Optimizing Geospatial Operations with Server-side Programming in HBase and Accumulo

James Hughes, CCRi

CCRi

DATA TO KNOWLEDGE

James Hughes

- CCRi's Director of Open Source Programs
- Working in geospatial software on the JVM for the last 7 years
- GeoMesa core committer / product owner
- SFCurve project lead
- JTS committer
- Contributor to GeoTools and GeoServer

Talk outline

- Background / Warm-up / What we are talking about
  - What is GeoMesa?
  - Quick Demo
- General Implementation Details
  - Indexing on Accumulo/HBase with Space Filling Curves
  - Filtering/transforming
    - Applying secondary filters
    - Changing output (projections / format changes)
  - Aggregations
    - Heatmaps
    - Stats
- Database specifics
  - Accumulo Implementation details
  - HBase Implementation details

Motivation

- What is geospatial?
- IoT based data examples?

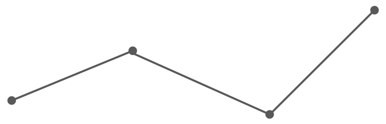
Spatial Data Types

Points

Locations
Events
Instantaneous
Positions

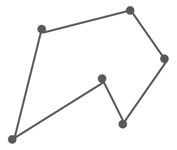
Lines

Road networks
Voyages
Trips
Trajectories



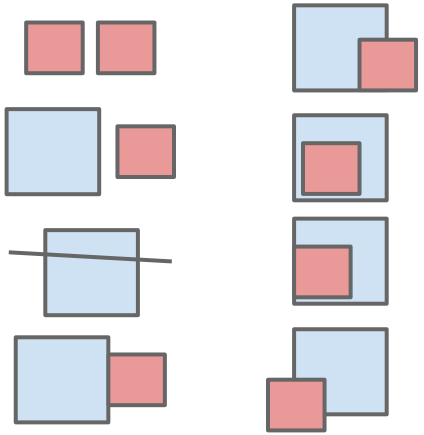
Polygons

Administrative Regions
Airspaces



Spatial Data Relationships

```
equals
disjoint
intersects
touches
```

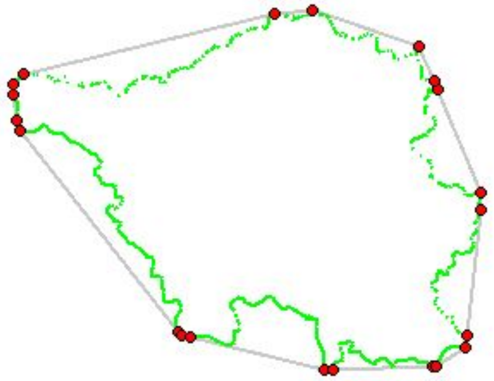


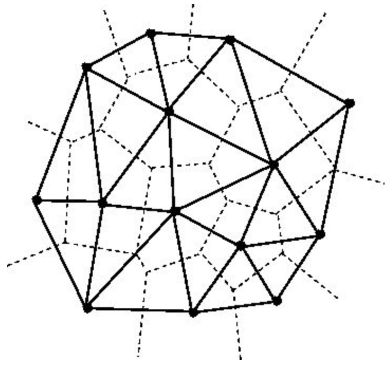
```
crosses
within
contains
overlaps
```

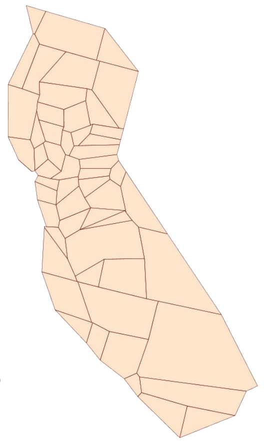
Topology Operations

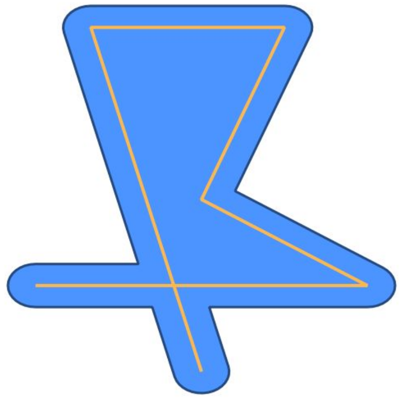
Algorithms

- Convex Hull
- Buffer
- Validation
- Dissolve
- Polygonization
- Simplification
- Triangulation
- Voronoi
- Linear Referencing
- and more...









GeoMesa

- GeoMesa Overview

What is GeoMesa?

A suite of tools for streaming, persisting, managing, and analyzing spatio-temporal data at scale

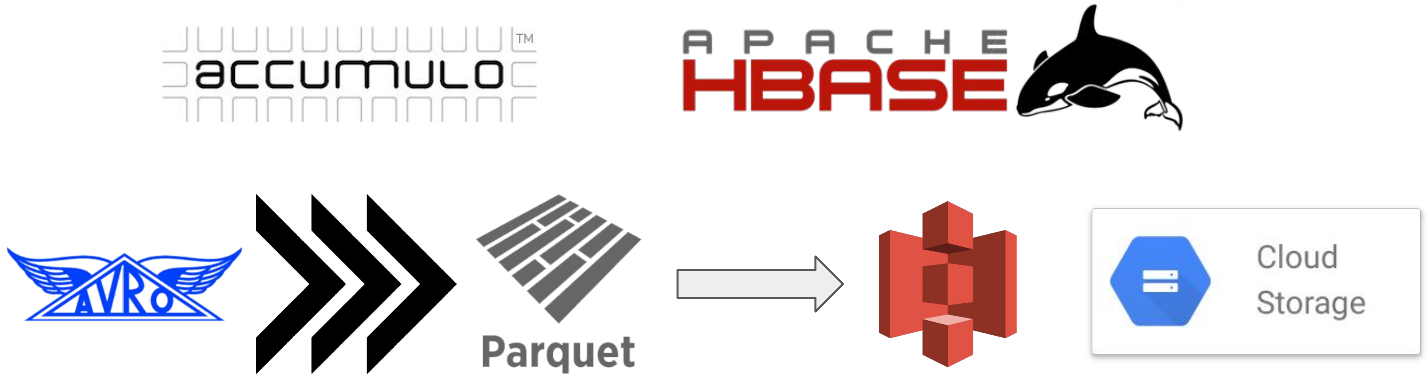
What is GeoMesa?

A suite of tools for **streaming**, persisting, managing, and analyzing spatio-temporal data at scale

CCRi DATA TO KNOWLEDGE

What is GeoMesa?

A suite of tools for streaming, **persisting**, managing, and analyzing spatio-temporal data at scale



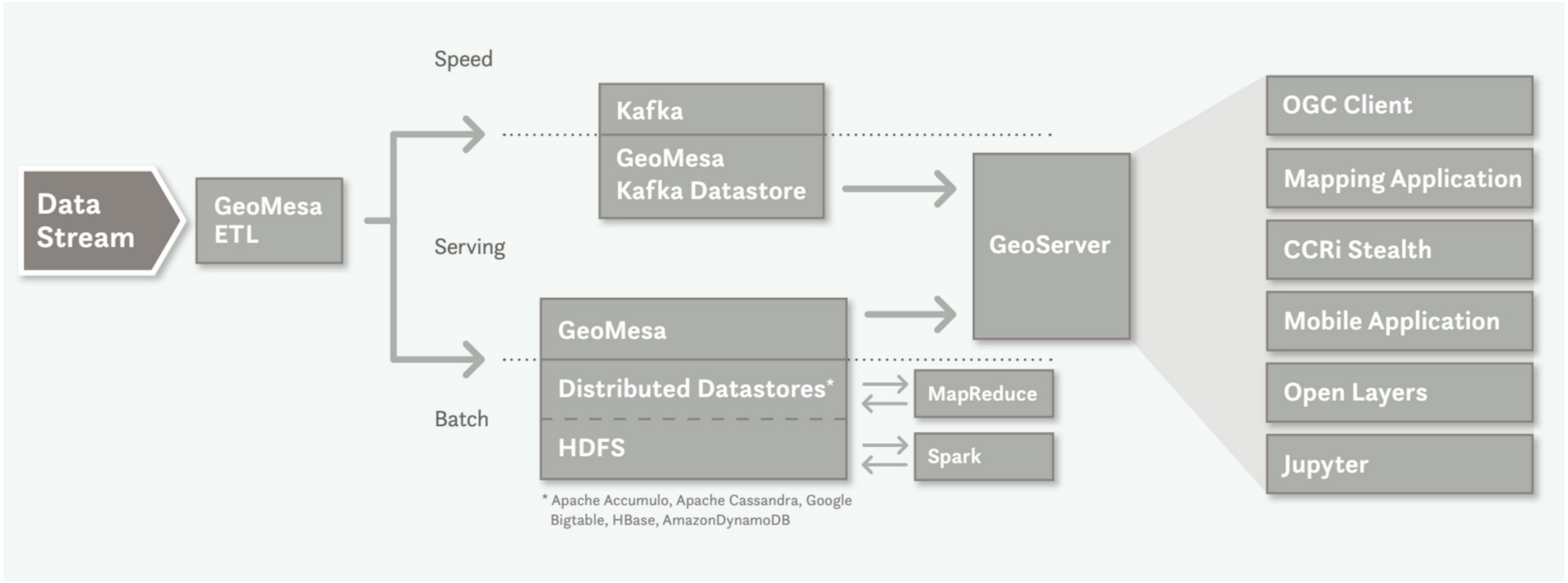
What is GeoMesa?

