Efficient Spatial Query Processing for Big Data

Kisung Lee †, Raghu K. Ganti ¶, Mudhakar Srivatsa ¶, Ling Liu †
†College of Computing, Georgia Institute of Technology, Atlanta, GA USA
¶IBM T. J. Watson Research Center, Yorktown Heights, NY USA
kslee@gatech.edu, {rganti, msrivats}@us.ibm.com, lingliu@cc.gatech.edu

ABSTRACT

Spatial queries are widely used in many data mining and analytics applications. However, a huge and growing size of spatial data makes it challenging to process the spatial queries efficiently. In this paper we present a lightweight and scalable spatial index for big data stored in distributed storage systems. Experimental results show the efficiency and effectiveness of our spatial indexing technique for different spatial queries.

1. INTRODUCTION

Many real-world and online activities are associated with their spatial information. For example, when we make or receive a call, the call information including its cell tower location is stored as a call detail record (CDR). Even a single tweet message of Twitter can be stored with its detailed location (i.e., latitude and longitude) [1]. To extract more valuable and meaningful information from such spatial data, spatial queries are widely used in many data mining and analytics applications. One of the most representative challenges for processing the spatial queries is that the amount of spatial data is increasing at an unprecedented rate, especially thanks to the widespread use of GPS-enabled smartphones. Due to this huge size of spatial data, we need new scalable techniques which can process the spatial queries efficiently.

To handle such huge spatial data, it is natural to utilize emerging distributed computing technologies such as Hadoop MapReduce, Hadoop Distributed File System (HDFS) and HBase. Several techniques have been proposed to support spatial queries on Hadoop MapReduce [7, 11, 4, 12] or HDFS [5, 6]. However, most of them require internal modification of underlying systems or frameworks to implement their indexing techniques based on, for example, R-trees. Those approaches not only increase the complexity and overhead of the modified storage systems but also are applicable only to a specific storage system.

To tackle the limitations of existing work, in this paper, we investigate the problem of developing efficient and scalable techniques for processing spatial queries over big spatial data. Specifically, we present a lightweight spatial index based on a hierarchical spatial data structure. Our spatial index has several advantages. First, it can be easily applied to existing storage systems without modifying their internal implementation and thus we can utilize existing systems as they are. Second, it provides simple yet highly efficient filtering, based on prefix matching, for finding only relevant spatial objects. Last but not the least, it supports efficient updates of spatial objects because it does not maintain any costly data structure such as trees. In this paper, we demonstrate how we implement the spatial index on top of HBase without modifying its internal implementation. We also provide experimental results to show the efficiency and effectiveness of our spatial indexing techniques.

SIGSPATIAL '14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA
Copyright 2014 ACM 978-1-4503-3131-9/14/11 ...$15.00
http://dx.doi.org/10.1145/2666310.2666481

2. PRELIMINARY

In this section, we give an overview of spatial queries, hierarchical spatial data structure and distributed storage systems. We also outline the related work.

2.1 Spatial Queries

There are many types of spatial queries, such as selection query, join query and k nearest neighbor (kNN) query, for different applications. Even though there are more spatial relations [8], in this paper, we focus on selected fundamental queries which are basis for many other spatial queries: *containing*, *containedIn*, *intersects* and *withinDistance*. Those queries are defined for any geometries including points, lines, rectangles and polygons. A ***containing(search geometry)*** query returns all spatial objects that contain the given search geometry. A ***containedIn(search geometry)*** query returns all spatial objects that are contained by the given search geometry (i.e., the converse of *containing*). An ***intersects(search geometry)*** query returns all spatial objects that intersect with the given search geometry. A ***withinDistance(search geometry, distance)*** query (or range query) returns all spatial objects that are within the given *distance* from the the given search geometry.

2.2 Hierarchical Spatial Data Structure

For our spatial indexing, we utilize a hierarchical spatial data structure, called *geohash* [2], which is a geocoding system for latitude and longitude. A geohash code, represented as a string, basically denotes a rectangle (bounding box) on the earth. It provides a spatial hierarchy and it can reduce the precision (i.e., represent a bigger rectangle) by removing characters from the end of the string. In other words, the longer the geohash code is, the smaller the bounding box represented by the code is. Another property of geohash is that two places with a long common geohash prefix are close each other. Similarly, nearby places *usually* share a similar prefix. However, it is not always guaranteed that two close places share a long common prefix.

