# dask-geomodeling

Casper van der Wel

Jan 17, 2024

# CONTENTS:


**1 About** **3**

**2 Contents** **5**

2.1 Installation . . . . . 5
2.2 Quickstart . . . . . 8
2.3 Views . . . . . 9
2.4 Blocks . . . . . 11
2.5 Raster Blocks . . . . . 14
2.6 Geometry and Series Blocks . . . . . 28

**3 Indices and tables** **43**

**Python Module Index** **45**

**Index** **47**

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 vector data, respectively through the baseclasses *RasterBlock()* and *GeometryBlock()*. Derived classes support operations such as grouping, basic math, shifting time, smoothing, reclassification, geometry operations, zonal statistics, and property field operations.

CHAPTER

ONE

# 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

TWO

# 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. 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.

### 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 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

### 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
```

### 2.1.7 Advanced: Running tests in VSCode for WSL2 (Ubuntu)

It is possible to run the tests (that reside in WSL2) but perform debugging in VSCode (Windows)

1 Install the Python extension in VSCode.

2 Open the `Test Explorer View` (beaker icon)

3 Press the `Configure Tests` button. Select `pytest` as test framework, and base the configuration on the existing `setup.cfg`

4 The tests should now be discovered, and by pressing the `Debug Tests` button, it is now possible to place breakpoints and step through the tests.

See the VSCode manual for python testing for explanation regarding running the tests.

A known issue can be found on StackOverflow.

## 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, 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
```

(continues on next page)

(continued from previous page)

```
Map(
    center=((extent[1] + extent[3]) / 2, (extent[0] + extent[2]) / 2),
    zoom=14,
    layers=[osm_layer, geoomdeling_layer]
)
```

Please consult the ipyleaflet 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",
    "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"
    ]
  },
  "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)
```

(continues on next page)

(continued from previous page)

```
    @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.

**property token**

Generates a unique and deterministic representation of this object

dask_geomodeling.core.graphs.**compute**(*graph*, *name*, **args*, ***kwargs*)

Compute a graph ({name: [func, arg1, arg2, ...]}) using the configured scheduler. See dask.config.

dask_geomodeling.core.graphs.**construct**(*graph*, *name*, *validate=True*)

Construct a Block with dependent Blocks from a graph and endpoint name.

## 2.5 Raster Blocks

RasterBlocks are the main component of raster operations. Most raster operations take one or more RasterBlocks as input and produce a single RasterBlock as output.

Raster-type blocks contain rasters with data in three dimensions. Besides the x- and y-axes they also have a temporal axis.

Internally, dask-geomodeling stores the raster data as NumPy arrays.

### 2.5.1 API Specification

Module containing the RasterBlock base classes.

**class** dask_geomodeling.raster.base.**RasterBlock**(**args*)

The base block for temporal rasters.

All RasterBlocks 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 is `None` or a dictionary with the following fields:

- (if mode was `"vals"`) `"values"`: a three dimensional numpy ndarray of shape `(bands, height, width)`
- (if mode was `"vals"`) `"no_data_value"`: a number that represents 'no data'. If the ndarray is a boolean, there is no 'no data' value.
- (if mode was `"time"'"`) `"time"`: a list of naive UTC datetimes corresponding to the time axis
- (if mode was `"meta"`) `"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.

Operation to combine multiple rasters into one along all three axes (x, y and temporal). To only fill 'no data' values of input rasters that have the same temporal resolution `dask_geomodeling.raster.elemwise.FillNoData` is preferred.

Values at equal timesteps in the contributing rasters are considered starting with the leftmost input raster. Therefore, values from rasters that are more 'to the right' are shown in the result. 'no data' values are transparent and will show data of rasters more 'to the left'.

**Parameters**

***args** (*list of RasterBlocks*) – list of rasters to be combined.

**Returns**

RasterBlock that combines all input rasters

### dask_geomodeling.raster.elemwise

Module containing elementwise raster blocks.

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

Add two rasters together or add a constant value to a raster.

Either one or both of the inputs should be a RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *number*) – Addition term a
- **b** (RasterBlock, *number*) – Addition term b

**Returns**

RasterBlock containing the result of the addition.

**class** dask_geomodeling.raster.elemwise.**And**(*a*, *b*)

Returns True where both inputs are True.

Either one or both of the inputs should be a boolean RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *boolean*) – Logical term a
- **b** (RasterBlock, *boolean*) – Logical term b

**Returns**

RasterBlock containing boolean values

**class** dask_geomodeling.raster.elemwise.**Divide**(*a*, *b*)

Divide two rasters or divide a raster by a constant value.

Either one or both of the inputs should be a RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *number*) – Numerator
- **b** (RasterBlock, *number*) – Denominator

**Returns**

RasterBlock containing the result of the division.

**class** dask_geomodeling.raster.elemwise.**Equal**(*a*, *b*)

Compares the values of two rasters and returns True for equal elements.

This operation can be used to compare two rasters or to compare a raster with a static value. Note that 'no data' is not equal to 'no data': False is returned if any of the two terms is 'no data'.

Either one or both of the inputs should be a RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *number*) – Comparison term a
- **b** (RasterBlock, *number*) – Comparison term b

**Returns**
RasterBlock containing boolean values

class dask_geomodeling.raster.elemwise.**Exp**(*x*)

Return e raised to the power of the raster values.

Out-of-range results (not representable by the resulting datatype) are set to *no data*.

**Parameters**
**x** (RasterBlock) – Raster

**Returns**
RasterBlock.

class dask_geomodeling.raster.elemwise.**FillNoData**(**args*)

Combines multiple rasters filling 'no data' values.

Values at equal timesteps in the contributing rasters are considered starting with the leftmost input raster. Therefore, values from rasters that are more 'to the right' are shown in the result. 'no data' values are transparent and will show data of rasters more 'to the left'.

The temporal properties of the rasters should be equal, however spatial properties can be different.

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

**Returns**
RasterBlock that combines values from the inputs.

class dask_geomodeling.raster.elemwise.**Greater**(*a*, *b*)

Compares the values of two rasters and returns True if an element in the first term is greater.

This operation can be used to compare two rasters or to compare a raster with a static value. Note that False is returned if any of the two terms is 'no data'.

Either one or both of the inputs should be a RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *number*) – Comparison term a
- **b** (RasterBlock, *number*) – Comparison term b

**Returns**
RasterBlock containing boolean values

class dask_geomodeling.raster.elemwise.**GreaterEqual**(*a*, *b*)

" Compares the values of two rasters and returns True if an element in the first term is greater or equal.

This operation can be used to compare two rasters or to compare a raster with a static value. Note that False is returned if any of the two terms is 'no data'.

Either one or both of the inputs should be a RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *number*) – Comparison term a
- **b** (RasterBlock, *number*) – Comparison term b

**Returns**
RasterBlock containing boolean values

class dask_geomodeling.raster.elemwise.**Invert**(*x*)

Logically invert a raster (swap True and False).

Takes a single input raster containing boolean values and outputs a boolean raster with the same spatial and temporal properties.

