GeoTrellis Documentation

Release 1.0.0

Azavea

Aug 10, 2018

Home

GeoTrellis is a Scala library and framework that uses Apache Spark to work with raster data. It is released under the Apache 2 License.

GeoTrellis reads, writes, and operates on raster data as fast as possible. It implements many Map Algebra operations as well as vector to raster or raster to vector operations.

GeoTrellis also provides tools to render rasters into PNGs or to store metadata about raster files as JSON. It aims to provide raster processing at web speeds (sub-second or less) with RESTful endpoints as well as provide fast batch processing of large raster data sets.

Please visit the project site for more information as well as some interactive demos.

CHAPTER 1

Why GeoTrellis?

Raster processing has traditionally been a slow task, which has prompted advances in vector data processing as an alternative. Raster data isn't going anywhere, however, with more and more satellite data made public every year. GeoTrellis is an answer to the growing need for **raster processing at scale.** We personally have handled terabyte-level data sets, but really we are only bound by the theoretical limits of Apache Spark. By *scale* then, we mean *arbitrarily large*.

Contact and Support

You can find more information and talk to developers (let us know what you're working on!) at:

- Gitter
- GeoTrellis mailing list

Hello Raster!

Here's a small example showing a routine focal operation over a single `Tile`:

```scala
scala> import geotrellis.raster._
import geotrellis.raster._

scala> import geotrellis.raster.render.ascii._
import geotrellis.raster.render.ascii._

scala> import geotrellis.raster.mapalgebra.focal._
import geotrellis.raster.mapalgebra.focal._

scala> val nd = NODATA
nd: Int = -2147483648

scala> val input = Array[Int](
     nd, 7, 1, 1,  3, 5, 9, 8, 2,
      9, 1, 1, 2,  2, 2, 4, 3, 5,
      3, 8, 1, 3,  3, 3, 1, 2, 2,
      2, 4, 7, 1, nd, 1, 8, 4, 3)
input: Array[Int] = Array(-2147483648, 7, 1, 1, 3, 5, 9, 8, 2, 9, 1, 1, 2,
2, 2, 4, 3, 5, 3, 8, 1, 3, 3, 3, 1, 2, 2, 2, 4, 7, 1, -2147483648, 1, 8, 4, 3)

scala> val iat = IntArrayTile(input, 9, 4)  // 9 and 4 here specify columns and rows
iat: geotrellis.raster.IntArrayTile = IntArrayTile([I@278434d0,9,4)

// The renderAscii method is mostly useful when you're working with small tiles
// which can be taken in at a glance.
scala> iat.renderAscii(AsciiArtEncoder.Palette.STIPLED)
res0: String =
```

```
scala> val focalNeighborhood = Square(1)  // a 3x3 square neighborhood
focalNeighborhood: geotrellis.raster.op.focal.Square =
 O  O  O
 O  O  O
 O  O  O

scala> val meanTile = iat.focalMean(focalNeighborhood)
meanTile: geotrellis.raster.Tile = DoubleArrayTile([D@7e31c125,9,4)

scala> meanTile.getDouble(0, 0)  // Should equal (1 + 7 + 9) / 3
res1: Double = 5.666666666666667
```

Ready? Setup a GeoTrellis development environment.

3.1 Changelog

3.1.1 2.0.0

API Changes

- `geotrellis.spark`

  - **Deprecation:** `geotrellis.slick` is being deprecated and will likely be moved to an external repo

  - **Change:** `geotrellis.slick.Projected` has been moved to `geotrellis.vector.Projected`

  - **Change:** The length of the key (the space-filling curve index or address) used for layer reading and writing has been extended from a fixed length of 8 bytes to an arbitrary length. This change affects not only the `geotrellis.spark` package, but all backends (excluding `geotrellis.geowave` and `geotrellis.geomesa`).

  - **New:** All focal operations now except an optional `partitioner` parameter.

  - **New:** `BufferTiless apply` methods and the `bufferTiles` methods now except an optional `partitioner` parameter.

  - **Change:** Reprojection has improved performance due to one less shuffle stage and lower memory usage.

  `TileRDDReproject` loses dependency on `TileReprojectMethods` in favor of `RasterRegionReproject` - **New:** CollectionLayerReader now has an SPI interface. - **New:** `ZoomResample` can now be used on `MultibandTileLayerRDD`s. - **New:** A `Partitioner` can be specified in the `reproject` methods of `TileLayerRDD`. - **New:** Compression `level` of GeoTiffs can be specified in the `DeflateCompression` constructor. - **Change::** The Ascii draw methods are now method extensions of `Tile`. - **Change:** Replace *geotrellis.util.Functor* with *cats.Functor*. - **Change:** Specifying the `maxTileSize` for a COGLayer that's to be written is now done via `COGLayerWriter.Options`

  which can be passed directly to the `write` methods.

  - **New:** The `resampleMethod` parameter has been added to `COGLayerWriter.Options`.

  - **Change:** Specifying the `compression` for a COGLayer that's to be written is now done via `COGLayerWriter.Options` which can be passed directly to the `write` methods.

  - **New:** A new type called `LayerType` has been created to help identify the nature of a layer (either Avro or COG).

- **New:** `LayerHeaders` now have an additional parameter: `layerType`.

- **Change:** The attribute name for `COGLayerStorageMetadata` is now `metadata` instead of `cog_metadata`.

- **New:** `AttributeStore` now has four new methods: `layerType`, `isCOGLayer`, `readCOGLayerAttributes`, and `writeCOGLayerAttributes`.

- **New:** Kryo serialization of geometry now uses a binary format to reduce shuffle block size.

- **Change:** Scalaz streams were replaced by fs2 streams.

- **Change:** Refactored HBaseInstance, now accepts a plain Hadoop Configuration object.

- **Change:** Refactored CassandraInstance, now accepts a getCluster function.

- **Change:** Use pureconfig to handle all work with configuration files.

- **Change:** Replace *geotrellis.util.Functor* with *cats.Functor*.

- **Remove:** `LayerUpdater` with its functionality covered by `LayerWriter` <https://github.com/locationtech/geotrellis/pull/2663>__

- **New:** Alter *geotrellis.spark.stitch.StitchRDDMethods* to allow *RDD[(K, V)]* to be stitched when not all tiles are of the same dimension.

- **Change:** Change *TilerMethods.tileToLayout* functions that accept *TileLayerMetadata* as an argument to return *RDD[(K, V)] with Metadata[M]* instead of *RDD[(K, V)]*.

- **New:** Introduce `Pyramid` class to provide a convenience wrapper for building raster pyramids.

- **Change:** Expose `attributeStore` parameter to LayerReader interface.

- **Change:** Added exponential backoffs in `S3RDDReader`.

- **Change:** Changed SinglebandGeoTiff and MultibandGeoTiff crop function behaviour to work properly with cases when extent to crop by doesn't intersect tiff extent.

- **Change:** All classes and objects in the `geowave` package now use the spelling: GeoWave in their names.

- **New:** Both `COGValueReader` and `OverzoomingCOGValueReader` now have the `readSubsetBands` method which allows users to read in a select number of bands in any order they choose.

- **New:** `COGLayerReader` now has the `readSubsetBands` and `querySubsetBands` methods which allow users to read in layers with the desired bands in the order they choose.

- **Change:** `COGLayerReader.baseRead` has been removed and has been replaced with `COGLayerReader.baseReadAllBands` and `COGLayerReader.baseReadSubsetBands`.

- **New:** `KeyBounds` now has the `rekey` method that will rekey the bounds from a source layout to a target layout.

- **Change:** The `TileLayerMetadata.fromRdd` method has been renamed to `TileLayerMetadata.fromRDD`.

- **Change:** The `KeyBounds.fromRdd` method has been renamed to `KeyBounds.fromRDD`.

- `geotrellis.raster`

  - **Change:** Removed `decompress` option from *GeoTiffReader* functions.

  - **New:** Kryo serialization of geometry now uses a binary format to reduce shuffle block size.

  - **Change:** Scalaz streams were replaced by fs2 streams.

- **New:** `GeoTiffMultibandTile` now has another `crop` method that takes a `GridBounds` and an `Array[Int]` that represents the band indices.

- **New:** `GeoTiff[MultibandTile]` can be written with `BandInterleave`, only `PixelInterleave` previously supported. <https://github.com/locationtech/geotrellis/pull/2767>

- **New:** `MultibandTile` now has a new method, `cropBands` that takes an Array of band indices and returns a cropped `MultibandTile` with the chosen bands.

- `geotrellis.spark-etl`

  - **Change**: Package is deprecated since GeoTrellis 2.0.

  - **Change: Input.maxTileSize is 256 by default to correspond HadoopGeoTiffRDD default behaviour.** Added `Input.partitionBytes` and it is set to `134217728` by default to correspond `HadoopGeoTiffRDD` default behaviour. Added `Output.bufferSize` option to set up a custom buffer size for the buffered reprojection.

Fixes

- StreamingHistogram.binCount now returns non-zero counts

- HilbertSpatialKeyIndex index offset - **Note:** Existing spatial layers using Hilbert index will need to be updated, see PR for directions.

- Fixed `CastException` that sometimes occured when reading cached attributes.

- Uncompressed GeoTiffMultibandTiles will now convert to the correct CellType.

- Calculating the Slope of a `Tile` when `targetCell` is `Data` will now produce the correct result.

- Introduce new hooks into AttributeStore API to allow for better performance in certain queries against catalogs with many layers

- `GeoTiffReader` can now read tiffs that are missing the `NewSubfileType` tag.

- Pyramiding code will once again respect resampling method and will now actually reduce shuffle volume by resampling tiles on map side of pyramid operation

- Uncompressed GeoTiffMultibandTiles will now convert to the correct CellType.

- COGLayer attributes can be accessed via the various read attribute methods in `AttributeStore` (ie `readMetadata`, `readHeader`, etc)

- The regex used to match files for the `HadoopLayerAttributeStore` and `FileLayerAttributeStore` has been expanded to include more characters.

- `HadoopAttributeStore.availableAttributes` has been fixed so that it'll now list all attribute files.

- Allow for simple features to be generated with a specified or random id with geometry stored in the standard field, "the_geom"

- Use a new Amazon SDK API to remove deprecation warnings.

- Fixed a bug in incorrect metadata fetch by COGLayerReaders that could lead to an incorrect data querying.

- Cropping RDDs with clamp=false now produces correct result.

- Fixed tiff reads in case RowsPerStrip tiff tag is not defined.

- Change aspect result to azimuth, i.e. start from due north and be clockwise.

- COG overviews generated in the `COGLayer.fromLayerRDD` method will now use the passed in `ResampleMethod`.

- Reading a GeoTiff with `streaming` will now work with files that are larger than `java.lang.Integer.MAX_VALUE`.

- `GeoTiffMultibandTile.crop` will now work with GeoTiffs that have tiled segments and band interleave.

- `GeoTiffMultibandTile.crop` will now return `ArrayMultibandTile`(s) with the correct number of bands.

- Imroved performance of `COGValueReader.readSubsetBands` when reading from S3.

1.2.1

2018 Jan 3

Fixes

- GeoTiffSegmentLayout.getIntersectingSegments bounds checking
- Fix for area of vectorizer that can throw topology exceptions
- Fix Tile.flipHorizontal for floating point tiles

3.1.2 1.2.0

2017 Nov 6

This release cycle saw a regular contributor Simeon Fitch elevated to official *Committer* status within GeoTrellis.

The team would like to thank him, along with our newest contributors Aaron Santos, Austin Heyne, Christos Charmatzis, Jessica Austin, and @mteldridge for helping make this release possible.

API Changes

- `geotrellis.raster`

  - **Deprecation:** `GridBounds.size` in favor of `GridBounds.sizeLong`.

  - **Deprecation:** `GridBounds.coords` in favor of `GridBounds.coordsIter`.

  - **New:** `GridBounds.offset` and `GridBounds.buffer` for creating a modified `GridBounds` from an existing one.

  - **New:** `ColorRamps.greyscale:  Int => ColorRamp`, which will generate a ramp when given some number of stops.

  - **New:** `ConstantTile.fromBytes` to create any type of `ConstantTile` from an `Array[Byte]`.

  - **New:** `Tile.rotate90:  Int => Tile`, `Tile.flipVertical:  Tile` and `Tile.flipHorizontal:  Tile`.

- `geotrellis.vector`

  - **New:** `Geometry.isEmpty:  Boolean`. This incurs much less overhead than previous ways of determining emptiness.

  - **New:** `Line.head` and `Line.last` for **efficiently** grabbing the first or last `Point` in the `Line`.

- `geotrellis.spark`

- **Deprecation:** The `LayerUpdater` trait hierarchy. Use `LayerWriter.update` or `LayerWriter.overwrite` instead.

- **Deprecation:** Every cache provided by `geotrellis.spark.util.cache`. These will be removed in favor of a pluggable cache in 2.0.

- **New:** `SpatialKey.extent: LayoutDefinition => Extent`

- **New:** `ValueReader.attributeStore: AttributeStore`

- **New:** `TileLayerRDD.toSpatialReduce: ((V, V) => V) => TileLayerRDD[SpatialKey]` for smarter folding of 3D tile layers into 2D tile layers.

- The often-used `apply` method overloads in `MapKeyTransform` have been given more descriptive aliases.

- **Change:** Querying a layer will now produce a result whose metadata will have an `Extent` and `KeyBounds` of the queried region and not of the whole layer.

- `geotrellis.vectortile` (experimental)

  - **New:** `VectorTile.toGeoJson` and `VectorTile.toIterable`.

 - Library simplified by assuming the codec backend will always be Protobuf.

New Features

Finally, the full marriage of the `vector`, `raster`, and `spark` packages! You can now transform an `RDD[Geometry]` into a writable GeoTrellis layer of `(SpatialKey, Tile)`!

```scala
val geoms: RDD[Geometry] = ...
val celltype: CellType = ...
val layout: LayoutDefinition = ...
val value: Double = ...  /* Value to fill the intersecting pixels with */

val layer: RDD[(SpatialKey, Tile)] with Metadata[LayoutDefinition] =
  geoms.rasterize(value, celltype, layout)
```

In a similar vein to the above, you can now transform an arbitrarily large collection of Geometries into a proper GeoTrellis layer, where the sections of each Geometry are clipped to fit inside their enclosing Extents.
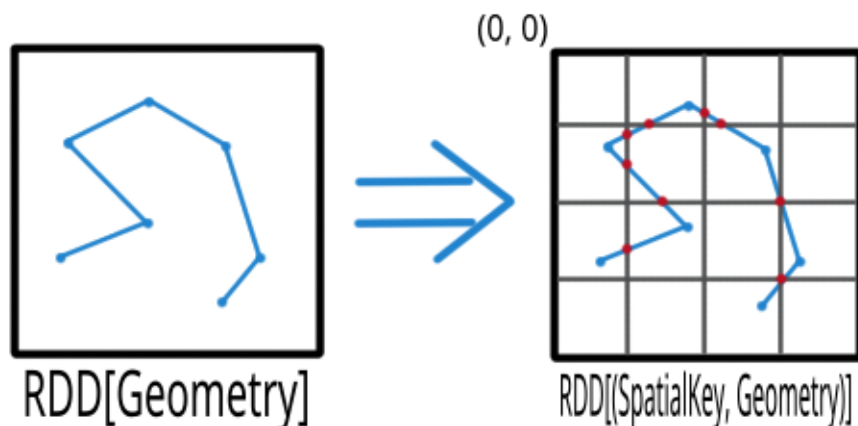
Here we can see a large `Line` being clipped into nine sublines. It's one method call:

```
import geotrellis.spark._

val layout: LayoutDefinition = ...   /* The definition of your grid */
val geoms: RDD[Geometry] = ...       /* Result of some previous work */

/* There are likely many clipped Geometries per SpatialKey... */
val layer: RDD[(SpatialKey, Geometry)] = geoms.clipToGrid(layout)

/* ... so we can group them! */
val grouped: RDD[(SpatialKey, Iterable[Geometry])] = layer.groupByKey
```

If clipping on the Extent boundaries is not what you want, there are ways to customize this. See the ClipToGrid entry in our Scaladocs.

A Viewshed shows "visibility" from some set vantage point, given an Elevation raster. Prior to GeoTrellis 1.2 this was possible at the individual `Tile` level but not the Layer (`RDD`) level. Now it is.

First, we need to think about the `Viewpoint` type:

```
import geotrellis.spark.viewshed._

val point: Viewpoint(
  x = ...,                  // some coordinate.
  y = ...,                  // some coordinate.
  viewHeight = 4000,        // 4 kilometres above the surface.
  angle = Math.PI / 2,      // direction that the "camera" faces (in radians). 0 ==
→east.
  fieldOfView = Math.PI / 2,  // angular width of the "view port".
  altitude = 0              // the height of points you're interested in seeing.
)
```

In other words:

- x, y, viewHeight: where are we?

- angle: what direction are we looking?

- fieldOfView: how wide are we looking?

- altitude: how high/low is the "target" of our viewing?

Given a `Seq[Viewpoint]` (the algorithm supports multiple simultaneous view points), we can do:

```
// Recall this common alias:
//   type TileLayerRDD[K] = RDD[(K, Tile)] with Metadata[TileLayerMetadata[K]]

val layer: TileLayerRDD[SpatialKey] = ...  /* Result of previous work */

val viewshed: TileLayerRDD[SpatialKey] = layer.viewshed(Seq(point))
```

We use *Euclidean Distance* to render a collection of points into a heatmap of proximities of some area. Say, of two roads crossing:

Prior to GeoTrellis 1.2, this was possible at the individual `Tile` level but not the Layer (`RDD`) level. Now it is.

```
/* Result of previous work. Potentially millions of points per SpatialKey. */
val points: RDD[(SpatialKey, Array[Coordinate])] = ...
val layout: LayoutDefinition = ...   /* The definition of your grid */

val layer: RDD[(SpatialKey, Tile)] = points.euclideanDistance(layout)
```
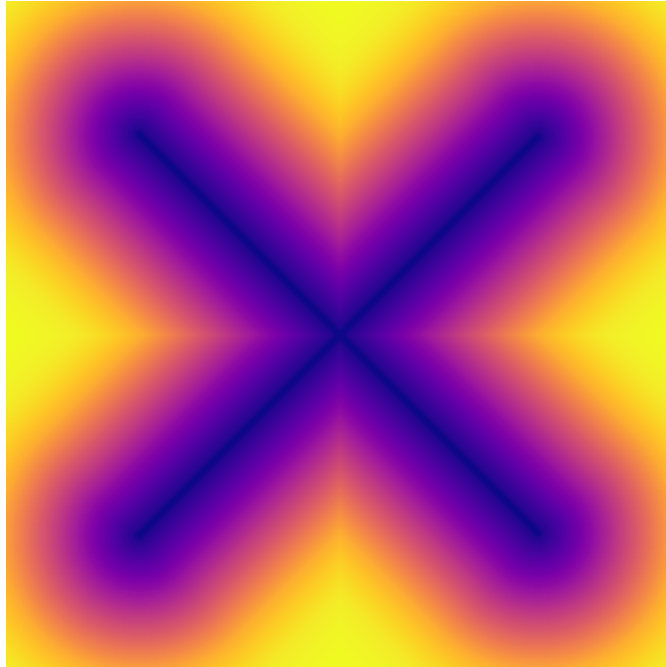
The following was possible prior to GeoTrellis 1.2:

```scala
val layer: TileLayerRDD[SpatialKey] = ...
val polygon: Polgyon = ...

/* The maximum value within some Polygon overlaid on a Tile layer */
val summary: Double = layer.polygonalMaxDouble(polygon)
```

The above is also now possible for layers keyed by `SpaceTimeKey` to form a "time series":

```scala
val layer: TileLayerRDD[SpaceTimeKey] = ...
val polygon: MultiPolygon = ...

/* The maximum value within some Polygonal area at each time slice */
val summary: Map[ZonedDateTime, Double] = layer.maxSeries(polygon)
```

A GeoTrellis `ValueReader` connects to some layer catalog and lets you read individual values (usually Tiles):

```scala
import geotrellis.spark.io.s3._

val store: AttributeStore = ...
val reader: Reader[SpatialKey, Tile] = S3ValueReader(store).reader(LayerId("my-catalog
→", 10))

val tile: Tile = reader.read(SpatialKey(10, 10))
```

However `.reader` is limited to zoom levels that actually exist for the given layer. Now you can use `.overzoomingReader` to go as deep as you like:

```scala
import geotrellis.raster.resample._

val reader: Reader[SpatialKey, Tile] =
  S3ValueReader(store).overzoomingReader(LayerId("my-catalog", 20), Average)
```

(continues on next page)

```scala
val tile: Tile = reader.read(SpatialKey(1000, 1000))
```

Have you ever wanted to "redraw" a grid over an established GeoTrellis layer? Say, this 16-tile Layer into a 4-tile one, both of 1024x1024 total pixels:
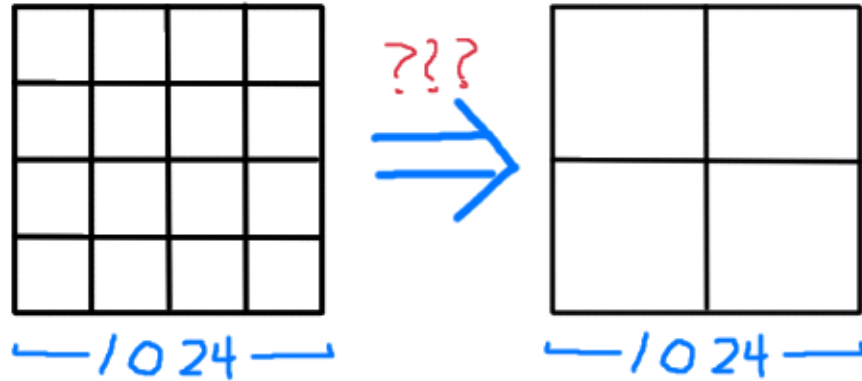
Prior to GeoTrellis 1.2, there was no official way to do this. Now you can use `.regrid`:

```scala
/* The result of some previous work. Say each Tile is 256x256. */
val layer: TileLayerRDD[SpatialKey] = ...

/* "Recut" the tiles so that each one is now 512x512.
 * No pixels are gained or lost, save some NODATA on the bottom
 * and right edges that may appear for padding purposes.
 */
val regridded: TileLayerRDD[SpatialKey] = layer.regrid(512)
```

You can also regrid to non-rectangular sizes:

```scala
val regridded: TileLayerRDD[SpatialKey] = layer.regrid(tileCols = 100, tileRows = 300)
```

It's common to find a subset of Tiles in a layer that are touched by some given `Polygon`:

```scala
val poly: Polygon = ???

val rdd: TileLayerRDD[SpatialKey] =
 layerReader
    .query[SpatialKey, Tile, TileLayerMetadata[SpatialKey]](Layer("name", zoom))
    .where(Intersects(poly))
    .result
```

Now you can perform this same operation with `Line`, `MultiLine`, and even `(Polygon, CRS)` to ensure that your Layer and Geometry always exist in the same projection.

Sometimes you just want to visualize a `Tile` without going through the song-and-dance of rendering it to a `.png`. The existing `Tile.asciiDraw` method *kind of* does that, except its output is all in numbers.

The new `Tile.renderAscii: Palette => String` method fulfills your heart's desire:

```scala
import geotrellis.raster._
import geotrellis.raster.io.geotiff._
import geotrellis.raster.render.ascii._

val tile: Tile = SinglebandGeoTiff("path/to/tiff.tiff").tile

// println(tile.renderAscii())  // the default
println(tile.renderAscii(AsciiArtEncoder.Palette.STIPLED))
```

Gorgious.

By adding some additional configuration, you can now use our HDFS Layer Backend to read and write GeoTrellis layers to Microsoft Azure's blob storage.

It's now possible to customize how our S3 backend communicates with S3.

GeoTrellis uses the Java Topology Suite for its vector processing. By default, JTS uses a "floating" PrecisionModel. When writing code that needs to be numerically robust, this default can lead to Topology Exceptions.

You can now use Typesafe Config to configure this to your application's needs. See here for the specifics.

- Kerberos authentication is available for properly configured Accumulo clusters
- Polygonal Summaries for MultibandTiles
- Filter GeoTiffRDDs by Geometry
- Can create ValueReaders via URIs through LayerProvides classes
- Can read/write GeoTiffs with Sinusoidal projections
- Can Resample via Sum operation

Fixes

- Negative grid bounds bug
- getSignedByteArray BugFix - fixes certain read problems
- Allow Merge Queue To Handle Larger Inputs
- Generate Windows That Conform To GeoTiff Segments
- Removed inefficient LayerFilter check
- Fixed issue with S3 URI not having a key prefix
- Improve S3 makePath function
- Fix S3GeoTiffRDD behavior with some options.
- Allow Contains(Point) query for temporal rdds
- Haversine formula fix
- Use Scaffeine instead of LRU cache in HadoopValueReader
- Fix GeoTiffInfo serialization issues
- Estimate partitions number based on GeoTiff segments
- Estimate partitions number basing on a desired partition size
- Pyramid operation preserves partitioning
- Don't constrain GridBounds size to IntMax x IntMax
- 4-Connected Line Drawing
- Added requirement for CRS implementations to provide a readable toString representation.
- Allow rasterizer to store Z value at double precision
- Changed scheme path file from /User -> current working dir

- Fix CRS parser and proj4 cea projection support

3.1.3 1.1.0

Features

- Spark Enabled Cost Distance
- Conforming Delaunay Triangulation
- Added a fractional-pixel rasterizer for polygons and multipolygons
- Added collections API mapalgebra local and masking functions
- Added withDefaultNoData method for CellTypes
- Moved Spark TestEnvironment to spark-testkit subproject for usage outside of GeoTrellis
- Add convenience overloads to GeoTiff companion object
- Added matplotlib's Magma, Inferno, Plasma, and Viridis color ramps
- Added library of land use classification color maps.
- Add MGRS encode/decode support to proj4
- Rasters write support to HDFS / S3
- Added Range-based reading of HTTP resources
- Improved the WKT parser that powers the WKT.getEpsgCode method
- Updated the geotrellis-geowave subproject to GeoWave 0.9.3
- Updated the geotrellis-geomesa subproject to GeoMesa 1.2.7
- Use H3 rather than Next Fit when building S3 partitions from paths
- Added delimiter option to S3InputFormat and S3GeoTiffRDD.
- Signed S3 Streaming for GeoTiff reader (HTTP with GET instead of HEAD request)
- Relaxed constraints to improve layer deletion capabilities
- Allow HadoopGeoTiffRDD and S3GeoTiffRDD to maintain additional key information such as file name
- Added API sugar for simplying construction of AvroRecordCodec
- Make compression optional for Avro encoding and decoding
- Optimization to avoid unspecialized Function3 usage in Hillshade, Slope and Aspect
- Updated multiple dependencies
- Upgraded ScalaPB version for VectorTile
- Added Avro codecs for ProjectedExtent and TemporalProjectedExtent and ConstantTile types
- Repartition in ETL when re-tiling increases layer resolution
- In GeoTiff reader, compute CellSize from TIFF tags
- Improved apply methods for constructing S3RangeReader
- Reorganized handling of CellType.name
- Documentation improvements, including porting the docts to reStructuredText

- Added top-level "Sinusoidal" CRS, commonly used with MODIS
- Added conditional to key bounds decomposition to detect full bounds query in Acccumulo.
- Support for the ability to specify output CRS via proj4 string.

Fixes

- Fixed issues that made GeoTiff streaming off of S3 slow and broken
- Give a better error message for CRS write failures
- Fix clipping logic during polygon layer query
- Fixed type for CRS authority in NAD83
- Moved JsonFormats for CellSize and CellType to their proper place
- Fixed polygon rasterization for complex polygon test cases
- Fixed issue with FileLayerDeleter
- Fixed issue with logger serialization
- Fixed bug in renderPng that caused incorrect rendering of non-floating-point rasters
- Don't allow illegal TileLayouts
- Prevent error from happening during Pyramiding
- Ensure tile columns are not zero when rounding
- Fixed malformed XML error that was happening after failed S3 ingest
- Fix issue with S3LayerDeleter deleting files outside of layer
- Fix TemporalProjectedExtentCodec to handling proj4 strings when CRS isn't available
- Fixed layoutForZoom to allow 0 zoom level
- Fixed MapKeyTransform to deal with points north and west of extent
- Fixed GeoTiff reading for GeoTiffs with model tie point and PixelIsPoint
- Fixed issue with reading tiny (4 pixel or less) GeoTiffs
- Fix usage of IntCachedColorMap in Indexed PNG encoding
- Ensure keyspace exists in CassandraRDDWriter
- Resolved repartitioning issue with HadoopGeoTiffRDD
- Fixed schema for intConstantTileCodec
- In HadoopAttributeStore, get absolute path for attributePath
- In AccumuloLayerDeleter, close batch deleter
- S3InputFormat - bucket names support period and dashes
- Fix TMS scheme min zoom level
- S3AttributeStore now handles ending slashes in prefix.
- Cell type NoData logic for unsigned byte / short not working properly
- CellSize values should not be truncated to integer when parsing from Json.
- Fixes to GeoTiff writing with original LZW compression.

- In ArrayTile.convert, debug instead of warn against floating point data loss.

- Fixes incorrect metadata update in a per-tile reprojection case

- Fix issue with duplicate tiles being read for File and Cassandra backends

- Move to a different Json Schema validator

- S3InputFormat does not filter according to extensions when partitionCount is used

- In S3GeoTiffReader, partitionBytes has no effect if maxTileSize is set

- Fixes typos with rasterizer extension methods

- Fix writing multiband GeoTiff with compression

- Fixed issue with BigTiff vs non-BigTiff offset value packing

API Changes

While we are trying to stick strictly to SemVer, there are slight API changes in this release. We felt that while this does break SemVer in the strictest sense, the change were not enough to warrant a 2.0 release. Our hope is in the future to be more cognizant of API changes for future releases.

- Made EPSG capatilization consistent in method names:

  - In `geotrellis.proj4.CRS`, changed `getEPSGCode` to `getEpsgCode`

  - In `geotrellis.proj4.io.wkt.WKT`, changed `fromEPSGCode` to `fromEpsgCode` and `getEPSGCode` to `getEpsgCode`

- Changed some internal but publicly visible classes dealing with GeoTiff reading

  - Changed `size` to `length` in `ArraySegmentBytes`

  - Replaced `foreach` on SegmentBytes with `getSegments`, which the caller can iterate over themselves

  - Changed `getDecompressedBytes` to `decompressGeoTiffSegment`

- **Changed some interal but publicly visible implicit classes and read methods around TiffTagReader**

  - Added as an implicit parameter to multiple locations, most publicly in *TiffTagReader.read(byteReader: ByteReader, tagsStartPosition: Long)(implicit ttos: TiffTagOffsetSize)*. Also changed that method from being generic to always taking a *Long* offset.

- Moved some misplaced implicit JsonFormats

  - Moved `CellTypeFormat` and `CellSizeFormat` from `` geotrellis.spark.etl.config.json`` in the `spark-etl` subproject to `geotrellis.raster.io.json.Implicits` in the `raster` subproject.

- Changed LazyLogger from the com.typesafe.scalalogging version to our own version

  - This shouldn't break any code, but technically is an API change.

3.1.4 1.0.0

Major Features

- GeoTools support

  - Add Support for GeoTools SimpleFeature #1495

  - Conversions between GeoTools GridCoverage2D and GeoTrellis Raster types #1502

- Streaming GeoTiff reading #1559
- Windowed GeoTiff ingests into GeoTrellis layers, allowing users to ingest large GeoTiffs #1763
    - Reading TiffTags via MappedByteBuffer #1541
    - Cropped Windowed GeoTiff Reading #1559
    - Added documentation to the GeoTiff* files #1560
    - Windowed GeoTiff Docs #1616
- GeoWave Raster/Vector support (experimental)
    - Create GeoWave Subproject #1542
    - Add vector capabilities to GeoWave support #1581
    - Fix GeoWave Tests #1665
- GeoMesa Vector support (experimental)
    - Create GeoMesa suproject #1621
- Moved to a JSON-configuration ETL process
    - ETL Refactor #1553
    - ETL Improvements and other issues fixes #1647
- Vector Tile reading and writing, file-based and as GeoTrellis layers in RDDs. #1622
- File Backends
    - Cassandra support #1452
    - HBase support #1586
- Collections API #1606
    - Collections polygonal summary functions #1614
    - Collections mapalgebra focal functions #1619
- Add `TileFeature` Type #1429
- Added Focal calculation target type #1601
- Triangulation
    - Voronoi diagrams and Delaunay triangulations #1545, #1699
    - Conforming Delaunay Triangulation #1848
- Euclidean distance tiles #1552
- Spark, Scala and Java version version support
    - Move to Spark 2; Scala 2.10 deprecation #1628
    - Java 7 deprecation #1640
- Color correction features:
    - Histogram Equalization #1668
    - Sigmoidal Contrast #1681
    - Histogram matching #1769

- `CollectNeighbors` feature, allowing users to group arbitrary values by the neighbor keys according to their SpatialComponent #1860

- **Documentation:** We moved to ReadTheDocs, and put a lot of work into making our docs significantly better. See them here.

Minor Additions

- Documentation improvements

  - Quickstart

  - Examples

    * Added example for translating from `SpaceTimeKey` to `SpatialKey` #1549

    * doc-examples subproject; example for tiling to GeoTiff #1564

    * Added example for focal operation on multiband layer. #1577

    * Projections, Extents, and Layout Definitions doc #1608

    * Added example of turning a list of features into GeoJson #1609

    * Example: `ShardingKeyIndex[K]` #1633

    * Example: `VoxelKey` #1639

- Introduce ADR concept

  - ADR: HDFS Raster Layers #1582

  - [ADR] Readers / Writers Multithreading #1613

- Fixes

  - Fixed some markdown docs #1625

  - `parseGeoJson` lives in geotrellis.vector.io #1649

- Parallelize reads for S3, File, and Cassandra backends #1607

- Kernel Density in Spark

- k-Nearest Neighbors

- Updated slick

- Added GeoTiff read/write support of TIFFTAG_PHOTOMETRIC via `GeoTiffOptions`. #1667

- Added ability to read/write color tables for GeoTIFFs encoded with palette photometric interpretation #1802

- Added `ColorMap` to String conversion #1512

- Add split by cols/rows to SplitMethods #1538

- Improved HDFS support #1556

- Added Vector Join operation for Spark #1610

- Added Histograms Over Fractions of RDDs of Tiles #1692

- Add `interpretAs` and `withNoData` methods to Tile #1702

- Changed GeoTiff reader to handle BigTiff #1753

- Added `BreakMap` for reclassification based on range values. #1760

- Allow custom save actions on ETL #1764

- Multiband histogram methods [#1784](#)

- `DelayedConvert` feature, allowing users to delay conversions on tiles until a map or combine operation, so that tiles are not iterated over unnecessarily [#1797](#)

- Add convenience overloads to GeoTiff companion object [#1840](#)

Fixes / Optimizations

- Fixed GeoTiff bug in reading NoData value if len = 4 [#1490](#)

- Add detail to avro exception message [#1505](#)

- Fix: The toSpatial Method gives metadata of type TileLayerMetadata[SpaceTimeKey]

    - Custom `Functor` Typeclass [#1643](#)

- Allow `Intersects(polygon:   Polygon)` in layer query [#1644](#)

- Optimize `ColorMap` [#1648](#)

- Make regex for s3 URLs handle s3/s3a/s3n [#1652](#)

- Fixed metadata handling on surface calculation for tile layer RDDs [#1684](#)

- Fixed reading GeoJson with 3d values [#1704](#)

- Fix to Bicubic Interpolation [#1708](#)

- Fixed: Band tags with values of length > 31 have additional white space added to them [#1756](#)

- Fixed NoData bug in tile merging logic [#1793](#)

- Fixed Non-Point Pixel + Partial Cell Rasterizer Bug [#1804](#)

New Committers

- metasim

- lokifacio

- aeffrig

- jpolchlo

- jbouffard

- vsimko

- longcmu

- miafg

3.1.5 0.10.3

- PR [#1611](#) Any `RDD` of `Tiles` can utilize Polygonal Summary methods. (@fosskers)

- PR [#1573](#) New `foreach` for `MultibandTile` which maps over each band at once. (@hjaekel)

- PR [#1600](#) New `mapBands` method to map more cleanly over the bands of a `MultibandTile`.

1.

3.1.6 0.10.2

- PR #1561 Fix to polygon sequence union, account that it can result in NoResult. (1)
- PR #1585 Removed warnings; add proper subtyping to GetComponent and SetComponent identity implicits; fix jai travis breakage. (1)
- PR #1569 Moved RDDLayoutMergeMethods functionality to object. (1)
- PR #1494 Add ETL option to specify upper zoom limit for raster layer ingestion (@mbertrand)
- PR #1571 Fix scallop upgrade issue in spark-etl (@pomadchin)
- PR #1543 Fix to Hadoop LayerMover (@pomadchin)

Special thanks to new contributor @mbertrand!

3.1.7 0.10.1

- PR #1451 Optimize reading from compressed Bit geotiffs (@shiraeeshi)
- PR #1454 Fix issues with IDW interpolation (@lokifacio)
- PR #1457 Store FastMapHistogram counts as longs (@jpolchlo)
- PR #1460 Fixes to user defined float/double CellType parsing (@echeipesh)
- PR #1461 Pass resampling method argument to merge in CutTiles (1)
- PR #1466 Handle Special Characters in proj4j (@jamesmcclain)
- PR #1468 Fix nodata values in codecs (@shiraeeshi)
- PR #1472 Fix typo in MultibandIngest.scala (@timothymschier)
- PR #1478 Fix month and year calculations (@shiraeeshi)
- PR #1483 Fix Rasterizer Bug (@jamesmcclain)
- PR #1485 Upgrade dependencies as part of our LocationTech CQ process (1)
- PR #1487 Handle entire layers of NODATA (@fosskers)
- PR #1493 Added support for int32raw cell types in CellType.fromString (@jpolchlo)
- PR #1496 Update slick (@adamkozuch, @moradology)
- PR #1498 Add ability to specify number of streaming buckets (@moradology)
- PR #1500 Add logic to ensure use of minval/avoid repetition of breaks (@moradology)
- PR #1501 SparkContext temporal GeoTiff format args (@echeipesh)
- PR #1510 Remove dep on cellType when specifying layoutExtent (@fosskers)
- PR #1529 LayerUpdater fix (@pomadchin)

Special thanks to new contributors @fosskers, @adamkozuch, @jpolchlo, @shiraeeshi, @lokifacio!

3.1.8 0.10.0

The long awaited GeoTrellis 0.10 release is here!

It's been a while since the 0.9 release of GeoTrellis, and there are many significant changes and improvements in this release. GeoTrellis has become an expansive suite of modular components that aide users in the building of geospatial

application in Scala, and as always we've focused specifically on high performance and distributed computing. This is the first official release that supports working with Apache Spark, and we are very pleased with the results that have come out of the decision to support Spark as our main distributed processing engine. Those of you who have been tuned in for a while know we started with a custom built processing engine based on Akka actors; this original execution engine still exists in 0.10 but is in a deprecated state in the geotrellis-engine subproject. Along with upgrading GeoTrellis to support Spark and handle arbitrarily-sized raster data sets, we've been making improvements and additions to core functionality, including adding vector and projection support.

It's been long enough that release notes, stating what has changed since 0.9, would be quite unwieldy. Instead I put together a list of features that GeoTrellis 0.10 supports. This is included in the README on the GeoTrellis Github, but I will put them here as well. It is organized by subproject, with more basic and core subprojects higher in the list, and the subprojects that rely on that core functionality later in the list, along with a high level description of each subproject.

geotrellis-proj4

- Represent a Coordinate Reference System (CRS) based on Ellipsoid, Datum, and Projection.

- Translate CRSs to and from proj4 string representations.

- Lookup CRS's based on EPSG and other codes.

- Transform `(x, y)` coordinates from one CRS to another.

geotrellis-vector

- Provides a scala idiomatic wrapper around JTS types: Point, Line (LineString in JTS), Polygon, MultiPoint, MultiLine (MultiLineString in JTS), MultiPolygon, GeometryCollection

- Methods for geometric operations supported in JTS, with results that provide a type-safe way to match over possible results of geometries.

- Provides a Feature type that is the composition of a geometry and a generic data type.

- Read and write geometries and features to and from GeoJSON.

- Read and write geometries to and from WKT and WKB.

- Reproject geometries between two CRSs.

- Geometric operations: Convex Hull, Densification, Simplification

- Perform Kriging interpolation on point values.

- Perform affine transformations of geometries

geotrellis-vector-testkit

- GeometryBuilder for building test geometries

- GeometryMatcher for scalatest unit tests, which aides in testing equality in geometries with an optional threshold.

geotrellis-raster

- Provides types to represent single- and multi-band rasters, supporting Bit, Byte, UByte, Short, UShort, Int, Float, and Double data, with either a constant NoData value (which improves performance) or a user defined NoData value.

- Treat a tile as a collection of values, by calling "map" and "foreach", along with floating point valued versions of those methods (separated out for performance).

- Combine raster data in generic ways.

- Render rasters via color ramps and color maps to PNG and JPG images.

- Read GeoTiffs with DEFLATE, LZW, and PackBits compression, including horizontal and floating point prediction for LZW and DEFLATE.

- Write GeoTiffs with DEFLATE or no compression.

- Reproject rasters from one CRS to another.

- Resample of raster data.

- Mask and Crop rasters.

- Split rasters into smaller tiles, and stitch tiles into larger rasters.

- Derive histograms from rasters in order to represent the distribution of values and create quantile breaks.

- Local Map Algebra operations: Abs, Acos, Add, And, Asin, Atan, Atan2, Ceil, Cos, Cosh, Defined, Divide, Equal, Floor, Greater, GreaterOrEqual, InverseMask, Less, LessOrEqual, Log, Majority, Mask, Max, MaxN, Mean, Min, MinN, Minority, Multiply, Negate, Not, Or, Pow, Round, Sin, Sinh, Sqrt, Subtract, Tan, Tanh, Undefined, Unequal, Variance, Variety, Xor, If

- Focal Map Algebra operations: Hillshade, Aspect, Slope, Convolve, Conway's Game of Life, Max, Mean, Median, Mode, Min, MoransI, StandardDeviation, Sum

- Zonal Map Algebra operations: ZonalHistogram, ZonalPercentage

- Operations that summarize raster data intersecting polygons: Min, Mean, Max, Sum.

- Cost distance operation based on a set of starting points and a friction raster.

- Hydrology operations: Accumulation, Fill, and FlowDirection.

- Rasterization of geometries and the ability to iterate over cell values covered by geometries.

- Vectorization of raster data.

- Kriging Interpolation of point data into rasters.

- Viewshed operation.

- RegionGroup operation.

geotrellis-raster-testkit

- Build test raster data.

- Assert raster data matches Array data or other rasters in scalatest.

geotrellis-spark

- Generic way to represent key value RDDs as layers, where the key represents a coordinate in space based on some uniform grid layout, optionally with a temporal component.

- Represent spatial or spatiotemporal raster data as an RDD of raster tiles.

- Generic architecture for saving/loading layers RDD data and metadata to/from various backends, using Spark's IO API with Space Filling Curve indexing to optimize storage retrieval (support for Hilbert curve and Z order curve SFCs). HDFS and local file system are supported backends by default, S3 and Accumulo are supported backends by the `geotrellis-s3` and `geotrellis-accumulo` projects, respectively.

- Query architecture that allows for simple querying of layer data by spatial or spatiotemporal bounds.

- Perform map algebra operations on layers of raster data, including all supported Map Algebra operations mentioned in the geotrellis-raster feature list.

- Perform seamless reprojection on raster layers, using neighboring tile information in the reprojection to avoid unwanted NoData cells.

- Pyramid up layers through zoom levels using various resampling methods.

- Types to reason about tiled raster layouts in various CRS's and schemes.

- Perform operations on raster RDD layers: crop, filter, join, mask, merge, partition, pyramid, render, resample, split, stitch, and tile.

- Polygonal summary over raster layers: Min, Mean, Max, Sum.

- Save spatially keyed RDDs of byte arrays to z/x/y files into HDFS or the local file system. Useful for saving PNGs off for use as map layers in web maps or for accessing GeoTiffs through z/x/y tile coordinates.

- Utilities around creating spark contexts for applications using GeoTrellis, including a Kryo registrator that registers most types.

geotrellis-spark-testkit

- Utility code to create test RDDs of raster data.

- Matching methods to test equality of RDDs of raster data in scalatest unit tests.

geotrellis-accumulo

- Save and load layers to and from Accumulo. Query large layers efficiently using the layer query API.

geotrellis-cassandra

Save and load layers to and from Casandra. Query large layers efficiently using the layer query API.

geotrellis-s3

- Save/load raster layers to/from the local filesystem or HDFS using Spark's IO API.

- Save spatially keyed RDDs of byte arrays to z/x/y files in S3. Useful for saving PNGs off for use as map layers in web maps.

geotrellis-etl

- Parse command line options for input and output of ETL (Extract, Transform, and Load) applications

- Utility methods that make ETL applications easier for the user to build.

- Work with input rasters from the local file system, HDFS, or S3

- Reproject input rasters using a per-tile reproject or a seamless reprojection that takes into account neighboring tiles.

- Transform input rasters into layers based on a ZXY layout scheme

- Save layers into Accumulo, S3, HDFS or the local file system.

geotrellis-shapefile

- Read geometry and feature data from shapefiles into GeoTrellis types using GeoTools.

geotrellis-slick

- Save and load geometry and feature data to and from PostGIS using the slick scala database library.

- Perform PostGIS `ST_` operations in PostGIS through scala.

3.2 Contributing

We value all kinds of contributions from the community, not just actual code. Perhaps the easiest and yet one of the most valuable ways of helping us improve GeoTrellis is to ask questions, voice concerns or propose improvements on the Mailing List.

If you do like to contribute actual code in the form of bug fixes, new features or other patches this page gives you more info on how to do it.

3.2.1 Building GeoTrellis

1. Install SBT (the master branch is currently built with SBT 0.13.12).

2. Check out this repository.

3. Pick the branch corresponding to the version you are targeting

4. Run `sbt test` to compile the suite and run all tests.

3.2.2 Style Guide

We try to follow the Scala Style Guide as closely as possible, although you will see some variations throughout the codebase. When in doubt, follow that guide.

3.2.3 Git Branching Model

The GeoTrellis team follows the standard practice of using the `master` branch as main integration branch.

3.2.4 Git Commit Messages

We follow the 'imperative present tense' style for commit messages. (e.g. "Add new EnterpriseWidgetLoader instance")

3.2.5 Issue Tracking

If you find a bug and would like to report it please go there and create an issue. As always, if you need some help join us on Gitter to chat with a developer.

3.2.6 Pull Requests

If you'd like to submit a code contribution please fork GeoTrellis and send us pull request against the `master` branch. Like any other open source project, we might ask you to go through some iterations of discussion and refinement before merging.

As part of the Eclipse IP Due Diligence process, you'll need to do some extra work to contribute. This is part of the requirement for Eclipse Foundation projects (see this page in the Eclipse wiki You'll need to sign up for an Eclipse account **with the same email you commit to github with**. See the `Eclipse Contributor Agreement` text below. Also, you'll need to signoff on your commits, using the `git commit -s` flag. See https://help.github.com/articles/signing-tags-using-gpg/ for more info.

3.2.7 Eclipse Contributor Agreement (ECA)

Contributions to the project, no matter what kind, are always very welcome. Everyone who contributes code to GeoTrellis will be asked to sign the Eclipse Contributor Agreement. You can electronically sign the Eclipse Contributor Agreement here.

3.2.8 Editing these Docs

Contributions to these docs are welcome as well. To build them on your own machine, ensure that `sphinx` and `make` are installed.

Installing Dependencies

Ubuntu 16.04

```
> sudo apt-get install python-sphinx python-sphinx-rtd-theme
```

Arch Linux

```
> sudo pacman -S python-sphinx python-sphinx_rtd_theme
```

MacOS

`brew` doesn't supply the sphinx binaries, so use `pip` here.

Pip

```
> pip install sphinx sphinx_rtd_theme
```

Building the Docs

Assuming you've cloned the GeoTrellis repo, you can now build the docs yourself. Steps:

1. Navigate to the `docs/` directory

2. Run `make html`

3. View the docs in your browser by opening `_build/html/index.html`

Note: Changes you make will not be automatically applied; you will have to rebuild the docs yourself. Luckily the docs build in about a second.

File Structure

When adding or editing documentation, keep in mind the following file structure:

- `docs/tutorials/` contains simple beginner tutorials with concrete goals

- `docs/guide/` contains detailed explanations of GeoTrellis concepts

- `docs/architecture` contains in-depth discussion on GeoTrellis implementation details

3.3 Setup

Welcome to GeoTrellis, the Scala library for high-performance geographic data processing. Being a library, users import GeoTrellis and write their own Scala applications with it. This guide will help you get up and running with a basic GeoTrellis development environment.

3.3.1 Requirements

- Java 8. GeoTrellis code won't function with Java 7 or below. You can test your Java version by entering the following in a Linux or Mac terminal:

```
> javac -version
javac 1.8.0_102
```

You want to see `1.8` like above.

- Apache Spark 2. This is if you plan to run ingests (as shown in our ETL tutorial) or write a serious application. Otherwise, fetching Spark dependencies for playing with GeoTrellis is handled automatically, as shown in our Quick-Start Guide.

