Magellan: Spark as a Geospatial Analytics Engine

Ram Sriharsha (@halfabrane)
Product Manager, Apache Spark @ Databricks

databricks™

Who Am I?

- Product Manager of Apache Spark @ Databricks
- PMC Member, Committer, Apache Spark
- Prior to Databricks
  - Hortonworks Architect, Spark and Data Science
    - Magellan, Geospatial Analytics on Spark
  - Yahoo Labs, Principal Research Scientist in Scalable Machine Learning
    - Login Risk Detection, Search Advertising Click Prediction, Online Clustering/ Classification.

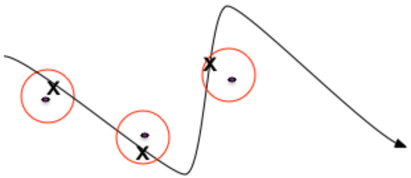
Agenda

- What is Geospatial Analytics?
- The basic operations in Magellan
- Some geometric algorithms used by Magellan
- Internals: How Magellan works with Spark SQL
- Upcoming work: Spatial Indices

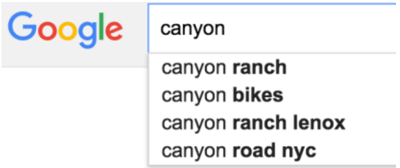
What is geospatial analytics?

UBER

How do pickup/ dropoff neighborhood hotspots evolve with time?



Correct GPS errors with more Accurate landmark measurements



Incorporate location in IR and search advertising

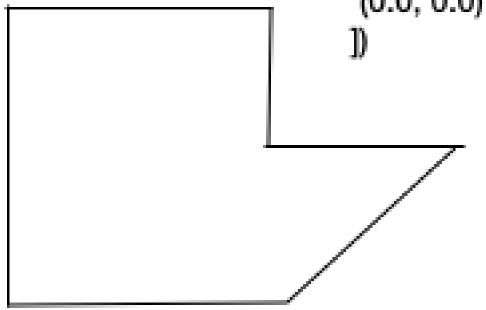
Do we need one more library?

- Spatial Analytics at scale is challenging
  - Single machine libraries not fast enough
  - No scalable implementations exist
- Ancient Data Formats
  - Do not leverage columnar storage, metadata hard to parse and index
  - No spatial indexing
- Geospatial Analytics is not simply BI anymore
  - (approx) Near neighbor queries
  - Map matching

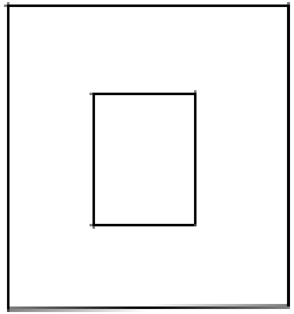
The basic operations

Introduction to Magellan

```
Polygon = (
  [],
  [(0.0, 0.0),(1.0, 0.0),
   (2.0, 1.0),(1.0, 1.0),
   (1.0, 2.0),(0.0, 2.0),
   (0.0, 0.0)
  ])
```



Introduction to Magellan



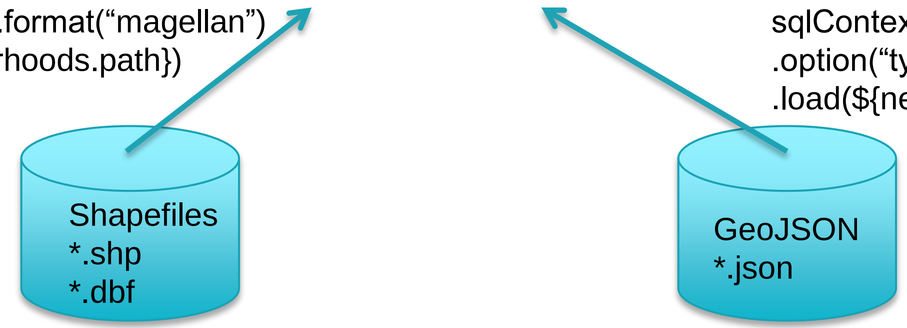
```
Polygon = (
  [0, 5],
  [(0.0, 0.0),(1.0, 0.0),
   (1.0, 2.0),(0.0, 2.0),
   (0.0, 0.0),
   (0.3, 0.3),
   (0.6, 0.3),
   (0.6, 0.9),
   (0.3, 0.9),
   (0.3, 0.3)
  ])
```

