Spatial Data Processing Frameworks - A Literature Survey

Ayman Zeidan

Department of Computer Science
The Graduate Center of the City University of New York
365 5th Ave, New York, NY 10016

Graduate Committee
Dr. Huy T. Vo, The City College of New York, New York, NY 10031
Dr. Feng Gu, The College of Staten Island, New York, NY 10314
Dr. Kaliappa Ravindran, The City College of New York, New York, NY 10031
ABSTRACT

Location-based applications and services have become an integral part of our lives. These applications extract meaningful information through the analysis of large location-tagged (spatial) datasets that are generated like never before. The term "Information Explosion" is often used to describe the sheer amount of data that is being made available to individuals, businesses, and other entities. Traditional computing and database systems fall short when it comes to the efficient handling of truly large datasets. Consequently, several high-performance parallel computing systems were developed with the goal of providing quick, accurate, and scalable solutions. Unfortunately, today’s state-of-the-art parallel processing systems are generic systems and are not well suited to perform efficient processing of large spatial datasets. Therefore, specialized frameworks are needed to empower these systems and improve spatial data processing. Instead of building parallel processing systems from the grounds up, already existing and stable systems like Apache Hadoop and Apache Spark are utilized.
# ABSTRACT

# LIST OF TABLES

# LIST OF ILLUSTRATIONS

<p>| 1 | Introduction | 1 |
| 2 | Challenges with spatial data analysis | 3 |
| 2.1 | The portion that is spatial data | 3 |
| 2.2 | Different shapes and sizes | 4 |
| 2.3 | Data at rest | 5 |
| 2.4 | Operations | 7 |
| 2.5 | Usability and Integration | 8 |
| 2.6 | Indexing | 9 |
| 2.7 | Temporal Aspect | 11 |
| 2.8 | Interactive vs Batch | 11 |
| 2.9 | Scalability | 12 |
| 2.10 | Reliability | 12 |
| 3 | Spatial Data Analysis Frameworks | 13 |
| 3.1 | Hadoop-Based Frameworks | 14 |
| 3.1.1 | Esri GIS Tools for Hadoop | 15 |
| 3.1.2 | Hadoop-GIS | 17 |
| 3.1.3 | SpatialHadoop | 19 |
| 3.1.4 | Features and Performance Summary | 21 |
| 3.2 | Spark-Based Frameworks | 22 |
| 3.2.1 | SpatialSpark | 23 |
| 3.2.2 | GeoSpark | 24 |</p>
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3</td>
<td>LocationSpark</td>
<td>27</td>
</tr>
<tr>
<td>3.2.4</td>
<td>STARK</td>
<td>29</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Simba</td>
<td>31</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Features and Performance Summary</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>Future Work</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>Summary</td>
<td>37</td>
</tr>
</tbody>
</table>
LIST OF TABLES

TABLE 1: Feature comparison of Hadoop-based frameworks .................. 22
TABLE 2: Feature comparison of Spark-based frameworks .................... 35
LIST OF ILLUSTRATIONS

FIGURE 1 : Selected Spatial Data Geometric Shapes .......................... 4
FIGURE 4 : Architecture of SpatialHadoop[37] ............................... 19
FIGURE 5 : GeoSpark Architecture[61] ....................................... 25
FIGURE 6 : LocationSpark Architecture[54] ................................. 27
FIGURE 7 : STARK Architecture[44] ........................................ 29
FIGURE 8 : Simba Architecture(Orange shaded boxes)[58] ............... 31
1 Introduction

Nearly every computer application generates some form of data. Depending on the application, this data can come from different sources like runtime log files, activity recordings, features usage, weather sensors, and/or driving habits. Over time, this data accumulates and grows to the point where it is too big to extract meaningful information using traditional techniques or using a single machine (node). When a dataset grows in size in a short amount of time, it is referred to as Big Data. The Oxford English Dictionary defines Big Data as "Extremely large datasets that may be analyzed computationally to reveal patterns, trends, and associations, especially relating to human behavior and interactions." While, the exact definition of big data is still somewhat subjective, everyone seems to agree on the 3Vs of big data – Volume, Variety, and Velocity. Some will even extend these to include the possible value that can be extracted from the data.

A significant portion of the data collected contains a spatial component that indicates the physical location where the data point was collected. This is crucial since knowing the data’s physical location increases the chance of extracting additional valuable information that would not be available otherwise. As a result, interest in collecting and analyzing big spatial data has increased significantly. In 2014, Facebook, the world’s largest social networking site, announced that their data warehouse can store 300 petabytes of data with 600 terabytes of daily incoming [26]. Facebook uses this data to improve features and produce targeted ads that are tailored to individual users. The data collected is tagged with various information including the location from where the user logged in, accessed a page, clicked on an ad, and the device’s specifications[27]. A Boeing 787 Dreamliner airplane can generate as much as 500 gigabytes of flight data by collecting location-tagged information from engines, sensors, fuel tanks, crew and passenger activities, and weather sensors[28]. Some of the data is analyzed on board during the flight to aid the pilot and crew while the rest is analyzed later to improve future flight experiences. Google houses one of the world’s largest data centers, processes over 40,000 queries per second, and can process over 20 terabytes of raw web data[29]. Many of Google’s services are free, but the data collected from the users’ interactions with these services is analyzed to produce targeted ads and develop and improve company services.
Many other examples exist that show how and why companies collect, store, and analyze data including spatial data. However, the ultimate goal is the same: unlock hidden values in these datasets to improve and invent. Data analysis must be done quickly and sometimes it is needed in real-time. Analyzing spatial data differs from other types due to a number of challenges. Mainly, the analysis must take into consideration the spatial attributes of the data, different shapes and sizes (i.e. Point, Polygon, LineString), and the different operations (i.e. Cluster, Distance, Join, kNN).

To that effect, specialized systems where invented to offer support to spatial data. For a long time, relational database systems (RDBMS) were a viable and attractive option for spatial data management. RDBMS like Oracle[17], PostGIS[18], and SQL Server[24] offer support to spatial data objects and operations. However, their capabilities are limited by the size and shape of the dataset. More recently, parallel processing on commodity hardware gained popularity for being inexpensive, easy to use, easy to maintain, and highly scalable.

Two of the most popular processing frameworks are Apache Hadoop[36] Spark[63]. They differ from each other in design and programming model, but they are both designed to handle generic datasets. While they can be used to process spatial datasets, results are achieved with considerable time and resource cost. Mainly, this is due to the lack of recognition and treatment of spatial data. To that effect, spatial frameworks were designed to work with Hadoop and Spark and make them recognize the different shapes and operations of spatial data. For Apache Hadoop, Esri GIS tools for Hadoop[9], Hadoop-GIS[30], and SpatialHadoop[37] utilize one or more layers from the Hadoop ecosystem to add spatial support. For Apache Spark, SpatialSpark[60], GeoSpark[61], LocationSpark[54], STARK[44], and Simba[58]) utilize Spark’s ecosystem to add spatial support.

All of these frameworks are similar in the sense that they allow for performing spatial operations against spatial objects. However, a number of drawbacks exist and as of the time of this writing, no single framework offers support to all spatial objects and operations. In section 2 we discuss some of the challenges researchers face when designing Hadoop and Spark spatial frameworks. In section 3 we survey a number of Hadoop-based (section 3.1) and Spark-based (section 3.2) frameworks.
2 Challenges with spatial data analysis

Analyzing spatial data differs from other types due to a number of factors like shapes, sizes, and operations. Extracting the spatial attribute from a dataset requires knowledge of the underlying structure. Information about where the data is stored, how it is stored, and the frequency of updates can affect the analysis process. Spatial objects can take on multiple different shapes and sometimes the dataset can be a hybrid of these shapes.

Any spatial data analysis system must be able to ultimately produce meaningful and accurate results in a timely manner. The scalability of the system is affected by decisions like what data should be kept in memory and in what shape should non-spatial data make it to the output. The system should also be optimized for streaming and/or batch processing. Usability of the system is also important as it will determine whether experts and/or non-experts can use it.

2.1 The portion that is spatial data

Spatial data is often part of a bigger picture. Depending on the analysis being done, the spatial attribute may be relevant at first, towards the end, or in between. For example, Twitter[25] data (called tweets) have been used for sentiment analysis (opinion mining) to predict election results[46, 56], foresee future stock prices[35], or collect product reviews[45]. Such an analysis can skip the spatial attribute of these tweets and produce meaningful results. However, taking into consideration the geographic locality of a tweet can significantly improve the results. It is more meaningful to consider tweets of users from certain cities (i.e. New York, London) who are reviewing products that are only available in those cities.

