Journal of Big Data

RESEARCH Open Access

Efficient spatial data partitioning for distributed kNN joins

Ayman Zeidan[1*] and Huy T. Vo[2]

*Correspondence:
azeidan@gradcenter.cuny.edu

[1] Department of Computer Science, CUNY Graduate Center, New York, USA
Full list of author information is available at the end of the article

Abstract

Parallel processing of large spatial datasets over distributed systems has become a core part of modern data analytic systems like Apache Hadoop and Apache Spark. The general-purpose design of these systems does not natively account for the data's spatial attributes and results in poor scalability, accuracy, or prolonged runtimes. Spatial extensions remedy the problem and introduce spatial data recognition and operations. At the core of a spatial extension, a locality-preserving spatial partitioner determines how to spatially group the dataset's objects into smaller chunks using the distributed system's available resources. Existing spatial extensions rely on data sampling and often mismanage non-spatial data by either overlooking their memory requirements or excluding them entirely. This work discusses the various challenges that face spatial data partitioning and proposes a novel spatial partitioner for effectively processing spatial queries over large spatial datasets. For evaluation, the proposed partitioner is integrated with the well-known k-Nearest Neighbor (kNN) spatial join query. Several experiments evaluate the proposal using real-world datasets. Our approach differs from existing proposals by (1) accounting for the dataset's unique spatial traits without sampling, (2) considering the computational overhead required to handle non-spatial data, (3) minimizing partition shuffles, (4) computing the optimal utilization of the available resources, and (5) achieving accurate results. This contributes to the problem of spatial data partitioning through (1) providing a comprehensive discussion of the problems facing spatial data partitioning and processing, (2) the development of a novel spatial partitioning technique for in-memory distributed processing, (3) an effective, built-in, load-balancing methodology that reduces spatial query skews, and (4) a Spark-based implementation of the proposed work with an accurate kNN spatial join query. Experimental tests show up to 1.48 times improvement in runtime as well as the accuracy of results.

Keywords: Big data, Spatial data, Spatial query, Indexing, Partitioning, Technique, Parallel processing, Load balancing, Distributed computing, Spark, NoSQL, kNN query, All kNN query

Introduction

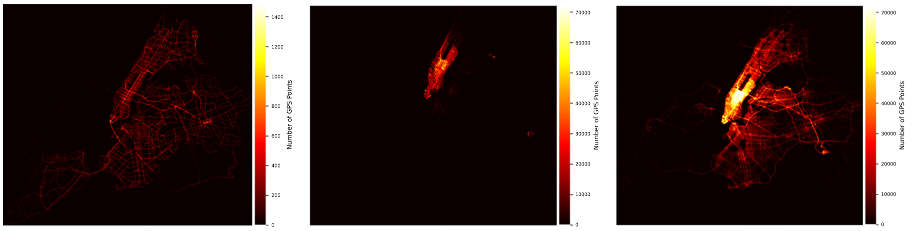
The world's data generation capabilities are rising rapidly. This increase has spurred researchers and businesses to make great strides to find new means for efficient and meaningful storage, retrieval, and analysis. A research report published in 2019 estimated the world's daily data collection rate at around 2.5 quintillion bytes and projected that over 150 zettabytes of data will need analysis by 2025 [1, 2]. The data stems from various sources like the 2.6 billion social media uses (a 2020 estimate), 74.4 million connected cars (a 2023 projection), and 2 billion Internet of Things devices (a 2018 estimate) [3–8].

A significant portion of the collected data, known as spatial data (or geospatial or geographic data), contains spatial attributes (e.g., latitude and longitude) that indicate the data's physical origins. Various works have shown the benefits of analyzing spatial data, making it one of the most valuable assets for enterprise and governmental agencies. A research report published in 2019 [9] estimates that the market value of geospatial solutions will exceed $502 billion by 2024. The projection includes software, hardware, and geospatial services with software solutions estimated to take the largest share.

Analyzing spatial data applies hypothesis testing and pattern discovery against the dataset's spatial topological, geometric, and geographic properties. Spatial analysis benefit businesses and government agencies who use the results to improve user experience, product, and service innovations, improve mobility, urban planning, and enhance security [4–8, 10, 11]. For efficiency, executing a spatial query against a large dataset uses distributed processing frameworks like Apache Hadoop (Hadoop) [12] and Apache Spark (Spark) [13]. Hadoop is a collection of software components that include the Hadoop Distributed File System (HDFS) and MapReduce (MR). HDFS and MR allow for the distributed processing of large datasets with limited in-memory computation and without automatic fault tolerance. Spark allows for in-memory distributed processing and automatic fault-tolerance. Spark's Resilient Distributed Dataset (RDD) is a read-only distributed data structure that divides data between the available processing nodes. A Spark application can manipulate an RDD through actions and transformation. Spark automatically keeps track of an RDD's transformations lineage and can recover the failed RDD.

Hadoop and Spark offer the bare necessities for distributed execution using non-spatially aware partitioning and processing. As a result, the spatial data loses its locality, and the query becomes sluggish or produces inaccurate results. Subsequently, a distributed system requires a specialized layer known as a spatial extension to introduce spatial object and operation processing. To properly partition spatial data, the extension must exhibit the ability to construct a spatial partitioner that spatially groups (co-locate) records from two (or more) large spatial datasets. Record grouping depends on the spatial query and must comply with the dataset's characteristics like object type, density, and the size of non-spatial data. Further complications may arise from the spatial dataset's non-uniform distribution where certain regions contain higher densities. Figure 1 illustrates this problem; the lighter areas indicate higher densities of GPS points which require particular awareness by the partitioner when distributing the records.

Several works have proposed scalable partitioning techniques to aid with the distributed execution of spatial queries. Most take a similar approach of sampling the dataset at varying rates (e.g., 1%, 5%, 10%). Next, information from the sampled records is generalized to construct a partitioning scheme over the entire dataset. The scheme acts as a global index that shows how to spatially group records into partitions and allows the query to choose relevant partitions. However, this process may suffer from performance



(a) BUS dataset (b) TAXI dataset (c) TLC dataset

The lighter regions indicate higher point densities and require particular attention by the partitioner.

Fig. 1 Density plots of GPS points from three real-world datasets

and accuracy drawbacks. First, sampling relies on a subset of the dataset and can cause skewness. Second, it does not account for processing overheads of shuffling, indexing, or query execution. Ignoring these limits may cause skewness some nodes to serialize excess data or overflow the memory and fail.

This publication proposes a technique for customizing and building a scalable spatial partitioner for distributed spatial queries. The partitioner does not rely on sampling, requires minimal input, and automatically adapts to the datasets' traits like mixed objects, non-spatial data, and high-density regions. Moreover, it considers computational overheads, preserves spatial locality, mitigates query skews, and yields accurate results. Furthermore, it improves scalability by allowing the query to navigate hundreds and even thousands of partitions. By design, the partitioner eliminates the problem of boundary-crossing objects from the partitioned dataset and allows objects from the querying dataset to visit only relevant partitions.

A multitude of spatial queries like range, *k*NN, join [14–18], map-matching [19], and data visualization [20, 21] benefit from the proposed partitioner. To demonstrate the effectiveness of the design, a *k*NN spatial join query is implemented to perform a nearest-neighbor search [22] under the constraint of the data's spatial and non-spatial attributes. Accuracy and runtimes experiments compare the results of the implementation to several existing works. The contributions of this work are:

- A discussion of the challenges that face the construction of an efficient, accurate, and scalable spatial data partitioner for in-memory distributed spatial processing.
- Devise a balanced solution to the design challenges to derive an optimal spatial partitioner for a given dataset.
- A ready to use *k*NN spatial join query for Apache Spark available on GitHub.[1]
- A thorough experimental runtime and accuracy study using real-world datasets.

The rest of the paper discusses, first, the definitions used in this work. The following section surveys previously published big spatial data partitioning and processing techniques. Next, the proposed partitioner section details the design principles of the work introduced. The implementation section discusses the in-memory *k*NN spatial join

[1] https://github.com/bdilab/Spark-knn.

query Apache Spark implementation. The scalability and precision evaluations section describes several experiments performed over several test datasets and compares the results with those obtained through the baseline approach and several popular techniques. Finally, the paper concludes with a discussion, summary, and future research plans. The appendix contains the extended tabular results referenced in the scalability and precision evaluations. The appendix also shows several algorithms that explain various steps.

Concepts and definitions

This word uses several concepts and definitions, including ones borrowed from Apache Spark's architecture [23] given. Spark is one of the most popular in-memory computing frameworks for large-scale distributed processing [24, 25]. Its API supports scripts written in Scala, Java, Python, R, SQL, C#, and F# [26].

Memory size units: The memory units in this work are all in Bytes.

Minimum Bounding Region (MBR): The smallest region that fully encompasses a set of spatial objects. The MBR gives a general idea of the extent of an object (or objects) and can act as a quick indicator to operations like intersects and contains. For the rest of the discussion, we will refer to the Minimum Bounding Region as simply the *MBR*.

Spatial partitioner: A scheme that dictates how and where to group the database's spatial records amongst the distributed system's processing nodes. To preserve locality, the partitioner uses the data's spatial attribute to group records by criteria like proximity, size, or type. For repeated uses, the process may save the partitioner to disk. For the rest of the discussion, we will refer to the spatial partitioner as simply the *partitioner*.

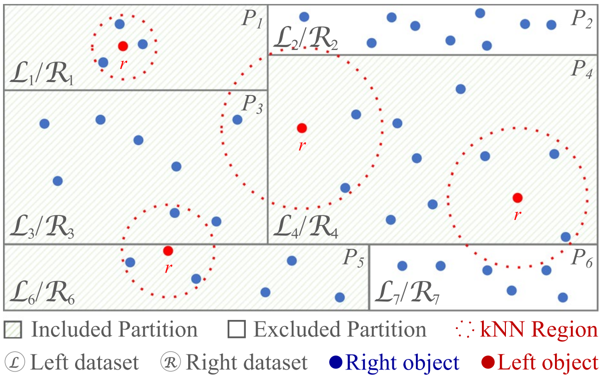
Shuffling: The process of moving data between the processing nodes. Shuffling is an expensive process that involves serialization, deserialization, network resources, and disk I/O operations.

Partitioning: The process of mapping and migrating the dataset's records to the proper partition as dictated by the partitioner. Partitioning requires the shuffling of one (or more) input datasets.

Pruning: A technique that allows a query to exclude some partitions that it deems irrelevant to its computations.

Partition: An atomic grouping of data. It is a logical division of the data retrieved from the underlying file system or constructed from the output of a previous processing task. A single executor's core processes one partition as a single task.

kNN Join query ($knn(\mathcal{L}, \mathcal{R}, k)$): An operation that takes as input two spatial datasets, $\mathcal{L}$ and $\mathcal{R}$, and $k \in \mathbb{N}, k > 1$. The query pairs each object $l \in \mathcal{L}$ with the k nearest objects from $\mathcal{R}$ sorted by proximity to l (e.g., Euclidean distance). Each l creates a search region with the smallest possible radius that encloses the k objects. Boundary-crossing regions require either shuffling the region or copying it to other partitions. Figure 2 shows several points from $\mathcal{L}$ after being matched with their nearest $k = 3$ neighbors from $\mathcal{R}$. The dotted circles represent the region's extent, which differs for each l and depends on the proximity of its 3 nearest neighbors.


P_1
P_2
$\mathcal{L}_1/\mathcal{R}_1$
$\mathcal{L}_2/\mathcal{R}_2$
P_3
P_4
r
r
r
r
$\mathcal{L}_3/\mathcal{R}_3$
$\mathcal{L}_4/\mathcal{R}_4$
P_5
P_6
$\mathcal{L}_6/\mathcal{R}_6$
$\mathcal{L}_7/\mathcal{R}_7$
▨ Included Partition
□ Excluded Partition
kNN Region
Ⓛ Left dataset
Ⓡ Right dataset
● Right object
● Left object

Each l object extends its region (dotted circle) depending on the proximity of its 3 nearest neighbors.

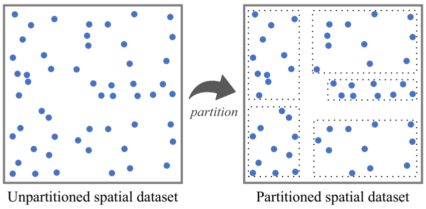
Fig. 2 Illustration of a spatial *k*NN join query between two datasets

All kNN join query ($kNN_{all}(\mathcal{L}, \mathcal{R}, k)$): A special case of the kNN spatial join query that takes as input two spatial datasets, $\mathcal{L}$ and $\mathcal{R}$, and $k \in \mathbb{N}, k > 1$. The query pairs each object $l \in \mathcal{L}$ as described for the kNN join query. In addition, kNN_{all} pairs each $r \in \mathcal{R}$ with the k nearest objects from $\mathcal{L}$ sorted by proximity to r. The query may execute as two consecutive kNN queries. However, for efficiency, it should account for both operations when partitioning and cache certain computations. Boundary-crossing objects may be present in either dataset; hence, object duplication or shuffling applies.

Related work

Partitioning is a powerful concept used to optimize many operations like distributed query processing, performance tuning of disk parallel systems [27], and query performance [28, 29]. This work studies the in-memory spatial partitioning problem in large spatial datasets and proposes a novel technique for building scalable partitioning over a given dataset.

Parallel processing of large spatial datasets is similar to the divide and conquer programming paradigm. Large data sets get broken into smaller, simpler parts that are processed individually; smaller solutions are then combined to solve the original problem. Thus, the challenge resets with constructing the best spatial partitioner with minimal overhead while maximizing resource utilization, reducing runtime, minimizing data skews, decreasing data shuffling, and producing accurate results.



Consider, for example, a spatial query over a large spatial dataset (e.g., terabytes in size) that wishes to find all objects within a specific search region (e.g., 1000*ft*). Without a partitioner, the query must examine all of the dataset's records (i.e., $O(n)$ operation). Alternatively, a spatial partitioner co-locates objects across partitions and allows the query to select those partitions within 1000*ft* of its center.

Similarly, a join spatial query operating over two spatial datasets can utilize the partitioner to co-locate objects from both datasets, thus avoiding an expensive complete scan operation (i.e., $O(n \times m)$ operation). Alternatively, a spatial partitioner regroups the first dataset's objects. Next, the second dataset's objects query the partitioner and merge with the partitions required to complete their query.

