

# Using GeoMesa on top of Apache Accumulo, HBase, Cassandra, and big data file formats for massive geospatial data

ApacheCon 2019  
James Hughes



# 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
- Committer representative to the LocationTech Steering Committee



# Talk outline

- Background
  - LocationTech Projects
  - Foundational LocationTech Projects
- GeoMesa
  - Distributed Databases (HBase/Accumulo)
  - Apache File formats
  - Streaming with Kafka
  - Analysis with Spark
- Other LocationTech big-data projects
  - GeoTrellis
  - GeoWave
  - RasterFrames

# What is LocationTech?

LocationTech is a working group of the Eclipse Software Foundation



LocationTech

Advanced Geospatial Software



Internet of Things



SCIENCE

eclipse.org

Scientific Research



Long Term Support



POLARSys

Embedded Systems



# LocationTech Projects

- **Foundational Libraries**

- JTS
- Spatial4j
- SFCurve
- Proj4j
- ImageN
- libspatialindex

- **Big Data Projects**

- GeoMesa
- GeoTrellis
- GeoWave
- RasterFrames

- **Other**

- GeoGig
- GeoPeril



# Foundational Libraries

Geometry  
Topology  
Indexing

- JTS,
- Spatial4J
- SFCurve

# LocationTech JTS

The JTS Topology Suite is a Java library for creating and manipulating vector geometry.

Provides implementations of

- OGC Geometry classes
- Basic relationships
- Topology operations

# LocationTech JTS License

LocationTech JTS is available under a dual-license:

- Eclipse Public License 1.0
- Eclipse Distribution License 1.0 (BSD style)

“The Eclipse Foundation makes available all content in this project ("Content"). Unless otherwise indicated below, the Content is provided to you under the terms and conditions of either the Eclipse Public License 1.0 ("EPL") or the Eclipse Distribution License 1.0 (a BSD Style License). For purposes of the EPL, "Program" will mean the Content.”

# JTS Data Types

Representations:

OGC Simple Features

Point

LineString

LinearRing

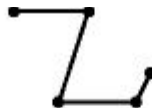
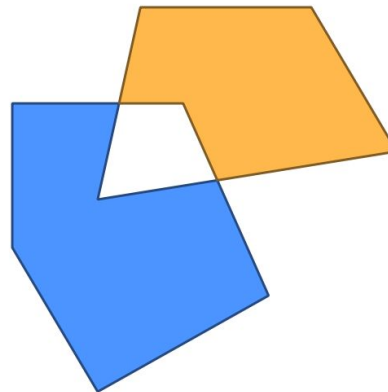
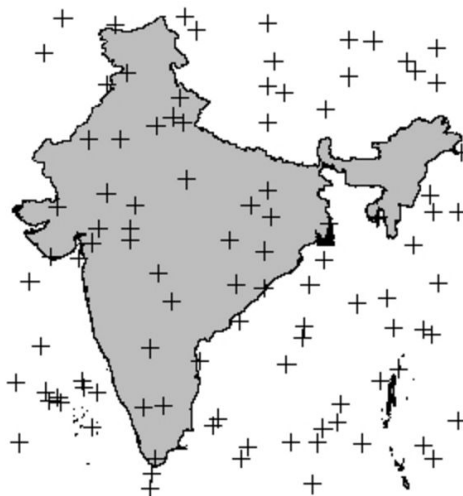
Polygon

MultiPoint

MultiLineString

MultiPolygon

GeometryCollection



# JTS Supported IO Formats

IO:

- WKT
- WKB
- GeoJSON
- KML
- GML2

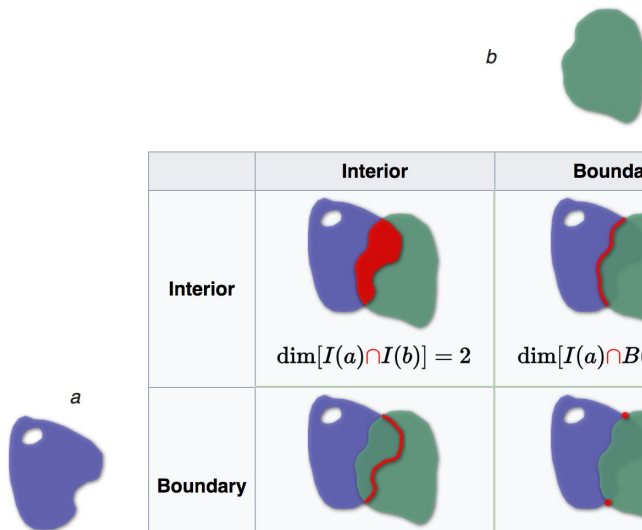
```
wkt_geom
Polygon ((-105.03792611059080286
39.78014782225491786, -105.04818400099962616
39.75856265597848704, -105.02284438556741009
39.75418720873850731, -105.01231287864754904
39.76789982851657612, -105.01364722199988933
39.78389171288461768, -105.03792611059080286
39.78014782225491786))
```








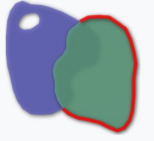
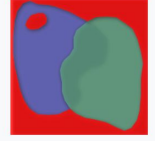
```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [
      -122.65335738658904,
      45.512083676585156
    ]
  },
  "properties": {
    "name": "Hungry Heart Cupcakes",
    "address": "1212 SE Hawthorne Boulevard",
    "website": "http://www.hungryheartcupcakes.com",
    "gluten free": "no"
  }
}
```

# JTS Topological Predicate Support

## Predicates (DE-9IM)

Equals  
 Disjoin  
 Intersects  
 Touches  
 Crosses  
 Within  
 Contains  
 Overlaps  
 Covers  
 CoveredBy



	Interior	Boundary	Exterior
Interior	 $\dim[I(a) \cap I(b)] = 2$	 $\dim[I(a) \cap B(b)] = 1$	 $\dim[I(a) \cap E(b)] = 2$
Boundary	 $\dim[B(a) \cap I(b)] = 1$	 $\dim[B(a) \cap B(b)] = 0$	 $\dim[B(a) \cap E(b)] = 1$
Exterior	 $\dim[E(a) \cap I(b)] = 2$	 $\dim[E(a) \cap B(b)] = 1$	 $\dim[E(a) \cap E(b)] = 2$

# JTS Topology Operations

## Algorithms

Convex Hull

Buffer

Validation

Dissolve

Overlay operations

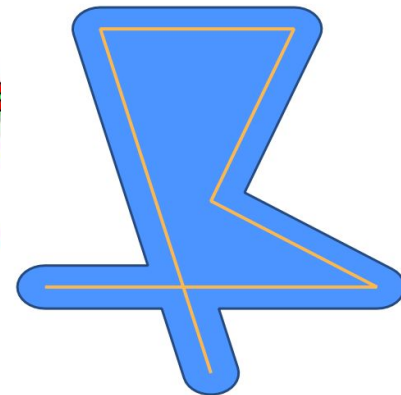
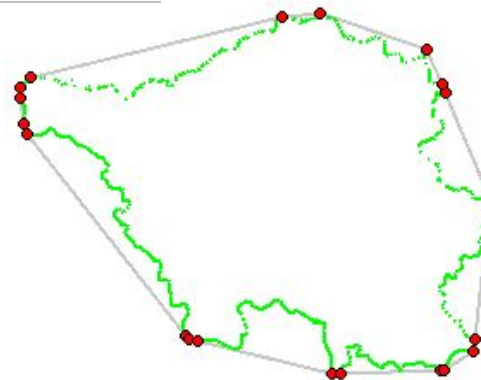
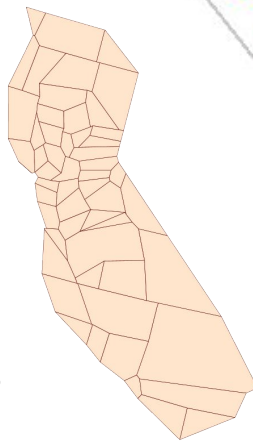
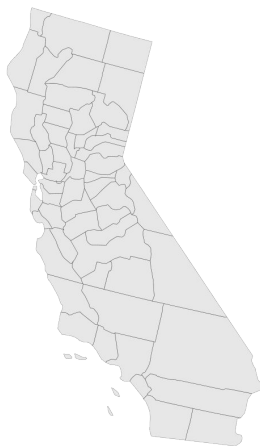
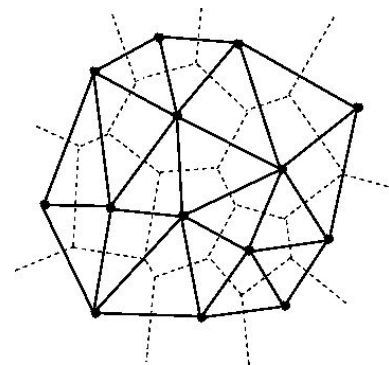
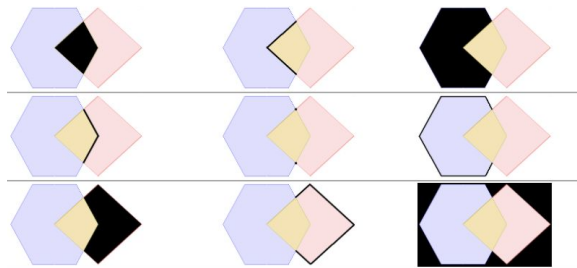
Polygonization

Simplification

Triangulation

Voronoi

Linear Referencing  
and more...