Reading from common data formats

polygon	metadata
([0], [(-122.4413024, 7.8066277), …])	neighborhood -> Marina
([0], [(-122.4111659, 37.8003388), …])	neighborhood -> North Beach

```
sqlContext.read.format(“magellan”)
.load(${neighborhoods.path})
```

```
sqlContext.read.format(“magellan”)
.option(“type”, “geojson”)
.load(${neighborhoods.path})
```



Geometric Expressions

polygon	metadata
([0], [(-122.4413024, 7.8066277), ...])	neighborhood -> Marina
([0], [(-122.4111659, 37.8003388), ...])	neighborhood -> North Beach

polygon	metadata
([0], [(-122.4111659, 37.8003388), ...])	neighborhood -> North Beach

```
neighborhoods.filter(
 point(-122.4111659, 37.8003388)
 within
  'polygon
).show()
```

Shape literal

Boolean Expression

Spatial Joins

polygon	metadata
([0], [(-122.4111659, 37.8003388), …])	neighborhood -> North Beach
([0], [(-122.4413024, 7.8066277), …])	neighborhood -> Marina

point
(-122.4111659, 37.8003388)
(-122.4343576, 37.8068007)

points.join(neighborhoods).
  where(‘point within ‘polygon).
  show()

point	polygon	metadata
(-122.4343576, 37.8068007)	([0], [(-122.4111659, 37.8003388), …])	neighborhood -> North Beach

Near neighbor queries

polygon	metadata
([0], [(-122.4111659, 37.8003388), …])	neighborhood -> North Beach
([0], [(-122.4413024, 7.8066277), …])	neighborhood -> Marina

→

point	polygon	metadata
(-122.4343576, 37.8068007)	([0], [(-122.4111659, 37.8003388), …])	neighborhood -> North Beach

```
neighborhoods.filter(
  point(-122.4111659, 37.8003388).buffer(0.1)
  intersects
  'polygon
).show()
```

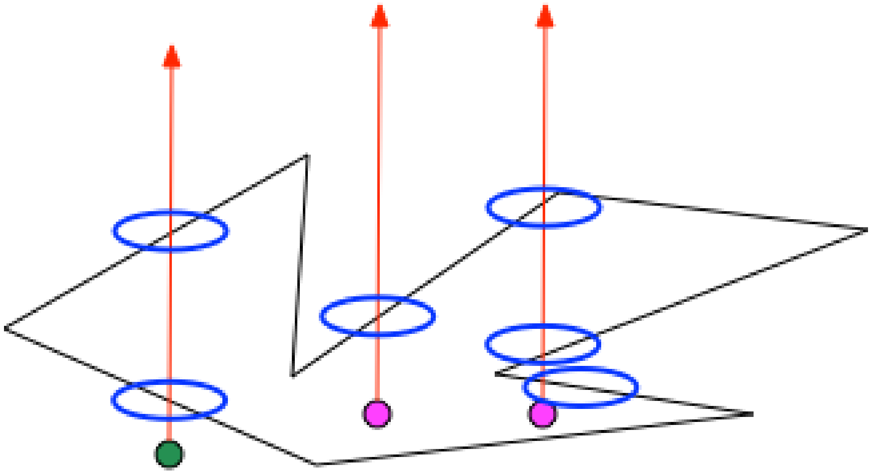
Advantage of embedding geometric queries in Spark SQL

```
val uber = sqlContext.read.format(“csv”).path(${uber.path})

val neighborhoods = sqlContext.read.format("magellan").
                        load(${neighborhoods.path}).
                        select($"polygon", $"metadata"). cache()
Val joined = neighborhoods.
                join(uber).
                where($"point".within($"polygon")).
                select($"tripId", $"timestamp", explode($"metadata").as(Seq("k", "v"))).
                withColumnRenamed("v", "neighborhood”)
```

algorithms

`point within `polygon



databricks

The Join

Inherits all join optimizations from Spark SQL

- if neighborhoods table is small, Broadcast Cartesian Join
- else Cartesian Join

status

databricks™

- Magellan 1.0.3 available as Spark Package.
- Scala , Spark 1.4
- Github: https://github.com/harsha2010/magellan
- Blog: http://hortonworks.com/blog/magellan-geospatial-analytics-in-spark/
- Notebook example: http://bit.ly/1GwLyrV
- Input Formats: ESRI Shapefile, GeoJSON, OSM-XML
- Please try it out and give feedback!

