How Good Are Modern Spatial Analytics Systems?

Varun Pandey
Technical University of Munich
pandey@in.tum.de

Andreas Kipf
Technical University of Munich
kipf@in.tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

ABSTRACT

Spatial data is pervasive. Large amount of spatial data is produced every day from GPS-enabled devices such as cell phones, cars, sensors, and various consumer based applications such as Uber, location-tagged posts in Facebook, Instagram, Snapchat, etc. This growth in spatial data coupled with the fact that spatial queries, analytical or transactional, can be computationally extensive has attracted enormous interest from the research community to develop systems that can efficiently process and analyze this data. In recent years a lot of spatial analytics systems have emerged. Existing work compares either limited features of these systems or the studies are outdated since new systems have emerged. In this work, we first explore the available modern spatial processing systems and then thoroughly compare them based on features and queries they support, using real-world datasets.

PVLDB Reference Format:
Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. How Good Are Modern Spatial Analytics Systems?. *PVLDB*, 11 (11): 1661-1673, 2018.
DOI: https://doi.org/10.14778/3236187.3236213

1. INTRODUCTION

There has been an explosion in the amount of spatial data being generated at the moment. It comes from the web, billions of phones, sensors, cars, satellites, and a huge array of various other sources. For example, NASA [16] provides climate projections since 1950 until 2100 for conducting studies of climate change impact. The dataset is approximately 17 TB in size. Gartner has also forecasted that there will be more than *20 billion* connected devices in 2020[1] which will lead to even more spatial data being generated. Location-based services are also on the rise. These services generate a large amount of location data on a daily basis. Foursquare, a popular cell phone application, has over 12 billion check-ins till date and has more than 105 million venues mapped around the world [7]. Uber, a Transportation Network Company (TNC), recently reported completing 5 billion rides [25] till date, more than doubling the reported 2 billion rides completed last year. Lyft, another TNC, now serves 1 million rides a day [13]. Twitter, a popular social media giant, generates approximately 10 million [23] geo-tagged tweets every day. The value of the data is ultimately tied to its use. Analyzing such large amount of spatial data can lead to variety of insights and better services as well as new products. An analysis [19] of NYC Taxi dataset and Uber barely scratches the surface of the information that is available in such datasets, which could lead to better business decisions. Analysis of geotagged tweets helps in predicting various trends, such as eating habits of people [24], and effects of temperature on happiness [2] to predicting the people who may need help during disasters [20]. Uber uses its data to provide better services by predicting price surges and making city transportation more efficient [27]. Moreover, these TNCs are now providing aggregated information about the trips taken in various cities to the Transportation Authorities of respective cities for analysis [26] [22]. Foursquare, which started as a check-in company has evolved its product Swarm to be a personal data collector of all the places a user has been to and has also launched Location Intelligence for brands to locate and message consumers.

forecast

The era of big spatial data has lead the research community to focus on developing systems that can efficiently analyze and process spatial data. Systems to manage and analyze big data have existed for a long time (Hadoop [8], Impala [9], Spark [35], however, spatial support in these systems had not existed. This lead to various Hadoop based spatial systems being developed (HadoopGIS [1], SpatialHadoop [5]). Similarly, there have been plenty of spatial processing and analytics systems that have been developed for Spark (SpatialSpark [31], GeoSpark [33], Simba [30], LocationSpark [21], and Magellan [14]). Spatial extensions for databases, have seen a similar trend with Oracle Spatial [17], MemSQL [15], Cassandra [3], and HyPer [18]. The general approach of building such systems is *on-top*, *from-scratch* and *built-in* and has been well documented in [6].

In this *Experiment and Analysis* paper we present:

- A brief survey of available modern spatial analytics systems, including two new systems that have not been covered in literature previously
- A thorough performance evaluation of the available systems using a real world dataset, focusing on major features that are supported by the systems

[1]http://www.gartner.com/newsroom/id/3165317

Proceedings of the VLDB Endowment, Vol. 11, No. 11

DOI: https://doi.org/10.14778/3236187.3236213

The remainder of this paper is structured as follows: Section 2 gives the motivation to carry out this study. Section 3 presents the spatial queries domain explaining which queries we consider for this study. Section 4 summarizes a broad variety of existing big spatial data analytics systems highlighting salient features of each system. Section 5 gives the details about the experimental setup and datasets used for evaluation of the spatial analytics systems. Section 6 gives the details about the performance evaluation of the systems which is followed by the conclusions in Section 7.

2. MOTIVATION

The aim of our study is to compare five Spark-based systems namely, SpatialSpark, GeoSpark, Simba, Magellan and LocationSpark, using five different spatial queries (range query, kNN query, spatial joins between various geometric datatypes, distance join, and kNN join) and four different datatypes (*points*, *linestrings*, *rectangles*, and *polygons*). Although we include SpatialHadoop and HadoopGIS in the brief survey of modern big data spatial analytics systems, we decided to omit them from evaluation. We only consider spatial analytics systems based on Spark for evaluation since Hadoop based systems like SpatialHadoop and HadoopGIS have consistently been shown to perform poorly compared to Spark based systems in existing work.

There have been multiple studies that compare these systems based on various queries and performance metrics but all of them are incomplete or only compare a limited features of the systems. SpatialSpark [31] implements two join algorithms, **point-in-polygon** and **point-to-polyline** distance join, and evaluates the two implementations. In the extended study [32], **point-in-polygon** and **polyline-with-polyline** intersection join performance is evaluated for Hadoop-GIS, SpatialHadoop, and SpatialSpark. In [33], GeoSpark compares itself with SpatialHadoop for intersection based join between **linestring-polygon** and kNN query performance. In [21] LocationSpark compares the kNN join performance against the state-of-the-art kNN join algorithms. Simba [30] evaluates itself with a variety of systems including Hadoop-GIS, SpatialHadoop, SpatialSpark, GeoSpark, and the state-of-the-art kNN join algorithms only on the point data type. Both Simba and LocationSpark support kNN joins but they have not been evaluated against each other. Simba does not support linestring and polygon datatypes yet. The join and range query performance comparison for these geometric objects are missing. Also, Simba only considers a small window of **selection ratio** for range queries, and only compares itself with SparkSQL variant for these windows. Moreover, all the performance comparison in the aforementioned studies were done using a **large cluster**, and a **scalability** study of these systems is missing.

Meanwhile, some of these systems have been actively developed and many optimizations have been added. Since the previous studies, GeoSpark has introduced many new datatypes and has also added a query optimizer. Also, Magellan [14] has gathered attention in the Free And Open Source Software for Geospatial[2] (FOSS4G) committee and has not been evaluated in any existing study.

To summarize, these are some open ended questions missing in the existing literature:

- How do the modern *in-memory* spatial analytics systems perform for all the **major** features that they **support**?
- How do these systems perform for **all** possible spatial join combinations of various geometric data types?
- **Where** is the time actually spent during various join queries?
- How well do these systems perform for **different** selection ratios for range queries for **different** geometric objects?
- What are the **memory costs** related to the systems?
- Do the memory costs have any **impact** on query performance?
- How well do these systems **scale** for the queries that they support?

We aim to fill this gap and compare the modern in-memory spatial analytics systems to present a complete study, while the experiment files and setup provided will make it easier for researchers to benchmark these systems against future spatial analytics systems or spatial algorithms.

3. QUERIES

For the queries, we consider four geometric features or datatypes: *points*, *linestrings*, *rectangles* and *polygons* subsets (or all) of which are supported in most of the evaluated systems.

The queries considered for evaluation are: single relation operations (range query, kNN query) and join operations (distance join, spatial joins and kNN join). There can be other spatial queries such as computational geometry operations, spatial data mining operations, and raster operations. These queries are well-defined in [6]. We do not consider these queries since the evaluated systems do not support these queries and evaluating systems that do is out of the scope of this paper. We will now briefly describe the set of queries that we consider for evaluation.

