datashader Documentation

Release 0.6.4

datashader contributors

Dec 05, 2017

Contents

1 FAQ 3

2 Other resources 5

Python Module Index 15

Datashader is a graphics pipeline system for creating meaningful representations of large datasets quickly and flexibly. Datashader breaks the creation of images into a series of explicit steps that allow computations to be done on intermediate representations. This approach allows accurate and effective visualizations to be produced automatically, and also makes it simple for data scientists to focus on particular data and relationships of interest in a principled way. Using highly optimized rendering routines written in Python but compiled to machine code using Numba, datashader makes it practical to work with extremely large datasets even on standard hardware.

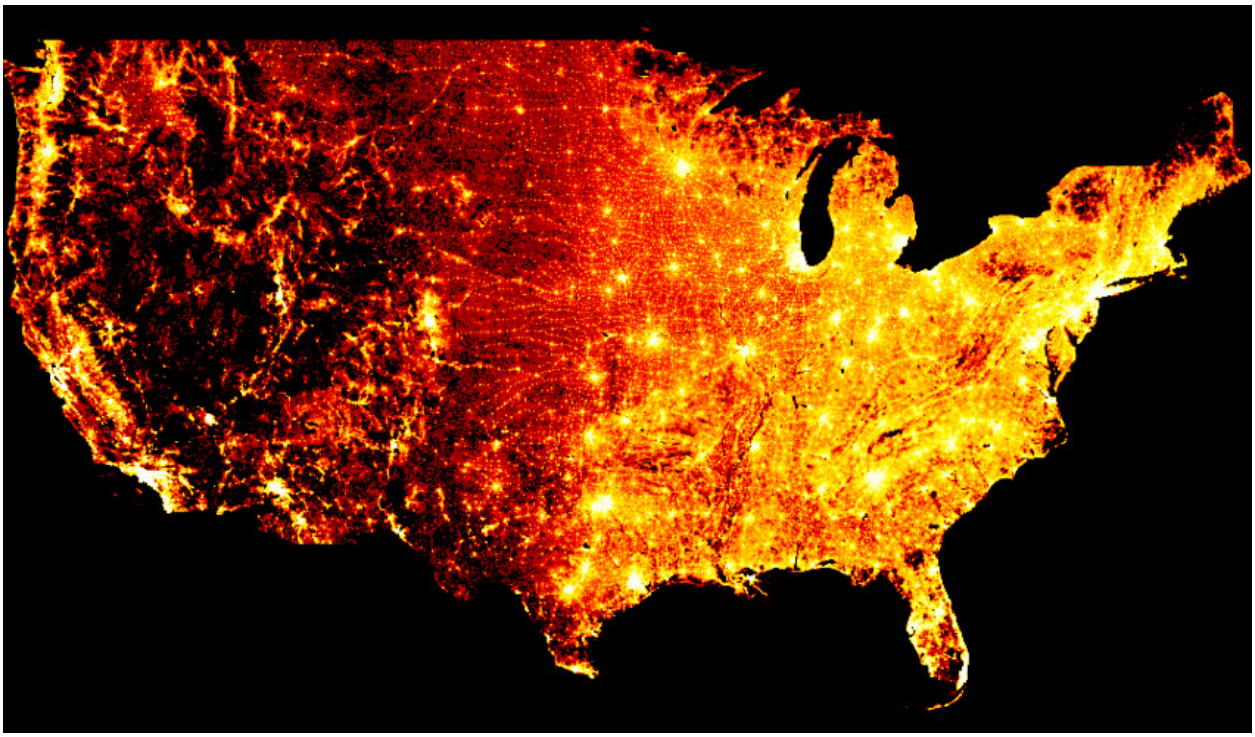
To make it concrete, here's an example of what datashader code looks like:

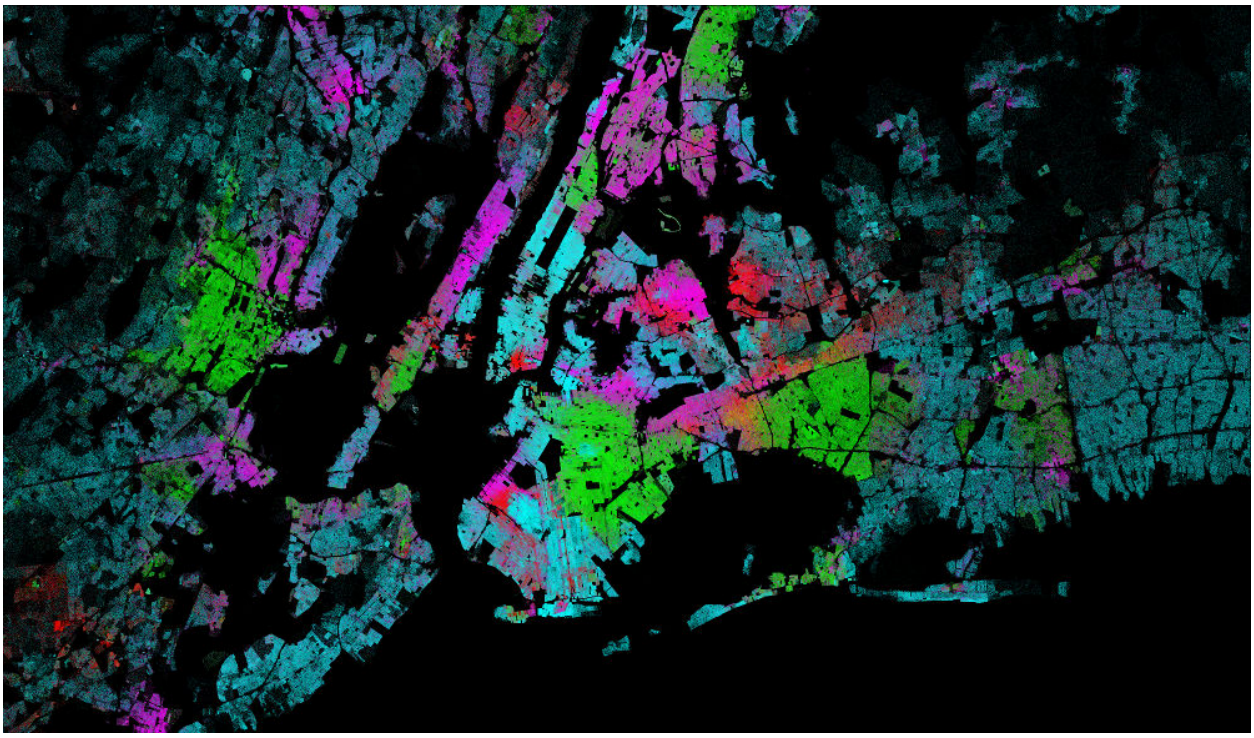
```
>>> import datashader as ds
>>> import datashader.transfer_functions as tf
>>> import pandas as pd
>>> df = pd.read_csv('user_data.csv')

>>> cvs = ds.Canvas(plot_width=400, plot_height=400)
>>> agg = cvs.points(df, 'x_col', 'y_col', ds.mean('z_col'))
>>> img = tf.shade(agg, cmap=['lightblue', 'darkblue'], how='log')
```

This code reads a data file into a Pandas dataframe `df`, and then projects the fields `x_col` and `y_col` onto the x and y dimensions of 400x400 grid, aggregating it by the mean value of the `z_col` of each datapoint. The results are rendered into an image where the minimum count will be plotted in `lightblue`, the maximum in `darkblue`, and ranging logarithmically in between.

