Distributed Spatial and Spatio-Temporal Join on Apache Spark

RANDALL T. WHITMAN, BRYAN G. MARSH, MICHAEL B. PARK, and ERIK G. HOEL, Esri, USA

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. To perform advanced analytics and gain insights, the temporal component becomes equally important as the spatial and attribute components. In this article, 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 [12], part of the ArcGIS Enterprise platform from version 10.5 onward.

CCS Concepts: • **Information systems** → **Spatial-temporal systems**; **Geographic information systems**; *Join algorithms*; • **Computing methodologies** → **Distributed algorithms**;

Additional Key Words and Phrases: Spatial join, spatio-temporal join, Hadoop, Spark, HDFS, distributed processing, geospatial and spatiotemporal databases

ACM Reference format:
Randall T. Whitman, Bryan G. Marsh, Michael B. Park, and Erik G. Hoel. 2019. Distributed Spatial and Spatio-Temporal Join on Apache Spark. *ACM Trans. Spatial Algorithms Syst.* 5, 1, Article 6 (June 2019), 28 pages.
https://doi.org/10.1145/3325135

1 INTRODUCTION

A spatial join [23] 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). More formally, given two spatial datasets R and S, the result of a spatial join of R and

Authors' addresses: R. T. Whitman, B. G. Marsh, M. B. Park, and E. G. Hoel, Esri, 380 New York Street, Redlands, CA, 92373; emails: {rwhitman, bmarsh, mpark, ehoel}@esri.com.

2374-0353/2019/06-ART6 $15.00
https://doi.org/10.1145/3325135

S is defined as:

$$R \bowtie_{pred} S = \{(r,s) | r \in R, s \in S, pred(r,s) = true\}, \quad (1)$$

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 1km 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 [45] 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 [23], as well as in the parallel [7, 22] and distributed environments [1]. For over 20 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 result 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 a 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 [4]. The two large datasets are divided into small pieces via a spatial decomposition, and each small piece is processed independently.

The rest of the article 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 presents some detailed performance analyses of the spatial join algorithms on very large real-world datasets using a cluster containing hundreds of CPU cores. Concluding remarks are contained in Section 5.

This article expands on our recent conference paper [41]. As data reduction is an important standard practice with big data volumes that motivate distributed-computing platforms, we added subsections about using spatial join to aggregate point data by polygon. We discuss our algorithms and implementation of aggregation, which can be done either after the spatial join or during the spatial join. Furthermore, we present experimental results with all implemented variants of spatial

join used for aggregation. In addition to discussing aggregation in detail, finally we add details of the observed runtime of selected experiments, broken down by job and stage.

2 RELATED WORK

Previous work has focused on distributed spatial indexing on Apache Hadoop [40]. For spatial data residing in the Hadoop Distributed File System (HDFS), a linear [16] quadtree [14] index was built in a series of distributed computations in Hadoop MapReduce and stored in distributed fashion on HDFS. The distributed spatial index implemented range and distance queries and *k*-NN but not spatial join. Also, it was implemented on Hadoop MapReduce rather than Spark.

Spatial Join with MapReduce (SJMR) introduced the first distributed spatial join on Hadoop using the MapReduce programming model [47]. 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 [10] optimized SJMR with a persistent spatial index (it supports grid files [29], R-trees [19], and R+-trees [34]) 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 [33]. 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 [4] features both 2D and 3D spatial join. The originally published work was implemented on Hadoop MapReduce and Hive. More recently, a successor is implemented on Spark, called SparkGIS [5, 6]. SparkGIS mitigates skew by making available various partitioning schemes as previously evaluated on MapReduce [3]. SparkGIS uniquely improves memory management in spatial-processing Spark jobs by spatially aware management of partitions loaded into memory rather than arbitrary spilling to disk. SparkGIS was benchmarked with medical pathology images and with OpenStreetMap (OSM) data [30].

GeoMesa [15] provides spatial indexing (Geohash-based [28]) and query capabilities on Apache Accumulo [2] and HBase [18], building upon Accumulo indexing. GeoMesa documentation describes a broadcast join on Spark [17] but not a partitioned join.

GeoSpark [44] is a framework for performing spatial joins, range, and *k*-NN queries. The framework supports quadtree 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 [35] 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 [11]. It has recently had indexing added to it to boost performance. Magellan also provides some coordinate transformation capabilities, most notably transforming between WGS84 and the NAD83 State Plane coordinate systems.

SpatialSpark [43] 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 a more traditional implementation based on Apache Impala [24], a distributed SQL processing engine.

STARK [20] 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 [13]. 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.

Instead of attempting balanced partitioning upfront, Spatial Join with Spark (SJS [46]) uses a regular grid, then measures partition load, and, when needed, re-partitions before executing the local join stage.

Multi-way spatial join algorithm with Spark [9] (MSJS) addresses the problem of performing multi-way spatial joins using the common technique of cascading sequences of pairwise spatial joins [26, 38]. 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 [42] 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, similarly 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 for SQL mode Simba requires 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 support spatio-temporal queries.

LocationSpark [36] 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, called Broadcast Join and Bin Join. The type of join used for any given operation is dependent on the effective size of both datasets.

We also discuss two types of operations or results. The first we call group join or one-to-many join, which returns one feature for every matching pair of target feature and join-side feature, like traditional spatial join. The second, called summary join or one-to-one join, returns only one result feature per target feature and is especially useful with datasets of big data volume. For group join, our distributed spatial join succeeds on all inputs except the case where both inputs are too big to fit in memory *and* one has highly coincident points. For summary join, we have achieved a spatial join implementation that succeeds for all skew and coincident-point characteristics of input data.

3.1 Broadcast Join

Broadcast join (Algorithm 1) is analogous to a map-side join in MapReduce programming [39]. A spatial join in which one of the input datasets is broadcast is implemented in a handful of the referenced works [17, 35, 43]. 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).

ALGORITHM 1: Broadcast Join

```
dataToBroadcast ← determineSmallerDataset().collect()
distributedData ← the other input dataset
indexingBroadcastDataJoiner ← MakeJoinerWithLazyBuildingQuadtreeIndex(dataToBroadcast)
broadcastedJoiner ← spark.broadcast(indexingBroadcastDataJoiner)

for partitionOfFeatures in distributedData do
    indexedJoiner ← broadcastedJoiner.value
    for partitionFeature in partitionOfFeatures do
        for candidateFeature in indexedJoiner.queryIntersects(partitionFeature) do
            if relationshipTest(partitionFeature, candidateFeature) then
                emit pair (partitionFeature, candidateFeature)
            end
        end
    end
end
```

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.

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, per being a Spark RDD.

Each partition in the distributed dataset is joined with the broadcast dataset. This join occurs local to each executor. In the case that the join has a spatial relationship, an in-memory quadtree index [11] 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, then the indexing step is not necessary. In Algorithm 1, the `relationshipTest` predicate function encapsulates both the spatial condition (if any) and the temporal condition (if any). For example, the relationship can be that the features intersect in both space and time. Or, another relationship could be, the features are both near in space, within 100 meters, and near in time, within 15 minutes.

Broadcast Join does not require shuffling either dataset to a spatial partitioning. While not incurring the cost of a shuffle is good, even better is avoiding the overloaded partitions that often occur when spatially partitioning a highly skewed dataset. Broadcast Join works with the pre-existing—hopefully even—partitioning of the input dataset. Thus it is expected to scale to very large sizes of *one* of the input datasets. Because real-world large-volume point datasets are typically highly skewed, it is highly advantageous to use Broadcast Join when the other dataset is small enough to broadcast successfully.

3.2 Bin Join

Bin Join (as contrasted with Broadcast Join) is analogous to a reduce-side join in MapReduce programming [39]. This method is used when both datasets are too large to fit into memory and a partitioned approach is required. Bin Join is variously referred to as partitioned spatial join or grid spatial join and is implemented in several of the referenced works [4, 5, 10, 20, 33, 36, 42–44, 46,

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...>	<v1a,v1b...>
K2	<u2a,u2b...>	<v2a,v2b...>

Fig. 1. Spark cogroup, as contrasted to Spark join method.