When running more involved applications, `spark-submit` should be on your `PATH`:

```
> which spark-submit
/bin/spark-submit
```

3.3.2 Using Scala

GeoTrellis is a Scala library, so naturally you must write your applications in Scala. If you're new to Scala, we recommend the following:

- The official Scala tutorials
- The Scala Cookbook as a handy language reference
- 99 Problems in Scala to develop basic skills in Functional Programming

3.3.3 GeoTrellis Project Template

The geotrellis-sbt-template repo provides a simple GeoTrellis project template. It can be used to experiment with GeoTrellis, or to write full applications. Get it with:

```
git clone https://github.com/geotrellis/geotrellis-sbt-template.git
```

You don't need `sbt` installed to write a GeoTrellis app, since this template includes an `sbt` bootstrap script. It is used like regular SBT, and comes with a few extra commands:

- Enter the SBT shell: `./sbt`
- Run tests: `./sbt test`
- Force Scala 2.11 (default): `./sbt -211`
- Force Scala 2.10: `./sbt -210`

3.3.4 À la Carte GeoTrellis Modules

GeoTrellis is actually a library suite made up of many modules. We've designed it such that you can depend on as much or as little of GeoTrellis as your project needs. To depend on a new module, add it to the `libraryDependencies` list in your `build.sbt`:

```
libraryDependencies ++= Seq(
    "org.locationtech.geotrellis" %% "geotrellis-spark"  % "1.0.0",
    "org.locationtech.geotrellis" %% "geotrellis-s3"     % "1.0.0", // now we can use␣
→Amazon S3!
    "org.apache.spark"            %% "spark-core"        % "2.1.0" % "provided",
    "org.scalatest"               %% "scalatest"         % "3.0.0" % "test"
)
```

Click here for a full list and explanation of each GeoTrellis module.

Now that you've gotten a simple GeoTrellis environment set up, it's time to get your feet wet with some of its capabilities.

3.4 Quick Start

For most users, it is not necessary to download the GeoTrellis source code to make use of it. The purpose of this document is to describe the fastest means to get a running environment for various use cases.

3.4.1 Wetting your Feet

By far, the quickest route to being able to play with GeoTrellis is to follow these steps:

- Use `git` to clone our project template repository:

```
git clone git@github.com:geotrellis/geotrellis-sbt-template
```

- Once available, from the root directory of the cloned repo, MacOSX and Linux users may launch the `sbt` script contained therein; this will start an SBT session, installing it if it has not already been.
- Once SBT is started, issue the `console` command; this will start the Scala interpreter.

At this point, you should be able to issue the command `import geotrellis.vector._` without raising an error. This will make the contents of that package available. For instance, one may create a point at the origin by typing `Point(0, 0)`.

This same project can be used as a template for writing simple programs. Under the project root directory is the `src` directory which has subtrees rooted at `src/main` and `src/test`. The former is where application code should be located, and the latter contains unit tests for any modules that demand it. The SBT documentation will describe how to run application or test code.

3.4.2 Hello Raster, Revisited

On the landing page, an example of an interactive session with GeoTrellis was shown. We're going to revisit that example here in more detail, using the various parts of that example as a means to highlight library features and to marshal beginners to the sections of interest in the documentation.

It is first necessary to expose functionality from the relevant packages (a complete list packages and the summary of their contents may be found here):

```scala
scala> import geotrellis.raster._
import geotrellis.raster._

scala> import geotrellis.raster.mapalgebra.focal._
import geotrellis.raster.mapalgebra.focal._
```

Much of GeoTrellis' core functionality lies in the raster library. Rasters are regular grids of data that have some notion of their spatial extent. When working with rasters, one can operate on the grid of data separately from the spatial information. The grid of data held inside a raster is called a Tile. We can create an example Tile as follows:

```scala
scala> val nd = NODATA
nd: Int = -2147483648

scala> val input = Array[Int](
     nd, 7, 1, 1, 3, 5, 9, 8, 2,
     9, 1, 1, 2, 2, 2, 4, 3, 5,
     3, 8, 1, 3, 3, 3, 1, 2, 2,
     2, 4, 7, 1, nd, 1, 8, 4, 3)
input: Array[Int] = Array(-2147483648, 7, 1, 1, 3, 5, 9, 8, 2, 9, 1, 1, 2,
2, 2, 4, 3, 5, 3, 8, 1, 3, 3, 3, 1, 2, 2, 2, 4, 7, 1, -2147483648, 1, 8, 4, 3)

scala> val iat = IntArrayTile(input, 9, 4)  // 9 and 4 here specify columns and rows
iat: geotrellis.raster.IntArrayTile = IntArrayTile(I@278434d0,9,4)

// The asciiDraw method is mostly useful when you're working with small tiles
// which can be taken in at a glance
scala> iat.asciiDraw()
res0: String =
"    ND      7      1      1      3      5      9      8      2
      9      1      1      2      2      2      4      3      5
      3      8      1      3      3      3      1      2      2
      2      4      7      1     ND      1      8      4      3
"
```

Note that not every cell location in a tile needs to be specified; this is the function of NODATA. Also be aware that NODATA's value varies by CellType. In this case, the use of IntArrayTile implies an IntCellType which defines NODATA as seen above.

As a GIS package, GeoTrellis provides a number of map algebra operations. In the following example, a neighborhood is defined as the region of interest for a focal operation, the focal mean operation is performed, and a value is queried:

```scala
scala> val focalNeighborhood = Square(1)  // a 3x3 square neighborhood
focalNeighborhood: geotrellis.raster.op.focal.Square =
 O  O  O
 O  O  O
 O  O  O

scala> val meanTile = iat.focalMean(focalNeighborhood)
meanTile: geotrellis.raster.Tile = DoubleArrayTile(D@7e31c125,9,4)

scala> meanTile.getDouble(0, 0)  // Should equal (1 + 7 + 9) / 3
res1: Double = 5.666666666666667
```

In this example, note that the NODATA value was simply ignored in the computation of the mean.

This is only a very simple example of what is possible with GeoTrellis. To learn more, it is recommended that the reader continue on with the core concepts section. Another example geared towards new users is available in the kernel density tutorial.

3.4.3 Using GeoTrellis with Apache Spark

GeoTrellis is meant for use in distributed environments employing Apache Spark. It's beyond the scope of a quickstart guide to describe how to set up or even to use Spark, but there are two paths to getting a REPL in which one can interact with Spark.

First: from the `geotrellis/geotrellis-sbt-template` project root directory, issue `./sbt` to start SBT. Once SBT is loaded, issue the `test:console` command. This will raise a REPL that will allow for the construction of a SparkContext using the following commands:

```
val conf = new org.apache.spark.SparkConf()
conf.setMaster("local[*]")
implicit val sc = geotrellis.spark.util.SparkUtils.createSparkContext("Test console",
→conf)
```

It will then be possible to issue a command such as `sc.parallelize(Array(1,2,3))`.

Alternatively, if you have source files inside a project directory tree, you may issue the `assembly` command from `sbt` to produce a fat .jar file, which will appear in the `target/scala-<version>/` directory under your current project's root directory. That jar file can be supplied to the Spark shell as part of the command `spark-shell --conf spark.serializer=org.apache.spark.serializer.KryoSerializer --jars <jarfile>`, provided you have Spark installed on your local machine. Fat jars created via `assembly` can be supplied as well to `spark-submit` commands for running jobs on a remote Spark master. Again, the ins-and-outs of Spark are beyond the scope of this document, but these pointers might provide useful jumping off points.

3.5 Kernel Density

This document provides a detailed example on how to build a raster from point data using kernel density estimation. Though that is the ostensible point, it also provides a brief introduction to working with rasters, including how to tile a raster and how to use the result as the basis for a computation in Spark.

Kernel density is one way to convert a set of points (an instance of vector data) into a raster. In this process, at every point in the point set, the contents of what is effectively a small Tile (called a Kernel) containing a predefined pattern are added to the grid cells surrounding the point in question (i.e., the kernel is centered on the tile cell containing the point and then added to the Tile). This is an example of a local map algebra operation. Assuming that the points were sampled according to a probability density function, and if the kernel is derived from a Gaussian function, this can develop a smooth approximation to the density function that the points were sampled from. (Alternatively, each point can be given a weight, and the kernel values can be scaled by that weight before being applied to the tile, which we will do below.)

To begin, let's generate a random collection of points with weights in the range (0, 32). To capture the idea of "a point with weight", we create a `PointFeature[Double]` (which is an alias of `Feature[Point, Double]`). Features, in general, combine a geometry with an attribute value in a type-safe way.

```
import geotrellis.vector._
import scala.util._

val extent = Extent(-109, 37, -102, 41) // Extent of Colorado

def randomPointFeature(extent: Extent): PointFeature[Double] = {
  def randInRange (low: Double, high: Double): Double = {
    val x = Random.nextDouble
    low * (1-x) + high * x
  }
```

(continues on next page)

```
  Feature(Point(randInRange(extent.xmin, extent.xmax),     // the geometry
                randInRange(extent.ymin, extent.ymax)),
        Random.nextInt % 16 + 16)                          // the weight (attribute)
}

val pts = (for (i <- 1 to 1000) yield randomPointFeature(extent)).toList
```

The choice of extent is largely arbitrary in this example, but note that the coordinates here are taken with respect to the standard (longitude, latitude) that we normally consider. Other coordinate representations are available, and it might be useful to investigate coordinate reference systems (CRSs) if you want more details. Some operations in GeoTrellis require that a CRS object be constructed to place your rasters in the proper context. For (longitude, latitude) coordinates, either `geotrellis.proj4.CRS.fromName("EPSG:4326")` or `geotrellis.proj4.LatLng` will generate the desired CRS.

Next, we will create a tile containing the kernel density estimate:

```
import geotrellis.raster._
import geotrellis.raster.mapalgebra.focal.Kernel

val kernelWidth: Int = 9

/* Gaussian kernel with std. deviation 1.5, amplitude 25 */
val kern: Kernel = Kernel.gaussian(kernelWidth, 1.5, 25)

val kde: Tile = pts.kernelDensity(kern, RasterExtent(extent, 700, 400))
```

This populates a 700x400 tile with the desired kernel density estimate. In order to view the resulting file, a simple method is to write the tile out to PNG or TIFF. In the following snippet, a PNG is created in the directory `sbt` was launched in (the working directory), where the values of the tile are mapped to colors that smoothly interpolate from blue to yellow to red.

```
import geotrellis.raster.render._

val colorMap = ColorMap(
  (0 to kde.findMinMax._2 by 4).toArray,
  ColorRamps.HeatmapBlueToYellowToRedSpectrum
)

kde.renderPng(colorMap).write("test.png")
```

The advantage of using a TIFF output is that it will be tagged with the extent and CRS, and the resulting image file can be overlaid on a map in a viewing program such as QGIS. That output is generated by the following statements.

```
import geotrellis.raster.io.geotiff._

GeoTiff(kde, extent, LatLng).write("test.tif")
```

3.5.1 Subdividing Tiles

The above example focuses on a toy problem that creates a small raster object. However, the purpose of GeoTrellis is to enable the processing of large-scale datasets. In order to work with large rasters, it will be necessary to subdivide a region into a grid of tiles so that the processing of each piece may be handled by different processors which may, for example, reside on remote machines in a cluster. This section explains some of the concepts involved in subdividing a raster into a set of tiles.

We will still use an `Extent` object to set the bounds of our raster patch in space, but we must now specify how that extent is broken up into a grid of `Tiles`. This requires a statement of the form:

```
import geotrellis.spark.tiling._

val tl = TileLayout(7, 4, 100, 100)
```

Here, we have specified a 7x4 grid of Tiles, each of which has 100x100 cells. This will eventually be used to divide the earlier monolithic 700x400 Tile (`kde`) into 28 uniformly-sized subtiles. The TileLayout is then combined with the extent in a `LayoutDefinition` object:

```
val ld = LayoutDefinition(extent, tl)
```

In preparation for reimplementing the previous kernel density estimation with this structure, we note that each point in `pts` lies at the center of a kernel, which covers some non-zero area around the points. We can think of each point/kernel pair as a small square extent centered at the point in question with a side length of 9 pixels (the arbitrary width we chose for our kernel earlier). Each pixel, however, covers some non-zero area of the map, which we can also think of as an extent with side lengths given in map coordinates. The dimensions of a pixel's extent are given by the `cellwidth` and `cellheight` members of a LayoutDefinition object.

By incorporating all these ideas, we can create the following function to generate the extent of the kernel centered at a given point:

```
def pointFeatureToExtent[D](kwidth: Double, ld: LayoutDefinition, ptf:␣
→PointFeature[D]): Extent = {
  val p = ptf.geom

  Extent(p.x - kwidth * ld.cellwidth / 2,
         p.y - kwidth * ld.cellheight / 2,
         p.x + kwidth * ld.cellwidth / 2,
         p.y + kwidth * ld.cellheight / 2)
}

def ptfToExtent[D](p: PointFeature[D]) = pointFeatureToExtent(9, ld, p)
```

When we consider the kernel extent of a point in the context of a LayoutDefinition, it's clear that a kernel's extent may overlap more than one tile in the layout. To discover the tiles that a given point's kernel extents overlap, LayoutDefinition provides a `mapTransform` object. Among the methods of mapTransform is the ability to determine the indices of the subtiles in the TileLayout that overlap a given extent. Note that the tiles in a given layout are indexed by `SpatialKeys`, which are effectively `(Int,Int)` pairs giving the (column,row) of each tile as follows:

```
+-------+-------+-------+         +-------+
| (0,0) | (1,0) | (2,0) | . . . | (6,0) |
+-------+-------+-------+         +-------+
| (0,1) | (1,1) | (2,1) | . . . | (6,1) |
+-------+-------+-------+         +-------+
    .       .       .       .         .
    .       .       .       .         .
    .       .       .       .         .
+-------+-------+-------+         +-------+
| (0,3) | (1,3) | (2,3) | . . . | (6,3) |
+-------+-------+-------+         +-------+
```

Specifically, in our running example, `ld.mapTransform(ptfToExtent(Feature(Point(-108, 38), 100.0)))` returns `GridBounds(0, 2, 1, 3)`, indicating that every cell with columns in the range [0,1] and rows in the range [2,3] are intersected by the kernel centered on the point (-108, 38)—that is, the 2x2 block of tiles at the lower left of the layout.

In order to proceed with the kernel density estimation, it is necessary to then convert the list of points into a collection of (SpatialKey, List[PointFeature[Double]]) that gathers all the points that have an effect on each subtile, as indexed by their SpatialKeys. The following snippet accomplishes that.

```
import geotrellis.spark._

def ptfToSpatialKey[D](ptf: PointFeature[D]): Seq[(SpatialKey,PointFeature[D])] = {
  val ptextent = ptfToExtent(ptf)
  val gridBounds = ld.mapTransform(ptextent)

  for {
    (c, r) <- gridBounds.coords
    if r < tl.totalRows
    if c < tl.totalCols
  } yield (SpatialKey(c,r), ptf)
}

val keyfeatures: Map[SpatialKey, List[PointFeature[Double]]] =
  pts
    .flatMap(ptfToSpatialKey)
    .groupBy(_._1)
    .map { case (sk, v) => (sk, v.unzip._2) }
```

Now, all the subtiles may be generated in the same fashion as the monolithic tile case above.

```
val keytiles = keyfeatures.map { case (sk, pfs) =>
  (sk, pfs.kernelDensity(
    kern,
    RasterExtent(ld.mapTransform(sk), tl.tileDimensions._1, tl.tileDimensions._2)
  ))
}
```

Finally, it is necessary to combine the results. Note that, in order to produce a 700x400 tile that is identical to the simpler, non-tiled case, every SpatialKey must be represented in the map, or the result may not span the full range. This is only necessary if it is important to generate a tile that spans the full extent.

```
import geotrellis.spark.stitch.TileLayoutStitcher

val tileList =
  for {
    r <- 0 until ld.layoutRows
    c <- 0 until ld.layoutCols
  } yield {
    val k = SpatialKey(c,r)
    (k, keytiles.getOrElse(k, IntArrayTile.empty(tl.tileCols, tl.tileRows)))
  }

val stitched = TileLayoutStitcher.stitch(tileList)._1
```

It is now the case that stitched is identical to kde.

3.5.2 Distributing Computation via Apache Spark

As mentioned, the reason for introducing this more complicated version of kernel density estimation was to enable distributed processing of the kernel stamping process. Each subtile could potentially be handled by a different node in a cluster, which would make sense if the dataset in question were, say, the location of individual trees, requiring a huge amount of working memory. By breaking the domain into smaller pieces, we can exploit the distributed framework

provided by Apache Spark to spread the task to a number of machines. This will also provide fault tolerant, rapid data processing for real-time and/or web-based applications. Spark is much too big a topic to discuss in any detail here, so the curious reader is advised to search the web for more information. Our concern falls on how to write code to exploit the structures provided by Spark, specifically *Resilient Distributed Datasets*, or RDDs. An RDD is a distributed collection, with all of the usual features of a collection—e.g., map, reduce—as well as distributed versions of certain sequential operations—e.g., `aggregate` is akin to `foldLeft` for standard collections. In order to create an RDD, one will call the `parallelize()` method on a `SparkContext` object, which can be generated by the following set of statements.

```scala
import org.apache.spark.{SparkConf, SparkContext}

val conf = new SparkConf().setMaster("local").setAppName("Kernel Density")
val sc = new SparkContext(conf)
```

In our case, we have a collection of PointFeatures that we wish to parallelize, so we issue the command

```scala
import org.apache.spark.rdd.RDD

val pointRdd = sc.parallelize(pts, 10)
```

Here, the 10 indicates that we want to distribute the data, as 10 partitions, to the available workers. A partition is a subset of the data in an RDD that will be processed by one of the workers, enabling parallel, distributed computation, assuming the existence of a pool of workers on a set of machines. If we exclude this value, the default parallelism will be used, which is typically the number of processors, though in this local example, it defaults to one.

In order to perform the same task as in the previous section, but in parallel, we will approach the problem in much the same way: points will be assigned an extent corresponding to the extent of the associated kernel, those points will be assigned SpatialKeys based on which subtiles their kernels overlap, and each kernel will be applied to the tile corresponding to its assigned SpatialKey. Earlier, this process was effected by a flatMap followed by a groupBy and then a map. This very same procedure could be used here, save for the fact that groupBy, when applied to an RDD, triggers an expensive, slow, network-intensive shuffling operation which collects items with the same key on a single node in the cluster. Instead, a fold-like operation will be used: `aggregateByKey`, which has a signature of `RDD[(K, U)] => T => ((U, T) => T, (T, T) => T) => RDD[(K, T)]`. That is, we begin with an RDD of key/value pairs, provide a "zero value" of type `T`, the type of the final result, and two functions: (1) a *sequential operator*, which uses a single value of type `U` to update an accumulated value of type `T`, and (2) a *combining operator*, which merges two accumulated states of type `T`. In our case, `U = PointFeature[Double]` and `T = Tile`; this implies that the insertion function is a kernel stamper and the merging function is a tile adder.

```scala
import geotrellis.raster.density.KernelStamper

def stampPointFeature(
  tile: MutableArrayTile,
  tup: (SpatialKey, PointFeature[Double])
): MutableArrayTile = {
  val (spatialKey, pointFeature) = tup
  val tileExtent = ld.mapTransform(spatialKey)
  val re = RasterExtent(tileExtent, tile)
  val result = tile.copy.asInstanceOf[MutableArrayTile]

  KernelStamper(result, kern)
    .stampKernelDouble(re.mapToGrid(pointFeature.geom), pointFeature.data)

  result
}

import geotrellis.raster.mapalgebra.local.LocalTileBinaryOp
```

```scala
object Adder extends LocalTileBinaryOp {
  def combine(z1: Int, z2: Int) = {
    if (isNoData(z1)) {
      z2
    } else if (isNoData(z2)) {
      z1
    } else {
      z1 + z2
    }
  }

  def combine(r1: Double, r2:Double) = {
    if (isNoData(r1)) {
      r2
    } else if (isNoData(r2)) {
      r1
    } else {
      r1 + r2
    }
  }
}

def sumTiles(t1: MutableArrayTile, t2: MutableArrayTile): MutableArrayTile = {
  Adder(t1, t2).asInstanceOf[MutableArrayTile]
}
```

Note that we require a custom Adder implementation because the built-in tile summation operator returns NODATA if either of the cells being added contain a NODATA value themselves.

Creating the desired result is then a matter of the following series of operations on `pointRdd`:

```scala
val tileRdd: RDD[(SpatialKey, Tile)] =
  pointRdd
    .flatMap(ptfToSpatialKey)
    .mapPartitions({ partition =>
      partition.map { case (spatialKey, pointFeature) =>
        (spatialKey, (spatialKey, pointFeature))
      }
    }, preservesPartitioning = true)
    .aggregateByKey(ArrayTile.empty(DoubleCellType, ld.tileCols, ld.tileRows))
                  (stampPointFeature, sumTiles)
    .mapValues{ tile: MutableArrayTile => tile.asInstanceOf[Tile] }
```

The `mapPartitions` operation simply applies a transformation to an RDD without triggering any kind of shuffling operation. Here, it is necessary to make the SpatialKey available to `stampPointFeature` so that it can properly determine the pixel location in the corresponding tile.

We would be finished here, except that RDDs inside GeoTrellis are required to carry along a Metadata object that describes the context of the RDD. This is created like so:

```scala
import geotrellis.proj4.LatLng

val metadata = TileLayerMetadata(DoubleCellType,
                                 ld,
                                 ld.extent,
                                 LatLng,
```

```
                     KeyBounds(SpatialKey(0,0),
                             SpatialKey(ld.layoutCols-1,
                                        ld.layoutRows-1)))
```

To combine the RDD and the metadata, we write `val resultRdd = ContextRDD(tileRdd, metadata)`.

This resulting RDD is essentially the object of interest, though it is possible to write `resultRDD.stitch` to produce a single merged tile. In the general case, however, the RDD may cover an area so large and in sufficient resolution that the result of stitching would be too large for working memory. In these sorts of applications, the usual work flow is to save the tiles off to one of the distributed back ends (Accumulo, S3, HDFS, etc.). Tiles thus stored may then be used in further processing steps or be served to applications (e.g., web mapping applications). If it is absolutely necessary, the individual tiles may be saved off as GeoTIFFs and stitched via an application like GDAL.

3.5.3 A Note on Running Example Code

To run the above test code, it is necessary to have a compatible environment. Spark code may experience failures if run solely in the Scala interpreter, as accessed through SBT's `console` command. One way to ensure proper execution is to run in `spark-shell`, a Scala environment which provides a SparkContext made available through the variable `sc`. Another way is to compile the application into a JAR file using `sbt assembly`, and to use `spark-submit`. This latter option is the preferred method for Spark applications, in general, but for the purposes of trying out the provided code samples, `spark-shell` is the more sensible choice. The use of `spark-submit` is beyond the scope of this documentation, but many resources are available on the internet for learning this tool.

In either event, it will be necessary to install Spark in your local environment to run the code above. Once that is done, you will need to clone the GeoTrellis repository from [Github](). From the root directory of that project, execute the provided `sbt` script. Once SBT is loaded, the following commands can be executed:

```
project spark-etl
assembly
```

This packages the required class files into a JAR file. Now, again from the GeoTrellis source tree root directory, issue the command

```
spark-shell --jars spark-etl/target/scala-2.11/geotrellis-spark-etl-assembly-
→[version].jar
```

From the resulting interpreter prompt, perform the following imports:

```scala
import geotrellis.raster._
import geotrellis.vector._
import geotrellis.proj4._
import geotrellis.spark._
import geotrellis.spark.util._
import geotrellis.spark.tiling._
```

It should then be possible to input the example code from above (excluding the creation of a SparkContext) and get the desired result.

3.5.4 A Note on Implementation

The procedures that we've been considering above have been implemented in GeoTrellis and are located in `raster/src/main/scala/geotrellis/raster/density/` and `spark/src/main/scala/geotrellis/spark/density`. This final implementation is more complete than the simple version presented here, as it handles type conversion for different tile cell types and is augmented with convenience functions that are

---

provided through the use of the `MethodExtensions` facility. Briefly, method extensions allow for implicit conversion between `Traversable[PointFeature[Num]]` (where `Num` is either `Int` or `Double`) and a wrapper class which provides a method `kernelDensity:   (Kernel, RasterExtent) => Tile`. Thus, any traversable collection can be treated as if it possesses a `kernelDensity` method. This pattern appears all throughout GeoTrellis, and provides some welcome syntactic sugar.

Furthermore, the final implementation is more flexible with regard to the type of data used. Both the PointFeature parameter and the Tile CellType may be of integral or floating-point type. See the code for details.

3.6 Reading GeoTiffs

This tutorial will go over how to read GeoTiff files using GeoTrellis on your local filesystem. It assumes that you already have the environment needed to run these examples. If not, please see our Setup Guide to get GeoTrellis working on your system. Also, this tutorial uses GeoTiffs from the `raster` project from GeoTrellis. If you have not already done so, please clone GeoTrellis here so that you can access the needed files.

One of the most common methods of storing geospatial information is through GeoTiffs. This is reflected throughout the GeoTrellis library where many of its features can work with GeoTiffs. Which would mean that there would have to be many different ways to read in GeoTiff, and indeed there are! In the following document, we will go over the methods needed to load in a GeoTiff from your local filesystem.

Before we start, open a Scala REPL in the Geotrellis directory.

3.6.1 Reading For the First Time

Reading a local GeoTiff is actually pretty easy. You can see how to do it below.

```
import geotrellis.raster.io.geotiff.reader.GeoTiffReader
import geotrellis.raster.io.geotiff._

val path: String = "path/to/geotrellis/raster/data/geotiff-test-files/lzw_int32.tif"
val geoTiff: SinglebandGeoTiff = GeoTiffReader.readSingleband(path)
```

And that's it! Not too bad at all really, just four lines of code. Even still, though, let's break this down line-by-line so we can see what exactly is going on.

```
import geotrellis.raster.io.geotiff.reader.GeoTiffReader
```

This import statement brings in `GeoTiffReader` from `geotrellis.raster.io.geotiff.reader` so that we can use it in the REPL. As the name implys, `GeoTiffReader` is the object that actually reads the GeoTiff. If you ever wonder about how we analyze and process GeoTiffs, then `geotrellis.raster.io.geotiff` would be the place to look. Here's a link.

```
import geotrellis.raster.io.geotiff._
```

The next import statement loads in various data types that we need so that we can assign them to our `val`s.

Okay, so we brought in the object that will give us our GeoTiff, now we just need to supply it what to read. This is where the next line of code comes into play.

```
val path: String = "path/to/geotrellis/raster/data/geotiff-test-files/lzw_int32.tif"
```

Our `path` variable is a `String` that contains the file path to a GeoTiff in `geotrellis.raster`. `GeoTiffReader` will use this value then to read in our GeoTiff. There are more types of paramters `GeoTiffReader` can accept, however. These are `Array[Byte]`s and `ByteReader`s. We will stick with

`Strings` for this lesson, but `Array[Byte]` is not that much different. It's just all of the bytes within your file held in an Array.

The last part of our four line coding escapade is:

```
val geoTiff: SinglebandGeoTiff = GeoTiffReader.readSingleband(path)
```

This line assigns the variable, `geoTiff`, to the file that is being read in. Notice the `geoTiff`'s type, though. It is `SinglebandGeoTiff`. Why does `geoTiff` have this type? It's because in GeoTrellis, `SinglebandGeoTiff`s and `MutlibandGeoTiff`s are two seperate subtypes of `GeoTiff`. In case you were wondering about the second `import` statement earlier, this is where is comes into play; as these two types are defined within `geotrellis.raster.io.geotiff`.

Great! We have a `SinglebandGeoTiff`. Let's say that we have a `MultibandGeoTiff`, though; let's use the code from above to read it.

```
import geotrellis.raster.io.geotiff.reader.GeoTiffReader
import geotrellis.raster.io.geotiff._

// now a MultibandGeoTiff!
val path: String = "path/to/raster/data/geotiff-test-files/3bands/3bands-striped-band.
↪tif"
val geoTiff = GeoTiffReader.readSingleband(path)
```

If we run this code, what do you think will happen? The result may surprise you, we get back a `SinglebandGeoTiff`! When told to read a `SinglebandGeoTiff` from a `MultibandGeoTiff` without a return type, the `GeoTiffReader` will just read in the first band of the file and return that. Thus, it is important to keep in mind what kind of GeoTiff you are working with, or else you could get back an incorrect result.

To remedy this issue, we just have to change the method call and return type so that `GeoTiffReader` will read in all of the bands of our GeoTiff.

```
val geoTiff: MultibandGeoTiff = GeoTiffReader.readMultiband(path)
```

And that's it! We now have our `MutlibandGeoTiff`.

Beginner Tip

A good way to ensure that your codes works properly is to give the return data type for each of your `vals` and `defs`. If by chance your return type and is different from what is actually returned, the compiler will throw an error. In addition, this will also make your code easier to read and understand for both you and others as well. Example:

```
val multiPath = "path/to/a/multiband/geotiff.tif"

// This will give you the wrong result!
val geoTiff = GeoTiffReader.readSingleband(multiPath)

// This will cause your compiler to throw an error
val geoTiff: MultibandGeoTiff = GeoTiffReader.readSingleband(multiPath)
```

Before we move on to the next section, I'd like to take moment and talk about an alternative way in which you can read in GeoTiffs. Both `SinglebandGeoTiffs` and `MultibandGeoTiffs` have their own `apply` methods, this means that you can give your parameter(s) directly to their companion objects and you'll get back a new instance of the class.

For `SinglebandGeoTiffs`:

```
import geotrellis.raster.io.geotiff.SinglebandGeoTiff

val path: String = "path/to/raster/data/geotiff-test-files/lzw_int32.tif"
val geoTiff: SinglebandGeoTiff = SinglebandGeoTiff(path)
```

There are two differences found within this code from the previous example. The first is this:

```
import geotrellis.raster.io.geotiff.SinglebandGeoTiff
```

As stated earlier, `SinglebandGeoTiff` and `MultibandGeoTiff` are found within a different folder of `geotrellis.raster.io.geotiff`. This is important to keep in mind when importing, as it can cause your code not to compile if you refer to the wrong sub-folder.

The second line that was changed is:

```
val geoTiff: SinglebandGeoTiff = SinglebandGeoTiff(path)
```

Here, we see `SinglebandGeoTiff`'s `apply` method being used on `path`. Which returns the same thing as `GeoTiffReader.readSingleband(path)`, but with less verbosity.

`MultibandGeoTiff`s are the exact same as their singleband counterparts.

```
import geotrellis.raster.io.geotiff.MultibandGeoTiff

val path: String = "raster/data/geotiff-test-files/3bands/3bands-striped-band.tif"
val geoTiff: MultibandGeoTiff = MultibandGeoTiff(path)
```

Our overview of basic GeoTiff reading is now done! But keep reading! For you have greater say over how your GeoTiff will be read than what has been shown. - - -

3.6.2 Expanding Our Vocab

We can read GeoTiffs, now what? Well, there's actually more that we can do when reading in a file. Sometimes you have a compressed GeoTiff, or other times you might want to read in only a sub-section of GeoTiff and not the whole thing. In either case, GeoTrellis can handle these issues with ease.

Dealing With Compressed GeoTiffs

Compression is a method in which data is stored with fewer bits and can then be uncompressed so that all data becomes available. This applies to GeoTiffs as well. When reading in a GeoTiff, you can state whether or not you want a compressed file to be uncompressed or not.

```
import geotrellis.raster.io.geotiff.reader.GeoTiffReader
import geotrellis.raster.io.geotiff._

// reading in a compressed GeoTiff and keeping it compressed
val compressedGeoTiff: SinglebandGeoTiff = GeoTiffReader.readSingleband("path/to/
↪compressed/geotiff.tif", false, false)

// reading in a compressed GeoTiff and uncompressing it
val compressedGeoTiff: SinglebandGeoTiff = GeoTiffReader.readSingleband("path/to/
↪compressed/geotiff.tif", true, false)
```

As you can see from the above code sample, the first `Boolean` value is what determines whether or not the file should be decompressed or not. What does the other `Boolean` value for? We'll get to that soon! For right now, though, we'll just focus on the first one.

Why would you want to leave a file compressed or have uncompressed when reading it? One of the benefits of using compressed GeoTiffs is that might lead to better performance depending on your system and the size of the file. Another instance where the compression is needed is if your file is over 4GB is size. This is because when a GeoTiff is uncompressed in GeoTrellis, it is stored in an Array. Anything over 4GB is larger than the max array size for Java, so trying read in anything bigger will cause your process to crash.

By default, decompression occurs on all read GeoTiffs. Thus, these two lines of code are the same.

```
// these will both return the same thing!
GeoTiffReader.readSingleband("path/to/compressed/geotiff.tif")
GeoTiffReader.readSingleband("path/to/compressed/geotiff.tif", true, false)
```

In addition, both `SinglebandGeoTiff` and `MultibandGeoTiff` have a method, `compressed`, that uncompresses a GeoTiff when it is read in.

```
SinglebandGeoTiff.compressed("path/to/compressed/geotiff.tif")
MultibandGeoTiff.compressed("path/to/compressed/geotiff.tif")
```

Streaming GeoTiffs

Remember that mysterious second parameter from earlier? It determines if a GeoTiff should be read in via streaming or not. What is streaming? Streaming is process of not reading in all of the data of a file at once, but rather getting the data as you need it. It's like a "lazy read". Why would you want this? The benefit of streaming is that it allows you to work with huge or just parts of files. In turn, this makes it possible to read in sub-sections of GeoTiffs and/or not having to worry about memory usage when working with large files.

Tips For Using This Feature

It is important to go over the strengths and weaknesses of this feature before use. If implemented well, the Windowed-GeoTiff Reader can save you a large amount of time. However, it can also lead to further problems if it is not used how it was intended.

It should first be stated that this reader was made to read in **sections** of a Geotiff. Therefore, reading in either the entire, or close to the whole file will either be comparable or slower than reading in the entire file at once and then cropping it. In addition, crashes may occur depending on the size of the file.

Reading in Small Files

Smaller files are GeoTiffs that are less than or equal to 4GB in isze. The way to best utilize the reader for these kinds of files differs from larger ones.

To gain optimum performance, the principle to follow is: **the smaller the area selected, the faster the reading will be**. What the exact performance increase will be depends on the bandtype of the file. The general pattern is that the larger the datatype is, quicker it will be at reading. Thus, a Float64 GeoTiff will be loaded at a faster rate than a UByte GeoTiff. There is one caveat to this rule, though. Bit bandtype is the smallest of all the bandtypes, yet it can be read in at speed that is similar to Float32.

For these files, 90% of the file is the cut off for all band and storage types. Anything more may cause performance declines.

Reading in Large Files

Whereas small files could be read in full using the reader, larger files cannot as they will crash whatever process you're running. The rules for these sorts of files are a bit more complicated than that of their smaller counterparts, but learning them will allow for much greater performance in your analysis.

One similarity that both large and small files share is that they have the same principle: **the smaller the area selected, the faster the reading will be**. However, while smaller files may experience slowdown if the selected area is too large, these bigger files will crash. Therefore, this principle must be applied more strictly than with the previous file sizes.

In large files, the pattern of performance increase is the reverse of the smaller files. Byte bandtype can not only read faster, but are able to read in larger areas than bigger bandtypes. Indeed, the area which you can select is limited to what the bandtype of the GeoTiff is. Hence, an additional principle applies for these large files: **the smaller the bandtype, the larger of an area you can select**. The exact size for each bandtype is not known, estimates have been given in the table bellow that should provide some indication as to what size to select.

BandType	Area Threshold Range In Cells
Byte	[5.76 * 109, 6.76 * 109)
Int16	[3.24 * 109, 2.56 * 109)
Int32	[1.44 * 109, 1.96 * 109)
UInt16	[1.96 * 109, 2.56 * 109)
UInt32	[1.44 * 109, 1.96 * 109)
Float32	[1.44 * 109, 1.96 * 109)
Float64	[3.6 * 108, 6.4 * 108)

How to Use This Feature

Using this feature is straight forward and easy. There are two ways to implement the WindowedReader: Supplying the desired extent with the path to the file, and cropping an already existing file that is read in through a stream.

Using Apply Methods

Supplying an extent with the file's path and having it being read in windowed can be done in the following ways:

```scala
val path: String = "path/to/my/geotiff.tif"
val e: Extent = Extent(0, 1, 2, 3)

// supplying the extent as an Extent

// if the file is singleband
SinglebandGeoTiff(path, e)
// or
GeoTiffReader.readSingleband(path, e)

// if the file is multiband
MultibandGeoTiff(path, e)
// or
GeoTiffReader.readMultiband(path, e)

// supplying the extent as an Option[Extent]

// if the file is singleband
SinglebandGeoTiff(path, Some(e))
```

(continues on next page)

```
// or
GeoTiffReader.readSingleband(path, Some(e))

// if the file is multiband
MultibandGeoTiff(path, Some(e))
// or
GeoTiffReader.readMultiband(path, Some(e))
```

Using Object Methods

Cropping an already loaded GeoTiff that was read in through Streaming. By using this method, the actual file isn't loaded into memory, but its data can still be accessed. Here's how to do the cropping:

```
val path: String = "path/to/my/geotiff.tif"
val e: Extent = Extent(0, 1, 2, 3)

// doing the reading and cropping in one line

// if the file is singleband
SinglebandGeoTiff.streaming(path).crop(e)
// or
GeoTiffReader.readSingleband(path, false, true).crop(e)

// if the file is multiband
MultibandGeoTiff.streaming(path).crop(e)
// or
GeoTiffReader.readMultiband(path, false, true).crop(e)

// doing the reading and cropping in two lines

// if the file is singleband
val sgt: SinglebandGeoTiff =
  SinglebandGeoTiff.streaming(path)
  // or
  GeoTiffReader.readSingleband(path, false, true)
sgt.crop(e)

// if the file is multiband
val mgt: MultibandGeoTiff =
  MultibandGeoTiff.streaming(path)
  // or
  GeoTiffReader.readMultiband(path, false, true)
mgt.crop(e)
```

3.6.3 Conclusion

That takes care of reading local GeoTiff files! It should be said, though, that what we went over here does not just apply to reading local files. In fact, reading in GeoTiffs from other sources have similar parameters that you can use to achieve the same goal.

3.7 Extract-Transform-Load (ETL)

This brief tutorial describes how to use GeoTrellis' Extract-Transform-Load ("ETL") functionality to create a GeoTrellis catalog. We will accomplish this in four steps:

1. we will build the ETL assembly from code in the GeoTrellis source tree,

2. we will compose JSON configuration files describing the input and output data,

3. we will perform the ingest, creating a GeoTrellis catalog, and

4. we will exercise the ingested data using a simple project.

It is assumed throughout this tutorial that Spark 2.0.0 or greater is installed, that the GDAL command line tools are installed, and that the GeoTrellis source tree has been locally cloned.

3.7.1 Local ETL

Build the ETL Assembly

Navigate into the GeoTrellis source tree, build the assembly, and copy it to the `/tmp` directory:

```
cd geotrellis
./sbt "project spark-etl" assembly
cp spark-etl/target/scala-2.11/geotrellis-spark-etl-assembly-1.0.0.jar /tmp
```

Although in this tutorial we have chosen to build this assembly directly from the GeoTrellis source tree, in some applications it may be desirable to create a class in one's own code base that uses or derives from `geotrellis.spark.etl.SinglebandIngest` or `geotrellis.spark.etl.MultibandIngest`, and use that custom class as the entry-point. Please see the Chatta Demo for an example of how to do that.

Compose JSON Configuration Files

The configuration files that we create in this section are intended for use with a single multiband GeoTiff image. Three JSON files are required: one describing the input data, one describing the output data, and one describing the backend(s) in which the catalog should be stored. Please see our more detailed ETL documentation for more information about the configuration files.

We will now create three files in the `/tmp/json` directory: `input.json`, `output.json`, and `backend-profiles.json`. (The respective schemas that those files must obey can be found here, here, and here.)

Here is `input.json`:

```
[{
    "format": "multiband-geotiff",
    "name": "example",
    "cache": "NONE",
    "backend": {
        "type": "hadoop",
        "path": "file:///tmp/rasters"
    }
}]
```

The value `multiband-geotiff` is associated with the `format` key. That is required if you want to access the data as an RDD of `SpatialKey-MultibandTile` pairs. Making that value `geotiff` instead of

multiband-geotiff would result in SpatialKey-Tile pairs. The value example associated with the key name gives the name of the layer(s) that will be created. The cache key gives the Spark caching strategy that will be used during the ETL process. Finally, the value associated with the backend key specifies where the data should be read from. In this case, the source data are stored in the directory /tmp/rasters on local filesystem and accessed via Hadoop.

Here is the output.json file:

```
{
    "backend": {
        "type": "hadoop",
        "path": "file:///tmp/catalog/"
    },
    "reprojectMethod": "buffered",
    "pyramid": true,
    "tileSize": 256,
    "keyIndexMethod": {
        "type": "zorder"
    },
    "resampleMethod": "cubic-spline",
    "layoutScheme": "zoomed",
    "crs": "EPSG:3857"
}
```

That file says that the catalog should be created on the local filesystem in the directory /tmp/catalog using Hadoop. The source data is pyramided so that layers of zoom level 0 through 12 are created in the catalog. The tiles are 256-by-256 pixels in size and are indexed in according to Z-order. Bicubic resampling (spline rather than convolutional) is used in the reprojection process, and the CRS associated with the layers is EPSG 3857 (a.k.a. Web Mercator).

Here is the backend-profiles.json file:

```
{
    "backend-profiles": []
}
```

In this case, we did not need to specify anything since we are using Hadoop for both input and output. It happens that Hadoop only needs to know the path to which it should read or write, and we provided that information in the input.json and output.json files. Other backends such as Cassandra and Accumulo require information to be provided in the backend-profiles.json file.

Create the Catalog

Before performing the ingest, we will first retile the source raster. This is not strictly required if the source image is small enough (probably less than 2GB), but is still good practice even if it is not required.

```
mkdir -p /tmp/rasters
gdal_retile.py source.tif -of GTiff -co compress=deflate -ps 256 256 -targetDir /tmp/
↪rasters
```

The result of this command is a collection of smaller GeoTiff tiles in the directory /tmp/rasters.

Now with all of the files that we need in place (/tmp/geotrellis-spark-etl-assembly-1.0.0.jar, /tmp/json/input.json, /tmp/json/output.json, /tmp/json/backend-profiles.json, and /tmp/rasters/*.tif) we are ready to perform the ingest. That can be done by typing:

```
rm -rf /tmp/catalog
$SPARK_HOME/bin/spark-submit \
   --class geotrellis.spark.etl.MultibandIngest \
   --master 'local[*]' \
   --driver-memory 16G \
   /tmp/geotrellis-spark-etl-assembly-1.0.0.jar \
   --input "file:///tmp/json/input.json" \
   --output "file:///tmp/json/output.json" \
   --backend-profiles "file:///tmp/json/backend-profiles.json"
```

After the `spark-submit` command completes, there should be a directory called `/tmp/catalog` which contains the catalog.

Optional: Exercise the Catalog

Clone or download this example code (a zipped version of which can be downloaded from here). The example code is a very simple project that shows how to read layers from an HDFS catalog, perform various computations on them, then dump them to disk so that they can be inspected.

Once obtained, the code can be built like this:

```
cd EtlTutorial
./sbt "project tutorial" assembly
cp tutorial/target/scala-2.11/tutorial-assembly-0.jar /tmp
```

The code can be run by typing:

```
mkdir -p /tmp/tif
$SPARK_HOME/bin/spark-submit \
   --class com.azavea.geotrellis.tutorial.EtlExercise \
   --master 'local[*]' \
   --driver-memory 16G \
   /tmp/tutorial-assembly-0.jar /tmp/catalog example 12
```

In the block above, `/tmp/catalog` is an HDFS URI pointing to the location of the catalog, `example` is the layer name, and `12` is the layer zoom level. After running the code, you should find a number of images in `/tmp/tif` which are GeoTiff renderings of the tiles of the raw layer, as well as the layer with various transformations applied to it.

3.7.2 GeoDocker ETL

The foregoing discussion showed how to ingest data to the local filesystem, albeit via Hadoop. In this section, we will give a basic example of how to use the ETL machinery to ingest into HDFS on GeoDocker. Throughout this section we will assume that the files that were previously created in the local `/tmp` directory (namely `/tmp/geotrellis-spark-etl-assembly-1.0.0.jar`, `/tmp/json/input.json`, `/tmp/json/output.json`, `/tmp/json/backend-profiles.json`, and `/tmp/rasters/*.tif`) still exist.

In addition to the dependencies needed to complete the steps given above, this section assumes that user has a recent version of `docker-compose` installed and working.

Edit `output.json`

Because we are planning to ingest into HDFS and not to the filesystem, we must modify the `output.json` file that we used previously. Edit `/tmp/json/output.json` so that it looks like this:

```
{
    "backend": {
        "type": "hadoop",
        "path": "hdfs://hdfs-name/catalog/"
    },
    "reprojectMethod": "buffered",
    "pyramid": true,
    "tileSize": 256,
    "keyIndexMethod": {
        "type": "zorder"
    },
    "resampleMethod": "cubic-spline",
    "layoutScheme": "zoomed",
    "crs": "EPSG:3857"
}
```

The only change is the value associated with the `path` key; it now points into HDFS instead of at the local filesystem.

Download `docker-compose.yml` File

We must now obtain a `docker-compose.yml` file. Download this file and move it to the `/tmp` directory. The directory location is important, because `docker-compose` will use that to name the network and containers that it creates.

Bring Up GeoDocker

With the `docker-compose.yml` file in place, we are now ready to start our GeoDocker instance:

```
cd /tmp
docker-compose up
```

After a period of time, the various Hadoop containers should be up and working.

Perform the Ingest

In a different terminal, we will now start another container:

```
docker run -it --rm --net=tmp_default -v $SPARK_HOME:/spark:ro -v /tmp:/tmp openjdk:8-
→jdk bash
```

Notice that the network name was derived from the name of the directory in which the `docker-compose up` command was run. The `--net=tmp_default` switch connects the just-started container to the bridge network that the GeoDocker cluster is running on. The `-v $SPARK_HOME:/spark:ro` switch mounts our local Spark installation at `/spark` within the container so that we can use it. The `-v /tmp:/tmp` switch mounts our host `/tmp` directory into the container so that we can use the data and jar files that are there.

Within the just-started container, we can now perform the ingest:

```
/spark/bin/spark-submit \
   --class geotrellis.spark.etl.MultibandIngest \
   --master 'local[*]' \
   --driver-memory 16G \
   /tmp/geotrellis-spark-etl-assembly-1.0.0.jar \
   --input "file:///tmp/json/input.json" \
```

(continues on next page)

```
    --output "file:///tmp/json/output.json" \
    --backend-profiles "file:///tmp/json/backend-profiles.json"
```

The only change versus what we did earlier is the location of the `spark-submit` binary.

Optional: Exercise the Catalog

Now, we can exercise the catalog:

```
rm -f /tmp/tif/*.tif
/spark/bin/spark-submit \
    --class com.azavea.geotrellis.tutorial.EtlExercise \
    --master 'local[*]' \
    --driver-memory 16G \
    /tmp/tutorial-assembly-0.jar /tmp/catalog example 12
```

The only differences form what we did earlier are the location of the `spark-submit` binary and URI specifying the location of the catalog.

3.8 Core Concepts

Geographical Information Systems (GIS), like any specialized field, has a wealth of jargon and unique concepts. When represented in software, these concepts can sometimes be skewed or expanded from their original forms. We give a thorough definition of many of the core concepts here, while referencing the Geotrellis objects and source files backing them.

This document aims to be informative to new and experienced GIS users alike. If GIS is brand, brand new to you, this document is a useful high level overview.

3.8.1 Basic Terms

- **Tile:** A grid of numeric *cells* that represent some data on the Earth.

- **Cell:** A single unit of data in some grid, also called a *Location* in GIS.

- **Layer:** or "Tile Layer", this is a grid (or cube) of *Tiles*.

- **Zoom Layer:** a *Tile Layer* at some zoom level.

- **Key:** Used to index a *Tile* in a grid (or cube) of them.

- **Key Index:** Used to transform higher-dimensional *Keys* into one dimension.

- **Metadata:** or "Layer Metadata", stores information critical to Tile Layer IO.

- **Layout Definition:** A description of a Tile grid (its dimensions, etc).

- **Extent:** or "Bounding Box", represents some area on the Earth.

- **Raster:** A *Tile* with an *Extent*.

- **Vector:** or "Geometry", these are Point, Line, and Polygon data.

- **Feature:** A *Geometry* with some associated metadata.

- **RDD:** "Resilient Distributed Datasets" from Apache Spark. Can be thought of as a highly distributed Scala `Seq`.

These definitions are expanded upon in other sections of this document.