**Parameters**
**x** (RasterBlock) – Boolean raster with values to invert

**Returns**
RasterBlock with boolean values opposite to the input raster.

class dask_geomodeling.raster.elemwise.**IsData**(*store*)

Returns True where raster has data.

Takes a single input raster and outputs a boolean raster with the same spatial and temporal properties.

**Parameters**
**store** (RasterBlock) – Input raster

**Returns**
RasterBlock with boolean values.

class dask_geomodeling.raster.elemwise.**IsNoData**(*store*)

Returns True where raster has no data.

Takes a single input raster and outputs a boolean raster with the same spatial and temporal properties.

**Parameters**
**store** (RasterBlock) – Input raster

**Returns**
RasterBlock with boolean values.

class dask_geomodeling.raster.elemwise.**Less**(*a*, *b*)

Compares the values of two rasters and returns True if an element in the first term is less.

This operation can be used to compare two rasters or to compare a raster with a static value. Note that False is returned if any of the two terms is 'no data'.

Either one or both of the inputs should be a RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *number*) – Comparison term a
- **b** (RasterBlock, *number*) – Comparison term b

**Returns**
RasterBlock containing boolean values

class dask_geomodeling.raster.elemwise.**LessEqual**(*a*, *b*)

Compares the values of two rasters and returns True if an element in the first term is less or equal.

This operation can be used to compare two rasters or to compare a raster with a static value. Note that False is returned if any of the two terms is 'no data'.

Either one or both of the inputs should be a RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *number*) – Comparison term a

- **b** (RasterBlock, *number*) – Comparison term b

**Returns**
RasterBlock containing boolean values

**class** dask_geomodeling.raster.elemwise.**Log**(*x*)

Return natural logarithm of the raster values.

Out-of-range results (not representable by the resulting datatype) are set to *no data* as well as the result of input values < 0.

**Parameters**
**x** (RasterBlock) – Raster

**Returns**
RasterBlock.

**class** dask_geomodeling.raster.elemwise.**Log10**(*x*)

Return the base 10 logarithm of the raster values.

Out-of-range results (not representable by the resulting datatype) are set to *no data* as well as the result of input values < 0.

**Parameters**
**x** (RasterBlock) – Raster

**Returns**
RasterBlock.

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

Multiply two rasters or multiply a raster by a constant value.

Either one or both of the inputs should be a RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *number*) – Multiplication factor a
- **b** (RasterBlock, *number*) – Multiplication factor b

**Returns**
RasterBlock containing the result of the multiplication.

**class** dask_geomodeling.raster.elemwise.**NotEqual**(*a*, *b*)

Compares the values of two rasters and returns False for equal elements.

This operation can be used to compare two rasters or to compare a raster with a static value. Note that 'no data' is not equal to 'no data': True is returned if any of the two terms is 'no data'.

Either one or both of the inputs should be a RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *number*) – Comparison term a
- **b** (RasterBlock, *number*) – Comparison term b

**Returns**
RasterBlock containing boolean values

class dask_geomodeling.raster.elemwise.**Or**(*a*, *b*)

Returns True where any of inputs is True.

Either one or both of the inputs should be a boolean RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *boolean*) – Logical term a
- **b** (RasterBlock, *boolean*) – Logical term b

**Returns**

RasterBlock containing boolean values

class dask_geomodeling.raster.elemwise.**Power**(*a*, *b*)

Exponential function with either a raster and a number or two rasters.

Either one or both of the inputs should be a RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *number*) – Base
- **b** (RasterBlock, *number*) – Exponent

**Returns**

RasterBlock containing the result of the exponential function.

class dask_geomodeling.raster.elemwise.**Subtract**(*a*, *b*)

Subtract two rasters or subtract a constant value from a raster

Either one or both of the inputs should be a RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *number*) – Term be subtracted from
- **b** (RasterBlock, *number*) – Term to be subtracted

**Returns**

RasterBlock containing the result of the function subtract.

class dask_geomodeling.raster.elemwise.**Xor**(*a*, *b*)

Exclusive or: returns True where exactly one of the inputs is True.

Where both inputs are True, False is returned.

Either one or both of the inputs should be a boolean RasterBlock. In case of two raster inputs the temporal properties of the rasters should be equal, however spatial properties can be different.

**Parameters**

- **a** (RasterBlock, *boolean*) – Logical term a
- **b** (RasterBlock, *boolean*) – Logical term b

**Returns**

RasterBlock containing boolean values

### dask_geomodeling.raster.misc

Module containing miscellaneous raster blocks.

**class** dask_geomodeling.raster.misc.**Classify**(*store*, *bins*, *right=False*)

Classify raster data into binned categories

Takes a RasterBlock and classifies its values based on bins. The bins are supplied as a list of increasing bin edges.

For each raster cell this operation returns the index of the bin to which the raster cell belongs. The lowest possible output cell value is 0, which means that the input value was lower than the lowest bin edge. The highest possible output value is equal to the number of supplied bin edges.

**Parameters**

- **store** (RasterBlock) – The raster whose cell values are to be classified
- **bins** (*list*) – An increasing list of bin edges
- **right** (*boolean*) – Whether the intervals include the right or the left bin edge, defaults to False.

**Returns**

RasterBlock with classified values

**class** dask_geomodeling.raster.misc.**Clip**(*store*, *source*)

Clip one raster to the extent of another raster.

Takes two raster inputs, one raster ('store') whose values are returned in the output and one raster ('source') that is used as the extent. Cells of the 'store' raster are replaced with 'no data' if there is no data in the 'source' raster.

If the 'source' raster is a boolean raster, False will result in 'no data'.

Note that the input rasters are required to have the same time resolution.

**Parameters**

- **store** (RasterBlock) – Raster whose values are clipped
- **source** (RasterBlock) – Raster that is used as the clipping mask

**Returns**

RasterBlock with clipped values.

**property period**

Return period datetime tuple.

**class** dask_geomodeling.raster.misc.**Mask**(*store*, *value*)

Replace values in a raster with a single constant value. 'no data' values are preserved.

**Parameters**

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

**Returns**

RasterBlock containing a single value

**class** dask_geomodeling.raster.misc.**MaskBelow**(*store*, *value*)

Converts raster cells below the supplied value to 'no data'.

Raster cells with values greater than or equal to the supplied value are returned unchanged.

**Parameters**

- **store** (RasterBlock) – The raster whose values are to be masked.
- **value** (*number*) – The constant value below which values are masked.

**Returns**
RasterBlock with cells below the input value converted to 'no data'.

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

Converts geometry source to raster

This operation is used to transform GeometryBlocks into RasterBlocks. Here geometries (from for example a shapefile) are converted to a raster, using the values from one of the columns.

Note that to rasterize floating point values, it is necessary to pass `dtype="float"`.

**Parameters**

- **source** (GeometryBlock) – The geometry source to be rasterized
- **column_name** (*string*) – The name of the column whose values will be returned in the raster. If column_name is not provided, a boolean raster will be generated indicating where there are geometries.
- **dtype** (*string*) – A numpy datatype specification to return the array. Defaults to 'int32' if column_name is provided, or to 'bool' otherwise.

