Evaluating Computational Geometry Libraries for Big Spatial Data Exploration

Yaming Zhang
yzhan737@ucr.edu
Computer Science and Engineering
University of California, Riverside
Riverside, California

Ahmed Eldawy
eldawy@ucr.edu
Computer Science and Engineering
University of California, Riverside
Riverside, California

ABSTRACT

With the rise of big spatial data, many systems were developed on Hadoop, Spark, Storm, Flink, and similar big data systems to handle big spatial data. At the core of all these systems, they use a computational geometry library to represent points, lines, and polygons, and to process them to evaluate spatial predicates and spatial analysis queries. This paper evaluates four computational geometry libraries to assess their suitability for various workloads in big spatial data exploration, namely, GEOS, JTS, Esri Geometry API, and GeoLite. The latter is a library that we built specifically for this paper to test some ideas that are not present in other libraries. For all the four libraries, we evaluate their computational efficiency and memory usage using a combination of micro- and macro-benchmarks on Spark. The paper gives recommendations on how to use these libraries for big spatial data exploration.

CCS CONCEPTS

• **Information systems → Spatial-temporal systems**; • **Theory of computation → Computational geometry**.

KEYWORDS

computational geometry, Java virtual machine, JVM, data exploration

ACM Reference Format:
Yaming Zhang and Ahmed Eldawy. 2020. Evaluating Computational Geometry Libraries for Big Spatial Data Exploration. In *Sixth International ACM SIGMOD Workshop on Managing and Mining Enriched Geo-Spatial Data (GeoRich'20), June 14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3403896.3403969

1 INTRODUCTION

Recently, there has been a notable rise in the amount of publicly available geospatial data from social networks, autonomous vehicles, and smart phones. As a result, many big geospatial data systems were developed on-top of existing big-data systems, Hadoop [11], Spark [32], Impala [10], Storm [21], and others.

All these systems were developed using existing computational geometry libraries. The two prominent ones are JTS [20] and Esri Geometry API [12]. Both are pure Java libraries which make them well-suited for most big data frameworks. JTS was developed way before any of the existing big data systems were developed but it remains to be an active project. Esri API was developed specifically for big data but did not have the long maintenance of JTS.

This paper evaluates four existing computational geometry libraries for exploratory analytics workloads. These workloads, have the following characteristics that we would like to evaluate.

- In-situ processing of textual data, e.g., in CSV or GeoJSON.
- Serialize/deserialize data in binary to move across machines.
- Scan a large amount of data in a streaming fashion.
- Keep a large amount of data in memory for analytic queries.
- Run a short and quick query on a small amount of data for new serverless systems such as AWS Lambda.

One of the main concerns that we wanted to address in this paper is that most big data platforms are built in Java or Scala which both run in the Java Runtime Environment (JRE). The JRE treats almost all variables as objects which raises a concern about the memory and processing overhead in handling all these objects, e.g., in the garbage collector. While hardcore database developers would prefer C++, Java claims to be very efficient in handling objects making it on-par or even better than C++. We can find that some big data systems, like Hadoop and AsterixDB, try to minimize object allocation to address this issue, while other systems, like Spark, are very generous in creating objects. Thus, we felt that this is something that needs to be addressed in details. In short, we found that JRE can indeed be very efficient but under specific circumstances as we details in this paper.

To test the above workloads, we run a set of micro- and macro-benchmarks on the four libraries, GEOS, JTS, Esri, and GeoLite and the results revealed some interesting findings. 1) When properly configured, JTS can provide the best balance between memory overhead and processing efficiency. 2) For streaming large data, Java libraries can be as efficient as C++. 3) The performance of Java seems to deteriorate quickly when a large number of active objects are kept. 4) For short queries, Java adds a notable overhead as compared to C++.

The full benchmark is publicly available at https://bitbucket.org/eldawy/geolitebenchmark revision `#aa23bbe`, tag '`georich20`' and the datasets are available at https://ucrstar.com.

The rest of this paper is organized as follows. Section 2 describes the related work. Section 3 details the micro- and macro-benchmarks. Sections 4 and 5 give the experimental setup and results, respectively. Finally, Section 6 concludes the paper.

