Data Science and Engineering
https://doi.org/10.1007/s41019-020-00147-9

How Good Are Modern Spatial Libraries?

Varun Pandey[1] · Alexander van Renen[1] · Andreas Kipf[2] · Alfons Kemper[1]

Received: 17 June 2020 / Revised: 1 October 2020 / Accepted: 16 October 2020
© The Author(s) 2020

Abstract
Many applications today like Uber, Yelp, Tinder, etc. rely on spatial data or locations from its users. These applications and services either build their own spatial data management systems or rely on existing solutions. JTS Topology Suite (JTS), its C++ port GEOS, Google S2, ESRI Geometry API, and Java Spatial Index (JSI) are some of the spatial processing libraries that these systems build upon. These applications and services depend on indexing capabilities available in these libraries for high-performance spatial query processing. In this work, we compare these libraries qualitatively and quantitatively based on four different spatial queries using two real world datasets. We also compare these libraries with an open-source implementation of the Vantage Point Tree—an index structure that has been well studied in image retrieval and nearest-neighbor search algorithms for high-dimensional data. We found that Vantage Point Trees are very competitive and even outperform the aforementioned libraries in two queries.

Keywords Spatial · Spatial libraries · Spatial data management

1 Introduction

In recent years, services such as recommending close-by social events, businesses, or restaurants as well as navigation, location-based mobile advertising, and social media platforms have fueled an exponential growth in location-enabled data. Industry giants like Google, Facebook, Uber, Foursquare, and Yelp are some of the various companies that provide such services. In order to handle location data from their users, these companies either build their own spatial data management systems from scratch, or rely on existing solutions.

The unprecedented rise of location-based services has led to a considerable amount of research efforts that have been focused on four broad areas; (1) systems that scale out [2–4, 9, 10, 17, 58, 59, 61, 69, 71, 72], (2) support for spatial processing in databases [14, 32, 35, 38, 41], (3) improving spatial query processing [12, 22–26, 42, 43, 46, 49, 62–64, 74], and (4) leveraging modern hardware and compiling techniques [6, 7, 27, 54–56, 73], to handle the increasing demands of applications today.

Some of the most popular spatial libraries are: JTS Topology Suite (JTS), its C++ port Geometry Engine Open Source (GEOS), Google S2 (S2), ESRI Geometry API, and Java Spatial Index (JSI). Today, these libraries are being used in a variety of services and research projects alike. We highlight the major services and research projects that use these libraries in Sect. 4. Many of the services that use these libraries are multi-million dollar business models, such as on-demand ride-hailing and dating applications. Moreover, many research efforts today in the systems community also use these libraries for their spatial-processing capabilities. Given how prevalent and relevant these libraries are in present-day services and systems, we believe it becomes a necessity to evaluate these libraries.

In this work, we extend the previous work done in [42]. We take an application-oriented approach in evaluating these libraries. Many open datasets such as Open Street Maps or NYC taxi rides datasets provide location information using **raw GPS coordinates**. Moreover, millions of GPS devices in use today send location information in the form of GPS coordinates. Thus, unless stated otherwise, we

✉ Varun Pandey
pandey@in.tum.de

Alexander van Renen
renen@in.tum.de

Andreas Kipf
kipf@mit.edu

Alfons Kemper
kemper@in.tum.de

[1] Technical University of Munich, Munich, Germany

[2] Massachusetts Institute of Technology, Cambridge, MA, USA

assume that applications receive raw GPS coordinates and have to process spatial queries based on them.

With this *Experiment and Analysis* paper we contribute:

- A study of problems arising when using planar geometry libraries directly with GPS coordinates.
- A survey of modern spatial libraries, highlighting their features and indexes.
- A thorough performance analysis of these libraries using four spatial queries: range, distance, *k*-NN, and a spatial join query.

The rest of the paper is structured as follows: Sect. 2 discusses the background for planar and spherical geometry, and identifies potential pitfalls when using these libraries. Section 3 formally defines the spatial queries we used for evaluation and presents practical examples of these queries. Section 4 introduces the aforementioned modern spatial libraries. Section 5 presents the experimental setup used for evaluation, which is followed by the evaluation itself in Sect. 6. In Sect. 7 we highlight a potential research area and discuss how distributed spatial query processing can be implemented using any spatial libraries. Section 8 discusses related work and is followed by takeaways and conclusions in Sect. 9.

2 Background

The libraries evaluated in this paper either use planar or spherical geometry. In this section, we describe what these two terms mean and why a naive usage of planar geometry libraries can introduce unintended errors.

2.1 Geometry Models

Earth can be projected onto many surfaces, but today the most widely adopted surfaces to project Earth on are planes and spheres.

Planar Geometry: is geometry on a plane. The basis of planar geometries is a plane, i.e., all the calculations on the geometries such as distance between geometries, area covered by a geometry, intersection between geometries is done on a plane using cartesian mathematics. In planar geometry, the distance between two points on a plane is a straight line distance between the points.

Spherical Geometry: is geometry on a sphere. The basis of spherical geometries is thus a sphere. On the sphere there are no straight lines as in case of a plane. In spaces involving curvature (such as spheres), straight lines are replaced by geodesics.

The shortest distance between two points on the surface of a sphere is called the great-circle distance or orthodromic distance [67].

To make planar geometries work with geographic data, Earth has to be projected onto a plane. There are multiple projections available, some of which are based on the area that they cover such as city based, region based, country based, and even on continental and global scale but they all come with different trade-offs [36]. Most notably, there is no planar projection that preserves distance. Projections can only *minimize* distance distortion. When working with planar geometries, it thus becomes essential to choose the right projection that is best suited to the application concerned.

Spherical geometries on the other hand work on spherical projections, which maps the points on Earth's surface to a perfect mathematical sphere. As Earth is not a perfect sphere, spherical projections of the Earth also create distortions, but are limited to a maximum distortion of 0.56% [52]. Spherical projections also preserve the correct topology of the Earth with no singularities and low distortions everywhere. An even more accurate projection of Earth is on an ellipsoid, but operations on ellipsoids are orders of magnitude slower than on a sphere. Spherical geometry are also slower than their planar geometry counterparts usually since the computations are on a sphere rather than on a plane. But spherical geometry is generally considered better suited to work with geographic data on a *global* scale.

2.2 When Can Things Go Wrong In Planar Geometries?

In this section, we show how applications can end up using planar geometry libraries in a wrong way. We motivate this by using an illustrative example of a ride-hailing application in two scenarios: operating in a city and on a global scale. We highlight potential pitfalls which can lead to applications getting wrong results.

Consider a ride-hailing application scenario in New York City that stores the location data as raw GPS coordinates (lat/long),[1] and matches riders with the nearest drivers using the *k*-NN query (we formally define *k*-NN query in Sect. 3.3). A part of *k*-NN query processing is the distance computation between two points, the user and the drivers in this case. Planar geometry libraries come with distance functions[2] that compute Euclidean distance. The application could naively compute Euclidean distance between two raw GPS coordinates, in which case, the distance would be in degrees and does not have any meaning. The correct

[1] Many open datasets today provide location information in lat/long format.

[2] JTS/GEOS do not support geodetic operations: https://locationtech.github.io/jts/jts-faq.html#geodetic_operations. ESRI geometry API has geodesic distance function: https://github.com/Esri/geometry-api-java/wiki.

approach is to project the raw GPS coordinates using a spatial reference system, such as EPSG:32118 [11] that *minimizes* the distance distortion for the New York area, and the measurement unit is in meters. The Euclidean distance can then be computed on the projected coordinates using the distance function in the planar geometry library. Another way is to compute the Haversine distance between the GPS coordinates, but it is slower to compute because it involves computing multiple sine and cosine operations.

Now as another example, consider the same application as in the previous example, but the application now operates at a global level and uses a planar geometry library. The application may naively start using EPSG:3857 [51] as the projected coordinate system, which projects the whole Earth onto a plane, and not just a city as in the case with EPSG:32118. In EPSG:3857, distances are only accurate along the equator, and the error increases with gain or loss in latitude. The application receives two ride requests, one in city A which lies on the equator, and the other in city B which is closer to the North (or the South) Pole where distance distortions are large (distances become larger than they actually are). While the distance computation will be correct for city A, for city B the distance distortions will be large. In EPSG:3857 the distance distortion can be significant. So if the application is using planar geometry, or more accurately using Euclidean distance, to compute the distance between the users and the drivers in city B, a user might not be assigned any driver as the application may wrongly interpret that the drivers are far away from the user, while in reality the driver might be parked next to the user. A better approach would be to detect during query processing that the user is in city B, and then transform coordinates into a reference system specific to the city as mentioned in the previous example to compute the distances.

A more **hidden** potential pitfall is while using a spatial index in a planar geometry library. Many popular spatial index structures in these libraries are either designed or implemented with Euclidean distance as a basis for distance computation during various types of index traversals, depending on the query. For example, the R-tree in Java Spatial Index (JSI) assumes Euclidean distance as the metric. So, if an application uses the R-tree to index GPS coordinates and issues a k-NN query to the R-tree, it is bound to get wrong results because the nearest-neighbor search algorithm in the index uses Euclidean distance. Similarly in JTS and GEOS, if a user does not provide a distance metric to the k-NN (or NN) query in the R-tree, the library uses Euclidean distance by default. These problems are further compounded because many other libraries utilize these spatial libraries. As an example, consider the description of STR-Packed R-tree in Shapely,[3] a popular python geospatial library which is used in more than 12 thousand projects on GitHub.[4] The description gives a simple example of R-tree for a nearest-neighbor query. The user might be using GPS coordinates in the R-tree, and might not be aware that the underlying library GEOS uses Euclidean distance as the metric for the nearest-neighbor queries and thus obtain an unintended error. The correct approach for using a spatial index that indexes geodectic coordinates is shown in [48].

