Spatio-Temporal Join on Apache Spark

Randall T. Whitman, Michael B. Park, Bryan G. Marsh, and Erik G. Hoel
Environmental Systems Research Institute (Esri)
{rwhitman, mpark, bmarsh, ehoel}@esri.com

ABSTRACT

Effective processing of extremely large volumes of spatial data has led to many organizations employing distributed processing frameworks. Apache Spark is one such open-source framework that is enjoying widespread adoption. Within this data space, it is important to note that most of the observational data (i.e., data collected by sensors, either moving or stationary) has a temporal component, or timestamp. In order to perform advanced analytics and gain insights, the temporal component becomes equally important as the spatial and attribute components. In this paper, we detail several variants of a spatial join operation that addresses both spatial, temporal, and attribute-based joins. Our spatial join technique differs from other approaches in that it combines spatial, temporal, and attribute predicates in the join operator.

In addition, our spatio-temporal join algorithm and implementation differs from others in that it runs in commercial off-the-shelf (COTS) application. The users of this functionality are assumed to be GIS analysts with little if any knowledge of the implementation details of spatio-temporal joins or distributed processing. They are comfortable using simple tools that do not provide the ability to tweak the configuration of the algorithm or processing environment. The spatio-temporal join algorithm behind the tool must always succeed, regardless of input data parameters (e.g., it can be highly irregularly distributed, contain large numbers of coincident points, it can be extremely large, etc.). These factors combine to place additional requirements on the algorithm that are uncommonly found in the traditional research environment. Our spatio-temporal join algorithm was shipped as part of the GeoAnalytics Server [9], part of the ArcGIS Enterprise 10.5 platform.

CCS CONCEPTS

• **Information Systems → Database management → Database applications**; *Spatial databases and GIS*; • **Information Systems → Information systems applications → Spatial-temporal systems**; *Geographic information systems*; • **Information Systems → Database management system engines → Database query processing**; *Join algorithms*;

KEYWORDS

Spatial join, spatio-temporal join, Hadoop, Spark, HDFS, distributed processing.

ACM Reference format:
Randall T. Whitman, Michael B. Park, Bryan G. Marsh, and Erik G. Hoel. 2017. Spatio-Temporal Join on Apache Spark. In *Proceedings of SIGSPATIAL'17, Los Angeles, CA, USA, November 7–10, 2017*, 10 pages.
https://doi.org/10.1145/3139958.3139963

SIGSPATIAL'17, November 7–10, 2017, Los Angeles Area, CA, USA

ACM ISBN 978-1-4503-5490-5/17/11…$15.00
https://doi.org/10.1145/3139958.3139963

1 INTRODUCTION

A spatial join [16] is an operation that takes two datasets of multi-dimensional objects in Euclidean space, and finds all pairs of objects satisfying the specified spatial relation between the objects (e.g., intersection or containment). More formally, given two spatial datasets R and S, the result of a spatial join of R and S is defined as:

$$R \bowtie_{pred} S = \{(r,s) | r \in R, s \in S, pred(r,s) = true\}$$

where *pred* is the spatial or spatio-temporal relationship.

A spatio-temporal attribute join incorporates spatial, temporal, and aspatial attribute relationships into the join condition. For example, given point locations of crime incidents with a time, join the crime data to itself, specifying a spatial relationship of crimes within 1 kilometer of each other, and that also occurred within 1 hour of each other, to determine if there are a sequence of crimes close to each other in space and time.

Apache Spark [32] is an open-source, distributed, in-memory computing framework and architecture. Spark enables developers to author software that can run across clusters of distributed machines with inherent data-parallelism and fault tolerance. Spark was developed to overcome inefficiencies related to the MapReduce programming model. MapReduce programming utilizes a linear dataflow with distributed programs; MapReduce programs read input data from disk, map a function across the data, reduce (combine) the results of the mapping operation, and store the reduced results to disk. This becomes problematic when working with algorithms that require sequences of MapReduce operations; there is considerable overhead when reading/writing intermediary results. Spark attempts to pipeline the processing of data partitions, keeping the intermediary results in memory, thus affording considerable opportunities for performance enhancement relative to the traditional MapReduce programming model. Although Spark is designed to exploit in-memory computing, it can accommodate reading and writing data from distributed storage when memory is exhausted; this also is utilized to support fault tolerance.

Spatial joins have been widely studied in both the standard sequential environment [16], as well as in the parallel [4, 15] and distributed environments [1]. For over twenty years, algorithms have been developed to take advantage of parallel and distributed processing architectures and software frameworks. The recent resurgence in interest in spatial join processing is the results of newfound interest in distributed, fault-tolerant computing frameworks such as Apache Hadoop, as well as the explosion in observational and IoT data.

With distributed processing architectures, there are two principal approaches that are employed when performing spatial joins. The first, termed a broadcast spatial join, is designed for joining a large dataset with another small dataset (e.g., political boundaries). The large dataset is partitioned across the processing nodes and the complete small dataset is broadcast to each of the nodes. This allows significant optimization opportunities. The second approach, termed a partitioned (or binned) spatial join is a more general technique that is used when joining two large datasets. Partitioned joins use a divide-and-conquer approach [2]. The two large datasets are divided into small pieces via a spatial decomposition, and each small piece is processed independently.

The rest of this paper is organized as follows. Section 2 reviews related work in the domain of distributed spatial joins. Section 3 describes our algorithm for spatial join using both broadcast and bin-based (or partition-based) approaches. In addition to describing the join algorithm, we present details of how to further optimize the process through direct summarization, pre-aggregation, and reference-point de-duplication. Section 4 details extending the spatial join algorithm to support temporal and aspatial components in the join (again, using both broadcast and bin-based approaches). Section 5 presents some detailed performance analyses of the spatial join algorithms on very large real-world datasets using a cluster containing hundreds of cores. Concluding remarks are contained in Section 6.

2 RELATED WORK

Our previous work [28] implemented range and distance queries and *k*-NN, but not spatial join. Also it was implemented on Hadoop MapReduce rather than Spark.

SJMR (Spatial Join with MapReduce) introduced the first distributed spatial join on Hadoop using the MapReduce programming model [33]. The algorithm, which did not employ a spatial index, evenly split the input datasets into disjoint partitions during the Map stage. This was accomplished using a tile-based spatial partitioning function. The algorithm then joined the partitions at the Reduce stage using a plane sweeping algorithm. The authors considered this approach to be a spatial analog to hash-merge and sort-merge.