Several Hadoop MR spatial extensions add spatial query partitioning and processing to Hadoop. SATO [30] offers several spatial partitioning strategies of spatial datasets. Hadoop-GIS [31] partitions its input dataset by placing records into grid tiles and assigning tiles to partitions. Its performance significantly improves when integrated with SATO. SpatialHadoop [32] partitions its input datasets into equal-sized data partitions such that each one fits into the underlying HDFS block size and pads small partitions to the required size.

Similarly, several Apache Spark [13] spatial extensions add spatial query partitioning and processing to its in-memory processing. Since its release in 2014, Spark has become one of the most popular big data distributed processing systems. Its prominent feature is the Resilient Distributed Dataset (RDD) which allows for in-memory processing and automatic fault tolerance. Table 1 summarizes the most referenced and publicly open-sourced proposed Spark spatial extensions found in the literature. Magellan [33] extends Spark SQL and DataFrame API.[2] It takes advantage of the SQL query optimizer, does not offer any partitioning, but allows users to build a Z-Order Curve to index their dataset. Magellan supports five spatial objects and three spatial operations; however, for polygonal shapes, Magellan relies on their MBRs instead of their precise geometric shapes.

GeoSpark [14] extends Spark RDDs and introduces Spatial RDD (SRDD). It supports five spatial objects and grid partitioning, which samples the SRDD and builds a grid on the master node. Next, it divides the grid into equal-sized cells, assigns each object from the SRDD to a cell, and copies objects if they intersect with multiple cells. Load balancing requires an additional optimization round over the grid to break large cells. Similarly, LocationSpark [16] integrates itself with Spark through spatially aware RDDs. However, it mainly aims at solving the query skew problem and introduces sFilter, a spatial bloom filter for load-balancing and reducing data shuffles. LocationSpark samples the input dataset and builds a spatial index (e.g., R-Tree), and partitions the first input

[2] Spark's DataFrames are an extension on top of RDDs and was as an improvement API; however, a newer Dataset API will replace DataFrames in future releases.

Table 1 Feature comparison of Spark-based *k*NN spatial extensions

Feature	**Magellan**	**GeoSpark**	**LocationSpark**	**STARK**	**Simba**	**SpPart_ kNN (Proposed)**
Base code	Scala	Java	Scala	Scala	Scala	Scala
Modifies spark's core	No	No	Yes	Yes	Yes	No
Data partitioning	No	Grid, R-tree, QuadTree, KDB-Tree	Grid, QuadTree	Grid, Cost-Based Binary Space	Grid, R-Tree, Kd-Tree	Grid
*k*NN Type	None	*k*NN Point	*k*NN join	*k*NN join	*k*NN join	*k*NN join
Data pruning	No	No	No	Yes	Yes	Yes
Carry non-spatial data	**No**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Accounts for non-spatial data overhead	**No**	**No**	**No**	**No**	**Yes**	**Yes**
Skew handling level	**None**	**Partition**	**Query**	**Partition**	**Partition**	**Partition**
Indexing options	Z-Order Curves	None, R-Tree, QuadTree	None, Grid, R-tree, QuadTree, IR-tree.	None, R-Tree	HashMaps, TreeMaps, R-Tree	R-Tree and QuadTree
Index persistence	No	Yes	Yes	Yes	Yes	Yes
Spatial objects	Point, LineString, Polygon, MultiPoint, MultiPolygon	Circle, LineString, Point, Polygon, Rectangle	Box and Point	Inherited from JTS	Point, MBR	Point
Geometry library	Built-in	JTSPlus	JTS	JTS	Built-in	Built-in
Spatial operation object accuracy	Point only MBR for others	Point, Polygon, and Circle only partial for LineString and MBR for Rectangles	Point only MBR for others	Point only MBR for others	Point only MBR for others	Built-in with spatial operation

dataset according to the number of leaf nodes in the spatial index. A *k*NN join query in GeoSpark and LocationSpark, computes the maximum distance between each point and the furthest *k* match found for that point from the partitioner. If a region intersects with multiple partitions, a duplicate of the region's center point gets duplicated to perform local *k*NN query on all partitions before, finally merging the results of the actual *k* matches.

STARK [17] is a spatio-temporal framework that aims to optimize queries for datasets with spatial and temporal components. However, the temporal component is not part of its partitioning strategy. STARK partitions its RDDs through grid or binary space partitioning (BSP) built from subsamples of the dataset. For load-balancing, users must specify the optimal number of dimensions and the number of partitions for each dimension. The grid partitioner suffers from query skews; therefore, STARK offers an implementation of the BSP proposed in [34] and requires the user to input the maximum cost for each partition. For load-balancing, BSP divides the dataset's MBR into small cells and evaluates all possible partitioning candidates. Finally, it selects the partitioner with the lowest cost difference.

Simba [35] works with Spark DataFrames, provides range and hash partitioners for two-dimensional spatial data, and STRPartitioner for multi-dimensional data. It supports four spatial objects and has three partitioning strategies. Simba partitions a dataset and groups close-by objects on the same partition while ensuring that each partition receives the same number of objects. Afterward, each partition builds a local index (e.g., R-Tree), loads all rows into an array, collects statistics, calculates the partitions' MBRs, and computes the number of records. Finally, the master node collects statistics from all partitions and constructs the global index. Simba offers *STRPartitioner* which samples the input dataset and constructs a partitioning scheme as described earlier. Finally, it runs the first iteration of the Sort-Tile-Recursive (STR) algorithm [36] over the sampled records. Simba determines the number of partitions through a cost-based analysis on a set of randomly selected records. The partitioner must produce partitions that have roughly the same size, fit in the executor's memory, and proportional to the number of workers in the cluster.

SparkNN [37] for Apache Spark, FML-kNN [38] for Apache Hadoop, Spark, and Flink, [39] for Cassandra NoSQL database, and [40] for Hive-based Big Data Warehousing systems present partitioning techniques that follow the common approach of (1) sampling one or both input datasets, (2) computing partition boundaries, and (3) partitioning the dataset. Boundary crossing objects are either duplicated or shuffled between partitions. The random selection process occurs in parallel; the sampling rate is design-dependant [41] but is often a configurable parameter with common values between $1-5\%$. The partitioner is often indexed which simplifies querying; common indexing structures include R-Tree [14, 16, 17, 35], QuadTree [14, 16, 17], IR-Tree [16, 42], Grid [14], TreeMap [35], Treap Tree [35, 43], and HashMap [35].

The proposed spatial partitioner in this work opts for a different flow from the common approach. First, it does not rely on training data and customizes itself according to the input dataset's unique traits of distribution, object types, and size. Instead of relying on the non-deterministic nature of sampling, the proposed process scans both input datasets in a distributed fashion and collects and returns relevant information to the master node. This reveals the dataset's exact distribution (concentration regions), which helps with computing the optimal number of partitions, pruning, and load balancing skewed datasets like those depicted in Fig. 1 without requiring additional optimization rounds.

Second, unlike [14, 16], the partitioner preserves the dataset and does not eliminate records that it can not match. Moreover, unlike [14, 16, 17, 33, 35, 44] it accounts for non-spatial data by computing the cost of storing and shuffling that data while executing the spatial query. Third, it automatically maximize utilization of the available computing resources without relying on the user to provide information such as the best number of partitions, index depth, or maximum size [17, 35, 37].

Fourth, network traffic is minimized by eliminating object duplication and limiting shuffling to one input dataset. Boundary-crossing objects from one dataset migrate towards the first. Finally, our proposed partitioner ensures that a spatial query produces accurate results compared to other works and the baseline. Certifying the accuracy proves the reliability of a spatial query. As part of this work, we report on the accuracy of the results obtained from the tested technique.

Challenges

A distributed spatial query starts by spatially partitioning its datasets before performing tasks against its partitions. Given the distinct traits of each dataset, the process must acknowledge the dataset's unique characteristics (e.g., format, the presence of non-spatial data, and its density regions). Moreover, the process should efficiently utilize the available hardware resources (e.g., executors, CPU, disk, and memory). Next, we discuss the challenges that face the construction and execution of a spatial partitioner and query.

Data storage format

Variety, one of big data's 3Vs [45], classifies big data into either structured, semi-structured, or unstructured. Data files may be compressed or encrypted and stored in one of many different formats like plain text, Well-Known Text (WKT), Comma Separated Value (CSV), or GeoJSON [46]. Accounting for all formats is impractical or tedious at best. Therefore, a process can either adopt one of the existing data formats or develop a custom input format [14, 16, 17, 35, 47].

Non-spatial data

A spatial query is often part of a series of analytical steps that uses spatial and non-spatial data. Although a spatial query primarily operates over the spatial attributes, it should not discard others and must account for them during computation. Consider, for instance, the case of the unmet taxi demands [48] where drivers may be far away from the best potential location. The driver can make better decisions when considering climate, traffic, special events, and past pickups. While this information may be irrelevant to the spatial query, it is crucial for later steps.

Memory utilization

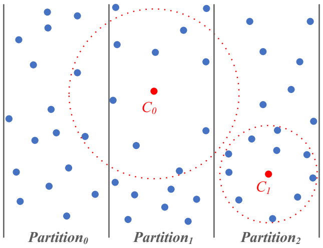
Although Random Access Memory (RAM) outperforms disk and network operations [49], its capacity is relatively smaller. Many partitioners do not closely account for the processing nodes' available RAM. Thus, it is counterproductive to attempt to use all of the available memory. In [50], the authors attempted to maximize memory usage in HBase [51] without accounting for growth due to computational overhead or shuffle restrictions like those seen in non-database distributed systems like Spark [52].[3] Our proposed partitioner assigns spatial objects to partitions such that each partition's final size never exceeds the node's memory storage fraction.

Spatial clustering

A distributed processing computation requires the grouping of related objects to the same partition. Hence, objects from both datasets must be brought together on the same executor to perform the necessary computations. Grouping objects is process-specific and performed using different measures like distance, equality, or shape. Moreover, several regrouping steps may be needed if objects require others located on different partitions. Several factors like load balancing, object duplication, and shuffling affect

[3] Before Spark version 2.4.0, users could not shuffle partitions larger than $2GB$ due to the serializer's use of byte arrays and ByteBuffers

performance and could slow execution. Our approach clusters objects by proximity; objects from the second dataset visit only the necessary partitions.



Query skew mitigation and load balancing

Load balancing calls for the even (or near-even) data distribution across the different partitions to reduce query skews. While it is difficult to achieve a perfect balance, the distribution should be as fair as possible, and account for any object duplication and shuffling. The figure to the right shows two search regions (dotted red line) centered at C_0 and C_1. Each region encloses the required 10 objects for this example. C_0 is located in $Partition_1$ and requires objects present on $Partition_0$. Therefore, objects from $Partition_0$ may be duplicated and sent over to $Partition_1$, or C_0 may be shuffled to $Partition_0$ after it finishes its computations on $Partition_1$. Object duplication increases the number of partitions or their size, avoids shuffling, but may require a final merge or filter of the final results. Shuffling, on the other hand, does not increase the number of partitions or their size but increases network traffic. Our proposal favors shuffling while attempting to minimize the number of shuffle rounds.

Optimal number of partitions

Initially, the distributed processing framework determines the number of partitions but allows users to adjust their number during execution. Choosing the optimal number of partitions is not a fixed process as it must account for the dataset's characteristics. Increasing the number of partitions reduces their size, underutilizes the executor's memory, and may increase the number of shuffle rounds. Reducing the number of partitions increases their size, causes the executor to write excess data to secondary storage, and may prolong serialization. Our proposal ensures that each partition receives approximately the same number of objects, ensures that the partition's final size never exceeds the executor's available memory, and adheres to any of the distributed processing framework's limitations.

Size of the partitioner

The spatial partitioner offers a global view of the distribution of records in a dataset. Often, it is constructed on the master node and distributed to the processing nodes. The information contained within the partitioner should be detailed enough to guide the spatial clustering of objects and guide the spatial query towards correctly completing its computations. Detailed partitioners contain precise information but require extended build times, a larger memory footprint, and additional time to broadcast it to the processing nodes. Less-detailed partitioners are faster to build and broadcast but may increase the number of shuffle rounds or produce inaccurate results. Moreover, it is preferred to limit the lifespan of the partitioner, extract its information early in the query process, and return its memory to the executor. As an added benefit, the partitioner may be written to disk and reused during subsequent queries over the same input datasets. Our proposal scans the entire dataset, collects relevant information, and summarizes it on the master node.

Spatial indexing

Indexing is an optimization technique used to speed up operations such as lookup, range, and *k*NN. Different spatial indexes perform better for different spatial objects. An R-Tree [53, 54] works best for polygonal shapes and works with their MBRs only. A K-dimensional tree (K-d tree) [55] organizes points in K-dimensional space. K-d trees are balanced trees that split the dimension plane and store points in leaf nodes. k-d trees are useful for point range and *k*NN quires. A Quadtree [56] is a two-dimensional tree that organizes objects in nodes with up to 4 child nodes. Quadtrees work best with point objects, but other variations exist to allow for polygonal object storage [57]. Grid indexing [58] splits the indexed space into equal-sized regions and works best with points. Our proposal starts with a Grid index that summarizes the dataset's distribution, identifies partition boundaries, and aids in identifying objects with boundary-crossing regions. Next, we transform the grid into a Quadtree for enhanced lookups.

Partition pruning

A spatial partitioner builds a distribution strategy over one dataset. Depending on the spatial query and the second dataset traits, some partitions from the first dataset may not contribute to the query's output. Therefore, the partitioner must allow the query to prune unneeded partitions without affecting accuracy or degrading execution time. An example of such an operation is shown in Fig. 2; the *k*NN query can safely exclude partitions P_2 and P_6 since they do not contribute to the results.

Proposed spatial partitioner

In this section, we detail a novel approach for constructing an efficient and scalable spatial partitioner for use with distributed in-memory spatial queries. The partitioner determines how to spatially divide the *right* objects across partitions; the *left* objects use the partitioner to migrate towards the correct partition(s). The partitioner accounts for the challenges described earlier and allows spatial queries to maximize resource utilization, decrease shuffling, and reduce execution time without compromising accuracy. In the next section, we discuss the steps of customizing the partitioner with a *k*NN spatial join query for execution over Apache Spark.