3 Queries

In this work we have considered four queries, namely, range, distance, k-nearest neighbor (k-NN) and a spatial point-in-polygon join query. We selected these four queries based on recent research in systems [69] and applications [73]. Simba [69] is a big spatial data analytics system that is optimized for storing location-data and considers (1) range, (2) distance, and (3) k-nearest neighbors Query (k-NN) queries. [73] showcases multiple motivating examples of spatial point-in-polygon join queries which are particularly useful for visual exploration and analysis of urban data.

3.1 Range Query

A range query takes a range r (i.e., min/max values for all dimensions N) and a set of geometric objects S. It returns all objects in S that are contained in the range r. Formally:

$$Range(r, S) = \{\ s | s \in S\ \wedge\ \forall n \in N :\ r[n].\min \le s[n] \le r[n].\max\ \}.$$

Practical Example: Retrieve all objects at current zoom level in a maps application (e.g., Google Maps) for a browser window.

3.2 Distance Query

A distance query takes a query point q, a distance d, and a set of geometric objects S. It returns all objects in S that lie within the distance d of query point q. Formally:

$$Distance(q, d, S) = \{\ s | s \in S\ \wedge\ \text{distance}(q, s) \le d \}.$$

Practical Example: Retrieve all dating profiles within 5 kilometers of a user's location.

[3] https://shapely.readthedocs.io/en/latest/manual.html#str-packed-r-tree.

[4] https://github.com/Toblerity/Shapely/network/dependents?package_id=UGFja2FnZS00OTkzMjI1MA%3D%3D.

Table 1 Selected features of the libraries

Features	S2	GEOS	ESRI	JTS	JSI	jvptree
Language	C++	C++	Java	Java	Java	Java
Indexes	ShapeIndex, PointIndex, RegionTermIndexer	STRtree, Quadtree	Quadtree	STRtree, Quadtree, k-d tree	R-Tree	Vantage Point Tree
Geometry Type	Spherical	Planar	Planar	Planar	Planar	Metric space
Geometry Model	Point, Line, Area, Geometry Collections	Point, Line, Area, Geometry Collections	Point, Line, Area, Geometry Collections	Point, Line, Area, Geometry Collections	Point, Area	Point
License	Apache v2.0	LGPL	Apache v2.0	Dual licence (EPL 1.0, BSD)	LGPL	MIT

Table 2 Selected features of all indexes

	S2	ESRI	JTS			JSI	jvptree
Feature	Point Index	Quadtree	k-d tree	Quadtree	STRtree	R-tree	jvptree
Implemen- tation	Linear Quadtree	Quadtree	k-d tree	MX-CIF Quadtree	STR packed R-tree	R-tree	VPTree
Geometry	Point	Rectangle	Point	Rectangle	Rectangle	Rectangle	Point
Native queries	Range, Distance, k-NN	Range	Range	Range	Range, k-NN	Range, k-NN	Distance, k-NN
Updates	Yes	Yes	Insert:Yes Delete:No	Yes	No insertion after build	Yes	No
Default Fanout	32	4	2	4	10	20–50	2

3.3 *k*-Nearest Neighbors Query

A k-NN query takes a set of points S, a query point q, and an integer $k \geq 1$ as input, and finds the k-nearest points in S to q. Formally:

$$k - NN(q,k,S) = \{\ s | s \in T \subseteq S\ \wedge\ |T| = k\ \wedge\ \forall t \in T,$$
$$\forall r \in S - T\ :\ \text{distance}(q,t) \leq \text{distance}(q,r)\}.$$

Practical Example: Find five closest pizzerias from a user's location.

3.4 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, within, or withindistance) and returns a set of all pairs (r, s) where $r \in R$, $s \in S$, and the join predicate θ is fulfilled. Formally:

$$R \bowtie_{\theta} S = \{(r,s)\ |\ r \in R,\ s \in S,\ \theta(r,s) \text{ holds}\}.$$

Practical Example: Given two datasets, taxi rides (R: points) and neighborhood boundaries (S: polygons), join the two datasets to find how many rides originate (θ: within) from each neighborhood.

4 Libraries

In the following section, we describe the major features of the evaluated libraries. We also highlight the major services, applications, and systems that use these libraries. Table 1 summarizes various features of the libraries, and Table 2 summarizes the features of the indexes found in these libraries.

4.1 ESRI Geometry API

ESRI Geometry API[5] is a planar geometry library written in Java. ESRI Geometry API comes with a rich support for multiple geometry datatypes, such as point, multipoint, line, polyline, polygon, and envelope and OGC variants of these datatypes. It has support for various topological operations, such as cut, difference, intersection, symmetric, union and various relational operations using DE-9IM matrix such as contains, crosses, overlaps etc. ESRI Geometry API also supports a variety of I/O formats, WKT, WKB, GeoJSON, ESRI shape and REST JSON. The geometry library also comes with Quadtree index which cannot be classified into

[5] https://github.com/Esri/geometry-api-java.

a particular type from the Quadtree family. The key property of any Quadtree is its decomposition rule, in ESRI Quadtree, a leaf node splits into four when the node element count reaches 5 elements, and they are pushed to the children quads if possible.

ESRI Geometry API is used in a variety of products by ESRI such as ArcGIS, ESRI GIS tools for Hadoop, and various ArcGIS APIs. It is also used by the Hive UDFs and by developers building geometry functions for third-party applications such as Cassandra, HBase, Storm, and many other Java-based "big data" applications.

4.2 Java Spatial Index

The Java Spatial Index (JSI)[6] is a main-memory optimized implementation of the R-tree [15]. JSI relies heavily on the trove4j[7] library to optimize performance and reduce the memory footprint. The code is open-source, and is released under the GNU Lesser General Public License, version 2.1 or later. The JSI spatial index is limited in features, and only supports a few operations. It is a lightweight R-tree implementation, specifically designed for the following features (in order of importance): fast intersection performance by using only main memory to store entries, low memory footprint, and fast updates. JSI's R-tree implementation avoids creating unnecessary objects by using primitive collections from the trove4j library. JSI only supports rectangle and point datatypes, and has support for only two predicates for refinement, intersects and contains. The R-tree index can be queried natively for ranges and *k*-NN.

We could not find any reference of JSI being used in a major system or service, which we believe is mostly due to its limited capabilities. Although limited in features, JSI is still regularly utilized in diverse research areas [28, 29, 33, 34, 57].

4.3 JTS Topology Suite and Geometry Engine Open Source

The JTS Topology Suite (JTS) is an open-source Java library that provides an object model for planar geometry together with a set of fundamental geometric functions. JTS conforms to the Simple Features Specification for SQL published by the Open GIS Consortium[8]. GEOS (Geometry Engine Open Source)[9] is a C++ port of the JTS Topology Suite (JTS). Both JTS and GEOS provide support for basic spatial datatypes such as points, linestrings and polygons along with indexes such as the STR packed R-tree [30] and MX-CIF Quadtree [31]. They also support a variety of geometry operations such as area, distance between geometries, length/perimeter, spatial predicates, overlay functions, and buffer computations. They also support a number of input/output formats including Well-Known Text (WKT), Well-Known Binary (WKB).

JTS is used in many modern distributed spatial analytics systems such as Hadoop-GIS [2], SpatialHadoop [9], GeoSpark [72], and SpatialSpark [71] and other research areas [55]. GEOS on the other hand is used in a number of database systems and their spatial extensions such as MonetDB, PostGIS, SpatiaLite, Ingres, and it is also used by a number of frameworks, applications, and proprietary packages.

JTS is used in many modern distributed spatial analytics systems such as Hadoop-GIS [2], SpatialHadoop [9], GeoSpark [72] and SpatialSpark [71] and other research areas [55]. GEOS on the other hand is used in a number of database systems and their spatial extensions such as MonetDB, PostGIS, SpatiaLite, Ingres. GeoPandas and Shapely, two popular geospatial libraries in python, internally use GEOS. It is also used by a number of frameworks, applications and proprietary packages.[10]

4.4 Google S2 Geometry

S2[11] is a library that is primarily designed to work with spherical geometry, i.e., shapes drawn on a sphere rather than on a planar 2D map, which makes it especially suitable for working with geographic data. S2 supports a variety of spatial datatypes including points, polylines, and polygons. It also has two index structures, namely (1) S2PointIndex to index collections of points in memory and is a variant of Linear Quadtree [31], and (2) S2ShapeIndex to index arbitrary collections of shapes, i.e., points, polylines and polygons in memory. S2 also defines a number of queries that can be issued against these indexes. Indexes also define iterators to allow for more fine-grained access. S2 also accepts input in lat/long (GPS) format.

In recent years, S2 has become a popular choice among various location-based services. It is used by on-demand ride-hailing services such as Uber [44] and GO-JEK [47]. It is also used by location-sharing applications like Zenly [50] (recently acquired by Snap Inc. [18]) and Foursquare [60]. Even popular games such as Pokémon GO [53], Ingress [1], and a popular location-based dating application Tinder [45] utilize S2. Moreover, S2 is also used by many database systems, including MemSQL [14], MongoDB [35], and HyPer's

6 https://github.com/aled/jsi.

