

---

RESEARCH PAPER

# SparkNN: A Distributed In-Memory Data Partitioning for KNN Queries on Big Spatial Data

Zaher Al Aghbari<sup>1</sup>, Tasneem Ismail<sup>1</sup> and Ibrahim Kamel<sup>2</sup>

<sup>1</sup> Department of Computer Science, University of Sharjah, AE

<sup>2</sup> Department of Computer Engineering, University of Sharjah, AE

Corresponding author: Zaher Al Aghbari (zaher@sharjah.ac.ae)

---

The increase in GPS-enabled devices and proliferation of location-based applications have resulted in an abundance of geotagged (spatial) data. As a consequence, numerous applications have emerged that utilize the spatial data to provide different types of location-based services. However, the huge amount of available spatial data presents a challenge to the efficiency of these location-based services. Although the advent of big data frameworks like Apache Spark has enabled the processing of large amounts of data efficiently, they are designed for general (non-spatial) data. That is due to the build-in data partitioning mechanism that does not take into account the spatial proximity of the data. Therefore, these big data frameworks cannot be readily used for spatial analytics such as efficiently answering spatial queries. To fill this gap, this paper proposes SparkNN, an in-memory partitioning and indexing system for answering spatial queries, such as  $K$ -nearest neighbor, on big spatial data. SparkNN is implemented on top of Apache Spark and consists of three layers to facilitate efficient spatial queries. The first layer is a spatial-aware partitioning layer, which partitions the spatial data into several partitions ensuring that the load of the partitions is balanced and data objects with close proximity are placed in the same, or neighboring, partitions. The second layer is a local indexing layer, which provides a spatial index inside each partition to speed up the data search within the partition. The third layer is a global index, which is placed in the master node of Spark to route spatial queries to the relevant partitions. The efficiency of SparkNN was evaluated by extensive experiments with big spatial datasets. The results show SparkNN significantly outperforms the state-of-the-art Spark system when evaluated on the same set of queries.

---

**Keywords:** spatial data; Apache Spark; global index; local index; partitioning; bounding boxes; kd-tree;  $k$ -nearest neighbors

---

## I. Introduction

The unparalleled popularity of online social media has resulted in the generation of vast amounts of data in various domains such as Banking, Government, Healthcare, Telecommunications, and Stock markets. Moreover, the widespread of devices that can acquire geotagged (spatial) data, e.g., mobile phones, wearable sensors and IoT sensors, have produced an enormous amount of spatial data. Such big data, whether spatial or non-spatial, requires significant attention to support queries efficiently. These queries are used in various applications and services such as social recommendations (Zhang & Chow 2015; Hammou et al. 2018), social community and event detection (Al Aghbari et al. 2019; Sapountzi & Psannis 2018), wireless sensor networks (Al Aghbari et al. 2012, 2013), image analysis (Dinges et al. 2011; Kubo et al. 2003), smart cities (Hanif et al. 2018; Alsaafin et al. 2018), and urban route planning Babar et al. (2019). As a result, spatial data mining has emerged, which deals with gaining knowledge, forming spatial relations, and mining interesting trends and patterns from geo-tagged data (Al Jawarneh et al. 2018; Garaeva et al. 2017).

The big data that is being produced by the aforementioned domains need to be exploited and queried in real-time to serve a wide spectrum of applications. Due to vast amount data, batch-processing systems cannot efficiently process such data in real time. That led to the significance of parallel computing frameworks such as MapReduce Dean & Ghemawat (2008) implemented on Hadoop White (2012). MapReduce processing approach of data is to read from and write to disk. Such a processing approach is very slow specially

for iterative data processing, which is required by most queries. This motivated the emergence of Apache Spark Zaharia et al. (2016), which is an in-memory, real-time cluster processing platform to manage big data and facilitate queries. Spark followed up on the framework of MapReduce Dean & Ghemawat (2008) with benefits of low latency due to its in-memory abstraction of Resilient Distributed Datasets (RDDs) Zaharia et al. (2012). It provides the scalability and fault tolerance. Moreover, it is especially useful for Interactive and Iterative tasks due to its in-memory processing where Hadoop has been reportedly deficient due to its use of disk while processing queries Zaharia et al. (2016).

The RDD is the fundamental data structure in Spark. It can be viewed as an immutable, fault-tolerant and distributed collection of objects. The immutability property of RDDs implies that they can be re-computed if there is a loss of any computing node in the cluster. Each dataset in a RDD is divided into logical partitions which are then computed on different nodes of the cluster. There are two built-in data partitioning mechanisms of one-dimensional RDDs across various data nodes in the cluster; namely Hash and Range partitioning. The hash partitioning attempts to spread data across all partitions by defining some function on the key-value pairs of RDDs. On the other hand, Range partitioning specifies data ranges for each partition and distributes the data accordingly.

These basic data partitioning approaches employed by Spark do not work well with spatial data due to its multi-dimensional property Zhang et al. (2017). That is the default Spark partitioning mechanisms ignores the spatial characteristics of the data. Although Spark balances the load between the different computing nodes, it randomly distributes the spatial data. Moreover, the concept of co-locality come into effect when querying spatial data, where each geographical location is viewed to be in vicinity or at a remote distance from another location. If the co-locality is taken into consideration when partitioning the data, neighboring data points would be placed in the same, or neighboring, partitions. Such proximity-aware partitioning of the data can significantly boost up spatial query performance. That is because all data points would be partitioned according to their inherent distance from other data points. So, for instance, if nearby locations from one area are requested, it would be easier to not go through the whole data to find nearby locations but only one, or few, partitions are searched.

To address this challenge, this paper proposes an in-memory partitioning and indexing system called SparkNN. SparkNN aims to efficiently support spatial queries such as  $k$  nearest neighbor queries. SparkNN consists of three layers, which are built on top of the Apache Spark system, namely, Spatial-aware data partitioning layer, Local indexing layer and Global index layer.

The Spatial-aware data partitioning layer uses the proximity information of the spatial data to partition the data among Spark's computing nodes such that data points with close proximity are stored in the same, or neighboring, partitions. In this layer, not only the partitioning is ensured to be spatial-aware, but also the load is balanced among the different partitions. This is achieved by utilizing a  $kd$ -tree to partition the data, which inherently takes the data skewness into consideration when partitioning the data among  $m$  computing nodes.

In the Local indexing layer, a local index of  $kd$ -tree is created for every data partition. Due to the huge number of spatial points in every partition, fast access methods are required to retrieve the relevant data point for a given query. These local indexes are used to efficiently answer spatial queries, particularly the KNN queries.  $kd$ -trees are most effectively used for  $k$  nearest neighbor queries where the  $k$  most similar data points to the query point are retrieved.

The Global index layer routes user queries to the relevant partition(s). Since the data partitions are spatial-aware that is points with close proximity are in the same partition, or in a neighboring partition. The global index layer determines to which partition(s) should a given query be sent based on the distance between the query point and borders of the neighboring partitions.