**Returns**
RasterBlock with values from 'column_name' or a boolean raster.

**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})
```

class dask_geomodeling.raster.misc.**RasterizeWKT**(*wkt*, *projection*)

Converts a single geometry to a raster mask

**Parameters**

- **wkt** (*string*) – the WKT representation of a geometry
- **projection** (*string*) – the projection of the geometry

**Returns**
RasterBlock with True for cells that are inside the geometry.

class dask_geomodeling.raster.misc.**Reclassify**(*store*, *data*, *select=False*)

Reclassify a raster of integer values.

This operation can be used to reclassify a classified raster into desired values. Reclassification is done by supplying a list of [from, to] pairs.

**Parameters**

- **store** (RasterBlock) – The raster whose cell values are to be reclassified
- **bins** (*list*) – A list of [from, to] pairs defining the reclassification. The from values can be of bool or int datatype; the to values can be of int or float datatype
- **select** (*boolean*) – Whether to set all non-reclassified cells to 'no data', defaults to False.

**Returns**
RasterBlock with reclassified values

**class** dask_geomodeling.raster.misc.**Step**(*store*, *left=0*, *right=1*, *value=0*, *at=None*)

Apply a step function to a raster.

This operation classifies the elements of a raster into three categories: less than, equal to, and greater than a value.

The step function is defined as follows, with x being the value of a raster cell:

- 'left' if $x <$ *value*
- 'at' if $x ==$ *value*
- 'right' if $x >$ *value*

**Parameters**

- **store** (RasterBlock) – The input raster
- **left** (*number*) – Value given to cells lower than the input value, defaults to 0
- **right** (*number*) – Value given to cells higher than the input value, defaults to 1
- **value** (*number*) – The constant value which raster cells are compared to, defaults to 0
- **at** (*number*) – Value given to cells equal to the input value, defaults to the average of left and right

**Returns**
RasterBlock containing three values; left, right and at.

## 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.

The raster pixel with its topleft corner at [x, y] will define ranges [x, x + dx) and (y - dy, y]. Here [dx, dy] denotes the (unsigned) pixel size. The topleft corner and top and left edges belong to a pixel.

**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.

The value at raster cell with its topleft corner at [x, y] is assumed to define a value for ranges [x, x + dx) and (y - dy, y]. Here [dx, dy] denotes the (unsigned) pixel size. The topleft corner and top and left edges belong to a pixel.

**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 spatial dilation on specific cell values.

Cells with values in the supplied list are spatially dilated by one cell in each direction, including diagonals.

Dilation is performed in the order of the values parameter.

**Parameters**

- **store** (RasterBlock) – Raster to perform dilation on.
- **values** (*list*) – Only cells with these values are dilated.

**Returns**

RasterBlock where cells in values list are dilated.

**See also:**

https://en.wikipedia.org/wiki/Dilation_%28morphology%29

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

Calculate a hillshade from the raster values.

**Parameters**

- **store** (RasterBlock) – Raster to which the hillshade algorithm is applied.
- **altitude** (*number*) – Light source altitude in degrees, defaults to 45.
- **azimuth** (*number*) – Light source azimuth in degrees, defaults to 315.
- **fill** (*number*) – Fill value to be used for 'no data' values.

**Returns**

Hillshaded raster

**See also:**

https://pro.arcgis.com/en/pro-app/tool-reference/3d-analyst/how-hillshade-works.htm

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

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

This can be used for visualization of sparse data.

**Parameters**

- **store** (RasterBlock) – Raster to which the filter is applied
- **size** (*integer*) – Diameter of the circular footprint. This should always be an odd number larger than 1.

**Returns**

RasterBlock with maximum values inside the footprint of each input cell.

class dask_geomodeling.raster.spatial.**Place**(*store*, *place_projection*, *anchor*, *coordinates*, *statistic='last'*)

Place an input raster at given coordinates

Note that if the store's projection is different from the requested one, the data will be reprojected before placing it at a different position.

**Parameters**

- **store** (RasterBlock) – Raster that will be placed.
- **place_projection** (*str*) – The projection in which this operation is done. This also specifies the projection of the `anchor` and `coordinates` args.
- **anchor** (*list of 2 numbers*) – The anchor into the source raster that will be placed at given coordinates.
- **coordinates** (*list of lists of 2 numbers*) – The target coordinates. The center of the bbox will be placed on each of these coordinates.
- **statistic** (*str*) – What method to use to merge overlapping rasters. One of: {"last", "first", "count", "sum", "mean", "min", "max", "argmin", "argmax", "product", "std", "var", "p<number>"}

**Returns**

RasterBlock with the source raster placed

**property geo_transform**

The native geo_transform of this block

Returns None if the store projection and place projections differ.

**property geometry**

Combined geometry in this block's native projection.

**property projection**

The native projection of this block.

Only returns something if the place projection equals the store projection

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

Smooth the values from a raster spatially using Gaussian smoothing.

**Parameters**

- **store** (RasterBlock) – Raster to be smoothed
- **size** (*number*) – The extent of the smoothing in meters. The 'sigma' value for the Gaussian kernal equals `size / 3`.
- **fill** (*number*) – 'no data' are replaced by this value during smoothing, defaults to 0.

**Returns**

RasterBlock with spatially smoothed values.

**See also:**

https://en.wikipedia.org/wiki/Gaussian_blur

## dask_geomodeling.raster.temporal

Module containing raster blocks for temporal operations.

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

Compute the cumulative of a raster over time.

Contrary to `dask_geomodeling.raster.temporal.TemporalAggregate`, in this operation the timedelta of the resulting raster equals the timedelta of the input raster. Cell values are accumulated over the supplied period. At the end of each period the accumulation is reset.

**Parameters**

- **source** (RasterBlock) – The input raster whose timesteps are accumulated.
- **statistic** (*string*) – The type of accumulation to perform. Can be `"sum"` or `"count"`. Defaults to `"sum"`.
- **frequency** (*string or None*) – The period over which accumulation is performed. Supply a pandas offset string (see the references below). If this value is None, the accumulation will continue indefinitely. Defaults to None.
- **timezone** (*string*) – Timezone in which the accumulation is performed, defaults to `"UTC"`.

**Returns**

RasterBlock with temporally accumulated data.

**See also:**

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

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

Shift a temporal raster by some timedelta.

A positive timedelta shifts into the future and a negative timedelta shifts into the past.

**Parameters**

- **store** (RasterBlock) – The store whose timestamps are to be shifted
- **time** (*integer*) – The timedelta to shift the store, in milliseconds.

**Returns**

RasterBlock with its timestamps shifted.

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

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

This operations allows to take the cell values from one raster ('store') and the temporal properties of another raster ('index').

If the store is not a temporal raster, its cell values are copied to each timestep of the index raster. If the store is also a temporal raster, this operation looks at each 'index' timestamp and takes the closest 'store' timestamp as cell values.

**Parameters**

- **store** (RasterBlock) – Return cell values from this raster
- **index** (RasterBlock) – Snap values to the timestamps from this raster

**Returns**

RasterBlock with temporal properties of the index.

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

Resample a raster in time.

This operation performs temporal aggregation of rasters, for example a hourly average of data that has a 5 minute resolution.. The timedelta of the resulting raster is determined by the 'frequency' parameter.

**Parameters**

- **source** (RasterBlock) – The input raster whose timesteps are aggregated
- **frequency** (*string or None*) – The frequency to resample to, as pandas offset string (see the references below). 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 raster period. Defaults to None.
- **statistic** (*string*) – The type of statistic to perform. Can be one of `{"sum", "count", "min", "max", "mean", "median", "std", "var", "p<percentile>"}`. Defaults to `"sum"`.
- **closed** (*string or None*) – Determines what side of the interval is closed. Can be `"left"` or `"right"`. The default depends on the frequency.
- **label** (*string or None*) – Determines what side of the interval is closed. Can be `"left"` or `"right"`. The default depends on the frequency.
- **timezone** (*string*) – Timezone to perform the resampling in, defaults to `"UTC"`.

**Returns**

RasterBlock with temporally aggregated data.

**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

## 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 attribute:

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

A geometry request contains the following fields:

- mode: one of `{"intersects", "centroid", "extent"}`
- 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 is a dictionary with the following fields:

- (if mode was `"intersects"` or `"centroid"`) `"features"`: a `GeoDataFrame` of features with properties
- (if mode was `"extent"`) `"extent"`: a tuple of 4 numbers `(min_x, min_y, max_x, max_y)` that represents the extent of the geometries that would be returned by an `"intersects"` request.
- (for all modes) `"projection"`: the EPSG or WKT representation of the projection.

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`.

**to_file**(**args*, ***kwargs*)

Utility function to export data from this block to a file on disk.

You need to specify the target file path as well as the extent geometry you want to save. Feature properties can be saved by providing a field mapping to the `fields` argument.

To stay within memory constraints or to parallelize an operation, the `tile_size` argument can be provided.

**Parameters**

- **url** (*str*) – The target file path. The extension determines the format. For supported formats, consult GeometryFileSink.supported_extensions.
- **fields** (*dict*) – a mapping that relates column names to output file field names field names, {<output file field name>: <column name>, ...}.
- **tile_size** (*int*) – Optionally use this for large exports to stay within memory constraints. The export is split in tiles of given size (units are determined by the projection). Finally the tiles are merged.
- **geometry** (*shapely Geometry*) – Limit exported objects to objects whose centroid intersects with this geometry.
- **projection** (*str*) – The projection as a WKT string or EPSG code. Sets the projection of the geometry argument, the target projection of the data, and the tiling projection.
- **start** (*datetime*) – start date as UTC datetime
- **stop** (*datetime*) – stop date as UTC datetime
- ****request** – see GeometryBlock request specification

**Relevant settings can be adapted as follows:**