A suite of tools for streaming, persisting, **managing**, and analyzing spatio-temporal data at scale

What is GeoMesa?

A suite of tools for streaming, persisting, managing, and **analyzing** spatio-temporal data at scale

Proposed Reference Architecture



* Apache Accumulo, Apache Cassandra, Google Bigtable, HBase, AmazonDynamoDB

Live Demo!

- Filtering by spatio-temporal constraints
- Filtering by attributes
- Aggregations
- Transformations

Indexing Geospatial Data

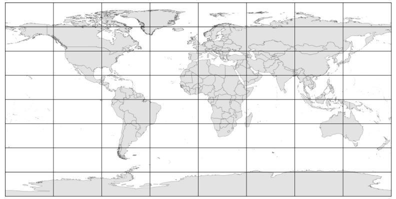
- Key Design using Space Filling Curves

Space Filling Curves (in one slide!)

- **Goal: Index 2+ dimensional data**
- **Approach: Use Space Filling Curves**

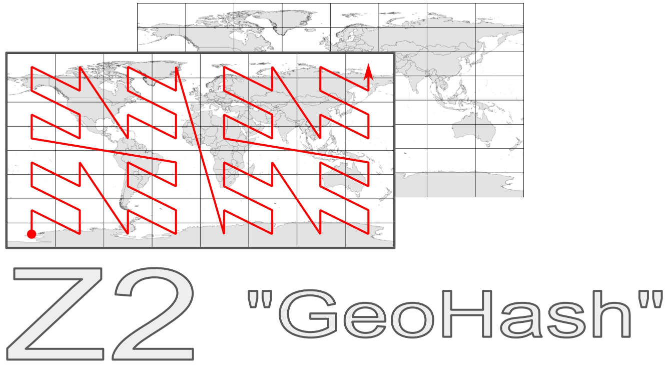
Space Filling Curves (in one slide!)

- **Goal: Index 2+ dimensional data**
- **Approach: Use Space Filling Curves**
- First, ‘grid’ the data space into bins.



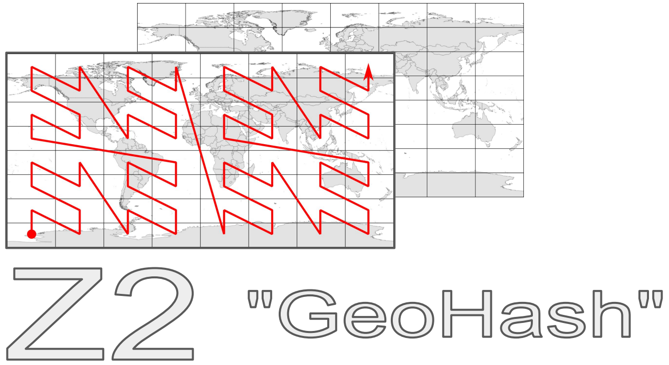
Space Filling Curves (in one slide!)

- **Goal: Index 2+ dimensional data**
- **Approach: Use Space Filling Curves**
- First, ‘grid’ the data space into bins.
- Next, order the grid cells with a space filling curve.
    - Label the grid cells by the order that the curve visits the them.
    - Associate the data in that grid cell with a byte representation of the label.



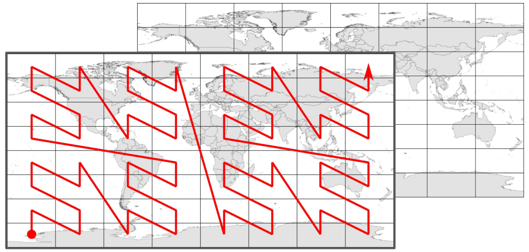
Space Filling Curves (in one slide!)

- **Goal: Index 2+ dimensional data**
- **Approach: Use Space Filling Curves**
- First, ‘grid’ the data space into bins.
- Next, order the grid cells with a space filling curve.
  - Label the grid cells by the order that the curve visits the them.
  - Associate the data in that grid cell with a byte representation of the label.
- We prefer “good” space filling curves:
  - Want recursive curves and locality.



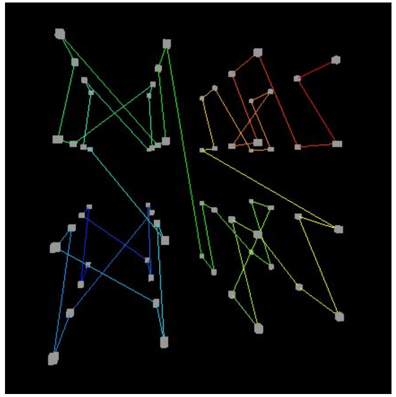
Space Filling Curves (in one slide!)

- **Goal: Index 2+ dimensional data**
- **Approach: Use Space Filling Curves**
- First, ‘grid’ the data space into bins.
- Next, order the grid cells with a space filling curve.
  - Label the grid cells by the order that the curve visits the them.
  - Associate the data in that grid cell with a byte representation of the label.
- We prefer “good” space filling curves:
  - Want recursive curves and locality.
- Space filling curves have higher dimensional analogs.



Z2 "GeoHash"

Z3

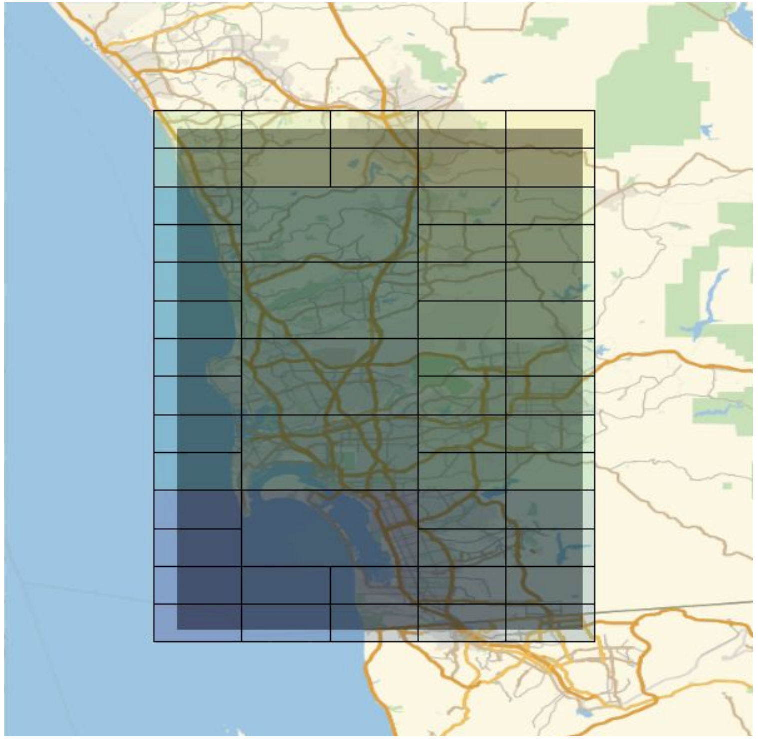


Query planning with Space Filling Curves

To query for points in the grey rectangle, the query planner enumerates a collection of index ranges which cover the area.