3.1 Range Query

A range query takes a range R and a set of geometric objects S, and returns all objects in S that lie in the range R. Formally,

$$Range(R, S) = \{\ s | s \in S,\ s \in R\ \}.$$

3.2 k Nearest Neighbors Query

A kNN query takes a set of points R, a query point q, and an integer k ≥ 1 as input, and finds the k nearest points in R to q. Formally,

$$kNN(R, q) = \{T \subseteq R, |T| = k \wedge \forall t \in T,$$
$$r \in R - T : d(q, t) \leq d(q, r)\}.$$

3.3 Spatial Join

A spatial join takes two input sets of spatial records R and S and a join predicate θ (e.g., overlap, intersect, contains,

[2]http://foss4g.org/

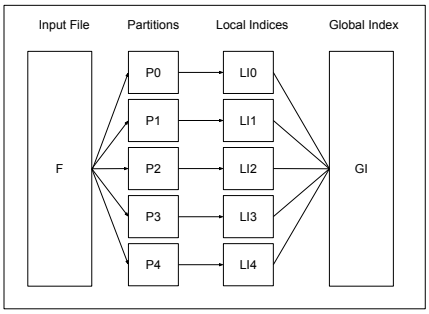


Figure 1: A generalized indexing scheme for distributed spatial analytics systems.

within, withindistance) and returns a set of all pairs (r,s) where r ∈ R, s ∈ S, and θ is true for (r,s). Formally,

$$R \bowtie_\theta S = \{ (r,s) \mid r \in R,\ s \in S,\ \theta(r,s)\ is\ true \}.$$

A distance join is a special case of spatial join where the join predicate is *withindistance*. For the sake of clarity, we will refer to distance join as is and do not include it in spatial joins.

3.4 k Nearest Neighbors Join

A kNN join takes two input sets of spatial records R and S and an integer k $\geq$ 1 and returns for all objects r ∈ R theirs k closest neighbors in S. Formally,

$$R \bowtie_{kNN} S = \{ (r,s) \mid r \in R,\ s \in kNN(S,r) \}.$$

4. SPATIAL ANALYTICS SYSTEMS

In this section, we briefly review the cluster-based systems that support spatial data management, queries and analytics over distributed data using a cluster of commodity machines. We study the various features, data partitioning and indexing schemes, and queries that are supported in these systems. Table 1 gives an overview of the features of the different spatial analytics systems.

An important point to make here is that distributed systems, generally, use a two level indexing scheme consisting of a global index in the master node and multiple local indices in the slave nodes. Figure 1 shows the generalized indexing scheme. The input file is first partitioned based on a partitioning scheme, each partition is then indexed using a specialized spatial index (e.g., R-tree, R+-tree, Quadtree etc.), and finally these local indices are indexed in a global index on the master node. This is also known as the preprocessing phase, wherein the data is loaded from the distributed file system, and is partitioned logically or physically which is useful for query processing. The quality and performance of partitioning techniques have been thoroughly covered in [4].

4.1 Hadoop-GIS

Hadoop-GIS [1] is a scalable and high-performance spatial data warehousing system for running large scale spatial queries on Hadoop. It was the first system based on Hadoop to support spatial queries. Hadoop-GIS treats Hadoop as a black box and relies on underlying architecture for processing. For partitioning, Hadoop-GIS uses a *uniform grid* to partition the space first and then map the objects to the tiles. If partitioning creates some high density tiles, these tiles are broken down into smaller tiles to handle this data skew. In [29], Hadoop-GIS added more partitioning techniques to provide flexibility to the system. Here, the input data is partitioned in four steps: Sample, Analyze, Tear and Optimize (SATO). 1-3% of the data is sampled and the density distribution of the dataset is computed. The Minimum Bounding Rectangles (MBR) from the sampled dataset are fed to the Analyzer which decides the optimum global partitioning scheme for the global partitions. In the Tear phase each global partition is further partitioned to create local partitions. The physical partitioning takes place in this step. In the Optimize phase the data is re-scanned and statistics about the partitions are collected to build the multi-level index. This is an example of dynamic partitioning and indexing, which takes into consideration the distribution and skew of spatial data. These indices are used to process the queries supported: range and spatial join queries. Hadoop-GIS supports points, rectangles, and polygons. Hadoop-GIS extends HiveQL with spatial query support and integrates the spatial query engine into Hive.

4.2 SpatialHadoop

SpatialHadoop [5] is a full-fledged MapReduce framework with native support for spatial data. It enriches Hadoop with spatial constructs and awareness of spatial data inside the core functionality of Hadoop and is thus able to obtain better performance than Hadoop-GIS since it has to deal with no layer overhead. SpatialHadoop partitions the dataset into n partitions that confirm to three conditions (i) each partition should fit one HDFS block (64MB), (ii) the objects close to each other in space should be assigned to same partition and, (iii) all partitions should be of similar size for load balancing purposes. The input dataset can be partitioned and indexed using either Grid Index, R-tree or R+-tree. Since, R-tree performs the best in most cases as reported in the publication, we will describe the partitioning phase using R-tree. SpatialHadoop bulk loads a sample from the input dataset into an in-memory R-tree using the Sort-Tile-Recursive (STR) algorithm. It computes the number of partitions, n, based on the size of the input file. It then fills the R-tree with degree d ($\sqrt{n}$) using the STR algorithm. The STR algorithm ensures that the tree is balanced and the degree d of the tree ensures that there are at least n nodes in the second level of the tree. The second level of the tree is used to physically partition the input dataset. In the physical partitioning step, each input record is assigned to a partition which requires the least enlargement to cover the record. After physical partitioning, each partition is bulk loaded into an R-tree using the STR algorithm and dumped to a file. The block in local index file is annotated with the MBR of its content. In the global indexing phase, all local indexed files are concatenated and the global index is created by bulk loading all the blocks into an R-tree using the annotated MBR as the index key. SpatialHadoop extends

Table 1: Overview of features in spatial analytics systems

	Hadoop-GIS	SpatialHadoop	SpatialSpark	GeoSpark	Magellan	SIMBA	LocationSpark
In-Memory Processing	No	No	Yes	Yes	Yes	Yes	Yes
Language	HiveSP	Pigeon	N.A.	N.A.	Extended SparkSQL	Extended SparkSQL	N.A.
Partitioning Techniques	SATO Framework (Multiple partitioning techniques)	Quad, STR, STR+, K-d, Hilbert, Z-curve	Uniform, Binary-Split, STR	Quad, KDB, R-tree, Voronoi, Uniform, Hilbert	Z-curve	STR	Uniform, R-tree, Quad
Index	R*-tree	R-tree	R-tree	R-tree, Quadtree	None	R-tree	R-tree, Quadtree, IR tree
Datatypes	Point, Rectangle, Polygon	Point, Rectangle, Polygon	Point, LineString, Rectangle, Polygon	Point, Rectangle, Polygon, LineString	Point, LineString, Polygon, MultiPoint, MultiPolygon	Point	Point, Rectangle
Queries	Range, Spatial Joins	Range, kNN, Spatial Joins	Range, Spatial Joins	Range, kNN, Spatial Joins, Distance Join	Range, Spatial Joins	Range, kNN, Distance Join, kNN Join	Range, kNN, Spatial Join, Distance Join, kNN Join

FileSplitter and *RecordReader* in Hadoop to support spatial records. *SpatialFileSplitter* uses the global index to prune out blocks that do not contribute to the query result. *SpatialRecordReader* exploits the local index in the partitions received from *SpatialFileSplitter* to efficiently process the query. It also extends Pig Latin, called Pigeon, with spatial support. SpatialHadoop supports range queries, kNN queries, and spatial joins. It has support for point, rectangle, and polygon datatypes.

4.3 SpatialSpark

SpatialSpark [31] is a lightweight implementation of spatial support in Apache Spark. It targets in-memory processing for higher performance. SpatialSpark supports multiple geometric objects including points, linestrings, polylines, rectangles, and polygons. It supports multiple spatial partitioning schemes fixed grid, binary split and STR partitioning. For indexing, SpatialSpark uses an R-tree. SpatialSpark offers a variety of operations on spatial datasets including range queries on all types of geometric objects, spatial joins between various geometric objects and distance joins. It supports 1NN queries but does not support kNN queries and kNN joins.

4.4 GeoSpark

GeoSpark [33] is an in-memory cluster computing framework based on Apache Spark for processing large spatial data. It consists of three layers: (i) Apache Spark Layer, (ii) Spatial RDD Layer, and (iii) Spatial Query Processing Layer. GeoSpark extends the core of Apache Spark to support spatial datatypes, indexes, and operations. GeoSpark extends the resilient distributed datasets (RDDs) to support spatial datatypes. Apache Spark Layer is responsible for native functions that are supported by Spark such as load/save data to persistent storage. Spatial RDD layer extends Spark with spatial RDDs (SRDDs) that can efficiently partition SRDD elements across machines and also introduces parallelized spatial transformations. GeoSpark introduces support for various types of spatial objects: points, linestrings, rectangles, and polygons. It also provides a Geometrical Operations library which has geometrical operations such as *Ovelap()* (find overlapping objects), *MinimumBoundingRectangle()* which returns the MBR of either every object in the SRDD or largest MBR encompassing every object in the SRDD, *Union()* which returns the union of all polygons in the SRDD. GeoSpark also comes with a query optimizer. GeoSpark supports multiple partitioning schemes including, Quadtree, KDB tree, R-tree, Voronoi, fixed grid, and Hilbert partitioning. GeoSpark has two indexes available, R-tree and Quadtree. GeoSpark has support for range queries, spatial join queries, and kNN queries. GeoSpark does not support kNN joins.