And here are some sample outputs for data from the 2010 US census, each constructed using a similar set of code:





Documentation for datashader is primarily provided in the form of Jupyter notebooks. To understand which plotting problems datashader helps you avoid, you can start with our Plotting Pitfalls notebook. To see the steps in the datashader pipeline in detail, you can start with our Pipeline notebook. Or you may want to start with detailed case studies of datashader in action, such as our NYC Taxi, US Census, and OpenSky notebooks. In most cases, the easiest way to use Datashader via the high-level HoloViews package, which lets you flexibly switch between Datashader and non-Datashader plots generated by Matplotlib or Bokeh. Additional notebooks showing how to use datashader for other applications or data types are viewable on Anaconda Cloud and can be downloaded in runnable form as described on the datashader examples page.

CHAPTER 1

FAQ

Q: When should I use datashader?

A: Datashader is designed for working with large datasets, for cases where it is most crucial to faithfully represent the *distribution* of your data. datashader can work easily with extremely large datasets, generating a fixed-size data structure (regardless of the original number of records) that gets transferred to your local browser for display. If you ever find yourself subsampling your data just so that you can plot it feasibly, or if you are forced for practical reasons to iterate over chunks of it rather than looking at all of it at once, then datashader can probably help you.

Q: When should I *not* use datashader?

A: If you have a very small number of data points (in the hundreds or thousands) or curves (in the tens or several tens, each with hundreds or thousands of points), then conventional plotting packages like Bokeh may be more suitable. With conventional browser-based packages, all of the data points are passed directly to the browser for display, allowing specific interaction with each curve or point, including display of metadata, linking to sources, etc. This approach offers the most flexibility *per point* or *per curve*, but rapidly runs into limitations on how much data can be processed by the browser, and how much can be displayed on screen and resolved by the human visual system. If you are not having such problems, i.e., your data is easily handled by your plotting infrastructure and you can easily see and work with all your data onscreen already, then you probably don't need datashader.

Q: Is datashader part of bokeh?

A: datashader is an independent project, focusing on generating aggregate arrays and representations of them as images. Bokeh is a complementary project, focusing on building browser-based visualizations and dashboards. Bokeh (along with other plotting packages) can display images rendered by datashader, providing axes, interactive zooming and panning, selection, legends, hover information, and so on. Sample bokeh-based plotting code is provided with datashader, but viewers for maptlotlib are already under development, and similar code could be developed for any other plotting package that can display images. The library can also be used separately, without any external plotting packages, generating images that can be displayed directly or saved to disk, or generating aggregate arrays suitable for further analysis.

CHAPTER 2

Other resources

You can watch a short talk about datashader on YouTube: Datashader: Revealing the Structure of Genuinely Big Data. The video, Visualizing Billions of Points of Data, and its slides from a February 2016 one-hour talk introducing Datashader are also available, but do not cover more recent extensions to the library.

Some of the original ideas for datashader were developed under the name Abstract Rendering, which is described in a 2014 SPIE VDA paper.

The source code for datashader is maintained at our Github site, and is documented using the API link on this page.

2.1 API

2.1.1 Entry Points

Canvas

Canvas([plot_width, plot_height, x_range, ...])	An abstract canvas representing the space in which to bin.
Canvas.line(source, x, y[, agg])	Compute a reduction by pixel, mapping data to pixels as a line.
Canvas.points(source, x, y[, agg])	Compute a reduction by pixel, mapping data to pixels as points.
Canvas.raster(source[, layer, ...])	Sample a raster dataset by canvas size and bounds.
Canvas.validate()	Check that parameter settings are valid for this object

Pipeline

Pipeline(df, glyph[, agg, transform_fn, ...])	A datashading pipeline callback.