Moreover, after the spatial analysis step is completed, the data can undergo additional non-spatial analysis. This naturally requires that the spatial processing system preserve any non-spatial data that was originally present. Doing so will have an impact on the performance and frameworks like Hadoop-GIS and SpatialSpark do not allow for non-spatial data to be carried through the computation steps.
2.2 Different shapes and sizes

One of the major challenges with spatial data is that it can take different shapes, sizes, and dimensions. Moreover, they can be formed from coordinate systems like Global Positioning System (GPS) coordinates or planer coordinates. The Open Geospatial Consortium[6] (OGC) aims at creating an open standard for geospatial content and services and offers certification of systems. By doing so, a system’s interoperability is increased, vendor’s confidence is improved, and users are assured that multiple systems can work together. Figure 1 depicts some of the most common forms of spatial data that a system should support.

Point: A Point object (Figure 1a) is the simplest spatial object and is the basic building object of other spatial objects. A Point consists of two coordinates like longitude and latitude or $(x, y)$. A single location on a map like a restaurant or a subway station can be represented as a Point.

LineSegment: A LineSegment object (Figure 1b) consists of two points with the first marking the beginning of a segment and the last marking its end. If a spatial object consists of two or more unconnected LineSegments, a MultiLineSegment (Figure 1c) object is used to represent its shape. A straight road can be represented as a LineSegment object.
**LineString:** A *LineString* object (Figure 1d) consists of two or more connected LineSegments that do not form a closed shape. The endpoint of the first LineSegment marks the beginning of the second one. If a spatial object consists of two or more unconnected LineStrings, a *MultiLineString* (Figure 1e) object is used to represent its shape. A city road that consists of multiple segments is represented as a LineString object.

**Polygon:** A *Polygon* object (Figure 1f) consists of multiple LineStrings that must form a closed shape. A Polygon can also contain another Polygon (Figure 1g). If a Polygon consists of two or more Polygons (Figure 1h) a *MultiPolygon* object is used to represent its coordinates. Countries, states, city blocks, and water ponds can all be represented using a Polygon object.

These shapes can be further used to form even more complex objects like *MultiPoint*, *Circle*, and *Curve*. Moreover, the dataset can be heterogeneously composed of different shapes which further complicates the analysis process. A good system must recognize spatial shapes for what they are in order to produce fast and meaningful results. A common approach to overcome the lack of support for other objects is to compute the object’s envelope (or Minimum Bounding Regions (MBR)) – the smallest rectangle that fully encompasses the object.

### 2.3 Data at rest

Data can be stored (disk, tape) in many different shapes and formats. A processing system must account for these variations without limiting its capabilities. This goes beyond whether the data is encrypted and/or compressed. A spatial processing system is no different and should account for datasets that are of single or multiple types, uniformly distributed or not . . . Determining the type(s) of spatial data can become expensive; the system can either make certain assumptions about the data and automatically load it, or rely on the user to preload their data. This technique is implemented in [61, 44, 58]; they allow users to manually parse and load the dataset or request that the framework automatically read and parse the datasets if it is in a specific format like WKT\(^1\), CSV\(^1\), GeoJSON\(^1\) . . . However, this feature is restricted to supported formats and does not allow for non-spatial data. In general, there are three different classifications of data (including spatial):
**Structured:** This type of data is well formed with a clear data model. The fields’ types are known and decide how they are stored (integral, currency, point, polygon . . .). It has the advantage of being easily generated, stored, queried, and analyzed. Structured data is ideal for use in a traditional RDBMS like MySQL, Oracle, and Microsoft SQL Server. However, even with a clear structure, these RDBMSs are limited by the amount of data they can store and process in a timely manner. In essence, one can only scale up/out so much before query times become noticeably lagging[43]. Other systems like NoSQL can also be used to process large datasets although they are not yet mature enough mainly because they try to merge RDBMS and distributed storage features into one system.

**Unstructured:** Unlike structured data, unstructured data is characterized by not having a clear data model. Their field types are hard to discern and sometimes impossible to assign them a type. Most of the data that is being generated nowadays is unstructured. Documents like photos, videos, web server logs, word documents, spreadsheets, and PDFs are some of the examples of unstructured data. Storing them in a traditional RDBMS is near impossible or impractical at best. As a result, they are mostly stored as either text or binary files that should be indexed and analyzed. Often, the analysis occurs multiple times with the original files kept intact. Systems like Hadoop[4], MapReduce[36], Spark[3], and Impala[12] were all designed to help with the management of unstructured data. Their techniques differ from simple distributed storage, key-value storage, document storage, or wide-column storage[40].

**Semi-structured:** A hybrid form of the two preceding types is a semi-structured dataset. It is not as well formed as structured data but a partial data model can be discerned. Depending on the data itself, semi-structured data can be managed using structured or unstructured techniques. Although, traditional RDBMS tend to lag when the size of the dataset exceeds a certain threshold. As a result, distributed systems like the ones mentioned for unstructured data are better suited. Examples of semi-structured data include E-mails, Metadata of documents (word, spreadsheet files, PDFs . . .), and media-file properties (time, location, size . . .).
2.4 Operations

Analyzing spatial data means performing different operations that depend on the spatial objects and desired results. Implementing these operations must take into consideration the object’s type, the mixture of the objects, and how to carry non-spatial information through the computation steps (if any).

Currently, no system can implement all the different possible combinations. Instead, some systems like [61, 54, 44, 58] will design their code in such a way that it can be extended to add additional support. Unsupported objects can be converted to generic Rectangle objects by computing their MBRs as done in [9, 30, 37, 60]. The more operations that a system can support, the more attractive it becomes; therefore, third-party libraries like Java Topology Suite (JTS)[15] or Geometry Engine - Open Source (GEOS)[8] are utilized to improve features. It is crucial to find a lowest common denominator such that more operations are supported with minimal code without affecting performance or increasing a system’s complexity. Some of the most common spatial operations are:

Range: In a range operation, the input is two sets of spatial objects $S$ and $R$. The output is a set of all records in $S$ that overlap $R$. Systems like [58] will implement this operation with a single dataset and a range query, while others will allow for two large datasets [61, 54, 44] or one small and one large [30, 60].

Contains: In a contains operation, the input is two spatial objects $O_1$ and $O_2$. The output is true if $O_2$ contains $O_1$, or false otherwise. Systems like [61, 60, 44] implement this operation for objects like point and polygon while [58, 30, 9] do not offer this operation.

$k$ Nearest Neighbor (kNN): kNN queries can take on different shapes like distance kNN and kNN join. The simplest form of a kNN operation consists of an input set of Point objects, $P$, a set of spatial objects $S$, and an integer value $k$. The output consists of all of the elements of $P$ along with the nearest $k$ objects from the set $S$ for each element in $P$. In [58] kNN join works for when one of the datasets is a set of point objects. Others like [61, 54, 44] allow for polygons and generic rectangle objects but only as a distance kNN join.
**Join**: In a join operation, the input is two sets of spatial objects $S$ and $R$ and a spatial predicate. The output is a pair of elements $(s, r)$ such that $s \in S$, $r \in R$ and the predicate is true. The predicate can be one of many like equals, larger than, within distance, or overlap. Due to the large range of predicates, only a small number is supported. In [60] a spatial join is implemented for intersect and within; in [61] only contains and intersects are supported. A better approach is taken in [44] where the predicate is a user-submitted function with some sample functions (i.e. distance) already implemented.

### 2.5 Usability and Integration

From a researchers point of view, making a spatial system easy to use and integrate is not very interesting. Once the processing system’s features are implemented and are stable, ease-of-use may be taken into consideration but only to allow end-users from a non-computer science background to interact with the system.

Since currently no system can cover all spatial objects, researchers aim at creating an easy-to-extend system. This allows others to write code to extend the functionality of the system to include new objects and operations. Some of the techniques used are supporting standards like OGC[6] as partially done in [9, 59, 37], Structured Query Language (SQL) as done in [58], extending existing high-level languages like Hive$^1$[55, 1] as done in [9, 30], Pig-Latin$^2$[50, 2] as done in [37], or integrating with APIs like Spark’s RDD$^3$[62, 19] as done in [61, 54, 44, 54], DataFrame$^4$[31, 21], and/or Spark SQL module$^5$[31, 21] as done in [58].

