geobeam

Release 1.1.2

traviswebb@google.com

May 16, 2023

CONTENTS:

CHAPTER

ONE

MODULE DOCUMENTATION

1.1 geobeam package

1.1.1 Subpackages

1.1.1.1 geobeam.examples package

Submodules

geobeam.examples.geodatabase_frd module

Loads FRD (Flood Risk Database) layers into Bigquery using the *FILE_LOADS* insertion method. *FILE_LOADS* should be used when individual geometries can be large and complex to avoid the size limits of the *STREAMING_INSERTS* method. See: https://cloud.google.com/bigquery/quotas#streaming_inserts.

`geobeam.examples.geodatabase_frd.`**`run`**(*pipeline_args*, *known_args*)
Run the pipeline

geobeam.examples.geotiff_dem module

Example pipeline that loads a DEM (digital elevation model) raster into Bigquery using the BlockSource method.

`geobeam.examples.geotiff_dem.`**`elev_to_centimeters`**(*element*)
Convert the floating-point meters into rounded centimeters to store as INT64 in order to support clustering on this value column (elev).

`geobeam.examples.geotiff_dem.`**`run`**(*pipeline_args*, *known_args*)
Run the pipeline. Invoked by the Beam runner.

geobeam.examples.geotiff_soilgrid module

Example pipeline that loads a Soil Grid raster that contains groundwater saturation values.

`geobeam.examples.geotiff_soilgrid.`**`run`**(*pipeline_args*, *known_args*)
Run the pipeline. Invoked by the Beam runner.

geobeam.examples.shapefile_nfhl module

Example pipeline that loads the NFHL (National Flood Hazard Layer) into BigQuery.

`geobeam.examples.shapefile_nfhl.run`(*pipeline_args*, *known_args*)
Invoked by the Beam runner

Module contents

A set of examples that demonstrate *geobeam* functionality.

Usage: python -m geobeam.examples.<example> –runner=PortableRunner . . . args

1.1.2 Submodules

1.1.3 geobeam.fn module

Beam functions, transforms, and filters that can be used to process geometries in your pipeline

class `geobeam.fn.DoBlockToPixelExterior`(**unused_args*, ***unused_kwargs*)
Bases: `apache_beam.transforms.core.DoFn`

process(*element*)
Decompose a raster block into individual pixels in order to store one pixel per row

`geobeam.fn.filter_invalid`(*element*)
Use with fn.make_valid to filter out geometries that are invalid, empty, or are out of bounds.

Example: .. code-block:: python

p | beam.Map(geobeam.fn.make_valid)

beam.Map(geobeam.fn.filter_invalid)

`geobeam.fn.format_rasterblock_record`(*element*, *band_mapping=None*)
Format the tuple received from a RasterBlockSource into a record that can be inserted into BigQuery or another database.

Args: band_mapping (dict, optional): a band number to band name mapping used to name the output columns. band numbers are 1-indexed, meaning they begin at 1 (by GDAL convention)

Example:

```
from geobeam.fn import format_rasterblock_record
band_mapping = {
    1: 'elevation',
    4: 'red',
    5: 'green'
}
p | beam.Map(format_rasterblock_record, band_mapping=band_mapping)
```

`geobeam.fn.format_rasterpixel_record`(*element*, *band_mapping=None*)

`geobeam.fn.format_rasterpolygon_record`(*element*, *band_type='int'*, *band_column=None*)
Format the tuple received from the geobeam raster source into a record that can be inserted into BigQuery or another database.

Args:

band_type (str, optional): Default to int. The data type of the raster band column to store in the database.

band_column (str, optional): the name of the raster band column

Example: .. code-block:: python

beam.Map(geobeam.fn.format_record, band_column='elev', band_type=float)

`geobeam.fn.`**`format_record`**(*element*, *geom_format='geojson'*)

Format the tuple received from the geobeam file source into a record that can be inserted into BigQuery or another database.

Example: .. code-block:: python

vector p | beam.Map(geobeam.fn.format_record)

`geobeam.fn.`**`make_valid`**(*element*, *drop_z=True*)

Attempt to make a geometry valid. Returns *None* if the geometry cannot be made valid.

Example: .. code-block:: python

p | beam.Map(geobeam.fn.make_valid)

beam.Map(geobeam.fn.filter_invalid)

`geobeam.fn.`**`pixel_to_ring`**(*i*, *j*, *xfrm*)

`geobeam.fn.`**`trim_polygons`**(*element*, *d=1e-07*, *cf=1.2*)

Remove extraneous artifacts, tails, etc. from otherwise valid polygons

Args: d (float, optional): trim distance cf (float, optional): corrective factor

Exmaple: .. code-block:: python

p | beam.Map(geobeam.fn.trim_polygons, d=0.00001, cf=1.2

1.1.4 geobeam.io module

This package contains Apache Beam I/O connectors for reading from spatial data files.