We have evaluated the efficiency of SparkNN with big spatial data and compared with the state-of-the-art Spark implementation. The result of the conducted experiments show that SparkNN significantly outperforms the Apache Spark system. Hence, the main contributions of this paper are:

- SparkNN, which is an in-memory partitioning and indexing system that efficiently supports the spatial queries.
- A spatial-aware partitioning algorithm for mapping spatially close data points to the same, or neighboring, partitions. Moreover, the partitioning algorithm ensure load balancing among the different partitions.
- Building a local index in each partition to speedup answering kNN queries and a global index to route queries to the relevant partition(s).
- Conducting extensive experiments to evaluate the efficiency of SparkNN on a big spatial datasets.

The rest of this paper is organized as follows. In Section II, a review of related distributed spatial systems that are developed on different frameworks is presented. Then, we introduce the architecture and different layers of SparkNN in Section III. Section IV presents the algorithms of the proposed SparkNN. The conducted experiments and the results are detailed in Section V. Finally, the paper is concluded and future directions to further extend this research are presented in Section VI.

## II. Literature Review

This section discusses the related work on big spatial data processing and distinguishes it from the proposed SparkNN system.

### A. Hadoop-based systems

To begin with, Hive, a data warehousing solution was built on top of Hadoop in 2009 by Thusoo et al. in Thusoo et al. (2009). This paradigm was built to make the low-level programming model of MapReduce more declarative by supporting queries written in a SQL like language. Spatial constructs were integrated into the Hive in Hadoop-GIS Aji et al. (2013). In this work, various types of fundamental and complex spatial queries like point, containment, joins, nearest neighbors etc. were supported on MapReduce by utilizing global and local indexing. Then, HBase (Hadoop database) George (2011) was developed as an open-source, distributed, and non-relational i.e. NoSQL database leveraging the capabilities of Hadoop Distributed File-System (HDFS). Following up on this work, MD-HBase Nishimura et al. (2011) was proposed exclusively to support spatial data operations. It employed *kd*-tree and quad-tree indexes over HBase Index and Data Storage layers, respectively. MD-HBase processes multi-dimensional range and *k*-NN queries with a latency of less than a second, which was one to two orders of magnitude faster than its MapReduce based implementation. The main limitation of these Hadoop-based spatial systems was that they treated Hadoop like a black box and therefore were restricted by the limitations of Hadoop.

SpatialHadoop Eldawy & Mokbel (2015) was introduced as an open source project which overcame the limitations of Hadoop-GIS and MD-HBase. It also used MapReduce for spatial data support. SpatialHadoop provided built-in spatial data awareness inside Hadoop base code. It supports various spatial queries like range, join, and *k*NN. Two-level global and local indexes with Grid file and R-tree were also implemented. Then, Parallel SECONDO Lu & Güting (2014) was proposed, which combined Hadoop and Secondo Güting et al. (2010) databases. It developed a PSFS (Parallel SECONDO File System) for storing and shuffling intermediate data to support multiple types of queries like map-matching and symbolic trajectory pattern matching. The performance of Parallel SECONDO was demonstrated using generated road networks with data from OpenStreetMap (OSM) (OpenStreetMap 2019). The work in Whitman et al. (2014) also used MapReduce to implement PMR quadtree on spatial data. In this work, range and *k*NN queries on Hadoop were supported. Then, a spatio-temporal database GeoMesa Hughes et al. (2015) was built on top of Hadoop. It was described as a suite of tools for persisting, querying, analyzing, and fusing spatio-temporal data. Furthermore, it supported non-point geometries, indexing of secondary attributes, and complex queries for both vector and raster datasets.

### B. Spark-based systems

After the foundation of Apache Spark, many researchers leveraged the in-memory capabilities of Spark to perform distributed computing on spatial datasets. Xie et al. Xie et al. (2014) employed quad-trees and R-trees to search spatial data using the framework of Spark. In their experiments they concluded quad-trees are more efficient on distributed systems. The various data management techniques to enable interactive and scalable processing of spatial data was investigated in Sarwat (2015). This work envisioned extending Apache Spark to support spatial query operations and is considered as a foundation for GeoSpark Yu et al. (2015). GeoSpark is a three-layer architecture atop of Apache Spark and supports different geometrical and spatial objects like Points, Polygons, and Rectangles. It implemented various query processing algorithms like range, *k*NN, and join. It was shown that GeoSpark outperforms SpatialHadoop Eldawy & Mokbel (2015) due the use of in-memory data processing by GopSpark. Also, Yu et al. proposed SpatialSpark You et al. (2015), which supports indexed spatial joins based on point-in-polygon test and point-to-polyline distance computation by using JTS library (JTS 2019) for spatial predicates and functions. Several indexing and filtering techniques were also implemented in SpatialSpark. However, it did not perform well on multiple computing nodes especially for data intensive processing. Its low scalability was due to the data communication overheads between the distributed computing nodes.

Another geospatial open source library developed by (Magellan 2015) made use of Spark SQL and DataFrame API of Spark to ensure efficient execution of spatial queries. Various spatial operators like

intersects, contains, covers, etc. were implemented along with multiple spatial data structures. Spark SQL and DataFrame was again leveraged to conduct relational processing in Simba (Spatial In-Memory Big data Analytics) Xie et al. (2016). A Query optimizer and SQL context module of Simba were used to support logical and cost-based optimizations for selecting efficient query plans and running multiple queries in parallel. Similarly, Tang et al. proposed LocationSpark Tang et al. (2016), which is a full-fledged library of memory management, spatial indexing, query execution and spatial analytics.

Support of spatio-temporal data process has been proposed by some researchers. For example, TrajSpark Zhang et al. (2017) was proposed, in which an index called IndexRDD for spatio-temporal data was introduced to manage trajectory segments. Furthermore, it provided a STPartitioner which used quad tree to group spatio-temporally close data into the same partition. The support of both global and local indexes was provided where the global index is stored at the master node and is updated when new data partitions arrived by keeping track of the data distribution through a time decay model. It was shown that TrajSpark outperforms its peers like GeoSpark and Simba on real and synthetic datasets in terms of query latency and scalability. Similarly, a system that supports spatio-temporal analytics on Spark, called the STARK Hagedorn et al. (2017), was proposed. This system provides alternative data partitioners (unbalanced grid and balanced binary space) to users. STARK used R-tree for spatial indexing, however different options were provided to the user to select if an index is to be built. Query operators of filter, join,  $k$ NN, and clustering were also developed in STARK.

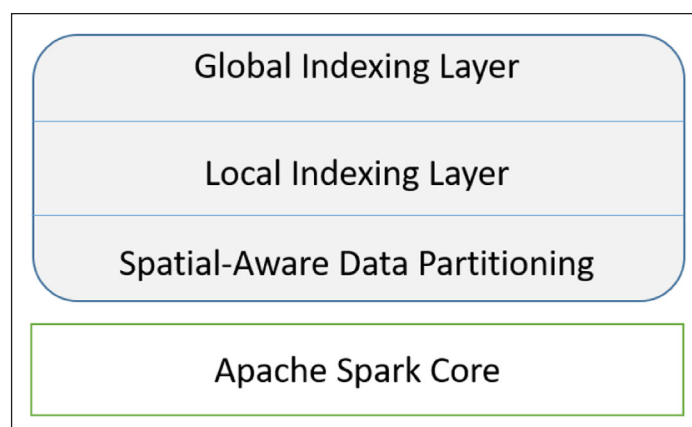
The proposed SparkNN is expected to perform better than systems like GeoSpark Yu et al. (2015) and SpatialHadoop Eldawy & Mokbel (2015) due to its indexing structure (kd-tree) that facilitates fast search since it is a tree and adapts well to skewed data since its nodes are adaptive to the point density of the space. On the other hand, GeoSpark, which uses uniform grid as its index, does not support skewed spatial data due to the uniformity of grid cell; thus, negatively affecting the efficiency of spatial queries. Further, it has been reported that GeoSpark outperforms SpatialHadoop.