```
>>> from dask import config
>>> config.set({"geomodeling.root": '/my/output/data/path'})
>>> config.set({"geomodeling.geometry-limit": 10000})
>>> config.set({"temporary_directory": '/my/alternative/tmp/dir'})
```

**class** dask_geomodeling.geometry.base.**GetSeriesBlock**(*source*, *name*)

Obtain a single feature property column from a GeometryBlock.

Provide a GeometryBlock with one or more columns. One of these columns can be read from this source into a SeriesBlock. This SeriesBlock can be used to run for example classifications.

**Parameters**

- **source** (GeometryBlock) – GeometryBlock with the column you want to load into the SeriesBlock.
- **name** (*str*) – Name of the column to load into the SeriesBlock.

**Returns**

SeriesBlock containing the single property column

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

A block that represents one column from a GeometryBlock.

Use this helper class to modify (or to use logic on) a specific feature property.

Use :class:`dask_geomodeling.geometry.base.GetSeriesBlock` to retrieve a SeriesBlock from a GeometryBlock and :class:`dask_geomodeling.geometry.base.SetSeriesBlock` to add a SeriesBlock to a GeometryBlock.

**class** dask_geomodeling.geometry.base.**SetSeriesBlock**(*source*, *column*, *value*, **args*)

Add one or multiple property columns (SeriesBlocks) to a GeometryBlock.

Provide the GeometryBlock that you want to add more properties to. Then provide the SeriesBlock(s) which you want to add to the GeometryBlock. The values of the SeriesBlock will be added to the features in the GeometryBlock automatically (if they are derived from the same geometries in previous operations, the features will have matching indexes so that each property is matched to the correct feature).

The value which is set can also be a single value, in which case each feature will get the same value as a property.

**Parameters**

- **source** (GeometryBlock) – The base GeometryBlock to which the SeriesBlock is added as a new column.
- **column** (*str*) – The name of the new column (if it exists, it will be overwritten)
- **value** (SeriesBlock, *number, str, bool*) – The SeriesBlock or constant value that has to be inserted in the destination column.
- ***args** – It is possible to repeat the "column" and "value" arguments multiple times to insert more than one column.

### Example

Add two columns to an existing `view` like this: `SetSeriesBlock(view, "column_1", series_1, "column_2", series_2)`.

**Returns**

The source GeometryBlock with additional property columns

## 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 statistics of a raster for each geometry in a geometry source.

A statistic is computed in a specific projection and with a specified raster cell size. If `projection` or `pixel_size` are not given, these default to the native projection of the provided raster source. The following cells are selected to perform the statistic (e.g. mean) on:

- Polygons: all raster cells whose center is inside the polygon
- Points: the raster cell (singular) that contains the point
- Linestrings: Bresenham's line algorithm is used

If this assignment leads to the situation that a geometry covers no raster cells (for instance with a polygon much smaller than the raster cell size), the geometry is reduced to a point by taking its centroid.

Should the combination of the requested pixel_size and the extent of the source geometry cause the required raster size to exceed `max_pixels`, the `pixel_size` can be adjusted automatically if `auto_pixel_size` is set to `True`, else (the default) a RuntimeError is raised.

Please note that for any field operation on the result of this block a GetSeriesBlock should be used to retrieve data from the added column. The name of the added column is determined by the `column_name` parameter.

**Parameters**

- **source** (GeometryBlock) – The geometry source for which the statistics are determined.
- **raster** (RasterBlock) – The raster source that is sampled.

- **statistic** (*str*) – The type of statistical analysis that should be performed. The options are: {"sum", "count", "min", "max", "mean", "median", "p<percentile>"}. Percentiles are provided for example as follows: "p50". Default "sum".
- **projection** (*str, optional*) – Projection to perform the aggregation in, for example "EPSG:28992". Defaults to the native projection of the supplied raster.
- **pixel_size** (*float, optional*) – The raster cell size used in the aggregation. Defaults to the cell size of the supplied raster.
- **max_pixels** (*int, optional*) – The maximum number of pixels (cells) in the aggregation. Defaults to the geomodeling.raster-limit setting.
- **column_name** (*str, optional*) – The name of the column where the result should be placed. Defaults to "agg".
- **auto_pixel_size** (*boolean*) – Determines whether the pixel size is adjusted automatically when "max_pixels" is exceeded. Default False.

**Returns**

GeometryBlock with aggregation results in an added column

**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*)

Compute statistics of a per-feature masked raster for each geometry in a geometry source.

Per feature, a threshold can be supplied to mask the raster with. Only values that exceed the threshold of a specific feature are included for the statistical value of that feature.

See :class:dask_geomodeling.geometry.aggregate.AggregateRaster for further information.

**Parameters**

- ***args** – See :class:dask_geomodeling.geometry.aggregate.AggregateRaster
- **threshold_name** (*str*) – The column that holds the thresholds.

**Returns**

GeometryBlock with aggregation results in an added column

### dask_geomodeling.geometry.constructive

Module containing geometry block constructive operations

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

Buffer ('expand') geometries with a given value.

A GeometryBlock and a buffer distance are provided. Each feature in the GeometryBlock is buffered with the distance provided, resulting in updated geometries.

**Parameters**

- **source** (GeometryBlock) – The source GeometryBlock whose geometry will be updated.
- **distance** (*float*) – The distance used to buffer all features. The distance is measured in the unit of the given projection (e.g. m, °).
- **projection** (*str*) – The projection used in the operation provided in the format: `"EPSG:28992"`.
- **resolution** (*integer, optional*) – The resolution of the buffer provided as the number of points used to represent a quarter of a circle. The default value is `16`.

**Returns**

GeometryBlock with buffered geometries.

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

Simplify geometries, mainly to make them computationally more efficient.

Provide a GeometryBlock and a tolerance value to simplify the geometries. As a result all features in the GeometryBlock are simplified.

**Parameters**

- **source** (GeometryBlock) – Source of the geometries to be simplified.
- **tolerance** (*float*) – The tolerance used in the simplification. If no tolerance is given the `"min_size"` request parameter is used.
- **preserve_topology** (*boolean, optional*) – Determines whether the topology should be preserved in the operation. Defaults to `True`.

**Returns**

GeometryBlock which was provided as input with a simplified geometry.

### dask_geomodeling.geometry.field_operations

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

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

Element-wise addition of SeriesBlock or number to another SeriesBlock.

**Parameters**

- **source** (SeriesBlock) – First addition term
- **other** (SeriesBlock *or number*) – Second addition term

**Returns**

SeriesBlock

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

Perform an elementwise logical AND between two SeriesBlocks.

If a feature has a True value in both SeriesBlocks, True is returned, else False is returned.

**Parameters**

- **source** (SeriesBlock) – First boolean term
- **other** (SeriesBlock) – Second boolean term

**Returns**

SeriesBlock with boolean values

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

Classify a value column into different bins

For example: every value below 3 becomes "A", every value between 3 and 5 becomes "B", and every value above 5 becomes "C".

The provided SeriesBlock will be classified according to the given classification parameters. These parameters consist of two lists, one with the edges of the classification bins (i.e. `[3, 5]`) and one with the desired class output (i.e. `["A", "B", "C"]`). The input data is then compared to the classification bins. In this example a value 1 is below 3 so it gets class `"A"`. A value 4 is between 3 and 5 so it gets label `"B"`.