Table 2 shows the required and optional input parameters provided by the user. The two input datasets (RDD_{Right} and RDD_{Left}) are any two spatial datasets that consist of the

Table 2 Required and optional input parameters

Parameter	Type	Default	Description
RDD_{Right}	RDD		A Spark RDD of spatial objects[4]
RDD_{Left}	RDD		A Spark RDD of spatial objects[4]
gridDim	Integer	100	Initial grid spatial index cell size for grouping the *right* dataset objects that are within *gridDim*.
maxPartSize	Integer	0	The partition's maximum memory size in Bytes. If set to 0, the process uses all available memory.
spatialIndexType	SpatialIndex	Quadtree	Type of spatial index to use (e.g., Quadtree or Kd-Tree). Supported types extend a interface.
Other			Parameters specific to the spatial query (e.g., *k* for *k*NN, *range* for range query)

supported object types. [4] *gridDim* is the initial width and height of the grid spatial index cells. *maxPartSize* is the absolute maximum amount of memory that a partition can reach. This value is crucial for some parallel processing frameworks (e.g., Spark version 2.4 or earlier) to prevent memory overflow failures. If not specified, the partitioner will compute the partition's optimal byte size based on the executor's available memory. *spatialIndexType* allows the user to choose one of the supported spatial indexing structures (e.g., Quadtree or Kd-Tree). Finally, *other* is any additional parameter(s) needed by the spatial query, such as the value of k in a kNN spatial join query or the value of *range* in a range query.

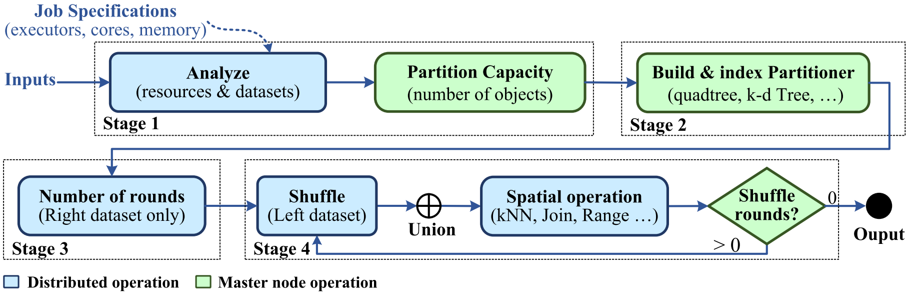
Figure 3 shows an overview of the steps followed by the proposed partitioner when embedded within a kNN spatial join query. The process consists of four phases. The first two stages analyze the input RDDs, estimate the partitions' capacity, and construct and index the partitioner. The final two stages relate to the execution of the kNN spatial join query, which requires computing the number of shuffle rounds and executing local query operations.

Stage 1: Analyzing the input datasets

The first stage in constructing the spatial partitioner examines both input datasets' spatial and non-spatial components, identifies the region with the highest density, estimates the cost of in-memory processing, and computes the total number of objects. Figure 4 highlights the steps taken during this stage. First, a parallel task indexes the *right* dataset's objects using a grid index with dimensions equal to the input parameter *gridDim*. Next, the master node receives the aggregated the results which show each grid cell's (X, Y) coordinates and the total number of objects grouped into that cell. The grid index in this stage offers three major advantages. First, it reduces the amount of shuffled data during the broadcast and aggregate operations. Second, knowing the number of objects that fall within each cell identifies the density region of the dataset. Third, the grid produces a summary that fits into the master node's memory.

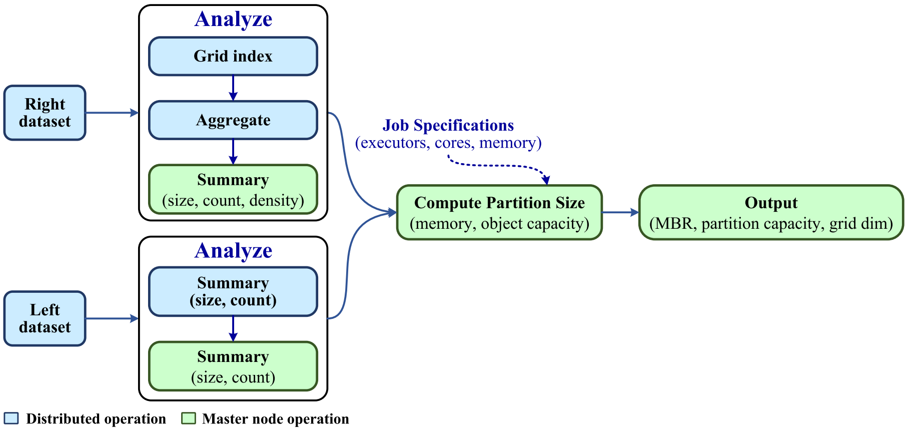
Next, the *left* dataset undergoes a similar but less detailed analysis process. We do not index the *left* dataset since we are only interested in its memory requirements when joined with the *right* dataset. The analysis step outputs:

[4] The partitioner works with custom light-weight spatial objects (e.g., Point) consisting of the coordinates (e.g., (X, Y)) and non-spatial data (e.g., trip records)



Stage 1 analyzes both input datasets and estimates the partition's object count; stage 2 constructs and indexes the spatial partitioner. Stage 3 computes the number of processing rounds; stage 4 executes the spatial query a number of times equal to that computed in Stage 3.

Fig. 3 Proposed partitioner stages (Stages 1 and 2) and spatial query (Stages 3 and 4)



Stage 1 analyzes both input datasets and produces information pertaining to the proper partitioning of the $right$ dataset while accounting for the merge step of the $left$ dataset during subsequent stages.

Fig. 4 Stage 1—analyzing the input datasets

Minimum bounding region (*right* dataset only): The MBR of the *right* (referred to as mbr_{Right} hereafter) shows the extent of the dataset's objects which is necessary for the construction of spatial indexes like the Quadtrees and K-d Trees. Only the *right*'s MBR is relevant since the *left* dataset is never indexed.

Maximum concentrations of objects (*right* dataset only): To mitigate query-skews without requiring users to know the exact distribution of their datasets; the process automatically finds the grid cell with the highest object density (referred to as $maxObjCount_{Right}$ hereafter). This value allows the adjustment the input parameter *gridDim* and reduce the partitions load imbalance in the next stage.

Largest record byte size (both datasets): In-memory computing degrades (or fails) if a task exceeds the available memory. Thus, the process should correctly estimate the byte size of cached structures and any operational costs. For efficiency, we assume that entries in both datasets occupy memory as much as the largest record in that dataset (referred to as $memMaxObj_{Right}$ and $memMaxObj_{Left}$ hereafter).

Total number of objects (both dataset): This value indicates the total number of objects in *right* and *left* datasets (referred to as $countObj_{Right}$ and $countObj_{Left}$ respectively). Knowing the exact number of objects aids in estimating the memory requirements and the maximum object capacity for each partition.

Using the analysis step outputs, we compute the optimal number of partitions and their object count without exceeding the available memory (or, *maxPartSize* if set by the user). First, we estimate the maximum memory size in bytes ($memMax_{Part}$) using formula 1. mem_{Exec} is the distributed job's assigned memory to each executor, $memOverhead_{Exec}$ is the executor's overhead memory necessary for its function as defined by the distributed system (e.g., Java Virtual Machine and reserved memory), and $countCores_{Exec}$ is the number of processing cores available for each executor. The value is finally divided by 2 to account for the merging of the *left* partitions.

$$memMax_{Part} = \frac{min(\frac{mem_{Exec} - memOverhead_{Exec}}{countCores_{Exec}}, maxPartSize)}{2} \tag{1}$$

Next, using $memMax_{Part}$ and formulas 2 and 3, we compute the number of partitions that both datasets require ($countPart_{Right}$, $countPart_{Left}$). $memOverhead_{SI}$ is the overhead associated with constructing the spatial index (e.g., internal nodes, lists, and pointers). The value is specific to the spatial index used (e.g., Quadtree or K-d Tree); if indexing is not required, then $memOverhead_{SI}$ is 0.

$$countPart_{Right} = \left\lceil \frac{(countObj_{Right} \times memMaxObj_{Right}) + memOverhead_{SI}}{memMax_{part}} \right\rceil \tag{2}$$

$$countPart_{Left} = \left\lceil \frac{(countObj_{Left} \times memMaxObj_{Left}) + memOverhead_{SI}}{memMax_{part}} \right\rceil \tag{3}$$

Using $countPart_{Right}$, $countPart_{Left}$, and formulas 4 and 5, we compute the minimum and the maximum number of partitions for the *right* dataset ($countPartMin_{Right}$ and $countPartMin_{Left}$). $countPartMin_{Right}$ is the *maximum* needed partitions between the two dataset. Doing so ensures that joining the two datasets will not overflow the executor's memory. The maximum number of partitions (formula 5) is the $countPartMin_{Right}$ adjusted to occupy all processing cores. This value may increase the number of partitions , but reduces the chance of idle cores.

$$countPartMin_{Right} = max(countPart_{Right}, countPart_{Left}) \tag{4}$$

$$countPartMax_{Right} = \left\lceil \frac{countPartMin_{Right}}{countCores_{all}} \right\rceil \times countCores_{all} \tag{5}$$

$$countCores_{all} = coutCores_{exec} \times count_{exec} \tag{6}$$

Finally, this stage outputs:

1. MBR of the *right* dataset (MBR_{Right}): This is the same value computed earlier.

2. Partition's minimum and maximum number of *right* dataset objects ($countObjMin_{Right}$ and $countObjMax_{Right}$): The range is produced using $countPartMax_{Right}$ and $countPartMin_{Right}$ and formulas 7 and 8.

$$countObjMin_{Right} = \left\lfloor \frac{countObj_{Right}}{countPartMax_{Right}} \right\rfloor \tag{7}$$

$$countObjMax_{Right} = \left\lfloor \frac{countObj_{Right}}{countPartMin_{Right}} \right\rfloor \tag{8}$$

3. Optimal grid dimension *gridDim*: If the maximum concentration of objects within a single grid square ($maxObjCount_{Right}$) is larger than the minimum number of objects per partition ($countObjMin_{Right}$), the load of the partitions may become unbalanced. In this case, we attempt to reduce the value of *gridDim*:

```
$rate \leftarrow \frac{maxObjCount_{Right}}{countObjMin_{Right}}$
if $rate > 1$ then
    $gridDim \leftarrow \frac{gridDim}{ceil(sqrt(rate)\ )}$        ▷ $gridDim$ is divided by $\lceil\sqrt{rate}\rceil$ since a single division of a
                                                                      square's sides splits that square into $4$. Two splits into $9$...
end if
```

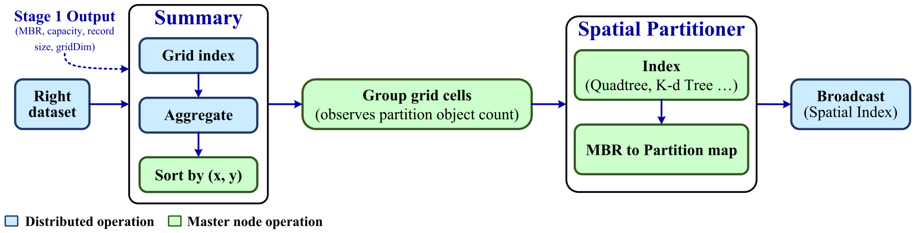
In the rare case of a highly skewed dataset, the adjusted value of *gridDim* may still produce a high concentration of objects (i.e., $rate > 1$). If this occurs, the process proceeds since recomputing $maxObjCount_{Right}$ requires multiple expensive computational rounds that will offset any gains from computing a partitioner without any query skews.

Stage 2: Constructing and indexing the partitioner

Using the output of state 1, stage 2 builds the partitioner over the *right* dataset. The partitioner (1) a spatial which simplifies lookup and (2) a Map that shows the partitions and their unique IDs. Each partition must receive and object count between $countObjMin_{Right}$ and $countObjMax_{Right}$. Figure 5 shows an overview of the steps taken during this stage which re-indexes the *right* dataset into a grid as described in stage 1, but with cell dimensions equal to the adjusted value of *gridDim*. The master node collects the aggregated summary of the grid and incrementally assigns non-empty grid cells to partitions.

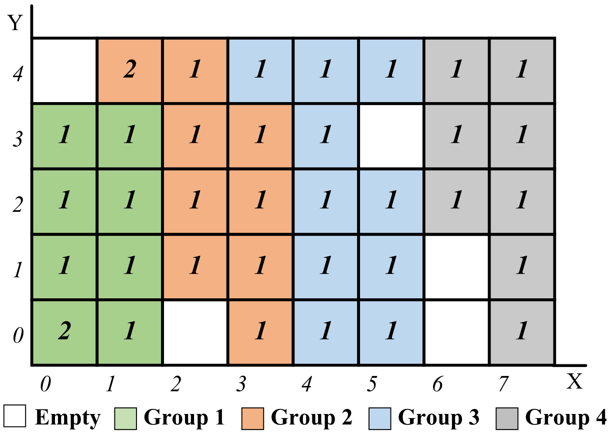
Figure 6 illustrates the grouping grid cells to partitions. The example shows a sample grid of 5 rows and 8 columns. Non-empty cells show the number of objects within. Assuming that the partition capacity is between 8 and 10, the first partition (green) receives a total of 9 objects since adding cell (1, 4) increases the sum to 11, which is greater than the maximum of 10. Subsequently, the partition 2 and 3 receive 10 objects each; the final partition gets the remaining 8. Simultaneously, a Map records the partitions' MBRs and their unique IDs as shown in table 3.

Finally, we index the grid (e.g., Quadtree or K-d Tree). This expedites lookup operations for boundary-crossing objects. Each entry in the spatial index records the grid cell's



Stage 2 builds the spatial partitioner using stage 1 computations. The stage indexes and only broadcasts the partitioner, which expedites querying on all processing nodes.

Fig. 5 Stage 2—constructing and indexing the partitioner



Y \ X	0	1	2	3	4	5	6	7
4	□	🟧 2	🟧 1	🟦 1	🟦 1	🟦 1	⬛ 1	⬛ 1
3	🟩 1	🟩 1	🟧 1	🟧 1	🟦 1	□	⬛ 1	⬛ 1
2	🟩 1	🟩 1	🟧 1	🟧 1	🟦 1	🟦 1	⬛ 1	⬛ 1
1	🟩 1	🟩 1	🟧 1	🟧 1	🟦 1	🟦 1	□	⬛ 1
0	🟩 2	🟩 1	□	🟧 1	🟦 1	🟦 1	□	⬛ 1

Example partition grouping for range 8 – 10. Partition 1 (green) receives 9 objects. Partitions 2 and 3 receive 10, and the partition 4 receive 8.

