Blocks Documentation

Release 0.6.5

Bradley Axen

May 29, 2019

Contents

Blocks provides a simple interface to read, organize, and manipulate structured data in files on local and cloud storage

CHAPTER 1

Install

```
pip install sq-blocks
```

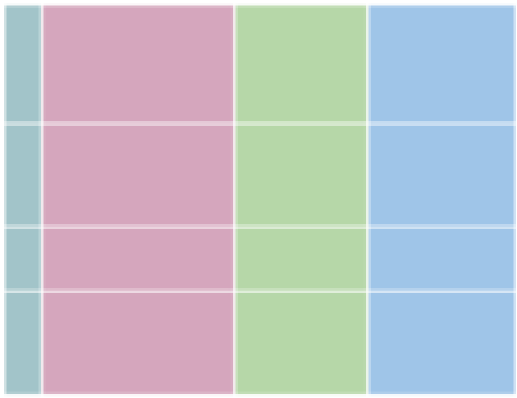
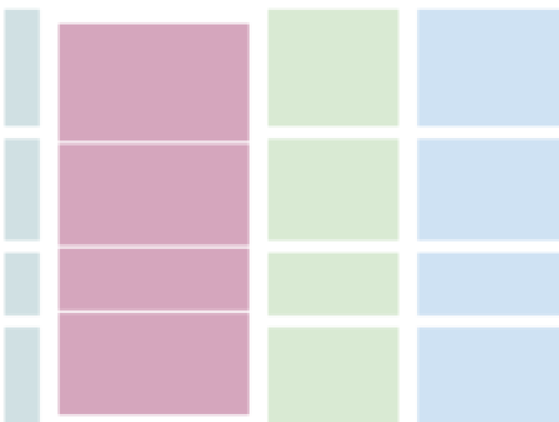
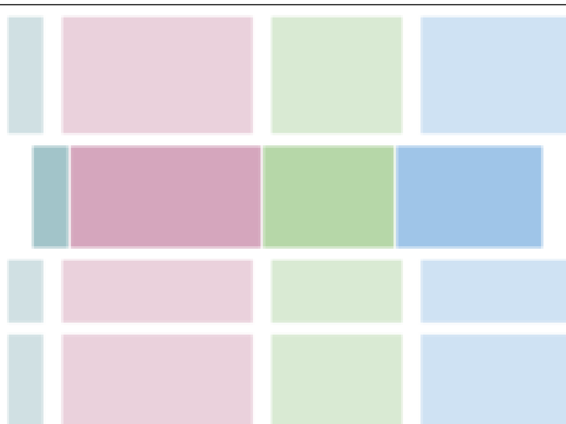
To enable GCS support make sure to also install the Google Cloud SDK

CHAPTER 2

Features

```
import blocks

# Load one or more files with the same interface
df = blocks.assemble('data.csv')
train = blocks.assemble('data/*[01].csv')
test = blocks.assemble('data/*[2-9].csv')

# With direct support for files on GCS
df = blocks.assemble('gs://mybucket/data.csv')
df = blocks.assemble('gs://mybucket/data/*.csv')
```

The interface emulates the tools you're used to from the command line, with full support for globbing and pattern matching. And blocks can handle more complicated structures as your data grows in complexity:

Layout	Recipe
	blocks.assemble('data/**')``
	blocks.assemble('data/g1/*')
	blocks.assemble('data/*/part_01.pq')
	

CHAPTER 3

Full Contents

3.1 Quickstart

3.1.1 Layout

In the simplest case, you might want to read your data from a single file. This is pretty easy in pandas, but blocks adds additional support for inferring file types and support cloud storage:

```
import pandas as pd
import blocks
df = blocks.assemble('data.pkl') # same as pd.read_pickle
df = blocks.assemble('gs://mybucket/data.parquet')
```

Many projects need to combine data stored in several files. To support this, blocks makes a few assumptions about your data. You've split it up into blocks, either into groups of columns (cgroups) or groups of rows (rgroups). You can read all this data into a single dataframe in memory with one command:

```
import blocks
blocks.assemble('data/')
```