4.5 Magellan

Magellan [14] is a distributed execution engine for spatial analytics on big data. It leverages modern database techniques in Apache Spark like efficient data layout, code generation, and query optimization in order to optimize spatial queries. Magellan extends SparkSQL to accommodate spatial datatypes, geometric predicates, and queries. Magellan has support for points, linestrings, rectangles, polygons, multipoints, and multipolygons. Magellan supports range queries and spatial joins but does not support kNN queries, distance joins, and kNN joins. Magellan also adds geometric predicates such as intersects, within, and contains. Magellan uses on the fly indexing of the geometrics objects but can also leverage the indices if they were persisted earlier. Magellan uses Z-order curve for indexing and appends a column to the dataset with the Z curve values. To perform join

queries efficiently, Magellan intercepts the query plan and overwrites it to use Z-curve index. It uses an inner join on the Z-curve and a predicate filter on top of the inner join, instead of a cross join between two input datasets. In this way, the Z-curve actually ends up acting as a spatial partitioning technique in Magellan.

4.6 SIMBA

Simba [30] (Spatial In-Memory Big Data Analytics) is a distributed in-memory analytics engine that supports spatial queries and analytics over big spatial data in Spark [3]. Simba extends Spark SQL to support spatial operations. Simba also adds spatial indices in RDDs and SQL context, which helps in reducing query latencies and increasing analytical throughput by executing queries in parallel. It also introduces logical and physical optimizers to select better query plans. Tables are represented as RDDs of records of the table, thus indexing records of the table means indexing elements of the RDDs. To partition the data, Simba uses a similar strategy as SpatialHadoop where an R-tree is constructed by sampling the input dataset and filled using the STR algorithm to get the first level of the tree that represents the partition boundaries. These boundaries are only the MBR of the sampled set, which are extended as each record is added to the partition. Simba provides flexibility to the user to specify its own partitioning scheme as well, since the *Partitioner* abstract class in Spark allows users to specify their own partitioning strategy. For indexing within an IndexRDD, Simba uses an R-tree by default. Finally, a global index is constructed by using the partition boundaries from the partitioner and statistics from the local index. Simba supports a variety of queries which include, range (rectangle and circle) queries, kNN queries (on points), distance joins (between points), and kNN joins (between points). Simba does not have support for spatial joins.

4.7 LocationSpark

LocationSpark [21] is a spatial data processing system based on Apache Spark. It provides a wide range of spatial features. It supports a rich set of spatial queries: range queries, kNN queries, spatial joins and kNN joins. LocationSpark introduces a spatial RDD layer named *LocationRDD* which can be cached in memory. LocationSpark has a query scheduler component which can detect if there is a query skew, by actively collecting statistical information from each partition. If it detects a hotspot partition for a query, a cost model evaluates the overhead of repartitioning and takes suitable action. For data partitioning, similar to other systems, LocationSpark samples the input dataset and partitions data accordingly. A user has the flexibility to choose between either a uniform grid or Quadtree as the partitioning scheme. It also provides flexibility for local indices. A user can choose between Fixed-Grid, R-tree, or a Quadtree for indexing the data locally within a partition. Furthermore, LocationSpark also has a spatial bloom filter termed *sFilter* embedded with the global and local indices to prune out more partitions for a query, which helps in reducing network communication costs. LocationSpark supports two geometric datatypes: points and rectangles. LocationSpark supports range queries (on points), kNN queries (on points), spatial joins between points and rectangles, and kNN joins (between points).

[3]Note: The latest stable Simba release is the standalone package outside of Spark (i.e. a library running on top of Spark) and we benchmark it and not the version in the original publication which is built inside Spark core

Table 2: Evaluated systems, their compatible Spark version, and defaults for the experiments

System	Version	Amazon EMR and Spark Version	Spatial Partitioning	Index
SpatialSpark	1.0	emr-5.9.0 (2.2.0)	STR	R-tree
GeoSpark	v1.1.3	emr-5.9.0 (2.2.0)	Quadtree	R-tree
Magellan	v1.0.5	emr-5.9.0 (2.2.0)	Z-curve	Z-curve
LocationSpark	the first version	emr-4.9.3 (1.6.3)	Quadtree	Quadtree
Simba	Standalone package compatible with Spark 2.1.x	emr-5.7.0 (2.1.2)	STR	R-tree

5. EXPERIMENTAL SETUP

5.1 Cluster Setup And Tuning Spark

To evaluate the systems we deploy variable sized clusters on Amazon AWS. We make use of the Amazon Elastic Map Reduce (EMR) framework to deploy the Spark cluster. The master node which runs the YARN resource manager for the cluster is an EC2 instance of type `m4.xlarge` that has 8 vCPUs and 16 GB main memory. For slave nodes we make use of `r4.8xlarge` EC2 instances which have 32 vCPUs and 244 GB main memory. We also attach 100 GB general purpose SSDs to each slave node. We deploy 1, 2, 4, 8, and 16 slaves nodes in the cluster to evaluate the systems and their scalability. We will only count the slave nodes in the cluster since the master node only runs the resource manager and is in no way responsible for any computation for the applications. We deploy the Spark applications in *cluster-mode* where the Spark *driver* is deployed on one of the slave nodes as *Application Master*. Since not all systems were compatible with latest Spark release we deployed clusters in different EMR versions. Table 2 shows the different systems evaluated, their compatible Spark versions and the default values we used for the experiments. The default number of partitions in every system has been set to 1024 for every query unless stated otherwise.

Amazon EMR cluster model has a master node and slave nodes. The master node runs the resource manager, by default YARN, which manages the cluster resources. The Spark applications are deployed on the slave nodes. The Spark execution model has two main components, the *driver* and the *executors*. The *driver* breaks up the work into tasks and assigns them to the *executors*. By default, Amazon EMR launches the cluster with *maximizeResourceAllocation*, which means that if there are four slave nodes in the cluster, then one node is selected as the *driver* and the other three as *executors*. This means that 25% of the cluster resources are dedicated to the bookkeeping tasks that the driver performs and only 75% of the resources are available for processing data. Moreover, having only three *executors* with all cores per node assigned to these executors in the cluster is not the optimal setting and often leads to poor HDFS through-

Table 3: Details of the datasets used for evaluation

Dataset	Geometry type	Number of records	Raw file size (GBs)	Total number of co-ordinates
OSM Nodes	Point	200 million	5.4	200 million
OSM Roads	LineString	70 million	18	803 million
OSM Buildings	Polygon	114 million	19	764 million
OSM Rectangles	Rectangle	114 million	14	573 million

put and failed Spark jobs[4]. We tuned every cluster in our experiments to utilize resources optimally by following the guidelines in the Cloudera Engineering Blog[5].

5.2 Datasets

To evaluate the systems we make use of the Open Street Maps (OSM) dataset made available by [5]. The OSM dataset comprises of `All Nodes` (*Points*), `Roads` (*LineStrings*), and `Buildings` (*Polygons*) datasets. The full OSM dataset contains *2.3 billions* points on earth (`All Nodes`), *70 million* roads and streets around the world (`Roads`), and *114 million* buildings (`Buildings`). We sampled a subset of *200 million* points from `All Nodes` dataset. We sampled the points from the dataset because some join results can be arbitrarily large with the full dataset and will not fit entirely in *driver's* memory. To extract the subset we make use of the `shuf` command in Linux. In addition to these datasets, we also generated a *Rectangle* dataset which is generated from the `Buildings` dataset by computing the minimum bounding rectangles of the polygons. We also needed to clean the datasets since some of the geometric objects did not comply with the OGC standard for spatial objects. To clean the datasets we used *Java Topology Suite* (JTS[6]) library which is OGC compliant. It is important to point out here is that certain systems only expect geometries in a particular format (such as Well-Known Text) as input, so we had to pre-process files in order to make them suitable as inputs for the different systems. In some cases the files sizes reduced because we had to strip the metadata from the files. Table 3 shows the details of the datasets used for evaluation. The datasets used for evaluation are available on our server[7] and all experiment files are available on GitHub[8]. We will refer to the datasets as Point, LineString, Rectangle, and Polygon datasets from now on.