GeoRich'20, June 14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-8035-5/20/06.
https://doi.org/10.1145/3403896.3403969

Table 1: Computational Geometry Libraries

Library	License	Language	Comments
JTS [20]	EPL	Java	
GEOS [15]	LGPL	C++	C++ port of JTS
Esri API [12]	Apache 2.0	Java	
GeoLite [9]	Apache 2.0	Java	
GeoTools [16]	LGPL	Java	Builds on JTS
GeoLatte [14]	LGPL	Java	
Spatial4J [28]	Apache 2.0	Java	Supports geodesic functions

Table 2: Big Spatial Data Systems

System	Base System	CG Library
PostGIS [24]	PostgreSQL	GEOS
SpatiaLite [29]	SQLite	GEOS
MonetDB [19]	N/A	GEOS
Vertexium [31]	Accumulo, Elasticsearch	JTS
GeoSpark [32]	Spark	JTS
Jena [4]	N/A	JTS, Spatial4J
GeoMesa [13]	Accumulo	JTS, Spatial4J
SpatialHadoop [11]	Hadoop	JTS, Esri Geometry API
Beast [8]	Spark & Hadoop	GeoLite, Esri, and JTS
Calcite [3]	N/A	Esri Geometry API
AsterixDB [1]	N/A	Esri Geometry API
Presto [25]	N/A	JTS, Esri Geometry API
Solr [5]	Lucene	Spatial4J
Elasticsearch [7]	N/A	Spatial4J
Titan [30]	N/A	Spatial4J

2 RELATED WORK

Computational Geometry (CG) Libraries: The Open Geospatial Consortium (OGC) defines an industry standard for representing and processing geometries for spatial database systems [22]. Therefore, big spatial data systems use existing computational geometry libraries to support this standard which includes Java Topology Suite (JTS) [20], Esri Geometry API [12], GeoTools [16], GeoLite [9], and GEOS [15] as summarized in Table 1. This paper focuses on the first four libraries. JTS and Esri are the most popular Java libraries. GEOS is a C++ port of JTS. Finally, we built GeoLite to test the functionality of mutable (reusable) objects which we could not find in the other libraries.

Big Spatial Data: Since the rise of big data, many systems were developed to extend these systems with big spatial data. In order to support SQL or SQL-like queries, these systems need to use one of the CG libraries mentioned in Table 1. Table 2 summarizes some of these systems to show which big data systems they extend and which computational geometry libraries they use. A base system of 'N/A' indicates that it is the core system that supports geospatial data not an extension. This paper can help the designers of these and other systems in choosing between the available computational geometry libraries.

Spatial Benchmarks: There has been some work in developing benchmarks for geospatial databases but none of them address the exploratory analytics workloads. Jackpine [26] proposes a benchmark for geospatial databases. Sequoia [2] is a storage benchmark that covers spatial data among others. BSD benchmark [27] focuses mainly on big spatial data systems. None of these benchmarks directly target the core computational geometry library used by any of these systems or the exploratory analytics workload that we consider in this paper.

3 BENCHMARKS

In this part, we would like to design a benchmark for exploratory analytic queries. These queries typically execute on raw files in *text format*, e.g., CSV. Some queries are *scan-based* and do not keep data in memory, and some are *iterative* queries that need to *store records in memory* and occasionally shuffle data between machines, e.g., clustering. Therefore, we propose the use of the following benchmarks.

3.1 Micro-benchmarks

create-polygons: This benchmark creates a set of random polygons to test the creation process.
wkt: Reads a CSV file and parses records represented in the Well-Known Text (WKT) standard format.
wkb: Parses a set of geometries from the Well-Known Binary (WKB) standard representation to test the efficiency of the shuffle.
point-count: Counts the total number of points in a set of in-memory geometries. This benchmark tests a simple CG operation that involves only integer calculations.
area: Computes the total area of a set of in-memory polygons. This benchmark tests a more complex CG operation that involves floating point calculations.