# JTS

# GEOS

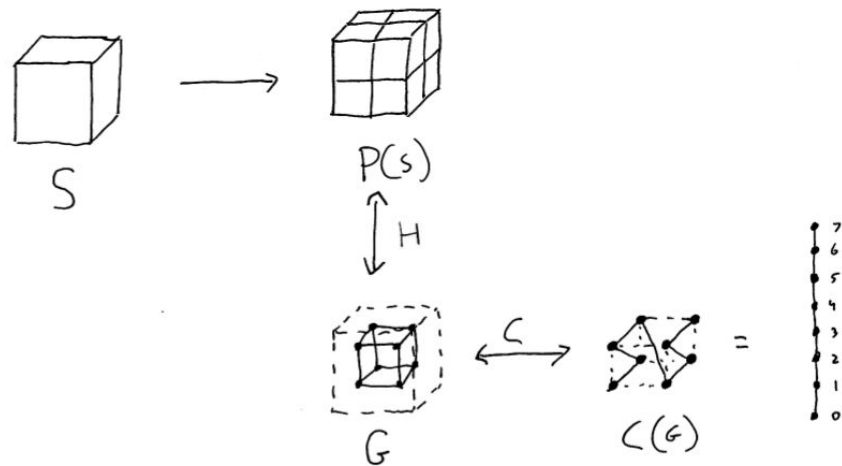
**JSTS**

# Spatial4j

- Spatial4j is a general purpose spatial / geospatial ASL licensed open-source Java library. Capabilities:
  - provide common shapes that can work in Euclidean and geodesic world models
  - provide distance calculations and other math
  - read & write shapes from formats like WKT and GeoJSON
- Came out of need to add spatial indexing to Apache Lucene

# SFCurve

- This library represents a collaborative attempt to create a solid, robust and modular library for dealing with space filling curves on the JVM.
- Implementations of space-filling curves:
  - Hilbert (2D)
  - Z-order (2D, 3D, nD)



# LocationTech GeoMesa

- GeoMesa Overview
- Reference Architecture
- Live Demo

## What is GeoMesa?

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



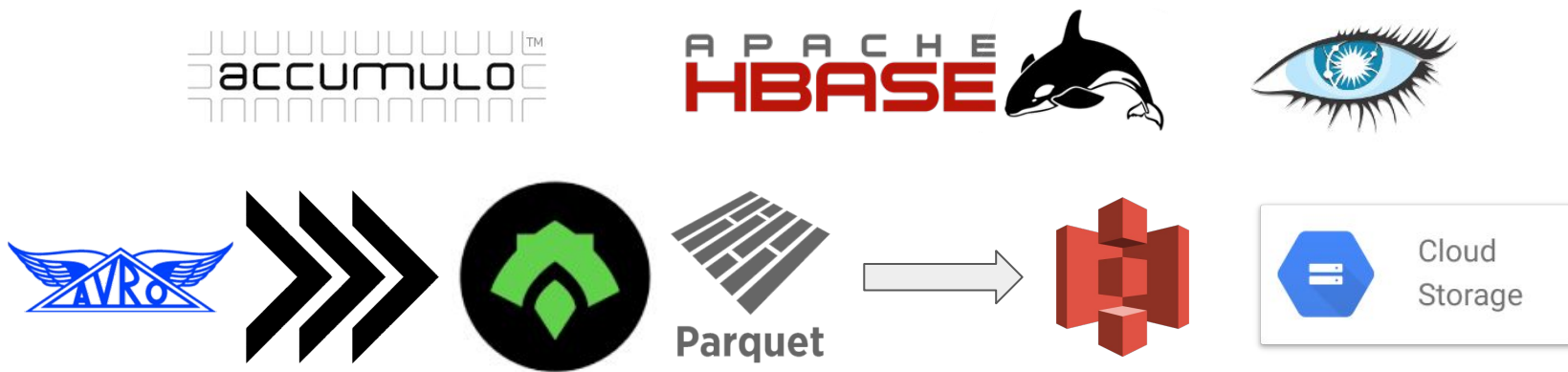
## What is GeoMesa?

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



# What is GeoMesa?

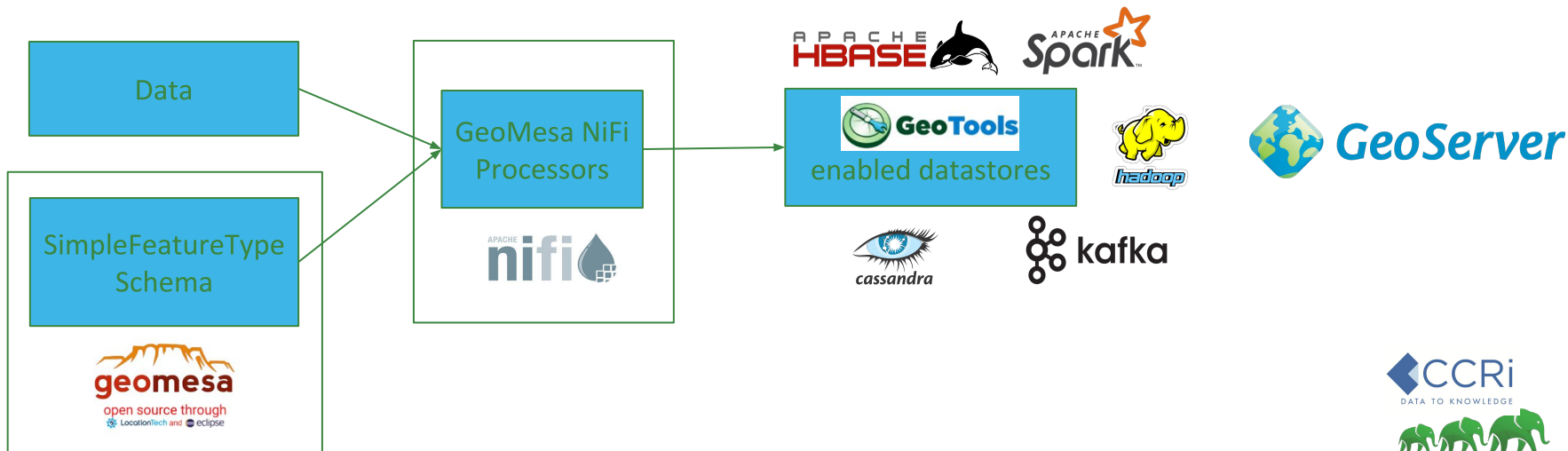
A suite of tools for streaming, persisting, **managing**, and analyzing spatio-temporal data at scale





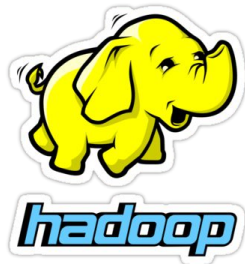
# GeoMesa NiFi

- GeoMesa-NiFi allows you to ingest data into GeoMesa straight from NiFi by leveraging custom processors.
- NiFi allows you to ingest data into GeoMesa from every source GeoMesa supports and more.

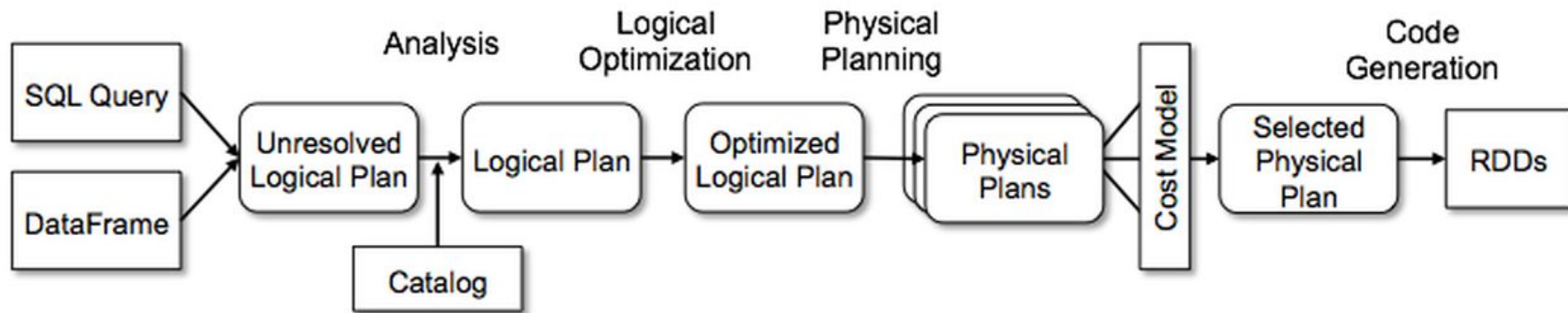


## What is GeoMesa?

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

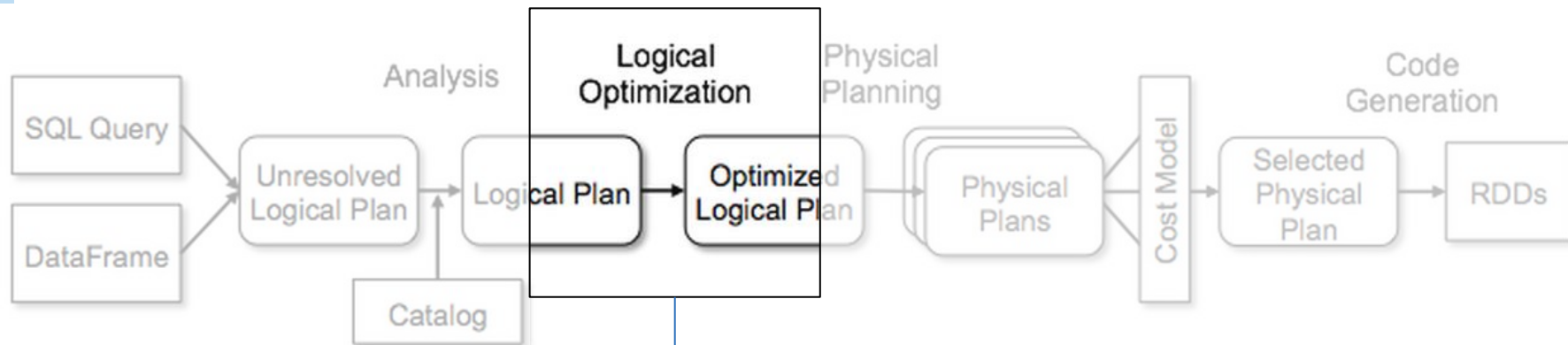


# Extending Spark's Catalyst Optimizer



<https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html>

# Extending Spark's Catalyst Optimizer



<https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html>

Catalyst exposes hooks to insert optimization rules in various points in the query processing logic.