We also ran the experiments with the US Census TIGER dataset provided by [5]. We used the LineString dataset from the TIGER dataset which contains approximately *70 million* linestrings in US. There are other datasets in TIGER but they are limited in size (less than 2 million spatial objects). To have a larger dataset to join with, we generated a rectangle dataset by computing bounding boxes of these linestrings. We also extracted *170 million* points that are in the US region from the OSM dataset to join with these datasets. Due to space constraint, we do not include the results of the queries on these datasets here, but they are available on the aforementioned GitHub page. The results are similar to the OSM dataset.

[4]https://databricks.com/session/top-5-mistakes-when-writing-spark-applications
[5]http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/
[6]https://github.com/locationtech/jts
[7]http://osm.db.in.tum.de/
[8]https://github.com/varpande/spatialanalytics

5.3 Spark Memory Management Model and Caching RDDs

Spark's memory management model is split into two parts, storage memory and execution memory. The amount of memory assigned to Spark after reserving memory for internal data structures is split into execution and storage memory. Execution memory refers to the memory that is assigned for computations such as joins, aggregations, shuffles, and sorts. Storage memory is the amount of memory that is used for caching the user datasets in memory. The total assigned memory is split 50/50 between storage and execution memory and is managed by *spark.memory.storageFraction* parameter. If no execution memory is needed, storage can acquire all the memory and vice versa. Execution memory can evict (spill to disk) storage blocks in case more execution memory is needed, and execution memory is immune to eviction.

Spark allows the user to cache (or persist) the RDDs in memory if they are used multiple times for computation. If sufficient storage memory is available in the cluster it is advisable to cache such RDDs. One purpose of choosing the AWS instance `r4.8xlarge` is that it comes with large memory so the RDDs can be cached. Even if RDDs do not fit in the memory in deserialized form, they can be serialized and persisted in memory. Most of the evaluated systems come with a custom serializer for the spatial RDDs which is based on Spark's KryoSerializer. Caching such RDDs is system specific and differs quite a bit because of different design choices. GeoSpark has an abstract SpatialRDD layer. It consists of three RDDs: *RawSpatialRDD*, *SpatialPartitionedRDD*, and *IndexedRDD*. When a SpatialRDD is initialized (e.g., new PointRDD()), the *RawSpatialRDD* gets populated. *SpatialPartitionedRDD* can be initialized by specifying the spatial partitioning technique, and then calling some action on the RDD. Initially for every type of query we keep the *RawSpatialRDD* in `MEMORY_ONLY` persistence level. Once the *SpatialPartitionedRDD* is generated, we unpersist the *RawSpatialRDD* as it is not needed in any query operation and keeping it in memory even in serialized form just incurs extra memory cost. We then populate the *IndexedRDD* and keep it along with the *SpatialPartitionedRDD* in memory at all times for all query operations. In Magellan, we make use of the dataframe API. We only persist a dataframe that contains the spatial object and the index for the object. LocationSpark has a *LocationRDD* and *QueryRDD* abstraction layer. We only cache these *LocationRDD* and *QueryRDD* for all query operations. In Simba, we make use of the dataframe API. For single relation operations we cache a dataframe with an index. For join operations we do not build an index on the dataframe, since Simba spatially partitions and indexes the datasets on the fly inside the join algorithms and does not utilize the persisted index. Spatially partitioning the data and building index on the dataframe is an overhead in case of join operations. In SpatialSpark, we cache the spatial RDDs which have a unique ID for every spatial element.

5.4 Performance Metrics

To measure and compare performance for single relation queries, we submit a batch of 100 queries and compute the

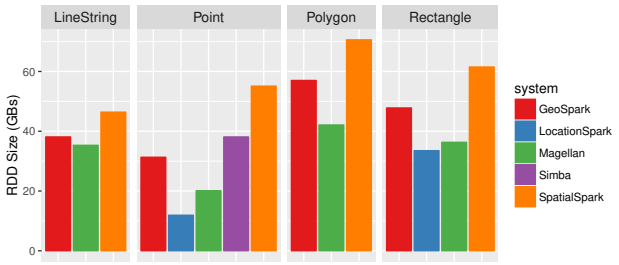


Figure 2: Memory footprint for various datasets

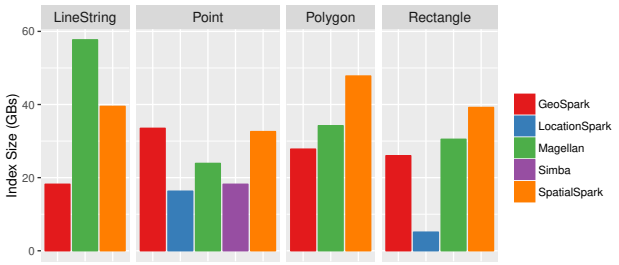


Figure 3: Indexing costs

throughput of the systems in queries/minute. To measure and compare the performance of different systems for join queries, we make use of the following six performance metrics:

- Preparation time: the *preparation time* is the total time spent by the system in reading the two datasets from HDFS, partitioning the input datasets, and indexing the partitions.
- Join time: the *join time* is the total amount of time spent by the system to complete the join query. This metric is a useful indication for use cases where the datasets are already indexed and the join queries need to run multiple times.
- Total runtime: the *total runtime* is the total of the two aforementioned performance metrics. It is a useful indication of end-to-end performance of the query.
- Shuffle write costs: the *shuffle write* is the sum of all written serialized data on all executors before transmitting to other executors at the end of a stage.
- Shuffle read costs: the *shuffle read* is the amount of read serialized data on all executors at the beginning of a stage in query execution.
- Peak Execution Memory: the *peak execution memory* is the maximum amount of memory used at any point in time for execution of a query.

In addition to the performance metrics, we also report the index sizes for the different datasets.

6. EVALUATION

6.1 Memory Costs

In this section, we report the in-memory consumption of the various data structures by caching the respective data structures and observing the Storage tab in the Spark Web UI[9]. Note that memory consumption for RDDs in Spark cannot be obtained programmatically, as it can only report approximate memory consumption of RDDs, hence these values are not available in the Scala codes for the systems in the GitHub page.

Figure 2 shows the raw spatial RDD sizes for various datasets for the systems. It is normal for Java objects to consume more memory than raw file size on disk[10]. We see most of the systems consume almost 3x more memory for every dataset. Also spatial partitioned RDDs add additional overhead because most of them use a replication-based technique to handle boundary objects. As mentioned before, the common technique for these systems to generate partitions is to sample the dataset first and decide spatial boundaries for the partitions based on this sampled data. When the spatial objects are loaded from the file system, they are mapped to these partitions. When a spatial object overlaps with multiple partitions, it is replicated to these multiple partitions, which increases the memory cost. Another point to make here is that both GeoSpark and SpatialSpark store JTS Geometry objects in the raw spatial RDD. The difference in their memory consumption is because SpatialSpark also adds a unique ID to each element in the RDD, which accounts for a slightly higher memory usage.

Figure 3 shows the index sizes for various systems for the different datasets. Simba and LocationSpark have the lowest memory consumption for indices for the Point dataset. LocationSpark only keeps the point coordinates and its two MBR coordinates in the Quadtree, and thus the indexing cost is low. Simba serializes its index (default persistence level is *MEM_AND_DISK_SER*) and thus the index cost is very low. An unusual case is Magellan's LineString index which consumes close to 60 GB of main memory compared to its indices for other datasets. The reason is that Magellan generates Z-curve to approximate the shapes. For Points, it has to generate one cell value for each coordinate. Polygons and Rectangles can be approximated using large cells and hence the cell counts for these geometric objects is low. For LineStrings, Magellan ends up generating cells for each coordinate in each linestring record in the dataset. There are a total of 803 million coordinates in the LineString dataset and hence Magellan ends up generating the same amount of cells for the LineString dataset.

6.2 Range Query Performance

To evaluate range queries, we varied the selection ratio (σ) to cover a wide range for selection. We generated six ranges for each dataset that cover six selection ratios for each dataset. In this experiment, we loaded and indexed the datasets in every system and do not include the costs to prepare them. We submit a batch of 100 queries for each range for each type of dataset and evaluate the query throughput in queries/minute.