- Magellan 1.0.4 release upcoming in end of June.
- Preview available in Databricks end of May.
- Spark 1.6, 2.0
- Python, Scala
- Tight integration with Tungsten’s memory layout
- Codegen for all operators
- Supports within, contains, intersects, shape literals, near neighbor queries

The internals

Shapes as Data Types

- Points, Polygons, Lines, Polylines are Spark SQL Data Types (UserDefinedType)
- Tungsten Encoding:
  - 8 bit type indicator (1 = point, 3 = polyline, 5 = polygon,…)
  - 16 * 4 bit bounding box (xmin, ymin, xmax, ymax)
  - For point, x and y coordinates = 2 * 16 bits
  - For polygon, indices, coordinates arrays = 8 * # of rings + 16 * # of points * 2
  - ...

databricks

Data Sources

- `SpatialRelation` extends `BaseRelation`, `PrunedFilteredScan`
- `GeoJSONRelation`, `ShapeFileRelation`, `OSMRelation` for GeoJSON, ESRIShapeFile, OSMXML
- Pushes predicates and filters down if possible
- Returns `Row[point, polygon, polyline, meta]`
- Metadata = `Map[String, String]`

`point within `polygon

- Create a case class **Within** that extends **BinaryExpression**
- Override **genCode(ctx: CodeGenContext, ev: GeneratedExpressionCode)** to return generated code
  - Generated code optimizes by bounding box intersections/ contains
  - Takes advantage of Tungsten format
- Make use of ctx to store expensive initialization and reusable objects

- Use implicit conversions to and from **Column**/

Within codegen

```
nullSafeCodeGen(ctx, ev, (c1, c2) => {
        s"" +
        s"Double lxmin = $c1.getDouble(1);" +
        s"Double lymin = $c1.getDouble(2);" +
        s"Double lxmax = $c1.getDouble(3);" +
        s"Double lymax = $c1.getDouble(4);" +
        s"Double rxmin = $c2.getDouble(1);" +
        s"Double rymin = $c2.getDouble(2);" +
        s"Double rxmax = $c2.getDouble(3);" +
        s"Double rymax = $c2.getDouble(4);" +
        s"Boolean within = false;" +
        s"if (rxmin <= lxmin && rymin <= lymin && rxmax >= lxmax && rymax >= lymax) {" +
        s"Integer ltype = $c1.getInt(0);" +
        s"Integer rtype = $c2.getInt(0);" +
        s"magellan.Shape leftShape = (magellan.Shape)" +
          s"((org.apache.spark.sql.types.UserDefinedType<magellan.Shape>)" +
          s"serializers.get(ltype)).deserialize($c1);" +
        s"magellan.Shape rightShape = (magellan.Shape)" +
          s"((org.apache.spark.sql.types.UserDefinedType<magellan.Shape>)" +
          s"serializers.get(rtype)).deserialize($c2);" +
        s"within = rightShape.contains(leftShape);" +
        s"}" +
        s"${ev.value} = within;"
      })
```

The next steps

The join revisited

What is the time complexity?

- m points, n polygons (assume average k edges)
- l partitions
- O(mn/l) computations of ‘point within ‘polygon
- O(ml) communication cost
- Each ‘point within ‘polygon costs O(k)
- Total cost = O(ml) + O(mnk/l)

=> O(m√n√k) cost, with O(√n√k) partitions

Optimization

Do we need to send every point to every partition?

Do we need to compute ‘point in ‘neighborhood for each neighborhood within a given partition?

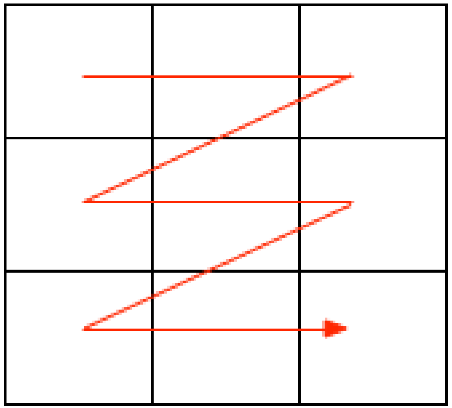
2D Indices

- Quad Trees
- R Trees
- Dimensional Reduction
  - Hashing
  - PCA
  - Space Filling Curves

