LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data*

Mingjie Tang†, Yongyang Yu†, Qutaibah M. Malluhi‡, Mourad Ouzzani⋆, Walid G. Aref†

†*Purdue University,* ‡*Qatar University,* ⋆*Qatar Computing Research Institute, HBKU*

{tang49, yu163, aref}@cs.purdue.edu, qmalluhi@qu.edu.qa, mouzzani@qf.org.qa

ABSTRACT

We present LocationSpark, a spatial data processing system built on top of Apache Spark, a widely used distributed data processing system. LocationSpark offers a rich set of spatial query operators, e.g., range search, kNN, spatio-textual operation, spatial-join, and kNN-join. To achieve high performance, LocationSpark employs various spatial indexes for in-memory data, and guarantees that immutable spatial indexes have low overhead with fault tolerance. In addition, we build two new layers over Spark, namely a query scheduler and a query executor. The query scheduler is responsible for mitigating skew in spatial queries, while the query executor selects the best plan based on the indexes and the nature of the spatial queries. Furthermore, to avoid unnecessary network communication overhead when processing overlapped spatial data, We embed an efficient spatial Bloom filter into LocationSpark's indexes. Finally, LocationSpark tracks frequently accessed spatial data, and dynamically flushes less frequently accessed data into disk. We evaluate our system on real workloads and demonstrate that it achieves an order of magnitude performance gain over a baseline framework.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Spatial data management

1. INTRODUCTION

Spatial computing [15] is becoming significantly important with the proliferation of mobile devices. The growing scale and importance of location data have driven the development of numerous specialized spatial data processing systems, e.g., SpatialHadoop [9], Hadoop-GIS [5] and MD-Hbase [14]. By taking advantage of the power and cost-effectiveness of MapReduce [8], these systems typically outperform spatial extensions on top of relational database systems by orders of magnitude [5].

These MapReduce-based systems enable users to run spatial queries using predefined high level spatial operators without having to worry about fault tolerance and computation distribution. However, these systems do not leverage the power of distributed memory, and are unable to reuse intermediate data [17, 10]. Nonetheless, data reuse is very common in spatial data processing. For example, spatial datasets, e.g., OpenStreetMap (>60G) and Point of Interest (POI, for short, >100G) [9], are usually large. It is unnecessary to read these datasets continuously from disk (e.g., using HDFS) for each query. Meanwhile, intermediate query results have to be written back to HDFS, and this directly impedes further data analysis.

To tackle the above challenges, we introduce LocationSpark, an efficient spatial data processing system built on top of Apache Spark [17]. Spark is a distributed computation framework that allows users to work on distributed in-memory data without worrying about data distribution and fault-tolerance. LocationSpark is built as a library on top of Spark (see Figure 1). It provides spatial query APIs on top of the standard dataflow operators. LocationSpark requires no modifications to Spark, revealing a general method to combine spatial data processing within distributed dataflow frameworks.

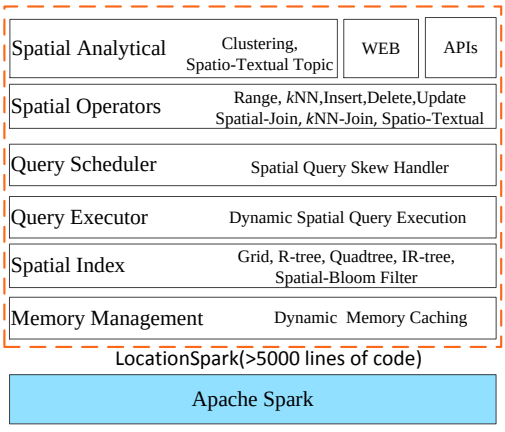


Figure 1: LocationSpark's layered architecture on top of Spark.

*This work is supported by QNRF Grant No. 4-1534-1-247 and National Science Foundation under Grant III-1117766.

Proceedings of the VLDB Endowment, Vol. 9, No. 10

Straightforward execution of spatial computations using Spark operators (i.e., transformations and actions) is inefficient for the following reasons: (1) the lack of spatial indexing, (2) the inability to handle spatial data skew, (3) the lack of spatial query optimization, and (4) the performance of unnecessary network communication due to spatial data overlap. To achieve performance speedup in LocationSpark, we introduce a range of optimizations. First, we build several types of global and local spatial indexes, e.g., Grid, R-tree, Quadtree, and IR-tree, to support efficient spatial queries over in-memory data. LocationSpark adopts immutable local spatial indexes, and achieves low overhead with fault-tolerance. Second, spatial data and queries are usually skewed. Hence, some data partitions receive more queries than others. Thus, LocationSpark has a query scheduler (including automatic skew analyzer and handler) to mitigate query skew. LocationSpark's query executor is responsible for choosing proper spatial algorithms based on the available spatial indexes and the registered queries. Skew is handled by distributing the load over the slave nodes. Furthermore, in order to reduce the communication cost when reading data that spans multiple partitions, LocationSpark uses a simple but efficient bloom filter (termed sFilter). sFilter can detect whether a spatial object is inside the spatial range or not. Finally, by exploring the access frequencies of spatial data, LocationSpark dynamically saves less frequently accessed data into HDFS, and this can reduce memory overhead and improve system performance. We evaluate LocationSpark using a real-world spatial dataset and compare it against a direct implementation, e.g., GeoSpark [11]. Our initial results show that LocationSpark can outperform GeoSpark by one order of magnitude.