A closely related topic to usability is integration. A processing system should allow for easy integration of input and output with other tasks. Spatial data is not always the first or final step. Often a spatial operation will start after some other operation and/or end before another task starts. For example, consider the problem of finding all restaurants along a driving route. While a spatial query is needed to find the restaurants (Points) around the different streets (LineSegment),

---

$^1$Hive is a SQL-like language for working with data stored on HDFS  
$^2$Pig-Latin is an abstraction layer intended to simplify MapReduce programming using easy to understand idioms  
$^3$Resilient Distributed Dataset (RDD) are read-only in-memory distributed data structures  
$^4$A DataFrame is a dataset that resembles a table in a RDBMS  
$^5$Spark SQL is a Spark module for working with structured data through a SQL-like manner
the search results may be further refined by looking at attributes like the type of food served, drive-thru vs. sit Down . . . Most systems surveyed here allow for some form of integration by either writing their output to HDFS for other tasks to use [9, 30, 37] or through RDD transformations as done in [61, 54, 44]

2.6 Indexing

Spatial indexing is a step in spatial data processing that is usually preferred to speed up operations. In most cases, spatial data is not pre-indexed, therefore, a spatial processing system can offer an implementation of one or more indexing techniques. These indexes can be built on the fly (live) and perhaps written to disk (persisted) in order to be used in subsequent processing tasks.

In a distributed spatial system, there are usually two levels of spatial indexing, global and local. Global indexing is used to perform an initial grouping of data based on their spatial relationship; this step is also referred to as partitioning because it partitions the data across the processing nodes. Depending on the processing system, the global index can be kept on the master node or broadcasted to all processing nodes [61]. The global index can be built by either sampling the dataset to avoid processing large amounts of data, or, if feasible, through a full scan of the dataset(s). All of the frameworks that we have surveyed employ sampling techniques when partitioning the data. The global index can, also, be used to exclude data that does not contribute to the output thus improving the overall performance. Local indexing, on the other hand, is utilized on the processing nodes after the data has been partitioned. This speeds the process since only a subset of the data on the local machine are considered. The results that were obtained through the local index can be further refined by calculating the exact relationship between objects.

Many indexing techniques exist and can even be used in conjunction with one another. Which technique to use depends on factors like on-disk or in-memory, type of dataset, and/or speed of construction/retrieval. Some of the most popular spatial indexing structures are:

**Grid**: A grid[52] spatial index is an indexing structure for organizing spatial objects. A number of different types of grids exist, but in its simplest form, a 2-D rectangle is divided into a number of contiguous and equal cells. Each cell is assigned a unique ID and can have a similar or different
size from the others. Spatial objects are assigned to one or more cell in the grid depending on their shape. Grid indexing is useful when the data is uniformly distributed.

**R-Tree:** An R-Tree[42] is a height-balanced tree used for indexing multidimensional objects like Points, Rectangles, and Polygons. Objects are inserted into the tree with their MBRs and Nearby MBRs are grouped together in order to expedite searching. R-Trees are popular in spatial data processing but produce approximate results and usually require a finer comparison step.

**R*Tree:** An R*-Tree[33] is a variation of an R-Tree that tries to minimize the MBR coverage and overlap. As a result, its querying times are faster but update times are slower. R*-Tree is better suited when the tree is queried more often than it is modified.

**Binary Space Partitioning:** Binary Space Partitioning (BSP) is a method of continuously dividing a plane into two or more halves. A record of how space is being divided is kept in a tree data structure to represent the BSP. Mainly it was invented for computer graphic rendering, but it can also be used for indexing spatial objects and is the basis of structures like Quadtree and K-D Tree.

**Quadtree:** A Quadtree[39] is a variation of a BSP where each node has zero (leaf) or exactly four nodes. A 2-D space is constantly divided into four regions such that the regions satisfy a certain condition (i.e. divide until each region has 0 or 1 points).

**K-D Tree:** A K-D Tree[34] is a variation of BSP and a generalization of a binary tree. Each node has zero (leaf) or exactly two nodes. A 2-D space is constantly divided into two such that the regions satisfy a certain condition (i.e. divide until each region has 1 point). Different from a Quadtree, a K-D Tree splits the node into two according to some mathematical equation like the mean of the node’s data.

**Space-Filling Curves:** Space-Filling Curves (SFC) is a technique to use a line in order to fill a 2-D space. One of the most popular SFC is the Hilbert Curve with the indexes indicating an initial clustering of spatial objects. The precision of the curve can be increased with the number of iterations, \( n \); however, this may decrease the performance of the index.
2.7 Temporal Aspect

Many of the spatial data collected have a time component which indicates the time that data was collected (timestamp). In some analysis, taking the timestamp into consideration produces more meaningful results because it focuses on, for example, more recent recordings.

Adding temporal support to spatial data querying comes with a set of unique challenges. For example, each spatial object must be able to hold information about its own timestamp. Partitioning the data will also need to take into account the time factor and instead of only using a spatial index like an R-Tree, a time index like an interval tree must be added. By doing so, the data processing workflow will change since, for example, a specific object must be duplicated if it spans multiple time segments. Performing the spatial query is also affected since the query should account for the specified time and include or exclude certain objects. Join, $k$NN, filter queries, will become spatio-temporal queries where the predicates must examine the time factor. Out of all the systems mentioned in this survey, only the Spark based systems in [44, 58] have taken the temporal aspect into consideration but in a very limited scope.

2.8 Interactive vs Batch

There are two types of spatial search that a system can offer, interactive and batch. An interactive search is a live search such that the search query is executed on demand. All systems surveyed here are designed for batch processing but some can be used for interactive searches [58, 61, 37]. An example of an interactive search is a person looking for shops in a specific neighborhood. The dataset containing all the shops in all areas is preprocessed and put on standby (Ram or disk). When the search query arrives, it is executed against the existing dataset and results are filtered accordingly. This type of search is ideal for one large dataset and one relatively small. Frameworks like [61, 37] have described a graphical interactive interface that they developed to demonstrate their techniques.

Batch processing involves multiple objects (one or more types) across two datasets. Objects from both sets are examined in order to determine their relationship. An example of this would be
finding out which of a given dataset of tweets originated near a body of water (lake, pond, rivers . . . ). This type of search usually calls for the preprocessing and indexing of both datasets in order to perform the spatial query.

Each one of these search types requires its own optimization. The size of the dataset(s) in question is key in both cases as the result might call for writing a portion of the dataset(s). The system must also be smart about the amount of memory available and how much of it is used for computations and for object caching. If the system is distributed, then distributed memory may call for some data shuffling between the physical machines. This requires high-speed data connections which may translate into time and monetary costs.

2.9 Scalability

A system’s scalability is measured by its ability to handle increasing amount of work without failure. Additionally, the system must be able to make use of any added resources it gets and put them to optimal use.

A modern-day analysis system must be scalable and handle terabytes or petabytes of data. Since RAM is usually not large enough to hold all of the data in memory physical disks are utilized. Distributed storage systems like HDFS were invented to store files as large as the physical storage units. Frameworks like MapReduce and Spark provide a common framework for developing distributed programs without the users worrying about the complex operations or low-level data communications and concurrency controls.

2.10 Reliability

Hardware reliability has come a long way since the early days of computing. Although possible, hardware failures are rare and their impact is further reduced by techniques like uninterrupted power supplies, load-balancing, disk redundancy, server cloning . . .

While a reliable analytic system requires reliable hardware, it should be able to recover from hardware and software faults without losing previous computations. An error or a crash should not require the complete restart of the job. Some of the techniques used include writing intermediate
results to permanent storage media or taking periodic backups. These techniques can be used to automatically restart the task from the point of failure. Spark provides fault-tolerance through its RDD technology which internally builds a lineage graph\(^6\). Spark-based systems automatically inherit and leverage this technology for reliability.

3 Spatial Data Analysis Frameworks

Finding a single machine that is able to process today’s large datasets is challenging; finding one to process large dataset in a timely manner is near impossible. This is because the processing performance is directly proportional to the available CPU, memory, and network resources. A single machine can only be scaled up so much before scaling out becomes necessary. Distributed computing systems were developed in order to speedup the computation process across independent machines. The idea is simple; instead of one machine processing the data in sequence, multiple machines work in parallel with each processing a small portion of the dataset.

Apache Hadoop and Apache Spark are two of the most widely used distributed computing frameworks. Both rely on Apache’s Hadoop Distributed File System[4] (HDFS) and offer operations that abstract the complex and error-prone procedures of low-level data communications and concurrency controls. They are data-neutral and allow users to write customized tasks that automatically distribute the workload across multiple machines. These machines (called processing nodes) are managed by one master node which decides how the workload is distributed and keeps track of the nodes’ progress.

