Voronoi-based Geospatial Query Processing with MapReduce

Afsin Akdogan, Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi
Integrated Media Systems Center, University of Southern California
Los Angeles, CA 90089
[aakdogan, demiryur, banaeika, shahabi]@usc.edu

Abstract

Geospatial queries (GQ) have been used in a wide variety of applications such as decision support systems, profile-based marketing, bioinformatics and GIS. Most of the existing query-answering approaches assume centralized processing on a single machine although GQs are intrinsically parallelizable. There are some approaches that have been designed for parallel databases and cluster systems; however, these only apply to the systems with limited parallel processing capability, far from that of the cloud-based platforms. In this paper, we study the problem of parallel geospatial query processing with the MapReduce programming model. Our proposed approach creates a spatial index, Voronoi diagram, for given data points in 2D space and enables efficient processing of a wide range of GQs. We evaluated the performance of our proposed techniques and correspondingly compared them with their closest related work while varying the number of employed nodes.

1. Introduction

With the recent advances in location-based services, the amount of geospatial data is rapidly growing. Geospatial queries (e.g., nearest neighbor queries and reverse nearest neighbor queries) are computationally complex problems which are time consuming to solve, especially with *large datasets*. On the other hand, we observe that a large variety of geospatial queries are intrinsically *parallelizable*.

Cloud computing enables a considerable reduction in operational expenses by providing *flexible* resources that can instantaneously scale up and down. The annual global market for cloud computing is estimated to surpass $100 billion in three years [3]. Thus, the big IT vendors including Google, IBM, Microsoft and Amazon are ramping up parallel programming infrastructures. Google's MapReduce programming model [1] provides a parallelization for processing large datasets and can do that at a very large scale. For example, Google processes 20 petabytes of data per day with MapReduce [20]. Given recent availability of cloud services and intrinsic parallel nature of geospatial queries, a variety of geospatial queries can be efficiently modeled using MapReduce. Suppose we want to answer a reverse nearest neighbor query (RNN). Given a query point q and a set of data points P, RNN query retrieves all the data points that have q as their nearest neighbors [9]. Each point p_i finds its nearest neighbor p_k efficiently in parallel and emits <p_k, p_i> in the map phase. Subsequently, all points having the same key, p_k will be grouped in the reduce phase by indetifying p_k's RNNs without any extra step.

Parallel spatial query processing has been studied in the contexts of parallel databases, cluster systems, and peer to peer systems as well as cloud platforms. The Paradise project developed at University of Wisconsin [14] runs on top of a parallel database, and consists of a query coordinator and one or more data servers. This architecture is not completely decentralized because of the single query coordinator that directs the query to the right data servers. The data points are partitioned among all data servers in the system using round-robin and hashing to provide a basic parallelism. However, the points in each partition are not locally indexed.

Meanwhile, several approaches that use distributed hierarchical index structures such as R-tree based P2PR-Tree and SD-Rtree [17, 18], kd-tree based k-RP [7], quad-tree based hQT* [15] have been proposed for parallel spatial query processing. The major problems with the tree-based approaches are as follows. First, they do not scale due to the traditional top-down search that overloads the nodes near the tree root, and fail to provide full decentralization. Whereas loose coupling and shared nothing architecture, which are widely believed to scale the best [19], make MapReduce clusters highly scalable. Second, most of these approaches support only the first Nearest Neighbor query.

There are a few recent studies that handle geospatial queries using MapReduce in the cloud computing context. However, these are not comprehensive studies. They either only construct R-tree index with MapReduce without supporting any type of query [13], or motivate the problem of geospatial query processing without specifically explaining how to process the queries [12]. Moreover, hierarchical indices like R-tree are structurally not suitable for MapReduce. Because processing a nearest neighbor query on a distributed R-tree requires navigation of the tree from top to bottom; and performing such a traversal needs the nodes to message to each other. Whereas, the nodes in a MapReduce cluster do not communicate with each other.

In this paper, we propose a MapReduce-based approach that both constructs a *flat* spatial index, Voronoi diagram (VD), and enables efficient parallel processing of

a wide range of spatial queries including reverse nearest neighbor (RNN), maximum reverse nearest neighbor (MaxRNN) and k nearest neighbor (kNN) queries. These types of queries are widely used in decision support systems, profile-based marketing, bioinformatics and GIS. For example, RNN query can help a franchise (e.g. Starbucks) decide where to put a new store in order to maximize the benefit to its customers. The experiments show that our approach outperforms its closest related work and scales well with the increasing number of nodes. For example, a VD can be generated by merging partial Voronoi Diagrams (PVD). Specifically, each mapper creates a PVD and a reducer combines all the PVDs to obtain a single VD. Clearly, dividing the most computationally complex piece of the algorithm, namely construction of PVDs, across multiple mappers improves the performance. To the best of our knowledge, our work is the first detailed attempt in processing geospatial queries with MapReduce in the cloud computing context.