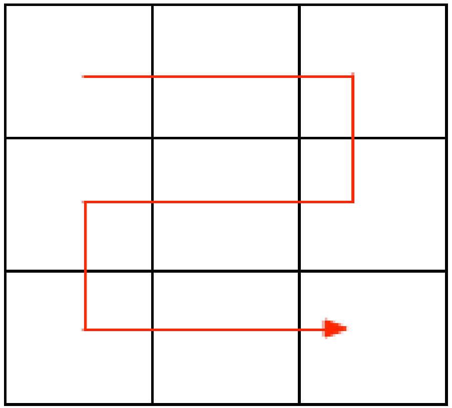
What does a good dimensional reduction need?

- Preserve (approximate) nearness in ambient space
- Enable range queries
- Little/ no collision

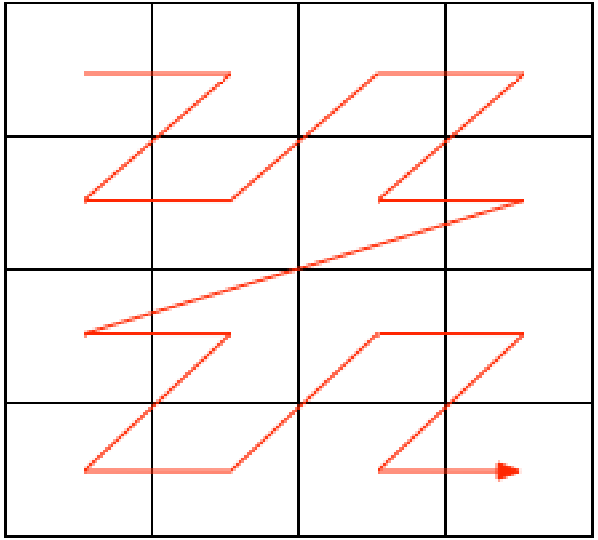
Row Order Curve



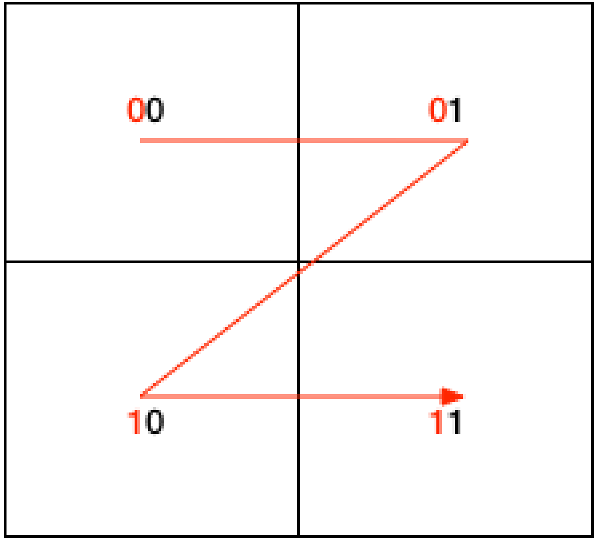
Snake Order Curve



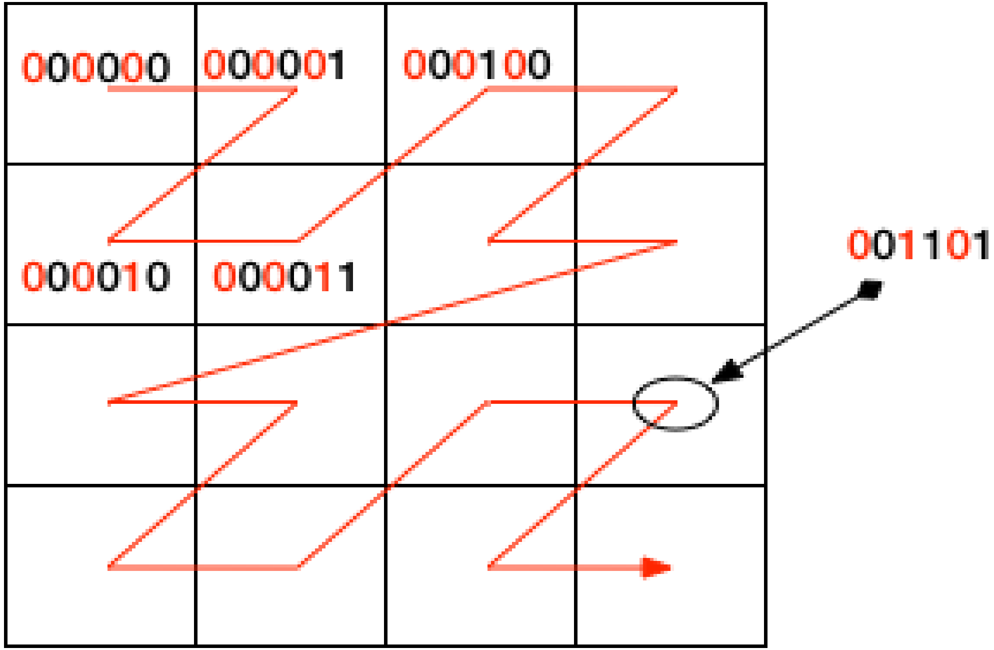
Z Order Curve



Binary Representation



Binary Representation



databricks

Properties

- Locality: two points differing in small # of bits => nearness
  - Converse not necessarily true
- Containment
- Efficient construction
- Nice bounds on precision

If you are familiar with GeoHash

Its nothing but a Z Order Curve!

Start with Bounding Box = (-180, -90, 180, 90) and compute Binary Representation.
Then convert to Base 32 encoded String
You Obtain the GeoHash

How to speed up join?

- Preprocess points:
  - Index each point to a unique geohash
- Preprocess polygons:
  - Index each polygon to a set of geohashes
- Inner join on geohash
- Filter out edge cases

