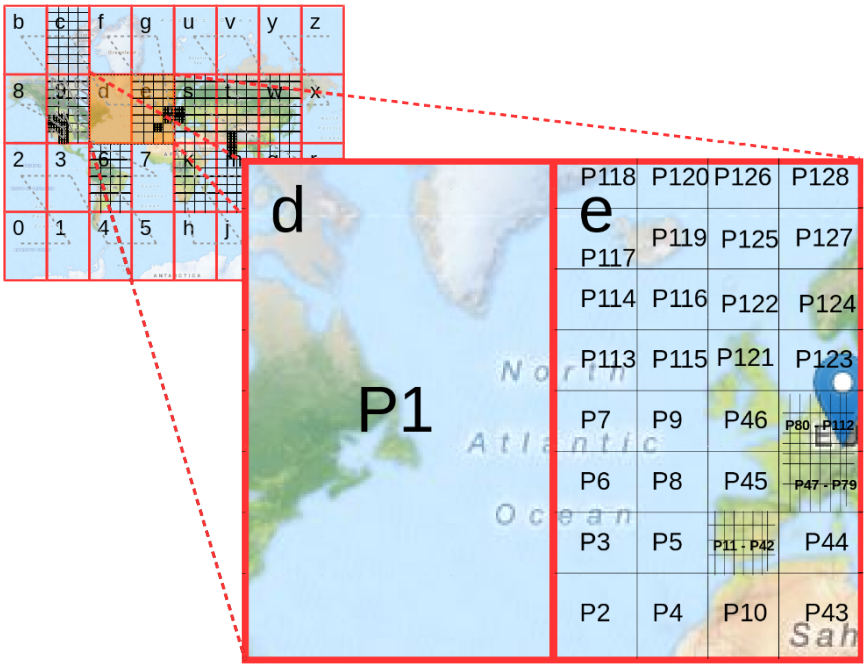


Figure 11: Enhanced optimization in creating Geohash Partitioning

So, we further enhance the algorithm to combine neighboring Geohash prefixes that could stay well as in a single partitions as demonstrated in the Figure 13.

Persistence after Custom Partitioning

Once a huge data set is partitioned based on any custom partitioner, it is necessary to persist the resulting RDD, either in memory or disk. If the RDD is not persisted, the subsequent usage of this RDD would result in repeated custom partitioning and involve reevaluation of the RDD's complete lineage. Hence failure to persist this RDD would nullify the benefits of `partitionBy()`, by incurring repeated partitioning and shuffling of records across the cluster. This would have been the case of not using any partitioner. [2]