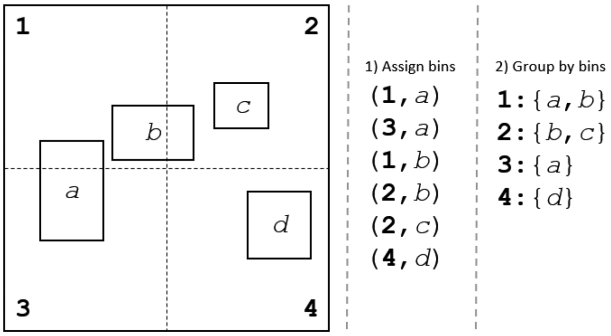


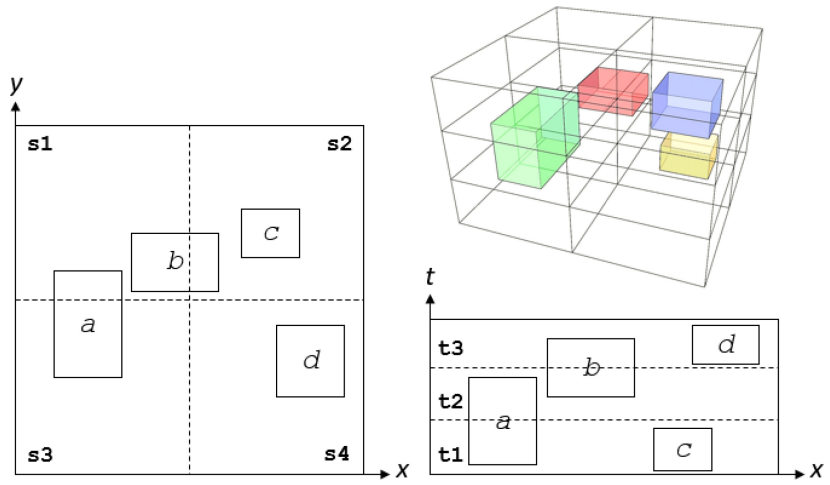
Fig. 2. Space-only binning of some rectangles.

47]. In GeoAnalytics, Bin Join implements distance join as well as the more strict sense of spatial join in which spatial objects have a topological relationship such as *intersects*. Pseudo-code for Bin Join appears in Algorithm 2.

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 spatial near radius specified by `padSpace`, and/or the temporal near span given by `padTime`), including bins that do not necessarily intersect the feature geometry, in space and/or in time.

The shape of a bin is defined separately and independently for XY and T. The XY shape (see Figure 2) is determined by one of two grid implementations as described in Section 3.3. T is defined as a temporal interval (e.g., 1s, 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.

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. Specifically, the binning stage produces RDDs of pairs of bin-ID and feature, which allows use of



a: { (s1,t1), (s3,t1), (s1,t2), (s3,t2) }
b: { (s1,t2), (s2,t2), (s1,t3), (s2,t3) }
c: { (s2,t1) }
d: { (s4,t3) }

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

the additional Spark functions available on type PairRDD, including the `cogroup` function. It is important to note that the bin ID is the key of the key-value pair, because we group on the bin ID.

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, then an on-the-fly in-memory quadtree index can be built on either side of the join to further filter the spatial relationships. Also as in Broadcast Join, likewise in Bin Join (Algorithm 2), the `relationship` predicate function encapsulates both the spatial condition (if any) and the temporal condition (if any).

De-duplication may be required to remove duplicate matches that occur when features end up in multiple bins. Potential duplicates can occur with any of lines, polygons, and inflated points (for near join). Specifically, duplicates can occur when neither of the inputs for binning is non-inflated point data.

We apply reference-point de-duplication [8, 10]—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.

ALGORITHM 2: Bin Join

```
binningMesh ← construct regular-grid or quadtree mesh
binnerLeft ← spark.broadcast(makeBinner(binningMesh, timeCycle, padSpace, padTime))
binnerRight ← spark.broadcast(makeBinner(binningMesh, timeCycle))
leftWithBins ← targetDataset.flatMap (feat ⇛ binnerLeft.value.pairWithBins(feat))
rightWithBins ← joinDataset.flatMap (feat ⇛ binnerRight.value.pairWithBins(feat))
groupedByBin ← leftWithBins.cogroup(rightWithBins)
for (binId, (targFeats, joinFeats)) in groupedByBin do
    targIndex ← buildQuadtreeIndex(targFeats)
    for joinFeature in joinFeats do
        for targFeature in targIndex.query(joinFeature) do
            if relationship(targFeature, joinFeature) and isUnique(binId, targFeature, joinFeature) then
                emit pair (targFeature, joinFeature)
            end
        end
    end
end
```

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 gap, 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).

3.3 Regular and Adaptive Binning

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 spatially 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. In our observation, temporal skew has never been an issue, in contrast to spatial skew.

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 that may adversely affect memory usage and system performance.

It may not always be cost effective to try to determine the characteristics of a dataset (for the purposes of choosing the binning method). If the join appears near the end of a long Spark pipeline, then the cost of sampling the dataset may be equal to the cost of evaluating the entire pipeline of the RDD passed as input to the join operation. 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.

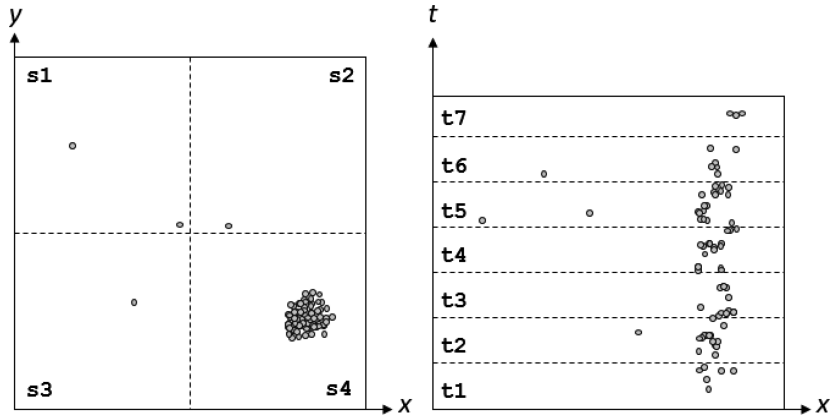


Fig. 4. Time slicing in spatiotemporal-join binning alleviates the bin overload that would occur in space-only join with skewed dense data.

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.

In the common use case of aggregate points to polygons, the cell size for regular grid space-only bin join is constrained as follows. The polygon dataset places a lower bound on the cell size, below which the replicated polygons will balloon so big as to cause out-of-memory. The point dataset may be dense and skewed so as to place an upper bound on the cell size, above which an overloaded bin will run out-of-memory. Only within the range between target-data-imposed lower bound and join-data-imposed upper bound can regular grid join succeed. If the target-imposed lower bound is in fact greater than the join-imposed upper bound, then no optimizer, however sophisticated, can choose a cell size at which regular grid join can succeed on the given target and join datasets.

A more sophisticated approach is a quadtree-based grid, as in the partitioning of VegaGiStore [48]. Using a PMR quadtree decomposition of space [27], 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 afterward. 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 [40]. The process of building the PMR quadtree in parallel across a collection of Spark partitions consists of the following five steps:

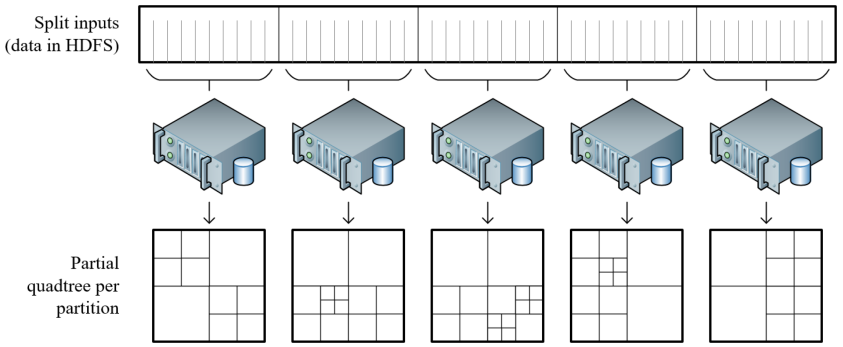


Fig. 5. Example highlighting the splitting of the source data into equal sized collections (step 1) and the partial quadtrees being built on each processing element.

(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.

For building a quadtree in a distributed manner, the input needs to be partitioned for the quadtree build—not to be confused with using the quadtree build result later for partitioning the spatial-join job following. 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 [16] 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 in that regard). 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.

