Designing Distributed Geospatial Data-Intensive Applications

Ph.D. Course, **2022**

Instructors:

Prof. **Luca Foschini**, Associate Professor &

Dr. **Isam Mashhour Al Jawarneh**, Postdoctoral Research Fellow

{isam.aljawarneh3, Luca.foschini}@unibo.it

Department of Computer Science and Engineering (DISI), Università di Bologna

Part 2

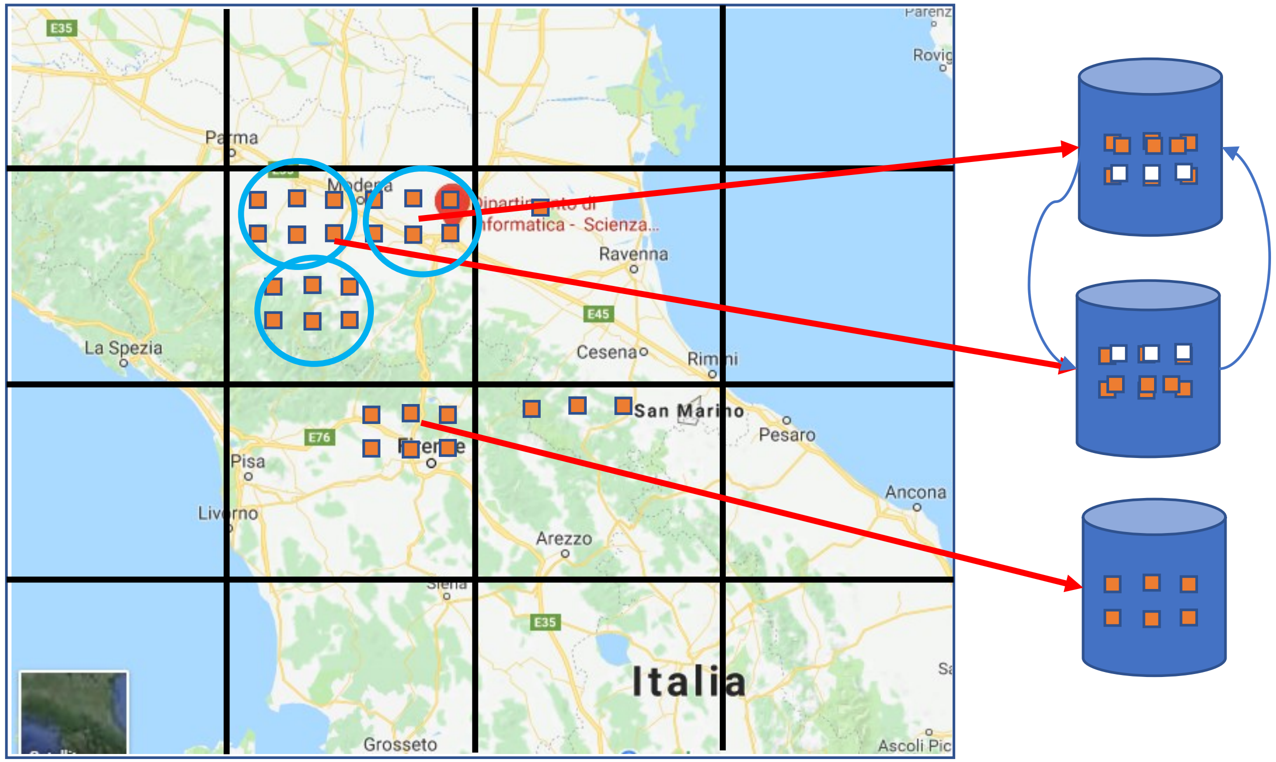
Designing highly efficient geospatial data-intensive solutions

27th July 2022

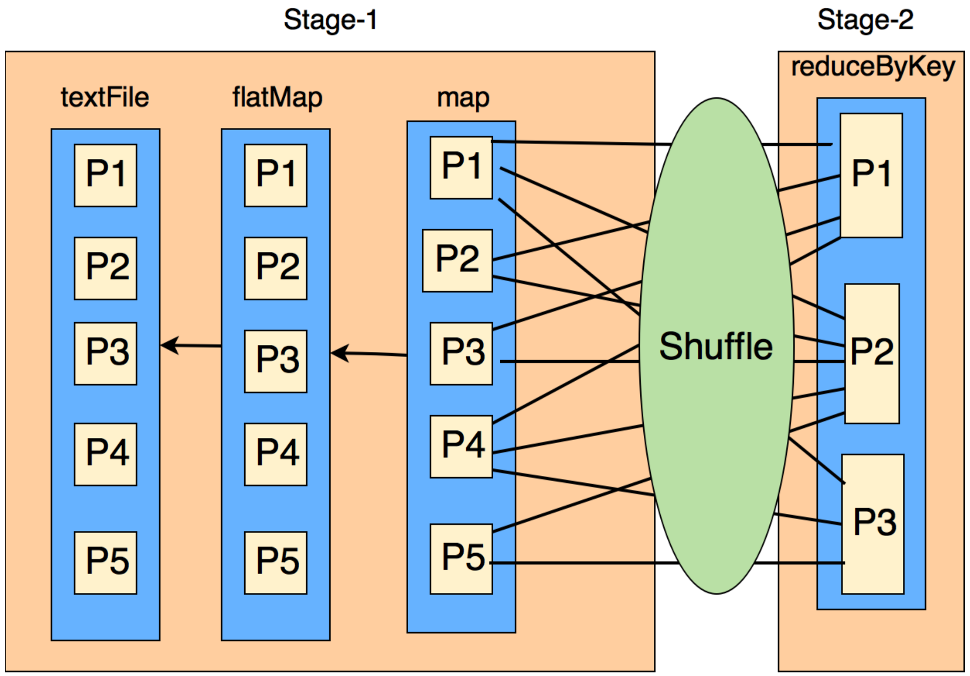
GeoMesa Spark spatial join

- GDELT is an archive containing location-indexed events from broadcast, print, and web news media worldwide dating back to 1979 until today
- FIPS Codes (shapefile) are Federal Information Processing Standard Publication codes, which uniquely identify **counties (polygons)** in the USA
- GDELT is a point geometries data set
  - How to tell which county each point belongs
    - By **join** GDELT points with the county that contains them from the FIPS **shapefile** → seems familiar?! (**Point in Polygon**)
- But which kind of join?!
  - **Join** in distributed settings is **costly**
    - Remember data **shuffling** is **computationally** **expensive**
    - Our target is to avoid shuffling as much as possible
      - Load **balancing** Vs. **co-locality** when **partitioning** geospatial data

Load balancing (smart city scenario)



In Spark join requires data to reside on the same partition



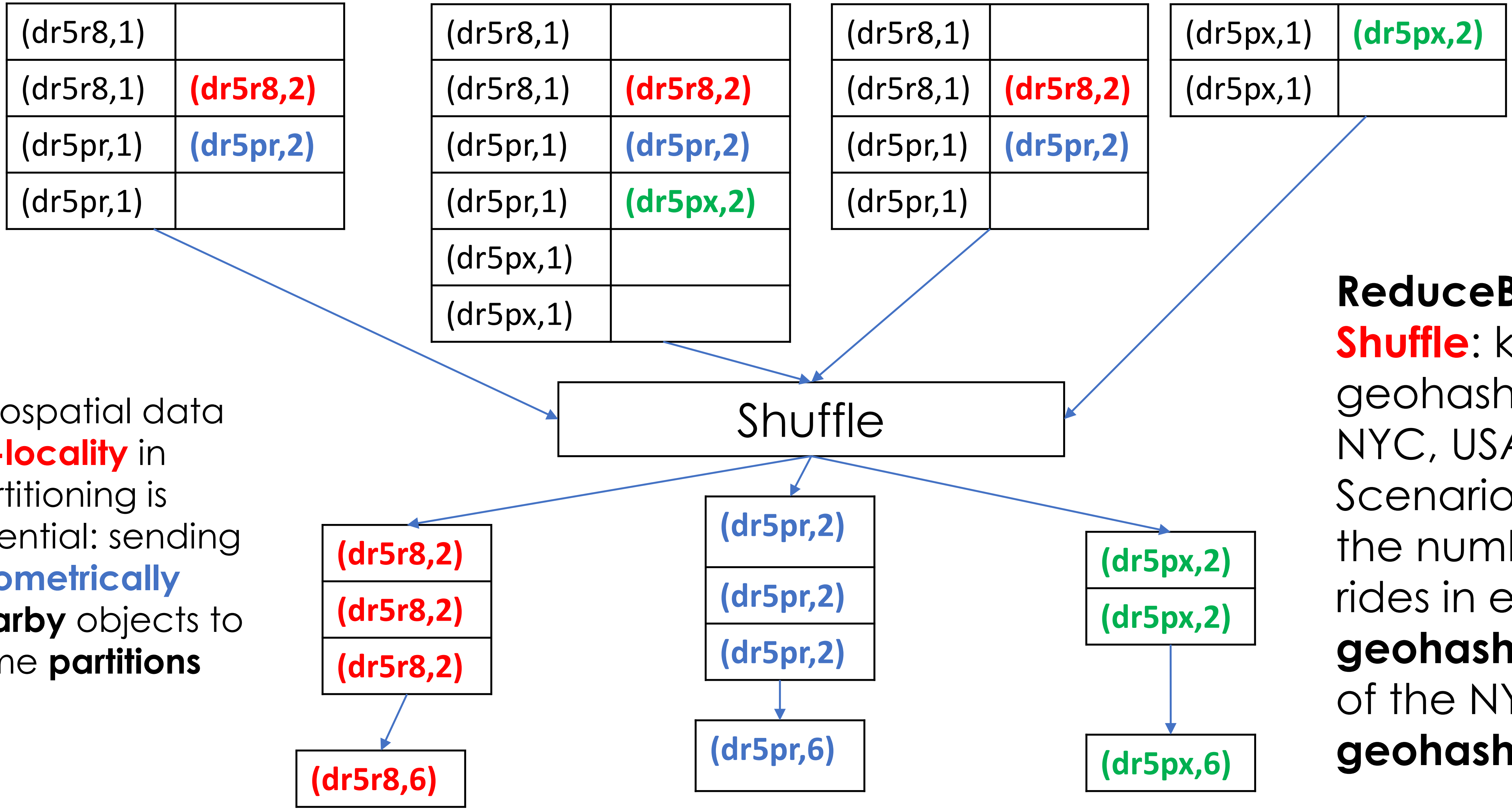
Is load balancing alone sufficient?!

Only load balancing = shuffling (huge toll) for co-location queries

Avoid shuffling

- So, lucky us, the number of **counties** is small to fit in the **fast memory**, circa 3000 records
  - we can **broadcast** the **counties (polygons)**
- In a conventional **Spark SQL join**, data is typically **shuffled** around the Spark Cluster **executors** depending on the **partitioners** of the **RDDs**,
- **Join key** is a geospatial field, Spark does not provide **over-the-counter partitioner** that can partition data in a way that preserves **spatial co-locality**
  - Shuffling data across nodes (and executors) is expensive,
    - Broadcasting small data (polygons) to each of the nodes, we obtain **performance gain**
  - Executors have a local copy of the data needed for join computation, hence shuffling is unneeded.
  - only useful for small broadcast data , such that it fits in the fast memory of the executors

Geospatial data **co-locality** in partitioning is essential: sending **geometrically nearby** objects to same **partitions**