Fig. 6 Grid cell grouping illustration

(X, Y) coordinates, the total number of spatial objects assigned to the cell, and the partition number where the cell resides. Finally, this stage outputs:

1. Broadcast indexed spatial partitioner: this is the indexed spatial partitioner that is made available on all processing nodes.
2. Partition mapping: A map of the partitions IDs and MBRs. The map is useful for quickly looking up a partition's information and aids partition pruning.

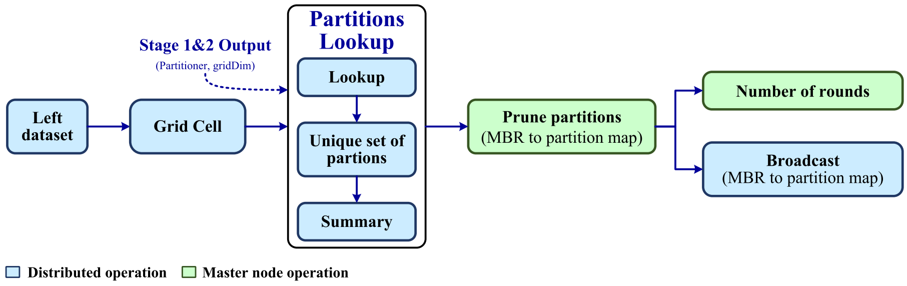
Stage 3: Computing the number rounds

Executing the spatial query starts by computing the number of shuffle rounds. This step is critical since the process does not duplicate boundary-crossing objects. Thus, the object with the longest partition list sets the number of shuffle rounds. For instance, in

Table 3 Partition MBR to ID mapping

Part. ID	MBR ends	Object count
0	(0, 0), (1, 3)	9
1	(1, 4), (3, 3)	10
2	(3, 4), (5, 4)	10
3	(6, 2), (7, 4)	8

An illustration of a map that shows the partition's MBRs and their unique IDs. The count column shown here for illustration; the implementation does not need to record the counts



Stage 3 concerns the spatial query. $left$ dataset boundary-crossing objects build a list of all partitions they cover. The object with the longest list sets the number of computational rounds. Simultaneously, the process maintains a list of all needed partitions and prunes the partition map produced in stage 2.

Fig. 7 Stage 3—Pruning and computing the number of computational rounds

a *k*NN spatial query, each object's region should extend to encompass the required *k* objects includes those residing on separate partitions (Fig. 2).

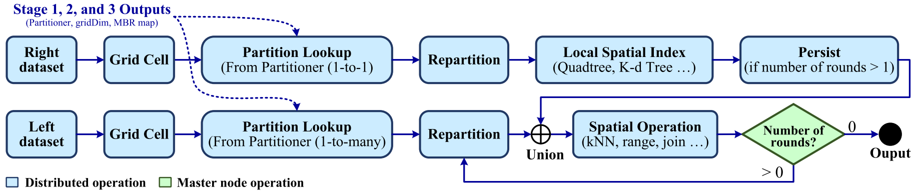
Figure 7 shows an overview of the steps taken during stage 3. Each *left* dataset object starts by computing its grid cell location and queries the spatial index to record the partitions to visit. Finally, the master node receives the aggregated results, which show the maximum length between all objects lists (*numRounds*) and a set of distinct partitions needed overall (*setNeededPartitions*).

Using the set of distinct partitions, *setNeededPartitions*, the master node attempts to update the partition mapping produced during stage 2 and removes partitions not present in *setNeededPartitions*. This pruning step allows the query to exclude unneeded partitions (if present). We should note here that pruning does not update the broadcast partitioner since the cost of unpersisting, rebuilding, and rebroadcasting the partitioner outweighs the cost of keeping the extra information. Finally, pruned map is broadcast to all processing nodes. Finally, this stage outputs:

1. Number of rounds: this is the length of the largest partition list (*numRounds*).
2. Broadcast partition mapping: this is the final pruned version of the hashmap.

Stage 4: Spatial query execution

The final stage executes the spatial query by repartitioning and grouping both datasets. Figure 8 shows an overview of the steps taken during this stage. The *right* dataset objects look up and migrate to their partitions, build a local spatial index, and persist their state.


Stage 1, 2, and 3 Outputs
(Partitioner, gridDim, MBR map)
Right dataset
Grid Cell
Partition Lookup (From Partitioner (1-to-1)
Repartition
Local Spatial Index (Quadtree, K-d Tree …)
Persist (if number of rounds > 1)
Left dataset
Grid Cell
Partition Lookup (From Partitioner (1-to-many)
Repartition
Union
Spatial Operation (kNN, range, join …)
Number of rounds?
0
> 0
Ouput
Distributed operation
Master node operation

Stage 4 partitions both datasets using the spatial partitioner. Optionally, each partition can build a local spatial index and persist its state in memory. Objects from the $left$ dataset select 1 (or more) partitions, merge with the $right$ dataset, and execute the spatial query locally. The process repeats a number of times equal to the number of rounds.

Fig. 8 Stage 4—executing the spatial query

Local indexing is optional but enhances lookups. Similarly, persistence is crucial for subsequent computations when *numRounds* > 1.

Likewise, the *left* dataset objects utilize the partitioner and select the partitions that each object requires. The lookup step is similar to the one performed in Stage 3; however, each object keeps the actual list of partitions sorted by proximity instead of just the count. Moreover, the order of the list is crucial since this ensures that the search region for each object can shrink once the number of matches reaches the required limit (e.g., k in kNN query). We should note that we opted to recompute the partition lists since persisting these computations may cause data spill to disk.

During this stage, the process accounts for query skews caused by the *left* dataset distribution. Depending on the distributed framework, selective exclusion of objects from the shuffle process may be prohibited. To mitigate skews, we ensure that the size of the partition list for all *left* objects equals to *numRounds*. Shorter lists receive randomly assigned partitions inserted at random places within the list while preserving the original order as shown in the following algorithm.

```
$rand \leftarrow RandomNumberGenerator()$
$listParts \leftarrow lookupFromPartitioner(spObj)$   ▷ $spObj$ is a $left$ dataset spatial object
while $listParts.length < numRounds$ do              ▷ Padding check
    $randIdx = rand.nextInt(numParts)$               ▷ numParts is the number of $right$ partitions
    $listParts.insert(rand.nextInt(listParts.length + 1), -randIdx)$
                                                     ▷ Padding partitions have negative ID.
                                                     ▷ Query operations are skipped there.
end while
```

After finalizing the list for all objects, the partitioner is removed from memory. This will increase the available memory on the executor and reduces the amount of serialization work that occurs during shuffling. Next, the process repartitions the dataset, and places each object on the first partition on its list. Next, both datasets join, and the *left* object query the *right* objects on that partition using the spatial index, if it exists, or through a full scan $O(n)$ operation. Finally, the steps of repartition, join, and query repeat a number of times equal to *numRounds*.

Apache Spark implementation − *k*NN spatial join query

The design details of our proposed partitioner apply to all distributed in-memory computing systems. In this section, we detail the implementation steps of the proposed partitioner for executing a spatial *k*NN join query of $two - dimensionalPoint$ spatial objects. We use this implementation in section to compare our work to existing *k*NN implementations on Apache Spark. The implementation provides interfaces that allow the addition of new spatial objects and indexes. We implement our spatial partitioner and *k*NN join query in Scala for execution with Apache Spark version 2.4.0.

Figure 9 shows an overview of the inputs and outputs to the implemented *k*NN spatial join query. RDD_1 and RDD_2 are Spark RDDs of type *Point*, $k \in \mathbb{N}, k > 1$, and RDD_{Out} is a Spark RDD of tuples consisting of all points from RDD_1 and a list of points from the RDD_2 sorted by proximity and size up to *k*.

Implementation of stage 1: analyzing the input datasets

Following the design details of stage 1, we compute MBR_{Right}, $countObjMin_{Right}$, $countObjMax_{Right}$, and *gridDim*. The computation occurs in parallel results collected on the master node. The executor's memory overhead ($MemOverhead_{Exec} = 300,000,000$) is set to the value specified by the Spark configuration [59, 60] and depicted in Figure 10, and compute mem_{Exec} using formula 9. $memJob_{Exec}$ is the amount of memory assigned to the Spark job; 0.60 and 0.50 are Spark's configuration parameter for the fraction of memory used for execution and storage and the amount of storage memory immune to eviction [59] respectively.

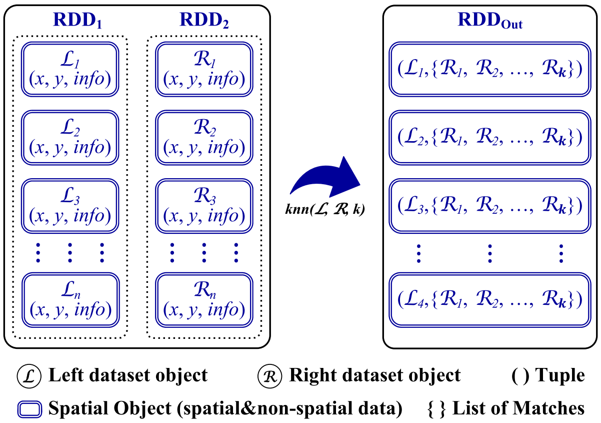
$$mem_{Exec} = (memJob_{Exec} - 300,000,000) \times 0.60 \times 0.50 \tag{9}$$

To estimate the memory size for objects, we use Spark's *SizeEstimator* [61]. However, due to some limitations whit estimating the size of deeply nested objects, we used *SizeEstimator* to estimate the object's shell and aggregate the results. Formula 10 shows the memory estimate for indexing the entire *right* dataset. $memEst_{SI}$ is an estimate of the memory required to index the dataset. This value is specific to the type and implementation of the spatial index and includes the cost of storing the index boundaries, internal nodes, and lists.

$$memEst_{Right} = countPartObj_{Right} \times memMaxObj_{Right} + memEst_{SI} \tag{10}$$

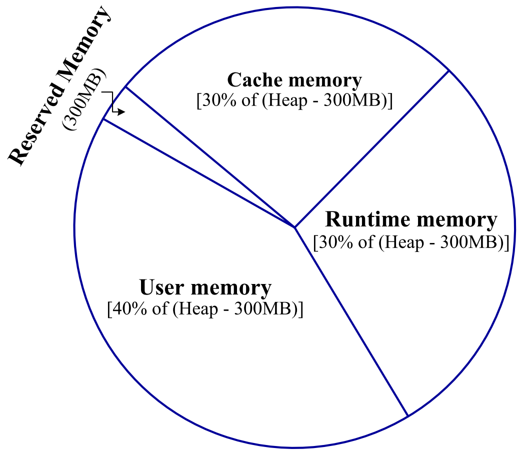
Implementation of stage 2: constructing and indexing the partitioner

Algorithm 1 in appendix C details the steps followed for building and indexing the partitioner. The inputs to the algorithm are the *right* RDD, the adjusted grid dimension, and the computed partition's size limits (*rightRDD*, *gridDim*, $countPointMin_{Right}$, and $countPointMax_{Right}$). A grid index groups the *right* dataset objects as described in the design details of stage 1 which produces an indexed partitioner and a map of partition IDs and their MBRs (*partitionerSI* and *mapMBR*).


RDD$_1$
RDD$_2$
$\mathcal{L}_1$ (x, y, info)
$\mathcal{L}_2$ (x, y, info)
$\mathcal{L}_3$ (x, y, info)
$\mathcal{L}_n$ (x, y, info)
$\mathcal{R}_1$ (x, y, info)
$\mathcal{R}_2$ (x, y, info)
$\mathcal{R}_3$ (x, y, info)
$\mathcal{R}_n$ (x, y, info)
knn($\mathcal{L}$, $\mathcal{R}$, k)
RDD$_{Out}$
($\mathcal{L}_1$,{$\mathcal{R}_1$, $\mathcal{R}_2$, …, $\mathcal{R}_k$})
($\mathcal{L}_2$,{$\mathcal{R}_1$, $\mathcal{R}_2$, …, $\mathcal{R}_k$})
($\mathcal{L}_3$,{$\mathcal{R}_1$, $\mathcal{R}_2$, …, $\mathcal{R}_k$})
($\mathcal{L}_4$,{$\mathcal{R}_1$, $\mathcal{R}_2$, …, $\mathcal{R}_k$})
$\mathcal{L}$ Left dataset object
$\mathcal{R}$ Right dataset object
( ) Tuple
Spatial Object (spatial&non-spatial data)
{ } List of Matches

The input is two Spark RDDs of type Point. The output is an RDD of all *left* dataset objects each paired with up to k points form the *right* dataset.

Fig. 9 *k*NN query input and output RDDs. The input is two Spark RDDs of type Point

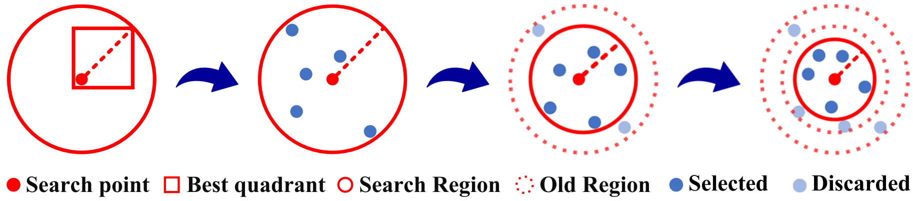


Each Spark executor reserves $300MB$; 40% of the remainder is given to the user's task, and the rest is split between caching and runtime RAM.

Fig. 10 Spark memory allocation

Stage 3: computing the number of processing rounds

Following the design details of stage 3, Algorithm 2 in Appendix C describes the computation process, which starts with the *left* dataset object computing their grid cell location, querying the spatial index, and selecting the number of partitions required to provide the required *k right* dataset objects. Simultaneously, the process maintains a distinct list of all partitions selected by the *left* objects.



The point starts by finding the smallest containing region with at least k objects. Next, the points searches the spatial index and shrinks the search regions after finding closer matches.

Fig. 11 Illustration of the *k*NN lookup process (Assuming $k = 5$)

The spatial index lookup process performed by each of the *left* objects is depicted in Fig. 11 and detailed in algorithm 3 in appendix C. The process finds the best region that contains at least *k* points. In a Quadtree, for example, the best region is the quadrant that contains the lookup point and has no fewer than *k* points. Next, we construct a search region with the search point as its center and dimensions *W*, *H* that contain the Quadtree's best region. Next, the point queries the partitioner starting from the best region, updates the *k* matches, and shrinks its search region. Next, the point queries the partitioner starting from the root element. Finally, we prune and broadcast *mapMBR*.