How values outside of the bins are classified depends on the length of the labels list. If the length of the labels equals the length of the binedges plus 1 (the above example), then values outside of the bins are classified to the first and last elements of the labels list. If the length of the labels equals the length of the bins minus 1, then values outside of the bins are classified to 'no data'.

**Parameters**

- **source** (SeriesBlock) – The (numeric) data which should be classified.
- **bins** (*list*) – The edges of the classification intervals (i.e. `[3, 5]`).
- **labels** (*list*) – The classification returned if a value falls in a specific bin (i.e. `["A", "B", "C"]`). The length of this list is either one larger or one less than the length of the `bins` argument. Labels should be unique. If labels are numeric, they are always converted to float to be able to deal with NaN values.
- **right** (*boolean, optional*) – Determines what side of the intervals are closed. Defaults to True (the right side of the bin is closed so a value assigned to the bin on the left if it is exactly on a bin edge).

**Returns**

A SeriesBlock with classified values instead of the original numbers.

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

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

See :class:`dask_geomodeling.geometry.field_operations.Classify` for further information.

**Parameters**

- **source** (GeometryBlock) – The GeometryBlock which contains the column which should be clasified as well as columns with the bin edges.
- **value_column** (*str*) – The column with (float) data which should be classified.
- **bin_columns** (*list*) – A list of columns that contain the bins for the classification. The order of the columns should be from low to high values.

- **labels** (*list*) – The classification returned if a value falls in a specific bin (i.e. `["A", "B", "C"]`). The length of this list is either one larger or one less than the length of the `bins` argument. Labels should be unique. If labels are numeric, they are always converted to float to be able to deal with NaN values.
- **right** (*boolean, optional*) – Determines what side of the intervals are closed. Defaults to True (the right side of the bin is closed so a value assigned to the bin on the left if it is exactly on a bin edge).

**Returns**
A SeriesBlock with classified values instead of the original floats.

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

Element-wise division of SeriesBlock or number with another SeriesBlock.

Note that if you want to divide a constant value by a SeriesBlock (like `3 / series`, you have to do `Multiply(3, Power(series, -1))`.

**Parameters**

- **source** (SeriesBlock) – Numerator
- **other** (SeriesBlock *or number*) – Denominator

**Returns**
SeriesBlock

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

Determine whether a SeriesBlock and a second SeriesBlock or a constant value are equal.

Note that 'no data' does not equal 'no data'.

**Parameters**

- **source** (SeriesBlock) – First comparison term
- **other** (SeriesBlock *or number*) – Second comparison term

**Returns**
SeriesBlock with boolean values

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

Element-wise integer division of SeriesBlock or number with another SeriesBlock.

The outcome of the division is converted to the closest integer below (i.e. 3.4 becomes 3, 3.9 becomes 3 and -3.4 becomes -4)

**Parameters**

- **source** (SeriesBlock) – Numerator
- **other** (SeriesBlock *or number*) – Denominator

**Returns**
SeriesBlock

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

Determine for each value in a SeriesBlock whether it is greater than a comparison value from a SeriesBlock or constant.

**Parameters**

- **source** (SeriesBlock) – First comparison term
- **other** (SeriesBlock *or number*) – Second comparison term

**Returns**
SeriesBlock with boolean values

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

Determine for each value in a SeriesBlock whether it is greater than or equal to a comparison value from a SeriesBlock or constant.

**Parameters**

- **source** (SeriesBlock) – First comparison term
- **other** (SeriesBlock *or number*) – Second comparison term

**Returns**
SeriesBlock with boolean values

class dask_geomodeling.geometry.field_operations.**Invert**(*source*, **args*)

Invert a boolean SeriesBlock (swap True and False)

**Parameters**
**source** (SeriesBlock) – SeriesBlock with boolean values.

**Returns**
SeriesBlock with boolean values

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

Determine for each value in a SeriesBlock whether it is less than a comparison value from a SeriesBlock or constant.

**Parameters**

- **source** (SeriesBlock) – First comparison term
- **other** (SeriesBlock *or number*) – Second comparison term

**Returns**
SeriesBlock with boolean values

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

Determine for each value in a SeriesBlock whether it is less than or equal to a comparison value from a SeriesBlock or constant.

**Parameters**

- **source** (SeriesBlock) – First comparison term
- **other** (SeriesBlock *or number*) – Second comparison term

**Returns**
SeriesBlock with boolean values

class dask_geomodeling.geometry.field_operations.**Mask**(*source*, *cond*, *other*)

Replace values in a SeriesBlock where values in another SeriesBlock are True.

Provide a source SeriesBlock, a conditional SeriesBlock (True/False) and a replacement value which can either be a SeriesBlock or a constant value. All entries in the source that correspond to a True value in the conditional are left unchanged. The values in the source that correspond to a True value in the conditional are replaced with the value from 'other'.

**Parameters**

- **source** (SeriesBlock) – Source SeriesBlock that is going to be updated

- **cond** (SeriesBlock) – Conditional SeriesBlock that determines whether features in the source SeriesBlock will be updated. If this is not boolean (True/False), then all data values (including 0) are interpreted as True. Missing values are always interpeted as False.
- **other** (SeriesBlock *or constant*) – The value that should be used as a replacement for the source SeriesBlock where the conditional SeriesBlock is True.

**Returns**
SeriesBlock with updated values where condition is True.

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

Element-wise modulo (remainder after division) of SeriesBlock or number with another SeriesBlock.

Example: if the input is `[31, 5.3, -4]` and the modulus is `3`, the outcome would be `[1, 2.3, 2]`. The outcome is always postive and less than the modulus.

**Parameters**

- **source** (SeriesBlock) – Number
- **other** (SeriesBlock *or number*) – Modulus

**Returns**
SeriesBlock

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

Element-wise multiplication of SeriesBlock or number with another SeriesBlock.

**Parameters**

- **source** (SeriesBlock) – First multiplication factor
- **other** (SeriesBlock *or number*) – Second multiplication factor

**Returns**
SeriesBlock

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

Determine whether a SeriesBlock and a second SeriesBlock or a constant value are not equal.

Note that 'no data' does not equal 'no data'.

**Parameters**

- **source** (SeriesBlock) – First comparison term
- **other** (SeriesBlock *or number*) – Second comparison term

**Returns**
SeriesBlock with boolean values

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

Perform an elementwise logical OR between two SeriesBlocks.

If a feature has a True value in any of the input SeriesBlocks, True is returned, else False is returned.

**Parameters**

- **source** (SeriesBlock) – First boolean term
- **other** (SeriesBlock) – Second boolean term

**Returns**
SeriesBlock with boolean values

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

Element-wise raise a SeriesBlock to the power of a number or another SeriesBlock.

For example, the inputs `[2, 4]` and `2` will give output `[4, 16]`.

**Parameters**

- **source** (SeriesBlock) – Base
- **other** (SeriesBlock *or number*) – Exponent

**Returns**
SeriesBlock

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) – SeriesBlock with float data that is rounded to the provided number of decimals.
- **decimals** (*int, optional*) – 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.

**Returns**
SeriesBlock with rounded values.

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

Element-wise subtraction of SeriesBlock or number with another SeriesBlock.

Note that if you want to subtract a SeriesBlock from a constant value (like `4 - series`, you have to do `Add(Multiply(series, -1), 4)`.

**Parameters**

- **source** (SeriesBlock) – First subtraction term