3.8.2 Tile Layers

Tile layers (of Rasters or otherwise) are represented in GeoTrellis with the type `RDD[(K, V)] with Metadata[M]`. This type is used extensively across the code base, and its contents form the deepest compositional hierarchy we have:
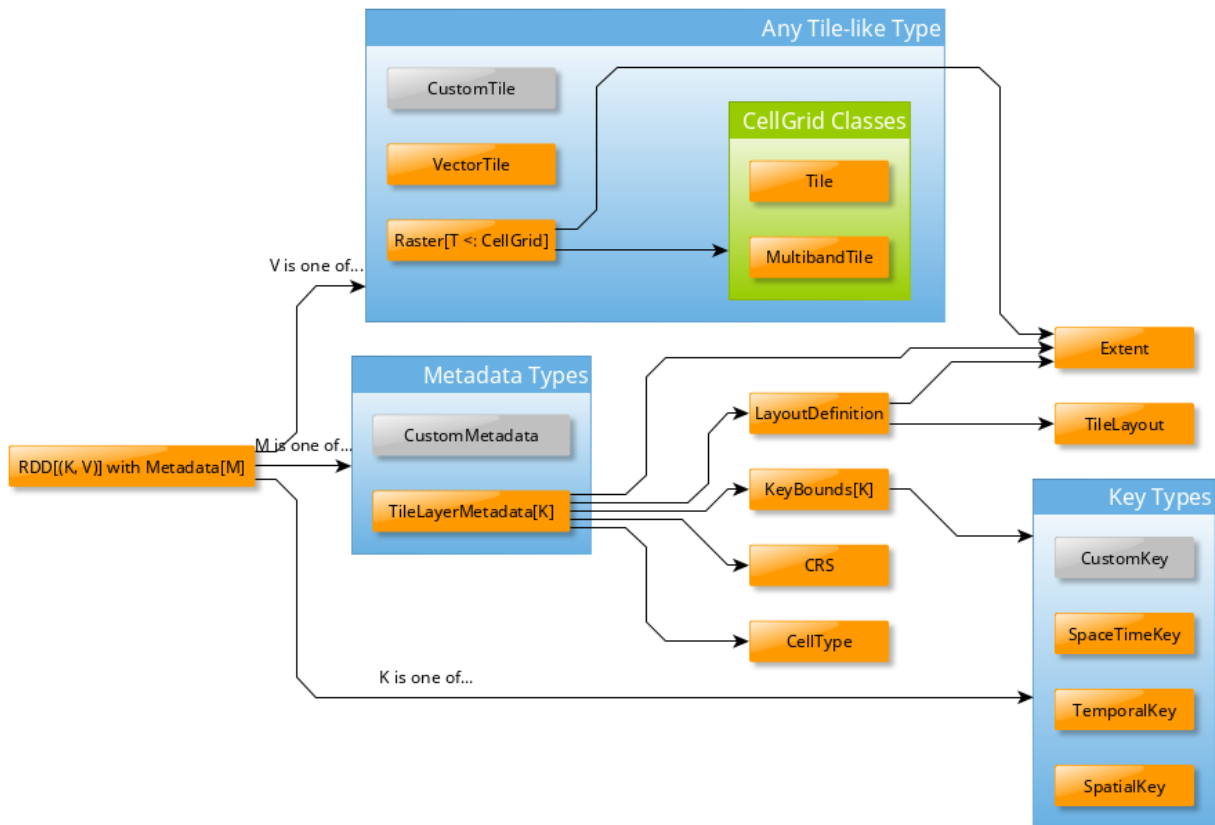
In this diagram:

- `CustomTile`, `CustomMetadata`, and `CustomKey` don't exist, they represent types that you could write yourself for your application.

- The `K` seen in several places is the same `K`.

- The type `RDD[(K, V)] with Metadata[M]` is a Scala *Anonymous Type*. In this case, it means `RDD` from Apache Spark with extra methods injected from the `Metadata` trait. This type is sometimes aliased in GeoTrellis as `ContextRDD`.

- `RDD[(K, V)]` resembles a Scala `Map[K, V]`, and in fact has further `Map`-like methods injected by Spark when it takes this shape. See Spark's PairRDDFunctions Scaladocs for those methods. **Note:** Unlike `Map`, the `K`s here are **not** guaranteed to be unique.

TileLayerRDD

A common specification of `RDD[(K, V)] with Metadata[M]` in GeoTrellis is as follows:

```
type TileLayerRDD[K] = RDD[(K, Tile)] with Metadata[TileLayerMetadata[K]]
```

This type represents a grid (or cube!) of `Tiles` on the earth, arranged according to some `K`. Features of this grid are:

- Grid location (`0, 0`) is the top-leftmost `Tile`.

- The `Tiles` exist in *some* CRS. In `TileLayerMetadata`, this is kept track of with an actual `CRS` field.

- In applications, `K` is mostly `SpatialKey` or `SpaceTimeKey`.

Tile Layer IO

Layer IO requires a Tile Layer Backend. Each backend has an `AttributeStore`, a `LayerReader`, and a `LayerWriter`.

Example setup (with our `File` system backend):

```scala
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.io.file._

val catalogPath: String = ...  /* Some location on your computer */

val store: AttributeStore = FileAttributeStore(catalogPath)

val reader = FileLayerReader(store)
val writer = FileLayerWriter(store)
```

Writing an entire layer:

```scala
/* Zoom level 13 */
val layerId = LayerId("myLayer", 13)

/* Produced from an ingest, etc. */
val rdd: TileLayerRDD[SpatialKey] = ...

/* Order your Tiles according to the Z-Curve Space Filling Curve */
val index: KeyIndex[SpatialKey] = ZCurveKeyIndexMethod.createIndex(rdd.metadata.
→bounds)

/* Returns `Unit` */
writer.write(layerId, rdd, index)
```

Reading an entire layer:

```scala
/* `.read` has many overloads, but this is the simplest */
val sameLayer: TileLayerRDD[SpatialKey] = reader.read(layerId)
```

Querying a layer (a "filtered" read):

```scala
/* Some area on the earth to constrain your query to */
val extent: Extent = ...

/* There are more types that can go into `where` */
val filteredLayer: TileLayerRDD[SpatialKey] =
  reader.query(layerId).where(Intersects(extent)).result
```

3.8.3 Keys and Key Indexes

Keys

As mentioned in the *Tile Layers* section, grids (or cubes) of `Tiles` on the earth are organized by keys. This key, often refered to generically as `K`, is typically a `SpatialKey` or a `SpaceTimeKey`:

```scala
case class SpatialKey(col: Int, row: Int)

case class SpaceTimeKey(col: Int, row: Int, instant: Long)
```

although there is nothing stopping you from defining your own key type.

Assuming some tile layer `Extent` on the earth, `SpatialKey(0, 0)` would index the top-leftmost `Tile` in the Tile grid.

When doing Layer IO, certain optimizations can be performed if we know that `Tiles` stored near each other in a filesystem or database (like Accumulo or HBase) are also spatially-close in the grid they're from. To make such a guarantee, we use a `KeyIndex`.

Key Indexes

A `KeyIndex` is a GeoTrellis `trait` that represents Space Filling Curves. They are a means by which to translate multi-dimensional indices into a single-dimensional one, while maintaining spatial locality. In GeoTrellis, we use these chiefly when writing Tile Layers to one of our Tile Layer Backends.

Although `KeyIndex` is often used in its generic `trait` form, we supply three underlying implementations.

Z-Curve

The Z-Curve is the simplest `KeyIndex` to use (and implement). It can be used with both `SpatialKey` and `SpaceTimeKey`.

```scala
val b0: KeyBounds[SpatialKey] = ... /* from `TileLayerRDD.metadata.bounds` */
val b1: KeyBounds[SpaceTimeKey] = ...

val i0: KeyIndex[SpatialKey] = ZCurveKeyIndexMethod.createIndex(b0)
val i1: KeyIndex[SpatialKey] = ZCurveKeyIndexMethod.byDay().createIndex(b1)

val k: SpatialKey = ...
val oneD: Long = i0.toIndex(k) /* A SpatialKey's 2D coords mapped to 1D */
```

Hilbert

Another well-known curve, available for both `SpatialKey` and `SpaceTimeKey`.

```scala
val b: KeyBounds[SpatialKey] = ...

val index: KeyIndex[SpatialKey] = HilbertKeyIndexMethod.createIndex(b)
```

Index Resolution Changes Index Order

Changing the resolution (in bits) of the index causes a rotation and/or reflection of the points with respect to curve-order. Take, for example the following code (which is actually derived from the testing codebase):

```
HilbertSpaceTimeKeyIndex(SpaceTimeKey(0,0,y2k), SpaceTimeKey(2,2,y2k.plusMillis(1)),2,
→1)
```

The last two arguments are the index resolutions. If that were changed to:

```
HilbertSpaceTimeKeyIndex(SpaceTimeKey(0,0,y2k), SpaceTimeKey(2,2,y2k.plusMillis(1)),3,
→1)
```

The index-order of the points would be different. The reasons behind this are ultimately technical, though you can imagine how a naive implementation of an index for, say, a 10x10 matrix (in terms of 100 numbers) would need to be reworked if you were to change the number of cells (100 would no longer be enough for an 11x11 matrix and the pattern for indexing you chose may no longer make sense). Obviously, this is complex and beyond the scope of GeoTrellis' concerns, which is why we lean on Google's `uzaygezen` library.

Beware the 62-bit Limit

Currently, the spatial and temporal resolution required to index the points, expressed in bits, must sum to 62 bits or fewer.

For example, the following code appears in `HilbertSpaceTimeKeyIndex.scala`:

```scala
@transient
lazy val chc = {
  val dimensionSpec =
    new MultiDimensionalSpec(
      List(
        xResolution,
        yResolution,
        temporalResolution
      ).map(new java.lang.Integer(_))
    )
}
```

where `xResolution`, `yResolution` and `temporalResolution` are numbers of bits required to express possible locations in each of those dimensions. If those three integers sum to more than 62 bits, an error will be thrown at runtime.

Row Major

Row Major is only available for `SpatialKey`, but provides the fastest `toIndex` lookup of the three curves. It doesn't however, give good locality guarantees, so should only be used when locality isn't as important to your application.

```scala
val b: KeyBounds[SpatialKey] = ...

val index: KeyIndex[SpatialKey] = RowMajorKeyIndexMethod.createIndex(b)
```
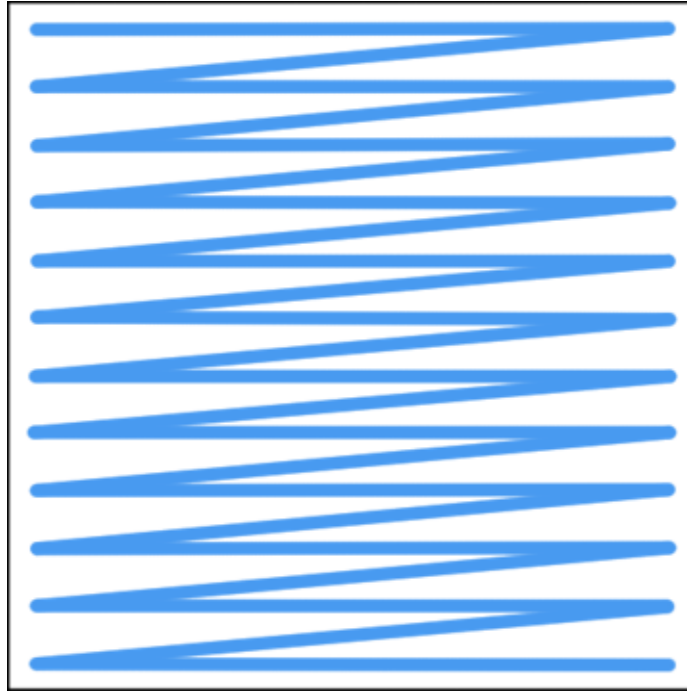
3.8.4 Tiles

`Tile` is a core GeoTrellis primitive. As mentioned in *Tile Layers*, a common specification of `RDD[(K, V)]` with `Metadata[M]` is:

```
type TileLayerRDD[K] = RDD[(K, Tile)] with Metadata[TileLayerMetadata[K]]
```

What is a `Tile` exactly? Below is a diagram of our `Tile` type hierarchy. As you can see, any `Tile` (via `CellGrid`) is effectively a grid of data cells:

The `Tile` trait has operations you'd expect for traversing and transforming this grid, like:

* `map:  (Int => Int) => Tile`

* `foreach:  (Int => Unit) => Unit`

* `combine:  Tile => ((Int, Int) => Int) => Tile`

* `color:  ColorMap => Tile`

Critically, a `Tile` must know how big it is, and what its underlying *Cell Type* is:

* `cols:  Int`

* `rows:  Int`

* `cellType:  CellType`

Fundamentally, the union of a `Tile` and `Extent` is how GeoTrellis defines a `Raster`:

```
case class Raster[+T <: CellGrid](tile: T, extent: Extent) extends CellGrid
```

For performance reasons, we have opted for `Tile` to hold its `CellType` as opposed to making `Tile` polymorphic on its underlying numeric type, for example like `trait Tile[T]`. The large type hierarchy above is what results from this decision. For more information, see our notes on Tile performance.
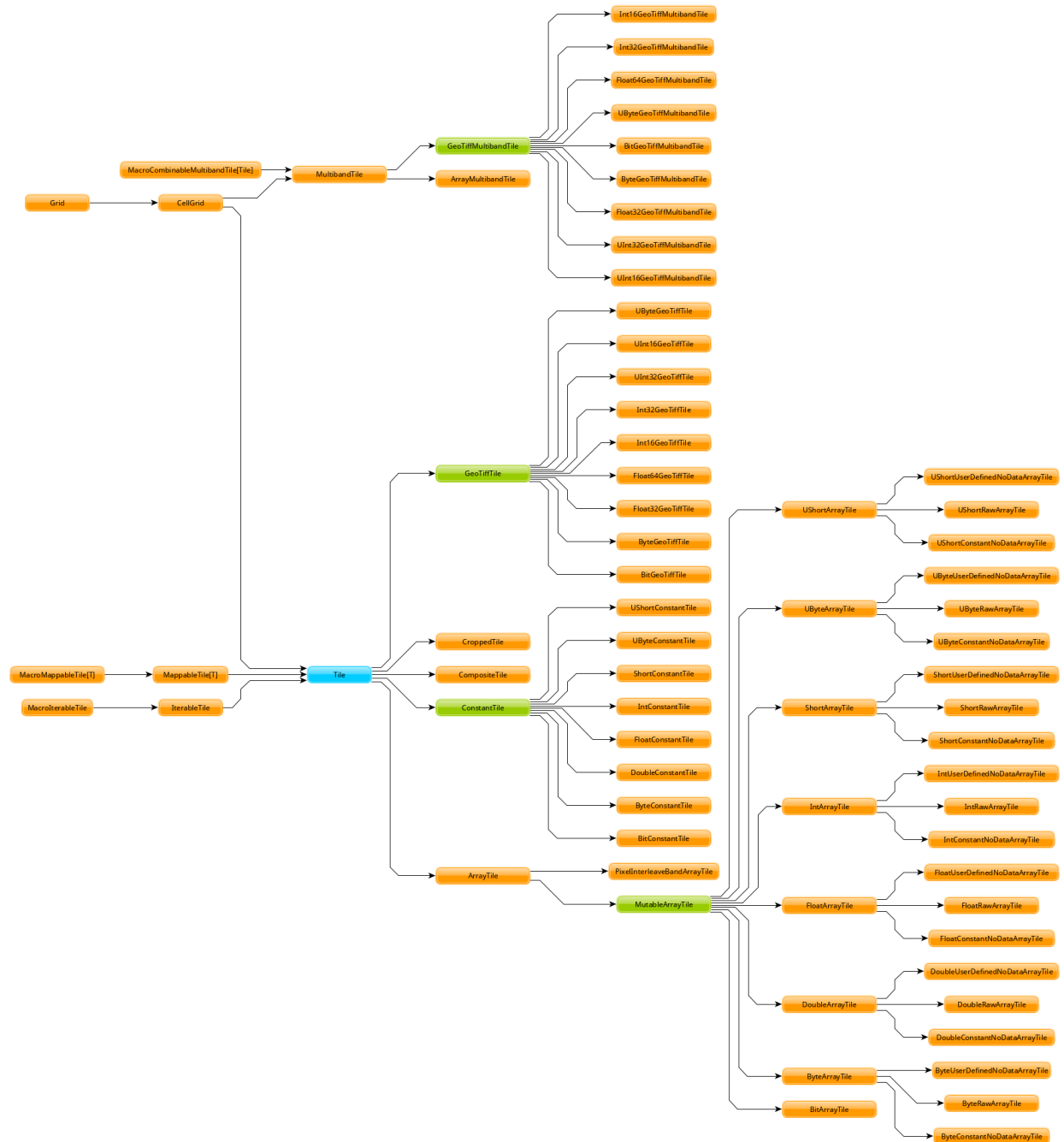
3.8.5 Cell Types

What is a Cell Type?

- A `CellType` is a data type plus a policy for handling cell values that may contain no data.

- By 'data type' we shall mean the underlying numerical representation of a `Tile`'s cells.

- `NoData`, for performance reasons, is not represented as a value outside the range of the underlying data type (as, e.g., `None`) - if each cell in some tile is a `Byte`, the `NoData` value of that tile will exist within the range [`Byte.MinValue` (-128), `Byte.MaxValue` (127)].

- If attempting to convert between `CellTypes`, see this note on `CellType` conversions.

	No NoData	Constant NoData	User Defined NoData
BitCells	`BitCellType`	N/A	N/A
ByteCells	`ByteCellType`	`ByteConstantNoDataCellType`	`ByteUserDefinedNoDataCellType`
Ubyte-Cells	`UByteCellType`	`UByteConstantNoDataCellType`	`UByteUserDefinedNoDataCellType`
ShortCells	`ShortCellType`	`ShortConstantNoDataCellType`	`ShortUserDefinedNoDataCellType`
UShort-Cells	`UShortCellType`	`UShortConstantNoDataCellType`	`UShortUserDefinedNoDataCellType`
IntCells	`IntCellType`	`IntConstantNoDataCellType`	`IntUserDefinedNoDataCellType`
FloatCells	`FloatCellType`	`FloatConstantNoDataCellType`	`FloatUserDefinedNoDataCellType`
Double-Cells	`DoubleCellType`	`DoubleConstantNoDataCellType`	`DoubleUserDefinedNoDataCellType`

The above table lists `CellType DataTypes` in the leftmost column and `NoData` policies along the top row. A couple of points are worth making here:

1. Bits are incapable of representing on, off, *and* some `NoData` value. As a consequence, there is no such thing as a Bit-backed tile which recognizes `NoData`.

2. While the types in the 'No NoData' and 'Constant NoData' are simply singleton objects that are passed around alongside tiles, the greater configurability of 'User Defined NoData' `CellTypes` means that they require a constructor specifying the value which will count as `NoData`.

Let's look to how this information can be used:

```scala
/** Here's an array we'll use to construct tiles */
val myData = Array(42, 1, 2, 3)

/** The GeoTrellis-default integer CellType
 *   Note that it represents `NoData` values with the smallest signed
 *   integer possible with 32 bits (Int.MinValue or -2147483648).
 */
val defaultCT = IntConstantNoDataCellType
val normalTile = IntArrayTile(myData, 2, 2, defaultCT)

/** A custom, 'user defined' NoData CellType for comparison; we will
 *    treat 42 as NoData for this one rather than Int.MinValue
 */
val customCellType = IntUserDefinedNoDataValue(42)
val customTile = IntArrayTile(myData, 2, 2, customCellType)

/** We should expect that the first (default celltype) tile has the value 42 at (0, 0)
 *    This is because 42 is just a regular value (as opposed to NoData)
```

```
 *    which means that the first value will be delivered without surprise
 */
assert(normalTile.get(0, 0) == 42)
assert(normalTile.getDouble(0, 0) == 42.0)

/** Here, the result is less obvious. Under the hood, GeoTrellis is
 *    inspecting the value to be returned at (0, 0) to see if it matches our
 *    `NoData` policy and, if it matches (it does, we defined NoData as
 *    42 above), return Int.MinValue (no matter your underlying type, `get`
 *    on a tile will return an `Int` and `getDouble` will return a `Double`).
 *
 *    The use of Int.MinValue and Double.NaN is a result of those being the
 *    GeoTrellis-blessed values for NoData - below, you'll find a chart that
 *    lists all such values in the rightmost column
 */
assert(customTile.get(0, 0) == Int.MinValue)
assert(customTile.getDouble(0, 0) == Double.NaN)
```

A point which is perhaps not intuitive is that `get` will *always* return an `Int` and `getDouble` will *always* return a `Double`. Representing NoData demands, therefore, that we map other celltypes' `NoData` values to the native, default `Int` and `Double` `NoData` values. `NoData` will be represented as `Int.MinValue` or `Double.Nan`.

Why you should care

In most programming contexts, it isn't all that useful to think carefully about the number of bits necessary to represent the data passed around by a program. A program tasked with keeping track of all the birthdays in an office or all the accidents on the New Jersey turnpike simply doesn't benefit from carefully considering whether the allocation of those extra few bits is *really* worth it. The costs for any lack of efficiency are more than offset by the savings in development time and effort. This insight - that computers have become fast enough for us to be forgiven for many of our programming sins - is, by now, truism.

An exception to this freedom from thinking too hard about implementation details is any software that tries, in earnest, to provide the tools for reading, writing, and working with large arrays of data. Rasters certainly fit the bill. Even relatively modest rasters can be made up of millions of underlying cells. Additionally, the semantics of a raster imply that each of these cells shares an underlying data type. These points - that rasters are made up of a great many cells and that they all share a backing data type - jointly suggest that a decision regarding the underlying data type could have profound consequences. More on these consequences *below*.

Compliance with the GeoTIFF standard is another reason that management of cell types is important for GeoTrellis. The most common format for persisting a raster is the GeoTIFF. A GeoTIFF is simply an array of data along with some useful tags (hence the 'tagged' of 'tagged image file format'). One of these tags specifies the size of each cell and how those bytes should be interpreted (i.e. whether the data for a byte includes its sign - positive or negative - or whether it counts up from 0 - and is therefore said to be 'unsigned').

In addition to keeping track of the memory used by each cell in a `Tile`, the cell type is where decisions about which values count as data (and which, if any, are treated as `NoData`). A value recognized as `NoData` will be ignored while mapping over tiles, carrying out focal operations on them, interpolating for values in their region, and just about all of the operations provided by GeoTrellis for working with `Tiles`.

Cell Type Performance

There are at least two major reasons for giving some thought to the types of data you'll be working with in a raster: persistence and performance.

Persistence is simple enough: smaller datatypes end up taking less space on disk. If you're going to represent a region with only `true`/`false` values on a raster whose values are `Doubles`, 63/64 bits will be wasted. Naively, this means somewhere around 63 times less data than if the most compact form possible had been chosen (the use of `BitCells` would be maximally efficient for representing the bivalent nature of boolean values). See the chart below for a sense of the relative sizes of these cell types.

The performance impacts of cell type selection matter in both a local and a distributed (spark) context. Locally, the memory footprint will mean that as larger cell types are used, smaller amounts of data can be held in memory and worked on at a given time and that more CPU cache misses are to be expected. This latter point - that CPU cache misses will increase - means that more time spent shuffling data from the memory to the processor (which is often a performance bottleneck). When running programs that leverage spark for compute distribution, larger data types mean more data to serialize and more data send over the (very slow, relatively speaking) network.

In the chart below, `DataTypes` are listed in the leftmost column and important characteristics for deciding between them can be found to the right. As you can see, the difference in size can be quite stark depending on the cell type that a tile is backed by. That extra space is the price paid for representing a larger range of values. Note that bit cells lack the sufficient representational resources to have a `NoData` value.

	Bits / Cell	512x512 Raster (mb)	Range (inclusive)	GeoTrellis NoData Value
BitCells	1	0.032768	[0, 1]	N/A
ByteCells	8	0.262144	[-128, 128]	-128 (`Byte.MinValue`)
UbyteCells	8	0.262144	[0, 255]	0
ShortCells	16	0.524288	[-32768, 32767]	-32768 (`Short.MinValue`)
UShortCells	16	0.524288	[0, 65535]	0
IntCells	32	1.048576	[-2147483648, 2147483647]	-2147483648 (`Int.MinValue`)
FloatCells	32	1.048576	[-3.40E38, 3.40E38]	Float.NaN
DoubleCells	64	2.097152	[-1.79E308, 1.79E308]	Double.NaN

One final point is worth making in the context of `CellType` performance: the `Constant` types are able to depend upon macros which inline comparisons and conversions. This minor difference can certainly be felt while iterating through millions and millions of cells. If possible, Constant `NoData` values are to be preferred. For convenience' sake, we've attempted to make the GeoTrellis-blessed `NoData` values as unobtrusive as possible a priori.

The limits of expected return types (discussed in the previous section) is used by macros to squeeze as much speed out of the JVM as possible. Check out our macros docs for more on our use of macros like `isData` and `isNoData`.

3.8.6 Raster Data

> "Yes raster is faster, but raster is vaster and vector just SEEMS more corrector." — C. Dana Tomlin

Rasters and Tiles

The entire purpose of `geotrellis.raster` is to provide primitive datatypes which implement, modify, and utilize rasters. In GeoTrellis, a raster is just a `Tile` with an associated `Extent`. A tile is just a two-dimensional collection of evenly spaced data. Tiles are a lot like certain sequences of sequences (this array of arrays is like a 3x3 tile):

```
// not real syntax
val myFirstTile = [[1,1,1],[1,2,2],[1,2,3]]
/** It probably looks more like your mental model if we stack them up:
```

(continues on next page)

```
 *  [[1,1,1],
 *   [1,2,2],
 *   [1,2,3]]
 */
```

In the `raster` module of GeoTrellis, the base type of tile is just `Tile`. All GeoTrellis compatible tiles will have inherited from that base class, so if you find yourself wondering what a given type of tile's powers are, that's a decent place to start your search. Here's an incomplete list of the types of things on offer:

- Mapping transformations of arbitrary complexity over the constituent cells

- Carrying out operations (side-effects) for each cell

- Querying a specific tile value

- Rescaling, resampling, cropping

As we've already discussed, tiles are made up of squares which contain values. We'll sometimes refer to these value-boxes as *cells*. And, just like cells in the body, though they are discrete units, they're most interesting when looked at from a more holistic perspective - rasters encode relations between values in a uniform space and it is usually these relations which most interest us. The code found in the `mapalgebra` submodule — discussed later in this document — is all about exploiting these spatial relations.

Working with Cell Values

One of the first questions you'll ask yourself when working with GeoTrellis is what kinds of representation best models the domain you're dealing with. What types of value do you need your raster to hold? This question is the province of GeoTrellis `CellType`s.

Building Your Own Tiles

With a grasp of tiles and `CellType`s, we've got all the conceptual tools necessary to construct our own tiles. Now, since a tile is a combination of a `CellType` with which its cells are encoded and their spatial arrangement, we will have to somehow combine `Tile` (which encodes our expectations about how cells sit with respect to one another) and the datatype of our choosing. Luckily, GeoTrellis has done this for us. To keep its users sane, the wise maintainers of GeoTrellis have organized `geotrellis.raster` such that fully reified tiles sit at the bottom of an pretty simple inheritance chain. Let's explore that inheritance so that you will know where to look when your intuitions lead you astray:

From `IntArrayTile.scala`:

```scala
abstract class IntArrayTile(
  val array: Array[Int],
  cols: Int,
  rows: Int
) extends MutableArrayTile { ... }
```

From `DoubleArrayTile.scala`:

```scala
abstract class DoubleArrayTile(
  val array: Array[Double],
  cols: Int,
  rows: Int
) extends MutableArrayTile { ... }
```

Tile Inheritance Structure

Both `IntArrayTile` and `DoubleArrayTile` are themselves extended by other child classes, but they are a good place to start. Critically, they are both `MutableArrayTiles`, which adds some nifty methods for in-place manipulation of cells (GeoTrellis is about performance, so this minor affront to the gods of immutability can be forgiven). From MutableArrayTile.scala:

```scala
trait MutableArrayTile extends ArrayTile { ... }
```

One level up is `ArrayTile`. It's handy because it implements the behavior which largely allows us to treat our tiles like big, long arrays of (arrays of) data. They also have the trait `Serializable`, which is neat any time you can't completely conduct your business within the neatly defined space-time of the JVM processes which are running on a single machine (this is the point of GeoTrellis' Spark integration). From ArrayTile.scala:

```scala
trait ArrayTile extends Tile with Serializable { ... }
```

At the top rung in our abstraction ladder we have `Tile`. You might be surprised how much we can say about tile behavior from the base of its inheritance tree, so the source is worth reading directly. From Tile.scala:

```scala
trait Tile extends CellGrid with ... { ... }
```

Where `CellGrid` and its parent `Grid` just declare something to be - you guessed it - a grid of numbers with an explicit `CellType`.

As it turns out, `CellType` is one of those things that we can *mostly* ignore once we've settled on which one is proper for our domain. After all, it appears as though there's very little difference between tiles that prefer int-like things and tiles that prefer double-like things.

> **CAUTION**: While it is true, in general, that operations are `CellType` agnostic, both `get` and `getDouble` are methods implemented on `Tile`. In effect, this means that you'll want to be careful when querying values. If you're working with int-like `CellTypes`, probably use `get`. If you're working with float-like `CellTypes`, usually you'll want `getDouble`.

Raster Examples

In the repl, you can try this out to construct a simple `Raster`:

```scala
import geotrellis.raster._
import geotrellis.vector._

scala> IntArrayTile(Array(1,2,3),1,3)
res0: geotrellis.raster.IntArrayTile = IntArrayTile([S@338514ad,1,3)

scala> IntArrayTile(Array(1,2,3),3,1)
res1: geotrellis.raster.IntArrayTile = IntArrayTile([S@736a81de,3,1)

scala> IntArrayTile(Array(1,2,3,4,5,6,7,8,9),3,3)
res2: geotrellis.raster.IntArrayTile = IntArrayTile([I@5466441b,3,3)

scala> Extent(0, 0, 1, 1)
res4: geotrellis.vector.Extent = Extent(0.0,0.0,1.0,1.0)

scala> Raster(res2, res4)
res5: geotrellis.raster.Raster = Raster(IntArrayTile([I@7b47ab7,1,3),Extent(0.0,0.0,1.
→0,1.0))
```

Here's a fun method for exploring your tiles:

```
scala> res0.asciiDraw()
res3: String =
"    1
     2
     3
"


scala> res2.asciiDraw()
res4: String =
"    1     2     3
     4     5     6
     7     8     9
"
```

That's probably enough to get started. `geotrellis.raster` is a pretty big place, so you'll benefit from spending a few hours playing with the tools it provides.

3.8.7 Vector Data

> "Raster is faster but vector is correcter." — Somebody

Features and Geometries

In addition to working with raster data, Geotrellis provides a number of tools for the creation, representation, and modification of vector data. The data types central to this functionality (`geotrellis.vector.Feature` and `geotrellis.vector.Geometry`) correspond - and not by accident - to certain objects found in the GeoJson spec. `Features` correspond to the objects listed under `features` in a geojson `FeatureCollection`. `Geometrys`, to `geometries` in a geojson `Feature`.

Geometries

The base `Geometry` class can be found in `Geometry.scala`. Concrete geometries include:

- `geotrellis.vector.Point`
- `geotrellis.vector.MultiPoint`
- `geotrellis.vector.Line`
- `geotrellis.vector.MultiLine`
- `geotrellis.vector.Polygon`
- `geotrellis.vector.MultiPolygon`
- `geotrellis.vector.GeometryCollection`

Working with these geometries is a relatively straightforward affair. Let's take a look:

```scala
import geotrellis.vector._

/** First, let's create a Point. Then, we'll use its intersection method.
  * Note: we are also using intersection's alias '&'.
  */
val myPoint = Point(1.0, 1.1) // Create a point
// Intersection method
```

(continues on next page)

```scala
val selfIntersection = myPoint intersection Point(1.0, 1.1)
// Intersection alias
val nonIntersection = myPoint & Point(200, 300)
```

At this point, the values `selfIntersection` and `nonIntersection` are `GeometryResult` containers. These containers are what many JTS operations on `Geometry` objects will wrap their results in. To idiomatically destructure these wrappers, we can use the `as[G <:  Geometry]` function which either returns `Some(G)` or `None`.

```scala
val pointIntersection = (Point(1.0, 2.0) & Point(1.0, 2.0)).as[Point]
val pointNonIntersection = (Point(1.0, 2.0) & Point(12.0, 4.0)).as[Point]

assert(pointIntersection == Some(Point(1.0, 2.0)))  // Either some point
assert(pointNonIntersection == None)                // Or nothing at all
```

As convenient as `as[G <:  Geometry]` is, it offers no guarantees about the domain over which it ranges. So, while you can expect a neatly packaged `Option[G <:  Geometry]`, it isn't necessarily the case that the `GeometryResult` object produced by a given set of operations is possibly convertable to the `Geometry` subtype you choose. For example, a `PointGeometryIntersectionResult.as[Polygon]` will *always* return `None`.

An alternative approach uses pattern matching and ensures an exhaustive check of the results. `geotrellis.vector.Results` contains a large [ADT](#) which encodes the possible outcomes for different types of outcomes. The result type of a JTS-dependent vector operation can be found somewhere on this tree to the effect that an exhaustive match can be carried out to determine the `Geometry` (excepting cases of `NoResult`, for which there is no `Geometry`).

For example, we note that a `Point`/`Point` intersection has the type `PointOrNoResult`. From this we can deduce that it is either a `Point` underneath or else nothing:

```scala
scala> import geotrellis.vector._
scala> p1 & p2 match {
     |    case PointResult(_) => println("A Point!")
     |    case NoResult => println("Sorry, no result.")
     | }
A Point!
```

Beyond the methods which come with any `Geometry` object there are implicits in many geotrellis modules which will extend Geometry capabilities. For instance, after importing `geotrellis.vector.io._`, it becomes possible to call the `toGeoJson` method on any `Geometry`:

```scala
import geotrellis.vector.io._
assert(Point(1,1).toGeoJson == """{"type":"Point","coordinates":[1.0,1.0]}""")
```

If you need to move from a geometry to a serialized representation or vice-versa, take a look at the `io` directory's contents. This naming convention for input and output is common throughout Geotrellis. So if you're trying to get spatial representations in or out of your program, spend some time seeing if the problem has already been solved.

Methods which are specific to certain subclasses of `Geometry` exist too. For example, `geotrellis.vector.MultiLine` is implicitly extended by `geotrellis.vector.op` such that this becomes possible:

```scala
import geotrellis.vector.op._
val myML = MultiLine.EMPTY
myML.unionGeometries
```

The following packages extend `Geometry` capabilities:

- geotrellis.vector.io.json

- geotrellis.vector.io.WKT

- geotrellis.vector.io.WKB

- geotrellis.vector.op

- geotrellis.vector.op.affine

- geotrellis.vector.reproject

Features

The `Feature` class is odd at first glance; it thinly wraps one of the afforementioned `Geometry` objects along with some type of data. Its purpose will be clear if you can keep in mind the importance of the geojson format of serialization which is now ubiquitous in the GIS software space. It can be found in `Feature.scala`.

Let's examine some source code so that this is all a bit clearer. From `geotrellis.vector.Feature.scala`:

```scala
abstract class Feature[D] {
  type G <: Geometry
  val geom: G ; val data: D
}

case class PointFeature[D](geom: Point, data: D) extends Feature[D] {type G = Point}
```

These type signatures tell us a good deal. Let's make this easy on ourselves and put our findings into a list. - The type G is some instance or other of `Geometry` (which we explored just above).

- The value, `geom`, which anything the compiler recognizes as a `Feature` must make available in its immediate closure must be of type `G`.

- As with `geom` the compiler will not be happy unless a `Feature` provides `data`.

- Whereas, with `geom`, we could say a good deal about the types of stuff (only things we call geometries) that would satisfy the compiler, we have nothing in particular to say about `D`.

Our difficulty with `D` is shared by the `Point`-focused feature, `PointFeature`. `PointFeature` uses `Point` (which is one of the concrete instances of `Geometry` introduced above) while telling us nothing at all about `data`'s type. This is just sugar for passing around a `Point` and some associated metadata.

Let's look at some code which does something with D (code which calls one of D's methods) so that we know what to expect. Remember: types are just contracts which the compiler is kind enough to enforce. In well-written code, types (and type variables) can tell us a great deal about what was in the head of the author.

There's only one `package` which does anything with `D`, so the constraints (and our job) should be relatively easy. In `geotrellis.vector.io.json.FeatureFormats` there are `ContextBounds` on `D` which ensure that they have JsonReader, JsonWriter, and JsonFormat implicits available (this is a typeclass, and it allows for something like type-safe duck-typing).

`D`'s purpose is clear enough: any `D` which comes with the tools necessary for json serialization and deserialization will suffice. In effect, `data` corresponds to the "properties" member of the geojson spec's `Feature` object.

If you can provide the serialization tools (that is, implicit conversions between some type (your `D`) and spray json), the `Feature` object in `geotrellis.vector` does the heavy lifting of embedding your (thus serializable) data into the larger structure which includes a `Geometry`. There's even support for geojson IDs: the "ID" member of a geojson Feature is represented by the keys of a `Map` from `String` to `Feature[D]`. Data in both the ID and non-ID variants of geojson Feature formats is easily transformed.

Submodules

These submodules define useful methods for dealing with the entities that call `geotrellis.vector` home:

- `geotrellis.vector.io` defines input/output (serialization) of geometries
- `geotrellis.vector.op` defines common operations on geometries
- `geotrellis.vector.reproject` defines methods for translating between projections

3.8.8 Catalogs

We call the basic output of an ingest a **Layer**, and many GeoTrellis operations *follow this idea*. Layers may be written in related groups we call **Pyramids**, which are made up of interpolations/extrapolations of some base Layer (i.e. different zoom levels). Finally, collections of Pyramids (or just single Layers) can be grouped in a **Catalog** in an organized fashion that allows for logical querying later.

While the term "Catalog" is not as pervasive as "Layer" in the GeoTrellis API, it deserves mention nonetheless as Catalogs are the result of normal GeoTrellis usage.

Catalog Organization

Our Landsat Tutorial produces a simple single-pyramid catalog on the filesystem at `data/catalog/` which we can use here as a reference. Running `tree -L 2` gives us a view of the directory layout:

```
.
├── attributes
│   ├── landsat__.__0__.__metadata.json
│   ├── landsat__.__10__.__metadata.json
│   ├── landsat__.__11__.__metadata.json
│   ├── landsat__.__12__.__metadata.json
│   ├── landsat__.__13__.__metadata.json
│   ├── landsat__.__1__.__metadata.json
│   ├── landsat__.__2__.__metadata.json
│   ├── landsat__.__3__.__metadata.json
│   ├── landsat__.__4__.__metadata.json
│   ├── landsat__.__5__.__metadata.json
│   ├── landsat__.__6__.__metadata.json
│   ├── landsat__.__7__.__metadata.json
│   ├── landsat__.__8__.__metadata.json
│   └── landsat__.__9__.__metadata.json
└── landsat
    ├── 0
    ├── 1
    ├── 10
    ├── 11
    ├── 12
    ├── 13
    ├── 2
    ├── 3
    ├── 4
    ├── 5
    ├── 6
    ├── 7
    ├── 8
    └── 9
```

```
16 directories, 14 files
```

The children of `landsat/` are directories, but we used `-L 2` to hide their contents. They actually contain thousands of `Tile` files, which are explained below.

Metadata

The metadata JSON files contain familiar information:

```
$ jshon < lansat__.__6__.__metadata.json
  [
    {
      "name": "landsat",
      "zoom": 6
    },
    {
      "header": {
        "format": "file",
        "keyClass": "geotrellis.spark.SpatialKey",
        "valueClass": "geotrellis.raster.MultibandTile",
        "path": "landsat/6"
      },
      "metadata": {
        "extent": {
          "xmin": 15454940.911194608,
          "ymin": 4146935.160646211,
          "xmax": 15762790.223459147,
          "ymax": 4454355.929947533
        },
        "layoutDefinition": { ... }
      },
      ... // more here
      "keyIndex": {
        "type": "zorder",
        "properties": {
          "keyBounds": {
            "minKey": { "col": 56, "row": 24 },
            "maxKey": { "col": 57, "row": 25 }
          }
        }
      },
      ... // more here
    }
  ]
```

Of note is the `header` block, which tells GeoTrellis where to look for and how to interpret the stored `Tiles`, and the `keyIndex` block which is critical for reading/writing specific ranges of tiles. For more information, see our *section on Key Indexes*.

As we have multiple storage backends, `header` can look different. Here's an example for a Layer ingested to S3:

```
... // more here
"header": {
   "format": "s3",
```

```
  "key": "catalog/nlcd-tms-epsg3857/6",
  "keyClass": "geotrellis.spark.SpatialKey",
  "valueClass": "geotrellis.raster.Tile",
  "bucket": "azavea-datahub"
},
... // more here
```

Tiles

From above, the numbered directories under `landsat/` contain serialized `Tile` files.

```
$ ls
attributes/  landsat/
$ cd landsat/6/
$ ls
1984  1985  1986  1987
$ du -sh *
12K     1984
8.0K    1985
44K     1986
16K     1987
```

Note: These `Tile` files are not images, but can be rendered by GeoTrellis into PNGs.

Notice that the four `Tile` files here have different sizes. Why might that be, if `Tiles` are all Rasters of the same dimension? The answer is that a `Tile` file can contain multiple tiles. Specifically, it is a serialized `Array[(K, V)]` of which `Array[(SpatialKey, Tile)]` is a common case. When or why multiple `Tiles` might be grouped into a single file like this is the result of the *Space Filling Curve* algorithm applied during ingest.

Separate Stores for Attributes and Tiles

The real story here is that layer attributes and the `Tiles` themselves don't need to be stored via the same backend. Indeed, when instantiating a Layer IO class like `S3LayerReader`, we notice that its `AttributeStore` parameter is type-agnostic:

```
class S3LayerReader(val attributeStore: AttributeStore)
```

So it's entirely possible to store your metadata with one service and your tiles with another. Due to the `header` block in each Layer's metadata, GeoTrellis will know how to fetch the `Tiles`, no matter how they're stored. This arrangement could be more performant/convenient for you, depending on your architecture.

3.8.9 Layout Definitions and Layout Schemes

Data structures: `LayoutDefinition`, `TileLayout`, `CellSize`

A Layout Definition describes the location, dimensions of, and organization of a tiled area of a map. Conceptually, the tiled area forms a grid, and the Layout Definitions describes that grid's area and cell width/height. These definitions can be used to chop a bundle of imagery into tiles suitable for being served out on a web map.

Within the context of GeoTrellis, the `LayoutDefinition` class extends `GridExtent`, and exposes methods for querying the sizes of the grid and grid cells. Those values are stored in the `TileLayout` (the grid description)

---

and `CellSize` classes respectively. `LayoutDefinitions` are used heavily during the raster reprojection process. Within the context of Geotrellis, the `LayoutDefinition` class extends `GridExtent`, and exposes methods for querying the sizes of the grid and grid cells. Those values are stored in the `TileLayout` (the grid description) and `CellSize` classes respectively. `LayoutDefinitions` are used heavily during the raster reprojection process.

What is a Layout Scheme?

The language here can be vexing, but a `LayoutScheme` can be thought of as a factory which produces `LayoutDefinitions`. It is the scheme according to which some layout definition must be defined - a layout definition definition, if you will. The most commonly used `LayoutScheme` is the `ZoomedLayoutScheme`, which provides the ability to generate `LayoutDefinitions` for the different zoom levels of a web-based map (e.g. Leaflet).

How are Layout Definitions used throughout Geotrellis?

Suppose that we've got a distributed collection of `ProjectedExtents` and `Tiles` which cover some contiguous area but which were derived from GeoTIFFs of varying sizes. We will sometimes describe operations like this as 'tiling'. The method which tiles a collection of imagery provided a `LayoutDefinition`, the underlying `CellType` of the produced tiles, and the `ResampleMethod` to use for generating data at new resolutions is `tileToLayout`. Let's take a look at its use:

```scala
val sourceTiles: RDD[(ProjectedExtent, Tile)] = ??? // Tiles from GeoTIFF
val cellType: CellType = IntCellType
val layout: LayoutDefinition = ???
val resamp: ResampleMethod = NearestNeighbor

val tiled: RDD[(SpatialKey, Tile)] =
  tiles.tileToLayout[SpatialKey](cellType, layout, resamp)
```

In essence, a `LayoutDefinition` is the minimum information required to describe the tiling of some map's area in Geotrellis. The `LayoutDefinition` class extends `GridExtent`, and exposes methods for querying the sizes of the grid and grid cells. Those values are stored in the `TileLayout` (the grid description) and `CellSize` classes respectively. `LayoutDefinitions` are most often encountered in raster reprojection processes.

3.8.10 Map Algebra

Map Algebra is a name given by Dr. Dana Tomlin in the 1980's to a way of manipulating and transforming raster data. There is a lot of literature out there, not least the book by the guy who "wrote the book" on map algebra, so we will only give a brief introduction here. GeoTrellis follows Dana's vision of map algebra operations, although there are many operations that fall outside of the realm of Map Algebra that it also supports.

Map Algebra operations fall into 3 general categories:

Local Operations

Local operations are ones that only take into account the information of on cell at a time. In the animation above, we can see that the blue and the yellow cell are combined, as they are corresponding cells in the two tiles. It wouldn't matter if the tiles were bigger or smaller - the only information necessary for that step in the local operation is the cell values that correspond to each other. A local operation happens for each cell value, so if the whole bottom tile was blue and the upper tile were yellow, then the resulting tile of the local operation would be green.

Focal Operations

Focal operations take into account a cell, and a neighborhood around that cell. A neighborhood can be defined as a square of a specific size, or include masks so that you can have things like circular or wedge-shaped neighborhoods. In the above animation, the neighborhood is a 5x5 square around the focal cell. The focal operation in the animation is a `focalSum`. The focal value is 0, and all of the other cells in the focal neighborhood; therefore the cell value of the result tile would be 8 at the cell corresponding to the focal cell of the input tile. This focal operation scans through each cell of the raster. You can imagine that along the border, the focal neighborhood goes outside of the bounds of the tile; in this case the neighborhood only considers the values that are covered by the neighborhood. GeoTrellis also supports the idea of an analysis area, which is the GridBounds that the focal operation carries over, in order to support composing tiles with border tiles in order to support distributed focal operation processing.

Zonal Operations

Zonal operations are ones that operate on two tiles: an input tile, and a zone tile. The values of the zone tile determine what zone each of the corresponding cells in the input tile belong to. For example, if you are doing a `zonalStatistics` operation, and the zonal tile has a distribution of zone 1, zone 2, and zone 3 values, we will get back the statistics such as mean, median and mode for all cells in the input tile that correspond to each of those zone values.

Using Map Algebra Operations

Map Algebra operations are defined as implicit methods on `Tile` or `Traversable[Tile]`, which are imported with `import geotrellis.raster._`.

```scala
import geotrellis.raster._

val tile1: Tile = ???
val tile2: Tile = ???

// If tile1 and tile2 are the same dimensions, we can combine
// them using local operations

tile1.localAdd(tile2)

// There are operators for some local operations.
// This is equivalent to the localAdd call above

tile1 + tile2

// There is a local operation called "reclassify" in literature,
// which transforms each value of the function.
// We actually have a map method defined on Tile,
// which serves this purpose.

tile1.map { z => z + 1 } // Map over integer values.

tile2.mapDouble { z => z + 1.1 } // Map over double values.

tile1.dualMap({ z => z + 1 })({ z => z + 1.1 }) // Call either the integer value or
↪double version, depending on cellType.

// You can also combine values in a generic way with the combine funciton.
```

```
// This is another local operation that is actually defined on Tile directly.

tile1.combine(tile2) { (z1, z2) => z1 + z2 }
```

The following packages are where Map Algebra operations are defined in GeoTrellis:

- geotrellis.raster.mapalgebra.local defines operations which act on a cell without regard to its spatial relations. Need to double every cell on a tile? This is the module you'll want to explore.

- geotrellis.raster.mapalgebra.focal defines operations which focus on two-dimensional windows (internally referred to as neighborhoods) of a raster's values to determine their outputs.

- geotrellis.raster.mapalgebra.zonal defines operations which apply over a zones as defined by corresponding cell values in the zones raster.

Conway's Game of Life can be seen as a focal operation in that each cell's value depends on neighboring cell values. Though focal operations will tend to look at a local region of this or that cell, they should not be confused with the operations which live in `geotrellis.raster.local` - those operations describe transformations over tiles which, for any step of the calculation, need only know the input value of the specific cell for which it is calculating an output (e.g. incrementing each cell's value by 1).

3.8.11 Vector Tiles

Invented by Mapbox, VectorTiles are a combination of the ideas of finite-sized tiles and vector geometries. Mapbox maintains the official implementation spec for VectorTile codecs. The specification is free and open source.

VectorTiles are advantageous over raster tiles in that:

- They are typically smaller to store

- They can be easily transformed (rotated, etc.) in real time

- They allow for continuous (as opposed to step-wise) zoom in Slippy Maps.

Raw VectorTile data is stored in the protobuf format. Any codec implementing the spec must decode and encode data according to this .proto schema.

GeoTrellis provides the `geotrellis-vectortile` module, a high-performance implementation of **Version 2.1** of the VectorTile spec. It features:

- Decoding of **Version 2** VectorTiles from Protobuf byte data into useful Geotrellis types.

- Lazy decoding of Geometries. Only parse what you need!

- Read/write VectorTile layers to/from any of our backends.

As of 2016 November, ingests of raw vector data into VectorTile sets aren't yet possible.

Small Example

```scala
import geotrellis.spark.SpatialKey
import geotrellis.spark.tiling.LayoutDefinition
import geotrellis.vector.Extent
import geotrellis.vectortile.VectorTile
import geotrellis.vectortile.protobuf._

val bytes: Array[Byte] = ...  // from some `.mvt` file
val key: SpatialKey = ...  // preknown
```

```scala
val layout: LayoutDefinition = ...  // preknown
val tileExtent: Extent = layout.mapTransform(key)

/* Decode Protobuf bytes. */
val tile: VectorTile = ProtobufTile.fromBytes(bytes, tileExtent)

/* Encode a VectorTile back into bytes. */
val encodedBytes: Array[Byte] = tile match {
  case t: ProtobufTile => t.toBytes
  case _ => ???  // Handle other backends or throw errors.
}
```

See our VectorTile Scaladocs for detailed usage information.

