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

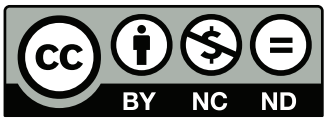
Databases and Distributed Systems
Department of Computer Science
Technical University of Darmstadt

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 5th July 2016

_____

(Hariharan Gandhi)

Acknowledgment

“Thanks to Gregor, Raghu, Robert, Professor Buchmann, Harini and friends! for playing a major role in supporting my Master studies”

-Hari

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[1] location sensors, emergence of social media and advancement in weather and navigation systems. More and more applications and businesses are providing location based services and personalised suggestions. Some of the 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' geographic 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** [1], to process geospatial datasets. Spark provides *resilient distributed dataset* (RDD) [2], which is a collection of dataset partitioned across the nodes of a cluster for parallel operation [3].

geospatial operations such as *contains, within, overlaps* etc involve 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 improves 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. [4]

[1] Global Positioning System

Contents

List of Figures

A Introduction

A.1 Motivation

Of late, geospatial data has seen an unprecedented growth rate owing to the increasing popularity of mobile devices. According to GSMA report 2016 [11], there were **7.6 billion** mobile subscriptions with 4.7 'unique' subscribers by the end of 2015, one billion more expected by 2020 and more interestingly, more than half of the world's population has mobile subscriptions with a penetration rate of more than 63% and operator revenue of one trillion dollars.

These devices are equipped with sophisticated *Global Positioning System(GPS)* which reports the exact location information of the device. This feature has made lives easier with tons of day to day applications providing *location based services*. Some of these apps, such as maps, navigation, social networking, have become integral part of our lives. The major source of data for these applications originates from end-users through opportunistic or participatory sensing. Besides mobile devices, almost every web application have also centralized their services to location based. Twitter trends in a particular country, traffic information for a highway route, weather forecast in a location/city - all involve spatial data associated with them.

With all these sudden burst in location data, comes the challenge of scalability and efficiency in processing these streams of data, and need for techniques that could efficiently process spatial queries. Big data processing and management has been in practice for years now and are providing solutions to most of the world's big data needs. Several distributed file systems - *Hadoop Distributed File System(HDFS), Google File System(GFS)* and distributed processing engines and databases like *Big Table, Hadoop, Spark* etc are in place to crunch the big data thrown at them. So, to handle our *big-spatial-data*, our obvious choice should be to use one of these distributed processing platforms. There are several spatial indexing techniques in spatial databases like

- Quadtrees,
- Grid files,
- R-trees

that are being used extensively to store and retrieve spatial data types in traditional Relational Database Management Systems(RDBMS). Several such approaches have also been proposed to use in big spatial data processing platforms like Hadoop MapReduce. Nonetheless, almost all of these approaches, demand certain level of modification to the internal implementation of their indexing schemes in these frameworks, hence leading to additional complexity. This would also result in system specific approaches [12] [13].

There are also huge possibilities to scale these data processing clusters as they are increasingly available and accessible via Cloud Computing. Besides all these available big data processing engine, there is no simple solution to efficiently handle spatial data in these parallel platforms without modifying their internals. Also, their working cannot be optimized as long as *location awareness* is introduced into these systems and they are extended to handle spatial data. One major challenge in such cluster based engines is to minimize shuffling of data between nodes of a cluster thereby reducing the expense of network cost, minimizing the number of parallel tasks launched and processed, efficient and quicker query processing by minimizing the number of partitions affected and number of records scanned. All these challenges could be solved by providing an efficient Partitioning algorithm that lays our spatial data into the nodes of a cluster such that spatial locality is retained within those partition. This would greatly help minimize shuffling of data point by performing partition local processing.

Currently there are no such partitioning algorithms in these parallel processing platforms that makes *location-aware partitioning* of very large spatial dataset. We, in our work, propose *(i)* a **custom Partitioning algorithm for Apache Spark**, that provides a lightweight spatial partitioning extension to spark without needing to modify any of spark's internal implementation. We study working of Spark, transformations affecting shuffling of data within nodes, design and implement our proposal to validate its effect. The proposed algorithm also load balances based on data distribution in the real world thereby preventing overloading of certain nodes over other. The ultimate goal of this partitioning algorithm on spark, is to witness an increase spatial query processing efficiency. *(ii)* We build a query translation layer that converts query in 2-D space to target our partitions and chooses candidate partitions for applying the query. The mismatching major portions of dataset are conveniently omitted from being considered for query processing. Our approach proposes optimizations at several stages to choose the right and most minimal set of partitions that would be needed to processing the query. This *query translation, preprocessing and optimization layer* leverages the knowledge of previous custom spatial partitioning we created.

A.2 Proposed Approach in this Thesis

We propose a custom spatial partitioning scheme, implement and test it with Apache Spark(in general, the algorithm aims to be used with any distributed partitioning scheme). It aims to provide an efficient spatial partitioning to overcome the problem of large size shuffling and reduced performance while handling geospatial data. Our approach based on ***Geohash encoding***, that encodes location points based on ***Z-order***, does not require any internal modification to existing implementation of Apache Spark. We extend sparks '$\mathrm{Partitioner}()$' class to provide a thin layer of custom partitioner that introduces location awareness to spark partitioner. This facilitates future geospatial queries on these partitioned RDD to be effective and substantially faster. The working of these techniques are discussed in detail by the forthcoming chapters.

We also propose and implement a query handling layer that leverages the knowledge of our spark spatial partitioning scheme and drastically reduces the number of parallel tasks launched on Spark nodes that do not contain data of our interest, minimizes number of records scanned, and eventually improves response time for queries. We also discuss about the observations from evaluation, limitations and challenges while using this approach. Our work mainly focus on *Point* spatial data type, which contains single latitude and longitude. However the approach could easily be extended to support complex spatial type without much effort.

A.3 Dissertation Road map

Following this introduction(**Chapter A**), where the motivation to work on this thesis and existing problems in that area are discussed, the rest of the dissertation is organized as follows:

Chapter B: explains background knowledge that would help to under existing principles, architecture of systems in use, available and previous works in the relevant areas or works that could you used to support our proposal.

Chapter C: proposes a new algorithm for distributed spatial partitioning. We then design and build the components of a custom partitioner for Spark that uses Geohashing technique to perform load aware partitioning on geospatial dataset.

Chapter D: proposes an effective query translation and optimization layer on Spark. We then demonstrate its the design and implementation which leverages the knowledge of partitioning scheme to minimize scans on RDD partitions.

Chapter E: evaluates the implementation of our proposed system on a multi-node Spark cluster running on Hadoop Distributed File System(HDFS) using New York Taxi dataset, to profile the performance and observations. We are throw light into limitations of our approach, best practices for efficiency and challenges that had to be tackled during the design and implementation of our approach.

Chapter F: concludes the dissertation with new enhanced ideas for future extensions.

B Background and Related Work

B.1 Geospatial data

In addition to the normal fields, geospatial data contain associated location information. Any point on Earth surface is represented, generally, using Geographic coordinate system with three common coordinate - Latitude, Longitude and Elevation. Map projections are then used to used to transform these coordinates from spherical to plane representation.

Approximate location coordinates of Mount Everest: 27.988056, 86.925278 [14]

Location has become an important property of any file and event. For events occurring, the information about the location of the event is added in the form of coordinates.

- photo taken at location
- route to a particular place on maps
- bad weather in a particular region

These coordinate information for any point on earth is fetched using Global Positioning Systems(GPS) [15]. With advancement in GPS enabled mobile devices and increased number of applications providing location based services, geospatial data have become inevitable part of data we gather. Techniques to handle and manage geospatial data types within databases have been in study and use in several spatial databases for quite a while. There are several representations for geometry, for example, Well-Known-Text is one of those formats and it can represent several distinct geometry objects.

Geometry object could be: [5]

Simple:

- Point
- LineString
- CircularString
- CompoundCurve
- Polygon
- CurvePolygon

(or) Collections:

- MultiPoint
- MultiLineString
- MultiPolygon
- GeometryCollection

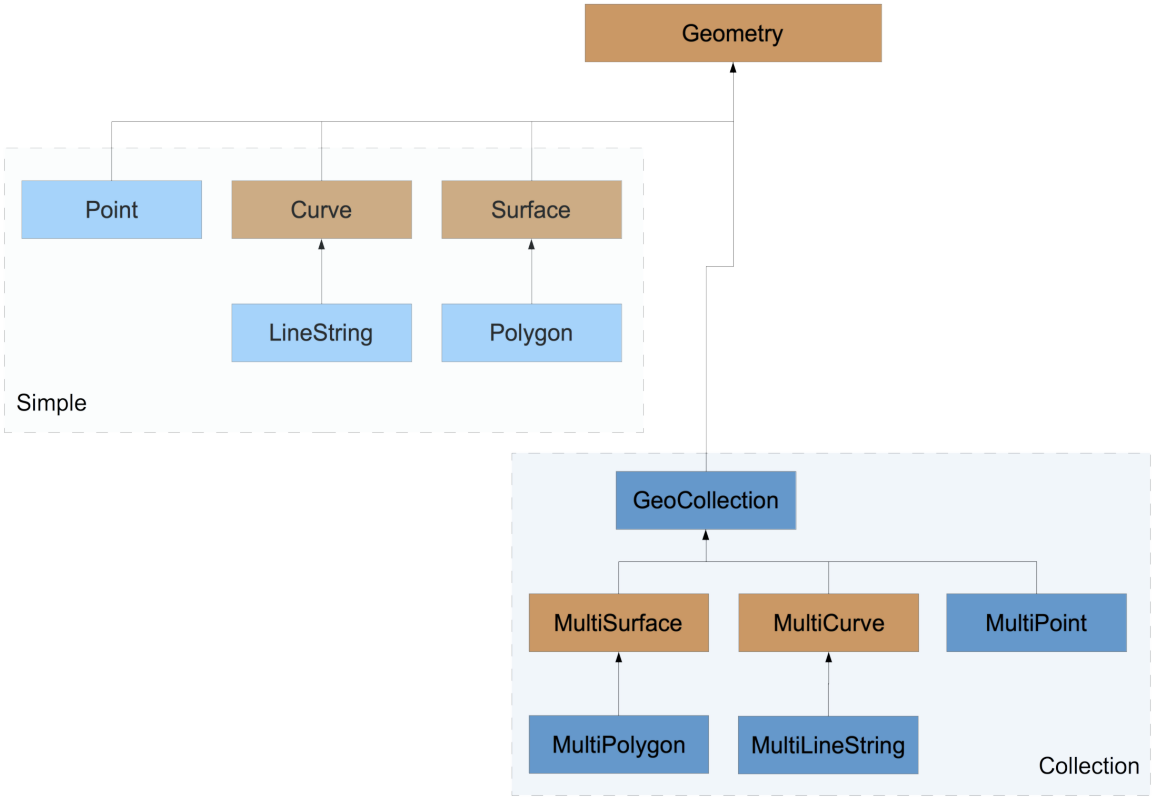


Figure 1: Various Spatial data types. Image inspired from [5]

Distributed computing platforms are gaining great importance these days w.r.t., spatial data processing. However there is limited or almost no support in distributed computing platforms like Apache Spark to efficiently handle geospatial data. This dissertation tries to solve one such problem involved with efficient handling of geospatial data without much internal modification to existing parallel platforms.

B.2 Apache Spark

The open source, large-scale data processing, cluster computing framework, Apache Spark [1], was developed at the University of California, Berkeley and is currently maintained by Apache Software Foundation [3] . Spark provides APIs for programming clusters with fault-tolerance, scheduling, and data parallelism [16]. Spark is designed mainly for applications that cannot be expressed efficiently as acyclic data flows and for those that reuse a working set of data across multiple parallel operation.

In short, Spark is most suited for applications that require *'Iterative Jobs'* and *'Iterative Analysis'*, such as most of the machine learning algorithms where each of the repeated jobs had to reload data from disk. Earlier cluster computing frameworks like Hadoop, would result in performance penalty in such a scenario. Spark addresses this issue by employing a new abstraction called *Resilient Distributed Dataset(RDD)* [1]

Resilient Distributed Dataset(RDD):

A resilient distributed dataset is an immutable collection of elements partitioned across a set of nodes in a cluster and could be rebuilt in case of partition is lost or node failure. Instead of existing in physical storage, RDDs exists as metadata that contain enough information to rebuild RDD from a reliable storage. [1].

RDDs support two kinds of operations:

1. **Transformations**: these are operations on RDDs that return a new RDD(computed lazily)

2. **Actions**: these are the actual operations that return some computed final value to the spark driver program [4]

Spark's building an RDD is lazy and ephemeral. This implies that when there is call for a transformation(say, Map operation) on an RDD, the task is not immediately performed. They are built only during an Action and the intermediate Transformation are just maintained as RDDs Directed Acyclic Graph (DAG) information. RDDs that needs to be reused, could be persisted in-memory or on disk(caching is also lazy and runs only on an action on RDD). By default, persisting an RDD would store them in memory; if they do not fit in memory then the remaining partitions of RDD are always re-computed on the fly and replaced in LRU fashion in the memory. It could also be configured to spill the computed RDD partitions, that do not fit in memory, into disks.

RDD's elements could be partitioned across machines based on each record's key. Partitioning is useful for locality optimizations. Two RDDs that would be joined are hash-partitioned in the same way. [2]. Spark provides applications control over partitioning and persisting of its RDDs. However, several operations, for example 'Joining of two RDDs' are supported only for Key-Value RDDs. During operations like groupByKey, reduceByKey etc, the resultant RDDs are either hash partitioned or range partitioned automatically.

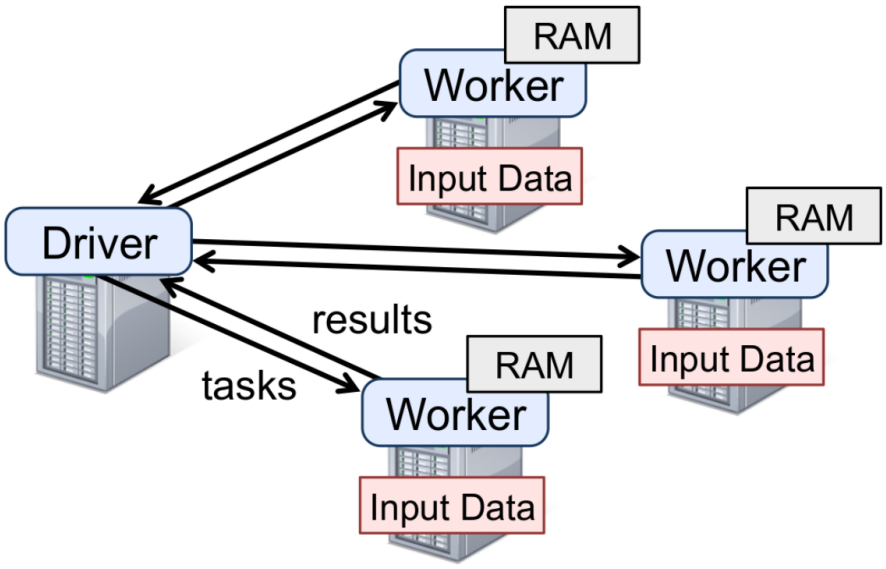


Figure 2: Spark Cluster: Workers read from a distributed file system and can persist RDD in memory. Image source [2]

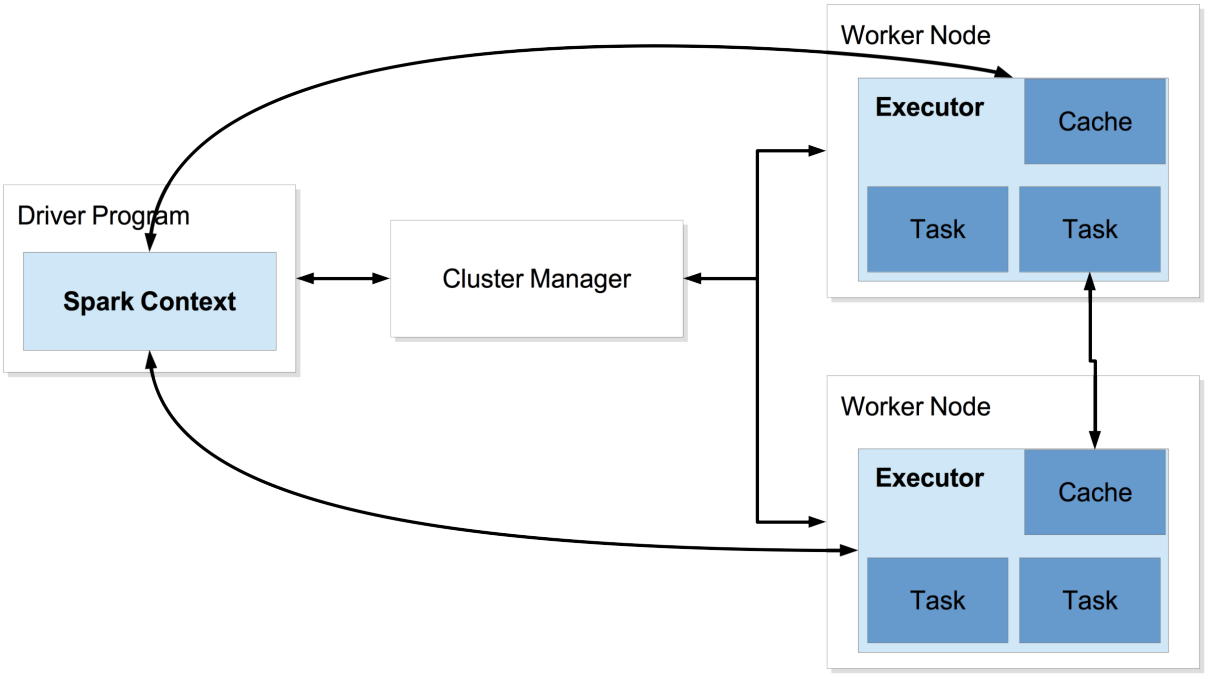


Figure 3: Components of Spark Cluster: Overview. Image redrawn from [6]

Impact of Data Locality on Spark performance

The location of data has a great impact on execution performance. Spark tends to export code to data's location to execute on them. When the executing code and its target data reside closer then the job execution is faster.