ReduceByKey

Shuffle: key is a geohash value in NYC, USA

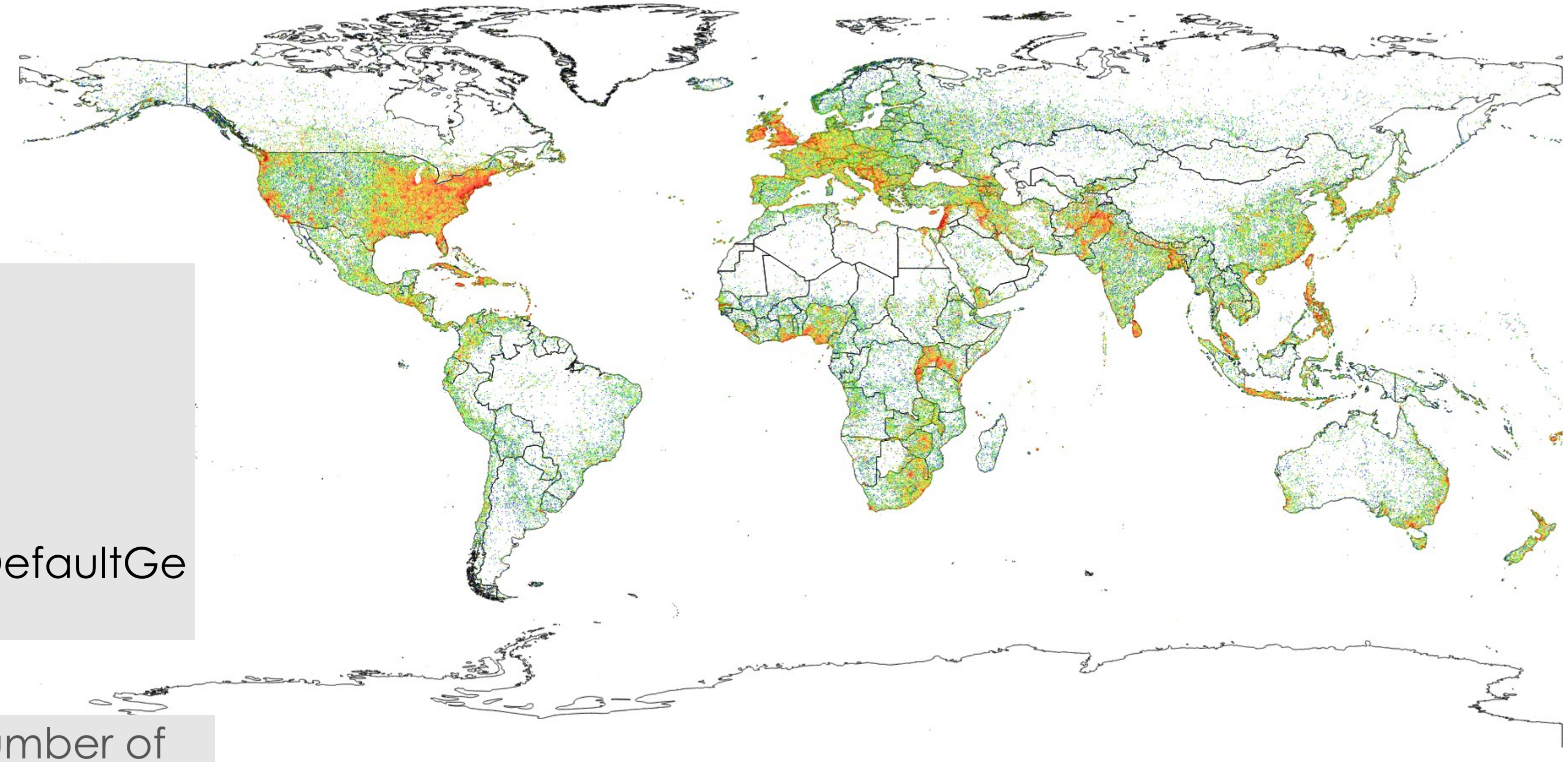
Scenario: **counting** the number of Taxi rides in each **geohash cell** (part of the NYC **geohash covering**)

GeoMesa with Spark

```
val f = ff.bbox("geom", -180, -90, 180, 90,
"EPSG:4326") val q = new Query("gdelt", f)
val queryRDD =
spatialRDDProvider.rdd(new
Configuration, sc, params, q, None)
```

```
//Project (in the relational sense)
the SimpleFeature to a 2-tuple
of (GeoHash, 1)

val discretized = queryRDD.map { f =>
(geomesa.utils.geohash.GeoHash(f.getDefaultGe
ometry.asInstanceOf[Point], 25), 1) }
```



```
//Then, group by grid cell and count the number of
features per cell.

val density = discretized.reduceByKey(_ + _)
density.collect.foreach(println)
```

[Code and Image source]

```scala
val fipsDF = spark.read.format("geomesa") .options(fipsParams) .option("geomesa.feature", "fips") .load()
val gdeltDF = spark.read.format("geomesa") .options(gdeltParams) .option("geomesa.feature", "gdelt")
.load()
```

```scala
import org.apache.spark.sql.functions.broadcast

val joinedDF = gdeltDF.join(broadcast(fipsDF), st_contains($"the_geom", $"geom"))
```

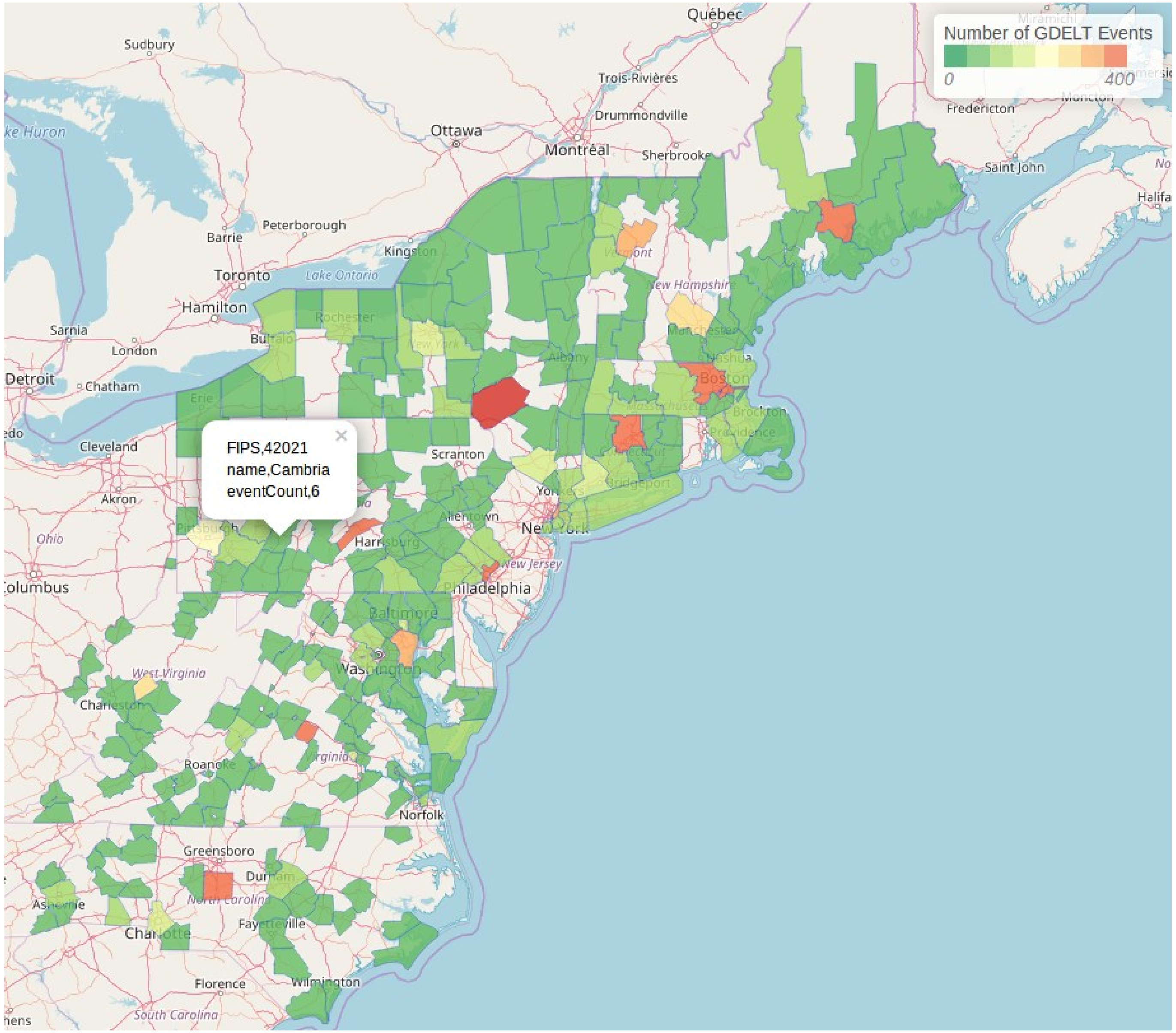
[Code source]()

st_contains takes two geometries as input, and it outputs whether the second geometry lies within the first one.

sending the FIPS data to each of the executors, then joining the two data sets based on whether the GDELT event occurred in the county

Aggregation

- A density map showing the distribution of GDELT events in the US
    - **Group** the data on FIPS (**polygons**) code
    - Counts distinct number of GDELT events (geospatial data **points**) in each polygon.
    - The result is used to **generate** a **geo-visualization** of the event density in each county (**polygon**)
- **Spatial join** is essential!



[Image Source](#)

Another spatial join example in GeoMesa

- NYC Taxi (**points**) is taxi trips data from NYC Taxi and Limo Commission
- GeoNames (**polygons**) is a geo-database consisting of circa 10 million geographical names
- Analysis that requires join
  - "Do taxi pickups centralize near certain points of interest?",
  - "Are people more likely to request a pickup or be dropped off at points of interest?".
- **Join** the two data sets (points, polygons) and aggregate **geo-statistics** over the result

Spatial non-equijoin

- GeoNames (**POI**) is a data set of points, and NYC Taxi offers the pickup and drop-off **points** of a taxi trip
  - it is unlikely that a trip starts or ends exactly on the labeled point of interest
  - So, equijoin is impossible
    - **D-within (within distance)** join → points (GeoNames and taxi trips) are within some tolerable **distance** of one another.

example

```
val joinedDF = geonamesNY .select(st_bufferPoint($"geom", lit(50)).as("buffer"), $"name", $"geonameId") .join(taxiDF, st_contains($"buffer", $"pickup_point"))
```

[Code source]

two UDFs

st_contains takes **two geometries** as input, and it outputs whether the second geometry **lies within** the first one.
st_bufferPoint takes a **point** and a **distance** in meters as input, and it outputs a circle around the point with radius equal to the provided distance.

transforms the geometry of each GeoName point into a **circle** with a radius of 50 meters and **joins** the result with the taxi records that had pickups anywhere in that circle

Now we have a **DataFrame** where each **point of interest (region, polygon)** in New York is combined with a **taxi record (spatial object, point)** where a pickup was issued from approximately that location.

Example: geo-stats

turn this into meaningful **statistics** about taxi habits in the **region**, we can do a GROUP BY operation and use some of **SparkSQL's aggregate functions**

Code source

```
val aggregateDF = joinedDF.groupBy($"geonameId")
.agg(first("name").as("name"), countDistinct($"trip_id")).as(s"numPickups"),
first("buffer").as("buffer"))
```

groups the data based on POI and **counts** the number of **distinct pickups**

```
val top10 = aggregateDF.orderBy($"numPickups".desc).take(10)
top10.foreach { row => println(row.getAs[String]("name") +
row.getAs[Int]("numPickups")) }
```

Top-N: Which POIs are popular depart locations, sort the results and look at the top ten