The remainder of this paper is organized as follows. Section 2 provides the background, and subsequently Sections 3 and 4 discuss the construction of Voronoi diagrams and the query processing methods, respectively. Section 5 presents the results of the experiments to verify and validate the performance and scalability of our proposed approach. Finally, Section 6 concludes the paper with the directions for future work.

2. Background

Before we present our geospatial query processing approach, in this section we briefly introduce Voronoi Diagrams and MapReduce to prepare for the rest of the discussion.

2.1. Voronoi diagrams

A Voronoi diagram decomposes a space into disjoint polygons based on the set of generators (i.e., data points). Given a set of generators S in the Euclidean space, Voronoi diagram associates all locations in the plane to their closest generator. Each generator s has a Voronoi polygon consisting of all points closer to s than to any other site. Hence, the nearest neighbor of any query point inside a Voronoi polygon is the generator of that polygon. The set of Voronoi polygons associated with all the generators is called the Voronoi diagram (VD) with respect to the generators set. The polygons are mutually exclusive except for their boundaries.

DEFINITION 1 Voronoi Polygon.

Given set of generators $P = \{p_1, p_2,\ldots,p_n\}$ *where* $2 < n < \infty$ *and* $p_i \neq p_j$ *for* $i \neq j$, $i, j = 1;\ldots, n$, *the Voronoi Polygon of* p_i *is* $VP(p_i) = \{p \mid d(p, p_i) \leq d(p, p_j)\}$ *for* $i \neq j$ *and* $p \in VP(p_i)$ *where* $d(p, p_i)$ *specifies the minimum distance between p and* p_i *in Euclidean space.*

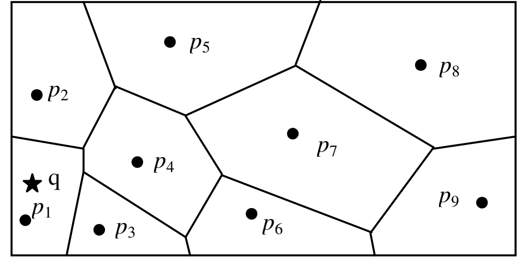


Figure 1: A sample Voronoi diagram

Figure 1 illustrates an example of a Voronoi diagram and its polygons for nine generators. Various approaches have been proposed to generate *VD* in Euclidean space. We use *divide and conquer* approach to efficiently compute *VD* [2]. Here we enlist VD's main properties that we use to establish our proposed solutions. The proofs for these properties can be found in [2].

Property 1: The Voronoi diagram for given set of generators is unique.

Property 2: Let n and n_e be the number of generators and Voronoi edges, respectively, then $n_e \leq 3n-6$.

Property 3: From property 2, and the fact that every Voronoi edge is shared by exactly two Voronoi polygons, the average number of Voronoi edges per Voronoi polygon is at most 6, i.e., $2(3n-6)/n = 6n-12/n \leq 6$. This states that on average, each generator has 6 adjacent generators. This property is especially useful to limit the number of candidate generators during kNN search.

Property 4: The nearest generator point of *pi* (e.g., *pj*) is among the generator points whose Voronoi polygons share Voronoi edges with *VP*(*pi*).

2.2. MapReduce

MapReduce [1] is a popular programming model for parallel processing of large datasets. Programs written with this model are automatically parallelized and executed on thousands of commodity machines, collectively referred to as a cluster. Since it is a simple but yet a powerful model, it has been used in a wide variety of application domains such as machine learning, data mining and search-engines.

Hadoop is a popular open source implementation of the MapReduce, which runs on top of a distributed storage system called Hadoop File System (HDFS) [8]. In HDFS, a file is broken into multiple blocks, called *split*, which are then distributed among the machines in the cluster. All splits are of the same size with the exception of the last one, and the split size is configurable per file. MapReduce consists of two user-defined functions, namely a map function and a reduce function. Given a MapReduce task, with Hadoop first each mapper is assigned to one or more splits depending on the number of machines in the cluster. Second, each mapper reads inputs provided as a <key, value> pair at a time, applies

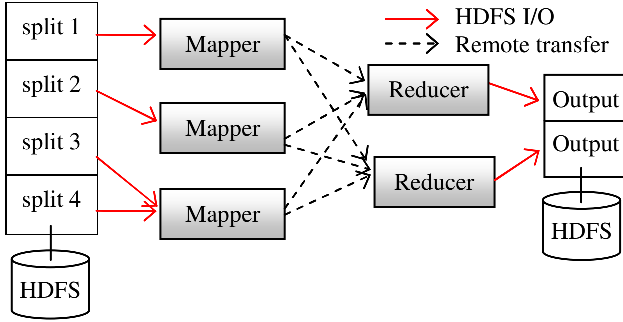


Figure 2: The data flow in Hadoop Architecture

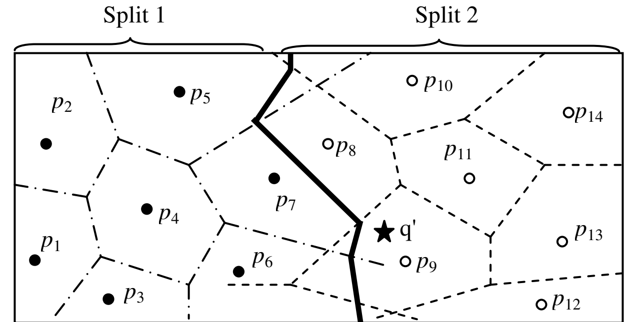
the map function to it and generates intermediate <*key, value*> pairs as the output. Finally, reducers fetch the output of the mappers and merge all of the intermediate values that share the same intermediate key, process them together and generate the final output. The input of mappers and the output of reducers are stored on HDFS. Figure 2 shows how Hadoop processes the data of four split in parallel where there are three map machines and two reduce machines.