Proximity to data to its processing code is categorized by Spark Documentation [17] as follows:(in the order of best to worst)

- PROCESS_LOCAL - data is located in the same process(JVM). Best locality possible
- NODE_LOCAL data is on the same node. Examples might be in HDFS on the same node, or in another executor on the same node. This is a little slower than PROCESS_LOCAL because the data has to travel between processes
- NO_PREF - data locality without any preference. All access to data are considered equal.
- RACK_LOCAL - data is located on the another server which resides in the same rack.
- ANY - data is not located in the same rack and is available elsewhere on the network.

Need for (Spatial)Partitioning: Shuffling

In a distributing computing platform, like Spark, one of the main concerns is to minimise the network traffic between clusters. With Spark, it is very crucial to partition, Pair RDD [18] data sets(key-value) since subsequent transformations on these PairRDDs involve data shuffling across the nodes in the cluster. When data set is partitioned and similar keys reside in the same partition(locality is maintained, in spatial terms), then transformations in these partitioned dataset results in minimised shuffling and substantially cheaper and faster processing.

Partitioning is greatly beneficial to several transformations that involve data shuffling across cluster nodes. Pair RDD functions [19] such as ***groupByKey, reduceByKey,combineByKey, cogroup, groupWith, join, leftOuterJoin, rightOuterJoin and lookup()*** are the transformations on Key-Value RDDs that benefit from partitioning. However, partitioning is not always effective. In most cases, it is effective when there is multiple reads on the partitioned data. [4].

Spark provides two default partitioning implementation, namely Hashing Partitioning and Range Partitioning. User could also extend the Partitioner class to create custom Partitioning scheme.

B.3 Spatial Indexing techniques

Spatial data handling has been in study and use for several years. This section lists down few of the spatial indexing techniques like R-Trees, Quad Trees, Grid files and their properties are summarized in Table 1

R-Tree: is similar to B-Tree with dynamic index where the search key are minimum bounding rectangles(MBR) in multidimensional. R-trees are extensively used in spatial databases and are efficient in processing Range Queries and kNN Queries.

Quadtree: is a hierarchical tree structure for indexing spatial points in 2-D space. Quad trees divides each region is subspaces recursively until a certain end criterion is met, like minimum number of points in a regions. Several databases including Oracle employ this in their spatial extensions.

Grid(Spatial index): Grids are shapes that divide 2-D surfaces into continuous cells each of which is assigned with a unique identifier used for spatial indexing [20, 21]. There are several types of Grids: Square/Rectangular, Triangular, hexagonal.

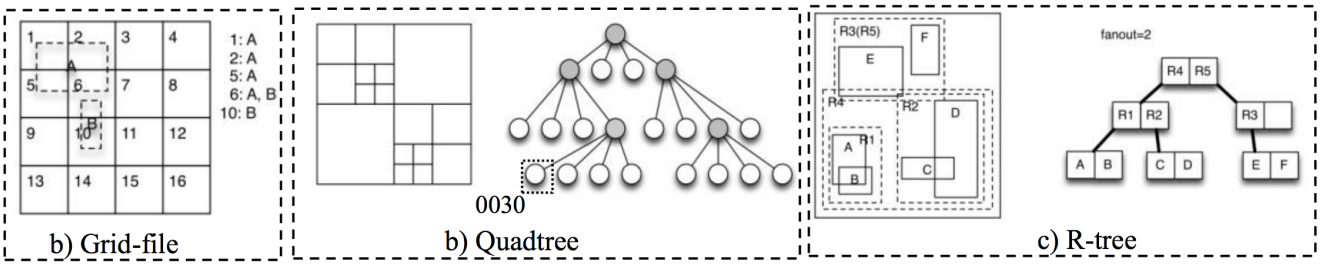


Figure 4: Various spatial indexing [7]

	R- Tree	Quad Tree	Grid files
Hierarchical Structure	Yes	Yes	No
Parallelization friendly	Poor	Medium	Good

Table 1: Properties of major spatial indexing scheme [7]

B.4 Space Filling Curves

Space Filling Curves helps in converting 2-Dimensional problems into single dimension.

First, recursively divide the initial or parent domain into four child sub-domains. Next, connect neighboring sub-domains with a space-filling curve.

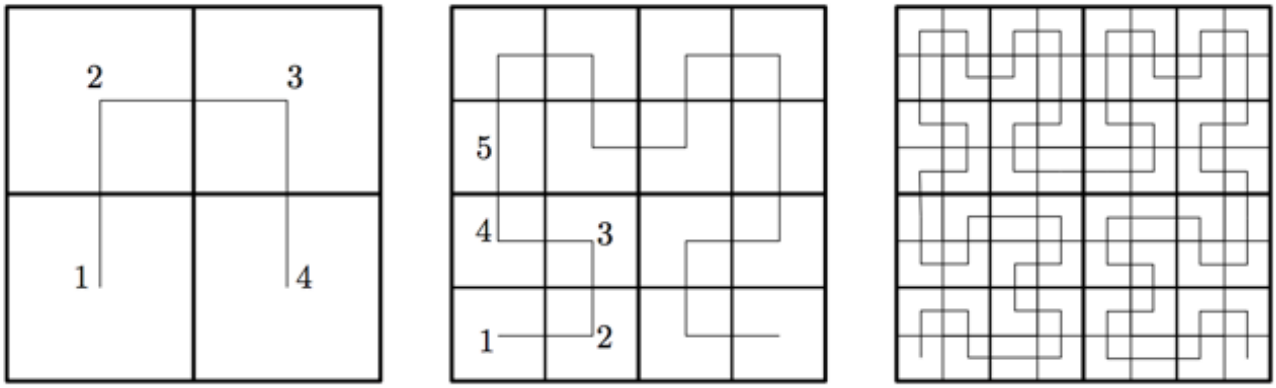


Figure 5: Space Filling Curve: Hilbert Curve of order 1, 2 and 3. Image source [8]

Like many of the multidimensional Query processing [22], we base our geospatial Query processing in Spark based on the characteristics of Space Filling Curves in Geohash. This helps in converting point query to be applied in the multi-dimensional space [22]. Additional benefit of applying Space Filling Curves, say Moore Curve [23,24], Hilbert Curve [25,26], is their recursive nature and it makes them more suitable for hierarchical indexing of Geo-spatial data [8].

The main benefit of applying Space Filling Curves on geospatial datatypes is that it does not require major changes to the existing internal data structure that were used to handle one dimensional non-geospatial data type. Once Space Filling Curves transforms Geo dimensions into single dimensions, it could be used with existing data structures, say B+ Trees, R-Tree etc [27]. For long Space Filling Curves are used for persistence and querying of multidimensional databases [28]. In modern databases, their application in geospatial data processing could be found in MongoDB [29], Solr [30], Oracle Spatial [31]

Z-order Curves

Z-curves was introduced by G. M. Morton as a function to map multidimensional data to single dimension while preserving the data points' locality. The plane traversal in this curve resembles a 'Z', see Fig:6, and hence the name. Data points' coordinates binary representation are interleaved to get it Z value. Once the z value is calculated into single dimension it could be used with existing data structure like B-Tree or Hash table. Z ordering could be used in hierarchical data structure similar to depth first traversal of a quad tree [9, 10]. We used the z ordered Geohash in combination with Hash table in our approach.

As we transform points according to Z-curve used in Geohash, they could be partitioned into clusters each representing a region. For any incoming query a sub-set of the clusters has to be retrieved and analysed to produce the result. This idea has adapted and used with Apache Spark to partition spatial data.

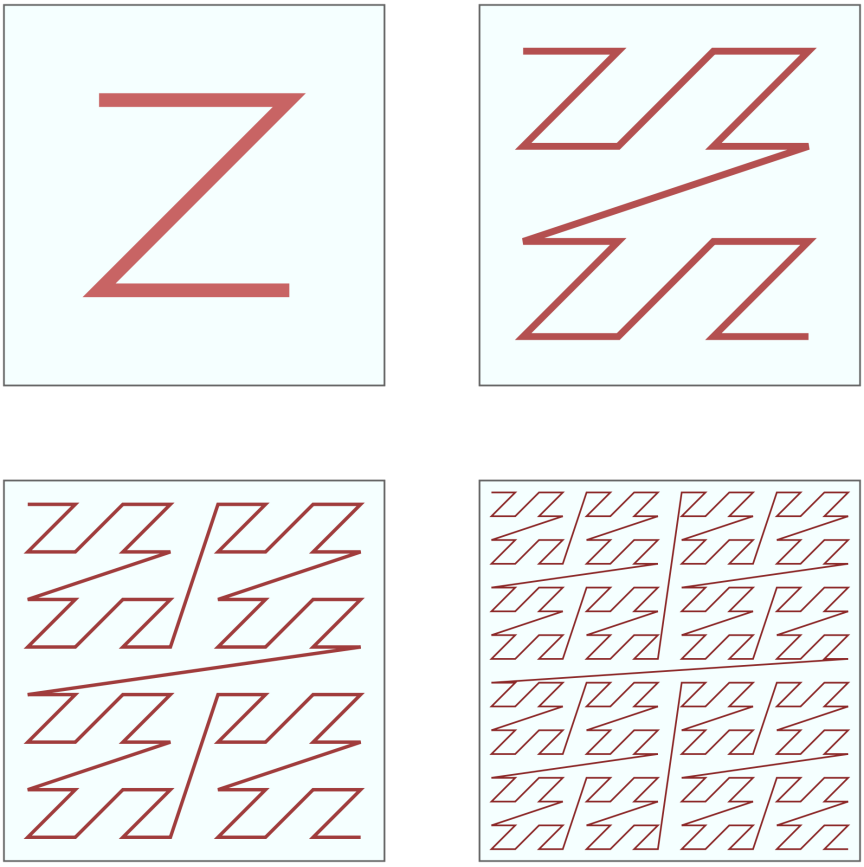


Figure 6: Space Filling Curve: Z Curve of order 1, 2, 3 and 4. Image source [9, 10]

B.5 Geohash

Geohash is a system for geocoding Latitude and Longitudes to represent spatial coordinates in a compact String or Binary representation. It was invented by Gustavo Niemeyer in 2008. [32] [12, 13] [33]

In our work, we use Geohash algorithms as hierarchical spatial data structure to partition geospatial big data based on the geographic region. Since the main focus of this work is partitioning, not indexing and we are dealing with points geometry Geohash is very useful. The algorithm controls data distribution in Spark by laying out data points to particular RDDs in the distributed clusters and also routing queries to specific RDD partition, allowing it to efficiently query few partitions.

Geohash as Hierarchical indexing structure: A Geohash code, in string format, represents a bounding rectangle grid on earth surface. Starting from resolution '1', Geohash string represents a grid on Earth surface

Geohash algorithm partitions Earth into a 8 X 4 = 32 grids at each level and traverses through the grids in z-order [9, 10] to produce the encoded Geohash string. At Geohash resolution '1', the length of the Geohash string is '1', denoting the z position of the grid.

The 32 grids in resolution '1' are further sub-divided into another 32 grids at Geohash resolution level '2' and making the two character long Geohash String. As the resolution of Geohash increases, the size of the grid(rectangular box) denoted by a Geohash String reduces making it more accurate in pointing to a particular place on earth. For instance, a Geohash of length '12' denotes a grid of approximately 7cm^2 near Equator.

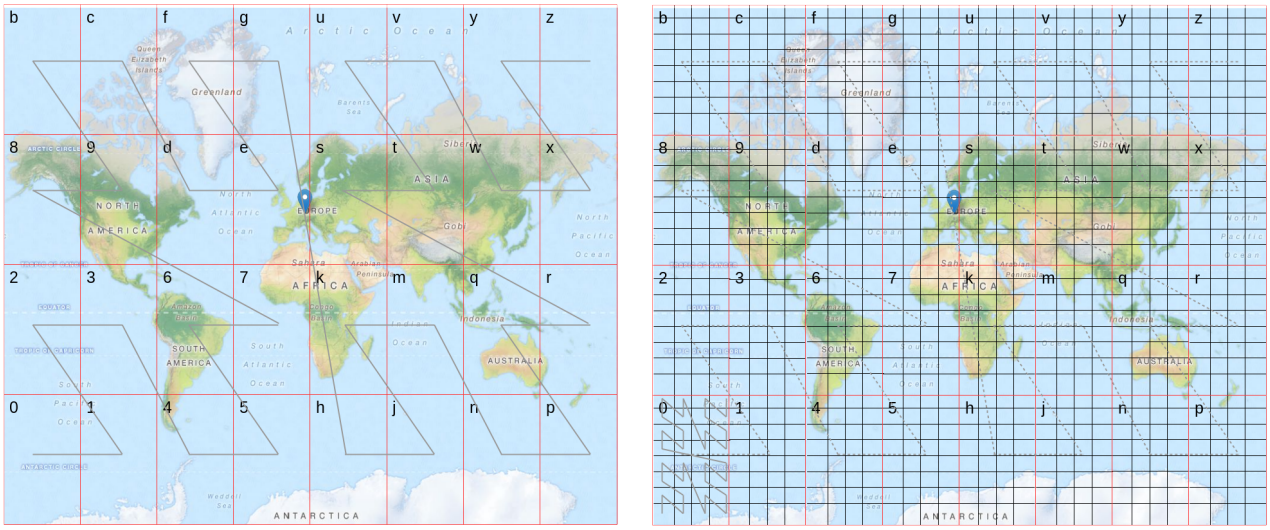


Figure 7: Geohash grid of resolution '1' and '2'

The below table 2 shows the dimensions of geohash cells for Geohash string lengths from 1 to 12. However, it is important to note that the dimensions of the cells varies from Equator to poles. [34]

GeoHash length	Area width x height
1	5,009.4km x 4,992.6km
2	1,252.3km x 624.1km
3	156.5km x 156km
4	39.1km x 19.5km
5	4.9km x 4.9km
6	1.2km x 609.4m
7	152.9m x 152.4m
8	38.2m x 19m
9	4.8m x 4.8m
10	1.2m x 59.5cm
11	14.9cm x 14.9cm
12	3.7cm x 1.9cm

Table 2: Geohash cell dimension for varied length of Geohash String(@Equator)

Properties of Geohash: : [35]

1. Geohashes could be used as Unique Identifiers and as a representation of Point data in databases [32]

2. As we remove characters from the right end of a Geohash string, it loses precision and covers a larger region.

3. Two Geohash strings with same prefix are closer to each other. However, the contrary is not true. Two Geohash strings with different prefix could be nearby too. This is true because of the longer Z curve distance between two grids in the Equatorial and Polar regions.

4. Exploiting the above property, Geohash index structure could be used for crude proximity search.

All these properties of Geohash encoding system are exploited in designing a geospatial Partitioning scheme in Apache Spark.

B.6 Related Work

Owing to the increased use cases for location aware data, many of the recent non-relational databases have focused on providing geospatial support by supporting spatial-indexing schemes. Geohash, out of the many schemes, is used as indexing scheme by MongoDB [29] and Apache Solr [30].