Figure 4 shows the range query performance for the different systems on a single node varying the selection ratio (σ) from 0.0001 to 100. Magellan does not have any optimization for range queries and ends up scanning all partitions for all selection ratios for all datasets. It serves as a baseline for other systems. For the point dataset, LocationSpark

[9]https://spark.apache.org/docs/latest/tuning.html
[10]https://spark.apache.org/docs/latest/tuning.html#memory-tuning

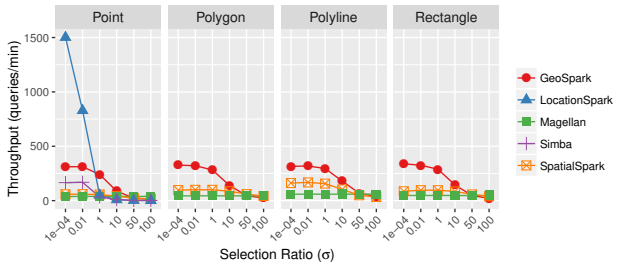


Figure 4: Range query performance on a single node for different selection ratio (σ)

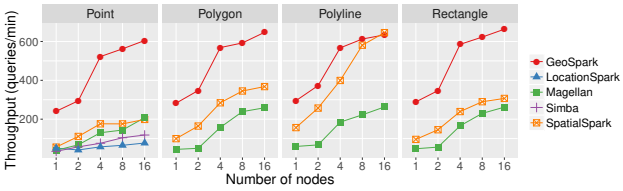


Figure 5: Range query performance for all geometric objects scaling up the number of nodes [selection ratio (σ) = 1.0]

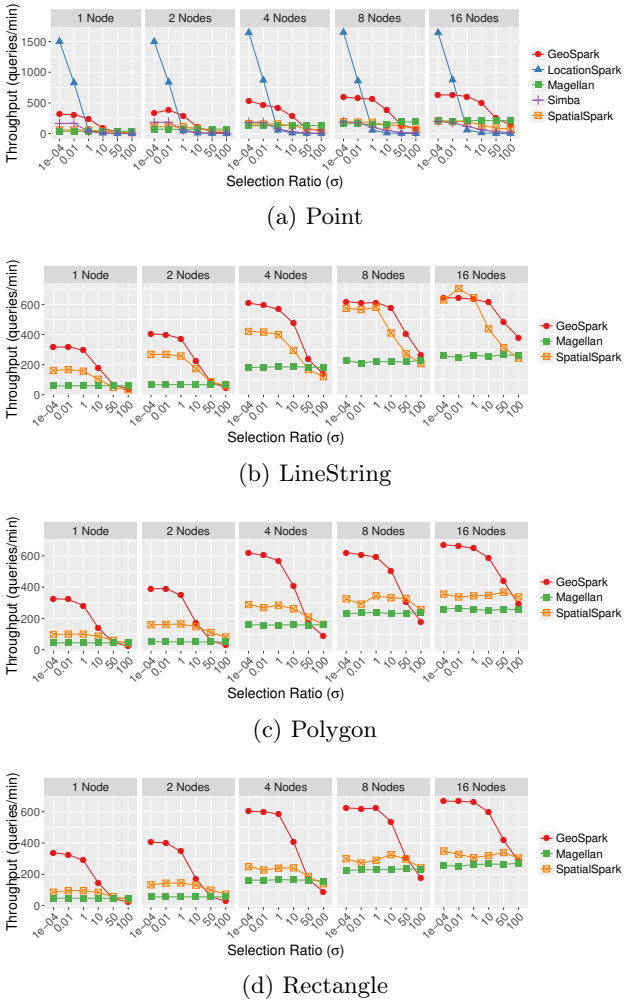
performs the best for selection ratios 0.0001 and 0.01. This is due to the *sFilter* (spatial bloom filter) that it puts on top of the global and local indices. The global index in the systems provides multiple overlapping partitions that intersect with the given range. LocationSpark can further filter out partitions using the *sFilter* from the global index and local indices that do not contribute to the answer and avoids unnecessary scans of partitions and network costs. As the selection ratio increases, LocationSpark's performance degrades as well, similar to other systems, which is expected as more partitions need to be scanned for higher selection ratios. GeoSpark performs better than Simba, because it utilizes the deserialized *IndexedRDD* compared to the serialized index that Simba uses to minimize memory costs and for fault tolerance. Figure 5 shows the range query performance for different datasets, fixing the selection ratio (σ) to 1.0 and scaling up the number of nodes in the cluster. This experiment shows that all the systems scale well with the number of nodes. The scalability is not perfectly linear, but it is acceptable.

Figure 6 shows the range query performance for different datasets for all selection ratio (σ) while scaling up the number of nodes in the cluster. GeoSpark dominates in performance for all the datasets, except in Points dataset where LocationSpark performs 5x better for low selection ratios.

6.3 kNN Query Performance

To study kNN query performance, we generate 100 random location points in the longitude range (-180.0,180.0) and the latitude (-90.0,90.0) range, issue the random locations to the systems and measure the throughput of the systems in queries/minute. We also vary the value of k between 5 and 50. Only three systems support kNN queries: GeoSpark, Simba, and LocationSpark.

Figure 7 shows the kNN query performance varying the value of k on a single node. It can be seen that LocationSpark's throughput fluctuates a lot and is not as stable compared to Simba and GeoSpark. We repeated the experiments multiple times and encountered performance spikes for all values of k. This can be attributed to the



(a) Point

(b) LineString

(c) Polygon

(d) Rectangle

Figure 6: Range query performance scaling up the number of nodes for different selection ratio (σ) on different datasets

sFilter that can significantly decrease the number of partitions to scan. GeoSpark utilizes the JTS library for most of its operations. GeoSpark uses *nearestNeighbour* function in JTS which uses the Branch-and-Bound tree traversal algorithm to provide efficient search for k nearest neighbor in the STRtree (*IndexedRDD* in GeoSpark). This means distance computation to other objects would only be limited to one (or at most two in case the query point overlaps with multiple partitions or is close to the boundaries of partitions). It then uses *takeOrdered* from the results to produce k nearest neighbors. Simba, on the other hand first computes a safe pruning bound to select partitions that contain at least k candidates. It then computes the tight pruning bound by issuing the kNN queries on the selected partitions. Again, similar to GeoSpark, the selected partitions are usually one or two for low values of k since most partitions would contain way more than k elements. Simba, also uses *takeOrdered* on distances to return the first k elements. The difference in their performance comes from the serialized index in Simba. Simba scans over the serialized index while GeoSpark has to scan the deserialized index. LocationSpark, can efficiently utilize the *sFilter* on global and local indices to reduce the distance computations to points in the *LocationRDD*. The fluctuation in performance is due to periodic updates to the

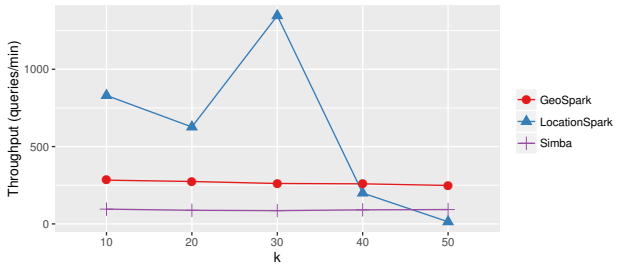


Figure 7: kNN query performance varying k

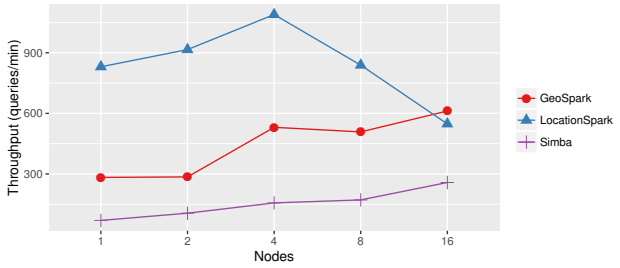


Figure 8: kNN query scalability with $k = 10$

sFilters. Figure 8 shows the kNN query performance scalability with value the of k fixed to 10.

6.4 Distance Join Performance

Only three systems support distance join queries: Simba, GeoSpark and SpatialSpark. Note that DJSpark (Distance Join) in Simba partitions the datasets inside the algorithm and thus we had to embed the timers to compute *Preparation Time* inside the join algorithm. This is the case with SpatialSpark as well. To measure the performance of distance joins we use the Points dataset. The distance for the query is set to 5 meters.