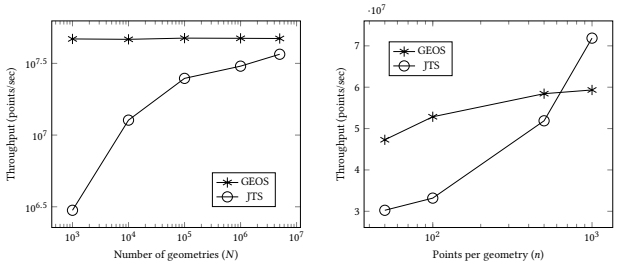
3.2 Macro-benchmarks

We also propose the following macro-benchmarks which are all implemented on Apache Spark.
rq: Runs a *range query* over a CSV input file with WKT geometries. This algorithm runs a simple scan-based algorithm for the input and counts the number of results.
sj: Runs *spatial join* between two CSV input files to find all intersecting polygons. This algorithm uses the Partition-based Spatial-merge (PBSM) algorithm [23] using a uniform grid of dimensions 360×180 over the world. It runs a plane-sweep algorithm in each partition and uses the reference-point to avoid duplicates [6]. It finally computes the intersection of each pair of records in the answer.

4 EXPERIMENTAL SETUP

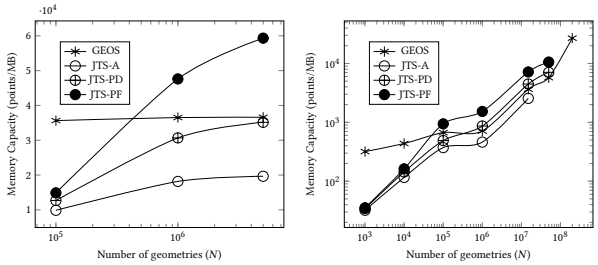
CG Libraries: For the micro-benchmarks, we consider one C++ library (GEOS 3.8.1) and three Java libraries (JTS 1.16.1, Esri Geometry API 2.2.3, and GeoLite 0.2.3).
Hardware Specification: We run all experiments on Amazon Web Services (AWS) with one master and 10 slaves all of type r5d.4xlarge which has 16 cores, 128 GB RAM, and 600GB SSD storage.
Datasets: For the *create-polygons* benchmark, we generate random polygons since we focus only on the performance not the shape of the polygon. For all other benchmarks, we use three real datasets. (1) `All-Objects` which contains 264M polygons with a size of 80GB. (2) `Parks` which contains 10M polygons with 7GB size. (3) `Lakes` which contains 377M polygons with 7GB size. All datasets are available on UCR-STAR [17].
Parameters: We use the following configurations some of which apply to specific libraries.

- **Number of geometries (N):** The number of polygons to create in the *create-polygons* benchmark, or the maximum number of geometries to process in other micro-benchmarks.
- **Points per polygon (n):** # of points to generate per polygon.



(a) Vary number of polygons (N) (b) Vary points per polygon (n)

Figure 1: Object creation throughput C++/Java



(a) Generated polygons (b) Real data

Figure 2: Memory capacity in C++ Vs Java

- **Discard:** When this Boolean flag is set, geometries are immediately discarded after they are generated/processed.
- **Reuse:** (Only for GeoLite and Esri) When this flag is set, the same geometry object is reused whenever possible to avoid creating too many objects.

Performance Metrics: We use two performance metrics, the *throughput (Θ)* in terms of number of processed points per second, and the *memory capacity (M)* in terms of number of points stored per megabyte. Both metrics consider the per-point performance to account for the varying sizes of geometries.

5 BENCHMARK RESULTS

This section presents the results of the benchmarks in three parts. First, we present the micro-benchmark results on GEOS and JTS to evaluate the difference between C++ and Java. Second, we present the micro-benchmark results on Java libraries. Finally, we present the results of the macro-benchmarks which are only done on Java.

We consider three configurations for JTS, JTS-A: which represents coordinates as an array of point objects, JTS-PD: which represents coordinates as a packed array of doubles, and JTS-PF: which represents coordinates as a packed array of floats. For clarity of the figures, we will only distinguish them when their results are significantly different. Otherwise, we will only present their average under the name 'JTS'.