- **other** (SeriesBlock *or number*) – Second subtraction term

**Returns**
SeriesBlock

class dask_geomodeling.geometry.field_operations.**Where**(*source*, *cond*, *other*)

Replace values in a SeriesBlock where values in another SeriesBlock are False.

Provide a source SeriesBlock, a conditional SeriesBlock (True/False) and a replacement value which can either be a SeriesBlock or a constant value. All entries in the source that correspond to a True value in the conditional are left unchanged. The values in the source that correspond to a False value in the conditional are replaced with the value from 'other'.

**Parameters**

- **source** (SeriesBlock) – Source SeriesBlock that is going to be updated
- **cond** (SeriesBlock) – Conditional SeriesBlock that determines whether features in the source SeriesBlock will be updated. If this is not boolean (True/False), then all data values (including 0) are interpreted as True. Missing values are always interpeted as False.
- **other** (SeriesBlock *or constant*) – The value that should be used as a replacement for the source SeriesBlock where the conditional SeriesBlock is False.

**Returns**
SeriesBlock with updated values where condition is False.

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

Perform an elementwise logical exclusive OR between two SeriesBlocks.

If a feature has a True value in precisely one of the input SeriesBlocks, True is returned, else False is returned.

**Parameters**

- **source** (SeriesBlock) – First boolean term
- **other** (SeriesBlock) – Second boolean term

**Returns**

SeriesBlock with boolean values

### dask_geomodeling.geometry.geom_operations

Module containing operations that return series from geometry fields

class dask_geomodeling.geometry.geom_operations.**Area**(*source*, *projection*)

Calculate the area of features in a GeometryBlock.

Provide a GeometryBlock and a projection. Returns the area of each individual geometry in the input block, in that projection.

**Parameters**

- **source** (GeometryBlock) – Source GeometryBlock which contains the features.
- **projection** (*str*) – Projection in which to compute the area (i.e. `"epsg:28992"`).

**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

Provide two GeometryBlocks with the same original source to make sure they can be matched on index. The additional SeriesBlocks that have been added to the GeometryBlock will be combined to one GeometryBlock that contains all the information.

**Parameters**

- **left** (GeometryBlock) – The left GeometryBlock to be combined.
- **right** (GeometryBlock) – The right GeometryBlock to be combined.
- **how** (*str, optional*) – The parameter that describes how the merge should be performed. There are four options:
  1. `"left"`: The resulting GeometryBlock will have all the features that are present in the left GeometryBlock, no matter the features in the right GeometryBlock.
  2. `"right"`: The resulting GeometryBlock will have all the features that are present in the right GeometryBlock, no matter the features in the left GeometryBlock.

3. `"inner"` (default): The outcome will contain all the features that are present in both input GeometryBlocks. Features that are absent in one of the GeometryBlocks will be absent in the result.
4. `outer`: The result will contain all the features which are present in one of the input GeometryBlocks.

- **suffixes** (*tuple, optional*) – Text to be added to the column names to distinguish whether they originate from the left or right GeometryBlock. Default: `("", "_right")`.

**Returns**
GeometryBlock that contains a combination of features and columns of the two input GeometryBlocks.

### 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) – The source GeometryBlock
- **size** (*float*) – The maximum size of a tile in units of the projection
- **projection** (*str*) – The projection as EPSG or WKT string in which to compute tiles (e.g. `"EPSG:28992"`)

**Returns**
GeometryBlock that 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*)

Calculate the geometric difference of two GeometryBlocks.

All geometries in the source GeometryBlock will be adapted by geometries with the same index from the second GeometryBlock. The difference operation removes any overlap between the geometries from the first geometry.

**Parameters**

- **source** (GeometryBlock) – First geometry source.
- **other** (GeometryBlock) – Second geometry source.

**Returns**
A GeometryBlock with altered geometries. Properties are preserved.

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

Calculate the intersection of a GeometryBlock with the request geometry.

Normally, geometries returned by a GeometryBlock may be partially outside of the requested geometry. This block ensures that the geometries are strictly inside the requested geometry by taking the intersection of each geometry with the request geometry.

**Parameters**
**source** (GeometryBlock) – Input geometry source.

**Returns**
A GeometryBlock with altered geometries. Properties are preserved.

### 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.

The input of this blocks is by default limited by the global geomodeling.geometry-limit setting.

**Parameters**

- **url** (*str*) – Path (URL) to the file. If relative, it is taken relative to the geomodeling.root setting.
- **layer** (*str, optional*) – The layer name in the source to select. If None, (default) the first layer is used.
- **id_field** (*str, optional*) – The field name to use as feature index. Default `"id"`.

**Relevant settings can be adapted as follows:**

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

**class** dask_geomodeling.geometry.sources.**GeometryWKTSource**(*wkt*, *projection*)

Converts a single geometry to a geometry source

**Parameters**

- **wkt** (*string*) – the WKT representation of a geometry
- **projection** (*string*) – the projection of the geometry

**Returns**
GeometryBlock

### dask_geomodeling.geometry.sinks

**class** dask_geomodeling.geometry.sinks.**GeometryFileSink**(*source*, *url*, *extension='shp'*, *fields=None*)

Write geometry data to files in a specified directory

Use GeometryFileSink.merge_files to merge tiles into one large file.

**Parameters**

- **source** (GeometryBlock) – The block the data is coming from
- **url** (*str*) – The target directory to put the files in. If relative, it is taken relative to the geomodeling.root setting.
- **extension** (*str*) – The file extension (defines the format), one of `{"shp", "gpkg", "geojson", "gml"}`. On some platforms, these options might be limited. For an accurate list, see `GeometryFileSink.supported_extensions`.
- **fields** (*dict*) – A mapping that relates column names to output file field names field names like `{<output file field name>: <column name>}`.

**Relevant settings can be adapted as follows:**

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

**static merge_files**(*path*, *target*, *remove_source=False*)

Merge files (the output of this Block) into one single file.

Optionally removes the source files.

dask_geomodeling.geometry.sinks.**to_file**(*source*, *url*, *fields=None*, *tile_size=None*, *dry_run=False*, ***request*)

Utility function to export data from a GeometryBlock to a file on disk.

You need to specify the target file path as well as the extent geometry you want to save. Feature properties can be saved by providing a field mapping to the `fields` argument.

To stay within memory constraints or to parallelize an operation, the `tile_size` argument can be provided.

**Parameters**

- **source** (GeometryBlock) – the block the data is coming from
- **url** (*str*) – The target file path. The extension determines the format. For supported formats, consult GeometryFileSink.supported_extensions.
- **fields** (*dict*) – a mapping that relates column names to output file field names field names, `{<output file field name>: <column name>, ...}`.
- **tile_size** (*int*) – Optionally use this for large exports to stay within memory constraints. The export is split in tiles of given size (units are determined by the projection). Finally the tiles are merged.
- **dry_run** (*bool*) – Do nothing, only validate the arguments.
- **geometry** (*shapely Geometry*) – Limit exported objects to objects whose centroid intersects with this geometry.
- **projection** (*str*) – The projection as a WKT string or EPSG code. Sets the projection of the geometry argument, the target projection of the data, and the tiling projection.
- **mode** (*str*) – one of `{"intersects", "centroid"}`, default "centroid"
- **start** (*datetime*) – start date as UTC datetime
- **stop** (*datetime*) – stop date as UTC datetime
- ****request** – see GeometryBlock request specification

**Relevant settings can be adapted as follows:**