Spark Implementation

- Define **SpatialJoinStrategy** that extends **org.apache.spark.sql.Strategy**
  - Add logic to decide when to trigger this join
    - Only trigger if geospatial queries
    - Only trigger if join is complex: if $n \sim O(1)$ then broadcast join is good enough
- Override **BinaryNode** to handle the physical execution plan ourselves
    - Override execute(): RDD to execute join and return results
- Stitch it up using **ExperimentalStrategies** in **SQLContext**

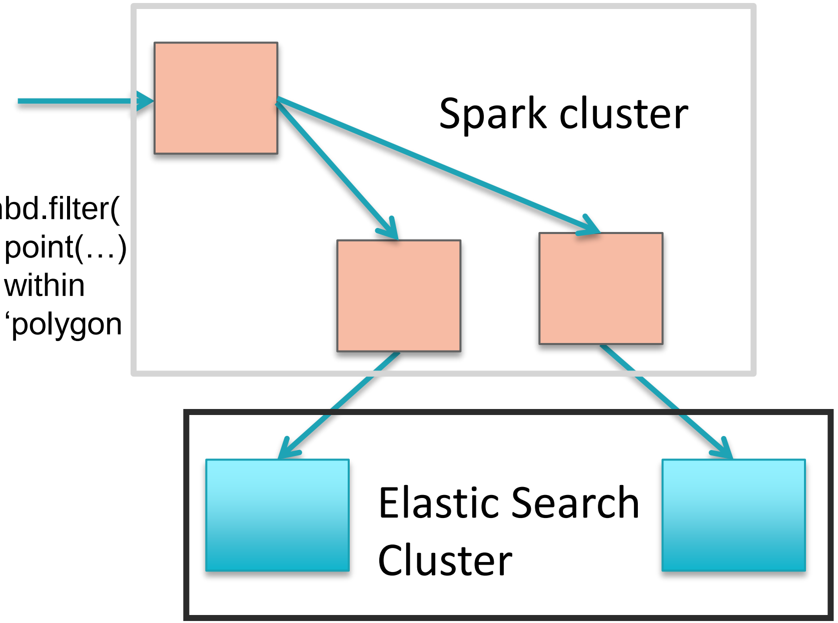
Persistent Indices

Often the geometries do not change (or change slowly)

Can we pre index them?

Overall architecture

```
nbd.filter(
  point(…)
  within
  ‘polygon
)
```




```
curl –XGET ‘http://…’ –d ‘{
  “query” : {
    “filtered” : {
      “filter” : {
        “geohash” : [“dr5rgb”]
      }
    }
  }
}’
```

Contributions to Magellan welcome!

- Algorithms
  - Map Matching
  - Persistent Indices
- Integration with Spark
  - Python API, R API?
  - Encoders
- Data Formats