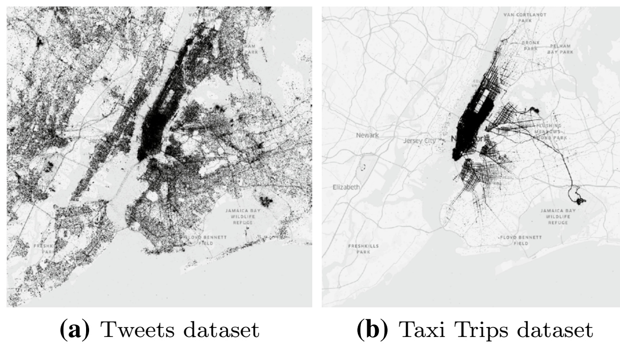
7 http://trove4j.sourceforge.net/html/overview.html.

8 https://www.opengeospatial.org/standards/sfa.

9 https://trac.osgeo.org/geos/.

10 https://trac.osgeo.org/geos/wiki/Applications/.

11 https://github.com/google/s2geometry.



(a) Tweets dataset **(b)** Taxi Trips dataset

Fig. 1 Datasets: NYC Taxi trips are clustered in central New York while Tweets are spread across the city

[21] geospatial extension HyPerSpace [41]. It has also been used in other research areas [22–24, 27, 68].

4.5 Vantage Point Tree

The vantage point tree [70] is based on metric space and has been well studied in image retrieval and nearest-neighbor search algorithms for high-dimensional data. It is a binary tree which is built recursively. At each node in the tree, the points are split into two equal-sized partitions, and are assigned to its two children. This process is repeated until no points are left or a certain threshold is reached. A node partitions its points by picking one point p at random, the vantage point. The points assigned to the node are then are sorted by their distance to the vantage point p. The resulting sorted array is then split in the middle and assigned to the two children. The distance of the split point from the vantage point p serves as the radius r for the node. All the points that are within the radius r (i.e., the left part of the sorted array) are assigned to the left child of the node, and the rest of the points are assigned to the right child. Based on this partitioning, the tree can then be traversed efficiently to answer distance and k-NN queries. We refer readers to [70] for more details on vantage point trees. We use the library jvptree[12] for an implementation of vantage point tree in our experiments.

5 Methodology

To benchmark the various libraries and measure memory costs, we use language specific open-source tools. For Java based libraries, we use the Java Microbenchmark Harness (JMH),[13] which is a framework for building, running, and analyzing benchmarks. To measure the memory consumption in Java, we use the Memory Measurer tool.[14] To benchmark C++ based libraries, we use Google Benchmark,[15] and for memory consumption of the indexes in C++, we use the Heap Profiler in TCMalloc.[16] TCMalloc overrides the `malloc` and `new` implementations, and can thus track the memory usage of an application from the amount of memory allocated/deallocated.

For evaluation, we used two location (points) datasets, the New York City Taxi Rides dataset [37] (NYC Taxi Rides) and geo-tagged tweets in the New York City area (NYC Tweets). NYC Taxi Rides contains 305 million rides from the years 2014 and 2015. NYC Tweets data was collected using Twitter's Developer API [65] and contains 83 million tweets. Figure 1 shows the distribution of the rides and tweets in the NYC region. It can be seen that the Taxi rides are mostly centered around central New York whereas the tweets are well distributed over the entire city.

We further generated query datasets that consist of ranges (bounding boxes) in case of range query, query points and distances in case of distance query, and query points in case of k-NN query. For range queries and distance queries, we created seven different query datasets for seven different selectivities, ranging from 0.0001 to 1% (i.e., the query selects 0.0001–1% of the data). These query datasets consist of one million queries each. We evaluate various indexes in the libraries by issuing these queries sequentially. We chose to generate a large number of queries to minimize the effect of caching tree nodes from a previously issued query. Testing with many queries is especially important in cases with low selectivity where many indexes achieve a throughput of more than 100,000 queries per second. The benchmark frameworks that we use for evaluation run a benchmark multiple number of times until the result is statistically stable. It is thus necessary that we have sufficient queries that do not touch the same nodes in the index structures, but rather exercises several paths in the indexes. To generate these datasets, we uniformly generated points within the New York City bounding box and continuously expanded the range or the distance, depending on which query dataset is being generated, to meet the selectivity requirements. For the k-NN query dataset, we uniformly generated points within the NYC bounding box. For the point-in-polygon spatial join query, we use 289 polygons of neighborhood boundaries in NYC. For planar geometry libraries, we projected the datasets to EPSG:32118 using *ogr2ogr* tool in GDAL. We used the *ogr2ogr* tool in GDAL to transform the lat/long coordinates in the datasets.

[12] https://github.com/jchambers/jvptree.

[13] https://openjdk.java.net/projects/code-tools/jmh/.

[14] https://github.com/msteindorfer/memory-measurer.

[15] https://github.com/google/benchmark.

[16] https://github.com/gperftools/gperftools.

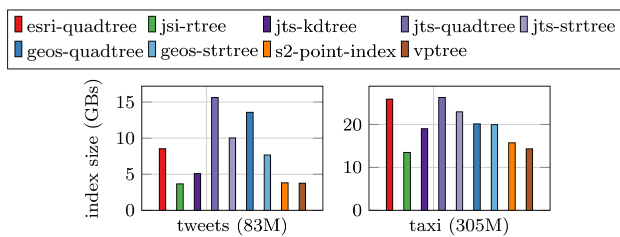


Fig. 2 Index sizes for the two datasets

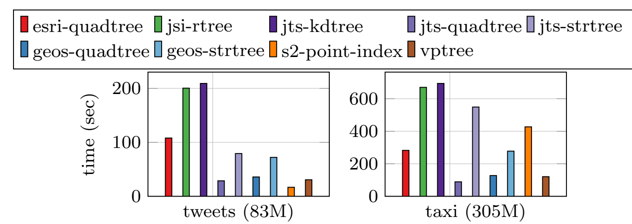


Fig. 3 Index building times for the two datasets

6 Evaluation

All experiments were run single threaded on a two-socket Ubuntu 18.04 machine with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 3.00 GHz turbo)[17] and 256 GB DDR3 RAM. We use the *numactl* command to bind the thread and memory to one node to avoid NUMA effects. CPU scaling was also disabled during benchmarking using the *cpupower* command.

We have benchmarked libraries written both in Java and C++. Although we have used language specific framework and tools to measure the performance of libraries, there are inherently many differences between the languages. As an example consider the size of an integer in the two languages. A type int Object in Java requires 16 bytes (depending on JVM implementation) while a type int in C++ requires 4 bytes. We ask the readers to carefully take such differences between languages into account while comparing performance of libraries written in different languages.

To evaluate the queries, we perform two experiments for each query. In the first experiment, we fix the selectivity of the query to 0.1% (we fix k to 10 in case of k-NN query) and vary the cardinality of the points dataset from 100,000 records to the maximum size of the dataset (i.e., 83 M records for Twitter dataset and 305 M for the Taxi dataset). In the second experiment, we fix the number of points to the maximum size of the dataset and vary the selectivity of the query from 0.0001 to 1% (we vary k from 1 to 10,000 in case of k-NN query). For all these experiments, we measure the throughput for each library in queries/s. In case of spatial join query, we report the join time in seconds. All query implementations are covered under the respective section. If a particular index does not support a query natively, the query is implemented using the *filter and refine* [39] approach.

6.1 Indexing Costs

ESRI Quadtree and JSI R-tree accept the rectangular range to index, and an identifier for the rectangular range, whereas other index structures are more liberal and allow users to put any user data along with the rectangular range. To be fair to all index structures, we only store the rectangular range to index and an identifier in every case and measure the size of these indexes in memory.

It is important at this point to categorize indexes in the libraries to better understand their behavior. Indexes in the libraries can be classified as: Point Access Methods (PAMs) and Spatial Access Methods (SAMs) [31]. PAMs are indexing methods that index point data, whereas SAMs index extended spatial objects such as rectangles, polygons etc. S2PointIndex, k-d tree and vptree are PAMs and the rest are SAMs. The indexes can also be categorized as space-driven (follow the embedding space hierarchy), or data-driven (follow the data space hierarchy). k-d tree and Quadtrees are space-driven structures and the rest of the indexes are data-driven.

Figure 2 shows the sizes of indexes in various libraries and Fig. 3 the time it takes to construct them. S2PointIndex, and vptree are PAMs which stores only points (at least two doubles) and hence the memory consumption is minimal. S2PointIndex is a B-tree that stores 64-bit integers (cell ids), and the overhead in inner nodes is minimal. jvptree only stores a vantage point, and a radius at every node, hence the intermediate nodes consume minimal memory. The rest of the indexes are SAMs and store rectangles and consume more memory than PAMs. This is expected, as the trees store rectangles.[18] each of which require storage of at least four doubles. Figure 2 also shows that the R-tree in JSI consumes very little memory even though it stores rectangles. JSI heavily relies on trove4j[19] collections, which are generally faster to access, and consumes much less memory than Java's Util collections. There are two reasons for low memory consumption. First is that (any) primitive collections store

[17] CPU: https://ark.intel.com/content/www/us/en/ark/products/75272/intel-xeon-processor-e5-2660-v2-25m-cache-2-20-ghz.html.