```
>>> from dask import config
>>> config.set({"geomodeling.root": '/my/output/data/path'})
>>> config.set({"temporary_directory": '/my/alternative/tmp/dir'})
```

### 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** (*str*) – Existing column in source.
- **key_mapping** (*dict*) – Mapping containing pairs {key_name: column_name}:

CHAPTER

THREE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## d

dask_geomodeling.core.graphs, 13
dask_geomodeling.geometry.aggregate, 30
dask_geomodeling.geometry.base, 28
dask_geomodeling.geometry.constructive, 32
dask_geomodeling.geometry.field_operations, 32
dask_geomodeling.geometry.geom_operations, 38
dask_geomodeling.geometry.merge, 38
dask_geomodeling.geometry.parallelize, 39
dask_geomodeling.geometry.set_operations, 39
dask_geomodeling.geometry.sinks, 40
dask_geomodeling.geometry.sources, 40
dask_geomodeling.geometry.text, 42
dask_geomodeling.raster.base, 14
dask_geomodeling.raster.combine, 15
dask_geomodeling.raster.elemwise, 16
dask_geomodeling.raster.misc, 21
dask_geomodeling.raster.sources, 23
dask_geomodeling.raster.spatial, 24
dask_geomodeling.raster.temporal, 26

# INDEX


## A

Add (*class in dask_geomodeling.geometry.field_operations*), 32
Add (*class in dask_geomodeling.raster.elemwise*), 16
AggregateRaster (*class in dask_geomodeling.geometry.aggregate*), 30
AggregateRasterAboveThreshold (*class in dask_geomodeling.geometry.aggregate*), 31
And (*class in dask_geomodeling.geometry.field_operations*), 32
And (*class in dask_geomodeling.raster.elemwise*), 16
Area (*class in dask_geomodeling.geometry.geom_operations*), 38

## B

Block (*class in dask_geomodeling.core.graphs*), 13
Buffer (*class in dask_geomodeling.geometry.constructive*), 32

## C

Classify (*class in dask_geomodeling.geometry.field_operations*), 33
Classify (*class in dask_geomodeling.raster.misc*), 21
ClassifyFromColumns (*class in dask_geomodeling.geometry.field_operations*), 33
Clip (*class in dask_geomodeling.raster.misc*), 21
compute() (*in module dask_geomodeling.core.graphs*), 14
construct() (*in module dask_geomodeling.core.graphs*), 14
Cumulative (*class in dask_geomodeling.raster.temporal*), 26

## D

dask_geomodeling.core.graphs
    module, 13
dask_geomodeling.geometry.aggregate
    module, 30
dask_geomodeling.geometry.base
    module, 28
dask_geomodeling.geometry.constructive
    module, 32
dask_geomodeling.geometry.field_operations
    module, 32
dask_geomodeling.geometry.geom_operations
    module, 38
dask_geomodeling.geometry.merge
    module, 38
dask_geomodeling.geometry.parallelize
    module, 39
dask_geomodeling.geometry.set_operations
    module, 39
dask_geomodeling.geometry.sinks
    module, 40
dask_geomodeling.geometry.sources
    module, 40
dask_geomodeling.geometry.text
    module, 42
dask_geomodeling.raster.base
    module, 14
dask_geomodeling.raster.combine
    module, 15
dask_geomodeling.raster.elemwise
    module, 16
dask_geomodeling.raster.misc
    module, 21
dask_geomodeling.raster.sources
    module, 23
dask_geomodeling.raster.spatial
    module, 24
dask_geomodeling.raster.temporal
    module, 26
deserialize() (*dask_geomodeling.core.graphs.Block class method*), 13
Difference (*class in dask_geomodeling.geometry.set_operations*), 39
Dilate (*class in dask_geomodeling.raster.spatial*), 24
Divide (*class in dask_geomodeling.geometry.field_operations*), 34
Divide (*class in dask_geomodeling.raster.elemwise*), 16

## E

Equal (*class in dask_geomodeling.geometry.field_operations*), 34
Equal (*class in dask_geomodeling.raster.elemwise*), 16
Exp (*class in dask_geomodeling.raster.elemwise*), 17

## F

FillNoData (*class in dask_geomodeling.raster.elemwise*), 17
FloorDivide (*class in dask_geomodeling.geometry.field_operations*), 34
from_import_path() (*dask_geomodeling.core.graphs.Block static method*), 13
from_json() (*dask_geomodeling.core.graphs.Block class method*), 13

## G

geo_transform (*dask_geomodeling.raster.spatial.Place property*), 25
geometry (*dask_geomodeling.raster.spatial.Place property*), 26
GeometryBlock (*class in dask_geomodeling.geometry.base*), 28
GeometryFileSink (*class in dask_geomodeling.geometry.sinks*), 40
GeometryFileSource (*class in dask_geomodeling.geometry.sources*), 40
GeometryTiler (*class in dask_geomodeling.geometry.parallelize*), 39
GeometryWKTSource (*class in dask_geomodeling.geometry.sources*), 40
get_compute_graph() (*dask_geomodeling.core.graphs.Block method*), 13
get_data() (*dask_geomodeling.core.graphs.Block method*), 13
get_graph() (*dask_geomodeling.core.graphs.Block method*), 13
get_import_path() (*dask_geomodeling.core.graphs.Block class method*), 13
get_sources_and_requests() (*dask_geomodeling.core.graphs.Block method*), 13