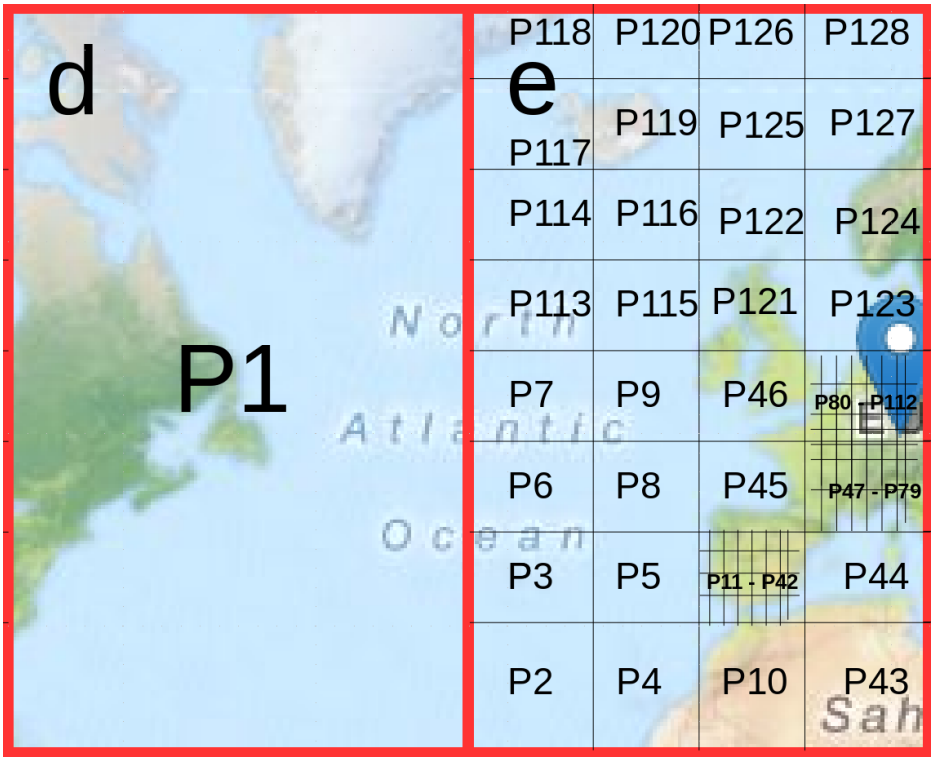


Figure 12: Geohash Partition 'e' is further partitioned to deeper resolution(has only 3 hotspot partitions)

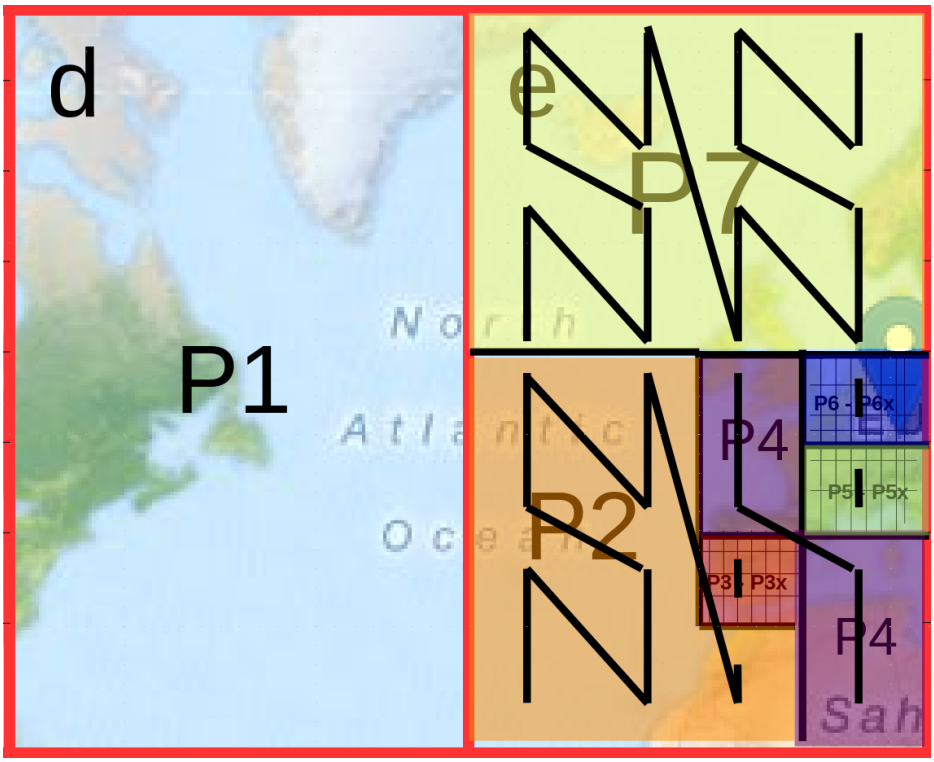


Figure 13: Previously under-fed 28 partitions combined and reduced to 3 joint partitions

C.4 Architecture Overview

The below architecture diagram gives an overview of all the components in Geohash based load aware partitioner