2.3 Distributed Storage Systems

A growing number of non-relational distributed databases (often called NoSQL databases) are proposed and widely used in many big data applications and analytics because they are designed to run on a large cluster of commodity hardware and fault-tolerant through data replication. One representative category of the NoSQL databases is the key-value store, in which data is stored in a schema-less way via an unique key that represents each row, such as Apache HBase, Apache Accumulo, Apache Cassandra, Google BigTable, Amazon DynamoDB, just to name a few. In this paper, our description is based on HBase, an open-source key-value store (or wide column store) originally derived from BigTable, because it is widely used by many big data applications. However, we believe that our spatial index is applicable to other key-value stores similarly because we use only keys for our index without modifying the internal structure of HBase.

2.4 Related Work

We classify existing spatial query processing techniques using distributed computing frameworks into two categories, based on their query types. The first category handles high selectivity queries, such as selection queries and kNN queries, in which only a small portion of spatial objects are returned as the result of spatial query processing. A few techniques have been proposed to process the high selectivity queries in HDFS [5, 6]. They are utilizing popular spatial indices such as an R-tree and its variants. The second category handles low selectivity queries which usually require at least one full scan of each dataset. One of the most representative low selectivity spatial queries is k nearest neighbor join (kNN join) which is to find, for each object in a dataset A, its k nearest neighbors in another dataset B. Several techniques have been proposed to process the kNN (or similar) joins using the MapReduce framework [7, 11, 4, 12].

3. SPATIAL QUERY PROCESSING

A spatial object basically includes its geometry and can have any additional information about the object, such as its name, address and phone number. In terms of the geometry, our spatial index supports most of generally used geometries including points, lines, rectangles, curves and polygons. Given a spatial object to be stored and indexed by our spatial index, we first calculate a set of minimum bounding boxes (i.e., geohash codes), called *minimum geohash set*, which fully cover the geometry of the spatial object. To prevent generating too many fine-grained bounding boxes to cover the geometry and thus increasing the overhead of managing the spatial object, we set the maximum number of bounding boxes for each geometry to 10 in the first prototype of our spatial index. The maximum number of bounding boxes for each geometry can be configured for different applications. Also, all the geohash codes included in a minimum geohash set have the same length and thus represent the same precision.

Similar to other indexing techniques such as R-trees, the query processing based on our spatial index basically consists of two main steps: *filter* step and *refinement* step. Given a spatial query Q, in the filter step, we find candidate spatial objects, which may satisfy the query condition of Q, by pruning non-qualifying spatial objects. In the refinement step, we examine each candidate spatial object to determine whether the object is actually satisfying the query condition of Q. We define the *precision* of query processing for Q as the ratio of actual spatial objects satisfying the query condition of Q to all evaluated candidate spatial objects.

To develop our spatial index on top of HBase, we propose to utilize HBase row keys to indicate the geohash codes for stored spatial objects. Specifically, given a spatial object SO to be stored and indexed by our spatial index, for each geohash code in its minimum geohash set $minGeohash(SO)$, we store the spatial object in the HBase row having the geohash code as its row key. We use an uniquely assigned identifier for the object as its column name (qualifier). We allow replication of spatial objects in multiple HBase rows for efficient processing of spatial queries as we will explain below. For example, if the minimum geohash set of a spatial object is {"dn5bpsby", "dn5bpsbv"}, we store the spatial object in two HBase rows whose keys are "dn5bpsby" and "dn5bpsbv". Note that our replication of spatial objects is not related to the data block replication of underlying HDFS for its fault-tolerance.