SpatialHadoop [7] optimized SJMR with a persistent spatial index (it supports grid files [20], R-trees [12], and R+trees [23]) that is pre-computed. SpatialHadoop was observed to be significantly faster computing spatial joins than SJMR. Mansour Raad has published an implementation of SJMR that uses on-the-fly spatial indexing rather than persistent indexing [22]. Each of these spatial join optimizations of SJMR are implemented on Hadoop MapReduce and do not utilize the in-memory architecture of Spark.

Hadoop-GIS [2], which is utilized in medical pathology imaging, features both 2D and 3D spatial join. The originally-published work was implemented on Hadoop MapReduce and Hive. More recently, research is also underway on Spark, with SparkGIS [3].

GeoSpark [31] is a framework for performing spatial joins, range, and *k*-NN queries. The framework supports quadtree [11] and R-tree indexing of the source data. Global or partitioning index is a regular grid; with local spatial indexing. Experiments showed that the local spatial indexing improved spatial join performance. In addition, GeoSpark was observed to outperform the MapReduce-based SpatialHadoop.

Magellan [24] is an open source library for geospatial analytics that uses Spark. It supports a broadcast join and is integrated with Spark SQL for a traditional, SQL user experience. Local computations are performed using the Java API in GIS Tools for Hadoop [8]. It has recently had indexing added to it to boost performance. Magellan also provide some coordinate transformation capabilities, most notably, transforming between WGS84 and the NAD83 State Plane coordinate systems.

SpatialSpark [30] supports both a broadcast spatial join and a partitioned spatial join on Spark. The partitioning is supported using either a fixed-grid, binary space partition, or a sort-tile approach. The authors demonstrated good scalability and performance gains relative to more traditional Impala-based implementation (Apache Impala [17] is a distributed SQL processing engine).

STARK [13] is a Spark-based framework that supports spatial joins, *k*-NN, and range queries on both spatial and spatio-temporal data. STARK supports three temporal operators: contains, containedBy, and intersects). It also supports the DBSCAN density-based spatial clusterer [10]. Differing from other frameworks in this domain, it retains the attribute data when resolving queries. STARK supports both a fixed grid and a binary space partitioner. STARK was observed to outperform GeoSpark and SpatialSpark when configured with live R-tree-based indexing and a binary space partitioner.

MSJS (multi-way spatial join algorithm with Spark [6]) addresses the problem of performing multi-way spatial joins using the common technique of cascading sequences of pairwise spatial joins [18, 26]. The authors showed that their Spark-based approach outperforms other MapReduce-based multi-way spatial join algorithms (e.g., Hadoop-GIS) on a modestly sized cluster. This is not a surprising result given the architectural optimizations that Spark-based algorithms offer over MapReduce-based approaches (i.e., in-memory optimizations and decreased I/O requirements).

Simba [29] offers range, distance (circle range), and *k*-NN queries as well as distance and *k*-NN joins. Two-level indexing, global and local, is employed, similar to the various indexing work on Hadoop MapReduce. Partitioning uses Sort-Tile-Recursive (STR) by default, but allows custom partitioning. For the local index, Simba introduces an IndexedRDD with in-partition fast random access. The local index consists of an in-memory and optionally persisted R-tree or kd-tree index. Spark core is untouched, but Simba makes changes to SparkSQL. Simba focuses

on distance join rather than spatial join. Point geometries are supported now; support for Line and Polygon geometries is awaiting future work. Simba optimizes spatial queries, but does not mention spatio-temporal queries.

LocationSpark [25] supports range query, *k*-NN, spatial join, and *k*-NN join. It claims to outperform GeoSpark by an order of magnitude. LocationSpark uses global and local indices - Grid, R-tree, Quadtree and IR-tree. LocationSpark stores spatial data in key-value pairs in which the key is the geometry. It does not claim support for near (distance) join, nor spatio-temporal join.

3 SPATIO-TEMPORAL JOIN

We employ two primary methods for spatiotemporal joins. The type of join used for any given operation is dependent on the effective size of both datasets.

3.1 Broadcast Join

Broadcast join is analogous to a map-side join in MapReduce programming [27]. It is the preferred method when at least one of the datasets can fit entirely into memory on each of the Spark executors (processing elements in the Spark architecture). Whether a dataset fits into memory depends on characteristics of the data (record count, attribute count, and vertex counts of the geometries), Spark configuration, and hardware (physical memory).

The first step in Broadcast Join is to estimate the size of one or both datasets to determine which one is smaller and can fit into memory. The cost of this step depends on several factors. The source of the data may provide an efficient way to estimate the size of the dataset, such as record counts or disk space utilization. However, often the data source provides no such information or the join is part of a larger Spark pipeline (a sequence of Spark transformations, which convert one RDD to another) and the cost of performing a count approximation cannot be determined.

Broadcast Join

```
 1:  val toBroadcast = determineSmallerDataset()
 2:  val distributed = the other input dataset
 3:  val broadcasted = spark.broadcast(toBroadcast.collect())

 4:  distributed.mapPartitions { partitionFeatures =>
 5:    val qt = BuildQuadtreeIndex(broadcasted.value)
 6:    partitionFeatures.map { partFeature =>
 7:      qt.query(partitionFeature.geometry).flatMap {
 8:        broadcastFeature =>
 9:          if (relationship(partFeature, broadcastFeature))
10:            Some((partFeature, broadcastFeature))
11:          else
12:            None
13:      }
14:    }
15:  }
```

Once the broadcast dataset has been identified, it is collected locally and then distributed in its entirety to each executor. The opposite dataset is distributed to partitions across the executors.

Each partition in the distributed dataset is joined with the broadcasted dataset. This join occurs local to each executor. In the case that the join has a spatial relationship, an in-memory quadtree index [8, 19] is generated. This index is generated lazily so that it is built only once per executor. The temporal relationship is applied after the index is queried and the spatial relationship has been applied. If no spatial relationship exists, the indexing step is not necessary.

3.2 Bin Join

Bin Join (as contrasted with Broadcast Join) is analogous to a reduce-side join in MapReduce programming [33]. This method is used when both datasets are too large to fit into memory and a partitioned approach is required.

PairRDD.join:

PairRDD[K,U], PairRDD[K,V] → PairRDD[K, (U,V)]

K1	U1a	V1a
K1	U1a	V1b
K1	U1b	V1a
K1	U1b	V1b

PairRDD.cogroup:

PairRDD[K,U], PairRDD[K,V] → PairRDD[K, (Iter[U],Iter[V])]

K1	<u1a,u1b…>	<u1a,u1b…>
K2	<u2a,u2b…>	<v2a,v2b…>

Figure 1: Spark cogroup, as contrasted to Spark join method.