Stage 4: *k*NN Spatial join query execution

Following the design details of stage 4, and using the spatial index, the *right* dataset points migrate to their respective partitions, build a local spatial index, and persist to the executor's main memory. Next, the *left* dataset points extract a list of partitions needed to find the *k right* dataset points. Lists with sizes less than *numRounds* receive their random partition padding as described earlier in algorithm 4.

Next, the processing rounds commence; each *left* dataset point migrates to the first partition on its list and joined with the persisted *right* dataset partitioner. The join occurs via Spark's union transformation. For efficiency, we ensure that Spark internally switches to the *PartitionerAwareUnionRDD* operation.[5] After the join operation, each point in the *left* dataset performs a *k*NN lookup operation against the local spatial index as described earlier in figure 11.

Once the *left* points finish their *k*NN lookups, the process repartitions the *left* dataset and sends each point to the next partition on its list. Next, the *left* dataset is joined with the persisted *right* dataset and performs a *k*NN lookup. The steps continue several times equal to the number of processing rounds computed during stage 3 (Algorithm 2. Finally, the output consists of all the *left* points, with each point containing a list of points from the *right* dataset sorted by proximity.

Query scalability and outcome precision evaluations

The scalability and accuracy of a spatial partitioner and query are of equal importance. Their value diminishes if the results accuracy is weak or fails to handle large datasets within an acceptable period. Using our Spark implementation of the *k*NN spatial join

[5] Spark performs *PartitionerAwareUnionRDD* when the two datasets were partitioned using the same partitioner

query, we perform several experiments and compare the runtimes and precision of our approach to those found in the literature. Furthermore, we study the effect of the presence of non-spatial data by repeating tests with and without non-spatial data. We repeat each test three times and record the standard deviation of the runtimes under the same input parameters. For the remainder of this section, we refer to our implementation as *SpPart_kNN*. We utilize our previous work on accuracy benchmarking [47] and classify the results into several categories that compare the outputs for similarity and completeness of records. The benchmark implementation is available on GitHub[1]; it analyzes the contents of two files and generates a report of several classifications. In this section, we limit our discussion to the following criteria:

- Total records: Shows the total number of distinct records in both files. Records with the same key, count once, but records found in one file count as one too.
- Exact match record: The number of records with exact matches in both input files. The number of matches from the *second* file matching those from the *first* file must equal a value specified as an input parameter. If there is at least one mismatch, the record cannot receive this classification.
- Mismatch records: The number of records from the *second* file that were not classified as *Exact Match Record*. The sum of this classification and the *Exact Match Record* classification equals to *Total Records*.
- Missing records: The number of records that were present in the *first* input file but were missing from the *second* input file. This classification indicates that the tested technique omitted records it could not match.

Evaluation setup

The source code for our partitioner and kNN spatial join query is available on GitHub[1]. We examined several implementations of existing works on spatial partitioning and spatial query execution whose source code was available for testing. The examined works were Magellan[6], GeoSpark[7], LocationSpark[8], STARK[9], and Simba[10]. Magellan does not support kNN query, and we exclude it from further discussion. We exclude STARK and Simba from many of our test results since their jobs either failed or terminated after executing for more than 180 minutes. We chose to stop jobs after 180 to adhere to our data center's usability rules and to acknowledge that real-world applications usually operate under time and budgetary constraints. GeoSpark relies on the JTS [11] library and its *nearestNeighbour* function [62]; it does not offer kNN join query over two datasets but does allow for a single-point kNN query.

We conduct our tests at the operational data facility of our research center. The cluster consists of 20 high-end nodes; each node has 24*TB* of disk space, 256GB of RAM, and 64

[6] https://github.com/harsha2010/magellan.

[7] https://mvnrepository.com/artifact/org.datasyslab/geospark/1.3.1.

[8] https://github.com/purduedb/LocationSpark.

[9] https://github.com/dbis-ilm/stark.

[10] https://github.com/InitialDLab/Simba.

[11] https://locationtech.github.io/jts/.

Table 4 Experimental dataset summary sorted by the dataset's file size

Dataset Name	Short Name	Summary	Observations
OSM POI [63]	POI	• 38.814 MB • 119, 319 Points	• Open Street Map (OSM) points of interest • New York City (NYC) only • GPS location of buildings, restaurants, shops ...
NYC Bus Trip Records [64]	BUS	• 22.147 GB • 221.715 Mil. Points	• Similar format to the TAXI dataset but denser (Buses run over fewer city streets) • Non-uniform distribution (Fig. 1a) • Good for testing the behavior with locations significantly overloaded than others.
NYC Taxi Trip Records [65]	TAXI	• 27.738 GB • 165.114 Mil. Points	• Non-uniform distribution (Fig. 1b) • Ideal for testing techniques that cannot handle the LARGE dataset
TLC TPEP and LPEP [65]	TLC	• 141.99 GB • 3.78 Bil. Points	• Non-uniform distribution (Fig. 1c) • 10.9 Mil duplicate records. • 158.9 Mil unmatchable records.

AMD cores (total 1, 200+ cores) running Cloudera Data Hub 6.1.0 with Apache Spark 2.4.0. Each job may use up to 250 cores and 1.6*TB* of RAM.

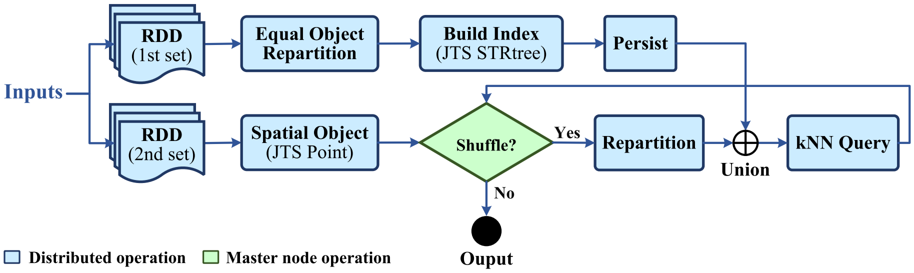
Datasets

Table 4 summarizes four real-world datasets employed in our evaluations. The datasets consist of GPS coordinates projected from WGS84 [66] to NAD83 datum [67] using the coordinate projection library PyProj [63]. The table lists the datasets in ascending order of their size. The first dataset, referred to as POI, consists of 119, 319 New York City (NYC) Points-of-Interest (POI) [68, 69] extracted from OpenStreeMap [63]. The remaining three datasets consist of records obtained from NYC Taxis [65] (TAXI and TLC) and Busses [64] (BUS). Each record in the later three datasets contains information about a single trip including, pick-up and drop-off locations, date/time, trip fair, number of passengers, and distance traveled.

The datasets consist of different sizes and densities. The BUS dataset size is close to the TAXI dataset; however, it is denser as busses only operate over selected city streets. If ignored, the density characteristic may cause skewness or load imbalance. The TLC dataset is the largest of the three (6.4 times larger than the BUS dataset) and contains far more GPS pings.

Baseline result design and implementation

Reliable accuracy evaluation requires result comparison against those certified as accurate (i.e., baseline). Due to the lack of baseline results for our datasets, we devised a technique to generate these results using exhaustive search; each point from one dataset examines every point from the other. Due to the computationally intensive nature of this approach, we extracted two small-scale POI datasets consisting of 100 and 200 POI points, respectively. For each point in each small-scale dataset, we perform a *k*NN query operation against the BUS dataset. The BUS dataset is denser and has fewer records than the TAXI and TLC datasets. The task for the 100 POI points finished its *k*NN query in 6.55 hours while the second task for the 200 POI points lasted 11.34 hours. We performed spatial object recognition and *k*NN lookup operations using LocationTech's JTS



Producing the baseline result follows an exhaustive search process. The process distributes the $right$ objects into several equal-sized partitions, indexing, and persistence in memory. The $left$ objects query each $right$ spatial index and selects the closest k objects.

Fig. 12 Workflow for producing the *k*NN baseline result.

library [70]. Namely, we used the library's *Point* object and *STRtree*, an R-Tree spatial index implementation.

Figure 12 shows the workflow for generating the baseline results using Spark. The process starts by evenly repartitioning the BUS dataset amongst several partitions of 300,000 points. We chose this number experimentally after several trials. Next, on each partition, we constructed an *STRtree* and persisted it in memory. Next, we transformed the POI dataset to JTS *Point* objects, repartitioned the dataset, and joined it with *STRtree* BUS partitions. Each point performs its *k*NN query steps and keeps up to 50 best matches. Once finished, the POI points migrate to the next partition; the process continues until all points query all *STRtree*. The final output consisted of the original trip record followed by its matches sorted by proximity.

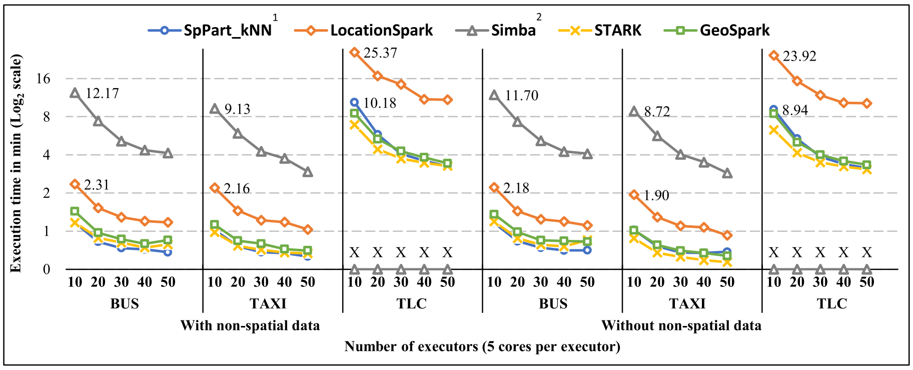
Spatial partitioner construction time evaluation

Our first round of evaluations examines the time needed to construct the spatial partitioner with and without non-spatial data. We fix the size of the input dataset (BUS, TAXI, TLC) and vary the number of executors for each subsequent test by adding 10 new executors (i.e., 10, 20, 30, 40, to 50) while fixing the number of cores per executor at 5[12]. The results of this test show how a technique analyzes the dataset and utilizes the available computing resources to build its spatial partitioner.

Figure 13 shows the average runtimes that each technique took to construct its partitioner for the three datasets BUS, TAXI, and TLC with and without non-spatial data. Each value is the average of three repeated tests with standard deviation ranges shown in Table 6. Values labeled with an "X" indicate that the test failed or terminated after executing 180 minutes.

Test results indicate awareness of the added computing resources, with runtimes remaining close or decreasing. SpPart_kNN, GeoSpark, and STARK finished with close runtimes. SpPart_kNN was 1.01 − 1.15 times faster than the following quickest technique (varies per test) during 10 of the 15 tests without non-spatial data, 1.02 − 1.17 times faster in 5 of the 15 tests with non-spatial data. Table 6 in appendix A shows that

[12] The 5 core per executor setting allows the executor to perform up to 5 parallel tasks. Increasing this number caused congestion within the executor and degraded performance



(1. Does not sample data. 2. Simba TLC tests failed or stopped after 180 minutes)

Test results indicate awareness of added computing resources with runtimes decreasing or remaining close. Overall, SpPart_kNN was $1.01 - 1.17$ times faster in 15 of the 30 tests with better standard deviation during 15 tests, SIMBA was better in 10 tests, failed all TLC tests, and showed worst standard deviation of repeated tests, STARK was best in 6 tests in runtime and standard deviation, and LocationSpark and GeoSpark were slowest with 4 and 5 better standard deviation respectively.

Fig. 13 Cost of building the partitioner with and without non-spatial data

SpPart_kNN exhibited better overall consistency during 15 of the repeated tests, SIMBA was better in 5 of the tests for both with and without non-spatial data, STARK was better in 1 and 5 tests with and without non-spatial data, respectively, and LocationSpark and GeoSpark showed the slowest times in all tests with GeoSpark finishing ahead. We should reemphasize that our approach performs a complete distributed dataset scan, collects minimal details, and does not rely on sampling as commonly done in the other tested techniques.

*k*NN join query—small-scale datasets

In this section, we report on small-scale datasets experiments to highlight the importance of assessing the accuracy of spatial query results. As noted earlier, we were unable to perform kNN query tests over all studied techniques due to either lack of support or failure. For these techniques, we obtain results by iteratively invoking their single point kNN spatial query method, namely, we use the *KNNQuery.SpatialKnnQuery*() and *kNNJoin*() methods for GeoSpark and STARK respectively. We omit reporting on runtime results in this section since iteratively invoking kNN point query is less efficient than a kNN spatial query

Accuracy comparison results

Table 5 summarizes the results of comparing the outputs of each of the tested techniques and the baseline matches for the small-scale dataset. The matches of SpPart_kNN, Simba, and GeoSpark agree with the baseline's results. STARK's matches agree with the baseline's $k = 10$ only but mismatched one record from both POI datasets for $k = 50$. The single mismatched record in STARK contained a point that was further than the one found by the baseline approach. LocationSpark's matches showed the worst results, agreeing with the baseline matches by 80% for $k = 10$ and $29 - 33\%$ for $k = 50$.

Table 5 Accuracy evaluation results—summary for small-scale query dataset

			—○— SpPart_kNN	—◇— LocationSpark	—△— Simba	—×— STARK	—□— GeoSpark
With non-spatial data	100 Points	10	**✓**	80.0%	✓	✓	✓
		50	**✓**	33.0%	✓	99.0%	✓
	200 Points	10	**✓**	80.0%	✓	✓	✓
		50	**✓**	29.5%	✓	99.5%	✓
Without non-spatial data	100 Points	10	**✓**	80.0%	✓	✓	✓
		50	**✓**	33.0%	✓	99.0%	✓
	200 Points	10	**✓**	80.0%	✓	✓	✓
		50	**✓**	29.5%	✓	99.5%	✓

A ✓ indicates that the results completely agree with the baseline matches

LocationSpark matched points that were further than those found through the baseline approach.