Geospark [36], one of the projects from DataSys Lab at Arizona State University, extends Apache Spark to provide geospatial support and claims to exhibit better performance in comparison to its hadoop based implementation, SpatialHadoop [37]. Geospark provides Spatial RDDs and Spatial operations on those RDDs. It allows to choose from R-Tree and Quad-Tree based indexing schemes.

The goal of our work is not to draw comparison with these systems. Our motto is to design a lightweight spatial partitioning scheme based on Geohash encoding technique, without much modification to internal implementation of Apache Spark. It is a high level design which is not restricted only to Spark and could be easily modified to use in any other distributed computing environment.

C Spatial Load Aware GeoHash Partitioner(GHP) for Spark

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:

- The dataset is scanned multiple times in key oriented operations like *joins*. [4]
- Considerable dataset size wherein benefits of partitioning outweighs the effort involved partitioning.
- And clearly when the application predominantly involves spatial queries than non-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 [38], since special distributed "*shuffle*" operations, such as grouping or aggregating the elements by a key.

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,

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

where $\mathrm{hasCode}()$ method is the java hashcode which returns an interger. $\mathrm{numPartitions}$ is the (desired) number of partitions we need the RDD to be partitioned into.

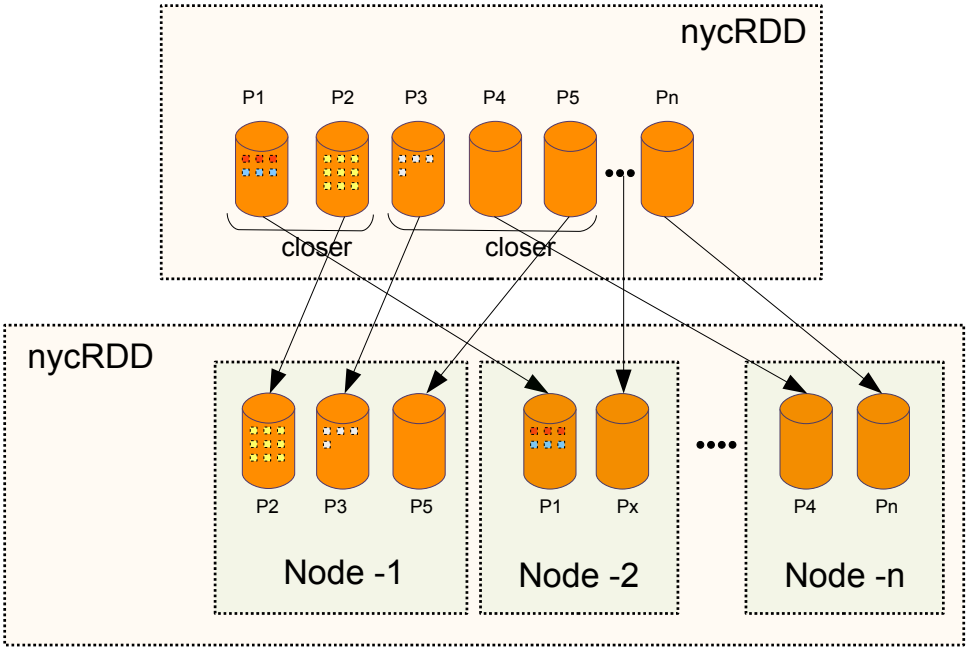


Figure 8: Hash key based Partitioning scenario in Spark

RangePartitioner: partitions sortable records by range into roughly equal ranges. The content of the RDD passed in is sampled to determine the range [39]. 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 could 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 knowledge(e.g., geospatial) to further cut down communication cost. Rest of this section 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. [4] However, it ensures that the set of keys resides in the same partition. For instance, keys that produce the 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. The properties of partitioning in Spark could be summarized as follows: [40]

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: [4]

- 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 B, 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.

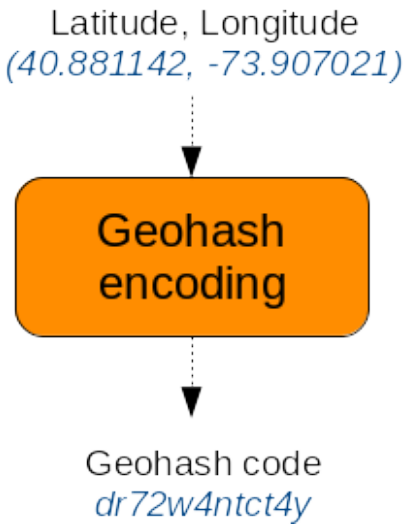


Figure 9: Geohash encoder to encode spatial points to Geohash Key

In our approach, we use Geohash of Latitude, Longitude of each record as its key using maximum resolution as seen in C.2 . We create a Key-Value RDD with the generated Geohash code as Key and the record itself as the value, as shown in table.

$$Geohash\ Key = GeoHashEncode.withCharacterPrecision(Lat, Long, 12^2).toBase32()$$

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

As discussed in the background section B of Geohash technique, this encoding scheme based on Z-curves, is used to preserve spatial locality. All data points with same Geohash prefix belong to the same geographical region. This encoding scheme also helps in identifying neighbour

[2] 12 - the maximum precision of 12 characters for 64bits Geohash

along the path of the z traversal. So, the next step is to 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 record's key and assign a partition number specific to that particular key. Since we need to create spatial awareness to partition allocation we cannot utilize the Java $\mathrm{hashcode}()$ 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 in representing grids on the earth surface, to create the partitions for RDD. 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 11

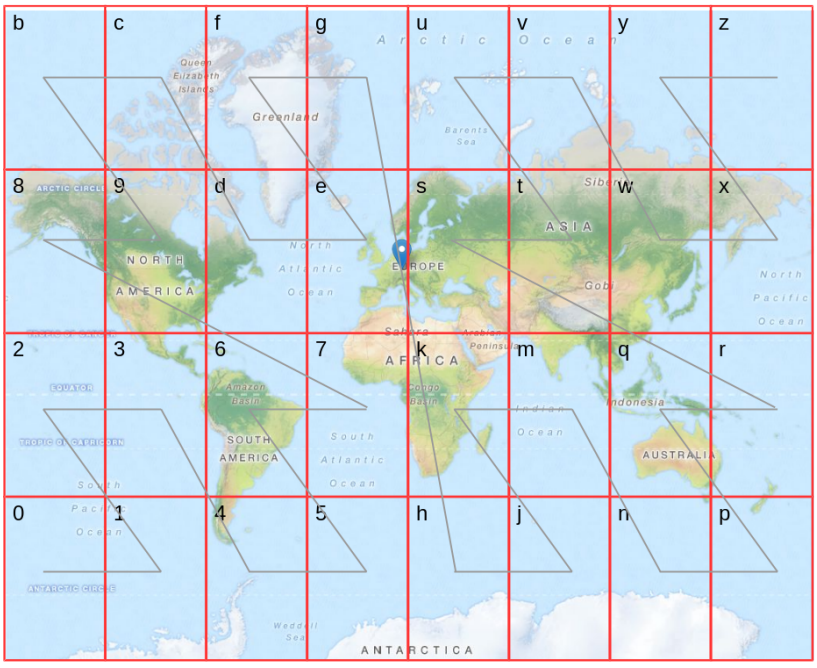


Figure 11: Geohash grid of resolution '1'

To proceed further deeper and to uniquely denote smaller region as a partition, we could simply increase the length 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 12. 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 Figure 13. This imbalance appears in spatial datasets with data point concentrated at hotspots, major cities and metropolitan areas.

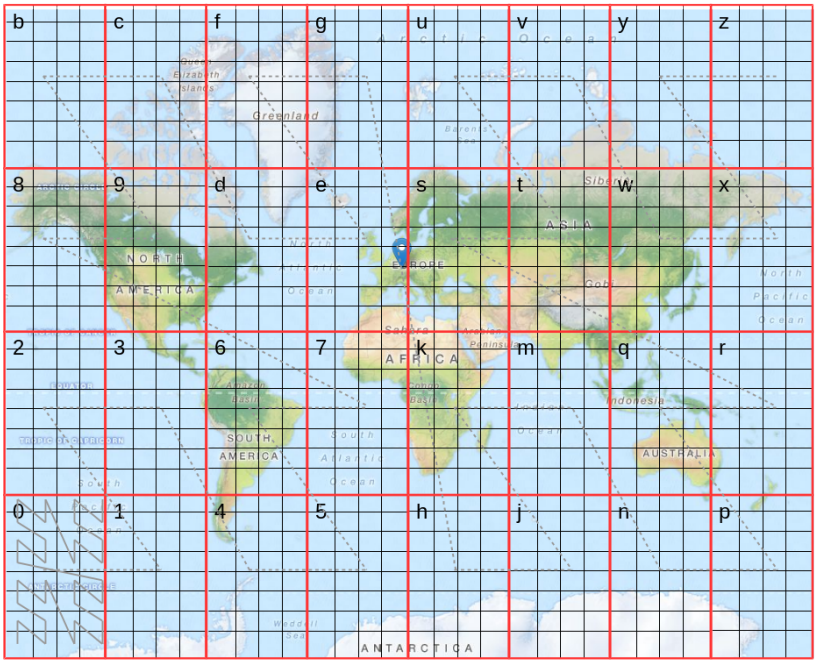


Figure 12: Geohash grid of resolution '2'

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 13, 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 14

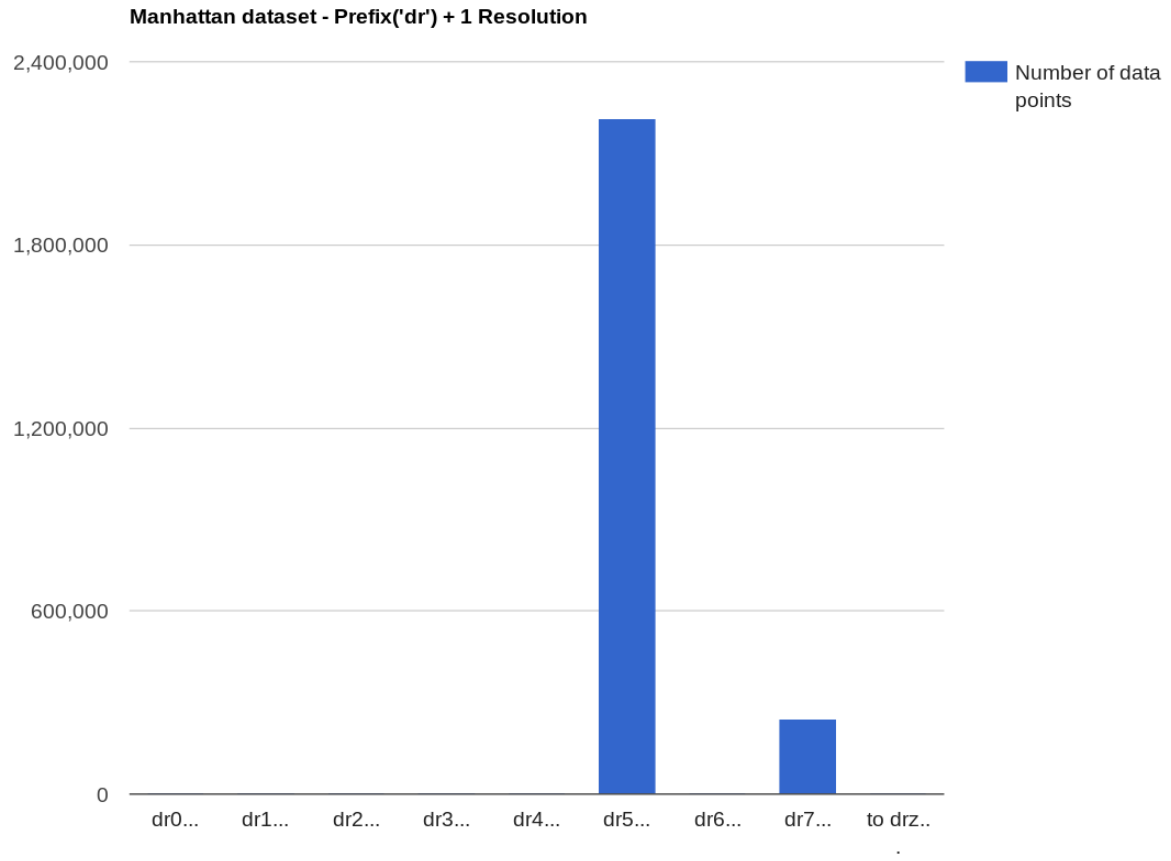


Figure 13: Imbalanced distribution of elements into partitions

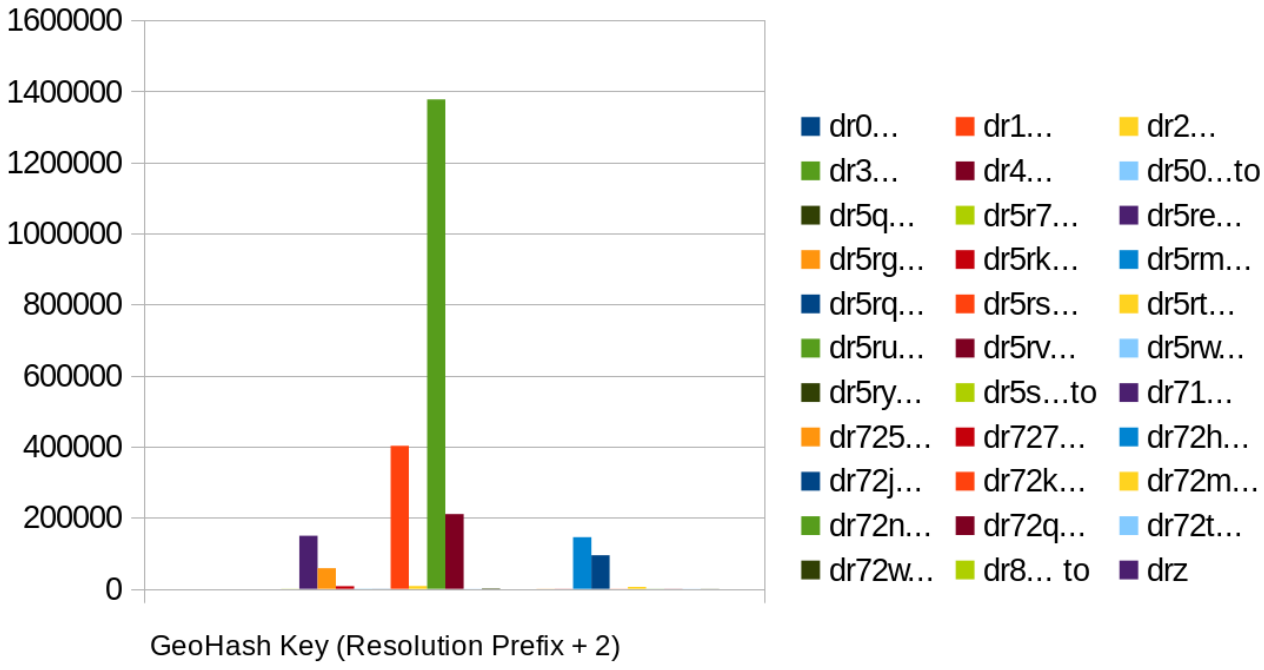


Figure 14: Increase in number of empty (or) imbalanced partitions

- This imbalance continues to exist as long as the load distribution pattern is ignored and we continue to simply increase the overall resolution of the Geohash grid

- Extremely varied sizes(number of records) of RDD partitions would result in some 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 [41]. Or even worse, we would hit time out on spark.locality.wait which would result in poor locality violating what was 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. [42]. However both these parameters - spark.speculation and spark.locality.wait - 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(*total number of records*) 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.

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 - 'keyRDD', 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.