# SQL optimizations for Spatial Predicates

SELECT

activity\_id, user\_id, geom, dtg

FROM

activities

WHERE

st\_contains(st\_makeBB0X(-78, 37, -77, 38), geom) AND

dtg > cast('2017-06-01' as timestamp) AND

dtg < cast('2017-06-05' as timestamp)

# SQL optimizations for Spatial Predicates

SELECT

activity\_id, user\_id, geom, dtg

FROM



WHERE

st\_contains(st\_makeBB0X(-78, 37, -77, 38), geom) AND

dtg > cast('2017-06-01' as timestamp) AND

dtg < cast('2017-06-05' as timestamp)

# SQL optimizations for Spatial Predicates

SELECT

activity\_id, user\_id, geom, dtg

Relational Projection

FROM

activities

WHERE

st\_contains(st\_makeBB0X(-78, 37, -77, 38), geom) AND

dtg > cast('2017-06-01' as timestamp) AND

dtg < cast('2017-06-05' as timestamp)

# SQL optimizations for Spatial Predicates

SELECT

activity\_id, user\_id, geom, dtg

FROM

activities

WHERE

st\_contains(st\_makeBB0X(-78, 37, -77, 38), geom) AND

dtg > cast('2017-06-01' as timestamp) AND

dtg < cast('2017-06-05' as timestamp)

Topological Predicate



# SQL optimizations for Spatial Predicates

SELECT

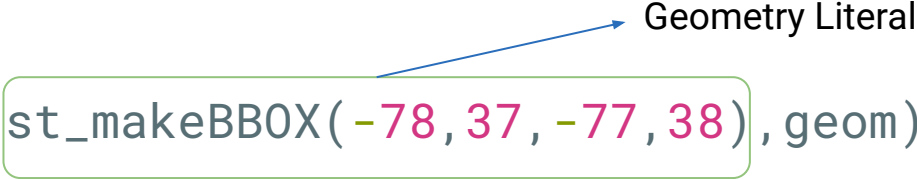
activity\_id, user\_id, geom, dtg

FROM

activities

WHERE

st\_contains(st\_makeBB0X(-78, 37, -77, 38), geom) AND  
dtg > cast('2017-06-01' as timestamp) AND  
dtg < cast('2017-06-05' as timestamp)



# SQL optimizations for Spatial Predicates

SELECT

activity\_id, user\_id, geom, dtg

FROM

activities

WHERE

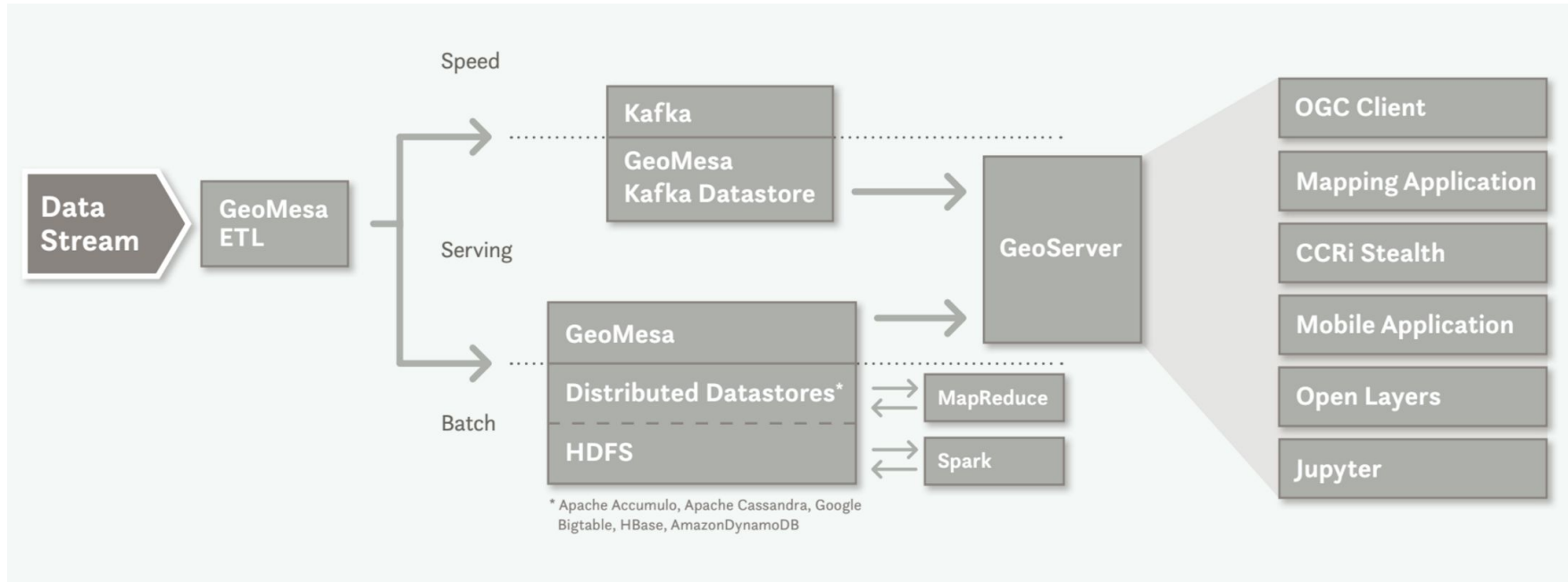
st\_contains(st\_makeBB0X(-78, 37, -77, 38), geom) AND

Date range predicate

dtg > cast('2017-06-01' as timestamp) AND

dtg < cast('2017-06-05' as timestamp)

# Proposed Reference Architecture



# Live Demo!

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

# GeoMesa's Persistence

- Distributed Databases
  - Accumulo, HBase, Cassandra
- File formats
  - Arrow, Avro, Orc, Parquet

# Indexing Geospatial Data In Key-Value Stores

Accumulo / HBase / Cassandra

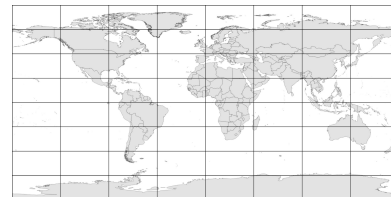
- Key Design using Space Filling Curves

# Space Filling Curves (in one slide!)

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

# Space Filling Curves (in one slide!)

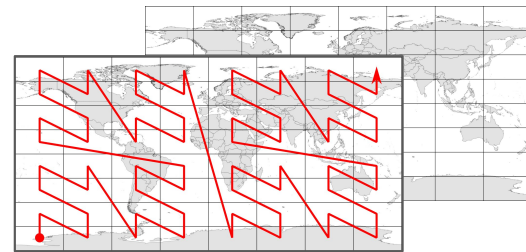
- **Goal: Index 2+ dimensional data**
- **Approach: Use Space Filling Curves**
- First, 'grid' the data space into bins.





# Space Filling Curves (in one slide!)

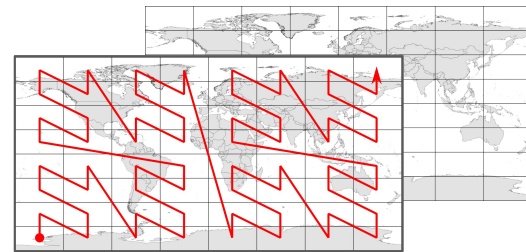
- **Goal: Index 2+ dimensional data**
- **Approach: Use Space Filling Curves**
- First, 'grid' the data space into bins.
- Next, order the grid cells with a space filling curve.
  - Label the grid cells by the order that the curve visits the them.
  - Associate the data in that grid cell with a byte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



Z2 "GeoHash"

# Space Filling Curves (in one slide!)

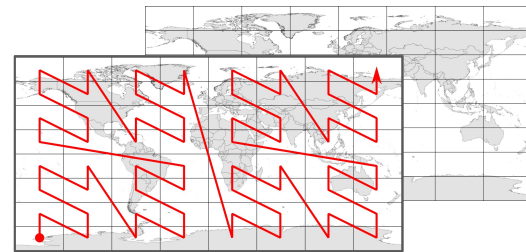
- **Goal: Index 2+ dimensional data**
- **Approach: Use Space Filling Curves**
- First, 'grid' the data space into bins.
- Next, order the grid cells with a space filling curve.
  - Label the grid cells by the order that the curve visits the them.
  - Associate the data in that grid cell with a byte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



Z2 "GeoHash"

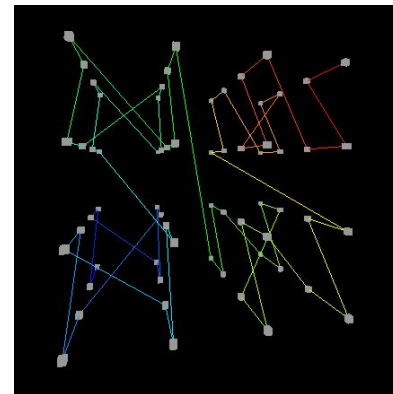
# Space Filling Curves (in one slide!)

- **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

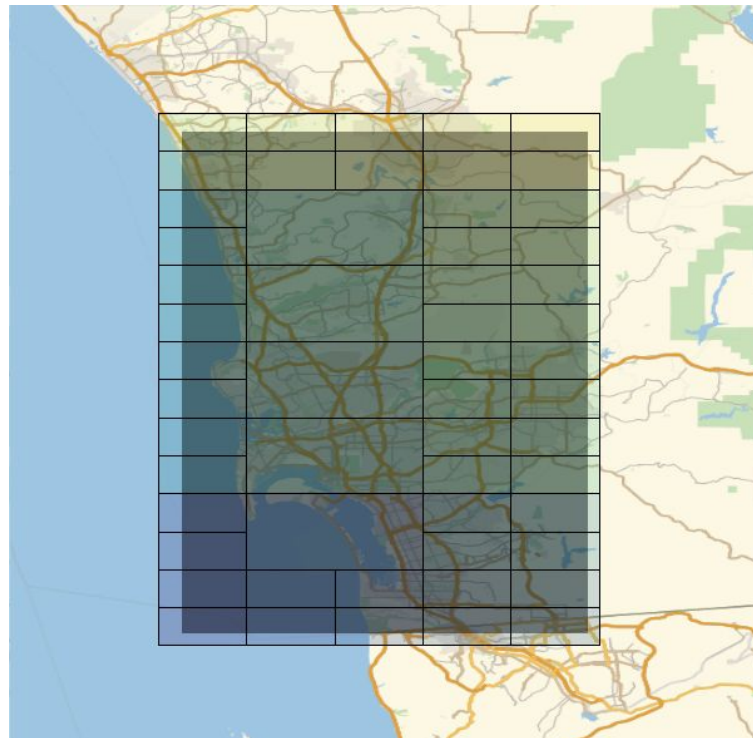


# Query planning with Space Filling Curves

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.

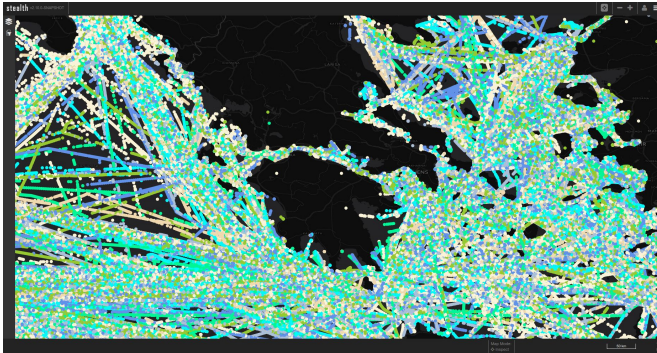
This paradigm is easily adapted to the specific implementations required by Accumulo and HBase.

