dask-geomodeling

Nov 15, 2019

Contents:

Dask-geomodeling is a collection of classes that are to be stacked together to create configurations for on-the-fly operations on geographical maps. By generating Dask compute graphs, these operation may be parallelized and (intermediate) results may be cached.

Multiple Block instances together make a view. Each Block has the `get_data` method that fetches the data in one go, as well as a `get_compute_graph` method that creates a graph to compute the data later.

Blocks are used for the on-the-fly modification of raster- and vectordata, respectively through the baseclasses *`RasterBlock()`* and *`GeometryBlock()`*. Derived classes support operations such has grouping basic math, shifting time, smoothing, reclassification, geometry operations, zonal statistics, and property field operations.

CHAPTER 1

About

This package was developed by Nelen & Schuurmans and is used commercially under the name Geoblocks. Please consult the Lizard website for more information about this product.

CHAPTER 2

Contents

2.1 Installation

2.1.1 Requirements

- python >= 3.5
- GDAL 2.* (with BigTIFF support)
- numpy
- scipy
- dask[delayed]
- pandas
- geopandas
- ipyleaflet, matplotlib, pillow (for the ipyleaflet plugin)

2.1.2 Anaconda (all platforms)

1. Install anaconda / miniconda
2. Start the *Anaconda Prompt* via the start menu
3. *conda config –add channels conda-forge*
4. *conda update conda*
5. *conda install python=3.6 gdal=2.4.1 scipy=1.3.1 dask-geomodeling ipyleaflet matplotlib pillow*

Note: The version pins of python, gdal and scipy are related to issues specific to Windows. On other platforms you may leave them out, although we do not recommend using Python 3.8 yet. If you need other python or GDAL versions

on windows: while *dask-geomodeling* itself is compatible with all current versions, you may may have a hard time getting it to work via Anaconda and it will probably be easier using the pip route listed below. Good luck out there.

2.1.3 Windows (pip)

The following recipe is still a work in progress:

1. Install Python 3.* (stable)
2. Install GDAL 2.* (MSVC 2015)
3. Add the GDAL installation path to your PATH variable
4. Start the command prompt
5. *pip install gdal==2.* dask-geomodeling ipyleaflet matplotlib pillow*
6. (optionally) *pip install ipyleaflet matplotlib pillow*

Note: You might need to setup your C++ compiler according to this

2.1.4 On the ipyleaflet plugin

dask-geomodeling comes with a ipyleaflet plugin for **'Jupyter<https://jupyter.org/>'_** so that you can show your generated views on a mapviewer. If you want to use it, install some additional dependencies:

```
$ conda [or pip] install jupyter ipyleaflet matplotlib pillow
```

And start your notebook server with the plugin:

```
$ jupyter notebook --NotebookApp.nbserver_extensions="{'dask_geomodeling.ipyleaflet_
↪plugin':True}"
```

Alternatively, you can add this extension to your **'Jupyter configuration<https://jupyter-notebook.readthedocs.io/en/stable/config_overview.html>'_**

2.1.5 Advanced: local setup with system Python (Ubuntu)

These instructions make use of the system-wide Python 3 interpreter.

    $ sudo apt install python3-pip python3-gdal

Install dask-geomodeling:

```
$ pip install --user dask-geomodeling[test,cityhash]
```

Run the tests:

```
$ pytest
```

2.1.6 Advanced: local setup for development (Ubuntu)

These instructions assume that `git`, `python3`, `pip`, and `virtualenv` are installed on your host machine.

Clone the dask-geomodeling repository:

```
$ git clone https://github.com/nens/dask-geomodeling
```

Make sure you have the GDAL libraries installed. On Ubuntu:

```
$ sudo apt install libgdal-dev
```

Take note of the GDAL version:

```
$ apt show libgdal-dev
```

Create and activate a virtualenv:

```
$ cd dask-geomodeling
$ virtualenv --python=python3 .venv
$ source .venv/bin/activate
```

Install PyGDAL with the correct version (example assumes GDAL 2.2.3):

```
(.venv) $ pip install pygdal==2.2.3.*
```

Install dask-geomodeling:

```
(.venv) $ pip install -e .[test,cityhash]
```

Run the tests:

```
(.venv) $ pytest
```

2.2 Quickstart

2.2.1 Constructing a view

A dask-geomodeling view can be constructed by creating a Block instance:

```
from dask_geomodeling.raster import RasterFileSource
source = RasterFileSource('/path/to/geotiff')
```

The view can now be used to obtain data from the specified file. More complex views can be created by nesting block instances:

```
from dask_geomodeling.raster import Smooth
smoothed = Smooth(source, 5)
smoothed_plus_two = smoothed + 2
```

2.2.2 Obtaining data from a view

To obtain data from a view directly, use the *get_data* method:

```
request = {
    "mode": "vals",
    "bbox": (138000, 480000, 139000, 481000),
    "projection": "epsg:28992",
    "width": 256,
    "height": 256
}
data = add.get_data(**request)
```

Which field to include in the request and what data to expect depends on the type of the block used. In this example, we used a RasterBlock. The request and response specifications are listed in the documentation of the specific block type.

2.2.3 Showing data on the map

If you a are using Jupyter and our ipyleaflet plugin (see **'installation'__**), you can inspect your dask-geomodeling View on an interactive map widget.

```
from ipyleaflet import Map, basemaps, basemap_to_tiles
from dask_geomodeling.ipyleaflet_plugin import GeomodelingLayer

# create the geomodeling layer and the background layer
# the 'styles' parameter refers to a matplotlib colormap;
# the 'vmin' and 'vmax' parameters determine the range of the colormap
geomodeling_layer = GeomodelingLayer(
    add, styles="viridis", vmin=0, vmax=10, opacity=0.5
)
osm_layer = basemap_to_tiles(basemaps.OpenStreetMap.Mapnik)

# center the map on the middle of the View's extent
extent = add.extent
Map(
    center=((extent[1] + extent[3]) / 2, (extent[0] + extent[2]) / 2),
    zoom=14,
    layers=[osm_layer, geoomdeling_layer]
)
```

Please consult the **'ipyleaflet<https://ipyleaflet.readthedocs.io>'_** docs for examples in how to add different basemaps, other layers, or add controls.