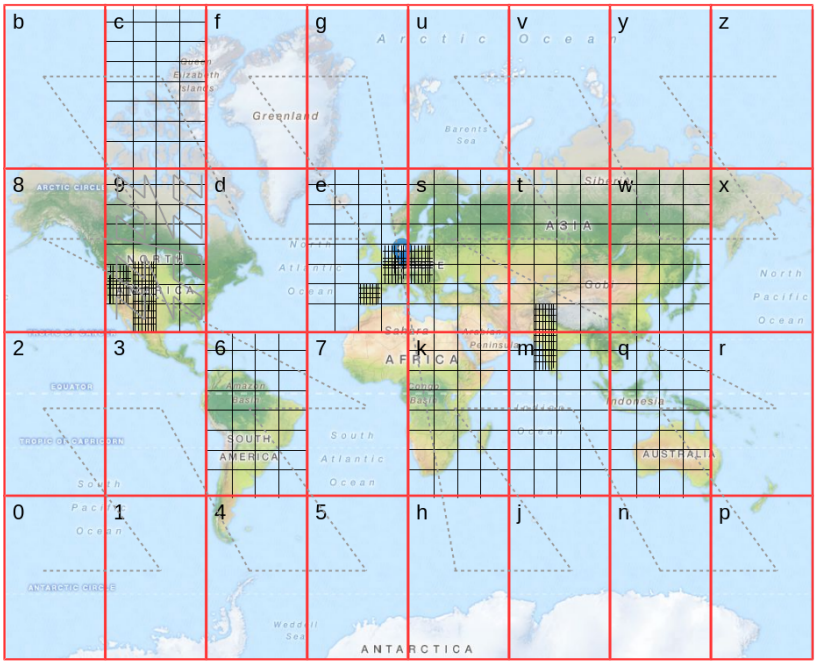


Figure 15: Geohash grids of varying resolution based on Load distribution

Determine the Threshold

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

$$Partitioning\ Threshold\ ^{3} = Total\ number\ of\ records\ /\ Number\ of\ default\ partitions$$

Since count operation involves scanning all the records, the *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 with the prefix size '1', count the number of elements in distinct keys (reduceByKey)

$$val\ level1\ =\ keysRDD.map(a\ =>\ (a.substring(0,1),\ 1)).reduceByKey(\_+\_)$$

[3] Average Record in each partition

- 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)

- Repeat the count operation by increasing the length of the code by one character for every iteration up to the predefined level of Geohash resolution(GH_{res}) or there are no segments that has more records than threshold.
- When the resolution level(GH_{res}) is reached, further filtering based on threshold is stopped, and all the left out Key prefix are added to the list.
- Suppose the resolution level, (GH_{res}), need to calculate load distribution is 'n', this process would involve 'n' times complete scan of dataset. So, to optimize it, we could create a *reduceByKey* on the n^{th} prefix length and calculating it in the reverse. This would involve only 1 complete scan of the data set.

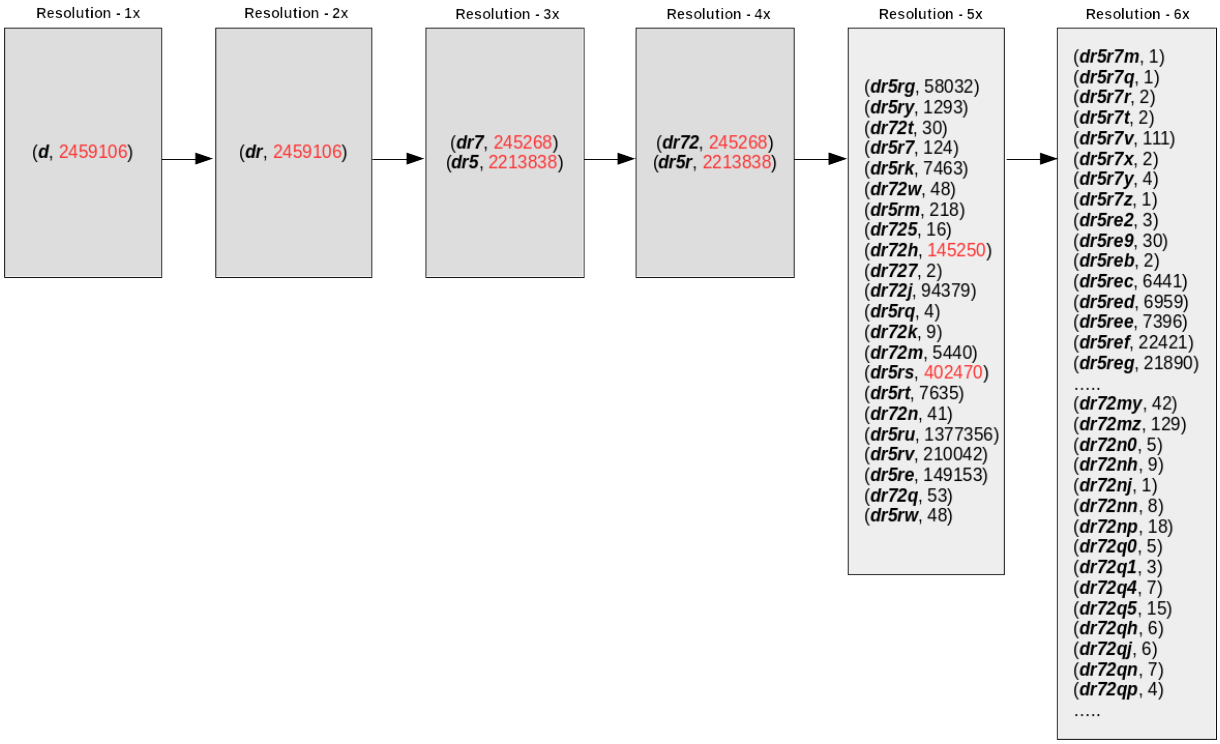


Figure 16: Levels of load distribution for increasing Geohash resolution

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.

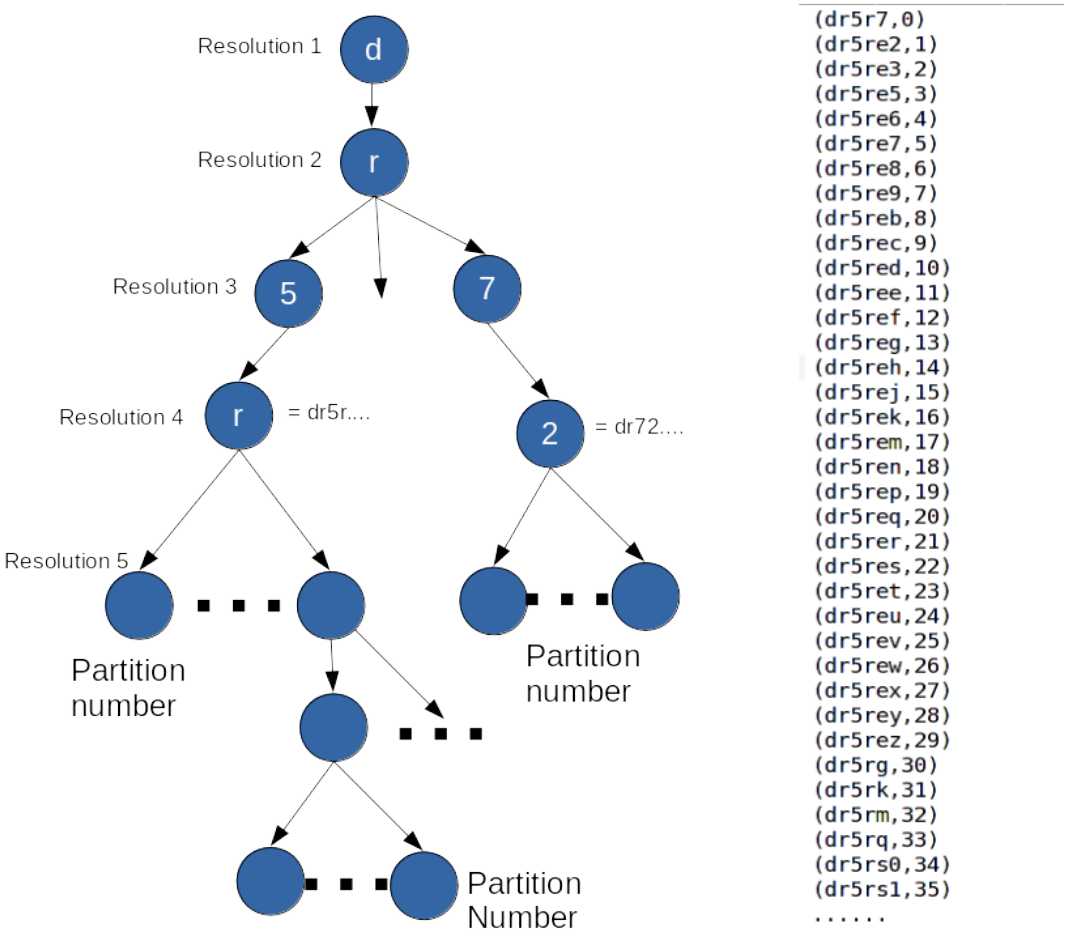


Figure 17: Geohash Partition mapper: variable length load aware Geohash key data structure

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 partitions with very less number of records, which could have otherwise been combined into a single partition, see Figure 18, 19

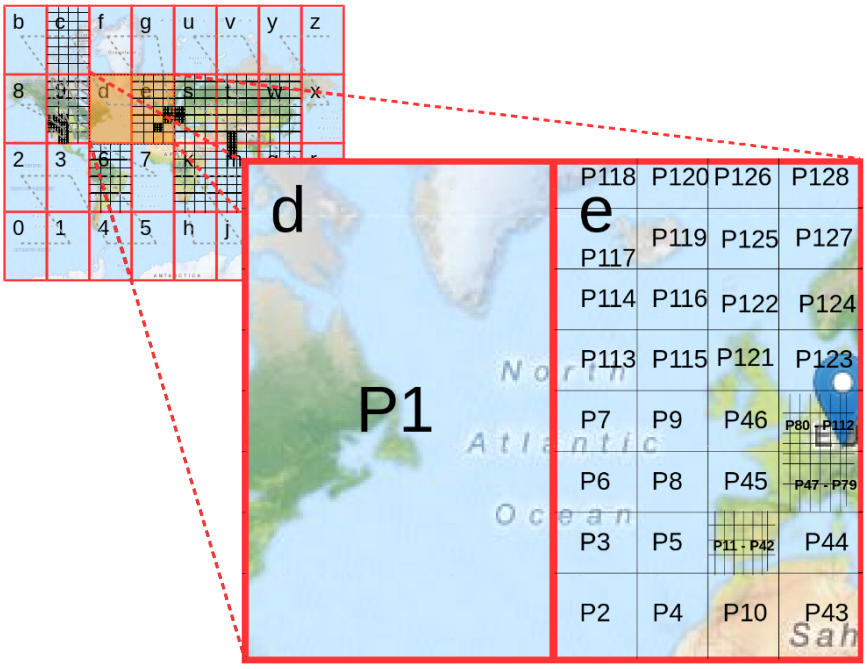


Figure 18: Enhanced optimization in creating Geohash Partitioning

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 20.

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 $\mathrm{partitionBy}()$, by incurring repeated partitioning and shuffling of records across the cluster. This would have been the case of not using any partitioner. [4]

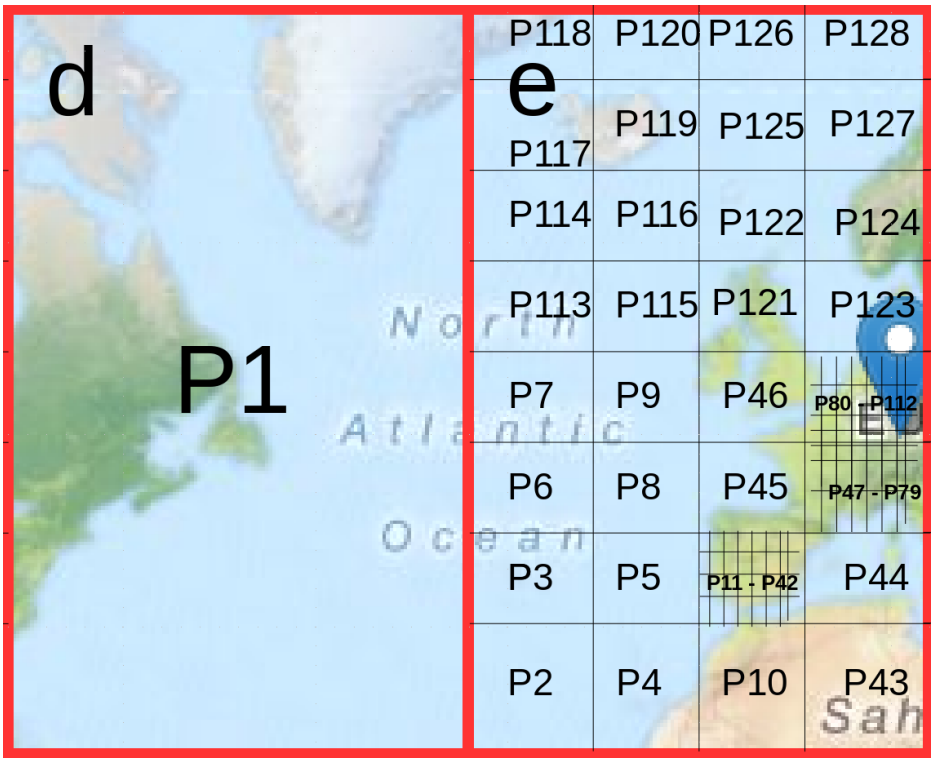


Figure 19: Geohash Partition 'e' is further partitioned to deeper resolution(has only 3 hotspot partitions)

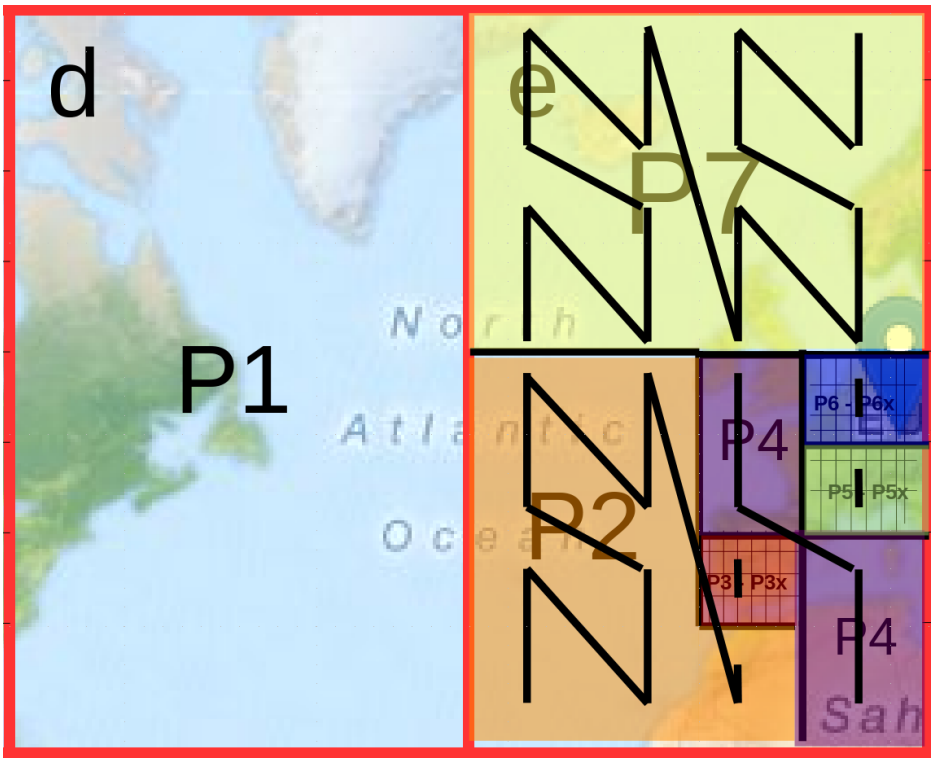


Figure 20: Previously under-fed 28 partitions combined and reduced to 3 joint partitions

C.4 Scala: Custom Geohash Partitioner

```
/**
* GeoHashPartitioner, extension of [[org.apache.spark.Partitioner]] that implements hash-based
* partitioning using GeoHash string as Key and input Map from load aware partition.
* Mapper is a variable depth tree structure depending on data distribution.

* Input Parameters: a 'mutable' Map[String,Long] which contains the variable length grid to
* partition information.

* [General Warning from Base Hash Partitioner]Java arrays have hashCodes that are based on the
    arrays'
* identities rather than their contents, so attempting to partition an RDD[Array[_]]
* or RDD[(Array[_], _)] using a HashPartitioner will produce an unexpected/incorrect result.
*
*/

import org.apache.spark.Partitioner

class GeoHashPartitioner(partitionerMap: scala.collection.Map[String,Long]) extends Partitioner {

   require(partitionerMap.size > 0, s"PartitionerMap ($partitionerMap) cannot be empty.")

   def numPartitions: Int = partitionerMap.size

/**
* Returns the corresponding partition number of the GeoHash Key.
* Defaults: GeoHash Key is null (or) there is no entry for the particular key in the
* partitionerMap.
* Defaults are mapped to the partition number 0.
*/
   def getPartition(key: Any): Int = key match {

      case null => 0

      case _ => {
         var default: Tuple2[String,Long] = ("default", 0)

         val keyPref = key.asInstanceOf[String];

         partitionerMap.find(i => keyPref.startsWith(i._1)).getOrElse(default)._2.toInt
      }

   }

   override def equals(other: Any): Boolean = other match {
      case h: GeoHashPartitioner => h.numPartitions == numPartitions
      case _ => false
   }
```

```
    override def hashCode: Int = numPartitions
}


/* [Warning] 'persist()' the Custom partitioned data
* Failure to persist an RDD after it has been transformed with partitionBy() will cause
* subsequent uses of the RDD to repeat the partitioning of the data.

* Without persistence, use of the partitioned RDD will cause reevaluation of the RDDs
* complete lineage.

* That would negate the advantage of partitionBy(), resulting in repeated partitioning and
* shuffling of data across the network, similar to what occurs without any specified partitioner.
*/
```