Note: Most queries won't line up perfectly with the gridding strategy.

Further filtering can be run on the tablet/region servers (next section)
or we can return 'loose' bounding box results (likely more quickly).



Server-Side Optimizations

Filtering and transforming records

- Pushing down data filters
  - Z2/Z3 filter
  - CQL Filters
- Projections

Filtering and transforming records overview

Using Accumulo iterators and HBase filters, it is possible to ***filter*** and ***map*** over the key-values pairs scanned.

This will let us apply fine-grained spatial filtering, filter by secondary predicates, and implement projections.

Pushing down filters

Let’s consider a query for **tankers** which are inside a bounding box for a given time period.

GeoMesa’s Z3 index is designed to provide a set of **key ranges** to scan which will cover the spatio-temporal range.

Additional information such as the **vessel type** is part of the value.

Using server-side programming, we can teach Accumulo and HBase how to understand the records and filter out undesirable records.

This **reduces network traffic** and **distributes** the work.

Projection

To handle projections in a query, Accumulo Iterators and HBase Filters can change the returned key-value pairs.

Changing the key is a bad idea.

Changing the value allows for GeoMesa to return a subset of the columns that a user is requesting.

GeoMesa Server-Side Filters

- Z2/Z3 filter
    - Scan ranges are not decomposed enough to be very accurate - fast bit-wise comparisons on the row key to filter out-of-bounds data
- CQL/Transform filter
    - If a predicate is not handled by the scan ranges or Z filters,
      then slower GeoTools CQL filters are applied to the serialized SimpleFeature in the row value
    - Relational projections (transforms) applied to reduce the amount of data sent back
- Other specialized filters
    - Age-off for expiring rows based on a SimpleFeature attribute
    - Attribute-key-value for populating a partial SimpleFeature with an attribute value from the row
    - Visibility filter for merging columns into a SimpleFeature when using attribute-level visibilities

Server-Side Optimizations

Aggregations

- Generating heatmaps
- Descriptive Stats
- Arrow format

Aggregations

Using Accumulo Iterators and HBase coprocessors, it is possible to aggregate multiple key-value pairs into one response. Effectively, this lets one implement ***map*** and ***reduce*** algorithms.

These aggregations include computing ***heatmaps***, ***stats***, and ***custom data formats***.

The ability to aggregate data can be composed with filtering and projections.

GeoMesa Aggregation Abstractions

Aggregation logic is implemented in a shared module, based on a lifecycle of

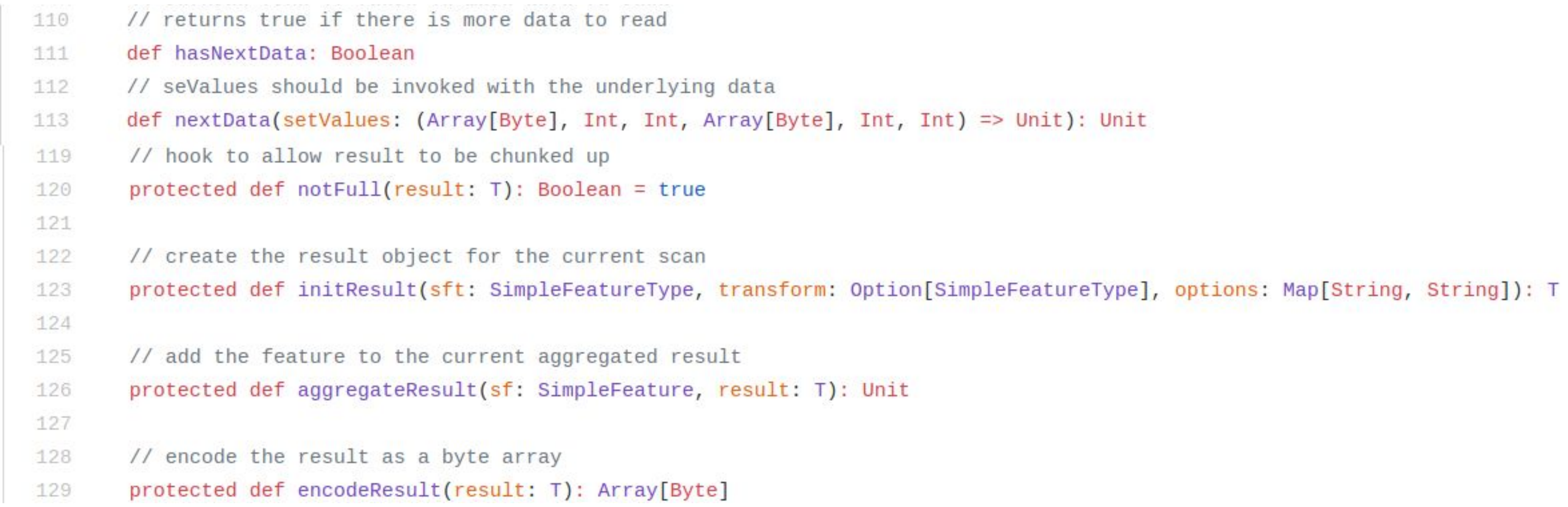
1. **Initialization**
2. **observing** some number of features
3. **aggregating** a result.

This paradigm is easily adapted to the specific implementations required by Accumulo and HBase.

Notably, all the algorithms we describe work in a single pass over the data.

GeoMesa Aggregation Abstractions

The main logic is contained in the AggregatingScan class:

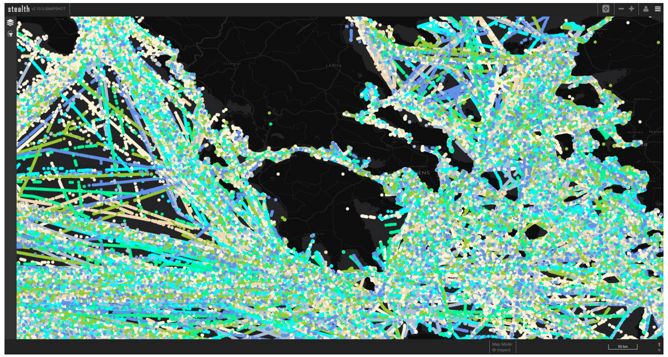


```
  // returns true if there is more data to read
  def hasNextData: Boolean
  // seValues should be invoked with the underlying data
  def nextData(setValues: (Array[Byte], Int, Int, Array[Byte], Int, Int) => Unit): Unit
  // hook to allow result to be chunked up
  protected def notFull(result: T): Boolean = true

  // create the result object for the current scan
  protected def initResult(sft: SimpleFeatureType, transform: Option[SimpleFeatureType], options: Map[String, String]): T

  // add the feature to the current aggregated result
  protected def aggregateResult(sf: SimpleFeature, result: T): Unit

  // encode the result as a byte array
  protected def encodeResult(result: T): Array[Byte]
```

Visualization Example: Heatmaps

Without powerful visualization options, big data is big nonsense.

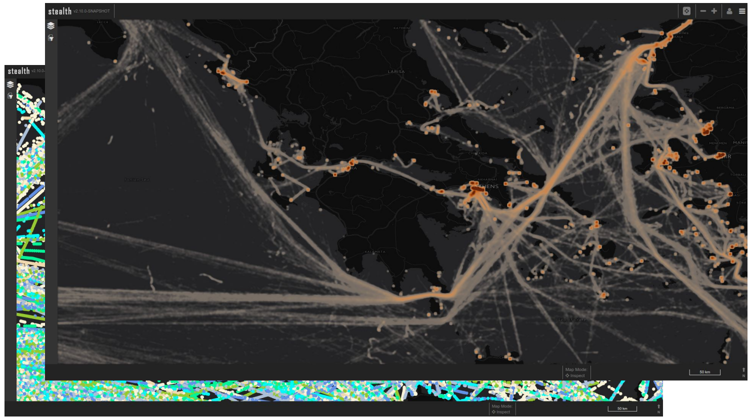
Consider this view of shipping in the Mediterranean sea



Visualization Example: Heatmaps

Without powerful visualization options, big data is big nonsense.