Implementation Assumptions

This particular implementation of the VectorTile spec makes the following assumptions:

- Geometries are implicitly encoded in ''some" Coordinate Reference system. That is, there is no such thing as a "projectionless" VectorTile. When decoding a VectorTile, we must provide a Geotrellis [[Extent]] that represents the Tile's area on a map. With this, the grid coordinates stored in the VectorTile's Geometry are shifted from their original [0,4096] range to actual world coordinates in the Extent's CRS.

- The id field in VectorTile Features doesn't matter.

- UNKNOWN geometries are safe to ignore.

- If a VectorTile geometry list marked as POINT has only one pair of coordinates, it will be decoded as a Geotrellis Point. If it has more than one pair, it will be decoded as a MultiPoint. Likewise for the LINESTRING and POLYGON types. A complaint has been made about the spec regarding this, and future versions may include a difference between single and multi geometries.

3.8.12 GeoTiffs

GeoTiffs are a type of Tiff image file that contain image data pertaining to satellite, aerial, and elevation data among other types of geospatial information. The additional pieces of metadata that are needed to store and display this information is what sets GeoTiffs apart from normal Tiffs. For instance, the positions of geographic features on the screen and how they are projected are two such pieces of data that can be found within a GeoTiff, but is absent from a normal Tiff file.

GeoTiff File Format

Because GeoTiffs are Tiffs with extended features, they both have the same file structure. There exist three components that can be found in all Tiff files: the header, the image file directory, and the actual image data. Within these files, the directories and image data can be found at any point within the file; regardless of how the images are presented when the file is opened and viewed. The header is the only section which has a constant location, and that is at the begining of the file.

File Header

As stated earlier, the header is found at the beginning of every Tiff file, including GeoTiffs. All Tiff files have the exact same header size of 8 bytes. The first two bytes of the header are used to determine the ByteOrder of the file, also known as "Endianness". After these two, comes the next two bytes which are used to determine the file's magic

number. `.tif`, `.txt`, `.shp`, and all other file types have a unique identifier number that tells the program kind of file it was given. For Tiff files, the magic number is 42. Due to how the other components can be situated anywhere within the file, the last 4 bytes of the header provide the offset value that points to the first file directory. Without this offset, it would be impossible to read a Tiff file.

Image File Directory

For every image found in a Tiff file there exists a corresponding image file directory for that picture. Each property listed in the directory is referred to as a `Tag`. `Tags` contain information on, but not limited to, the image size, compression types, and the type of color plan. Since we're working with Geotiffs, geo-spatial information is also documented within the `Tags`. These directories can vary in size, as users can create their own tags and each image in the file does not need to have exact same tags.

Other than image attributes, the file directory holds two offset values that play a role in reading the file. One points to where the actual image itself is located, and the other shows where the the next file directory can be found.

Image Data

A Tiff file can store any number of images within a single file, including none at all. In the case of GeoTiffs, the images themselves are almost always stored as bitmap data. It is important to understand that there are two ways in which the actual image data is formatted within the file. The two methods are: Striped and Tiled.

Striped

Striped storage breaks the image into segments of long, vertical bands that stretch the entire width of the picture. Contained within them are columns of bitmapped image data. If your GeoTiff file was created before the realse of Tiff 6.0, then this is the data storage method in which it most likely uses.

If an image has strip storage, then its corresponding file directory contains the tags: `RowsPerStrip`, `StripOffsets`, and `StripByteCount`. All three of these are needed to read that given segment. The first one is the number of rows that are contained within the strips. Every strip within an image must have the same number of rows within it except for the last one in certain instances. `StripOffsets` is an array of offsets that shows where each strip starts within the file. The last tag, `ByteSegmentCount`, is also an array of values that contains the size of each strip in terms of Bytes.

Tiled

Tiff 6.0 introduced a new way to arrange and store data within a Tiff, tiled storage. These rectangular segments have both a height and a width that must be divisible by 16. There are instances where the tiled grid does not fit the image exactly. When this occurs, padding is added around the image so as to meet the requirement of each tile having dimensions of a factor of 16.

As with stips, tiles have specific tags that are needed in order to process each segment. These new tags are: `TileWidth`, `TileLength`, `TileOffsets`, and `TileByteCounts`. `TileWidth` is the number of columns and `TileLength` is the number of rows that are found within the specified tile. As with striped, `TileOffsets` and `TileByteCounts` are arrays that contain the begining offset and the byte count of each tile in the image, respectively.

Layout: Columns and Rows

At a high level, there exist two ways to refer to a location within GeoTiffs. One is to use Map coordinates which are X and Y values. X's are oriented along the horizontal axis and run from west to east while Y's are on the vertical axis

and run from south to north. Thus the further east you are, the larger your X value; and the more north you are the larger your Y value.

The other method is to use the grid coordinate system. This technique of measurement uses Cols and Rows to describe the relative location of things. Cols run east to west whereas Rows run north to south. This then means that Cols increase as you go east to west, and rows increase as you go north to south.

Each (X, Y) pair corresponds to some real location on the planet. Cols and rows, on the other hand, are ways of specifying location *within the image* rather than by reference to any actual location. For more on coordinate systems supported by GeoTiffs, check out the relevant parts of the spec.

Big Tiffs

In order to qualify as a BigTiff, your file needs to be **at least 4gb in size or larger**. At this size, the methods used to store and find data are different. The accommodation that is made is to change the size of the various offsets and byte counts of each segment. For a normal Tiff, this size is 32-bits, but BigTiffs have these sizes at 64-bit. GeoTrellis transparently supports BigTiffs, so so you shouldn't need to worry about size.

Cloud Optimized GeoTiffs

Just as the GeoTiff is a subset of Tiff meant to convey information not only about image values but also the spatial extent of that imagery, Cloud Optimized GeoTiffs (COGs for short) are a nascent subset of GeoTiff meant to increase their expressiveness, ease of use, and portability through further standardization. We call these GeoTiffs "cloud optimized" because the features they add allow for remote access to GeoTiff that, with the help of HTTP GET range requests, access the parts of a tiff you're interested without consuming large portions of the image which are irrelevant to your computation.

COGs are thus capable of serving as a self-describing backing for raster layers. The only cost associated with the use of COGs over GeoTrellis' Avro-based layers is the extra effort related to metadata retrieval and munging (metadata for each individual GeoTiff will need to be collected as opposed to the monolithic metadata of Avro layers, which is read and handled once for the entire layer).

The COG specification (which is not a 100% complete as of the writing of this documentation) defines required tags and means of access (a server accepting GET range requests). These required features are necessary to even support remotely reading subsets of the overall image data from some remote Tiff.

COG requirements: - Tiled storage of image data - Overviews at different levels of resolution - Infrastructure capable of handling GET Range requests

```scala
// Constructing a COG from a non-COG tiff
val nonCog = SinglebandGeoTiff(path = file:///path/to/my/tiff.tif)
val almostCog = nonCog.withStorageMethod(Tiled)

// most likely either NearestNeighbor or BilinearInterpolation; depends on context
val resampleMethod: ResampleMethod = ???
val fullCog = almostCog.withOverviews(resampleMethod)
```

> A note on sidecar files > The spec seems to indicate that overviews be part of the GeoTiff itself to > count as a COG. In practice, things are messier than that. Content providers > aren't always going to want to rewrite their tiffs to stuff generated > overviews into them. The practical upshot of this is that separate overview > files should be supported (GDAL will actually inspect some canonical relative > paths within the directory of the Tiff being read).

```scala
// Constructing a COG with sidecar overviews
val mainTile = SinglebandGeoTiff(path = file:///path/to/my/file.tif)
val overview1 = SinglebandGeoTiff(path = file:///path/to/my/file.tif.ovr1)
```

(continues on next page)

```scala
val overview2 = SinglebandGeoTiff(path = file:///path/to/my/file.tif.ovr2)
val tileWithOverview = mainTile.withOverviews(List(overview1, overview2))
```

Structured vs Unstructured COGs

Historically, Geotrellis layers have been backed by specially encoded Avro layers which are were designed to maximize the performance of distributed reading and querying. With the advent of the COG and the development of tooling to support this subset of the GeoTiff spec, however, the advantages of depending upon a bespoke raster format are less obvious than they once were. Avro support is likely to continue, but support for applications backed by COGs are a priority for continued GeoTrellis development.

To this end, GeoTrellis is introducing the notion of a 'structured' COG layer. Structured COG layers are actually a collection of COGs tiled out in a consistent manner and described through common (GeoTrellis-specific) metadata which is designed to enhance query performance for larger layers by allowing GeoTrellis programs to infer information about underlying, individual COG files without having to read multiple of them.

Structured COG metadata: - cellType: Underlying Tiff celltype (width of cell representation and NoData strategy) - zoomRangeInfos: A map from some range of supported zoom levels to a collection of key extents - layoutScheme: The scheme by which individual COG tiles are cut for this layer - extent: The overall extent of all underlying COGs - crs: The projection of all underlying COGs

```scala
// We'll need to get a layer from somewhere
val layer: RDD[(K, V)] with Metadata[TileLayerMetadata[K]] = ???
// The native resolution for this layer (assumes standard TMS zoom levels)
val baseZoom = 8

// With that, we should be able to construct a 'structured' COG layer
val structured: CogLayer[K, V] = CogLayer.fromLayerRDD(layer, baseZoom)
```

Further Readings

- For more information on the Tiff file format
- For more information on the GeoTiff file format
- For more information on the Cloud Optimized GeoTiff file format

3.8.13 Typeclasses

Typeclasses are a common feature of Functional Programming. As stated in the FAQ, typeclasses group data types by what they can *do*, as opposed to by what they *are*. If traditional OO inheritance arranges classes in a tree hierarchy, typeclasses arrange them in a graph.

Typeclasses are realized in Scala through a combination of `traits` and `implicit` class wrappings. A typeclass constraint is visible in a class/method signature like this:

```scala
class Foo[A: Order](a: A) { ... }
```

Meaning that `Foo` can accept any `A`, so long as it is "orderable". In reality, this in syntactic sugar for the following:

```scala
class Foo[A](a: A)(implicit ev: Order[A]) { ... }
```

Here's a real-world example from GeoTrellis code:

```
protected def _write[
  K: AvroRecordCodec: JsonFormat: ClassTag,
  V: AvroRecordCodec: ClassTag,
  M: JsonFormat: GetComponent[?, Bounds[K]]
](layerId: LayerId, rdd: RDD[(K, V)] with Metadata[M], keyIndex: KeyIndex[K]): Unit =
→{ ... }
```

A few things to notice:

- Multiple constraints can be given to a single type variable: `K: Foo: Bar: Baz`
- `?` refers to `M`, helping the compiler with type inference. Unfortunately `M: GetComponent[M, Bounds[K]]` is not syntactically possible

Below is a description of the most-used typeclasses used in GeoTrellis. All are written by us, unless otherwise stated.

ClassTag

Built-in from `scala.reflect`. This allows classes to maintain some type information at runtime, which in GeoTrellis is important for serialization. You will never need to use this directly, but may have to annotate your methods with it (the compiler will let you know).

JsonFormat

From the `spray` library. This constraint says that its type can be converted to and from JSON, like this:

```
def toJsonAndBack[A: JsonFormat](a: A): A = {
  val json: Value = a.toJson

  json.convertTo[A]
}
```

AvroRecordCodec

Any type that can be serialized by Apache Avro. While references to `AvroRecordCodec` appear frequently through GeoTrellis code, you will never need to use its methods. They are used internally by our Tile Layer Backends and Spark.

Boundable

Always used on `K`, `Boundable` means your key type has a finite bound.

```
trait Boundable[K] extends Serializable {
  def minBound(p1: K, p2: K): K

  def maxBound(p1: K, p2: K): K
...  // etc
}
```

Component

`Component` is a bare-bones `Lens`. A `Lens` is a pair of functions that allow one to generically get and set values in a data structure. They are particularly useful for nested data structures. `Component` looks like this:

```scala
trait Component[T, C] extends GetComponent[T, C] with SetComponent[T, C]
```

Which reads as "if I have a `T`, I can read a `C` out of it" and "if I have a `T`, I can write some `C` back into it". The lenses we provide are as follows:

- `SpatialComponent[T]` - read a `SpatialKey` out of a some `T` (usually `SpatialKey` or `SpaceTimeKey`)
- `TemporalComponent[T]` - read a `TemporalKey` of some `T` (usually `SpaceTimeKey`)

Cats

There is a wide variety of standard typeclasses employed by the functional programming community. Rather than implement them ourselves, we have elected to depend on the Cats project to provide this extra functionality. We intend to provide as much compatibility with Cats as is reasonable without sacrificing readability for users who are not functional programming mavens. Initially, we rely on *Functor's, 'Semigroup's, and 'Monoid's, but there is some use of the 'IO* monad in limited parts of the code base. Please see the documentation for Cats for more information.

3.8.14 More Core Concepts

CRS

Data Structures: `CRS`, `LatLng`, `WebMercator`, `ConusAlbers`

In GIS, a *projection* is a mathematical transformation of Latitude/Longitude coordinates on a sphere onto some other flat plane. Such a plane is naturally useful for representing a map of the earth in 2D. A projection is defined by a *Coordinate Reference System* (CRS), which holds some extra information useful for reprojection. CRSs themselves have static definitions, have agreed-upon string representations, and are usually made public by standards bodies or companies. They can be looked up at SpatialReference.org.

A *reprojection* is the transformation of coorindates in one CRS to another. To do so, coordinates are first converted to those of a sphere. Every CRS knows how to convert between its coordinates and a sphere's, so a transformation `CRS.A -> CRS.B -> CRS.A` is actually `CRS.A -> Sphere -> CRS.B -> Sphere -> CRS.A`. Naturally some floating point error does accumulate during this process.

Within the context of GeoTrellis, the main projection-related object is the `CRS` trait. It stores related `CRS` objects from underlying libraries, and also provides the means for defining custom reprojection methods, should the need arise.

Here is an example of using a `CRS` to reproject a `Line`:

```scala
val wm = Line(...)  // A `LineString` vector object in WebMercator.
val ll: Line = wm.reproject(WebMercator, LatLng)  // The Line reprojected into LatLng.
```

Extents

Data structures: `Extent`, `ProjectedExtent`, `TemporalProjectedExtent`, `GridExtent`, `RasterExtent`

An `Extent` is a rectangular section of a 2D projection of the Earth. It is represented by two coordinate pairs that are its "min" and "max" corners in some Coorindate Reference System. "min" and "max" here are CRS specific, as the location of the point `(0,0)` varies between different CRS. An Extent can also be referred to as a *Bounding Box*.

Within the context of GeoTrellis, the points within an `Extent` always implicitely belong to some `CRS`, while a `ProjectedExtent` holds both the original `Extent` and its current `CRS`.

Here are some useful `Extent` operations, among many more:

- `Extent.translate: (Double, Double) => Extent`
- `Extent.distance: Extent => Double`
- `Extent.contains: Extent => Boolean`
- `Extent.intersection: Extent => Option[Extent]`
- `ProjectedExtent.reproject: CRS => Extent`

`Extent`s are most often used to represent the area of an entire Tile layer, and also the individual `Tile`s themselves (especially in the case of `Raster`s).

3.9 Using Rasters

This document serves as a complete guide to using `Raster`s in GeoTrellis.

3.9.1 Rendering

Rendering Common Image Formats

At some point, you'll want to output a visual representation of the tiles you're processing. Likely, that's why you're reading this bit of documentation. Luckily enough, geotrellis provides some methods which make the process as painless as possible. Currently, both PNG and JPG formats are supported.

To begin writing your tiles out as PNGs and JPGs, there are just a few things to keep in mind. As elsewhere throughout geotrellis, the functionality in this module is added through implicit class extension. `import geotrellis.raster._` will import the necessary methods off of the core types like `renderToPng`, and the the types like `ColorRamp` and `ColorMap` live in `geotrellis.raster.render`.

First Steps

Let's say that the tile you've got is an integer tile and that the integers in it are all *actually* hex codes for RGBA colors. In this case, your task is nearly complete. The following code should be sufficient:

```
import geotrellis.raster._

// Generate the tile - let's paint it red with #FF0000FF
// (red = 0xFF or 255; green = 0x00 or 0; blue = 0x00 or 0; and alpha = 0xFF or 255,
→which is completely opaque)
val hexColored: IntArrayTile = IntArrayTile.fill(0xFF0000FF, 100, 100)

// Making a PNG
val png: Png = hexColored.renderPng

// JPG variation
val jpg: Jpg = hexColorsHere.renderJpg
```

A `Png` and `Jpg` type represent the `Array[Byte]` that is the binary encoding of the image. You can get to the bytes by calling the `bytes` property, e.g. `png.bytes` or `jpg.bytes`. It's useful to use the bytes directly if you are, say, returning PNG data from a web service. These image format types also have a `write` method that can be called to write that array of bytes to the file system, therefore writing out a `PNG` or `JPG` representation of our `Tile` to the filesystem.

Clearly this won't suffice for the majority of use-cases. In general, you're more likely to be working on tiles whose cells encode information having only an incidental relation to human vision. In these cases, you'll need to tell `renderPng` and `renderJpg` how the values in your tile relate to the colors you'd like in your image. To this end, there are arguments you can provide to the render method which will tell geotrellis how to color cells for your tile.

ColorRamp

A ColorRamp describes a range of colors that can be used to paint values of a tile. You can use a built-in color ramp, or construct one with your own palette with the API described below.

Built-in color ramps

GeoTrellis provides built-in color ramps in the `ColorRamps` object. These are provided to ease the transition from developer to cartographer. However, you need not feel constrained by these and can use your own color palettes as well. There are many good resources online for selecting color ramps.

Color Schemes

#2586AB	#118C8C	#2791C3	#569543
#4EA3C8	#429D91	#5DA1CA	#9EBD4D
#7FB8D4	#61AF96	#83B2D1	#BBCA7A
#ADD8EA	#75C59B	#A8C5D8	#D9E2B2
#C8E1E7	#A2CF9F	#CCDBE0	#E4E7C4
#EDECEA	#C5DAA3	#E9D3C1	#E6D6BE
#F0E7BB	#E6E5A7	#DCAD92	#E3C193
#F5CF7D	#E3D28F	#D08B6C	#DFAC6C
#F9B737	#E0C078	#C66E4B	#DB9842
#E68F2D	#DDAD62	#BD4E2E	#B96230
#D76B27	#D29953		
	#CA8746		
	#C2773B		

From left to right

Blue to Orange

An 11-step diverging color ramp from blue to gray to orange. The gray critical class in the middle clearly shows a median or zero value. Example uses include temperature, climate, elevation, or other color ranges where it is necessary to distinguish categories with multiple hues.

Blue to Red

A 10-step diverging color ramp from blue to red. Example uses include elections and politics, voter swing, climate or temperature, or other color ranges where it is necessary to distinguish categories with multiple hues.

Green to Red-Orange

A 10-step diverging color ramp from green to red-orange. Example uses include elections and politics, voter swing, climate or temperature, or other color ranges where it is necessary to distinguish categories with multiple hues.

Green to Orange

A 13-step diverging color ramp from green to orange. Example uses include elevation, relief maps, topography, or other color ranges where it is necessary to distinguish categories with multiple hues.

Sequential Color Schemes

	#FFFFFF
	#FBEDD1
	#F7E0A9
	#EFD299
	#E8C58B
	#E0B97E
	#F2924D
	#C97877
	#946196
	#2AB7D6
	#474040

	#E8EDDB
	#DCE8D4
	#BEDBAD
	#A0CF88
	#81C561
	#4BAF48
	#1CA049
	#3A6D35

	#F7DA22
	#ECBE1D
	#E77124
	#D54927
	#CF3A27
	#A33936
	#7F182A
	#68101A

	#2A2E7F
	#3D5AA9
	#4698D3
	#39C6F0
	#76C9B3
	#A8D050
	#F6EB14
	#FCB017
	#F16022
	#EE2C24
	#7D1416

	#68101A
	#7F182A
	#A33936
	#CF3A27
	#D54927
	#E77124
	#ECBE1D
	#F7DA22
	#F6EDB1
	#FFFFFF

	#A52278
	#993086
	#8C3C97
	#6D328A
	#4E2B81
	#3B264B
	#180B11
	#FFFFFF

From left to right

Light to Dark - Sunset

An 11-step sequential color ramp showing intensity from light to dark. This color ramp is perfect for showing density where it is critical to highlight very different values with bold colors at the higher, darker end of the ramp. Example uses include population density, accessibility, or ranking.

Light to Dark - Green

A basic 8-step sequential color ramp showing light to dark in shades of green. Example uses include density, ordered data, ranking, or any map where darker colors represent higher data values and lighter colors represent lower data values, generally.

Yellow to Red - Heatmap

An 8-step sequential heatmap from yellow to dark red. Great for heatmaps on a light basemap where the hottest values are more opaque or dark. Also useful for sequential color ranges where the lowest value is the median or zero value.

Blue to Yellow to Red Spectrum - Heatmap

An 11-step heatmap from blue to yellow to red. Great for showing a wide range of values with clear differences in hue.

Dark Red to Yellow-White - Heatmap

A 10-step sequential heatmap from dark red to yellow to white. Great for heatmaps where the hottest values should look more vibrant or intense.

Light Purple to Dark Purple to White

An 8-step sequential heatmap to show intensity with shades of purple with white as the "hottest" value. Great for light or gray basemaps, or where the highest value needs to be called out visually.

Qualitative or Categorical Schemes

#B29CC3	#CEE1E8
#4F8EBB	#7CBCB5
#8F9238	#82B36D
#C18437	#94C279
#B5D6B1	#D1DE8D
#D378A6	#EDECC3
#D4563C	#CCAFB4
#F9BE47	#C99884

From left to right

Bold Land Use

An 8-hue qualitative scheme used to show a clear difference in categories that are unordered or very different. Example uses include zoning, land use, land cover, or maps where all categories or groups are equal in visual strength/magnitude.

Muted Terrain

An 8-hue qualitative scheme used to show different kinds of map topology or features. This is generally used to show landforms, terrain, and topology.

Viridis, Magma, Plasma and Inferno

The Viridis, Magma, Plasma and Inferno color ramps were developed for matplotlib, and are incorporated into our default color ramp set. You can read more about *these color ramps here <https://bids.github.io/colormap/>*.

Custom Color Ramps

You can create your own color ramp with a list of integar values, constructed using our RBG or RGBA helper objects.

```
val colorRamp =
  ColorRamp(
    RGB(0,255,0),
    RGB(63, 255 ,51),
    RGB(102,255,102),
    RGB(178, 255,102),
    RGB(255,255,0),
    RGB(255,255,51),
    RGB(255,153, 51),
    RGB(255,128,0),
    RGB(255,51,51),
    RGB(255,0,0)
  )
```

You can also do things like set the number of stops in a gradient between colors, and set an alpha gradient. This example sets a 100 color stops that interpolates colors between red and blue, with an alpha value that starts at totally opaque for the red values, and ends at 0xAA alpha for blue values:

```
val colorRamp =
    ColorRamp(0xFF0000FF, 0x0000FFFF)
      .stops(100)
      .setAlphaGradient(0xFF, 0xAA)
```

There are many online and offline resources for generating color palettes for cartography including:

- Carto Colors
- ColorBrewer 2.0
- Cartographer's Toolkit: Colors, Typography, Patterns, by Gretchen N. Peterson
- Designing Better Maps, by Cynthia A. Brewer
- Designed Maps: A Sourcebook, by Cynthia A. Brewer

RGBA vs RGB values

One way to represent a color is as an RGB hex value, as often seen in CSS or graphics programs. For example, the color red is represented by #FF0000 (or, in scala, 0xFF0000).

Internally to GeoTrellis, colors are represented as RGBA values, which includes a value for transparency. These can be represented with 8 instead of 6 hex characters (with the alpha opacity value being the last two charcters) such as 0xFF0000FF for opaque red. When using the programming interface, just be sure to keep the distinction in mind.

You can create RGB and RGBA colors in a variety of ways:

```scala
import geotrellis.raster.render._

val color1: Int = RGB(r = 255, g = 170, b = 85)
val color2: Int = RGBA(0xFF, 0xAA, 0x55, 0xFF)
val color3: Int = 0xFFAA55FF
assert(color1 == color2 && color2 == color3)
```

ColorMap

A `ColorMap` is what actually determines how the values of a tile to colors. It constitutes a mapping between class break values and color stops, as well as some options that determine how to color raster values.

ColorMap Options

The options available for a ColorMap are a class boundary type, which determines how those class break values (one of `GreaterThan`, `GreaterThanOrEqualTo`, `LessThan`, `LessThanOrEqualTo`, or `Exact`), an option that defines what color NoData values should be colored, as well as an option for a "fallback" color, which determines the color of any value that doesn't fit to the color map. Also, if the `strict` option is true, then no fallback color is used, and the code will throw an exception if a value does not fit the color map. The default values of these options are:

```scala
val colorMapDefaultOptions =
  ColorMap.Options(
    classBoundaryType = LessThan,
    noDataColor = 0x00000000, // transparent
    fallbackColor = 0x00000000, // transparent
    strict = false
  )
```

To examplify the options, let's look at how two different color ramps will color values.

```scala
import geotrellis.render._

// Constructs a ColorMap with default options,
// and a set of mapped values to color stops.
val colorMap1 =
  ColorMap(
    Map(
      3.5 -> RGB(0,255,0),
      7.5 -> RGB(63,255,51),
      11.5 -> RGB(102,255,102),
      15.5 -> RGB(178,255,102),
      19.5 -> RGB(255,255,0),
      23.5 -> RGB(255,255,51),
      26.5 -> RGB(255,153,51),
```

(continues on next page)

```scala
        31.5 -> RGB(255,128,0),
        35.0 -> RGB(255,51,51),
        40.0 -> RGB(255,0,0)
      )
    )

// The same color map, but this time considering the class boundary type
// as GreaterThanOrEqualTo, and with a fallback and nodata color.
val colorMap2 =
  ColorMap(
    Map(
        3.5 -> RGB(0,255,0),
        7.5 -> RGB(63,255,51),
        11.5 -> RGB(102,255,102),
        15.5 -> RGB(178,255,102),
        19.5 -> RGB(255,255,0),
        23.5 -> RGB(255,255,51),
        26.5 -> RGB(255,153,51),
        31.5 -> RGB(255,128,0),
        35.0 -> RGB(255,51,51),
        40.0 -> RGB(255,0,0)
      ),
    ColorMap.Options(
      classBoundaryType = GreaterThanOrEqualTo,
      noDataColor = 0xFFFFFFFF,
      fallbackColor = 0xFFFFFFFF
    )
)
```

If we were to use the `mapDouble` method of the color maps to find color values of the following points, we'd see the following:

```scala
scala> colorMap1.mapDouble(2.0) == RGB(0, 255, 0)
res1: Boolean = true

scala> colorMap2.mapDouble(2.0) == 0xFFFFFFFF
res2: Boolean = true
```

Because `colorMap1` has the `LessThan` class boundary type, `2.0` will map to the color value of `3.5`. However, because `colorMap2` is based on the `GreaterThanOrEqualTo` class boundary type, and `2.0` is not greater than or equal to any of the mapped values, it maps `2.0` to the `fallbackColor`.

```scala
scala> colorMap1.mapDouble(23.5) == RGB(255,153,51)
res4: Boolean = true

scala> colorMap2.mapDouble(23.5) == RGB(255,255,51)
res5: Boolean = true
```

If we map a value that is on a class border, we can see that the `LessThan` color map maps the to the lowest class break value that our value is still less than (`26.5`), and for the `GreaterThanOrEqualTo` color map, since our value is equal to a class break value, we return the color associated with that value.

Creating a ColorMap based on Histograms

One useful way to create ColorMaps is based on a `Histogram` of a tile. Using a histogram, we can compute the quantile breaks that match up to the color stops of a `ColorRamp`, and therefore paint a tile based on quantiles instead of something like equal intervals. You can use the `fromQuantileBreaks` method to create a `ColorMap` from both a `Histogram[Int]` and `Histogram[Double]`

Here is an example of creating a `ColorMap` from a `ColorRamp` and a `Histogram[Int]`, in which we define a ramp from red to blue, set the number of stops to 10, and convert it to a color map based on quantile breaks:

```scala
val tile: Tile = ???

val colorMap = ColorMap.fromQuantileBreaks(tile.histogram, ColorRamp(0xFF0000FF,
→0x0000FFFF).stops(10))
```

Here is another way to do the same thing, using the `ColorRamp.toColorMap`:

```scala
val tile: Tile = ???

val colorMap: ColorMap =
  ColorRamp(0xFF0000FF, 0x0000FFFF)
    .stops(10)
    .toColorMap(tile.histogram)
```

PNG and JPG Settings

It might be useful to tweak the rendering of images for some use cases. In light of this fact, both png and jpg expose a `Settings` classes (`geotrellis.raster.render.jpg.Settings` and `geotrellis.raster.render.png.Settings`) which provide a means to tune image encoding.

In general, messing with this just isn't necessary. If you're unsure, there's a good chance this featureset isn't for you.

PNG Settings

`png.Settings` allows you to specify a `ColorType` (bit depth and masks) and a `Filter`. These can both be read about on the W3 specification and [png Wikipedia page](#).

JPEG Settings

`jpg.Settings` allow specification of the compressionQuality (a Double from 0 to 1.0) and whether or not Huffman tables are to be computed on each run - often referred to as 'optimized' rendering. By default, a compressionQuality of 0.7 is used and Huffman table optimization is not used.

3.9.2 Reprojecting

Overview

Many core GeoTrellis data types can be reprojected. To reproject a `Line`:

```
val wm: Line = ...

val ll: Line = wm.reproject(WebMercator, LatLng)  /* The Line reprojected into LatLng␣
↪*/
```

To reproject a `Tile`:

```
val wm: Tile = ...
val extent: Extent = ....  /* Area covered by the Tile */
val raster: Raster[Tile] = Raster(wm, extent)  /* A Raster is a "location-aware" Tile␣
↪*/

val ll: Raster[Tile] = raster.reproject(WebMercator, LatLng)
```

To reproject an `Extent`:

```
val wm: Extent = ...

val ll: Extent = wm.reproject(WebMercator, LatLng)
```

See the pattern?

A GeoTrellis "layer" (some `RDD[(K, V)]`) conceptually represents a single, giant piece of imagery. In this form however it finds itself cut up into individual `V` (usually `Tile`) and indexed by `K` (usually `SpatialKey`). Is such a giant, cut-up raster still reprojectable? Certainly:

```
// Recall this common alias:
//   type TileLayerRDD[K] = RDD[(K, Tile)] with Metadata[TileLayerMetadata[K]]
val wm: TileLayerRDD[SpatialKey] = ...  /* Result of previous work */
val layout: LayoutDefinition = ...      /* Size/shape of your new grid */

val (zoom, ll): (Int, TileLayerRDD[SpatialKey]) = wm.reproject(LatLng, layout)
```

Let's break down the last line some more:

```
val (zoom, ll): (Int, TileLayerRDD[SpatialKey]) = wm.reproject(LatLng, layout)
     [1]                                                       [2]      [3]
```

- [1]: When a `ZoomedLayoutScheme` was given instead of a `LayoutDefinition`, this `zoom` value is the nearest zoom level that the layer was reprojected to. Otherwise, this value is `0`.

- [2]: We only need to provide the target `CRS` here, since a `TileLayerRDD` implicitely knows its own projection.

- [3]: Providing a `LayoutDefinition` allows us to define the shape of the grid that results from the reproject. As an overload, this method can accept a `LayoutScheme` instead.

You may also have a different formulation if your data source is a giant GeoTiff on S3, and not a pre-ingested GeoTrellis layer. In that case, you'd have:

```
/* Extract Tiles efficiently from a remote GeoTiff */
val wm: RDD[(ProjectedExtent, Tile)] = S3GeoTiffRDD.spatial("s3-bucket-name", "your-
↪image.tiff")

val ll: RDD[(ProjectedExtent, Tile)] = wm.reproject(LatLng)
```

`.reproject` here doesn't need a source `CRS`, since this information is encoded in each `ProjectedExtent`.

On Layer Reprojection

It's important to note that the resulting RDDs in this last example and the one previous to it are different. The latter is not keyed, and so doesn't necessarily represent some original, unified raster. It is not a "layer" in the usual GeoTrellis sense, and the projection is done per-Tile. With `TileLayerRDD`, the Tiles were automatically given buffer pixels during reprojection in order to avoid artifacts (i.e. seams, dead pixels, etc.).

Futhermore, we can expect the reprojected Tiles in the `RDD[(ProjectedExtent, Tile)]` to be skewed:
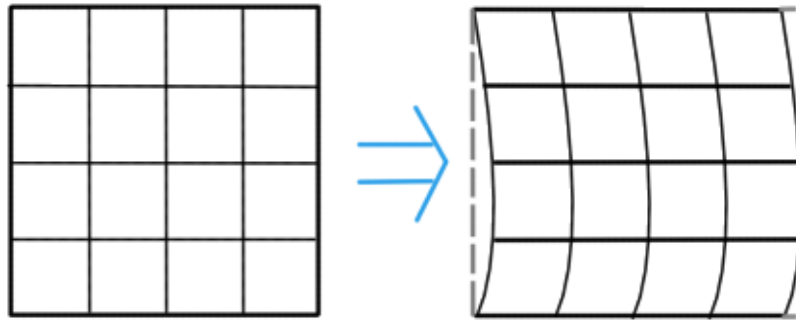
where the border regions of each `Tile` are almost certainly patches of `NODATA`. This is undesirable but also fixable, thanks to `.tileToLayout`:

```scala
val wm: RDD[(ProjectedExtent, Tile)] = S3GeoTiffRDD.spatial("s3-bucket-name", "your-
↪image.tiff")

val ll: RDD[(ProjectedExtent, Tile)] = wm.reproject(LatLng)

/* The value type held within the tiles. Example: `IntCellType` */
val ct: CellType = ???

/* The size and shape of your desired grid. */
val layout: LayoutDefinition = ???

/* Reorganize your reprojected Tiles into a true grid. */
val layer: RDD[(SpatialKey, Tile)] = ll.tileToLayout(ct, layout)
```

`.tileToLayout` works by recutting each source `Tile` into potentially many target tiles, associating each with a `SpatialKey`, and merging any that have matching keys. This merging is what clears out our `NODATA`.

And now we've returned to a nice keyed Layer! Due to fewer Spark shuffles, this method technically has better performance than when we had a `TileLayerRDD` to begin with. However, you will not be freed from potential seams or other image artifacts.

3.9.3 Resampling

Often, when working with raster data, it is useful to change the resolution, crop the data, transform the data to a different projection, or to do all of that at once. This all relies on our ability to resample, which is the act of changing the spatial resolution and layout of the raster cells, and interpolating the values of the modified cells from the original cells. For everything there is a price, however, and changing the resolution of a tile is no exception: there will (almost) always be a loss of information (or representativeness) when conducting an operation which changes the number of cells on a tile.

Upsampling vs Downsampling

Intuitively, there are two ways that you might resample a tile. You might:

1. increase the number of cells

2. decrease the number of cells

Increasing the number of cells produces more information at the cost of being only probabilistically representative of the underlying data whose points are being used to generate new values. We can call this upsampling (because we're increasing the samples for a given representation of this or that state of affairs). Typically, upsampling is handled through interpolation of new values based on the old ones.

The opposite, downsampling, involves a loss of information. Fewer points of data are tasked with representing the same states of affair as the tile on which the downsampling is carried out. Downsampling is a common strategy in digital compression.

Aggregate vs Point-Based Resampling

In geotrellis, `ResampleMethod` is an ADT (through a sealed trait in `Resample.scala`) which branches into `PointResampleMethod` and `AggregateResampleMethod`. The aggregate methods of resampling are all suited for downsampling only. For every extra cell created by upsampling with an `AggregateResampleMethod`, the resulting tile is **absolutely certain** to contain a `NODATA` cell. This is because for each additional cell produced in an aggregated resampling of a tile, a bounding box is generated which determines the output cell's value on the basis of an aggregate of the data captured within said bounding box. The more cells produced through resampling, the smaller an aggregate bounding box. The more cells produced through resampling, the less likely it is that this box will capture any values to aggregate over.

What we call 'point' resampling doesn't necessarily require a box within which data is aggregated. Rather, a point is specified for which a value is calculated on the basis of nearby value(s). Those nearby values may or may not be weighted by their distance from the point in question. These methods are suitable for both upsampling and downsampling.

Remember What Your Data Represents

Along with the formal characteristics of these methods, it is important to keep in mind the specific character of the data that you're working with. After all, it doesn't make sense to use a method like `Bilinear` resampling if you're dealing primarily with categorical data. In this instance, your best bet is to choose an aggregate method (and keep in mind that the values generated don't necessarily mean the same thing as the data being operated on) or a forgiving (though unsophisticated) method like `NearestNeighbor`.

3.9.4 Histograms

It's often useful to derive a histogram from rasters, which represents a distribution of the values of the raster in a more compact form. In GeoTrellis, we differentiate between a `Histogram[Int]`, which represents the exact counts of integer values, and a `Histogram[Double]`, which represents a grouping of values into a discrete number of buckets. These types are in the `geotrellis.raster.histogram` package.

The default implementation of `Histogram[Int]` is the `FastMapHistogram`, developed by Erik Osheim, which utilizes a growable array structure for keeping track of the values and counts.

The default implementation of `Histogram[Double]` is the `StreamingHistogram`, developed by James McClain and based on the paper `Ben-Haim, Yael, and Elad Tom-Tov. "A streaming parallel decision tree algorithm." The Journal of Machine Learning Research 11 (2010): 849-872.`.

Histograms can give statistics such as min, max, median, mode and median. It also can derive quantile breaks, as dsecribed in the next section.

Quantile Breaks

Dividing a histogram distribution into quantile breaks attempts to classify values into some number of buckets, where the number of values classified into each bucket are generally equal. This can be useful in representing the distribution of the values of a raster.

For instance, say we had a tile with mostly values between 1 and 100, but there were a few values that were 255. We want to color the raster with 3 values: low values with red, middle values with green, and high values with blue. In other words, we want to classify each of the raster values into one of three categories: red, green and blue. One technique, called equal interval classification, consists of splitting up the range of values (1 - 255) into the number of equal intervals as target classifications (3). This would give us a range intervals of 1 - 85 for red, 86 - 170 for green, and 171 - 255 for blue. This corresponds to "breaks" values of 85, 170, and 255. Because the values are mostly between 1 and 100, most of our raster would be colored red. This may not show the contrast of the dataset that we would like.

Another technique for doing this is called quantile break classification; this makes use of the quantile breaks we can derive from our histogram. The quantile breaks will concentrate on making the number of values per "bin" equal, instead of the range of the interval. With this method, we might end up seeing breaks more like 15, 75, 255, depending on the distribution of the values.

For a code example, this is how we would do exactly what we talked about: color a raster tile into red, green and blue values based on it's quantile breaks:

```scala
import geotrellis.raster.histogram._
import geotrellis.raster.render._

val tile: Tile = ??? // Some raster tile
val histogram: Histogram[Int] = tile.histogram

val colorRamp: ColorRamp =
  ColorRamp(
    RGB(r=0xFF, b=0x00, g=0x00),
    RGB(r=0x00, b=0xFF, g=0x00),
    RGB(r=0x00, b=0x00, g=0xFF)
  )

val colorMap = ColorMap.fromQuantileBreaks(histogram, colorRamp)

val coloredTile: Tile = tile.color(colorMap)
```

3.9.5 Kriging Interpolation

These docs are about **Raster** Kriging interpolation.

The process of Kriging interpolation for point interpolation is explained in the geotrellis.vector.interpolation package documnetation.

Kriging Methods

The Kriging methods are largely classified into different types in the way the mean(mu) and the covariance values of the object are dealt with.

```
// Array of sample points with given data
val points: Array[PointFeature[Double]] = ...
/** Supported is also present for
  * val points: Traversable[PointFeature[D]] = ... //where D <% Double
  */

// The raster extent to be kriged
val extent = Extent(xMin, yMin, xMax, yMax)
val cols: Int = ...
val rows: Int = ...
val rasterExtent = RasterExtent(extent, cols, rows)
```

There exist four major kinds of Kriging interpolation techniques, namely:

Simple Kriging

```
// Simple kriging, a tile  set with the Kriging prediction per cell is returned
val sv: Semivariogram = NonLinearSemivariogram(points, 30000, 0, Spherical)

val krigingVal: Tile = points.simpleKriging(rasterExtent, 5000, sv)

/**
  * The user can also do Simple Kriging using :
  * points.simpleKriging(rasterExtent)
  * points.simpleKriging(rasterExtent, bandwidth)
  * points.simpleKriging(rasterExtent, Semivariogram)
  * points.simpleKriging(rasterExtent, bandwidth, Semivariogram)
  */
```

It belong to the class of Simple Spatial Prediction Models.

The simple kriging is based on the assumption that the underlying stochastic process is entirely *known* and the spatial trend is constant, viz. the mean and covariance values of the entire interpolation set is constant (using solely the sample points)

```
mu(s) = mu               known; s belongs to R
cov[eps(s), eps(s')]     known; s, s' belongs to R
```

Ordinary Kriging

```
// Ordinary kriging, a tile  set with the Kriging prediction per cell is returned
val sv: Semivariogram = NonLinearSemivariogram(points, 30000, 0, Spherical)

val krigingVal: Tile = points.ordinaryKriging(rasterExtent, 5000, sv)

/**
  * The user can also do Ordinary Kriging using :
  * points.ordinaryKriging(rasterExtent)
  * points.ordinaryKriging(rasterExtent, bandwidth)
  * points.ordinaryKriging(rasterExtent, Semivariogram)
  * points.ordinaryKriging(rasterExtent, bandwidth, Semivariogram)
  */
```

It belong to the class of Simple Spatial Prediction Models.

This method differs from the Simple Kriging appraoch in that, the constant mean is assumed to be unknown and is estimated within the model.

```
mu(s) = mu              unknown; s belongs to R
cov[eps(s), eps(s')]    known; s, s' belongs to R
```

Universal Kriging

```scala
// Universal kriging, a tile  set with the Kriging prediction per cell is returned
val attrFunc: (Double, Double) => Array[Double] = {
  (x, y) => Array(x, y, x * x, x * y, y * y)
}

val krigingVal: Tile = points.universalKriging(rasterExtent, attrFunc, 50, Spherical)

/**
  * The user can also do Universal Kriging using :
  * points.universalKriging(rasterExtent)
  * points.universalKriging(rasterExtent, bandwidth)
  * points.universalKriging(rasterExtent, model)
  * points.universalKriging(rasterExtent, bandwidth, model)
  * points.universalKriging(rasterExtent, attrFunc)
  * points.universalKriging(rasterExtent, attrFunc, bandwidth)
  * points.universalKriging(rasterExtent, attrFunc, model)
  * points.universalKriging(rasterExtent, attrFunc, bandwidth, model)
  */
```

It belongs to the class of General Spatial Prediction Models.

This model allows for explicit variation in the trend function (mean function) constructed as a linear function of spatial attributes; with the covariance values assumed to be known. This model computes the prediction using

For example if:

```
x(s) = [1, s1, s2, s1 * s1, s2 * s2, s1 * s2]'
mu(s) = beta0 + beta1*s1 + beta2*s2 + beta3*s1*s1 + beta4*s2*s2 + beta5*s1*s2
```

Here, the "linear" refers to the linearity in parameters (beta).

```
mu(s) = x(s)' * beta,   beta unknown; s belongs to R
cov[eps(s), eps(s')]    known; s, s' belongs to R
```

Geostatistical Kriging

```scala
// Geostatistical kriging, a tile  set with the Kriging prediction per cell is
↪returned
val attrFunc: (Double, Double) => Array[Double] = {
  (x, y) => Array(x, y, x * x, x * y, y * y)
}

val krigingVal: Tile = points.geoKriging(rasterExtent, attrFunc, 50, Spherical)

/**
  * The user can also do Universal Kriging using :
```

(continues on next page)

```
* points.geoKriging(rasterExtent)
* points.geoKriging(rasterExtent, bandwidth)
* points.geoKriging(rasterExtent, model)
* points.geoKriging(rasterExtent, bandwidth, model)
* points.geoKriging(rasterExtent, attrFunc)
* points.geoKriging(rasterExtent, attrFunc, bandwidth)
* points.geoKriging(rasterExtent, attrFunc, model)
* points.geoKriging(rasterExtent, attrFunc, bandwidth, model)
*/
```

It belong to the class of General Spatial Prediction Models.

This model relaxes the assumption that the covariance is known. Thus, the beta values and covariances are simultaneously evaluated and is computationally more intensive.

```
mu(s) = x(s)' * beta,    beta unknown; s belongs to R
cov[eps(s), eps(s')]     unknown; s, s' belongs to R
```

Attribute Functions (Universal, Geostatistical Kriging):

The `attrFunc` function is the attribute function, which is used for evaluating non-constant spatial trend structures. Unlike the Simple and Ordinary Kriging models which rely only on the residual values for evaluating the spatial structures, the General Spatial Models may be modelled by the user based on the data (viz. evaluating the beta variable to be used for interpolation).

In case the user does not specify an attribute function, by default the function used is a quadratic trend function for `Point(s1, s2)`:

```
mu(s) = beta0 + beta1*s1 + beta2*s2 + beta3*s1*s1 + beta4*s2*s2 + beta5*s1*s2
```

General example of a trend function is :

```
mu(s) = beta0 + Sigma[ beta_j * (s1^n_j) * (s2^m_j) ]
```

Example to understand the attribute Functions

Consider a problem statement of interpolating the ground water levels of Venice. It is easy to arrive at the conclusion that it depends on three major factors; namely, the elevation from the ground, the industries' water intake, the island's water consumption. First of all, we would like to map the coordinate system into another coordinate system such that generation of the relevant attributes becomes easier (please note that the user may use any method for generating the set of attribute functions; in this case we have used coordinate transformation before the actual calculation).

```
val c1: Double = 0.01 * (0.873 * (x - 418) - 0.488 * (y - 458))
val c2: Double = 0.01 * (0.488 * (x - 418) + 0.873 * (y - 458))
```

Image taken from `Smith, T.E., (2014) Notebook on Spatial Data Analysis [online]` `http://www.seas.upenn.edu/~ese502/#notebook`
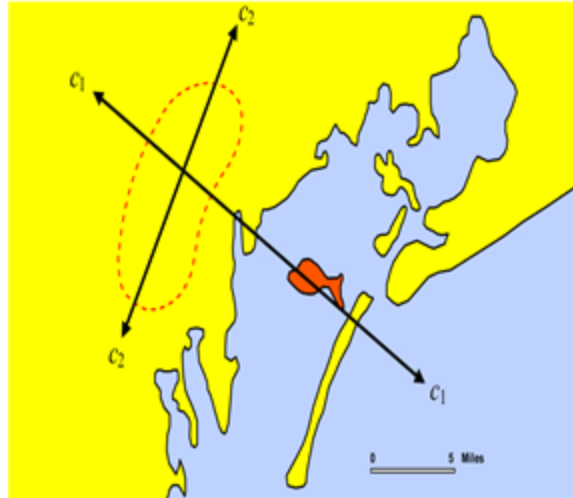
Elevation

Fig. 1: Coordinate Mapping

```
/** Estimate of the elevation's contribution to groundwater level
 * [10 * exp(-c1)]
 */
val elevation: Double = math.exp(-1 * c1)
```

Industry draw down (water usage of industry)
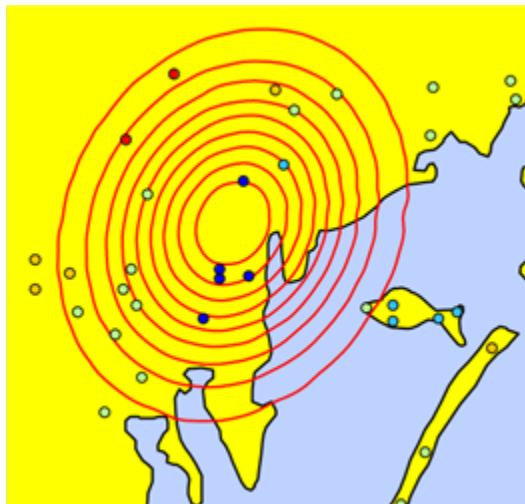
Fig. 2: Industry Draw Down

Image taken from `Smith, T.E., (2014) Notebook on Spatial Data Analysis [online]`
`http://www.seas.upenn.edu/~ese502/#notebook`

```
/** Estimate of the industries' contribution to groundwater level
 * exp{ -1.0 * [(1.5)*c1^2 - c2^2]}
 */
val industryDrawDown: Double = math.exp(-1.5 * c1 * c1 - c2 * c2)
```

Island draw down (water usage of Venice)

Fig. 3: Venice Draw Down

Image taken from `Smith, T.E., (2014) Notebook on Spatial Data Analysis [online]` `http://www.seas.upenn.edu/~ese502/#notebook`

```
/** Estimate of the island's contribution to groundwater level
  * //exp{-1.0 * (sqrt((s1-560)^2 + (s2-390)^2) / 35)^8 }
  */
val islandDrawDown: Double =
    math.exp(-1 * math.pow(math.sqrt(math.pow(x - 560, 2) + math.pow(y - 390, 2)) /␣
↪35, 8))
```

The Final Attribute Function

Thus for a `Point(s1, s2)`:

`Array(elevation, industryDrawDown, islandDrawDown)` is the set of attributes.