### III. SparkNN

The aim of SparkNN is to support efficient spatial queries such as KNN. The state-of-the-art Apache Spark framework is efficient in processing non-spatial big data due to its in-memory data processing approach. However, the efficiency of Apache Spark degrades significantly because of the partitioning mechanism, which does not take into consideration the co-locality of data points and thus ignores their spatial characteristics during the data partitioning process. Therefore, SparkNN addresses this challenge by extending the Apache Spark framework to boost up the efficiency of processing spatial queries, particularly the KNN query. SparkNN is built on top of the Apache Spark core and consists of three layers, as shown in **Figure 1**: Spatial-aware data partitioning layer, Local indexing layer and Global index layer.

#### A. Spatial-Aware Data Partitioning Layer

In this layer, spatial-aware data partitioning mechanism is developed to ensure that data points that are in close proximity in the real 2-dimensional space are mapped to the same, or neighboring, partitions. That is the partitioning mechanism of SparkNN maintains the respective inherent distances between the points in the space. This is achieved by using kd-tree, which is a binary data structure that recursively partitions all the



**Figure 1:** Extension of Spark core to implement SparkNN.

data points into two halves by hyperplanes that are perpendicular to the coordinate axes Hajebi et al. (2011). Although other spatial data partitioning techniques, such as the R-tree, can be used, the kd-tree is faster for spatial point data and requires less memory space. A kd-tree is formed from a set of multi-dimensional points by selecting a splitting data value using only one of the dimensions at each level of the tree. Normally, this procedure stops after  $\log(n)$  levels, where  $n$  is the number of spatial data points. However, the proposed spatial-aware partitioning mechanism stops the splitting process of the kd-tree when the number of leaf nodes is equal to the number of available computing nodes  $m$  in SparkNN. Where  $m$  is a user defined variable set according to the desired number of partitions.

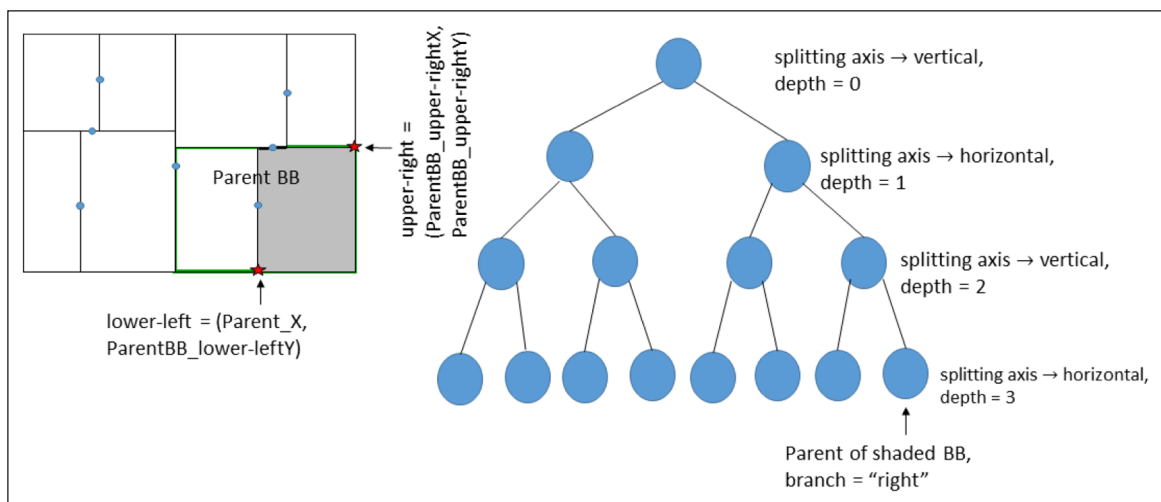
When the kd-tree grow to  $m$  leaf nodes, each leaf node corresponds to one partition. That is the spatial region represented by each leaf node determines the lower and upper bounds (boundary) of the partition. Note that the set of points  $n$  used to determine the spatial boundaries of the kd-tree nodes is a random sample extracted from the real spatial data, which is initially stored by in the RDDs of Apache Spark. In case of static spatial datasets, the SparkNN partition boundaries are determined based on the extracted sample of spatial data points. However, if the spatial data points are dynamic, a periodic rebalancing mechanism can be applied to ensure data balancing among the different SparkNN partitions. That is periodically the boundaries of neighboring partitions are adjusted to balance the data loads of the partitions.

Based on the determined boundaries of leaf nodes, which are mapped to the SparkNN partitions, the spatial data points are mapped to the relevant SparkNN partition. These partitions correspond to the spatial-aware RDDs, which we refer to as SARDDs. The spatial data is loaded into Spark as latitude and longitude coordinates. Each spatial point is mapped to one SARDD, which has a predefined boundary called here Bounding Box (BB). A BB is defined by the *lower-left*  $(x_{min}, y_{min})$  and *upper-right*  $(x_{max}, y_{max})$  coordinates of the region owned by a leaf node in the kd-tree. The number of these leaf nodes is determined by the depth of the tree and is specified by the user to tune the number of partitions. The depth of the tree is initialized as 0 and incremented by 1 when the tree is recursively grow down another level. The following condition is used to determine whether the tree needs to be grow down to another level or is it the tree growth should stop. if the number of leaf nodes, which corresponds to the desired number of partitions  $m$ , has been reached that is equivalent to  $2^{depth}$ , the kd-tree stops growing. Otherwise, the leaf nodes of the tree are split to grow down one level.

$$2^{depth} \begin{cases} \geq m, & \text{Stop growing} \\ < m, & \text{Grow down another level} \end{cases}$$

The approach to partition the data by creating bounding boxes is illustrated in **Figure 2**.

In **Figure 2**, a tree of depth 2 results in 4 leaf nodes. The formulation of *lower-left* and *upper-right* parameters of the grey BB, which is shown in the **Figure 2**, are determined by using conditional statements according to the depth of the tree, the splitting axis defining the direction of split, and finally the branch



**Figure 2:** Partitioning the spatial data into BBs.

(left or right) that would be created next. The skewness of data is an issue with kd-trees, however a simple solution is to implement a balanced kd tree. To build a balanced kd-tree, it is necessary to find the median of the data for each recursive split of the data.

After the creation of these boxes, a key-value pair SARDD is created by mapping every spatial point to one of the SARDDs, where each SARDD corresponds to one BB.