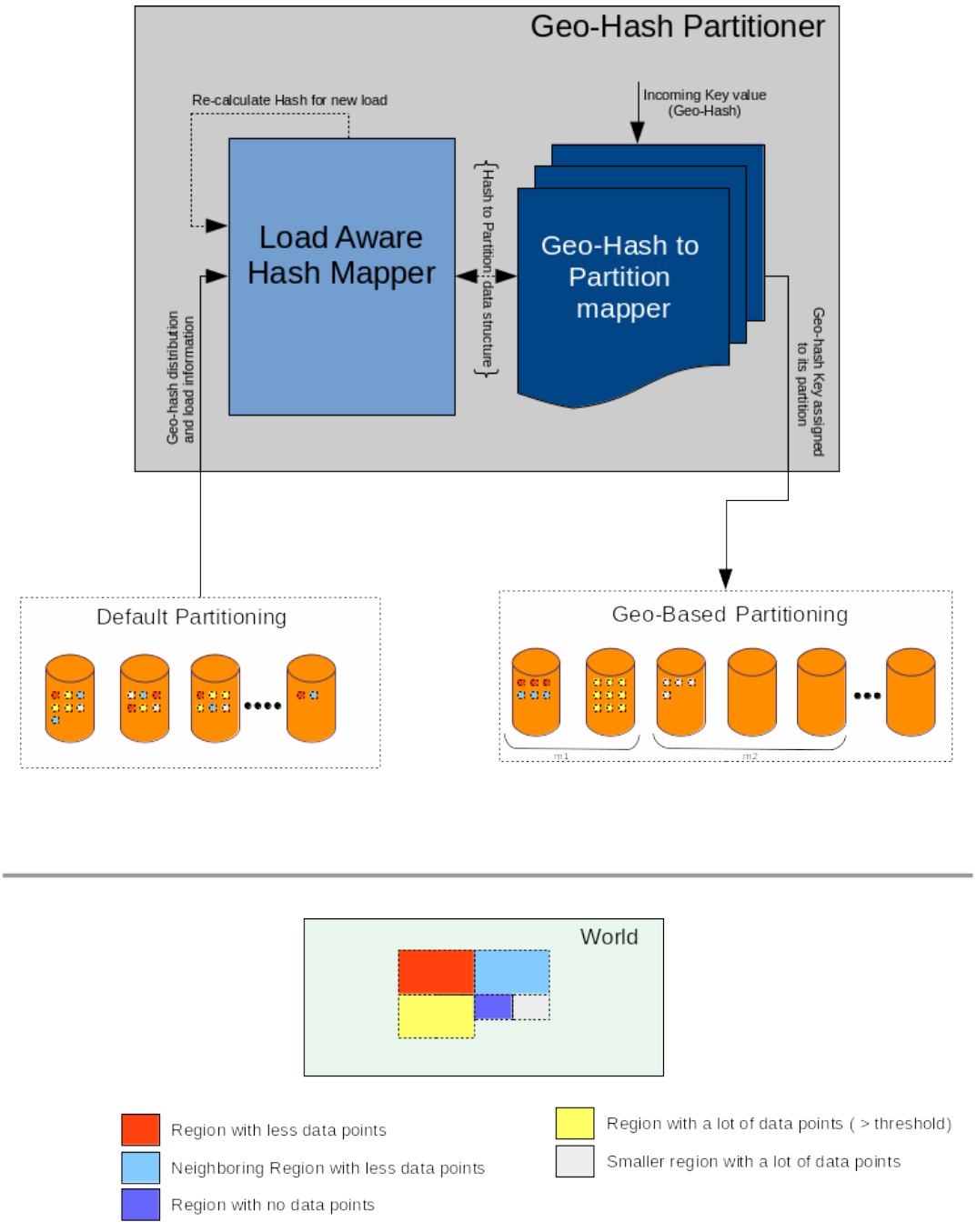


Figure 14: Architecture Overview: Geohash Partitioner

D GeoHash Query Optimizer: Design and Implementation

In this chapter, we look into the design and implementation of the Query Layer that leverages knowledge from Geohash Partitioning to perform efficient querying

D.1 Partition Pruning

After running Geohash based partitioning of our dataset, subsequent operations on the resultant RDD would result in reduced network communication. Moreover, the application has a clear knowledge on the partitioning logic - how the dataset is partitioned and in which specific partition the data for a particular region resides.

The general behavior of spark is that whenever there is an incoming tasks, it is launched on all partitions. For instance, when there is filter to extract all the restaurants in Chicago, the filtering task in launched on all the partitions even on the ones which holds records only from Australia. Now that the application has control on partitions based on the custom Geohash partitioner, we could leverage this knowledge to affect only those partitions which are of our concern.