2. RELATED WORK

Spatial data management has extensively been studied for decades. Sowell et al. survey iterative spatial-joins in memory [16]. Recently, there has been considerable interest in supporting spatial data management over Hadoop MapReduce [8]. Afrati and Ullman [4] have proposed a framework that computes a multi-join query in a single computation round. Liu et al. [13] study how to partition spatial data using Voronoi diagrams to speedup k-nearest-neighbor joins over MapReduce. Hadoop-GIS [5] supports spatial queries in Hadoop by using a uniform grid index. SpatialHadoop [9] builds global and local spatial indexes, and modifies the HDFS record reader to read data more efficiently. MD-Hbase [14] extends HBase to support spatial data updates and queries. Hadoop MapReduce is good at data processing for high throughput and fault-tolerance. Yet, Hadoop MapReduce has to write intermediate data into HDFS, and hence impedes the performance of applications that require pipelines of multiple MapReduce jobs.

Taking advantage of the very large memory pools available in modern machines, Spark [17] and Spark-related systems (e.g., Graphx [10], Spark-SQL [7] and DStream [18]) are developed to overcome the drawbacks of MapReduce in specific application domains. In order to process big spatial data more efficiently, it is natural to develop a novel and efficient spatial data management system based on Spark. Therefore, several prototypes support spatial operations over Spark, e.g., GeoSpark [11], SpatialSpark [3], Magellan [2], and GeoTrellis [1]. However, some important factors that impede system performance, mainly, query skew, and excessive and unoptimized network and I/O communication overheads are not addressed in these systems. Our demo system, LocationSpark, addresses these issues, and offers enhancements over the above systems.

3. OVERVIEW OF LocationSpark

3.1 Data Model and Data Types

LocationSpark stores spatial data as key-value pairs. A spatial tuple, say t_i, contains a spatial geometric key and a related value, namely k_i and v_i, respectively. The spatial data type of key k_i can be a two-dimensional point, e.g., latitude-longitude, a line-segment, a poly-line, a rectangle, or a polygon. The value type v_i can be specified by the user, e.g., a text type if the data tuple is a tweet.

3.2 Spatial Queries

LocationSpark supports spatial querying, spatial data updates, and spatial analytics. LocationSpark provides a rich set of spatial queries including spatial range, spatial kNN, spatial-join, and kNN-join. Moreover, it supports data updates and spatio-textual operations. Due to the importance of spatial data analysis, LocationSpark provides spatial data analysis functions including spatial data clustering, spatial data skyline computation and spatio-textual topic summarization. More complex spatial analysis functions can be added to the LocationSpark's framework.

3.3 Demonstration Overview

The demonstration consists of three different components, namely, basic spatial queries, join operators, and spatial analysis. For basic spatial queries, we showcase how to read large spatial data, and how users interact with the system by issuing basic spatial queries in a map-based or terminal interface. For spatial-join and kNN-join, users may join datasets from various sources that are either streamed or that are static. Finally, we show how to handle spatial analytics queries.

4. MAIN FEATURES OF LocationSpark

Figure 1 illustrates the architecture of LocationSpark. It adopts a layered design and implementation. We briefly explain key technical ideas within each layer.

4.1 Query Scheduler

Skew is a major issue that influences the runtime performance of parallel computations. Recently, SkewTune [12] has been proposed to deal with skew in the MapReduce platform, and AQWA [6] has been developed to address the spatial query skew in MapReduce platform. In this demonstration, we focus on two types of skew. The first type is unbalanced data partitioning for which we develop spatial indexes (Section 4.3) to partition spatial data in a balanced way. The second type is query skew, where some queries are unevenly distributed in space, and certain data partitions are overwhelmed by a small number of queries. Query skew is very common in spatial data processing, and is found to deteriorate the runtime performance of spatial queries.