Both, Hadoop and Spark, are suitable for most types of datasets since they allow users to write custom code for their datasets and operations. However, this presents a problem for spatial datasets since the processing framework needs to recognize the spatial object’s shape in order to process it. To that effect, spatial frameworks were developed to utilize Hadoop or Spark to allow for efficient spatial processing. From an end user’s viewpoint, all frameworks perform the same task; they take in as input one or more datasets, perform a specific spatial operation, and finally produce

\(^{6}\)A lineage graph is a Directed Acyclic Graph (DAG) that shows the different phases of RDD transformations from the start RDD to the end RDD
the results. Users will differentiate the frameworks by speed, accuracy, and supported objects and operations.

The differences between the spatial frameworks are due to a number of reasons. Each framework implements its own techniques which may be an improvement of another framework or simply introduce new ones. Some integrate better with the underlying structures by working directly with the framework’s core API; others will simply build on top of the framework and avoid the core. Support of objects and operations is also subjective and may be due to limitations with the underlying algorithm being implemented or simply because the researchers wanted to target a specific problem.

3.1 Hadoop-Based Frameworks

In 2003, Google published a paper that details a new proprietary distributed file system called Google File System (GFS)[41]. It had several advantages, but mainly it was able to store and retrieve truly large files quickly and safely. Files in GFS are split into segments of 64 megabytes and replicated across different servers. By doing so, it eliminated single-point-failures and provided high availability and scalability. Additionally, GFS does not require specialized hardware and can run on inexpensive commodities hardware which makes it extremely attractive. In 2006, Yahoo engineers were able to implement their own version of GFS called Hadoop Distributed File System (HDFS)[53]. The project was then donated to the Apache Software Foundation who is now in charge of maintaining it[4].

A viable large-file storage solution is incomplete without an effective way to process files. For GFS, Google developed a data processing model called MapReduce[36]. Apache followed in their footsteps and created their own, but similar, version of MapReduce\(^7\). The idea of MapReduce is to take the program to the data instead of the traditional way of bringing the data to the program. Such a model sparked the development of many fast and parallel data processing techniques.

Ever since its release, Hadoop has proven to be an excellent system for processing big datasets regardless of the dataset’s type. The range of applications that utilize Hadoop are many and include

\(^7\)HDFS and MapReduce are usually referred to as just Hadoop
machine learning\cite{47}, sorting of terabyte datasets\cite{47}, stock market data analysis and prediction\cite{32}, and big data analysis \cite{48}. Naturally, spatial data was no exception, and Hadoop can be used to process them. However, Hadoop does not recognize spatial data; therefore the time it takes it to process spatial data is slower than it should. A better approach is to read spatial data from HDFS and transform them into runtime spatial objects. Afterward, specialized spatial query engines execute parallel techniques against these object to produce the desired results. Frameworks like Esri GIS Tools for Hadoop\cite{9}, Hadoop-GIS\cite{30}, and SpatialHadoop\cite{37} do just that and empower Hadoop to become spatially aware thus improving results and runtimes.

3.1.1 Esri GIS Tools for Hadoop

Esri GIS tools for Hadoop is a set of tools published by Environmental Systems Research Institute (Esri)\cite{8}. One of their most popular products is a software called ArcGIS\cite{5} which is used for working with and creating geographical maps. In order to harness the power of Hadoop, Esri released a set of tools for performing spatial operations on Hadoop and import the results into ArcGIS. They are designed to provide spatial functionality that is OGC compliant similar to those found in geospatial database systems like PostGIS and Oracle Spatial.

The Esri GIS Tools framework consists of three layers(Figure 2). The Esri Geometry API for Java layer allows MapReduce jobs to become spatially aware through defining geometry objects (i.e. Point, Polygon), spatial operations (i.e. intersect, join), and spatial indexing (i.e. QuadTree, HashTable). The Spatial Framework for Hadoop layer consists of a set of Hive User Defined Functions (UDF) that enable users to write spatial queries in HQL\cite{9}. The Geoprocessing Tools for Hadoop layer offers a set of tools for data connectivity between Hadoop and ArcGIS, submit workflow jobs, and convert data to and from JSON\cite{10}. Unlike the previous two layers, the Geoprocessing Tools for Hadoop is implemented in Python rather than Java.

A job in Esri GIS Tools for Hadoop consists of writing SQL-Like queries using HQL. Quires are then translated into spatially-aware MapReduce tasks that extract relevant data. For example,

\footnote{Esri is a software company specializing in Geographic Information System software and services. \url{https://www.esri.com}}

\footnote{Hive Query Language (HQL) is a SQL-like language for Hadoop}

\footnote{JSON: JavaScritp Object Notation \url{https://www.json.org}}
in the case of $k$NN query involving Points and Polygons datasets, a single Map and Reduce jobs locally index the entire Polygon dataset in the memory of the processing nodes and points are then sequentially checked to determine which Polygon they fall within. The reducer can then perform a job like aggregating the number of points within each Polygon. The reducer causes lots of data shuffling to occur as Points get routed to the proper processing node. As with any Hadoop task, the results are finally written back to HDFS. The format of the output is in JSON which makes it easy for ArcGIS to import and process the results.

The Esri GIS Tools for Hadoop can be imported as a library and included in a user’s MapReduce task. However, they were developed to extend the capabilities of ArcGIS. Some of the tasks implemented may or may not employ a global index. For example, the $k$NN query involving Points and Polygons, a global grid is not utilized. In an aggregate hotspot query, a grid global index is used which increases the number of map tasks. Esri GIS Tools for Hadoop is intended for geometry filtering and therefore is unable to support very large datasets. Any dataset that nears a terabyte in size cannot be processed.
3.1.2 Hadoop-GIS

Hadoop-GIS is a framework for processing spatial datasets on Hadoop. It aims to create a fast and scalable framework for processing spatial datasets in a warehousing system that is already running Hadoop. Its architecture (Figure 3) consists of three major layers built on top of Hadoop – Query Language, Query Translation, and Query Engine. The Query Language layer extends the Hadoop Hive language to introduce support for spatial objects and operations. Users are able to write spatial queries directly in Hive which simplified the MapReduce writing process. The Query Translation layer optimizes the Hive code and translates it into proper MapReduce tasks in order to perform the query. Finally, the Real-time Spatial Query Engine (RESQUE) performs tasks like spatial indexing, query execution, and spatial boundary handling. The source code for these layers[10] is a mix of code written in Java, C++, and Python and utilizes the open source libraries LibSpatialIndex[14] and GEOS[8]. Through the use of these libraries, Hadoop-GIS can reuse already existing code written in languages other than Java (C++ and Python) and allows users to run programs written in these languages. Running Hadoop-GIS tasks is requires some preliminary setup where all libraries have to be pre-installed and the proper environment variables setup[11].

Hadoop-GIS works by applying a series of MapReduce jobs with each job starting by reading a file from HDFS and ending by writing results to a new file to HDFS. This is necessary because Hadoop-GIS streams its input data and relies on HDFS/MapReduce which must write intermediate
results to disk. While this feature achieves fault tolerance, it is I/O intensive. Moreover, streaming is not efficient compared to direct HDFS read, but Hadoop-GIS requires it since it uses and allows non-Java tasks.

Hadoop-GIS starts by scanning all records from both datasets and applying any filtering operations. The filtered records from both datasets are then sampled and indexed based on a grid built from the sampled data. The indexes of both datasets are used to build a global index which is then used to partition both datasets. This step places objects into groups (called buckets or tiles [30]). The spatial objects’ MBR are calculated and overlapping MBRs are placed in the same bucket. Each bucket is assigned a unique ID for identification and the final results are written to disk. This step relies on Hadoop streaming which is slower than direct HDFS read. Moreover, results through sampling are useful if the data itself is uniformly distributed, which is hardly the case with spatial data. The assignment of buckets relies on the objects’ MBRs which is not accurate and can produce a large number of false-positives. Duplicates can also arise during this step; Hadoop-GIS remedies this by sorting the final results and filtering out duplicates.

After each object is assigned to a grid ID, Hadoop-GIS shuffles the data such that objects with the same ID are placed on the same partition. This step involves reading the files from the previous step. Since the datasets are not uniformly distributed, Hadoop-GIS tries to lessen the effect of data skew by splitting large partitions into two or more smaller sub-partitions. The overhead associated with this step seriously degrades the performance as it requires reading and writing files from HDFS as well as data shuffling.