`class` `geobeam.io.`**`ESRIServerSource`**(*file_pattern*, *skip_reproject=False*, *in_epsg=None*, *in_proj=None*, ***kwargs*)

Bases: `apache_beam.io.filebasedsource.FileBasedSource`

A Beam FileBasedSource for reading layers from an ESRI ArcGIS Server.

The given file(s) should be a link to a specific layer from the ArcGIS Server REST API (ex. https://services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services/USA_States_Generalized/FeatureServer/0)

p | beam.io.Read(ESRIServerSource(file_pattern))

beam.Map(print)

Args:

skip_reproject (bool, optional): Defaults to *False*. True to return *geom* in its original projection.

in_epsg (int, optional): override the source projection with an EPSG code.

in_proj (str, optional): override the source projection with a PROJ4 string.

Yields: generator of (*props*, *geom*) tuples. *props* is a *dict* containing all of the feature properties. *geom* is the geometry.

read_records (*file_name*, *range_tracker*)

Returns a generator of records created by reading file 'file_name'.

Args:

file_name: a `string` that gives the name of the file to be read. Method `FileBasedSource.open_file()` must be used to open the file and create a seekable file object.

offset_range_tracker: a object of type `OffsetRangeTracker`. This defines the byte range of the file that should be read. See documentation in `iobase.BoundedSource.read()` for more information on reading records while complying to the range defined by a given `RangeTracker`.

Returns: an iterator that gives the records read from the given file.

class geobeam.io.**GeoJSONSource** (*file_pattern*, *skip_reproject=False*, *in_epsg=None*, *in_proj=None*, ***kwargs*)

Bases: `apache_beam.io.filebasedsource.FileBasedSource`

A Beam FileBasedSource for reading GeoJSON Files.

The given file(s) should be a .geojson file.

p | beam.io.Read(GeoJSONSource(file_pattern))

beam.Map(print)

Args:

skip_reproject (bool, optional): Defaults to *False*. True to return *geom* in its original projection.

in_epsg (int, optional): override the source projection with an EPSG code.

in_proj (str, optional): override the source projection with a PROJ4 string.

Yields: generator of (*props*, *geom*) tuples. *props* is a *dict* containing all of the feature properties. *geom* is the geometry.

read_records (*file_name*, *range_tracker*)

Returns a generator of records created by reading file 'file_name'.

Args:

file_name: a `string` that gives the name of the file to be read. Method `FileBasedSource.open_file()` must be used to open the file and create a seekable file object.

offset_range_tracker: a object of type `OffsetRangeTracker`. This defines the byte range of the file that should be read. See documentation in `iobase.BoundedSource.read()` for more information on reading records while complying to the range defined by a given `RangeTracker`.

Returns: an iterator that gives the records read from the given file.

class geobeam.io.**GeodatabaseSource** (*file_pattern*, *gdb_name=None*, *layer_name=None*, *in_epsg=None*, *in_proj=None*, *skip_reproject=False*, ***kwargs*)

Bases: `apache_beam.io.filebasedsource.FileBasedSource`

A Beam FileBasedSource for reading geodatabases.

The given file(s) should be a zip archive containing .gdb geodatabase directory.

p | beam.io.Read(GeodatabaseSource(file_pattern))

beam.Map(print)

Args:

gdb_name (str): Required. the name of the .gdb directory in the archive, e.g. *FRD_510104_Coastal_GeoDatabase_20160708.gdb*

layer_name (str): Required. the name of the layer you want to read from the gdb, e.g. *S_CSLF_Ar*

skip_reproject (bool, optional): Defaults to *False*. True to return *geom* in its original projection.

in_epsg (int, optional): override the source projection with an EPSG code.

in_proj (str, optional): override the source projection with a PROJ4 string.

Yields: generator of (*props*, *geom*) tuples. *props* is a *dict* containing all of the feature properties. *geom* is the geometry.

read_records(*file_name*, *range_tracker*)

Returns a generator of records created by reading file 'file_name'.

Args:

file_name: a `string` that gives the name of the file to be read. Method `FileBasedSource.open_file()` must be used to open the file and create a seekable file object.

offset_range_tracker: a object of type `OffsetRangeTracker`. This defines the byte range of the file that should be read. See documentation in `iobase.BoundedSource.read()` for more information on reading records while complying to the range defined by a given `RangeTracker`.