Hotel Gansevoort has the **most taxi pickups**

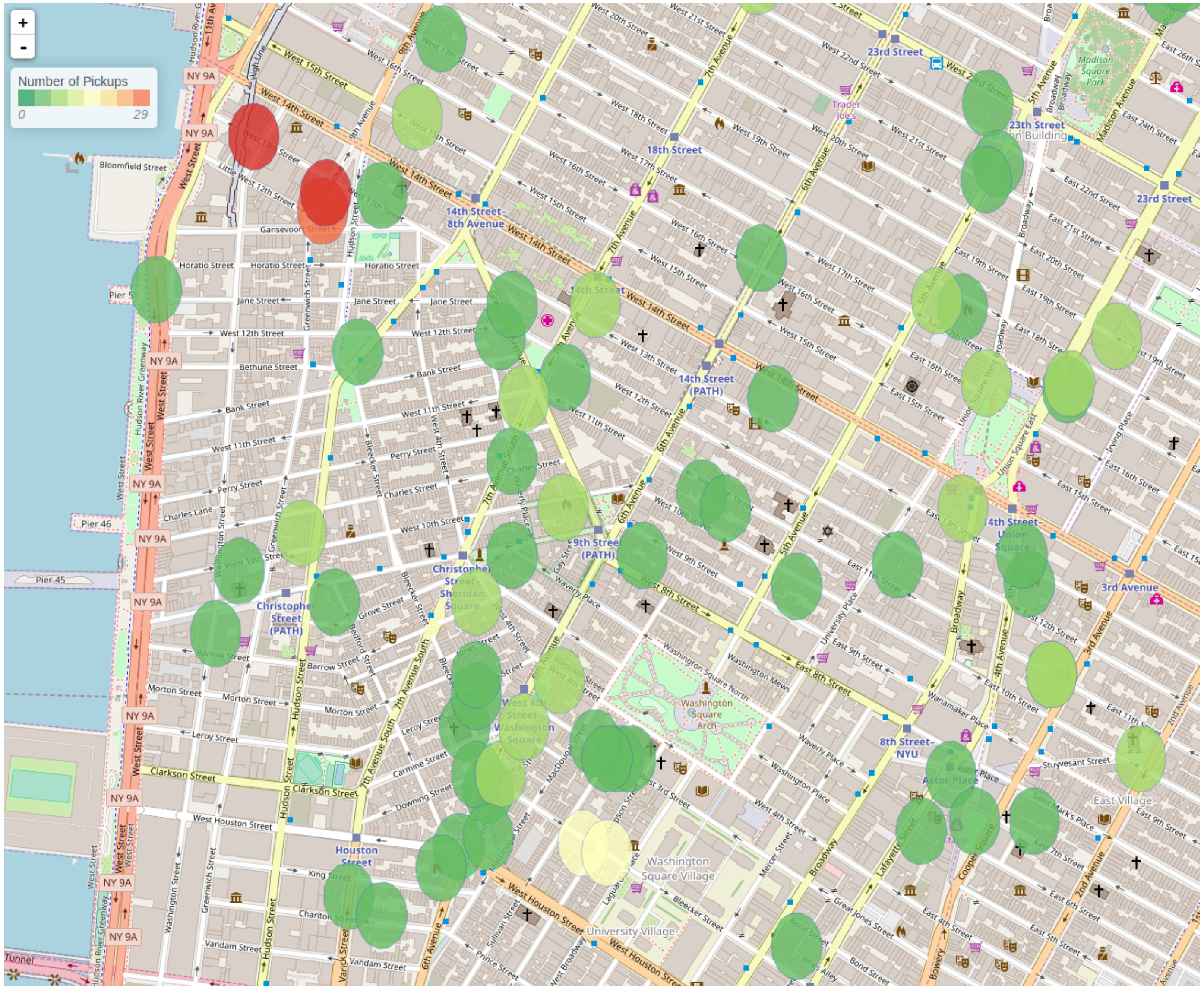
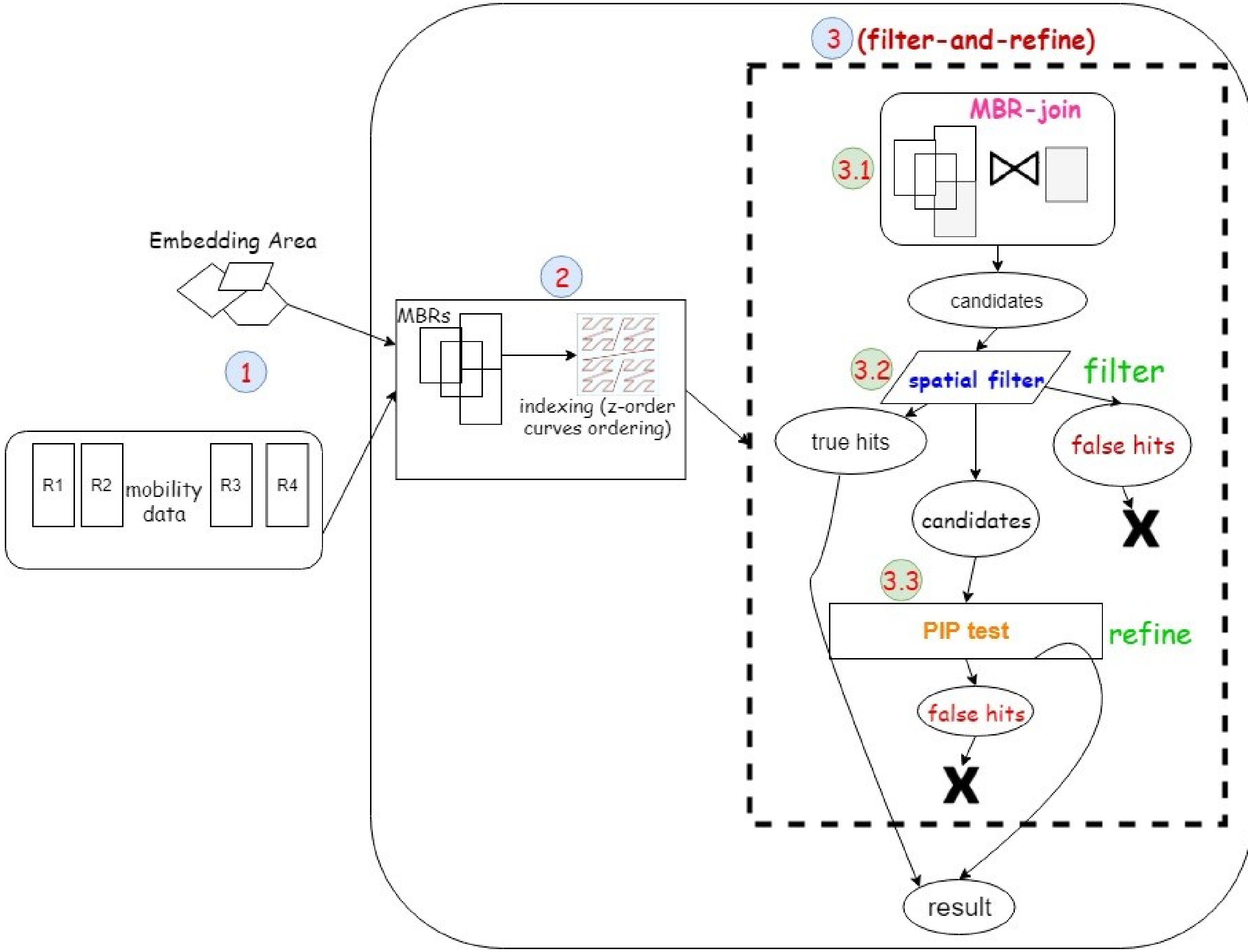


Image source

Filter-and-refine approach for spatial join

- Based on **dimensionality reduction**
  - Compute **MBR** for every **point**
  - Compute **MBR covering** of the **embedding area**
  - Perform a cheap **equi-join** to find which points fall within the embedding area (filter)
  - Use the ray casting algorithm to exclude **false positives** (refine)
- Adopted by Spark's Magellan and Geomesa



Filter-refine spatial join with Spark on GeoMesa: with QoS guarantees

- >200M NYC taxi trips

pickup_datetime	dropoff_datetime	passenger_count	trip_distance	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude
2016-03-26 15:39:13	2016-03-26 15:51:44	2	1.22	-73.99749755859375	40.756813049316406	-73.9789047241211	40.75257110595703
2016-03-26 17:33:38	2016-03-26 17:45:17	1	3.2	-73.86327362060547	40.76980972290039	-73.91075897216797	40.772361755371094
2016-03-28 10:47:20	2016-03-28 11:03:16	1	2	-73.98033142089844	40.76011276245117	-73.99227905273438	40.73797607421875

[Table source]()

Geospatial data skewness

- ~110M pickups are in a single geohash (Manhattan)

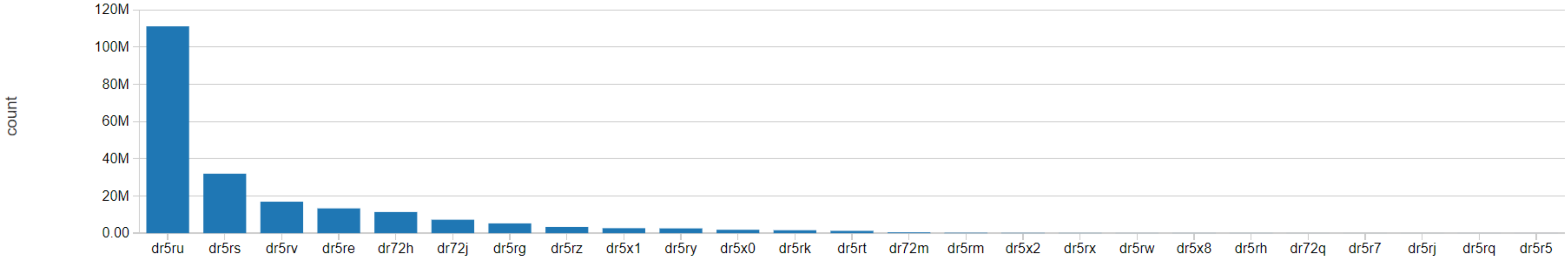


Figure source

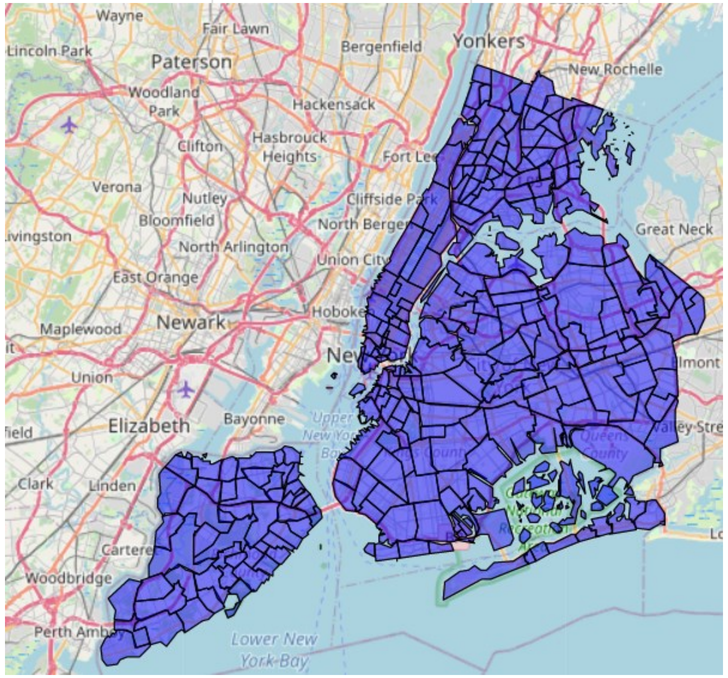
Polygon Data

OBJECTID	Shape_Leng	the_geom	Shape_Area	zone	LocationID	borough
	0.116357453189	MULTIPOLYGON (((-74.1844529999996 40.694995999999904, -74.18448899999999 40.69509499999987, -74.18449799999996 40.69518499999987, -74.18438099999997 40.69587799999989, -74.18428199999994 40.6962109999999, -74.18402099999997 40.697074999999884, -74.18391299999996 40.69750699999986, -74.18375099999997 40.69779499999988, -74.18363399999998 40.6983259999999, -74.18356199999994 40.698451999999875, -74.18354399999998 40.69855999999988, -74.18350799999996 40.69870399999992, -74.18327399999998 40.70008999999988, -74.18315699999994 40.701214999999884, -74.18316599999997 40.702384999999886, -74.18313899999998 40.7026279999999, -74.18309399999998 40.7028529999999, -74.18299499999995 40.70315899999985, -74.18284199999994 40.70346499999989, -74.18264399999998 40.70373499999988, -74.18242799999996 40.70395099999992, -74.18220299999996 40.704139999999896, -74.18203199999994 40.70425699999987, -74.18180699999994 40.7043919999999, -74.18157299999996 40.70449999999988, -74.18132099999997 40.70460799999...	0.0007823067885	Newark Airport	1	EWR
2	0.43346966679	MULTIPOLYGON (((-73.82337597260663 40.63898704717672, -73.82277105438692 40.63557691408512, -73.82265046764824	0.00486634037837	Jamaica Bay	2	Queens

Table source

• NYC Neighborhood Polygon Data

Image source



- We want to associate each pickup (point) with the appropriate NYC taxi zone (polygon)
    - This "*if point is within polygon*" query predicate would require comparision of >200M points to ~250 polygons in our example. (i.e. **worst case 50,000,000,000 expensive comparisons**)
- To constain joins we are leveraging the precomputed **geohash** information to significantly "prune" the solution space.
- We then evaluate the **geospatial predicate** st_contains($"polygon", $"pickupPoint") (to filter out false positives)
-

- get all GeoHashes Intersecting a Polygon
- Add the `polygon` Geometry Column using GeoMesa + explode intersecting GeoHashes

neighborhood	polygon	geohashes	geohash
Newark Airport	MULTIPOLYGON (((-74.18445299999996 40.694995999999904, -74.18448899999999 40.69509499999987, -74.18449799999996 40.69518499999987, -74.18438099999997 40.69587799999989, -74.18428199999994 40.6962109999999, -74.18402099999997 40.697074999999884, -74.18391299999996 40.69750699999986, -74.18375099999997 40.69779499999988, -74.18363399999998 40.6983259999999, -74.18356199999994 40.698451999999875, -74.18354399999998 40.69855999999988, -74.18350799999996 40.69870399999992, -74.18327399999998 40.70008999999988, -74.18315699999994 40.701214999999884, -74.18316599999997 40.702384999999886, -74.18313899999998 40.7026279999999, -74.18309399999998 40.7028529999999, -74.18299499999995 40.70315899999985, -74.18284199999994 40.70346499999989, -74.18264399999998 40.70373499999988, -74.18242799999996 40.70395099999992, -74.18220299999996 40.704139999999896, -74.18203199999994 40.70425699999987, -74.18180699999994 40.7043919999999, -74.18157299999996 40.70449999999988, -74.18132099999997 40.70460799999...	["dr5r8","dr5pr","dr5px","dr5r2"]	dr5r8
Newark Airport	MULTIPOLYGON (((-74.18445299999996 40.694995999999904, -74.18448899999999 40.69509499999987, -74.18449799999996 40.69518499999987, -74.18438099999997 40.69587799999989, -74.18428199999994 40.6962109999999, -74.18402099999997 40.697074999999884, -74.18391299999996	["dr5r8","dr5pr","dr5px","dr5r2"]	dr5pr

[Table source](#)