3. Constructing Voronoi Diagram with MapReduce

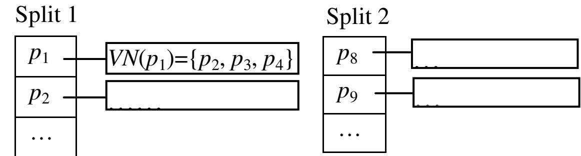
Voronoi Diagram construction is inherently suitable for MapReduce modeling because a *VD* can be obtained by merging multiple partial Voronoi diagrams (*PVD*). Specifically, each of *PVD*s can be created by the mappers in parallel and the reducer can combine them into a single *VD*. In this section, we first show how a *VD* is built with MapReduce using *divide and conquer* method. Second, we discuss the modification that we did (in Hadoop's map phase) to improve sequential reading of inputs.

One of the techniques used to generate a Voronoi diagram is *divide and conquer* paradigm. The main idea behind the divide and conquer method is as follows. Given a set of data points *P* as an input in Euclidean space, first the points are sorted in increasing order by X coordinate. Next *P* is separated into several subsets of equal size. Subsequently, a *PVD* is generated for the points of each subset, and then all of the *PVD*s are merged to obtain the final *VD* for *P*.

MapReduce Solution: Given a set of data points sorted by X coordinate, each mapper reads an input split in the format of <data_point, dummy_value> (i.e., <key, value> pair). Note that the dummy_value does not have any purpose, i.e., it is used to follow the input format of MapReduce. Subsequently, each mapper generates a *PVD* for the data points in its split, marks the boundary polygons to be later used in the merge phase and emits the generated *PVD*s in the form of <constant_key, PVD_i> where *i* denotes the split number. The constant_key is common to all *PVD*s, so that all *PVD*s can be grouped together and merged in the subsequent reduce step.



(a)



(b)

Figure 3: a) Merger of left and right *PVD*s b) A Voronoi diagram distributed on HDFS which will read by the mappers as input

Finally, a single reducer aggregates all *PVD*s in the same group and combines them into a single *VD*. In the merger phase, the boundary cells are detected first with a sequential scan, and then new Voronoi edges and vertices are generated by deleting superfluous boundary portions from *PVD*s. As the final output, the reducer emits each point and its Voronoi neighbors (*VN*).

Figure 3a illustrates parallel *VD* generation with two mappers and one reducer. Given a set of data points P= $\{p_1, p_2, \ldots, p_{13}, p_{14}\}$ as input, first the points are sorted, then they are divided into two splits equal in size. The first mapper reads the data points in *Split 1*, generates a *PVD* and emits <1, PVD_1>. The key value "*1*" is constant for all mappers. Likewise the second mapper emits <1, PVD_2>. Next the reducer aggregates these *PVD*s and merges them into a single *VD*. The final output of the reducer is in the following <key, value> format: <p_i, $VNs(p_i)$> where $VNs(p_i)$ represents the set of Voronoi neighbors of p_i. For example, for point p_1 the output is < p_1, $VN(p_1)=\{p_2, p_3, p_4\}$> and for point p_{14} the output is < p_{14}, $VN(p_{14})=\{p_{10}, p_{11}, p_{13}\}$>. Analogously, a <key, value> pair is generated for each data point.

Once the reducer outputs the result, the output is also split into multiple blocks and scattered across the machines in the cluster to prepare for query processing. Suppose the output is divided into two equally sized splits. Consequently, the first split will contain the Voronoi polygons of $\{p_1, \ldots, p_7\}$ and the second will include those of $\{p_8, \ldots, p_{14}\}$. Figure 3b shows how the generated *VD* is stored on HDFS.

Modification of Hadoop: With Hadoop, a mapper reads only one <key, value> pair at a time from a split, and subsequently generates an intermediate output for that <key, value> pair. Once an output is generated for a <key,

value> pair, that pair cannot be accessed again by the mapper. With our modification on Hadoop's implementation, a mapper processes all data points in its splits simultaneously rather than one at a time. Therefore, Voronoi polygons (for all data points in a split) are constructed together in the map phase. This modification improves other types of operations as well. For example, in order to sum a set of numbers given as input, mappers emit the partial sums of the numbers in their splits. Then a reducer combines all partial sums and emits the final result. However, without the modification discussed above, mappers only collect all the numbers in the same group and a reducer sums up all the numbers. Thus, a single reducer will be overloaded and mappers will stay idle.