5.1 C++ Vs Java

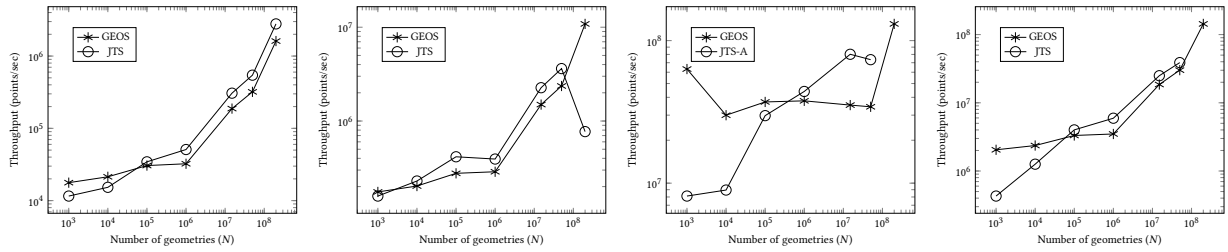
This part focuses on the comparison between C++ and Java. We compare JTS (in Java) and GEOS (its C++ port) which we think is a fair comparison since both are stable libraries that use the same algorithms. Our goal is to evaluate three types of overhead that JRE might introduce: (1) Object creation overhead, (2) Object memory storage, and (3) Processing overhead.

Object creation overhead. We use the *create-polygons* benchmark and measure the throughput as we vary number of polygons (N) and points per polygon (n). We enable the *discard* option to keep the memory usage low. First, in Figure 1(a) we vary N from 1,000 to 15 million and fix points-per-polygon $n = 50$. GEOS maintains a stable throughput regardless of the number of geometries. On the other hand, the throughput of JTS steadily increases as the number of geometries increases. This indicates a fixed overhead in JTS (or Java) which has a bigger effect for a few polygons but its effect diminishes as more polygons are created.

Second, in Figure 1(b) we fix number of polygons $N = 1M$ and vary n from 50 to 1,000. In this case, the throughput of both GEOS and JTS increases since the number of polygons objects remains fixed which reduces the overhead of creating these objects as compared to the total number of points created.

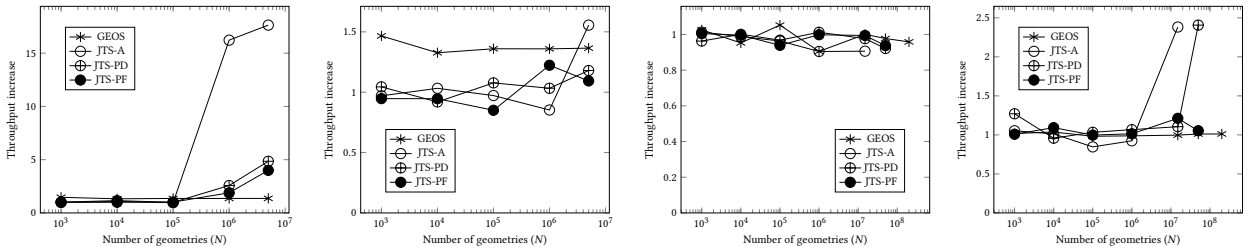
Memory Overhead on Objects. Figure 2(a) runs the *create-polygons* benchmark while keeping all objects in memory and measure the memory capacity in term of number of points per MB. Again, GEOS keeps a stable memory capacity while the capacity of JTS increases with the number of generated polygons N. This is due to the constant overhead of the JVM that diminishes when the data size gets bigger. JTS-A gives a much lower capacity due to the huge overhead of creating an individual object for each point. JTS-PD is almost similar to GEOS since it keeps coordinates in a primitive array of doubles. JTS-PF provides an even better capacity since it uses the less accurate 32-bit float data type. Figure 2(b) measures the memory capacity of the *wkt* benchmark when loading the real data `all-objects`. We observe an increase in throughput for all libraries since the latter polygons in the `all-objects` dataset tend to have more points per polygon which increases the memory capacity. However, as the data size increases even more, all JTS variations eventually run out of memory ($> 50GB$). JTS-A even fails on a smaller input due to the excessive number of objects that it creates.

JRE Performance Overhead. Figure 3 shows the four compute-centric benchmarks, namely, *wkt*, *wkb*, *point-count*, and *area*. We did not find a significant difference between the three variants of JTS so we only report the average for clarity. We expected that the compiled GEOS library in C++ would be much faster than the semi-interpreted JTS library in Java. However, we were surprised to find that JTS can often be more efficient. In particular, JTS is only more efficient when it works when a medium to large number of polygons, e.g., around 100K or more objects. This is because the fixed overhead that Java adds which diminishes as the data size increases. Additionally, when the number objects grows to fill most of the allotted memory, the garbage collector starts to be more aggressive and it slows down JTS again as compared to GEOS. In short, GEOS is more predictable but JTS can often be more efficient. In Figure 3(b), the performance of JTS drops at 200M polygons due to the additional memory needed to store the binary data.