2.1.2 Edge Bundling

directly_connect_edges
hammer_bundle

2.1.3 Glyphs

Point

Point(x, y)	A point, with center at x and y.
Point.inputs	
Point.validate(in_dshape)	

Line

Line(x, y)	A line, with vertices defined by x and y.
Line.inputs	
Line.validate(in_dshape)	

2.1.4 Reductions

any([column])	Whether any elements in column map to each bin.
count([column])	Count elements in each bin.
count_cat(column)	Count of all elements in column, grouped by category.
m2(column)	Sum of square differences from the mean of all elements in column.
max(column)	Maximum value of all elements in column.
mean(column)	Mean of all elements in column.
min(column)	Minimum value of all elements in column.
std(column)	Standard Deviation of all elements in column.
sum(column)	Sum of all elements in column.
summary(**kwargs)	A collection of named reductions.
var(column)	Variance of all elements in column.

2.1.5 Transfer Functions

Image

Image(data[, coords, dims, name, attrs, ...])
Image.to_bytesio([format, origin])
Image.to_pil([origin])

Other

dynspread(img[, threshold, max_px, shape, ...])	Spread pixels in an image dynamically based on the image density.
set_background(img[, color, name])	Return a new image, with the background set to *color*.

Continued on next page

Table 2.8 – continued from previous page

shade(agg[, cmap, color_key, how, alpha, ...])	Convert a DataArray to an image by choosing an RGBA pixel color for each value.
spread(img[, px, shape, how, mask, name])	Spread pixels in an image.
stack(*imgs, **kwargs)	Combine images together, overlaying later images onto earlier ones.

2.1.6 Definitions

class datashader.**Canvas**(*plot_width=600*, *plot_height=600*, *x_range=None*, *y_range=None*, *x_axis_type='linear'*, *y_axis_type='linear'*)

An abstract canvas representing the space in which to bin.

Parameters **plot_width, plot_height** : int, optional

Width and height of the output aggregate in pixels.

x_range, y_range : tuple, optional

A tuple representing the bounds inclusive space `[min, max]` along the axis.

x_axis_type, y_axis_type : str, optional

The type of the axis. Valid options are `'linear'` [default], and `'log'`.

class datashader.**Pipeline**(*df*, *glyph*, *agg=<datashader.reductions.count object>*, *transform_fn=<function identity>*, *color_fn=<function shade>*, *spread_fn=<function dynspread>*, *width_scale=1.0*, *height_scale=1.0*)

A datashading pipeline callback.

Given a declarative specification, creates a callable with the following signature:

```
callback(x_range, y_range, width, height)
```

where `x_range` and `y_range` form the bounding box on the viewport, and `width` and `height` specify the output image dimensions.

Parameters **df** : pandas.DataFrame, dask.DataFrame

glyph : Glyph

The glyph to bin by.

agg : Reduction, optional

The reduction to compute per-pixel. Default is `count()`.

transform_fn : callable, optional

A callable that takes the computed aggregate as an argument, and returns another aggregate. This can be used to do preprocessing before passing to the `color_fn` function.

color_fn : callable, optional

A callable that takes the output of `tranform_fn`, and returns an `Image` object. Default is `shade`.

spread_fn : callable, optional

A callable that takes the output of `color_fn`, and returns another `Image` object. Default is `dynspread`.

height_scale: float, optional

Factor by which to scale the provided height

width_scale: float, optional

Factor by which to scale the provided width

class datashader.glyphs.**Point**(*x*, *y*)
A point, with center at x and y.

Points map each record to a single bin. Points falling exactly on the upper bounds are treated as a special case, mapping into the previous bin rather than being cropped off.

Parameters **x, y** : str

Column names for the x and y coordinates of each point.

class datashader.glyphs.**Line**(*x*, *y*)
A line, with vertices defined by x and y.

Parameters **x, y** : str

Column names for the x and y coordinates of each vertex.

class datashader.reductions.**count**(*column=None*)
Count elements in each bin.

Parameters **column** : str, optional

If provided, only counts elements in column that are not NaN. Otherwise, counts every element.

class datashader.reductions.**any**(*column=None*)
Whether any elements in column map to each bin.