### B. Global indexing

A kd-tree that is formed to create the  $m$  partitions, which are SARDDs of SparkNN, is used as a global index to route user queries to the relevant SARDDs. Where  $m$  is a user defined variable set according to the desired number of partitions. Since the data partitions are spatial-aware that is points with close proximity are in the same partition, or in a neighboring partition. Note that the inherent distances between the spatial points are preserved in the SARDDs. The global index layer determines to which partition(s) should a given query be sent based on the distance between the query point and borders of the neighboring partitions.

First the the partition in which the query point resides is determined. Then, the KNN points are retrieved. Also, the distance  $d_i$  between the query point and every neighboring partition is computed. If  $d_i$  between the query and a certain partition is less than distance  $r$  between the query point and farthest point ( $k^{th}$  point), then  $k$ NN query is routed to this neighboring partition. **Figure 3** shows the global indexing mechanism, where the query point represented by a red star is contained in the BB P2. Since  $r$  intersects with the neighboring BBs (the BBs P4, P5 and P7), the  $k$ NN query will be sent these BBs.

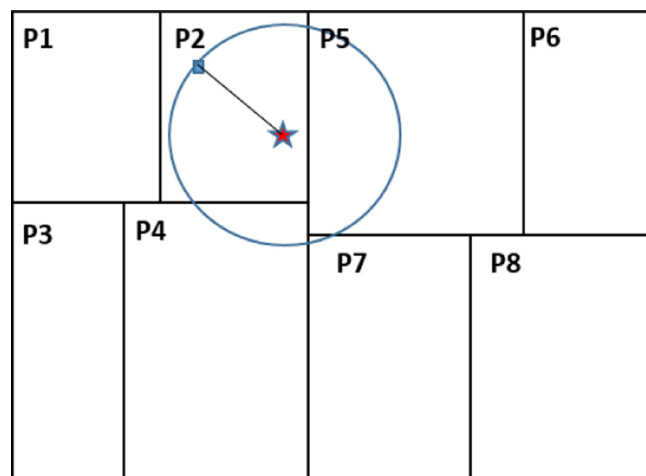
### C. Local indexing

Due to the huge number of spatial points in every partition, fast access methods are required to retrieve the relevant data point for a given query. A local index is created for each of the SARDDs (partitions) by using kd-trees. These local indexes are used to efficiently answer spatial queries, particularly the KNN queries. kd-trees are most effectively used for  $k$  nearest neighbor queries where the  $k$  most similar data points to the query point are retrieved.

This SARDD with the local index is saved in memory by persisting it for easy access for all of the queries. By caching the local index of SARDD, it does not need to be re-computed for each query and can be fetched from the memory when required. Note that in our experiments, we varied the size of the dataset to measure the impact of the data size. Therefore, the depth and the required memory space of a kd-tree are dependent on the size of the dataset.

## IV. KNN Querying

In SparkNN, each SARDD stores the spatial point of local certain region in the space, in addition to the local index build to access the spatial point. A local index at each SARDD is used to answer the user's KNN queries. That is given a set of  $n$  data points  $P = p_1, p_2, \dots, p_n$  and query point  $q$ , SparkNN computes and a subset  $S \in P$  of points that contains the  $k$  nearest points ( $k$ NN) to  $q$ .



**Figure 3:** Global indexing of the partitions.

$$\forall p_i \in S, p_j \in P \setminus S : dist(q, p_i) \leq dist(q, p_j)$$

Where  $dist(.)$  is a function that returns the Euclidean distance between two spatial points. If  $p_1 = (x_1, y_1)$  and  $p_2 = (x_2, y_2)$ , then the distance between  $p_1$  and  $p_2$  is given by:  $dist(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ . The proposed  $k$ NN algorithm (see **Algorithm 1**) utilizes a bounded-priority queue (Ryan LeCompte 2019), which retains the top  $k$  points from the priority queue of nearest points. These points are prioritized by their distances  $dist(.)$  from the query point  $q$ . So, the data point with the least distance from the query point is the highest priority element in the bounded-priority queue and vice-versa. The **Algorithm 1** is used to find the  $k$ NN from the SARDD, which are persisted in memory.

In the  $k$ NN from SARDDs algorithm, the function  $Knearest$  takes as input three arguments: the local index  $node$ , the query point  $needle$ , and the number of nearest neighbors  $k$ . The nearest neighbors to be found will be stored in the  $bpq$  which uses a heap implementation to maintain a priority queue of  $k$  objects ordered

---

**Algorithm 1:**  $k$ NN from SARDDs.
 

---

**Input:**  $node :=$  root of the tree in a RDD partition,  $needle := \{n_0, n_1, \dots\}$  the query point,  $k :=$  an integer value corresponding to the number of nearest neighbors to find

**Output:**  $bpq :=$  A bounded priority queue containing the  $k$ -nearest neighbors

**Function**  $Knearest (node, needle, k) :$

Declare  $bpq$  as a *BoundedPriorityQueue* to contain candidate nearest neighbors

Set the size of  $bpq$  to  $k$

**Function**  $nearest (node) :$

default  $\leftarrow node$

**if** default == *NULL* **then**

**return** NearestPoint

**else**

    Enqueue default into  $bpq$

**end**

**if**  $n_i \leq default_i$  **then**

    NearestPoint  $\leftarrow nearest (left(node))$

**else**

    NearestPoint  $\leftarrow nearest (right(node))$

**end**

**if**  $bpq$  is not full **OR**  $|default_i - n_i| < distance(needle, head(bpq))$  **then**

**if**  $n_i \leq default_i$  **then**

        NearestPoint  $\leftarrow nearest (right(node))$

**else**

        NearestPoint  $\leftarrow nearest (left(node))$

**end**

**return** NearestPoint

**end**

$Knearest (node, needle, k)$

**return**  $bpq$

by the distance of spatial points from the query point. So that the points at the head of the queue termed as *head(bpq)* has the least priority and is the farthest from the query point. The size of this *bpq* is set to  $k$ .

Initially, the *default* is set to root node of the local *kd*-tree and it is enqueued into the *bpq*. The splitting axis dimension  $n_i$  of needle is compared with the same dimension of current node i.e.  $default_i$ . If this *needle's* value is less or equal than the node, then the *left(node)* is recursively searched and its value is obtained as *NearestPoint*, however if *needle's* value is greater than the node, then the *right(node)* is recursively searched and its value is obtained as *NearestPoint*. Finally, it is determined whether there is a point closer to the needle at the other side of the tree by using the distance of current point with the *needle*. The other branch of the tree is only searched if the *bpq* isn't full or if the  $|default_i - n_i|$  is less than the distance of farthest point from the needle, i.e., *head(bpq)*. After the *nearest* procedure ends, when leaf node is reached and it is determined that there is no better branch to find a candidate point to be included in the *kNN*, the *bpq* is returned.