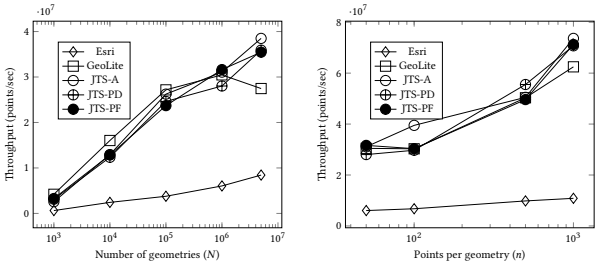
(a) WKT Parsing (b) WKB Parsing (c) Point Count (d) Area Computation

Figure 3: The computational performance (throughput) of GEOS (C++) and JTS (Java)



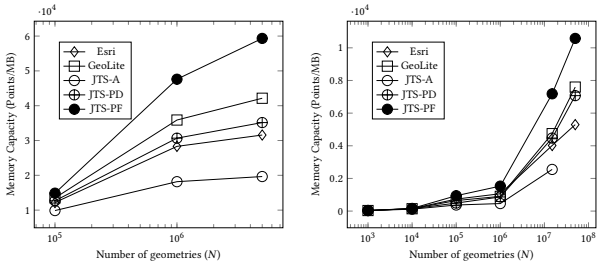
(a) Polygon creation. Xms=2G (b) Polygon creation. Xms=Xmx=50G (c) WKT Parsing. Xms=Xmx=50G (d) WKB Parsing. Xms=Xmx=50G

Figure 4: Effect of discarding objects on the throughput (Θ) in both C++ and Java



(a) Vary number of polygons (*N*) (b) Vary points per polygon (*n*)

Figure 5: Throughput of *create-polygons* for Java libraries



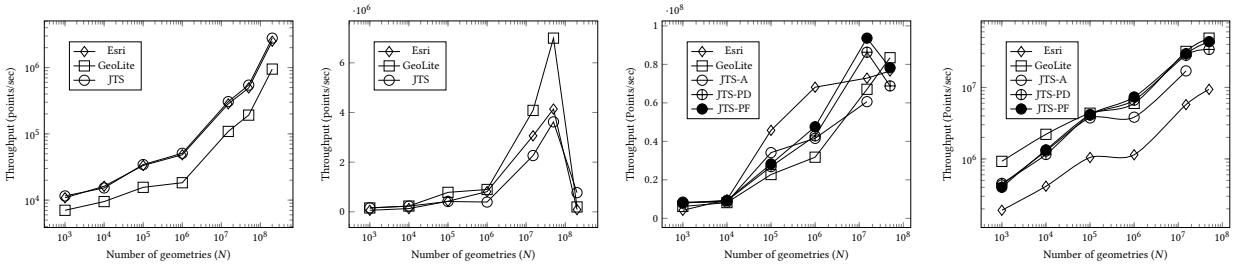
(a) Synthetic data ($n = 50$) (b) Real data

Figure 6: Memory capacity for Java libraries

Effect of Object Discard: In this part, we study the effect of discarding objects in both Java and C++. Java discards objects through the garbage collector while in C++ we have to destroy the object manually. We measure the ratio of throughput increase when we discard objects. If that ratio is higher than one, it indicates that the discard option improved the throughput.