2.2.4 Delayed evaluation

Dask-geomodeling revolves around *lazy data evaluation*. Each Block first evaluates what needs to be done for certain request, storing that in a *compute graph*. This graph can then be evaluated to obtain the data. The data is evaluated with dask, and the specification of the compute graph also comes from dask. For more information about how a graph works, consult the dask documentation:

We use the previous example to demonstrate how this works:

```
import dask
request = {
    "mode": "vals",
    "bbox": (138000, 480000, 139000, 481000),
    "projection": "epsg:28992",
```

(continues on next page)

(continued from previous page)

```
    "width": 256,
    "height": 256
}
graph, name = add.get_compute_graph(**request)
data = dask.get(graph, [name])
```

Here, we first generate a compute graph using dask-geomodeling, then evaluate the graph using dask. The power of this two-step procedure is twofold:

1. Dask supports threaded, multiprocessing, and distributed schedulers. Consult the dask documentation to try these out.
2. The *name* is a unique identifier of this computation: this can easily be used in caching methods.

2.3 Views

A View is a combination of one or more Blocks. For instance:

```
from dask_geomodeling.raster import RasterFileSource, Group
source_1 = RasterFileSource("path/to/some/tiff")
source_2 = RasterFileSource("path/to/another/tiff")
view = Group(source_1, source_1)
```

2.3.1 View serialization

A View consists of several Block instances that reference each other. To serialize this we use Dask's graph format. This graph format replaces the nested structure by a flat dictionary with internal references

Warning: A serialized view looks much alike a compute graph. Don't get confused!

An example using the above view definition:

```
serialized_view = view.serialize()
```

```
{
  "version": 2,
  "graph": {
    "RasterFileSource_300b56b278d49bea13eff68f8cf52f90": [
      "dask_geomodeling.raster.sources.RasterFileSource",
      "file:///path/to/some/tiff"
    ],
    "RasterFileSource_9dc43c9bd98e2f9069e2b0c879d76cb1": [
      "dask_geomodeling.raster.sources.RasterFileSource",
      "file:///path/to/another/tiff"
    ],
    "Group_0d0a99fe65bffe87dd045627c27bcbbb": [
      "dask_geomodeling.raster.combine.Group",
      "RasterFileSource_300b56b278d49bea13eff68f8cf52f90",
      "RasterFileSource_9dc43c9bd98e2f9069e2b0c879d76cb1"
    ]
  },
```

(continues on next page)

(continued from previous page)

```
  "name": "Group_0d0a99fe65bffe87dd045627c27bcbbb"
}
```

The above "view graph" contains all three operations that we defined together with their arguments. The names are automatically generated and contain a hash which is useful to uniquely determine the block. Also, we see the "name", that points to the endpoint block.

To deserialize the view:

```
from dask_geomodeling.core import Block
view = Block.deserialize(serialized_view)
```

The methods `Block.to_json` and `Block.from_json`, or `Block.get_graph` and `dask_geomodeling.construct` serve the same purpose, but they in/output different object types.

2.4 Blocks

2.4.1 The Block class

To write a new `Block` subclass, we need to write the following:

1. the `__init__` that validates the arguments when constructing the block
2. the `get_sources_and_requests` that processes the request
3. the `process` that processes the data
4. a number of attributes such as `extent` and `period`

About the 2-step data processing

The `get_sources_and_requests` method of any block is called recursively from `get_compute_graph` and feeds the request from the block to its sources. It does so by returning a list of (source, request) tuples. During the data evaluation each of these 2-tuples will be converted to a single data object which is supplied to the `process` function.

First, an example in words. We construct a View `add = RasterFileSource('path/to/geotiff') + 2.4` and ask it the following:

- give me a 256x256 raster at location (138000, 480000)

We do that by calling `get_data`, which calls `get_compute_graph`, which calls `get_sources_and_requests` on each block instance recursively.

First `add.get_sources_and_requests` would respond with the following:

- I will need a 256x256 raster at location (138000, 480000) from `RasterFileSource('/path/to/geotiff')`
- I will need 2.4

Then, on recursion, the `RasterFileSource.get_sources_and_requests` would respond:

- I will give you the 256x256 raster at location (138000, 480000)

These small subtasks get summarized in a compute graph, which is returned by `get_compute_graph`. Then `get_data` feeds that compute graph to dask.

Dask will evaluate this graph by calling the `process` methods on each block:

1. A raster is loaded using `RasterFileSource.process`
2. This, together with the number 2.4, is given to `Add.process`
3. The resulting raster is presented to the user.

Implementation example

As an example, we use a simplified Dilate block, which adds a buffer of 1 pixel around all pixels of given value:

```
class Dilate(RasterBlock):
    def __init__(self, source, value):
        assert isinstance(source, RasterBlock):
        value = float(value)
        super().__init__(source, value)

    @property
    def source(self):
        return self.args[0]

    @property
    def value(self):
        return self.args[1]

    def get_sources_and_requests(self, **request):
        new_request = expand_request_pixels(request, radius=1)
        return [(self.store, new_request), (self.value, None)]

    @staticmethod
    def process(data, values=None):
        # handle empty data cases
        if data is None or values is None or 'values' not in data:
            return data
        # perform the dilation
        original = data['values']
        dilated = original.copy()
        dilated[ndimage.binary_dilation(original == value)] = value
        dilated = dilated[:, 1:-1, 1:-1]
        return {'values': dilated, 'no_data_value': data['no_data_value']}

    @property
    def extent(self):
        return self.source.extent

    @property
    def period(self):
        return self.source.period
```

In this example, we see all the essentials of a Block implementation.