A bin is a location in space (XY) and/or time (T) that serves as a key for Spark's cogroup operation. In Spark, cogroup is an operation on a pair of key-value RDDs that produces for each key, a data pair consisting of a collection of the left-side values and a collection of the right-side values for that key (see Figure 1). Features that intersect a bin are grouped together; this results in every feature from that bin ending up on the same partition. A feature can however intersect multiple bins and be duplicated across multiple partitions. For near join, features are assigned to bins near the feature (within the near radius), including bins that do not necessarily intersect the feature geometry.

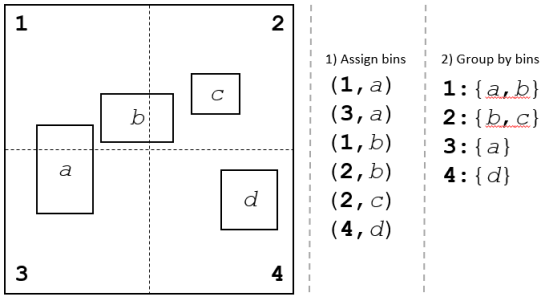

Figure 2: Space-only binning of some rectangles.

The shape of a bin is defined separately for XY and T. The XY shape (see Figure 2) is determined by one of two grid implementations as described below. T is defined as a temporal interval (e.g., 1 second, 1 day, 1 month, etc.) and is equivalent to an extrusion of the XY shape (see Figure 3). In the 3D representation in the figure, *a* is green, *b* is red, *c* is yellow, and *d* is blue.

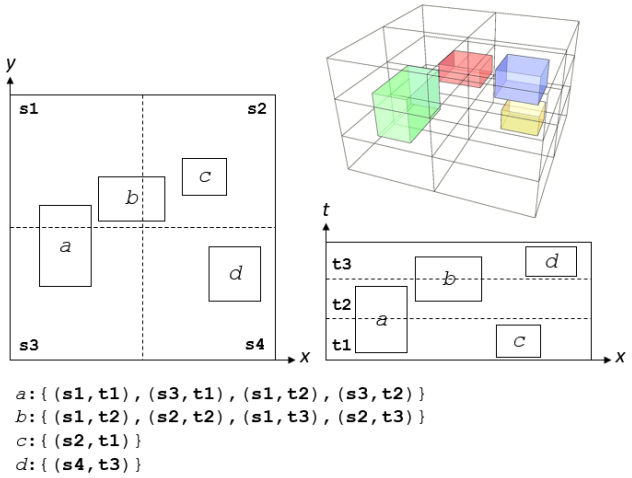

Figure 3: Spatio-temporal binning of the four rectangles shown in Figure 2 (this example presumes the rectangles have a temporal component).

When joining two datasets, the same binning operation is applied to both sides. This ensures that features from the left side and right side of the join are grouped together in the same bins. Once the features are grouped together, a local join is used to further filter the related features. Like the Broadcast Join, if a spatial relationship is defined, a quadtree index can be built on either side of the join to further filter the spatial relationships.

Bin Join

```
1:  val binningMesh = construct regular-grid or quadtree mesh
2:  val binnerLeft = spark.broadcast(
       makeBinner(binningMesh, timeCycle, padSpace, padTime))
3:  val binnerRight =
       spark.broadcast(makeBinner(binningMesh, timeCycle))
4:  val leftWithBins = targetDataset.flatMap ( feat =>
       binnerLeft.value.bins(feat).map{(_,feat)} )
5:  val rightWithBins = joinDataset.flatMap ( feat =>
       binnerRight.value.bins(feat).map{(_,feat)} )
6:  val groupedByBin = leftWithBins.cogroup(rightWithBins)

7:  groupedByBin.flatMap {
       case (binId, (targFeats, joinFeats)) =>
8:    val targIndex = buildQuadtreeIndex(targFeats)
9:    for {
10:      joinFeature <- joinFeats
11:      targFeature <- targIndex.query(joinFeature)
12:      if relationshipTest(targFeature, joinFeature)
13:      if isUnique(binId, targFeature, joinFeature)
14:    } yield (targFeature, joinFeature)
15: }
```

A final de-duplication step may be required to remove duplicate matches that occur when features end up in multiple bins. We apply reference-point de-duplication [5, 7] – which accomplishes de-duplication with no cross-node communication and no post-pass – with generalizations and optimizations. For a pair of candidate geometries, a target geometry and a join geometry (which have already been determined to intersect), choose a reference point. It is a mostly arbitrary choice, but must be well-defined given a geometry pair (A, B), so that the same point would be chosen for inputs (A, B) in any partition across the shared-nothing, distributed architecture. The chosen reference point will be associated with exactly one bin. In the bin that owns the reference point, yield the candidate feature pair as a result; but in any other bin not owning the reference point, that may have the feature pair as a candidate, it is ignored as a duplicate. The reference point commonly corresponds to the lower left corner of the rectangle formed from the intersection of the two minimum bounding rectangles (or MBRs) of the two geometries. We however assign only those bins that intersect the geometry itself, which is commonly a strict subset of the bins intersecting the MBR. We do not choose the corner of the MBR intersection, but instead a point of the intersection of the geometries themselves. For de-duplication in spatio-temporal Bin Join, reference-point de-duplication can be applied separately in the temporal and spatial dimensions. The spatio-temporal object constitutes an extruded prism (in X-Y-T space) rather than an arbitrary pseudo-3D shape. Another variant of de-duplication arises with Near Join, in which the join relation is defined as the distance between the two geometries, and/or the time difference, being at most a specified limit. For geometries A and B, whose distance is known to be at most *d*, choose a reference point. It suffices to choose a point of A that is at most distance *d* from B (the "point of A" can be any point intersecting A, not necessarily a vertex).

Aggregation of point data to polygons, is more common than one-to-many join, at least with very large data volumes. Direct-summarization means accumulating the summary statistics during the cogroup callback, rather than materializing the one-to-many results first (and summarizing afterward). Bin overload (a bin with many coincident points – e.g., as is found in the GDELT dataset – Global Database of Events, Language, and Tone), is an issue no matter how small the cell size. This problem can be resolved by pre-aggregating before direct-summarization. Pre-aggregate coincident points into a micro-cell, then collapse the micro-cell to a point, in order to continue operating on it as a point dataset.