For combining the quadrants of the partial quadtrees into subtrees of a complete quadtree index (step 3), the quadtree entries are re-partitioned. (Again, this [pre-]partitioning is separate from later using the resulting quadtree as the partitioner for spatial join.) A custom partitioner assigns the entry from the partial linear quadtrees 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

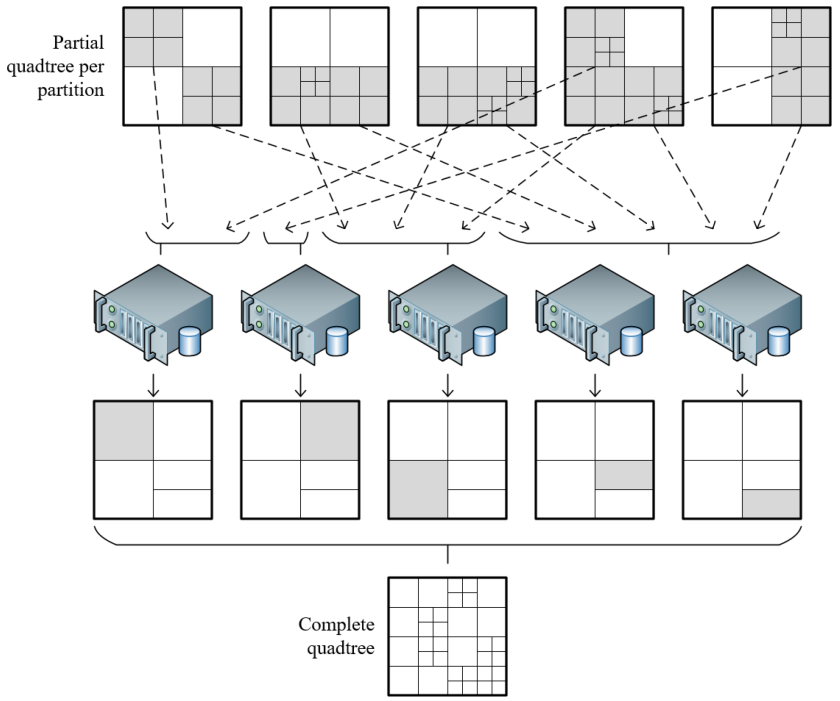


Fig. 6. Example showing entries from the partial quadtrees being shuffled and merged on the collection of processing elements. The combined result (the complete quadtree) is also shown.

extent into the desired number of partitions by alternating horizontal and vertical splits, like a PR k-d tree decomposition [31], with splitting-direction alternation order to be compatible with the N-order of the quadtree being built.

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 that 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 [21].

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, and so on. 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).

3.4 Post Aggregation/Summarization

In analysis of data of big data volume, an important aspect is data reduction—in other words, producing a result small enough to be comprehensible to a human. With a large volume of point data, it is typically useful to aggregate the points to a collection of regions. One way is to aggregate the points to a regular mesh of squares or hexagons, as in the Aggregate Points to Bins capability of ArcGIS GeoAnalytics. The other way is to aggregate the points to a set of polygons that partition space and that is accomplished by Spatial Join—as in Aggregate Points to Polygons.

With very large data volumes that require data reduction to be useful, aggregation of point data to polygons, is more common than one-to-many join. The straightforward way to produce an aggregated or summarized result is to first perform a grouping one-to-many join, and then afterward, for each target-side feature (typically polygon), aggregate the join-side features that were grouped with it. The grouping spatial join can be done with either Broadcast Join or Bin Join, and is the same as done for one-to-many join, except for the addition of one detail. Before binning and joining, each feature from the target side (typically polygons) is assigned a unique ID, as with the Spark function `zipWithUniqueId`. The target polygon may be assigned to multiple bins, and the unique ID is needed as the key for aggregating join-side features paired with that target-side polygon across all such bins.

Algorithm 3 shows post-aggregation simplified by aggregating only counts rather than various statistics on the several attributes. The aggregation is done with the Spark higher-order function `PairRDD.combineByKey`, which takes three callback functions as arguments. The `makeCombo` function simply assigns a count of one in the new attribute field, which is appended to the existing attributes of the target-side feature, to make a combiner feature out of one single join-side feature. The `mergeValue` function merges one join-side feature into a combiner feature by incrementing its count by one. Last, the `mergeCombos` function merges two combiner features by adding their counts.

Besides record count, ArcGIS GeoAnalytics supports several per-field statistical summaries: arithmetic mean, variance, standard deviation, minimum, and maximum. The accumulator for each statistic accumulates the information required for the resulting statistic, as the value for each feature is passed in. For example, the arithmetic mean statistic accumulates the sum and count of the values for its field. Afterward, the result can be queried; in the case of arithmetic mean, the result is the quotient of the accumulated sum and the accumulated count.

ALGORITHM 3: Post Aggregation

```
function makeCombo(targetFeature, joinFeature):
    return makeFeature(targetFeature.attributes.append(1), targetFeature.geometry)

function mergeValue(aggFeature, joinFeature):
    return aggFeature.withLastAttr(1 + aggrFeature.getLastAttr)

function mergeCombos(agg1, agg2):
    return agg1.withLastAttr(agg1.getLastAttr + agg2.getLastAttr)

targetData.assignIds
grouped ← FeatureJoin.group(targetData, joinData)
byId ← grouped as key-value pairs where the key is the ID assigned to the target feature
summarized ← byId.combineByKey(makeCombo, mergeValue, mergeCombos)
```

3.5 Direct Summarization with Pre-Aggregation

We have earlier noted that in especially dense regions, the bin assigned such dense points can easily become overloaded and cause Bin Join to suffer from straggler tasks and even fail due to running out of memory. We further noted that an adaptive quadtree mesh can help with dense and skewed datasets by using small bins on dense areas but without using such small bins everywhere such that the profusion of intermediate records causes slowdowns and failures. Such small bins help with non-coincident dense data but cannot help with highly coincident points. Bin overload is an issue no matter how small the cell size in the case of highly coincident points, such as found in the well-known publicly available dataset GDELT [32].

Direct-summarization means accumulating the summary statistics during the `cogroup` callback rather than materializing the one-to-many results first (and summarizing afterward). This limits the cardinality of the output of a bin to the number of target-side features assigned to the bin rather than the product of the number of target-side features and join-side features. That by itself does not suffice to resolve the case of highly coincident points. However, direct summarization allows us to additionally employ another technique that we call pre-aggregation.

As seen in Algorithm 4, we pre-aggregate coincident points into a micro-cell and then collapse the micro-cell to a point to continue operating on it as a point dataset. This way, even in the case of millions of point features coincident in the same location, they will be collapsed to one single point with pre-aggregated field values. Thus, from all the points in this micro-cell, the bin assigned during Bin Join will receive one point with pre-aggregated field values rather than getting overloaded by millions of separate point features.

For the size of the micro-cell for pre-aggregation, we choose the tolerance value of the spatial reference, which runs about 1mm. We recognize that collapsing an input point to the center of its assigned micro-cell could effectively move the point coordinates by a distance of up to $\sqrt{2}/2$ times the side length of the micro-cell. From a theoretical standpoint, one might say that this introduces an approximate result. However, the reality of real-world data is that coordinates are measured rather than declared—and the error bars inherent in measurement exceed the tolerance of the spatial reference. In fact, in real-world data we have observed that coincident points typically result from snapping the observed locations to some lower-resolution set of possible locations; in the extreme case of GDELT, one of the motivating examples for developing pre-aggregation, locations are snapped to a representative point for a city, municipality, or other such well-known toponym. Thus, more accurate would be to say that join results are always approximate because the coordinates are approximate and that pre-aggregation can cause a slight change in results for those pairs of spatial objects that are borderline as to whether they intersect. Trying to determine the