Returns: an iterator that gives the records read from the given file.

class geobeam.io.**RasterBlockSource**(*file_pattern*, *bidx=None*, *in_epsg=None*, *in_proj=None*, *skip_reproject=False*, *return_block_transform=False*, ***kwargs*)

Bases: `apache_beam.io.filebasedsource.FileBasedSource`

A Beam FileBasedSource for reading pixel blocks from raster files in equally-sized blocks.

The raster file is read in MxN blocks, and each band is read as a MxN array. M and N are determined by the input raster file and are not runtime configurable.

RasterBlockSource optimizes for pipeline speed, and is best for ELT processes where further processing can be done by the sink (e.g. BigQuery).

p | beam.io.Read(RasterBlockSource(file_pattern))

beam.Map(print)

Args: file_pattern (str): required, passed to FileBasedSource. bidx (int[], optional): Defaults to *1*. The band indexes to read from

the raster.

skip_reproject (bool, optional): Defaults to *False*. True to return *geom* in its original projection. This can be useful if your data is in a bespoke CRS that requires a custom reprojection, or if you want to join/clip with other spatial data in the same projection. Note: you will need to manually reproject all geometries to EPSG:4326 in order to store it in BigQuery.

in_epsg (int, optional): override the source projection with an EPSG code.

in_proj (str, optional): override the source projection with a PROJ4 string.

return_block_transform (bool, optional): set True to return the window transform object and window dimensions in order to (re)-project individual pixels in a later pipeline stage.

Yields: generator of (*value*, *geom*) tuples. *value* is a 3D array; the first dimension is the band index, and the remaining two dimensions represent the pixel values for the band.

read_records (*file_name*, *range_tracker*)

Returns a generator of records created by reading file 'file_name'.

Args:

file_name: a `string` that gives the name of the file to be read. Method `FileBasedSource.open_file()` must be used to open the file and create a seekable file object.

offset_range_tracker: a object of type `OffsetRangeTracker`. This defines the byte range of the file that should be read. See documentation in `iobase.BoundedSource.read()` for more information on reading records while complying to the range defined by a given `RangeTracker`.

Returns: an iterator that gives the records read from the given file.

class geobeam.io.**RasterPolygonSource** (*file_pattern*, *bidx=1*, *in_epsg=None*, *in_proj=None*, ***kwargs*)

Bases: `apache_beam.io.filebasedsource.FileBasedSource`

A Beam FileBasedSource for reading pixels grouped by value from raster files.

The raster file is read in blocks and each block is polygonized. Each polygon is returned as a (*value*, *geom*) tuple, where *value* is the band value of the polygonized pixels, and *geom* is the Polygon geometry that corresponds to the *value*.

RasterPolygonSource optimizes for immediate usability of the output, and so is most suitable for ETL processes; it will be slower than RasterBlockSource, produce more rows of output, and can only read one band at a time.

p | beam.io.Read(RasterPolygonSource(file_pattern))

beam.Map(print)

Args: file_pattern (str): required, passed to FileBasedSource. bidx (int, optional): Defaults to *1*. The band index to read from

the raster.

skip_reproject (bool, optional): Defaults to *False*. True to return *geom* in its original projection. This can be useful if your data is in a bespoke CRS that requires a custom reprojection, or if you want to join/clip with other spatial data in the same projection. Note: you will need to manually reproject all geometries to EPSG:4326 in order to store it in BigQuery.

in_epsg (int, optional): override the source projection with an EPSG code.

in_proj (str, optional): override the source projection with a PROJ4 string.

Yields: generator of (*value*, *geom*) tuples. The data type of *value* is determined by the raster band it came from.

read_records (*file_name*, *range_tracker*)

Returns a generator of records created by reading file 'file_name'.

Args:

file_name: a `string` that gives the name of the file to be read. Method `FileBasedSource.open_file()` must be used to open the file and create a seekable file object.

offset_range_tracker: a object of type `OffsetRangeTracker`. This defines the byte range of the file that should be read. See documentation in `iobase.BoundedSource.read()`

for more information on reading records while complying to the range defined by a given `RangeTracker`.

Returns: an iterator that gives the records read from the given file.

class `geobeam.io.`**`ShapefileSource`**(*file_pattern*, *layer_name=None*, *skip_reproject=False*, *in_epsg=None*, *in_proj=None*, ***kwargs*)

Bases: `apache_beam.io.filebasedsource.FileBasedSource`

A Beam FileBasedSource for reading shapefiles.