[18] We store points from the datasets as degerate rectangles in SAMs.

[19] http://trove4j.sourceforge.net/html/benchmarks.shtml.

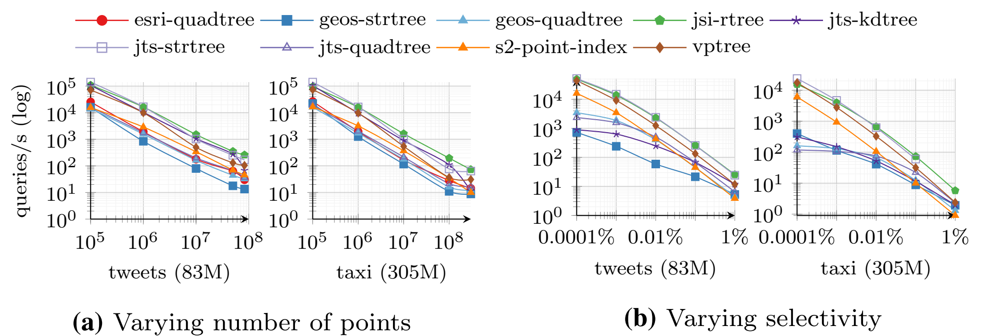


Fig. 4 Range query performance varying the number of points and selectivity of the query rectangle for NYC Taxi and Twitter Datasets

Table 3 *CPU Counters*: Range query datasize = 50M tweets, selectivity = 0.1%, 1 thread, normalized by the number of range queries

	Cycles	ipc	instr	L1 miss	LLC miss	Branch miss
esri-quadtree	116	0.84	98	1.34	0.54	0.08
geos-quadtree	105	0.75	79	0.97	0.75	0.09
geos-strtree	236	0.37	88	4.04	2.68	0.51
geos-cfstrtree	91	0.87	80	1.21	0.57	0.46
jsi-rtree	8	1.25	10	0.13	0.06	0.03
jts-kdtree	8	1.12	9	0.14	0.02	0.04
jts-quadtree	68	1.17	80	0.82	0.27	0.19
jts-strtree	31	0.81	25	0.42	0.22	0.01
s2-pointindex	44	1.34	59	0.42	0.05	0.36
vptree	30	0.70	21	0.68	0.21	0.05

All values are in millions except IPC

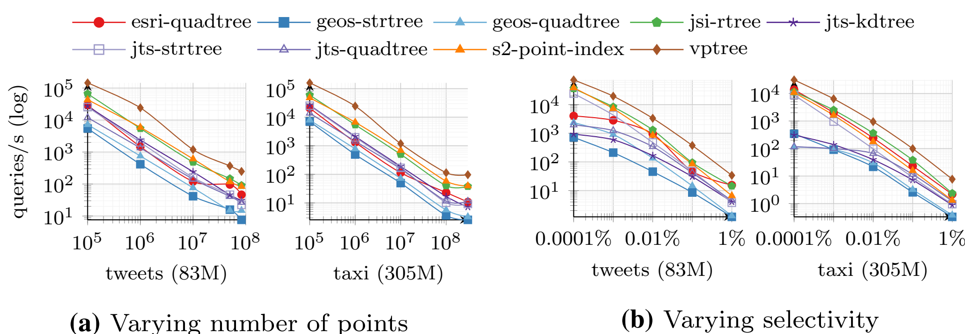
data directly in an array of primitives (int, long, double, char), and thus only a single reference to an array of primitives is needed instead of an array of references to data objects. JSI also uses floating-point precision while the other index structure use double precision values. Second, each primitive data element consumes less memory than the Object (e.g., type int only requires 4 bytes instead of 16 bytes object Integer). The reason for better performance is that trove4j avoids boxing and unboxing elements every time a primitive value is queried to/from the collection. It can also be seen that the space-driven indexes, i.e., Quadtrees and k-d tree, consumes more memory compared to the other index structures. Since space-driven structures divide the space they index, more internal nodes are formed as they keep dividing the space until a certain threshold is not met for the leaf node size.

Index construction times have been measured using the benchmarking frameworks, and are averaged over several runs until the runtime is statistically stable. For both Taxi and Twitter datasets, jvptree is the fastest to construct, whereas k-d tree and STRtree in JTS, Quadtree in ESRI geometry API and R-tree in JSI are among the slowest to construct for all datasets.

6.2 Range Query

Implementation: All indexes, except for jvptree, natively provide an interface for range queries. To implement range queries in jvptree we first compute the centroid q of the query rectangle. Next, we determine the distance of the centroid q to one of the rectangle's corner vertices. The resulting circle (q, d) is always larger then the range query rectangle and can therefore be used as a *filter* to retrieve a list with qualifying points. This list is then *refined* to determine which points are actually contained in the range query rectangle. As mentioned earlier, k-d tree in JTS keeps a count of points, in case of duplicate points (up to a certain distance tolerance), rather than creating a new node for the duplicate points. We make sure that we materialize all such points for the range query, but we do use them as an optimization in distance and join query to reduce the refinement costs (i.e., skip refinement for duplicate points if one point qualifies the refinement check).

Another point to mention here is that Quadtree implementation in ESRI geometry API requires tuning. The initialization of the Quadtree expects a height parameter for the index. As mentioned in Sect. 5, we generated range queries with varying selectivities from 0.0001 to 1%. We ran



(a) Varying number of points

(b) Varying selectivity

Fig. 5 Distance query performance varying the number of points and selectivity of the query rectangle for NYC Taxi Dataset and Twitter Datasets

all these range queries from selectivity 0.0001–1% on both datasets, and varied the height of the Quadtree from 1 to 64 for both datasets and for each selectivity. We then ranked these heights based on the lowest query runtime for each query selectivity, and compute the aggregated rank of all heights across all selectivities. We then selected the height with the lowest rank for both datasets. We found that the Quadtree performed best with heights 18 and 9 for the Taxi and Tweets datasets respectively.

Analysis: Fig. 4 shows the range query performance of various libraries on the Taxi and Twitter datasets. For both datasets, JSI R-tree show the best throughput numbers (259.87 and 72.779 queries per second, respectively, for Twitter and Taxi dataset for 0.1% selectivity). JSI R-tree is optimized for main memory usage for range queries and has the least height of all indexes (5 and 7 in the two datasets). Many of the tree nodes are cached and it suffers from the least number of cache misses as shown in Table 3.

An interesting case in the results is the low query throughput of GEOS STRtree (17.8315 queries per second in the Tweets dataset for 50 M points and 0.1% selectivity). GEOS STRtree is much slower than the JTS STRtree. Upon investigation, we found that the reason for the low query throughput of STRtree in GEOS is an implementation artifact. It can be seen in Table 3 that GEOS STRtree suffers from a large number of LLC misses, 2.68 million in the Twitter dataset and 1.28 million in the Taxi dataset (not shown in table). R-trees in general store multiple rectangles at every node. When the tree is queried, the decision to explore the branches from each node in the tree is based on whether the query range overlaps any of these rectangles. In both cases, JTS and GEOS, every node in the STRtree contains a maximum of 10 such rectangles by default. GEOS STRtree stores a vector of **pointers** to these rectangles at every node.

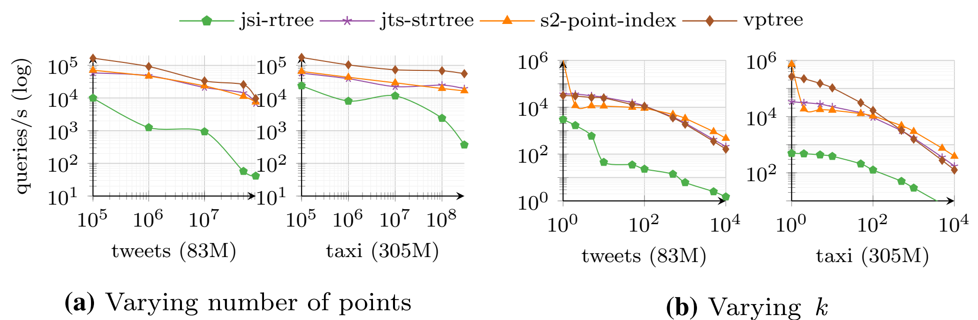
At every node, the algorithm in the range query iterates over these pointers, retrieves these rectangles from memory and checks whether there is any overlap with the query range and then based on the overlap explores the various branches from the node. Retrieving these rectangles from memory causes many cache misses in GEOS STRtree during the query execution. To validate this, we implemented a cache-friendly STRtree (designated as cfstrtree in Table 3) in GEOS on top of the existing tree. We basically introduced another vector at every node in the tree, which stores the objects of these rectangles in contiguous memory. We replaced the logic to check for overlap to use these rectangle objects rather than the pointers to the rectangles. This reduces the number of LLC misses in the CFSTRtree relative to STRtree, by a large number as can be seen in Table 3.