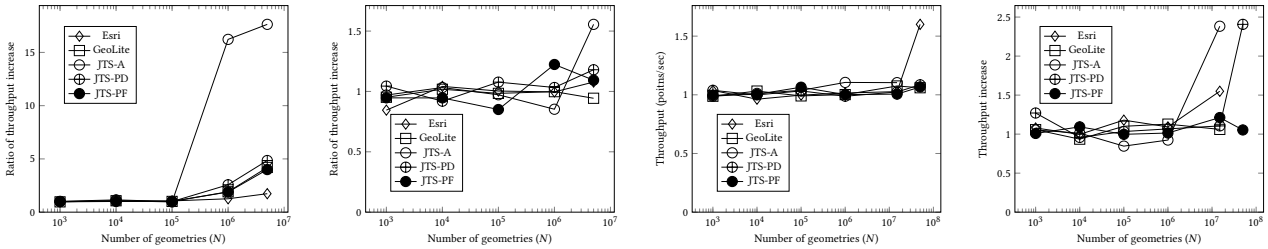
Figure 4(a) shows the throughput increase with the *create-polygons* benchmark when the memory of the JVM is initialized at 2GB (JVM option '-Xms=2g'). JTS runs faster as the input size increases since the garbage collector will need to deal with a huge number of objects when not discarded. GEOS also gains around 1.5x speedup since it uses less memory when objects are discarded. In Figure 4(b) we run the same experiment again but this time we initialize the JVM memory at 50GB, similar to the maximum memory allowed (JVM options '-Xms=50g -Xmx=50g'). In this case, there is no significant performance gain since the garbage collector will not need to run until the memory usage gets close to the 50GB limit. For the WKT and WKB benchmarks in Figures 4(c) and 4(d), respectively, no significant improvement was found since we initialize the memory to 50GB. However, for very large input that fills almost all the memory, the discard option yields a higher performance since it indicates less objects to maintain in the memory space.

Key findings: Java can indeed run as fast as C++ because it can handle its memory space more efficiently. However, the performance of Java degrades when the total number of active objects in memory becomes very large. Also, we found that the performance of C++ is much higher for both processing and memory efficiency for relatively small inputs which makes it more suitable for serverless processing, i.e., AWS Lambda.



(a) WKT parsing (b) WKB parsing (c) Point count (d) Area calculation

Figure 7: Throughput for compute-centric workloads in Java libraries



(a) Polygon creation (Xms=2g) (b) Polygon creation (Xms=Xmx) (c) WKT parsing (Xms=Xmx) (d) WKB parsing (Xms=Xmx)

Figure 8: Throughput increase when discarding objects immediately after creation in Java libraries

5.2 Performance of Java-based Libraries

Object Creation Overhead: Figure 5 shows the performance of the *create-polygons* benchmark as we vary the number of polygons and number of points per polygon. The only one that stands out is the Esri library which is significantly slower than all other libraries. Furthermore, the performance of JTS-A drops significantly when the number of objects is very high, e.g., 1000 points per polygon, due to the overhead of maintaining all these objects.

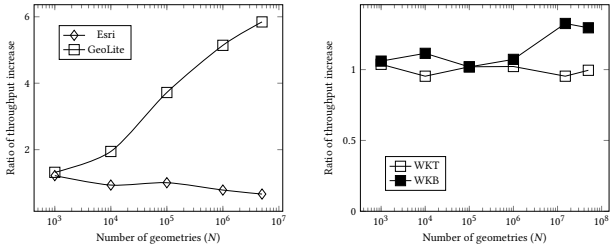
Memory Capacity: Figure 6 shows the memory capacity for storing generated polygons and real geometries. For both cases, JTS-A provides the lowest memory capacity due to the overhead of storing a lot of objects. The three other libraries, JTS-PD, Esri, and GeoLite, provide a higher capacity as they store coordinates as arrays of doubles. Finally, and as expected, JTS-PF provides the highest capacity since it uses 32-bit float coordinates. With real data in in Figure 6(b), JTS-A fails for 50M polygons since it runs out of memory. *All* of them fail to load 200M polygons in 50GB of memory.

Processing Efficiency: Figure 7 shows the results of the four compute-centric benchmarks, *wkt*, *wkb*, *point count*, and *area*. For WKT parsing in Figure Figure 7(a), both JTS and Esri provide the same throughput while GeoLite is significantly slower. The reason for that is that GeoLite uses a parser based on JavaCC which adds a significant overhead for detecting errors. The other two libraries use a hand-crafted parser which happens to work well for WKT and is also much faster. For WKB parsing (Figure 7(b)), GeoLite is slightly faster than other libraries due to the fewer objects it creates. As the input size increases, the performance of all libraries decreases due to the high memory usage which triggers the garbage collector more frequently.