If all of your files are in one directory, then the rows will be concatenated:

```
data
├── part.00.pq
├── part.01.pq
└── part.02.pq
```

If your files actually contain the same rows but store different columns, you should place them in different folders with corresponding names:

```
data
├── g0
│   └── part.00.pq
```

(continues on next page)

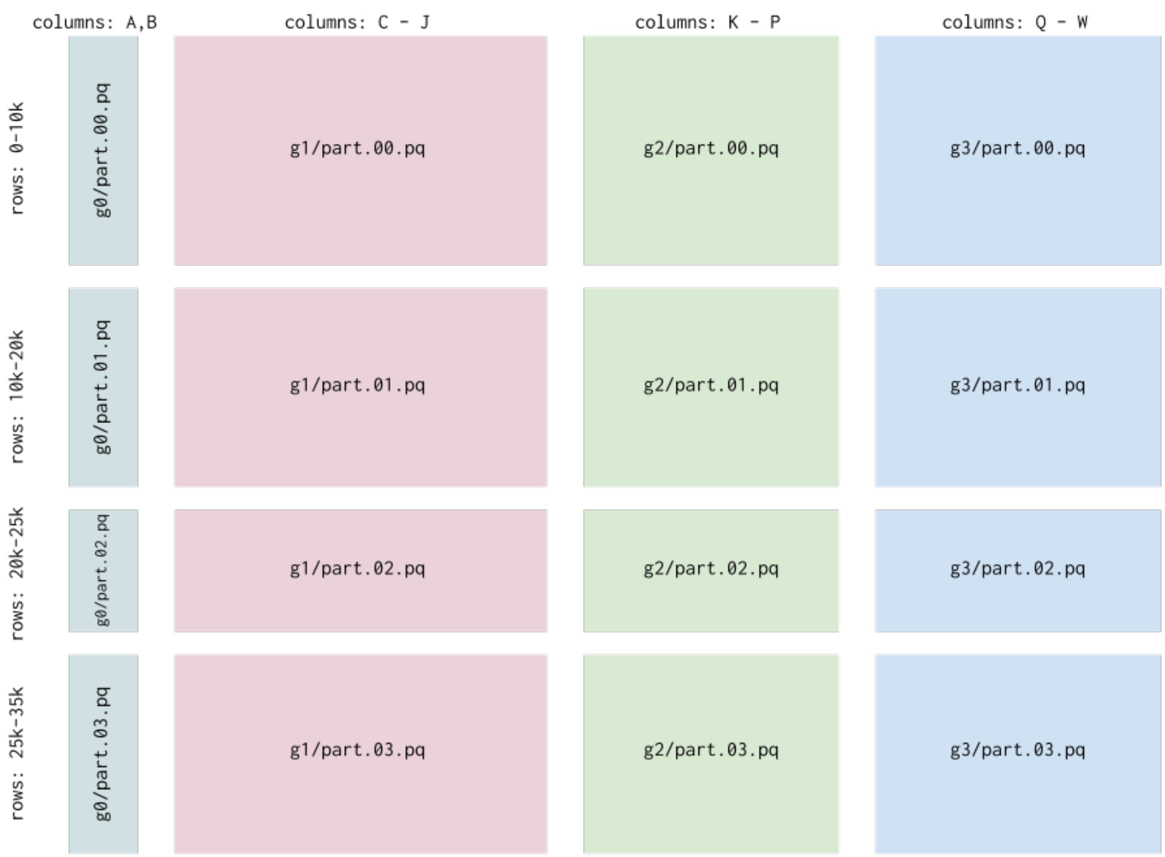
(continued from previous page)

```
├── g1
│   └── part.00.pq
└── g2
    └── part.00.pq
```

In the most general case you can do both, laying out your data in multiple cgroups and rgroups - where each rgroup should contain the same logical rows (e.g. different attributes of the same event)

```
- data
  ├── g0
  │   ├── part.00.pq
  │   ├── part.01.pq
  │   ├── part.02.pq
  │   └── part.03.pq
  ├── g1
  │   ├── part.00.pq
  │   ├── part.01.pq
  │   ├── part.02.pq
  │   └── part.03.pq
  ├── g2
  │   ├── part.00.pq
  │   ├── part.01.pq
  │   ├── part.02.pq
  │   └── part.03.pq
  └── g3
      ├── part.00.pq
      ├── part.01.pq
      ├── part.02.pq
      └── part.03.pq
```

This corresponds to the following dataframe structure:



This pattern generalizes very well when you start collecting data from multiple sources and with enough content that the entire dataset won't comfortably fit into memory at once.