STRtree implementation in JTS does not suffer from this. In both libraries, GEOS and JTS, the algorithm for constructing and traversing the trees are the same, but the difference in performance stems from how memory management works in the JVM. Every node in JTS STRtree stores the rectangle objects in a *List*. Lists in Java store the references to the objects, so logically it is similar to storing a vector of pointers in C++. But where this differs is that JVM makes a distinction between small and large objects during object allocation [66]. The limit for when an object is considered large depends on the JVM version, the heap size, the garbage collection strategy and the platform used, but is usually somewhere between two and 128 kB. Small objects are allocated in thread local areas (TLAs). The thread local areas are free chunks reserved from the heap and given to a Java thread for exclusive use. The thread can then allocate objects in its TLA without synchronizing with other threads. The size of the rectangle objects in JTS is 48 bytes each. This means that the rectangle objects qualify as small objects and are in contiguous memory. Only the access to the first rectangle causes a cache miss, and the other objects are most likely brought into memory as a side effect of that cache miss (speculative loading).

6.3 Distance Query

Implementation: S2PointIndex and jvptree provide native support for distance queries, so we directly issue the query point and the distance to these two indexes. The other indexes do not support distance query natively. To implement distance queries in these indexes, we again use the

Fig. 6 *k*-NN query performance varying the number of points and *k* for NYC Taxi and Twitter Datasets



filter and refine paradigm. We first filter using a rectangle, whose corner vertices are at a distance of d from the query point q. We issue a range query to the various range based indexes using this rectangle. We then refine the resulting candidate set of points by using a *withinDistance* predicate (available in ESRI Geometry API, JTS, and GEOS). For JSI, we implemented our own predicate, which computes the Euclidean distance for all candidate points from the query point and checks if the candidate point is within distance $d * d$ rather than d from the query point. This helps in skipping the square root operation to calculate Euclidean distance.

Analysis: Fig. 5 shows the distance query performance on Taxi and Twitter datasets. The performance for distance query is dominated by range query lookup for most indexes, apart from S2PointIndex and jvptree. These index support distance queries natively, i.e., have specialized tree traversal algorithms for distance query. For other indexes, we deploy the filter and refine paradigm. The performance of these indexes thus follows directly from the range query performance. JSI R-tree is slightly better than JTS k-d tree as we optimize the Euclidean distance computation by skipping the square root operation. We would also advise the readers to use this approach for refinement in GEOS as well. The *isWithinDistance* function in GEOS returns whether two geometries are within a certain distance from each other. By profiling the function we noticed that this function makes six *malloc()* calls, for every candidate point, which degrades the performance. By using our own predicate distance function, we were able to speed up distance query by up to 2× in GEOS. In many geometric operations, GEOS frequently allocates and deallocates memory, which is an overhead. This problem in memory management was also observed by [71], where authors use GEOS to introduce spatial processing in Impala.

6.4 *k*-NN Query

Implementation: Out of all the available indexes, only S2PointIndex, JTS STRtree, JSI R-tree, and jvptree support *k*-NN queries natively. We directly issue the query point to these indexes and measure their performance. We did not implement any tree traversal algorithms for any other available tree because we wanted to measure the performance of the libraries without making any changes to the library source code.

Analysis: Fig. 6 shows *k*-NN query performance of various indexes on the Taxi and Twitter datasets. jvptree again takes the crown as the best performing index for *k*-NN queries, with S2PointIndex close behind. It can be observed that for the Twitter dataset the performance of JSI R-tree fluctuates quite a bit. This can be explained by how the nearest-neighbor algorithm works in JSI R-tree (and also in JTS STRtree), which is known as branch-and-bound traversal. The algorithm starts with adding the root node to a priority queue of size k. The algorithm, then iterates over the tree continuously adding nodes until the priority queue is full. The algorithm then continues traversing the tree observing nodes, replacing the current farthest node in the queue with the current node being looked at, if it is closer. JSI R-tree is a *dynamic* tree, it is built in a top-down fashion (spatial objects are inserted from the root to the leaf), and the nodes are split (or merged) based on various factors. It is evident that several R-trees can represent same set of data rectangles [16], depending on insertion order and grouping of data objects into leaves. The JSI R-trees for different sized datasets are therefore vastly different. Thus, during the tree traversal for *k*-NN query, sometimes a large number of branches from a node can be dropped since they are not closer than the current farthest node in the priority queue and sometimes they cannot be dropped. This can lead to multiple search paths to be evaluated and hence the fluctuation in performance. JTS STRtree packed R-tree does not suffer from this because it is a type of *static* R-tree. It is built bottom-up and once built, no more elements can be added. STRtree is built by first sorting the leaf node in the x dimension, and then dividing the data in vertical slices, each containing an equal number of points. Within each slice, the data is sorted in the y dimension, and again divided into slices containing an equal number of points. The tree is then built on top of these slices by packing a predefined number of slices into

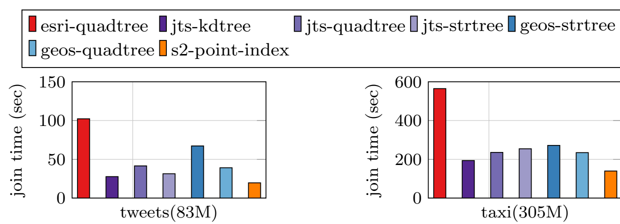


Fig. 7 Join query performance for NYC Taxi and Twitter Datasets

nodes. The difference in tree node boundaries is still there in JTS STRtree but is more profound in the lower levels of the tree, rather than at various levels as in the case of JSI R-tree. Thus, JSI R-tree can sometime quickly discard branches at the top of the tree and other times it cannot, and this is reflected in the query throughput.

6.5 Point-In-Polygon Join Query

Implementation: In S2, we used the S2ShapeIndex, instead of S2PointIndex, which provides a native interface for the contains predicate. S2ShapeIndex[20] stores a map from S2CellId to the set of shapes that intersect that cell. The shapes are identified by a ShapeId. As shapes are added to the index, their cell ids are computed and added along with the shape id to the index. When a query point is issued against the index it retrieves the cells that contain the query point and identifies the shape(s) that this containing cell belongs to using the shape id. For other indexes, we again use the filter and refine approach. For GEOS and JTS we use PreparedGeometry[21] to index line segments of all individual polygons, which helps in accelerating the refinement check. In JTS, we also use k-d tree's points snapping technique to skip refinement for duplicate points in case one point qualifies or disqualifies the predicate check. In ESRI implementation, we use AcceleratedGeometry and set its *accelDegree* to *enumHot*[22] for the fastest containment performance.

Analysis: Fig. 7 shows joins query performance on the Taxi and the Twitter datasets. Spatial join queries are notoriously expensive and this is reflected in the figure. For join queries S2ShapeIndex performs the best. As mentioned earlier, we skip the refinement check for duplicate points if one such point qualifies (or disqualifies) the refinement check and that is why it does slightly better than the other indexes. S2ShapeIndex natively supports the containment query and traverses the index appropriately and does not have to deal with refining many candidate set of points. The performance of other indexes follows from the range query performance. JTS/GEOS STRtree and Quadtree perform better than ESRI Quadtree because the refinement using PreparedGeometry is faster than AcceleratedGeometry in ESRI.

7 Discussion

In this section, we first discuss a research direction that we believe might not be getting the attention in the community that it should, before we outline how to use modern spatial libraries as building blocks for building distributed spatial systems.

7.1 Why Refinement Should Be Looked At?

As we learned in the past sections, the modern spatial libraries provide index structures which arrange spatial objects in a way that the access time to these geometric objects reduces. But we also learned that these index structures only support a limited set of native queries (range lookup and *k*-NN query in most cases). In other queries, such as distance query and spatial joins, these index structures primarily act as filters. The resulting candidate set of points (or geometries) after the filter phase needs to be further refined based on a spatial predicate. For distance query, the predicate is *withinDistance*, and for spatial joins, the predicate can be one of many predicates, such as contains, intersects, overlaps, etc. For these queries, we used the filter and refine paradigm. The set of geometry objects from the candidate set that do not qualify the predicate check are known as *false drops* and the ones that do are known as *candidate hits*. Generally, we can determine how good these indexes are for such queries by analyzing the ratio of number of false drops to the number of candidate hits. If the ratio is more than 1, it can be deduced that the amount of work being done for *false drops* is more than for *candidate hits*. This work done can be classified as an overhead, and the goal is to *minimize* this overhead.

In this study we also looked at an index structure, namely, Vantage Point Tree, which is specially designed to answer distance and *k*-NN queries. We saw in Sect. 6 that for distance queries an open-source implementation of VPTree, performs 2.48× better for the Taxi dataset (and 2.74× for the Twitter dataset) than its closest competitors S2PointIndex and JSI R-tree. Please note that in JSI R-tree we even skipped the overhead of square root operation in Euclidean distance computation. This is because jvptree reduces the overhead of *false drops* during the index lookup itself. In essence, the index structure completely skips the refinement phase for distance and *k*-NN queries and does not have to deal with *false drops*. This shows that if an index structure is

[20] http://s2geometry.io/devguide/s2shapeindex.html.

[21] https://locationtech.github.io/jts/javadoc/org/locationtech/jts/geom/prep/PreparedGeometry.html.

[22] https://esri.github.io/geometry-api-java/javadoc/com/esri/core/geometry/Geometry.GeometryAccelerationDegree.html.