- The `__init__` checks the types of the provided arguments and calls the `super().__init__` that further initializes the block.
- The `get_sources_and_requests` expands the request with 1 pixel, so that dilation will have no edge effects. It returns two (source, request) tuples.
- The `process` (static)method takes the amount arguments equal to the length of the list that `get_sources_and_requests` produces. It does the actual work and returns a data response.
- Some attributes like `extent` and `period` need manual specification, as they might change through the block.

- The class derives from `RasterBlock`, which sets the type of block, and through that its request/response schema and its required attributes.

2.4.2 Block types specification

A block type sets three things:

1. the response schema: e.g. "RasterBlock.process returns a dictionary with a numpy array and a no data value"
2. the request schema: e.g. "RasterBlock.get_sources_and_requests expects a dictionary with the fields 'mode', 'bbox', 'projection', 'height', 'width'"
3. the attributes to be implemented on each block

This is not enforced at the code level, it is up to the developer to stick to this specification. The specification is written down in the type baseclass *`RasterBlock()`* or *`GeometryBlock()`*.

2.4.3 API specification

Module containing the core graphs.

class dask_geomodeling.core.graphs.**Block**(**args*)

A class that generates dask-like compute graphs for given requests.

Arguments (args) are always stored in `self.args`. If a request is passed into the Block using the `get_data` or (the lazy version) `get_compute_graph` method, the Block figures out what args are actually necessary to evaluate the request, and what requests need to be sent to those args. This happens in the method `get_sources_and_requests`.

After the requests have been evaluated, the data comes back and is passed into the `process` method.

classmethod deserialize(*val*, *validate=False*)

Deserialize this block from a dict containing version, graph and name

static from_import_path(*path*)

Deserialize the Block by importing it from given path.

classmethod from_json(*val*, ***kwargs*)

Construct a graph from a json stream.

get_compute_graph(*cached_compute_graph=None*, ***request*)

Lazy version of get_data, returns a compute graph dict, that can be evaluated with *compute* (or dask's get function).

The dictionary has keys in the form `name_token` and values in the form `tuple(process, *args)`, where `args` are the precise arguments that need to be passed to `process`, with the exception that args may reference to other keys in the dictionary.

get_data(***request*)

Directly evaluate the request and return the data.

get_graph(*serialize=False*)

Generate a graph that defines this Block and its dependencies in a dictionary.

The dictionary has keys in the form `name_token` and values in the form `tuple(Block class, *args)`, where `args` are the precise arguments that were used to construct the Block, with the exception that args may also reference other keys in the dictionary.

If serialize == True, the Block classes will be replaced by their corresponding import paths.

classmethod get_import_path()
Serialize the Block by returning its import path.

get_sources_and_requests(***request*)
Adapt the request and/or select the sources to be computed. The request is allowed to differ per source.

This function should return an iterable of (source, request). For sources that are no Block instance, the request is ignored.

Exceptions raised here will be raised before actual computation starts. (at .get_compute_graph(request)).

static process(*data*)
Overridden to modify data from sources in unlimited ways.

Default implementation passes single-source unaltered data.

serialize()
Serialize this block into a dict containing version, graph and name

to_json(***kwargs*)
Dump the graph to a json stream.

token
Generates a unique and deterministic representation of this object

dask_geomodeling.core.graphs.**construct**(*graph*, *name*, *validate=True*)
Construct a Block with dependent Blocks from a graph and endpoint name.

dask_geomodeling.core.graphs.**compute**(*graph*, *name*, **args*, ***kwargs*)
Compute a graph ({name: [func, arg1, arg2, ...]}) using dask.get_sync

2.5 Raster Blocks

Raster-type blocks contain rasters with a time axis. Internally, the raster data is stored as NumPy arrays.

2.5.1 API Specification

Module containing the raster block base classes.

class dask_geomodeling.raster.base.**RasterBlock**(**args*)
The base block for temporal rasters.

All raster blocks must be derived from this base class and must implement the following attributes:

- `period`: a tuple of datetimes
- `timedelta`: a datetime.timedelta (or None if nonequidistant)
- `extent`: a tuple `(x1, y1, x2, y2)`
- `dtype`: a numpy dtype object
- `fillvalue`: a number
- `geometry`: OGR Geometry
- `projection`: WKT string
- `geo_transform`: a tuple of 6 numbers

These attributes are `None` if the raster is empty.

A raster data request contains the following fields:

- mode: values (`'vals'`), time (`'time'`) or metadata (`'meta'`)
- bbox: bounding box `(x1, y1, x2, y2)`
- projection: wkt spatial reference
- width: data width
- height: data height
- start: start date as naive UTC datetime
- stop: stop date as naive UTC datetime

The data response contains the following:

- if mode was `'vals'`: a three dimensional array of shape `(bands, height, width)`
- if mode was `'time'`: a list of naive UTC datetimes corresponding to the time axis
- if mode was `'meta'`: a list of metadata values corresponding to the time axis

dask_geomodeling.raster.combine

Module containing raster blocks that combine rasters.

class dask_geomodeling.raster.combine.**Group**(**args*)
Combine multiple rasters into a single one.

Parameters **args** (*list of RasterBlock*) – multiple RasterBlocks to be combined.

Values at equal timesteps in the contributing rasters are pasted left to right. Therefore values from rasters that are more 'to the left' are shadowed by values from rasters more 'to the right'. However, 'no data' values are transparent and do not shadow underlying data values.

In the case of aligned equidistant time characteristics, the procedure will use slicing in the processing of the result for optimum performance.

dask_geomodeling.raster.elemwise

Module containing elementwise raster blocks.

class dask_geomodeling.raster.elemwise.**Add**(*a*, *b*)
Add a value.

Parameters

- **a** (RasterBlock, *scalar*) – Addition parameter a
- **b** (RasterBlock, *scalar*) – Addition parameter b

At least one of the parameters should be a RasterBlock. If the params are both RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**Subtract**(*a*, *b*)
Subtract a constant value from a store or vice versa.