In case the intuition for a relevant `attrFunc` is not clear; the user need not supply an `attrFunc`, by default the following attribute Function is used :

```
// For a Point(x, y), the set of default attributes is :
Array(x, y, x * x, x * y, y * y)
```

The default function would use the data values of the given sample points and construct a spatial structure trying to mirror the actual attribute characteristics.

3.10 Using Vectors

3.10.1 Numerical Precision and Topology Exceptions

When doing any significant amount of geometric computation, there is the potential of encountering JTS TopologyExceptions or other numerical hiccups that arise, essentially, from asking too much of floating point numerical

representations. JTS offers the ability to snap coordinates to a grid, the scale of whose cells may be specified by the user. This can often fix problems stemming from numerical error which makes two points that should be the same appear different due to miniscule rounding errors.

However, GeoTrellis defaults to allowing the full floating point representation of coordinates. To enable the fixed precision scheme, one must create an `application.conf` in `your-project/src/main/resources/` containing the following lines:

```
geotrellis.jts.precision.type="fixed"
geotrellis.jts.precision.scale=<scale factor>
```

The scale factor should be `10^x` where `x` is the number of decimal places of precision you wish to keep. The default scale factor is 1e12, indicating that 12 decimal places after the decimal point should be kept. Values less than 1 (negative exponents) allow for very coarse grids. For example, a scale factor of 1e-2 will round coordinate components to the nearest hundred.

Note that `geotrellis.jts.precision.type` make take on the value `floating` for the default double precision case, or `floating_single` to use single precision floating point values.

3.10.2 Parsing GeoJson

GeoTrellis includes good support for serializing and deserializing geometry to/from GeoJson within the `geotrellis.vector.io.json` package. Utilizing these features requires some instruction, however, since the interface may not be immediately apparent from the type signatures.

Serializing to GeoJson

All `Geometry` and `Feature` objects in `geotrellis.vector` have a method extension providing a `toGeoJson` method. This means that:

```scala
import geotrellis.vector.io._

Polygon((10.0, 10.0), (10.0, 20.0), (30.0, 30.0), (10.0, 10.0)).toGeoJson
```

is valid, in this case yielding:

```json
{"type":"Polygon","coordinates":[[[10.0,10.0],[10.0,20.0],[30.0,30.0],[10.0,10.0]]]}
```

Issuing `.toGeoJson` on `Feature` instances, requires that the type parameter supplied to the feature meets certain requirements. For example, `PointFeature(Point(0,0), 3)` will succeed, but to tag a Feature with arbitrary data, that data must be encapsulated in a case class. That class must also be registered with the Json reading infrastructure provided by `spray`. The following example achieves these goals:

```scala
import geotrellis.vector.io.json._

case class UserData(data: Int)
implicit val boxedValue = jsonFormat1(UserData)

PointFeature(Point(0,0), UserData(13))
```

Case classes with more than one argument would require the variants of `jsonFormat1` for classes of higher arity. The output of the above snippet is:

```json
{"type":"Feature","geometry":{"type":"Point","coordinates":[0.0,0.0]},"properties":{
↪"data":13}}
```

where the property has a single field named `data`. Upon deserialization, it will be necessary for the data member of the feature to have fields with names that are compatible with the members of the feature's data type.

This is all necessary underpinning, but note that it is generally desirable to (de)serialize collections of features. The serialization can be achieved by calling `.toGeoJson` on a `Seq[Feature[G, T]]`. The result is a Json representation of a FeatureCollection.

Deserializing from GeoJson

The return trip from a string representation can be accomplished by another method extension provided for strings: `parseGeoJson[T]`. The only requirement for using this method is that the type of `T` must match the contents of the Json string. If the Json string represents some `Geometry` subclass (i.e., `Point`, `MultiPolygon`, etc), then that type should be supplied to `parseGeoJson`. This will work to make the return trip from any of the Json strings produced above.

Again, it is generally more interesting to consider Json strings that contain `FeatureCollection` structures. These require more complex code. Consider the following Json string:

```scala
val fc: String = """{
                  |  "type": "FeatureCollection",
                  |  "features": [
                  |    {
                  |      "type": "Feature",
                  |      "geometry": { "type": "Point", "coordinates": [1.0, 2.0] },
                  |      "properties": { "someProp": 14 },
                  |      "id": "target_12a53e"
                  |    }, {
                  |      "type": "Feature",
                  |      "geometry": { "type": "Point", "coordinates": [2.0, 7.0] },
                  |      "properties": { "someProp": 5 },
                  |      "id": "target_32a63e"
                  |    }
                  |  ]
                  |}""".stripMargin
```

Decoding this structure will require the use of either `JsonFeatureCollection` or `JsonFeatureCollectionMap`; the former will return queries as a `Seq[Feature[G, T]]`, while the latter will return a `Map[String, Feature[G, T]]` where the key is the `id` field of each feature. After calling:

```scala
val collection = fc.parseGeoJson[JsonFeatureCollectionMap]
```

it will be necessary to extract the desired features from `collection`. In order to maintain type safety, these results are pulled using accessors such as `.getAllPoints`, `.getAllMultiLineFeatures`, and so on. Each geometry and feature type requires the use of a different method call.

As in the case of serialization, to extract the feature data from this example string, we must create a case class with an integer member named `someProp` and register it using `jsonFormat1`.

```scala
case class SomeProp(someProp: Int)
implicit val boxedToRead = jsonFormat1(SomeProp)

collection.getAllPointFeatures[SomeProp]
```

A Note on Creating JsonFeatureCollectionMaps

It is straightforward to create FeatureCollection representations, as illustrated above. Simply package your features into a `Seq` and call `toGeoJson`. In order to name those features, however, it requires that a JsonFeatureCollectionMap be explicitly created. For instance:

```
val fcMap = JsonFeatureCollectionMap(Seq("bob" -> Feature(Point(0,0), UserData(13))))
```

Unfortunately, the `toGeoJson` method is not extended to `JsonFeatureCollectionMap`, so we are forced to call `fcMap.toJson.toString` to get the same functionality. The return of that call is:

```
{
  "type": "FeatureCollection",
  "features": [{
    "type": "Feature",
    "geometry": {
      "type": "Point",
      "coordinates": [0.0, 0.0]
    },
    "properties": {
      "data": 13
    },
    "id": "bob"
  }]
}
```

3.10.3 Working with Vectors in Spark

While GeoTrellis is focused on working with raster data in spark, we do have some functionality for working with vecto data in spark.

ClipToGrid

If you have an `RDD[Geometry]` or `RDD[Feature[Geometry, D]]`, you may want to cut up the geometries according to `SpatialKey` s, so that you can join that data to other raster or vector sources in an efficient way. To do this, you can use the `rdd.clipToGrid` methods.

For example, if you want to read GeoTiffs on S3, and find the sum of raster values under each of the polygons, you could use the following technique:

```
import geotrellis.raster._
import geotrellis.spark._
import geotrellis.spark.tiling._
import geotrellis.vector._

import org.apache.spark.HashPartitioner
import org.apache.spark.rdd.RDD

import java.net.URI
import java.util.UUID

// The extends of the GeoTiffs, along with the URIs
val geoTiffUris: RDD[Feature[Polygon, URI]] = ???
val polygons: RDD[Feature[Polygon, UUID]] = ???
```

```scala
// Choosing the appropriately resolute layout for the data is here considered a␣
↪client concern.
val layout: LayoutDefinition = ???

// Abbreviation for the code to read the window of the GeoTiff off of S3
def read(uri: URI, window: Extent): Raster[Tile] = ???

val groupedPolys: RDD[(SpatialKey, Iterable[MultiPolygonFeature[UUID]])] =
  polygons
    .clipToGrid(layout)
    .flatMap { case (key, feature) =>
      val mpFeature: Option[MultiPolygonFeature[UUID]] =
        feature.geom match {
          case p: Polygon => Some(feature.mapGeom(_ => MultiPolygon(p)))
          case mp: MultiPolygon => Some(feature.mapGeom(_ => mp))
          case _ => None
        }
      mpFeature.map { mp => (key, mp) }
    }
    .groupByKey(new HashPartitioner(1000))

val rastersToKeys: RDD[(SpatialKey, URI)] =
  geoTiffUris
    .clipToGrid(layout)
    .flatMap { case (key, feature) =>
      // Filter out any non-polygonal intersections.
      // Also, we will do the window read from the SpatialKey extent, so throw out␣
↪polygon.
      feature.geom match {
        case p: Polygon => Some((key, feature.data))
        case mp: MultiPolygon => Some((key, feature.data))
        case _ => None
      }
    }

val joined: RDD[(SpatialKey, (Iterable[MultiPolygonFeature[UUID]], URI))] =
  groupedPolys
    .join(rastersToKeys)

val totals: Map[UUID, Long] =
  joined
    .flatMap { case (key, (features, uri)) =>
      val raster = read(uri, layout.mapTransform.keyToExtent(key))

      features.map { case Feature(mp, uuid) =>
        (uuid, raster.tile.polygonalSum(raster.extent, mp).toLong)
      }
    }
    .reduceByKey(_ + _)
    .collect
    .toMap
```
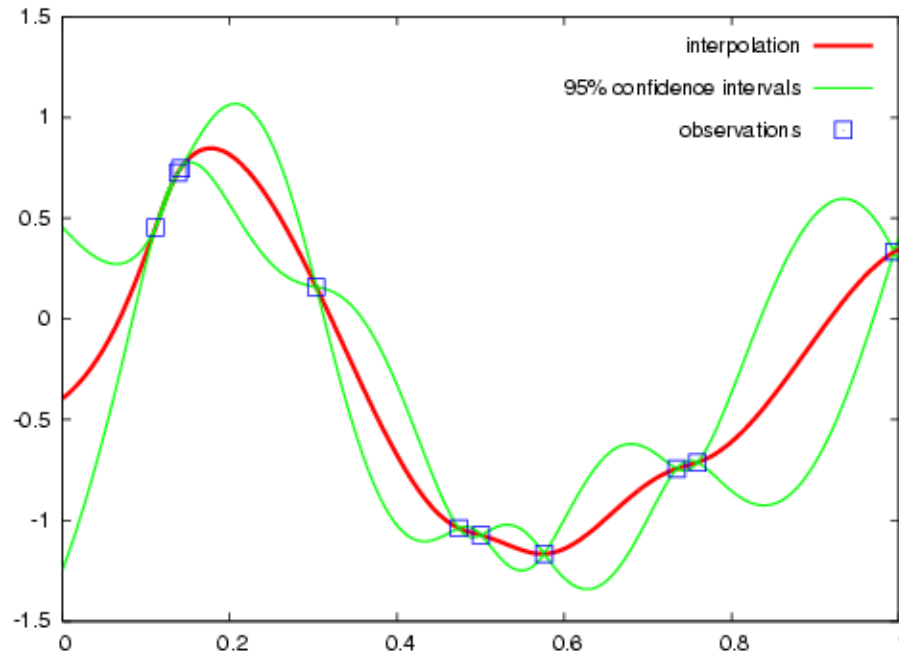
3.10.4 Kriging Interpolation

Semivariograms

This method of interpolation is based on constructing Semivariograms. For grasping the structure of spatial dependencies of the known data-points, semivariograms are constructed.

First, the sample data-points' spatial structure to be captured is converted to an empirical semivariogram, which is then fit to explicit/theoretical semivariogram models.

Two types of Semivariograms are developed :

- Linear Semivariogram

- Non-Linear Semivariograms

Empirical Semivariogram

```scala
//(The array of sample points)
val points: Array[PointFeature[Double]] = ???

/** The empirical semivariogram generation
  * "maxDistanceBandwidth" denotes the maximum inter-point distance relationship
  * that one wants to capture in the empirical semivariogram.
  */
val es: EmpiricalVariogram = EmpiricalVariogram.nonlinear(points,␣
→maxDistanceBandwidth, binMaxCount)
```

The sample-data point used for training the Kriging Models are clustered into groups(aka bins) and the data-values associated with each of the data-points are aggregated into the bin's value. There are various ways of constructing the bins, i.e. equal bin-size(same number of points in each of the bins); or equal lag-size(the bins are separated from each other by a certain fixed separation, and the samples with the inter-points separation fall into the corresponding bins).

In case, there are outlier points in the sample data, the equal bin-size approach assures that the points' influence is tamed down; however in the second approach, the outliers would have to be associated with weights (which is computationally more intensive).

The final structure of the empirical variogram has an array of tuples :

```
(h, k)
where h => Inter-point distance separation
      k => The variogram's data-value (used for covariogram construction)
```

Once the empirical semivariograms have been evaluated, these are fitted into the theoretical semivariogram models (the fitting is carried out into those models which best resemble the empirical semivariogram's curve generate).

Linear Semivariogram

```
/** "radius" denotes the maximum inter-point distance to be
  * captured into the semivariogram
  * "lag" denotes the inter-bin distance
  */
val points: Array[PointFeature[Double]] = ...
val linearSV = LinearSemivariogram(points, radius, lag)
```

This is the simplest of all types of explicit semivariogram models and does not very accurately capture the spatial structure, since the data is rarely linearly changing. This consists of the points' being modelled using simple regression into a straight line. The linear semivariogram has linear dependency on the free variable (inter-point distance) and is represented by:

```
f(x) = slope * x + intercept
```

Non-Linear Semivariogram

```
/**
  * ModelType can be any of the models from
  * "Gaussian", "Circular", "Spherical", "Exponential" and "Wave"
  */
val points: Array[PointFeature[Double]] = ...
val nonLinearSV: Semivariogram =
    NonLinearSemivariogram(points, 30000, 0, [[ModelType]])
```

Most often the empirical variograms can not be adequately represented by the use of linear variograms. The non-linear variograms are then used to model the empirical semivariograms for use in Kriging intepolations. These have non-linear dependencies on the free variable (inter-point distance).

In case the empirical semivariogram has been previously constructed, it can be fitted into the semivariogram models by :

```
val svSpherical: Semivariogram =
    Semivariogram.fit(empiricalSemivariogram, Spherical)
```

The popular types of Non-Linear Semivariograms are :

```
(h in each of the function definition denotes the inter-point distances)
```

Gaussian Semivariogram

```scala
// For explicit/theoretical Gaussian Semivariogram
val gaussianSV: Semivariogram =
    NonLinearSemivariogram(range, sill, nugget, Gaussian)
```

The formulation of the Gaussian model is :

```
                    | 0                                      , h = 0
gamma(h; r, s, a) = |
                    | a + (s - a) {1 - e^(-h^2 / r^2)}  , h > 0
```

Circular Semivariogram

```scala
//For explicit/theoretical Circular Semivariogram
val circularSV: Semivariogram =
    NonLinearSemivariogram(range, sill, nugget, Circular)
```

```
                         | 0                                                                       ⌴
↪            , h = 0
                         |
                         |                       |                                              ___
↪_____  |
                         |                       |       2                 | h |      2h          /  ⌴
↪ h^2   |
gamme(h; r, s, a) = | a + (s - a) * |1 - ----- * cos_inverse|---| + -------- *   /1 -⌴
↪-----   | , 0 < h <= r
                         |                       |      pi                 | r |    pi * r    \/   ⌴
↪ r^2   |
                         |                       |                                               ⌴
↪        |
                         |
                         | s                                                                      ⌴
↪          , h > r
```

Spherical Semivariogram

```scala
// For explicit/theoretical Spherical Semivariogram
val sphericalSV: Semivariogram = NonLinearSemivariogram(range, sill, nugget,⌴
↪Spherical)
```

```
                    | 0                                , h = 0
                    |                  | 3h      h^3  |
gamma(h; r, s, a) = | a + (s - a) |---- - ------- | , 0 < h <= r
                    |                  | 2r     2r^3  |
                    | s                                , h > r
```

Exponential Semivariogram

```
// For explicit/theoretical Exponential Semivariogram
val exponentialSV: Semivariogram = NonLinearSemivariogram(range, sill, nugget,␣
↪Exponential)
```

```
                      | 0                                    , h = 0
gamma(h; r, s, a) = |
                      | a + (s - a) {1 - e^(-3 * h / r)}    , h > 0
```

Wave Semivariogram

```
//For explicit/theoretical Exponential Semivariogram
//For wave, range (viz. r) = wave (viz. w)
val waveSV: Semivariogram =
    NonLinearSemivariogram(range, sill, nugget, Wave)
```

```
                      | 0                                    , h = 0
                      |
gamma(h; w, s, a) = |                    |        sin(h / w)  |
                      | a + (s - a) |1 - w ------------ | , h > 0
                      |                    |           h       |
```

Notes on Semivariogram fitting

The empirical semivariogram tuples generated are fitted into the semivariogram models using Levenberg Marquardt Optimization. This internally uses jacobian (differential) functions corresponding to each of the individual models for finding the optimum range, sill and nugget values of the fitted semivariogram.

```
// For the Spherical model
val model: ModelType = Spherical
valueFunc(r: Double, s: Double, a: Double): (Double) => Double =
    NonLinearSemivariogram.explicitModel(r, s, a, model)
```

The Levenberg Optimizer uses this to reach to the global minima much faster as compared to unguided optimization.

In case, the initial fitting of the empirical semivariogram generates a negative nugget value, then the process is re-run after forcing the nugget value to go to zero (since mathematically, a negative nugget value is absurd).

Kriging Methods

Once the semivariograms have been constructed using the known point's values, the kriging methods can be invoked.

The methods are largely classified into different types in the way the mean(mu) and the covariance values of the object are dealt with.

```
//Array of sample points with given data
val points: Array[PointFeature[Double]] = ...

//Array of points to be kriged
val location: Array[Point] = ...
```

There exist four major kinds of Kriging interpolation techniques, namely :

Simple Kriging

```scala
//Simple kriging, tuples of (prediction, variance) per prediction point
val sv: Semivariogram = NonLinearSemivariogram(points, 30000, 0, Spherical)

val krigingVal: Array[(Double, Double)] =
    new SimpleKriging(points, 5000, sv)
      .predict(location)
/**
  * The user can also do Simple Kriging using :
  * new SimpleKriging(points).predict(location)
  * new SimpleKriging(points, bandwidth).predict(location)
  * new SimpleKriging(points, sv).predict(location)
  * new SimpleKriging(points, bandwidth, sv).predict(location)
  */
```

It belongs to the class of Simple Spatial Prediction Models.

The simple kriging is based on the assumption that the underlying stochastic process is entirely *known* and the spatial trend is constant, viz. the mean and covariance values of the entire interpolation set is constant (using solely the sample points)

```
mu(s) = mu             known; s belongs to R
cov[eps(s), eps(s')]   known; s, s' belongs to R
```

Ordinary Kriging

```scala
//Ordinary kriging, tuples of (prediction, variance) per prediction point
val sv: Semivariogram = NonLinearSemivariogram(points, 30000, 0, Spherical)

val krigingVal: Array[(Double, Double)] =
    new OrdinaryKriging(points, 5000, sv)
      .predict(location)
/**
  * The user can also do Ordinary Kriging using :
  * new OrdinaryKriging(points).predict(location)
  * new OrdinaryKriging(points, bandwidth).predict(location)
  * new OrdinaryKriging(points, sv).predict(location)
  * new OrdinaryKriging(points, bandwidth, sv).predict(location)
  */
```

It belongs to the class of Simple Spatial Prediction Models.

This method differs from the Simple Kriging appraoch in that, the constant mean is assumed to be unknown and is estimated within the model.

```
mu(s) = mu             unknown; s belongs to R
cov[eps(s), eps(s')]   known; s, s' belongs to R
```

Universal Kriging

```scala
//Universal kriging, tuples of (prediction, variance) per prediction point
```

```scala
val attrFunc: (Double, Double) => Array[Double] = {
  (x, y) => Array(x, y, x * x, x * y, y * y)
}

val krigingVal: Array[(Double, Double)] =
    new UniversalKriging(points, attrFunc, 50, Spherical)
      .predict(location)
/**
  * The user can also do Universal Kriging using :
  * new UniversalKriging(points).predict(location)
  * new UniversalKriging(points, bandwidth).predict(location)
  * new UniversalKriging(points, model).predict(location)
  * new UniversalKriging(points, bandwidth, model).predict(location)
  * new UniversalKriging(points, attrFunc).predict(location)
  * new UniversalKriging(points, attrFunc, bandwidth).predict(location)
  * new UniversalKriging(points, attrFunc, model).predict(location)
  * new UniversalKriging(points, attrFunc, bandwidth, model).predict(location)
  */
```

It belongs to the class of General Spatial Prediction Models.

This model allows for explicit variation in the trend function (mean function) constructed as a linear function of spatial attributes; with the covariance values assumed to be known.

For example if :

```
x(s) = [1, s1, s2, s1 * s1, s2 * s2, s1 * s2]'
mu(s) = beta0 + beta1*s1 + beta2*s2 + beta3*s1*s1 + beta4*s2*s2 + beta5*s1*s2
```

Here, the "linear" refers to the linearity in parameters (beta).

```
mu(s) = x(s)' * beta,    beta unknown; s belongs to R
cov[eps(s), eps(s')]     known; s, s' belongs to R
```

The `attrFunc` function is the attribute function, which is used for evaluating non-constant spatial trend structures. Unlike the Simple and Ordinary Kriging models which rely only on the residual values for evaluating the spatial structures, the General Spatial Models may be modelled by the user based on the data (viz. evaluating the beta variable to be used for interpolation).

In case the user does not specify an attribute function, by default the function used is a quadratic trend function for Point(s1, s2) :

```
mu(s) = beta0 + beta1*s1 + beta2*s2 + beta3*s1*s1 + beta4*s2*s2 + beta5*s1*s2
```

General example of a trend function is :

```
mu(s) = beta0 + Sigma[ beta_j * (s1^n_j) * (s2^m_j) ]
```

An elaborate example for understanding the `attrFunc` is mentioned in the readme file in `geotrellis.raster.interpolation` along with detailed illustrations.

Geostatistical Kriging

```scala
//Geostatistical kriging, tuples of (prediction, variance) per prediction point
val attrFunc: (Double, Double) => Array[Double] = {
  (x, y) => Array(x, y, x * x, x * y, y * y)
}
```

```scala
val krigingVal: Array[(Double, Double)] =
    new GeoKriging(points, attrFunc, 50, Spherical)
      .predict(location)
/**
  * Geostatistical Kriging can also be done using:
  * new GeoKriging(points).predict(location)
  * new GeoKriging(points, bandwidth).predict(location)
  * new GeoKriging(points, model).predict(location)
  * new GeoKriging(points, bandwidth, model).predict(location)
  * new GeoKriging(points, attrFunc).predict(location)
  * new GeoKriging(points, attrFunc, bandwidth).predict(location)
  * new GeoKriging(points, attrFunc, model).predict(location)
  * new GeoKriging(points, attrFunc, bandwidth, model).predict(location)
  */
```

It belongs to the class of General Spatial Prediction Models.

This model relaxes the assumption that the covariance is known. Thus, the beta values and covariances are simultaneously evaluated and is computationally more intensive.

```
mu(s) = x(s)' * beta,    beta unknown; s belongs to R
cov[eps(s), eps(s')]     unknown; s, s' belongs to R
```

3.10.5 Delaunay Triangulations, Voronoi Diagrams, and Euclidean Distance

When working with vector data, it is often necessary to establish sensible interconnections among a collection of discrete points in ^{2} (the Euclidean plane). This operation supports nearest neighbor operations, linear interpolation among irregularly sampled data, and Euclidean distance, to name only a few applications.

For this reason, GeoTrellis provides a means to compute the Delaunay triangulation of a set of points. Letting be the input set of points, the Delaunay triangulation is a partitioning of the convex hull of into triangular regions (a partition that completely covers the convex hull with no overlaps). Each triangle, `T`, has a unique circle passing through all of its vertices that we call the *circumscribing circle* of `T`. The defining property of a Delaunay triangulation is that each `T` has a circumscribing circle that contains no points of in their interiors (note that the vertices of `T` are on the boundary of the circumscribing circle, not in the interior). Among the most important properties of a Delaunay triangulation is its relationship to the Voronoi diagram. The Voronoi diagram is another partitioning of ^{2} based on the points in . This time, the partitioning is composed of convex polygonal regions—one for each point in —that completely cover the plane (some of the convex regions are half open, which is to say that they may extend to infinity in certain directions). The Delaunay triangulation of is the *dual* to the Voronoi diagram of . This means that elements of the Delaunay triangulation have a one-to-one correspondence with the elements of the Voronoi diagram. Letting `DT()` be the Delaunay triangulation of and `V()` be the Voronoi diagram of , we have that each vertex of `DT()` corresponds to a polygonal region of `V()` (called the *Voronoi cell* of), each edge to an edge, and each triangle to a vertex. The number of edges emanating from a vertex in `DT()` gives the number of sides of the corresponding polygonal region in `V()`. The corresponding edges of each structure are perpendicular. The Voronoi vertex corresponding to a triangle of `DT()` is the center of that triangle's circumscribing circle. And if there are no more than 3 points of lying on any circle in the plane (a condition called *general position*), then there are no more than 3 edges emanating from any vertex of `V()`, which matches the number of sides in each planar region of `DT()`. (If we are not in general position, not all vertices of `V()` will be distinct and some Voronoi edges may have zero length.) The dual relationship between `DT()` and `V()` is important because it means that we may compute whichever structure that is easiest and simply derive the other in a straightforward manner. As it happens, it is generally easier to compute Delaunay triangulations, and we have implemented a very fast method for doing just that. Specifically, we employ the divide-and-conquer approach to computing the Delaunay triangulation based on Guibas and Stolfi's 1985 ACM Transactions on Graphics paper.

Mesh Representation

Delaunay triangulations are represented using half edges, a common data structure for encoding polygonal meshes. Half edges are so called because, when attempting to represent an edge from vertex A to vertex B, we require two complementary half edges: one pointing to A and one pointing to B. Half edges are connected into *loops*, one for each face in the mesh; so given a half edge, the loop may be iterated over. Surprisingly, these three pieces of information are enough to create a mesh that can be easily navigated, though the class of meshes that may be represented are limited to orientable (having an inside and an outside—i.e., no Möbius strips), manifold surfaces (for any point on the surface, the intersection of a small 3-d ball around the point and the surface is a disc—i.e., no more than two faces share an edge, faces sharing a vertex must be contiguous). We also take on the convention that when viewed from the "outside" of the surface, the edges of a loop traverse the facet vertices in counterclockwise order. But note that if a mesh has a boundary, as is the case with Delaunay triangulations, there is a boundary loop that navigates the vertices of the boundary in clockwise order.
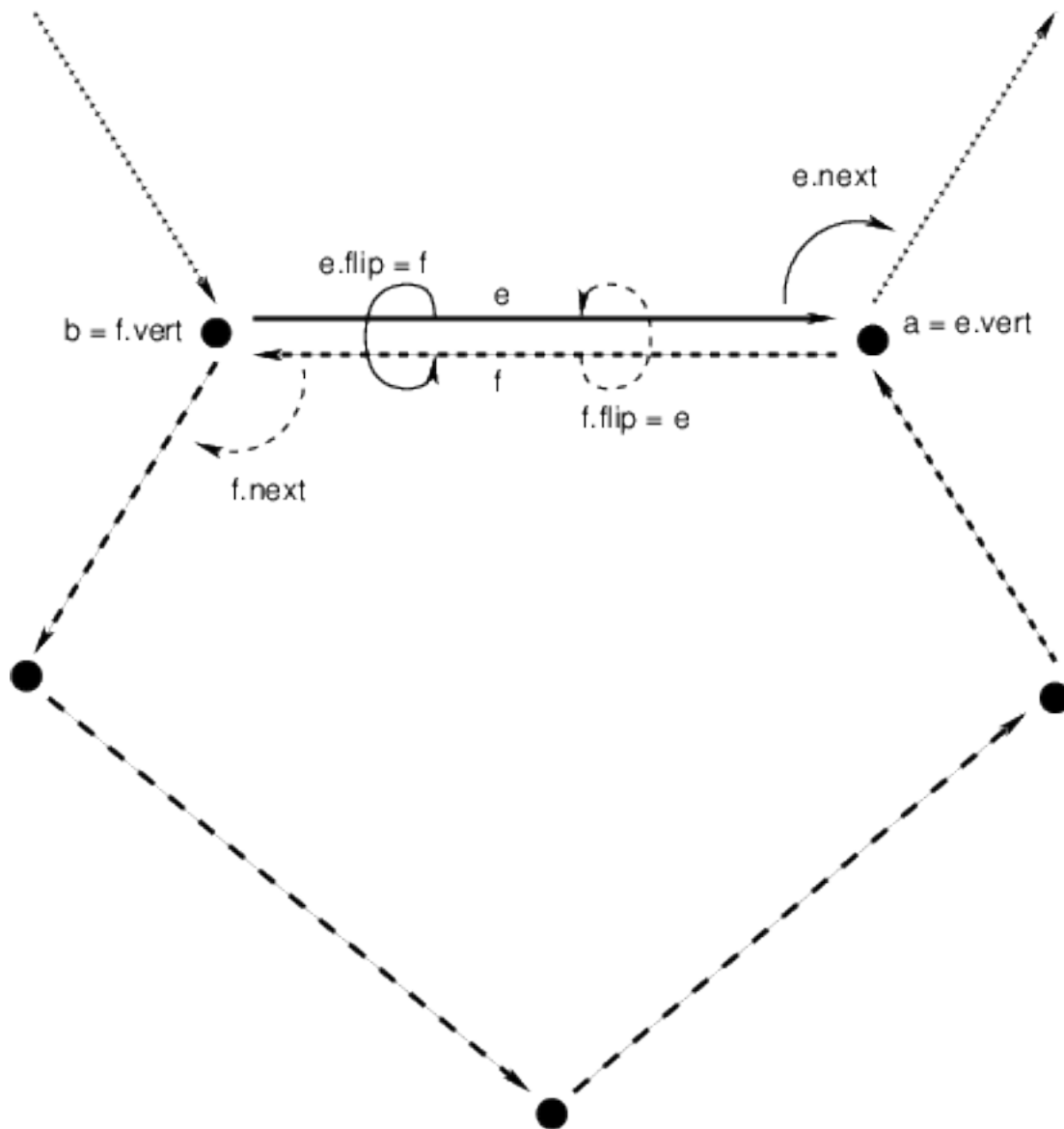
There are two means to represent a half edge mesh in GeoTrellis: the object-based HalfEdge structure, and the faster, more space efficient, but less generic HalfEdgeTable structure. The latter constitutes the core of our mesh structures, but the former has some uses for small-scale applications for intrepid users.

Delaunay Triangulations

The intent for our DelaunayTriangulation implementation is that we be able to easily handle triangulations over 10s or 100s of millions of points (though the latter scale especially may require distribution via Spark to do so in a reasonable time/memory envelope). Smaller scale applications can easily compute Delaunay triangulations of arrays of JTS Coordinates (GeoTrellis Points are too heavyweight given the scale of our intended applications, though they may be converted to Coordinates via `_.jtsGeom.getCoordinate`) using our method extensions:

```scala
val coordinates: Array[Coordinate] = ???
val triangulation = coordinates.delaunayTriangulation
```

`DelaunayTriangulation` objects contain a field `halfEdgeTable` of type `HalfEdgeTable` which can be used to interrogate the mesh structure. It is, however, necessary to have an entry point into this structure. Typically, we either use the `boundary` field of the triangulation object, or we call `triangulation.halfEdgeTable.edgeIncidentTo(v)`, where `v` is the index of a vertex (`triangulation.liveVertices` gives a `Set[Int]` listing the indices of vertices present in the triangulation). From there, the standard half edge navigation operations are available:

```scala
import triangulation.halfEdgeTable._

e = edgeIncidentTo(???)

getFlip(e)                                // Returns the complementary half edge of e
assert(e == getFlip(getFlip(e)))          // Identity regarding complementary edges
assert(getSrc(e) == getDest(getFlip(e)))  // Vertices of half edges are sane

getNext(e)                                // Returns the next half edge in the
                                          // triangle
assert(e == getNext(getNext(getNext(e)))) // True if e is an edge of a triangle
assert(getPrev(e) == getNext(getNext(e))) // True if e is an edge of a triangle

assert(rotCWSrc(e) == getNext(getFlip(e)) // Find the edge next in clockwise order
                                          // around the source vertex of e
                                          // sharing the same destination vertex
```

See the HalfEdgeTable documentation for more details.

Finally, triangulations obviously contain triangles. For ease of use, triangulation objects have a `triangles` field (or method) which return a `Seq[(Int, Int, Int)]` containing triples of vertex indices that are the vertices of all the triangles in the triangulation (the indices are listed in counterclockwise order).

Simplification

When the Coordinates composing a triangulation have a meaningful z-coordinate, it may be of interest to reduce the number of points in the mesh representation while inflicting the smallest amount of change to the surface. We accomplish this by sorting vertices according to their error, which is derived from a quadric error metric (see Garland, Michael, and Paul S. Heckbert. "Surface simplification using quadric error metrics." Proceedings of the 24th annual conference on Computer graphics and interactive techniques. ACM Press/Addison-Wesley Publishing Co., 1997). We remove the vertices with the smallest error using a Delaunay-preserving vertex removal, and iteratively apply this process until a desired number of vertices are removed.

Voronoi Diagrams

As mentioned, a Voronoi diagram is directly derived from a DelaunayTriangulation object. The VoronoiDiagram class is a thin veneer that exists only to extract the polygonal Voronoi cells corresponding to each vertex. Because of the

possibility of unbounded Voronoi cells around the boundaries of the Delaunay triangulation, we have opted to specify an extent at the time of construction of the VoronoiDiagram to which all the Voronoi cells will be clipped. Voronoi cells may be gathered individually, or all at once. These cells may also be collected with or without their corresponding point from the initial point set.

Euclidean Distance and Interpolation

A strong motivation for implementing Delaunay triangulations is to be able to furnish certain vector-to-raster operations.

EuclideanDistance allows us to build a raster where each tile cell contains the distance from that cell to the closest point in a point set. This is accomplished by rasterizing Voronoi cells using a distance function. Euclidean distance tiles may be computed using either the `coordinates.euclideanDistance(re: RasterExtent)` method extension or the `EuclideanDistanceTile(re: RasterExtent)` apply method.
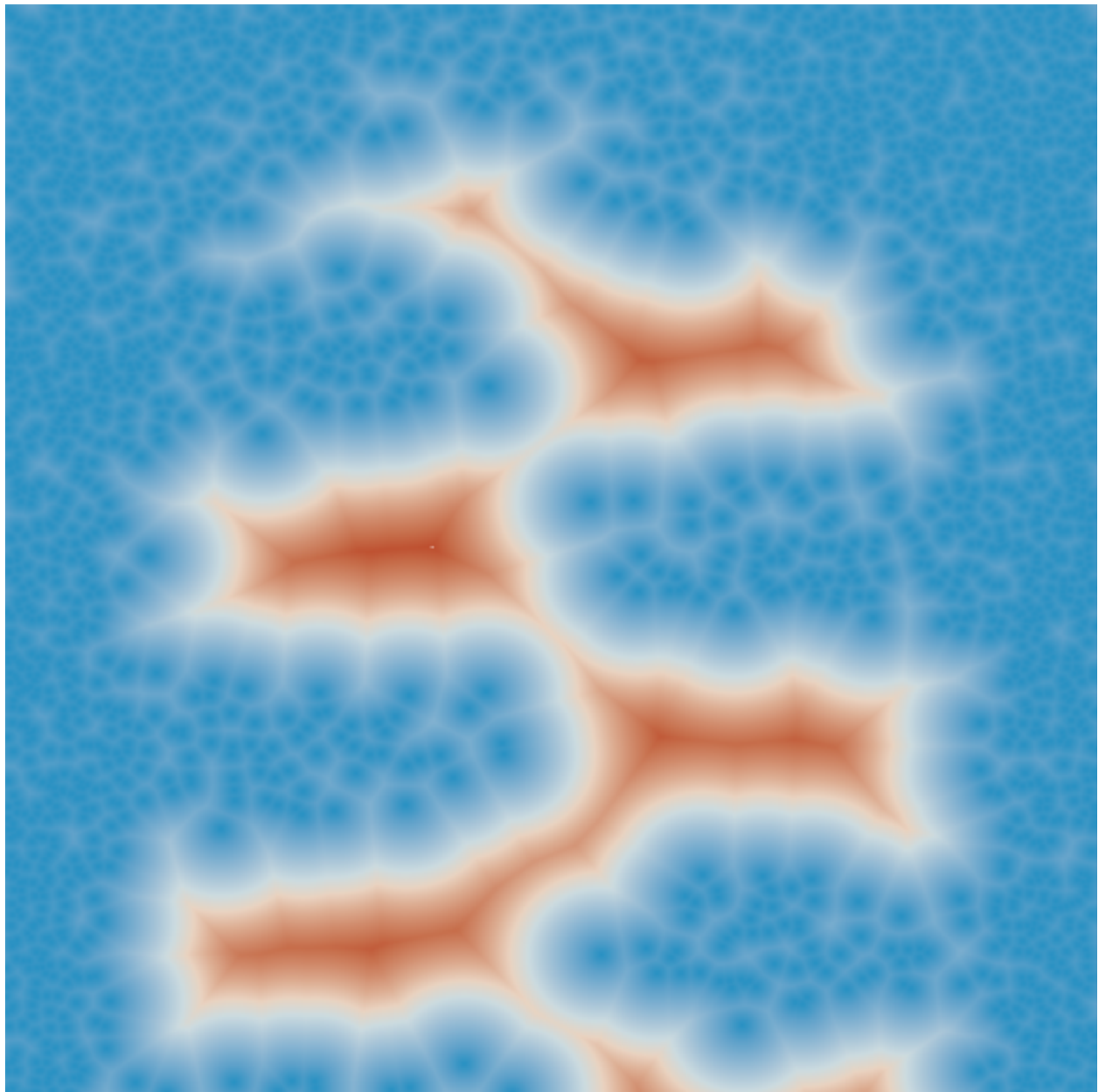
The other main class of vector-to-raster functions enabled by Delaunay triangulations is linear interpolation of unstructured samples from some function. We use the z-coordinate of our input points to store a Double attribute for each point, and we rasterize the Delaunay triangles to produce the final interpolation. The most obvious candidate is to use the z-coordinates to indicate the elevation of points on the globe; the rasterization of these values is a digital elevation map. This is the TIN algorithm for DEM generation. Using this method, we would apply one of the methods in `geotrellis.raster.triangulation.DelaunayRasterizer`.
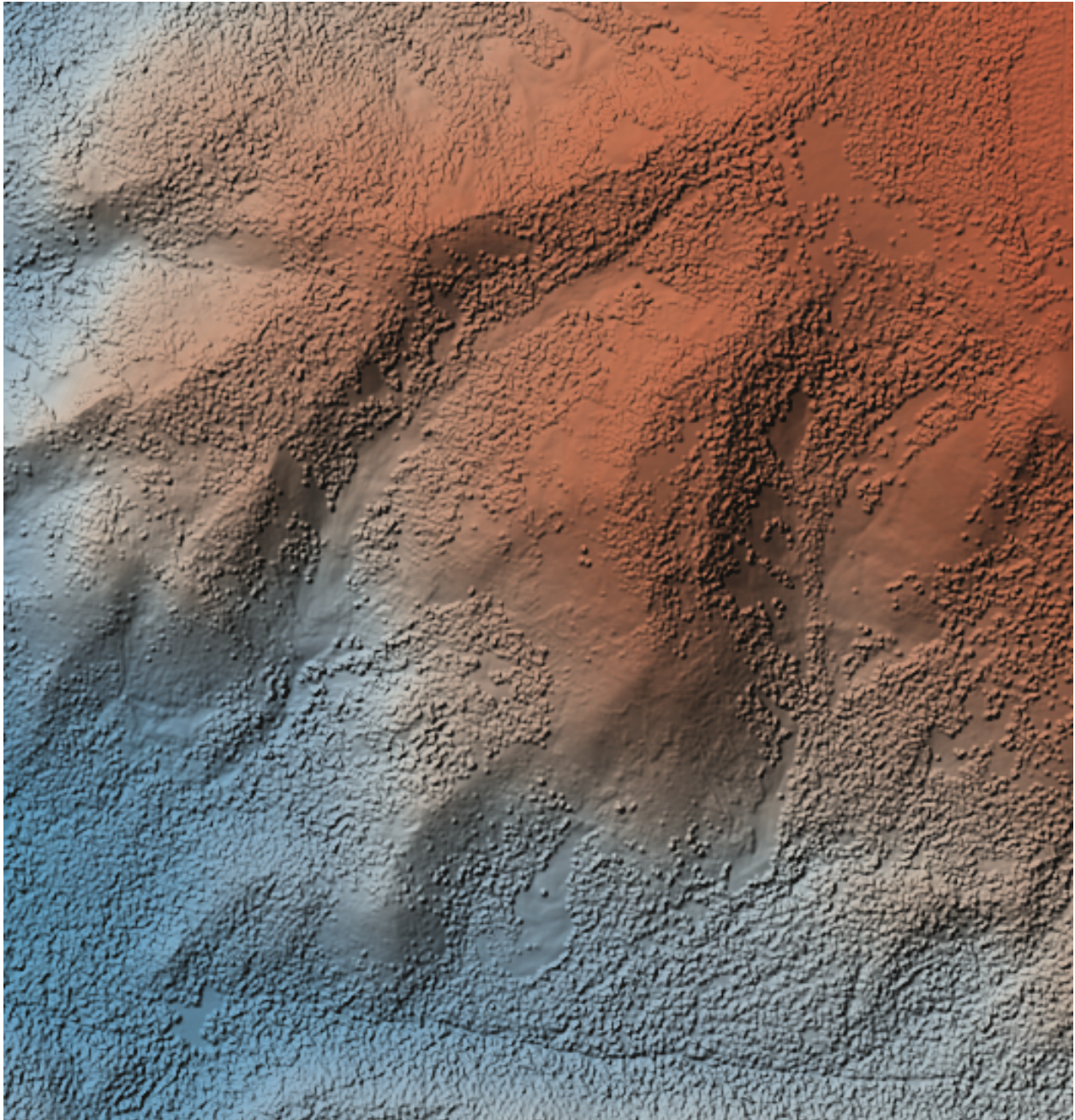
(The above image has been hillshaded to better show the detail in the elevation raster.)

The major advantage of using triangulations to interpolate is that it more gracefully handles areas with few or no samples, in contrast to a method such as inverse distance weighted interpolation, a raster-based technique. This is common when dealing with LiDAR samples that include water, which has spotty coverage due to the reflectance of water.

Distributed Computation

Among the design goals for this package was the need to handle extremely large point sets—on the order of 100s of millions of points. To accomplish this end, we opted for a distributed solution using Spark. Generally speaking, this interface will require the user to cut the incoming point set according to some LayoutDefinition into an `RDD[(SpatialKey, Array[Coordinate])]`. After triangulating each grid cell individually, facilities are provided to join the results—though in certain cases, the results will not be as expected (see Known Limitations below).

Given an `RDD[(SpatialKey, DelaunayTriangulation)]`, one is meant to apply the `collectNeighbors()` method to generate a map of nearby grid cells, keyed by `geotrellis.util.Direction`. These maps are then taken as input to StitchedDelaunay's apply method. This will join a 3x3 neighborhood of triangulations into a single triangulation by creating new triangles that fill in the gaps between the component triangulations. For instance, if we begin with the following collection of triangulations
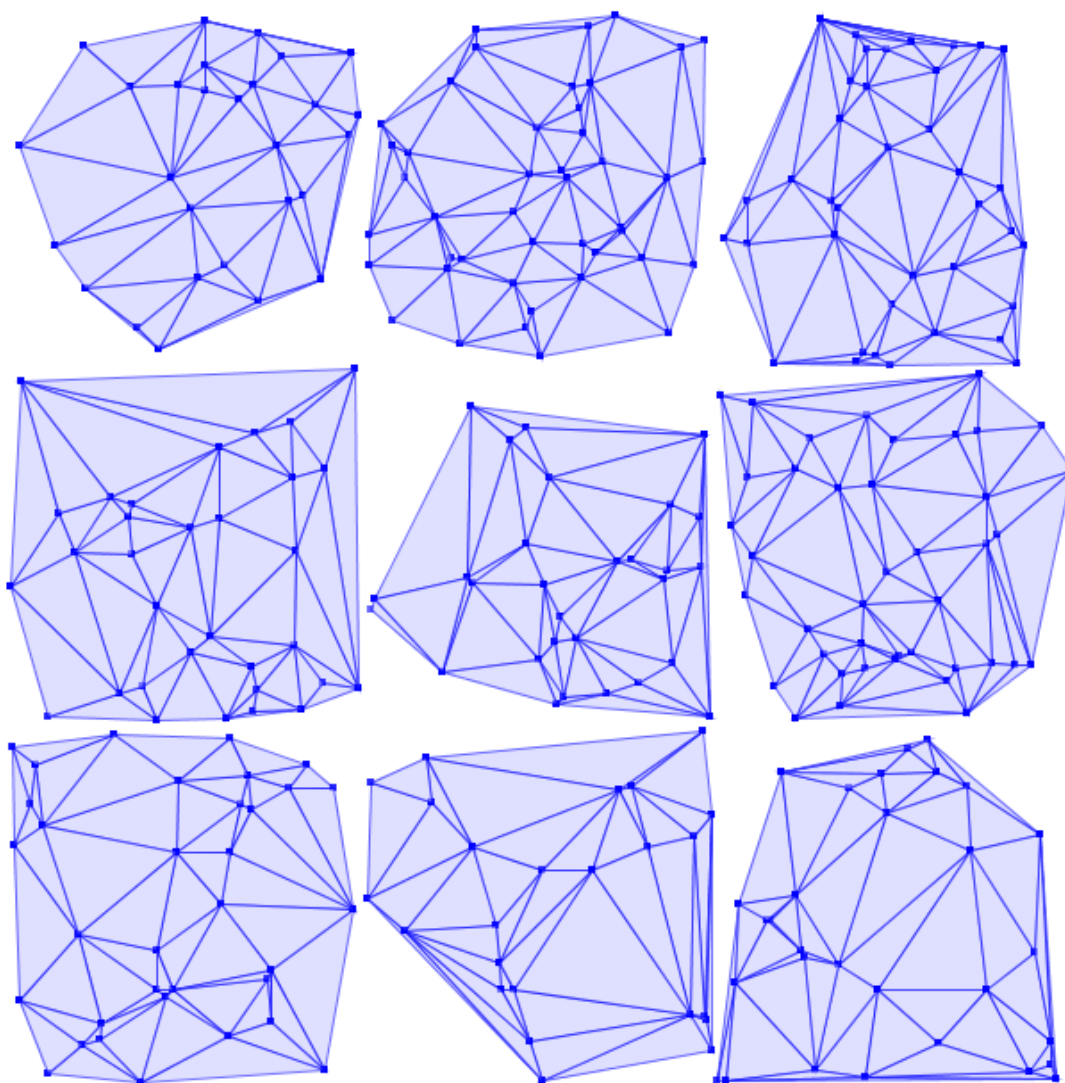
The stitch operation creates the stitch triangles shown in red below:
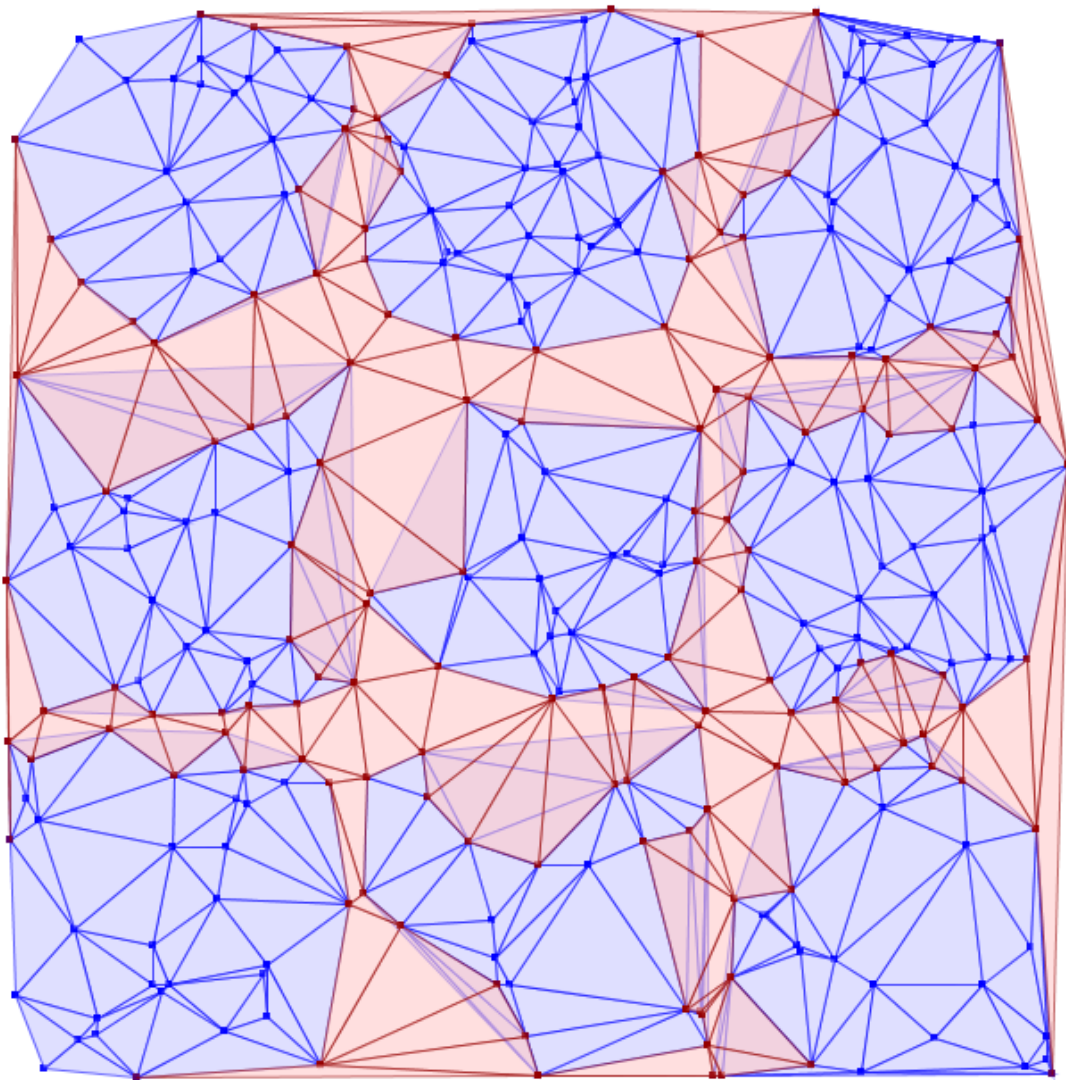
Notice that the stitch triangles overlap the base triangulations. This is expected since not all the base triangles are Delaunay with respect to the merged triangulation. Also keep in mind that in its current incarnation, StitchedDelaunay instances' `triangles` element contains only these fill triangles, not the triangles of the base triangulations.