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

# 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.

# Optimizing Big Data Formats for Vector Data

- File formats overview
- Spatial extensions to file formats

# Specialized Big data file formats



# Benefits of big data file formats

- Columnar layouts
- Dictionary encoding
- Efficient compression
- Structured
- Optimized filtering on read
- Language interoperability

# Benefits of big data file formats

- Columnar layouts
- Dictionary encoding
- Efficient compression
- Structured
- Optimized filtering on read
- Language interoperability

## One problem!

- No spatial types!

# Row vs Columnar Layouts

- Row layout
  - All the data for a single **record** is contiguous
  - Easier to write and stream
- Columnar layout
  - All the data for a single **column** is contiguous
  - Can be compressed much more efficiently
  - Requires much less I/O for filtering and projections

# Row vs Columnar Layouts

	session_id	timestamp	source_ip
Row 1	1331246660	3/8/2012 2:44PM	99.155.155.225
Row 2	1331246351	3/8/2012 2:38PM	65.87.165.114
Row 3	1331244570	3/8/2012 2:09PM	71.10.106.181
Row 4	1331261196	3/8/2012 6:46PM	76.102.156.138

Source: Apache Arrow

# Row vs Columnar Layouts

	session_id	timestamp	source_ip
Row 1	1331246660	3/8/2012 2:44PM	99.155.155.225
Row 2	1331246351	3/8/2012 2:38PM	65.87.165.114
Row 3	1331244570	3/8/2012 2:09PM	71.10.106.181
Row 4	1331261196	3/8/2012 6:46PM	76.102.156.138

Row 1	1331246660	3/8/2012 2:44PM	99.155.155.225	session_id	1331246660	1331246351	1331244570	1331261196
Row 2	1331246351	3/8/2012 2:38PM	65.87.165.114	timestamp	3/8/2012 2:44PM	3/8/2012 2:38PM	3/8/2012 2:09PM	3/8/2012 6:46PM
Row 3	1331244570	3/8/2012 2:09PM	71.10.106.181	source_ip	99.155.155.225	65.87.165.114	71.10.106.181	76.102.156.138
Row 4	1331261196	3/8/2012 6:46PM	76.102.156.138					

Source: Apache Arrow



# Apache Avro

- Row-based layout
- Schemas
  - Embedded (file format) or centralized (message format)
  - Supports versioning and evolution
- Optimal for streaming data (i.e. Apache Kafka), as each message is self-contained



# Apache Parquet

- Column-based layout
- Optimized for Hadoop/Spark
- Schema is embedded in the file
- Per-column compression
- Push-down predicates during read
- Column chunking allows skipping I/O



# Apache Orc

- Column-based layout
- Optimized for Hadoop/Hive
- Optimized for streaming reads
- Per-column compression
- File-level indices
- Push-down predicates during read
- Column stripes provide parallelism



# Apache Arrow

- Column-based layout
- Optimized for in-memory use
- IPC file format
- Dictionary encoding
- Zero-copy reads



# Spatial File Formats in GeoMesa

- No native spatial types
- Geometries are built up with lists of primitive columns
- Similar to GeoJSON, can be read without special type awareness

# Spatial File Formats in GeoMesa

- Points
  - Stored as two columns of type Double, one for X and one for Y
  - Arrow - stored as tuples (FixedSizeList)
- Allows for push-down filtering against each dimension

# Spatial File Formats in GeoMesa

- Points
  - Stored as two columns of type `Double`, one for X and one for Y
  - Arrow - stored as tuples (`FixedSizeList`)
- Allows for push-down filtering against each dimension
- LineStrings, MultiPoints
  - Stored as two columns of type `List[Double]`

# Spatial File Formats in GeoMesa

- Points
  - Stored as two columns of type `Double`, one for X and one for Y
  - Arrow - stored as tuples (`FixedSizeList`)
- Allows for push-down filtering against each dimension
- LineStrings, MultiPoints
  - Stored as two columns of type `List[Double]`
- MultiLineStrings, Polygons
  - Stored as two double precision `List[List[Double]]` columns
- MultiPolygons
  - Stored as two double precision `List[List[List[Double]]` columns



# Spatial File Formats in GeoMesa

- Points
  - Stored as two columns of type `Double`, one for X and one for Y
  - Arrow - stored as tuples (`FixedSizeList`)
- Allows for push-down filtering against each dimension
- LineStrings, MultiPoints
  - Stored as two columns of type `List[Double]`
- MultiLineStrings, Polygons
  - Stored as two double precision `List[List[Double]]` columns
- MultiPolygons
  - Stored as two double precision `List[List[List[Double]]` columns
- Avro - row based - WKB/TWKB/WKT

# Reading and Writing Spatial Formats

- Parquet
  - [geomesa-fs-storage-parquet](#) - SimpleFeatureParquetWriter, FilteringReader
- Orc
  - [geomesa-fs-storage-orc](#) - OrcFileSystemReader/Writer
- Arrow
  - [geomesa-arrow-jts](#) - PointVector, LineStringVector, etc
- Avro
  - [geomesa-feature-avro](#) - AvroFeatureSerializer, AvroDataFileReader/Writer

# Reading and Writing Spatial Formats

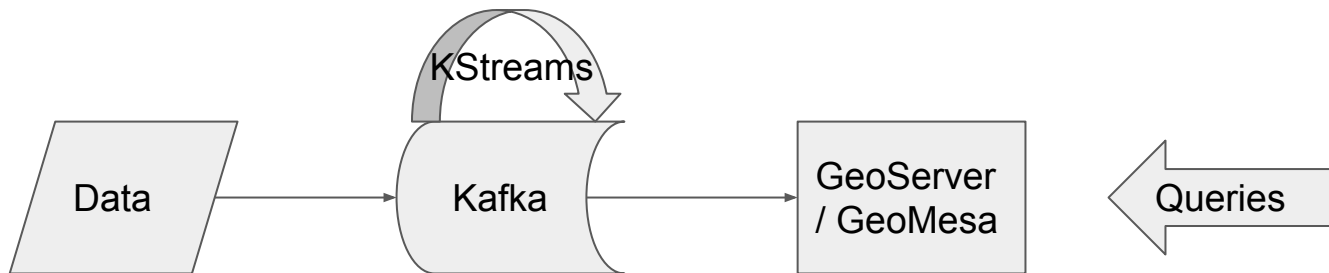
- Parquet/Orc
  - GeoMesa file system data store
  - GeoMesa CLI export/ingest
- Arrow
  - WFS/WPS requests through GeoServer
  - GeoMesa CLI export
- Avro
  - WFS/WPS requests through GeoServer
  - GeoMesa CLI export/ingest
- Standard format tools

# Spatial File Format Use Cases

- Streaming Data
- Spark analytics
- ETL and tiered storage
- Data visualization

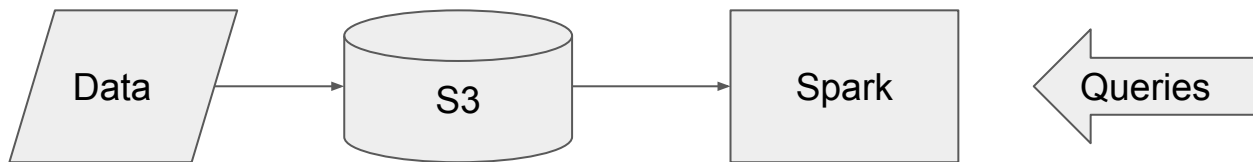
# Streaming Data - Apache Avro

- Each message is a single record (row based)
- Apache Kafka/Streams for data exchange
- Confluent schema registry is used for managing schemas
  - Small header per message uniquely identifies schema
  - Schema evolution for adding/removing fields
- GeoMesa Kafka data store for in-memory indexing



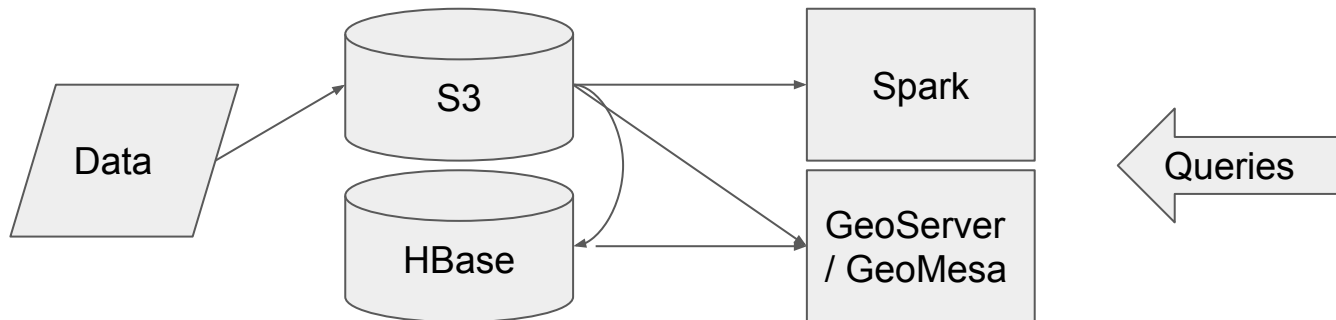
# Spark Analytics - Apache Parquet and Orc

- GeoMesa Spark integration adds spatial UDFs/UDTs
  - `st_contains`, `st_point`, etc
- Native input formats provide high throughput
- Relational projections take advantage of columnar layouts
- Predicates are pushed down into the file reads



# Tiered Storage and ETL - Apache Parquet and Orc

- Data is pre-processed into S3 using the GeoMesa converter library to create Parquet or Orc
- Processed files are ingested directly from S3 into HBase
- Processed files are accessed with the GeoMesa file system data store for large-scale analytics
- Data age-off is used to keep your HBase cluster small
- Merged view data store shows combined HBase + S3