Once the data is split into the proper partitions, Hadoop-GIS builds a local R-Tree index on each of the partitions. This index is used to query one dataset against the other which speeds up the query processing step which performed by the query engine (RESQUE). The engine utilizes the GEOS library to compute the actual relationship between the objects (i.e. distance). In order to remove duplicates, a sort process is performed before writing unique results one final time to HDFS.
SpatialHadoop is a spatial data processing framework for Hadoop. It offers a tighter interaction with Hadoop than Hadoop-GIS and Esri GIS Tools for Hadoop via the use of low-level Hadoop APIs. Tasks in SpatialHadoop recognize spatial operations directly and passes operations to the built-in query engine. It’s architecture (Figure 3) consists of four layers – language, storage, MapReduce, and operations. The storage layer provides a mechanism to index input files and writes them back to HDFS. This layer is I/O intensive but necessary in order to persist results. The MapReduce layer extends Hadoop’s MapReduce by adding two new components (SpatialFileSplitter and SpatialRecordReader) to allow for distributed spatial query processing. The operations layer introduces a number of spatial operations (i.e. Range, kNN, Join) and a number of spatial objects (i.e. Point, Rectangle, Polygon). This is the layer that executes steps for performing the specified query. Finally, the language layer (called Pigeon) extends Hadoop’s Pig Latin language – A SQL-like high-level language intended to simplify MapReduce programming in Hadoop. Pigeon introduces new constructs through a set of user-defined functions that create the spatial types and operations. The addition of the Pigeon language will require users to have a good understanding of Hadoop and Pig Latin programming before learning the new constructs.

SpatialHadoop relies on configuration files and comes pre-configured to run with any spatial dataset.
on all versions of Hadoop[13]. While common operations are supported, users may wish to change the configuration files and fine-tune the framework depending on the task at hand. This, again, requires good Hadoop experience and knowledge of spatial data programming. It is also tedious if the configuration needs to change depending on the task at hand. For example, sample ratio is controlled by the configuration `spatialHadoop.storage.SampleRatio` with default 0.01. R-Tree indexing is controlled by the configuration `spatialHadoop.storage.RTreeBuildMode` which has two options, fast which requires more memory but less time and light which uses less memory with more time.

SpatialHadoop starts by building a partitioning scheme that takes into consideration the HDFS block size (64MB), the proximity of spatial objects, and the number of objects in each partition. This step will ensure that nearby spatial objects are assigned to the same partition. To avoid large indexes, SpatialHadoop only uses a sample of both datasets. Results are written to HDFS until they are read again for the next phase. After the data is partitioning, a local index is built for each partition. Because of the previous step, the size of the local index will not exceed 64MB and hence will be treated by HDFS as a single block when written to HDFS. If the size is less than 64MB, SpatialHadoop will pad the block with 0s to fill the entire block. After the local indexes are built and written to HDFS, the global index is built by merging all files into one single file using HDFS’s `concat` command. The global index file is then loaded into the Master node’s main memory where it will be utilized to index the spatial data blocks using their MBR. The partitioning scheme that is followed here relies heavily on HDFS for data persistence. However, this degrades the performance since disk IOs are expensive and if the input files change, the indexing will no longer be valid.

After the data is correctly partitions, SpatialHadoop follows a similar approach to that of Hadoop-GIS. A local index is built on each partition and then queried in order to discern an initial relationship between the objects. Finally, the spatial library JTS[15] is used to compute the actual relationship between the objects. Duplicate may arise due to objects overlapping multiple grid cells. To remedy this, SpatialHadoop runs a duplicate avoidance technique which requires the computation of the intersection between the resulting record and the query area. Records are added to the final result only if the top-left corner of the intersection is inside the partition boundaries.
3.1.4 Features and Performance Summary

The major aim of the previously mentioned frameworks is to provide spatial support on Hadoop. Their approaches differ in ways like the objects and operations they support, techniques they use, the underlying languages, required expertise level . . . . Table 1 shows a high-level summary of these features; however, all of these frameworks suffer from the same drawback of relying on HDFS for fault tolerance.

A number of experiments were done to compare these in terms of speed and scalability. In [60], a number of spatial datasets were used to gauge the performance of Hadoop-GIS and SpatialHadoop with a maximum sized dataset of 6.9GB. Hadoop-GIS was not able to process this dataset, but SpatialHadoop succeeded. In the same experiment, the authors reduced the size of the dataset to 1/12 the size in order to gauge Hadoop-GIS’s performance. In this test, SpatialHadoop proved that it can outperform Hadoop-GIS. The authors concluded that the problem is due to Hadoop-GIS’s intensive I/O, streaming approach, and use of the GEOS library.

A more detailed experiment was done in [58] which compared a number of non-Hadoop based frameworks along with Hadoop-GIS and SpatialHadoop. The experiments showed that SpatialHadoop is better compared to Hadoop-GIS. The first experiment focused on the index construction time of the frameworks and showed that SpatialHadoop is faster than Hadoop-GIS using a dataset of 4.4 billion records. A second experiment compared the local index sizes and showed that SpatialHadoop requires slightly less memory than Hadoop-GIS. However, Hadoop-GIS uses slightly less memory for its global index. Another two experiments focused on throughput and latency when performing Range and kNN queries. Both frameworks produced results that are close to one another in the Range test, but Hadoop-GIS failed the kNN test. The final experiment tested the Join operation, and the results showed SpatialHadoop to be the better framework with Hadoop-GIS failing to complete the operation.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Esri GIS Tools</th>
<th>Hadoop-GIS</th>
<th>SpatialHadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release Date</td>
<td>2013</td>
<td>2013</td>
<td>2014</td>
</tr>
<tr>
<td>Last Update</td>
<td>2018</td>
<td>2012</td>
<td>2018</td>
</tr>
<tr>
<td>Integration Approach</td>
<td>On Top of Hadoop</td>
<td>On Top of Hadoop</td>
<td>Into Hadoop</td>
</tr>
<tr>
<td>Language Integration</td>
<td>HiveQL</td>
<td>HiveQL</td>
<td>Pigeon (Pig Latin)</td>
</tr>
<tr>
<td>OGC Compliant</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Geometry Library</td>
<td>Esri Geometry API</td>
<td>GEOS</td>
<td>JTS</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global Indexing</td>
<td>Grid (Partial)</td>
<td>Grid</td>
<td>Grid, R-Tree</td>
</tr>
<tr>
<td>Local Indexing</td>
<td>PMR Quadtree</td>
<td>R-Tree</td>
<td>R-Tree or R+-Tree</td>
</tr>
<tr>
<td>Index Persistence</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Data Pruning</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Carry non-spatial data</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Skew Handling Level</td>
<td>Partition</td>
<td>Partition</td>
<td>Partition</td>
</tr>
<tr>
<td>Mixed object Query</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Base Code</td>
<td>Java, Python (non-spatial tools)</td>
<td>Java, Python, and C++</td>
<td>Java</td>
</tr>
<tr>
<td>Allows user-defined opera-</td>
<td>No, (base-code modifications required)</td>
<td>No, (base-code modifications required)</td>
<td>No, (base-code modifications required)</td>
</tr>
<tr>
<td>tions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Installation</td>
<td>None</td>
<td>GEOS, libspatialindex</td>
<td>None</td>
</tr>
<tr>
<td>Configuration</td>
<td>No</td>
<td>System environment variables and Configuration Files</td>
<td>Configuration Files</td>
</tr>
<tr>
<td>Required Expertise Level</td>
<td>Regular Hive</td>
<td>Regular Hadoop</td>
<td>Advanced Hadoop</td>
</tr>
<tr>
<td>Spatial Objects</td>
<td>Point, Polygon, Line, Envelope (MBR)</td>
<td>Point, Box, Polygon</td>
<td>Point, Circle, Rectangle, LineString</td>
</tr>
<tr>
<td>Spatial Operations</td>
<td>Range, $k$NN, Join</td>
<td>Range, $k$NN, Join</td>
<td>Range, $k$NN, Join</td>
</tr>
</tbody>
</table>

Table 1: Feature comparison of Hadoop-based frameworks

3.2 Spark-Based Frameworks

Apache Hadoop gained considerable attention from users and researchers and became one of the most popular distributed processing frameworks for large datasets. However, in 2013 this attention began to shift when Apache released the first version of Apache Spark (Spark). Spark is compatible with Hadoop but, more importantly, it solves two major drawbacks in Hadoop: (1) the need for intermediate data writes to HDFS between tasks to achieve fault tolerance and (2) in-memory data processing which was limited in Hadoop.

At the core of Spark is a technology called Resilient Distributed Dataset (RDD)[62, 19]. RDDs are read-only collections of data that are distributed across different computing nodes. RDDs live in the memory of processing nodes in a parallel computing cluster. Each is processed indepen-
dently with the possibility of moving data into and out of the nodes. There are two groups of operations on RDDs; transformations and actions. Transformations (i.e. map, filter, union) are lazy operations which are not executed immediately; once executed they transform the RDD into a new RDD. Actions (i.e. foreach, count, reduce), on the other hand, are operations that trigger the transformations. Spark achieves fault-tolerance through building a lineage graph.