Because the interior of these base triangulations is often not needed, and they can be very large structures, to reduce shuffle volume during the distributed operation, we introduced the BoundaryDelaunay structure. These are derived from DelaunayTriangulations and an extent that entirely contains the triangulation, and inside which no points will be added in a subsequent stitch operation. The BoundaryDelaunay object will be a reduced mesh where the interior is empty. This is for context, as it is not recommended to interact with BoundaryDelaunay objects directly; that way madness lies. Nonetheless, it is an important operation to include due to the massive memory savings and reduced network traffic.

The immediate application of StitchedDelaunay is the ability to perform both EuclideanDistance and interpolation tasks in a distributed context. We provide the `euclideanDistance(ld:  LayoutDefinition)` method extension taking an `RDD[(SpatialKey, Array[Coordinate])]` to an `RDD[(SpatialKey, Tile)]` (also available as the apply method on the `EuclideanDistance` object in the `geotrellis.spark.distance` package). The following image is one tile from such a Euclidean distance RDD. Notice that around the tile boundary, we are seeing the influence of points from outside the tile's extent.
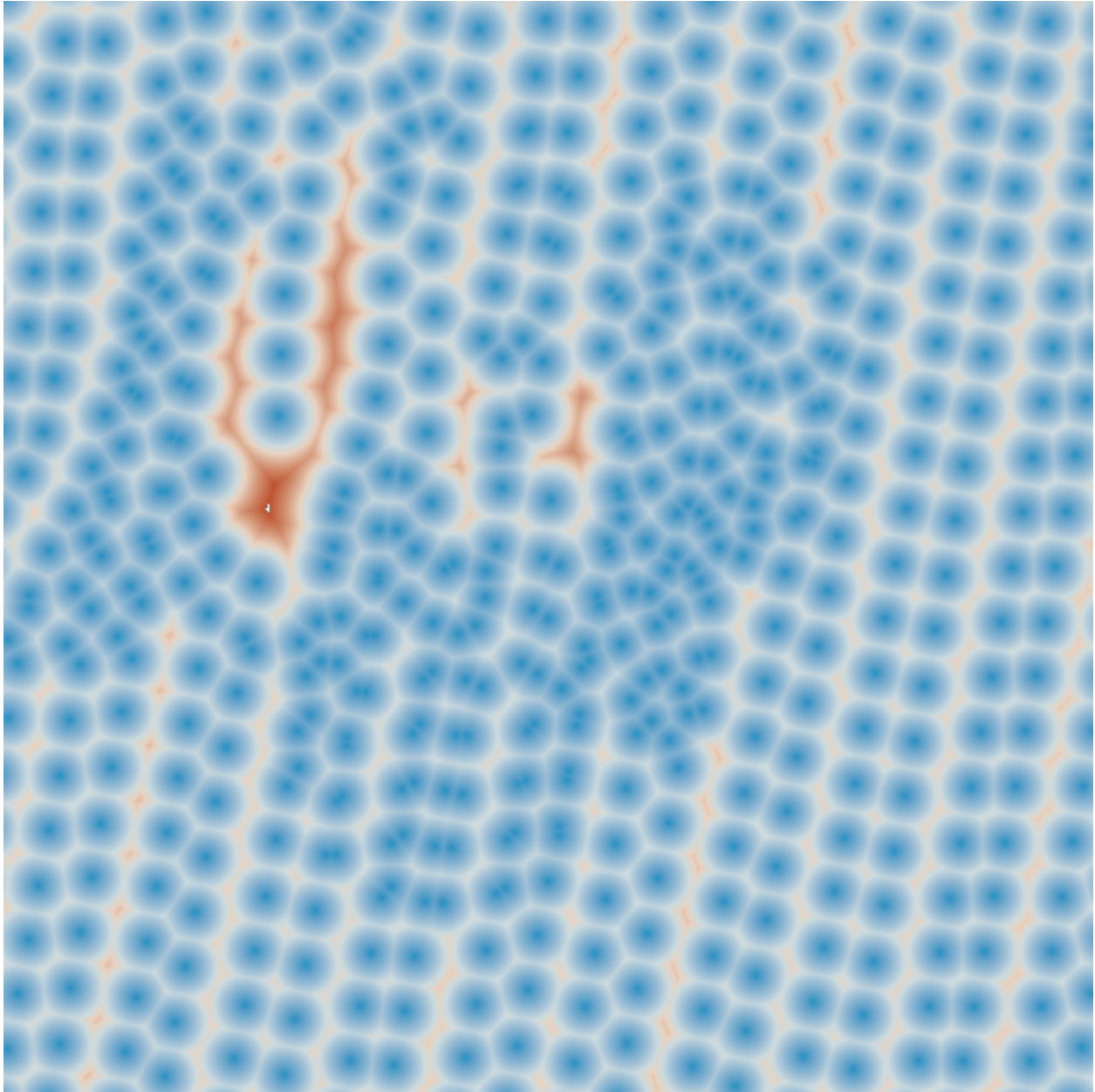
Keep in mind that one can rasterize non-point geometries as the basis for generic Euclidean distance computations, though this might start to be cost prohibitive if there are many polygonal areas in the input set.

Known Limitations

When designing this component, our aim was to handle the triangulation of dense, very large clouds of points with only small regions (relative to the layout definition) without any samples. That is to say, if there are occasional, isolated SpatialKeys that have no points, there is unlikely to be a problem. Multiple contiguous SpatialKeys with no points may cause problems. Specifically, in the case of Euclidean distance, if a tile has influence from outside the 3x3 area, there is likely to be errors. In the best case, there will be missing tiles, in the worst case, the Euclidean distance will simply be incorrect in certain areas.
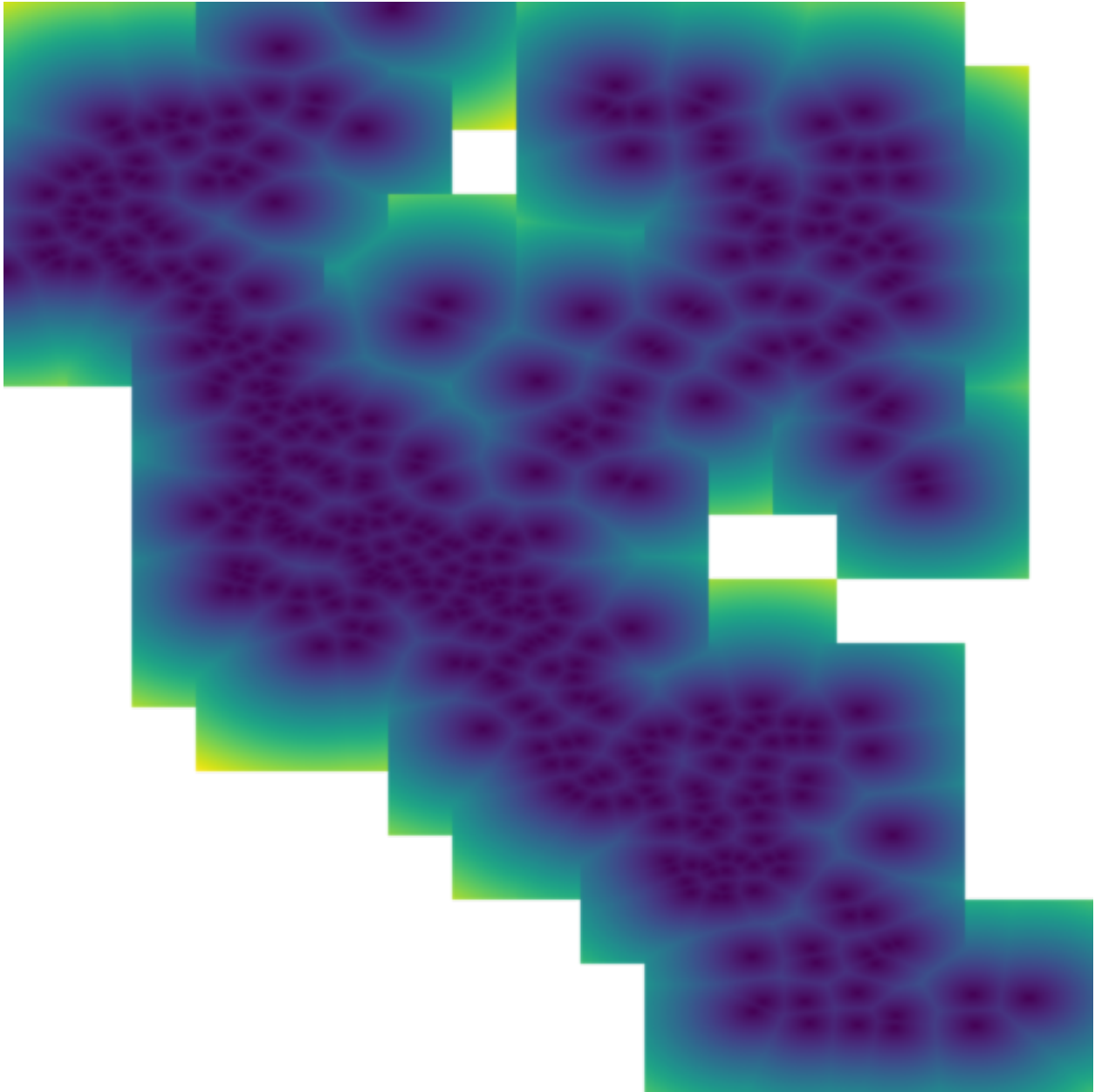
In this example, one can see that there are clear discontinuities in the values along some tile boundaries. The upshot is that these erroneous tiles are generated when `(SpatialKey(c, r), Array.empty[Coordinate])` is included in the source RDD. If the spatial key is simply not present, no tile will be generated at that location, and the incidence of erroneous tiles will be reduced, though not necessarily eliminated.

In cases where the point sample is small enough to be triangulated efficiently on a single node, we recommend using `geotrellis.spark.distance.SparseEuclideanDistance` to produce the Euclidean distance tiles. This will produce the desired result.

Numerical Issues

When dealing with large, real-world point sets (particularly LiDAR data), one is likely to encounter triangulation errors that arise from numerical issues. We have done our best to be conscious of the numerical issues that surround these triangulation operations, including porting Jonathan Shewchuk's robust predicates to Java, and offering some

sensible numerical thresholds and tolerance parameters (not always accessible from the interface). Specifically, the DelaunayTriangulation object allows a distance threshold to be set, defining when two points are considered the same (only one will be retained, with no means of allowing the user to select which one).

The two most common errors will arise from points that are too close together for the numerical predicates to distinguish them, but too far apart to be considered a single point. Notably, during distributed tasks, this will produce stitch triangles which overlap the patches being joined. These errors arise from a known place in the code and can be dealt with by altering numerical thresholds, but there is currently no handle in the interface for setting these values.

3.11 Spark and GeoTrellis

This documentation series describes the use of the vast `geotrellis.spark` module.

3.11.1 On Distributed Computation

Note: Distributed computing is difficult to get right. Luckily, we are able to lean on the `RDD` abstraction provided by Apache Spark to simplify matters somewhat. Still, the conceptual difficulties in `geotrellis.spark` are arguably as great as can be found in any part of the GeoTrellis library. As such, the discussion in this portion of the documentation assumes a passing familiarity with the key concepts of `geotrellis.raster`. If this is a difficulty, please refer to the documentation for the geotrellis.raster package.

Consider the (relatively) simple case of carrying out local addition on two raster tiles. In the abstract, this merely involves adding together corresponding values from two different `Tiles`. Practically, things can quickly become more complex: what if one `Tile`'s data covers a larger extent than the other? In general, how do we determine what 'corresponding values' means in such a context? (Some specifics related to this question are covered in the `geotrellis.spark` *documentation on joins*)

What we need, then, is to pass around tiles as well as some kind of associated data. In addition, the `Tile` abstraction makes sense only in a particular place (in space and/or time) - the data in my `Tile` represents the elevation of terrain in this or that actual place which has such and such spatial relations to other `Tiles` that represent neighboring terrain elevations. If your application for finding directions displayed street data for Beijing in the middle of downtown Philadelphia, it would be extremely difficult to actually use. From the perspective of application performance during spatially-aware computations (say, for instance, that I want to compute the average elevation for every `Tile`'s cell within a five mile radius of a target location) it is also useful to have an index which provides a sort of shortcut for tracking down the relevant information.

The need for intelligently indexed tiles is especially great when thinking about distributing work over those tiles across multiple computers. The tightest bottleneck in such a configuration is the communication between different nodes in the network. What follows is that reducing the likelihood of communication between nodes is one of the best ways to improve performance. Having intelligently indexed tilesets allows us to partition data according to expectations about which `Tiles` each node will require to calculate its results.

Hopefully you're convinced that for a wide variety of GeoTrellis use-cases it makes sense to pass around tiles with indices to which they correspond as well as metadata. This set of concerns is encoded in the type system as `RDD[(K, V)] with Metadata[M]`.

For more information on this type, see Tile Layers.

3.11.2 Writing Layers

The underlying purpose of `geotrellis.spark.io` is to provide reading and writing capability for instances of `RDD[(K, V)] with Metadata[M]` into one of the distributed storage formats.

GeoTrellis provides an abstraction for writing layers, `LayerWriter`, that the various backends implement. There are a set of overloads that you can call when writing layers, but generally you need to have the target `LayerId` that you will be writing to, and the `RDD[(K, V)] with Metadata[M]` that you want to write. Note that the `K`, `V`, and `M` concrete types need to have all of the context bounds satisfied; see the method signature in code or look to the implicit argument list in the ScalaDocs to find what the context bounds are (although if you are not using custom types, on the required imports should be necessary to satisfy these conditions). The overloaded methods allow you to optionally specify how the key index will be created, or to supply your own `KeyIndex`.

Key Index

A `KeyIndex` determines how your N-dimensional key (the `K` in `RDD[(K, V)] with Metadtaa[M]`) will be translated to a space filling curve index, represented by a `Long`. It also determines how N-dimensional queries (represented by `KeyBounds` with some minimum key and maximum key) will translate to a set of ranges of `Long` index values.

There are two types of key indexes that GeoTrellis supports, which represent the two types of space filling curves supported: Z-order Curves and Hilbert Curves. The Z-order curves can be used for 2 and 3 dimensional spaces (e.g. those represented by `SpatialKeys` or `SpaceTimeKeys`). Hilbert curves can represent N-dimensions, although there is currently a limitation in place that requires the index to fit into a single `Long` value.

In order to index the space of an `RDD[(K, V)] with Metadata[M]`, we need to know the bounds of the space, as well as the index method to use.

The LayerWriter methods that do not take a `KeyIndex` will derive the bounds of the layer to be written by the layer itself. This is fine if the layer elements span the entire space that the layer will ever need to write to. If you have a larger space that represents the layer, for instance if you want to write elements to the layer that will be outside the bounds of the original layer RDD, you will need to create a `KeyIndex` manually that represents the entire area of the space.

For example, say we have a spatio-temporal raster layer that only contains elements that partially inhabit the date range for which we will want the layer to encompass. We can use the `TileLayout` from the layer in combination with a date range that we know to be sufficient, and create a key index.

```
import geotrellis.raster.Tile
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.io.index.ZCurveKeyIndexMethod
import geotrellis.util._
import org.apache.spark.rdd.RDD
import org.joda.time.DateTime

val layer: RDD[(SpaceTimeKey, Tile)] with Metadata[TileLayerMetadata[SpaceTimeKey]] =␣
→???

// Create the key index with our date range
val minDate: DateTime = new DateTime(2010, 12, 1, 0, 0)
val maxDate: DateTime = new DateTime(2010, 12, 1, 0, 0)

val indexKeyBounds: KeyBounds[SpaceTimeKey] = {
  val KeyBounds(minKey, maxKey) = layer.metadata.bounds.get // assuming non-empty␣
→layer
  KeyBounds(
    minKey.setComponent[TemporalKey](minDate),
    maxKey.setComponent[TemporalKey](maxDate)
  )
}
```

```
val keyIndex =
  ZCurveKeyIndexMethod.byMonth
    .createIndex(indexKeyBounds)

val writer: LayerWriter[LayerId] = ???
val layerId: LayerId = ???

writer.write(layerId, layer, keyIndex)
```

3.11.3 Reindexing Layers

If a layer was written with bounds on a key index that needs to be expanded, you can reindex that layer. The `LayerReindexer` implementation of the backend you are using can be passed in a `KeyIndex`, which can be constructed similarly to the example above.

3.11.4 Reading Layers

Layer readers read all or part of a persisted layer back into `RDD[(K, V)] with Metadata[M]`. All layer readers extend the FilteringLayerReader trait which in turn extends LayerReader. The former type should be used when abstracting over the specific back-end implementation of a reader with region query support, and the latter when referring to a reader that may only read the layers fully.

In order to read a layer correctly some metadata regarding the type and format of the values must be stored as well as metadata regarding layer properties. All layer readers lean on instances of AttributeStore to provide this functionality. As a convenience each concrete type of a `LayerReader` will provide a constructor that will instantiate an `AttributeStore` of the same type with reasonable defaults. For instance `S3LayerReader` constructor, which requires S3 bucket and prefix parameters, would instantiate an `S3AttributeStore` with the bucket and prefix.

LayerReader

```
import geotrellis.raster._
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.io.s3._

val reader: FilteringLayerReader[LayerId] = S3LayerReader("my-bucket", "catalog-prefix
↪")

val rdd: RDD[(SpatialKey, Tile)] with Metadata[TileLayerMetadata[SpatialKey]] =
  reader.read[SpatialKey, Tile, TileLayerMetadata[SpatialKey]](LayerId("NLCD", 10))
```

Type signature of `rdd` variable can be inferred from the assignment and may be omitted but the type parameters for the `read` method can not be inferred and are required. Furthermore, the `reader.read` method will use these explicitly provided type parameters to find implicit type class instances that will allow it to read records of that format.

It's important to note that as a result of call to `reader.read` some IO will happen right away in order to read the layer attributes from the `AttributeStore`. However, the result of the call is an RDD, a description of the distributed collection at some point in the future. Consequently the distributed store (like HDFS or S3) will not touched until some spark "action" is called on either `rdd` or one of its descendants.

But what happens when IO gremlins strike and the type of the record stored does not match the type parameter? It depends. The layer reader will do its best to read the layer as instructed, possibly failing. Most likely this effort will

result in `org.apache.avro.AvroTypeException` if the Avro schema of the specified value does not match the schema of the stored value or a `spray.json.DeserializationException` if the JSON format of the metadata does not match the JSON value stored in the `AttributeStore`. This behavior is somewhat unhelpful but it future proofs the persisted data in so far that records may be reified into types that differ from their original implementations and names, as long as correct their formats are specified correctly for the records written.

If the type of the layer can not be assumed to be known it is possible to inspect the layer through `reader.attributeStore` field.

```
val header = reader.attributeStore.readHeader[LayerHeader]
assert(header.keyClass == "geotrellis.spark.SpatialKey")
assert(header.valueClass == "geotrellis.raster.Tile")
```

LayerReader.reader

In addition to `reader.read` there exists a `reader.reader` method defined as follows:

```
def reader[
  K: AvroRecordCodec: Boundable: JsonFormat: ClassTag,
  V: AvroRecordCodec: ClassTag,
  M: JsonFormat: GetComponent[?, Bounds[K]]
]: Reader[ID, RDD[(K, V)] with Metadata[M]] =
  new Reader[ID, RDD[(K, V)] with Metadata[M]] {
    def read(id: ID): RDD[(K, V)] with Metadata[M] =
      LayerReader.this.read[K, V, M](id)
  }
```

In effect we would be using a reader to produce a reader, but critically the `read` method on the constructed reader does not have any type class parameters. This is essentially a way to close over all of the formats for `K`, `V`, and `M` such that a "clean" reader can be passed to modules where those formats are not available in the implicit scope.

FilteringLayerReader

```
import geotrellis.vector._
import geotrellis.spark.io._
import geotrellis.spark.io.s3._

val reader: FilteringLayerReader[LayerId] = S3LayerReader("my-bucket", "catalog-prefix
↪")
val layerId = LayerId("NLCD", 10)

val rdd: RDD[(SpatialKey, Tile)] with Metadata[TileLayerMetadata[SpatialKey]] =
  reader
    .query[SpatialKey, Tile, TileLayerMetadata[SpatialKey]](LayerId("NLCD", 10))

    .result
```

When using the `.query` method the expected return types must still be provided just like we did when calling `.read`, however instead of producing an `RDD` it produced an instance of LayerQuery which is essentially a query builder in a fluent style, allowing for multiple '.where' clauses to be specified. Only when `.result` is called will an `RDD` object be produced. When multiple `.where` clauses are used, the query specified their intersection.

This behavior allows us to build queries that filter on space and time independently.

```
import org.joda.time.DateTime

val time1: DateTime = ???
val time2: DateTime = ???

val rdd: RDD[(SpaceTimeKey, Tile)] with Metadata[TileLayerMetadata[SpaceTimeKey]] =
  reader
    .query[SpaceTimeKey, Tile, TileLayerMetadata[SpaceTimeKey]](LayerId("Climate_
→CCSM4-RCP45-Temperature-Max", 8))
    .where(Intersects(Extent(-85.32,41.27,-80.79,43.42)))
    .where(Between(time1, time2))
    .result
```

Other query filters are supported through the LayerFilter type class. Implemented instances include:

- `Contains`: Tile which contains a point

- `Between`: Tiles between two dates

- `At`: Tiles at a a specific date

- `Intersects`: Tiles intersecting . . .

- `KeyBounds`

- `GridBounds`

- `Extent`

- `Polygon`

3.11.5 Value Readers

Unlike layer readers, which produce a future distributed collection, an `RDD`, a tile reader for a layer is essentially a reader provider. The provided reader is able to read a single value from a specified layer.

```
import geotrellis.raster._
import geotrellis.spark._
import geotrellis.spark.io.s3._

val attributeStore = S3AttributeStore("my-bucket", "catalog-prefix")
val nlcdReader: Reader[SpatialKey, Tile] = S3ValueReader[SpatialKey,
→Tile](attributeStore, LayerId("NLCD", 10))
val tile: Tile = nlcdReader.read(SpatialKey(1,2))
```

`ValueReader` class is very useful for creating an endpoint for a tile server because it both provides a cheap low latency access to saved tiles and does not require an instance of `SparkContext` to operate.

If you wish to abstract over the backend specific arguments but delay specification of the key and value types you may use an alternative constructor like os:

```
val attributeStore = S3AttributeStore("my-bucket", "catalog-prefix")
val readerProvider: ValueReader[LayerId] = S3ValueReader(attributeStore)
val nlcdReader: Reader[SpatialKey, Tile] = readerProvider.reader[SpatialKey,
→Tile](LayerId("NLCD", 10))
val tile: Tile = nlcdReader.read(SpatialKey(1,2))
```

The idea is similar to the `LayerReader.reader` method except in this case we're producing a reader for single tiles. Additionally it must be noted that the layer metadata is accessed during the construction of the `Reader[SpatialKey, Tile]` and saved for all future calls to read a tile.

Reader Threads

Cassandra and S3 Layer RDDReaders / RDDWriters are configurable by threads amount. It's a programm setting, that can be different for a certain machine (depends on resources available). Configuration could be set in the `reference.conf` / `application.conf` file of your app, default settings available in a `reference.conf` file of each backend subproject (we use TypeSafe Config). For a File backend only RDDReader is configurable, For Accumulo - only RDDWriter (Socket Strategy). For all backends CollectionReaders are configurable as well. By default thread pool size per each configurable reader / writer equals by virtual machine cpu cores available. Word `default` means thread per cpu core, it can be changed to any integer value.

Default configuration example:

```
geotrellis.accumulo.threads {
  collection.read = default
  rdd.write       = default
}
geotrellis.file.threads {
  collection.read = default
  rdd.read        = default
}
geotrellis.hadoop.threads {
  collection.read = default
}
geotrellis.cassandra.threads {
  collection.read = default
  rdd {
    write = default
    read  = default
  }
}
geotrellis.s3.threads {
  collection.read = default
  rdd {
    write = default
    read  = default
  }
}
```

Cassandra has additional configuration settings:

And additional connections parameters for `Cassandra`:

```
geotrellis.cassandra {
  keyspace            = "geotrellis"
  replicationStrategy = "SimpleStrategy"
  replicationFactor   = 1
  localDc             = "datacenter1"
  usedHostsPerRemoteDc = 0
  allowRemoteDCsForLocalConsistencyLevel = false
}
```

Consider using `hbase.client.scanner.caching` parameter for `HBase` as it may increase scan performance.

3.11.6 RDD Joins

In `geotrellis.spark` we represent a raster layer as a distributed collection of non-overlapping tiles indexed by keys according to some `TileLayout`. For instance a raster layer is represented as `RDD[(SpatialKey, Tile)]`.

With this setup, we can represent certain decisions about how operations between layers should be performed in terms of the sort of 'join' to be performed.
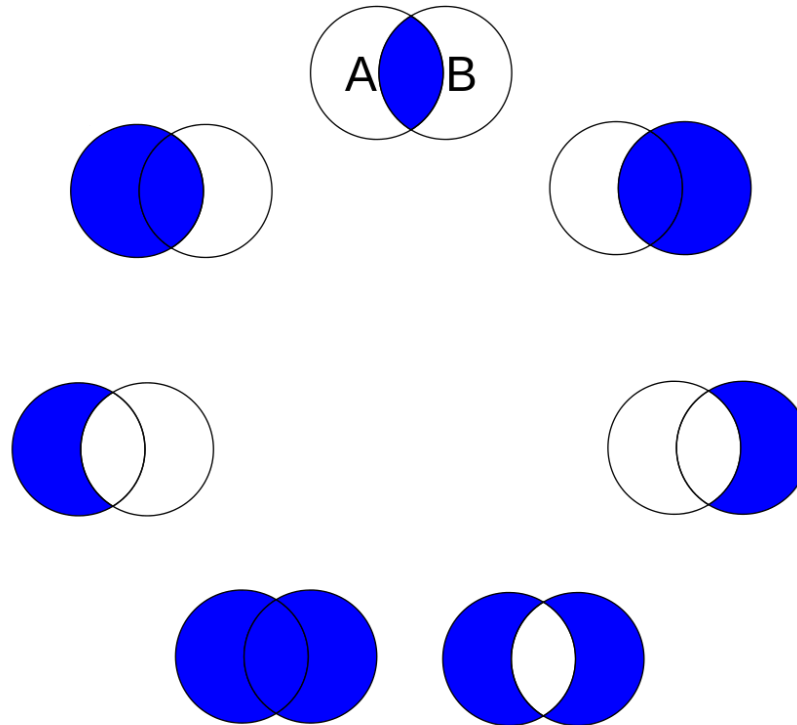
Fig. 4: visualized joins

First, we'll set the stage for a discussion of joins in `geotrellis.spark` with a discussion of how metadata is used in this context.

Metadata

A previously tiled and saved `RasterRDD` read in through an instance of `geotrellis.spark.io.LayerReader` will be mixed in with the `Metadata[RasterMetaData]` trait. This metadata describes the TileLayout used by the layer, the extent it covers, the CRS of its projection, and what the CellType of each tile is. This metadata allows us to verify that we're working with compatible layers.

```
import org.apache.spark._
import org.apache.spark.rdd._

import geotrellis.raster._
import geotrellis.spark.io._
import geotrellis.spark.io.s3._

implicit val sc: SparkContext = ???

val reader : S3LayerReader[SpatialKey, Tile, TileLayerMetadata[SpatialKey]] =
```

(continues on next page)

```
  S3LayerReader.spatial("bucket", "prefix")

def getLayerId(idx: Int): LayerId = ???

val rdd1 =
  reader.read(getLayerId(1))

val rdd2: RDD[(SpatialKey, Tile)] with Metadata[TileLayerMetadata] =
  reader.read(getLayerId(2))

val rdd3: TileLayerRDD[SpatialKey] =
  reader.read(getLayerId(3))
```

Default Joins

GeoTrellis provides an API for interaction with RDDs of tiles as a single unit. Where possible, we attempt to provide symbolic methods where their meaning is obvious and explicit method names in *all* cases.

```
import geotrellis.spark.mapalgebra.local._

 rdd1 + 1            // add 1 to every cell in the tiled raster
 rdd1 localAdd 1    // explicit method name for above operation
 rdd1 + rdd2        // do a cell wise local addition between two rasters
 rdd1 localAdd rdd2 // explicit method name for above operation
 List(rdd1, rdd2, rdd3).localAdd
 // all results are of type RDD[(SpatialKey, Tile)]
```

Other supported operations can been found in the `geotrellis.spark.mapalgebra` package and its sub-packages.

In order to provide this concise and intuitive syntax for map algebra operations between two layers some assumptions need to be made regarding the mechanics of the join. So, by default, GeoTrellis will use the spark implementation of inner join deferring to spark for the production of an appropriate partitioner for the result. Thus, if two layers being operated on are not aligned the result of the operation will contain **only** the intersecting tiles.

Explicit Joins

In cases where it is important to control the type of join a more explicit method is required. We make a direct call to `geotrellis.raster.mapalgebra.local.Add.apply` to perform per tile operations.

Because all binary operations must have the shape of `(V, V) => R` we provide an extension method on `RDD[(K, (V, V))]` that decomposes the tuple resulting from the join and uses it to call a function taking two arguments.

```
import geotrellis.raster.mapalgebra.local._

// using spark API
rdd1.join(rdd2).mapValues { case (tile1: Tile, tile2: Tile) => Add(tile1, tile2) }

// using GeoTrellis method extensions
rdd1.join(rdd2).combineValues(Add(_, _))
```

Left Join

Another reason to want to control a join is to perform an update of a larger layer with a smaller layer, performing an operation where two intersect and capturing resulting values.

This case is captured by a left outer join. If the right-side of the join row tuple is `None` we return the left-side tile unchanged. Consequently the extension method `updateValues` will only accept operations with signature of `(V, V) => V`.

```
// using spark API
rdd1.leftOuterJoin(rdd2).mapValues { case (tile1: Tile, optionTile: Option[Tile]) =>
  optionTile.fold(tile1)(Add(tile1, _))
}

// using GeoTrellis method extensions
rdd1.leftOuterJoin(rdd2).updateValues(Add(_, _))
```

Spatial Join

Given that we know the key bounds of our RDD, from accompanying `TileLayerMetadata`, before performing the join we may use a spark `Partitioner` that performs space partitioning. Such a partitioner has a number of benefits over standard `HashPartitioner`:

- Scales the number of partitions with the number of records in the RDD

- Produces partitions with spatial locality which allow:

    - Faster focal operations

    - Shuffle free joins with other spatially partitioned RDDs

    - Efficient spatial region filtering

Because the partitioner requires ability to extract `Bounds` of the original RDD from it's `Metadata` it is able to provide the `Bounds` of the join result. Since the result of a join may be empty the user must match on the resulting `Bounds` object to find out if it's `EmptyBounds` or `KeyBounds[SpatialKey]`.

```
import geotrellis.spark.partitioner._

val joinRes: RDD[(SpatialKey, (Tile, Tile))] with Metadata[Bounds[SpatialKey]] =
  rdd1.spatialJoin(rdd2)

val leftJoinRes: RDD[(SpatialKey, (Tile, Option[Tile])] with
↪Metadata[Bounds[SpatialKey]] =
  rdd1.spatialLeftOuterJoin(rdd2)
```

Manipulating Metadata

Metadata is provided when loading a layer from a GeoTrellis layer reader and is required when writing a layer through a GeoTrellis layer writer. The user bears responsibility that it is preserved and remains consistent through transformations if such behavior is desired.

The concrete implementation of `RDD[(K, V)] with Metadata[M]` signature in GeoTrellis is `ContextRDD[K, V, M]`

```
val rdd: RDD[(SpatialKey, Tile)] = rdd1 localAdd rdd2
val rddWithContext: RDD[(SpatialKey, Tile)] with Metadata[TileLayerMetadata] =
  ContextRDD(rdd, rdd1.metadata)
```

Preserving Metadata Through Operations

There are extension methods in `RDD[(K, V]] with Metadata[M]` that allow either changing rdd while preserving metadata or changing metadata while preserving the rdd.

```
// .withContext preserves the RDD context, the Metadata
val rddWithContext1: RDD[(SpatialKey, Tile)] with Metadata[TileLayerMetadata] =
  rdd1.withContext { _ localAdd rdd2 }

val rddWithContext2: RDD[(SpatialKey, Tile)] with Metadata[TileLayerMetadata] =
  rdd1.withContext { _ localAdd rdd2 localAdd rdd3 }


// .mapContext allows to chain changing Metadata after an operation
// example: localEqual will produce tiles with CellType of TypeBit
val rddWithContext3: RDD[(SpatialKey, Tile)] with Metadata[TileLayerMetadata] =
  rdd1
    .withContext { _ localEqual 123 }
    .mapContext { metadata: TileLayerMetadata => metadata.copy(cellType = TypeBit) }
```

Preserving Metadata Through Spatial Joins

Since spatial joins produce metadata, in contrast to vanilla spark joins, we must use `.withContext` wrapper at every transformation in order to allow the updated `Bounds` to flow to the end where they can be used.

For instance lets assume we wrote `updateLayout` that combines `Bounds[SpatialKey]` and `LayoutDefinition` from `TileLayerMetadata` to produce an RDD with updated, smaller `TileLayout`.

```
def updateLayout(md: TileLayerMetadata, bounds: Bounds[SpatialKey]):␣
→TileLayerMetadata = ???

val rddWithContext: RDD[(SpatialKey, Tile)] with Metadata[TileLayerMetadata] =
  rdd1
    .spatialJoin(rdd2).withContext { _.combineValues(Add(_, _)) }
    .spatialJoin(rdd3).withContext { _.combineValues(Add(_, _)) }
    .mapContext{ bounds => updateLayout(rdd1.metadata, bounds) }
```

3.11.7 Example Use Cases

Summaries from Temporal Overlap of Tiles

Sometimes you'd like to take a layer that has multiple tiles over the same spatial area through time, and reduce it down to a layer that has only value per pixel, using some method of combining overlapping pixels. For instance, you might want to find the maximum values of a pixel over time.

The following example shows an example of taking temperature data over time, and calculating the maximum temperature per pixel for the layer:

```
import geotrellis.raster._
import geotrellis.spark._
import geotrellis.util._

import org.apache.spark.rdd.RDD

val temperaturePerMonth: TileLayerRDD[SpaceTimeKey] = ???

val maximumTemperature: RDD[(SpatialKey, Tile)] =
  temperaturePerMonth
    .map { case (key, tile) =>
      // Get the spatial component of the SpaceTimeKey, which turns it into SpatialKey
      (key.getComponent[SpatialKey], tile)
    }
    // Now we have all the tiles that cover the same area with the same key.
    // Simply reduce by the key with a localMax
    .reduceByKey(_.localMax(_))
```

Stiching Tiles into a single GeoTiff

This example will show how to start with an `RDD[(ProjectedExtent, Tile)]` and end with a stitched together GeoTiff.

Note: Stitching together an RDD can produce a tile that is far bigger than the driver program's memory can handle. You should only do this with small layers, or a filtered RDD.

```
import geotrellis.raster._
import geotrellis.raster.io.geotiff._
import geotrellis.raster.resample._
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.tiling._
import geotrellis.vector._
import org.apache.spark.HashPartitioner
import org.apache.spark.rdd.RDD

val rdd: RDD[(ProjectedExtent, Tile)] = ???

// Tile this RDD to a grid layout. This will transform our raster data into a
// common grid format, and merge any overlapping data.

// We'll be tiling to a 512 x 512 tile size, and using the RDD's bounds as the tile␣
→bounds.
val layoutScheme = FloatingLayoutScheme(512)

// We gather the metadata that we will be targeting with the tiling here.
// The return also gives us a zoom level, which we ignore.
val (_: Int, metadata: TileLayerMetadata[SpatialKey]) =
  rdd.collectMetadata[SpatialKey](layoutScheme)

// Here we set some options for our tiling.
// For this example, we will set the target partitioner to one
// that has the same number of partitions as our original RDD.
val tilerOptions =
```

```
  Tiler.Options(
    resampleMethod = Bilinear,
    partitioner = new HashPartitioner(rdd.partitions.length)
  )

// Now we tile to an RDD with a SpaceTimeKey.

val tiledRdd =
  rdd.tileToLayout[SpatialKey](metadata, tilerOptions)


// At this point, we want to combine our RDD and our Metadata to get a␣
→TileLayerRDD[SpatialKey]

val layerRdd: TileLayerRDD[SpatialKey] =
  ContextRDD(tiledRdd, metadata)

// Now we can save this layer off to a GeoTrellis backend (Accumulo, HDFS, S3, etc)
// In this example, though, we're going to just filter it by some bounding box
// and then save the result as a GeoTiff.

val areaOfInterest: Extent = ???

val raster: Raster[Tile] =
  layerRdd
    .filter()                               // Use the filter/query API to
    .where(Intersects(areaOfInterest))   // filter so that only tiles intersecting
    .result                                 // the Extent are contained in the result
    .stitch                // Stitch together this RDD into a Raster[Tile]

GeoTiff(raster, metadata.crs).write("/some/path/result.tif")
```

Median Filter over Multiband Imagery

This example shows how to take some multiband imagery that exists in a layer, filter it with some upper bound threshold, and then apply a 5x5 median filter.

```
import geotrellis.spark._
import geotrellis.raster._
import geotrellis.raster.mapalgebra.focal.Square

val imageLayer: MultibandTileLayerRDD[SpaceTimeKey] = ???
val neighborhood = Square(2)

val resultLayer: MultibandTileLayerRDD[SpaceTimeKey] =
  imageLayer
    .withContext { rdd =>
      rdd.mapValues { tile =>
        tile.map { (band, z) =>
          if(z > 10000) NODATA
          else z
        }
      }
      .bufferTiles(neighborhood.extent)
      .mapValues { bufferedTile =>
```

```
      bufferedTile.tile.mapBands { case (_, band) =>
        band.focalMedian(neighborhood, Some(bufferedTile.targetArea))
      }
    }
  }
```

Region Query and NDVI Calculation

```
import geotrellis.raster._
import geotrellis.raster.io.geotiff._
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.util._
import geotrellis.vector._
import org.joda.time.DateTime

val region: MultiPolygon = ???
val layerReader: FilteringLayerReader[LayerId] = ???
val layerId: LayerId = LayerId("layerName", 18) // Querying zoom 18 data

val queryResult: MultibandTileLayerRDD[SpaceTimeKey] =
  layerReader.query[SpaceTimeKey, MultibandTile,
→TileLayerMetadata[SpaceTimeKey]](layerId)
    .where(Intersects(region))
    .where(Between(new DateTime(2016, 3, 1, 0, 0, 0), new DateTime(2016, 4, 1, 0, 0)))
    .result

val raster: Raster[Tile] =
  queryResult
    .mask(region)
    .withContext { rdd =>
      rdd
        .mapValues { tile =>
          // Assume band band 4 is red and band 5 is NIR
          tile.convert(DoubleConstantNoDataCellType).combine(4, 5) { (r, nir) =>
            (nir - r) / (nir + r)
          }
        }
        .map { case (key, tile) => (key.getComponent[SpatialKey], tile) }
        .reduceByKey(_.localMax(_))
    }
    .stitch

GeoTiff(raster, queryResult.metadata.crs).write("/path/to/result.tif")
```

3.12 The ETL Tool

When working with GeoTrellis, often the first task is to load a set of rasters to perform reprojection, mosaicing and pyramiding before saving them as a GeoTrellis layer. It is possible, and not too difficult, to use core GreoTrellis features to write a program to accomplish this task. However, after writing a number of such programs we noticed two patterns emerge:

- Often an individual ETL process will require some modification that is orthogonal to the core ETL logic

- When designing an ETL process it is useful to first run it a smaller dataset, perhaps locally, as a verification

- Once written it would be useful to re-run the same ETL process with different input and output storage media

To assist these patterns `spark-etl` project implements a plugin architecture for tile input sources and output sinks which allows you to write a compact ETL program without having to specify the type and the configuration of the input and output at compile time. The ETL process is broken into three stages: `load`, `tile`, and `save`. This affords an opportunity to modify the dataset using any of the GeoTrellis operations in between the stages.

3.12.1 Sample ETL Application

```scala
import geotrellis.raster.Tile
import geotrellis.spark._
import geotrellis.spark.etl.Etl
import geotrellis.spark.etl.config.EtlConf
import geotrellis.spark.util.SparkUtils
import geotrellis.vector.ProjectedExtent
import org.apache.spark.SparkConf

object GeoTrellisETL {
  type I = ProjectedExtent // or TemporalProjectedExtent for temporal ingest
  type K = SpatialKey // or SpaceTimeKey for temporal ingest
  type V = Tile // or MultibandTile to ingest multiband tile
  def main(args: Array[String]): Unit = {
    implicit val sc = SparkUtils.createSparkContext("GeoTrellis ETL", new
→SparkConf(true))
    try {
      EtlConf(args) foreach { conf =>
        /* parse command line arguments */
        val etl = Etl(conf)
        /* load source tiles using input module specified */
        val sourceTiles = etl.load[I, V]
        /* perform the reprojection and mosaicing step to fit tiles to LayoutScheme
→specified */
        val (zoom, tiled) = etl.tile[I, V, K](sourceTiles)
        /* save and optionally pyramid the mosaiced layer */
        etl.save[K, V](LayerId(etl.input.name, zoom), tiled)
      }
    } finally {
      sc.stop()
    }
  }
}
```

Above is just `Etl.ingest` function implementation, so it is possible to rewrite same functionality:

```scala
import geotrellis.spark._
import geotrellis.raster.Tile
import geotrellis.spark.util.SparkUtils
import geotrellis.vector.ProjectedExtent
import org.apache.spark.SparkConf

object SinglebandIngest {
  def main(args: Array[String]): Unit = {
    implicit val sc = SparkUtils.createSparkContext("GeoTrellis ETL SinglebandIngest",
→ new SparkConf(true))
    try {
```

(continues on next page)

```
      Etl.ingest[ProjectedExtent, SpatialKey, Tile](args)
    } finally {
      sc.stop()
    }
  }
}
```

`Etl.ingest` function can be used with following types variations:

- `Etl.ingest[ProjectedExtent, SpatialKey, Tile]`

- `Etl.ingest[ProjectedExtent, SpatialKey, MultibandTile]`

- `Etl.ingest[TemporalProjectedExtent, SpaceTimeKey, Tile]`

- `Etl.ingest[TemporalProjectedExtent, SpaceTimeKey, MultibandTile]`

For temporal ingest `TemporalProjectedExtent` and `SpaceTimeKey` should be used, for spatial ingest `ProjectedExtent` and `SpatialKey`.

3.12.2 User-defined ETL Configs

The above sample application can be placed in a new SBT project that has a dependency on `"org.locationtech.geotrellis" %% "geotrellis-spark-etl" % s"$VERSION"` in addition to dependency on `spark-core`. and built into an assembly with `sbt-assembly` plugin. You should be careful to include a `assemblyMergeStrategy` for sbt assembly plugin as it is provided in spark-etl build file.

At this point you would create a seperate `App` object for each one of your ETL configs.

3.12.3 Built-in ETL Configs

For convinence and as an example the `spark-etl` project provides two `App` objects that perform vanilla ETL:

- `geotrellis.spark.etl.SinglebandIngest`

- `geotrellis.spark.etl.MultibandIngest`

You may use them by building an assembly jar of `spark-etl` project as follows:

```
cd geotrellis
./sbt
sbt> project spark-etl
sbt> assembly
```

The assembly jar will be placed in `geotrellis/spark-etl/target/scala-2.11` directory.

3.12.4 Running the Spark Job

For maximum flexibility it is desirable to run spark jobs with `spark-submit`. In order to achieve this `spark-core` dependency must be listed as `provided` and `sbt-assembly` plugin used to create the fat jar as described above. Once the assembly jar is read outputs and inputs can be setup through command line arguments like so:

```
#!/bin/sh
export JAR="geotrellis-etl-assembly-1.0.0-SNAPSHOT.jar"
```

```
spark-submit \
--class geotrellis.spark.etl.SinglebandIngest \
--master local[*] \
--driver-memory 2G \
$JAR \
--backend-profiles "file://backend-profiles.json" \
--input "file://input.json" \
--output "file://output.json"
```

Note that the arguments before the `$JAR` configure `SparkContext` and arguments after configure GeoTrellis ETL inputs and outputs.

Command Line Arguments

Option	Description
backend-profiles	Path to a json file (local fs / hdfs) with credentials for ingest datasets (required field)
input	Path to a json file (local fs / hdfs) with datasets to ingest, with optional credentials
output	Path to a json file (local fs / hdfs) with output backend params to ingest, with optional credentials

3.12.5 Backend Profiles JSON

```
{
  "backend-profiles": [{
    "name": "accumulo-name",
    "type": "accumulo",
    "zookeepers": "zookeepers",
    "instance": "instance",
    "user": "user",
    "password": "password"
  },
  {
    "name": "cassandra-name",
    "type": "cassandra",
    "allowRemoteDCsForLocalConsistencyLevel": false,
    "localDc": "datacenter1",
    "usedHostsPerRemoteDc": 0,
    "hosts": "hosts",
    "replicationStrategy": "SimpleStrategy",
    "replicationFactor": 1,
    "user": "user",
    "password": "password"
  }]
}
```

Sets of *named* profiles for each backend.

3.12.6 Output JSON

```
{
   "backend":{
      "type":"accumulo",
      "path":"output",
      "profile":"accumulo-name"
   },
   "breaks":"0:ffffe5ff;0.1:f7fcb9ff;0.2:d9f0a3ff;0.3:addd8eff;0.4:78c679ff;0.
→5:41ab5dff;0.6:238443ff;0.7:006837ff;1:004529ff",
   "reprojectMethod":"buffered",
   "cellSize":{
      "width":256.0,
      "height":256.0
   },
   "encoding":"geotiff",
   "tileSize":256,
   "layoutExtent":{
      "xmin":1.0,
      "ymin":2.0,
      "xmax":3.0,
      "ymax":4.0
   },
   "tileLayout":{
      "layoutCols": 360,
      "layoutRows": 180,
      "tileCols":   240,
      "tileRows":   240
   },
   "resolutionThreshold":0.1,
   "pyramid":true,
   "resampleMethod":"nearest-neighbor",
   "keyIndexMethod":{
      "type":"zorder"
   },
   "layoutScheme":"zoomed",
   "cellType":"int8",
   "crs":"EPSG:3857"
}
```

Key	Value
backend	Backend description is presented below
breaks	Breaks string for `render` output (optional field)
partitions	Partitions number during pyramid build
reprojectMethod	`buffered`, `per-tile`
cellSize	Cell size
encoding	`png`, `geotiff` for `render` output
tileSize	Tile size (optional field)If not set, the default size of output tiles is 256x256
layoutExtent	Layout extent (optional field)
tileLayout	Tile layout to specify layout grid (optional field)
resolutionThreshold	Resolution for user defined Layout Scheme (optional field)
pyramid	`true`, `false` - ingest with or without building a pyramid
resampleMethod	`nearest-neighbo r`, `bilinear`, `cubic-convoluti on`, `cubic-spline`, `lanczos`
keyIndexMethod	`zorder`, `row-major`, `hilbert`
layoutScheme	`tms`, `floating` (optional field)
cellType	`int8`, `int16`, etc... (optional field)
crs	Destination crs name (example: EPSG:3857) (optional field)

Backend Keyword

Key	Value
type	Input backend type (file / hadoop / s3 / accumulo / cassandra)
path	Input path (local path / hdfs), or s3:// url
profile	Profile name to use for input

Supported Layout Schemes

Layout Scheme	Options
zoomed	Zoomed layout scheme
floating	Floating layout scheme in a native projection

KeyIndex Methods

Key	Options
type	`zorder`, `row-major`, `hilbert`
temporalResolution	Temporal resolution for temporal indexing (optional field)
timeTag	Time tag name for input geotiff tiles (optional field)
timeFormat	Time format to parse time stored in time tag geotiff tag (optional field)

3.12.7 Input JSON

```
[{
  "format": "geotiff",
```

```
  "name": "test",
  "cache": "NONE",
  "noData": 0.0,
  "clip": {
    "xmin":1.0,
    "ymin":2.0,
    "xmax":3.0,
    "ymax":4.0
  },
  "backend": {
    "type": "hadoop",
    "path": "input"
  }
}]
```

Key	Value
format	Format of the tile files to be read (ex: geotiff)
name	Input dataset name
cache	Spark RDD cache strategy
noData	NoData value
clip	Extent in target CRS to clip the input source
crs	Destination crs name (example: EPSG:3857) (optional field)
maxTleSize	Inputs will be broken up into smaller tiles of the given size (optional field)(example: 256 returns 256x256 tiles)
numParti-tions	How many partitions Spark should make when repartitioning (optional field)

Supported Formats

Format	Options
geotiff	Spatial ingest
temporal-geotiff	Temporal ingest

Supported Inputs

Input	Options
hadoop	path (local path / hdfs)
s3	s3:// url

Supported Outputs

Output	Options
hadoop	Path
accumulo	Table name
cassandra	Table name with keysapce (keyspace.tablename)
s3	s3:// url
render	Path

Accumulo Output

Accumulo output module has two write strategies:

- `hdfs` strategy uses Accumulo bulk import
- `socket` strategy uses Accumulo `BatchWriter`

When using `hdfs` strategy `ingestPath` argument will be used as the temporary directory where records will be written for use by Accumulo bulk import. This directory should ideally be an HDFS path.