The given file(s) should be a zip archive containing the .shp file alongside the .dbf and .prj files.

p | beam.io.Read(ShapefileSource(file_pattern))

beam.Map(print)

Args:

layer_name (str, optional): the name of the layer you want to read. Required the zipfile contains multiple layers.

skip_reproject (bool, optional): Defaults to *False*. True to return *geom* in its original projection.

in_epsg (int, optional): override the source projection with an EPSG code.

in_proj (str, optional): override the source projection with a PROJ4 string.

Yields: generator of (*props*, *geom*) tuples. *props* is a *dict* containing all of the feature properties. *geom* is the geometry.

`read_records`(*file_name*, *range_tracker*)

Returns a generator of records created by reading file 'file_name'.

Args:

file_name: a `string` that gives the name of the file to be read. Method `FileBasedSource.open_file()` must be used to open the file and create a seekable file object.

offset_range_tracker: a object of type `OffsetRangeTracker`. This defines the byte range of the file that should be read. See documentation in `iobase.BoundedSource.read()` for more information on reading records while complying to the range defined by a given `RangeTracker`.

Returns: an iterator that gives the records read from the given file.

1.1.5 geobeam.util module

This module contains utility functions that make working with geosaptial data in Google Cloud easier.

`geobeam.util.`**`get_bigquery_raster_schema`**(*band_column='value'*, *band_type='INT64'*)

Generate Bigquery table schema for a raster

`geobeam.util.`**`get_bigquery_schema`**(*filepath*, *layer_name=None*, *gdb_name=None*)

Generate a Bigquery table schema from a geospatial file

python -m geobeam.util get_bigquery_schema . . . args

Args: filepath (str): full path to the input file layer_name (str, optional): name of the layer, if file contains

multiple layers

Returns: dict: the schema, convertable to json by json.dumps(schema, indent=2)

`geobeam.util.`**`get_bigquery_schema_dataflow`**(*gcs_url*, *layer_name=None*, *gdb_name=None*)

Generate a Bigquery table schema from a geospatial file hosted on a Google Cloud Storage bucket

from apache_beam.io.gcp.bigquery_tools import parse_table_schema_from_json

table_schema = parse_table_schema_from_json(get_bigquery_schema_dataflow(known_args.gcs_url, known_args.layer_name))

Args: filepath (str): full path to the input file hosted on Google Cloud Storage layer_name (str, optional): name of the layer, if file contains

multiple layers

Returns: JSON: the schema in JSON that can be passed to the schema argument in WriteToBigQuery. Must use the parse_table_schema_from_json() from apache_beam.io.gcp.bigquery_tools

1.1.6 Module contents

geobeam root namespace.

CHAPTER

TWO

README

geobeam adds GIS capabilities to your Apache Beam pipelines.

2.1 What does geobeam do?

`geobeam` enables you to ingest and analyze massive amounts of geospatial data in parallel using Dataflow. geobeam provides a set of FileBasedSource classes that make it easy to read, process, and write geospatial data, and provides a set of helpful Apache Beam transforms and utilities that make it easier to process GIS data in your Dataflow pipelines.

See the Full Documentation for complete API specification.

2.1.1 Requirements

- Apache Beam 2.46+
- Python 3.8+

  Note: Make sure the Python version used to run the pipeline matches the version in the built container.

2.1.2 Supported input types

File format	Data type	Geobeam class
`tiff`	raster	`RasterBlockSource` and `RasterPolygonSource`
`shp`	vector	`ShapefileSource`
`gdb`	vector	`GeodatabaseSource`
`json`	vector	`GeoJSONSource`
URL	vector	`ESRIServerSource`

2.1.3 Included libraries

`geobeam` includes several python modules that allow you to perform a wide variety of operations and analyses on your geospatial data.

Module	Version	Description
gdal	3.5.2	python bindings for GDAL
rasterio	1.3.2	reads and writes geospatial raster data
fiona	1.8.21	reads and writes geospatial vector data
shapely	1.8.4	manipulation and analysis of geometric objects in the cartesian plane
esridump	1.11.0	read layer from ESRI server

2.2 How to Use

2.2.1 1. Install the module

```
pip install geobeam
```

2.2.2 2. Write your pipeline

Write a normal Apache Beam pipeline using one of `geobeam`s file sources. See ``*geobeam/examples*` <https://github.com/GoogleCloudPlatform/dataflow-geobeam/tree/main/geobeam/examples>`_ for inspiration.