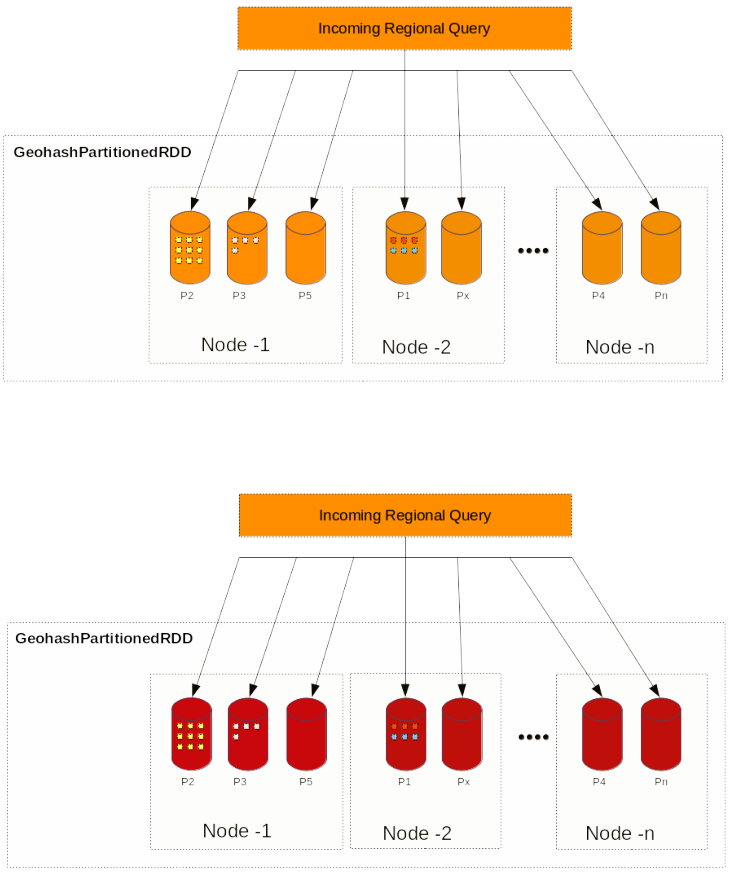


Figure 15: Spark's default behavior to launch tasks in all partitions

Spark provides an api, **PartitionPruningRDD<T>**, which is an RDD used to prune existing RDD partitions in order to prevent launching tasks on all partitions [8]. This helps us to launch tasks only on partitions that are of concern for the filter operation in the execution DAG.

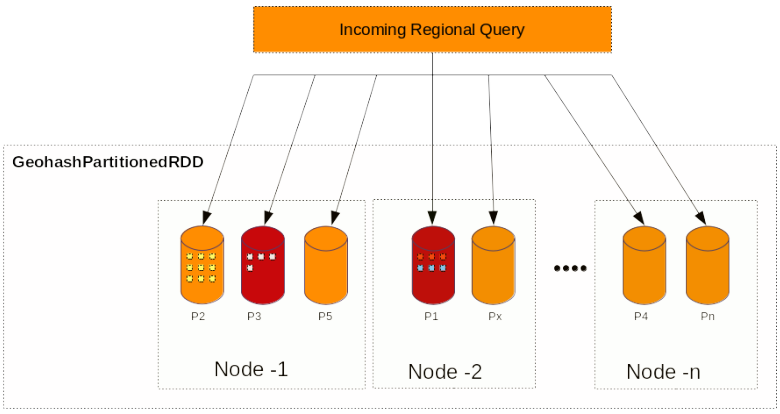


Figure 16: Spark Partition Pruning RDDs launching tasks on selected partitions

Our approach builds a Query Translator to translate incoming queries into geographic region and in turn use these regions to pick potential candidate partitions, based on the information from Geohash Partition mapper, previously built for partitioning. We then build a query optimizer module which prunes the candidate partitions and launches tasks on these smaller subset of partitions.

D.2 Geohash Query Translator

The purpose of this module is to analyze the incoming task or query and mark down the regions of interest for that particular query. In the following section, we shall take a look into how the translation works when we have a spatial '***Within***' query (say, show me all the restaurants *150 meters* from *here*)

- This type of query provides us with the point of interest*(p)* and the desired distance*(d)*.
- In our approach, we make use of spatial distance calculation function like *Haversine formula* [9] (or) *Vincenty's formulae* [10] in our distance calculation between any two points on the earth.
- From the point of interest, calculate a circle(c) with a radius of the d. This query circle(c) is created by all points that are a distance d from the point of interest(p).
- Using this query circle, a minimum bounding box that completely covers circle of interest is calculated. The mathematical formula for calculating bounding box coordinates are derived from [11] and clearly explained in the work [12]