Layout Scheme

GeoTrellis is able to tile layers in either `ZoomedLayoutScheme`, matching TMS pyramid, or `FloatingLayoutScheme`, matching the native resolution of input raster. These alternatives may be selecting by using the `layoutScheme` option.

Note that `ZoomedLayoutScheme` needs to know the world extent, which it gets from the CRS, in order to build the TMS pyramid layout. This will likely cause resampling of input rasters to match the resolution of the TMS levels.

On other hand `FloatingLayoutScheme` will discover the native resolution and extent and partition it by given tile size without resampling.

3.12.8 User-Defined Layout

You may bypass the layout scheme logic by providing `layoutExtent` and either a `tileLayout` or a `cellSize` and `tileSize` to fully define the layout and start the tiling process. The user may optionally specify an output `cellType` as well (default case uses the input `cellType`).

3.12.9 Reprojection

`spark-etl` project supports two methods of reprojection: `buffered` and `per-tile`. They provide a trade-off between accuracy and flexibility.

Buffered reprojection method is able to sample pixels past the tile boundaries by performing a neighborhood join. This method is the default and produces the best results. However it requires that all of the source tiles share the same CRS.

Per tile reproject method can not consider pixels past the individual tile boundaries, even if they exist elsewhere in the dataset. Any pixels past the tile boundaries will be as `NODATA` when interpolating. This restriction allows for source tiles to have a different projections per tile. This is an effective way to unify the projections for instance when projection from multiple UTM projections to WebMercator.

3.12.10 Rendering a Layer

`render` output module is different from other modules in that it does not save a GeoTrellis layer but rather provides a way to render a layer, after tiling and projection, to a set of images. This is useful to either verify the ETL process or render a TMS pyramid.

The `path` module argument is actually a path template, that allows the following substitution:

- `{x}` tile x coordinate
- `{y}` tile y coordinate
- `{z}` layer zoom level
- `{name}` layer name

A sample render output configuration template could be:

```
{
  "path": "s3://tms-bucket/layers/{name}/{z}-{x}-{y}.png",
  "ingestType": {
    "format":"geotiff",
    "output":"render"
  }
}
```

3.12.11 Extension

In order to provide your own input or output modules you must extend InputPlugin and OutputPlugin and register them in the `Etl` constructor via a `TypedModule`.

3.12.12 Examples

Standard ETL assembly provides two classes to ingest objects: class to ingest singleband tiles and class to ingest multiband tiles. The class name to ingest singleband tiles is `geotrellis.spark.etl.SinglebandIngest` and to ingest multiband tiles is `geotrellis.spark.etl.MultibandIngest`.

Every example can be launched using:

```
#!/bin/sh
export JAR="geotrellis-etl-assembly-0.10-SNAPSHOT.jar"

spark-submit \
--class geotrellis.spark.etl.{SinglebandIngest | MultibandIngest} \
--master local[*] \
--driver-memory 2G \
$JAR \
--input "file://input.json" \
--output "file://output.json" \
--backend-profiles "file://backend-profiles.json"
```

Example Backend Profile

`backend-profiles.json`:

```
{
  "backend-profiles":[
    {
      "name":"accumulo-name",
      "type":"accumulo",
      "zookeepers":"zookeepers",
      "instance":"instance",
      "user":"user",
      "password":"password"
    },
    {
      "name":"cassandra-name",
      "type":"cassandra",
      "allowRemoteDCsForLocalConsistencyLevel":false,
      "localDc":"datacenter1",
```

(continues on next page)

```
            "usedHostsPerRemoteDc":0,
            "hosts":"hosts",
            "replicationStrategy":"SimpleStrategy",
            "replicationFactor":1,
            "user":"user",
            "password":"password"
        }
    ]
}
```

Example Output JSON

`output.json`:

```
{
   "backend":{
      "type":"accumulo",
      "path":"output",
      "profile":"accumulo-name"
   },
   "breaks":"0:ffffe5ff;0.1:f7fcb9ff;0.2:d9f0a3ff;0.3:addd8eff;0.4:78c679ff;0.
→5:41ab5dff;0.6:238443ff;0.7:006837ff;1:004529ff",
   "reprojectMethod":"buffered",
   "cellSize":{
      "width":256.0,
      "height":256.0
   },
   "encoding":"geotiff",
   "tileSize":256,
   "layoutExtent":{
      "xmin":1.0,
      "ymin":2.0,
      "xmax":3.0,
      "ymax":4.0
   },
   "resolutionThreshold":0.1,
   "pyramid":true,
   "resampleMethod":"nearest-neighbor",
   "keyIndexMethod":{
      "type":"zorder"
   },
   "layoutScheme":"zoomed",
   "cellType":"int8",
   "crs":"EPSG:3857"
}
```

Example Input JSON

`input.json`:

```
{
  "format": "geotiff",
  "name": "test",
  "cache": "NONE",
```

```
    "noData": 0.0,
    "backend": {
      "type": "hadoop",
      "path": "input"
  }
}
```

Backend JSON examples (local fs)

```
"backend": {
  "type": "hadoop",
  "path": "file:///Data/nlcd/tiles"
}
```

Backend JSON example (hdfs)

```
"backend": {
  "type": "hadoop",
  "path": "hdfs://nlcd/tiles"
}
```

Backend JSON example (s3)

```
"backend": {
  "type": "s3",
  "path": "s3://com.azavea.datahub/catalog"
}
```

Backend JSON example (accumulo)

```
"backend": {
  "type": "accumulo",
  "profile": "accumulo-gis",
  "path": "nlcdtable"
}
```

Backend JSON example (set of PNGs into S3)

```
"backend": {
  "type": "render",
  "path": "s3://tms-bucket/layers/{name}/{z}-{x}-{y}.png"
}
```

Backend JSON example (set of PNGs into hdfs or local fs)

```
"backend": {
  "type": "render",
  "path": "hdfs://path/layers/{name}/{z}-{x}-{y}.png"
}
```

3.13 Extending GeoTrellis Types

3.13.1 Custom Keys

Want to jump straight to a code example? See VoxelKey.scala

Keys are used to index (or "give a position to") tiles in a tile layer. Typically these tiles are arranged in some conceptual grid, for instance in a two-dimensional matrix via a SpatialKey. There is also a SpaceTimeKey, which arranges tiles in a cube of two spatial dimensions and one time dimension.

In this way, keys define how a tile layer is shaped. Here, we provide an example of how to define a new key type, should you want a custom one for your application.

The VoxelKey type

A voxel is the 3D analogue to a 2D pixel. By defining a new `VoxelKey` type, we can create grids of tiles that have a 3D spatial relationship. The class definition itself is simple:

```
case class VoxelKey(x: Int, y: Int, z: Int)
```

Key usage in many GeoTrellis operations is done generically with a `K` type parameter, for instance in the `S3LayerReader` class:

```
/* Read a tile layer from S3 via a given `LayerId`. Function signature slightly␣
→simplified. */
S3LayerReader.read[K: Boundable: JsonFormat, V, M]: LayerId => RDD[(K, V)] with␣
→Metadata[M]
```

Where the pattern `[A: Trait1: Trait2: ...]` means that for whichever `A` you end up using, it must have an implicit instance of `Trait1` and `Trait2` (and any others) in scope. Really it's just syntactic sugar for `[A](implicit ev0: Trait1[A], ev1: Trait2[A], ...)`. The `read` method above would be used in real life like:

```
val reader: S3LayerReader = ...

// The type on `rdd` is often left off for brevity.
val rdd: RDD[(SpatialKey, MultibandTile)] with Metadata[LayoutDefinition] =
    reader.read[SpatialKey, MultibandTile, LayoutDefinition]("someLayer")
```

Boundable, `SpatialComponent`, and `JsonFormat` are frequent constraints on keys. Let's give those typeclasses some implementations:

```
import geotrellis.spark._
import spray.json._

// A companion object is a good place for typeclass instances.
object VoxelKey {

  // What are the minimum and maximum possible keys in the key space?
  implicit object Boundable extends Boundable[VoxelKey] {
    def minBound(a: VoxelKey, b: VoxelKey) = {
      VoxelKey(math.min(a.x, b.x), math.min(a.y, b.y), math.min(a.z, b.z))
    }

    def maxBound(a: VoxelKey, b: VoxelKey) = {
```

(continues on next page)

```
      VoxelKey(math.max(a.x, b.x), math.max(a.y, b.y), math.max(a.z, b.z))
    }
  }

  /** JSON Conversion */
  implicit object VoxelKeyFormat extends RootJsonFormat[VoxelKey] {
    // See full example for real code.
    def write(k: VoxelKey) = ...

    def read(value: JsValue) = ...
  }

  /** Since [[VoxelKey]] has x and y coordinates, it can take advantage of
    * the [[SpatialComponent]] lens. Lenses are essentially "getters and setters"
    * that can be used in highly generic code.
    */
  implicit val spatialComponent = {
    Component[VoxelKey, SpatialKey](
      /* "get" a SpatialKey from VoxelKey */
      k => SpatialKey(k.x, k.y),
      /* "set" (x,y) spatial elements of a VoxelKey */
      (k, sk) => VoxelKey(sk.col, sk.row, k.z)
    )
  }
}
```

With these, `VoxelKey` is now (almost) usable as a key type in GeoTrellis.

A Z-Curve SFC for VoxelKey

Many operations require a KeyIndex as well, which are usually implemented with some hardcoded key type.
`VoxelKey` would need one as well, which we will back by a Z-Curve for this example:

```
/** A [[KeyIndex]] based on [[VoxelKey]]. */
class ZVoxelKeyIndex(val keyBounds: KeyBounds[VoxelKey]) extends KeyIndex[VoxelKey] {
  /* ''Z3'' here is a convenient shorthand for any 3-dimensional key. */
  private def toZ(k: VoxelKey): Z3 = Z3(k.x, k.y, k.z)

  def toIndex(k: VoxelKey): Long = toZ(k).z

  def indexRanges(keyRange: (VoxelKey, VoxelKey)): Seq[(Long, Long)] =
    Z3.zranges(toZ(keyRange._1), toZ(keyRange._2))
}
```

And with a `KeyIndex` written, it will of course need its own `JsonFormat`, which demands some additional glue to
make fully functional. For more details, see ShardingKeyIndex.scala.

We now have a new fully functional key type which defines a tile cube of three spatial dimensions. Of course, there
is nothing stopping you from defining a key in any way you like: it could have three spatial and one time dimension
(`EinsteinKey`?) or even ten spatial dimensions (`StringTheoryKey`?). Happy tiling.

3.13.2 Custom KeyIndexes

Want to dive right into code? See: ShardingKeyIndex.scala

The KeyIndex trait

The `KeyIndex` trait is high-level representation of Space Filling Curves, and for us it is critical to Tile layer input/output. As of GeoTrellis `1.0.0`, its subclasses are:

- `ZSpatialKeyIndex`
- `ZSpaceTimeKeyIndex`
- `HilbertSpatialKeyIndex`
- `HilbertSpaceTimeKeyIndex`
- `RowMajorSpatialKeyIndex`

While the subclass constructors can be used directly when creating an index, we always reference them generically elsewhere as `KeyIndex`. For instance, when we write an `RDD`, we need to supply a generic `KeyIndex`:

```
S3LayerWriter.write[K, V, M]: (LayerId, RDD[(K, V)] with Metadata[M], KeyIndex[K]) =>␣
→Unit
```

but when we read or update, we don't:

```
S3LayerReader.read[K, V, M]: LayerId => RDD[(K, V)] with Metadata[M]

S3LayerWriter.update[K, V, M]: (LayerId, RDD[(K, V)] with Metadata[M]) => Unit
```

Luckily for the end user of GeoTrellis, this means they don't need to keep track of which `KeyIndex` subclass they used when they initially wrote the layer. The `KeyIndex` itself is stored a JSON, and critically, **(de)serialized generically**. Meaning:

```
/* Instantiate as the parent trait */
val index0: KeyIndex[SpatialKey] = new ZSpatialKeyIndex(KeyBounds(
    SpatialKey(0, 0),
    SpatialKey(9, 9)
))

/* Serializes at the trait level, not the subclass */
val json: JsValue = index0.toJson

/* Deserialize generically */
val index1: KeyIndex[SpatialKey] = json.convertTo[KeyIndex[SpatialKey]]

index0 == index1  // true
```

Extending KeyIndex

To achieve the above, GeoTrellis has a central `JsonFormat` registry for the `KeyIndex` subclasses. When creating a new KeyIndex type, we need to:

0. Write the index type itself, extending `KeyIndex`
1. Write a standard `spray.json.JsonFormat` for it
2. Write a *Registrator* class that registers our new Format with GeoTrellis

To extend `KeyIndex`, we need to supply implementations for three methods:

```
/* Most often passed in as an argument ''val'' */
def keyBounds: KeyBounds[K] = ???

/* The 1-dimensional index in the SFC of a given key */
def toIndex(key: K): Long = ???

/* Ranges of results of `toIndex` */
def indexRanges(keyRange: (K, K)): Seq[(Long, Long)] = ???
```

where K will typically be hard-coded as either SpatialKey or SpaceTimeKey, unless you've defined some custom key type for your application. K is generic in our example ShardingKeyIndex, since it holds an inner KeyIndex:

```
class ShardingKeyIndex[K](val inner: KeyIndex[K], val shardCount: Int) extends␣
→KeyIndex[K] { ... }
```

Writing and Registering a JsonFormat

Supplying a JsonFormat for our new type is fairly ordinary, with a few caveats:

```
import spray.json._

class ShardingKeyIndexFormat[K: JsonFormat: ClassTag] extends␣
→RootJsonFormat[ShardingKeyIndex[K]] {
  /* This is the foundation of the reflection-based deserialization process */
  val TYPE_NAME = "sharding"

  /* Your `write` function must follow this format, with two fields
   * `type` and `properties`. The `properties` JsObject can contain anything.
   */
  def write(index: ShardingKeyIndex[K]): JsValue = {
    JsObject(
      "type" -> JsString(TYPE_NAME),
      "properties" -> JsObject(
        "inner" -> index.inner.toJson,
        "shardCount" -> JsNumber(index.shardCount)
      )
    )
  }

  /* You should check the deserialized `typeName` matches the original */
  def read(value: JsValue): ShardingKeyIndex[K] = {
    value.asJsObject.getFields("type", "properties") match {
      case Seq(JsString(typeName), properties) if typeName == TYPE_NAME => {
        properties.asJsObject.getFields("inner", "shardCount") match {
          case Seq(inner, JsNumber(shardCount)) =>
            new ShardingKeyIndex(inner.convertTo[KeyIndex[K]], shardCount.toInt)
          case _ => throw new DeserializationException("Couldn't deserialize␣
→ShardingKeyIndex.")
        }
      }
      case _ => throw new DeserializationException("Wrong KeyIndex type:␣
→ShardingKeyIndex expected.")
    }
  }
}
```

Note: Our `Format` here only has a `K` constraint because of our inner `KeyIndex`. Yours likely won't.

Now for the final piece of the puzzle, the format Registrator. With the above in place, it's quite simple:

```
import geotrellis.spark.io.json._

/* This class must have no arguments! */
class ShardingKeyIndexRegistrator extends KeyIndexRegistrator {
  def register(keyIndexRegistry: KeyIndexRegistry): Unit = {
    implicit val spaceFormat = new ShardingKeyIndexFormat[SpatialKey]()
    implicit val timeFormat = new ShardingKeyIndexFormat[SpaceTimeKey]()

    keyIndexRegistry.register(
      KeyIndexFormatEntry[SpatialKey, ShardingKeyIndex[SpatialKey]](spaceFormat.TYPE_
↪NAME)
    )
    keyIndexRegistry.register(
      KeyIndexFormatEntry[SpaceTimeKey, ShardingKeyIndex[SpaceTimeKey]](timeFormat.
↪TYPE_NAME)
    )
  }
}
```

At its simplest for an Index with a hard-coded key type, a registrator could look like:

```
class MyKeyIndexRegistrator extends KeyIndexRegistrator {
  def register(keyIndexRegistry: KeyIndexRegistry): Unit = {
    implicit val format = new MyKeyIndexFormat()

    keyIndexRegistry.register(
      KeyIndexFormatEntry[SpatialKey, MyKeyIndex](format.TYPE_NAME)
    )
  }
}
```

Plugging a Registrator in

GeoTrellis needs to know about your new Registrator. This is done through an `application.conf` in `your-project/src/main/resources/`:

```
// in `application.conf`
geotrellis.spark.io.index.registrator="geotrellis.doc.examples.spark.
↪ShardingKeyIndexRegistrator"
```

GeoTrellis will automatically detect the presence of this file, and use your Registrator.

Testing

Writing unit tests for your new Format is the best way to ensure you've set up everything correctly. Tests for `ShardingKeyIndex` can be found in `doc-examples/src/test/scala/geotrellis/doc/examples/spark/ShardingKeyIndexSpec.scala`, and can be ran in sbt with:

```
geotrellis > project doc-examples
doc-examples > testOnly geotrellis.doc.examples.spark.ShardingKeyIndexSpec
```

3.14 GeoTrellis Module Hierarchy

This is a full list of all GeoTrellis modules. While there is some interdependence between them, you can depend on as many (or as few) of them as you want in your `build.sbt`.

3.14.1 geotrellis-accumulo

Allows the use of Apache Accumulo as a Tile layer backend.

Provides: `geotrellis.spark.io.accumulo.*`

- Save and load layers to and from Accumulo. Query large layers efficiently using the layer query API.

3.14.2 geotrellis-cassandra

Allows the use of Apache Cassandra as a Tile layer backend.

Provides: `geotrellis.spark.io.cassandra.*`

- Save and load layers to and from Cassandra. Query large layers efficiently using the layer query API.

3.14.3 geotrellis-etl

A command-line tool for streamlining the ingest process.

Provides: `geotrellis.spark.etl.*`

- Parse command line options for input and output of ETL (Extract, Transform, and Load) applications
- Utility methods that make ETL applications easier for the user to build.
- Work with input rasters from the local file system, HDFS, or S3
- Reproject input rasters using a per-tile reproject or a seamless reprojection that takes into account neighboring tiles.
- Transform input rasters into layers based on a ZXY layout scheme
- Save layers into Accumulo, S3, HDFS or the local file system.

3.14.4 geotrellis-geomesa

Experimental. GeoTrellis compatibility for the distributed feature store GeoMesa.

Provides: `geotrellis.spark.io.geomesa.*`

- Save and load `RDD`s of features to and from GeoMesa.

3.14.5 geotrellis-hbase

Allows the use of Apache HBase as a Tile layer backend.

Provides: `geotrellis.spark.io.hbase.*`

- Save and load layers to and from HBase. Query large layers efficiently using the layer query API.

3.14.6 geotrellis-proj4

Provides: `geotrellis.proj4.*`, `org.osgeo.proj4.*` (Java)

- Represent a Coordinate Reference System (CRS) based on Ellipsoid, Datum, and Projection.

- Translate CRSs to and from proj4 string representations.

- Lookup CRS's based on EPSG and other codes.

- Transform `(x, y)` coordinates from one CRS to another.

3.14.7 geotrellis-raster

Types and algorithms for Raster processing.

Provides: `geotrellis.raster.*`

- Provides types to represent single- and multi-band rasters, supporting Bit, Byte, UByte, Short, UShort, Int, Float, and Double data, with either a constant NoData value (which improves performance) or a user defined NoData value.

- Treat a tile as a collection of values, by calling "map" and "foreach", along with floating point valued versions of those methods (separated out for performance).

- Combine raster data in generic ways.

- Render rasters via color ramps and color maps to PNG and JPG images.

- Read GeoTiffs with DEFLATE, LZW, and PackBits compression, including horizontal and floating point prediction for LZW and DEFLATE.

- Write GeoTiffs with DEFLATE or no compression.

- Reproject rasters from one CRS to another.

- Resample of raster data.

- Mask and Crop rasters.

- Split rasters into smaller tiles, and stitch tiles into larger rasters.

- Derive histograms from rasters in order to represent the distribution of values and create quantile breaks.

- Local Map Algebra operations: Abs, Acos, Add, And, Asin, Atan, Atan2, Ceil, Cos, Cosh, Defined, Divide, Equal, Floor, Greater, GreaterOrEqual, InverseMask, Less, LessOrEqual, Log, Majority, Mask, Max, MaxN, Mean, Min, MinN, Minority, Multiply, Negate, Not, Or, Pow, Round, Sin, Sinh, Sqrt, Subtract, Tan, Tanh, Undefined, Unequal, Variance, Variety, Xor, If

- Focal Map Algebra operations: Hillshade, Aspect, Slope, Convolve, Conway's Game of Life, Max, Mean, Median, Mode, Min, MoransI, StandardDeviation, Sum

- Zonal Map Algebra operations: ZonalHistogram, ZonalPercentage

- Operations that summarize raster data intersecting polygons: Min, Mean, Max, Sum.

- Cost distance operation based on a set of starting points and a friction raster.

- Hydrology operations: Accumulation, Fill, and FlowDirection.

- Rasterization of geometries and the ability to iterate over cell values covered by geometries.

- Vectorization of raster data.

- Kriging Interpolation of point data into rasters.

- Viewshed operation.

- RegionGroup operation.

3.14.8 geotrellis-raster-testkit

Integration tests for `geotrellis-raster`.

- Build test raster data.

- Assert raster data matches Array data or other rasters in scalatest.

3.14.9 geotrellis-s3

Allows the use of Amazon S3 as a Tile layer backend.

Provides: `geotrellis.spark.io.s3.*`

- Save/load raster layers to/from the local filesystem or HDFS using Spark's IO API.

- Save spatially keyed RDDs of byte arrays to z/x/y files in S3. Useful for saving PNGs off for use as map layers in web maps.

3.14.10 geotrellis-shapefile

Provides: `geotrellis.shapefile.*`

- Read geometry and feature data from shapefiles into GeoTrellis types using GeoTools.

3.14.11 geotrellis-slick

Adds PostGis support for Slick use with GeoTrellis.

Provides: `geotrellis.slick.*`

- Save and load geometry and feature data to and from PostGIS using the slick scala database library.

- Perform PostGIS `ST_` operations in PostGIS through scala.

3.14.12 geotrellis-spark

Tile layer algorithms powered by Apache Spark.

Provides: `geotrellis.spark.*`

- Generic way to represent key value RDDs as layers, where the key represents a coordinate in space based on some uniform grid layout, optionally with a temporal component.

- Represent spatial or spatiotemporal raster data as an RDD of raster tiles.

- Generic architecture for saving/loading layers RDD data and metadata to/from various backends, using Spark's IO API with Space Filling Curve indexing to optimize storage retrieval (support for Hilbert curve and Z order curve SFCs). HDFS and local file system are supported backends by default, S3 and Accumulo are supported backends by the `geotrellis-s3` and `geotrellis-accumulo` projects, respectively.

- Query architecture that allows for simple querying of layer data by spatial or spatiotemporal bounds.

- Perform map algebra operations on layers of raster data, including all supported Map Algebra operations mentioned in the geotrellis-raster feature list.

- Perform seamless reprojection on raster layers, using neighboring tile information in the reprojection to avoid unwanted NoData cells.

- Pyramid up layers through zoom levels using various resampling methods.

- Types to reason about tiled raster layouts in various CRS's and schemes.

- Perform operations on raster RDD layers: crop, filter, join, mask, merge, partition, pyramid, render, resample, split, stitch, and tile.

- Polygonal summary over raster layers: Min, Mean, Max, Sum.

- Save spatially keyed RDDs of byte arrays to z/x/y files into HDFS or the local file system. Useful for saving PNGs off for use as map layers in web maps or for accessing GeoTiffs through z/x/y tile coordinates.

- Utilities around creating spark contexts for applications using GeoTrellis, including a Kryo registrator that registers most types.

3.14.13 geotrellis-spark-testkit

Integration tests for `geotrellis-spark`.

- Utility code to create test RDDs of raster data.

- Matching methods to test equality of RDDs of raster data in scalatest unit tests.

3.14.14 geotrellis-geotools

Provides: `geotrellis.geotools.*`

3.14.15 geotrellis-vector

Types and algorithms for processing Vector data.

Provides: `geotrellis.vector.*`

- Provides a scala idiomatic wrapper around JTS types: Point, Line (LineString in JTS), Polygon, MultiPoint, MultiLine (MultiLineString in JTS), MultiPolygon, GeometryCollection

- Methods for geometric operations supported in JTS, with results that provide a type-safe way to match over possible results of geometries.

- Provides a Feature type that is the composition of a geometry and a generic data type.

- Read and write geometries and features to and from GeoJSON.

- Read and write geometries to and from WKT and WKB.

- Reproject geometries between two CRSs.

- Geometric operations: Convex Hull, Densification, Simplification

- Perform Kriging interpolation on point values.

- Perform affine transformations of geometries

3.14.16 geotrellis-vector-testkit

Integration tests for `geotrellis-vector`.

- GeometryBuilder for building test geometries
- GeometryMatcher for scalatest unit tests, which aides in testing equality in geometries with an optional threshold.

3.14.17 geotrellis-vectortile

Experimental. A full Mapbox VectorTile codec.

Provides: `geotrellis.vectortile.*`

- Lazy decoding
- Read/write `VectorTile` tile layers from any tile backend

3.14.18 geotrellis-util

Plumbing for other GeoTrellis modules.

Provides: `geotrellis.util.*`

- Data structures missing from Scala
- Lenses
- Constants

3.14.19 geotrellis-geowave

Experimental. GeoTrellis compatibility for the distributed feature store GeoWave.

Provides: `geotrellis.spark.io.geowave.*`

- Save and load `RDD`s of features to and from GeoWave.

3.15 Tile Layer Backends

GeoTrellis isn't picky about how you store your data. This guide describes the various tile layer backends we support, how to use them, and why you might choose one over the other.

3.15.1 To Be or not to Be a Backend

The Scala classes that underpin each backend all inherit from the same group of traits, meaning they agree on behaviour:

- `AttributeStore` - save and access layer attributes (metadata, etc.)
- `LayerReader` - read `RDD[(K, V)] with Metadata[M]`
- `LayerWriter` - write `RDD[(K, V)] with Metadata[M]`
- `LayerReindexer`

- `LayerCopier`
- `LayerDeleter`
- `LayerMover`
- `LayerManager`

The top three are used most often, with the `AttributeStore` being a key piece to every other class.
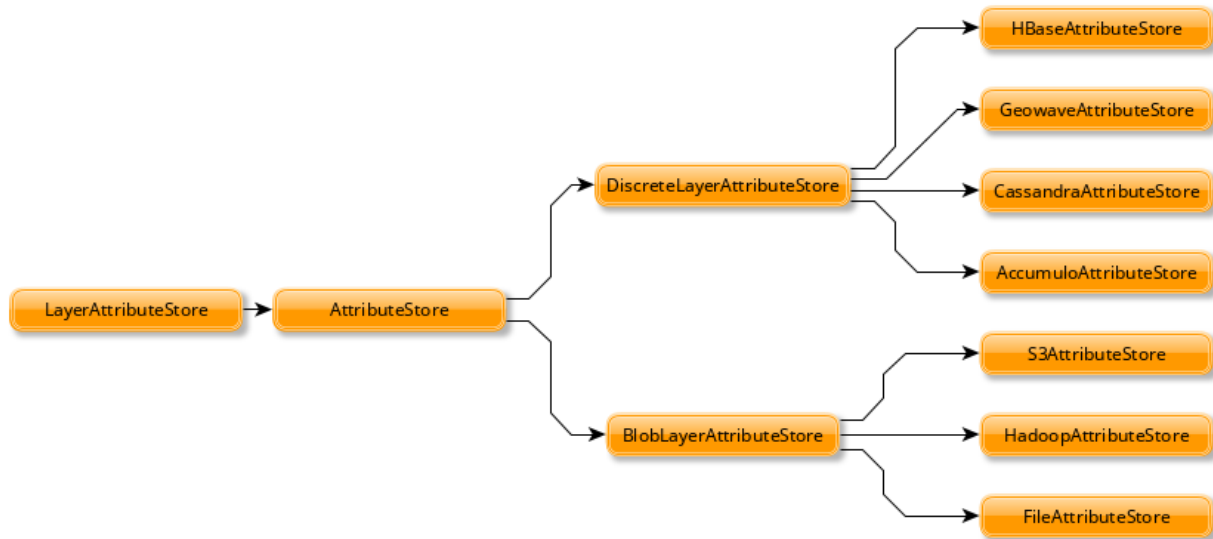
By default, the stored attributes are:

- Metadata
- Header (different per backend)
- Key Index
- Schema

`BlobLayerAttributeStore` stores all attributes in a single JSON object. `DiscreteLayerAttributeStore` stores each attribute as a seperate object (say, a column in the case of databases).

3.15.2 Backend URI

Instances of `AttributeStore`, `LayerWriter`, and `` `LayerReader `` can be created from a URI.

```
val uri = new URI(""s3://bucket/catalog")
val store = AttributeStore(uri)
val reader = LayerReader(uri)
val writer = LayerWriter(uri)
val values = ValueReader(uri)
```

The backends are identified by the scheme portion of the URI. The address and path is used to identify the resource and query parameters are used to configure it.

Backend	URI
Hadoop	`hdfs://path/to/catalog`
S3.	`s3://bucket/catalog-key`
File	`file://tmp/local-catalog`
Accumulo	`accumulo://[user[:password]@]zookeeper/instance-name[?` `attributes=table1[&layers=table2]]`
Cassandra	`cassandra://[user:password@]zookeeper[:port][/keyspace][?` `attributes=table1[&layers=table2]]`
HBase	`hbase://zookeeper[:port][?master=host][?attributes=table1[&layers=table2]]`

Backends that use a database need two extra settings: `attributes` specifies which table is used for storing layer attributes. It is optional with a default value. `layers` specifies which table to write values to. It is required when creating a `LayerWriter`.

Note: When building an assembly make sure to define a merge strategy for concatenating service files

---

```
assemblyMergeStrategy in assembly := {
   case s if x.startsWith("META-INF/services") => MergeStrategy.concat
   case "reference.conf" => MergeStrategy.concat
   case "META-INF/MANIFEST.MF" => MergeStrategy.discard
   case "META-INF/ECLIPSEF.RSA" => MergeStrategy.discard
   case "META-INF/ECLIPSEF.SF" => MergeStrategy.discard
   case _ => MergeStrategy.first
}
```

3.15.3 File System

Choose your file system if: you want to perform tests, data ingests, or data processing locally on your computer.

This is the simplest backend, only requiring a path to read and write tiles to:

```
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.io.file._

val catalogPath: String = ...

val store: AttributeStore = FileAttributeStore(catalogPath)

val reader = FileLayerReader(store)
val writer = FileLayerWriter(store)
```

PROs:

- Simple

- Built in: available from the `geotrellis-spark` package

CONs:

- Not suitable for use in Production.

3.15.4 HDFS

Choose HDFS if: you want a simple setup and fast write speed or the ability to write and read from Microsoft Azure's Blob storage.

The Hadoop Distributed File System (HDFS), as the name implies, presents a view to the programmer as if their entire cluster were one giant file system.

```
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.io.hadoop._

val rootPath: Path = ...
val config: Configuration = ...

/* The `config` argument is optional */
val store: AttributeStore = HadoopAttributeStore(rootPath, config)

val reader = HadoopLayerReader(store)
val writer = HadoopLayerWriter(rootPath, store)
```

Additionally, HDFS provides the ability to connect to Azure Blob storage through the use of the Hadoop Azure connector. Similar to S3 this allows you to leverage cloud storage as a backend. This capability requires addition configuration to use that is outlined in the Hadoop Azure Support documentation. Notably you will need to:

- Provide Azure credentials in Hadoop's *core-site.xml*.

- [Optional] Configure the Hadoop's *fs.defaultFS* to use your *wasb* or *wasbs* url. This allows for bare paths to the configured location to be used.

- Provide additional dependencies, including the *hadoop-azure* module and the Azure Storage SDK for Java.

PROs:

- Built in: available from the `geotrellis-spark` package

- Simple production environment

- Fast writes

- Can also be used locally (good for testing)

- Supports `hadoop >= 2.6`

- Supports Microsoft Azure's Blob Storage

CONs

- Slower read speed than alternatives

- Inefficient layer update functionality

3.15.5 S3

Choose S3 if: you have large amounts of data to store, can pay for external storage, and want to access the data from anywhere.

Amazon S3. Provided you can pay for their service, S3 is the simplest backend to put into production. There are no external processes, and it allows your data and application to live on different clusters. Data replication is handled automatically. If your application runs on AWS, it can also access S3 data for free.

The GeoTrellis team recommends the S3 backend as the first consideration when putting a system into production.

```
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.io.s3._  /* from the `geotrellis-s3 package */

val bucket: String = ...
val prefix: String = ...

implicit val sc: SparkContext = ...

val store: AttributeStore = S3AttributeStore(bucket, prefix)

val reader = S3LayerReader(store) /* Needs the implicit SparkContext */
val writer = S3LayerWriter(store)
```

PROs:

- Your application can access your data from anywhere in the world

- Simple production environment; no external processes

- Fast enough to back a real-time tile server

CONs:

- May be cost-prohibitive, depending on your usage

3.15.6 Accumulo

Choose Accumulo if: you want fast reads and are willing to put in the setup effort.

Apache Accumulo. This is a popular choice in the GIS world, and is the most battle-tested backend within GeoTrellis. It requires more mental and physical overhead to put into production, but is quite performant and provides unique features. To work with GeoTrellis, it requires an external Accumulo process to be running.

```
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.io.accumulo._  /* from the `geotrellis-accumulo` package */
import org.apache.accumulo.core.client.security.tokens._

val instanceName: String = ...
val zookeeper: String = ...
val user: String = ...
val token: AuthenticationToken = new PasswordToken(pwd)

val dataTable: String = ...

val strat: AccumuloWriteStrategy = HdfsWriteStrategy()  /* Or SocketWriteStrategy */
val opts: AccumuloLayerWriter.Options = AccumuloLayerWriter.Options(strat)

implicit val sc: SparkContext = ...
implicit val instance = AccumuloInstance(
  instanceName,
  zookeeper,
  user,
  token
)

val store: AttributeStore = AccumuloAttributeStore(instance)
```

(continues on next page)

```
val reader = AccumuloLayerReader(instance)
val writer = AccumuloLayerWriter(instance, dataTable, opts)
```

PROs:

- Fast reads

- Popular in GIS

- Fine-grained field access authentication support

- Supports 1 Exobyte cell size

- Supports `accumulo >= 1.7`

CONs:

- Complex production environment

- Requires external processes

3.15.7 Cassandra

Choose Cassandra if: you want a simple(r) production environment, or already have a Cassandra cluster.

Apache Cassandra. Cassandra is a fast, column-based NoSQL database. It is likely the most performant of our backends, although this has yet to be confirmed. To work with GeoTrellis, it requires an external Cassandra process to be running.

Note: As of 2016 October 26, our Cassandra support is still relatively new.

```
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.io.cassandra._ /* from the `geotrellis-cassandra package */

val instance: CassandraInstance = ...
val keyspace: String = ...
val attrTable: String = ...
val dataTable: String = ...

implicit val sc: SparkContext = ...

val store: AttributeStore = CassandraAttributeStore(instance, keyspace, attrTable)

val reader = CassandraLayerReader(store) /* Needs the implicit SparkContext */
val writer = CassandraLayerWriter(store, instance, keyspace, dataTable)
```

PROs:

- Simple(r) production environment; no HDFS, zookeepers, etc.

- Popular as a NoSQL database

- Supports `cassandra >= 3`

CONs:

- Requires external processes

3.15.8 HBase

Choose HBase if: you have a pre-existing HBase cluster.

Apache HBase, a "Big Table" implementation based on HDFS. To work with GeoTrellis, HBase requires external processes much like Accumulo.

Note: As of 2016 October 26, our HBase support is still relatively new.

```
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.io.hbase._ /* from the `geotrellis-hbase package */

val instance: HBaseInstance = ...
val attrTable: String = ...
val dataTable: String = ...

implicit val sc: SparkContext = ...

val store: AttributeStore = HBaseAttributeStore(instance, attrTable)

val reader = HBaseLayerReader(store) /* Needs the implicit SparkContext */
val writer = HBaseLayerWriter(store, dataTable)
```

PROs:

- More user friendly than Accumulo
- Supports `hbase >= 1.2`

CONs:

- Slower than Cassandra
- Requires external processes

3.16 Vector Data Backends

GeoTrellis supports two well-known distributed vector-feature stores: GeoMesa and GeoWave. A question that often arises in the vector processing world is: "Which should I use?" At first glance, it can be hard to tell the difference, apart from "one is Java and the other is Scala". The real answer is, of course, "it depends".

In the fall of 2016, our team was tasked with an official comparison of the two. It was our goal to increase awareness of their respective strengths and weaknesses, so that both teams can focus on their strengths during development, and the public can make an easier choice. We analysed a number of angles, including:

- Feature set
- Performance
- Ease of use
- Project maturity

The full report should be made public in Q1/Q2 of 2017.

While developing applications directly with these projects is quite a different experience, in terms of our GeoTrellis interfaces for each project (as a vector data backend), they support essentially the same feature set (GeoWave optionally supports reading/writing Raster layers).

Keep in mind that as of 2016 October 25, both of these GeoTrellis modules are still experimental.

3.16.1 GeoMesa

```
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.io.geomesa._

val instance: GeoMesaInstance(
  tableName = ...,
  instanceName = ...,
  zookeepers = ...,
  users = ...,
  password = ...,
  useMock = ...
)

val reader = new GeoMesaFeatureReader(instance)
val writer = new GeoMesaFeatureWriter(instance)

val id: LayerId = ...
val query: Query = ... /* GeoMesa query type */

val spatialFeatureType: SimpleFeatureType = ... /* from geomesa - see their docs */

/* for some generic D, following GeoTrellis `Feature[G, D]` */
val res: RDD[SimpleFeature] = reader.read[Point, D](
  id,
  spatialFeatureType,
  query
)
```

3.16.2 GeoWave

```scala
import geotrellis.spark._
import geotrellis.spark.io._
import geotrellis.spark.io.geowave._

val res: RDD[Feature[G, Map[String, Object]]] = GeoWaveFeatureRDDReader.read(
  zookeepers = ...,
  accumuloInstanceName = ...,
  accumuloInstanceUser = ...,
  accumuloInstancePass = ...,
  gwNamespace = ...,
  simpleFeatureType = ... /* from geowave */
)
```

3.17 Frequently Asked Questions

3.17.1 How do I install GeoTrellis?

Sadly, you can't. GeoTrellis is a developer toolkit/library/framework used to develop applications in Scala against geospatial data large and small. To use it, you need it listed as a dependency in your project config, like any other library. See our Setup Tutorial on how to do this.

3.17.2 How do I convert a `Tile`'s `CellType`?

Question: Let's say I have a tile with incorrect `CellType` information or that, for whatever reason, I need to change it. How can I convert a `Tile`'s `CellType`? Which methods can I use?

Answer: There are two distinct flavors of 'conversion' which GeoTrellis supports for moving between `CellTypes`: `convert` and `interpretAs`. In what follows we will try to limit any confusion about just what differentiates these two methods and describe which should be used under what circumstances.

Elsewhere, we've said that the `CellType` is just a piece of metadata carried around alongside a `Tile` which helps GeoTrellis to keep track of how that `Tile`'s array should be interacted with. The distinction between `interpretAs` and `convert` relates to how smart GeoTrellis should be while swapping out one `CellType` for another.

Broadly, `convert` assumes that your `Tile`'s `CellType` is accurate and that you'd like the semantics of your `Tile` to remain invariant under the conversion in question. For example, imagine that we've got categorical data whose cardinality is equal to the cardinality of `Byte` (254 assuming we reserved a spot for `NoData`). Let's fiat, too, that the `CellType` we're using is `ByteConstantNoData`. What happens if we want to add a 255th category? Unless we abandon `NoData` (usually not the right move), it would seem we're out of options so long as we use `ByteCells`. Instead, we should call `convert` on that tile and tell it that we'd like to transpose all `Byte` values to `Short` values. All of the numbers will remain the same with the exception of any `Byte.MinValue` cells, which will be turned into `Short.MinValue` in accordance with the new `CellType`'s chosen `NoData` value. This frees up quite a bit of extra room for categories and allows us to continue working with our data in nearly the same manner as before conversion.

`interpretAs` is a method that was written to resolve a different problem. If your `Tile` is associated with an incorrect `CellType` (as can often happen when reading GeoTIFFs that lack proper, accurate headers), `interpretAs` provides a means for attaching the correct metadata to your `Tile` *without trusting the pre-interpretation metadata*. The "conversion" carried out through `interpretAs` does *not* try to do anything intelligent. There can be no guarantee that meaning is preserved through reinterpretation - in fact, the primary use case for `interpretAs` is to attach the correct metadata to a `Tile` which is improperly labelled for whatever reason.

An interesting consequence is that you can certainly move between data types (not just policies for handling `NoData`) by way of `interpretAs` but that, because the original metadata is not accurate, the default, naive conversion (`_.toInt`, `_.toFloat`, etc.) must be depended upon.

```scala
/** getRaw is a method that allows us to see the values regardless of
if, semantically, they are properly treated as non-data. We use it here
simply to expose the mechanics of the transformation 'under the hood' */

val myData = Array(42, 2, 3, 4)
val tileBefore = IntArrayTile(myData, 2, 2, IntUserDefinedNoDataValue(42))

/** While the value in (0, 0) is NoData, it is now 1 instead of 42
  *   (which matches our new CellType's expectations)
  */
val converted = tileBefore.convert(IntUserDefinedNoData(1))
assert(converted.getRaw.get(0, 0) != converted.get(0, 0))
```

(continues on next page)

```
/** Here, the first value is still 42. But because the NoData value is
  *  now 1, the first value is no longer treated as NoData
  *  (which matches our new CellType's expectations) */
val interpreted = tileBefore.interpretAs(IntUserDefinedNoData(1))
assert(interpreted.getRaw.get(0, 0) == interpreted.get(0, 0))
```

TL;DR: If your `CellType` is just wrong, reinterpret the meaning of your underlying cells with a call to `interpretAs`. If you trust your `CellType` and wish for its semantics to be preserved through transformation, use `convert`.

3.17.3 How do I import GeoTrellis methods?

Question: In some of the GeoTrellis sample code and certainly in example projects, it looks like some GeoTrellis types have more methods than they really do. If I create an `IntArrayTile`, it doesn't have most of the methods that it should - I can't reproject, resample, or carry out map algebra operations - why is that and how can I fix it?

Answer: Scala is a weird language. It is both object oriented (there's an inheritance tree which binds together the various types of `Tile`) and functional (harder to define, exactly, but there's plenty of sugar for dealing with functions). The phenomenon of apparently missing methods is an upshot of the fact that many of the behaviors bestowed upon GeoTrellis types come from the more functional structure of typeclasses rather than the stricter, more brittle, and more familiar standard inheritance structure.

Roughly, if OO structures of inheritance define what can be done in virtue of *what a thing is*, typeclasses allow us to define an object's behavior in virtue of *what it can do*. Within Scala's type system, this differing expectation can be found between a function which takes a `T` where `T <: Duck` (the `T` that is expected must be a duck or one of its subtypes) and a function which takes `T` where `T: Quacks` (the `T` that is expected must be able to quack, regardless of what it is).

If this sounds a lot like duck-typing, that's because it is. But, whereas method extension through duck-typing in other languages is a somewhat risky affair (runtime errors abound), Scala's type system allows us to be every bit as certain of the behavior in our typeclasses as we would be were the methods defined within the body of some class, itself.

Unlike the rather straightforward means for defining typeclasses which exist in some languages (e.g. Haskell), Scala's typeclasses depend upon implicitly applying pieces of code which happen to be in scope. The details can get confusing and are unnecessary for most work with GeoTrellis. If you're interested in understanding the problem at a deeper level, check out this excellent article.

Because the entire typeclass infrastructure depends upon implicits, all you need to worry about is importing the proper set of classes which define the behavior in question. Let's look to a concrete example. Note the difference in import statements:

This does not compile.

```
import geotrellis.vector._

val feature = Feature[Point, Int](Point(1, 2), 42)
feature.toGeoJson  // not allowed, method extension not in scope
```

This does.

```
import geotrellis.vector._
import geotrellis.vector.io._

val feature = Feature[Point, Int](Point(1, 2), 42)
feature.toGeoJson  // returns geojson, as expected
```

TL;DR: Make sure you're importing the appropriate implicits. They define methods that extend GeoTrellis types.

3.17.4 How do I resolve dependency compatibility issues (Guava, etc.)?

Full possible exception message:

```
Caused by: java.lang.IllegalStateException: Detected Guava issue #1635
which indicates that a version of Guava less than 16.01 is in use.  This
introduces codec resolution issues and potentially other incompatibility
issues in the driver. Please upgrade to Guava 16.01 or later.
```

GeoTrellis depends on a huge number of complex dependencies that may cause dependency hell. One of such dependency is the Guava library. `GeoTrellis ETL` and `GeoTrellis Cassandra` depend on `Guava 16.01`, but Hadoop depends on `Guava 11.0.2` which causes runtime issues due to library incompatibility. When two different versions of the same library are both available in the Spark classpath and in a fat assembly jar, Spark will use library version from its classpath.

There are two possible solutions:

1. To `shade` the conflicting library (example below shades Guava in all GeoTrellis related deps, this idea can be extrapolated on all conflicting libraries):

```
assemblyShadeRules in assembly := {
  val shadePackage = "com.azavea.shaded.demo"
  Seq(
    ShadeRule.rename("com.google.common.**" -> s"$shadePackage.google.common.@1")
      .inLibrary(
        "com.azavea.geotrellis" %% "geotrellis-cassandra" % gtVersion,
        "com.github.fge" % "json-schema-validator" % "2.2.6"
      ).inAll
  )
}
```

2. To use spark.driver.userClassPathFirst. It's an experimental Spark property to force Spark using all deps from the fat assembly jar.

3.18 Example Archive

This is a collection of code snippets that represent common GeoTrellis tasks.

3.18.1 Get a Layer from a GeoTiff on S3

Motivation: You have a potentially large GeoTiff on S3 that you'd like to form a proper keyed GeoTrellis layer out of. While you're at it, you'd like to change the projection of the imagery too.

New as of GeoTrellis 1.1, `S3GeoTiffRDD` provides a powerful set of functions for fetching Tiles from the cloud. Here's an example that encompasses your desired work-flow:

```
def layerFromS3(bucket: String, key: String)(implicit sc: SparkContext): (Int,
→TileLayerRDD[SpatialKey]) = {
 /* Extract Tiles efficiently from a remote GeoTiff. */
 val ungridded: RDD[(ProjectedExtent, Tile)] =
   S3GeoTiffRDD.spatial(bucket, key)
```

```scala
/* Say it was in WebMercator and we want it in ConusAlbers. */
val reprojected: RDD[(ProjectedExtent, Tile)] =
  ungridded.reproject(ConusAlbers)

/* Our layer will need metadata. Luckily, this can be derived mostly for free. */
val (zoom, meta): (Int, TileLayerMetadata[SpatialKey]) =
  TileLayerMetadata.fromRDD(reprojected, ZoomedLayoutScheme(ConusAlbers))

/* Recut our Tiles to form a proper gridded "layer". */
val gridded: RDD[(SpatialKey, Tile)] = reprojected.tileToLayout(meta)

(zoom, ContextRDD(gridded, meta))
}
```

Want to pyramid up to zoom-level 0 while you're at it?

```scala
def pyramid(bucket: String, key: String)(implicit sc: SparkContext): Stream[(Int,
→TileLayerRDD[SpatialKey])] = {
 val (zoom, layer) = layerFromS3(bucket, key)

 Pyramid.levelStream(layer, ZoomedLayoutScheme(ConusAlbers), zoom)
}
```

3.18.2 Get the `Extent` of a `Tile`

Motivation: Given a pair `(SpatialKey, Tile)`, you'd like to know the geographic area on the earth that they cover (the `Extent`). This `Extent` has many uses within GeoTrellis.