Parameters **column** : str, optional

If provided, only elements in column that are NaN are skipped.

class datashader.reductions.**sum**(*column*)
Sum of all elements in column.

Parameters **column** : str

Name of the column to aggregate over. Column data type must be numeric. NaN values in the column are skipped.

class datashader.reductions.**m2**(*column*)
Sum of square differences from the mean of all elements in column.

Intermediate value for computing var and std, not intended to be used on its own.

Parameters **column** : str

Name of the column to aggregate over. Column data type must be numeric. NaN values in the column are skipped.

class datashader.reductions.**min**(*column*)
Minimum value of all elements in column.

Parameters **column** : str

Name of the column to aggregate over. Column data type must be numeric. NaN values in the column are skipped.

class datashader.reductions.**max**(*column*)
Maximum value of all elements in column.

Parameters **column** : str

Name of the column to aggregate over. Column data type must be numeric. `NaN` values in the column are skipped.

class datashader.reductions.**mean**(*column*)
Mean of all elements in `column`.

Parameters **column** : str

Name of the column to aggregate over. Column data type must be numeric. `NaN` values in the column are skipped.

class datashader.reductions.**var**(*column*)
Variance of all elements in `column`.

Parameters **column** : str

Name of the column to aggregate over. Column data type must be numeric. `NaN` values in the column are skipped.

class datashader.reductions.**std**(*column*)
Standard Deviation of all elements in `column`.

Parameters **column** : str

Name of the column to aggregate over. Column data type must be numeric. `NaN` values in the column are skipped.

class datashader.reductions.**count_cat**(*column*)
Count of all elements in `column`, grouped by category.

Parameters **column** : str

Name of the column to aggregate over. Column data type must be categorical. Resulting aggregate has a outer dimension axis along the categories present.

class datashader.reductions.**summary**(***kwargs*)
A collection of named reductions.

Computes all aggregates simultaneously, output is stored as a `xarray.Dataset`.

Examples

A reduction for computing the mean of column "a", and the sum of column "b" for each bin, all in a single pass.

```
>>> import datashader as ds
>>> red = ds.summary(mean_a=ds.mean('a'), sum_b=ds.sum('b'))
```

datashader.transfer_functions.**stack**(**imgs*, ***kwargs*)
Combine images together, overlaying later images onto earlier ones.

Parameters **imgs** : iterable of Image

The images to combine.

how : str, optional

The compositing operator to combine pixels. Default is *'over'*.

datashader.transfer_functions.**shade**(*agg, cmap=['lightblue', 'darkblue'], color_key=['#e41a1c', '#377eb8', '#4daf4a', '#984ea3', '#ff7f00', '#ffff33', '#a65628', '#f781bf', '#999999', '#66c2a5', '#fc8d62', '#8da0cb', '#a6d854', '#ffd92f', '#e5c494', '#ffffb3', '#fb8072', '#fdb462', '#fccde5', '#d9d9d9', '#ccebc5', '#ffed6f'], how='eq_hist', alpha=255, min_alpha=40, span=None, name=None*)

Convert a DataArray to an image by choosing an RGBA pixel color for each value.

Requires a DataArray with a single data dimension, here called the "value", indexed using either 2D or 3D coordinates.

For a DataArray with 2D coordinates, the RGB channels are computed from the values by interpolated lookup into the given colormap `cmap`. The A channel is then set to the given fixed `alpha` value for all non-zero values, and to zero for all zero values.

DataArrays with 3D coordinates are expected to contain values distributed over different categories that are indexed by the additional coordinate. Such an array would reduce to the 2D-coordinate case if collapsed across the categories (e.g. if one did `aggc.sum(dim='cat')` for a categorical dimension `cat`). The RGB channels for the uncollapsed, 3D case are computed by averaging the colors in the provided `color_key` (with one color per category), weighted by the array's value for that category. The A channel is then computed from the array's total value collapsed across all categories at that location, ranging from the specified `min_alpha` to the maximum alpha value (255).