Consider this view of shipping in the Mediterranean sea



Generating Heatmaps

Heatmaps are implemented in DensityScan.

For the scan, we set up a 2D grid array representing the pixels to be displayed. On the region/tablet servers, each feature increments the count of any cells intersecting its geometry. The resulting grid is returned as a serialized array of 64-bit integers, minimizing the data transfer back to the client.

The client process merges the grids from each scan range, then normalizes the data to produce an image.

Since less data is transmitted, **heatmaps are generally faster**.

Statistical Queries

We support a flexible stats API that includes **counts**, **min/max** values, **enumerations**, **top-k** (StreamSummary), **frequency** (CountMinSketch), **histograms** and **descriptive statistics**. We use well-known streaming algorithms backed by data structures that can be serialized and merged together.

Statistical queries are implemented in StatsScan.

On the region/tablet servers, we set up the data structure and then add each feature as we scan. The client receives the serialized stats, merges them together, and displays them as either JSON or a Stat instance that can be accessed programmatically.

Arrow Format

Apache Arrow is a columnar, in-memory data format that GeoMesa supports as an output type. In particular, it can be used to drive complex in-browser visualizations. Arrow scans are implemented in ArrowScan.

With Arrow, the data returned from the region/tablet servers is similar in size to a normal query. However, the processing required to generate Arrow files can be distributed across the cluster instead of being done in the client.

As we scan, each feature is added to an in-memory Arrow vector. When we hit the configured batch size, the current vector is serialized into the Arrow IPC format and sent back to the client. All the client needs to do is to create a header and then concatenate the batches into a single response.

Server-Side Optimizations

Data

- Row Values
- Tables/compactions

Row Values

Our first approach was to store each SimpleFeature attribute in a separate column. However, this proved to be slow to scan.

Even when skipping columns for projections, they are still loaded off disk.

Column groups seem promising, but they kill performance if you query more than one.

Row Values

Our second (and current) approach is to store the entire serialized SimpleFeature in one column.

Further optimizations:

- Lazy deserialization - SimpleFeature implementation that wraps the row value and only deserializes each attribute as needed
- Feature ID is already stored in the row key to prevent row collisions, so don’t also store it in the row value
- Use BSON for JSON serialization, along with JsonPath extractors
- Support for TWKB geometry serialization to save space

Tables/Compactions

When dealing with streaming data sources, continuously writing data to a table will cause a lot of compactions.

Table partitioning can mitigate this by creating a new table per time period (e.g. day/week), extracted from the SimpleFeature. Generally only the most recent table(s) will be compacted.

For frequent updates to existing features, the GeoMesa Lambda store uses Kafka as a medium-term cache before persisting to the key-value store. This reduces the cluster load significantly.

Accumulo Server Side Programming

- Accumulo Iterator Review
- GeoMesa’s Accumulo iteraors

Accumulo Iterators

“**Iterators** provide a modular mechanism for adding functionality to be executed by **TabletServers** when scanning or compacting data. This allows users to efficiently summarize, filter, and aggregate data.” -- Accumulo 1.7 documentation

Part of the modularity is that the iterators can be stacked:
the output of one can be wired into the next.

Example: The first iterator might read from disk, the second could filter with Authorizations, and a final iterator could filter by column family.

Other notes:

- Iterators provided a sorted view of the key/values.
- Iterator code can be loaded from HDFS and namespaced!

GeoMesa Data Requests

A request to GeoMesa consists of two broad pieces:

1. A filter restricting the data to act on, e.g.:
   a. Records in Maryland with ‘Accumulo’ in the text field.
   b. Records during the first week of 2016.
2. A request for ‘how’ to return the data, e.g.:
   a. Return the full records
   b. Return a subset of the record (either a projection or ‘bin’ file format)
   c. Return a histogram
   d. Return a heatmap / kernel density

Generally, a filter can be handled partially by selecting which ranges to scan; the remainder can be handled by an Iterator.

Modifications to selected data can also be handled by a GeoMesa Iterator.

Initial GeoMesa Iterator design

The first pass of GeoMesa iterators separated concerns into separate iterators.

The GeoMesa query planner assembled a stack of iterators to achieve the desired result.

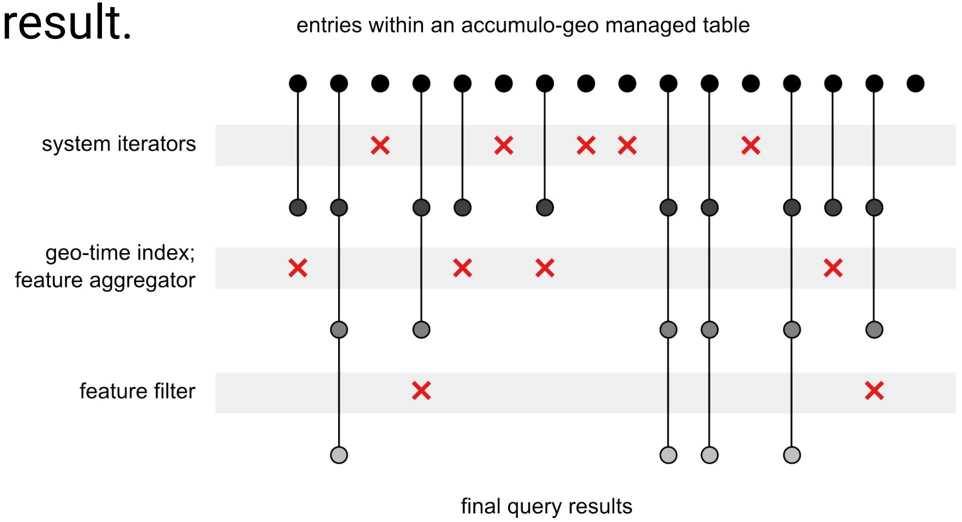


Image from “Spatio-temporal Indexing in Non-relational Distributed Databases” by Anthony Fox, Chris Eichelberger, James Hughes, Skylar Lyon

Second GeoMesa Iterator design

The key benefit to having decomposed iterators is that they are easier to understand and re-mix.

In terms of performance, each one needs to understand the bytes in the Key and Value. In many cases, this will lead to additional serialization/deserialization.

Now, we prefer to write Iterators which handle transforming the underlying data into what the client code is expecting in one go.

Lessons learned about Iterators

1. Using fewer iterators in the stack can be beneficial
2. Using lazy evaluation / deserialization for filtering Values can power speed improvements.
3. Iterators take in Sorted Keys + Values and *must* produce Sorted Keys and Values.

HBase Server Side Programming

- HBase Filter and Coprocessor Review
- GeoMesa HBase Filter
- GeoMesa HBase Coprocessor

HBase Filter Info

HBase filters are restricted to the ability to skip/include rows, and to transform a row before returning it. Anything more complicated requires a Coprocessor.

In contrast to Accumulo, where iterators are configured with a map of options, HBase requires custom serialization code for each filter implementation.

HBase Filter Info

The main GeoMesa filters are:

- org.locationtech.geomesa.hbase.filters.CqlTransformFilter
  - Filters rows based on GeoTools CQL
  - Transforms rows based on relational projections
- org.locationtech.geomesa.hbase.filters.Z2HBaseFilter
  - Compares Z-values against the row key
- org.locationtech.geomesa.hbase.filters.Z3HBaseFilter
  - Compares Z-values against the row key

HBase Coprocessor Info

Coprocessors are not trivial to implement or invoke, and can starve your cluster if it is not configured correctly.

GeoMesa implements a harness to invoke a coprocessor, receive the results, and handle any errors:

- org.locationtech.geomesa.hbase.coprocessor.GeoMesaCoprocessor

An adapter layer links the common aggregating code to the coprocessor API:

- org.locationtech.geomesa.hbase.coprocessor.aggregators.HBaseAggregator

HBase Coprocessor Info

GeoMesa defines a single Protobuf coprocessor endpoint, modeled around the Accumulo iterator lifecycle. The aggregator class and a map of options are passed to the endpoint.