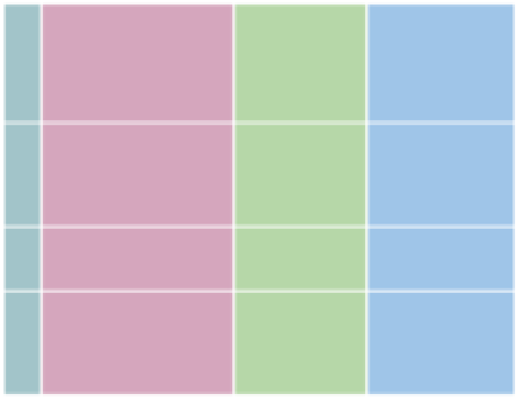
Blocks supports multiple data formats, including `csv`, `hdf5`, `pickle`, and `parquet`. Reads from these files are handled by `pandas` libraries, so they support all of the options you expect like headers, index columns, etc. All of the `blocks` interfaces below support passing keyword args to the read functions for the files (see the docstrings). The files can be local (referenced by normal paths) or on GCS (referenced by paths like `gs://bucket`).

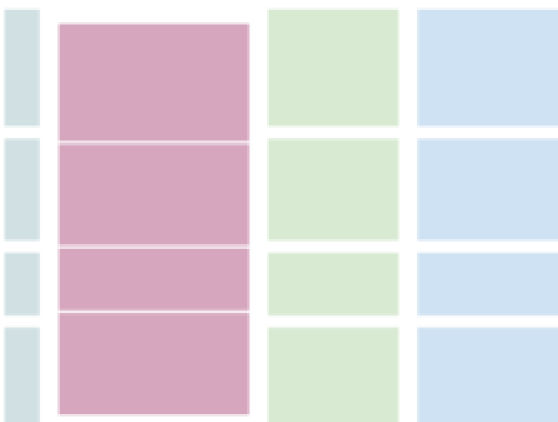
Note that rgroups are combined by simple concatenation, and cgroups are combined by a "natural left join": any shared columns are considered join keys. Key-based merging only makes sense with named columns, so make sure any CSVs you use have a column header if you want to join cgroups.

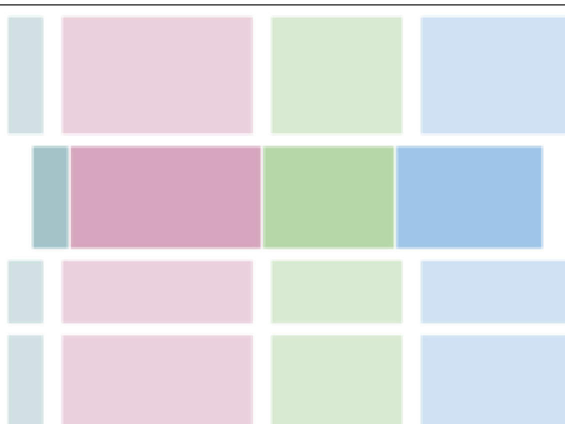
3.1.2 Read

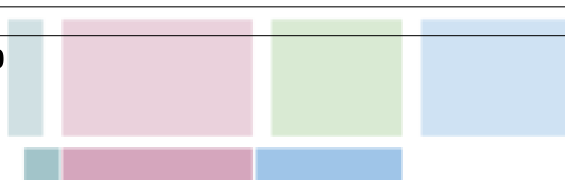
Assemble

Assemble is the primary data reading command, and can handle any of the layouts above. You can select subsets of the data using glob patterns or the `cgroups` and `rgroups` arguments:

Layout	Recipe
	`blocks.assemble('data/')`

	`blocks.assemble('data/g1/*')`
	`# or`
	`blocks.assemble('data/', cgroups=['g1'])`

	`blocks.assemble('data/*/part.01.pq')`
	`# or`
	`blocks.assemble('data/', rgroups=['part.`
	`↪01.pq'])`

Iterate

Blocks also has an iterative option for performing operations on each of the blocks without loading them all into memory at once:

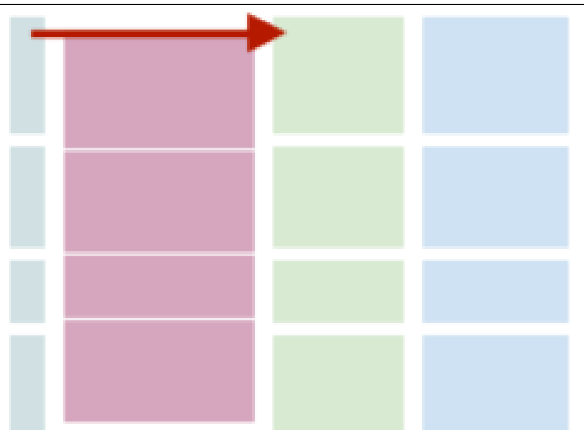
```
import blocks

for cgroup, rgroup, df in blocks.iterate('data/'):
    print(df.shape)
```

`iterate` supports the same syntax and features as `assemble` above, but instead of returning a merged dataframe, it returns an iterator of `(rgroup, cgroup, dataframe)` where the `rgroup` and `cgroup` are the names of the groups (`'g0'` and `'part.00.pq'` from above).

`iterate` can also operate on multiple axes - the default is to iterate over every block separately. But if you specify `axis=0`, then iterate will combine cgroups and iterate over rgroups, and for `axis=1` it will iterate over the cgroups while combining any rgroups.

<table>
<tr><th>Direction</th><th>Recipe</th></tr>
<tr><td></td><td>

```
# iterate over one dataframe per rgroup
for rgroup, df in  blocks.iterate('gs://
↪path/to/data', axis=0):
    print(df.shape)
```

</td></tr>
<tr><td></td><td>

```
# iterate over one dataframe per cgroup
for cgroup, df in  blocks.iterate('gs://
↪path/to/data', axis=1):
    print(df.shape)
```

</td></tr>
</table>

Partitioned

Dask provides a great interface to a partitioned dataframe, and you can use blocks' simple syntax to build a `dask.dataframe`. Checkout the dask documentation for details on how to use the resulting object.

```
import blocks

# need to have separately installed dask
dask_df = blocks.partitioned('data/*/part_0[1-4].pq')

dask_df.groupby('category').mean().compute()
```

3.1.3 Write

Place

If you want to put a dataframe into a single file, use `place`:

```
import blocks

blocks.place(df, 'data/part_00.pq')
blocks.place(df, 'gs://mybucket/data/part_00.pq')
```

Like with `assemble` for a single file, this is easy in `pandas`, but `blocks` infers the file type and has support for cloud storage.

Divide

For paritioning your data, blocks also has a divide function. You'd use this to split up a single large dataframe in memory into many rgroups and/or cgroups on disk, to help with parallelizing analysis. By default the blocks are written as `parquet` files, but you can specify other extensions including `.hdf5`, `.csv`, and `.pkl`.

```
import blocks

# divide into just row groups
blocks.divide(df, 'data/', n_rgroup=3)
```

```
data
├── part_00.pq
├── part_01.pq
└── part_02.pq
```