ALGORITHM 4: Space-Only Bin Join – Direct Summary with Pre-Aggregation

```
targetData.assignIds
statComb ← initialize statistics combiner for requested statistics

binnerPre ← makeBinner(ultra-fine mesh with tolerance as cell size)
preBinned ← joinDataset.flatMap(feat ⇛ binnerPre.withBins(feat))
interimData ← preBinned.combineByKey(statComb.makeCombo, statComb.mergeValue,
    statComb.mergeCombos)
preAggr ← interimData.map((bin, summaries) ⇛ Feature(summaries, binnerPre.centerPoint(bin)))

binningMesh ← construct regular-grid or quadtree mesh
binnerLeft ← makeBinner(binningMesh, padSpace)
binnerRight ← makeBinner(binningMesh)
leftWithBins ← targetDataset.flatMap(feat ⇛ binnerLeft.withBins(feat))
rightWithBins ← preAggr.flatMap(feat ⇛ binnerRight.withBins(feat))          // n.b. preAggr
groupedByBin ← leftWithBins.cogroup(rightWithBins)

joined ← for partitionOfPairings in groupedByBin do
    for (binId, (targFeats, joinFeats)) in partitionOfPairings do
        targIndex ← buildQuadtreeIndex(targFeats)
        for targFeature in targFeats do
            initialize entry to empty in table of statistics accumulators
        end
        for joinFeature in joinFeats do
            for targFeature in targIndex.query(joinFeature) do
                if relationshipTest(targFeature, joinFeature) andisUnique(binId, targFeature, joinFeature)
                then
                    update stats accumulator for targFeature,with the attributes of joinFeature
                end
            end
        end
    end
    emit table of target features with their corresponding array of accumulated statistics
end

merged ← joined.combineByKey(identity, statComb.mergeCombos, statComb.mergeCombos)
summarized ← merged.map((feat, joinStats) ⇛ feat.append(joinStats.getResultOfEach))
```

incidence of deviation from exact spatial join aggregation results would be a complex probabilistic calculation that would need information on the specific devices used for measuring both datasets, with their precision and accuracy, to develop a probability density function of the actual location of a point given its measured coordinates. Furthermore, in a GIS, a point, which is theoretically zero-dimensional, represents a real-world object whose base is orders of magnitude bigger than the spatial-reference tolerance. We conclude that for real-world analysis, an effective change of location by less than the tolerance is deemed immaterial to the result of spatial join aggregation.

In the callback to Spark `cogroup`, as in grouping Bin Join, we check the feature relationship and then we check uniqueness by reference-point de-duplication. But instead of yielding pairs of features, we accumulate the requested statistics on fields of the attributes. For example, one of the statistics may be the mean of a numerical field called "magnitude." At the start of the callback, we initialize a table of statistics accumulators, in which we have for each of the bin's target features,

a collection of accumulators for statistics on the fields of join-side features. After updating such accumulators with the field values of the matching join-side features, we yield the table, which consists of key-value pairs where the key is the target-side feature and the value is the collection of statistics accumulated from join-side features.

As with post-aggregation, likewise with pre-aggregation, the target polygon feature may have been assigned to several bins; therefore post-merging is required. So, in pre-aggregation, as with post-aggregation, each target feature has been assigned a unique ID before binning and joining. The partial result after the per-bin direct summarization, is a `PairRDD`, in which the key is the ID that had been assigned to the polygon, and the value is a polygon feature whose geometry (and time if any) is the geometry (and time or null) of the original polygon of that ID but whose attributes are statistics accumulators rather than final numerical values. For each target feature, we combine or merge the summary statistics using the summary-statistic accumulators for each of the result fields. This is done with Spark `PairRDD.combineByKey`, which will combine all the partial-result features that have the same previously assigned ID. Due to direct summarization, the first callback argument to Spark `combineByKey` is the identity function, and the remaining two callbacks are both the method for merging existing combinations (which for post-aggregation, was the last argument to `combineByKey`), provided by a statistics combiner object that was constructed for the requested statistics.

3.6 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 many opportunities to tweak the configuration of the algorithm or processing environment (note that 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, it can 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, then we failover to a more expensive, but robust bin join.

4 PERFORMANCE COMPARISON

To test scalability and runtime, we benchmarked with a couple of big real-world datasets. First, we used the New York City Taxi and Limousine Commission's taxi dataset [37]. It consists of about 1.2 billion records from 2009 to 2015, detailing both trip and fare data for every taxi trip recorded in the five boroughs of New York City. The data are available for download as a collection of CSV files. The trip data contain 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 contain 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 on-the-fly on input; but in reality

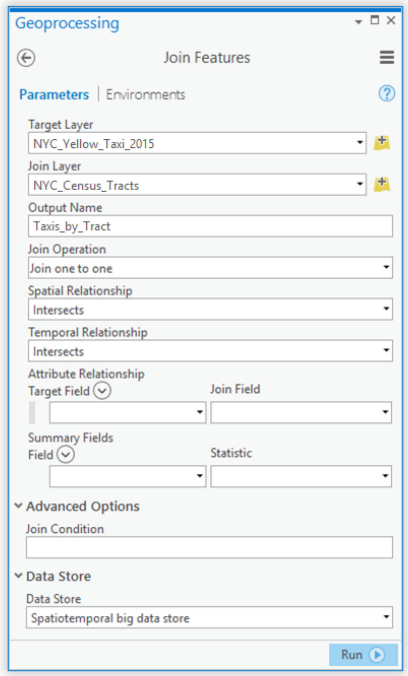


Fig. 7. Example of an easy-to-use spatio-temporal join tool (taken from ArcGIS Pro).

that did little to avoid the impact of outliers, because the taxi data also contain other unrealistic outliers.

We also used GDELT, originally known as the Global Database of Events, Language, and Tone [25]. It is a human-curated database and has many point features coincident at the same location in urban areas. We used a 2013 snapshot of GDELT containing approximately 210 million features.

All tests were performed on a Spark 2.2.0 Standalone cluster with 18 worker nodes (72 CPU cores with 144 hyperthreads) and 16GB of memory per executor. Results were calculated and counted and then discarded without persisting—in real-world use of the product, writing the output can also take substantial time.

4.1 Polygons Grouping Points

We compared our algorithms and variants in joining various sizes of NYC taxi 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. In any case, this test confirmed the expectation that Broadcast Join runs fastest when it can complete, and that Bin Join scales to larger inputs with a quadtree mesh than with a regular grid, when the large set of points has a skewed distribution.

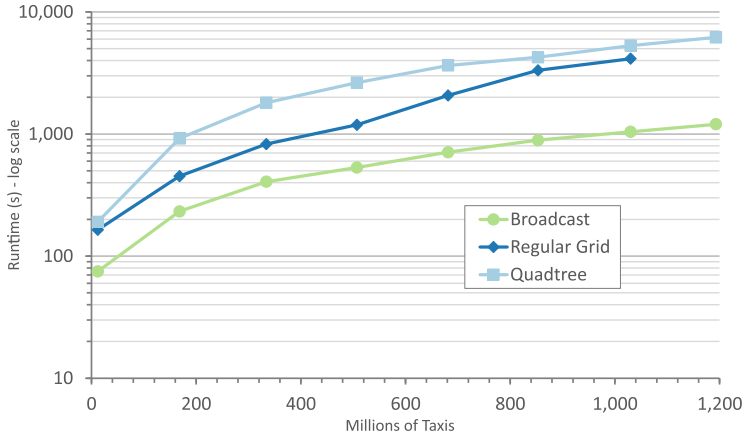


Fig. 8. Joining New York City taxi points (1.2 billion) to census tracts (2166).

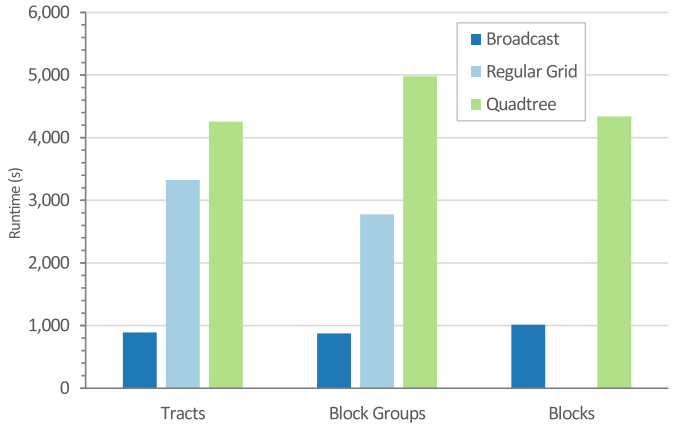


Fig. 9. Joining 700 million New York City taxis (five years) with Census geographies (2,200 census tracts, 6,500 block groups, and 38,800 blocks).

We benchmarked grouping join on a varying number of polygons, with a fixed number of taxi points (about 700 million taxis in five years of data). Specifically, the polygons are the successively more local polygon meshes of the U.S. Census Bureau: approximately 2,200 census tracts, 6,500 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 more-numerous 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. In any case, we observed the same expected results when varying the polygon dataset as when varying the point dataset—Broadcast Join runs fastest when it can complete, and Bin Join scales to larger inputs with a quadtree mesh than with a regular grid.