There are several things to consider when choosing the best binning method. Spatial distribution can affect the number of features that fall into any given bin. A dense area can result in a large number of features assigned to some bins. Processing the features assigned to an overloaded bin, consumes a large amount of memory – resulting in straggler tasks, or, in the worst case, failures.

Spatial distribution becomes less of a problem when a temporal relationship is specified (see Figure 4). Dense areas in space (e.g., the southeast corner) are often distributed more evenly in time, resulting in more reasonably sized bins.

Geometry size is another consideration. A large polygon or long line has the potential to intersect more bins than a point. The more bins that a geometry falls into, the more times it will be duplicated which may adversely affect memory usage and system performance.

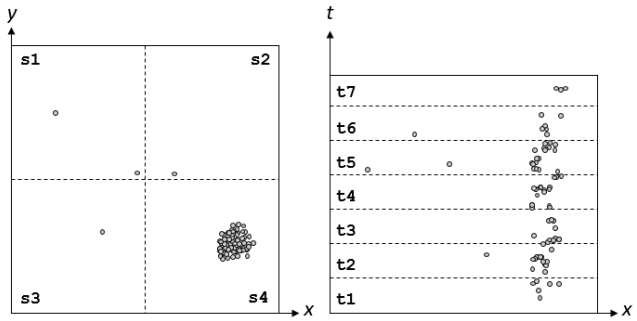

Figure 4: Time slicing in spatiotemporal-join binning alleviates the bin overload that would occur in space-only join with skewed dense data.

It may not always be cost effective to try to determine the characteristics of a dataset. If the join appears near the end of a long Spark pipeline, the cost of sampling the dataset may be equal to the cost of evaluating the entire pipeline. Furthermore, each dataset will have different characteristics and both need to be considered.

Two grid types (regular and adaptive) are employed to define the XY dimensions of a bin. A grid implementation has one primary function: given a geometry, return the set of grid cells that the geometry intersects.

The simplest implementation is a regular grid. A regular grid can be computed on the fly without considering the characteristics of either dataset. However, choosing an optimal grid cell size can have a significant impact on join performance. A larger cell size results in less duplication of features across bins, but suffers from overloaded partitions in spatially dense areas of the dataset. A smaller grid cell size generally decreases the partition sizes, but can cause large geometries to be duplicated more. In some cases, especially with space-only joins, regular-grid Bin Join will fail, irrespective of the cell size.

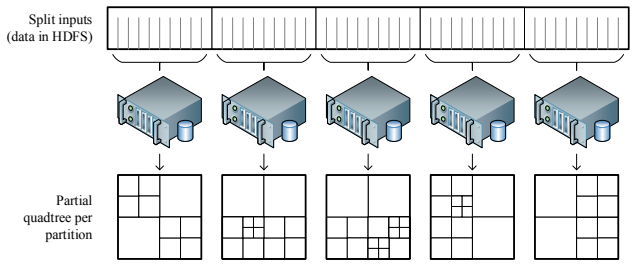

Figure 5: Example highlighting the splitting of the source data into equal sized collections (step 1) and the partial quadtrees being built on each processing element.

A more sophisticated approach is a quadtree-based grid, as in the partitioning of VegaGiStore [34]. Using a PMR quadtree decomposition of space [19], each quadrant becomes a cell in the grid. Building the quadtree requires a pre-pass through the data, but results in an adaptive grid where dense areas have smaller cells, and sparse areas have larger cells. Performance in the average case is generally equal or better than the regular grid, even when accounting for the cost of data pre-pass. In the worst case (e.g., datasets with extreme skew between dense and sparse areas), the quadtree-based grid performs better and succeeds, where the regular grid may fail due to severe memory pressure.

A PMR quadtree is built in a distributed manner using the Spark API. The quadtree mesh differs from a quadtree index in that it is used only to partition space, not to look up individual spatial objects in searches afterwards. This means that only the sequence of quadrants is of interest, and the entries of the quadtree mesh can be key-only rather than key-value.

The implementation of building the quadtree mesh is adapted from our previous quadtree build algorithm that used the MapReduce programming model [28]. The process of building the PMR quadtree in parallel across a collection of Spark partitions consists of the following five steps:

1. Split the unorganized source data into evenly sized collections, physically distributing one collection to each processing element in the cluster.
2. On each processing element, build a partial quadtree on the data in the partition.
3. Assign and distribute each quadrant in each partial quadtree to a processing element for later assembly of the complete quadtree, by active spatial-block partitioning.
4. On each processing element, combine the received unsorted and potentially overlapping quadrants into a locally consistent/valid quadtree.
5. The ordered collection of spatially-partitioned quadtree portions, constitutes complete global PMR quadtree; filter down to keys only by dropping the value from the key-value pairs.

The native splitting capability of the Hadoop system is used to split the unordered source data into equal sized collections (step 1), for data stored in HDFS; for other data sources, the input driver needs to support partitioning/splitting. The number of collections/partitions is not constrained to be equal to the number of processing elements (nodes, CPU cores, etc.) in the cluster.

Using the Spark API, we build a PMR quadtree on each partition of the input, similar to the Map phase of our index on Hadoop MapReduce (step 2). This is done for each partition independently, resulting in a set of partial quadtrees, one for each partition (see Figure 5). At maximum depth, the value of the quadtree entry is the count only, rather than a collection of MBRs. Each partial quadtree is converted from a Java TreeMap to a collection of linear quadtree [11] entries. The entries of the linear quadtree have a composite key consisting of a Peano key of the quadrant, and the node depth. Note that input objects whose bounding rectangle intersects more than one quadrant, contribute to multiple quadrants in the quadtree (similar to the R+-tree). At maximum depth, the key-count pair generates one linear quadtree entry only, and the value of the key-value pair is the count. Entries at non-max depth generate a linear quadtree entry for each MBR.

A custom partitioner is defined whose purpose is to assign each entry in a partial linear quadtree to the processing element that will combine the quadrants of the partial quadtrees into subtrees of a

complete quadtree index (step 3). The partitioner assigns the entry to the [pre-]partition for which the Peano key of the entry lies within the [pre-]partition boundaries. Preliminary partitions are defined by splitting the extent into the desired number of partitions, by alternating horizontal and vertical splits, like a PR k-d tree decomposition [21], with splitting-direction alternation order to be compatible with the N-order of the quadtree being built.

In order to maintain the invariant that every leaf quadrant is wholly contained within a pre-partition/shard-region - and ensure that every object is sent to the correct partition when building the quadtree - shard regions (partitions) are composed from quadrants at sharding depth - and the sharding depth can be at most the minimum depth for insertion into the quadtree.