Parameters

- **a** (RasterBlock, *scalar*) – Subtraction parameter a

- **b** (RasterBlock, *scalar*) – Subtraction parameter b

At least one of the parameters should be a RasterBlock. If the params are both RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**Multiply**(*a*, *b*)
Multiply by a value.

Parameters

- **a** (RasterBlock, *scalar*) – Multiplication parameter a
- **b** (RasterBlock, *scalar*) – Multiplication parameter b

At least one of the parameters should be a RasterBlock. If the params are both RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**Divide**(*a*, *b*)
Divide Store by a constant value or vice versa.

Parameters

- **a** (RasterBlock, *scalar*) – Division parameter a
- **b** (RasterBlock, *scalar*) – Division parameter b

At least one of the parameters should be a RasterBlock. If the params are both RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**Power**(*a*, *b*)
Raise each number in a to the power b.

Parameters

- **a** (RasterBlock, *scalar*) – base
- **b** (RasterBlock, *scalar*) – exponent

At least one of the parameters should be a RasterBlock. If the params are both RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**FillNoData**(**args*)
Combines multiple rasters, filling in nodata values.

Parameters **args** (*list of RasterBlock*) – list of raster sources to be combined.

Values at equal timesteps in the contributing rasters are pasted left to right. Therefore values from rasters that are more 'to the left' are shadowed by values from rasters more 'to the right'. However, 'no data' values are transparent and do not shadow underlying data values.

class dask_geomodeling.raster.elemwise.**Equal**(*a*, *b*)
Compares the values of two stores and returns True if they are equal.

Note that "no data" is not equal to "no data".

Parameters

- **a** (RasterBlock, *scalar*) – Comparison parameter a
- **b** (RasterBlock, *scalar*) – Comparison parameter b

At least one of the parameters should be a RasterBlock. If the params are both RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**NotEqual**(*a*, *b*)
Compares the values of two stores and returns False if they are equal.

Note that "no data" is not equal to "no data".

Parameters

- **a** (RasterBlock, *scalar*) – Comparison parameter a
- **b** (RasterBlock, *scalar*) – Comparison parameter b

At least one of the parameters should be a RasterBlock. If the params are both RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**Greater**(*a*, *b*)
Returns True if a is greater than b.

Note that "no data" will always return False

Parameters

- **a** (RasterBlock, *scalar*) – Comparison parameter a
- **b** (RasterBlock, *scalar*) – Comparison parameter b

At least one of the parameters should be a RasterBlock. If the params are both RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**GreaterEqual**(*a*, *b*)
Returns True if a is greater than or equal to b.

Note that "no data" will always return False

Parameters

- **a** (RasterBlock, *scalar*) – Comparison parameter a
- **b** (RasterBlock, *scalar*) – Comparison parameter b

At least one of the parameters should be a RasterBlock. If the params are both RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**Less**(*a*, *b*)
Returns True if a is less than b.

Note that "no data" will always return False

Parameters

- **a** (RasterBlock, *scalar*) – Comparison parameter a
- **b** (RasterBlock, *scalar*) – Comparison parameter b

At least one of the parameters should be a RasterBlock. If the params are both RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**LessEqual**(*a*, *b*)
Returns True if a is less than or equal to b.

Note that "no data" will always return False

Parameters

- **a** (RasterBlock, *scalar*) – Comparison parameter a
- **b** (RasterBlock, *scalar*) – Comparison parameter b

At least one of the parameters should be a RasterBlock. If the params are both RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**Invert**(*x*)
Swaps False and True ("not x" or "~x").

Parameters **x** (RasterBlock) – raster data to invert

class dask_geomodeling.raster.elemwise.**And**(*a*, *b*)
Returns True where a and b are True.

Parameters

- **a** (RasterBlock, *boolean*) – Comparison parameter a
- **b** (RasterBlock, *boolean*) – Comparison parameter b

All input parameters should have a boolean dtype and at least one of the parameters should be a RasterBlock. If both are RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**Or**(*a*, *b*)
Returns True where a or b are True.

Parameters

- **a** (RasterBlock, *boolean*) – Comparison parameter a
- **b** (RasterBlock, *boolean*) – Comparison parameter b

All input parameters should have a boolean dtype and at least one of the parameters should be a RasterBlock. If both are RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**Xor**(*a*, *b*)
Returns True where either a or b is True (exclusive-or)

Parameters

- **a** (RasterBlock, *boolean*) – Comparison parameter a
- **b** (RasterBlock, *boolean*) – Comparison parameter b

All input parameters should have a boolean dtype and at least one of the parameters should be a RasterBlock. If both are RasterBlocks, they should share exactly the same time structure. The Snap block can be used to accomplish this.

class dask_geomodeling.raster.elemwise.**IsData**(*store*)
Returns True where raster has data.

Parameters **store** (RasterBlock) –

class dask_geomodeling.raster.elemwise.**IsNoData**(*store*)
Returns True where raster has no data.

Parameters **store** (RasterBlock) –

dask_geomodeling.raster.misc

Module containing miscellaneous raster blocks.

class dask_geomodeling.raster.misc.**Clip**(*store*, *source*)
Make result values 'no data' if it is 'no data' in the source. If source is a boolean mask, False values become 'no data'.

Parameters

- **store** (RasterBlock) – The store-like whose values are to be converted.

- **source** (RasterBlock) – The store-like as the source for 'data or no data'

extent
Intersection of bounding boxes of 'store' and 'source'.

geometry
Intersection of geometries of 'store' and 'source'.

class dask_geomodeling.raster.misc.**Classify**(*store*, *bins*, *right=False*)
Classify raster data into a binned categories

Parameters

- **store** (RasterBlock) – rasterdata to classify
- **bins** (*list*) – a 1-dimensional and monotonic list of bin edges
- **right** (*boolean*) – whether the intervals include the right or the left bin edge

See also: https://docs.scipy.org/doc/numpy/reference/generated/numpy.digitize.html

class dask_geomodeling.raster.misc.**Reclassify**(*store*, *data*, *select=False*)
Reclassify integer data to integers or floats.