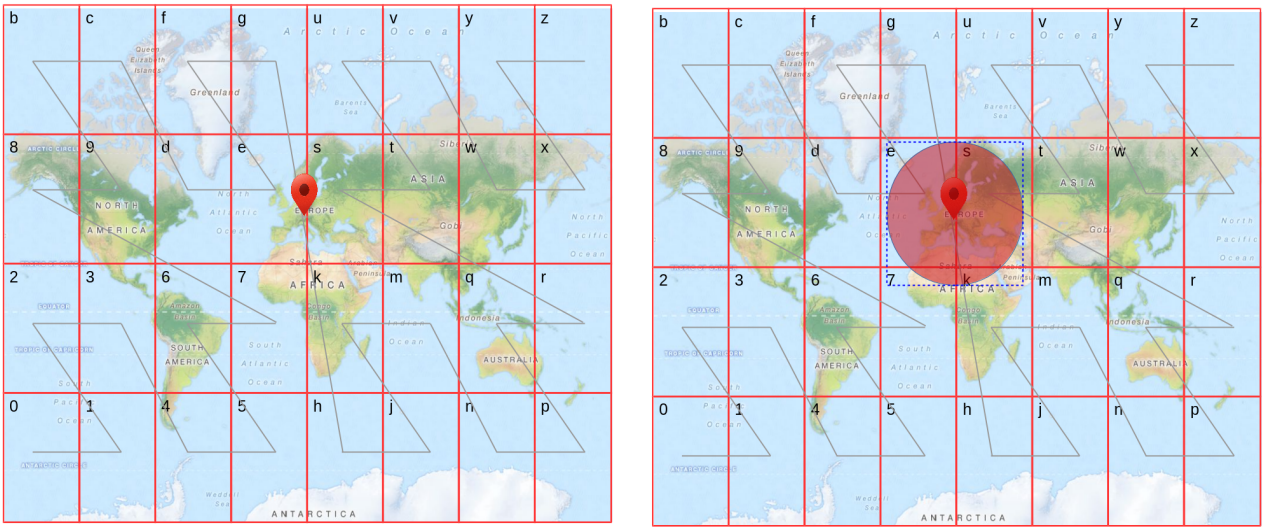


Figure 17: Query translated into Bounding Box for region of interest

Now query has been translated into a bounding box and this query bounding box contains all the data points that are of interest to us. The coordinates of this derived bounding box is fed as input by the next module, ***Geohash Query Optimizer***, to pick candidate partitions to run the tasks on.

D.3 Geohash Query Optimizer

Identifying partitions corresponding to Query Bounding Box, pruning them from the Geohash partitioned RDD and launching the tasks on them is the purpose of Geohash Query Optimizer. The bounding box is evaluated against the Geohash Partition mapper to determine the list of partitions. The details of this procedure as listed below:

- The query bounding box has four coordinates and these are encoded into their respective Geohash string.
- The longest common prefix is calculated from the four coordinates' Geohash. This prefix covers all the Geohashes of the data points that are required for this query.

NorthWest:	**dr5r**uj4477kd
SouthWest:	**dr5ru**46ne2ux
SouthEast:	**dr5ru**6ryw0cp
NorthEast:	**dr5ru**mpfq534
Query Bounding Box Prefix:	***dr5ru***

Table 1: Query Bounding Box Prefix

- The prefix is evaluated against the key values in Geohash Partition mapper and all the matching partition numbers are retrieved as depicted in Figure 19