Unlike Hadoop MapReduce, sorting is not inherent in the Spark operation; we would have to call a sort method if we would want a streaming merge sort exactly as done in our index on MapReduce. Instead, we forego the sort, and insert the entries in a region-partial quadtree (in other words, partitioned spatially - step 4) - this time not data-partial (that is, split arbitrarily by the input driver) - but rather the complete data for one quadrant or subtree out of the whole tree (see Figure 6). Due to having built the interim data-partial quadtrees, the linear quadtree entries we need to merge, contain a first-pass approximation to the depth at the entry will lie in the final quadtree. Thus, building the quadtree-region is in effect a hybrid between quadtree insertion and quadtree bulk-loading [14].

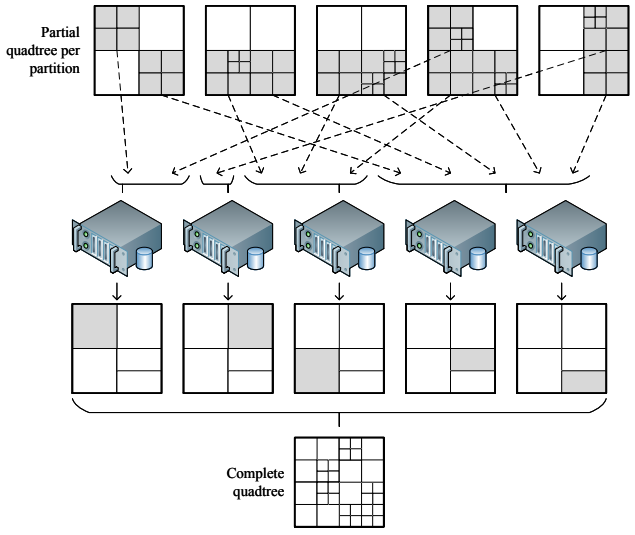

Figure 6: Example showing entries from the partial quadtrees being shuffled and merged on the collection of processing elements. The combined result (the complete quadtree) is also shown.

Finally, we filter the linear quadtree down to keys only, by keeping only the key from the key-value pairs (step 5). Thus, the size of the entire quadrant mesh can be reduced to fit in memory, such that we can broadcast it for use in the join phase of Quadtree-mesh Bin Join.

Strictly speaking, it would be possible for the last stage of the quadtree build (step 4) to build a quadtree spatial-portion without the data-partial interim quadtree (step 1) having been built beforehand. The interim quadtree entry provides a first-pass approximation to the depth at which each entry will lie in the final, complete quadtree, thus lessening the amount of quadrant splitting that must be done at the latter stage.

It is important to note that the PMR quadtree can be readily optimized when one realizes that much of the data in the "big data" space is generated by stationary sensors, moving objects, etc. Essentially, these produce features/observations with point geometry. Polygonal data in this space is less common, other than in the filtering role (e.g., legal geometries like census tracts, zip code or city boundaries, etc.). Because of this, it is possible to further optimize the quadtree index record to utilize a point geometry rather than a bounding rectangle as part of the value. This makes a substantial difference in the memory consumed when building a quadtree on a big dataset.

After building the quadrant mesh, it is broadcast as an array of quadrants, for use in binning. In binning, we search for quadrants, not for indexed spatial objects. We search the quadrant array, to yield a collection of quadrants that intersect the search geometry. The search is done by a standard quadtree-search algorithm (which returns quadrants, without having to proceed to look up individual spatial objects). When binning the geometry of a target feature, any absent/implicit empty quadrants, can simply be ignored, thus early-filtering out the target features that are disjoint with all the non-empty quadrants (that were generated from the join-side features).

3.3 Other Design Considerations

When developing software for commercial off-the-shelf (COTS) applications, there are many additional design considerations that impact the spatio-temporal join algorithm (and algorithms in general). Commercial software applications in the GIS domain assume that the users of the functionality are GIS analysts and not computer scientists. GIS analysts generally have modest (or better) experience with scripting languages such as Python, but have little knowledge of the implementation details of spatio-temporal joins, much less distributed processing. They are comfortable using straightforward tools (see Figure 7) that generally do not provide the many opportunities to tweak the configuration of the algorithm or processing environment (note – in certain domains such as geostatistics, GIS analysts and spatial statisticians are provided with this capability and are comfortable doing so).

Given the intended user (a normal GIS analyst), the spatio-temporal join algorithm behind the tool must always succeed, regardless of input data parameters (e.g., it can be highly irregularly distributed, contain large numbers of coincident points, it can be extremely large, etc.). These factors combine to place additional requirements on the algorithm that are uncommonly found in the traditional research environment. This results in more sophisticated binning techniques that can accommodate irregular distributions of data (e.g., the PMR quadtree, or a BSP tree). In addition, failover techniques may be employed. For example, if memory exhaustion

occurs during a broadcast join, we failover to a more expensive, but robust bin join.

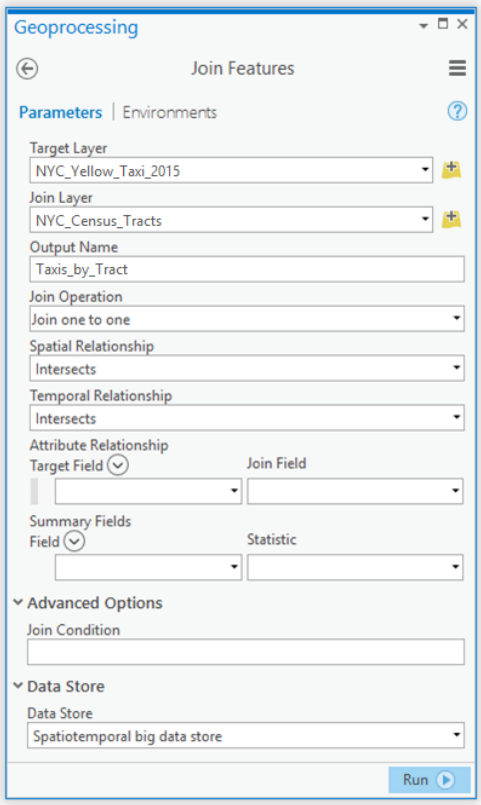

Figure 7: Example of an easy-to-use spatio-temporal join tool (taken from ArcGIS Pro).