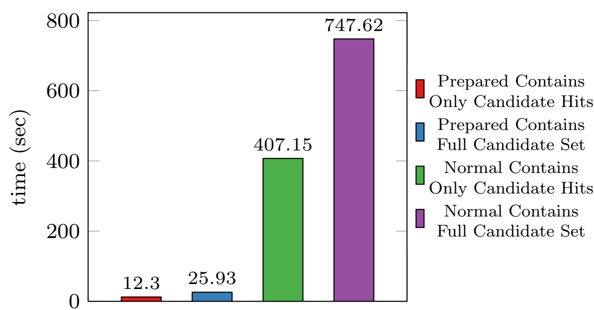


Fig. 8 Refinement costs for Midtown Manhattan Polygon for NYC Taxi Dataset using various contains functions in JTS

built to answer certain queries, and no refinement is needed, the performance implications can be large.

Recent research acknowledges [5, 62] that there is potential in accelerating the refinement step for join queries. We consider the spatial point-in-polygon join query here, where filter and refinement is also required for some indexes. In point-in-polygon join, after the filter phase, the candidates set of points is typically refined using an algorithm known as ray tracing. In this algorithm, a line (ray) is drawn from the query point to a point known to be outside the polygon, and then the number of intersections of this line with all edges in the polygon is counted. This algorithm is linear with the number of edges in the polygon. So if the cardinality of the filtered candidate set of points after filtering from the index is n, then the work to be done is $\mathcal{O}(nk)$, where k is the number of edges in the polygon. If the n is large or if the polygons are complex, with a large number of edges then this has the potential to become a bottleneck. Midtown Manhattan is one of the neighborhoods in NYC that is highly skewed for the Taxi and the Twitter dataset alike. Using the bounding box of Midtown Manhattan and querying any range-based index (i.e., which can be queried using a range, in this case MBR of Midtown Manhattan) as a filter with 305 million taxi rides, yields a candidate set with 78.35 million points. The final result after refinement has 42.55 million points (*candidate hits*), with **35.8 million** points being *false drops*.

Using Midtown Manhattan as a query polygon, we carried out an experiment to determine the costs of refinement using various contains functions in JTS and the results are shown in Fig. 8. In PreparedGeometry, the individual geometry objects are indexed and the indexing scheme varies based on the geometry datatype. For example, for polygons, PreparedGeometry indexes the line segments of the polygons. If the refinement step can be skipped for *false drops*, there is gain of 2.10× in query performance (12.3 s without false drops vs. 25.93 s with false drops). The figure also shows the effect of indexing individual polygons. If line segments in polygons are not indexed, the polygon contains function takes 747.62 s compared to 2.93 s (28.83× improvement).

There are two potential research directions for improving point-in-polygon spatial join queries. As mentioned earlier, the potential work to be done in the refinement phase after filtering is $\mathcal{O}(nk)$. We can either try to reduce n or k (or both). Some of the recent research work [22–24, 73] tries to address the former and skip the refinement phase altogether. The latter is addressed to some extent in the libraries via PreparedGeometry (in JTS and GEOS) and AcceleratedGeomerty (in ESRI Geometry API). There is also a research [75] work that show that the refinement step can be improved by using interval trees to index the polygon line segments.

7.2 Distributed Spatial Analytics Systems

In the past few years, a number of big spatial analytics systems have emerged. While they differ in some architectural design aspects, many of the core fundamentals remain the same in terms of building a distributed spatial processing system. In this section, we briefly highlight these fundamentals and how a distributed spatial analytics system can be built from scratch using the libraries studied in this work. A cluster of commodity machines coordinating to complete a task generally have the following structure: a master node (the coordinator) and multiple worker nodes. Big spatial analytics systems today also deploy the same cluster setup since they are primarily built on big data infrastructures in the form of Hadoop, Apache Spark, Impala etc. There are three main components to designing a big spatial analytics system: (1) Partitioning Technique, (2) Index Structures, and (3) Supported Datatypes and Queries.

Index structures, as we saw in this work, are important for answering spatial queries. Spatial indexes allow access to the desired spatial objects in sub-linear time and thus accelerate spatial query processing. Index structures hence form an integral part of any spatial processing system, whether it be a relational database system, or a distributed spatial processing system. Spatial partitioning is also an important part of distributed spatial processing system, which we now discuss in detail:

7.2.1 Spatial Partitioning

Spatially partitioning the input dataset(s) is an important aspect of distributed spatial processing system. Since there are multiple worker nodes in a cluster, an input dataset should be partitioned to fully utilize the parallel computing capability of the cluster. Also, since many of the spatial datasets are inherently spatially skewed, it is important to

Table 4 Strengths/Weaknesses of the Libraries

Library	Strengths	Weaknesses
ESRI	(1) Active development and support (2) Full geometric types, refinements, and operations	(1) Quadtree requires tuning
JSI	(1) R-tree performance as a filter	(1) No active development (2) No geometric refinements
GEOS and JTS	(1) Active development and support (2) Full geometric types, refinements, and operations	(1) Memory management in GEOS requires improvement
jvptree	(1) Best distance and *k*-NN performance	(1) No geometric refinements
S2	(1) Best suited for geographic data (2) Active development and support (3) Many practical queries natively supported	

partition them spatially. A naive grid partitioning would introduce skew in some individual grid cells, and thus leads to the *straggling* nodes in the cluster, which would affect the overall query efficiency.

How is it done?: The usual practice today to build partitions is to sample the input dataset and to determine partitions based on the sample. Previous research [8] has shown that sampling 1% of the input dataset is sufficient to produce high-quality partitions. To further delve into detail, we will walk through an example, using an R-tree index. After sampling the input dataset, an R-tree is built on the sample. Sampling helps in capturing the density distribution of the input dataset, and indexing the samples in an R-tree spatially partitions the sample, thereby providing the *partitions boundaries* of the sample dataset. The minimum bounding rectangle (MBR) of the leaves of the R-tree are then used as the *partition boundaries*. Once the partition boundaries have been determined, the input dataset can then be loaded in parallel using these boundaries. Now since these partition boundaries were determined using only a sample of the dataset, and the input dataset may contain spatial objects that do not lie, or even overlap multiple partition boundaries. The common practice today is to expand the partition boundaries or duplicate the object in multiple partitions. We refer the readers to [8] to understand the trade-offs related to such decisions. Once the partitions have been built, the individual partition are indexed in an R-tree. The index does not necessarily have to be an R-tree but for the sake of continuity, we continue using the R-tree as an example. These index within individual partitions are called *local index* (i.e., local to a partition). Once these local indexes have been built, finally a *global index* is built using the spatial extent of these local indexes. We walked through an example with R-tree as the index to determine the partition boundaries, but R-tree may not be the best partitioning scheme in certain scenarios. We refer the readers to [8], which thoroughly compares and evaluates various spatial partitioning techniques.

Why is it done?: Spatial partitioning an input dataset helps in query processing. To better understand the importance of spatial partitioning, we will walk through an example. Consider a large input dataset, and a range query is issued to determine which spatial objects in the input dataset lie within the given range. The *global index* is first used to determine which partitions the input range overlaps and then the overlapping partitions can be scanned with the given range. This saves unnecessary scans of the partitions that do not overlap the input range. This is a very simple example, but things get more complex when join queries are considered. A join query can be processed as follows: the global indexes of the two datasets are first consulted to determine the partitions that overlap each other and then these partitions can be joined in parallel. This is again a very simple example of how the spatial partitions and indexes can be used to process a join query, and how to avoid joining partitions that do not spatially overlap. In reality, the systems deploy query optimizers that determine the best way to join the two datasets. For example, when the two global indexes are considered to determine which partitions overlap, it could very well be that a large number of partition pairs overlap (since they are two different datasets). A system may choose to repartition one dataset to minimize these overlapping partition pairs. These are design choices that these systems make, and they are based on various trade-offs. We refer the readers to the individual systems to better understand these design choices and trade-offs.

8 Related Work

To the best of our knowledge, no previous work in literature has evaluated the spatial libraries studied here empirically. One research work [19] compares indexing

techniques for big spatial data, where the authors consider many big spatial data systems and one spatial library JSI, only to report the performance of each system/library on a standalone basis. The authors in [71] implement spatial query processing in Apache Spark, and Apache Impala using JTS and GEOS, respectively. They do observe some of the implementation differences between JTS and GEOS, but largely the work is about a comparative study of spatial processing in Spark and Impala. Another research work [40] compares five Spark based spatial analytics systems, some of which use JTS library for spatial query processing. The authors in [13] shows how to efficiently implement distance join queries in distributed spatial data management systems. A research work [20] compares Quadtree and R-tree as filters in Oracle Spatial.

9 Conclusions

In this work we empirically compared popular spatial libraries using four different queries: range query, distance query, *k*-NN query, and a point-in-polygon join query. We performed a thorough experimental evaluation of these libraries using two real-world points datasets. While we evaluated the libraries on the point datatype, there are other datatypes (such as linestring, polylines etc.) in the libraries that should also be evaluated. We leave evaluating the libraries on other geometric datatypes for future work. Table 4 summarizes the strengths and weaknesses of the spatial libraries.