C.5 Architecture Overview

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

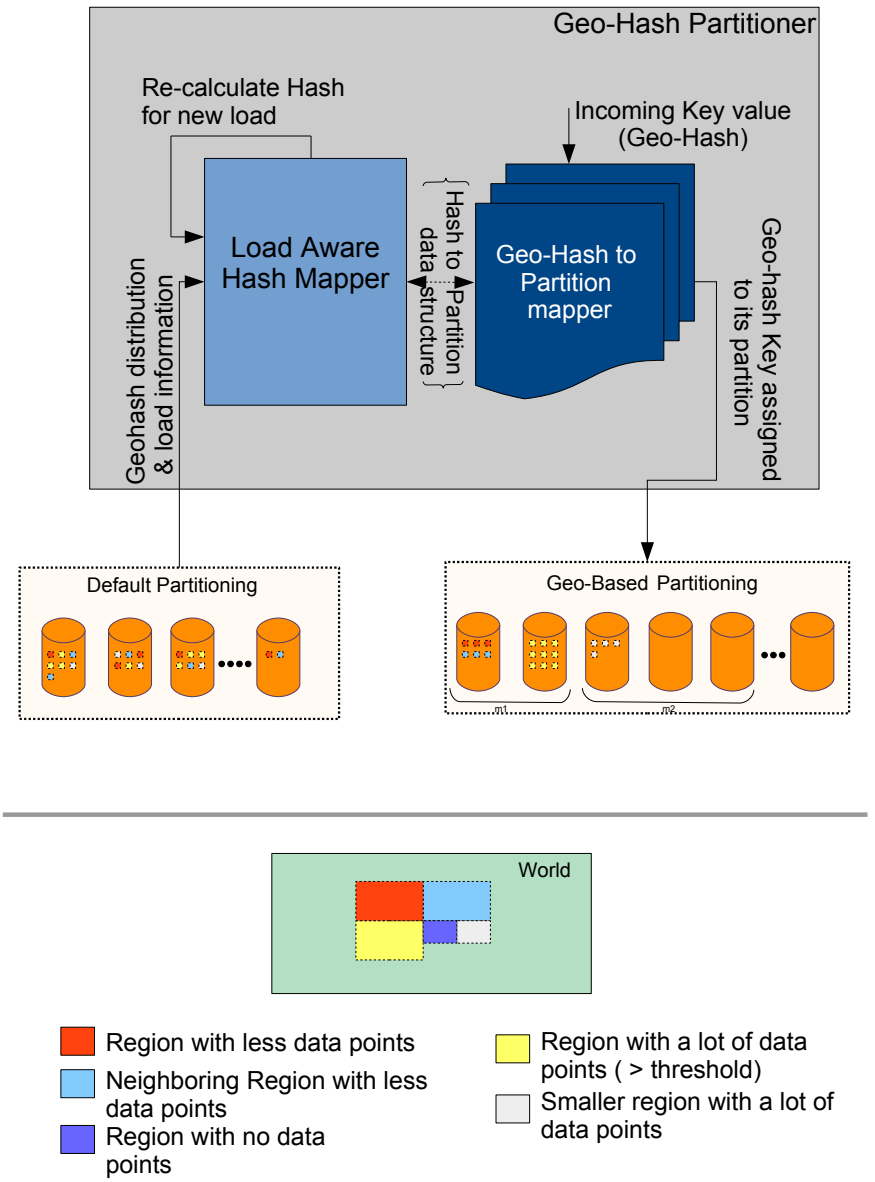


Figure 21: Architecture Overview: Geohash Partitioner

D Optimized Spatial Query Pre-processing leveraging GHP

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.

Spark provides an api, $\mathrm{PartitionPruningRDD}<\mathrm{T}>$, which is an RDD used to prune existing RDD partitions in order to prevent launching tasks on all partitions [43]. This helps us to launch tasks only on partitions that are of concern for the filter operation in the execution DAG(Directed Acyclic Graph).

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 within a circle of *150 meters* around *me*)

- This type of query provides us with the point of interest*(p)* and the desired distance*(d)*.

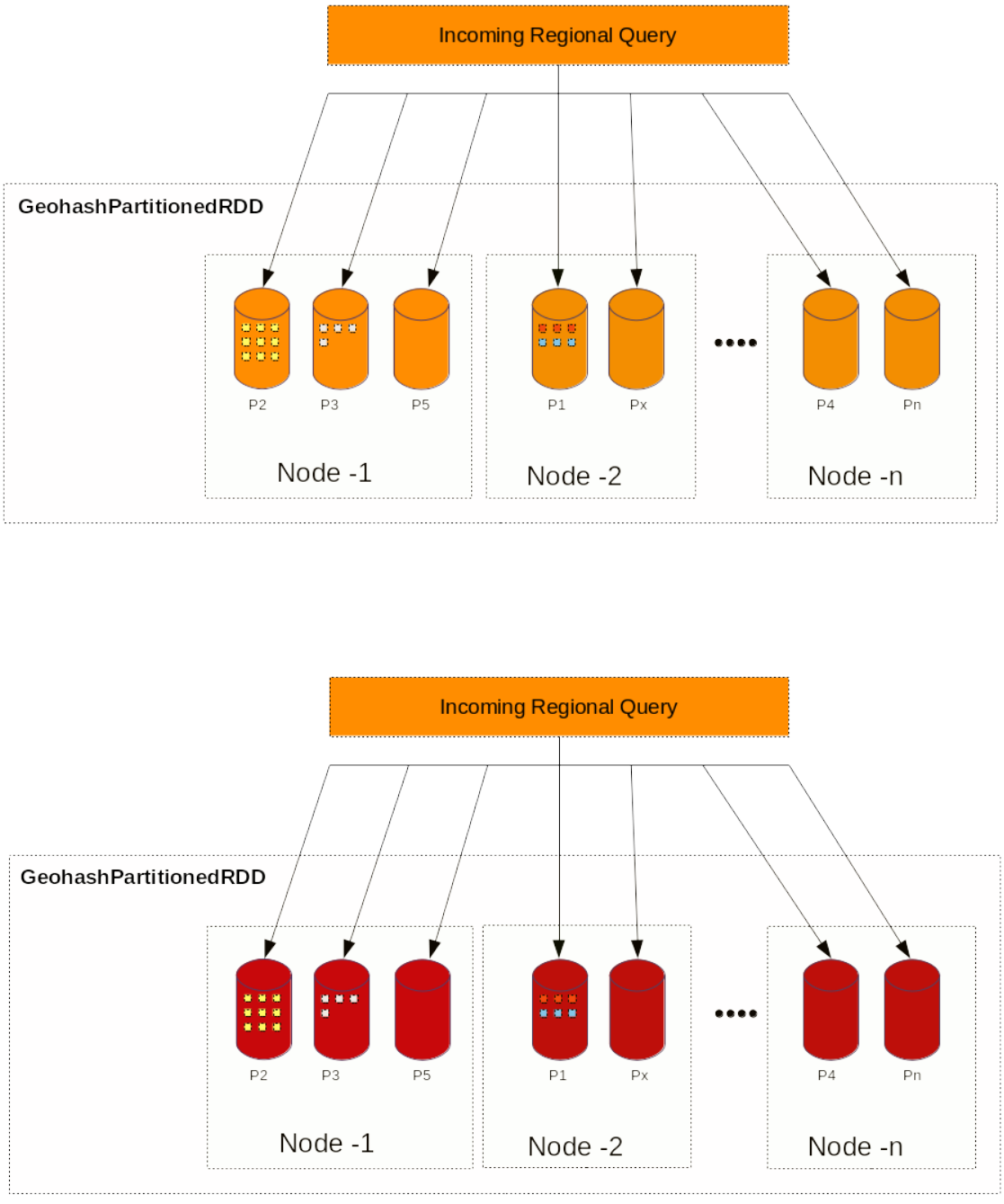


Figure 22: Spark's default behavior to launch tasks in all partitions

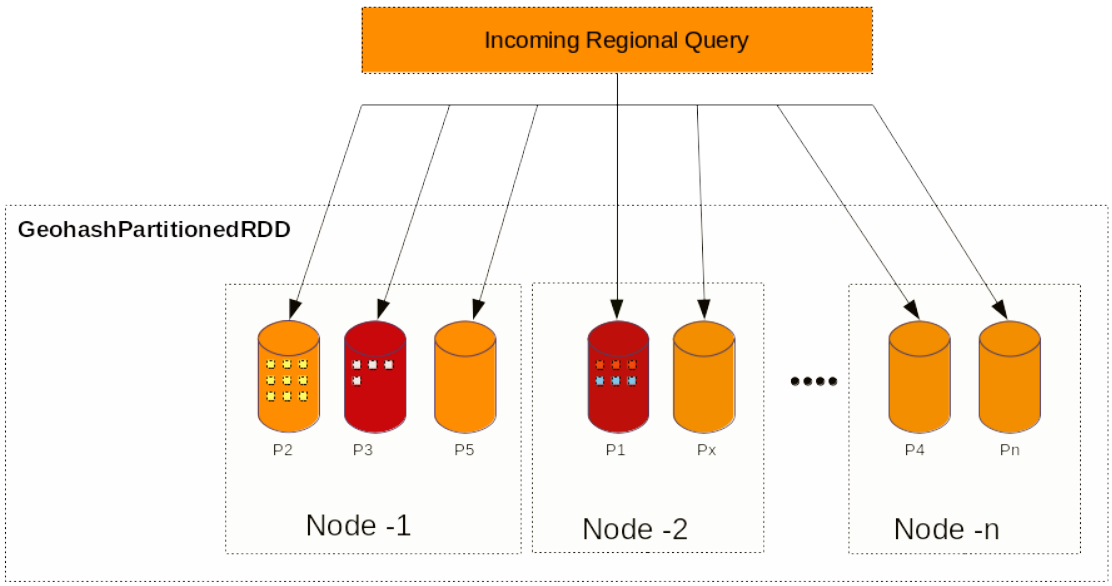


Figure 23: Spark Partition Pruning RDDs launching tasks on selected partitions

- In our approach, we make use of spatial distance calculation function *Haversine formula* [44, 45] when we calculate distance between two point and more accurate *Vincenty's formulae* [46, 47], which considers Earth as oblate sphere, in query translation module.
- 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 within 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 [48] and clearly explained in the work [49]

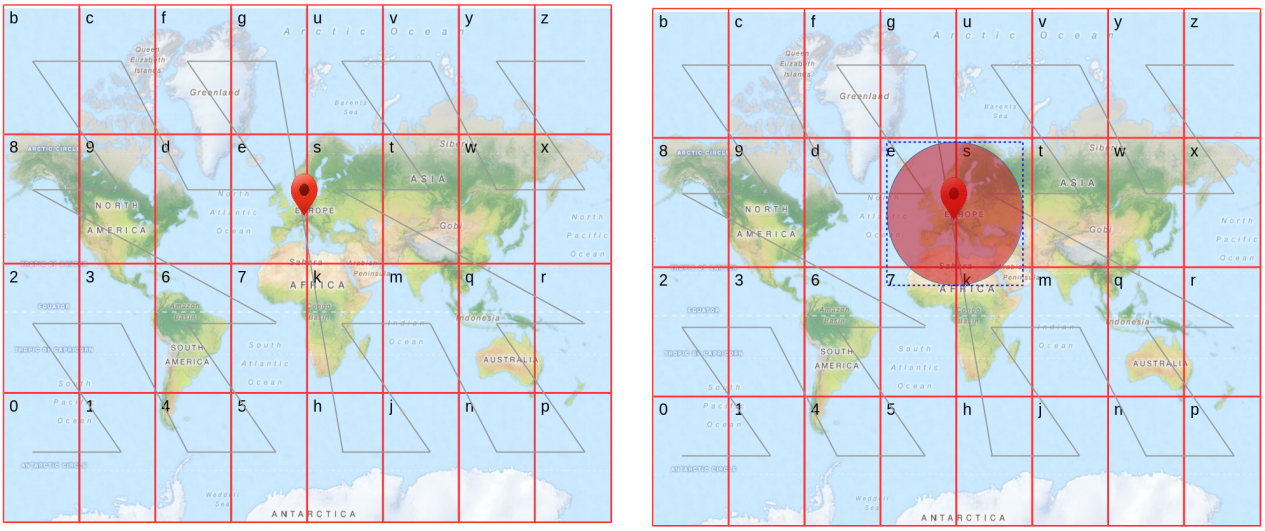


Figure 24: Query translated into Bounding Box for region of interest

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:	**dr5ru**j4477kd
SouthWest:	**dr5ru**46ne2ux
SouthEast:	**dr5ru**6ryw0cp
NorthEast:	**dr5ru**mpfq534
Query Bounding Box Prefix:	***dr5ru***

Table 3: 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 26

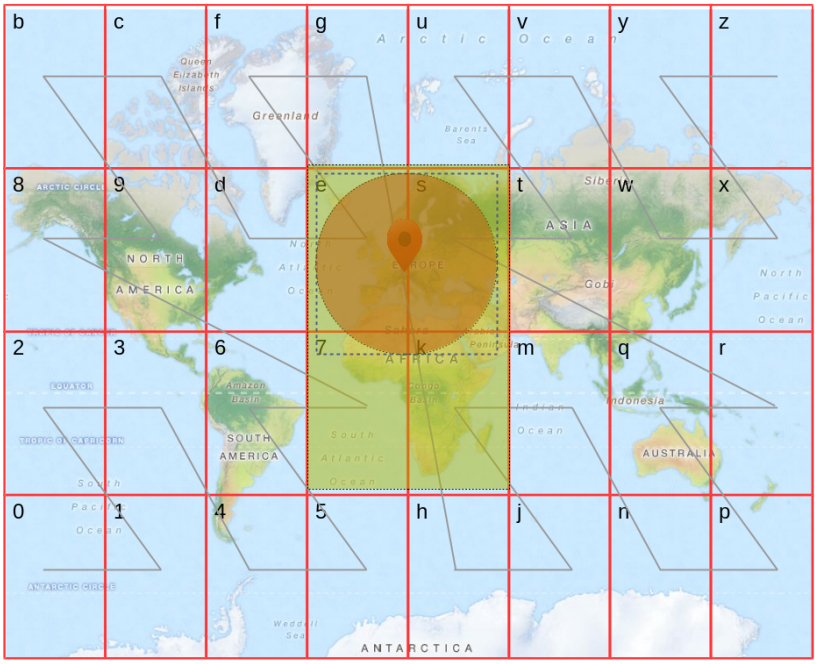


Figure 25: Translating Query into Regions of interest(Bounding Box)

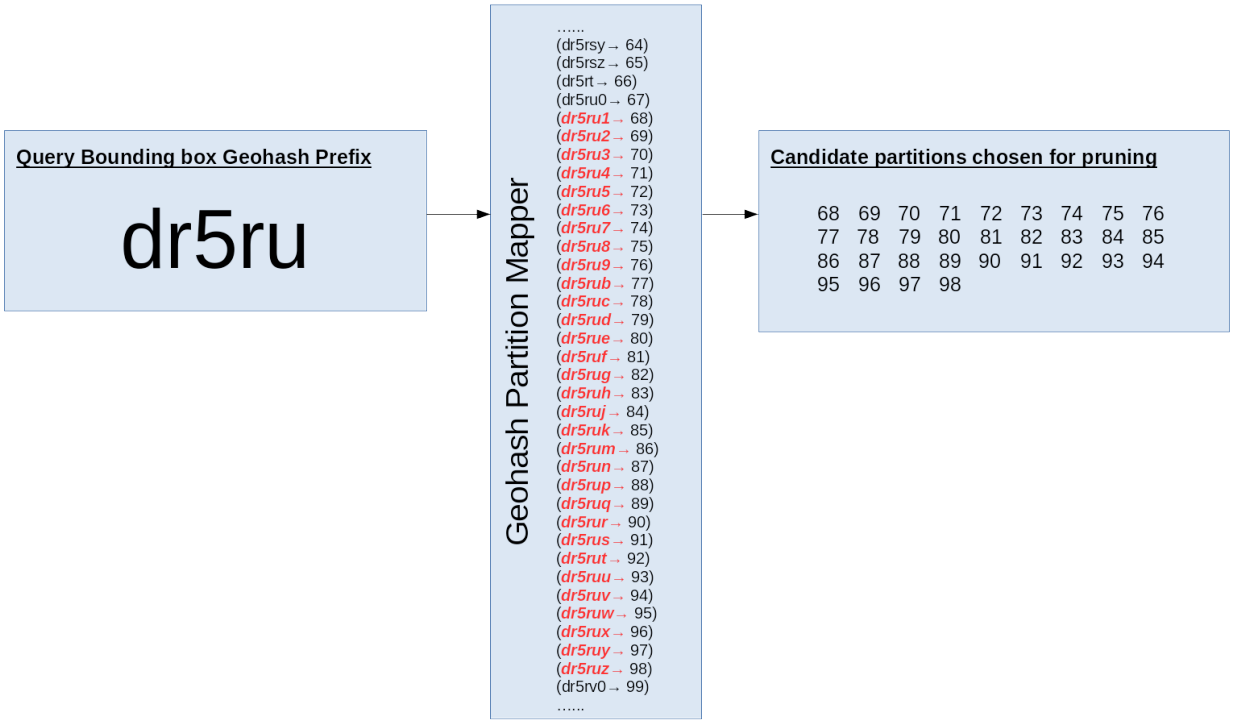


Figure 26: Choosing partitions to prune based on Query Bounding Box prefix

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 27. 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, like grid '***7***', with no further inner partitions has to be considered as whole.