4 PERFORMANCE COMPARISON

In order to test scalability and runtime, we used the New York City Taxi and Limousine Commission's taxi dataset, http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml. It consists of about 1.2 billion records from 2009-2015, detailing both trip and fare data for every taxi trip recorded in the seven boroughs of New York City. The data is available for download as a collection of CSV files. The trip data contains information such as hack license, pickup date/time, dropoff date/time, passenger count, trip duration, trip distance, and pickup and dropoff latitude/longitude. The fare data contains hack license, pickup date/time, fare, tip amount, tolls, and total trip cost. For the purposes of benchmarking, outliers at the origin were filtered out, but in reality that did little to avoid the impact of outliers, because the taxi data contains other unrealistic outliers also.

All tests were performed on a Spark cluster with 18 Worker nodes (144 cores) and 16 GB of memory per executor. Results were calculated and counted, then discarded without persisting – in real-world use of the product, writing the output can also take substantial time.

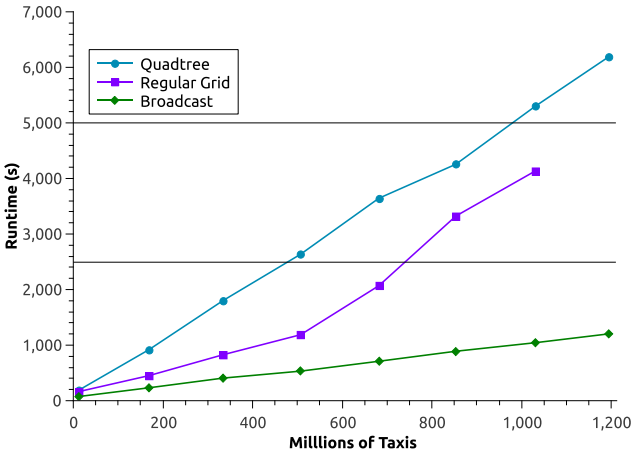

Figure 8: Joining NYC taxis to census tracts.

We compared our algorithms and variants in joining various sizes of data to a polygon dataset consisting of 2166 census tracts in the New York City area (see Figure 8). In this case Bin Join ran faster with regular-grid mesh than with quadtree mesh, up to about a billion points. (In development we have seen some cases where regular grid is faster and other cases where quadtree is faster.) However, quadtree bin join scaled farther than regular-grid bin join, due to adapting to the skew of the point data. Broadcast Join outperformed Bin Join because only one dataset is large, and the polygon dataset is of modest size. Our conjecture is that Broadcast Join would scale indefinitely in the size of one dataset, while Bin Join with quadtree mesh would drop off at some point, in like manner as, albeit at a higher feature count (and higher density) than, with regular grid.

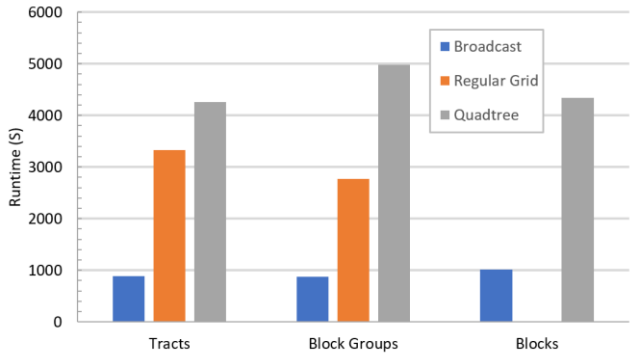

Figure 9: Joining ~700 million NYC taxis (five years) with Census geographies (~2200 census tracts, 6500 block groups, and 38,800 blocks).

Next we joined with a fixed number of taxi points (about 700M taxis in five years of data), to a varying number of polygons. Specifically, the polygons are the successively more local polygon meshes of the U.S. Census Bureau: approximately 2200 census tracts, 6500 block groups, and 38,800 tabulation blocks in the New York City area (Figure 9). It was possible to broadcast all 38,800 block polygons, and Broadcast Join ran faster

than either variant of Bin Join, for all three polygon datasets. Regular-grid Bin Join ran faster than quadtree-mesh Bin Join with the census tracts and block groups, but ran out of memory with the census blocks. With quadtree, Bin Join succeeded on all three Census polygon datasets. Our conjecture is that upon obtaining or generating larger polygon datasets, Quadtree-mesh Bin Join would scale to a larger polygon input dataset than would Broadcast Join.

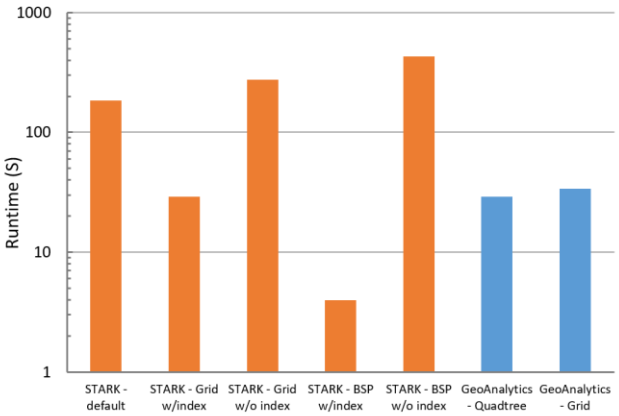

Figure 10: NYC taxi self join (1M taxis subset).

We prepared benchmarks to compare GeoAnalytics spatial join to STARK, which had published results of outperforming GeoSpark and SpatialSpark. Figure 10 shows the runtime of a self join, with a relation of geometry intersects, on a 1-million feature subset of the taxi data, using five variants of the STARK algorithm and two variants of our Bin Join, namely using regular-grid and quadtree mesh respectively. At this modest data size, STARK with BSP partitioning and indexing ran fastest, followed by the two variants of our spatial join implementation. We attempted the same comparison with larger subsets of the taxi data, including specifically a one-month subset, containing 12 million points. With BSP partitioning in STARK, we were unable to get the STARK job to launch on the 12-million feature self join (in several attempts). With the default algorithm, and with regular grid (with or without live indexing), STARK ran for over 30 minutes before our tester killed the job. Our Bin Join with a quadtree mesh completed successfully in 73 seconds.

We benchmarked space-only near join against Simba, which refers to it as distance join (Figure 11). The near radius used was 20 meters. Up to a size of about 5M taxis, Simba outperformed GeoAnalytics on this space-only near self join; but Simba was unable to complete the join with more than about 7M taxis. (Recognizing that Simba published results of 10M × 10M distance join with OSM data, we conjecture that the dropoff at a lower point count results from the greater density and skew of the taxi data.) GeoAnalytics scaled to a space-only near join with 100M taxis (Figure 12).