Spark is written using the Scala functional programming language which is ideal for parallel programming. This, along with the previously mentioned features, made Spark one of the most popular big data processing frameworks. Naturally, spatial data processing is one of the areas that become interested in Spark. Similar to Hadoop, Spark is a generic framework that leaves the specific operations details to the user. It offers a safe and convenient way to parallelize programs across processing nodes without having to worry about low-level communication, concurrency control, or fault tolerance. Spatial operations like join, union, and even kNN can be performed on Spark as-is; however, results are achieved at a considerable resource and time overheads. Therefore, specialized frameworks were developed to run on top of Spark to make it spatially aware and ultimately achieve quicker and more accurate results.

3.2.1 SpatialSpark

SpatialSpark is one of the earliest works on spatial data processing frameworks to take advantage of Spark’s in-memory processing. It is written in Scala and released in 2015 with the aim of providing spatial operations for running on Apache Spark by performing in-memory operations. Its current code release shows that it is able to perform spatial join queries on two datasets. SpatialSpark has two modes of spatial join operations; broadcast and partitioned spatial join. Broadcast spatial join is ideal for use with one small dataset (i.e. city or county boundaries) and one large dataset (i.e, geo-tagged tweets). In a partitioned spatial join, two large datasets are partitioned and individually processed across available computing nodes.

SpatialSpark starts by sampling one of its datasets and computes the MBR of each partition. The MBRs are used to build a global spatial index which assigns each partition an ID and is then broadcasted to all processing nodes. Once the index is broadcasted, partitions will query the index
for each spatial object in order to determine which partition the object should be sent to. The
global index can be written to HDFS in order to be used in subsequent tasks. SpatialSpark uses
the `groupByKey` transformation to group objects with the same partition ID on the same partition.
Then the `join` method is used to join both datasets together. Overall, this process has a number
of drawbacks. First, similar to the Hadoop frameworks, the sampling of the dataset is only useful
in the rare case of uniform data distribution. Second, the broadcast method adds a networking
overhead which may increase processing time if the sampled dataset is large. Third, it is memory
intensive especially because of the use of broadcast and `groupByKey`. These operations require
that the data be saved in memory and if the memory is not large enough to hold the indexes and
objects, SpatialSpark will fail.

After the datasets are partitioned, the local join process matches objects from both datasets.
Initially, this step relies on the objects’ MBRs but can utilize the JTS library to compute an
accurate relationship between the objects (i.e. Euclidean distance). Depending on the user’s
choice, SpatialSpark can build a local index before performing this computation. Overall, this
step is fairly quick especially when objects are matched by their MBRs. Moreover, due to the
sampling technique, some partitions might become overloaded more than others which will increase
the processing time.

3.2.2 GeoSpark

GeoSpark[61] is a cluster computing Spark framework written in Java for processing large spatial
datasets. GeoSpark’s architecture (Figure 5) consists of two layers built on top of Spark, Spatial
RDD (SRDD) and Spatial Query Processing. The Spark layer remains unmodified and no instal-
lion is required as tasks use GeoSpark by including it as a library. The SRDD layer extends
Spark’s RDD class to enable RDDs to support spatial objects (Point, Polygon, Circle, Line, Rect-
gle) and spatial operations (Join, kNN, Range). The spatial query processing layer carries the
task of performing spatial queries against data in SRDDs.

GeoSpark starts by creating SRDDs for the input datasets automatically or via custom procedures.
For automatic SRDD creation, GeoSpark parses and builds spatial objects from input files if they
are in a recognizable format (i.e. CSV\textsuperscript{11}, WKT\textsuperscript{12}, GeoJSON\textsuperscript{13}). Alternatively, the user can parse their input data, build the spatial objects, and then construct the SRDDs. Automatic SRDD creation might seem useful at first, but, in fact, it is much more restrictive. For example, a file that is in CSV format must conform to a specific CSV style; namely, each row should be the spatial object’s coordinates without any non-spatial data. In essence, this feature seems to enforce structure on spatial data which is mostly not structured.

Once the SRDDs are built, GeoSpark partitions these SRDDs by building a global grid over the entire dataset. The grid is built by sampling the datasets and computing the MBR for the entire sample. Then the MBR is partitioning such that each box has a unique ID and contains about the same number of spatial objects. Then, GeoSpark examines objects in both datasets, computes its MBR, and assigns it to a specific box in the grid. If an object falls within multiple grid cells, a copy of that object is made and assigned to the overlapping cells. This step is very computing intensive requiring an initial pass over the first dataset in order to sample and build the global grid followed by another pass to assign each object a grid box ID. In addition, this step will generate duplicate objects to account for an object’s MBR spanning multiple grid cells. This increases the required resources (computation, memory, shuffling) by the framework and calls for a filtering process before the final results are produced.

\textsuperscript{11}Comma Separated Value
\textsuperscript{12}Well-Known Text
\textsuperscript{13}A format for encoding a variety of geographic data structures \url{http://geojson.org}
After the objects in the SRDDs are assigned to their perspective grid cells, GeoSpark will examine the objects within these SRDDs in order to decide whether an index is needed. This process is carried out for each of SRDDs such that the index is only built if the cost of building the index (scan time and memory) improves the overall query execution time. While this step is intended to speed up the query, it may, overall, affect the performance of the framework. The decision to index the SRDD requires a partial or full scan of the spatial objects in that SRDD. Due to the nature of Spark, unless data is cached, it will need to be computed the next time it is needed. Therefore, either the memory requirement or time complexity of the task must increase. It does not seem that users of GeoSpark have control over this step other than to specify the type of index that should be used when GeoSpark decides to build the index.

With spatial objects stored in their respective SRDDs (with or without the index), the spatial query processing layer begins executing the required operation. GeoSpark will follow certain steps that depend on the type of the query. For range queries, the query MBR is computed and then broadcasted to all SRDDs to check their spatial objects against that MBR. For join queries, the SRDDs are joined using their grid IDs. Afterward, spatial objects are compared using their own MBRs in order to decide if they overlap. For $k$NN queries, the framework computes the distance between the spatial objects and keeps the best $k$ matches (uses heap-based top-k algorithm). Afterward, different SRDDs from different nodes are grouped and the $k$ overall results are kept. Naturally, the memory and/or read operations requirements will vary depending on the query’s implementation. Overall, GeoSpark seems memory intensive as it caches data that it will need in future steps.

As a final step and right before producing the results, GeoSpark filters out duplicates that were due to the partitioning and query execution steps. This is a necessary step since any duplicate results will affect the accuracy of the results. In order to perform this step properly, the framework incurs additional computing overhead to group, sort, and filter the data. GeoSpark does not perform a finer computation step to compute the actual relation between the objects. Instead, its results rely on the object’s MBR and leaves any further refinements to the user.
3.2.3 LocationSpark

LocationSpark[54] is a Spark framework for processing large spatial datasets. Its architecture (Figure 6) consists of two layers built on top of Spark – Query Scheduler and Query Executor. Both layers are implemented in Scala with the major aim at solving the query skew problem. The Spark layer remains unmodified and no installation is required as tasks use LocationSpark as a library. The query scheduler layer distributes the data across the different computing nodes in a balanced way. The query executor selects the best execution plan based on the type of query required and the index that was used to index the data.

Data must be parsed and loaded into LocationSpark by creating RDDs of spatial objects that it understands. Currently, LocationSpark can support Box and Point spatial objects with kNN join query. The Box object can be used as a generic spatial object capable of representing any spatial object after calculating that object’s MBR. This means that the results of any query are not based on the object’s actual boundaries which produce incomplete results.

With data loaded into RDDs, LocationSpark proceeds to collect random statistical information from each partition using the query type and data points. This information is used to build a global index with equal sized points and identifies potential problematic spots (called hotspots) in the data partitions. Based on these hotspots, a cost-based-model calculates the overhead of repartitioning the hotspot data by reallocating underutilized nodes. The user can specify either
a grid or a region quadtree as the global index type. While fast, the process of building a global index from random data samples suffers from two major drawbacks. First, it has the overhead of having to pass through the dataset (or at least the sample records) or persist the data in RAM for the subsequent pass. Second, random selection results are inherently nondeterministic with each run producing different results.

LocationSpark allows the grid index to be written to disk in order to speedup future operations. With the global index built, LocationSpark partitions the entire dataset equally between the available processing nodes. This step examines each object in the dataset to figure out which processing node it should be redirected to. In order to perform the spatial join queries, LocationSpark duplicates the outer table and sends it across to the processing nodes. It does this assuming that the outer table is smaller and contains the query objects and the inner table is the queried dataset. Because this is a memory and communication intensive step, LocationSpark embeds in the global index a spatial bloom-filter ($sFilter$). The $sFilter$ allows for testing if a point falls within a given spatial range; if it falls outside the query boundaries, it is not duplicated.