There is no clear winner for each of the considered queries, and this is mostly because all the indexes available in the libraries do not support all these queries natively (i.e., do not have specialized tree traversal algorithms for each query). ESRI geometry API and JTS/GEOS are complete planar geometry libraries, and are rich in features. They support multiple datatypes, and have a variety of topological and geometry operations. They are also under active development and have a community for support. They do, however, come with some drawbacks. ESRI Quadtree has to be tuned for the dataset that it indexes, and memory management in GEOS could be improved. We also identified a difference in implementation of GEOS STRtree and JTS STRtree which has performance implications. Although both index structures implement the same algorithm for tree traversal, the difference in performance stems from memory management in Java. To validate this, we implemented a cache friendly version of GEOS STRtree and highlight the improvement in performance. The R-tree in JSI exhibited the best performance for range lookups, however, JSI is very limited in features, and is also not under active development. We also highlighted the difference in *k*-NN query performance of JSI R-tree and JTS STRtree. Both index structures implement branch-and-bound traversal to answer *k*-NN query and the difference in performance is due to the structure of the trees being different. JSI R-tree is a dynamic tree (constructed top-down by inserting objects at the root node), and the nodes are split (or merged) based on multiple factors, whereas JTS STRtree is a static tree (constructed bottom-up by sorting points in both dimensions, partitioning into slices, and packing the slices to nodes). Google S2 is a spherical geometry library and is best suited to work with geographic data. It is under active development and is used in many multimillion-dollar industries. It also has many practically used queries that are implemented natively on various indexes. Finally, jvptree, is a library that implements the Vantage Point Tree. It exhibits the best performance for distance and *k*-NN queries as it is specifically designed to answer these queries. The index can only be used as a filter for other queries, and users have to implement their own refinement operations for such queries.

We also identified areas of potential pitfalls when using the planar geometry libraries which can be critical from the perspective of actual users, either of these libraries or any system that is based on them. Particularly for distance computations, the differences can be significant when using planar geometry for processing GPS coordinates. Many important business decisions might be based on the outcome of such queries and there are potentially hundreds of users and companies that are using software that is based on these state-of-the-art spatial libraries. While these libraries and systems correctly execute what they are designed to do, users should be aware of how to use them correctly.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. A Comprehensive Guide to S2 Cells and Pokémon GO (2019). https://pokemongohub.net/post/article/comprehensive-guide-s2-cells-pokemon-go/
2. Aji A, Wang F, Vo H, Lee R, Liu Q, Zhang X, Saltz JH (2013) Hadoop-gis: a high performance spatial data warehousing system over mapreduce. PVLDB 6(11):1009–1020. https://doi.org/10.14778/2536222.2536227

3. Amemiya K, Nakao A (2020) Layer-integrated edge distributed data store for real-time and stateful services. In: NOMS 2020—IEEE/IFIP network operations and management symposium, pp 1–9. IEEE. https://doi.org/10.1109/NOMS47738.2020.9110436
4. Boric N, Gildhoff H, Karavelas M, Pandis I, Tsalouchidou I (2020) Unified spatial analytics from heterogeneous sources with amazon redshift. In: Proceedings of the 2020 international conference on management of data, SIGMOD conference 2020, pp 2781–2784. ACM. https://doi.org/10.1145/3318464.3384704
5. Bouros P, Mamoulis N (2019) Spatial joins: What's next? SIGSPATIAL Special 11(1):13–21
6. Doraiswamy H, Freire J (2020) A gpu-friendly geometric data model and algebra for spatial queries. In: Proceedings of the 2020 international conference on management of data, SIGMOD conference 2020, pp 1875–1885. ACM. https://doi.org/10.1145/3318464.3389774
7. Doraiswamy H, Freire J (2020) A gpu-friendly geometric data model and algebra for spatial queries: extended version. CoRR arXiv:2004.03630
8. Eldawy A, Alarabi L, Mokbel MF (2015) Spatial partitioning techniques in spatial hadoop. PVLDB 8(12):1602–1605. https://doi.org/10.14778/2824032.2824057
9. Eldawy A, Mokbel MF (2015) Spatialhadoop: a mapreduce framework for spatial data. In: ICDE 2015, Seoul, South Korea, April 13–17, 2015, pp 1352–1363. IEEE Computer Society. https://doi.org/10.1109/ICDE.2015.7113382
10. Eldawy A, Sabek I, Elganainy M, Bakeer A, Abdelmotaleb A, Mokbel MF (2017) Sphinx: empowering impala for efficient execution of SQL queries on big spatial data. In: SSTD 2017. https://doi.org/10.1007/978-3-319-64367-0_4
11. EPSG:32118—NAD83/New York Long Island. https://spatialreference.org/ref/epsg/32118/
12. García-García F, Corral A, Iribarne L, Vassilakopoulos M (2020) Improving distance-join query processing with voronoi-diagram based partitioning in spatialhadoop. Future Gener Comput Syst 111:723–740. https://doi.org/10.1016/j.future.2019.10.037
13. García-García F, Corral A, Iribarne L, Vassilakopoulos M, Manolopoulos Y (2020) Efficient distance join query processing in distributed spatial data management systems. Inf Sci 512:985–1008. https://doi.org/10.1016/j.ins.2019.10.030
14. Gomes D (2019) MemSQL Live: Nikita Shamgunov on the Data Engineering Podcast. https://www.memsql.com/blog/memsql-live-nikita-shamgunov-on-the-data-engineering-podcast/
15. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: SIGMOD'84. https://doi.org/10.1145/602259.602266
16. Hadjieleftheriou M, Manolopoulos Y, Theodoridis Y, Tsotras VJ (2017) R-trees: a dynamic index structure for spatial searching, pp 1805–1817. Springer. https://doi.org/10.1007/978-3-319-17885-1_1151
17. Hagedorn S, Götze P, Sattler K (2017) The STARK framework for spatio-temporal data analytics on spark. In: Datenbanksysteme für Business, Technologie und Web (BTW 2017)
18. Heath A (2017) Snap confirms that it paid $213 million to buy Zenly and $135 million for Placed. https://www.businessinsider.com/snapchat-paid-213-million-for-zenly-and-135-million-for-placed-2017-8/
19. Jhummarwala A, Alkathiri M, Karamta M, Potdar MB (2016) Comparative evaluation of various indexing techniques of geospatial vector data for processing in distributed computing environment. In: Proceedings of the 9th annual ACM India conference, 2016, pp 167–172. https://doi.org/10.1145/2998476.2998493
20. Kanth KVR, Ravada S, Abugov D (2002) Quadtree and r-tree indexes in oracle spatial: a comparison using GIS data. In: Proceedings of the 2002 ACM SIGMOD international conference on management of data, 2002, pp 546–557. ACM. https://doi.org/10.1145/564691.564755
21. Kemper A, Neumann T (2011) Hyper: a hybrid oltp&olap main memory database system based on virtual memory snapshots. In: Proceedings of the 27th international conference on data engineering, ICDE 2011, pp 195–206
22. Kipf A, Lang H, Pandey V, Persa RA, Anneser C, Zacharatou ET, Doraiswamy H, Boncz PA, Neumann T, Kemper A (2020) Adaptive main-memory indexing for high-performance point-polygon joins. In: Proceedings of the 23nd international conference on extending database technology, EDBT 2020, pp 347–358. OpenProceedings.org. https://doi.org/10.5441/002/edbt.2020.31
23. Kipf A, Lang H, Pandey V, Persa RA, Boncz PA, Neumann T, Kemper A (2018) Adaptive geospatial joins for modern hardware. CoRR arxiv:1802.09488
24. Kipf A, Lang H, Pandey V, Persa RA, Boncz PA, Neumann T, Kemper A (2018) Approximate geospatial joins with precision guarantees. In: 34th IEEE international conference on data engineering, ICDE 2018, pp 1360–1363. https://doi.org/10.1109/ICDE.2018.00150
25. Kipf A, Pandey V, Böttcher J, Braun L, Neumann T, Kemper A (2017) Analytics on fast data: Main-memory database systems versus modern streaming systems. In: EDBT 2017, pp 49–60. OpenProceedings.org. https://doi.org/10.5441/002/edbt.2017.06
26. Kipf A, Pandey V, Böttcher J, Braun L, Neumann T, Kemper A (2019) Scalable analytics on fast data. ACM Trans Database Syst 44(1):1:1–1:35. https://doi.org/10.1145/3283811
27. Lang H, Kipf A, Passing L, Boncz PA, Neumann T, Kemper A (2018) Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. In: Proceedings of the 14th international workshop on data management on new hardware, 2018, pp 5:1–5:8. ACM. https://doi.org/10.1145/3211922.3211928
28. Lee K, Ganti RK, Srivatsa M, Liu L (2014) Efficient spatial query processing for big data. In: Proceedings of the 22nd ACM SIGSPATIAL, 2014. https://doi.org/10.1145/2666310.2666481
29. Lee K, Liu L, Ganti RK, Srivatsa M, Zhang Q, Zhou Y, Wang Q (2019) Lightweight indexing and querying services for big spatial data. IEEE Trans Serv Comput 12(3):343–355. https://doi.org/10.1109/TSC.2016.2637332
30. Leutenegger ST, Edgington JM, López MA (1997) STR: a simple and efficient algorithm for r-tree packing. In: Proceedings of the thirteenth international conference on data engineering, April 7–11, 1997, Birmingham, UK, pp 497–506. IEEE Computer Society. https://doi.org/10.1109/ICDE.1997.582015
31. Liu L, Özsu MT (eds) (2018) Encyclopedia of database systems, 2nd edn. Springer. https://doi.org/10.1007/978-1-4614-8265-9
32. Makris A, Tserpes K, Spiliopoulos G, Anagnostopoulos D (2019) Performance evaluation of mongodb and postgresql for spatio-temporal data. In: Proceedings of the workshops of the EDBT/ICDT 2019 joint conference, EDBT/ICDT 2019, Lisbon, Portugal, March 26, 2019, CEUR Workshop Proceedings, vol 2322. CEUR-WS.org
33. Malensek M, Pallickara SL, Pallickara S (2013) Polygon-based query evaluation over geospatial data using distributed hash tables. In: IEEE/ACM 6th international conference on utility and cloud computing, UCC, 2013. https://doi.org/10.1109/UCC.2013.46
34. Malensek M, Pallickara SL, Pallickara S (2014) Evaluating geospatial geometry and proximity queries using distributed hash tables. Comput Sci Eng 16(4):53–61. https://doi.org/10.1109/MCSE.2014.48
35. MongoDB Releases—New Geo Features in MongoDB 2.4 (2013) https://www.mongodb.com/blog/post/new-geo-features-in-mongodb-24/