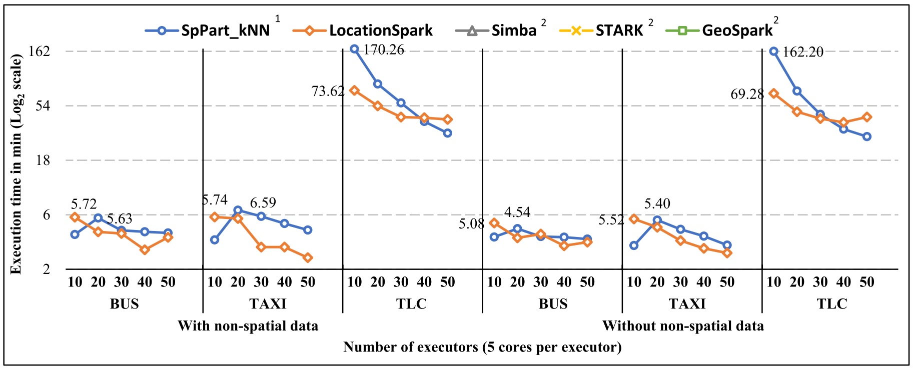
Table 9 in Appendix B shows the results of comparing the outputs of all techniques against *each other*. The results of SpPart_kNN, Simba, and GeoSpark agreed with each other and with STARK matches for $k = 10$ but missed one record for $k = 50$. These techniques agreed with LocationSpark matches by 21% and 71% for $k = 10$ and by 21.5%, and 72.5% for $k = 50$. These results were in line with our findings during later accuracy evaluations which lack baseline results. In these experiments, we consider the output with the closest matches as the better approach.

*k*NN join query-scalability evaluation

The scalability test examines the runtime and accuracy of the partitioner and the kNN spatial join query under different processing configurations. The tests fix the input size to the entire dataset (BUS, TAXI, and TLC) and gradually increase the number of executors by 10 with $k = 10$.

Figure 14 shows the average runtimes for successful tests. The runtimes overall decreased with the addition of new executors. SpPart_kNN showed a runtime spike in the BUS and TAXI between $1.18 - 1.82$ times over the previous tests with 10 executors. Upon examining the runtime logs, we noticed that during these tests, the partitioner increased its estimated optimal number of partitions by 33%. The cause of the misestimate is in the subprocedure that tries to adjust the number of partitions to become divisible by the number of cores. This increase, in turn, required 2 additional shuffle rounds. Adjusting the number of partitions relative to the cores' count increases the utilization of cores (i.e., fewer idle cores); however, depending on the dataset's density and size, it may slightly increase the number of shuffle rounds. This behavior was not a factor in subsequent experiments during the 30, 40, and 50 executors or for the TLC tests. We will study this behavior further and propose a fix in later versions.

SpPart_kNN was 1.08—1.59 times faster than LocationSpark in 4 of the 15 tests without non-spatial data and 1.06—2.31 times slower in the remaining 11. In the tests with



(1. Does not sample data. 2. Unsupported operation, failed, or stopped test)

Test results indicate that the runtimes decreased or remained close with the addition of new executors except for SpPart_kNN during the BUS and TAXI tests with 20 executors due to an anomaly in computing the number of partitions. While LocationSpark showed faster runtimes, its performance advantage diminishes due to its poor accuracy results.

Fig. 14 Scalability test with and without non-spatial data ($k = 10$)

non-spatial data, SpPart_kNN was 1.05—1.71 times faster in 5 out of the 15 tests and 1.09—2.34 times slower in the remaining 10. However, LocationSpark tests were found unreliable due to their low accuracy score, as discussed next. Table 7 in appendix A shows that SpPart_kNN exhibited better overall consistency than LocationSpark during 27 of the repeated tests.

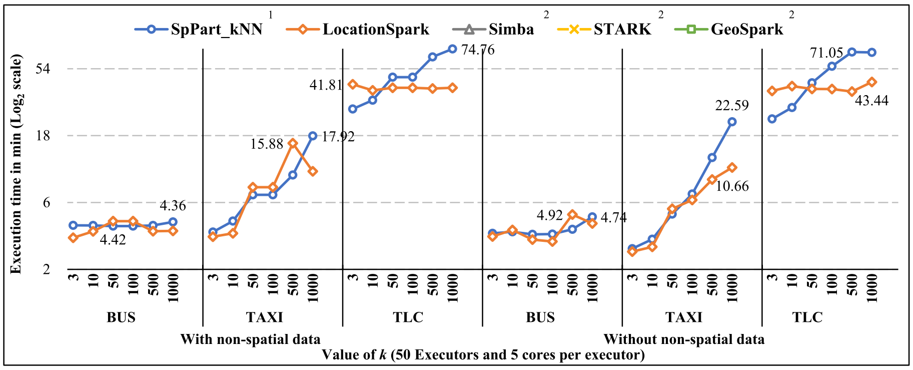
Accuracy comparison results

Table 10 in Appendix B shows the results of comparing the outputs of successful tests. For tests with non-spatial data, LocationSpark and SpPart_kNN results agreed, at best, by 81.50% of the POI records for the BUS dataset tests, 82.01% for the TAXI dataset tests, and 85.69% for the TLC dataset tests. We independently examined and verified the remaining parts of the results, and found that SpPart_kNN matched points with closer matches than LocationSpark. In other words, SpPart_kNN found better matches than LocationSpark in 18.50% of the POI records for the BUS dataset tests, 17.99% for the TAXI dataset tests, and 14.31% for the TLC dataset tests.

For the tests without non-spatial data, the percentages were close to those with spatial data. LocationSpark and SpPart_kNN results agreed, at best, by 80.67% of the POI records for the BUS dataset tests, 82.12% for the TAXI dataset tests, and 85.63% for the TLC dataset tests. Similar to the tests with non-spatial data, SpPart_kNN produced closer matches for the remaining parts of the results.

*k*NN join query evaluation—varying the value of *k*

Our final round of evaluations examines the effect of increasing the value of k for a kNN spatial join query. We observe how the kNN query utilizes the partitioner to account for



(1. Does not sample data. 2. Unsupported operation, failed, or stopped test)

Test results show that runtimes for SpPart_kNN increased or remained close as the the value of k increased. For LocationSpark, runtimes were less than expected especially for the TLC dataset. This translated into poor accuracy results compared to SpPart_kNN

Fig. 15 Varying the value of *k* with and without non-spatial data

the increase in k, which expands the range for each query point as k grows. This, in turn, increases the number of boundary-crossing regions, decreases the chance of partition pruning, and requires additional shuffling rounds or object duplication.

Accuracy comparison results

Figure 15 shows the average runtimes of several kNN queries using six different values for k (3, 10, 50, 100, 500, and 1, 000). We fix the input size to the entire dataset (BUS, TAXI, and TLC) and set the number of executors to 50. The BUS dataset showed the best runtime consistency except for LocationSpark during the test with $k = 1000$ without non-spatial data. For the TAXI dataset, both SpPart_kNN and LocationSpark showed increasing runtimes except for LocationSpark during the test with $k = 1000$ with non-spatial data. We should reiterate that the BUS and TAXI datasets are similar in size; however, the BUS dataset is denser as it spans fewer.

The most interesting observation is seen in the runtimes of LocationSpark for the largest dataset (TLC). During these tests, the runtimes remained very close even as k increased. This behavior was clarified when we reviewed the accuracy results and noticed the degraded accuracy of the results compared to SpPart_kNN. SpPart_kNN was 1.08—1.69 times faster during 7 of the 18 tests with non-spatial data and 1.03—1.58 times faster during 5 of the 18 tests without non-spatial data. Table 8 in appendix A shows that SpPart_kNN exhibited better overall runtime variations with a range of 0.016—3.289 compared to 0.039—11.362 for LocationSpark.

Accuracy comparison results

Table 11 in appendix B shows the outputs comparison results for the successful tests. LocationSpark accuracy degraded as k increased compared to SpPart_kNN. For $k = 3$

for the tests with non-spatial data, LocationSpark agreed with SpPart_kNN by 91.19%, 95.31%, and 96.66% of the POI records for the BUS, TAXI, and TLC datasets respectively. For $k = 1000$, LocationSpark agreed with SpPart_kNN by 0%, 14.31%, and 0% for the three datasets respectively.

For the tests without non-spatial data, the percentages were close to those with spatial data. For $k = 3$, LocationSpark and SpPart_kNN results agreed, at best, by 90.86% for the BUS dataset, 95.29% for the TAXI dataset, and 96.67% for the TLC dataset. For the remaining portion, SpPart_kNN produces results with closer matches; 9.14%, 4.71%, and 3.33%. Overall, LocationSpark produced a higher percentage of imprecise matches than SpPart_kNN as the value of k increased.

Discussion

The proposed partitioner exhibits performance gains over existing works due to several factors. First, it customizes itself to the dataset's exact traits without sampling the input datasets or relying on training data. Second, it gathers precise information about the dataset's object types, size, and high-density locations. Third, it naturally accounts for query skews and load balancing and avoids the need for additional optimization rounds that could offset runtime gains. Fourth, the partitioner computes the dataset's memory byte size requirements, including its non-spatial data.

Fifth, the construction of the partitioner is independent of the spatial query type. This flexibility allows any query to extract information from the partitioner and build a list of partitions to visit to avoid object duplication and minimize shuffling. Sixth, the size of the partitioner is minimal and only stores relevant information about objects' location and counts. Finally, the partitioner performs most computations like grid assignment and aggregation in parallel. The master node only collects results, distributes the load, and indexes the information to improve lookup.

In addition to the performance gains, the proposed partitioner allows the spatial query to produce accurate results. The comprehensive information offered by the partitioner permit complex queries like the kNN spatial join query to learn enough about which partitions contribute to the output. A boundary-crossing object that overlaps a partition can learn the exact number of objects that may fall within its region. Users can choose to allow the partitioner to select the proper grid size or adjust it for coarse or fine sizes.

We plan to enhance our proposed spatial partitioner and address some of its current limitations. The partitioner can account for density spots within the database. However, less dense datasets may increase the size of the indexed partitioner, which in turn requires additional partitions and shuffling. In extremely skewed datasets, the partitioner may bypass the object size estimate. While the two problems will not cause the distributed task to fail or reduce accuracy, it may increase the runtime and space complexities. The solution is for the user to adjust the input parameter *griDim* to an optimal setting specific to the input datasets. In addition, the density affects the complexity of the spatial index used. For instance, Quadtrees work well when they are balanced or near-balanced. Dense datasets may cause the tree to grow in one specific direction more than others.

While trajectory matching is not part of the studied problem; however, given the flexibility of the design, trajectory analysis is easily applied to the output of the spatial query. Moreover, the design is not limited to one spatial type; adding additional types may be introduced as the partitioner can account for mixed-object datasets. Finally, the partitioner does not currently recognize the temporal attribute within the dataset. Such a feature is planned for future work to integrate and ensure the scalability and accuracy of spatial and temporal partitioning.

Conclusion and future work

The preceding work proposed a novel approach to spatial data partitioning for in-memory distributed processing frameworks and detailed several challenges facing partitioner construction, scalability, and query processing. The proposed solution takes a balanced approach to the challenges and does not compromise the spatial query's accuracy. The implementation section discussed the details for integrating the partitioner with a *k*NN spatial join query for execution on Apache Spark called SpPart_kNN.

Several experiments were conducted using real-world datasets and compared to several popular and publicly available spatial partitioning techniques for Apache Spark. Although SpPart_kNN does not rely on sampling, its partitioner construction time was up to 1.17 during 15 of the 30 experiments and up to 1.50 times slower in the rest of the tests. In the scalability tests, SpPart_kNN was up to 1.59 times faster than LocationSpark in 9 of the 30 experiments and up to 2.34 times slower. However, the accuracy results of LocationSpark were unreliable, which explains its runtime advantage. During the experiments that varied the value of *k*, the results were similar; SpPart_kNN was up to 1.69 times faster during 12 of the 24 tests and up to 2.12 times slower than LocationSpark during the remaining tests. However, the accuracy results of LocationSpark showed weaker results compared to SpPart_kNN, which degraded as the value of *k* increased.

For future releases, we plan to continue our study into spatial partitioning and address the limitations encountered during this work. We plan to investigate finding a more efficient estimate of the dataset's memory requirements (e.g., mean or mode instead of maximum). We believe that this may reduce the partition size estimate, which in turn will decrease the number of partitions and shuffle rounds. The memory size estimator provided by Spark (i.e., *SizeEstimator*) may overestimate objects sizes; thus, a preciser size estimator is needed. We also plan to introduce support for new spatial shapes like LineStrings and Polygons and expand the list of supported operations to others like *join* and *intersect*. Finally, for better usability with Spark, we plan to offer tighter integration with Spark's RDD through Scala's Domain-Specific Languages (DSL)[13].

[13] https://www.scala-lang.org/old/node/1403.