Due to these optimizations, LocationSpark requires a large amount of memory to work and store its spatial data and indexes. Therefore, and in order to reduce the memory requirements, LocationSpark monitors access frequencies (time and number of hits) for each of the spatial objects. Objects with low frequencies are serialized from memory to disk. With this step, it is clear that the framework tries to reduce its memory usage, however, it comes at a greater expense since it increases the amount of disk IO which is slower than memory access.

As a final step and right before producing the results, LocationSpark filters out duplicates that were generated due to the global partitioning step. This is a necessary step since any duplicates will affect the accuracy of the results. Additionally, LocationSpark’s output is currently limited to only counting the number of points that fall within a specific Box. By doing so it discards the spatial objects that were loaded into its RDDs and used during the computation.
3.2.4 STARK

STARK[44] is a Spark framework for processing large spatial datasets. It differs from other frameworks in taking into consideration the temporal attribute of spatial data (spatio-temporal framework). It is written using the Scala language and tightly integrates itself with the Spark API such that RDDs are automatically transformed into spatially-aware RDDs. It does this by taking advantage of Scala’s implicit conversions – a technique that allows new methods to be added to existing types.

STARK’s architecture (Figure 7) consists of four layers built on top of Spark – spatial RDD, predicates, distance functions, language. The Spatial RDD layer adds spatial functionality to Spark’s RDDs. At the core of these RDDs is an object called STObject which contains the spatio-temporal information of the spatial object. The time attribute of the object can be left blank and subsequently ignored by STARK’s query. The predicates layer adds a number of predicates (i.e. distance, intersects) to the spatial operations join and filter. The distance functions provide a set of pre-programmed distance functions to be used with the predicate operations. The idea behind this approach is to provide support to data of different coordinate systems (Cartesian and geodetic) which require different distance metrics for accurate computations. The spatial partitioner layer decides on the best way to partition the objects across the different computing nodes. Currently, STARK works with the spatial attribute and ignores the temporal attribute when partitioning or indexing the datasets. Finally, a new language integration called Piglet extends Pig Latin in order to add support to spatial data programming. Piglet adds a new geometry data type and new filter, join, and indexing operators.
A job in STARK starts by accepting a RDD of type \((STObject, Object)\). The first element \((STObject)\) holds the spatio-temporal information and the second element \((Object)\) can be set to any type and will only be carried through the computation steps. The RDD must be in this form in order for STARK to work since Scala’s implicit conversion will not recognize the spatial RDD as such. With the Spatial RDDs built, STARK builds a global index using the spatial attribute stored in \(STObject\) in order to be able to spatially partition the datasets. STARK offers two types of global indexing; grid which divides the dataset into equally-sized boxes and is not optimized for partition skews, and cost-binary space partitioning (BSP). BSP divides the dataset into boxes of equal number objects thus providing a partition balancing techniques that mitigate partition skews. The user can select which technique to use, but that would require the user to have density knowledge about the data.

STARK tries not duplicate objects that span multiple partitions. If an object like a Polygon breaks off into another partition(s), it is assigned to the partition where its centroid falls, and then the object is virtually pruned. To compensate for this, STARK will record the extent of a partition and use it in the query execution phase. By doing so, additional memory and processing time are required to store and compute the extent of the partition with every newly added object. Additionally, this process will grow the size of the partition which depending on the objects assigned to it, may cause its size to grow to the point where it must be split.

After the data is partitioned according to the global index, STARK performs the query operations on each partition. The user can choose to index the spatial data on each partition using an R-Tree. STARK recommends running the query without in-memory indexing if the cost of building an querying the index exceeds that of querying all items. If an R-Tree is used, then the R-Tree is queried and an initial relationship between the objects is derived. Since these results are based on the object’s MBR, the results are refined further. STARK does this automatically on the local partitions by computing the actual relationship between the objects (i.e. distance).
3.2.5 Simba

Simba[58] is a Spark framework for large spatial data analysis. Its aim is to introduce a framework with a simple programming interface, low latency, high throughput, and scalability. Unlike the previously mentioned frameworks, Simba does not integrate directly with Spark’s RDDs as it is built to work with Spark DataFrame[22] and Spark SQL[31]. Currently, Simba only supports spatial operations over point and rectangular objects. Its architecture[58] (Figure 8) consists of a number of components to provide native spatial operations – SQL Parser, Spatial Operations, Query Optimizer, and Index Manager.

Simba’s SQL Parser layer allows users to run spatial queries using SQL-like statements by adding support to spatial keywords and grammar to Spark SQL (i.e. Point, Polygon, Range, kNN, Join). A similar process adds grammar support to the DataFrame API. The Index Manager layer provides the necessary utilities for users to build global and local indexes like R-Tree, HashMap, and TreeMap. These indexes can be built and dropped anytime using the provided abstraction IndexRDD and can be written to disk in order to speed up future operations. The Spatial Operations layer implements a number of spatial operations over point and rectangular objects. The Query Optimizer layer extends Spark SQL Catalyst optimizer in order to provide a Cost-Based Optimization (CBO) techniques for optimizing complex spatial queries.

Simba tasks start with a relation either from an abstract syntax tree returned by the SQL parser or the DataFrame API. Relation’s attributes that have not been matched with a type or an input table are assigned a type using the Catalyst and a Catalog object which tracks tables in all data sources. Afterward, the logical optimizer produces an optimized logical plan through standard rule-based
optimization like constant folding, predicate pushdown, spatial distance pruning. The logical plan is then optimized via non-spatial rules (constant folding, predicate pushdown) and spatial rules (distance pruning). The optimized logical plan is then turned into a one or more physical plans based on criteria like spatial operation support and physical operators inherited from Spark SQL. In the case of multiple physical plans, CBO is applied taking into consideration the choice of indexes and random dataset statistics collected from Spark’s CacheManager and Simba’s Index Manager. The optimal plan is then selected, however, since this step relies on random data samples, it is possible that the execution plan could change when the same task is executed again. Finally, the selected physical plan is transformed into an RDD object which is treated as a table with objects in the RDD as the rows. The RDD can be written to HDFS and reused again to skip this process in subsequent runs.

Simba utilizes the 2-phase indexing approach, global and local. Datasets are treated as tables with records represented as Row objects; a table is then basically an RDD of type Row. To index a table, Row objects within an RDD are packed into an array which, also, makes sampling quick. This undoubtedly increases the memory requirements and introduces and overhead, but Simba states that their experiments show the overhead to be negligible. Initially, Simba partitions tables such that close by objects are assigned to the same partitions while balancing the load across all partitions. Afterward, each partition builds a local index (i.e. R-Tree), loads all rows into an array, collects statistics, calculates the partitions’ MBRs, and computes the number of records. Finally, the global index is built by having each partition report its statistics back to the master node which will build the index (R-tree or Grid). The global index is kept in the memory of the master node and is used to prune irrelevant partitions for an input query. As an added feature, the global index can be written to disk and loaded directly for future tasks.

Spatial queries execution in Simba is type dependent and utilizes the global and local indexes. kNN queries utilize the global index to prune irrelevant partitions and the local index to improve performance. A circle is drawn around the point and the global index is used to select the best partitions that cover at least the required $k$ within that partitions MBR. Candidates are selected on each partition after calculating the actual distance; results from all the partitions are then
combined on the master node and the top best \( k \) are returned. For distance join queries, Simba uses the global index to get an initial approximation of how to join the two datasets. The results of this step is a set of possible pairs \((i, j)\) which may contribute to the solution with each pair assigned a partition ID. Then, pairs with similar partition IDs are sent to the same processing node where the precise distance between the points is calculated. \( k \)NN joins are implemented using three different approaches. The baseline method is the simplest and the least efficient as it uses the block nested loop \( k \)NN join in Spark. The Voronoi \( k \)NN Join and z-Value \( k \)NN Join method is faster than the baseline method but produces approximate results. The R-Tree \( k \)NN Join method provided faster and better results. It partitions a dataset into \( n \) partitions using a Sort-Tile-Recursive algorithm for load balancing and preserving locality. Then a distance bound is calculated for each partition in order to derive a subset of the results. The distance is calculated by finding the furthest point from the center of each partition’s MBR; the results are then sent back to the master node and finally utilize an R-Tree to find a subset of the results on each partition. Finally, Spark’s \texttt{zipPartitions} is invoked, a new R-Tree is built, a local \( k \)NN is executed, and the union of the results produces the query’s output.