4. Query processing

In this section, we discuss our proposed MapReduce-based approaches to answer a variety of spatial queries using Voronoi diagrams. For each query type, we first define the problem and then discuss our approach.

4.1. Reverse Nearest Neighbor Query (RNN)

Given a query point q and set of data points P, reverse nearest neighbor (RNN) query retrieves all data points p ∈ P that have q as their nearest neighbors [9]. There are two cases of RNN queries, namely, monochromatic RNN and bichromatic RNN. In this paper we focus on monochromatic RNN (MRNN) which has been extensively studied in spatial databases [5, 9]. With MRNN, all objects are of the same type. Therefore, a point p is considered an RNN of q if there is no other point p′ where d (p, p′) ≤ d (p, q). RNN query has been used in a wide range of applciations such as decision support systems, profile-based marketing, bioinformatics, and optimal location. For example, McDonald's has a marketing application in which the issue is to determine the business impact of restaurants to each other. A simple task would be to determine the customers who would be likely to use restaurants.

Figure 4 illustrates the computation of reverse nearest neighbor for a given set of data points in 2D space where each data point is considered a query point as well. The search expands from each point p_i simultaneously until p_k, the point closest to p_i, is found. Performing this task for each point p_i results in a circle centered at p_i with a radius $|p_i, p_k|$. Therefore, the nearest neighbor to each point lies on the perimeter of its corresponding circle. Given that p_k is p_i's nearest neighbor, p_k's reverse nearest neighbor includes p_i and possibly other points whose nearest neighbors are p_k as well. For example, p_5 is the nearest neighbors of p_2, p_3 and p_4 in Figure 4; therefore, p_5's RNNs are p_2, p_3 and p_4.

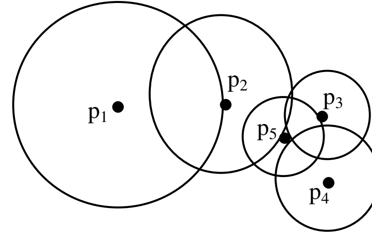


Figure 4: The reverse nearest neighbor query

MapReduce Solution: The RNN problem defined above is intrinsically applicable to MapReduce. Given that each mapping operation is independent of the others and all maps can be performed in parallel, first each point p_i finds its nearest neighbor p_k efficiently and emits <p_k, p_i> in the map phase. Next, all points having the same key, p_k will be grouped in the reduce phase by indetifying p_k's RNNs without any extra step. The detailed map and reduce steps for processing an RNN query is as follows. Suppose the input to the map function is:

<p, Voronoi Neighbors (*VN*s)>, $\forall p \in P$

Map: Each point p_i finds p_k as its nearest neighbor. Recall that the nearest neighbor of a point lies within its *VN*s (see Property 4) and on average the number of *VN*s cannot exceed 6 (see Property 3). Therefore, in order to find the nearest neighbor we need to check only 6 points on average. Once p_k is found, the mapper emits the pair <p_k, p_i>, which states that p_k's reverse nearest neighbors include p_i.

Reduce: The reducer aggregates all pairs with the same key p_k (i.e., the result of each nearest neighbor search) and emits a set <$p_{k1}, p_{k2}, \ldots, p_{km}$>, which contains all points p_{ij} $0<j<m+1$ whose nearest neighbor is p_k.

For example, p_2, p_3 and p_4 all have p_5 as their nearest neighbor in Figure 4. In the map phase, these points will locate p_5 as the nearest neighbor and emit the <key, value> pairs: <p_5, p_2>, <p_5, p_3>, <p_5, p_4>, respectively. Then in the reduce phase, every point with the same key, p_5 will be aggregated as <p_5, $\{p_2, p_3, p_4\}$>. This group forms the RNNs of p_5.

4.2. Maximizing Reverse Nearest Neighbor (MaxRNN)

A MaxRNN query [6] locates the optimal region *A* such that when a new point *p* is inserted in A, the number of RNNs for *p* is maximized. This query is also known as the *optimal location problem*. The optimal region can be found using nearest neighbor circles.

DEFINITION 2 Nearest Neighbor Circle (NNC)
Given a point p and its nearest neighbor p′, NNC of p is the circle centered at p′ with radius |p, p′|.

Given a set of data points P, each point p_i finds its nearest neighbor p_k, and computes an NNC based on p_k. The region which is intersected by the highest number of

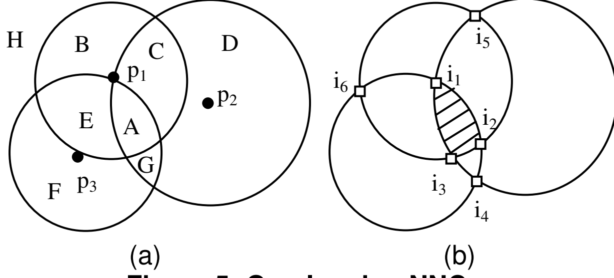


Figure 5: Overlapping NNCs.