GetSeriesBlock (*class in dask_geomodeling.geometry.base*), 29
Greater (*class in dask_geomodeling.geometry.field_operations*), 34
Greater (*class in dask_geomodeling.raster.elemwise*), 17
GreaterEqual (*class in dask_geomodeling.geometry.field_operations*), 35
GreaterEqual (*class in dask_geomodeling.raster.elemwise*), 17
Group (*class in dask_geomodeling.raster.combine*), 15

## H

HillShade (*class in dask_geomodeling.raster.spatial*), 24

## I

Intersection (*class in dask_geomodeling.geometry.set_operations*), 39
Invert (*class in dask_geomodeling.geometry.field_operations*), 35
Invert (*class in dask_geomodeling.raster.elemwise*), 17
IsData (*class in dask_geomodeling.raster.elemwise*), 18
IsNoData (*class in dask_geomodeling.raster.elemwise*), 18

## L

Less (*class in dask_geomodeling.geometry.field_operations*), 35
Less (*class in dask_geomodeling.raster.elemwise*), 18
LessEqual (*class in dask_geomodeling.geometry.field_operations*), 35
LessEqual (*class in dask_geomodeling.raster.elemwise*), 18
Log (*class in dask_geomodeling.raster.elemwise*), 19
Log10 (*class in dask_geomodeling.raster.elemwise*), 19

## M

Mask (*class in dask_geomodeling.geometry.field_operations*), 35
Mask (*class in dask_geomodeling.raster.misc*), 21
MaskBelow (*class in dask_geomodeling.raster.misc*), 21
MemorySource (*class in dask_geomodeling.raster.sources*), 23
merge_files() (*dask_geomodeling.geometry.sinks.GeometryFileSink static method*), 41
MergeGeometryBlocks (*class in dask_geomodeling.geometry.merge*), 38
module
    dask_geomodeling.core.graphs, 13
    dask_geomodeling.geometry.aggregate, 30
    dask_geomodeling.geometry.base, 28
    dask_geomodeling.geometry.constructive, 32
    dask_geomodeling.geometry.field_operations, 32
    dask_geomodeling.geometry.geom_operations, 38
    dask_geomodeling.geometry.merge, 38
    dask_geomodeling.geometry.parallelize, 39

dask_geomodeling.geometry.set_operations, 39
dask_geomodeling.geometry.sinks, 40
dask_geomodeling.geometry.sources, 40
dask_geomodeling.geometry.text, 42
dask_geomodeling.raster.base, 14
dask_geomodeling.raster.combine, 15
dask_geomodeling.raster.elemwise, 16
dask_geomodeling.raster.misc, 21
dask_geomodeling.raster.sources, 23
dask_geomodeling.raster.spatial, 24
dask_geomodeling.raster.temporal, 26
Modulo (*class in dask_geomodeling.geometry.field_operations*), 36
MovingMax (*class in dask_geomodeling.raster.spatial*), 25
Multiply (*class in dask_geomodeling.geometry.field_operations*), 36
Multiply (*class in dask_geomodeling.raster.elemwise*), 19

## N

NotEqual (*class in dask_geomodeling.geometry.field_operations*), 36
NotEqual (*class in dask_geomodeling.raster.elemwise*), 19

## O

Or (*class in dask_geomodeling.geometry.field_operations*), 36
Or (*class in dask_geomodeling.raster.elemwise*), 19

## P

ParseTextColumn (*class in dask_geomodeling.geometry.text*), 42
period (*dask_geomodeling.raster.misc.Clip property*), 21
Place (*class in dask_geomodeling.raster.spatial*), 25
Power (*class in dask_geomodeling.geometry.field_operations*), 36
Power (*class in dask_geomodeling.raster.elemwise*), 20
process() (*dask_geomodeling.core.graphs.Block static method*), 14
projection (*dask_geomodeling.raster.spatial.Place property*), 26

## R

RasterBlock (*class in dask_geomodeling.raster.base*), 14
RasterFileSource (*class in dask_geomodeling.raster.sources*), 24
Rasterize (*class in dask_geomodeling.raster.misc*), 22
RasterizeWKT (*class in dask_geomodeling.raster.misc*), 22
Reclassify (*class in dask_geomodeling.raster.misc*), 22
Round (*class in dask_geomodeling.geometry.field_operations*), 37

## S

serialize() (*dask_geomodeling.core.graphs.Block method*), 14
SeriesBlock (*class in dask_geomodeling.geometry.base*), 29
SetSeriesBlock (*class in dask_geomodeling.geometry.base*), 29
Shift (*class in dask_geomodeling.raster.temporal*), 26
Simplify (*class in dask_geomodeling.geometry.constructive*), 32
Smooth (*class in dask_geomodeling.raster.spatial*), 26
Snap (*class in dask_geomodeling.raster.temporal*), 27
Step (*class in dask_geomodeling.raster.misc*), 23
Subtract (*class in dask_geomodeling.geometry.field_operations*), 37
Subtract (*class in dask_geomodeling.raster.elemwise*), 20

## T

TemporalAggregate (*class in dask_geomodeling.raster.temporal*), 27
to_file() (*dask_geomodeling.geometry.base.GeometryBlock method*), 28
to_file() (*in module dask_geomodeling.geometry.sinks*), 41
to_json() (*dask_geomodeling.core.graphs.Block method*), 14
token (*dask_geomodeling.core.graphs.Block property*), 14

## W

Where (*class in dask_geomodeling.geometry.field_operations*), 37

## X

Xor (*class in dask_geomodeling.geometry.field_operations*), 37
Xor (*class in dask_geomodeling.raster.elemwise*), 20