Figure 7(c) shows the result of the point counting benchmark which is very simple and runs at almost the same speed for all libraries. However, JTS either fails or becomes slower for very large input. Finally, for area calculation (Figure 7(d)), Esri library was significantly slower than other libraries. When we investigated the code, we found that all libraries use the same algorithm but the slowdown of Esri algorithm was attributed to the use of the Kahan summation algorithm [18], which reduces the numerical error in adding up the floating point numbers.

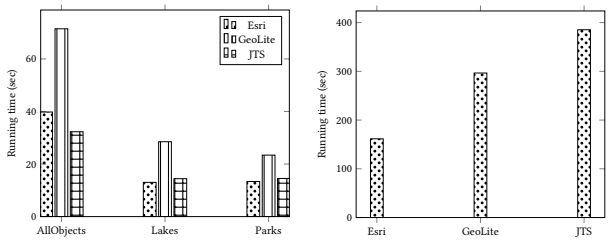
Effect of discarding objects: In Figure 8(a) we run the *create-polygons* benchmark while initializing the JVM memory at 2GB. In this case, we start to see a significant increase in throughput because the garbage collector needs to work with more polygons when the discard option is disabled. In Figure 8(b) we set the initial JVM memory to 50GB (similar to the maximum) which results in fewer runs of the garbage collector, hence, almost no benefit of discarding objecting. In Figures 8(c) and 8(d) we run the WKT and WKB, respectively. In both cases, enabling the discard option does not yield a significant improvement until the number of geometries is so large in which case the discard option saves in the overhead of the garbage collector.

Effect of reusing objects: This experiment verifies if reusing the Java objects and avoid creating additional objects is beneficial. This feature is only available in GeoLite and partially in Esri. Figure 9(a) runs the *create-polygons* benchmark and measures the increased throughput when reusing objects. Esri went a little slower when



(a) Polygon Creation (b) WKT/WKB parsing in GeoLite

Figure 9: Effect of object reuse in Esri and GeoLite



(a) Range Query (RQ) (b) Spatial Join (SJ) (parks⋈lakes)

Figure 10: Results of the macro-benchmarks

reusing objects while GeoLite became up-to 6x faster. Figure 9(b) shows the increased throughput with the WKT and WKB benchmarks. WKT showed no improvement but WKB showed a slight improvement. Overall, we think that the benefit of reusing objects is strongly tied to very light-weight processing which is another way for saying that object creation in Java is indeed very light-weight.

Key findings: JTS-PD seems to provide the best balance between computation and memory. While reusing objects can be beneficial, it needs to be applied only when running a light-weight workload.

5.3 Macro-benchmarks

Figure 10 shows the results of the two macro-benchmarks, range query (rq) and spatial join (sj), on Spark for the three Java libraries. Figure 10(a) shows the running time of *rq* on three input files, *lakes*, *parks*, and *all-objects*. JTS and Esri provided almost the same performance but JTS was slightly faster. On the other hand, for *sj* in Figure 10(b), Esri is apparently the fastest while JTS is the slowest. However, when we futher investigated this issue, we found that JTS produces significantly more results than Esri. It seems that the spatial logic is not similar, even though it is supposed to be exactly the same. Thus, we decided to make no conclusion for spatial join and we call upon the community (and ourselves) to make a further investigation to the logic of those CG libraries but this goes beyond the scope of this paper.

6 CONCLUSION AND FINAL REMARKS

This paper ran a set of micro- and macro-benchmarks on four computational geometry libraries, GEOS, JTS, Esri, and GeoLite. The goal is to find the most efficiently library for exploratory analytic workloads. When comparing C++ to Java, we found that C++ is significantly more efficient for very small and very large workloads. Among the Java libraries, we found JTS to provide the best balance between memory and computation. Finally, we found that these libraries differ in their spatial logic which is not supposed to be and we think that a thorough investigation can be done for the logic of these libraries.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (NSF) under grants IIS-1838222 and CNS-1924694 and by the USDA National Institute of Food and Agriculture, AFRI award number A1521.