NNCs is the answer to MaxRNN query. Figure 5a shows an example of MaxRNN with three points $\{p_1, p_2, p_3\}$. The nearest neighbors of p_1, p_2 and p_3 are p_3, p_1 and p_1, respectively. The boundaries of NNCs decompose the space into disjoint regions labeled as A, B, C, D, E, F, G and H. As shown in Figure 5b, region A, the overlapping region of all circles, is maximizing the size of RNN for the new point and is represented by a set of intersection points of NNCs which are covered by the maximum number of circles, i.e. $\{i_1, i_2, i_3\}$.

MapReduce Solution: The main motivation behind parallelizing a MaxRNN query with MapReduce is that MaxRNN queries need to process large datasets in its entirety which may result in an unreasonable response time. For example, in a recent study it has been showed that the computation of a MaxRNN query with 200,000 points takes several hours [6]. Whereas, our MapReduce-based approach can answer the same query in a few minutes with 32 nodes and the performance can be improved as more nodes join the cluster (see Section 5.2.3).

With our approach, we compute a MaxRNN query in two MapReduce steps where the output of each step is given to the following step as input. First step finds the nearest neighbors of every point and computes the radiuses of the NNCs. The second step finds the intersection points that represent the optimal region. The detailed map and reduce steps are as follows. Suppose the input to the map function is in the following form:

<p, Voronoi Neighbors (*VN*s)>, $\forall p \in P$.

Step 1: Each point p finds p' as its nearest neighbor among the *VN*s, and sets $|p, p'|$ as the radius r of its NNC. Subsequently, for each point p_{vn} in the *VN*s of p, <p_{vn}, $\{p, r\}$> (i.e. <key, value>) is emitted as the intermediate output by the mappers. Subsequently, the reducers aggregate the intermediate output, such that for every p and its *VN*s a radius value is set. The final output of the reduce phase is as follows.

<(p, r), {Voronoi Neighbors with radius of their NNCs}>, $\forall p \in P$

Step 2: First, each point p checks its *VN*s in order to find the NNCs with which it overlaps using the radius values computed in the first step. Two circles centered at p_1 and p_2 with radiuses r_1 and r_2 overlap if $|p_1, p_2| < r_1 + r_2$.

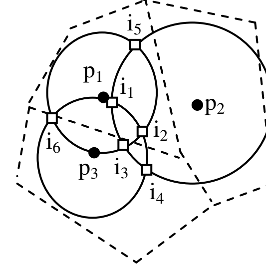


Figure 6: MaxRNN computation

We expand our search to neighbors' neighbors in stages as follows. For each point p_{vn} in the *VN*s of p, the mappers emit <p_{vn}, $\{VNs - p_{vn}\}$> as output. Subsequently, the reducers receive the input from the mappers and emit it as is. This expansion is repeated until all overlapping NNCs are found for every point p. Once the expansion is terminated, the mappers compute the intersection points.

Subsequently, we count the number of NNCs covering each of the intersection points to identify the weights. An intersection point i can only be covered by the NNCs which overlap the NNC on which i exists. Since we have already found all NNCs at the end of the expansion phase, we find the weights of each i and emit <contant_key, (i, w(i))> pairs as output of the map phase where w(i) represents the weight of the intersection point i. At the concluding reduce step, all intersection points are grouped together since they have the same constant_key, and then the highest weigthed intersection points are found and emitted as the final output.

Figure 6 shows NNCs and the underlying Voronoi polygons with three data points. At the first step, p_1 finds p_3 as the nearest neighbor among its *VN*s $\{p_2, p_3\}$ and sets $|p_1, p_3|$ as the radius of its NNC. The final output of the first MapReduce step for p_1 is <(p_1, $|p_1, p_3|$), {(p_2, r_2), (p_3, r_3)}>. In the map phase of the second step, p_1 checks its *VN*s $\{p_2, p_3\}$, finds that it overlaps $\{p_2, p_3\}$ and computes the intersection points $\{i_2, i_3, i_5, i_6\}$. The search for p_1 can expand to neighbors' neighbors, namely neighbors of $\{p_2, p_3\}$; however, for the sake of clarity we do not demonstrate the expansion phase. Each intersection point i then computes its weight by counting the NNCs covering i. For example, i_2 is covered by the NNCs of $\{p_1, p_2, p_2\}$, and $w(i_2)=3$. For each intersection point i, the second map phase emits <1, (i, w(i))> pairs, i.e., <1, (i_2, 3)>, <1, (i_3, 3)>, <1, (i_5, 2)>, and <1, (i_6, 2>. The final reduce phase combines all pairs, finds the intersection points with the heighest weights who are emitted as the final output, i.e., $\{i_1, i_2, i_3\}$.