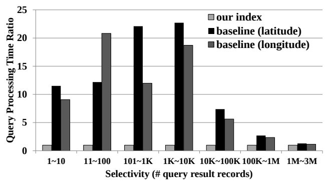
According to the definition of the geohash, longer geohash codes will be generated for smaller geometries. If there are many spatial objects associated with a tiny geometry, a huge number of HBase rows having a long row key may be created to store the objects and each row will likely include only a few spatial objects. Since too many HBase rows can aggravate the performance of our spatial query processing, we need to control the number of HBase rows. To limit the number of HBase rows, we utilize the hierarchical feature of the geohash codes. By setting the maximum length of geohash codes (i.e., length of HBase row keys), we can store those spatial objects associated with a tiny geometry in HBase rows representing a bigger rectangle and thus reduce the number of HBase rows.

To execute spatial queries for the stored and indexed spatial objects in HBase, we utilize the properties of the geohash codes to find only relevant HBase rows and thus reduce the search space considerably. Let us assume that a spatial query Q with its search geometry Q_G is given. We first calculate the minimum geohash set of Q which fully cover Q_G. If the query is *containing(search geometry)*, we select only those HBase rows whose row key is a prefix of one of the geohash codes in the minimum geohash set. This is because those spatial objects which contain the search geometry should have at least the same or larger rectangles than the search geometry. As we explained above, a geohash code representing a rectangle is a prefix of those geohash codes representing the sub-rectangles of the rectangle. Therefore, we can efficiently select candidate HBase rows which may store spatial objects containing the search geometry, using the prefix match. Specifically, to find candidate HBase rows, we scan all possible prefixes for each geohash code in the minimum geohash set. For example, for a geohash code "dn5b" included in the minimum geohash set, we scan for key "d", "dn", "dn5" and "dn5b". Finally, for each candidate HBase row, we read all spatial objects stored in the row and return those spatial objects which *actually* contain the search geometry.

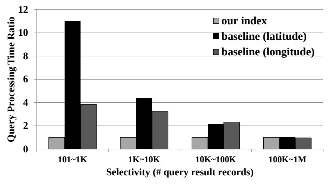
If the query is *containedIn(search geometry)*, an intuitive approach is to select only those HBase rows whose row key includes one of the geohash codes, included in the minimum geohash set, as its prefix because *containedIn* is the converse of *containing*. However, we need to take into account that we set the maximum length of geohash codes to prevent generating too many small HBase rows. For example, let us assume that the minimum geohash set of a spatial object is {"dn5bpsby"} and the spatial object is stored in a HBase row whose row key is "dn5bp" because the maximum length of geohash codes is 5. Also, assume that a *containedIn(search geometry)* query in which the minimum geohash set of the search geometry is {"dn5bpsb"} is given and the search geometry actually contains the spatial object. Based on the intuitive approach, we cannot select the HBase row "dn5bp" because "dn5bp" does not include "dn5bpsb" as its prefix. To tackle this problem, we also apply the maximum length to the geohash codes included in the minimum geohash set of the spatial query (from "dn5bpsb" to "dn5bp" in the previous example) and then use the intuitive approach. When we select candidate HBase rows whose row key includes one of the geohash codes, included in the minimum geohash set of the spatial query, as its prefix, we utilize a range scan of HBase for each geohash code. Specifically, for each geohash code included in the minimum geohas set, we execute a range scan whose start row is the geohash code and end row is the lexicographically next geohash code, having the same length, to access all HBase rows whose row key has the geohash code as its prefix. For example, for a geohash code "dn5b", we execute a range scan from "dn5b" to "dn5c". For each selected HBase row, we read the stored spatial objects in the row and return those spatial objects which are *actually* contained in the search geometry.

If the query is *intersects(search geometry)*, we consider both prefix cases when we select candidate HBase rows. This is because, if there is any intersecting region between the search geometry and the geometry of a spatial object, both geometries should have a rectangle(s) (i.e., geohash code) which includes the intersecting region and any two different rectangles including the same region should have their hierarchy (i.e., one is the sub-rectangle of the other) according to the definition of the geohash codes. Since we do not know which geometry has a bigger rectangle covering the intersecting region until we evaluate the spatial object, we select those HBase rows, as candidate rows, whose row key is a prefix of one of the geohash codes included in the minimum geohash set of the spatial query *or* includes one of the geohash codes as its prefix. For each selected HBase row, we read the stored spatial objects in the row and return those spatial objects which are *actually* intersecting with the search geometry.