2.2.3 3. Run

2.2.3.1 Run locally

```
python -m geobeam.examples.geotiff_dem \
  --gcs_url gs://geobeam/examples/dem-clipped-test.tif \
  --dataset examples \
  --table dem \
  --band_column elev \
  --runner DirectRunner \
  --temp_location <temp gs://> \
  --project <project_id>
```

Note: Some of the provided examples may take a very long time to run locally...

2.2.3.2 Run in Dataflow

Write a Dockerfile

This will run in Dataflow as a custom container based on the ``*dataflow-geobeam/base*` <Dockerfile>`_ image. It is recommended that you publish your own container based on the Dockerfile in this repository and store it in your project's GCR registry.

```
FROM gcr.io/dataflow-geobeam/base

RUN pip install geobeam

COPY requirements.txt .
RUN pip install -r requirements.txt
```

(continues on next page)

(continued from previous page)

```
COPY . .
```

```
# build locally with docker
docker build -t gcr.io/<project_id>/geobeam
docker push gcr.io/<project_id>/geobeam

# or build with Cloud Build
gcloud builds submit --timeout 3600s --worker_machine_type n1-highcpu-8
```

2.2.3.3 Start the Dataflow job

```
# run the geotiff_soilgrid example in dataflow
python -m geobeam.examples.geotiff_soilgrid \
  --gcs_url gs://geobeam/examples/AWCh3_M_sl1_250m_ll.tif \
  --dataset examples \
  --table soilgrid \
  --band_column h3 \
  --runner DataflowRunner \
  --sdk_container_image gcr.io/dataflow-geobeam/base \
  --temp_location <temp bucket> \
  --service_account_email <service account> \
  --region us-central1 \
  --max_num_workers 2 \
  --worker_machine_type c2-standard-30 \
```

2.3 Examples

```
def run(options):
  from geobeam.io import RasterBlockSource
  from geobeam.fn import format_rasterblock_record

  with beam.Pipeline(options) as p:
    (p  | 'ReadRaster' >> beam.io.Read(RasterBlockSource(gcs_url))
        | 'FormatRecord' >> beam.Map(format_rasterblock_record)
        | 'WriteToBigquery' >> beam.io.WriteToBigQuery('geo.dem'))
```

```
def run(options):
  from geobeam.io import ShapefileSource
  from geobeam.fn import make_valid, filter_invalid, format_record

  with beam.Pipeline(options) as p:
    (p  | 'ReadShapefile' >> beam.io.Read(ShapefileSource(gcs_url))
        | 'Validate' >> beam.Map(make_valid)
        | 'FilterInvalid' >> beam.Filter(filter_invalid)
        | 'FormatRecord' >> beam.Map(format_record)
        | 'WriteToBigquery' >> beam.io.WriteToBigQuery('geo.parcel'))
```

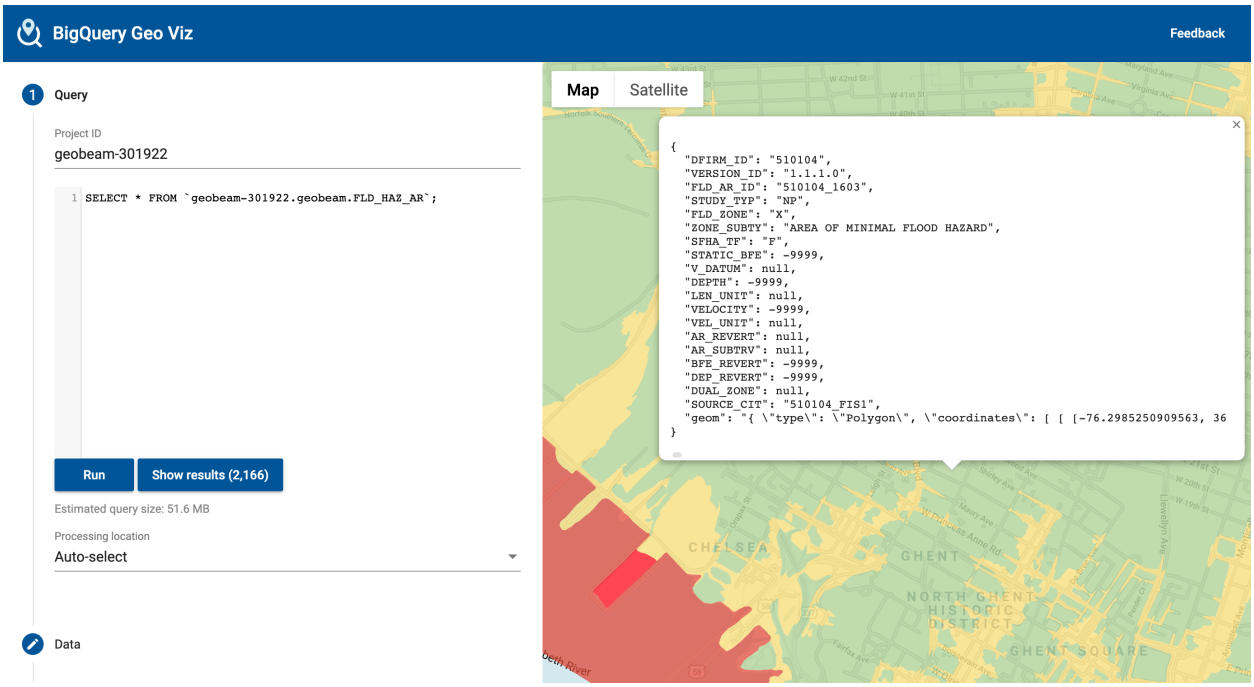
See `geobeam/examples/` for complete examples.