4.3. k Nearest Neighbor Query (kNN)

Given a query point q and a set of data points P, k Nearest Neighbor (kNN) query finds the k closest data points $p_i \in P$ to q where $d(q, p_i) \le d(q, p)$. Voronoi diagram has been showed to be an efficient method to

solve kNN problem [4]. The *first* NN is found by locating q into corresponding Voronoi polygon (*VP*). The *second* NN to *any* location inside a Voronoi polygon *VP* (p_i) is among the *VN*s of p_i (see Property 4). For example, in Figure 1 where the *first* NN of q is p_1, the *second* NN of q is among the *VN*s (i.e., p_2, p_3, p_4) of p_1. Suppose the *second* NN of q is p_2. Then the third nearest neighbor of q be among the Voronoi neighbors of $\{p_1, p_2\}$. Consecutive nearest neighbors of q can then be found using the same iterative approach.

MapReduce Solution: We address kNN problem with one MapReduce step if all NNs of q are in the same split. Additional steps might be required for some query points which have NNs in more than one split. The map and reduce steps are discussed below. Suppose the input to the map function is as follows:

<p, Voronoi Neighbors (*VN*s)>, $\forall p \in P$

Map: Suppose we answer a 3NN query with two mappers and one reducer for query point q_1 in Figure 7. First, each mapper reads its split. All data points in the split are put in a hash map where each point is a key and *VN*s are values. Subsequently, the point p_1, the first NN, is found where q_1 is inside *VP* (p_1). Notice that only one split can have the *VP*(p_1) and only one mapper processing that split can locate q. The second mapper can simultaneously process the query point q_2 and other queries as well. Clearly, processing a batch of query points simultaneosuly improves the throughput considerably. However, for simplicity, we will move on the discussion assuming that there is only one query point. The second NN is among the *VN*s of p_1, $\{p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$. Suppose the second NN is p_2. The third NN is among the *VN*s of $\{p_1, p_2\}$. The *VN*s of p_1 are already known. The neighbors of p_2 are efficiently found by using the hash map. A distance from q to every point in the set of *VN*s is calculated and the point with the minimum distance is found as the third NN, i.e., $\{p_5\}$. Once all the neighbors are found, the mapper emits the query point q as key and the neighbor list as value, i.e., <q_1, $\{p_1, p_2, p_5\}$>. In some cases, a query point can be close to the boundary of a split (see q_3 in Figure 7) and some of its nearest neighbors can be in the neighboring split. In order to handle this case, when a point is found as a nearest neighbor, we check each of its *VN*s in the hash map if they are in the current split. If not, we flag the missing points and reprocess them in an additional MapReduce step as described below.

Reduce: Receives the query point as key and its neighbors as value, and emits them as they are.

After the reduce step, the final output is checked if the query contains any flagged point. If some of the neighbors are in another split, an aditional MapReduce step is required. In the second map phase of such a case, the query comes with its current kNNs and flagged points. The flagged points are located in the Voronoi diagram and the new candidate points are found. Next, the new set

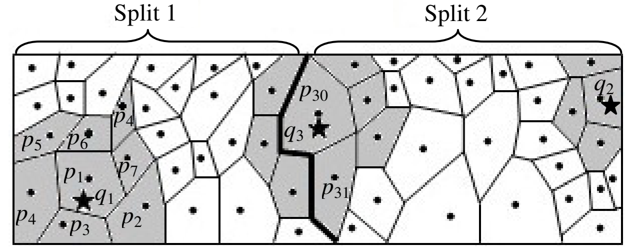


Figure 7: A visualization of processing multiple kNN queries on Hadoop

of kNNs is selected among the current kNNs and the candidate set. Finally, the mapper emits the new result.

5. PERFORMANCE EVALUATION

5.1. Experimental setup

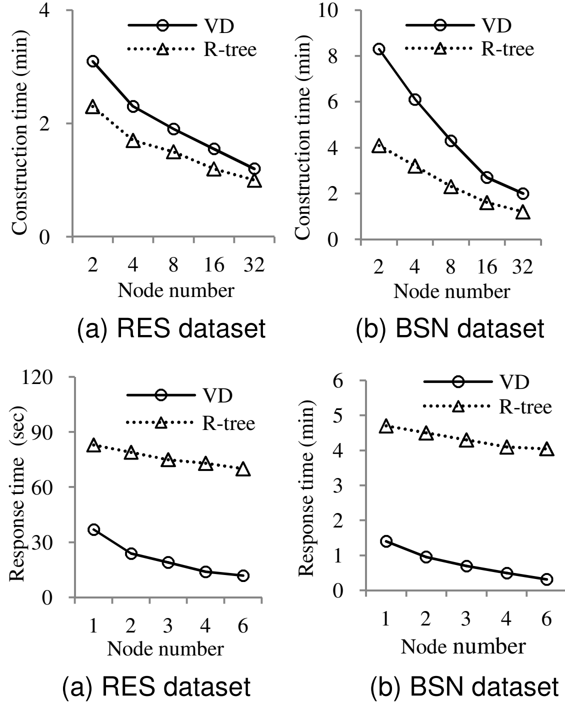
In our experiments, we used four real Navteq datasets that consist of all financial institutions (FIN), auto services (AUT), restaurants (RES), other businesses (BSN) in the entire U.S., containing approximately 190000, 250000, 450000 and 1300000 data points, respectively. With these datasets, we evaluated the performance of our proposed techniques and correspondingly compared them with their closest related work while varying the number of employed nodes in the cluster.