For a *withinDistance(search geometry, distance)* query, we first calculate the minimum geohash set which covers the extended geometry computed by adding the *distance* to the search geometry. Then, similar to the *intersects* query processing, we select those HBase rows, as candidate rows, whose row key is a prefix of one of the geohash codes included in the minimum geohash set *or* includes one of the geohash codes as its prefix. For each selected HBase row, we read the stored spatial objects in the row and return those spatial objects which are *actually* within the *distance* from the the search geometry.



(a) *withinDistance* queries



(b) *containedIn* queries

Figure 1: Query Processing Time

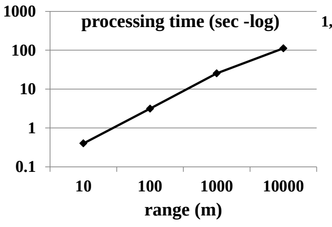
4. EXPERIMENTAL EVALUATION

For evaluation of our spatial index on top of HBase, we use HBase (Version 0.96) and Hadoop (Version 1.0.4) running on Java 1.6.0, installed on a cluster of 11 physical machines (one master machine) on Emulab [10]: each has 12GB RAM, one 2.4 GHz 64-bit quad core Xeon E5530 processor and two 7200 rpm SATA disks (500GB and 250GB). We run HBase RegionServers on the same machines as DataNodes and a ZooKeeper ensemble of 3 machines. For each setting and each query, our spatial query processing time indicates the fastest time after running five cold runs to remove any possible bias posed by OS and/or network activity. We use GeoLife GPS Trajectories (GeoLife in short) [13] and San Francisco taxi cab traces (SFTaxi in short) [9] for our experiments. GeoLife and SFTaxi contain 24,876,977 and 11,219,955 GPS point records respectively.

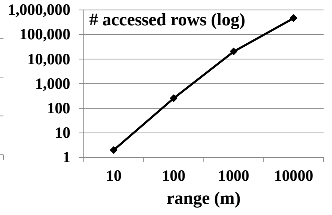
We first present spatial query processing performance using our index on top of HBase running on HDFS. As our baseline approach, we store the spatial objects using their latitude (or longitude) as a row key of HBase (i.e., one dimensional index). We choose this approach as our baseline because it can be also implemented without modifying HBase and, similar to our spatial index, HBase range scans can be utilized for fair comparisons. For example, given a *containedIn* query, we use the leftmost and rightmost latitudes (or longitudes) of the query geometry as the start and end row keys of a HBase range scan respectively.

We implement a Hadoop MapReduce job to efficiently store the spatial objects in HBase. Also, we represent each geohash code as a binary array, instead of a string, to efficiently handle geohash codes. By default, we empirically choose 40 bits as the maximum length of geohash codes because we think that value strikes a balance between the number of rows and the number of columns of each row. 2,608,848, 4,744,257 and 4,886,185 HBase rows are generated to store the spatial objects using our index, the latitude-based baseline approach and longitude-based baseline approach respectively.

In this paper, we report the results of *withinDistance* and *containedIn* queries. We generate 300 *withinDistance* queries by randomly selecting a point in the datasets and using a distance of 10m, 100m or 1km. This generation process guarantees that we get at least one point record as the output of each query execution. We also generate 100 *containedIn* queries by randomly selecting two points in the datasets and using them as the lower-left and upper-right points of a rectangle.



(a) Processing Time



(b) # accessed rows

Figure 2: Effects of different distances

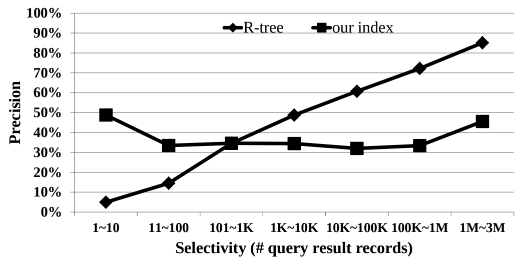


Figure 3: Precision comparison (*withinDistance*)