Efficient spatial join

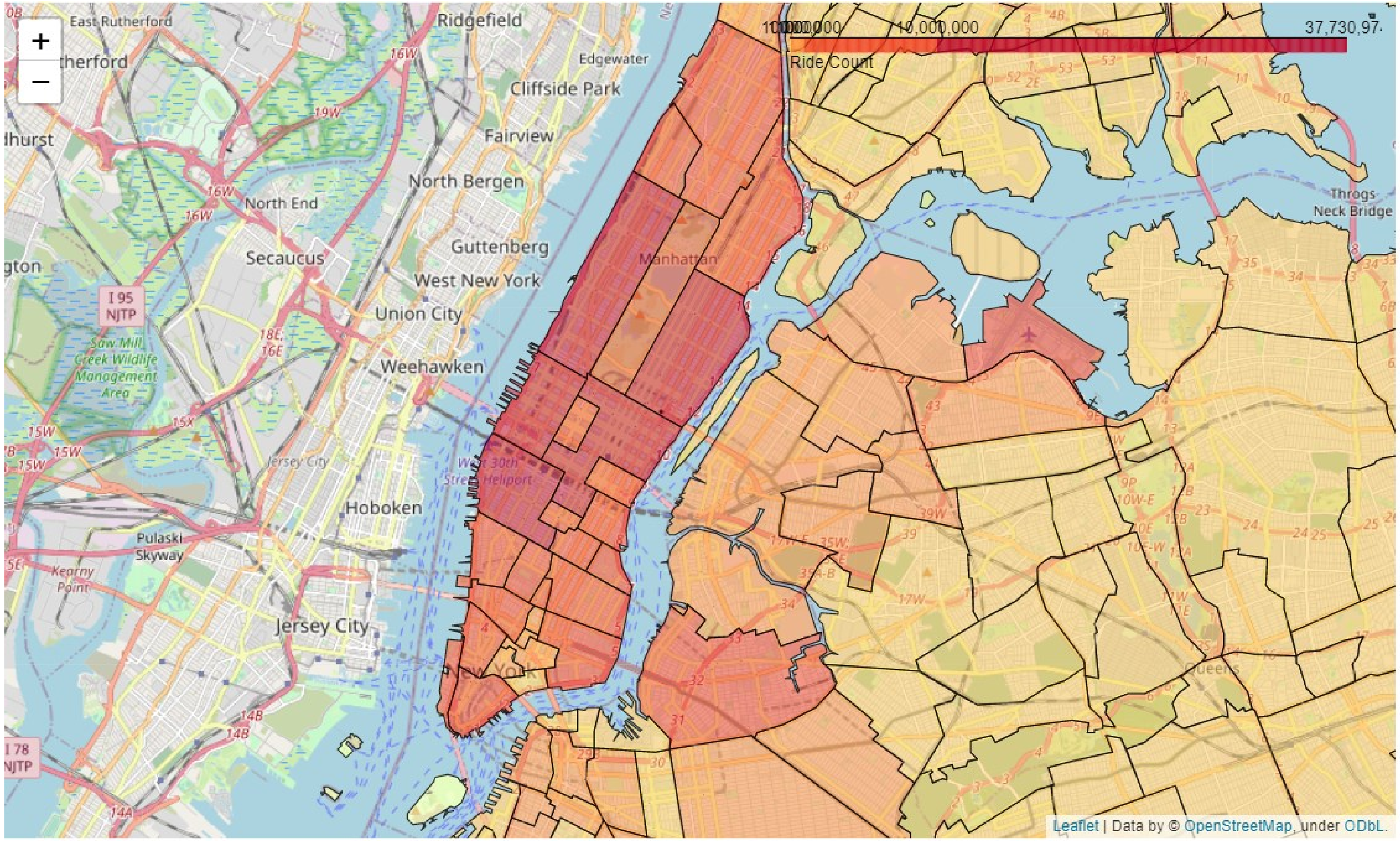
- Spatial **Join**: predicate = **Point within Polygon**
- Trips → spatial **points**
- neighborhoodsDF → **polygons**
- **===** → **filter** stage
- **st_contains** → **refinement** stage (**ray casting**) → **expensive**

```
val joined = trips.join( neighborhoodsDF.as("R"),
// short circuit on geohash and apply geospatial predicate
when necessary $"L.pickup_geohash_25" === $"R.geohash"
&& st_contains($"polygon", $"pickupPoint") )
```

pickupPoint	dropoffPoint	pickup_geohash_25	dropoff_geohash_25	neighborhood
POINT (-73.99749755859375 40.756813049316406)	POINT (-73.9789047241211 40.75257110595703)	dr5ru	dr5ru	East Chelsea
POINT (-73.86327362060547 40.76980972290039)	POINT (-73.91075897216797 40.772361755371094)	dr5rz	dr5ry	LaGuardia Airport
POINT (-73.98033142089844 40.76011276245117)	POINT (-73.99227905273438 40.73797607421875)	dr5ru	dr5ru	Times Sq/Theatre District

Table source

Geo-visualization of spatial join results



Map Pickup Density by Neighborhood

Image source

Summary: GeoMesa

- GeoMesa also provides RDD API, DataFrame API and Spatial SQL API so that the user can run spatial queries on Apache Spark.
- supports range query and join query.
- use R-Tree spatial partitioning technique to decrease the computation overhead.
  - uses a grid file as the local index per DataFrame partition. Grid file is a simple 2D index but cannot well handle spatial data skewness in contrast to R-Tree or Quad-Tree index.
  - does not remove duplicates introduced by partitioning the data and hence cannot guarantee
    join query accuracy.
- GeoMesa does not support parallel map rendering. Its user has to collect the big dataset to a single machine then visualize it as a low resolution map image.

Apache Sedona (previously GeoSpark)

- Apache Sedona is a **cluster computing** system (**full-fledged**) for **loading**, **processing** and **analyzing** large-scale spatial data.
  - Extends existing **Cloud-based** computing systems, such as Apache **Spark** and Apache **Flink**,
  - Extends the core engine of Apache **Spark** and **SparkSQL** to support **spatial data types**, **indexes**, and **geometrical operations** at scale.
    - Extends the Resilient Distributed Datasets (**RDDs**) concept to support **spatial** data.
    - Out-of-the-box Spatial Resilient Distributed Dataset (**SRDD**), which provides **in-house** support for **geometrical** and **distance** operations necessary for **processing geospatial data**
    - **Spatial RDD** provides an Application Programming Interface (**API**) for Apache **Spark** programmers to easily develop their spatial analysis programs using **operational** (e.g., Java and **Scala**) and **declarative** (i.e., **SQL**) languages
  - Map visualization function of GeoSpark creates high resolution maps in parallel (**GeoSparkViz**)

Sedona (previously GeoSpark)

- Over-the-counter **distributed Spatial Datasets** and **Spatial SQL** that efficiently **load**, **process**, and **analyze** large-scale spatial data in distributed computing environments
- ETL , partitioning, indexing, in-memory storing are supported intrinsically in GeoSpark and do not need direct intervention of the user, leaving the logistics handling to the underlying engine

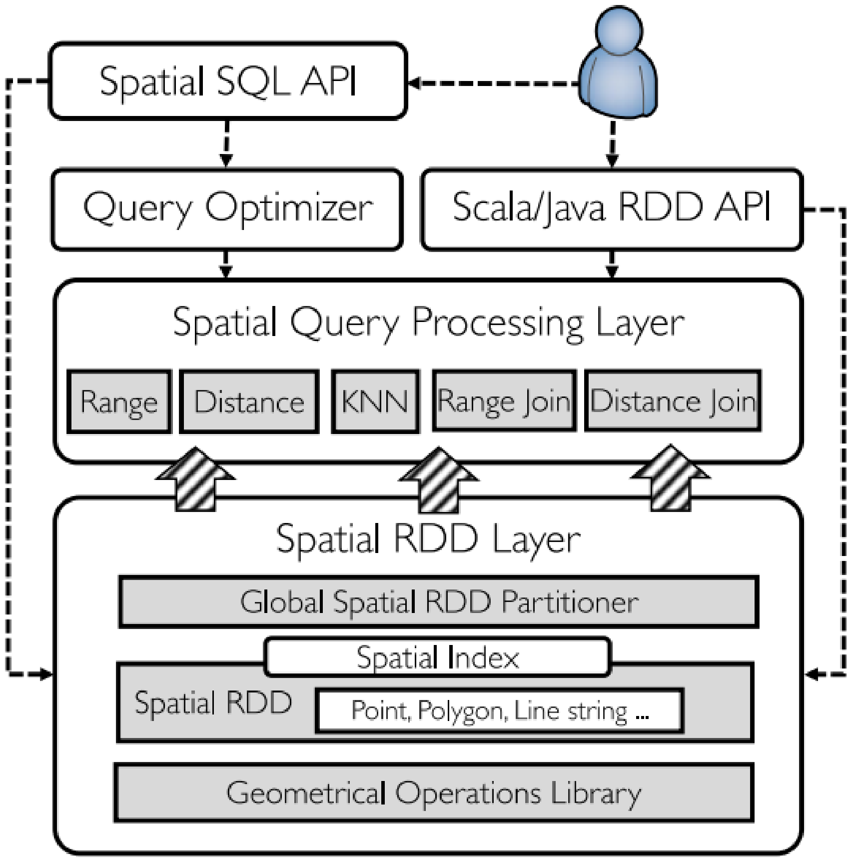
- Consists of three **layers**: **Spark** Layer, **Spatial RDD** Layer and **Spatial Query Processing** Layer
  - **Spatial RDD** Layer → three novel Spatial Resilient Distributed Datasets (**SRDDs**) which extend plain Spark RDD for supporting geometrical and spatial objects with data partitioning and indexing (pointRDD, RectangleRDD, PolygonRDD)
  - **Spatial Query Processing** Layer executes spatial queries (e.g., **Spatial Join**) on **SRDDs**
    - spatial **aggregation**, **autocorrelation** and **co-location**

GeoSpark Layered architecture

- The Spatial Resilient Distributed Dataset (**SRDD**) Layer
  - Extends Spark with Spatial RDDs (SRDDs) which efficiently partitions spatial data objects across a Spark computing cluster
- The Spatial **Query Processing** Layer
  - Execute spatial query predicates on Spatial RDDs
  - Efficient implementation of common spatial query predicates, e.g., range, distance, spatial k-nearest neighbors, **range join (within)** and **distance join** (**within distance**).
  - Novel optimizer that considers the **running time cost** and shuffles several queries to select a performant query execution plan
    - Two types of optimizations

    (1) **cost-based join** query **optimization**: selecting the fastest spatial join algorithm depending on Spatial RDDs input size

    (2) Predicate **pushdown**: detect the spatial **predicates** which **filter** the spatial data and **push** them **down** to the beginning of the spatial query plan (near data sources) to reduce data size and avoid shuffling as much as possible



Apache Sedona architecture

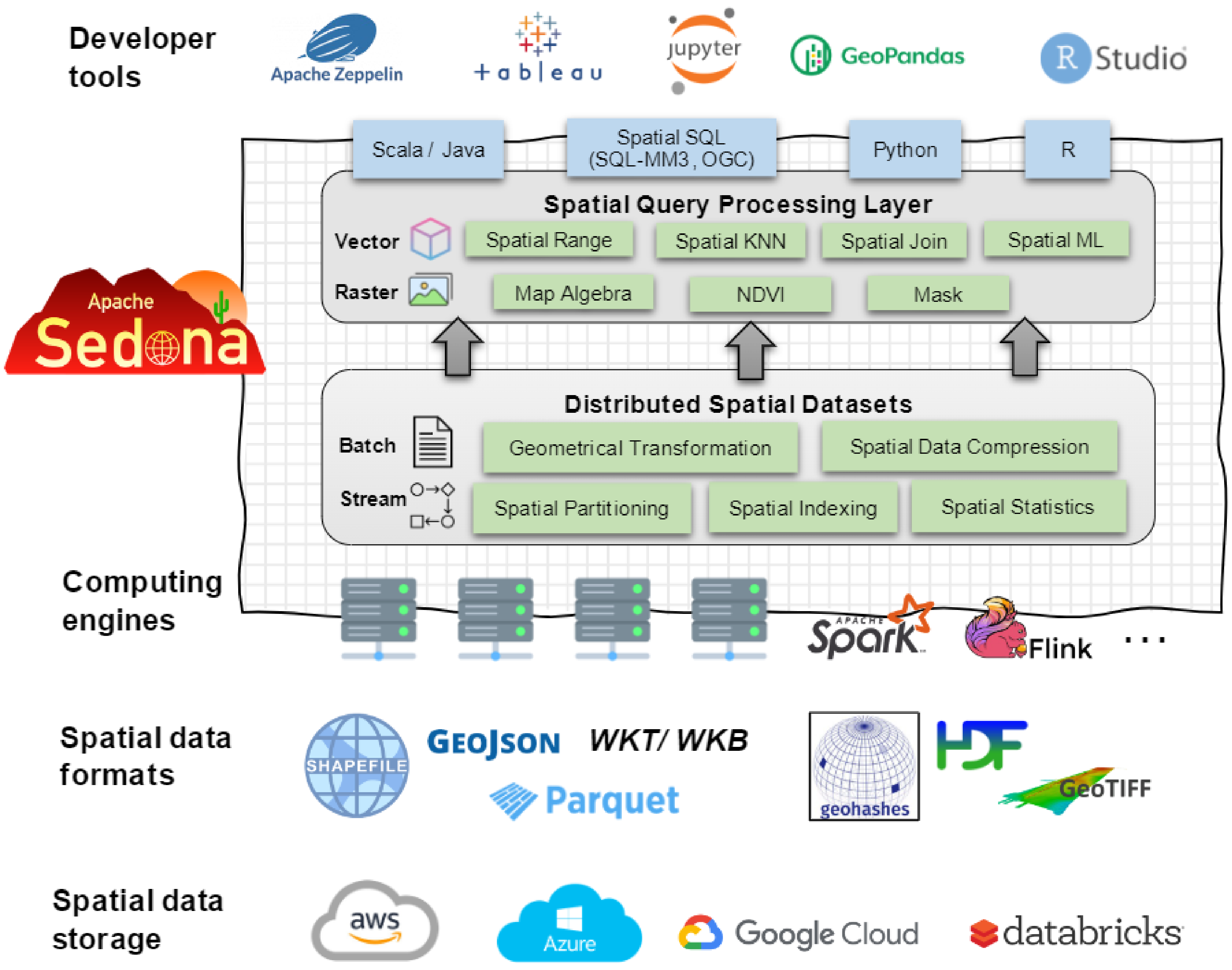


Image source

spatial RDD

- **PointRDD:** 2D Point objects (representing points on the surface of the earth), and their format is as follows: **<Longitude, Latitude>**
- **RectangleRDD:** **regularly sized** rectangular objects, format: <PointA(Longitute, Latitude), PointB(Longitute, Latitude)>
- **PolygonRDD:** **irregularly sized** format : <**PointA**(Longitute,Latitude), **PointB**(Longitute,Latitude), **PointC**…>

id	geom
1	**POINT** (21 52)
1	**POINT** (23 42)
1	**POINT** (26 32)

id	geom
1	**MULTIPOINT** ((19.511463 51.765158), (19.446408 51.779752))

id	geom
1	**LINESTRING** (10 10, 20 20, 10 40)

id	geom
1	**POLYGON** ((19.51121 51.76426, 19.51056 51.76583, 19.51216 51.76599, 19.5128 51.76448, 19.51121 51.76426))

What is missing!