A number of example pipelines are available in the `geobeam/examples/` folder. To run them in your Google Cloud project, run the included terraform file to set up the Bigquery dataset and tables used by the example pipelines.

Open up Bigquery GeoViz to visualize your data.

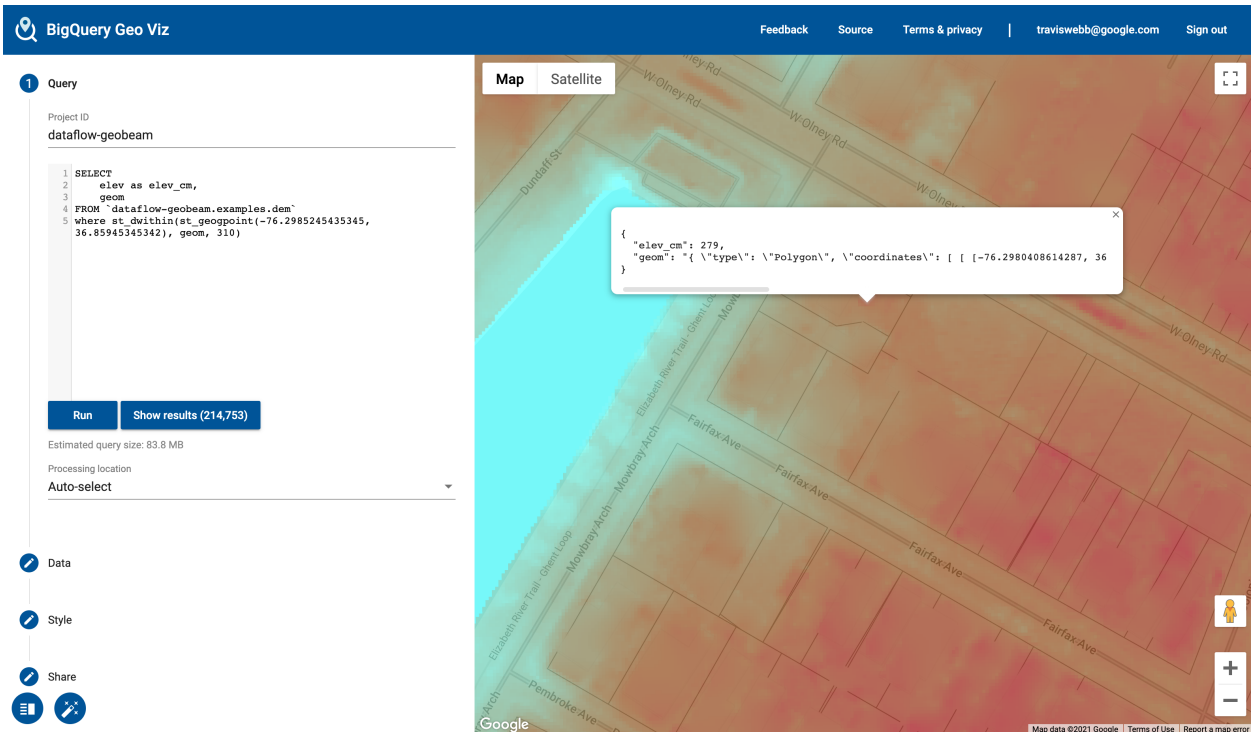
2.3.1 Shapefile Example

The National Flood Hazard Layer loaded from a shapefile. Example pipeline at ``*geobeam/examples/shapefile_nfhl.py*` <https://github.com/GoogleCloudPlatform/dataflow-geobeam/blob/main/geobeam/examples/shapefile_nfhl.py>`_



2.3.2 Raster Example

The Digital Elevation Model is a high-resolution model of elevation measurements at 1-meter resolution. (Values converted to centimeters). Example pipeline: ``*geobeam/examples/geotiff_dem.py*` <https://github.com/GoogleCloudPlatform/dataflow-geobeam/blob/main/geobeam/examples/geotiff_dem.py>`_.