Each aggregating scan requires a trivial adapter implementation:

- HBaseDensityAggregator
- HBaseStatsAggregator
- HBaseArrowAggregator

Thanks!

James Hughes

- jhughes@ccri.com
- http://geomesa.org
- http://gitter.im/locationtech/geomesa

Backup Slides

Integration with MapReduce / Spark

- GeoMesa + Spark Setup
- GeoMesa + Spark Analytics
- GeoMesa powered notebooks (Jupyter and Zeppelin)

GeoMesa MapReduce and Spark Support

Using Accumulo Iterators, we've seen how one can easily perform simple 'MapReduce' style jobs without needing more infrastructure.

NB: Those tasks are limited. One can filter inputs, transform/map records and aggregate partial results on each tablet server.

To implement more complex processes, we look to MapReduce and Spark.

GeoMesa MapReduce and Spark Support

Using Accumulo Iterators, we've seen how one can easily perform simple 'MapReduce' style jobs without needing more infrastructure.

NB: Those tasks are limited. One can filter inputs, transform/map records and aggregate partial results on each tablet server.

To implement more complex processes, we look to MapReduce and Spark.

Accumulo Implements the MapReduce InputFormat interface.

GeoMesa MapReduce and Spark Support

Using Accumulo Iterators, we've seen how one can easily perform simple 'MapReduce' style jobs without needing more infrastructure.

NB: Those tasks are limited. One can filter inputs, transform/map records and aggregate partial results on each tablet server.

To implement more complex processes, we look to MapReduce and Spark.

Accumulo Implements the MapReduce InputFormat interface.

Spark provides a way to change InputFormats into RDDs.

GeoMesa MapReduce and Spark Support

Using Accumulo Iterators, we've seen how one can easily perform simple 'MapReduce' style jobs without needing more infrastructure.

NB: Those tasks are limited. One can filter inputs, transform/map records and aggregate partial results on each tablet server.

To implement more complex processes, we look to MapReduce and Spark.

Accumulo Implements the MapReduce InputFormat interface.

Spark provides a way to change InputFormats into RDDs.

Winning!

GeoMesa Spark Example 1: Time Series

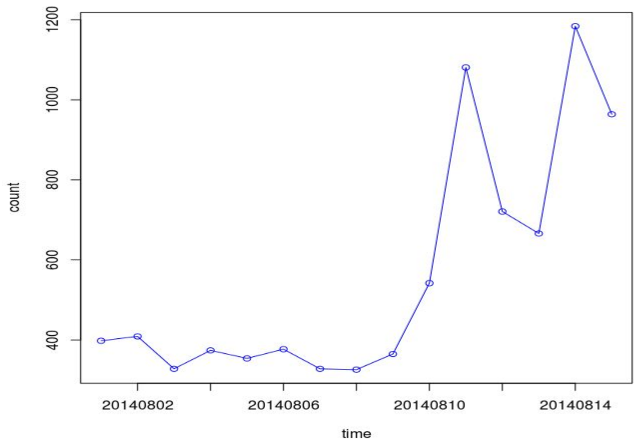
Step 1: Get an RDD[SimpleFeature]

```
// Get a handle to the data store
val params = Map(
  "instanceId" -> "myinstance",
  "zookeepers" -> "zoo1,zoo2,zoo3",
  "user"       -> "username",
  "password"   -> "password",
  "tableName"  -> "geomesa_catalog")

val ds = DataStoreFinder.getDataStore(params).asInstanceOf[AccumuloDataStore]

// Construct a CQL query to filter by bounding box
val ff = CommonFactoryFinder.getFilterFactory2
val f = ff.bbox("geom",  -90.32023,38.72009,-90.23957,38.77019, "EPSG:4326")
val q = new Query(feature, f)
val conf = new Configuration
val sconf = init(new SparkConf(true), ds)
val sc = new SparkContext(sconf)

val queryRDD = geomesa.compute.spark.GeoMesaSpark.rdd(conf, sconf, ds, query)
```

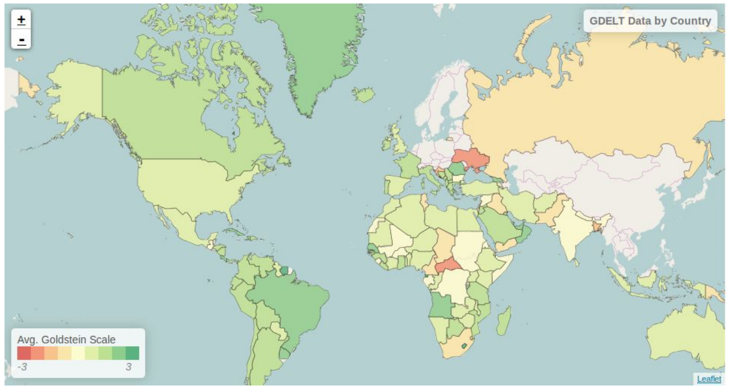
Step 2: Calculate the time series

```
// Convert RDD[SimpleFeature] to RDD[(String, SimpleFeature)] where the first
// element of the tuple is the date to the day resolution
val dayAndFeature = queryRDD.mapPartitions { iter =>
  val df = new SimpleDateFormat("yyyyMMdd")
  val ff = CommonFactoryFinder.getFilterFactory2
  val exp = ff.property("dtg")
  iter.map { f => (df.format(exp.evaluate(f).asInstanceOf[java.util.Date]), f) }
}

// Aggregate and output
val groupedByDay = dayAndFeature.groupBy { case (date, _) => date }
val countByDay = groupedByDay.map { case (date, iter) => (date, iter.size) }
countByDay.collect.foreach(println)
```

Step 3: Plot the time series in R.



GeoMesa Spark Example 2: Aggregating by Regions



Using one dataset (country boundaries) to group another (here, GDELT) is effectively a join.

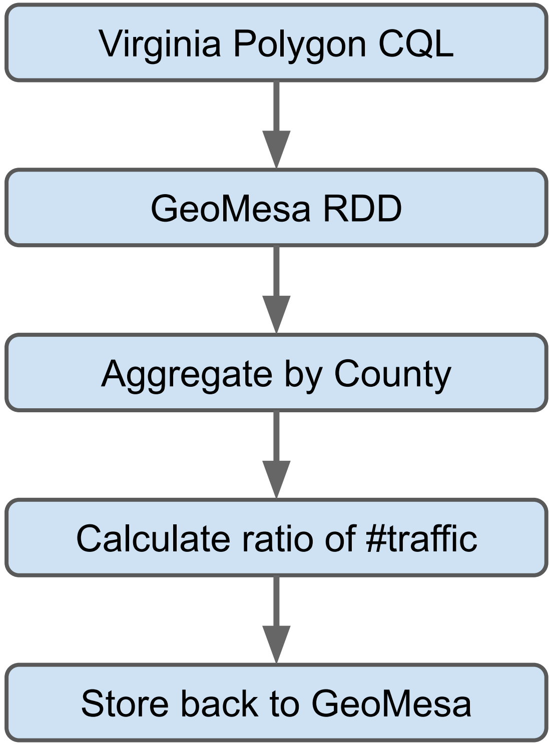
Our summer intern, Atallah, worked out the details of doing this analysis in Spark and created a tutorial and blog post.