- **Heterogeneous data sources**
  - Various file (**CSV**, **GeoJSON** , **NetCDF**, **GRIB** and ESRI **Shapefile**)
    - Spark does not **over-the-counter** understand those formats for spatial data.
- **Spatial partitioning**
  - Default data partitioner in Spark does not **preserve the spatial proximity** objects (spatial co-locality)
- **Spatial indexing**
  - Spark does not natively support **spatial indexing** such (e.g., **Quad-Tree** and **R-Tree**).
    - Maintaining a tree-based spatial index imposes additional 15% **storage space overhead**
      - A **global spatial index** for all spatial objects in the master node of the computing cluster is not a good idea

Example challenge in native Spark

Spatial **KNN** query: 20 nearest neighbor objects for a **query point** (5.0, 7.0) from **points** table

SELECT * **FROM** points
ORDERED BY (points .x – 5.0) * (points .x – 5.0) +
(points .y - 7.0) * (points .y - 7.0)
LIMIT 20.

Partitioning

- State-of-the-art **spatial data partitioning** techniques: uniform **grid**, **R-tree**, **Quad-Tree**, and **KDB-Tree**.
  - Partitions data based upon the **spatial proximity** among
    spatial objects to achieve load balancing in the Spark cluster
- Partitions a Spatial RDD in accordance with spatial data distribution
- Group spatial objects into the same partition based upon their spatial proximity (**spatial proximity preservation**)
  - Spatial partitioning speeds up spatial join query
  - A performant spatial partitioning approach keeps Spatial RDD partitions load-balanced

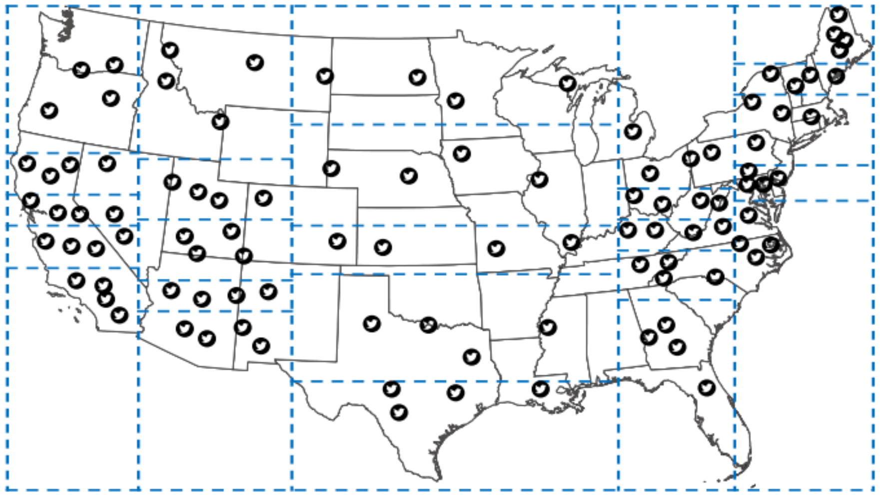
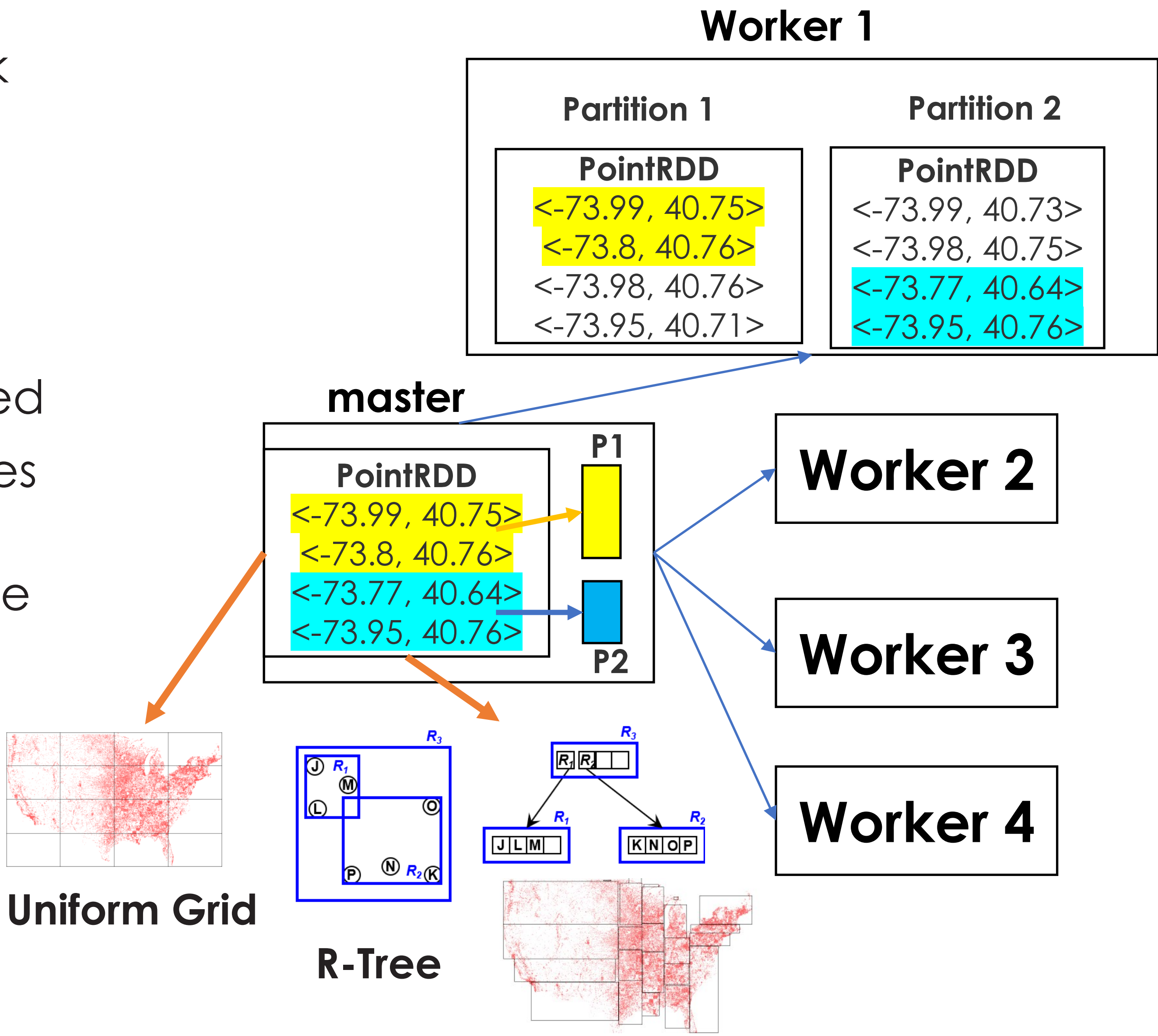


Image source
tweets in U.S.A

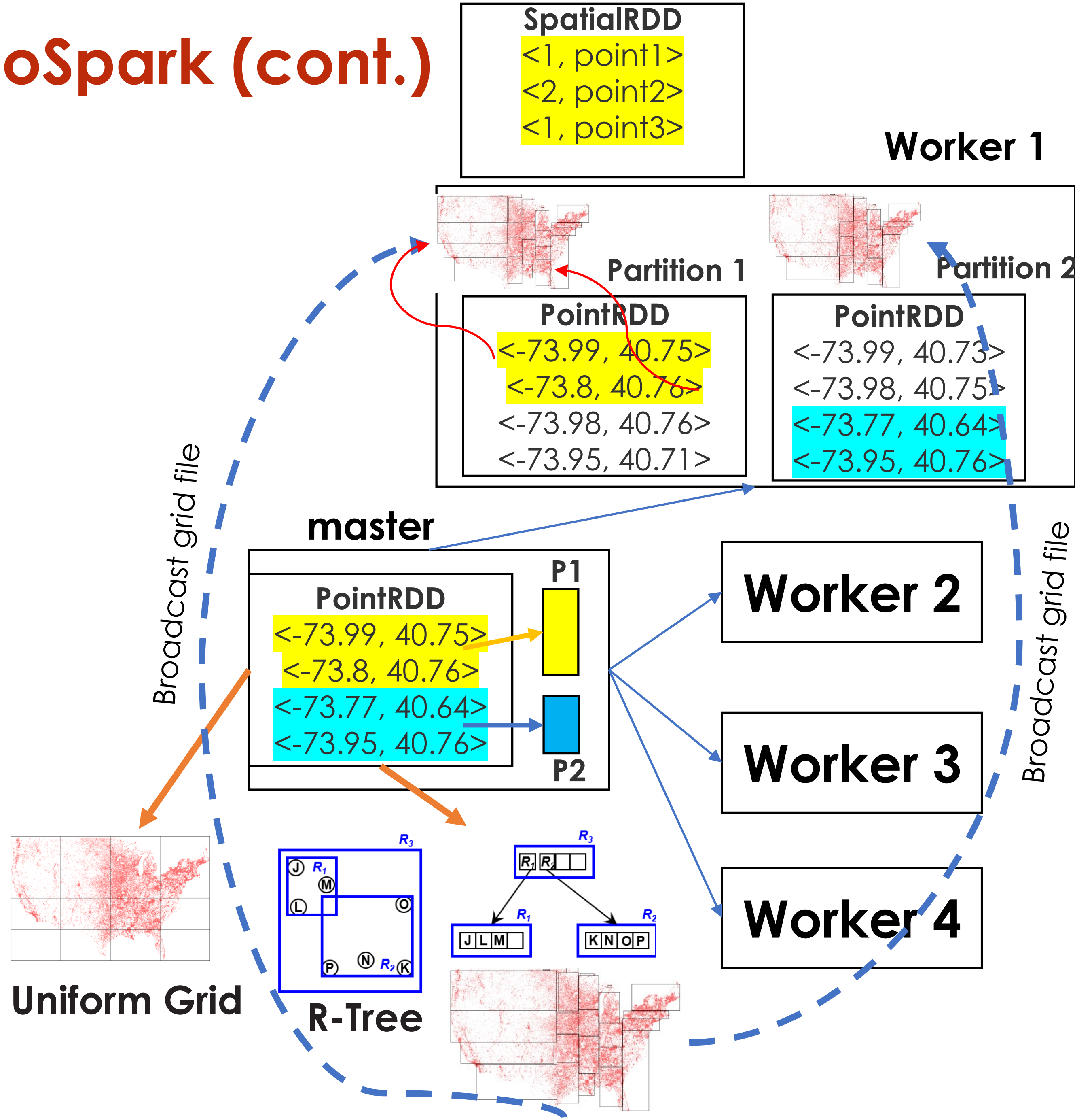
Spatial-aware partitioning in GeoSpark

- **Step 1: Building a global spatial grid file**
  - Samples Spatial RDDs in partitions to Spark master → a subset of entire Spatial RDD
  - The subset has the same data distribution of the original Spatial RDD
    - Load balancing & spatial locality (objects space proximity) are preserved
    - after sampling, a spatial data structures is applied to **divide the sampled data into partitions** at the Spark master node (Uniform **Grid** , tree-based - **R-Tree**, **Quad-Tree**, **KD-Tree**)
      - Tree-based → collects the leaf node boundaries into a **grid file**