2.4 Included Transforms

The `geobeam.fn` module includes several Beam Transforms that you can use in your pipelines.

Module	Description
`geobeam.fn.make_valid`	Attempt to make all geometries valid.
`geobeam.fn.filter_invalid`	Filter out invalid geometries that cannot be made valid
`geobeam.fn.format_record`	Format the (props, geom) tuple received from a vector source into a `dict` that can be inserted into the destination table
`geobeam.fn.format_rasterblock_record`	Format the output record for blocks read from `RasterBlockSource`
`geobeam.fn.format_rasterpolygon_record`	Format the output record for blocks read from `RasterPolygonSource`

2.5 Execution parameters

Each FileSource accepts several parameters that you can use to configure how your data is loaded and processed. These can be parsed as pipeline arguments and passed into the respective FileSources as seen in the examples pipelines.

Parameter	Input type	Description	Default	Required?
`skip_reproject`	All	True to skip reprojection during read	`False`	No
`in_epsg`	All	An EPSG integer to override the input source CRS to reproject from		No
`in_proj`	All	A PROJ string to override the input source CRS		No
`band_number`	Raster	The raster band to read from	`1`	No
`include_nodata`	Raster	True to include `nodata` values	`False`	No
`return_block_transform`	Raster	True to include rasterio `transform` object with each block to use with `geobeam.fn.format_rasterpixel_record`	`False`	No
`layer_name`	Vector	Name of layer to read		Yes, for shapefiles
`gdb_name`	Vector	Name of geodatabase directory in a gdb zip archive		Yes, for GDB files

2.6 License

This is not an officially supported Google product, though support will be provided on a best-effort basis.

CHAPTER

THREE

EXAMPLES

CHAPTER

FOUR

GEOBEAM EXAMPLES

4.1 shapefile_parcel

Load a shapefile of county parcels into Bigquery

4.1.1 Run locally

```
python -m geobeam.examples.shapefile_parcel \
  --runner DirectRunner \
  --project <your project>
  --temp_location gs://geobeam-pipeline-tmp \
  --gcs_url gs://geobeam/examples/ghent-parcels-shp.zip \
  --layer_name Property_Information \
  --dataset examples \
  --table parcel
```

4.1.2 Run in Dataflow

```
python -m geobeam.examples.shapefile_parcel \
  --runner DataflowRunner \
  --sdk_container_image gcr.io/dataflow-geobeam/base \
  --project dataflow-geobeam \
  --temp_location <your temp bucket>
  --service_account_email <your service account>
  --region us-central1
  --gcs_url gs://geobeam/examples/ghent-parcels-shp.zip \
  --layer_name Property_Information \
  --dataset examples \
  --table parcel
```

4.2 geodatabase_frd

4.2.1 Run locally

```
python -m geobeam.examples.geodatabase_frd \
  --runner DirectRunner \
  --project <your project> \
  --temp_location <your temp bucket> \
  --gcs_url gs://geobeam/examples/FRD_510104_Coastal_GeoDatabase_20160708.zip \
  --dataset examples \
  --table CSLF_Ar \
  --gdb_name FRD_510104_Coastal_GeoDatabase_20160708.gdb \
  --layer_name S_CSLF_Ar
```

4.2.2 Run in Dataflow

```
python -m geobeam.examples.geodatabase_frd \
  --project <your project> \
  --runner DataflowRunner \
  --sdk_container_image gcr.io/dataflow-geobeam/base \
  --temp_location <your temp bucket> \
  --service_account_email <your service account> \
  --region us-central1 \
  --gcs_url gs://geobeam/examples/FRD_510104_Coastal_GeoDatabase_20160708.zip \
  --gdb_name FRD_510104_Coastal_GeoDatabase_20160708.gdb \
  --layer_name S_CSLF_Ar \
  --dataset examples \
  --table CSLF_Ar
```

4.3 geotiff_dem

Load a Digital Elevation Model (DEM) raster into Bigquery

4.3.1 Run Locally

```
python -m geobeam.examples.geotiff_dem \
  --runner DirectRunner \
  --temp_location <your temp bucket> \
  --project <your project> \
  --gcs_url gs://geobeam/examples/ghent-dem-1m.tif \
```

