Sparsity Documentation

Release 0.19.0

Datarevenue

Sep 10, 2018

Contents

Sparsity builds on top of Pandas and Scipy to provide DataFrame-like API to work with numerical homogeneous sparse data.

Sparsity provides Pandas-like indexing capabilities and group transformations on Scipy csr matrices. This has proven to be extremely efficient as shown below.

Furthermore we provide a distributed implementation of this data structure by relying on the Dask framework. This includes distributed sorting, partitioning, grouping and much more.

Although we try to mimic the Pandas DataFrame API, some operations and parameters don't make sense on sparse or homogeneous data. Thus some interfaces might be changed slightly in their semantics and/or inputs.

CHAPTER 1

Install

Sparsity is available from PyPi:

```
# Install using pip
$ pip install sparsity
```

CHAPTER 2

Contents

2.1 About Sparsity

2.1.1 Motivation

Many tasks, especially in data analytics and machine learning domain, make use of sparse data structures to support the input of high dimensional data.

This project was started to build an efficient homogeneous sparse data processing pipeline. As of today Dask has no support for something as a sparse dataframe. We process big amounts of high-dimensional data on a daily basis at Datarevenue and our favourite language and ETL framework are Python and Dask. After chaining many function calls on scipy.sparse csr matrices that involved handling of indices and column names to produce a sparse data pipeline, we decided to start this project.

This package might be especially useful to you if you have very big amounts of sparse data such as clickstream data, categorical timeseries, log data or similarly sparse data.

2.1.2 Comparison to Pandas SparseDataFrame

Pandas has its own implementation of sparse data structures. Unfortunately this structures perform quite badly with a groupby-sum aggregation which we use frequently. Furthermore doing a groupby on a Pandas SparseDataFrame returns a dense DataFrame. This makes chaining many groupby operations over multiple files cumbersome and less efficient. Consider following example:

```
In [1]: import sparsity
   ...: import pandas as pd
   ...: import numpy as np
   ...:

In [2]: data = np.random.random(size=(1000,10))
   ...: data[data < 0.95] = 0
   ...: uids = np.random.randint(0,100,1000)
```

(continues on next page)

(continued from previous page)

```
   ...: combined_data = np.hstack([uids.reshape(-1,1),data])
   ...: columns = ['id'] + list(map(str, range(10)))
   ...:
   ...: sdf = pd.SparseDataFrame(combined_data, columns = columns, default_fill_
↪value=0)
   ...:

In [3]: %%timeit
   ...: sdf.groupby('id').sum()
   ...:
1 loop, best of 3: 462 ms per loop

In [4]: res = sdf.groupby('id').sum()
   ...: res.values.nbytes
   ...:
Out[4]: 7920

In [5]: data = np.random.random(size=(1000,10))
   ...: data[data < 0.95] = 0
   ...: uids = np.random.randint(0,100,1000)
   ...: sdf = sparsity.SparseFrame(data, columns=np.asarray(list(map(str,␣
↪range(10)))), index=uids)
   ...:

In [6]: %%timeit
   ...: sdf.groupby_sum()
   ...:
The slowest run took 4.20 times longer than the fastest.
1000 loops, best of 3: 1.25 ms per loop

In [7]: res = sdf.groupby_sum()
   ...: res.__sizeof__()
   ...:
Out[7]: 6128
```

2.2 Sparsity User Guide

2.2.1 Creating a SparseFrame

Create a SparseFrame from numpy array:

```
>>> import sparsity
>>> import numpy as np

>>> a = np.random.rand(10, 5)
>>> a[a < 0.9] = 0
>>> sf = sparsity.SparseFrame(a, index=np.arange(10, 20), columns=list('ABCDE'))
>>> sf
      A         B    C         D         E
10  0.0  0.000000  0.0  0.000000  0.000000
11  0.0  0.962851  0.0  0.000000  0.000000
12  0.0  0.858180  0.0  0.867824  0.930348
13  0.0  0.000000  0.0  0.000000  0.968163
14  0.0  0.000000  0.0  0.000000  0.985610
```