Appendix

Standard deviation tables

Table 6 Standard deviation for repeated tests—partition build cost (Fig. 13)

	With non-spatial data					Without non-spatial data				
	—○—	—◇—	—△—	—×—	—□—	—○—	—◇—	—△—	—×—	—□—
Bus	0.011–0.027	0.000–0.048	0.021–0.126	0.022–0.060	0.008–0.050	0.003–0.018	0.014–0.103	0.000–0.000	0.008–0.223	0.021–0.286
Taxi	0.003–0.012	0.008–0.028	0.028–0.110	0.007–0.018	0.004–0.021	0.000–0.024	0.014–0.017	0.014–0.150	0.004–0.027	0.004–0.015
TLC	0.008–0.637	0.229–0.974	0.000–0.000	0.008–0.223	0.021–0.286	0.008–0.488	0.342–0.902	0.000–0.000	0.008–0.055	0.027–0.049

—○— SpPart_kNN —◇— LocationSpark —△— Simba —×— STARK —□— GeoSpark

Table 7 Standard deviation for repeated tests—Scalability (Figure 14)

	With non-spatial data					Without non-spatial data				
	—○—	—◇—	—△—	—×—	—□—	—○—	—◇—	—△—	—×—	—□—
Bus	0.042–0.122	0.109–0.617	0.000 –0.000	0.000–0.000	0.000–0.000	0.024–0.146	0.055–0.766	0.000–0.000	0.000–0.000	0.000–0.000
Taxi	0.049–0.270	0.418–1.029	0.000–0.000	0.000–0.000	0.000–0.000	0.008–0.461	0.111–0.599	0.000–0.000	0.000–0.000	0.000–0.000
TLC	0.279–1.270	0.922–5.063	0.000–0.000	0.000–0.000	0.000–0.000	0.140–1.481	1.681–6.452	0.000–0.000	0.000–0.000	0.000–0.000

—○— SpPart_kNN —◇—LocationSpark —△—Simba —×—STARK—□— GeoSpark

Table 8 Standard deviation for repeated tests – Varying the value of *k* (Figure 15)

	With non–spatial data					Without non–spatial data				
	—○—	—◇—	—△—	—×—	—□—	—○—	—◇—	—△—	—×—	—□—
Bus	0.034–0.083	0.068–0.818	0.000–0.000	0.000–0.000	0.000–0.000	0.039–0.082	0.039–1.857	0.000–0.000	0.000–0.000	0.000–0.000
Taxi	0.016–0.310	0.082–11.362	0.000–0.000	0.000–0.000	0.000–0.000	0.016–0.143	0.110–1.845	0.000–0.000	0.000–0.000	0.000–0.000
TLC	0.532–2.453	0.582–2.117	0.000–0.000	0.000–0.000	0.000–0.000	0.111–3.289	1.341–5.295	0.000–0.000	0.000–0.000	0.000–0.000

—○— SpPart_kNN —◇— LocationSpark —△— Simba —×— STARK —□— GeoSpark

Accuracy comparison tables

Table 9 Accuracy evaluation results—small-scale query test

K		Percentage of records											
		Exact Match	Mis-match	Missing	Exact Match	Mis-match	Missing	Exact Match	Mis-match	Missing	Exact Match	Mis-match	Missing
		With non-spatial data			Without non-spatial data			With non-spatial data			Without non-spatial data		
		—○— SpPart_kNN VS —△— Simba						—○— SpPart_kNN VS —□— GeoSpark					
100 Records	10	100%	0.00%	0.00%	100%	0.00%	0.00%	100%	0.00%	0.00%	100%	0.00%	0.00%
	50	100%	0.00%	0.00%	100%	0.00%	0.00%	100%	0.00%	0.00%	100%	0.00%	0.00%
200 Records	10	100%	0.00%	0.00%	100%	0.00%	0.00%	100%	0.00%	0.00%	100%	0.00%	0.00%
	50	100%	0.00%	0.00%	100%	0.00%	0.00%	100%	0.00%	0.00%	100%	0.00%	0.00%
		—○— SpPart_kNN VS —×— STARK						—○— SpPart_kNN VS —◇— LocationSpark					
100 Records	10	100%	100%	0.00%	100%	0.00%	0.00%	80.00%	20.00%	0.00%	79.00%	21.00%	0.00%
	50	99.00%	99.00%	1.00%	99.00%	1.00%	0.00%	33.00%	67.00%	0.00%	29.00%	71.00%	0.00%
200 Records	10	100%	100%	0.00%	100%	0.00%	0.00%	80.00%	20.00%	0.00%	78.50%	21.50%	0.00%
	50	99.50%	99.50%	0.50%	99.50%	0.50%	0.00%	29.50%	70.50%	0.00%	27.50%	72.50%	0.00%
		—□— GeoSpark VS —×— STARK						—□— GeoSpark VS —◇— LocationSpark					
100 Records	10	100%	0.00%	0.00%	100%	0.00%	0.00%	80.00%	20.00%	0.00%	79.00%	21.00%	0.00%
	50	99.00%	1.00%	0.00%	99.00%	1.00%	0.00%	33.00%	67.00%	0.00%	29.00%	71.00%	0.00%
200 Records	10	100%	0.00%	0.00%	100%	0.00%	0.00%	80.00%	20.00%	0.00%	78.50%	21.50%	0.00%
	50	99.50%	0.50%	0.00%	99.50%	0.50%	0.00%	29.50%	70.50%	0.00%	27.50%	72.50%	0.00%
		—△— Simba VS —×— STARK						—△— Simba VS —◇— LocationSpark					
100 Records	10	100%	0.00%	0.00%	100%	0.00%	0.00%	80.00%	80.00%	20.00%	79.00%	21.00%	0.00%
	50	99.00%	1.00%	0.00%	99.00%	1.00%	0.00%	33.00%	33.00%	67.00%	29.00%	71.00%	0.00%
200 Records	10	100%	0.00%	0.00%	100%	0.00%	0.00%	80.00%	80.00%	20.00%	78.50%	21.50%	0.00%
	50	99.50%	0.50%	0.00%	99.50%	0.50%	0.00%	29.50%	29.50%	70.50%	27.50%	72.50%	0.00%

Table 9 (continued)

		△ Simba VS □ GeoSpark						× STARK VS ◇ LocationSpark					
100 Records	10	100%	0.00%	0.00%	100%	0.00%	0.00%	80.00%	20.00%	0.00%	79.00%	21.00%	0.00%
	50	100%	0.00%	0.00%	100%	0.00%	0.00%	33.00%	67.00%	0.00%	29.00%	71.00%	0.00%
200 Records	10	100%	0.00%	0.00%	100%	0.00%	0.00%	80.00%	20.00%	0.00%	78.50%	21.50%	0.00%
	50	100%	0.00%	0.00%	100%	0.00%	0.00%	29.50%	70.50%	0.00%	27.50%	72.50%	0.00%

Table 10 Accuracy evaluation results—scalability test

	Executors	Exact match	Mis-match	Missing	Exact match	Mis-match	Missing
		With non-spatial data*			Without non-spatial data*		
		—○— SpPart_kNN VS —◇— LocationSpark					
BUS Records (119, 319)	10	80.72%	19.28%	0.01%	80.54%	19.46%	0.01%
	20	81.30%	18.70%	0.01%	80.43%	19.57%	0.01%
	30	81.26%	18.74%	0.01%	80.67%	19.33%	0.01%
	40	81.19%	18.81%	0.01%	80.40%	19.60%	0.01%
	50	81.50%	18.50%	0.01%	80.61%	19.39%	0.01%
TAXI Records (119, 319)	10	81.98%	18.02%	0.00%	81.92%	18.08%	0.00%
	20	81.96%	18.04%	0.00%	81.93%	18.07%	0.00%
	30	82.00%	18.00%	0.00%	82.12%	17.88%	0.00%
	40	82.01%	17.99%	0.00%	82.01%	17.99%	0.00%
	50	81.97%	18.03%	0.00%	82.10%	17.90%	0.00%
TLC Records(119, 319)	10	85.67%	14.33%	0.00%	85.62%	14.38%	0.00%
	20	85.67%	14.33%	0.00%	85.62%	14.38%	0.00%
	30	85.68%	14.32%	0.00%	85.63%	14.37%	0.00%
	40	85.69%	14.31%	0.00%	85.62%	14.38%	0.00%
	50	85.69%	14.31%	0.00%	85.61%	14.39%	0.00%

*Simba, STARK, GeoSpark results omitted for lack of support or exceeding 180 minutes of runtime

Table 11 Accuracy evaluation results - Varying the value of *k* test

	K	Exact match	Mis-match	Missing	Exact match	Mis-match	Missing
		With non-spatial data*			Without non-spatial data*		
		—○— SpPart_kNN VS —◇— LocationSpark					
BUS Records (119, 319)	3	91.19%	8.81%	0.01%	90.86%	9.14%	0.01%
	10	81.72%	18.28%	0.01%	80.60%	19.40%	0.01%
	50	30.00%	70.00%	0.01%	25.46%	74.54%	0.01%
	100	10.17%	89.83%	0.01%	5.64%	94.36%	0.01%
	500	0.01%	99.99%	0.01%	0%	100.00%	0.01%
	1000	0%	100.00%	0.01%	0%	100.00%	0.01%
TAXI Records (119, 319)	3	95.31%	4.69%	0%	95.29%	4.71%	0%
	10	81.95%	18.05%	0%	82.02%	17.98%	0%
	50	49.19%	50.81%	0%	49.06%	50.94%	0%
	100	38.64%	61.36%	0%	38.44%	61.56%	0%
	500	26.42%	73.58%	0%	25.98%	74.02%	0%
	1000	15.61%	84.39%	0%	14.31%	85.69%	0%
TLC Records(119, 319)	3	96.66%	3.34%	0%	96.67%	3.33%	0%
	10	85.67%	14.33%	0%	85.61%	14.39%	0%
	50	33.30%	66.70%	0%	33.00%	67.00%	0%
	100	17.09%	82.91%	0%	16.85%	83.15%	0%
	500	0.25%	99.75%	0%	0.18%	99.82%	0%
	1000	0.01%	99.99%	0%	0.00%	100.00%	0%

*Simba, STARK, GeoSpark results omitted for lack of support or exceeding 180 min of runtime

Algorithms

Algorithm 1 Building and Indexing the Partitioner

```
▷ An algorithm illustrating the $right$ object assignment to partitions. The grid assignment starts in
▷ parallel with the master node receiving occupied cells and their object counts. Each partition receives
▷ objects between the specified limits and added to the spatial index.
▷ The time complexity of the algorithm greatly depends on the $right$ distribution but has a complexity
▷ of $O(n)$, where $n$ is the number of $right$ points

function BUILDPARTITIONER($rightRDD, gridDim, countPointMin_{Right}, countPointMax_{Right}$)

    $partitionerSI \leftarrow SpatialIndex()$      ▷ Create a new Spatial Index
    $currPartNum, currPartSize \leftarrow 0$  ▷ Partition and number of points assigned counters.
    $mapMBR \leftarrow newMap()$               ▷ Maps partitions IDs (key) and their MBRs (value)
    $rightRDD$
        $.mapPartitions($           ▷ $rInt$ rounds to the nearest integer
            $\_.map(point \Rightarrow ((rInt(point.x/gridDim), rInt(point.y/gridDim)), 1L)))$
        $.reduceByKey(\_ + \_)$      ▷ Counts the number of points in each cell
        $.sortByKey()$              ▷ Sorts by the cells' $(X, Y)$ coordinates
        $.collect()$                ▷ Brings the summarized information to the master node
        $.map(xyCount \rightarrow \{$
                if($currPartNum == 0$ OR $currPartSize \geq countPointMin_{Right}$ OR
                        $currPartSize + xyCount.count > countPointMax_{Right}$) then

                    $currPartNum \leftarrow currPartNum + 1$
                    $currPartSize \leftarrow xyCount.count$
                    $mapMBR.add(newMBR(xyCount))$
                else

                    $currPartSize \leftarrow currPartSize + xyCount.count$
                    $mapMBR.last.updateMBR(xyCount)$
                endif

                $partitionerSI.insert(xyCount, currPartNum)$
            $\})$
    return partitionerSI, mapMBR
end function
```

Algorithm 2 Computing the Number of Processing Rounds

```
▷ An algorithm for illustrating the distributed computation of the number of rounds. Each object in
▷ the $left$ object computes its grid location and queries the spatial partitioner and finds the partitions
▷ needed to satisfy its $k$ matches. Finally, the master node receives a unique set of the partitions
▷ needed by all objects along with the size of the longest overall list.
▷ The time complexity is $O(n)$ where $n$ is the number of $left$ dataset objects on the partition

function COMPUTEROUNDS($leftRDD, gridDim, partitionerSI$)

    $setPartitionID \leftarrow Set()$     ▷ A unique list of needed partitions

    $numRounds \leftarrow 0$              ▷ The size of the longest list of partitions

  $(setPartitionID, maxPartition) =$
    $leftRDD$
        $.mapPartitions($
            $\_.map(point \rightarrow ((rInt(point.x/gridDim), rInt(point.y/gridDim))))$
                                 ▷ $rInt$ rounds to the nearest integer
        $.distinct()$                ▷ Avoids repetitive lookups from $partitionerSI$
        $.mapPartitions($
            $\_.map(gridCell \rightarrow setPartitions \leftarrow knn(partitionerSI, gridCell, k).toSet$
                                 ▷ Finds closest $k$ grid cells and returns their partition assignments
    $(setPartitions, setPartitions.size)$
  $))$
  $.treeReduce((x, y) \rightarrow (mergeSets(x._1, y._1), max(x._2, y._2)))$
                                 ▷ Merges the results on the master node
  return $setPartitionID, numRounds$
end function
```

Algorithm 3 kNN Lookup Within Spatial Index

```
▷ An algorithm for illustrating $k$NN lookup from a Quadtree index. First, the $gridCellXY$ finds the
▷ region that contains it had has at least $k$ points. Next, $gridCellXY$ builds a search area that
▷ encloses the best region and search all the child quadrants of the best region that intersect the
▷ search area. Add all points that fall within the search region. Once their size reaches $k$, shrink the
▷ search area. Once finished with the best region, repeat the search steps with the remainder of the
▷ Quadtree regions.
▷ The time complexity of the algorithm depends on dataset distribution, how it effects building the
▷ Quadtree, and the value of $k$. In the worst case, the time complexity is $O(n \times m)$, where $n$ is the
▷ number of points in the $left$ dataset, and $m$ is the number of points in the spatial index.

function KNN($partitionerSI, gridCellXY, k$)

    $bestLoc \leftarrow partitionerSI.bestLocation(gridCellXY, k)$
                                      ▷ Finds the best location in the spatial index with at least $k$ points

    $searchRegion \leftarrow circularRegion(gridCellXY, bestLoc)$

    for each cell $\in bestLoc$ do
        if $searchRegion.intersects(cell)$ then
            for each point $\in cell$ do
                if $searchRegion.contains(point)$ then
                    $searchRegion.addToList(cell)$
                    shrink region if list size reaches $k$
                end if
            end for
        end if
    end for

    for each cell $\in partitionerSI.root$ do
        if $cell \neq bestLoc$ AND $searchRegion.intersects(cell)$ then
            for each point $\in cell$ do
                if $searchRegion.contains(point)$ then
                    $searchRegion.addToList(cell)$
                    shrink region if list size reaches $k$
                end if
            end for
        end if
    end for
end function
```

Algorithm 4 Padding the List of Partitions to Visit

```
▷ An algorithm illustrating the padding of lists with sizes $< numRounds$. Each short list randomly
▷ receives partition assignments in random locations within the list without disrupting the existing
▷ order. Padding partitions are made negative to distinguish them from the original ones.
▷ The time complexity for this algorithm is constant ($O(numRounds)$). When applied by the
▷ $left$ objects, the complexity is $O(n)$.

$rand \leftarrow RandomNumberGenerator()$
$listParts \leftarrow lookup\_from\_partitioner()$
while $listParts.length < numRounds$ do
    $randIdx = rand.nextInt(numParts)$    ▷ numParts is the number of $right$ dataset partitions
    $listParts.insert(rand.nextInt(listParts.length + 1), -randIdx)$
            ▷ Negative IDs indicate a padding partition. Objects should not perform operations there
end while
```

Abbreviations

TLC: New York City Taxi and Limousine Commission; TPEP/LPEP: Taxicab and Livery Passenger Enhancement Programs.