This picture shows 'stability' of a region from GDELT Goldstein values

http://www.ccri.com/2016/08/17/new-geomesa-tutorial-aggregating-visualizing-data/
http://www.geomesa.org/documentation/tutorials/shallow-join.html

GeoMesa Spark Example 3: Aggregating Tweets about #traffic

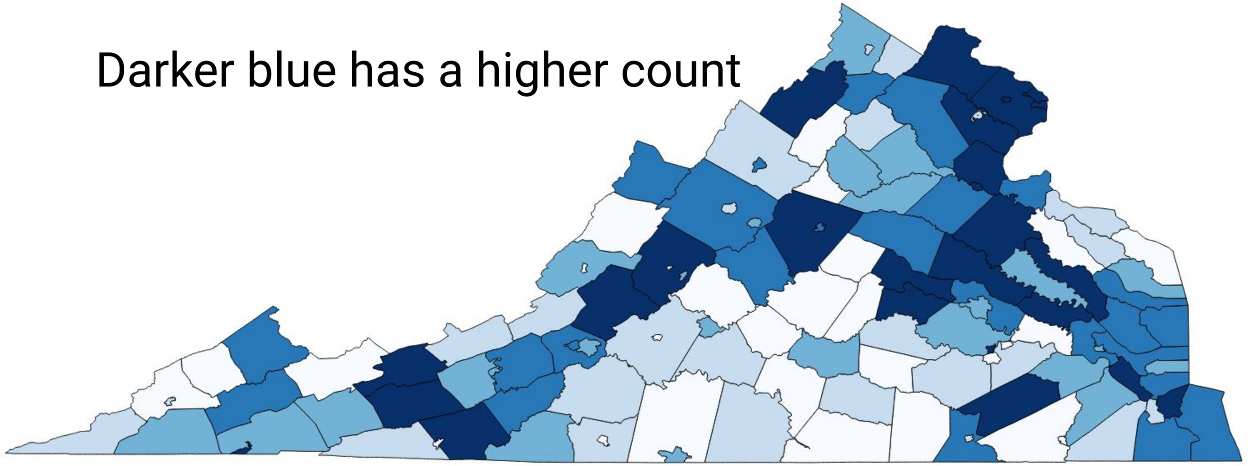
```scala
// Create a list of Counties and iterator of
// booleans for if tweets are traffic related
val countyAndTweets = geoMesaRDD.mapPartitions { iter =>
  iter.flatMap { sf =>
    getCounty(sf).map { county =>
      (county, isTrafficRelated(sf))
    }
  }
}

// Group by Counties and calculate percentage of
// tweets that are True (traffic related)
countyAndTweets.groupBy(_._1)
.map { case (county, iter) =>
  val pctTraffic = iter.count(_._2) / iter.size
  (count, pctTraffic)
 }
.map { case (county, pct) => (c, getGeom(county), pct) }
.saveLayer()
```



GeoMesa Spark Example 3: Aggregating Tweets about #traffic

#traffic by Virginia county

Darker blue has a higher count



Interactive Data Discovery at Scale in GeoMesa Notebooks

Writing (and debugging!) MapReduce / Spark jobs is slow and requires expertise.

A long development cycle for an analytic saps energy and creativity.

The answer to both is interactive ‘notebook’ servers like Apache Zeppelin and Jupyter (formerly iPython Notebook).

Interactive Data Discovery at Scale in GeoMesa Notebooks

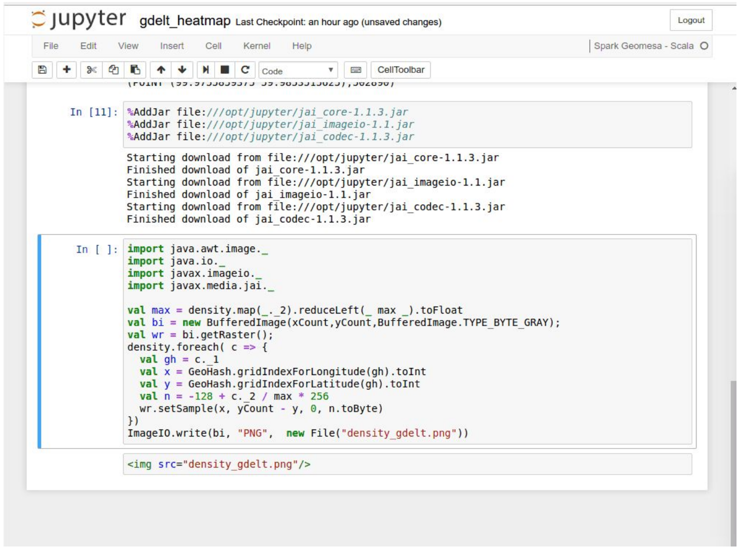
Writing (and debugging!) MapReduce / Spark jobs is slow and requires expertise.

A long development cycle for an analytic saps energy and creativity.

The answer to both is interactive 'notebook' servers like Apache Zeppelin and Jupyter

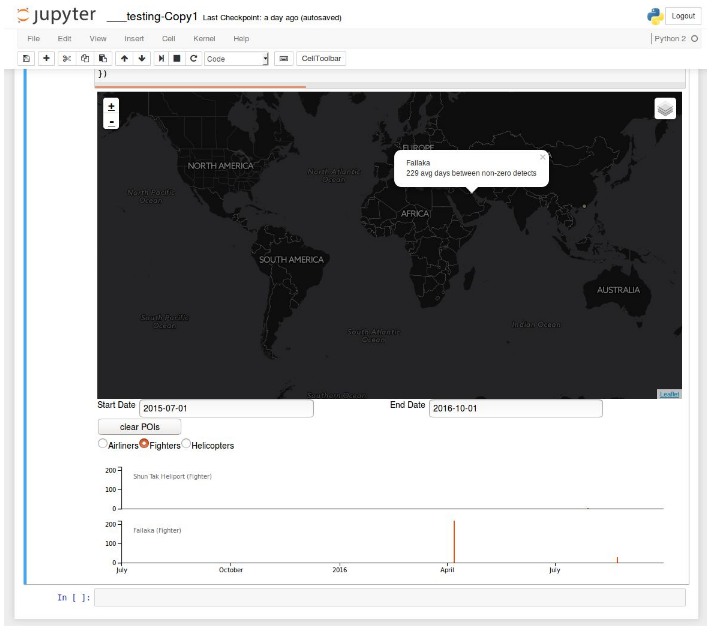
There are two big things to work out:

1. Getting the right libraries on the classpath.
2. Wiring up visualizations.


jupyter
gdelt_heatmap Last Checkpoint: an hour ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Help
Spark Geomesa - Scala
Code
CellToolbar

```
In [11]: %AddJar file:///opt/jupyter/jai_core-1.1.3.jar
         %AddJar file:///opt/jupyter/jai_imageio-1.1.jar
         %AddJar file:///opt/jupyter/jai_codec-1.1.3.jar

         Starting download from file:///opt/jupyter/jai_core-1.1.3.jar
         Finished download of jai_core-1.1.3.jar
         Starting download from file:///opt/jupyter/jai_imageio-1.1.jar
         Finished download of jai_imageio-1.1.jar
         Starting download from file:///opt/jupyter/jai_codec-1.1.3.jar
         Finished download of jai_codec-1.1.3.jar

In [ ]:  import java.awt.image._
         import java.io._
         import javax.imageio._
         import javax.media.jai._

         val max = density.map(_._2).reduceLeft(_ max _).toFloat
         val bi = new BufferedImage(xCount,yCount,BufferedImage.TYPE_BYTE_GRAY);
         val wr = bi.getRaster();
         density.foreach( c => {
           val gh = c._1
           val x = GeoHash.gridIndexForLongitude(gh).toInt
           val y = GeoHash.gridIndexForLatitude(gh).toInt
           val n = -128 + c._2 / max * 256
           wr.setSample(x, yCount - y, 0, n.toByte)
         })
         ImageIO.write(bi, "PNG",  new File("density_gdelt.png"))

         <img src="density_gdelt.png"/>
```

Interactive Data Discovery at Scale in GeoMesa Notebooks


jupyter ___testing-Copy1 Last Checkpoint: a day ago (autosaved)
Logout
File Edit View Insert Cell Kernel Help
Python 2
Code
CellToolbar
NORTH AMERICA
EUROPE
AFRICA
SOUTH AMERICA
AUSTRALIA
Failaka
229 avg days between non-zero detects
Start Date 2015-07-01
End Date 2016-10-01
clear POIs
Airliners Fighters Helicopters
Shun Tak Heliport (Fighter)
Failaka (Fighter)
July October 2016 April July
In [ ]:

GeoMesa Notebook Roadmap:

- Improved JavaScript integration
- D3.js and other visualization libraries
- OpenLayers and Leaflet
- Python Bindings

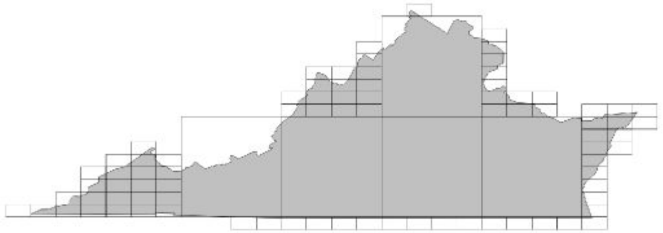
Backup Slides

Indexing non-point geometries

Indexing non-point geometries: XZ Index

Most approaches to indexing non-point geometries involve covering the geometry with a number of grid cells and storing a copy with each index.