In LocationSpark, we build a new layer, termed the query scheduler, to mitigate and deal with query skew. LocationSpark identifies potential hotspot data partitions by dynamically collecting statistical information from each partition

(i.e., number of queries and data points). Next, we develop a cost model to evaluate the overhead of repartitioning the hotspot partitions. Then, the scheduler can choose a set of partitions to be further reallocated to workers with an affordable cost. Therefore, execution on hotspot partitions and non-hotspot partitions can start to run at the same time, then query processing time is optimized by overlapping the processing of multiple jobs.

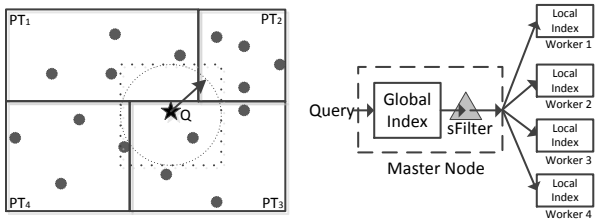
4.2 Query Executor

After spatial queries and related data partitions are scheduled into slave nodes, query executors execute specific query evaluation plans in slave nodes. For a spatial join, one data partition is usually associated with more than one spatial range query. It is unfeasible to execute a spatial range query over indexes separately for each query. Often, it makes sense to build an index over the queries (e.g., a set of range queries), and traverse the query index and the data index simultaneously. However, building query and data indexes is always computationally expensive, and requires high memory usage. LocationSpark evaluates the runtime and memory usage trade-offs for the various alternatives. Then, it chooses and executed the better execution plan on each slave node.

4.3 Spatial Indexing

LocationSpark builds two layers of spatial indexes (global and local) as given in Figure 2(b). The global index partitions data among the various nodes. To build a global index, LocationSpark samples the underlying data to learn the data distribution in space. Then, LocationSpark builds the global index to ensure that each data partition has the same amount of data. LocationSpark provides a grid and a region quadtree as the global index. In addition, each data partition has a local index. The type of the local index can be specified by users to match the needs of various application scenarios, e.g., a grid local index, an R-tree, a variant of the quadtree, or an IR-tree. Notice that in order to support data update, each version of spatial index can be persistent to disk for fault-tolerance. Thus, these spatial indexes are immutable and are implemented based on the path copy approach. By following the layered design of LocationSpark in Figure 1, it is convenient to add other types of spatial indexes in the future.

Spatial Bloom Filter. The Bloom filter is widely used for testing whether a data tuple is contained in a set or not. For spatial data processing, say processing a spatial range query, we need to retrieve that data partitions that overlap the query range. In Figure 2(a), the range query centered at Q overlaps four partitions. Typically, one can evaluate this query in one of two approaches can be used. The first approach is to replicate all points within Distance r from the boundary of each data partition into the neighboring partitions, where r is a parameter reflecting the radius of a query rectangle. Therefore, data tuples along the boundary (i.e., PT_1, PT_2, PT_3, PT_4 in Figure 2(a)) are duplicated to their neighboring partitions. The second approach is to broadcast query point Q into each overlapping partition. Next, a postprocessing step is followed to merge the query results returned from each partition. Both approaches mandate unnecessary network communication. In this demonstration, we propose a new data structure, termed spatial bloom filter (sFilter, for short). sFilter can answer whether a spatial



(a) (b)

Figure 2: Spatial Bloom filter in LocationSpark. (a) Range query Q overlapping four partitions, (b) sFilter along with the global index.

point is contained inside a spatial range or not. sFilter is embedded into the global spatial index of LocationSpark as illustrated in Figure 2(b). This can save unnecessary network communication.

4.4 Memory Management for Spatial Data

Memory is a precious resource for distributed in-memory data management systems. To save memory, Spark allows users to choose different storage levels. For example, the storage level MEMORY_AND_DISK can persist data into disk when data cannot fit into memory. Then, the system can read persistent data from disk when needed. But this policy does not consider query access patterns, and is not efficient for spatial data. For example, consider the POI dataset for the USA with more than 100GB. Certain partitions, e.g., Chicago or San Francisco, are queried more frequently than the other partitions. To detect and deal with this situation, access frequencies and corresponding time stamps are recorded in the spatial index. Then, LocationSpark detects the frequently accessed data by aggregating access frequencies. Finally, LocationSpark dynamically caches frequently accessed data into memory, and stores the less frequently used data into disk.

5. DEMONSTRATION AND EVALUATION

We demonstrate LocationSpark on real-world spatial data analytics scenarios, illustrating its applicability, ease of use, and performance.

5.1 Demonstration Scenarios

LocationSpark is demonstrated on two real spatial datasets. The first dataset is Twitter tweets that are gathered over a period of nearly 20 months (from January 2013 to July 2014). The size of the tweets dataset is 1.5 Billion tweets totaling around 250 GB. The second dataset is OpenStreetMap that represents the map features of the whole world, where each spatial object is identified by its coordinates (longitude, latitude) and an object ID. The OpenStreetMap dataset contains 1.7 Billion points and takes 62.3 GB of disk space.