It is worth mentioning that a range query can be easily computed using a *kNN* query. A range query of a spatial point dataset returns all points inside a query region  $R$ . A *kNN* query returns the  $k$  closest points to a query point  $q$ . To find the points in a given region (rectangle)  $R$ , a *kNN* query is executed on a given query point  $q$ . the result of the *kNN* search is a set of  $k$  nearest points that define a circle centered at  $q$  with radius equal to the distance from  $q$  to its  $k^{th}$  nearest neighbor. The area of the circle define a range. By adjusting value of  $k$ , the area of a circle can cover the required range to be computed Bae et al. (2007). Thus, the set of *kNN* points represents a superset of the result of the range query  $q$ . Then, a second phase of filtering is executed to remove the points that are outside  $R$  would result in a set of points that is the exact answer to the range query. As explained above, it is possible to retrieve the result set of a range query using multiple *kNN* queries, however a native range query executed directly on the data would be more efficient. As discussed in Bae et al. (2007), the number of *kNN* queries that retrieve a range result can be reduced by selecting a relatively large value of  $k$ . Nevertheless, this would return extra non-relevant points (false positives), which requires a second phase of filtering to remove them.

Note that it is possible that the query may run in parallel in more than one partition depending on the location of the query point  $q$  and value of  $k$ . In such a case, each partition that is executing the same query will retrieve its local *kNN* candidate set and return it to the master Spark node that received the query. The master will aggregate the local *kNN* results and find the final set of *kNN*. This is easily computed by a one pass comparison,  $O(n)$ , over the aggregated local *kNN* results to extract the global *kNN* set, which is the answer to the query.

## V. Experiments and Results

### A. Experimental Setup

The experiments were conducted on 64-bit Ubuntu 16.04 LTS with a memory of 31.3 GiB and a processor of Intel® Xeon(R) CPU E5-2630 v3 @ 2.40GHz × 32. The Spark version used is 2.3.2 using the Scala 2.11.8. The Standalone deployment of Spark was used in Client mode. These experiments were conducted on a cluster with 2 nodes; one with 32 cores and the other with 16 cores.

### B. Dataset

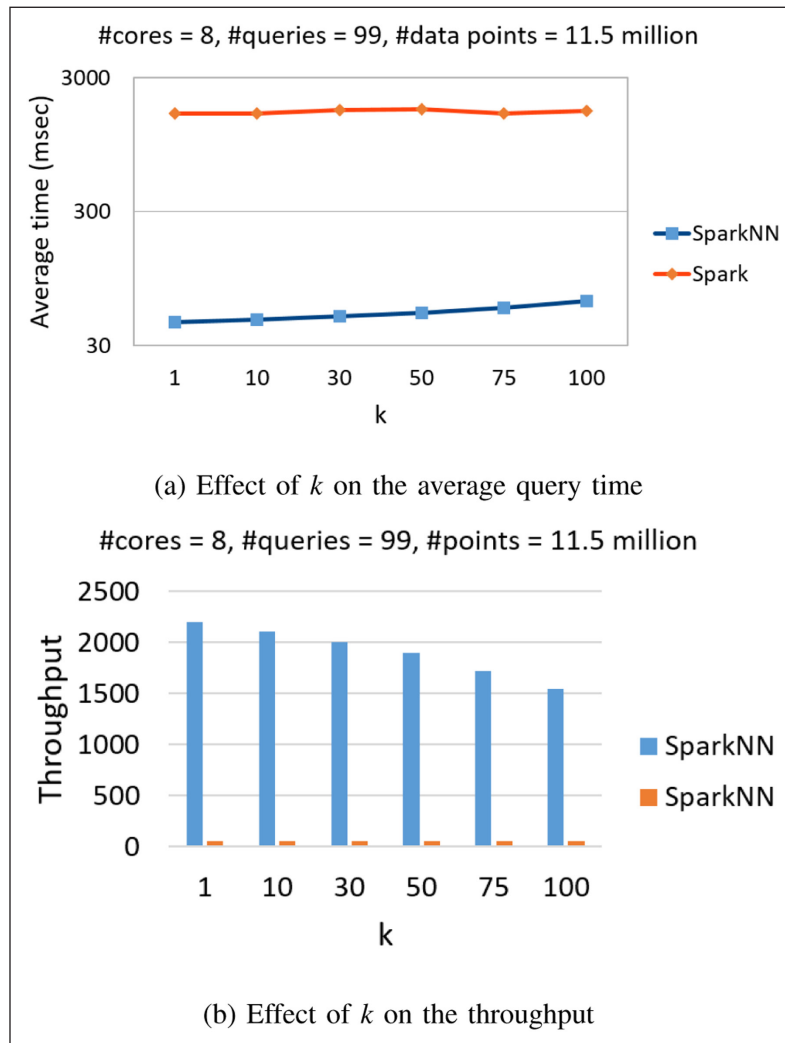
The spatial dataset obtained for this research was obtained from (Free Downloadable Databases MaxMind Developer Site) and the database used was GeoLite2 City. A CSV downloadable file was obtained on which data transformations were performed to select the latitude and longitude columns. This total number of collected spatial data points for testing purposes is 11.5 million points. The average querying times taken in the following experiments were computed over an average of 99 queries selected randomly from the data points.

### C. Impact of $k$

The first experiment was performed by varying  $k$  to measure and study the query processing time and throughput. The result of the *kNN* queries shown in **Figure 4a** depicts a comparison between the Apache Spark referred to as *Spark* with SparkNN. A total of 8 executor cores were used for this experiment for all of the data points. The values of  $k$  were taken as 1, 10, 30, 50, 75, and 100, as shown on x-axis in **Figure 4a**. Due to the huge difference in performance values, a logarithmic scale is used to compare the two systems. That is the values of processing *kNN* queries using SparkNN was in the range of milliseconds while the processing time of Apache Spark is in minutes.

The significant gain in performance when using SparkNN is due to the spatial-aware partitioning of data points that narrowed down the search to one, or few, SARDDs. Moreover, the local indexes that reside inside the SARDDs facilitated fast access to the relevant points to the query. In case of Apache Spark, the query