REFERENCES

[1] Sattam Alsubaiee et al. 2014. AsterixDB: A Scalable, Open Source BDMS. *PVLDB* 7, 14 (2014), 1905–1916.
[2] Jean T. Anderson and Michael Stonebraker. 1994. SEQUOIA 2000 Metadata Schema for Satellite Images. *SIGMOD Rec.* 23, 4 (1994), 42–48.
[3] Apache. 2020. Apache Calcite: Dynamic data management framework. https://calcite.apache.org/.
[4] Apache. 2020. Apache Jena. https://jena.apache.org/.
[5] Apache. 2020. Apache Solr. https://lucene.apache.org/solr/.
[6] Jens-Peter Dittrich and Bernhard Seeger. 2000. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE*. IEEE Computer Society, San Diego, CA, 535–546.
[7] Elasticsearch. 2020. Elasticsearch. https://www.elastic.co/.
[8] Ahmed Eldawy. 2020. Beast: Big Exploratory Analytics for Spatio-temporal data. http://bitbucket.org/eldawy/beast/.
[9] Ahmed Eldawy. 2020. GeoLite: A light-weight computational geometry library. https://bitbucket.org/mehradae/beast/src/master/geolite/.
[10] Ahmed Eldawy et al. 2017. Sphinx: Empowering Impala for Efficient Execution of SQL Queries on Big Spatial Data. In *SSTD*. Springer, Arlington, VA, 65–83.
[11] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce Framework for Spatial Data. In *ICDE*. IEEE Computer Society, Seoul, South Korea, 1352–1363.
[12] Esri. 2020. Esri Geometry API. https://github.com/Esri/geometry-api-java.
[13] Anthony D. Fox et al. 2013. Spatio-temporal indexing in non-relational distributed databases. In *IEEE BigData*. IEEE Computer Society, Santa Clara, CA, 291–299.
[14] GeoLatte. 2020. GeoLatte: Open Source GIS Components for Java. http://www.geolatte.org/.
[15] GEOS. 2020. Geometry Engine, Open Source (GEOS). https://trac.osgeo.org/geos/.
[16] GeoTools. 2020. GeoTools The Open Source Java GIS Toolkit. https://www.geotools.org/.
[17] Saheli Ghosh et al. 2019. UCR-STAR: The UCR Spatio-Temporal Active Repository. *ACM SIGSPATIAL Special* 11, 2 (2019), 34–40.
[18] Nicholas J. Higham. 2002. *Accuracy and stability of numerical algorithms, Second Edition*. SIAM, Manchester, England.
[19] Stratos Idreos et al. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45.
[20] LocationTech. 2020. Java Topology Suite (JTS). https://locationtech.github.io/jts/.
[21] Ahmed R. Mahmood et al. 2015. Tornado: A Distributed Spatio-Textual Stream Processing System. *PVLDB* 8, 12 (2015), 2020–2023.
[22] OGC. 2020. Open Geospatial Consortium (OGC) Simple Feature Access. https://www.ogc.org/standards/sfs/.
[23] Jignesh M. Patel and David J. DeWitt. 1996. Partition Based Spatial-Merge Join. In *SIGMOD*. ACM Press, Quebec, Canada, 259–270.
[24] PostGIS. 2020. PostGIS: Spatial and Geographic Objects for PostgreSQL. https://postgis.net/.
[25] Presto. 2020. Presto: Distributed SQL Query Engine for Big Data. https://prestodb.io/.
[26] Suprio Ray et al. 2011. Jackpine: A benchmark to evaluate spatial database performance. In *ICDE*. IEEE Computer Society, Hannover, Germany, 1139–1150.
[27] Shashi Shekhar et al. 2012. Benchmarking Spatial Big Data. In *WKDB*, Vol. 8163. Springer, Pune, India, 81–93.
[28] Spatial4J. 2020. Spatial4j. https://projects.eclipse.org/projects/locationtech.spatial4j/.
[29] SpatiaLite. 2020. SpatiaLite. https://www.gaia-gis.it/fossil/libspatialite/index/.
[30] Titan. 2020. Titan: Distributed Graph Database. https://titan.thinkaurelius.com/.
[31] Vertexium. 2020. Vertexium. http://vertexium.org/.
[32] Jia Yu et al. 2015. GeoSpark: A Cluster Computing Framework for Processing Large-scale Spatial Data. In *SIGSPATIAL*. ACM, Bellevue, WA, 70:1–70:4.