Parameters

- **store** (RasterBlock) – rasterdata to reclassify
- **data** (*list*) – list of (from, to) values defining the reclassification the from values can be of bool or int datatype; the to values can be of int or float datatype
- **select** (*bool*) – leave only reclassified values, set others to 'no data'. Default False.

class dask_geomodeling.raster.misc.**Mask**(*store*, *value*)
Convert 'data' values to a single constant value.

Parameters

- **store** (*Store*) – The store whose values are to be converted.
- **value** (*number*) – The constant value to be given to 'data' values.

class dask_geomodeling.raster.misc.**MaskBelow**(*store*, *value*)
Mask data below some value.

Parameters

- **store** (*Store*) – The store whose values are to be masked.
- **value** (*number*) – The threshold value. Values below this will be masked.

class dask_geomodeling.raster.misc.**Step**(*store*, *left=0*, *right=1*, *location=0*, *at=None*)
Block for a constant step function with one discontinuity.

The step function is defined as: - left if x < location - at if x == location - right if x > location

Parameters

- **store** (RasterBlock) – The raster whose values are the input to the step function.
- **left** (*number*) – value left of the discontinuity
- **right** (*number*) – value right of the discontinuity
- **location** (*number*) – location of the discontinuity
- **at** (*number*) – value at the discontinuity

class dask_geomodeling.raster.misc.**Rasterize**(*source*, *column_name=None*, *dtype=None*, *limit=None*)

Converts geometry source to raster

Parameters

- **source** (GeometryBlock) – geometry source
- **column_name** (*string*) – column from the geometry source to rasterize. If column_name is not provided, a boolean raster will be returned indicating where there are geometries.
- **dtype** (*string*) – a numpy datatype specification to return the array. Defaults to 'int32' if column_name is not, else it defaults to 'bool'.
- **limit** (*int*) – the maximum number of geometries. Defaults to the geomodeling.goemetry-limit setting.

Returns a raster containing values from 'column_name' or True/False.

To rasterize floating point values, it is necessary to pass dtype='float'.

See also: https://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html

The global geometry-limit setting can be adapted as follows:

```
>>> from dask import config
>>> config.set({"geomodeling.geometry-limit": 100000})
```

dask_geomodeling.raster.sources

Module containing raster sources.

class dask_geomodeling.raster.sources.**MemorySource**(*data*, *no_data_value*, *projection*, *pixel_size*, *pixel_origin*, *time_first=0*, *time_delta=None*, *metadata=None*)

A raster source that interfaces data from memory.

Nodata values are supported, but when upsampling the data, these are assumed to be 0 biasing data edges towards 0.

Parameters

- **data** (*number or ndarray*) – the pixel values this value will be transformed in a 3D array (t, y, x)
- **no_data_value** (*number*) – the pixel value that designates 'no data'
- **projection** (*str*) – the projection of the given pixel values
- **pixel_size** (*float or length-2 iterable of floats*) – the size of one pixel (in units given by projection) if x and y pixel sizes differ, provide them in (x, y) order
- **pixel_origin** (*length-2 iterable of floats*) – the location (x, y) of pixel with index (0, 0)
- **time_first** (*integer or naive datetime*) – the timestamp of the first frame in data (in milliseconds since 1-1-1970)
- **time_delta** (*integer or timedelta or NoneType*) – the difference between two consecutive frames (in ms)

- **metadata** (*list or NoneType*) – a list of metadata corresponding to the input frames

class dask_geomodeling.raster.sources.**RasterFileSource**(*url*, *time_first=0*, *time_delta=300000*)

A raster source that interfaces data from a file path.

Parameters

- **url** (*str*) – the path to the file. File paths have to be contained inside the current root setting. Relative paths are interpreted relative to this setting (but internally stored as absolute paths).
- **time_first** (*integer or datetime*) – the timestamp of the first frame in data (in milliseconds since 1-1-1970), defaults to 1-1-1970
- **time_delta** (*integer or timedelta*) – the difference between two consecutive frames (in ms), defaults to 5 minutes

The global root path can be adapted as follows:

```
>>> from dask import config
>>> config.set({"geomodeling.root": "/my/data/path"})
```

Note that this object keeps a file handle open. If you need to close the file handle, call block.close_dataset (or dereference the whole object).

dask_geomodeling.raster.spatial

Module containing raster blocks for spatial operations.

class dask_geomodeling.raster.spatial.**Dilate**(*store*, *values*)

Perform binary dilation on specific values. Dilation is done in the order of the values parameter.

Parameters

- **store** (RasterBlock) – raster to perform dilation on
- **values** (*list*) – only dilate pixels that have these values

class dask_geomodeling.raster.spatial.**Smooth**(*store*, *size*, *fill=0*)

Smooth the values from a raster spatially using gaussian smoothing.

Parameters

- **store** (*Store*) – raster to smooth
- **size** (*scalar*) – size of the smoothing in meters. the 'sigma' of the gaussian kernel equals `size / 3`.
- **fill** (*scalar*) – fill value to be used for 'no data' values during smoothing. The output will not have 'no data' values.

The challenge is to mitigate the edge effects whilst remaining performant. If the necessary margin for smoothing is more than 6 pixels in any direction, the approach used here is requesting a zoomed out geometry, smooth that and zoom back in to the original region of interest.

class dask_geomodeling.raster.spatial.**MovingMax**(*store*, *size*)

Apply a spatial maximum filter to the data using a circular footprint.

Parameters

- **store** (*int*) – raster to apply the maximum filter on

- **size** – diameter of the circular footprint in pixels

This block can be used for visualization of sparse data.

class dask_geomodeling.raster.spatial.**HillShade**(*store*, *altitude=45*, *azimuth=315*, *fill=0*)

Calculate a hillshade from the raster values.