Figure 9 shows the distance join cost breakdown for these systems while scaling up the number of nodes. For distance join, Simba samples both datasets and partitions the two datasets, R and S, using the STR algorithm. Simba then produces partition pairs (i,j) such that r $\in$ R_i, s $\in$ S_j and distance(r,s) $\leq D$ (where D is the distance for the join). After generating these candidate pairs, Simba generates a combined partition P = (R_i,S_j) for each pair (i,j) and broadcasts them to the workers for local join processing. In local join processing, Simba creates local indices for S_j on the fly, and uses R_i to probe into the index to produce the final result. SpatialSpark samples data from only one input dataset and uses partition MBRs to build a spatial index to assign partition IDs for each data item on both sides of the join. This index is broadcasted to all nodes. The data items in both dataset are used to query the index to compute the partition ID that each data item should be assigned to. This assignment of the partition IDs is done using the STR algorithm. The partitioned data items are grouped based on the partition ID on both sides of the join using *groupByKey* function of the RDD. Then the partitions on the two sides are joined into one using the hash-based *join* on the partition IDs (one-to-one integer matching). Finally, these combined partitions are sent to the nodes for local join processing where local indices are built for the partitions and geometric refinement is done. This is how SpatialSpark handles all types of joins (including spatial joins). The spatial predicate (intersects

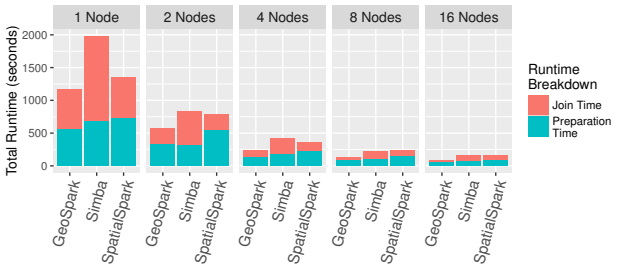


Figure 9: Distance join cost breakdown scaling up the number of nodes

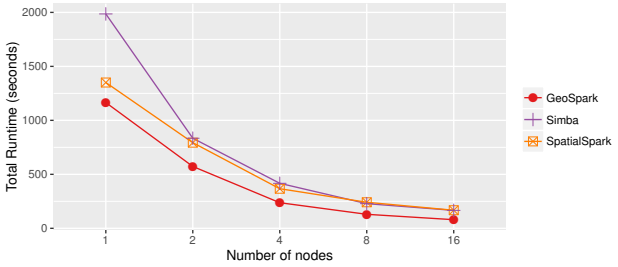


Figure 10: Distance join scalability

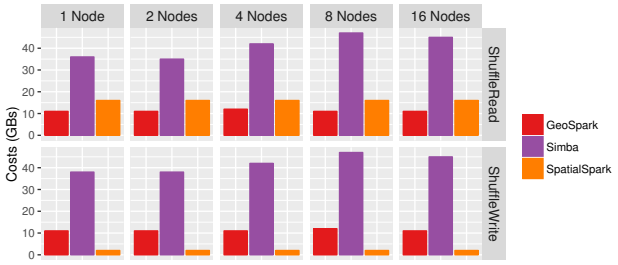


Figure 11: Distance join shuffle costs

or withindistance) for refinement is handled in the local join processing phase. GeoSpark handles the joins in a similar way. An advantage with GeoSpark is that user has more control since it exposes the spatial partitioning and index RDD APIs for applications. This means that distance join (or any join) can be called multiple times without incurring extra costs of partitioning and indexing the RDDs again. In the case of SpatialSpark and Simba, the partitions and the indices are created on the fly which means that partitioning and indexing is tightly coupled with the join algorithm. This implies that the input datasets will be partitioned again, in case distance join has to be invoked again. SpatialSpark and Simba can be tuned to reuse the partitions from the previous join query, but this would require changes to the systems source code rather than the application code. Figure 10 shows the scalability of the systems for distance join query based on *Total Join Time* and Figure 11 shows the shuffle read and shuffle write costs related to the systems. It can be seen that Simba has the highest shuffling costs. The peak memory consumption by GeoSpark, SpatialSpark, and Simba for distance join are **149 GB**, **287 GB**, and **211 GB** respectively for the three systems.

6.5 Spatial Joins Performance

In this experiment, we measure spatial join performance for all possible combinations of geometric objects. To evaluate the systems, we make use of the *intersects* predicate

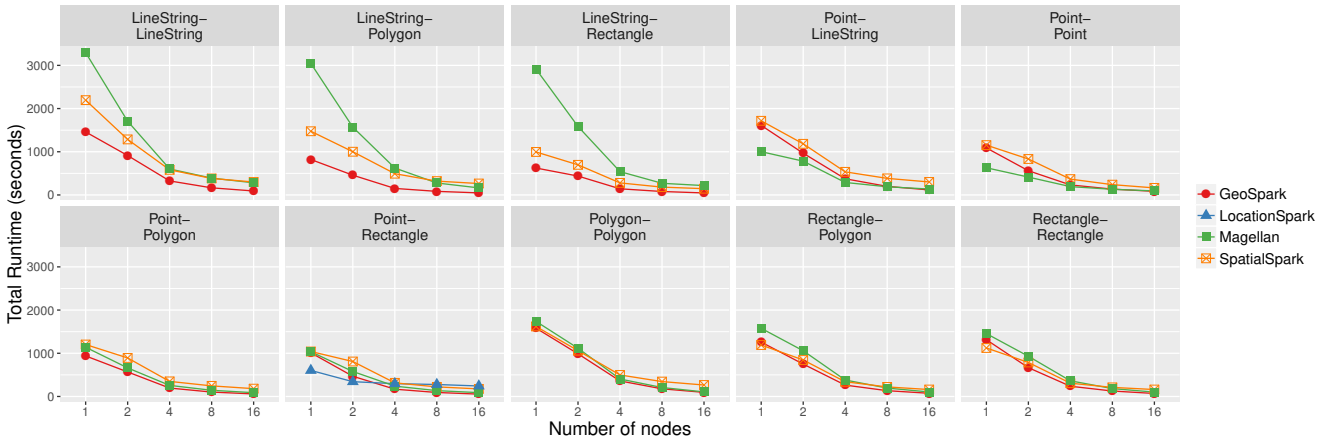


Figure 12: Scalability of all spatial joins for different systems while scaling up the number of nodes