Parameters **agg** : DataArray

cmap : list of colors or matplotlib.colors.Colormap, optional

The colormap to use for 2D agg arrays. Can be either a list of colors (specified either by name, RGBA hexcode, or as a tuple of `(red, green, blue)` values.), or a matplotlib colormap object. Default is `["lightblue", "darkblue"]`.

color_key : dict or iterable

The colors to use for a 3D (categorical) agg array. Can be either a `dict` mapping from field name to colors, or an iterable of colors in the same order as the record fields, and including at least that many distinct colors.

how : str or callable, optional

The interpolation method to use, for the `cmap` of a 2D DataArray or the alpha channel of a 3D DataArray. Valid strings are 'eq_hist' [default], 'cbrt' (cube root), 'log' (logarithmic), and 'linear'. Callables take 2 arguments - a 2-dimensional array of magnitudes at each pixel, and a boolean mask array indicating missingness. They should return a numeric array of the same shape, with `NaN` values where the mask was True.

alpha : int, optional

Value between 0 - 255 representing the alpha value to use for colormapped pixels that contain data (i.e. non-NaN values). Regardless of this value, `NaN` values are set to be fully transparent when doing colormapping.

min_alpha : float, optional

The minimum alpha value to use for non-empty pixels when doing colormapping, in [0, 255]. Use a higher value to avoid undersaturation, i.e. poorly visible low-value datapoints, at the expense of the overall dynamic range.

span : list of min-max range, optional

Min and max data values to use for colormap interpolation, when wishing to override autoranging.

name : string name, optional

Optional string name to give to the Image object to return, to label results for display.

datashader.transfer_functions.**set_background**(*img*, *color=None*, *name=None*)

Return a new image, with the background set to *color*.

Parameters **img** : Image

color : color name or tuple, optional

The background color. Can be specified either by name, hexcode, or as a tuple of `(red, green, blue)` values.

datashader.transfer_functions.**spread**(*img*, *px=1*, *shape='circle'*, *how='over'*, *mask=None*, *name=None*)

Spread pixels in an image.

Spreading expands each pixel a certain number of pixels on all sides according to a given shape, merging pixels using a specified compositing operator. This can be useful to make sparse plots more visible.

Parameters **img** : Image

px : int, optional

Number of pixels to spread on all sides

shape : str, optional

The shape to spread by. Options are 'circle' [default] or 'square'.

how : str, optional

The name of the compositing operator to use when combining pixels.

mask : ndarray, shape (M, M), optional

The mask to spread over. If provided, this mask is used instead of generating one based on *px* and *shape*. Must be a square array with odd dimensions. Pixels are spread from the center of the mask to locations where the mask is True.

name : string name, optional

Optional string name to give to the Image object to return, to label results for display.

datashader.transfer_functions.**dynspread**(*img*, *threshold=0.5*, *max_px=3*, *shape='circle'*, *how='over'*, *name=None*)

Spread pixels in an image dynamically based on the image density.

Spreading expands each pixel a certain number of pixels on all sides according to a given shape, merging pixels using a specified compositing operator. This can be useful to make sparse plots more visible. Dynamic spreading determines how many pixels to spread based on a density heuristic. Spreading starts at 1 pixel, and stops when the fraction of adjacent non-empty pixels reaches the specified threshold, or the max_px is reached, whichever comes first.

Parameters **img** : Image

threshold : float, optional

A tuning parameter in [0, 1], with higher values giving more spreading.

max_px : int, optional

Maximum number of pixels to spread on all sides.

shape : str, optional

The shape to spread by. Options are 'circle' [default] or 'square'.

how : str, optional

The name of the compositing operator to use when combining pixels.

2.2 Performance

Datashader is designed to make it simple to work with even very large datasets. To get good performance, it is essential that each step in the overall processing pipeline be set up appropriately. Below we share some of our suggestions based on our own Benchmarking and optimization experience, which should help you obtain suitable performance in your own work.