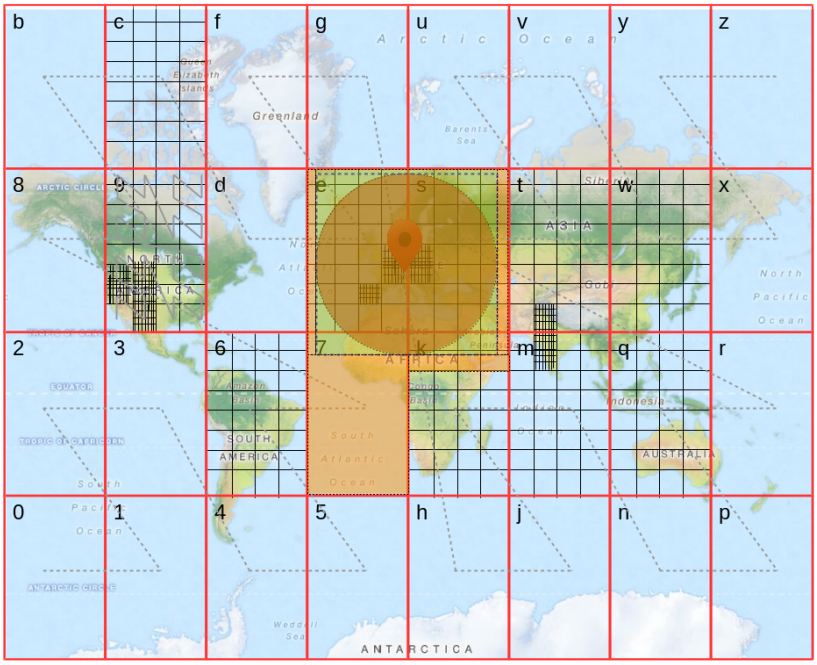


Figure 27: Enhanced candidate partitions selection based on varied grid resolution information

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 23. This results in considerably less workload and leading to fewer scans in obtaining the same results and in turn enhancing performance. The efficiency of *Geohash Partitioning and Query optimizing* approach could be evidently noticed from the evaluation results. The later section E 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

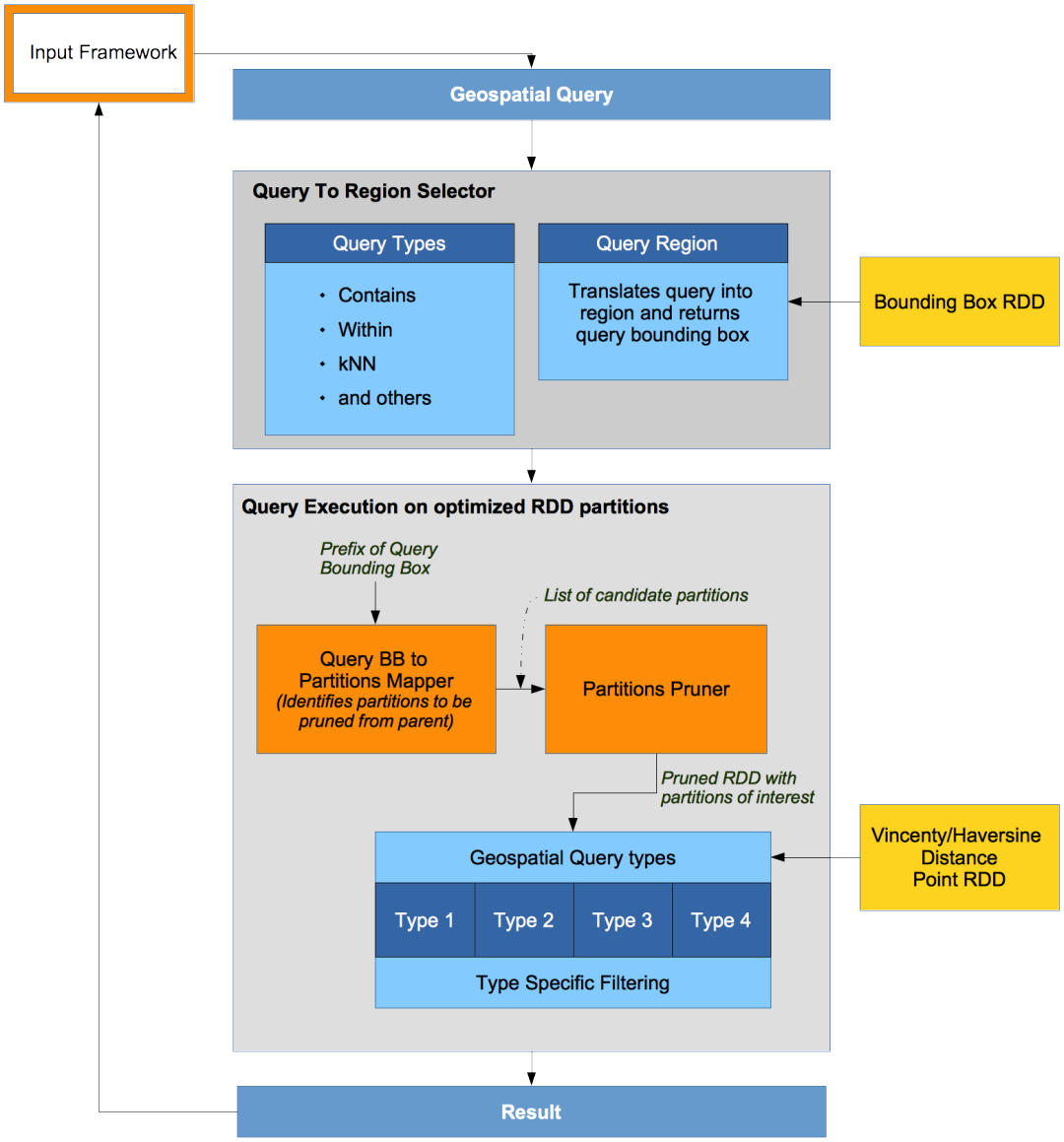


Figure 28: Architecture Overview: Geohash Query Optimizer

E Evaluating our approach on a Spark & HDFS cluster

E.1 System Footprint

E.1.1 Environment

The design is evaluated on virtual machines at IBM Research and Development Lab in Germany.

System Property	Value
Number of VMs	*5*
Operating System	*RedHat Enterprise Linux, 3.10.0-327.13.1.el7.x86_64*
Memory	*16GB (X 5)*
CPU	*9 (X 5)*
Disk	*250GB (X 5)*

Table 4: Test System Specifications

Spark cluster with a master and 5 worker nodes are configured in these machines with the following specifications:

System Property	Value
Spark Version	*1.6.1(Pre Built for Hadoop 2.6)*
Deploy mode	*Cluster*
Cluster Manager	*Standalone*
Number of Worker nodes	*5*
Application Language	*Scala, Version 2.10.5*
Java	*Java-1.8.0 Openjdk-1.8.0.91-0.b14.el7_2.x86_64*
Input datasource	*HDFS*
Executor Memory	*12GB (X 5)*

Table 5: Spark Cluster Specifications

The input spatial dataset is stored and retrieved from HDFS layer below Spark

System Property	Value
Hadoop Version	*2.6.4*
Number of Worker Data-nodes	*5*

Table 6: Hadoop HDFS Specifications

E.1.2 Setup Overview

The overall test environment setup is depicted in the Figure 29

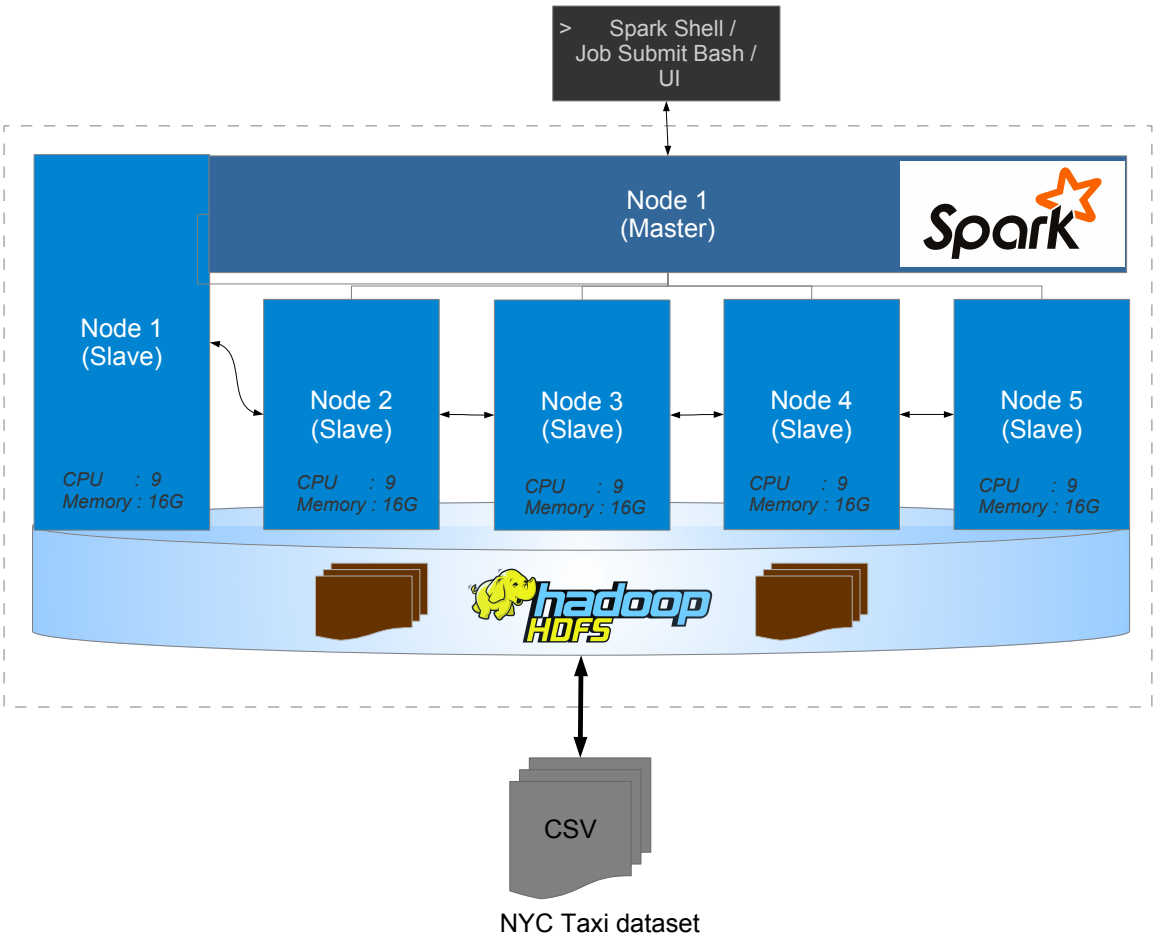


Figure 29: Apache Spark clusters running on Top of HDFS(Node 1 acts as both master and slave)

E.1.3 Dataset

The evaluation used Input dataset with geospatial point data from New York Taxi dataset [50].

The dataset is in CSV format and it contains records with details about trips made, with month level segregation for any particular year. Each record corresponds to a trip and fields of these records includes

- **Time**(Pick up and Drop)
- **Location** (Pick up and Drop) - *Spatial Point(latitude, longitude)*
- **Trip Distance**,
- **Trip Fare**,
- **Payment Type** etc

The evaluation was performed on varying sizes of dataset,

- Small (just for coding the logic),
- Medium (~15 million records) and
- Large (~175 million records).

The spatial data from the raw dataset comes with few corrupted GPS recording, locating irrelevant locations. Before using them for evaluation, we use spark to cleanse the dataset to remove all empty and erroneous location references. The cleansed dataset is then uploaded and persisted into HDFS.

We create a *Spark context* object, which instructs Spark how to access a cluster, and *textFile* method to read data from HDFS layer

```
//reading from HDFS data source
val inputNycdata = "hdfs://ghp.evaluationsetup.com:9000/nyc/large"
val nycRdd = sc.textFile(inputNycdata).persist()
```

Fields	Record Value
medallion	96472
hack_license	8EB9CE00A5AD729095BC542FC5FDB5F4
vendor_id	0766C20575FB088022E9B7D889D899A3
rate_code	VTS
store_and_fwd_flag	1
pickup_datetime	2013-01-01 10:31:00
dropoff_datetime	2013-01-01 10:45:00
passenger_count	2
trip_time_in_secs	840
trip_distance	4.67
pickup_longitude	-73.991516
pickup_latitude	40.75798
dropoff_longitude	-74.016251
dropoff_latitude	40.706989
payment_type	CSH
fare_amount	16.5
surcharge	0
mta_tax	0.5
tip_amount	0
tolls_amount	0
total_amount	17

Table 7: A single record of NYC Taxi dataset

E.2 Results

Once the dataset is read from HDFS, the necessary spatial fields are GeoHash encoded and a default Key Value RDD is created with GeoHash value as the key. Then a new RDD is created by partitioning the default RDD, based on GeoHash partitioner logic. Queries are executed against both these RDDs. The following metrics are observed.

1. Query Response Time
2. Time taken to perform Geohash based partition
3. Number of scans to process queries
4. Factors affecting Locality of data during execution
5. Impact of Shuffling before and after Geohash partitioning

Besides the Query response time, parameters like are calculated. Then the locality of tasks during the query execution on these differently partitioned RDD is observed.

Spatial ***Within*** query was executed on these RDDs and results are observed.

> *"Return all the Taxi* ***on boarding events*** *that happened within a radius of* ***200 meters*** *around* ***Time Square(40.758895,-73.9872836)****"*

Any evaluation execution runs in parallel for three different partitioners logic,

- Default partitioning (resulting from the Key Value RDD)
- GeoHash Partitioning (load aware location based partitioning)
- GeoHash Partitioning with Query optimization and results are evaluated.

Improvement in Spatial Query Response time

As a first metric for the evaluation, *Response time* of spatial queries is measured for different dataset sizes and on RDD with (i) Default partitioning (ii) Geohash partitioning and (iii) Geohash partitioning with optimized querying.