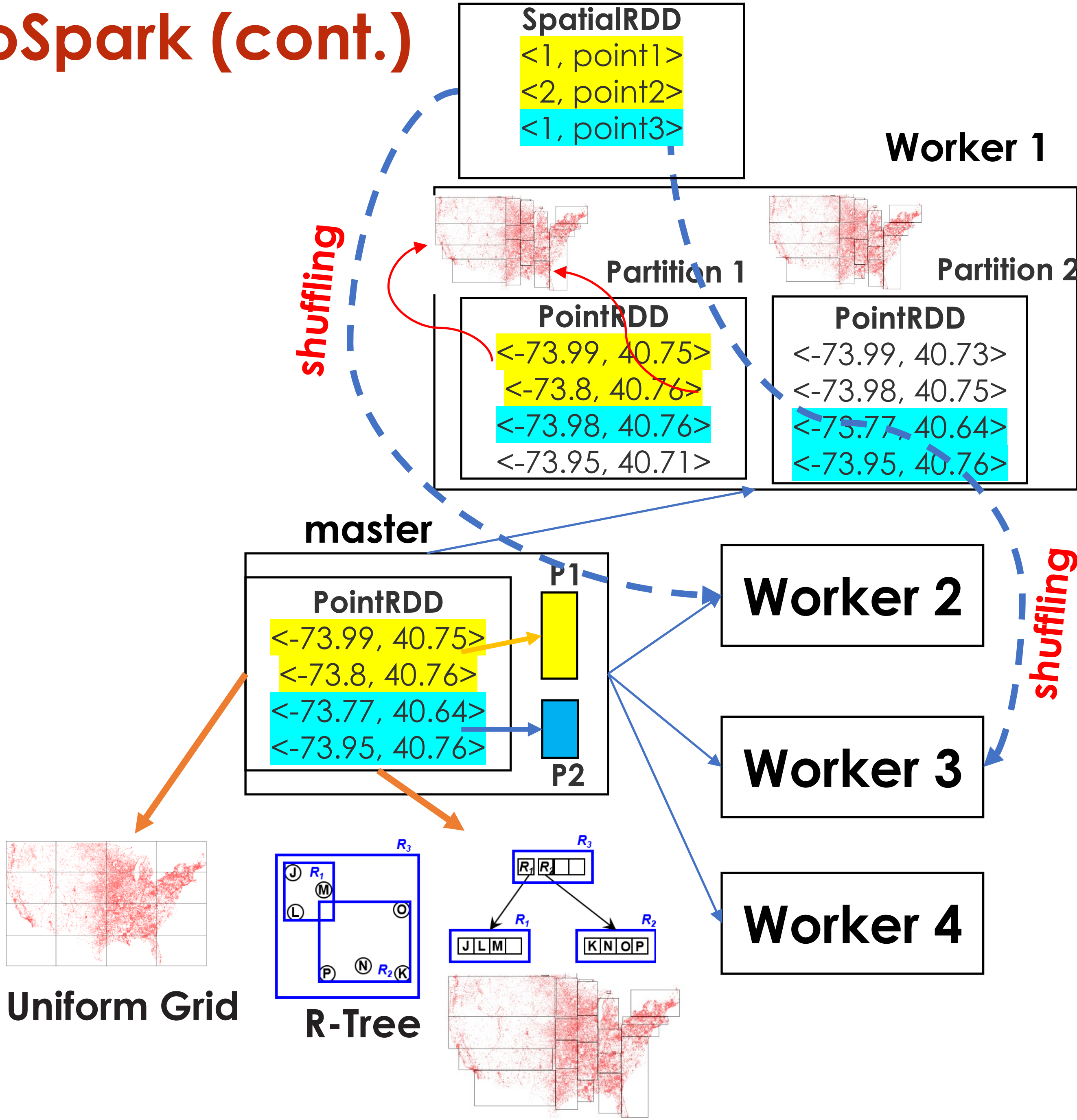
Spatial-aware partitioning in GeoSpark (cont.)

- **Step 2: Assigning a grid cell ID to each object**
  - After building a global grid file, check the grid cell to which each spatial objects **belongs (PIP test)**, then **repartition** the Spatial RDD considering the grid cells IDs
  - **Broadcast** the **grid** files to every original Spatial RDD partition in **worker** nodes
  - **Check** every local spatial object against the grid file. Store the result in a new Spatial RDD in the <Key, Value> format
    - If a local spatial object **intersects** (**spatial predicate**) a grid cell, assign a grid cell ID to the object with the <cell ID, object> format



Spatial-aware partitioning in GeoSpark (cont.)

- **Step 3: Re-partitioning SRDD across the cluster**
  - repartition the Spatial RDD by Key (grid cell ID)
    - spatial objects with **same Key** (falling within the same grid cell are sent to the **same partition** (spatial **co-locality**, preserving **proximity**).
    - Huge data **shuffling** across the cluster

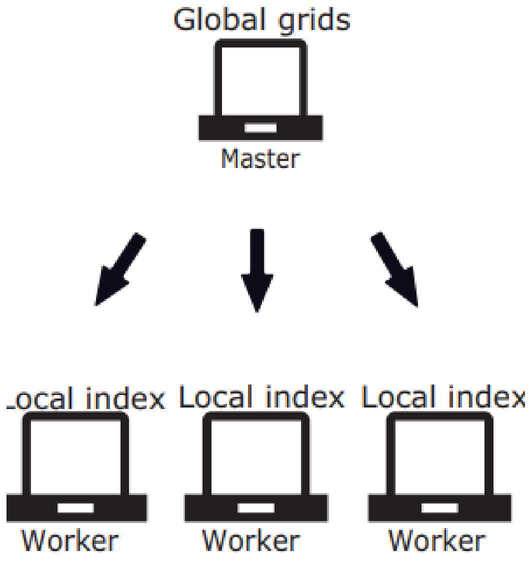


Summary of spatial data partitioning in GeoSpark

- We have one global grid for data partitioning
- Spatial proximity is preserved as it follows:
  - Divide the embedding space into non-equally sized grid cells which construct a **global grid file**
  - Check each object in the SpatialRDD and attach this object to the grid cell with which it intersects
- **Preserving spatial proximity** guarantees reducing the **data shuffling** across the cluster worker nodes and avoiding geometrical calculations on partitions that do not have relevant data

SRDD Indexing

- **R-tree** → groups **nearby objects (preserving spatial proximity)** and represent them with a **MBR** in the next higher-level node of the tree
  - Objects **MBRs** that do not **intersect** with a **higher-level node** MBR can not **intersect** with any of the objects in its **lower-levels** (**child** nodes)
- Spatial objects are organized using their MBRs instead of their real geometries
  - Queries utilizing the **spatial index** respect the **filter-refinement** approach
    - **Filter** → search for candidate spatial objects MBRs that intersect (or are contained within) with the query object's MBRs (MBR-join, cheap)
    - **Refinement** → check the **spatial relation** between the **candidate** objects (resulted from the **MBR-join**) and the query object (**real geometries**) and retrieve only spatial objects (real geometries) that **geometrically satisfy** the required **spatial relationship** (within, intersect, etc.,)
- Spatial IndexRDDs → **quadtree** and **R-tree**
  - **Local indexes** (local spatial indexes, e.g., R-Tree or Quad-Tree) are created for each **SRDD** data **partition**
    - based on a tradeoff between indexing overhead (space & time) and query selectivity
    - **speed up** performance gain



	Spatial data type	Approach	Spatial indexing	Queries	Optimization	Temporal attribute	Streaming processing
GeoSpark [34], [35], [37]	Generic	RDD, DataFrame	Two-level	Range, Join, KNN	Query optimizer, object serializer	Not optimized	Not optimized
Simba [32]	Generic	DataFrame	Two-level	Range, Join, KNN, KNN join	Query optimizer	Not optimized	Not optimized
LocationSpark [29]	Generic	DataFrame	Two-level	Range, Join, KNN, KNN join	Query optimizer	Not optimized	Not optimized
GeoMesa [12]	Generic	RDD, DataFrame	Global grid file	Range,Join	-	Not optimized	Not optimized
Magellan [17]	Generic	DataFrame	-	Range,Join	-	Not optimized	Not optimized
SpatialSpark [33]	Generic	RDD	Two-level	Range, Join	-	-	-
SparkGIS [7]	Generic	RDD	Two-level	Range, Join, KNN	Resource-aware query rewriter	-	-
DST [31]	Trajectory	DataFrame	Two-level	Similarity search	-	Not optimized	Not optimized
DITA [27]	Trajectory	DataFrame	Two-level	Similarity join	Query optimizer	Not optimized	Not optimized
SciSpark [20]	Satellite image	RDD	-	Filter, Join	-	Not optimized	-
GeoSparkViz [36]	Raster map	RDD	-	Range, Join, Overlay	-	-	-
Geotrellis [14]	Raster map	RDD	-	Cropping, Warping, Map algebra	-	Not optimized	-
BinJoin [30]	Generic	RDD	Local index	Join	Query optimizer	Optimized	-

Source

Feature name	GeoSpark	Simba	Magellan	Spatial Spark	GeoMesa	Spatial Hadoop	Parallel Secondo	Hadoop GIS
RDD API	✓	✗	✗	✓	✓	✗	✗	✗
DataFrame API	✓	✓	✓	✗	✓	✗	✗	✗
Spatial SQL [11, 28]	✓	✗	✗	✗	✓	✓	✗	✗
Query optimization	✓	✓	✓	✗	✓	✗	✓	✗
Complex geometrical operations	✓	✗	✗	✗	✓	✓	✗	✗
Spatial indexing	R-Tree Quad-Tree	R-Tree Quad-Tree	✗	R-Tree	Grid file	R-Tree Quad-Tree	R-Tree	R-tree
Spatial partitioning	Multiple	Multiple	Z-Curve	R-Tree	R-Tree	Multiple	Uniform	SATO
Range / Distance query	✓	✓	✓	✓	✓	✓	✓	✓
KNN query	✓	✓	✗	✗	✗	✓	✗	✓
Range / Distance Join	✓	✓	✓	✓	✓	✓	✓	✓

Table source

Spatial Query Processing

- Supports spatial queries (e.g., Range query and Join query) for large-scale spatial datasets
  - **range** query, distance query, K Nearest Neighbors (**KNN**) query, **range join** query (**within** predicate) and distance **join** query (**within distance** predicate)

  - leverages the grid partitioned Spatial RDDs spatial indexing
- **Spatial Range Query**
  - Load target dataset,
  - partition data,
  - create a spatial index on each SRDD partition, if necessary,
  - **broadcast** the **query window** to each SRDD partition,
    - broadcasts the query window to each machine in the cluster

  - check the spatial predicate in each partition, and
    - if a spatial index exists, it follows the Filter and Refine model
    - truly qualified spatial objects are returned as the partition of
  - remove spatial objects duplicates that existed due to the data partitioning phase

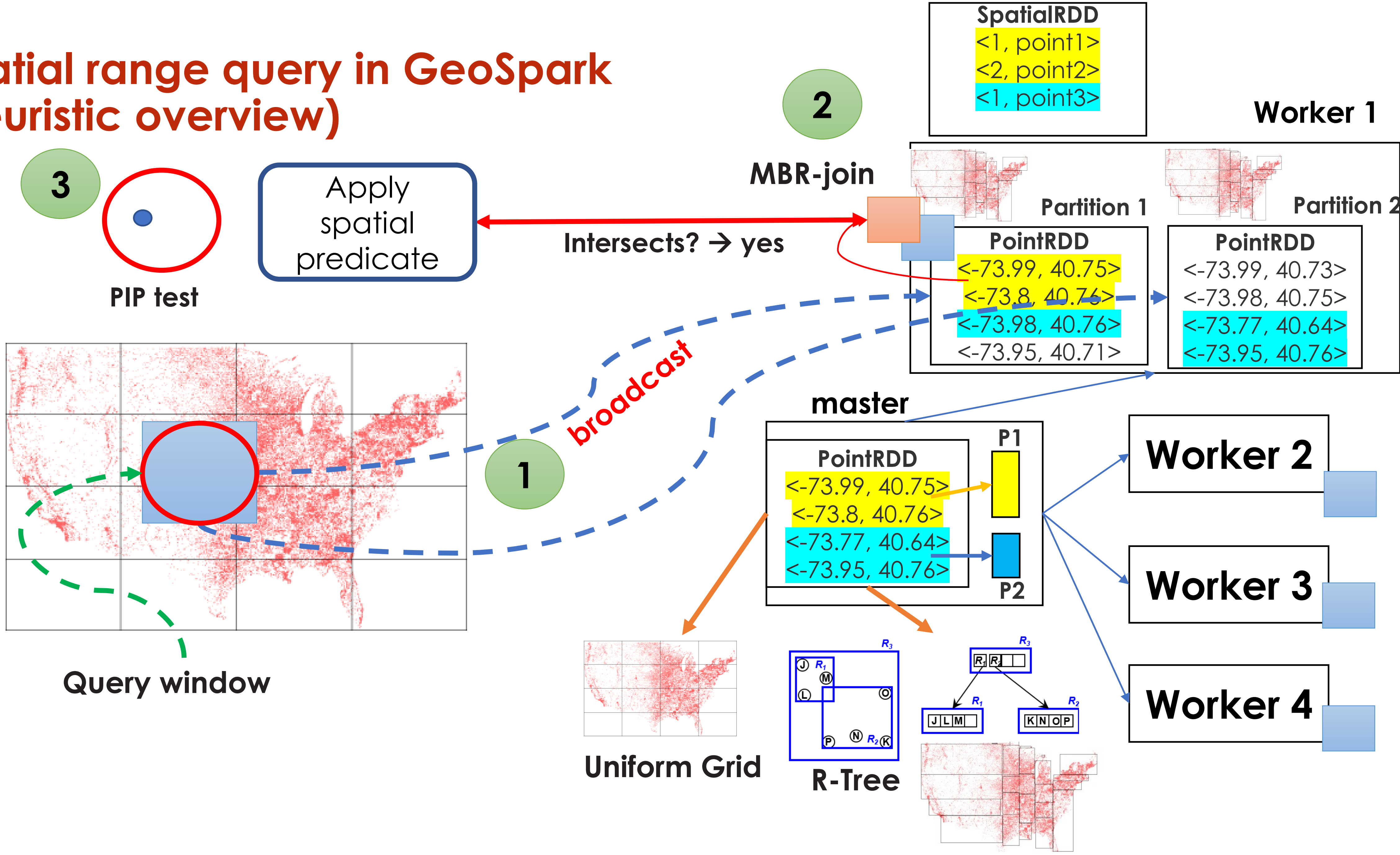
Spatial range query in GeoSpark

Algorithm 2: Range query and distance query

```
Data: A query window A, a Spatial RDD B and spatial relation predicate
Result: A Spatial RDD that contains objects that satisfy the predicate
foreach partition in the SRDD B do
    if an index exists then
        // Filter phase
        Query the spatial index of this partition using the window A's MBR;
        // Refine phase
        Check the spatial relation predicate using real shapes of A and candidate
          objects;
    else
        foreach object in this partition do
            Check spatial relation predicate between this object and A;
            Record this object if it is qualified;
Generate the result Spatial RDD;
```