We conducted our experiments on the Amazon EC2 cluster with extra large instances that run on 64 bit Fedora 8 Linux Operating System with 15 GB memory, 4 virtual cores, and 4 disks with 1,690 GB storage. The I/O performance of EC2 varies from 40 MB/s to 140 MB/s during a day depending on the hour [10]. In order to obtain consistent results, we conducted all experiments in off-peak hours. All experiments are implemented using Hadoop version 0.20.1. The configuration of Hadoop can considerably influence the performance of the query processing techniques. Accordingly, we performed all of our experiments with the same setup, where the replication factor of each file is set to 1 in order to avoid writing overhead and to enable compression of the data transferred between mappers and reducers. Moreover, each node runs two map instances and one reduce instance.

5.2. Results

5.2.1. Voronoi-based Index

In this experiment, we compared the performance of our proposed indexing approach (VD) with the MapReduce-based R-tree [13] in terms of index construction time and query response time. Figure 8a illustrates the effect of varying the number of nodes in the cluster on the construction time of VD and R-tree, given the RES dataset. As shown, the construction times of both



(a) RES dataset (b) BSN dataset

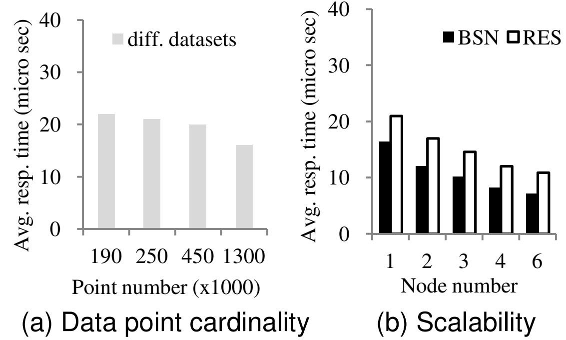
(a) RES dataset (b) BSN dataset

Figure 8: ab) VD & R-tree construction cd) NN query on VD & R-tree

index structures decrease almost linearly as more parallelism is induced by the new nodes. Construction of the R-tree index takes less time than that of VD. This is because the mappers generating *PVD*s execute expensive computations and there is only one reducer merging the *PVD*s in the cluster. Figure 8b depicts a similar observation with the BSN dataset. Even though R-Tree is faster to build, our experiments show that VD outperforms R-tree in query response time. Figure 8c and 8d depict the time to compute the nearest neighbor (as an example query) for every point in the RES and BSN datasets, respectively. As shown, in both cases VD is at least five times faster than R-tree and R-tree does not scale well. This is because with VD, points can immediatelly find their nearest neighbors in their VNs in the map phase; however, R-tree does not guarantee that nearest neighbor of a point p is in the same split with p. Thus, with R-tree a reducer always verifies the answer.

5.2.2. RNN

In this experiment, we study the impact of varying the number of nodes as well as size of the datasets on average response time of our proposed RNN query processing technique. Our approach processes RNN query for every data point simultaneously rather than a single data point. Figure 9a shows the average response time of processing a reverse nearest neighbor query for every data point on a single node without parallelism. As shown, the average response time slightly decreases with the increasing number of data points; hence, the overall throughput increases. With the state-of-the-art centralized approach, FITCH [16], an RNN query for a single data point runs in a few seconds given a similar dataset in size with RES. Clearly, processing an RNN query for all the data points in a dataset decreases the average response time by several orders of magnitude. Figure 9b illustrates the effect of node number on the average response time of processing an RNN query for BSN and RES datasets. As shown, the response time decreases linearly as more nodes join the cluster.



(a) Data point cardinality (b) Scalability

Figure 9: RNN response time

5.2.3. MaxRNN

In this section, we study the performance of our technique for MaxRNN query answering by varying node numbers with both BSN and RES datasets. We compare our Voronoi based approach with the best-known centralized algorithm, MaxOverlap [6], which uses R-tree index structure. Note that the response time of MaxRNN is high (in orders of minutes). Both approaches implement similar algorithms with different data structures; therefore, the performance on a single machine without parallelism is similar as well. However, our approach is parallelizable and shows linear scalability in response time utilizing VD. Figure 10 illustrates the execution time with varying node numbers and datasets. As shown, parallelization reduces the response time *at linear rate* for both datasets. As the number of nodes doubles, we observe a performance increase from %30 up to %50.

5.2.4. kNN

In this experiment, we compare our approach (VD) with the existing MapReduce based kNN search (MRK) [12] in terms of response time. We study the impact of k and query cardinality on response time. Figure 11a presents the response of kNN queries (with BSN dataset) processed on a single node for varying k. As shown, VD outperforms MRK significantly. This is because, for a given number of data points *P* and given number of query

points Q, MRK always outputs $P\times Q$ <key,value> pairs in the map phase independent of k. Due to the massive amount of data generated by the mappers, the response time is high. The effect of k is not a substantial factor for VD as long as all nearest neighbors of a query point is in the same split. Otherwise, additional MapReduce steps might be required. Thus, for larger k values, the chance of having nearest neighbors in more than one split is also higher. Figure 11b shows the impact of query cardinality with uniform distribution on response time. MRK does not scale with the increasing number of queries due to the same reason discussed above. As more queries are processed simultaneously, the throughput increases. This is because, before the queries are processed, the entire data is already fetched into the memory.