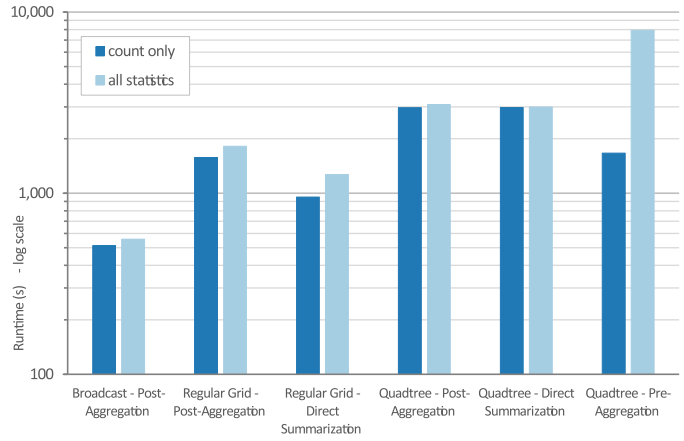


Fig. 10. Aggregating New York City taxis (507 million) to census tracts (2166)—count only versus all statistics on all attribute fields.

4.2 Polygons Aggregating Points

The previous subsection refers to traditional or group join yielding all pairs of matching features. Next, we benchmarked comparisons of algorithm variants for aggregating various sizes of NYC taxi data by the census tracts. Specifically, we calculate two types of aggregation results: first, the count only, and, second, all applicable summary statistics on all the fields of the taxi feature attributes). Furthermore, we compare and contrast post-aggregation (after grouping), direct summarization only (without pre-aggregation), and direct-summarization with pre-aggregation.

Figure 10 compares the case of the user merely requesting the count of taxi points per polygon versus requesting all statistics relevant to all the attribute fields of the taxi data. Note that regular-grid bin join with pre-aggregation is omitted, because it was unable to complete for summarizing all statistics, whereas all of the other variants completed for both count-only and all-statistics. For post-aggregation and for direct summarization without pre-aggregation, the additional time required to summarize all statistics was moderate. However, pre-aggregating direct summarization (which succeeded only with quadtree mesh), the difference was striking. For count only (with quadtree), pre-aggregation ran in about *half* the time of the alternatives. In contrast, when calculating all statistics (with quadtree), pre-aggregation took over *twice* the time as the others. We conjecture that such slowdown is caused by a combination of the extra step to pre-aggregate and the extra memory and shuffle required to carry around all the partially computed statistics on the attributes. In summary, the graph indicates that trying to choose only one variant for all summarizing joins does not suffice to achieve optimal performance.

We then proceed to consider what would be the most scalable and efficient join algorithm variant for each of such cases—the case where the user requests counts only (see Figure 11) and the case where the user requests all statistics on all fields (see Figure 12).

Broadcast join has not been implemented with direct summarization, so Broadcast Join was run only with post-aggregation. Broadcast join, unsurprisingly, ran faster than any variant of Bin Join—for both count only and all statistics. Comparison of aggregating variants of Bin Join is nevertheless interesting for the case that larger polygon datasets would require Bin Join or that a general-purpose join planner/optimizer is unable to feasibly determine the size of the polygon dataset.

For aggregation with count only, bin join completed for all input sizes with regular grid, when using direct summarization, either with or without pre-aggregation. Furthermore, bin join completed faster with regular grid than with quadtree mesh. However, noting that with

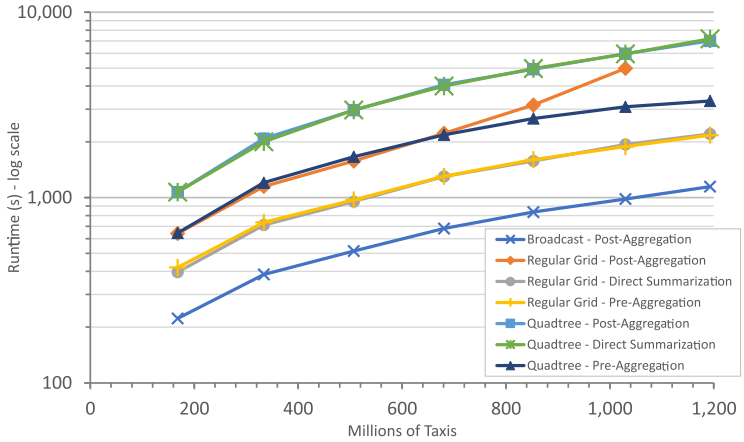


Fig. 11. Aggregating New York City taxis to census tracts—*count only*. Comparison between post-aggregation and direct summarization with and without pre-aggregation and also between broadcast join and bin join with regular grid and quadtree binning. (Almost the same: Post-Aggregation and Direct Summarization with Quadtree; Direct Summarization and Pre-Aggregation with Regular Grid.)

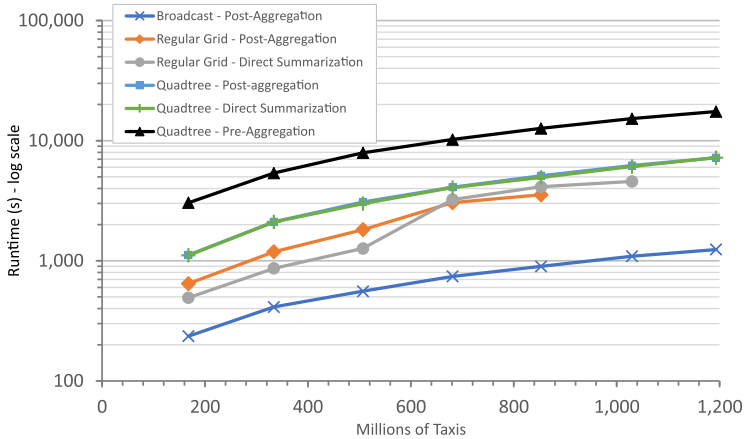


Fig. 12. Aggregating New York City taxis to census tracts—*all statistics*. Comparison between post-aggregation and direct summarization with and without pre-aggregation and also between broadcast join and bin join with regular grid and quadtree binning. (Almost the same: Post-Aggregation and Direct Summarization with Quadtree.)

post-aggregation, bin join scaled to larger input size with quadtree mesh than with regular grid, it is our conjecture that with yet-larger input of similarly extreme skew, quadtree mesh could likely outscale regular grid, likewise with direct summarization. We additionally note that when using the quadtree mesh, the join completed faster with pre-aggregation than with either alternative for aggregating only the count.

Again, Figure 11 indicates that for aggregating counts only, broadcast join is fastest when it can complete and that, with bin join, it ran faster with regular grid than with quadtree for the range of inputs tested (for direct summarization with or without pre-aggregation—rather than with post-aggregation).

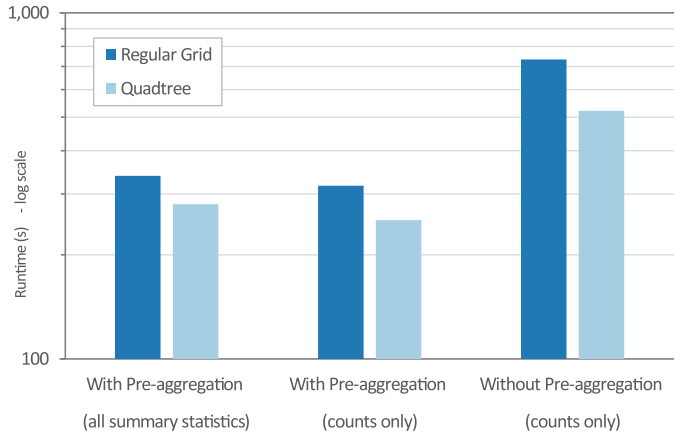


Fig. 13. ZCTA (33,000) aggregating GDELT (210 million), using direct summarization with and without pre-aggregation.

For aggregating all statistics on all fields, bin join with quadtree mesh clearly outscaled bin join with regular grid. With a regular grid, post-aggregation succeeded up to about 900 million points (fewer than with the non-aggregating grouping join), and direct summarization without pre-aggregation made it possible for regular-grid bin join to aggregate taxi points up to about as many as with group-only join (about one billion). Pre-aggregating bin join did not succeed with regular grid, not with any of the input sizes attempted. Bin join with quadtree mesh succeeded in all tests of aggregating taxi points by census block on all sizes up to the full 1.2 billion. With the quadtree, direct summarization without pre-aggregation had the same runtime as grouping followed by post-aggregation. However, pre-aggregating direct summarization took over twice as long. Recall that pre-aggregation was implemented with the objective of robustness to highly coincident points; this benchmark indicates there can be a performance penalty for using it even when not needed.