4.3.2 Run in Dataflow

```
python -m geobeam.examples.geotiff_dem \
  --runner DataflowRunner \
  --sdk_container_image gcr.io/dataflow-geobeam/base \
  --project dataflow-geobeam \
  --temp_location gs://geobeam-pipeline-tmp/ \
  --service_account_email dataflow-runner@dataflow-geobeam.iam.gserviceaccount.com \
  --region us-central1 \
  --gcs_url gs://geobeam/examples/dem-clipped-test.tif \
  --dataset examples \
  --table dem \
  --schema 'elev:INT64,geom:GEOGRAPHY'
  --max_num_workers 3 \
  --worker_machine_type c2-standard-30 \
```

4.4 geotiff_soilgrid

4.4.1 Run Locally

```
python -m geobeam.examples.geotiff_soilgrid \
  --runner DirectRunner \
  --project <your project> \
  --temp_location <your temp bucket> \
  --gcs_url gs://geobeam/examples/soilgrid-test-clipped.tif \
  --dataset examples \
  --table soilgrid \
  --band_column h3
```

4.4.2 Run in Dataflow

```
python -m geobeam.examples.geotiff_soilgrid \
  --runner DataflowRunner \
  --sdk_container_image gcr.io/dataflow-geobeam/base \
  --temp_location <your temp bucket> \
  --project <your project> \
  --service_account_email <your service account> \
  --region us-central1 \
  --worker_machine_type c2-standard-8 \
  --gcs_url gs://geobeam/examples/soilgrid-test-clipped.tif \
  --dataset examples \
  --table soilgrid \
  --band_column h3
```

4.5 shapefile_nfhl

4.5.1 Run Locally

```
python -m geobeam.examples.shapefile_nfhl \
  --runner DirectRunner \
  --project <your project> \
  --temp_location <your temp bucket> \
  --gcs_url gs://geobeam/examples/510104_20170217.zip \
  --dataset examples \
  --table FLD_HAZ_AR \
  --layer_name S_FLD_HAZ_AR
```

4.5.2 Run in Dataflow

```
python -m geobeam.examples.shapefile_nfhl \
  --runner DataflowRunner \
  --project <your project> \
  --temp_location <your temp bucket> \
  --sdk_container_image gcr.io/dataflow-geobeam/base \
  --service_account_email <your service account> \
  --gcs_url gs://geobeam/examples/510104_20170217.zip \
  --layer_name S_FLD_HAZ_AR \
  --dataset examples \
  --table FLD_HAZ_AR
```

4.6 streaming_pubsub

4.6.1 Run Locally

4.6.1.1 setup pubsub emulator

```
gcloud components install pubsub-emulator
gcloud beta emulators pubsub start &
curl -X PUT -v http://localhost:8085/v1/projects/example-project/topics/example-topic
```

4.6.1.2 run pipeline

```
PUBSUB_EMULATOR_HOST=localhost:8085 python -m geobeam.examples.streaming_pubsub \
  --runner DirectRunner \
  --streaming \
  --in_proj "+proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0 +x_0=0.0 +y_0=0␣
↪+k=1.0 +units=m +nadgrids=@null +wktext  +no_defs"
```

4.6.1.3 publish messages to topic

```
curl -v \
  -X POST \
  -H "Content-Type: application/json" \
  -d "@geobeam/examples/pubsub_emulator_messages.json" \
  http://localhost:8085/v1/projects/example-project/topics/example-topic:publish
```

4.6.1.4 unfortunate note

There is a bug in the pubsub emulator, or the directrunner, that garbles the message timestamp from the emulator. You may not be able to use the emulator for testing until this is resolved.

4.7 geojson_stormwater

```
bq mk --table <dataset>.stormwater geobeam/examples/stormwater_schema.json
```

4.7.1 Run Locally

```
python -m geobeam.examples.geojson_stormwater \
  --runner DirectRunner \
  --project <your project> \
  --temp_location <your temp bucket> \
  --gcs_url gs://geobeam/examples/Stormwater_Pipes.geojson \
  --dataset examples \
  --table stormwater \
```

4.7.2 Run in Dataflow

```
python -m geobeam.examples.geojson_stormwater \
  --runner DataflowRunner \
  --project <your project> \
  --temp_location <your temp bucket> \
  --sdk_container_image gcr.io/dataflow-geobeam/base \
  --service_account_email <your service account> \
  --gcs_url gs://geobeam/examples/Stormwater_Pipes.geojson \
  --dataset examples \
  --table stormwater
```

CHAPTER

FIVE

INDICES AND TABLES

- search

PYTHON MODULE INDEX

g

INDEX

S

T