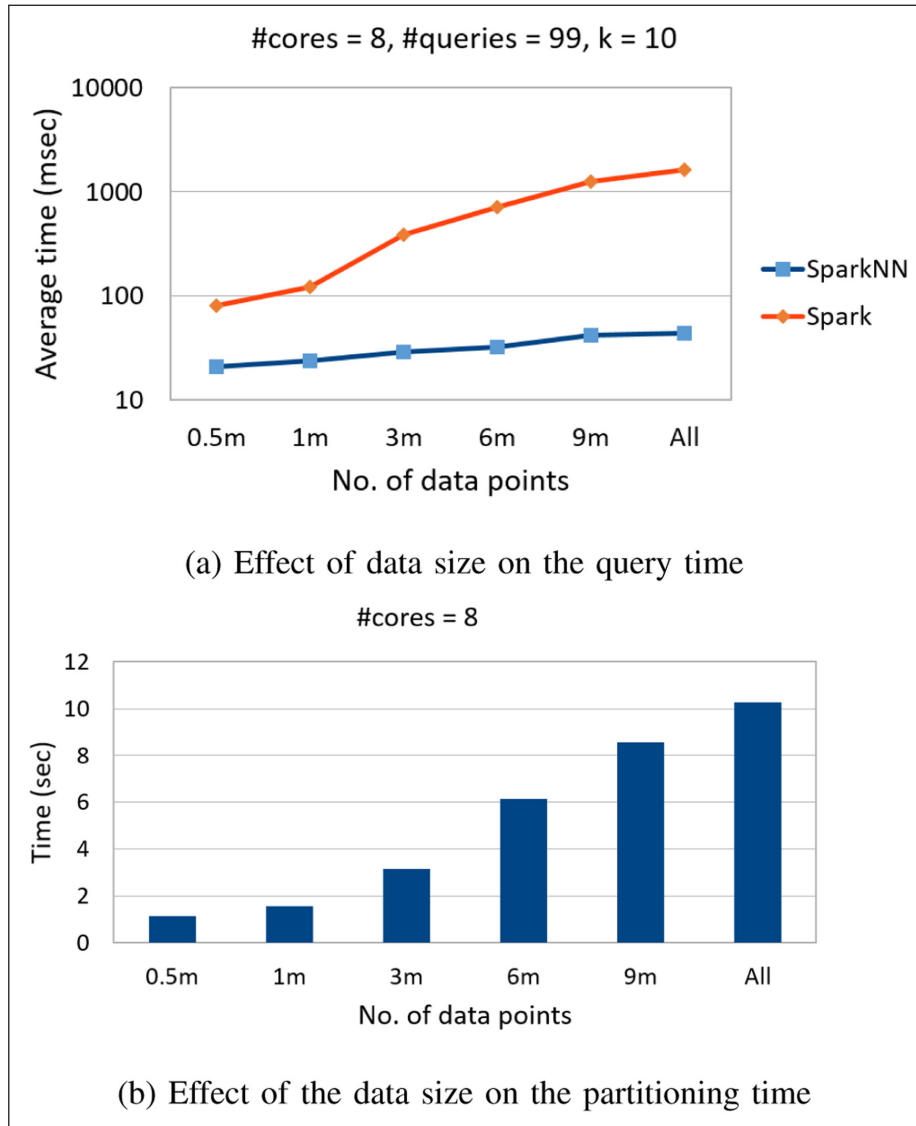
**Figure 4:** Impact of  $k$ .

latency of  $k$ NN was almost constant when  $k$  was varied from 1 to 100, as shown by the red line in **Figure 4a**. This is because data points are randomly stored inside the RDDs of Apache Spark and thus processing a query regardless of the value of  $k$  requires checking all the data points in all RDDs. So the variation of  $k$  has no impact on the processing times. On the other hand, when using SparkNN, as  $k$  is increased, querying time increases almost linearly. It is worth noting that if we have an extreme case where the value of  $k$  is very large relative to the size of the database, where  $r$  overlaps with all partitions, this would cause all partitions to execute the same query.

**Figure 4b** illustrates the query throughput results of Apache Spark and SparkNN systems. Where *throughput* is the number of queries that are processed in a second. For this experiment, we used eight cores to store the 11.5 million spatial data points. The x-axis shows the variation of  $k$ , and the red bars depict the throughput of Apache Spark, while the blue bars represent the throughput of SparkNN. Clearly, the query throughput of SparkNN is much higher (in the range of 1500 to more than 2000 queries per second) than that of Apache Spark. However, the throughput of SparkNN slightly decreases when  $k$  is increased from 1 to 100 due to the fact that more time is required to find a larger number of nearest neighbors.

#### D. Impact of data size

The second set of experiments was performed to compare the effects of data size on querying time for both SparkNN and Apache Spark. In addition, the time to perform the data partitioning which mapped all the points to their respective SARDDs by building a  $k$ d-tree was also analyzed. The value of  $k$  was kept constant for this experiment with a value of 10. In this experiment, eight executor cores were used while the number of input data points were varied from 0.5 million, 1 million, 3 million, 6 million, 9 million, and 11.5 million, as shown by the x-axis of **Figure 5a**.



**Figure 5:** Impact of data size on data load time.

The scale of y-axis was set to logarithmic due the huge difference in performance between SparkNN and Apache Spark. The y-axis illustrates the latency variations with the different data sizes. When increasing the data size, the average time in milliseconds increases for both the sequential version and SparkNN. For the Apache Spark, shown by the red line in **Figure 5a**, the significant increase in latency is due to the fact that it takes more time to process the data points as the size increases. This is expected since Apache Spark requires to process all the spatial data point in the SARDDs. On the other hand, SparkNN, which is shown by the blue line, the query time only slightly increases as the data size increases because it requires to process the  $k$  nearest neighbors using relatively larger local indices.

The time to load the data points by building bounding boxes for different data sizes is illustrated in **Figure 5b**. In this experiment, we used eight cores to load the data into the partitions. The data size is shown on x-axis, while the data load time is shown on the y-axis. The time to build the  $k$ d-tree and load the data into the desired number of partitions (SARDDs) increases when data size increases.

### E. Scalability

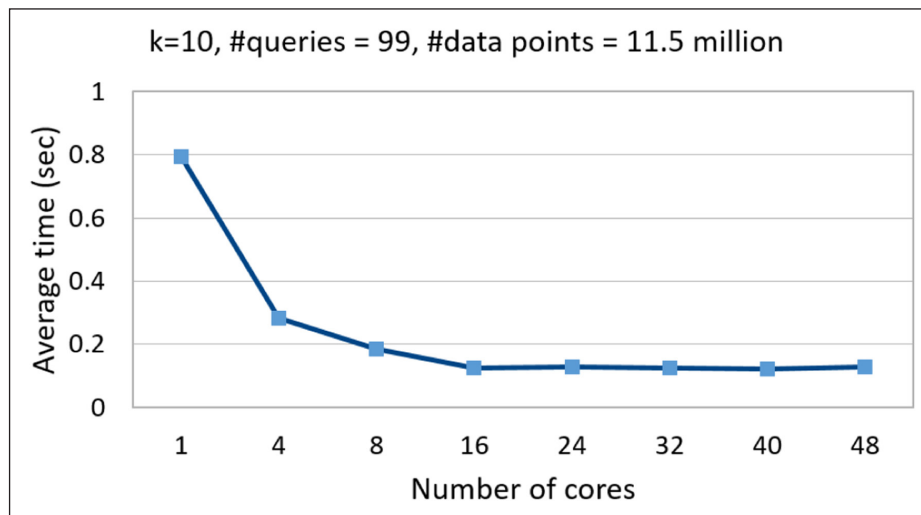
The third experiment was performed to test the Scalability of SparkNN, which is depicted in **Figure 6**. This goal of this experiment was to find out how the SparkNN scales as more executor cores are used. For this experiment, a constant value of  $k = 10$  was used for all queries on the 11.5 million data points. The y-axis depicts the latency in seconds. As shown in **Figure 6**, the average time of processing  $k$ NN queries decreases as the number of cores increases up to 16 executor cores, however, after that the latency is almost constant.