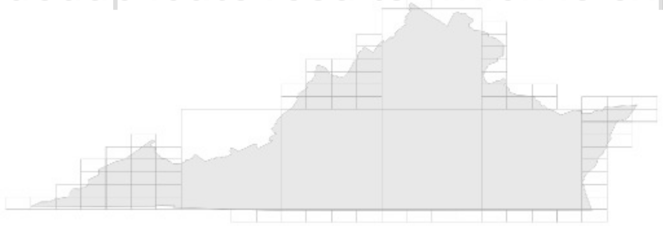
This means that the client has to deduplicate results which is expensive.



Indexing non-point geometries: XZ Index

Most approaches to indexing non-point geometries involve covering the geometry with a number of grid cells and storing a copy with each index.

This means that the client has to deduplicate results which is expensive.



Böhm, Klump, and Kriegel describe an indexing strategy allows such geometries to be stored once.

GeoMesa has implemented this strategy in XZ2 (spatial-only) and XZ3 (spatio-temporal) tables.

The key is to store data by resolution, separate geometries by size, and then index them by their lower left corner.

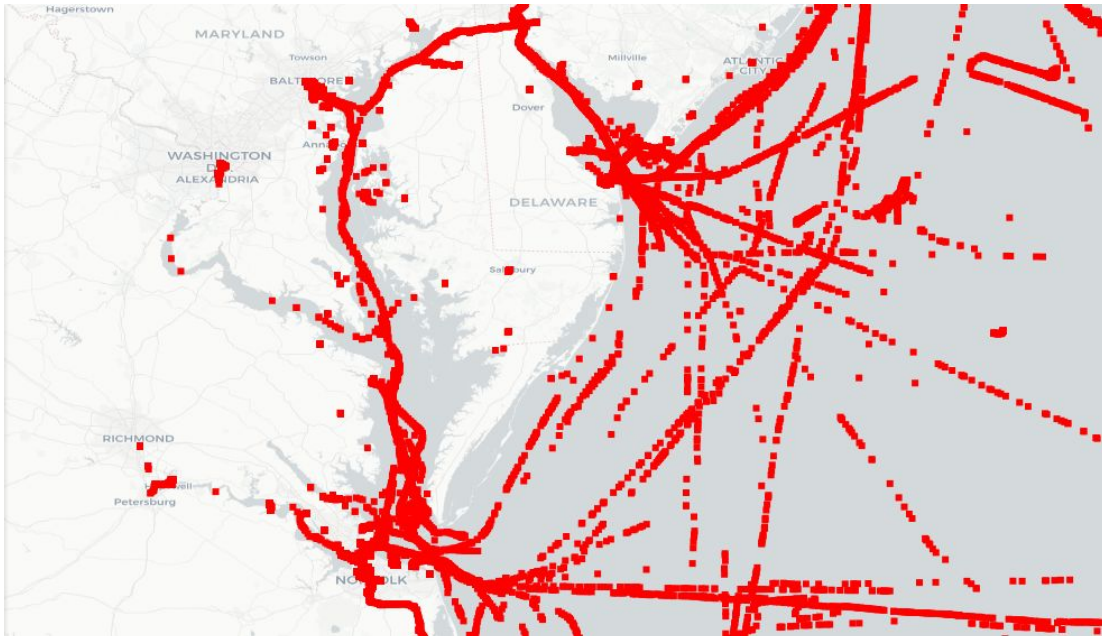
This does require consideration on the query planning side, but avoiding deduplication is worth the trade-off.

For more details, see Böhm, Klump, and Kriegel. “XZ-ordering: a space-filling curve for objects with spatial extension.” 6th. Int. Symposium on Large Spatial Databases (SSD), 1999, Hong Kong, China. (http://www.dbs.ifi.lmu.de/Publikationen/Boehm/Ordering_99.pdf)

Demo Backup Slides

CCRi
DATA TO KNOWLEDGE

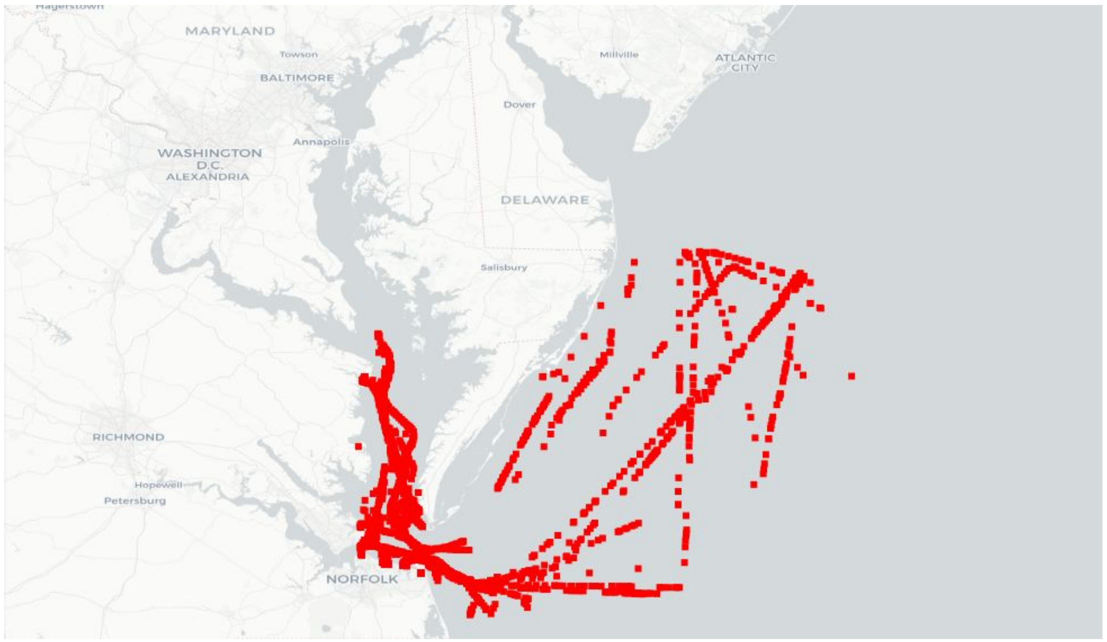
Query by bounding box



Here the viewport is used as the spatial bounds for the query.

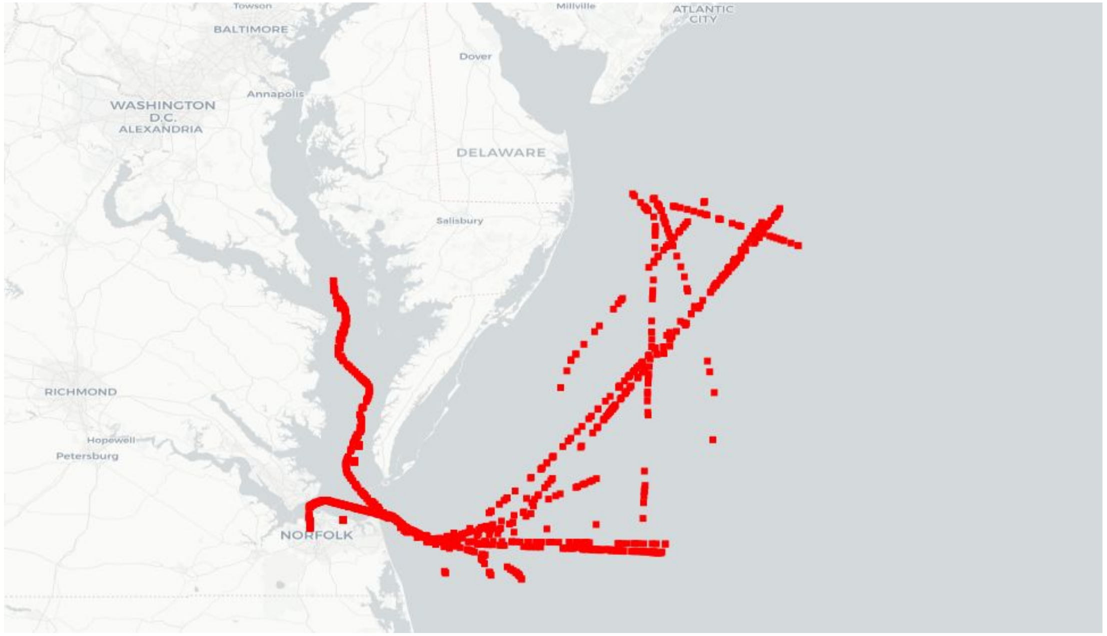
The time range is a 12 hour period on Monday.