Divide can also handle column groups:

```
# split into 10 rgroups and specific cgroups
cgroup_columns = {
    'g0': ['id', 'timestamp', 'metadata'],
    'g1': ['id', 'timestamp', 'feature0', 'feature1'],
    'g2': ['id', 'timestamp', 'feature2', 'feature3'],
    'g3': ['id', 'timestamp', 'feature4', 'feature5', 'feature6'],
}
blocks.divide(df, 'data/', 4, cgroup_columns=cgroup_columns)
```

```
─ data
  ├── g0
  │   ├── part.00.pq
  │   ├── part.01.pq
  │   ├── part.02.pq
  │   └── part.03.pq
  ├── g1
  │   ├── part.00.pq
  │   ├── part.01.pq
  │   ├── part.02.pq
  │   └── part.03.pq
  ├── g2
  │   ├── part.00.pq
  │   ├── part.01.pq
```

(continues on next page)

(continued from previous page)

```
│       ├── part.02.pq
│       └── part.03.pq
└── g3
        ├── part.00.pq
        ├── part.01.pq
        ├── part.02.pq
        └── part.03.pq
```

3.2 Examples

3.2.1 Inspect Data

You can use assemble to grab a small subset of your data

```
import blocks

df = blocks.assemble('data/*/part_00.pq')
df.describe()
```

This works great when dealing with data staged on GCS

```
import blocks

df = blocks.assemble('gs://bucket/*/part_00.pq')
df.describe()
```

3.2.2 Large Datasets

It's common to end up with a dataset that won't easily fit into memory. But you often still need to calculate aggregate statistics on that data. For example, you might need to get a unique list of categories in one of your fields.

Iterate makes this easy:

```
import blocks

uniques = set()
for _, _, block in blocks.iterate('data/'):
    uniques |= set(block['feature'])
```

or maybe you want to parallelize the process

```
import blocks
from multiprocessing import Pool

def unique_f1(block):
    return set(block[-1]['feature'])

uniques_per_block = Pool(4).map(unique_f1, blocks.iterate('data/'))
uniques = reduce(lambda a, b: a | b, uniques_per_block)
```

And if you have dask installed the parallelization is even easier

```
import blocks

uniques = blocks.partitioned('data')['feature'].unique().compute()
```

3.2.3 Batch Training

If you're working with a tool like Keras, you might want to train a model on an iterator of batches without every loading more than one partition into memory:

```
import blocks

def batch_generator(path):
    for _, df in blocks.iterate(path, axis=0):
        while df.shape[0] >= nbatch:
            # Grab a sample and drop from original
            sub = df.sample(nbatch)
            df.drop(sub.index, inplace=True)
            yield sub.values

model.fit_generator(
    generator=batch_generator('train/'),
    validation_data=batch_generator('validate/'),
)
```

If you use an efficient file format like `parquet`, this simple code will be suprisingly fast. You should make sure that you don't use multiple cgroups in a situation like this, however, because merging can slow down the process.

3.2.4 Combining

If you end up with a dataset with multiple column groups, say because you grabbed your data from multiple sources, you may want to merge accross those groups. However it is expensive to do this by loading the whole dataset into memory. If you use the blocks structure you can merge each row partition separately and then save to new files. You can even subdivide those files into smaller row groups to ensure that they don't grow too large:

```
import blocks

offset = 0
for _, df in blocks.iterate(path, axis=0):
    blocks.divide(df, 'combined/', n_rgroup=10, rgroup_offset=offset)
    rgroup_offset += 10
```

3.2.5 Filesystem

Blocks provide a default filesystem that supports local files and GCS files. If you need additional functionality, you can create a custom filesystem instance:

```
import blocks
from blocks.filesystem import GCSFileSystem

fs = GCSFileSystem()
df = blocks.assemble('gs://bucket/data/', filesystem=fs)
```

The default filesystem has support for GCS, and you can implement your own FileSystem class by inheriting from `blocks.filesystem.FileSystem`. This can be used to extend blocks to additional cloud platforms, to support encryption/decryption, etc...