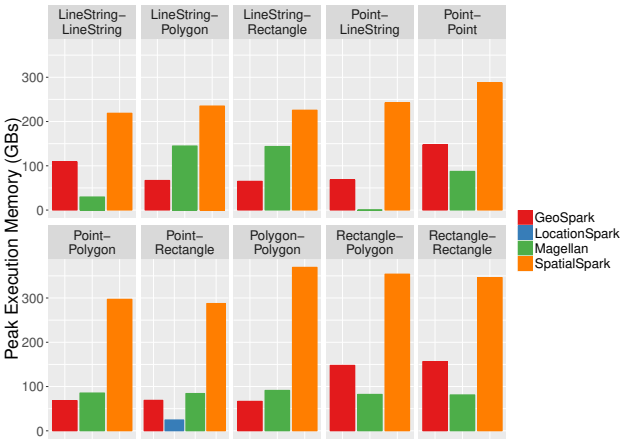


Figure 13: Spatial joins peak execution memory consumption

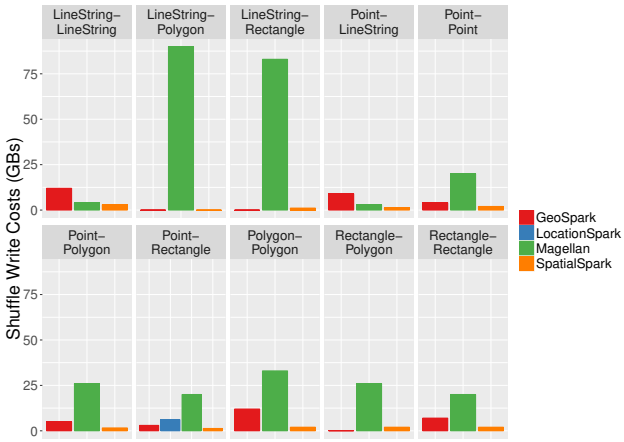


Figure 15: Spatial joins shuffle write costs

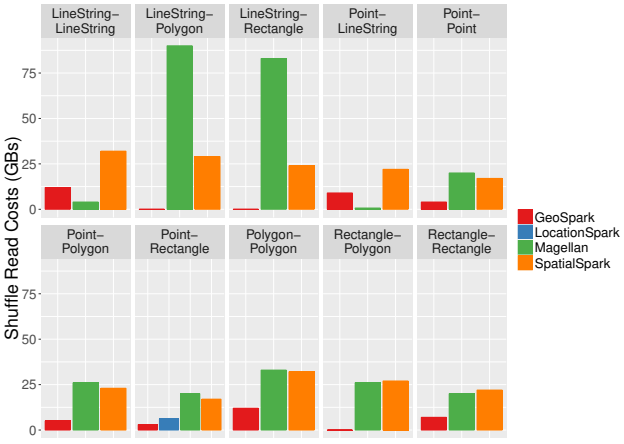


Figure 14: Spatial joins shuffle read costs

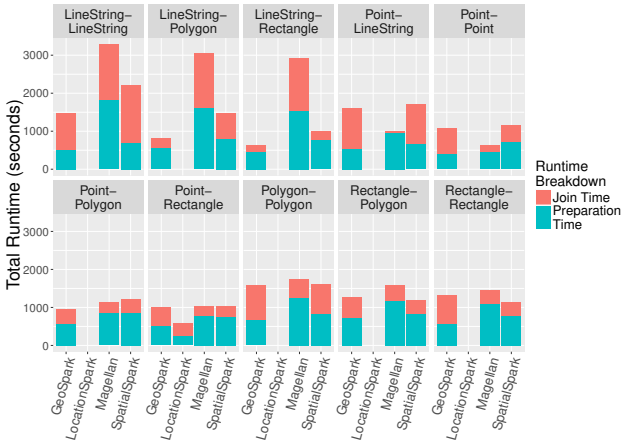


Figure 16: Total runtime cost breakdown for spatial joins between various geometric objects on a single node

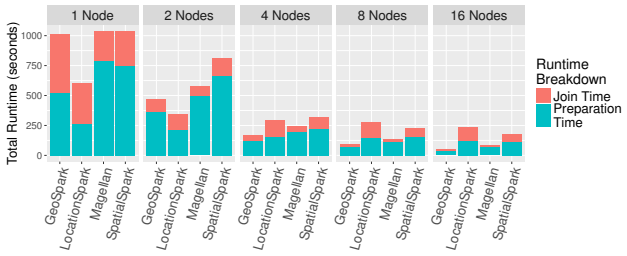


Figure 17: *Point-Rectangle* spatial join cost breakdown scaling up the number of nodes

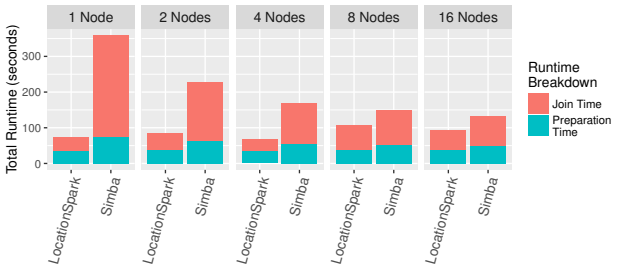


Figure 18: kNN join cost breakdown scaling up the number of nodes

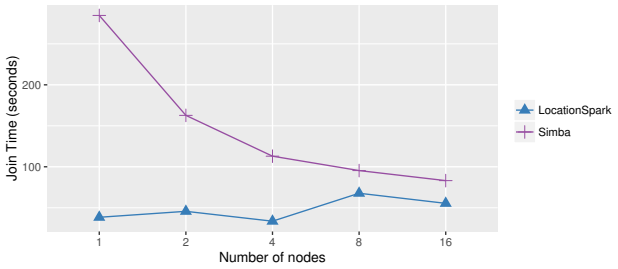


Figure 19: kNN join scalability

in every case. We study the *Preparation Time*, *Join Time*, *Peak Execution Memory* consumption, *Shuffle Write* costs and *Shuffle Read* costs for evaluating join query performance. It is also important to mention here is that at the time of writing, Magellan does not have a full implementation of *Point-LineString* and *LineString-LineString* join. These joins only work for the filter phase where join partners can be filtered out based on the Z-curve value but no exact intersection test takes place. The results produced are only an approximation of the actual join.

Figure 12 shows the scalability of all possible spatial joins based on *Total Runtime*. Figure 13 shows the peak execution memory consumption, Figure 14 shows the shuffle write costs and Figure 15 shows the shuffle read costs related to the systems. Figure 16 shows the spatial joins cost breakdown and join performance for different systems on a single node and Figure 17 shows the *Point-Rectangle* join performance for different systems while scaling up the number of nodes in the cluster.

It can be seen that SpatialSpark has the highest *Peak Execution Memory* consumption, like in the case of distance join. It can also be seen that Magellan has high *Shuffling* costs compared to the other systems, especially in the case of joins that involve *LineStrings*. For the join, Magellan rewrites the plan, to use Z-curve value as the key and adds a *filter* that checks if the curves intersect or not. If the curves intersect, only then does Magellan check whether the spatial objects actually intersect or not. In the refinement phase, Magellan ends up shuffling a lot of data. Note that very little data is shuffled for *Point-LineString* and *LineString-LineString* joins, since these join only have the filter phase. Although, Geospark has high memory consumption for input RDDs, it does not exhibit high *Peak Execution Memory* consumption or high *Shuffling* costs. In almost all cases, GeoSpark performs best for the spatial joins if *Total runtime* is considered.

From the figures we can also see that LocationSpark outperforms the closest competitor GeoSpark for *Point-Rectangle* join (the only supported spatial join in LocationSpark) by 1.5x. This is due to couple of reasons. Firstly, its has low memory related costs. Secondly, LocationSpark has two abstract spatial layers, *LocationRDD* (for locations or points) and *queryRDD* (for rectangles), and a query scheduler. *LocationRDD* is globally and locally indexed along with their embedded *sFilters*. The query scheduler first estimates the cost of query runtime using sampled queries and partitions from *queryRDD* and *LocationRDD*. LocationSpark uses reservoir sampling [28] to sample queries from the *queryRDD* and partitions from the *LocationRDD* and estimates the runtime costs if queries are executed on the sampled partitions. The costs are estimated using techniques proposed in [12]. It then takes the partitions with high query runtime estimates and estimates the cost of repartitioning these partitions and computes the runtime costs to run sampled queries on repartitioned partitions. If the estimated cost of repartitioning and runtime is less than the previously estimated runtime costs, it adds the repartitioning step in the execution plan. This ensures that not only the partition sizes are well balanced but also the amount of work on the executors is also more or less well balanced. Secondly, LocationSpark also filters out multiple partitions for a tuple from the *queryRDD* to join against using its *sFilter*, in a similar way as it does it in the case of range and kNN queries.

6.6 kNN Join Performance

Only two systems support kNN join: Simba and LocationSpark. For kNN join query we fix the value of k to 5 and measure the join performance for the two systems. Another point to make here is that the kNN Join query for the Points datasets (200 million points) crashed in Simba. We will explain the reason later. Since, Simba [30] used a maximum of 10 million points (for both datasets) in their evaluation of kNN join, we decided to do the same. We sampled 10 million points from the Points dataset and then ran the kNN join query on them. Since we reduced the dataset to 10 million points for both datasets we had to run multiple experiments to determine a good number of partitions for Simba. LocationSpark does not require tuning the number of partitions for the *LocationRDDs* as the query scheduler and optimizer already does it and overwrites the number of partitions specified by the user. On the other hand Simba, by default, sets the number of join and index partitions to 200 each. We found that 50 partitions performed the best for Simba for 10 million points.