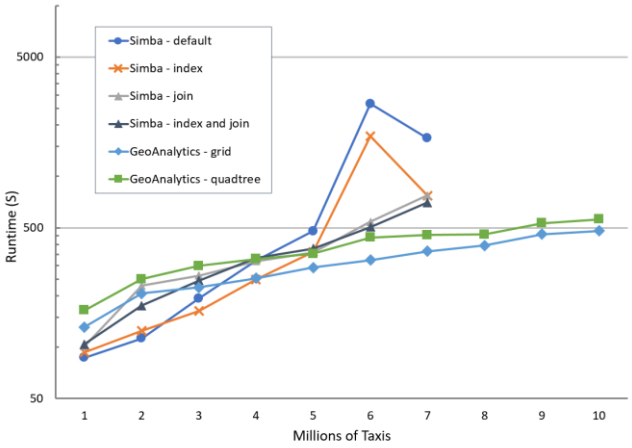

Figure 11: Simba and GeoAnalytics – space-only near join (note that Simba is run in four configurations, with and without providing tuning parameters for number of index and/or join partitions).

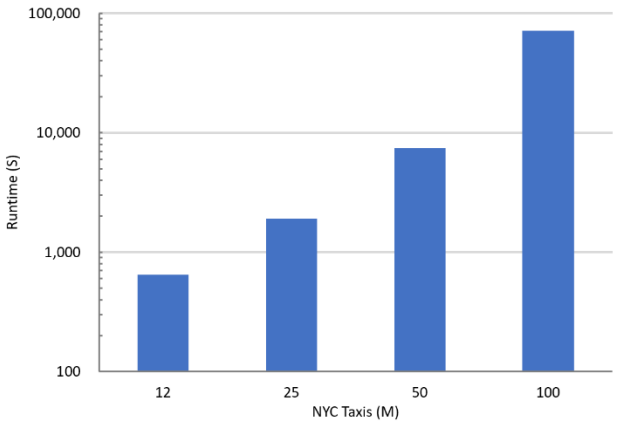

Figure 12: GeoAnalytics space-only self join with quadtree.

Figure 13 shows a benchmark of the scalability of our Spatio-temporal Join, on a self join with a spatial near radius of 20 meters and a temporal near span of 10 minutes. Our join is able to scale past a billion, in the size of **both** inputs, without tuning parameters - on commodity hardware with modest memory. The quadtree mesh outscales (and outperforms) the regular grid on such size and skew. Interestingly enough, spatio-temporal join scales past space-only join, because the temporal condition alleviates spatial skew. To our knowledge, the spatio-temporal bin join with quadtree mesh in ArcGIS GeoAnalytics is the only general spatial join implementation that can complete a spatio-temporal near join on inputs where both datasets exceed 1 billion points.

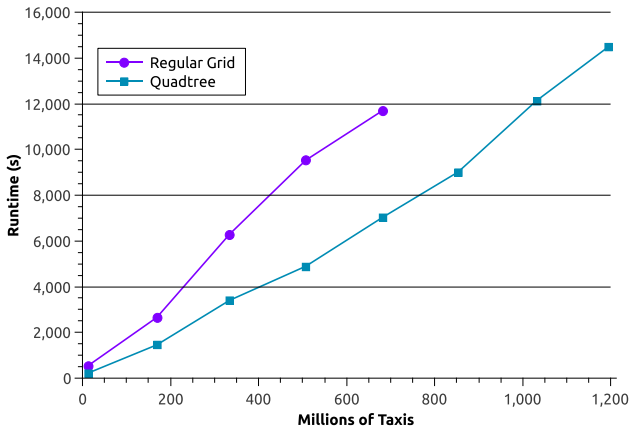

Figure 13: NYC taxis spatio-temporal near self join.

5 CONCLUSION AND FUTURE WORK

In this paper, we have detailed a Spark-based implementation of a spatio-temporal attribute join that runs in a distributed manner across a Hadoop cluster. We have demonstrated robust, scalable, and performant spatial join – in a comprehensive architecture that is usable by real-world GIS analysts who need not be experts in algorithms or spatial join.

Also, our results demonstrate that the most effective and efficient distributed spatial join algorithm depends on the characteristics of the two input datasets. Broadcast Join is generally fastest when one of the datasets is modest in size (and only one is large), but cannot complete when both data sets are large. Among different binning techniques for Bin Join, regular-grid mesh is faster for some inputs – but quadtree mesh is faster for other inputs, and in fact the quadtree mesh can succeed in some cases where the regular grid would run out of memory.

We believe that our spatial join scales to greater dataset size than other implementations of spatial join on Spark, having benchmarked with STARK and Simba.

An interesting observation is that spatio-temporal near join was able to scale to larger input sizes than space-only near join, because the temporal condition alleviates the effects of spatial skew.

Further work will yet further improve the robustness, scalability, and performance of our spatio-temporal join. Compressing or clipping polygon geometries may hasten the shuffle between binning and local refinement.

For near join, we can benchmark the effect of near radius, as well as the already-benchmarked input size.

In one-to-many space-only join on input with highly-coincident points, bins can be fragmented non-spatially. The stacked points would be assigned to only one bin fragment, and features of the other dataset would have to be sent to all fragments of the bin. Counts of features by bin could determine the necessity of fragmenting.

For bin join with quadtree mesh, the quadtree mesh can be built on a sample of the larger dataset, rather than on its entirety – in which case the sample would have to adequately reflect the skew of the dataset. The benefit would be reducing the time to build the quadtree mesh, and possibly at least as important, potentially reducing the size of the interim quadtree as it is being built, and reducing the risk of running out of memory during the quadtree build phase. It would entail foregoing the early filtering of disjoint target features - instead it would be necessary to include implicit quadrants in the search result during binning, in case the quadrant was in fact non-sampled rather than empty (empty with the sample but non-empty with the entire dataset).

A more sophisticated join planner/optimizer could choose better the join algorithm or variant to use for a specific pair of input datasets. It could make a more aggressive, but necessarily accurate, determination of whether broadcast join can be utilized. For bin join, it could choose between regular grid and quadtree mesh based on characteristics of the input datasets. A hints-enabled planner/optimizer could use information that a GIS Analyst would likely know about the data - such as order-of-magnitude feature count and some idea of uniformity versus skew - without requiring the user to have any expertise about algorithms nor implementation.

Acknowledgements

We wish to thank other key development staff within Esri that have contributed to this work through discussion and critical feedback. These people include Bill Moreland, Lauren Bennett, Mansour Raad, Mark Janikas, Sarah Ambrose, Sergey Tolstov, and Sud Menon.