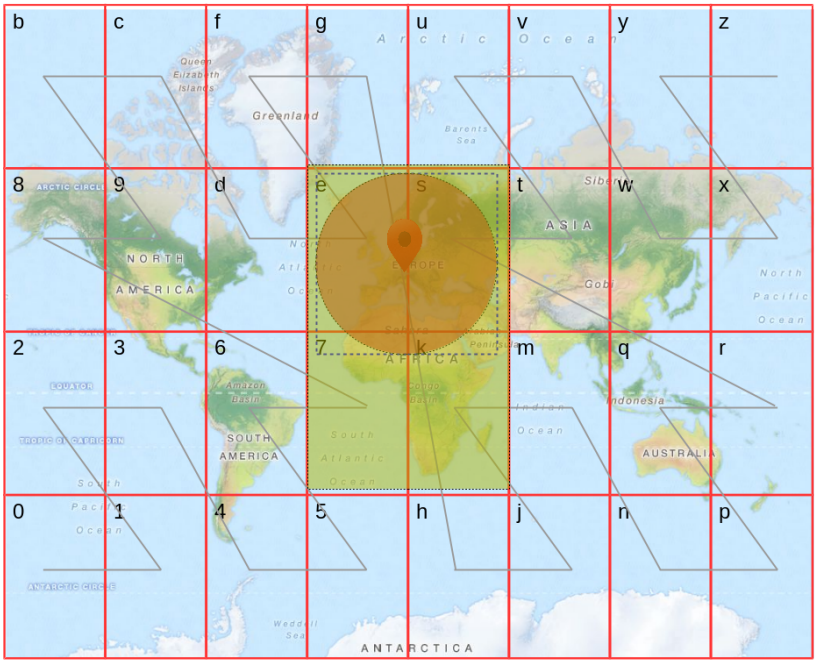


Figure 18: Translating Query into Regions of interest(Bounding Box)

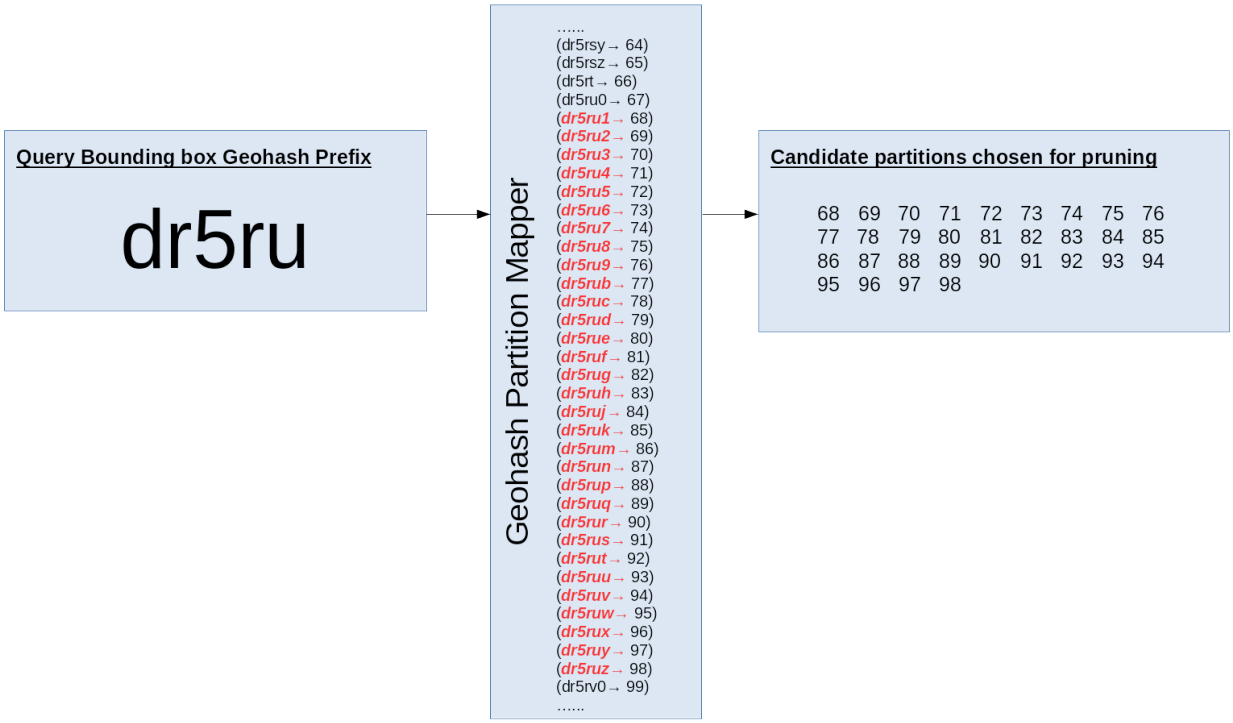


Figure 19: Choosing partitions to prune based on Query Bounding Box prefix

D.3.1 Enhanced Optimization

This candidate partition section procedure could be further optimized by efficiently ignoring partitions that are within the query bounding box prefix but out of Query circle. This could be achieved by taking advantage of the knowledge about varied levels of Geohash grid resolution and dropping out partitions that are not needed for the query as shown in the Figure 20. As we could see, only *4 out of 32* inner partitions of grid '***k***' are of interest for this query and so the remaining 28 partitions could be easily dropped from consideration. Bigger grids, grid '***7***', with no further inner partitions has to be considered as whole.