Figure 18 shows the kNN join cost breakdown and Figure 19 shows the scalability of the systems based on *Join Time*. LocationSpark balances the work among the Spark

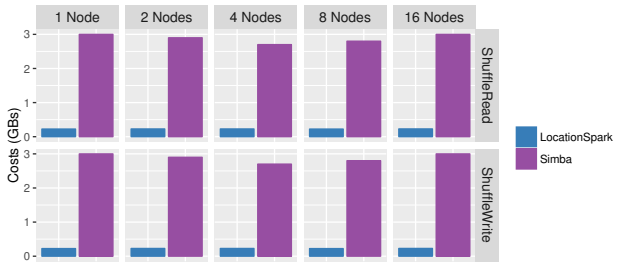


Figure 20: kNN join shuffle costs

workers well, using the query cost estimation mentioned previously in Section 6.5. Simba is not able to do so, and ends up creating overloaded partitions because of duplicated points. This can be attributed to how the kNN join algorithm (*RKJSpark*) works in Simba: Let the two datasets be R and S. RKJSpark algorithm tries to find n partitions of S to pair with n partitions of R, such that these paired partitions can be combined into one RDD partition using *zipPartitions* and then kNN join can be run on them locally. The pairing is done by computing distance bounds (γ). Simba partitions R into n partitions (R_n) and computes a distance bound (γ_i) for each partition R_i in two steps. First, for each partition R_i, the algorithm computes the distance of centroid (C_i) of the MBR (minimum bounding rectangle) of the partition to the furthest point in the partition (we denote this distance as D_{i1}). Second, it samples a set of points from S and builds an R-tree on the sampled dataset. It then computes the kNN of the centroid (C_i) of each partition (R_i) from the sampled dataset using the R-tree and selects the distance of the furthest kth neighbor (D_{i2}). The distance bound (γ_i) is then set to $2D_{i1} + D_{i2}$. Note that the distance bound is different for each partition. The algorithm then partitions S into n partitions based on

$$S_i = \{\ s | s \in S,\ distance(C_i, s) \leq \gamma_i\ \}$$

This means that for every s ∈ S, *RKJSpark* includes a **copy** of s in S_i if distance(C_i,s) ≤ γ_i. This creates a lot of duplicated points in the partitions for S and leads to more and redundant computations. This is also the reason, why Simba crashes for the Points dataset (200 million) where it simply runs out of heap space because of a lot of duplicated points.

Figure 19 shows the scalability of the systems for kNN join query based on *Join Time*. It can be noticed that LocationSpark shows a slight increase in runtime for 8 and 16 nodes. This is due to the communication cost where more *executors* return the local result to the *driver*.

Figure 20 shows the shuffle costs for each system. It can be seen that Simba has a higher *Shuffling* related costs as compared to LocationSpark. The peak memory consumption for LocationSpark and Simba is **2.24 GB** and **1.75 GB** respectively for the two systems.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we evaluated five Spark based spatial analytics systems. We performed an experimental evaluation of these systems using real-world datasets. Table 4 summarizes the strengths and weaknesses of the systems. From our experience, GeoSpark comes close to a complete spatial analytics systems because of data types and queries supported and the control user has while writing applications. It also

Table 4: Strengths and Weaknesses

System	Strengths	Weaknesses
GeoSpark	Query optimizer Scales well Rich in features Active development	No kNN join
Simba	Query optimizer Scales well	Limited data types No recent development
LocationSpark	Query optimizer and scheduler Spatial bloom filter	Limited data types No recent development
Magellan	Join query optimizer Low join time Scales well Active development	High shuffle costs High preparation times No range query optimization
SpatialSpark	Scales well	No recent development High memory costs

exhibits the best performance in most cases. There are a few drawbacks though. First, it consumes a large amount of memory for the input datasets. Second, GeoSpark does not support kNN joins yet. Magellan also exhibits good performance for some spatial joins especially if only *Join Time* is considered, but it does not have any optimization for range queries. Also, it does not support kNN queries, distance joins and kNN joins. Moreover, Magellan has very high shuffling related costs. An advantage of GeoSpark and Magellan is that they are actively under development. LocationSpark is interesting since it has a very good query scheduler and optimizer. Also, it has a spatial bloom filter *sFilter* which brings the query costs down. The aforementioned systems may look to incorporate such filters in their system as well. Again, the limitation is that it has limited data types and there has not been any development recently. Simba, like LocationSpark, has very limited data types (only points) and does not support spatial joins. SpatialSpark is competitive but has high *Peak Execution Memory* consumption. Moreover, there has been no active development. We also see that all the systems evaluated scale pretty well with more resources.

A recent development in the area of spatial joins has been in the area of approximate and adaptive joins with precision guarantees [34] [11] [10]. The motivation behind such joins is that many applications today do not require the join results to be accurate and only need an approximation to make certain decisions. The systems studied in this paper may look to add such joins for such applications. Another interesting field is the area of trajectory similarity search, and an operator for such queries in these systems would be a welcome addition for many users. Also, Postgres with its extension PostGIS is rich with a variety of spatial operators that the Spark based spatial systems do not currently have and could be implemented in the future.

8. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable feedback.

This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

9. REFERENCES

[1] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz. Hadoop-gis: A high performance spatial data warehousing system over mapreduce. *PVLDB*, 6(11):1009–1020, 2013.

[2] P. Baylis. Temperature and temperament: Evidence from a billion tweets. *Energy Institute at HAAS working paper*, 2015.

[3] M. B. Brahim, W. Drira, F. Filali, and N. Hamdi. Spatial data extension for cassandra nosql database. *J. Big Data*, 3:11, 2016.

[4] A. Eldawy, L. Alarabi, and M. F. Mokbel. Spatial partitioning techniques in spatial hadoop. *PVLDB*, 8(12):1602–1605, 2015.

[5] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1352–1363, 2015.

[6] A. Eldawy and M. F. Mokbel. The era of big spatial data. *PVLDB*, 10(12):1992–1995, 2017.

[7] *Foursquare*. `https://foursquare.com/about`.

[8] *Apache Hadoop*. `https://hadoop.apache.org`.

[9] *Apache Impala*. `https://impala.apache.org/`.

[10] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. Boncz, T. Neumann, and A. Kemper. Approximate geospatial joins with precision guarantees. In *34rd IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, 2018.

[11] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. A. Boncz, T. Neumann, and A. Kemper. Adaptive geospatial joins for modern hardware. *CoRR*, abs/1802.09488, 2018.

[12] Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 75–86, 2010.

[13] *One Million Rides a Day*. `https://blog.lyft.com/posts/one-million-rides-a-day`.

[14] *Magellan: Geospatial Analytics Using Spark*. `https://github.com/harsha2010/magellan`.

[15] *MemSQL Geospatial*. `http://www.memsql.com/content/geospatial/`.

[16] *NASA OpenNEX*. `https://nex.nasa.gov/nex/static/htdocs/site/extra/opennex/`.

[17] *Spatial And Graph Analytics With Oracle Database 18c*. `http://download.oracle.com/otndocs/products/spatial/pdf/Spatial_and_Graph_TWP_18c.pdf`.

[18] V. Pandey, A. Kipf, D. Vorona, T. Mühlbauer, T. Neumann, and A. Kemper. High-performance geospatial analytics in hyperspace. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2145–2148, 2016.

[19] T. Schneider. *Analyzing 1.1 Billion NYC Taxi and Uber Trips, with a Vengeance*. http://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/.

[20] J. P. Singh, Y. K. Dwivedi, N. P. Rana, A. Kumar, and K. K. Kapoor. Event classification and location prediction from tweets during disasters. *Annals of Operations Research*, pages 1–21, 2017.

[21] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref. Locationspark: A distributed in-memory data management system for big spatial data. *PVLDB*, 9(13):1565–1568, 2016.

[22] *TNCs TODAY : DATA EXPLORER*. `http://tncstoday.sfcta.org/`.

[23] *Making the most detailed tweet map ever*. `https://blog.mapbox.com/making-the-most-detailed-tweet-map-ever-b54da237c5ac`.

[24] *How Twitter shapes the food trends we follow*. `http://www.fooddive.com/news/social-media-eating-behavior/428596/`.

[25] *Uber Hits 5 Billion Rides Milestone*. `https://www.uber.com/en-SG/blog/uber-hits-5-billion-rides-milestone/`.

[26] *UBER Movement*. `https://movement.uber.com/`.

[27] *ENGINEERING INTELLIGENCE THROUGH DATA VISUALIZATION AT UBER*. `https://eng.uber.com/data-viz-intel/`.

[28] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[29] H. Vo, A. Aji, and F. Wang. 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, Dallas/Fort Worth, TX, USA, November 4-7, 2014*, pages 545–548, 2014.

[30] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1071–1085, 2016.

[31] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17, 2015*, pages 34–41, 2015.

[32] S. You, J. Zhang, and L. Gruenwald. Spatial join query processing in cloud: Analyzing design choices and performance comparisons. In *44th International Conference on Parallel Processing Workshops, ICPPW 2015, Beijing, China, September 1-4, 2015*, pages 90–97, 2015.

[33] J. Yu, J. Wu, and M. Sarwat. Geospark: a cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, pages 70:1–70:4, 2015.

[34] E. T. Zacharatou, H. Doraiswamy, A. Ailamaki, C. T. Silva, and J. Freire. GPU rasterization for real-time spatial aggregation over arbitrary polygons. *PVLDB*, 11(3):352–365, 2017.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.