[Algorithm source]()

Spatial range query in GeoSpark (heuristic overview)



Example range query in GeoSpark

Spatial query: find all **counties** that are **within** the given **polygon**

Code source

SQL API

```
spatialDf = sparkSession.sql(
  """
    |SELECT *
    |FROM spatialdf
    |WHERE ST_Contains (ST_PolygonFromEnvelope(1.0,100.0,1000.0,1100.0), newcountyshape)
  """.stripMargin)
spatialDf.createOrReplaceTempView("spatialdf")
spatialDf.show()
```

Source code

SRDD API

```
val rangeQueryWindow = new Envelope(-90.01, -80.01, 30.01, 40.01)
val considerBoundaryIntersection = false // Only return gemeotries fully covered by the window
val buildOnSpatialPartitionedRDD = false // Set to TRUE only if run join query
spatialRDD.buildIndex(IndexType.QUADTREE, buildOnSpatialPartitionedRDD)

val usingIndex = true
var queryResult = RangeQuery.SpatialRangeQuery(spatialRDD, rangeQueryWindow,
considerBoundaryIntersection, usingIndex)
```

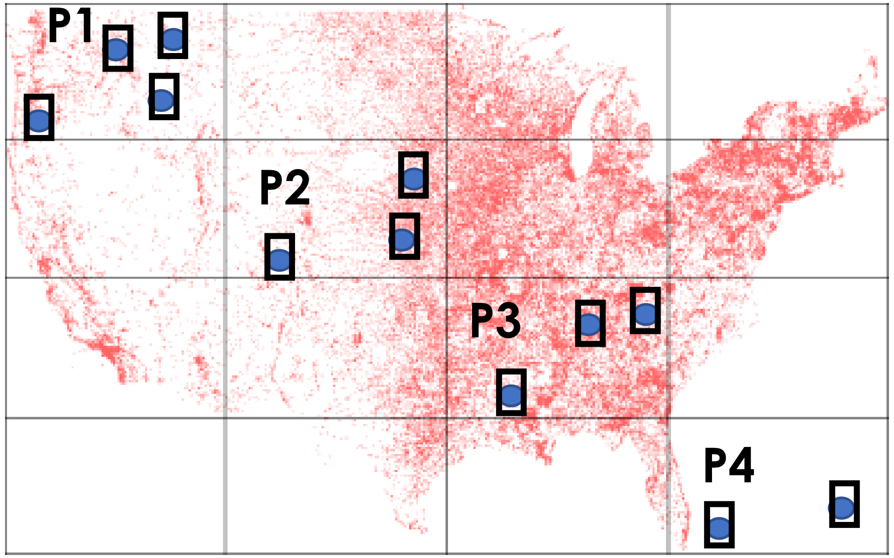
The **output** format → another **SpatialRDD**.

Spatial Join Query algorithm in GeoSpark

- **Partition** data from two input SRDDs and create local spatial indexes
- **Join** the two SRDDs by their **keys** (grid **cell IDs**) → **MBR-join**
- Calculates the spatial relations of candidates (**refine**)

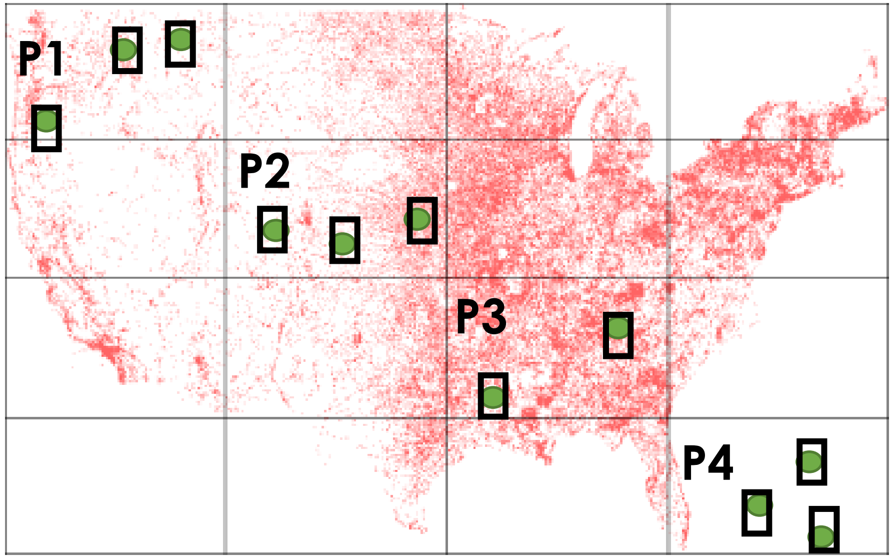
p 1 in A → all taxi **stations** in cell 1

SRDD A (stations)



merge partition 1 from A and B into a bigger partition with two sub-partitions

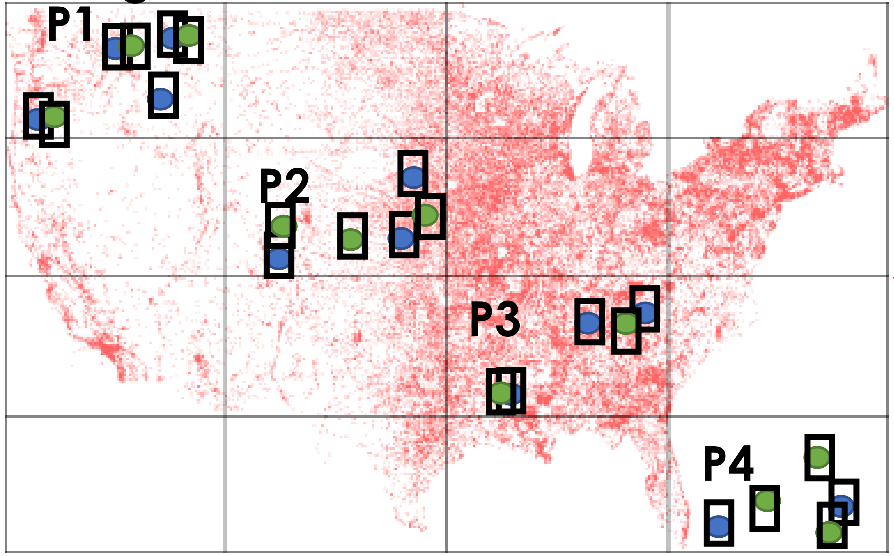
SRDD B (pickup points)



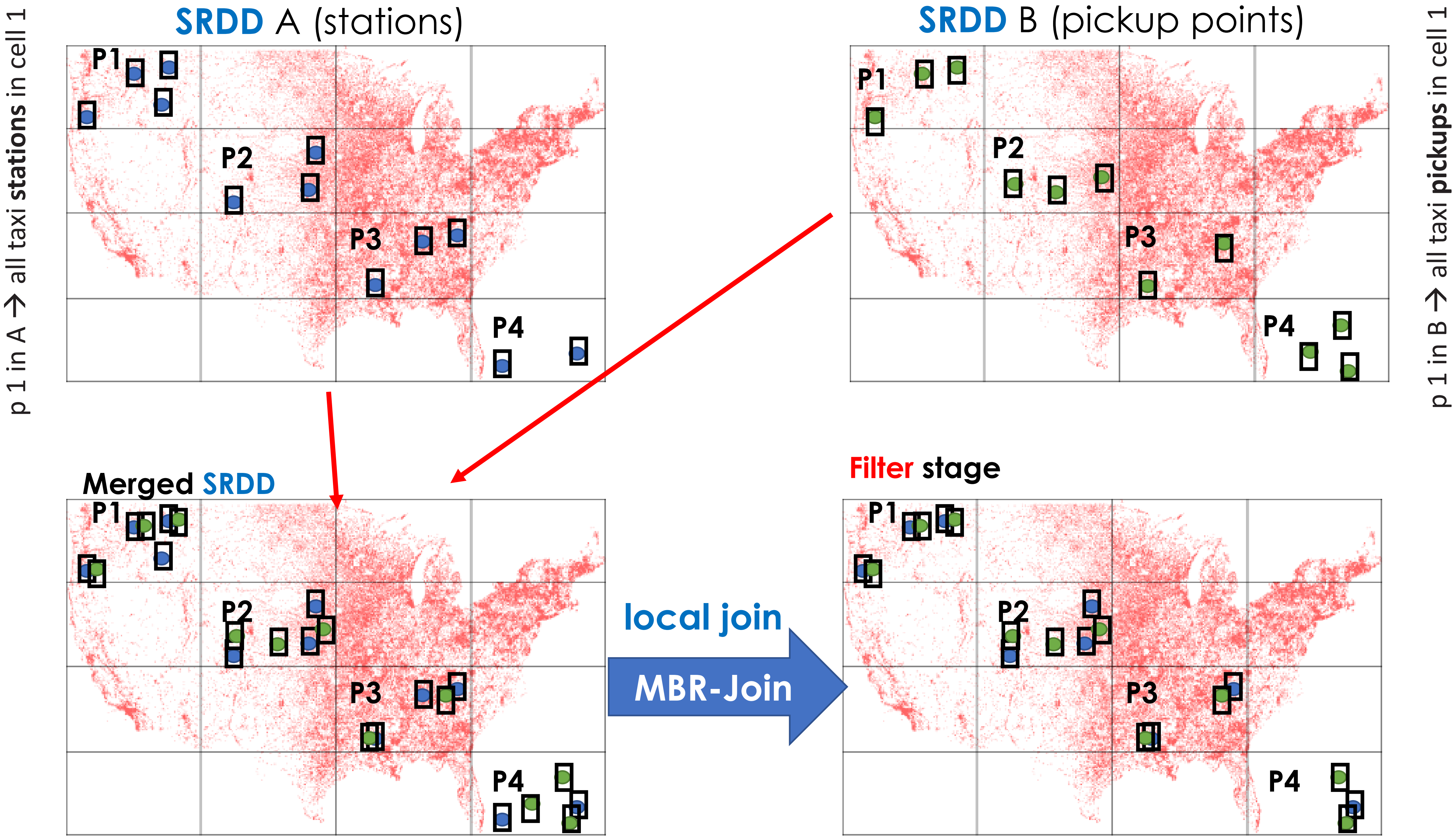
p 1 in B → all taxi **pickups** in cell 1

range join query :The pickup point falls inside the taxi stop station

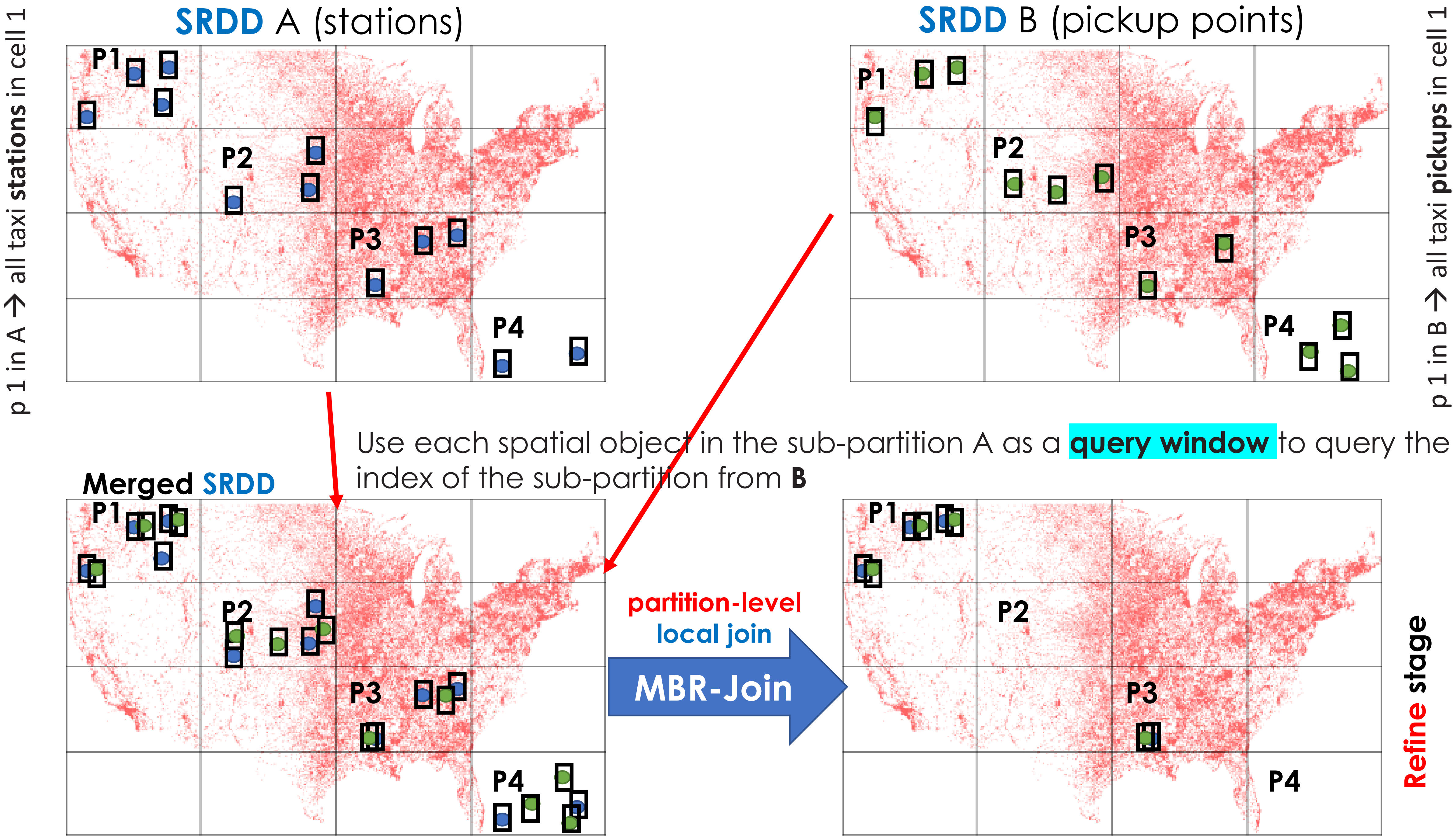
Merged SRDD



the data in Partition1 from A are disjoint from all B's partitions (except 1) → they belong to different grid cells