REFERENCES

[1] Abel, D. J., Ooi, B. C., Tan, K.-L., Power, R., and Yu, J. X. 1995. Spatial join strategies in distributed spatial DBMS. In *Advances in Spatial Databases – 4th International Symposium, SSD'95*, vol. 1619 of Springer-Verlag Lecture Notes in Computer Science. Portland, ME, 348–367.

[2] Aji A., Wang, F., Vo H.,Lee, R., Liu, Q., Zhang, X. and Saltz, J. 2013. Hadoop-GIS: a high performance spatial data warehousing system over mapreduce. In *Proceedings of the VLDB Endowment,* 6 (11), (pp. 1009-1020).

[3] Baig F., Mehrotra M., Vo H., Wang F., Saltz J., Kurc T. 2016. SparkGIS: efficient comparison and evaluation of algorithm results in tissue image analysis studies. In *Biomedical Data Management and Graph Online Querying*. Big-O(Q) 2015, DMAH 2015. Lecture Notes in Computer Science, vol 9579. Springer.

[4] Brinkhoff, T., Kriegel, H. P., and Seeger, B. 1996. Parallel processing of spatial joins using r-trees. In *Proceedings of the 12th International Conference on Data Engineering* (pp. 258-265). IEEE.

[5] Dittrich, J. P., and Seeger, B. 2000. Data redundancy and duplicate detection in spatial join processing. In *Data Engineering, 2000. Proceedings of the 16th International Conference on* (pp. 535-546). IEEE.

[6] Du, Z., Zhao, X., Ye, X., Zhou, J., Zhang, F., and Liu, R. 2017. An effective high-performance multiway spatial join algorithm with spark. *ISPRS International Journal of Geo-Information*, 6 (4): 96.

[7] Eldawy, A., and Mokbel, M. F. 2015. SpatialHadoop: a mapreduce framework for spatial data. In *Data Engineering*

(ICDE), 2015 IEEE 31st International Conference on (pp. 1352-1363). IEEE.

[8] Esri. 2013. GIS Tools for Hadoop. https://github.com/Esri/gis-tools-for-hadoop (referenced 2017/06).

[9] Esri. 2016. ArcGIS GeoAnalytics Server. http://server.arcgis.com/en/server/latest/get-started/windows/what-is-arcgis-geoanalytics-server-.htm (referenced 2017/06).

[10] Ester, M., Kriegel, H. P., Sander, J., and Xu, X. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96),* pp. 226-231.

[11] Gargantini, I. 1982. An effective way to represent quadtrees. *Communications of the ACM* 25, 12 (December 1982), 905-910. DOI= http://doi.acm.org/10.1145/358728.358741.

[12] Guttman, A. 1984. "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, (pp. 47-57).

[13] Hagedorn, S., Götze, P., and Sattler, K. U. 2017. The STARK framework for spatio-temporal data analytics on spark. In *Proceedings of the 17th Conference on Database Systems for Business, Technology, and the Web (BTW 2017),* Stuttgart, Germany, March 2017.

[14] Hjaltason, G., and Samet, H. 2002. Speeding up construction of PMR quadtree-based spatial indexes. VLDB Journal, 11, 2 (October 2002), 109-137. DOI=http://dx.doi.org/10.1007/s00778-002-0067-8.

[15] Hoel, E. and Samet, H. 1994. Data-parallel spatial join algorithms. In *Proceedings of the 23rd International Conference on Parallel Processing*. Vol. 3. St. Charles, IL, 227–234.

[16] Jacox, E. H., and Samet, H. 2007. Spatial join techniques. *ACM Transactions on Database Systems (TODS)*, 32(1), 7.

[17] Kornacker, M., and Erickson, J. 2012. Cloudera Impala: Real Time Queries in Apache Hadoop, For Real. http://blog. cloudera. com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real.

[18] Mamoulis, N. and Papadias, D. 2001. Multiway spatial joins. *ACM Transactions on Database Systems* 26, 4 (Dec.), 424–475.

[19] Nelson, R. C., and Samet, H. 1986. A consistent hierarchical representation for vector data. In *ACM SIGGRAPH Computer Graphics,* 20 (4), pp. 197-206). ACM.

[20] Nievergelt, J., Hinterberger, H., and Sevcik, K. C. 1984. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS),* 9(1), 38-71.

[21] Orenstein, Jack A. "Multidimensional tries used for associative searching." Information Processing Letters 14.4 (1982): 150-157.

[22] Raad, M. 2013. BigData Spatial Joins, Blog post. http://thunderheadxpler.blogspot.com/2013/10/bigdata-spatial-joins.html (referenced 2017/06).

[23] Sellis, T. K., Roussopoulos, N., and Faloutsos, C. 1987. The R+-tree: a dynamic index for multi-dimensional objects, in *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB),* (pp. 507-518).

[24] Sriharsha, R., 2015. Magellan: geospatial analytics on spark, https://hortonworks.com/blog/magellan-geospatial-analytics-in-spark/ (referenced 2017/06).

[25] Tang, MingJie, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani and Walid G. Aref. "LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data." PVLDB 9 (2016): 1565-1568.

[26] Valduriez, P., and Gardarin, G. 1984. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems (TODS)*, 9(1), 133-161.

[27] White, T. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc., Sebastopol, CA, USA.

[28] Whitman, R. T., Park, M. B., Ambrose, S. M., and Hoel, E. G. 2014. Spatial indexing and analytics on hadoop. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (pp. 73-82). ACM. DOI=http://dx.doi.org/10.1145/2666310.2666387.

[29] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). ACM, New York, NY, USA, 1071-1085. DOI: https://doi.org/10.1145/2882903.2915237

[30] You, S., Zhang, J., and Gruenwald, L. 2015. Large-scale spatial join query processing in cloud. In *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on* (pp. 34-41). IEEE.

[31] Yu, J., Wu, J., and 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* (p. 70). ACM.

[32] Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., and Stoica, I. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing* (HotCloud '10), Boston, June 2010.

[33] Zhang, S., Han, J., Liu, Z., Wang, K., and Xu, Z. 2009. SJMR: Parallelizing spatial join with mapreduce on clusters. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE international conference on* (pp. 1-8). IEEE.

[34] Zhong, Yunqin, Jizhong Han, Tieying Zhang, Zhenhua Li, Jinyun Fang and Guihai Chen. "Towards Parallel Spatial Query Processing for Big Spatial Data." 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (2012): 2085-2094.