Summary conclusions about summarizing all statistics on all fields, from Figure 12, are as follows: Broadcast join is fastest when it can complete; and with bin join, on large skewed inputs, the quadtree mesh is needed, but pre-aggregation can cause a slowdown.

For testing the case of highly coincident points, we used GDELT, which is characterized by a large number of point features coincident at many locations. We aggregated GDELT points by United States ZIP Code Tabulation Areas (ZCTA), consisting of about 33,000 polygons (with an average vertex count roughly 1500; see Figure 13). It was not possible to complete a grouping join of ZCTA and GDELT—and thus not post-aggregation, either; we are unable to broadcast ZCTA, and bin join failed with both regular grid and quadtree mesh, because it runs out of memory when processing the bin containing the many coincident points, no matter how small the bin size. For aggregating count only, direct summarization succeeded both with and without pre-aggregation. However, for aggregating all summary statistics on all fields of GDELT, direct summarization was insufficient by itself but succeeded with pre-aggregation. Also of note, in all cases where this aggregation join completed successfully, it ran faster with quadtree mesh than with regular grid. Similar results were seen when aggregating GDELT by the United States Census Bureau Tabulation Blocks, a dataset with a much higher polygon count, about 11 million polygons (with an average vertex count roughly 60; see Figure 14). For summarizing highly coincident points by any set of polygons, we conclude that pre-aggregation is necessary for summarizing all the statistics, in addition to being faster for count only.

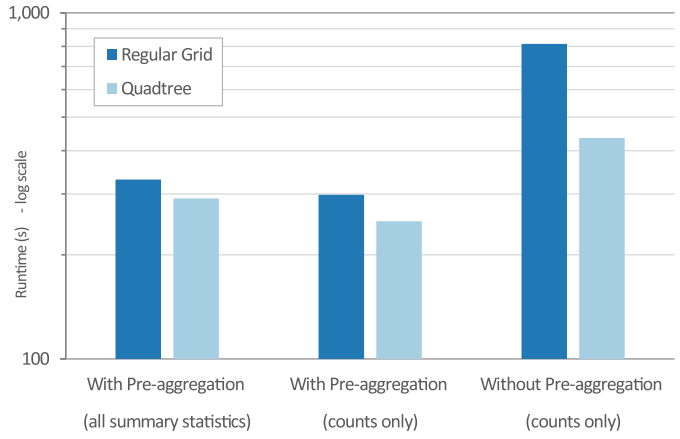


Fig. 14. Census blocks (11 million) aggregating GDELT (210 million) using direct summarization with and without pre-aggregation.

To recap, our experiments indicate that the recommended aggregation approach, based on input data characteristics, is as follows. If one dataset can be determined to fit in memory, then Broadcast Join followed by post-aggregation is fastest, as well as most robust to any characteristics of the other dataset. Otherwise, proceed to choose among the Bin Join variants for joining datasets both of which can be big. If the point dataset is characterized by highly coincident points, then quadtree bin join with pre-aggregation is the variant that completes reliably and fastest. Finally, the case where the point data is not highly coincident, even if possibly dense and skewed: For counts only, regular-grid bin join with direct summarization ran fastest while succeeding in all tests, with a tossup between using or not using pre-aggregation with the direct summarization; whereas for calculating all statistics, the quadtree mesh was needed for bin join to succeed on the larger end of the input range, and the fastest aggregation variant was a tossup between *post*-aggregation and direct summarization *without* pre-aggregation.

4.3 Self and Distance/Near Join of Points

We prepared benchmarks to compare GeoAnalytics spatial join to STARK, which had published results of outperforming GeoSpark and SpatialSpark. Figure 15 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 73s. Although one configuration of STARK ran fastest at modest data size, STARK was observed to have scalability issues when the input size increased, and GeoAnalytics outscaled STARK substantially.

We benchmarked space-only near join against Simba, which refers to it as distance join (Figure 16). Input was subsets of the NYC taxi points, and the near radius used was 20m. Up to a size of about five million taxis, Simba outperformed GeoAnalytics on this space-only near self-join; but Simba was unable to complete the join with more than about seven million taxis (recognizing that

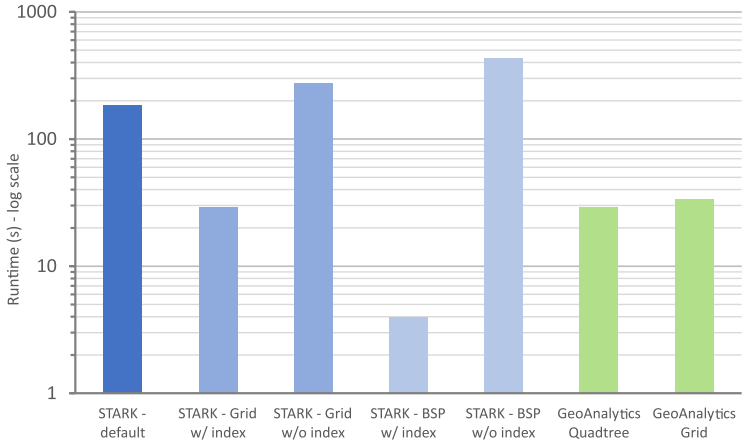


Fig. 15. STARK and GeoAnalytics—New York City taxi self-join (one million taxis subset).

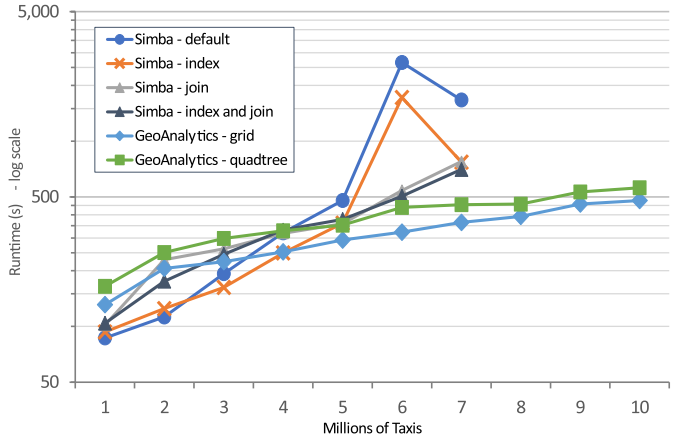


Fig. 16. Simba and GeoAnalytics—space-only near join (note that Simba is run in four configurations, with and without providing tuning parameters for number of index and/or join partitions).

Simba published results of 10-million-point by 10-million-point 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 100 million taxis (Figure 17). We conclude that GeoAnalytics scales to larger dense and skewed data than does Simba.

Figure 18 shows a benchmark of the scalability of our Spatio-temporal Join, in performing a self-join on subsets of the NYC taxi points, ranging from the data of 1 month to all 7 years. The input parameters were a spatial near radius of 20m and a temporal near span of 10 minutes. Our join is able to scale past a billion, in the size of input to self-join, 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 (Figure 18) scales past space-only join (Figure 17), 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 self-join on an input dataset exceeding one billion points.

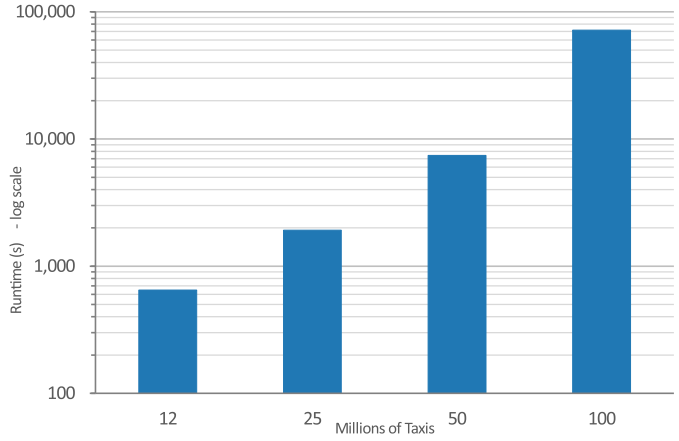


Fig. 17. GeoAnalytics spatial self-join with quadtree.