Acknowledgements
Not applicable.

Author contributions

AZ and HTV: design, software, experiment, evaluation, analysis, writing. Both the authors read and approved the final manuscript.

Funding
This work was supported by Vingroup Innovation Award VINIF.2019.20.

Availability of data and materials

Source code and experiments: https://github.com/bdilab/Spark-knn. OpenStreetMap: https://wiki.openstreetmap.org/wiki/Downloading_data. OpenStreetMap, Researcher Information: https://wiki.openstreetmap.org/wiki/Researcher_Information. TLC Trip Records: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page. Bus Trip Recrods: http://web.mta.info/developers/MTA-Bus-Time-historical-data.html

Declarations

Ethics approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Author details
[1]Department of Computer Science, CUNY Graduate Center, New York, USA. [2]Department of Computer Science, CUNY City College, New York, USA.

Received: 19 November 2021 Accepted: 23 March 2022
Published online: 02 June 2022

References

1. Bernard Marr Fc. How much data do we create every day the mindblowing stats everyone should read. https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=356c32f960ba.
2. Rohit Kulkarni Fc. Big data goes big. https://www.forbes.com/sites/rkulkarni/2019/02/07/big-data-goes-big/?sh=7284031320d7
3. Tankovska HS. Number of social media users 2025 Statista. https://www.statista.com/statistics/278414/number-of-worldwide-social-network-users/
4. Kim G-H, Trimi S, Chung J-H. Big-data applications in the government sector. Commun ACM. 2014;57(3):78–85. https://doi.org/10.1145/2500873.
5. Zheng Y, Liu Y, Yuan J, Xie X. Urban computing with taxicabs. In: Proceedings of the 13th International Conference on Ubiquitous Computing, 2011; pp. 89–98. https://doi.org/10.1145/2030112.2030126.
6. Zhang D, Zhao J, Zhang F, He T. comobile: Real-time human mobility modeling at urban scale using multi-view learning. In: Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, 2015; pp. 1–10. https://doi.org/10.1145/2820783.2820821.
7. Yuan J, Zheng Y, Zhang C, Xie W, Xie X, Sun G, Huang Y. T-drive: driving directions based on taxi trajectories. In: Proceedings of the 18th SIGSPATIAL international conference on advances in geographic information systems, 2010; 99–108. https://doi.org/10.1145/1869790.1869807.
8. Huang Y, Powell JW. Detecting regions of disequilibrium in taxi services under uncertainty. In: Proceedings of the 20th International conference on advances in geographic information systems, 2012; pp. 139–148. https://doi.org/10.1145/2424321.2424340.
9. Markets and Markets. Geospatial solutions market worth \$502.6 Billion by 2024 - Exclusive report by markets and marketsTM. 2019. https://www.prnewswire.com/news-releases/geospatial-solutions-market-worth-502-6-billion-by-2024--exclusive- report-by-marketsandmarkets-300895569.html .
10. Shiftehfar R. Uber's big data platform: 100+ petabytes with minute latency. 2018. https://eng.uber.com/uber-big-data-platform/ .
11. Li B, Zhang D, Sun L, Chen C, Li S, Qi G, Yang Q. Hunting or waiting? discovering passenger-finding strategies from a large-scale real-world taxi dataset. In: 2011 IEEE International conference on pervasive computing and communications workshops (PERCOM Workshops), IEEE. 2011; pp. 63–68. https://doi.org/10.1109/PERCOMW.2011.5766967.
12. Hadoop A. Apache Hadoop. https://hadoop.apache.org/.
13. Foundation TAS. Apache spark unified analytics engine for big data. https://spark.apache.org/.

14. Yu J, Wu J, Sarwat M. 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, 2015; pp. 1–4. https://doi.org/10.1145/2820783.2820860.
15. Huang Z, Chen Y, Wan L, Peng X. Geospark sql: an effective framework enabling spatial queries on spark. ISPRS Int J Geo Inform. 2017;6(9):285. https://doi.org/10.3390/ijgi6090285.
16. Tang M, Yu Y, Malluhi QM, Ouzzani M, Aref WG. Locationspark: a distributed in-memory data management system for big spatial data. Proc VLDB Endow. 2016;9(13):1565–8. https://doi.org/10.14778/3007263.3007310.
17. Hagedorn S, Gotze P, Sattler K-U. The stark framework for spatio-temporal data analytics on spark. Datenbanksysteme für Business, Technologie und Web (BTW 2017). 2017.
18. Jacox EH, Samet H. Spatial join techniques. ACM Trans Database Syst. 2007;32(1):7. https://doi.org/10.1145/1206049.1206056.
19. Zeidan A, Lagerspetz E, Zhao K, Nurmi P, Tarkoma S, Vo HT. Geomatch: efficient large-scale map matching on apache spark. ACM Trans Data Sci. 2020;1(3):1–30. https://doi.org/10.1145/3402904.
20. Shekhar S, Lu C, Tan X, Chawla S, Vatsavai R. A visualization tool for spatial data warehouses. Geogr Data Min Knowl Dis. 2001;73:16–72.
21. Eldawy A, Mokbel MF, Jonathan C. Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In: 2016 IEEE 32nd International Conference on Data Engineering (ICDE). IEEE. 2016; pp. 601–612 . https://doi.org/10.1109/ICDE.2016.7498274.
22. Roussopoulos N, Kelley S, Vincent F. Nearest neighbor queries. In: Proceedings of the 1995 ACM SIGMOD International conference on management of data, 1995; pp. 71–79. https://doi.org/10.1145/223784.223794.
23. Hadoop A. Cluster mode overview - spark 2.4.0 Documentation.html. http://spark.apache.org/docs/2.4.0/cluster-overview.html. 2020.
24. Jelvix. Top 10 big data frameworks in 2021 | Jelvix. https://jelvix.com/blog/top-5-big-data-frameworks.
25. Forbes. Spark or hadoop – which is the best big data framework? https://www.forbes.com/sites/bernardmarr/2015/06/22/spark-or-hadoop-which-is-the-best-big-data-framework/?sh=55ab4da7127e
26. Microsoft. What is apache spark? | Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/spark/what-is-spark.
27. Scheuermann P, Weikum G, Zabback P. Data partitioning and load balancing in parallel disk systems. VLDB J. 1998;7(1):48–66. https://doi.org/10.1007/s007780050053.
28. Lee K, Liu L. Scaling queries over big rdf graphs with semantic hash partitioning. Proc VLDB Endow. 2013;6(14):1894–905. https://doi.org/10.14778/2556549.2556571.
29. Abadi DJ, Marcus A, Madden SR, Hollenbach K. Scalable semantic web data management using vertical partitioning. In: Proceedings of the 33rd international conference on very large data bases, 2007; pp. 411–422.
30. Vo H, Aji A, Wang F. SATO: a spatial data partitioning framework for scalable query processing. In: Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems. SIGSPATIAL '14, pp. 545–548. New York; ACM. https://doi.org/10.1145/2666310.2666365, 2014.
31. Aji A, Wang F, Vo H, Lee R, Liu Q, Zhang X, Saltz J. Hadoop gis: a high performance spatial data warehousing system over mapreduce. Proc VLDB Endow. 2013;6(11):1009–20.
32. Eldawy A. Spatialhadoop: towards flexible and scalable spatial processing using mapreduce. In: Proceedings of the 2014 SIGMOD PhD symposium, ACM 2014; pp. 46–50. https://doi.org/10.1145/2602622.2602625.
33. Magellan. GitHub—harsha2010/magellan: geo spatial data analytics on spark. https://github.com/harsha2010/magellan
34. He Y, Tan H, Luo W, Feng S, Fan J. Mr-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data. Front Comput Sci. 2014;8(1):83–99. https://doi.org/10.1007/s11704-013-3158-3.
35. Xie D, Li F, Yao B, Li G, Zhou L, Guo M. Simba: Efficient in-memory spatial analytics. In: Proceedings of the 2016 international conference on management of data, 2016; pp. 1071–1085. https://doi.org/10.1145/2882903.2915237.
36. Leutenegger ST, Lopez MA, Edgington J. Str: a simple and efficient algorithm for r-tree packing. In: Proceedings 13th international conference on data engineering. IEEE. 1997; pp. 497–506. https://doi.org/10.1109/ICDE.1997.582015.
37. Al Aghbari Z, Ismail T, Kamel I. Sparknn: a distributed in-memory data partitioning for knn queries on big spatial data. Data Sci J. 2020;19(1):00. https://doi.org/10.5334/dsj-2020-035.
38. Chatzigeorgakidis G, Karagiorgou S, Athanasiou S, Skiadopoulos S. Fml-knn: scalable machine learning on big data using k-nearest neighbor joins. J Big Data. 2018;5(1):1–27. https://doi.org/10.1186/s40537-018-0115-x.
39. Ben Brahim M, Drira W, Filali F, Hamdi N. Spatial data extension for cassandra nosql database. J Big Data. 2016;3(1):1–16. https://doi.org/10.1186/s40537-016-0045-4.
40. Costa E, Costa C, Santos MY. Evaluating partitioning and bucketing strategies for hive-based big data warehousing systems. J Big Data. 2019;6(1):1–38. https://doi.org/10.1186/s40537-019-0196-1.
41. Minasny B, McBratney AB, Walvoort DJ. The variance quadtree algorithm: use for spatial sampling design. Comput Geosci. 2007;33(3):383–92. https://doi.org/10.1016/j.cageo.2006.08.009.
42. Li Z, Lee KC, Zheng B, Lee W-C, Lee D, Wang X. Ir-tree: an efficient index for geographic document search. IEEE Trans knowl Data Eng. 2010;23(4):585–99. https://doi.org/10.1109/TKDE.2010.149.
43. Aragon CR, Seidel R. Randomized search trees In: FOCS. 1989;30:540–5. https://doi.org/10.1007/BF01940876.
44. NJordan72. harsha2010: GitHub—harsha2010/magellan: geo spatial data analytics on spark. https://github.com/harsha2010/magellan.
45. Li J, Xu L, Tang L, Wang S, Li L. Big data in tourism research: a literature review. Tour Manag. 2018;68:301–23. https://doi.org/10.1016/j.tourman.2018.03.009.
46. GeoJSON. GeoJSON. https://geojson.org/. (undefined 11/3/2021 23:28).
47. Zeidan A, Lagerspetz E, Zhao K, Nurmi P, Tarkoma S, Vo HT. Geomatch: Efficient large-scale map matching on apache spark. In: 2018 IEEE International Conference on Big Data (Big Data). IEEE. 2018; pp. 384–391. https://doi.org/10.1109/BigData.2018.8622488.
48. Chang H-w, Tai Y-c, Hsu JY-j. Context-aware taxi demand hotspots prediction. Int J Bus Intell Data Min. 2010;5(1):3–18. https://doi.org/10.1504/IJBIDM.2010.030296.

49. Zhang H, Chen G, Ooi BC, Tan K-L, Zhang M. In-memory big data management and processing: a survey. IEEE Tran Knowl Data Eng. 2015;27(7):1920–48. https://doi.org/10.1109/TKDE.2015.2427795.
50. Cahsai A, Ntarmos N, Anagnostopoulos C, Triantafillou P. Scaling k-nearest neighbours queries (the right way). In: 2017 IEEE 37th international conference on distributed computing systems (ICDCS). IEEE. 2017; pp. 1419–1430 . https://doi.org/10.1109/ICDCS.2017.267.
51. George L. HBase: the definitive guide: random access to your planet-size data. " O'Reilly Media, Inc.".
52. Spark A. [SPARK-6235] Address various 2G limits–ASF JIRA. https://issues.apache.org/jira/browse/SPARK-6235
53. Guttman A. R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International conference on management of data, 1984; pp. 47–57. https://doi.org/10.1145/602259.602266.
54. Beckmann N, Kriegel H, Schneider R, Seeger B. The r*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD international conference on management of data, 1990; pp. 322–331. https://doi.org/10.1145/93597.98741.
55. Bentley JL. Multidimensional binary search trees used for associative searching. Commun ACM. 1975;18(9):509–17. https://doi.org/10.1145/361002.361007.
56. Finkel RA, Bentley JL. Quad trees a data structure for retrieval on composite keys. Acta inform. 1974;4(1):1–9. https://doi.org/10.1007/BF00288933.
57. Samet H. An overview of quadtrees, octrees, and related hierarchical data structures. Theoretical foundations of computer graphics and CAD, 1988; 51–68. https://doi.org/10.1007/978-3-642-83539-1_2.
58. Rigaux P, Scholl M, Voisard A. Spatial databases: with application to GIS. Elsevier.
59. Spark A. Tuning—spark 2.4.0 documentation. https://spark.apache.org/docs/2.4.0/tuning.html#memory-management-overview.
60. Foundation TAS. configuration—spark 2.4.5 documentation. https://spark.apache.org/docs/latest/configuration.html
61. Foundation TAS. Spark 2.4.5 JavaDoc. https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/util/SizeEstimator.html.
62. Pandey V, Kipf A, Neumann T, Kemper A. How good are modern spatial analytics systems? Proc VLDB Endow. 2018;11(11):1661–73. https://doi.org/10.14778/3236187.3236213.
63. OpenStreetMap: researcher information—OpenStreetMap Wiki 2022. https://wiki.openstreetmap.org/wiki/Researcher_Information.
64. MTA. MTA Bus Time® Historical data 2022. http://web.mta.info/developers/MTA-Bus-Time-historical-data.html.
65. MTA. TLC trip record data—TLC 2022. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.
66. Kumar M. World geodetic system 1984: a modern and accurate global reference frame. Mar Geod. 1988;12(2):117–26. https://doi.org/10.1080/15210608809379580.
67. U.S. Department of Commerce, N.O., Administration, A.: North American Datum of 1983 (NAD 83) - Horizontal and Geometric Datums - Datums - National Geodetic Survey (2022). https://geodesy.noaa.gov/datums/horizontal/north-american-datum-1983.shtml.
68. Wiki O. Points of interest—OpenStreetMap Wiki. https://wiki.openstreetmap.org/wiki/Points_of_interest.
69. OpenStreetMap contributors: OpenStreetMap. 2017. https://www.openstreetmap.org.
70. LocationTech: LocationTech JTS Topology Suite | projects.eclipse.org. https://projects.eclipse.org/projects/locationtech.jts.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.