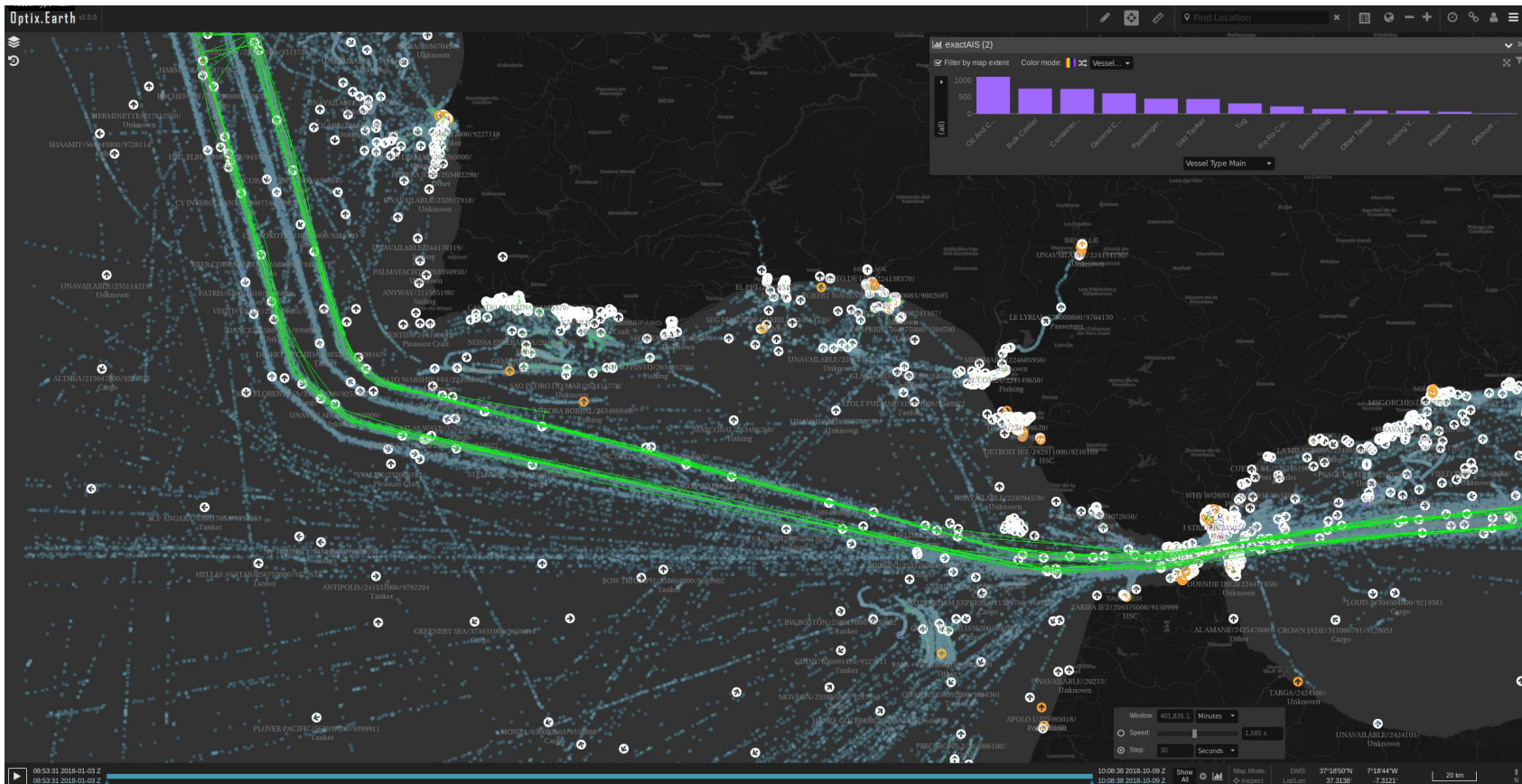


# Data Visualization - Apache Arrow

- Query Arrow IPC data through WFS/WPS
  - Distributed aggregation used where possible
- Arrow-js wraps the raw bytes and exposes the underlying data
- Can efficiently filter, sort, count, etc to display maps, histograms, timelapses



# Data Visualization in browser with Apache Arrow



# Big-Data LocationTech Projects

- GeoMesa
- GeoTrellis
- GeoWave
- RasterFrames



GeoWave

---

# What is GeoWave?

An open source framework that leverages the scalability of key-value stores for effective storage, retrieval, and analysis of massive geospatial datasets

At its core, GeoWave handles spatial and spatiotemporal indexing within distributed key-value stores with natural integrations for various popular frameworks



GeoWave bridges the gap between popular geospatial platforms and distributed processing frameworks



GeoTrellis

---

---

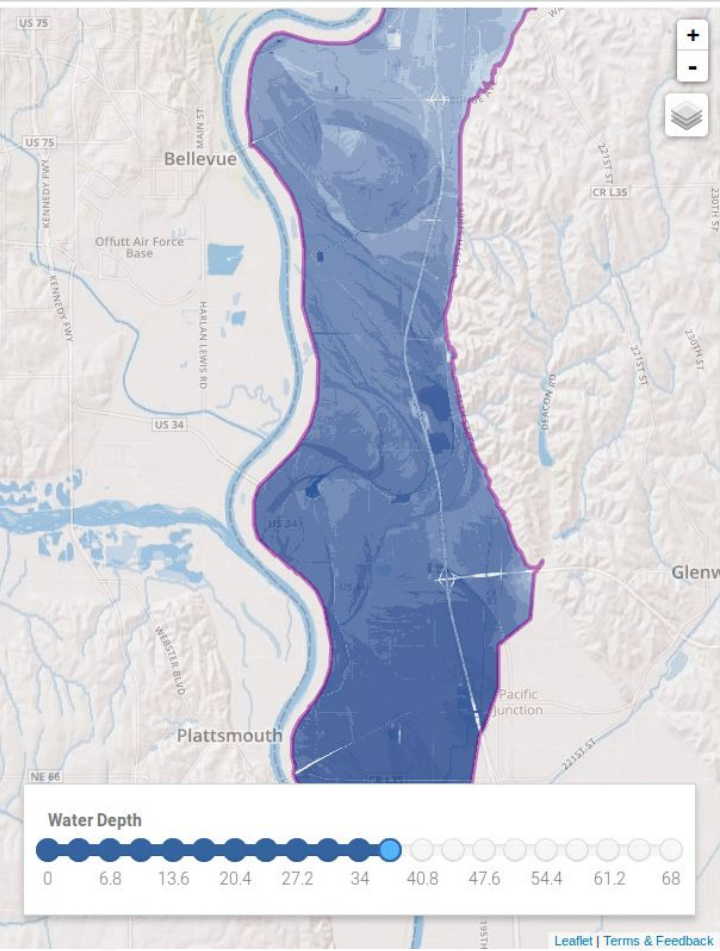
# GeoTrellis

- a Scala library for geospatial data types and operations.
- enables Spark with geospatial capabilities
- Build tile servers for on-the-fly transformations of COGs
- storage and query raster from HDFS, Accumulo, Cassandra and S3

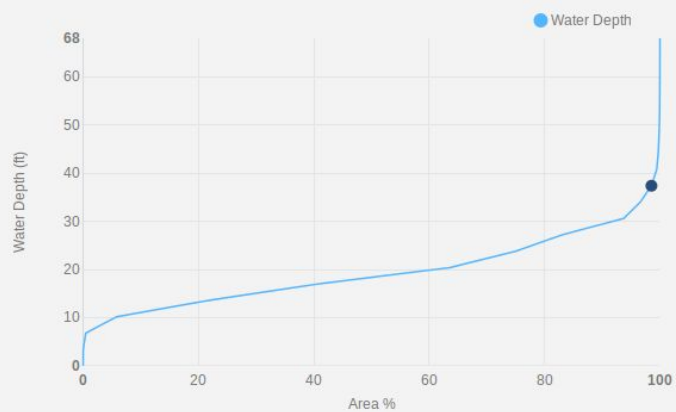


Reset

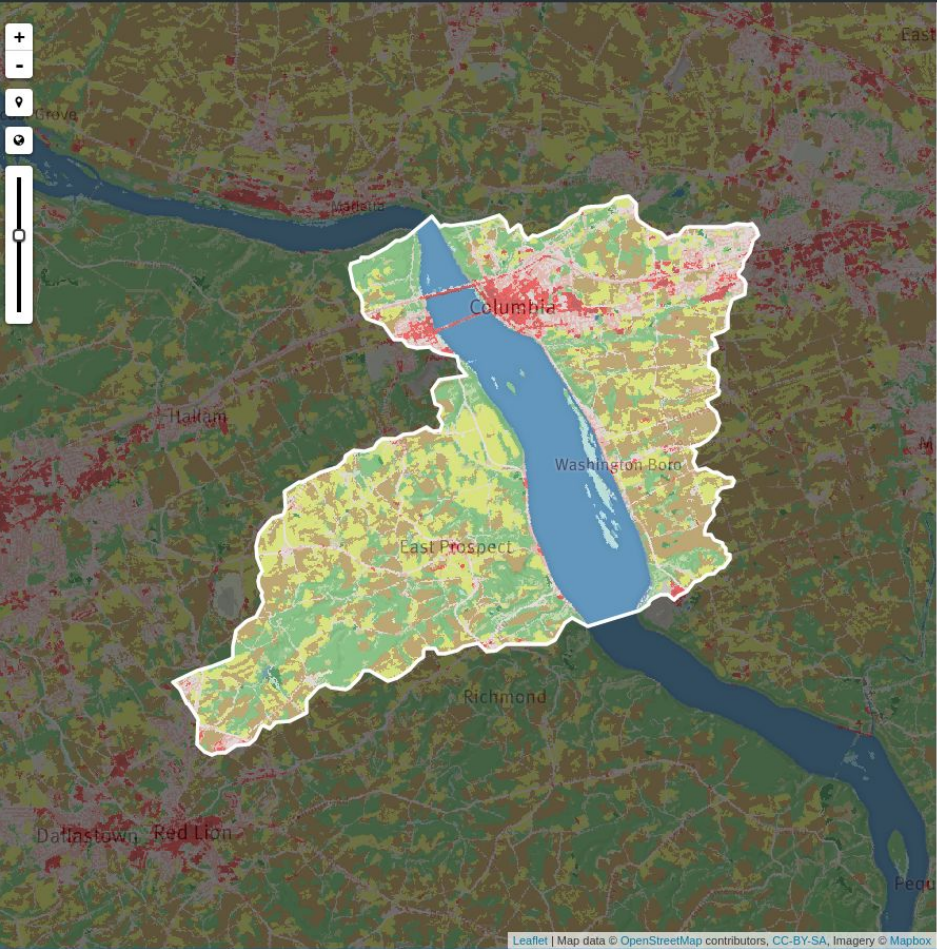
23,685 acres processed in 7.4 seconds.