Parameters

- **store** (RasterBlock) – RasterBlock object
- **size** (*scalar*) – size of the effect, in projected units
- **altitude** (*scalar*) – Light source altitude in degrees. Default 45.
- **azimuth** (*scalar*) – Light source azimuth in degrees. Default 315.
- **fill** (*scalar*) – fill value to be used for 'no data' values during hillshading

Hillshade, adapted from gdal algorithm. Smooths prior to shading to keep it good looking even when zoomed beyond 1:1.

dask_geomodeling.raster.temporal

Module containing raster blocks for temporal operations.

class dask_geomodeling.raster.temporal.**Snap**(*store*, *index*)

Snap the time structure of a raster to that of another raster.

Parameters

- **store** (RasterBlock) – Return values from this block.
- **index** (RasterBlock) – Snap values to the times from this block.

On get_data, get_meta or get_time requests, it will return the data and meta from the block supplied as the store parameter, but apparently have the time structure of the Store supplied as the index parameter.

In contrast to the group block, this block does not take advantage of aligned bands in any way. Therefore the perfomance will be noticeably worse, in particular when requesting long timeslices.

class dask_geomodeling.raster.temporal.**Shift**(*store*, *time*)

Shift the the store by some timedelta.

Parameters

- **store** (*Store*) – The source whose time is to be modified.
- **time** (*integer*) – The time to shift the store, in milliseconds.

Modifies the source's properties and queries it such that its data appears to be shifted towards the future in case of a positive time parameter.

class dask_geomodeling.raster.temporal.**TemporalAggregate**(*source*, *frequency*, *statistic='sum'*, *closed=None*, *label=None*, *timezone='UTC'*)

Geoblock that resamples rasters in time.

Parameters

- **source** (RasterBlock) – The source whose time is to be modified.

- **frequency** (*string or NoneType*) – the frequency to resample to, as pandas offset string if this value is None, this block will return the temporal statistic over the complete time range, with output timestamp at the end of the source's period.
- **statistic** (*string*) – the type of statistic to perform. Can be `'sum'`, `'count'`, `'min'`, `'max'`, `'mean'`, `'median'`, `'p<percentile>'`
- **closed** (*string or NoneType*) – {`None`, `'left'`, `'right'`}. Determines what side of the interval is closed when resampling.
- **label** (*string or NoneType*) – {`None`, `'left'`, `'right'`}. Determines what side of the interval is used as output datetime.
- **timezone** (*string*) – timezone to perform the resampling in

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.resample.html https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-objects

class dask_geomodeling.raster.temporal.**Cumulative**(*source*, *statistic='sum'*, *frequency=None*, *timezone='UTC'*)

Geoblock that computes a cumulative of a raster over time.

Parameters

- **source** (RasterBlock) – The source whose time is to be modified.
- **statistic** (*string*) – the type of statistic to perform. Can be `'sum'`, `'count'`
- **frequency** (*string or NoneType*) – the frequency at which to restart the cumulative. if this value is None, the cumulative will continue indefinitely
- **timezone** (*string*) – timezone to restart the cumulative

See also: https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-objects

2.6 Geometry and Series Blocks

Geometry-type blocks contain sets of geometries, optionally with `'start'` and `'end'` fields and other properties. Internally, geometry data is stored in GeoPandas GeoDataframes.

2.6.1 API Specification

Module containing the base geometry block classes.

class dask_geomodeling.geometry.base.**GeometryBlock**(**args*)

The base block for geometries

All geometry blocks must be derived from this base class and must implement the following attributes:

- `columns`: a set of column names to expect in the dataframe

A geometry request contains the following fields:

- mode: `'intersects'` or `'extent'`. default `'intersects'`.
- geometry: limit returned objects to objects that intersect with this shapely geometry object
- projection: projection to return the geometries in as WKT string
- limit: the maximum number of geometries

- min_size: geometries with a bbox that is smaller than this on all sides are left out
- start: start date as UTC datetime
- stop: stop date as UTC datetime
- filters: dict of Django ORM-like filters on properties (e.g. `id=598`)

The data response contains the following:

- if mode was `'intersects'`: a DataFrame of features with properties
- if mode was `'extent'`: the bbox that contains all features

To be able to perform operations on properties, there is a helper type called''SeriesBlock''. This is the block equivalent of a `pandas.Series`. You can get a `SeriesBlock` from a `GeometryBlock`, perform operations on it, and set it back into a `GeometryBlock`.

class dask_geomodeling.geometry.base.**SeriesBlock**(**args*)
A helper block for GeometryBlocks, representing one single field

class dask_geomodeling.geometry.base.**GetSeriesBlock**(*source*, *name*)
Get a column from a GeometryBlock.

Parameters

- **source** (GeometryBlock) – GeometryBlock
- **name** (*string*) – name of the column to get

Returns SeriesBlock containing the property column

class dask_geomodeling.geometry.base.**SetSeriesBlock**(*source*, *column*, *value*, **args*)
Set one or multiple columns (SeriesBlocks) in a GeometryBlock.

Parameters

- **source** (GeometryBlock) – source to add the extra columns to
- **column** (*string*) – name of the column to be set
- **value** (SeriesBlock, *scalar*) – series or constant value to set
- **args** – string, SeriesBlock, . . . , repeated multiple times

Returns the source GeometryBlock with additional property columns

Example:

```
>>> SetSeriesBlock(view, 'column_1', series_1, 'column_2', series_2)
```

dask_geomodeling.geometry.aggregate

Module containing raster blocks that aggregate rasters.

class dask_geomodeling.geometry.aggregate.**AggregateRaster**(*source*, *raster*, *statistic='sum'*, *projection=None*, *pixel_size=None*, *max_pixels=None*, *column_name='agg'*, *auto_pixel_size=False*, **args*)
Compute zonal statistics and add them to the geometry properties

Parameters

- **source** (GeometryBlock) – the source of geometry data
- **raster** (RasterBlock) – the source of raster data
- **statistic** (*string*) – the type of statistic to perform. can be `'sum'`, `'count'`, `'min'`, `'max'`, `'mean'`, `'median'`, `'p<percentile>'`.
- **projection** (*string or None*) – the projection to perform the aggregation in
- **pixel_size** (*float or None*) – the pixel size to perform aggregation in
- **max_pixels** (*int or None*) – the maximum number of pixels to use for aggregation. defaults to the geomodeling.raster-limit setting.
- **column_name** (*string*) – the name of the column to output the results
- **auto_pixel_size** (*boolean*) – determines whether the pixel_size is adjusted when a raster is too large. Default False.