Below, we illustrate how Scala commands work for the basic spatial queries based on LocationSpark. The attendees load the spatial data and then build a spatial RDD. Next, the attendees can execute various spatial queries e.g., range queries, kNN queries, or spatial join over the loaded data. In addition, user-defined functions (UDFs) are used to filter tuples during spatial query processing. Finally, attendees can run spatial data analysis queries (e.g., density-based clustering or spatio-textual topic detection) based on the provided APIs.

LocationSpark is open-source, and can be downloaded from https://github.com/merlintang/SpatialSpark. More examples can be found in the codebase.

```
val datardd=spark.textFile(datafile) // Load spatial data
val spatialRDD = SpatialRDD(datardd).cache() // Generate SpatialRDD
val box = Box(23.10094f,-86.8612f, 32.41f, -85.222f)
spatialRDD.rangeFilter(box, udf) // Spatial range query with udf
val k=100
val Q=Point(21.10334f,-86.3332f)
spatialRDD.knnFilter(Q,k,udf)// kNN Query
val rectangles=spark.textFile(sjoinqueryfiles)
spatialRDD.sjoin(rectangles) // Spatial-join
val knnqueries=spark.textFile(knnqueryfiles)
spatialRDD.knnjoin(knnqueries, k) // kNN-join
```

Code 1: Basic Spatial Queries in LocationSpark.

	Twitter(Seconds)		OpenStreetMap(Seconds)	
	k=100	k=200	k=100	k=200
PGBJ	3422	3549	5588	5668
LocationSpark	284	345	420	574

Table 1: Performance of kNN-join in LocationSpark and PGBJ.

5.2 Performance

This part of the demo will center around LocationSpark's performance. Attendees can observe the system performance and resource utilization on a range of input datasets and cluster scenarios. Experiments are conducted on a cluster[1] that consists of 6 Dell compute nodes with two 8-core Intel E5-2650v2 CPUs, 32 GB of memory, and 48TB of local storage per node for a total cluster capacity of 288TB. The Spark version is 1.5.0 with Yarn cluster resource management.

We compare the performance of LocationSpark with other spatial systems, mainly using techniques from GeoSpark [11] over Spark, and SpatialSpark [3]. Attendees of VLDB can see the details of system performance comparison from various aspects including query response time, memory usage, and data shuffling cost. In Table 1, we report performance results for kNN-join on the Twitter and OpenStreetMap datasets. LocationSpark is compared with a state-of-art kNN-join approach. LocationSpark performs up to an order-of-magnitude better than PGBJ. The speedup in LocationSpark is attributed to the various optimizations in Section 4.

6. REFERENCES

[1] Geotrellis. `https://github.com/geotrellis/geotrellis`.

[2] Magellan. `https://github.com/harsha2010/magellan`.

[3] Spatialspark. `http://simin.me/projects/spatialspark/`.

[4] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. Technical report, National Technical University of Athens, Stanford University, December 2009.

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

[6] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. AQWA: adaptive query-workload-aware partitioning of big spatial data. *PVLDB*, 8(13):2062–2073, 2015.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *SIGMOD '15*, pages 1383–1394, New York, NY, USA, 2015. ACM.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04*. USENIX Association, 2004.

[9] A. Eldawy and M. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE'15*, pages 1352–1363, April 2015.

[10] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI'14*, pages 599–613, Broomfield, CO, Oct. 2014. USENIX Association.

[11] M. S. Jia Yu, Jinxuan Wu. Geospark: A cluster computing framework for processing large-scale spatial data. In *ACM SIGSPATIAL'15*, Seattle, WA.

[12] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *SIGMOD '12*, pages 25–36, New York, NY, USA, 2012. ACM.

[13] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proc. VLDB Endow.*, 5(10):1016–1027, June 2012.

[14] S. Nishimura, S. Das, D. Agrawal, and A. Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *MDM'12*, volume 1, pages 7–16, 2011.

[15] S. Shekhar, S. K. Feiner, and W. G. Aref. Spatial computing. *Commun. ACM*, 59(1):72–81, 2016.

[16] B. Sowell, M. V. Salles, T. Cao, A. Demers, and J. Gehrke. An experimental analysis of iterated spatial joins in main memory. *Proc. VLDB Endow.*, 6(14):1882–1893, Sept. 2013.

[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12*, pages 15–28, San Jose, CA, 2012. USENIX.

[18] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP'13*, pages 423–438, New York, NY, USA, 2013. ACM.

[1]https://www.rcac.purdue.edu/compute/hathi/