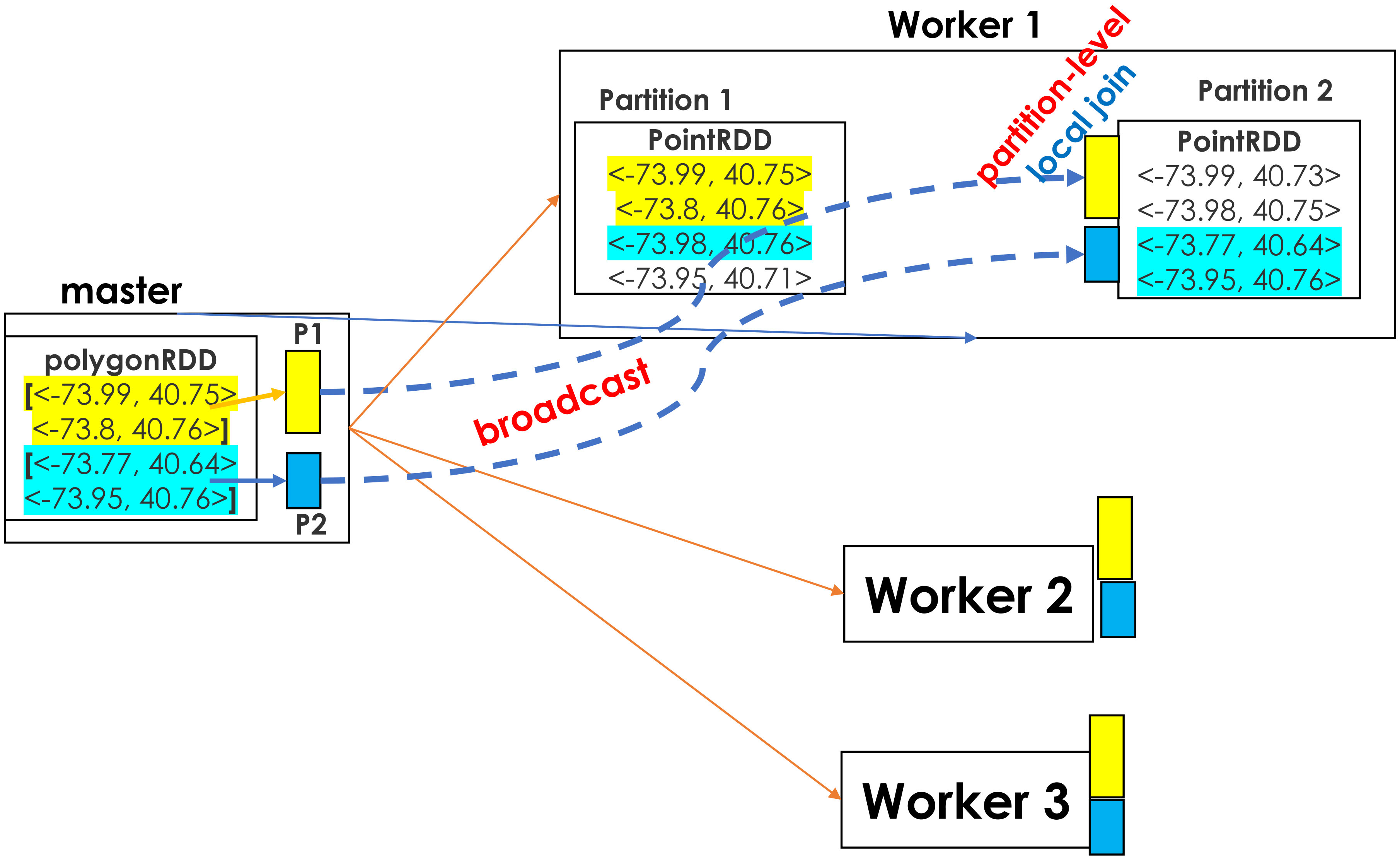


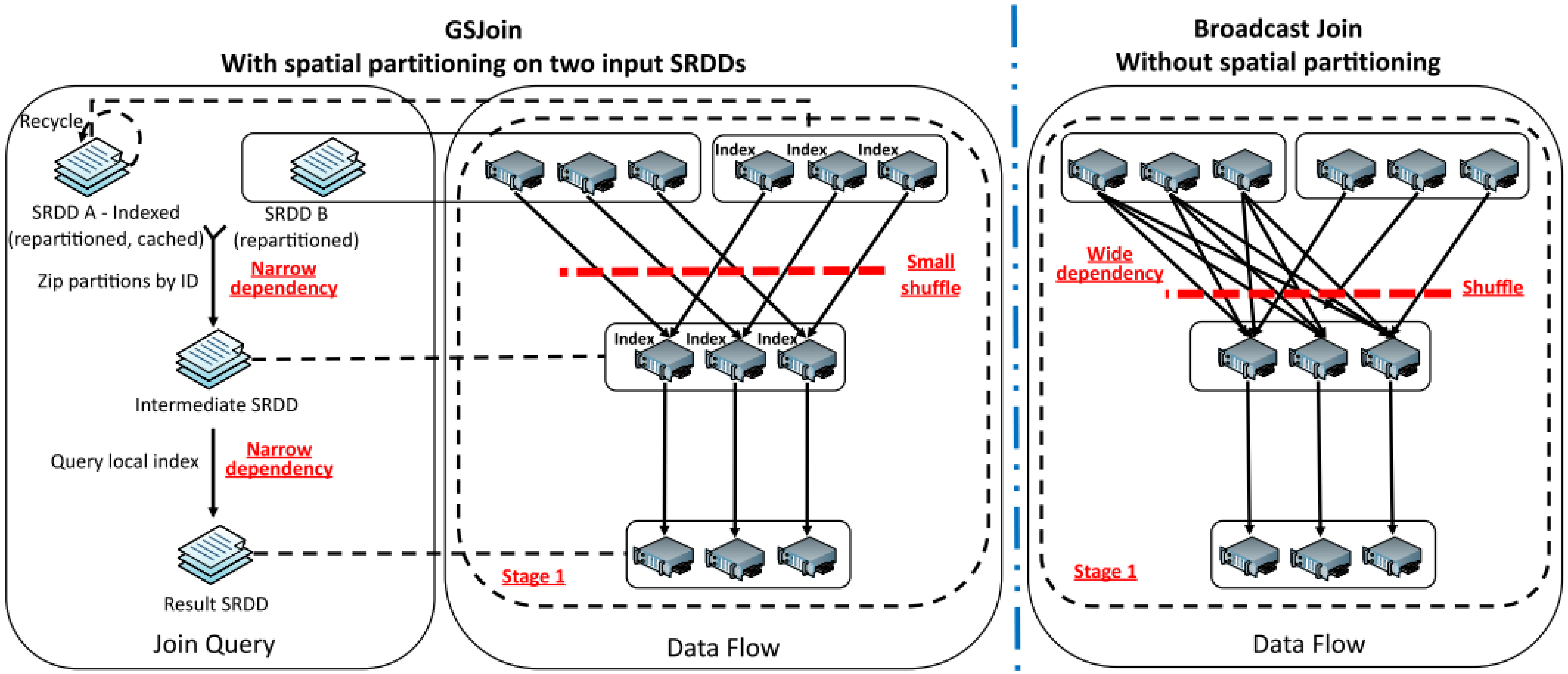
SRDD A (stations)
SRDD B (pickup points)
p 1 in A → all taxi stations in cell 1
p 1 in B → all taxi pickups in cell 1
P1
P2
P3
P4
Merged SRDD
Filter stage
local join
MBR-Join



SRDD A (stations)
SRDD B (pickup points)
p 1 in A → all taxi stations in cell 1
p 1 in B → all taxi pickups in cell 1
P1
P2
P3
P4
Use each spatial object in the sub-partition A as a query window to query the index of the sub-partition from B
Merged SRDD
partition-level
local join
MBR-Join
Refine stage

Broadcast range spatial join in GeoSpark





Join query DAG and data flow

[Image source]()

Range join query examples in GeoSpark

Spatial range join query: Find geometries from A and geometries from B such that each geometry pair satisfies a certain **predicate (contains, intersects)**

Most predicates supported by GeoSpark SQL can trigger a **range join**

All join queries in GeoSparkSQL are **inner joins (**matching values in both tables)

```
SELECT *
FROM polygondf, pointdf
WHERE ST_Contains(polygondf.polygonshape,pointdf.pointshape)
```

```
SELECT *
FROM polygondf, pointdf
WHERE ST_Intersects(polygondf.polygonshape,pointdf.pointshape)
```

Code source

Spatial distance join query in GeoSpark

Spatial **distance join query**: Find geometries from A and geometries from B such that the internal Euclidean distance of each geometry pair is less or equal than a certain distance

```
// fully within a certain distance
SELECT *
FROM pointdf1, pointdf2
WHERE ST_Distance(pointdf1.pointshape1,pointdf2.pointshape2) < 2

// intersects within a certain distance
SELECT *
FROM pointdf1, pointdf2
WHERE ST_Distance(pointdf1.pointshape1,pointdf2.pointshape2) <= 2
```

Code source

Spatial join in RDD terms: GeoSpark

val considerBoundaryIntersection = false // Only return geometries **fully covered** by each **query window** in queryWindowRDD
val **usingIndex** = true
queryWindowRDD.**buildIndex**(IndexType.**QUADTREE**, buildOnSpatialPartitionedRDD)

objectRDD.**spatialPartitioning**(GridType.**KDBTREE**)
queryWindowRDD.**spatialPartitioning**(objectRDD.getPartitioner)

val result = JoinQuery.**SpatialJoinQueryFlat**(**objectRDD**, **queryWindowRDD**, usingIndex, considerBoundaryIntersection)

[Code source]()

Output (PairRDD)

Point,Polygon
Point,Polygon
Point,Polygon
Polygon,Polygon
LineString,LineString
Polygon,LineString
...

left → geometry from **objectRDD**
right → geometry from the **queryWindowRDD**

Spatial KNN Query

- uses a **heap-based top-k** algorithm
  - contains two phases: **selection** and **merging (sorting)**
- It takes a **partitioned SRDD**, a **point** and a number (k) as inputs
- Calculate the **nearest** objects around query point ,
  - in **selection** phase, for each SRDD partition **calculate distances** between every **object** to **query point** ,
  - Maintain a **local heap (local priority queue)** by adding/removing **objects** based on their **distances** in relative to the **query point**
  - This priority queue maintains the nearest spatial objects to query point
  - **merge** results from all partition, keep the nearest K objects that have the shortest distances to the query point

Algorithm 3: K nearest neighbor (KNN) query

```
Data: A query center object A, a Spatial RDD B, the number K
Result: A list of K spatial objects
/* Step 1: Selection phase */
foreach partition in the SRDD B do
    if an index exists then
        Return K nearest neigbors of A by querying the index of this partition;
    else
        foreach object in this partition do
            Check the distance between this object and A;
            Maintain a priority queue that stores the top K nearest neighbors;
/* Step 2: Sorting phase */
Sort the spatial objects in the intermediate Spatial RDD C based on their distances
  to A;
Return the top K objects in C
```

[Algorithm source]()

Spatial KNN query in GeoSpark

Spatial kNN query: 5 nearest neighbor of the given polygon

Code source

```
spatialDf = sparkSession.sql(
  """
    |SELECT countyname, ST_Distance(ST_PolygonFromEnvelope(1.0,100.0,1000.0,1100.0), newcountyshape) AS distance
    |FROM spatialdf
    |ORDER BY distance DESC
    |LIMIT 5
  """.stripMargin)
spatialDf.createOrReplaceTempView("spatialdf")
spatialDf.show()
```

SQL API

```
val geometryFactory = new GeometryFactory()
val pointObject = geometryFactory.createPoint(new Coordinate(-84.01, 34.01))
val K = 1000 // K Nearest Neighbors

val buildOnSpatialPartitionedRDD = false // Set to TRUE only if run join query
objectRDD.buildIndex(IndexType.RTREE, buildOnSpatialPartitionedRDD)

val usingIndex = true
val result = KNNQuery.SpatialKnnQuery(objectRDD, pointObject, K, usingIndex)
```

SRDD API