Water Depth	Area %	Area acres	Structures Impacted
0.0 feet	0.00%	0 acres	0
3.4 feet	0.02%	4 acres	0
6.8 feet	0.45%	106 acres	2
10.2 feet	5.80%	1,373 acres	5
13.6 feet	21.76%	5,154 acres	23
17.0 feet	40.72%	9,644 acres	49
20.4 feet	63.58%	15,059 acres	76
23.8 feet	75.00%	17,764 acres	100
27.2 feet	83.05%	19,671 acres	150







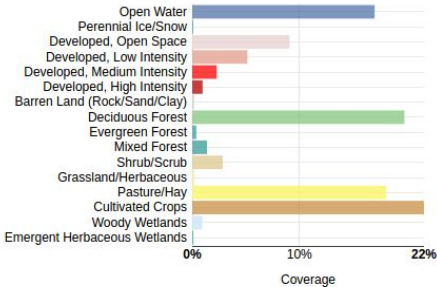
### Analyze

Back

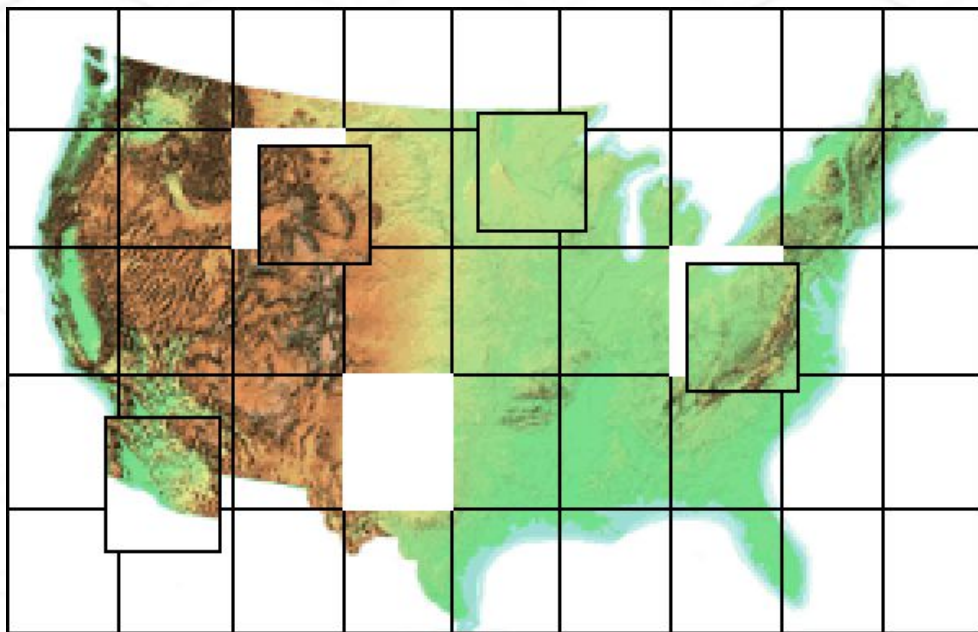
Model

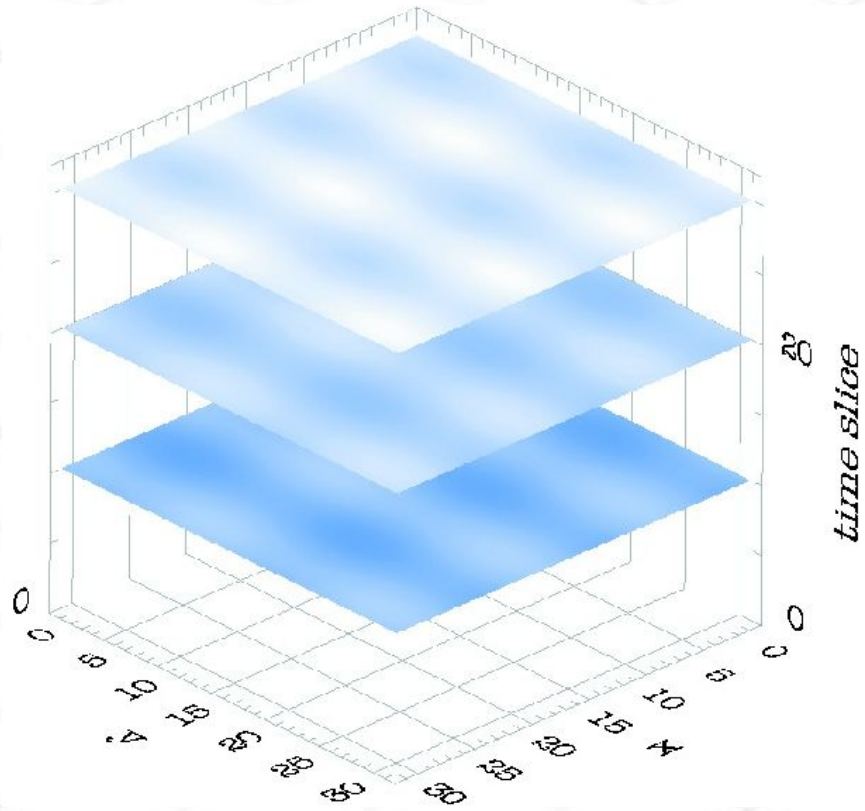
LAND SOIL

### HUC-12 Subwatershed: Cabin Creek-Susquehanna River Total Area 130 km<sup>2</sup>

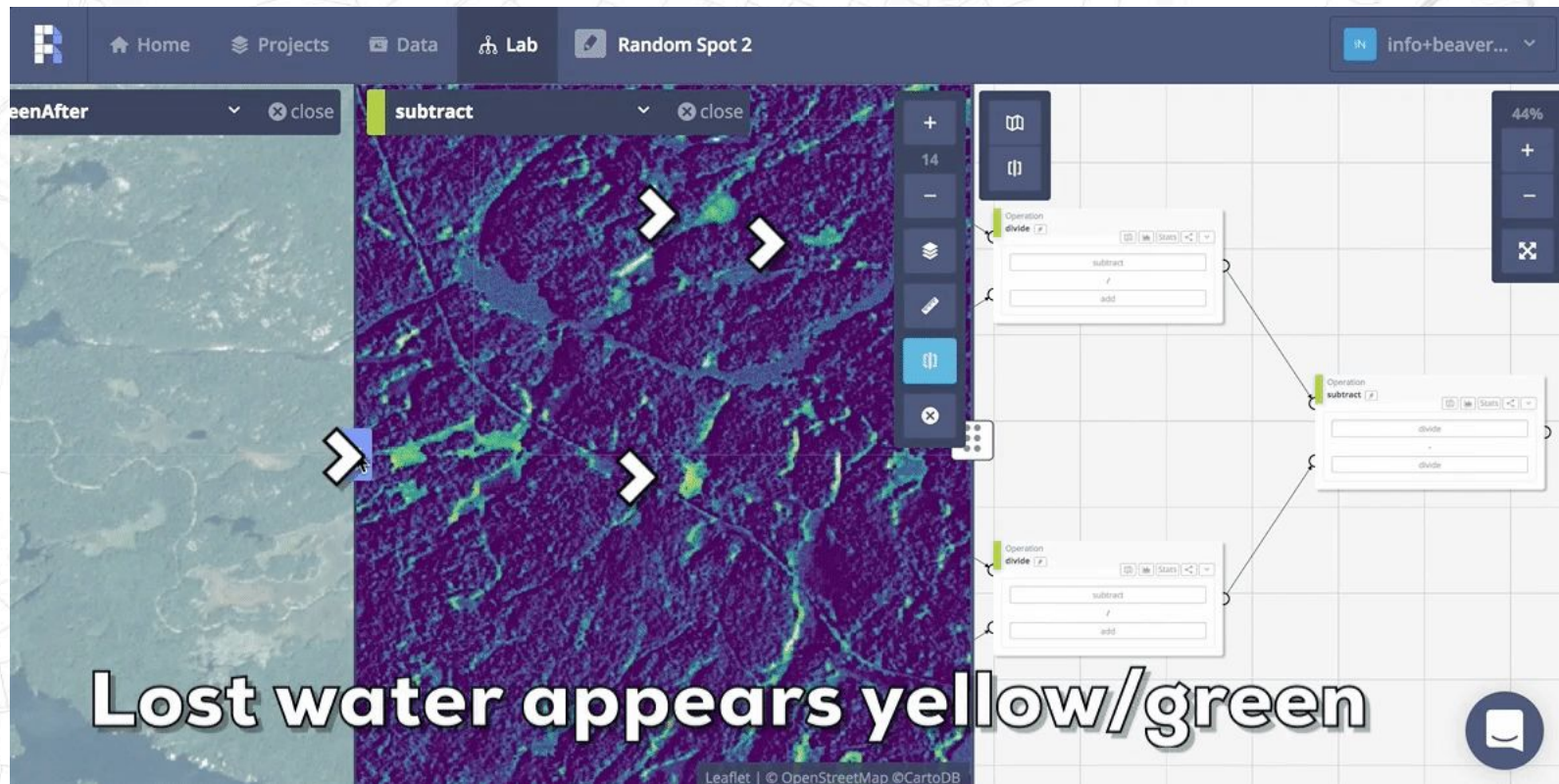


Type	Area (km <sup>2</sup> )	Coverage (%)
Open Water	21.99	17.0
Perennial Ice/Snow	0.00	0.0
Developed, Open Space	11.73	9.1
Developed, Low Intensity	6.65	5.1
Developed, Medium Intensity	2.94	2.3
Developed, High Intensity	1.25	1.0
Barren Land (Rock/Sand/Clay)	0.22	0.2
Deciduous Forest	25.58	19.8
Evergreen Forest	0.49	0.4
Mixed Forest	1.78	1.4