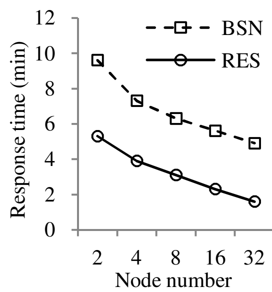
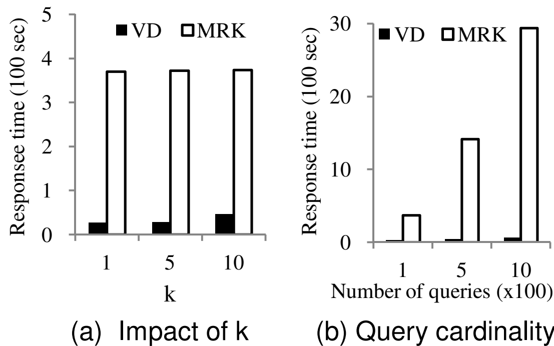


Figure 10. Effect of node number on MaxRNN



(a) Impact of k (b) Query cardinality

Figure 11: kNN

6. Conclusions

In this paper we investigated the problem of handling and effecient querying of massive spatial data in a cloud architecture. To this end, we have presented a distributed Voronoi index and techniques to answer three types of geospatial queries. Our experiments show that our Vdapproach significantly improves the performance, that the query types we have presented are suitable to solve with MapReduce, and a *linear scale-up* can be achieved in the response times as more nodes are used.

A natural next step is to support other types of queries such as bichromatic reverse nearest neighbor, skyline, reverse k nearest neighbor.

Acknowledgments. This research has been funded in part by an unrestricted cash gift from Google, in part by NSF grant CNS-0831505 (CyberTrust) and in part from METRANS Transportation Center, under grants from USDOT and Caltrans. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

7. References

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI, 2004

[2] A. Okabe, B. Boots, K. Sugihara, and S.N. Chiu. Spatial Tessellations: Concepts and Applications of Voronoi Diagrams. Wiley, 2000.

[3] K. Rangan.The Cloud Wars: $100+ billion at stake. Tech. Rep. Merrill Lynch, May 2008.

[4] M.R. Kolahdouzan and C. Shahabi. Voronoi-Based k Nearest Neighbor Search for Spatial Network Databases. Proceedings of the 30th Very Large DataBase, pages 840-851, 2004.

[5] Y. Tao, D. Papadias, and X. Lian. Reverse kNN Search in Arbitrary Dimensionality. In VLDB, 2004.

[6] R.C. Wong, M.T. Ozsu, P.S. Yu, A.W. Fu, and L. Liu. Efficient Method for Maximizing Bichromatic Reverse Nearest Neighbor. In VLDB, 1999.

[7] W. Litwin and M. A. Neimat. K-RP*S: A Scalable Distributed Data Structure for High-Performance Multi Attribute Access. In Proc. Intl. Conf. on Parallel and Distributed Inf. Systems (PDIS), pages 120-131, 1996.

[8] http://hadoop.apache.org/.

[9] F. Korn and S. Muthukrishnan. Influence Set Based on Reverse Nearest Neighbor Queries. In SIGMOD, 2000.

[10] D. Jiang, B. Chin, O. L. Shi, and S. Wu. The Performance of MapReduce: An In-depth Study. VLDB, 2010., in press.

[11] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-136, 2009.

[12] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Fend. Spatial Queries Evaluation with MapReduce. In International Conference on Grid and Cooperative Computing, 2009.

[13] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on Processing Spatial Data with MapReduce. In SSDBM, 2009.

[14] J.M. Patel. Building a Scalable Geospatial Database System. In SIGMOD, 1997.

[15] J. S. Karlsson. hQT*: A Scalable Distributed Data Structure for High-Performance Spatial Accesses. In Foundations of Data Organization and Algorithms (FODO), 1998.

[16] W. Wu, F. Yang, C.Y. Chan, and K.L. Tan. FITCH: Evaluating Reverse k-Nearest-Neighbor Queries on Location Data. VLDB, 2008.

[17] A. Mondal, Y. Lifu, and M. Kitsuregawa. P2PR-Tree: An R-Tree-Based Spatial Index for Peer-to-Peer Environments. In EDBT, 2004.

[18] C. Mouza, W. Litwin, and P Rigaux. SD-Rtree: A Scalable Distributed Rtree. In ICDE, 2007.

[19] S. Madden, D. Dewitt, and M. Stonebraker. Database parallelism choices greatly impact scalability. DatabaseColumnBlog.www.databasecolumn.com/2007/10/ database-paralelism-choices.html.

[20] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. Commun. ACM, 51(1):107-113,2008.