This solution assumes you have a `LayoutDefinition` on hand. If you're working with a `TileLayerRDD`, as many GeoTrellis operations do, then grabbing one is just:

```scala
val layer: TileLayerRDD[SpatialKey] = ...  /* Output of previous work */

val layout: LayoutDefinition = layer.metadata.layout
```

Now to find the `Extent` of the key/tile pair:

```scala
val (key, tile): (SpatialKey, Tile) = ...

val extent1: Extent = key.extent(layout)        /* As of GeoTrellis 1.2 */
val extent2: Extent = layout.mapTransform(key)  /* For versions < 1.2 */
```

3.18.3 Create a Layer from a single `Tile`

Motivation: You've gotten a single `Tile` as output from some GeoTrellis function. Now you'd like to promote it as-is as a GeoTrellis layer, perhaps for easy output.

First, it is assumed that you know the projection (`CRS`) that your `Tile` is in, and that you've calculated its `Extent`. If so, you can construct a `ProjectedRaster`:

```scala
val tile: Tile = ...
val crs: CRS = ...
```

```scala
val extent: Extent = ...

val raster: ProjectedRaster[Tile] = ProjectedRaster(Raster(tile, extent), crs)
```

This is the minimum amount of information required to construct a Layer. A function that does that could look like this:

```scala
import geotrellis.raster._
import geotrellis.spark._
import geotrellis.spark.SpatialKey._   /* To get a `Boundable` instance for␣
↪`SpatialKey` */
import geotrellis.spark.tiling._
import geotrellis.vector._
import org.apache.spark._
import org.apache.spark.rdd._

/** Convert an in-memory `Tile` into a GeoTrellis layer. */
def toLayer(pr: ProjectedRaster[Tile])(implicit sc: SparkContext):␣
↪TileLayerRDD[SpatialKey] = {

  val layer: RDD[(ProjectedExtent, Tile)] =
    sc.parallelize(List((ProjectedExtent(pr.raster.extent, pr.crs), pr.raster.tile)))

  val scheme: LayoutScheme = ZoomedLayoutScheme(pr.crs)

  /* The metadata, plus the zoom level corresponding to the size of the Tile.
   * We don't need the zoom level here, but it deserves a note.
   */
  val meta: (Int, TileLayerMetadata[SpatialKey]) = layer.collectMetadata(scheme)

  ContextRDD(layer.tileToLayout[SpatialKey](meta._2), meta._2)
}
```

3.18.4 Work with S3 using a custom S3Client configuration

Motivation: You would like to work with assets on S3, but you want to use an S3 client (or clients) with a configuration (various configurations) different from the default client configuration.

This can be accomplished by sub-classing `S3AttributeStore` and/or `S3ValueReader`, perhaps anonymously.

```scala
import geotrellis.spark.io.s3._
import com.amazonaws.services.s3.{AmazonS3Client=>AWSAmazonS3Client}

val aws: AWSAmazonS3Client = ??? /* Special configuration happens here */
val specialS3client = new AmazonS3Client(aws)

val attributeStore = new S3AttributeStore("my-bucket", "my-prefix") {
  override def s3Client = specialS3Client
}

val valueReader = new S3ValueReader(attributeStore) {
  override def s3Client = specialS3Client
}
```

3.18.5 Saving the Tiles of a Layer as GeoTiffs to S3

Motivation: You would like to save the `Tiles` of your layer as GeoTiffs to a S3.

If the size of your `Tiles` are fine and you're to save them:

```scala
import geotrellis.raster.io.geotiff._
import geotrellis.spark.io.s3._

val rdd: TileLayerRDD[SpatialKey] = ???

// Convert the values of the layer to SinglebandGeoTiffs
val geoTiffRDD: RDD[(K, SinglebandGeoTiff)] = rdd.toGeoTiffs()

// Convert the GeoTiffs to Array[Byte]]
val byteRDD: RDD[(K, Array[Byte])] = geoTiffRDD.mapValues { _.toByteArray }

// In order to save files to S3, we need a function that converts the
// Keys of the layer to URIs of their associated values.
val keyToURI = (k: SpatialKey) => s"s3://path/to/geotiffs/${k.col}_${k.row}.tif"

byteRDD.saveToS3(keyToURI)
```

If you'd like the size of the `Tiles` in the layer to be a different size before saving:

```scala
import geotrellis.raster.io.geotiff._
import geotrellis.spark.io.s3._
import geotrellis.spark.regrid._

val rdd: TileLayerRDD[SpatialKey] = ???

// Regrid the Tiles so that they are 512x512
val regridedRDD: TileLayerRDD[SpatialKey] = rdd.regrid(512, 512)

// Convert the values of the layer to SinglebandGeoTiffs
val geoTiffRDD: RDD[(K, SinglebandGeoTiff)] = regridedRDD.toGeoTiffs()

// Convert the GeoTiffs to Array[Byte]]
val byteRDD: RDD[(K, Array[Byte])] = geoTiffRDD.mapValues { _.toByteArray }

// In order to save files to S3, we need a function that converts the
// Keys of the layer to URIs of their associated values.
val keyToURI = (k: SpatialKey) => s"s3://path/to/geotiffs/${k.col}_${k.row}.tif"

byteRDD.saveToS3(keyToURI)
```

3.19 Architecture Decision Records

These are detailed records of technical decisions made in the past regarding GeoTrellis.

3.19.1 0001 - Streaming Writes

Context

To write streaming data (e.g. `RDD[(K, V)]`) to an S3 backend it is necessary to map over rdd partitions and to send multiple async PUT requests for all elements of a certain partition, it is important to synchronize these requests in order to be sure, that after calling a `writer` function all data was ingested (or at least attempted). Http status error `503 Service Unavailable` requires resending a certain PUT request (with exponential backoff) due to possible network problems this error was caused by. `Accumulo` and `Cassandra` writers work in a similar fashion.

To handle this situation we use the `Task` abstraction from Scalaz, which uses it's own `Future` implementation. The purpose of this research is to determine the possibility of removing the heavy `Scalaz` dependency. In a near future we will likely depend on the `Cats` library, which is lighter, more modular, and covers much of the same ground as Scalaz. Thus, to depend on `Scalaz` is not ideal.

Decision

We started by a moving from Scalaz `Task` to an implementation based on the scala standard library `Future` abstraction. Because `List[Future[A]]` is convertible to `Future[List[A]]` it was thought that this simpler home-grown solution might be a workable alternative.

Every `Future` is basically some calculation that needs to be submitted to a thread pool. When you call `(fA: Future[A]).flatMap(a => fB: Future[B])`, both `Future[A]` and `Future[B]` need to be submitted to the thread pool, even though they are not running concurrently and could run on the same thread. If `Future` was unsuccessful it is possible to define recovery strategy (in case of `S3` it is neccesary).

We faced two problems: difficulties in `Future` synchronization (`Future.await`) and in `Future` delay functionality (as we want an exponential backoff in the `S3` backend case).

We can await a `Future` until it's done (`Duration.Inf`), but we can not be sure that `Future` was completed exactly at this point (for some reason - this needs further investigation - it completes a bit earlier/later).

Having a threadpool of `Futures` and having some `List[Future[A]`, awaiting of these `Futures` does not guarantees completeness of each `Future` of a threadpool. Recovering a `Future` we produce a *new* `Future`, so that recoved `Futures` and recursive `Futures` are *new* `Futures` in the same threadpool. It isn't obvious how to await all *necessary* `Futures`. Another problem is *delayed* `Futures`, in fact such behaviour can only be achieved by creating *blocking* `Futures`. As a workaround to such a situation, and to avoid *blocking* `Futures`, it is possible to use a `Timer`, but in fact that would be a sort of separate `Future` pool.

Let's observe Scalaz `Task` more closely, and compare it to native scala `Futures`. With `Task` we recieve a bit more control over calculations. In fact `Task` is not a concurrently running computation, it's a description of a computation, a lazy sequence of instructions that may or may not include instructions to submit some of calculations to thread pools. When you call `(tA: Task[A]).flatMap(a => tB: Task[B])`, the `Task[B]` will by default just continue running on the same thread that was already executing `Task[A]`. Calling `Task.fork` pushes the task into the thread pool. Scalaz `Tasks` operates with their own `Future` implementation. Thus, having a stream of `Tasks` provides more control over concurrent computations.

Some implementations were written, but each had synchronization problems. This attempt to get rid of the Scalaz dependency is not as trival as we had anticipated.

This is not a critical decision and, if necessary, we can come back to it later.

Consequences

All implementations based on `Futures` are non-trival, and it requires time to implement a correct write stream based on native `Futures`. Here are the two simplest and most transparent implementation variants, but both have

synchronization problems.

Scalaz `Tasks` seem to be better suited to our needs. `Tasks` run on demand, and there is no requirement of instant submission of `Tasks` into a thread pool. As described above, `Task` is a lazy sequence of intructions and some of them could submit calculations into a thread pool. Currently it makes sense to depend on Scalaz.

3.19.2 0002 - HDFS Raster Layers

Context

Raster layer is a regular grid of raster tiles, represented as a `RDD[(K, V)]` where `K` contains the column, row, and/or time. Raster layer storage scheme must support two forms of queries with different requirements:

1. Distributed bounding box queries

    - Minimum time between start of the query and time at which records are inspected for a match

    - Minimum number of records discarded during query refinement stage

2. Key/Value look-ups

    - Clear mapping from any K to a single block file

    - Efficient seeks to any random value in the layer

HDFS does not provide any active index management so we must carefully define a storage and indexing scheme that supports both of those cases.

Decision

The design builds on an established pattern of mapping a multi-dimensional tile key to a one-dimensional index using a space filling curve (SFC). This requires definition of bounding spatial extent and resolution but provides a total ordering for our records.

MapFiles

The layer will be sorted and written to multiple Hadoop MapFiles. `MapFile` consist of two files:

- `data` file is a `SequenceFile` of `LongWritable` and `BytesWritable` key/value pairs where the key is the SFC index and value bytes are Avro encoded `Vector[(K,V)]` where all `K`s map to the given SFC index.

- `index` file is a `SequenceFile` which maps a `LongWritable` in seen in `data` file to its offset at some defined `indexInterval`.

When `MapFile` is open the `index` is read fully and allows fast random seeks into the `data` file.

Each map file will consequently correspond to an SFC range from from first to last key stored in the file. Because the whole layer is sorted before being written we can assume that that ranges covered by the map files are exclusive.

It will be important to know which SFC range each file corresponds to and to avoid creating an addition overall index file we record the value of the first SFC index stored in the map file as part of the file name.

We experimented with using a bloom filter index, but it did not appear appropriate. Because each file will be restricted to be no bigger than a single HDFS block (64M/128M) the time to compute and store the bloom filter does not offer any speed improvements on per-file basis.

Single Value Queries

In a single value query we are given an instance of `K` and we must produce a corresponding `V` or an error. The first step is to locate the `MapFile` which potentially contains `(K, V)` record. Because the layer records are indexed by their SFC index we map `K` to `i:  Long` and determine which file contains potential match by examining the file listing and finding the file with maximum starting index that is less than equal `i`. At this point the `MapFile` must be opened and queried for the key.

The file listing is a comparatively expensive operation that is cached when we create a `Reader[K, V]` instance for a given layer from `HadoopValueReader`. Additionally as we maintain an LRU cache of `MapFiless` as we open them to satisfy client requests. Because SFC preserves some spatial locality of the records, geographically close records are likely to be close in SFC index, and we expect key/value queries to be geographically grouped, for instance requests from a map viewer. This leads us to expect that `MapFile` LRU cache can have a high hit-rate.

Once we have located a record with matching SFC index we must verify that it contains a matching `K`. This is important because several distinct values of `K` can potentially map to the same SFC index.

Bounding Box Queries

To implement bounding box queries we extend `FileInputFormat`, the critical task is to filter the potential file list to remove any files which do not have a possible match. This step happens on the Spark driver process so it is good to perform this task without opening the files themselves. Again we exploit the fact that file names contain the first index written and assume that a file covers SFC range from that value until the starting index of the file with the next closest index.

Next the query bounding box is decomposed into separate list of SFC ranges. A single contiguous bounding box will likely decompose into many hundreds or even thousands of SFC ranges. These ranges represent all of the points on SFC index which intersect the query region. Finally we discard any `MapFile` whose SFC index range does not intersect the the bounding box SFC ranges.

The job of inspecting each `MapFile` is distributed to executors which perform in-sync traversal of query SFC ranges and file records until the end of each candidate file is reached. The resulting list of records is checked against the original bounding box as a query refinement step.

Layer Writing

When writing a layer we will receive `RDD[(K, V)] with Metadata[M]` with unknown partitioning. It is possible that two records which will map to the same SFC index are in fact located on different partitions.

Before writing we must ensure that all records that map to a given SFC index value reside on the same partition and we are able to write them in order. This can be expressed as a `rdd.groupByKey(k => sfcIndex(k)).sortByKey`. However we can avoid the double shuffle implied here by partitioning the `rdd` on SFC index of each record and defining partition breaks by inspecting dataset bounding box which is a required part of `M`. This approach is similar to using `RangePartitioner` but without the requirement of record sampling. Critically we instruct Spark to sort the records by their SFC index during the single shuffle cause by repartitioning.

With records thus partitioned and sorted we can start writing them to `MapFiles`. Each produced file will have the name of `part-r-<partition number>-<first record index>`. This is trivial to do because we have the encoded record when we need to open the file for writing. Additionally we keep track to number of bytes written to each file so we can close it and roll over to a new file if the next record written is about to cross the HDFS block boundary. Keeping files to a single block is a standard advise that optimizes their locality, it is now not possible to have a single file that is stored across two HDFS nodes.

Consequences

This storage strategy provides key features which are important for performance:

- Writing is handled using a single shuffle, which is minimum required to get consistency

- Sorting the records allows us to view them as exclusive ranges and filter large number of files without opening them

- Storing index information in the file name allows us to perform query planning without using a secondary index or opening any of the individual files

- Individual files are guaranteed to never exceed block boundary

- There is a clear and efficient mapping from any `K` to a file potentially containing the matching record

Testing showed that `HadoopValueReader` LRU caching strategy is effective and it provides sufficient performance to support serving a rendered tile layer to a web client directly from HDFS. It is likely that this performance can be further improved by adding an actor-based caching layer to re-order the requests and read `MapFiles` in order.

Because each file represents an exclusive range and there is no layer wide index to be updated there is a possibility of doing an incremental layer update where we only change those `MapFiles` which intersect with the updated records.

3.19.3 0003 - Readers / Writers Multithreading

Context

Not all GeoTrellis readers and writers implemented using MR jobs (Accumulo RDDReader, Hadoop RDDReaders), but using socket reads as well. This (socket) this approach allows to define paralelizm level depending on system configuration, like CPU, RAM, FS. In case of `RDDReaders`, that would be threads amount per rdd partition, in case of `CollectionReaders`, that would be threads amount per whole collection.

All numbers are more impericall rather than have strong theory approvals. Test cluster works in a local network to exclude possible network issues. Reads tested on ~900 objects per read request of landsat tiles (test project).

Test cluster

- Apache Spark 1.6.2

- Apache Hadoop 2.7.2

- Apache Accumulo 1.7.1

- Cassandra 3.7

Decision

Was benchmarked functions calls performace depending on RAM / and CPU cores availble.

File Backend

`FileCollectionReader` optimal (or reasonable in most cases) pool size equal to cores number. As well there could be FS restrictions, that depends on a certain FS settings.

- *collection.reader: number of CPU cores available to the virtual machine*

- *rdd.reader / writer: number of CPU cores available to the virtual machine*

Hadoop Backend

In case of `Hadoop` we can use up to 16 threads without reall significant memory usage increment, as `HadoopCollectionReader` keeps in cache up to 16 `MapFile.Readers` by default (by design). However using more than 16 threads would not improve performance signifiicantly.

- *collection.reader: number of CPU cores available to the virtual machine*

S3 Backend

`S3` threads number is limited only by the backpressure, and that's an impericall number to have max performance and not to have lots of useless failed requests.

- *collection.reader: number of CPU cores available to the virtual machine, <= 8*
- *rdd.reader / writer: number of CPU cores available to the virtual machine, <= 8*

Accumulo Backend

Numbers in the table provided are average for warmup calls. Same results valid for all backends supported, and the main really performance valueable configuration property is avaible CPU cores, results table:

4 CPU cores result (m3.xlarge):

Threads	Reads time (ms)	Comment
4	~15,541	•
8	~18,541	~500mb+ of ram usage to previous
32	~20,120	~500mb+ of ram usage to previous

8 CPU cores result (m3.2xlarge):

Threads	Reads time (ms)	Comment
4	~12,532	•
8	~9,541	~500mb+ of ram usage to previous
32	~10,610	~500mb+ of ram usage to previous

- *collection.reader: number of CPU cores available to the virtual machine*

Cassandra Backend

4 CPU cores result (m3.xlarge):

Threads	Reads time (ms)	Comment
4	~7,622	•
8	~9,511	Higher load on a driver node + (+ ~500mb of ram usage to previous)
32	~13,261	Higher load on a driver node + (+ ~500mb of ram usage to previous)

8 CPU cores result (m3.2xlarge):

Threads	Reads time (ms)	Comment
4	~8,100	•
8	~4,541	Higher load on a driver node + (+ ~500mb of ram usage to previous)
32	~7,610	Higher load on a driver node + (+ ~500mb of ram usage to previous)

- *collection.reader: number of CPU cores available to the virtual machine*

- *rdd.reader / writer: number of CPU cores available to the virtual machine*

Conclusion

For all backends performance result are pretty similar to `Accumulo` and `Cassandra` backend numbers. In order not to duplicate data these numbers were omitted. Thread pool size mostly depend on CPU cores availble, less on RAM. In order not to loose performane should not be used threads more than CPU cores availble for java machine, otherwise that can lead to significant performance loss.

3.19.4 0004 - Spark Streaming

Context

During the current `GeoTrellis` Ingest process, data is ingested as a huge chunk of data. During Ingest the tiling shuffle breaks the memory allocation the process is very likely to crash. One of reasons to investigate Streaming to ingest data incrementally, definitely that would be slower, as it requires layer *update*. Another point that Spark Streaming supports checkpoiniting: potentially we can persist application context and in case of failure to recover context.

Decision

Was implemented a beta API and we tried to use `Spark Streaming` in Landsat EMR Demo project.

API

`Spark Streaming` API provides easy wrapping / reusing of GeoTrellis RDD functions, as `DStream` implements:

```
def foreachRDD(foreachFunc: (RDD[T])  Unit): Unit
def transform[U](transformFunc: (RDD[T])  RDD[U])(implicit arg0: ClassTag[U]):␣
→DStream[U]
def transformWith[U, V](other: DStream[U], transformFunc: (RDD[T], RDD[U]) ␣
→RDD[V])(implicit arg0: ClassTag[U], arg1: ClassTag[V]): DStream[V]
```

And other higher ordered functions, which makes possible GeoTrellis `RDD` functions usage without reimplementing, but by just with wrapping our core api.

Basic terms and important Spark Streaming setting to control performance:

- `Batch interval` - the interval at which the streaming API will update (socket / folder / receiver period lookup) data

- `Window` - data between times

- `receiver.maxRate` - maximum rate (number of records per second) at which each receiver will receive data

- `backpressure.enabled` - in fact just dynamically sets `receiver.maxRate`, upper bounded `receiver.maxRate` that can be processed by cluster

- `Streaming Events` - some events that can be set (exmpl. on microbatch complete), description would be provided below

At first the idea was to ingest tiles as chunked batches and to process these batches sequentially. Instead of a common writer was used `WriterOrUpdater`. The problem was to control somehow the stream, and to have only one "batch" processed. But it turned out that it is not possible via standard Spark Streaming API. The problem is that input batch immediately splited into microbatches and into lots of jobs, and it is not possible to understand the real state of the current ingest process.

The consequence of `Spark Streaming` usage was just slowing down the ingest process (as input was parallelized, and instead of just one write, in a common ingest process, `WriteOrUpdate` was called every time, and mostly `Update` was used).

As a workaround was tried to setup listeners on a `DStream`:

```scala
trait StreamingListener {
  /** Called when a receiver has been started */
  def onReceiverStarted(receiverStarted: StreamingListenerReceiverStarted): Unit

  /** Called when a receiver has reported an error */
  def onReceiverError(receiverError: StreamingListenerReceiverError): Unit

  /** Called when a receiver has been stopped */
  def onReceiverStopped(receiverStopped: StreamingListenerReceiverStopped): Unit

  /** Called when a batch of jobs has been submitted for processing. */
  def onBatchSubmitted(batchSubmitted: StreamingListenerBatchSubmitted): Unit

  /** Called when processing of a batch of jobs has started.  */
  def onBatchStarted(batchStarted: StreamingListenerBatchStarted): Unit

  /** Called when processing of a batch of jobs has completed. */
  def onBatchCompleted(batchCompleted: StreamingListenerBatchCompleted): Unit

  /** Called when processing of a job of a batch has started. */
  def onOutputOperationStarted(
    outputOperationStarted: StreamingListenerOutputOperationStarted): Unit

  /** Called when processing of a job of a batch has completed. */
  def onOutputOperationCompleted(
    outputOperationCompleted: StreamingListenerOutputOperationCompleted): Unit
}
```

The most interesting events are `onBatchCompleted` an `onOutputOperationCompleted`, but they called not on the *input* batch, but on microbatches (empirically proofed information).

Conclusion

The use case of controlling memory and the Ingest process is possible, but not using `Spark Streaming` only. The logic of holding the input batch should be handled by some additional service, and in fact that means that we can't control directly memory usage using `Spark Streaming`. However it is still an interesting tool and we have a use case of monitoring some folder / socket / other source and receiving tiles in some batches periodically and potentially, an interesting `Feature` Ingest API can be implemented.

As a solution for Landsat EMR Demo project it is possible to set up certain batches (example: by 5 tiles), and certain batch interval (example: each 10 minutes), but that solution not prevents application from increased memory usage.

3.19.5 0005 - ETL Pipelne

Context

The current GeoTrellis ETL does not allow us to determine ETL as a pipeline of transformations / actions. This document describes a new approach (inspired by PDAL Pipeline) with a new ETL JSON description.

The new ETL would be in fact a Pipeline, which would allow flexible input sources / transofrmations / write steps definitions, in addition a flexible way to apply user defined functions would be introduced.

The motivation to write this ADR appeared since there are much more common and generic things to do during the ingest process, and it is possible to cover them all. That would help to reduce unnecessary code writing, would allow to provide a flexible GeoTrellis CLI tool (it's not very convenient and obvious how to use current ETL definitions).

Decision

We can divide the Pipeline steps into three categories:

- Read
- Transform
- Write

And steps which can be done during the steps above are the following (it's a rough list):

- Read input from a backend (`read.{file | hadoop | s3}`)
- Group input sources (`read.group`)
- Reproject buffered (`transform.reproject.buffered`)
- Reproject per-tile (`transform.reproject.per-tile`)
- Tile to layout (`transform.tile`)
- Write to a backend (`write.{file | hadoop | s3 | accumulo | cassandra | hbase}.{save | update | reindex}`)
- Write into tiff / png / jps (`write.render`)
- Save

It is possible to represent all these steps as JSON objects, and an array of these objects would be instructions Pipeline. There still would be two different json inputs: `pipeline.json` and `backend-profiles.json` as it seems to be a reasonable way to divide parameters semantically. `backend-profiles.json` is an optional file, as some times it can be useful to setup credentials as ENV variables.

backend-profiles.json

Without significant changes, as it already provides minimal information about backends credentials. However this file can be optional, as in some cases credentials can be setuped as ENV variables.

pipeline.json

This file would be a set of sequential instructions, to read input data, transform it, and to write. Each transformation can be applied to a previous transformation result type and in some cases can rename / change number of separate sources to which functions would be applied.

Scheme with Pipeline work description (arrows are actions):
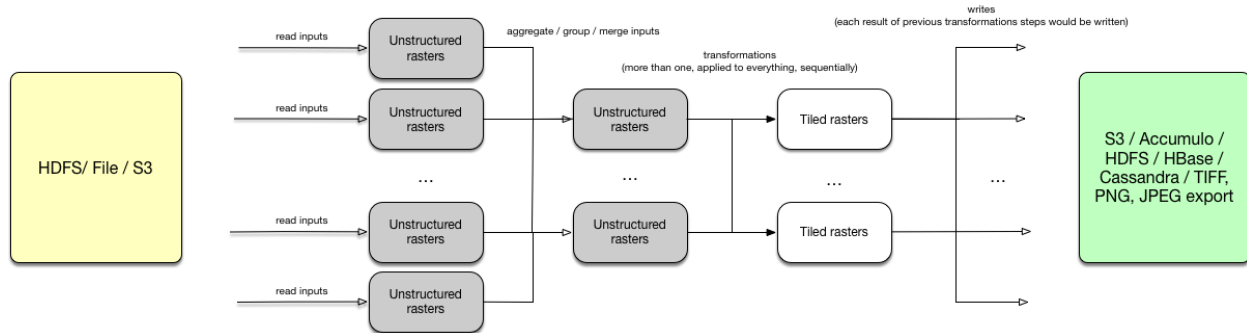
The first attempt to describe it as JSON:

Read steps

Read definition:

```
{
  "type": "read.{hadoop | s3 | file}",
  "profile": "credentials profile name",
  "path": "{uri-path}",
  "crs": "input source CRS",
  "tag": "tag-name" // tag means current layer name
}
```

Each input can be corresponded with some input alias tag, after the sequence of read steps, should be applied a step which would define transformation functions applications behaviour (should all transformations be applied per each input or inputs would be grouped somehow)

Transformation steps

Transformations can be applied to the data of different types. Accoridng to GeoTrellis type system we can divide rasters into

1.   Unstructured   rasters   (`RDD[({ProjectedExtent | TemporalProjectedExtent}, {Tile | MultibandTile})]`) 1.   Tiled   rasters   (`RDD[({SpatialKey | SpaceTimeKey}, {Tile | MultibandTile})]`)

Group / Aggregate definition:

```
{
  "type": "transform.group",
  "tags": ["{tags array}"],
  "tag": "tag-name" // in fact renames inputs into a new name
}
```

Can be done several group functions, all further transformation steps would be applied to all inputs (by default), or to a specified tag.

Merge definition:

```
{
  "type": "transform.merge",
  "tags": ["{tags array}"],
  "tag": "tag-name"
}
```

Merges input into one multiband rdd.

Map definition:

```
{
  "type": "transform.map",
  "function": "org.azavea.custommap"
  "tags": ["{tags array}"],
  "tag": "tag-name"
}
```

Map values function, can be applied to input and to tiled (K, V) pairs.

- *function* — path to a user defined or inbuilt function

Reproject definition:

Buffered reproject can be applied only to tiled rasters:

```
{
  "type": "transform.reproject.buffered",
  "crs": "{EPSG code | EPSG name | proj4 string}"
}
```

Per-tile reproject can be applied only to unstructured rasters:

```
{
  "type": "transform.reproject.per-tile",
  "crs": "{EPSG code | EPSG name | proj4 string}"
}
```

- *crs* — {EPSG code | EPSG name | proj4 string} destination CRS

Tile to layout:

```
{
  "type": "transform.tile",
  "resampleMethod": "{nearest-neighbor | bilinear | cubic-convolution | cubic-spline␣
↪| lanczos}",
  "layoutScheme": "zoomed",
  "tileSize": 256,
  "cellSize": {
```

(continues on next page)

```
      "width": 0.5,
      "height": 0.5
    },
    "partitions": 5000
}
```

- *tileSize* — destination tile size [optional field]

- *resampleMethod* — {nearest-neighbor | bilinear | cubic-convolution | cubic-spline | lanczos} methods are possible

- *cellSize* — cellSize [optional field]

- *partitions* — partitions number after tiling [optional field]

Write steps

Render definition:

```
{
  "type": "write.render",
  "format": "{tiff | png}",
  "path": "{path | pattern}"
}
```

- *format* — {tiff | png} supported formats

- *path* — {path | pattern} output path, can be specified as a pattern

{Save | Update | Reindex} definition:

```
{
  "type": "write.{file | hadoop | s3 | accumulo | cassandra | hbase}.{save | update |␣
→reindex}",
  "name": "layer name",
  "profile": "credentials profile name",
  "uri": "uri with or without credentials",
  "pyramid": "{true | false}",
  "maxZoom": 19,
  "keyIndexMethod": {
    "type": "{zorder | hilbert}",
    "temporalResolution": 86400000
  }
}
```

- *name* — layer name, all inputs would be saved / updated / reindexed with that name

- *profile* — profile name, can be specified in the backend-profiles.json, default profiles available: {file | hadoop | s3}

- *uri* — path or table name

- *maxZoom* — max zoom level [optional field]

- *keyIndexMethod:*

    - *type* — {zorder | hilbert}

    - *temporalResolution* — temporal resolution in ms, if specified it would be a temporal index [optional field]

Pipeline example:

Chapter 3. Hello Raster!

```
[
  {
    "type": "read.hadoop",
    "profile": "credentials profile name",
    "path": "{uri-path}",
    "crs": "input source CRS",
    "tag": "A"
  },
  {
    "type": "read.s3",
    "profile": "credentials profile name",
    "path": "{uri-path}",
    "crs": "input source CRS",
    "tag": "B"
  },
  {
    "type": "read.s3",
    "profile": "credentials profile name",
    "path": "{uri-path}",
    "crs": "input source CRS",
    "tag": ""
  },
  {
    "type": "transform.group",
    "tags": ["A", "B"], // no more tags A and B in the context of this pipeline
    "tag": "D"
  },
  {
    "type": "transform.reproject.buffered",
    "crs": "{EPSG code | EPSG name | proj4 string}"
  },
  {
    "type": "transform.tile",
    "resampleMethod": "{nearest-neighbor | bilinear | cubic-convolution | cubic-
→spline | lanczos}",
    "layoutScheme": "zoomed",
    "tileSize": 256,
    "cellSize": {
      "width": 0.5,
      "height": 0.5
    },
    "partitions": 5000
  },
  {
    "type": "write.render",
    "format": "{tiff | png}",
    "path": "{path | pattern}"
  },
  {
    "type": "write.{save | update | reindex}",
    "name": "layer name",
    "profile": "credentials profile name",
    "uri": "uri with or without credentials",
    "pyramid": "{true | false}",
    "keyIndexMethod": {
      "type": "{zorder | hilbert}",
      "temporalResolution": 86400000
```

```
    },
    "maxZoom": 19
  },
  {
    "type": "write.{save | update | reindex}",
    "name": "layer name",
    "profile": "credentials profile name",
    "uri": "uri with or without credentials",
    "pyramid": "{true | false}",
    "keyIndexMethod": {
      "type": "{zorder | hilbert}",
      "temporalResolution": 86400000
    },
    "maxZoom": 19
  }
]
```

Problems

It worked in the PDAL Pipeline case, as all operations worked on the same types. Input sources could be different, but the internal type is always the same, the means that transformation functions have always the same type (as that would be the most significant problem in the GeoTrellis case (as inputs can be of a different type, and as a consequence there are two problems: such functions application and such functions result type)).

During the attempt to implement rough version, was tried to unify typed and untyped ASTs (it was not obvious that in different cases different type information would be required), and different AST interpreters should be used (for typed and untyped cases).

The following problems were figured out:

- Types have to be described in the JSON representation

- Internal AST should be typed (for internal Scala DSL) and untyped (with erased types, for JSON DSL, and with type information in the JSON fields)

- The generic approach in JSON description should be followed, so the type would be a classname, and during JSON parsing the correct AST Node representation should be loaded (for example using Java class loader).

- As a consequence interpreters for typed and untyped ASTs should be different.

The second attempt to describe it as JSON:

There still a question how internal AST nodes should be represented.

1. The correct Node (typed or untyped) would be picked during JSON parsing

2. Json would be parsed into the set of internal case classes, and after that converted into typed ot untyped AST nodes.

Json descriptions would be quite the same, but into the all *type* fields should be inserted actual class name, and in such class name path (package names) this class type can be hidden.

For example:

```
"type": "geotrellis.pipeline.singleband.temporal.Reproject" // singleband and
→temporal quite important information
```

It is not quite obvious how and when such class would be loaded, but it's possible, and it's possible to cast it into necessary type due to type information available.

The following interpreter for untyped case can be implemented (just a snippet and definitely not the final version):

```
def interpretUntyped = {
  // everything should keep ordering
  val reads: List[Read] = List()
  val transformations: List[Transform] = List()
  val writes: List[Write] = List()

  val inputs: List[(String, RDD[Any])] = reads.map { r =>
    // make instance of a class and typed; after that it's possible to erase types
    ↪again.
    // Class.forName(r.`type`).newInstance
    // read =>
    null: (String, RDD[Any])
  }

  // along with common transform operations there can be arguable aggregate functions,
  // to rename multiple inputs
  // or to merge them into multiband input

  val reorogonizedInputs = transformations.flatMap {
    case t: TransformGroup =>
      // make instance of a class and typed; after that it's possible to erase types
      ↪again.
      // Class.forName(r.`type`).newInstance
      // List[(String, RDD[Any])] => List[(String, RDD[Any])] function applied
      // casting of RDD can be incapsulated into this functions
      null: List[(String, RDD[Any])]

    case t: TransformMerge =>
      // make instance of a class and typed; after that it's possible to erase types
      ↪again.
      // Class.forName(r.`type`).newInstance
      // List[(String, RDD[Any])] => List[(String, RDD[Any])] function applied
      // casting of RDD can be incapsulated into this functions
      null: List[(String, RDD[Any])]

      // no transofmration steps applied
    case _ => null: List[(String, RDD[Any])]
  }

  val generalTransformations: List[(String, RDD[Any])] = reorogonizedInputs.map {
    case p @ (tag, rdd) =>
      transformations.foldLeft(p) { case (acc, tr: Transform) =>
        // make instance of a class and typed; after that it's possible to erase
        ↪types again.
        // Class.forName(r.`type`).newInstance
        // List[(String, RDD[Any])] => List[(String, RDD[Any])] functions applied
        // casting of RDD can be incapsulated into this functions
        // String as a first tuple argument can be used to be sure that
        ↪transformation can be applied
        // runtime exceptions can happen: class not found, or type can't be casted
        // shapeless.cast function can be used(?)

        // tr.instance.apply(acc)

        null: (String, RDD[Any])
      }
```

(continues on next page)

```
  }

  writes.collect { case w: Write =>
    // make instance of a class and typed; after that it's possible to erase types␣
→again.
    // Class.forName(r.`type`).newInstance
    // List[(String, RDD[Any])] => Boolean // Unit

    ()
  }
```

There can be cases to omit `write` steps, or to return `RDD`s after all transformations applied in any case. Such approach would allow to write the following expression:

```
// json can be AST / or a JSON string
val rdd: TileLayerRDD[SpatialKey] = Pipeline.execute[TileLayerRDD[SpatialKey](json)
```

However there is still a lot to explore. The most significant questions still:

- Interpreters versions
- When to class load a class with necessary type (during initial JSON parsing or after)
- How to load classes dynamically (just a Class Loader or Java SPI)

Conclusion

The current `input.json` and `output.json` files were unified into `pipeline.json` and `backend-profiles.json` seems to be already fine.A new `pipeline.json` would allow us to construct Pipelines similar to what PDAL allows. In addition, such an approach allows us to not have complicated API extensions (aka user defined steps), which can be implemented not only be implementing separate steps itself, but just by implementing necessary functions for the `transform.map` instruction.

3.20 Proj4 Implementation

GeoTrellis relies heavily on the Proj4J library, which in turn borrows much of its implementation from the proj.4 c library. There is a correspondence between proj.4 functions and Proj4J classes, although it is not direct since C and Java coding conventions vary.

Note: Currently the GeoTrellis team maintains a fork of Proj4J in the GeoTrellis source repository, rather than relying on an official release. This includes some added projection parameters and other improvements to make proj4j more suitable for use in a distributed context such as marking appropriate objects with the `java.io.Serializable` marker interface.

The format of parameters passed to proj.4 command line tools is also supported by Proj4J, although it is not 100% compatible with all parameters. In some cases invalid parameters may cause exceptions, in others they may cause incorrect results.

3.20.1 What makes a Coordinate Reference System?

Any time you load a coordinate reference system in Proj4J you are creating an instance of the `CoordinateReferenceSystem` class. `CoordinateReferenceSystem` is a wrapper around two types:

- `Datum` which defines a coordinate system anchored to the Earth's surface

- `Projection` which defines the mapping we are using between that curved surface and 2-dimensional space. Projections in Proj4J support many parameters including units to be used, axis reordering, and some that are specific to individual projection types.

While it is technically possible to create a `CoordinateReferenceSystem` by manipulating `Projection` and `Datum` instances in Java code, typical usage is to use the `Proj4Parser` class to create one from proj.4 parameters.

Note that in contrast to the Proj4J implementation of a `CoordinateReferenceSystem` containing objects, all the coordinate system parameters are contained in the `PJ` struct in proj.4.

3.20.2 Datum

A `Datum` in Proj4J contains a reference `Ellipsoid` (model of the Earth as a mathematical surface with known equatorial radius and polar radius) and defines a mathematical transform to and from WGS84 latitude/longitude coordinates. This can be a simple 3-parameter transform (affine translation,) a 7-parameter transform (affine translate + rotate + scale,) or a `Grid` mapping part of the world's surface to latitude/longitude. Proj4's `+ellps +datum +nadgrids` and `+towgs84` parameters all affect the `Datum` in the parsed projection. In proj.4 the datum information is flattened into the `PJ` struct rather than separated out to a separate entity.

Projection

A `Projection` in Proj4J represents a formula for projecting geodetic coordinates (latitude/longitude/distance from center of the earth) to some 2D coordinate system. The Java encoding of these is a `Projection` base class with subclasses for each supported formula; eg `MercatorProjection`. The `+proj` parameter determines which projection class is instantiated. Aside from this and the datum parameters, all supported parameters affect fields of the `Projection`. In proj.4 the projection function is represented as pointers to setup, transform, inverse transform, and teardown functions, with these families of functions being implemented in one C source file per projection.

EPSG Codes

The EPSG database is released as a collection of XML files and periodically updated. The proj4 project seems to have automatic means to convert the XML parameter definitions to proj4 parameter lists, and ships a file containing one epsg coordinate system definition per line in `nad/epsg`. For Proj4J we have simply copied this file directly.

Testing

The tests for Proj4J are mostly Scala ports of JUnit tests with hand-constructed inputs and outputs. Of particular interest are the tests in the `MetaCRSTest` which reads input parameters and expected results from CSV files, making it a little easier to manage large test suites. The `GenerateTestCases.scala` file in the tests directory uses the `cs2cs` command line tool to perform sample conversions in each supported coordinate reference system for cross-validation. If you're looking to improve Proj4J's consistency with proj.4 a good place to start is the `proj4-epsg.csv` dataset in `src/test/resources/` - changing `failing` to `passing` on any line in that file will generate one test failure that you can investigate. Furthermore there are tests marked with the ScalaTest `ignore` function in many of the other test suites that would ideally be enabled and passing.

3.20.3 Further Reading

For some general information on coordinate systems and geospatial projections, see:

- Snyder, 1987: Map projection; a working manual

- Map projections

- proj.4 Wiki

3.21 High Performance Scala

3.21.1 Macros

Note: Because scala macros require a separate stage of compilation, they've been broken out into their own package in GeoTrellis. Otherwise, the functionality to be found there fits most neatly into `geotrellis.raster`.

Why Macros?

Macros are complex and harder to read than most code. As such, it is reasonable to demand justification when they are employed and to be suspicious of their necessity. Here are some reasons you'll find macros in GeoTrellis:

Boxing and Unboxing

The main purpose for all of the macros employed throughout GeoTrellis (though mostly in `geotrellis.raster`) is to avoid the JVM's so-called 'boxing' of primitive types. Boxing, in other contexts, is often called 'wrapping' and it involves passing around primitive values (which normally are lightweight and which require no special setup to work with) inside objects that are far heavier (a JVM double is 8 bytes while the boxed variant requires 24 bytes!) and which require processing time to unwrap.

Readability and Consistency of Performant Code

Above, it was pointed out that macros are harder to read. This is true, but there are some special circumstances in which their use can improve readability and help to ensure consistency. When writing performant code, it is often not possible to stay DRY (Don't Repeat Yourself). This adds significant burdens to future modifications of shared behavior (you have to change code *all over the library*) and it reduces readability by exploding the sheer amount of text which must be read to make sense of a given portion of code.

How Macros are Used

NoData Checks

Throughout `geotrellis.raster`, there are lots of checks about whether or not a given value is data or whether its value represents `NoData`.

```
isData(Int.MinValue)    // false
isNoData(Int.MinValue)  // true

isData(Double.NaN)      // false
isNoData(Double.NaN)    // true
```

This macro provides inlined code which checks to see if a given value is the GeoTrellis-internal notion of `NoData`. `Int.MinValue` and `Double.NaN` are the two `NoData` values Geotrellis `isData` and `isNoData` check against.

Type Conversion

Similar to the `NoData` checks mentioned above, type conversion macros inline functionality which converts `NoData` values for different `CellTypes` (see the documentation about celltypes for more on the different `NoData` values). This is a boon to performance and it reduces the lines of code fairly significantly.

Instead of this:

```
val someValue: Int = ???
val asFloat =
  if (someValue == Int.MinValue) Float.NaN
  else someValue.toFloat
```

We can write:

```
val someValue: Int = ???
val asFloat = i2f(someValue)
```

Tile Macros

Unlike the above macros, tile macros don't appreciably improve readability. They've been introduced merely to overcome shortcomings in certain boxing-behaviors in the scala compiler and understanding their behavior isn't necessary to read/understand the GeoTrellis codebase.

3.21.2 Micro-Optimizations

Loops

In Scala "'*for*'-loops" are more than just loops <http://docs.scala-lang.org/tutorials/FAQ/yield.html>'__. A commonly-seen feature throughout the codebase is the use of `cfor`- or `while`-loops where it would seem that an ordinary "`for`-loop" would be sufficient; we avoid using them in most cases because the flexibility of Scala's `for` construct can come at a cost.

For example, the following simple `for`-loop

```
for(i <- 0 to 100; j <- 0 to 100) { println(i+j) }
```

does not just just put the value 0 into a couple of variables, execute the loop body, increment the variables and as appropriate, and either branch or fall-through as appropriate. Instead, the Scala compiler generates objects representing the ranges of the outer- and inner-loops, as well as closures representing the interior of each loop. That results in something like this:

```
(0 to 100).foreach({ x => (0 to 100).foreach({ y => println(x+y) }) })
```

which can lead to unnecessary allocation and garbage collection. In the case of more complicated `for`-loops, the translation rules can even result in boxing of primitive loop variables.

The `cfor` construct from the Spire library avoids this problem because it is translated into the `while` construct, which does not incur the same potential performance penalties as the `for` construct.

Specialization

Another strategy that we imply to avoid unnecessary boxing is use of the `@specialized` decorator.

An example is the `Histogram[T]` type, which is used to compute either integer- or double-valued histograms. The declaration of that type looks something like this:

```
abstract trait Histogram[@specialized (Int, Double) T <: AnyVal] { ... }
```

The `@specialized` decorator and its two arguments tell the compiler that it should generate three versions of this trait instead of just one: `Histogram[Int]`, `Histogram[Double]` and the customary generic version `Histogram[T]`. Although this multiplies the amount of bytecode associated with this type by roughly a factor of three, it provides the great advantage of preventing boxing of (most) arguments and variables of type `T`. In addition, specialization also opens up additional opportunities for optimization in circumstances where the compiler knows that it is dealing with a particular primitive type instead of a object.

Mutable Types

Although use of immutable data structures is preferred in Scala, there are places in the codebase where mutable data structures have been used for performance reasons. This pattern frequently manifests as use of `foreach` on a collection rather than `filter` and/or `map`. This is helpful because less allocation of intermediate objects reduces garbage collection pressure.

3.21.3 The Tile Hierarchy

One of the most broadly-visible performance-related architectural features present in GeoTrellis is the tile hierarchy. Prompted by concerns similar to those which motivated the use of the `@specialized` decorator, this hierarchy is designed to prevent unnecessary boxing. The hierarchy provides a structure of relationships between tiles of conceptually similar types, for example `IntArrayTiles` and `DoubleArrayTile`, but they are connected via type-neutral traits rather than traits or base classes with a type parameter.

As brief example of the advantage that is provided, the types `IntArrayTile` and `DoubleArrayTile` both share a common ancestor, `ArrayTile`, which guarantees that they provide an `apply` method. That method is used to index the underlying array. In the case of `IntArrayTile` it directly indexes the array and in the case of `DoubleArrayTile` the array is indexed and then the retrieved value is converted form a `double` to an `Int` and returned. A reliable interface is provided, but without the risk of boxing that use of a type parameter would have.

Along similar lines, the fact that `IntArrayTile` and `UByteGeoTiffTile` share a common ancestor `Tile` guarantees that they both provide the method `foreach`, which allows a function to be applied to each pixel of a tile. This is possible even though those two types are backed by very different data structures: an array for the first one and complex TIFF structure for the second.

Some of the tile-related code is partially-auto generated using Miles Sabin's Boilerplate mechanism. In particular, this mechanism is used to generate the code related to `TileCombiners`.
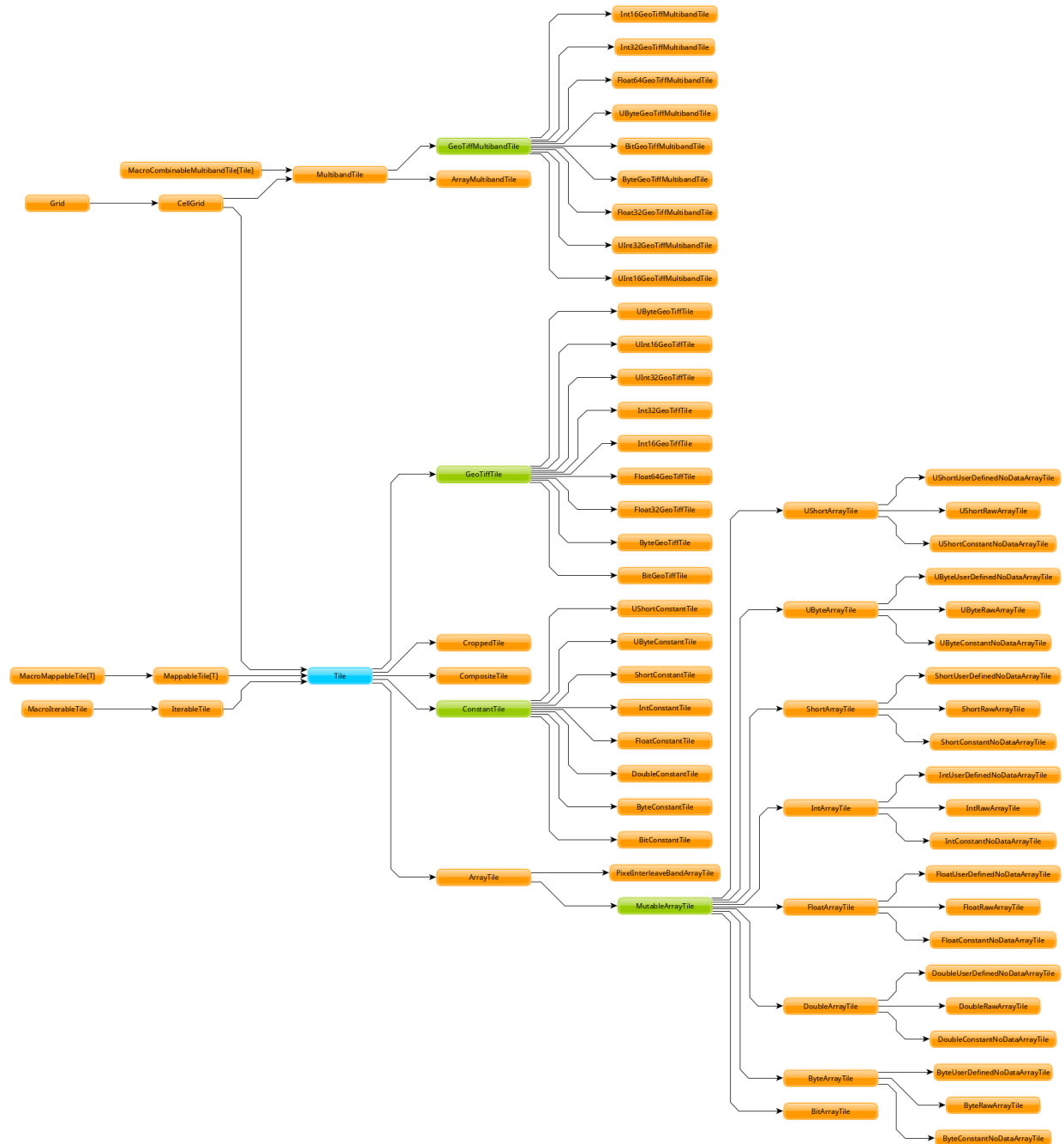
Fig. 5: tile-hierarchy

3.21.4 Spark

The two principal Spark-related performance optimizations used throughout the GeoTrellis codebase concern improved serialization performance and avoiding shuffles.

In order to improve serialization performance, we do two things: we use Kryo serialization instead of standard Java serialization and we preregister classes with Kryo.

Kryo serialization is faster and more compact than standard Java serialization. Preregistration of classes with Kryo provides a good performance boost because it reduces the amount of network traffic. When a class is not preregistered with Kryo, that class' entire name must be transmitted along with the a serialized representation of that type. However when a class is preregistered, an index into the list of preregistered classes can be sent instead of the full name.

In order to reduces shuffles, we prefer `aggregateByKey` or `reduceByKey` over `groupByKey` as recommended by the Spark documentations.