Returns GeometryBlock with aggregation results in `column_name`

The currently implemented statistics are sum, count, min, max, mean, median, and percentile. If projection or max_resolution are not given, these are taken from the provided RasterBlock.

The count statistic calculates the number of active cells in the raster. A percentile statistic can be selected using text value starting with 'p' followed by something that can be parsed as a float value, for example `'p33.3'`.

Only geometries that intersect the requested bbox are aggregated. Aggregation is done in a specified projection and with a specified pixel size.

Should the combination of the requested pixel_size and the extent of the source geometry cause the requested raster size to exceed max_pixels, the pixel_size is adjusted automatically if `auto_pixel_size = True`, else a RuntimeError is raised.

The global raster-limit setting can be adapted as follows:

```
>>> from dask import config
>>> config.set({"geomodeling.raster-limit": 10 ** 9})
```

class dask_geomodeling.geometry.aggregate.**AggregateRasterAboveThreshold**(*source*, *raster*, *statistic='sum'*, *projection=None*, *pixel_size=None*, *max_pixels=None*, *column_name='agg'*, *auto_pixel_size=False*, *threshold_name=None*)

Aggregate raster values ignoring values below some threshold. The thresholds are supplied per geometry.

Parameters

- **source** (GeometryBlock) – the source of geometry data
- **raster** (RasterBlock) – the source of raster data

- **statistic** (*string*) – the type of statistic to perform. can be `'sum'`, `'count'`, `'min'`, `'max'`, `'mean'`, `'median'`, `'p<percentile>'`.
- **projection** (*string*) – the projection to perform the aggregation in
- **pixel_size** (*float*) – the pixel size to perform aggregation in
- **max_pixels** (*int*) – the maximum number of pixels to use for aggregation
- **column_name** (*string*) – the name of the column to output the results
- **auto_pixel_size** (*boolean*) – determines whether the pixel_size is adjusted when a raster is too large. Default False.
- **threshold_name** (*string*) – the name of the column with the thresholds

Returns GeometryBlock with aggregation results in `column_name`

See also: *dask_geomodeling.geometry.aggregate.AggregateRaster*

dask_geomodeling.geometry.constructive

Module containing geometry block constructive operations

class dask_geomodeling.geometry.constructive.**Buffer**(*source*, *distance*, *projection*, *resolution=16*)

Buffer geometries.

Parameters

- **source** (GeometryBlock) – the geometry source
- **distance** (*float*) – a distance measure in the given projection.
- **projection** (*string*) – an EPSG or WKT string, e.g. EPSG:28992.
- **resolution** (*int*) – quarter circle segments. Default is 16.

distance
Buffer distance.

The unit (e.g. m, °) is determined by the projection.

projection
Projection used for buffering.

resolution
Buffer resolution.

class dask_geomodeling.geometry.constructive.**Simplify**(*source*, *tolerance=None*, *preserve_topology=True*)

Simplify geometries up to given tolerance.

Parameters

- **source** (GeometryBlock) – the geometry source
- **tolerance** (*float*) – the simplification tolerance. if no tolerance is given, the `min_size` request param is used.
- **preserve_topology** (*boolean*) – whether to preserve topology. Default True.

dask_geomodeling.geometry.field_operations

Module containing geometry block operations that act on non-geometry fields

class dask_geomodeling.geometry.field_operations.**Classify**(*source*, *bins*, *labels*, *right=True*)

Classify a continuous-valued property into binned categories

Parameters

- **source** (SeriesBlock) – source data to classify
- **bins** (*list*) – a 1-dimensional and monotonic list of bins. How values outside of the bins are classified, depends on the length of the labels. If len(labels) = len(bins) - 1, then values outside of the bins are classified to NaN. If len(labels) = len(bins) + 1, then values outside of the bins are classified to the first and last elements of the labels list.
- **labels** (*list*) – the labels for the returned bins
- **right** (*boolean*) – whether the intervals include the right or the left bin edge

See also: https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.cut.html

class dask_geomodeling.geometry.field_operations.**ClassifyFromColumns**(*source*, *value_column*, *bin_columns*, *labels*, *right=True*)

Classify a continuous-valued property based on bins located in different columns.

Parameters

- **source** (GeometryBlock) – geometry source to classify
- **value_column** (*string*) – the column name that contains values to classify
- **bin_columns** (*list*) – column names in which the bins are stored. The bins values need to be sorted in increasing order.
- **labels** (*list*) – specifies the labels for the returned bins
- **right** (*boolean*) – whether the intervals include the right or the left bin edge Default True.

See also: *dask_geomodeling.geometry.field_operations.Classify*

class dask_geomodeling.geometry.field_operations.**Add**(*source*, *other*)

Addition of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.add.html

class dask_geomodeling.geometry.field_operations.**Subtract**(*source*, *other*)

Subtraction of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.subtract.html

class dask_geomodeling.geometry.field_operations.**Multiply**(*source*, *other*)
Multiplication of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.multiply.html

class dask_geomodeling.geometry.field_operations.**Divide**(*source*, *other*)
Floating division of series and other, element-wise.