3.2.6 Features and Performance Summary

Much like the Hadoop-based frameworks, Spark-based frameworks aim at simplifying and speeding up the processing of spatial data. Different from the Hadoop frameworks, Spark frameworks rely on in-memory data processing first (RDD) then HDFS. The techniques these frameworks use, language features they offer, and operations they support are all directly affected by the underlying Spark system. Table 2 shows a high-level summary of the frameworks’ features.

Each of these frameworks discussed why its technique(s) are better than those of the ones that came before it. These discussions were then backed by experiments that used one or more large spatial datasets. In [44], the STARK framework is compared to GeoSpark and SpatialSpark using a dataset containing 50 million polygons. The experiment put the frameworks under different tests to examine the different indexing modes. Results showed that STARK performs better when used with live indexing. SpatialSpark was reported to be limited in its functionality since it only supports
a limited number of operations without an index (contains, within distance). GeoSpark was also problematic in the sense that it was not able to process the entire dataset. This was attributed to the excessive caching of data that its algorithm follows.

In [58], a number of experiments were done to compare Simba, GeoSpark, and SpatialSpark. The experiments used three datasets with varying sizes to compare the time and memory costs of building the indexes (local, global), throughput, and latency. For the cost of building the indexes, the experiment used a dataset of 1 billion records. The results showed the geopark’s indexing is slightly faster than all others but only because it relied on a sample of the dataset and skipped the global index. Simba was close behind followed by SpatialSpark. The experiment also tested Simba’s cost of multidimensional indexing and found that the time increases linearly as the number of dimensions increased from 2 to 6.

In the next experiment, the frameworks’ RAM requirements of the indexes were measured for varying data sizes. The experiment showed that most of the memory consumed by the local indexes across the different processing nodes. SpatialSpark’s global and local indexes were slightly better than Simba’s. GeoSpark’s local index consumed the most memory out of all frameworks tested.

The throughput and latency experiment used 500 million records to test range and kNN queries on a number of frameworks. Simba finished its operations in far less time than the others. Simba’s throughput was better, followed by SpatialSpark, followed by GeoSpark. Latency results were similar with Simba requiring less time than SpatialSpark and GeoSpark. The experiment also tested Simba’s cost of multidimensional indexing and found that throughput decreases and latency increases as the number of dimensions increased from 2 to 6 for both query types.
<table>
<thead>
<tr>
<th>Feature</th>
<th>SpatialSpark</th>
<th>GeoSpark</th>
<th>LocationSpark</th>
<th>STARK</th>
<th>Simba</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Update</td>
<td>2017</td>
<td>2018</td>
<td>2017</td>
<td>2018</td>
<td>2018</td>
</tr>
<tr>
<td>Integration Approach</td>
<td>On Top of</td>
<td>On Top of</td>
<td>Into Spark</td>
<td>Into Spark</td>
<td>Into Spark</td>
</tr>
<tr>
<td></td>
<td>Spark</td>
<td>Spark</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Language Integration</td>
<td>-</td>
<td>-</td>
<td>Scala (Spatial RDD)</td>
<td>Piglet and Scala via RDD integration</td>
<td>DataFrame and Spark SQL</td>
</tr>
<tr>
<td>OGC Compliant</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Geometry Library</td>
<td>JTS</td>
<td>JTSPlus</td>
<td>JTS</td>
<td>JTS</td>
<td>Built-in</td>
</tr>
<tr>
<td>Global Indexing</td>
<td>Grid, K-D</td>
<td>Grid</td>
<td>Grid, Quad-Tree</td>
<td>Grid, Cost-Based Binary Space, R-Tree</td>
<td>Grid, R-Tree, KD-tree</td>
</tr>
<tr>
<td>Options</td>
<td>None, R-Tree</td>
<td>None, R-Tree, Quad-Tree</td>
<td>None, Grid, R-tree, Quad-Tree, IR-tree.</td>
<td>None, R-Tree</td>
<td>HashMaps, TreeMaps, R-Tree</td>
</tr>
<tr>
<td>Index Persistence</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Data Pruning</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Carry non-spatial data</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Skew Handling Level</td>
<td>-</td>
<td>Partition</td>
<td>Query</td>
<td>Partition</td>
<td>Partition</td>
</tr>
<tr>
<td>Mixed object Query</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Base Code</td>
<td>Scala</td>
<td>Java</td>
<td>Scala</td>
<td>Scala</td>
<td>Scala</td>
</tr>
<tr>
<td>Allow for new operations</td>
<td>No</td>
<td>Yes (SRDD and query processing layers)</td>
<td>Yes (Spatial Objects, Query Scheduler, Query Executor)</td>
<td>Yes (Spatial-RDD Functions and Piglet)</td>
<td>Yes (Requires source code modification)</td>
</tr>
<tr>
<td>Installation</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Configuration</td>
<td>In code</td>
<td>In code</td>
<td>In code</td>
<td>In code</td>
<td>In code</td>
</tr>
<tr>
<td>Required Expertise Level</td>
<td>Moderate</td>
<td>Regular Spark</td>
<td>Regular Spark</td>
<td>Regular Spark</td>
<td>Regular Spark DataFrame</td>
</tr>
<tr>
<td>Spatial Objects</td>
<td>MBR</td>
<td>Circle, LineString, Point, Polygon, and Rectangle</td>
<td>Box and Point</td>
<td>Inherited from JTS</td>
<td>Circle, MBR, Point, Polygon</td>
</tr>
<tr>
<td>Spatial Operations</td>
<td>Range, Join, kNN</td>
<td>Range, Join, kNN</td>
<td>Range, kNN, Join, kNN-join</td>
<td>Filtering, Join, kNN, Clustering</td>
<td>Range, Join, kNN</td>
</tr>
</tbody>
</table>

Table 2: Feature comparison of Spark-based frameworks
4 Future Work

The field of spatial data analysis is rapidly changing. The various frameworks discussed here aim at simplifying the analysis process with each framework claiming that its approach is better for spatial data processing. However, it is unclear how these frameworks compare to one another under similar tests. It would be interesting to put all of the frameworks to test under the same conditions using the same cluster center.

The various experiments reported in the test sections of the frameworks use different dataset sizes and environment configurations. For future work, we would like to apply the same dataset to all of the frameworks and observe their usability, runtimes, and behaviors. While they all claim that they support truly large datasets, an exact definition is not given. For instance, in [60] the experiments use workstations of 10 nodes with 15 gigabytes of memory and a number of datasets with the largest being 23.8 gigabytes. In [44], experiments are performed using 16 nodes with 16 gigabytes of memory and a dataset containing 34 million entries.

In addition, we would like to put these frameworks under different scalability tests. First, the number of nodes is fixed while we vary the size of the input datasets. Second, the size of the datasets is fixed while the number of nodes is linearly increased. Such tests would give an indication of the frameworks’ scalability. Tests can, also, focus on the frameworks’ performance when using different indexes similar to those reported in [58]. If the framework offers index caching, a number of tests can gauge the performance variations when the framework skips the indexing step.

Some frameworks offer batch and/or live processing which would be worth investigating. Since both approaches are different, a framework that is better at live queries may not perform as well for batch queries. In addition, the usability factor is important as it indicates how easy it is to launch and perform multiple queries.

Finally, none of the experiments that we have studied showed how accurate are their results. The framework’s performance is only as reliable as its results. As an accuracy measure, we would like to compare each of the frameworks’ results to those obtained via a traditional naive approach. Such
a result can be obtained from running non-optimized queries and simply focus on pairing objects together for 100% accuracy. Moreover, the examination should look at how much of the input data makes it to the output. For example, does the framework drop objects that are unmatched and/or does it maintain the object’s boundaries by not just computing and working with its MBR.

5 Summary

In this paper, we surveyed a number of frameworks that make Apache Hadoop and Apache Spark spatially aware. Spatial data analysis is a field that has recently picked up new momentum due to the recent explosion of the amount of spatial data being recorded. The release of new parallel execution frameworks has remotivated researchers to produce spatial analysis systems that are fast, accurate, reliable, and scalable. This task is not trivial due to a number of challenges like the need to recognize the many types of spatial objects, support of large number of operations, different shapes a dataset can take, indexing, multidimensional objects, scalability, and reliability.

Apache Hadoop and Spark are two of the most popular big data processing frameworks; their underlying structure is open-sourced, easy to use, scalable, and shields the user from much of the worrying of parallel programming. Because of that, they are generic data-processing frameworks and are not suitable for fast spatial data processing. To that effect, specialized frameworks have been developed to make them spatially aware. Each of these frameworks offers its own features that vary in usability, objects, operations, indexing . . . (Tables 1 and 2).
Bibliography