Query by polygon



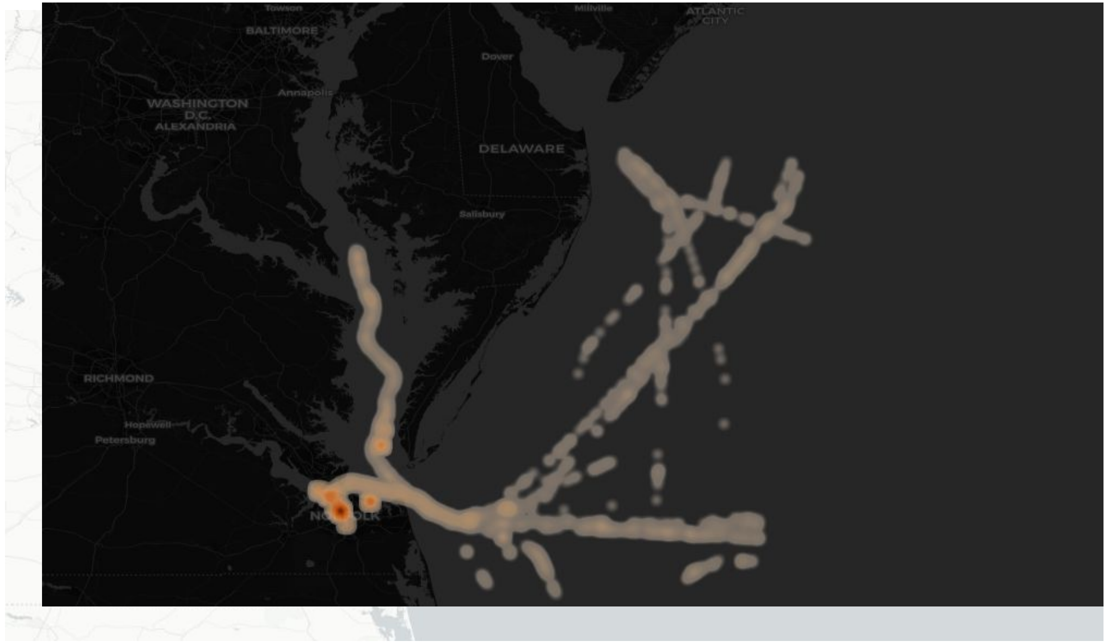
Here we further restrict the query region by an arbitrary polygon

Query by polygon and vessel type



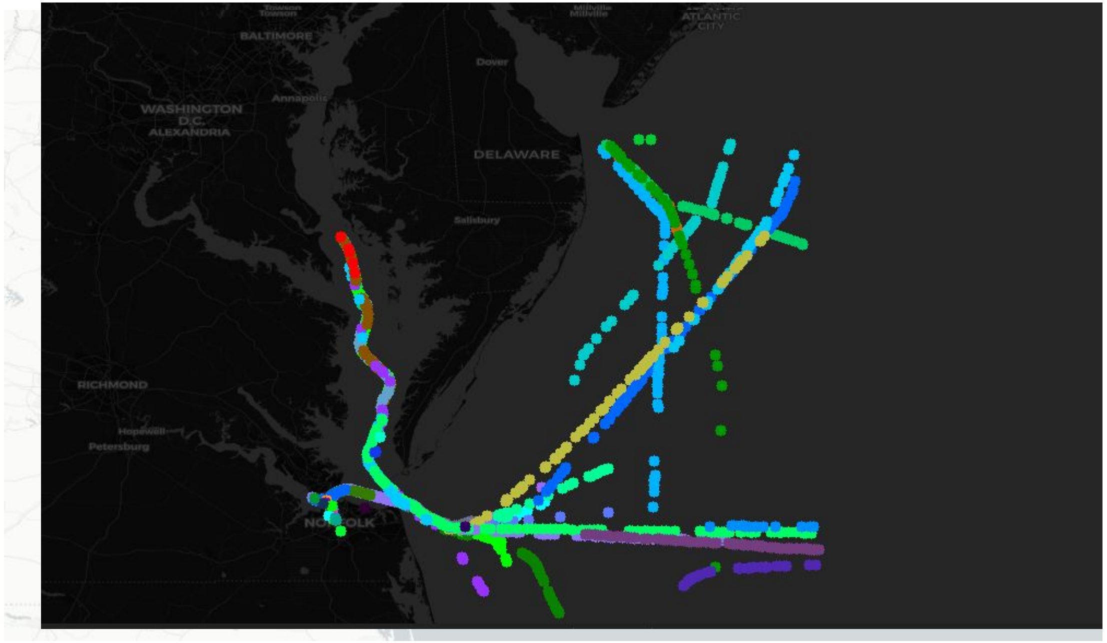
Here, we have added a clause to restrict to **cargo** vessels

Query by polygon and vessel type (heatmap)



Heatmaps can be generated

Query by polygon and vessel type (Apache Arrow format)

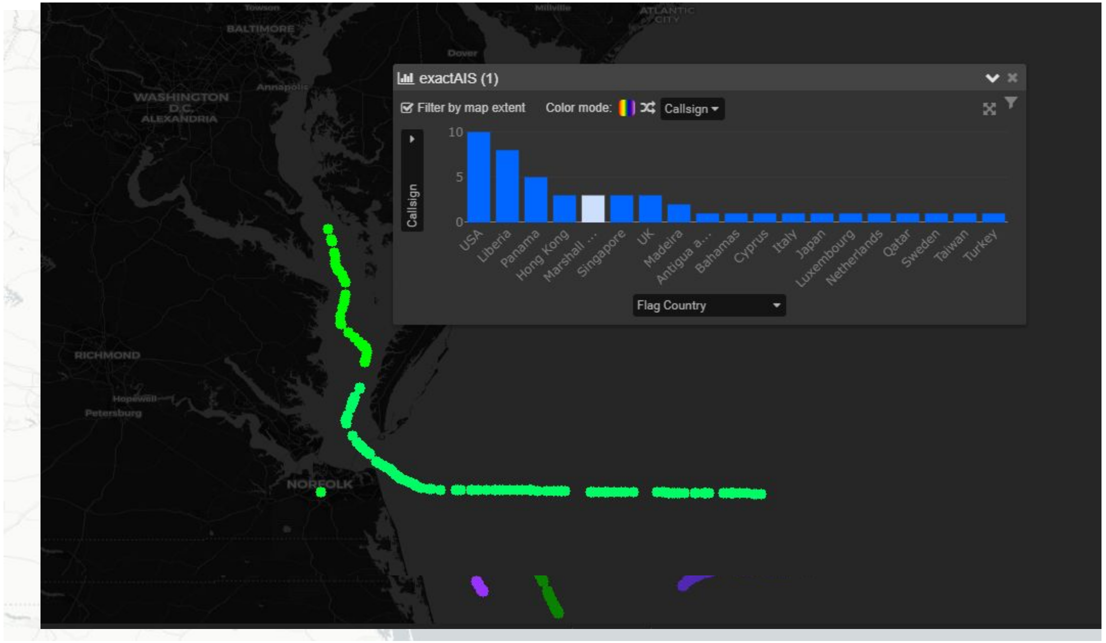


Data can be returned in a number of formats.

The Apache Arrow format allows for rapid access in JavaScript.

Here, points are colored by callsign.

Query by polygon and vessel type (Apache Arrow format)



Apache Arrow allows for in browser data exploration.

This histogram shows callsigns grouped by country.

Selections in the histogram can influence the map.