2.2.1 File formats

Based on our testing with various file formats, we recommend storing any large columnar datasets in the Apache Parquet format when possible, using the fastparquet library with "Snappy" compression:

```
>>> import dask.dataframe as dd
>>> dd.to_parquet(filename, df, compression="SNAPPY")
```

If your data includes categorical values that take on a limited, fixed number of possible values (e.g. "Male", "Female"), With parquet, categorical columns use a more memory-efficient data representation and are optimized for common operations such as sorting and finding uniques. Before saving, just convert the column as follows:

```
>>> df[colname] = df[colname].astype('category')
```

By default, numerical datasets typically use 64-bit floats, but many applications do not require 64-bit precision when aggregating over a very large number of datapoints to show a distribution. Using 32-bit floats reduces storage and memory requirements in half, and also typically greatly speeds up computations because only half as much data needs to be accessed in memory. If applicable to your particular situation, just convert the data type before generating the file:

```
>>> df[colname] = df[colname].astype(numpy.float32)
```

2.2.2 Single machine

Datashader supports both Pandas and Dask dataframes, but Dask dataframes typically give higher performance even on a single machine, because it makes good use of all available cores, and it also supports out-of-core operation for datasets larger than memory.

Dasks works on chunks of the data at any one time, called partitions. With dask on a single machine, a rule of thumb for the number of partitions to use is `multiprocessing.cpu_count()`, which allows Dask to use one thread per core for parallelizing computations.

When the entire dataset fits into memory at once, you can persist the data as a Dask dataframe prior to passing it into datashader, to ensure that data only needs to be loaded once:

```
>>> from dask import dataframe as dd
>>> import multiprocessing as mp
>>> dask_df = dd.from_pandas(df, npartitions=mp.cpu_count())
>>> dask_df.persist()
...
>>> cvs = datashader.Canvas(...)
>>> agg = cvs.points(dask_df, ...)
```

When the entire dataset doesn't fit into memory at once, you should not use *persist*. In our tests of this scenario, dask's distributed scheduler gave better performance than the default scheduler, even on single machines:

```
>>> from dask import distributed
>>> import multiprocessing as mp
>>> cluster = distributed.LocalCluster(n_workers=mp.cpu_count(), threads_per_worker=1)
>>> dask_client = distributed.Client(cluster)
>>> dask_df = dd.from_pandas(df, npartitions=mp.cpu_count()) # Note no "persist"
...
>>> cvs = datashader.Canvas(...)
>>> agg = cvs.points(dask_df, ...)
```

2.2.3 Multiple machines

To use multiple nodes (different machines) at once, you can use a Dask dataframe with the distributed scheduler as shown above. `client.persist(dask_df)` may help in certain cases, but if you are doing profiling of the aggregation step, be sure to include `distributed.wait()` to block until the data is read into RAM on each worker.

Python Module Index

d

datashader.transfer_functions, 9

Index

A

any (class in datashader.reductions), 8

C

Canvas (class in datashader), 7
count (class in datashader.reductions), 8
count_cat (class in datashader.reductions), 9

D

datashader.transfer_functions (module), 9
dynspread() (in module datashader.transfer_functions), 11

L

Line (class in datashader.glyphs), 8

M

m2 (class in datashader.reductions), 8
max (class in datashader.reductions), 8
mean (class in datashader.reductions), 9
min (class in datashader.reductions), 8

P

Pipeline (class in datashader), 7
Point (class in datashader.glyphs), 8

S

set_background() (in module datashader.transfer_functions), 11
shade() (in module datashader.transfer_functions), 9
spread() (in module datashader.transfer_functions), 11
stack() (in module datashader.transfer_functions), 9
std (class in datashader.reductions), 9
sum (class in datashader.reductions), 8
summary (class in datashader.reductions), 9

V

var (class in datashader.reductions), 9