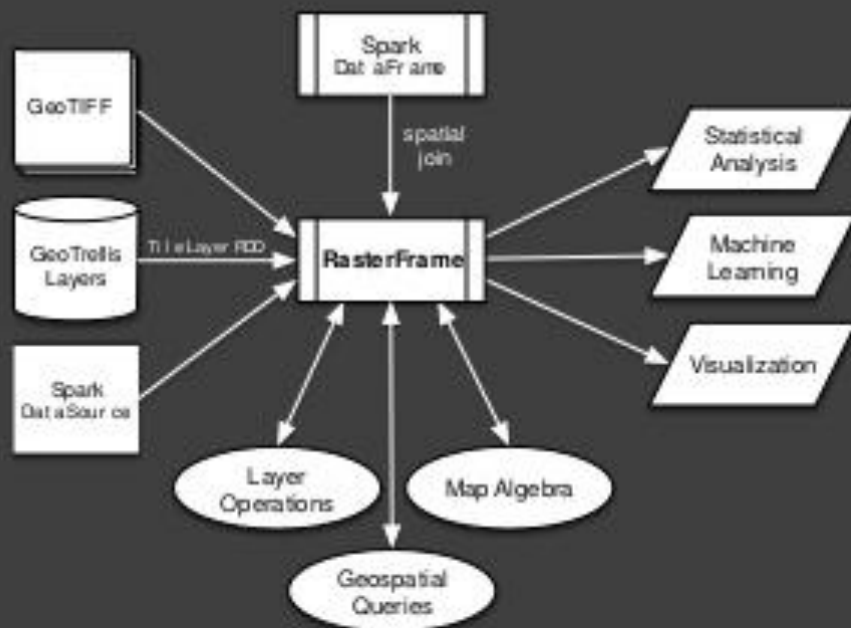


# RasterFrames



# RasterFrames

- Incubating LocationTech project
- Provides ability to work with global-scale remote sensing imagery in a convenient yet scalable format
- Integrates with multiple data sources and libraries, including Spark ML, GeoTrellis Map Algebra and GeoMesa Spark-JTS
- Python, Scala and SQL APIs



# Polyglot API



```
SELECT spatial_key,  
       rf_localAggMin(red) as red_min,  
       rf_localAggMax(red) as red_max,  
       rf_localAggMean(red) as red_mean  
FROM df  
GROUP BY spatial_key
```



```
df.groupby(df.spatial_key).agg( \  
    localAggMin(df.red).alias('red_min'), \  
    localAggMax(df.red).alias('red_max'), \  
    localAggMean(df.red).alias('red_mean'))
```



```
df.groupby("spatial_key").agg(  
    localAggMin($"red") as "red_min",  
    localAggMax($"red") as "red_max",  
    localAggMean($"red") as "red_mean")
```

# RasterFrame Take-Aways

- DataFrames lower cognitive friction when modeling. Good Ergonomics!
- Rich set of raster processing primitives
- Support for descriptive and predictive analysis
- Via `spark-shell`, Jupyter Notebook, Zeppelin, etc. can interact with data and iterate over solution
- It scales!
- Many more examples at <http://rasterframes.io>



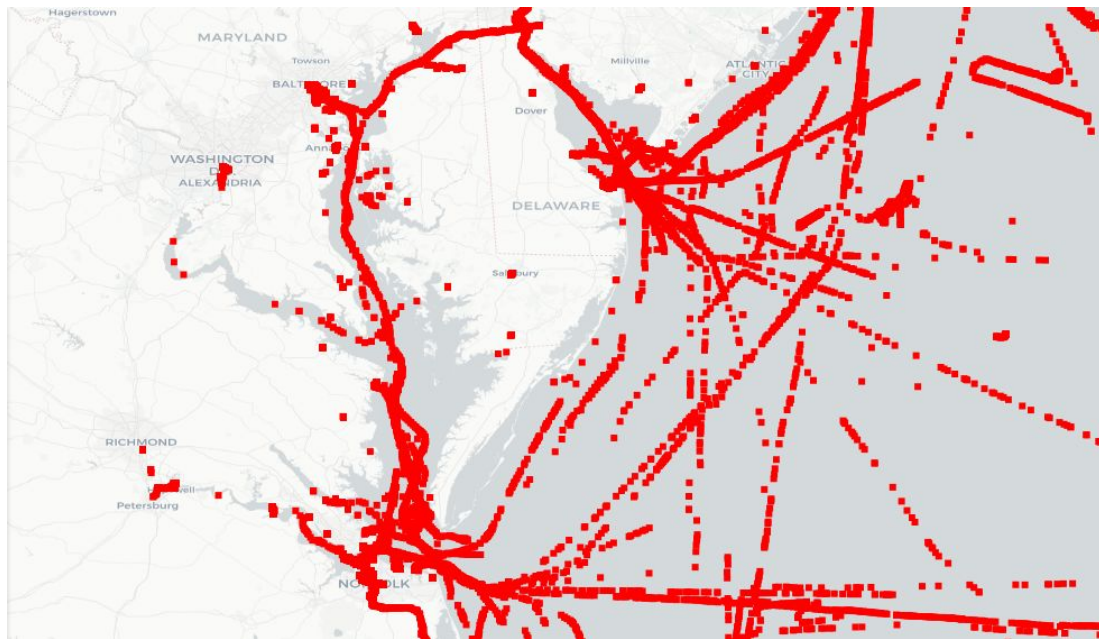
# Thanks!

James Hughes

- [jhughes@ccri.com](mailto:jhughes@ccri.com)
- <http://geomesa.org>
- <http://gitter.im/locationtech/geomesa>
- <https://github.com/locationtech/geomesa/>
- @CCR\_inc

# Demo Backup Slides

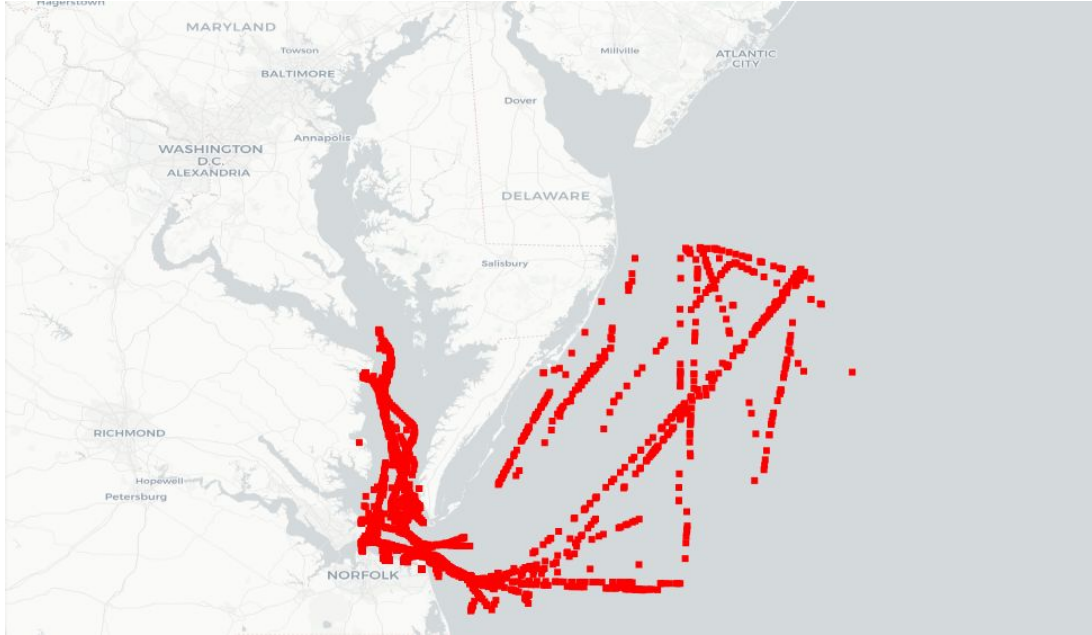
# Query by bounding box



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



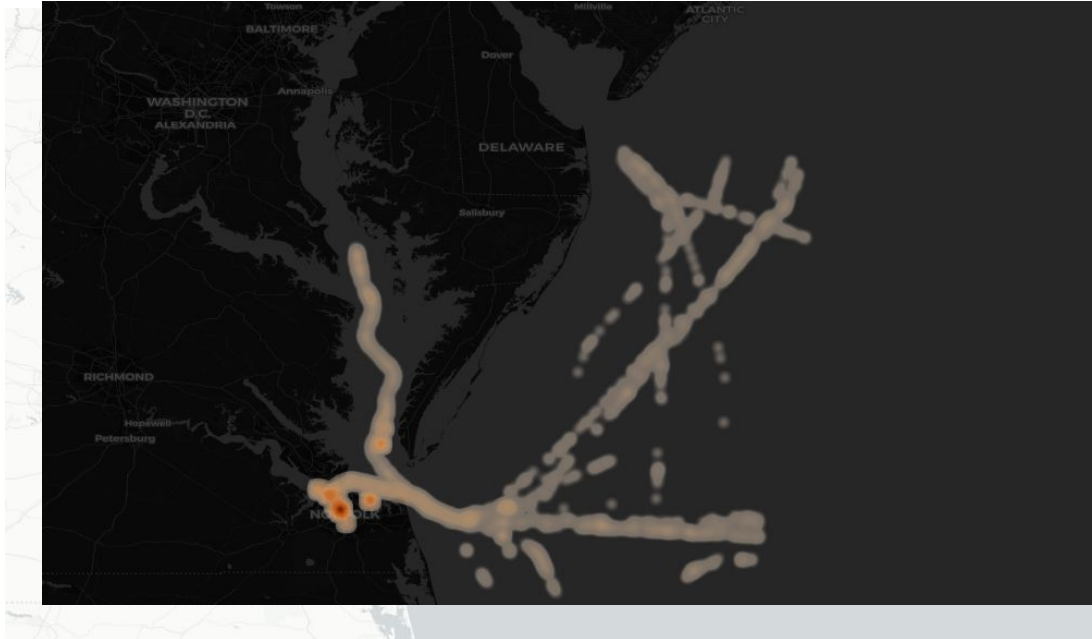
Here we further restrict the query region by an arbitrary polygon

# 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.



# Backup Slides

Integration with MapReduce / Spark

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

# GeoMesa MapReduce and Spark Support

accumulo



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.