For brevity, we first categorize the queries based on their selectivity and then compare our query processing performance with that of the baseline approach using the ratio of their query processing times where we set our query processing time to 1, as shown in Fig. 1. The query processing with our spatial index is more than one order of magnitude faster than both the latitude-based and longitude-based baseline approaches, on average, for those *withinDistance* queries which select less than 10,000 records, as shown in Fig. 1(a). As we decrease the selectivity of queries, the performance gain of our spatial index also drops because retrieving a large number of rows for query evaluation is inevitable. However, the query processing with our spatial index is still 30% faster than the latitude-based baseline approach, on average, for those *withinDistance* queries which select more than 1 million records. For *containedIn* queries, even though our query processing is still more than one order of magnitude faster than the latitude-based baseline approach for queries having high selectivity as shown in Fig. 1(b), its performance gain is generally smaller than that for *withinDistance* queries. This is primarily because *containedIn* queries usually cover a wider region than *withinDistance* queries and thus the pruning power of the baseline approaches is higher for *containedIn* queries. Specifically, the average precisions (i.e., the ratio of true positives to all evaluated candidate spatial objects) of the latitude-based baseline approach are 8% and 12% for *withinDistance* queries and *containedIn* queries respectively.

Fig. 2 shows the query processing results using different distances for the same query point of a *withinDistance* query. The query processing time understandably increases as we enlarge the query region because more HBase rows are accessed and thus more candidate records are evaluated for query processing.

Finally, we compare the pruning power of our spatial index with that of an R-tree-based index. We use an open source R-tree implementation [3] for this evaluation. We want to emphasize that the focus of this paper is on the scalable and lightweight spatial index which can be easily applied to existing systems without modifying their internal implementation. Outperforming the pruning power of R-tree-based indices is not the purpose of this paper because R-tree-based indices maintain expensive data structures and mostly require internal and complicated modification of the storage systems. Nevertheless, the precision results in Fig. 3 show that our index has one order of magnitude higher precision than the R-tree-based index for those queries having very high selectivity (selecting less than 10 records). Our spatial index demonstrates relatively consistent precision for different selectivity levels while the R-tree-based index has higher precision for less selective queries.

5. CONCLUSION

In this paper we have proposed efficient and scalable spatial indexing techniques for big data stored in distributed storage systems. Based on a hierarchical spatial data structure, called geohash, we have presented how we develop a lightweight spatial index for big data stored in a distributed file system, especially on top of HBase.

6. ACKNOWLEDGMENTS

This work was performed while Kisung Lee was an intern at IBM Research T.J. Watson.

7. REFERENCES

[1] Geo Developer Guidelines. https://dev.twitter.com/terms/geo-developer-guidelines.

[2] Geohash. http://geohash.org/.

[3] JSI RTree Library. http://jsi.sourceforge.net/.

[4] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-Based Geospatial Query Processing with MapReduce. In *CLOUDCOM '10*, 2010.

[5] H. Liao, J. Han, and J. Fang. Multi-dimensional Index on Hadoop Distributed File System. In *NAS '10*, 2010.

[6] X. Liu, J. Han, Y. Zhong, C. Han, and X. He. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. In *CLUSTER '09*, 2009.

[7] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient Processing of K Nearest Neighbor Joins Using MapReduce. *Proc. VLDB Endow.*, 5(10), June 2012.

[8] D. Papadias, T. Sellis, Y. Theodoridis, and M. J. Egenhofer. Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees. *SIGMOD Rec.*, 24(2), May 1995.

[9] M. Piorkowski, N. Sarafijanovoc-Djukic, and M. Grossglauser. A Parsimonious Model of Mobile Partitioned Networks with Clustering. In *COMSNETS'09*.

[10] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *SIGOPS Oper. Syst. Rev.*, 36, 2002.

[11] C. Zhang, F. Li, and J. Jestes. Efficient Parallel kNN Joins for Large Data in MapReduce. In *EDBT '12*, 2012.

[12] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. Spatial Queries Evaluation with MapReduce. In *GCC '09*, 2009.

[13] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining Interesting Locations and Travel Sequences from GPS Trajectories. In *WWW '09*, 2009.