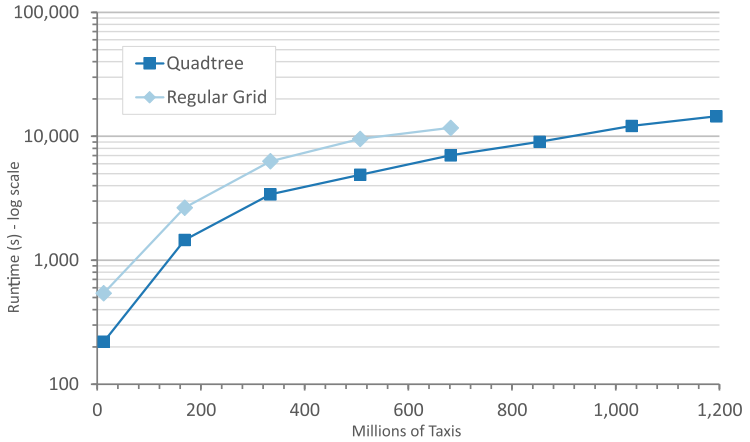


Fig. 18. New York City taxi points spatio-temporal near self-join.

4.4 Spark Jobs and Stages

Besides comparing different variants of our spatial join algorithms, it is interesting to examine the time taken by different steps of the join algorithm. In Spark, a computation comprises one or more Spark jobs, each of which comprises one or more Spark stages. Figure 19 displays the runtime of each of the three Spark jobs in one-to-many spatio-temporal self-join of NYC taxi points in bin join with quadtree mesh. The final job, which is the spatial join itself, takes about twice as long as the prelimary job that builds the quadtree mesh, which in turn takes several times longer than the preparatory job that calculates the data extent (needed for quadtree build) while reading the data off disk. Figure 20 splits up the runtime of the same spatio-temporal near joins into even greater granularity by Spark stage. There are two stages each for the ingest and quadtree-build jobs, and then, for the spatial join job, a stage for binning each of the inputs followed by a stage for the per-bin local spatial join calculations. The per-stage information confirms that the final stage of bin-local joins does take longer than any one of the preceding stages. However, the final stage does take up less than half of the total runtime, which is to say less than the sum of all the other stages that lead up to it. The second-most-costly stage is seen to be merging partial quadtrees in

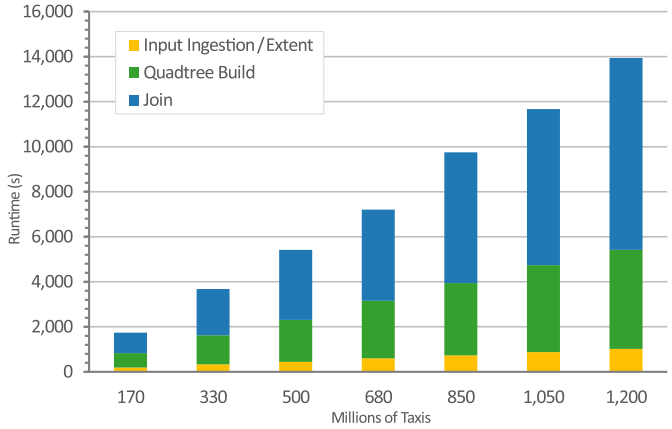


Fig. 19. New York City taxis spatio-temporal near self-join by Spark job.

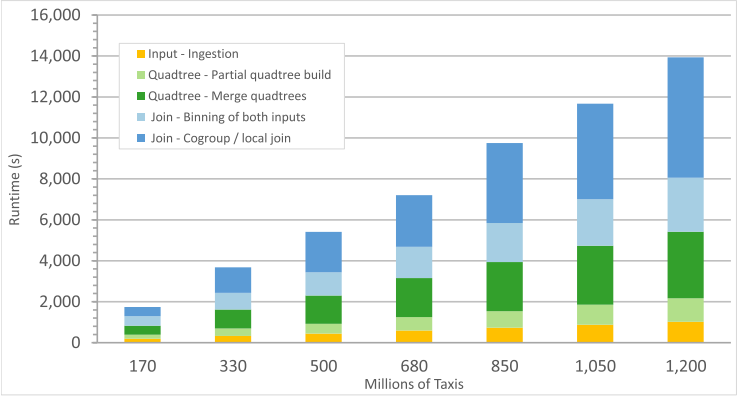


Fig. 20. New York City taxis spatio-temporal near self-join by Spark stage.

the distributed build of the quadtree mesh. The two binning stages (on target features and join features) are interlaced by Spark, and the time reported for the longer one by the Spark Web UI, is the elapsed time for completing both stages; thus the binning stages are combined in the graph using the time of the longer-running one. The ratio of runtime by stage is remarkably consistent across the range of sizes of taxi-point inputs tested. Other metrics such as Spark shuffle costs are not provided here, as the elapsed time is what is important to the end user.

5 CONCLUSION AND FUTURE WORK

In this article, 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. For group (one-to-many) join, our distributed spatial join succeeds on all inputs except the very specific case where both inputs are too big to fit in memory *and* one has highly coincident points. For summary (one-to-one) join, we have achieved a spatial join implementation that succeeds for all skew and coincident-point characteristics of input data.

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 evaluated three approaches to aggregation for one-to-one spatial join. The most straightforward is post-aggregation of the one-to-many spatial join result. The alternate is direct summarization inline with testing the geometry relation in local per-bin join, which technique we evaluated both with and without the optional pre-aggregation of coincident points. For inputs such that one dataset can be broadcast, Broadcast Join with post-aggregation is fastest. Direct summarization only (without pre-aggregation), in bin join, had equivalent results to post-aggregation in some cases and in other cases showed improvements to scalability or robustness to coincident points. The effectiveness of pre-aggregation with direct summarization, varied greatly by other factors. When aggregating all statistics on all fields, pre-aggregation with regular grid was the worst option of all, failing all tests. On a dense and skewed but not highly coincident dataset, using quadtree mesh, pre-aggregation was much faster for merely counting the join matches but much slower for calculating all statistics on all fields of the point features. On a dataset with highly coincident points, being aggregated by a polygon dataset too big to broadcast, pre-aggregation was the only effective approach. We observed that implementing such a general-purpose planner/optimizer for spatial join is far from straightforward.

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.

The spatio-temporal bin join with quadtree mesh in ArcGIS GeoAnalytics is, to our knowledge, the only general spatial join implementation that can complete a spatio-temporal near self-join on an input dataset containing over a billion points.

5.1 Future Work

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. Using a clipped polygon in per-bin local join would require a post-join on ID so as to emit the complete geometry for the result. Once a post-join on ID is in place, it would also be possible to drop the target-side feature attributes during a one-to-many join operation without attribute criteria, thus reducing memory consumption, and then recover the attributes for inclusion in the result, as part of the post-join on ID. Reducing memory usage during the join operation could furthermore allow more aggressive utilization of Broadcast Join.

For near join, we can benchmark the effect of near radius, as well as the already-benchmarked input size. For near self-join, instead of inflating one input side by the full distance, we could inflate both sides by half the distance and use a single binning stage instead of having two binning stages. The caveat is that de-duplication would have to be applied when both inputs are inflated.

One-to-many space-only join on input with highly coincident points can be pursued in a couple ways. 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. Alternately, the input points can be associated with center points of micro-cells, similarly to that of pre-aggregation in

direct summary. The join operation would be performed on these micro-cell centers. Then in a post-pass, all features of the other dataset that match a given micro-cell center will be returned with each of the points pre-associated with the micro-cell.

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).

The local join component of bin join could build a quadtree on both sides of the per-bin input rather than on only one side. Then, instead of streaming through the features of one side and for each one querying the quadtree built on the other side, the local join could walk the quadtrees in Peano-key order.

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.

ACKNOWLEDGMENTS

We thank key development staff within Esri who 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] David J. Abel, Beng Chin Ooi, Kian-Lee Tan, Robert Power, and Jeffrey X. Yu. 1995. Spatial join strategies in distributed spatial DBMS. In *Proceedings of the 4th International Symposium on Advances in Spatial Databases (SSD'95)*. Springer-Verlag, London, UK, 348–367. http://dl.acm.org/citation.cfm?id=647224.718929

[2] Accumulo. 2012. Apache Accumulo. Retrieved May 17, 2018 from https://accumulo.apache.org.

[3] Hoang Vo, Ablimit Aji, and Fusheng Wang. 2014. SATO: A spatial data partitioning framework for scalable query processing. In *Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'14)*. ACM, 545–548.

[4] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop GIS: A high performance spatial data warehousing system over mapreduce. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1009–1020. DOI: https://doi.org/10.14778/2536222.2536227

[5] Furqan Baig, Mudit Mehrotra, Hoang Vo, Fusheng Wang, Joel Saltz, and Tahsin Kurc. 2015. SparkGIS: Efficient comparison and evaluation of algorithm results in tissue image analysis studies. In *Proceedings of the VLDB Workshop on Biomedical Data Management and Graph Online Querying (DMAH'15)*, Vol. 9579. Springer.