3.3 Core

`blocks.core.assemble`(*path*, *cgroups=None*, *rgroups=None*, *read_args={}*, *cgroup_args={}*, *merge='inner'*, *filesystem=<blocks.filesystem.GCSFileSystem object at 0x7f0563e7c090>*)

Assemble multiple dataframe blocks into a single frame

Each file included in the path (or subdirs of that path) is combined into a single dataframe by first concatenating over row groups and then merging over cgroups. The merges are performed in the order of listed cgroups if provided, otherwise in alphabetic order. Files are opened by a method inferred from their extension

Parameters

path [str] The glob-able path to all datafiles to assemble into a frame e.g. gs://example//, gs://example/*/part.0.pq, gs://example/c[1-2]/* See the README for a more detailed explanation

cgroups [list of str, optional] The list of cgroups (folder names) to include from the glob path

rgroups [list of str, optional] The list of rgroups (file names) to include from the glob path

read_args [optional] Any additional keyword args to pass to the read function

cgroup_args [{cgroup: kwargs}, optional] Any cgroup specific read arguments, where each key is the name of the cgroup and each value is a dictionary of keyword args

merge [one of 'left', 'right', 'outer', 'inner', default 'inner'] The merge strategy to pass to pandas.merge

filesystem [blocks.filesystem.FileSystem or similar] A filesystem object that implements the blocks.FileSystem API

Returns

data [pd.DataFrame] The combined dataframe from all the blocks

`blocks.core.divide`(*df*, *path*, *n_rgroup=1*, *rgroup_offset=0*, *cgroup_columns=None*, *extension='.pq'*, *convert=False*, *filesystem=<blocks.filesystem.GCSFileSystem object at 0x7f0562f277d0>*, *prefix=None*, ***write_args*)

Split a dataframe into rgroups/cgroups and save to disk

Note that this splitting does not preserve the original index, so make sure to have another column to track values

Parameters

df [pd.DataFrame] The data to divide

path [str] Path to the directory (possibly on GCS) in which to place the columns

n_rgroup [int, default 1] The number of row groups to partition the data into The rgroups will have approximately equal sizes

rgroup_offset [int, default 0] The index to start from in the name of file parts e.g. If rgroup_offset=10 then the first file will be *part_00010.pq*

cgroup_columns [{cgroup: list of column names}] The column lists to form cgroups; if None, do not make cgroups Each key is the name of the cgroup, and each value is the list of columns to include To reassemble later make sure to include join keys for each cgroup

extension [str, default .pq] The file extension for the dataframe (file type inferred from this extension

convert [bool, default False] If true attempt to coerce types to numeric. This can avoid issues with ambiguous object columns but requires additional time

filesystem [blocks.filesystem.FileSystem or similar] A filesystem object that implements the blocks.FileSystem API

write_args [dict] Any additional args to pass to the write function

`blocks.core.`**`iterate`**(*path*, *axis=-1*, *cgroups=None*, *rgroups=None*, *read_args={}*, *cgroup_args={}*, *merge='inner'*, *filesystem=<blocks.filesystem.GCSFileSystem object at 0x7f0562f27710>*)

Iterate over dataframe blocks

Each file include in the path (or subdirs of that path) is opened as a dataframe and returned in a generator of (cname, rname, dataframe). Files are opened by a method inferred from their extension

Parameters

path [str] The glob-able path to all datafiles to assemble into a frame e.g. gs://example//, gs://example/*/part.0.pq, gs://example/c[1-2]/* See the README for a more detailed explanation

axis [int, default -1] The axis to iterate along If -1 (the default), iterate over both columns and rows If 0, iterate over the rgroups, combining any cgroups If 1, iterate over the cgroups, combining any rgroups

cgroups [list of str, or {str: args} optional] The list of cgroups (folder names) to include from the glob path

rgroups [list of str, optional] The list of rgroups (file names) to include from the glob path

read_args [dict, optional] Any additional keyword args to pass to the read function

cgroup_args [{cgroup: kwargs}, optional] Any cgroup specific read arguments, where each key is the name of the cgroup and each value is a dictionary of keyword args

merge [one of 'left', 'right', 'outer', 'inner', default 'inner'] The merge strategy to pass to pandas.merge, only used when axis=0

filesystem [blocks.filesystem.FileSystem or similar] A filesystem object that implements the blocks.FileSystem API