36. Moore L (1997) Transverse mercator projections and us geological survey digital products. US Geological Survey, Professional Paper
37. NYC Taxi and Limousine Commission (TLC)—TLC Trip Record Data (2019) https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
38. Oracle Spatial and Graph Spatial Features (2019) https://www.oracle.com/technetwork/database/options/spatialandgraph/overview/spatialfeatures-1902020.html/
39. Orenstein JA (1989) Redundancy in spatial databases. In: Proceedings of the 1989 ACM SIGMOD international conference on management of data, 1989. https://doi.org/10.1145/67544.66954
40. Pandey V, Kipf A, Neumann T, Kemper A (2018) How good are modern spatial analytics systems? PVLDB 11(11):1661–1673. https://doi.org/10.14778/3236187.3236213
41. Pandey V, Kipf A, Vorona D, Mühlbauer T, Neumann T, Kemper A (2016) 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. https://doi.org/10.1145/2882903.2899412
42. Pandey V, van Renen A, Kipf A, Kemper A (2020) An evaluation of modern spatial libraries. In: Database systems for advanced applications—25th international conference, DASFAA 2020, Jeju, South Korea, Sept 24–27, 2020, Proceedings, Part II, Lecture Notes in Computer Science, vol 12113, pp 711–727. Springer. https://doi.org/10.1007/978-3-030-59416-9_46
43. Pandey V, van Renen A, Kipf A, Sabek I, Ding J, Kemper A (2020) The case for learned spatial indexes. CoRR arXiv:2008.10349
44. Ranney M (2015) Scaling uber's real-time market platform. https://www.infoq.com/presentations/uber-market-platform/
45. Ren F, Li X, Thomson D, Geng D (2018) Geosharded recommendations part 1: sharding approach. https://tech.gotinder.com/geosharded-recommendations-part-1-sharding-approach-2/
46. Richly K (2019) Optimized spatio-temporal data structures for hybrid transactional and analytical workloads on columnar in-memory databases. In: VLDB 2019 PhD workshop, CEUR workshop proceedings, vol 2399. CEUR-WS.org. http://ceur-ws.org/Vol-2399/paper10.pdf
47. Saxena S (2017) Appreciating the geo/S2 library. https://blog.gojekengineering.com/fe-f0e4a909d56f
48. Schubert E, Zimek A, Kriegel H (2013) Geodetic distance queries on r-trees for indexing geographic data. In: Advances in spatial and temporal databases—13th international symposium, SSTD 2013, Munich, Germany, Aug 21–23, 2013. Proceedings, pp 146–164. https://doi.org/10.1007/978-3-642-40235-7_9
49. Sidlauskas D, Chester S, Zacharatou ET, Ailamaki A (2018) Improving spatial data processing by clipping minimum bounding boxes. In: 34th IEEE international conference on data engineering, ICDE 2018, pp 425–436. IEEE Computer Society. https://doi.org/10.1109/ICDE.2018.00046
50. Sinton A (2018) Geospatial indexing on Hilbert curves. https://blog.zen.ly/geospatial-indexing-on-hilbert-curves-2379b929addc/
51. SR-ORG:6864|EPSG:3857. https://spatialreference.org/ref/sr-org/6864/
52. S2Geometry Overview—Spherical Geometry. https://s2geometry.io/about/overview/
53. S2 cells and Pokémon GO (2018). https://pokemongohub.net/post/wiki/s2-cells-pokemon-go/
54. Tahboub RY, Essertel GM, Rompf T (2018) How to architect a query compiler, revisited. In: Proceedings of the 2018 international conference on management of data, SIGMOD conference 2018, Houston, TX, USA, June 10–15, 2018, pp 307–322. ACM. https://doi.org/10.1145/3183713.3196893
55. Tahboub RY, Rompf T (2016) On supporting compilation in spatial query engines: (vision paper). In: Proceedings of the 24th ACM SIGSPATIAL international conference on advances in geographic information systems, GIS 2016, Burlingame, California, USA, Oct 31–Nov 3, 2016. https://doi.org/10.1145/2996913.2996945
56. Tahboub RY, Rompf T (2020) Architecting a query compiler for spatial workloads. In: Proceedings of the 2020 international conference on management of data, SIGMOD conference 2020, pp 2103–2118. ACM. https://doi.org/10.1145/3318464.3389701
57. Tang M, Tahboub RY, Aref WG, Atallah MJ, Malluhi QM, Ouzzani M, Silva YN (2016) Similarity group-by operators for multi-dimensional relational data. IEEE Trans Knowl Data Eng. https://doi.org/10.1109/TKDE.2015.2480400
58. Tang M, Yu Y, Malluhi QM, Ouzzani M, Aref WG (2016) Locationspark: A distributed in-memory data management system for big spatial data. PVLDB 9(13):1565–1568. https://doi.org/10.14778/3007263.3007310
59. Theocharidis K, Liagouris J, Mamoulis N, Bouros P, Terrovitis M (2019) SRX: efficient management of spatial RDF data. VLDB J 28(5):703–733. https://doi.org/10.1007/s00778-019-00554-z
60. Titlow JP (2013) How foursquare is building a humane map framework to rival google. https://www.fastcompany.com/3007394/how-foursquare-building-humane-map-framework-rival-googles/
61. Toliopoulos T, Nikolaidis N, Michailidou A, Seitaridis A, Gounaris A, Bassiliades N, Georgiadis A, Liotopoulos F (2020) Developing a real-time traffic reporting and forecasting back-end system. In: Research challenges in information science—14th international conference, RCIS 2020, Limassol, Cyprus, Sept 23–25, 2020, Proceedings, Lecture Notes in Business Information Processing, vol 385, pp 58–75. Springer. https://doi.org/10.1007/978-3-030-50316-1_4
62. Tsitsigkos D, Bouros P, Mamoulis N, Terrovitis M (2019) Parallel in-memory evaluation of spatial joins. CoRR arXiv:1908.11740
63. Tsitsigkos D, Bouros P, Mamoulis N, Terrovitis M (2019) Parallel in-memory evaluation of spatial joins. In: Proceedings of the 27th ACM SIGSPATIAL international conference on advances in geographic information systems, SIGSPATIAL 2019, Chicago, IL, USA, Nov 5–8, 2019, pp 516–519. ACM. https://doi.org/10.1145/3347146.3359343
64. Tsitsigkos D, Lampropoulos K, Bouros P, Mamoulis N, Terrovitis M (2020) A two-level spatial in-memory index. CoRR arXiv:2005.08600
65. Tutorials (2020) Filtering tweets by location. https://developer.twitter.com/en/docs/tutorials/filtering-tweets-by-location
66. Understanding Memory Management—Oracle. https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html/
67. Weisstein EW (2002) Great circle. https://mathworld.wolfram.com/GreatCircle.html
68. Winter C, Kipf A, Neumann T, Kemper A (2019) Geoblocks: a query-driven storage layout for geospatial data. CoRR arXiv:1908.07753
69. Xie D, Li F, Yao B, Li G, Zhou L, Guo M (2016) 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. https://doi.org/10.1145/2882903.2915237
70. Yianilos PN (1993) Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proceedings of the fourth annual ACM/SIGACT-SIAM symposium on discrete algorithms, 25–27 Jan 1993, Austin, Texas, USA
71. You S, Zhang J, Gruenwald L (2015) 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. https://doi.org/10.1109/ICDEW.2015.7129541

72. Yu J, Wu J, Sarwat M (2015) Geospark: a cluster computing framework for processing large-scale spatial data. In: Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems, Bellevue, WA, USA, Nov 3–6, 2015. https://doi.org/10.1145/2820783.2820860
73. Zacharatou ET, Doraiswamy H, Ailamaki A, Silva CT, Freire J (2017) GPU rasterization for real-time spatial aggregation over arbitrary polygons. PVLDB 11(3):352–365. https://doi.org/10.14778/3157794.3157803
74. Zacharatou ET, Sidlauskas D, Tauheed F, Heinis T, Ailamaki A (2019) Efficient bundled spatial range queries. In: ACM SIGSPATIAL 2019, pp 139–148. ACM. https://doi.org/10.1145/3347146.3359077
75. Zhou T, Wei H, Zhang H, Wang Y, Zhu Y, Guan H, Chen H (2013) Point-polygon topological relationship query using hierarchical indices. In: 21st SIGSPATIAL international conference on advances in geographic information systems, SIGSPATIAL 2013, Orlando, FL, USA, Nov 5–8, 2013, pp 562–565. https://doi.org/10.1145/2525314.2527263