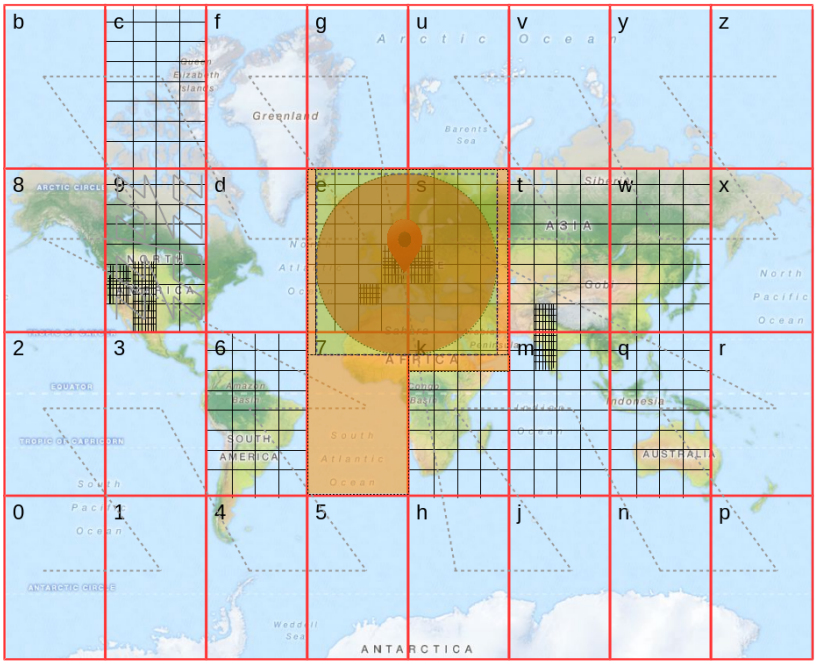


Figure 20: Enhanced candidate partitions selection based on varied grid resolution information

Once the candidate partitions are identified by using either of these methods, they are pruned out from the large partitioned RDD and task is launched on the resulting, pruned RDD as shown in Figure 16. This results in a much lesser workload and reduced scans needed to obtain the same results and in turn enhances performance. The efficiency of *Geohash Partitioning and Query optimizing* approach could be evidently noticed from the evaluation results. The following section shows the results of evaluation for the test cases run on New York Taxi Data.

D.4 Architecture Overview

The below architecture shows the complete components of the Geohash based Query optimizer