### F. Impact of SARDDs

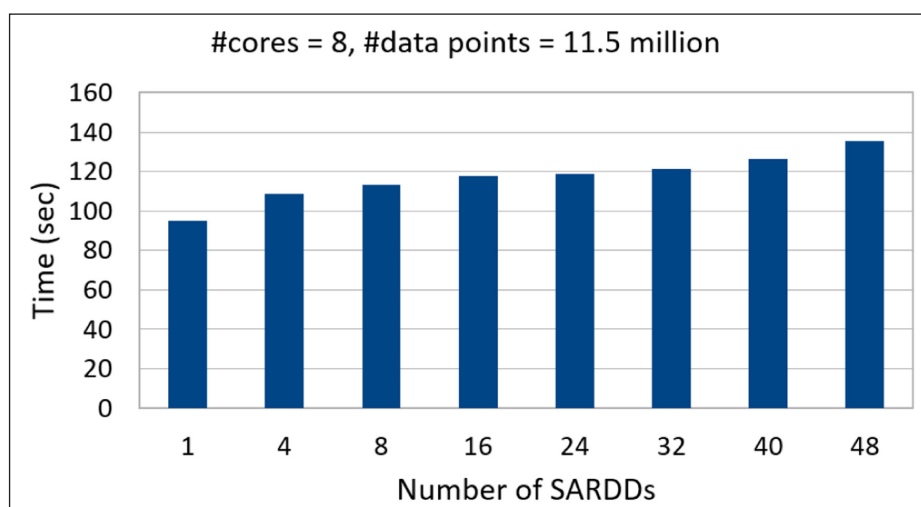
Lastly, an experiment was conducted to study the impact of increasing number of SARDDs (partitions) on the time to build the tree that partitions the spatial data at the master node. The number of executor cores were kept constant for this experiment with a value of 4, however the number of BBs which determined the number of generated SARDDs during the partitioning process were increased from 1 to 48. **Figure 7** shows the results of this experiment with the x-axis showing the number of SARDDs, which were specified to the values: 1, 4, 8, 16, 24, 32, 40, and 48. The y-axis shows the query time in seconds. A slight increase in time to build the tree at the master node was observed when the partitions were increased. This slight increase is because it takes a longer time to build a deeper tree when the desired number of SARDDs increases. For instance, for 1 SARDD, only 1 node of the tree is built to determine 1 BB for the whole region, while for 8 SARDDs, a 3-level tree would be built to compute 8 BBs. Note that the number of SARDDs (x-axis values in **Figure 7**) can follow and go up to  $2^{\text{depth}}$  of the kd-tree, however it can be less depending on the available cores (machines) and the point density of the space.

## VI. Conclusion and Future Directions

The paper proposed SparkNN, which is an in-memory partitioning and indexing system to process  $K$ -nearest neighbor spatial queries on big spatial data. SparkNN extended the state-of-the-art Apache Spark to support efficient storage and processing of spatial queries. This extension consists of three layers to be built on top of Apache Spark. As a result, the storage of data is spatial-aware that led to the creation of a new concept



**Figure 6:** Effect of no. of cores on the scalability of SpakNN.



**Figure 7:** Effect of the number of SARDD on the partitioning time.

of SARDDs. Since the geographical boundaries of the SARDDs were determined by the partitioning *kd*-tree, the SARDDs are balanced in terms of load. Moreover, SparkNN built a local index inside each SARDD and a global index in the master node of SparkNN to route user queries to the relevant SARDDs. Our experiments have shown that SparkNN significantly improved the average query time over that of the Apache Spark. In addition, SparkNN significantly outperformed Apache Spark in terms of query throughput, scalability toward data size and increasing the number of execution cores.

In the future, the Spark Streaming and Spark SQL components of Apache Spark can be used to achieve further structure into the data used and to ingest real-time data in mini-batches. Also, spatio-temporal data can be used which logs the location details of moving objects at different times.

## Competing Interests

The authors have no competing interests to declare.

## References

- Aji, A, Wang, F, Vo, H, Lee, R, Liu, Q, Zhang, X and Saltz, J.** 2013. Hadoop gis: A high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11): 1009–1020. DOI: <https://doi.org/10.14778/2536222.2536227>
- Al Aghbari, Z, Bahutair, M and Kamel, I.** 2019. Geosimmr: A mapreduce algorithm for detecting communities based on distance and interest in social networks. *Data Science Journal*, 18(1): 1–10. DOI: <https://doi.org/10.5334/dsj-2019-013>
- Al Aghbari, Z, Kamel, I and Awad, T.** 2012. On clustering large number of data streams. *Intelligent Data Analysis*, 16(1): 69–91. DOI: <https://doi.org/10.3233/IDA-2011-0511>
- Al Aghbari, Z, Kamel, I and Elbaroni, W.** 2013. Energy-efficient distributed wireless sensor network scheme for cluster detection. *International Journal of Parallel, Emergent and Distributed Systems*, 28: 1: 1–28. DOI: <https://doi.org/10.1080/17445760.2012.729584>
- Al Jawarneh, IM, Bellavista, P, Corradi, A, Foschini, L, Montanari, R and Zanotti, A.** 2018. In-memory Spatial-Aware Framework for Processing Proximity-Alike Queries in Big Spatial Data. In: *2018 IEEE 23<sup>rd</sup> International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 1–6. IEEE. DOI: <https://doi.org/10.1109/CAMAD.2018.8514950>
- Alsaafin, A, Khedr, AM and Al Aghbari, Z.** 2018. Distributed trajectory design for data gathering using mobile sink in wireless sensor networks. *AEU-International Journal of Electronics and Communications*, 96: 1–12. DOI: <https://doi.org/10.1016/j.aeue.2018.09.005>
- Babar, M, Arif, F, Jan, M, Tan, Z and Khan, F.** 2019. Urban data management system: Towards big data analytics for internet of things based smart urban environment using customized hadoop. *Future Generation Computer Systems*, 96: 398–409. DOI: <https://doi.org/10.1016/j.future.2019.02.035>
- Bae, WD, Alkobaisi, S, Kim, SH, Narayanappa, S and Shahabi, C.** 2007. Supporting range queries on web data using *k*-nearest neighbor search. In: *International Symposium on Web and Wireless Geographical Information Systems*, 61–75. Springer. DOI: [https://doi.org/10.1007/978-3-540-76925-5\\_5](https://doi.org/10.1007/978-3-540-76925-5_5)
- Dean, J and Ghemawat, S.** 2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1): 107–113. DOI: <https://doi.org/10.1145/1327452.1327492>
- Dinges, L, Al-Hamadi, A, Elzobi, M, Al Aghbari, Z and Mustafa, H.** 2011. Offline automatic segmentation based recognition of handwritten Arabic words. *International Journal of Signal Processing, Image Processing and Pattern Recognition*, 4(4): 131–143.
- Eldawy, A and Mokbel, MF.** 2015. Spatialhadoop: A mapreduce framework for spatial data. In: *2015 IEEE 31st international conference on Data Engineering*, 1352–1363. IEEE. DOI: <https://doi.org/10.1109/ICDE.2015.7113382>
- Garaeva, A, Makhmutova, F, Anikin, I and Sattler, K-U.** 2017. A framework for co-location patterns mining in big spatial data. In: *2017 XX IEEE International Conference on Soft Computing and Measurements (SCM)*, 477–480. IEEE. DOI: <https://doi.org/10.1109/SCM.2017.7970622>
- Geolite2 free downloadable databases maxmind developer site. <https://dev.maxmind.com/geoip/geoip2/geolite2/>. Accessed: 2019-05-22.
- George, L.** 2011. *HBase: the definitive guide: random access to your planetsize data*. "O'Reilly Media, Inc."
- Güting, RH, Behr, T, Düntgen, C and others.** 2010. SECONDO: A Platform for Moving Objects Database Research and for Publishing and Integrating Research Implementations. *IEEE Data Eng. Bull.* 33(2): 56–63.