Returns

data [generator] A generator of (cname, rname, dataframe) for each collected path If axis=0, yields (rname, dataframe) If axis=1, yields (cname, dataframe)

`blocks.core.`**`partitioned`**(*path*, *cgroups=None*, *rgroups=None*, *read_args={}*, *cgroup_args={}*, *merge='inner'*, *filesystem=<blocks.filesystem.GCSFileSystem object at 0x7f0562f27750>*)

Return a partitioned dask dataframe, where each partition is a row group

The results are the same as iterate with axis=0, except that it returns a dask dataframe instead of a generator. Note that this requires dask to be installed

Parameters

path [str] The glob-able path to all datafiles to assemble into a frame e.g. gs://example//, gs://example/*/part.0.pq, gs://example/c[1-2]/* See the README for a more detailed explanation

cgroups [list of str, or {str: args} optional] The list of cgroups (folder names) to include from the glob path

rgroups [list of str, optional] The list of rgroups (file names) to include from the glob path

read_args [dict, optional] Any additional keyword args to pass to the read function

cgroup_args [{cgroup: kwargs}, optional] Any cgroup specific read arguments, where each key is the name of the cgroup and each value is a dictionary of keyword args

merge [one of 'left', 'right', 'outer', 'inner', default 'inner'] The merge strategy to pass to pandas.merge, only used when axis=0

filesystem [blocks.filesystem.FileSystem or similar] A filesystem object that implements the blocks.FileSystem API

Returns

data [dask.dataframe] A dask dataframe partitioned by row groups, with all cgroups merged

`blocks.core.`**`place`**(*df*, *path*, *filesystem=<blocks.filesystem.GCSFileSystem object at 0x7f0562f27790>*, ***write_args*)

Place a dataframe block onto the filesystem at the specified path

Parameters

df [pd.DataFrame] The data to place

path [str] Path to the directory (possibly on GCS) in which to place the columns

write_args [dict] Any additional args to pass to the write function

filesystem [blocks.filesystem.FileSystem or similar] A filesystem object that implements the blocks.FileSystem API

3.4 Filesystem

class `blocks.filesystem.`**`DataFile`**

Bases: `tuple`

Attributes

`handle` Alias for field number 1

`path` Alias for field number 0

Methods

`count`()	
`index`()	Raises ValueError if the value is not present.

`handle`

Alias for field number 1

`path`

Alias for field number 0

class `blocks.filesystem.`**`FileSystem`**

Bases: `object`

The required interface for any filesystem implementation

See GCSFileSystem for a full implementation. This FileSystem is intended to be extendable to support cloud file systems, encryption strategies, etc. . .

Methods

`access`(self, paths)	Access multiple paths as file-like objects
`ls`(self, path)	List files correspond to path, including glob wildcards
`store`(self, bucket, files)	Store multiple data objects

access (*self*, *paths*)
Access multiple paths as file-like objects

This allows for optimization like parallel downloads

Parameters

paths: list of str The paths of the files to access

Returns

files: list of DataFile A list of datafile instances, one for each input path

ls (*self*, *path*)
List files correspond to path, including glob wildcards

Parameters

path [str] The path to the file or directory to list; supports wildcards

store (*self*, *bucket*, *files*)
Store multiple data objects

This allows for optimizations when storing several files

Parameters

bucket [str] The GCS bucket to use to store the files

files [list of str] The file names to store

Returns

datafiles [contextmanager] A contextmanager that will yield datafiles and place them on the filesystem when finished

class blocks.filesystem.**GCSFileSystem** (*parallel=True*, *quiet=True*)
Bases: `blocks.filesystem.FileSystem`

File system interface that supports both local and GCS files

This implementation uses subprocess and gsutil, which has excellent performance. However this can lead to problems in very multi-threaded applications and might not be as portable. For a python native implementation use GCSNativeFileSystem

Methods

`access`(self, paths)	Access multiple paths as file-like objects
`cp`(self, sources, dest[, recursive])	Copy the files in sources to dest
`local`(self, path)	Check if the path is available as a local file
`ls`(self, path)	List files correspond to path, including glob wildcards
`open`(*args, **kwds)	Access path as a file-like object
`rm`(self, paths[, recursive])	Remove the files at paths
`store`(*args, **kwds)	Create file stores that will be written to the filesystem on close