ACCUMULO



# 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.

ACCUMULO



# 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.

ACCUMULO



---

## 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)

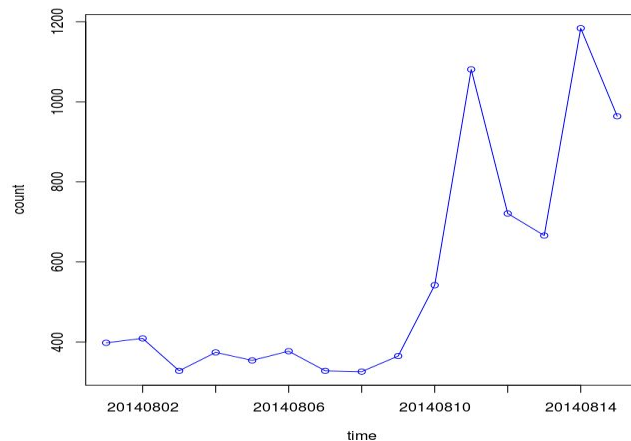
val queryRDD = geomesa.compute.spark.GeoMesaSpark.rdd(conf, sconf, ds, query)
```

## 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.

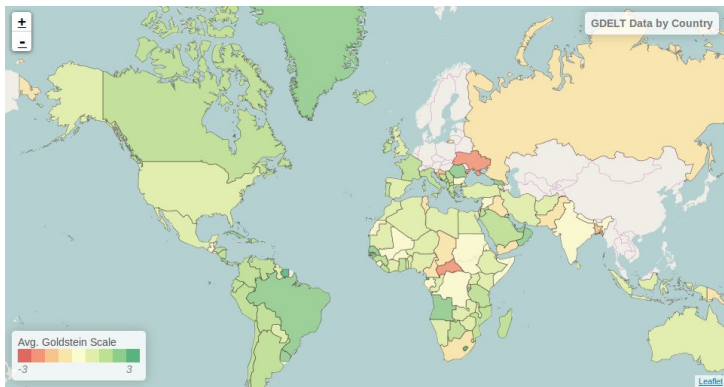


## GeoMesa Spark Example 2: Aggregating by Regions

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.

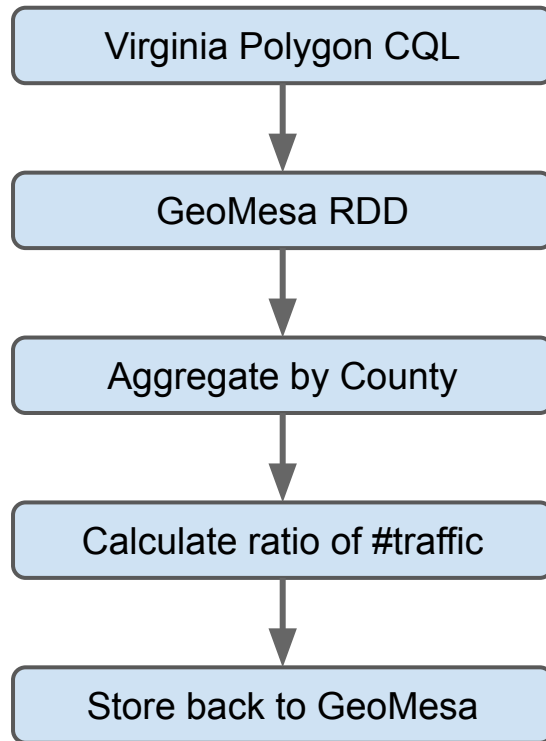
This picture shows 'stability' of a region from GDELT Goldstein values



# GeoMesa Spark Example 3: Aggregating Tweets about #traffic

```
// Create a list of Counties and iterator of
// booleans for if tweets are traffic related
val countyAndTweets = geoMesaRDD.mapPartitions { iter =>
  iter.flatMap { sf =>
    getCountry(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
  (county, pctTraffic)
}
.map { case (county, pct) => (c, getGeom(county), pct) }
.saveLayer()
```

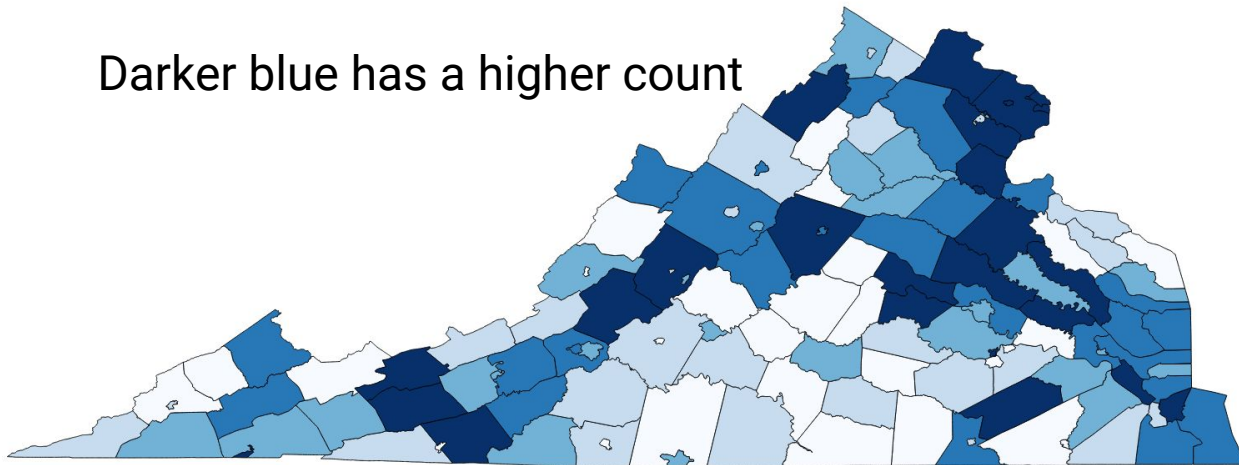




# GeoMesa Spark Example 3: Aggregating Tweets about #traffic

#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

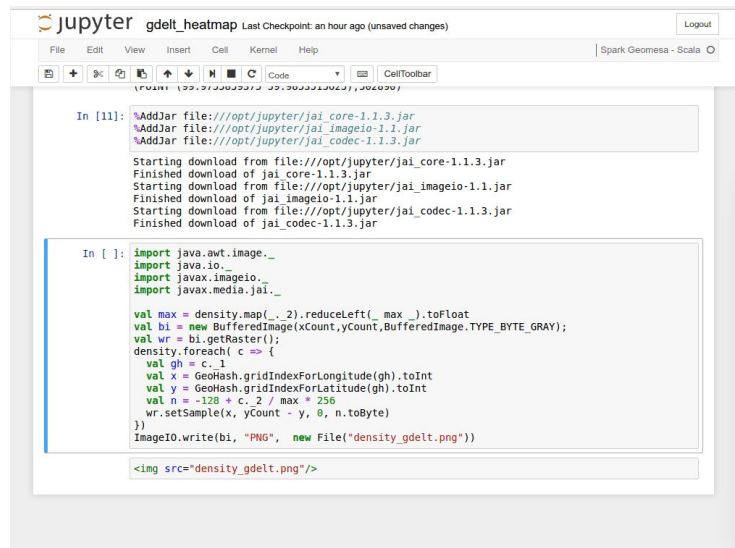
Writing (and debugging!) MapReduce / Spark jobs is slow and requires expertise.

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

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

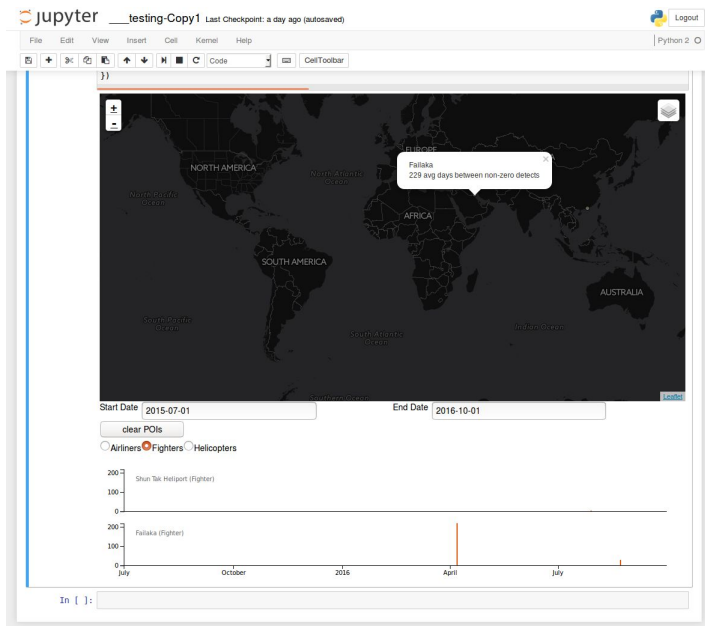
There are two big things to work out:

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



The screenshot shows a Jupyter Notebook window titled 'jupyter gdelt\_heatmap'. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with icons for file operations and code execution. The notebook content is divided into two cells. The first cell, labeled 'In [11]:', contains three lines of code to add JAR files to the classpath: `%AddJar file:///opt/jupyter/jai_core-1.1.3.jar`, `%AddJar file:///opt/jupyter/jai_imageio-1.1.jar`, and `%AddJar file:///opt/jupyter/jai_codec-1.1.3.jar`. Below the code, the output shows the progress of downloading each JAR file. The second cell, labeled 'In [ ]:', contains a block of Java code that imports necessary classes, processes a density map, and writes the result to a PNG file. The code includes comments and uses the `ImageIO` and `File` classes. At the bottom of the cell, there is a small HTML snippet: ``.

# Interactive Data Discovery at Scale in GeoMesa Notebooks



## GeoMesa Notebook Roadmap:

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

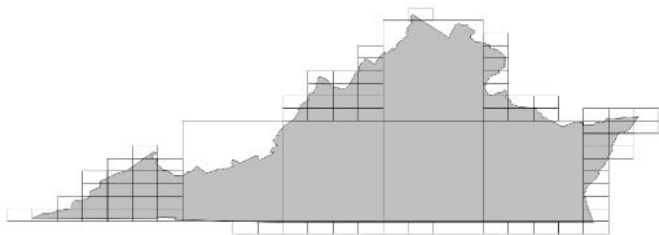
# Backup Slides

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.

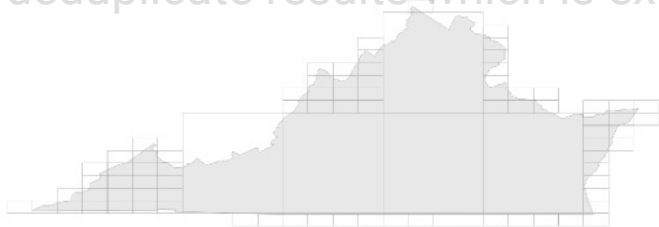
This means that the client has to deduplicate results which is expensive.



# 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](http://www.dbs.ifi.lmu.de/Publikationen/Boehm/Ordering_99.pdf))

# Backup Slides

More HBase / Accumulo details



# 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 iterators

# 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](http://org.locationtech.geomesa.hbase.filters.CqlTransformFilter)
  - Filters rows based on GeoTools CQL
  - Transforms rows based on relational projections
- [org.locationtech.geomesa.hbase.filters.Z2HBaseFilter](http://org.locationtech.geomesa.hbase.filters.Z2HBaseFilter)
  - Compares Z-values against the row key
- [org.locationtech.geomesa.hbase.filters.Z3HBaseFilter](http://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](http://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](http://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](#)