Putting source in the divisor is not possible: please use the Power for that instead.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.divide.html

class dask_geomodeling.geometry.field_operations.**FloorDivide**(*source*, *other*)
Integer (floor) division of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.floordiv.html

class dask_geomodeling.geometry.field_operations.**Power**(*source*, *other*)
Power (exponent) of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.pow.html

class dask_geomodeling.geometry.field_operations.**Modulo**(*source*, *other*)
Modulo of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.mod.html

class dask_geomodeling.geometry.field_operations.**Equal**(*source*, *other*)
Equal to of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.eq.html

class dask_geomodeling.geometry.field_operations.**NotEqual**(*source*, *other*)
Not equal to of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.ne.html

class dask_geomodeling.geometry.field_operations.**Greater**(*source*, *other*)
Greater than of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.gt.html

class dask_geomodeling.geometry.field_operations.**GreaterEqual**(*source*, *other*)
Greater than or equal to of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.ge.html

class dask_geomodeling.geometry.field_operations.**Less**(*source*, *other*)
Less than of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.lt.html

class dask_geomodeling.geometry.field_operations.**LessEqual**(*source*, *other*)
Less than or equal to of series and other, element-wise.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.le.html

class dask_geomodeling.geometry.field_operations.**And**(*source*, *other*)
Logical AND between series and other.

class dask_geomodeling.geometry.field_operations.**Or**(*source*, *other*)
Logical OR between series and other.

class dask_geomodeling.geometry.field_operations.**Xor**(*source*, *other*)
Logical XOR between series and other.

class dask_geomodeling.geometry.field_operations.**Invert**(*source*, **args*)
Logical NOT operation on a series.

class dask_geomodeling.geometry.field_operations.**Where**(*source*, *cond*, *other*)
Replace values where the condition is False.

Parameters

- **source** (SeriesBlock) – source data
- **cond** (SeriesBlock) – condition that determines whether to keep values from source
- **other** (SeriesBlock, *scalar*) – entries where cond is False are replaced with the corresponding value from other.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.where.html

class dask_geomodeling.geometry.field_operations.**Mask**(*source*, *cond*, *other*)
Replace values where the condition is True.

Parameters

- **source** (SeriesBlock) – source data
- **cond** (SeriesBlock) – condition that determines whether to mask values from source

- **other** (SeriesBlock, *scalar*) – entries where cond is True are replaced with the corresponding value from other.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.mask.html

class dask_geomodeling.geometry.field_operations.**Round**(*source*, *decimals=0*)
Round each value in a SeriesBlock to the given number of decimals

Parameters

- **source** (SeriesBlock) – source data
- **decimals** (*int*) – number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

See also: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.round.html

dask_geomodeling.geometry.geom_operations

Module containing operations that return series from geometry fields

class dask_geomodeling.geometry.geom_operations.**Area**(*source*, *projection*)
Block that calculates the area of geometries.

Parameters

- **source** (GeometryBlock) – geometry data
- **projection** (*string*) – projection as EPSG or WKT string to compute area in

Returns SeriesBlock with only the computed area

dask_geomodeling.geometry.merge

Module containing merge operation that act on geometry blocks

class dask_geomodeling.geometry.merge.**MergeGeometryBlocks**(*left*, *right*, *how='inner'*, *suffixes=('', '_right')*)
Merge two GeometryBlocks into one by index

Parameters

- **left** (GeometryBlock) – left geometry data to merge
- **right** (GeometryBlock) – right geometry data to merge
- **how** (*string*) – type of merge to be performed. One of `'left'`, `'right'`, `'outer'`, `'inner'`. Default `'inner'`.
- **suffixes** (*tuple*) – suffix to apply to overlapping column names in the left and right side, respectively. Default `('', '_right')`.

See also merge: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.merge.html

dask_geomodeling.geometry.parallelize

Module containing blocks that parallelize non-geometry fields

class dask_geomodeling.geometry.parallelize.**GeometryTiler**(*source*, *size*, *projection*)

Parallelize operations on a GeometryBlock by tiling the request

Parameters

- **source** (GeometryBlock) – GeometryBlock
- **size** (*float*) – the max size of a tile in units of the projection
- **projection** (*string*) – the projection as EPSG or WKT string in which to compute tiles

Only supports 'centroid' and 'extent' request modes.

dask_geomodeling.geometry.set_operations

Module containing geometry block set operations

class dask_geomodeling.geometry.set_operations.**Difference**(*source*, *other*)

Block that calculates the difference of two GeometryBlocks.

The resulting GeometryBlock will have all geometries in `source`, and if there are geometries with the same ID in `other`, the geometries will be adapted using the Difference operation.

class dask_geomodeling.geometry.set_operations.**Intersection**(*source*, *other=None*)

Block that intersects geometries with the requested geometry.

Parameters **source** (GeometryBlock) – the source of geometry data

dask_geomodeling.geometry.sources

Module containing geometry sources.

class dask_geomodeling.geometry.sources.**GeometryFileSource**(*url*, *layer=None*, *id_field='id'*)

A geometry source that opens a geometry file from disk.

Parameters

- **url** – URL to the file. File paths have to be contained inside the current root setting. Relative paths are interpreted relative to this setting but internally stored as absolute paths).
- **layer** (*string*) – the layer_name in the json to use as source. If None, the first layer is used.
- **id_field** (*string*) – the field name to use as unique ID. Default `'id'`.

The input of these blocks is by default limited to 10000 geometries.

Relevant settings can be adapted as follows:

```
>>> from dask import config
>>> config.set({"geomodeling.root": '/my/data/path'})
>>> config.set({"geomodeling.geometry-limit": 100000})
```

dask_geomodeling.geometry.text

Module containing text column operations that act on geometry blocks

class dask_geomodeling.geometry.text.**ParseTextColumn**(*source*, *source_column*, *key_mapping*)

Parses a text column into (possibly multiple) value columns.

Key, value pairs need to be separated by an equal (=) sign.

Parameters

- **source** (GeometryBlock) – data source
- **source_column** (*string*) – existing column in source.
- **key_mapping** (*dict*) – mapping containing pairs {key_name: column_name}: key_name: existing key in the text to be parsed. column_name: name of the new column created that contains the parsed value.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

O

P

R

S

T

W

X