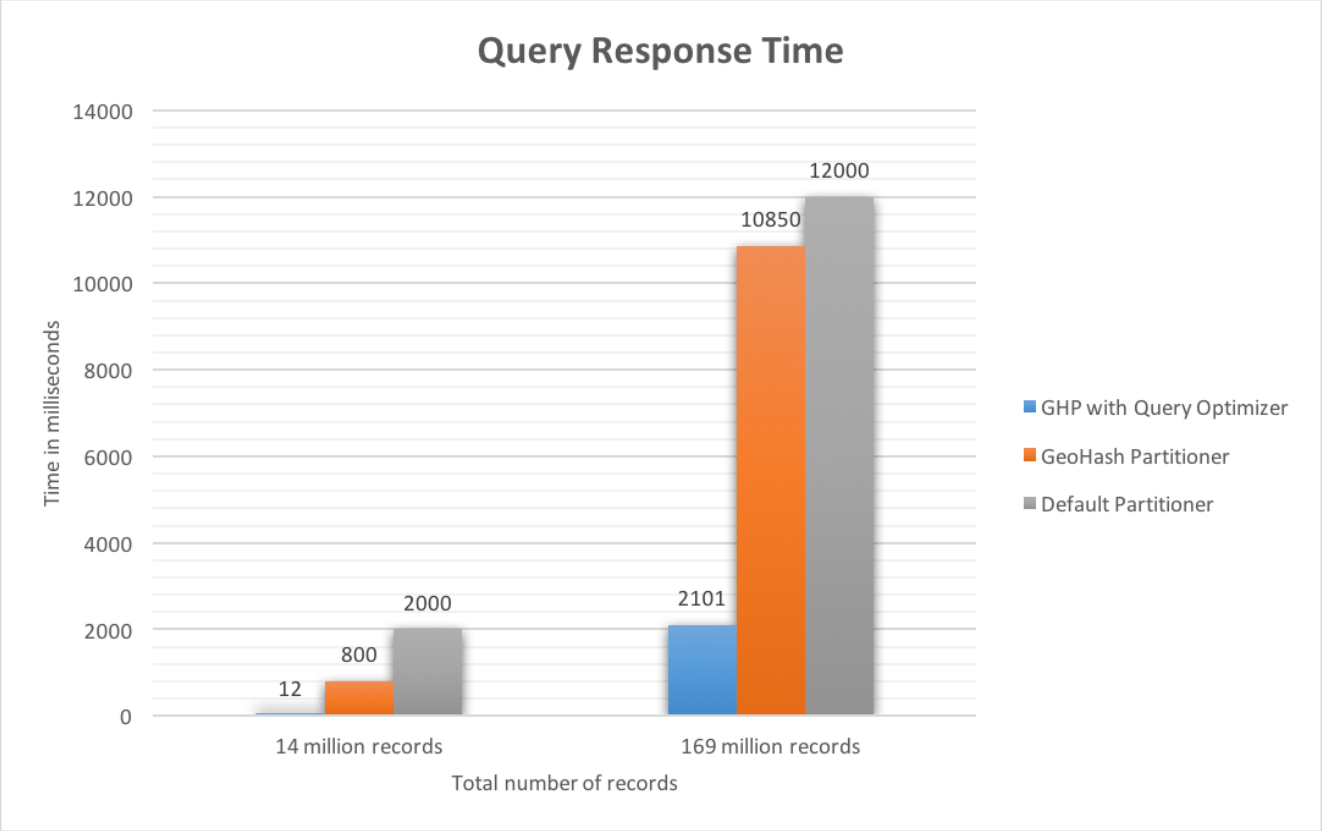


Figure 30: Improved Spatial Query response time

The following were inferred from the results,

- There was no drastic improvement in query response time in for Geohash Partitioned RDDs as against default partitioned RDDs. The considerable reduction in response time with GHP RDDs is due to the minimized shuffling, and shuffling induced read writes. For medium sized data set, there was an improvement of more than 50%, on the other hand, for larger dataset the improvement was around 10%

- On contrary, the response time of queries on GeoHash partitioned RDDs with Query preprocessor displayed noticeable reduction in response time. The improvement was prominently seen in all sizes of datasets.

Reduction in number of records scanned

The main focus of the Query layer is to minimize the number of partitions affected by an incoming spatial reads. We choose only the candidate partitions that are of interest to the query region. However it is not possible to completely avoid partitions that do not contain data for the read request. The effort is focused instead on not missing out any potential partition with data for that query.

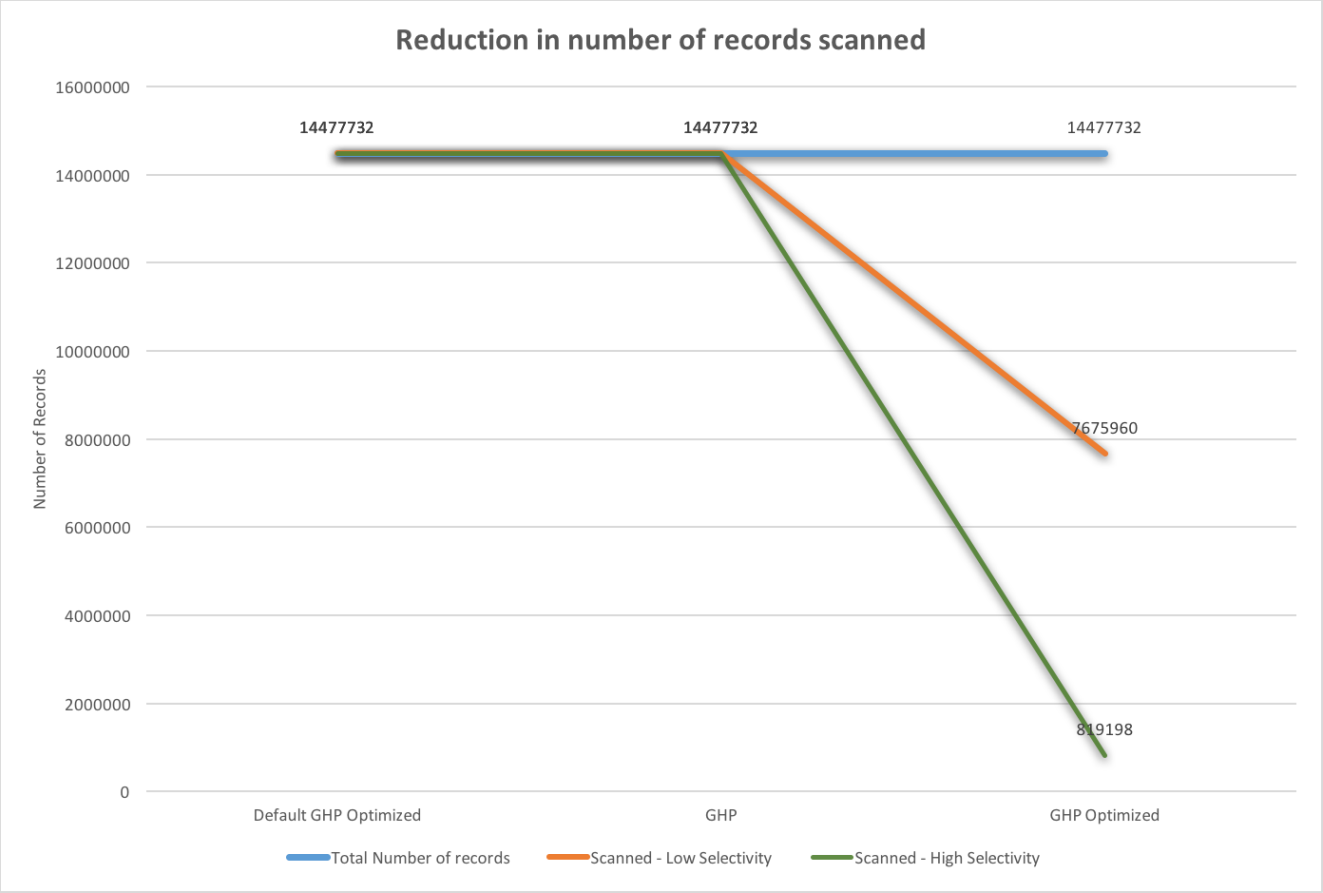


Figure 31: Reduced number of scanning records

By pre-processing the query, we prune a minimal set of RDD partitions upfront and run query on them. Results show that there has been a tremendous decrease in the number of records scanned and number of task executed for completion of the same query. This also reduced the response time and resource utilization, like the number of tasks launched to process the same query and produce the same result. The percentage decrease in number of records scanned varies depending on the Query, suppose we have a query to calculate revenue for the entire NYC region, then there is always a complete scan.

Reduction Locality of Spark tasks and minimized shuffling

The following observations were made with regards to locality of tasks

- The GeoHash partitioned RDD(with optimized querying) achieves 'Process_Local' locality (see section B.2 for explanation), which is the best possible locality, in most scenarios.
- However since we perform *Hash based* partitioning, and since load balancing is performed only up to a predefined level of Geohash resolution(GH_{res}) of order 6(*configurable*), there are few region with much denser points even after load balanced drilling. This results in few partitions with very high number of records, taking longer time to process than rest of partitions. Spark speculation tries to re-launch any of the slow running task(or) threshold to wait on locality would transfer task over network to other less-local nodes resulting in Any level(*worst*) locality (see section B.2 for explanation). This is avoided by configuring a higher locality wait time - spark.locality.wait (default 3s) and ensuring spark.speculation is set to false (or) a higher timeout value for spark.speculation.interval (default 100ms).

The advantage of partitioning based on domain knowledge is to minimize network cost when there is a domain specific query. This network cost is incurred due to the shuffling of records across partitions and in turn nodes. According to spark's programming guide [18], the operations which could cause shuffling are:

- **Repartition** operations: repartition, coalesce
- **ByKey** operations: groupByKey, reduceByKey
- **Join** operations: cogroup, join

To analyze the impact on reduction in shuffling read/write, we create a another query which would involve one of the above mentioned operations. Consider the following scenario and its query, a taxi company needs to find region wise income for all trips.

Simple *reduceByKey()* operation::

> *"List down the Region wise sum of all the trip income. (For simplicity, suppose that regions are specified in terms of Geohash grids.)"*

Simple *join()* operation::

> *"List down all the trips that started 350 meters around Time Square(40.758895,-73.9872836) and trip distance is less than 5 kilometers"*

From Figure 32, we could see that there is around 75% reduction in size of Shuffled Read and Shuffled Write after the dataset is Geohash partitioned for reduceByKey() operation. We should, however, remember that there is always an initial Shuffle involved once during custom partitioning. However, once they are partitioned, there is a reduced shuffling for all future queries on the RDD.

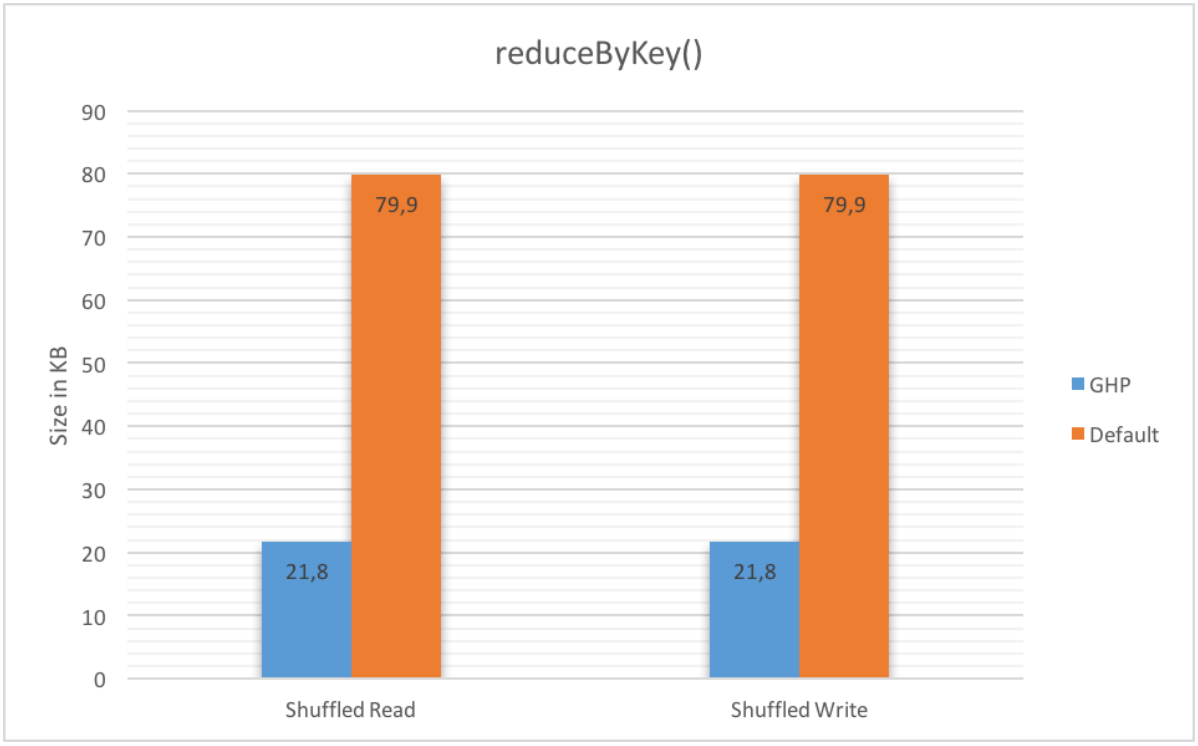


Figure 32: Shuffling read/write Behavior: reduceByKey()

When we analyze the case for join() operations, Figure 33 shows that there is a very high penalty for join operations in terms of Shuffled read/write when there is no location based partitioning. For instance, when we join two RDDs based on location key, the default partitioning results in very large shuffle write upfront. This is retained as long as the query is repeated. However, if there is another query involving join, then again a huge shuffle write is involved. This is greatly minimized when records are already shuffled to placed closer to their neighbors by our partitioning scheme. The values shown in Figure 33 is for all the trips that cost less than $10. The size of shuffle varies depending on the number of records in joining and selection criteria on the joining RDDs. Not only does this involve network cost but also affects response time since it involves writing shuffled values to disk. Figure 34

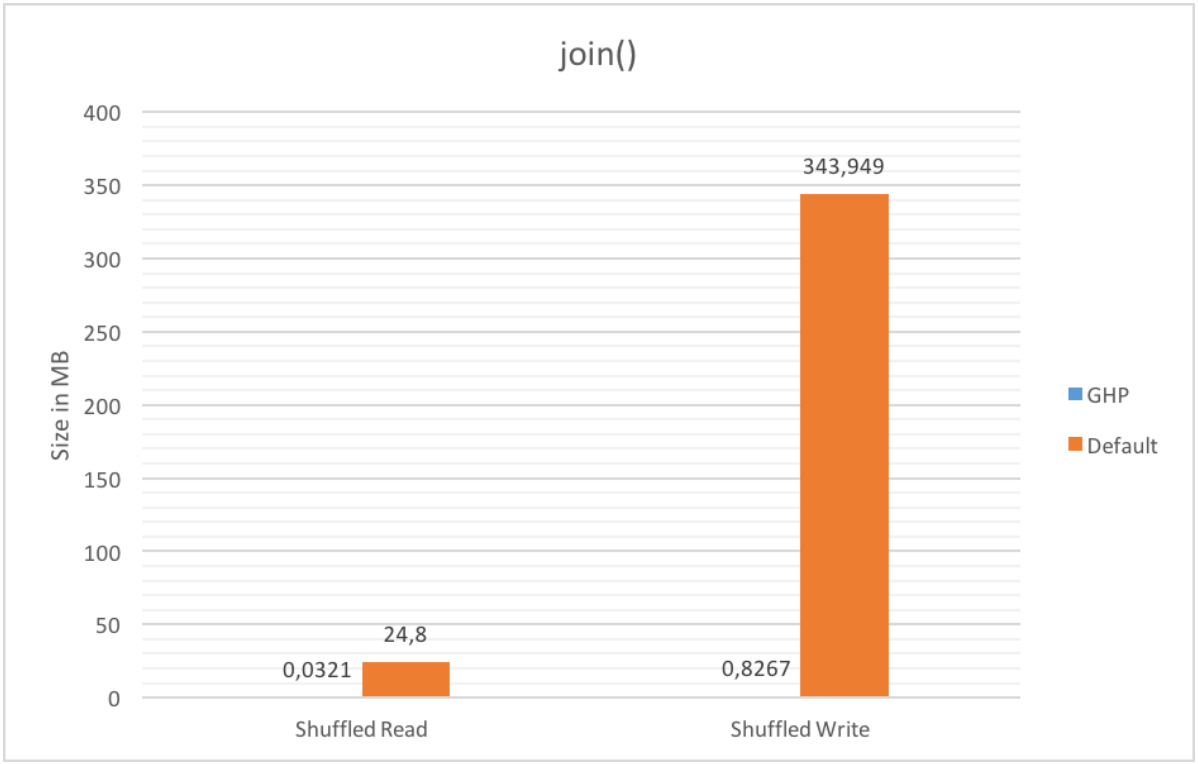


Figure 33: Shuffling read/write Behavior: join()

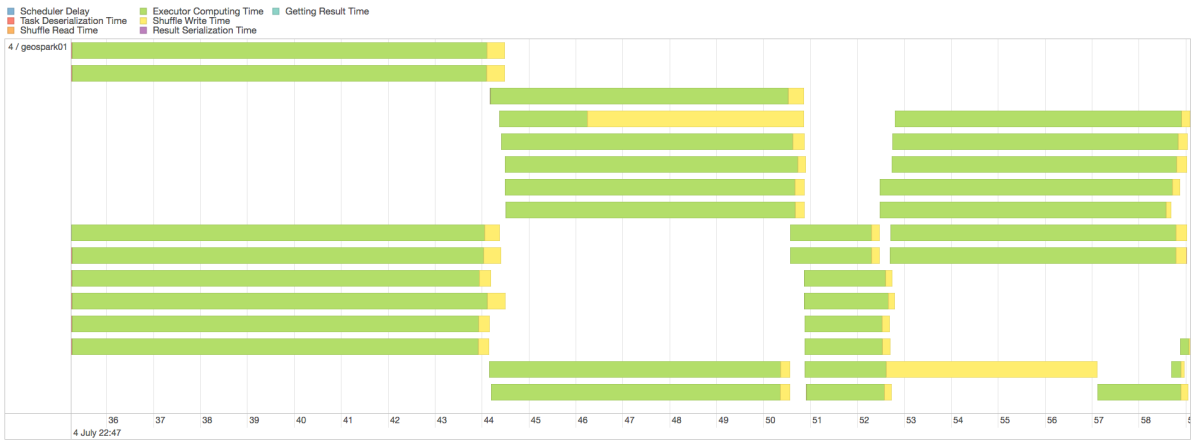


Figure 34: Timeline view on Spark UI indicating shuffle Write Time

Partitioning is not always beneficial: Benefits of Partitioning with multiple Reads

As we could infer, running custom partitioning takes considerably longer than default partitioning due to the steps involved: (i) additional transformations, (ii) learning about the load to performing load balancing, (iii) increase or decrease in size of partitions and (iv) redistribution of data points, there is a considerable increase in the time Spark takes to partition relative to the default partitioning.