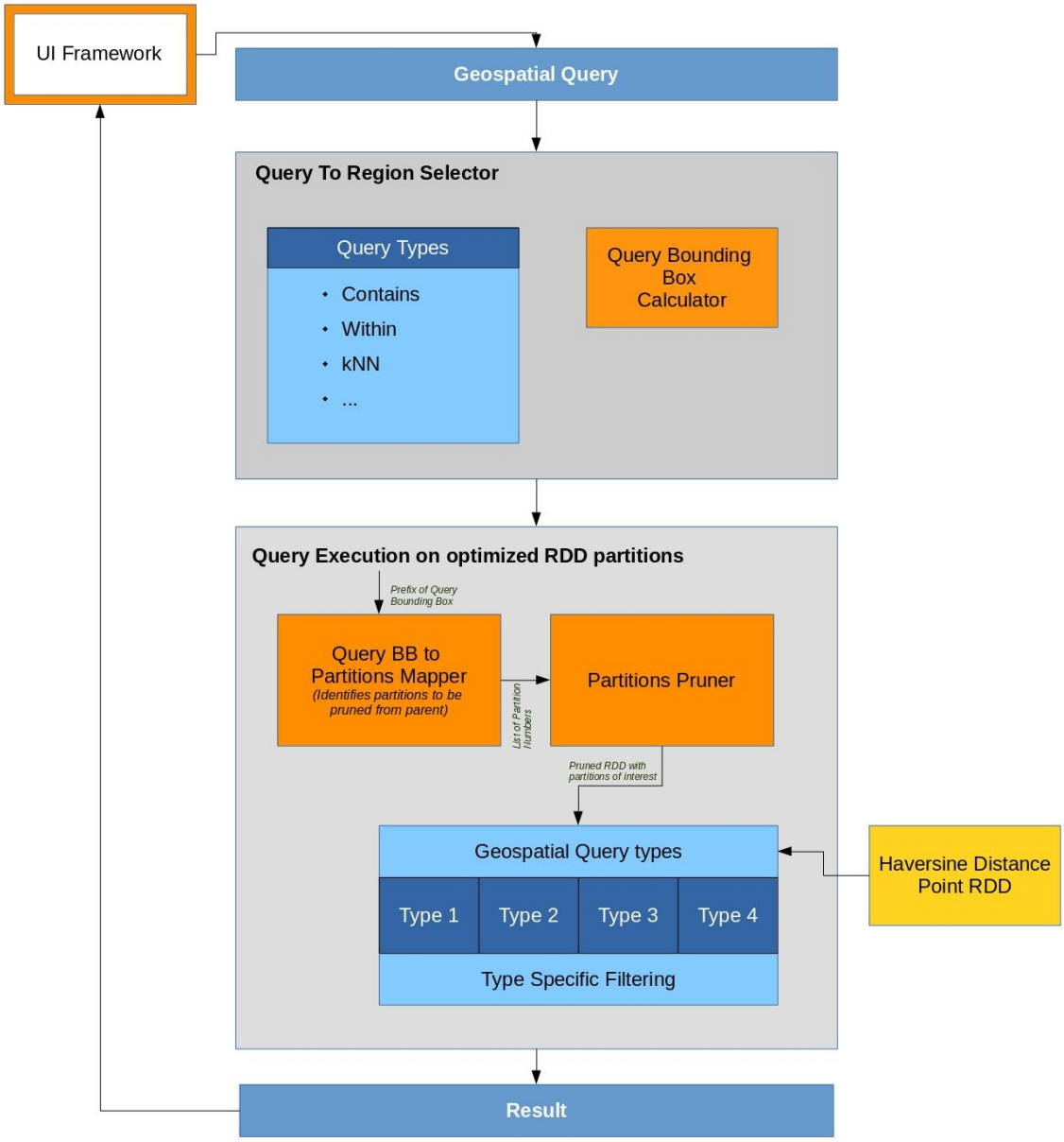


Figure 21: Architecture Overview: Geohash Query Optimizer

E Evaluation and Results

UNDER PROOF READING

E.1 System Footprint

E.2 Results

E.3 Limitations

F Conclusions

UNDER PROOF READING

F.1 Contributions

F.2 Future Work

References

[1] "Apache Spark." http://spark.apache.org/. Accessed: 2016-05-30.

[2] Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis*. " O'Reilly Media, Inc.", 2015.

[3] "JavaPairRDD: Apache Spark - org.apache.spark.api.java." http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.api.java.JavaPairRDD. Accessed: 2016-05-30.

[4] "Partitioner: Apache Spark- org.apache.spark.Partitioner." http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.Partitioner. Accessed: 2016-05-30.

[5] Heather Miller, "Shuffling, partitioning, and closures - parallel programming and data analysis." http://heather.miller.am/teaching/cs212/slides/week20.pdf, 03 2015. Accessed: 2016-05-30.

[6] Aaron Davidson, Andrew Or, "Optimizing shuffle performance in spark," UC Berkeley 2013.

[7] "Configuration - Spark 1.6.1 Documentation." http://spark.apache.org/docs/latest/configuration.html#scheduling. Accessed: 2016-05-30.

[8] "PartitionPruningRDD: Apache Spark." https://spark.apache.org/docs/latest/api/java/org/apache/spark/rdd/PartitionPruningRDD.html. Accessed: 2016-05-30.

[9] Wikipedia, "Haversine formula — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Haversine%20formula&oldid=719099044, 2016. Accessed 30-May-2016.

[10] Wikipedia, "Vincenty's formulae — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Vincenty's%20formulae&oldid=717944025, 2016. Accessed 30-May-2016.

[11] I.N. Bronshtein , K.A. Semendyayev, Gerhard Musiol, Heiner Mühlig, *Handbook of Mathematics*. Springer-Verlag Berlin Heidelberg, sixth ed., 2015. eBook ISBN: 978-3-662-46221-8.

[12] Jan Philip Matuschek, "Finding points within a distance of a latitude/longitude using bounding coordinates." http://janmatuschek.de/LatitudeLongitudeBoundingCoordinates#Longitude. Accessed 30-May-2016.