GCS = 'gs://'

access (*self*, *paths*)

Access multiple paths as file-like objects

This allows for optimization like parallel downloads

Parameters

paths: list of str The paths of the files to access

Returns

files: list of DataFile A list of datafile instances, one for each input path

cp (*self*, *sources*, *dest*, *recursive=False*)

Copy the files in sources to dest

Parameters

sources [list of str] The list of paths to copy

dest [str] The destination for the copy of source(s)

recursive [bool] If true, recursively copy any directories

local (*self*, *path*)

Check if the path is available as a local file

ls (*self*, *path*)

List files correspond to path, including glob wildcards

Parameters

path [str] The path to the file or directory to list; supports wildcards

open (**args*, ***kwds*)

Access path as a file-like object

Parameters

path: str The path of the file to access

mode: str The file mode for the opened file

Returns

file: file A python file opened to the provided path (uses a local temporary copy that is removed)

rm (*self*, *paths*, *recursive=False*)

Remove the files at paths

Parameters

paths [list of str] The paths to remove

recursive [bool, default False] If true, recursively remove any directories

store (**args*, ***kwds*)

Create file stores that will be written to the filesystem on close

This allows for optimizations when storing several files

Parameters

bucket [str] The path of the bucket (on GCS) or folder (local) to store the data in

files [list of str] The filenames to create

Returns

datafiles [contextmanager] A context manager that yields datafiles and when the context is closed they are written to GCS

class blocks.filesystem.**GCSNativeFileSystem** (**args*, ***kwargs*)

Bases: *blocks.filesystem.GCSFileSystem*

File system interface that supports GCS and local files

This uses the native python cloud storage library for read and write, rather than gsutil. The performance is significantly slower when doing any operations over several files (especially copy), but is thread-safe for applications which are already parallelized. It stores the files entirely in memory rather than using tempfiles.

Methods

access(self, paths)	Access multiple paths as file-like objects
cp(self, sources, dest[, recursive])	Copy the files in sources (recursively) to dest
local(self, path)	Check if the path is available as a local file
ls(self, path)	List all files at the specified path, supports globbing
open(*args, **kwds)	Access paths as a file-like object
rm(self, paths[, recursive])	Remove the files at paths
store(*args, **kwds)	Create file stores that will be written to the filesystem on close

client	
copy_single	
is_dir	
rm_single	

access (*self*, *paths*)

Access multiple paths as file-like objects

This allows for optimization like parallel downloads. To help track which files came from which objects, this returns instances of Datafile

Parameters

paths: list of str The paths of the files to access

Returns

files: list of DataFile A list of datafile instances, one for each input path

client (*self*)

copy_single (*self*, *source*, *dest*)

cp (*self*, *sources*, *dest*, *recursive=False*)
Copy the files in sources (recursively) to dest

Parameters

sources [list of str] The list of paths to copy, which can be directories

dest [str] The destination for the copy of source(s)

recursive [bool, default False] If true, recursively copy directories

is_dir (*self*, *path*)

ls (*self*, *path*)
List all files at the specified path, supports globbing

open (**args*, ***kwds*)
Access paths as a file-like object

Parameters

path: str The path of the file to access

mode: str The file mode for the opened file

Returns

file: BytesIO A BytesIO handle for the specified path, works like a file object

rm (*self*, *paths*, *recursive=False*)
Remove the files at paths

Parameters

paths [list of str] The paths to remove

recursive [bool, default False] If true, recursively remove any directories

rm_single (*self*, *path*)

store (**args*, ***kwds*)
Create file stores that will be written to the filesystem on close

This allows for optimizations when storing several files

Parameters

bucket [str] The path of the bucket (on GCS) or folder (local) to store the data in

files [list of str] The filenames to create

Returns

datafiles [contextmanager] A context manager that yields datafiles and when the context is closed they are written to GCS

Python Module Index

b

Index