So, we should take into consideration also the time it took to partition the dataset. However this is a one time process and later spatial querying on these partitions are effective in terms of response time and minimized shuffling. So, when we calculate the overall benefit the partitioning provides, we consider the impact on multiple reads(spatial queries, filters and joins) on this Geohash Partitioned RDD.

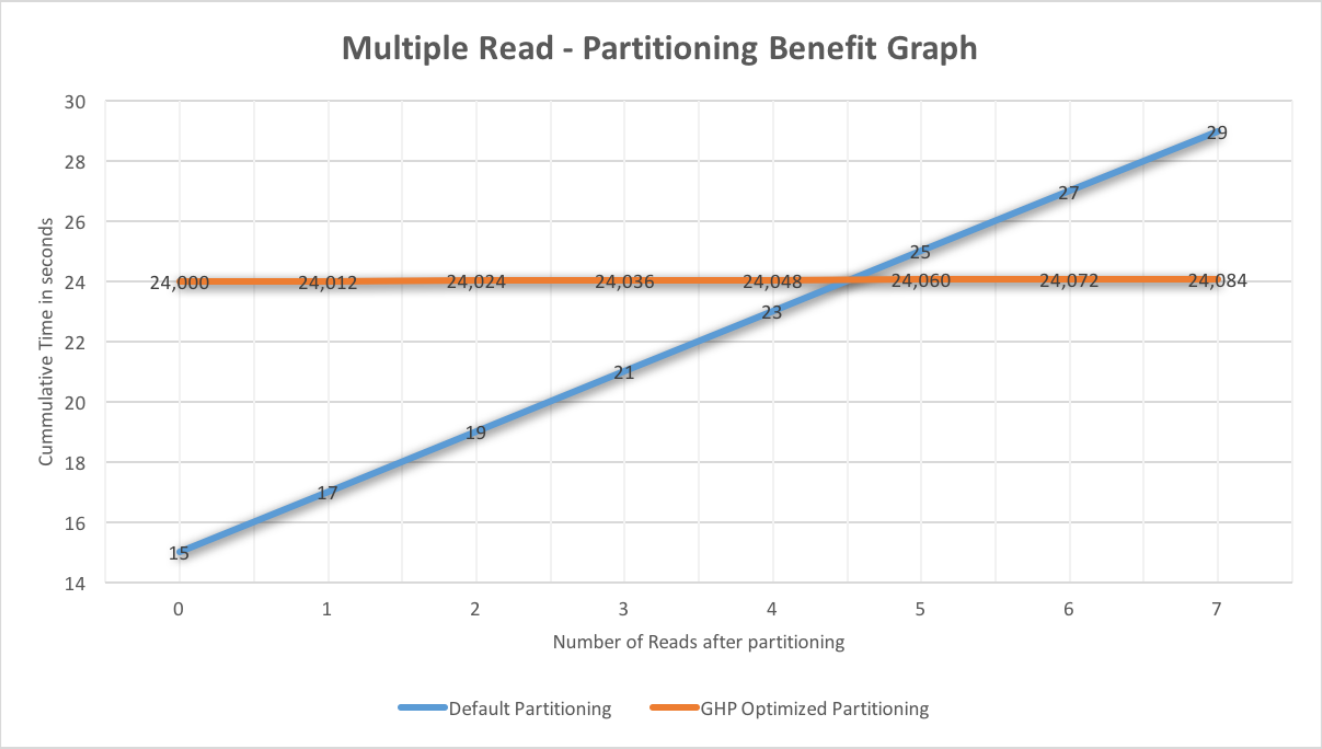


Figure 35: Partitioning benefit with multiple reads

From the graph it is obvious that

- the reason to partition has to be solid
- the application must contain multiple Spatial reads on these partitioned RDDs
- need to maintain the context of partition through the application's lifespan

E.3 Challenges and Limitations

The realization of the proposed approached imposed few challenges and limitations. The thesis work tried to solve those challenges with simpler solutions and optimizations and the rest of technical challenges that are beyond the scope of this work have been documented as limitations in this section.

Inherent imbalance with Hashing

Hash based partitioning alone, generally results in an imbalanced partitioning. There could be more records corresponding to one particular hash value. This is greatly possible with geospatial data set with data points concentrated around major cities and temporally in hotspot areas. Hence there is a need to address this issue and we propose that the algorithm learns the data distribution of the input dataset and limits the number of records to a hash key based on a threshold value. This is achieved by curves filling the earth with variable orders(increased curve order in areas of density and reduced in sparse areas.)

Need for minimizing scans during load balancing

During the load aware balancing phase, there is a need to scan through the complete dataset multiple times to encode latitude, longitude to Geohash, calculate Geohash prefix, understand the load distribution under each prefix cluster. Such multiple scanning during partitioning are acceptable if the size of dataset is considerably small. However for large data sets with more than 100 million records, would result is *n x 100 millions scans* if we are scanning them for *'n'* difference operations. The algorithm addresses this problem by drilling deeper scan once and building in reverse for coarser resolutions as explained in figure 30.

Custom partitioner: Need to maintain metadata

The default hash partitioning algorithm in spark depends on Java hash code. For every incoming record, the hash code of its key is generated. Each hash code which is an integer, is mapped on to one the available partitions using

$$Partition\ number = Hashcode\ /\ No.of\ partitions.$$

In this case, the hashcode-to-partition mapping could always be calculated dynamically.

However, in our case, since hashcode is a geohash string and we perform a load balanced hash partitioning we always need the data structure that holds the load aware geohash to RDD partitions mapping metadata. We need this metadata to perform partitioning.

Custom partitioner: Need to retain spark context

Once the spark Job is completed the Geohash partitioned RDD has to be persisted on to a disk (or) HDFS. When we read these RDDs back again within a different spark context, the RDD would retain the partitioner and partitions but not the same corresponding partition number. In this case, the metadata with mapping from Geohash to Partition number becomes irrelevant. Hence we are need to maintain the same spark context throughout the life span of the application or the life span of spark cluster. This is the normal case with any Spark application, yet it is a limitation.

F Conclusions

F.1 Contribution

This thesis work acknowledges the unprecedented growth in the usage of geospatial data and seeks modern distributed computing platform like Spark to efficiently process them. In this process, the work identifies challenges and limitations posed by these platforms in handling geospatial data. The need for retaining locality of geo-points with RDD partitions are emphasized and realized.

A custom partitioning algorithm to retain *partition-level* locality has be designed, implemented and evaluated. The partition algorithm uses Geohash encoding scheme, which in-turn is based on z order Space Filling Curve, to transform the dimensionality of spatial data and leverages from '*Hierarchical Prefix*' property of Geohash encoding to efficiently layout spatial data into RDD partitions. In order to avoid imbalance in the partition sizes, the algorithm proposes a load aware balancing technique using variable resolution Geohash grids. This helps in avoiding starvation in few CPUs while others are overloaded. The algorithm is optimized at several areas to minimize the scan needed to study the load distribution on earth surface. Once the data set is custom partitioned using Geohash Partitioner, further spatial transformation queries on these partitions results in minimized shuffled read/write and achieve very low locality possible($\mathrm{Process\_Local}$).

Further more, the later part of the thesis, proposes an efficient spatial query processing algorithm for any transformation that act on these Geohash partitioned RDDs. This layer leverages the knowledge of Geohash partitioning meta-data which is a hash table mapping geohash code to RDD partition number. It translates queries into 2D space and draws a query region that are affected by the query. This query region is then evaluated against the partitioner information to choose a minimal subset of partitions to run the spark tasks. This approach drastically reduces the number of records scanned and substantially increases query response time.

The thesis work further explains the limitations of our approach like the need to retain partitioning metadata across context and concludes the dissertation by proposing various ideas for future extension of this work.

Supporting additional Geometry types

Our dissertation focuses mainly on spatial data with **Point** geometry. This could be extended to support composite Geometries like LineString, Polygons, Circles which would represent Transport routes, city boundary, travel zones respectively. This idea would involve calculating a overlapping geohash for each composite geometry, say a *polygon* would be represented by the geohash of a Minimum Bounding Geohash Rectangle.

Extending to other distributed platforms

The Geohash based partitioning was implemented on Apache Spark to distribute spatial data into partitions in an ordered fashion. The same algorithm could be ported to other similar parallel computing platforms like Hadoop, Hbase, or other MPP Databases to support spatial partitioning and their performance could be studied.

Building a Geo processing stack

Once partitioning is completed the partitions could be persisted into a distributed database like Elastic, MongoDB and further create a secondary Spatial *Indexing* for efficient search and retrieval of Spatial data for visualization, resulting in a complete stack for processing, indexing, visualizing spatial data.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, (San Jose, CA), pp. 15–28, USENIX, 2012.

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

[4] Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis*. https://library.oreilly.com/book/0636920028512/learning-spark/toc: " O'Reilly Media, Inc.", 2015. "Chapter 4".

[5] Microsoft, "Spatial data types overview." https://msdn.microsoft.com/en-us/library/bb964711.aspx, 2016. Accessed: 2016-05-30, Updated: June 2, 2016.

[6] "Cluster Mode Overview - Spark 1.6.1 Documentation." http://spark.apache.org/docs/latest/cluster-overview.html. Accessed: 2016-05-30.

[7] S. You, *Large-Scale Spatial Data Management on Modern Parallel and Distributed Platforms*. PhD thesis, Graduate Center, City University of New York, 1 2016. An optional note.

[8] "Kevin Buchin", *"Organizing Point Sets: Space-Filling Curves, Delaunay Tessellations of Random Point Sets, and Flow Complexes"*. PhD thesis, "Fachbereich Mathematik und Informatik der Freien Universität Berlin", "Institut für Informatik Freie Universität Berlin TakustraÃ§e 9 14195 Berlin buchin@inf.fu-berlin.de", "2007".

[9] Ramsak, Frank and Markl, Volker and Fenk, Robert and Zirkel, Martin and Elhardt, Klaus and Bayer, Rudolf, "Integrating the ub-tree into a database system kernel," in *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, (San Francisco, CA, USA), pp. 263–272, Morgan Kaufmann Publishers Inc., 2000.

[10] Wikipedia, "Z order curves — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Z-order_curve, 2016. Accessed 30-May-2016.

[11] GSMA, "Gsma: The mobile economy 2016." http://www.gsmamobileeconomy.com/, 2016. Accessed: 2016-05-30.

[12] K. Lee, R. K. Ganti, M. Srivatsa, and L. Liu, "Efficient spatial query processing for big data," in *Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '14, (New York, NY, USA), pp. 469–472,

ACM, 2014.

[13] A. Fox and C. Eichelberger and J. Hughes and S. Lyon, "Spatio-temporal indexing in non-relational distributed databases," in *Big Data, 2013 IEEE International Conference on*, pp. 291–299, Oct 2013.

[14] , "Geohack - mount everest." https://tools.wmflabs.org/geohack/geohack.php?pagename=Mount_Everest&params=27_59_17_N_86_55_31_E_type:mountain_scale:100000_, 2016. Accessed: 2016-05-30.

[15] Wikipedia, "Global Positioning System — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Global_Positioning_System, 2016. Accessed 30-May-2016.

[16] Wikipedia, "Apache Spark — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Apache_Spark, 2016. Accessed 30-May-2016.

[17] "Tuning - Spark 1.6.2 Documentation." http://spark.apache.org/docs/latest/tuning.html#data-locality. Accessed: 2016-05-30.

[18] "Working with Key-Value Pairs: Apache Spark Programming Guide." http://spark.apache.org/docs/latest/programming-guide.html#working-with-key-value-pairs. Accessed: 2016-05-30.

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

[20] Kevin Sahr and Denis White and A. Jon Kimerling, "Geodesic discrete global grid systems," *Cartography and Geographic Information Science*, vol. 30, no. 2, pp. 121–134, 2003.

[21] Wikipedia, "Grid Spatial Index — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Grid_(spatial_index), 2016. Accessed 30-May-2016.

[22] Pan Xu and Srikanta Tirthapura, "On the optimality of clustering properties of space filling curves," in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pp. 215–224, 2012.

[23] Rose, N.J., "Hilbert-type space-filling curves," 2010. http://www4.ncsu.edu/ njrose/pdfFiles/HilbertCurve.pdf.

[24] Wikipedia, "Moore curves — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Moore_curve, 2016. Accessed 30-May-2016.

[25] B. Moon and H. V. Jagadish and C. Faloutsos and J. H. Saltz, "Analysis of the clustering properties of the hilbert space-filling curve," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, pp. 124–141, Jan 2001.

[26] Wikipedia, "Hilbert curves — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Hilbert_curve, 2016. Accessed 30-May-2016.

[27] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer, *Space filling curves and their use in the design of geometric data structures*, pp. 36–48. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995.

[28] Lawder, Jonathan K. and King, Peter J. H., "Using space-filling curves for multi-dimensional indexing," in *Proceedings of the 17th British National Conferenc on Databases: Advances in Databases*, BNCOD 17, (London, UK, UK), pp. 20–35, Springer-Verlag, 2000.

[29] , "Mongodb: Calculation of geohash values for 2d indexes." https://docs.mongodb.com/manual/core/geospatial-indexes/#geospatial-indexes-geohash, 2016. Accessed: 2016-05-30.

[30] Created by Cassandra Targett, last modified by David Smiley on May 18, 2016, "Spatial search - apache solr." https://cwiki.apache.org/confluence/display/solr/Spatial+Search, 2016. Accessed: 2016-05-30.

[31] , "Oracle spatial and graph." http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/index.html, 2016. Accessed: 2016-05-30.

[32] Wikipedia, "Geohash — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Geohash, 2016. Accessed 30-May-2016.

[33] Z. Balkić, D. Šoštarić, and G. Horvat, *Agent and Multi-Agent Systems. Technologies and Applications: 6th KES International Conference, KES-AMSTA 2012,Dubrovnik, Croatia, June 25-27, 2012. Proceedings*, ch. GeoHash and UUID Identifier for Multi-Agent Systems, pp. 290–298. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[34] , "Elastic: Geohash grid aggregation." https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-geohashgrid-aggregation.html, 2016. Accessed: 2016-05-30.

[35] , "Geohash: Tips and tricks." http://geohash.org/site/tips.html, 2016. Accessed: 2016-05-30.

[36] Jia Yu, Jinxuan Wu, Mohamed Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '15, (New York, NY, USA), pp. 70:1–70:4, ACM, 2015.

[37] Ahmed Eldawy, Mohamed F. Mokbel, "SpatialHadoop: A MapReduce Framework for Spatial Data," in *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pp. 1352–1363, 2015.

[38] "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.

[39] "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.

[40] 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.

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

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

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

[44] R.W. Sinnott, "Virtues of the haversine," *Sky and Telescope*, vol. 68, no. 2, p. 159, 1984.

[45] Wikipedia, "Haversine formula — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Haversine_formula, 2016. Accessed 30-May-2016.

[46] Karney, C.F.F. and Deakin, R.E., "F.w. bessel (1825): The calculation of longitude and latitude from geodesic measurements," *Astronomische Nachrichten*, vol. 331, no. 8, pp. 852–861, 2010.

[47] 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.

[48] 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.

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

[50] NYC Taxi & Limousine Commission, "Spatial data: Tlc trip record data." http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml, 2016. Accessed: 2016-05-30, CSV format.

List of Tables