[6] Furqan Baig, Hoang Vo, Tahsin Kurc, Joel Saltz, and Fusheng Wang. 2017. SparkGIS: Resource aware efficient in-memory spatial query processing. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'17)*. ACM, New York, NY, Article 28, 10 pages. DOI: https://doi.org/10.1145/3139958.3140019

[7] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1996. Parallel processing of spatial joins using R-trees. In *Proceedings of the 12th International Conference on Data Engineering (ICDE'96)*. IEEE Computer Society, Los Alamitos, CA, 258–265. http://dl.acm.org/citation.cfm?id=645481.655583

[8] J. P. Dittrich and Bernhard Seeger. 2000. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the 16th International Conference on Data Engineering (ICDE'00)*. IEEE Computer Society, Los Alamitos, CA, 535. http://dl.acm.org/citation.cfm?id=846219.847395.

[9] Zhenhong Du, Xianwei Zhao, Xinyue Ye, Jingwei Zhou, Feng Zhang, and Renyi Liu. 2017. An effective high-performance multiway spatial join algorithm with Spark. *ISPRS Int. J. Geo-Inf.* 6, 4 (Mar. 2017), 96. DOI: https://doi.org/10.3390/ijgi6040096

[10] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering (ICDE'15)*, Vol. 2015-May. IEEE Computer Society, 1352–1363. DOI: https://doi.org/10.1109/ICDE.2015.7113382

[11] Esri. 2013. GIS Tools for Hadoop. Retrieved April 11, 2018 from https://github.com/Esri/gis-tools-for-hadoop.

[12] Esri. 2016. ArcGIS GeoAnalytics Server. Retrieved April 11, 2018 from http://server.arcgis.com/en/server/latest/get-started/windows/what-is-arcgis-geoanalytics-server-.htm.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters 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)*. AAAI Press, 226–231. http://dl.acm.org/citation.cfm?id=3001460.3001507.

[14] R. A. Finkel and J. L. Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta Inf.* 4, 1 (01 Mar 1974), 1–9. DOI: https://doi.org/10.1007/BF00288933

[15] A. Fox, C. Eichelberger, J. Hughes, and S. Lyon. 2013. Spatio-temporal indexing in non-relational distributed databases. In *Proceedings of the 2013 IEEE International Conference on Big Data*. 291–299. DOI: https://doi.org/10.1109/BigData.2013.6691586

[16] Irene Gargantini. 1982. An effective way to represent quadtrees. *Commun. ACM* 25, 12 (Dec. 1982), 905–910. DOI: https://doi.org/10.1145/358728.358741

[17] GeoMesa. 2017. GeoMesa Spark: Aggregating and Visualizing Data. Retrieved May 17, 2018 from http://www.geomesa.org/documentation/tutorials/shallow-join.html.

[18] Lars George. 2011. *HBase: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[19] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD'84)*. ACM, New York, NY, 47–57. DOI: https://doi.org/10.1145/602259.602266

[20] Stefan Hagedorn, Philipp Götze, and Kai-Uwe Sattler. 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'17)*.

[21] Gisli R. Hjaltason and Hanan Samet. 2002. Speeding up construction of PMR quadtree-based spatial indexes. *VLDB J.* 11, 2 (Oct. 2002), 109–137. DOI: https://doi.org/10.1007/s00778-002-0067-8

[22] Erik G. Hoel and Hanan Samet. 1994. Data-parallel spatial join algorithms. In *Proceedings of the 1994 International Conference on Parallel Processing (ICPP'94)*, Vol. 3. IEEE Computer Society, Los Alamitos, CA, 227–234. DOI: https://doi.org/10.1109/ICPP.1994.82

[23] Edwin H. Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Trans. Database Syst.* 32, 1, Article 7 (Mar. 2007). DOI: https://doi.org/10.1145/1206049.1206056

[24] M. Kornacker and J. Erickson. 2012. Cloudera Impala: Real Time Queries in Apache Hadoop, For Real. Retrieved April 11, 2018 from http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real.

[25] Kalev Leetaru and Philip A. Schrodt. 2013. GDELT: Global data on events, language, and tone, 1979–2012. In *Proceedings of the International Studies Association Annual Conference* (2013).

[26] Nikos Mamoulis and Dimitris Papadias. 2001. Multiway spatial joins. *ACM Trans. Database Syst.* 26, 4 (Dec. 2001), 424–475. DOI: https://doi.org/10.1145/503099.503101

[27] Randal C. Nelson and Hanan Samet. 1986. A consistent hierarchical representation for vector data. *SIGGRAPH Comput. Graph.* 20, 4 (Aug. 1986), 197–206. DOI: https://doi.org/10.1145/15886.15908

[28] Gustavo Niemeyer. 2008. Geohash. Retrieved June 6, 2018 from https://en.wikipedia.org/wiki/Geohash.

[29] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.* 9, 1 (Mar. 1984), 38–71. DOI: https://doi.org/10.1145/348.318586

[30] OpenStreetMap. 2004. OpenStreetMap. Retrieved May 17, 2018 from https://openstreetmap.org.

[31] Jack A. Orenstein. 1982. Multidimensional tries used for associative searching. *Inf. Process. Lett.* 14, 4 (1982), 150–157. DOI: https://doi.org/10.1016/0020-0190(82)90027-8

[32] GDELT Project. 2014. The GDELT Project. Retrieved May 3, 2018 from https://www.gdeltproject.org/.

[33] Mansour Raad. 2013. BigData Spatial Joins. Retrieved April 11, 2018 from http://thunderheadxpler.blogspot.com/2013/10/bigdata-spatial-joins.html.

[34] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB'87)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 507–518. http://dl.acm.org/citation.cfm?id=645914.671636

[35] R. Sriharsha. 2015. Magellan: Geospatial Analytics on Spark. Retrieved May 1, 2018 from https://hortonworks.com/blog/magellan-geospatial-analytics-in-spark.

[36] Mingjie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A distributed in-memory data management system for big spatial data. *Proc. VLDB Endow.* 9, 13 (Sep. 2016), 1565–1568. DOI: https://doi.org/10.14778/3007263.3007310

[37] New York City Taxi and Limousine Commission. 2016. TLC Trip Record Data. Retrieved May 3, 2018 from http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[38] Patrick Valduriez and Georges Gardarin. 1984. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Syst.* 9, 1 (Mar. 1984), 133–161. DOI: https://doi.org/10.1145/348.318590

[39] Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[40] Randall T. Whitman, Michael B. Park, Sarah M. Ambrose, and Erik G. Hoel. 2014. Spatial indexing and analytics on Hadoop. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'14)*. ACM, New York, NY, 73–82. DOI: https://doi.org/10.1145/2666310.2666387

[41] Randall T. Whitman, Michael B. Park, Bryan G. Marsh, and Erik G. Hoel. 2017. Spatio-temporal join on Apache Spark. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'17)*. ACM, New York, NY, Article 20, 10 pages. DOI: https://doi.org/10.1145/3139958.3139963

[42] 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, 1071–1085. DOI: https://doi.org/10.1145/2882903.2915237

[43] Simin You, Jianting Zhang, and Le Gruenwald. 2015. Large-scale spatial join query processing in Cloud. In *Proceedings of the 2015 31st IEEE International Conference on Data Engineering Workshops* (2015), 34–41.

[44] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 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 (SIGSPATIAL'15)*. ACM, New York, NY, Article 70, 4 pages. DOI: https://doi.org/10.1145/2820783.2820860

[45] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

[46] Renyi Liu, Feng Zhang, Zhenhong Du, Jingwei Zhou, and Xinyue Ye. 2016. A new design of high-performance large-scale GIS computing at a finer spatial granularity: A case study of spatial join with spark for sustainability. *Sustainability (2071-1050)* 8, 9 (2016).

[47] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. 2009. SJMR: Parallelizing spatial join with mapreduce on clusters. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'09)*. IEEE, 1–8.

[48] Yunqin Zhong, Jizhong Han, Tieying Zhang, Zhenhua Li, Jinyun Fang, and Guihai Chen. 2012. Towards parallel spatial query processing for big spatial data. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW'12)*. IEEE Computer Society, Los Alamitos, CA, 2085–2094. DOI: https://doi.org/10.1109/IPDPSW.2012.245

Received June 2018; revised March 2019; accepted March 2019