- Hagedorn, S, Götze, P and Sattler, K-U.** 2017. The STARK framework for spatio-temporal data analytics on spark. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*.
- Hajebi, K, Abbasi-Yadkori, Y, Shahbazi, H and Zhang, H.** 2011. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In: *Twenty-Second International Joint Conference on Artificial Intelligence*.
- Hammou, B, Lahcen, A and Mouline, S.** 2018. Apra: An approximate parallel recommendation algorithm for big data. *Knowledge-Based Systems*, 157: 10–19. DOI: <https://doi.org/10.1016/j.knosys.2018.05.006>
- Hanif, S, Khedr, AM, Al Aghbari, Z and Agrawal, DP.** 2018. Opportunistically exploiting internet of things for wireless sensor network routing in smart cities. *Journal of Sensor and Actuator Networks*, 7(4): 46. DOI: <https://doi.org/10.3390/jsan7040046>
- Hughes, JN, Annex, A, Eichelberger, CN, Fox, A, Hulbert, A and Ronquest, M.** 2015. Geomesa: A distributed architecture for spatio-temporal fusion. In: *Geospatial Informatics, Fusion, and Motion Video Analytics V*, 9473: 94730F. International Society for Optics and Photonics. DOI: <https://doi.org/10.1117/12.2177233>
- JTS Topology Suite.** <https://www.osgeo.org/projects/jts/>. Accessed: 2019-06-18.
- Kubo, M, Aghbari, Z, Makinouchi, A and Oh, K-S.** 2003. Content-based image retrieval technique using wavelet-based shift and brightness invariant edge feature. *International Journal of Wavelets, Multiresolution and Information Processing*, 1(2): 163–178. DOI: <https://doi.org/10.1142/S0219691303000141>
- Lu, J and Güting, RH.** 2014. Parallel secondo: A practical system for largescale processing of moving objects. In: *2014 IEEE 30th International Conference on Data Engineering*, 1190–1193. IEEE. DOI: <https://doi.org/10.1109/ICDE.2014.6816738>
- Magellan: Geospatial Analytics on Spark.** Oct. 2015. <https://hortonworks.com/blog/magellan-geospatial-analytics-in-spark/>. Accessed: 2019-06-18.
- Nishimura, S, Das, S, Agrawal, D and El Abbadi, A.** 2011. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In: *2011 IEEE 12th International Conference on Mobile Data Management*, 1: 7–16. IEEE. DOI: <https://doi.org/10.1109/MDM.2011.41>
- OpenStreetMap.** <https://www.openstreetmap.org/>. Accessed: 2019-06-18.
- Ryan LeCompte.** Bounded priority queue in scala. <https://gist.github.com/ryanlecompte/5746241>. Accessed: 2019-06-08.
- Sapountzi, A and Psannis, K.** 2018. Social networking data analysis tools and challenges. *Future Generation Computer Systems*, 86: 893–913. DOI: <https://doi.org/10.1016/j.future.2016.10.019>
- Sarwat, M.** 2015. Interactive and Scalable Exploration of Big Spatial Data—A Data Management Perspective. In: *2015 16th IEEE International Conference on Mobile Data Management*, 1: 263–270. IEEE. DOI: <https://doi.org/10.1109/MDM.2015.67>
- Tang, M, Yu, Y, Malluhi, QM, Ouzzani, M and Aref, WG.** 2016. Locationspark: A distributed in-memory data management system for big spatial data. *Proceedings of the VLDB Endowment*, 9(13): 1565–1568. DOI: <https://doi.org/10.14778/3007263.3007310>
- Thusoo, A, Sarma, JS, Jain, N, Shao, Z, Chakka, P, Anthony, S, Liu, H, Wyckoff, P and Murthy, R.** 2009. Hive: a warehousing solution over a mapreduce framework. *Proceedings of the VLDB Endowment*, 2(2): 1626–1629. DOI: <https://doi.org/10.14778/1687553.1687609>
- White, T.** 2012. *Hadoop: The definitive guide*. “O’Reilly Media, Inc.”
- Whitman, RT, Park, MB, Ambrose, SM and Hoel, EG.** 2014. Spatial indexing and analytics on Hadoop. In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 73–82. ACM. DOI: <https://doi.org/10.1145/2666310.2666387>
- Xie, D, Li, F, Yao, B, Li, G, Zhou, L and Guo, M.** 2016. Simba: Efficient in-memory spatial analytics. In: *Proceedings of the 2016 International Conference on Management of Data*, 1071–1085. ACM. DOI: <https://doi.org/10.1145/2882903.2915237>
- Xie, X, Xiong, Z, Hu, X, Zhou, G and Ni, J.** 2014. On massive spatial data retrieval based on spark. In: *International Conference on Web-Age Information Management*, 200–208. Springer. DOI: [https://doi.org/10.1007/978-3-319-11538-2\\_19](https://doi.org/10.1007/978-3-319-11538-2_19)
- You, S, Zhang, J and Gruenwald, L.** 2015. Large-scale spatial join query processing in cloud. In: *2015 31st IEEE International Conference on Data Engineering Workshops*, 34–41. IEEE. DOI: <https://doi.org/10.1109/ICDEW.2015.7129541>
- Yu, J, Wu, J and Sarwat, M.** 2015. 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*, 70. ACM. DOI: <https://doi.org/10.1145/2820783.2820860>

- Zaharia, M, Chowdhury, M, Das, T, Dave, A, Ma, J, McCauley, M, Franklin, MJ, Shenker, S and Stoica, I.** 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2–2. USENIX Association.
- Zaharia, M, Xin, RS, Wendell, P, Das, T, Armbrust, M, Dave, A, Meng, X, Rosen, J, Venkataraman, S, Franklin, MJ and others.** 2016. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11): 56–65. DOI: <https://doi.org/10.1145/2934664>
- Zhang, J-D and Chow, C-Y.** 2015. Geosoca: Exploiting geographical, social and categorical correlations for point-of-interest recommendations. In: *SIGIR*, ACM. DOI: <https://doi.org/10.1145/2766462.2767711>
- Zhang, Z, Jin, C, Mao, J, Yang, X and Zhou, A.** 2017. Trajspark: A scalable and efficient in-memory management system for big trajectory data. In: *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data*, 11–26. Springer. DOI: [https://doi.org/10.1007/978-3-319-63579-8\\_2](https://doi.org/10.1007/978-3-319-63579-8_2)


**How to cite this article:** Al Aghbari, Z, Ismail, T and Kamel, I. 2020. SparkNN: A Distributed In-Memory Data Partitioning for KNN Queries on Big Spatial Data. *Data Science Journal*, 19: 35, pp.1–14. DOI: <https://doi.org/10.5334/dsj-2020-035>

**Submitted:** 09 October 2019

**Accepted:** 28 July 2020

**Published:** 24 August 2020

**Copyright:** © 2020 The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC-BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited. See <http://creativecommons.org/licenses/by/4.0/>.

 *Data Science Journal* is a peer-reviewed open access journal published by Ubiquity Press.

**OPEN ACCESS** 