(continues on next page)

(continued from previous page)

```
[10x5 SparseFrame of type '<class 'float64'>'
 with 10 stored elements in Compressed Sparse Row format]
```

You can also create a SparseFrame from Pandas DataFrame. Index and columns will be preserved:

```
>>> import pandas as pd

>>> df = pd.DataFrame(a, index=np.arange(10, 20), columns=list('ABCDE'))
>>> sparsity.SparseFrame(df)
      A         B    C         D         E
10  0.0  0.000000  0.0  0.000000  0.000000
11  0.0  0.962851  0.0  0.000000  0.000000
12  0.0  0.858180  0.0  0.867824  0.930348
13  0.0  0.000000  0.0  0.000000  0.968163
14  0.0  0.000000  0.0  0.000000  0.985610
[10x5 SparseFrame of type '<class 'float64'>'
 with 10 stored elements in Compressed Sparse Row format]
```

Initialization from Scipy CSR matrix is also possible. If you don't pass index or columns, defaults will be used:

```
>>> import scipy.sparse

>>> csr = scipy.sparse.rand(10, 5, density=0.1, format='csr')
>>> sparsity.SparseFrame(csr)
          0    1         2    3    4
0  0.638314  0.0  0.000000  0.0  0.0
1  0.000000  0.0  0.000000  0.0  0.0
2  0.000000  0.0  0.043411  0.0  0.0
3  0.000000  0.0  0.000000  0.0  0.0
4  0.000000  0.0  0.222951  0.0  0.0
[10x5 SparseFrame of type '<class 'float64'>'
 with 5 stored elements in Compressed Sparse Row format]
```

2.2.2 Indexing

Indexing a SparseFrame with column name gives a new SparseFrame:

```
>>> sf['A']
      A
10  0.0
11  0.0
12  0.0
13  0.0
14  0.0
[10x1 SparseFrame of type '<class 'float64'>'
 with 0 stored elements in Compressed Sparse Row format]
```

Similarly for a list of column names:

```
>>> sf[['A', 'B']]
      A         B
10  0.0  0.000000
11  0.0  0.962851
12  0.0  0.858180
13  0.0  0.000000
```

(continues on next page)

(continued from previous page)

```
14  0.0  0.000000
[10x2 SparseFrame of type '<class 'float64'>'
 with 3 stored elements in Compressed Sparse Row format]
```

2.2.3 Basic arithmetic operations

Sum, mean, min and max methods are called on underlying Scipy CSR matrix object. They can be computed over whole SparseFrame or along columns/rows:

```
>>> sf.sum(axis=0)
matrix([[0.        , 2.79813655, 0.84659119, 2.8522892 , 2.88412053]])

>>> sf.mean(axis=1)
matrix([[0.        ],
        [0.19257014],
        [0.53127046],
        [0.19363253],
        [0.19712191],
        [0.        ],
        [0.19913979],
        [0.19542124],
        [0.        ],
        [0.36707143]])

>>> sf.min()
0.0

>>> sf.max()
0.9956989680903189
```

Add 2 SparseFrames:

```
>>> sf.add(sf)
      A         B    C         D         E
10  0.0  0.000000  0.0  0.000000  0.000000
11  0.0  1.925701  0.0  0.000000  0.000000
12  0.0  1.716359  0.0  1.735649  1.860697
13  0.0  0.000000  0.0  0.000000  1.936325
14  0.0  0.000000  0.0  0.000000  1.971219
[10x5 SparseFrame of type '<class 'float64'>'
 with 10 stored elements in Compressed Sparse Row format]
```

Multiply each row/column by a number:

```
>>> sf.multiply(np.arange(10), axis='index')
      A         B    C         D         E
10  0.0  0.000000  0.0  0.000000  0.000000
11  0.0  0.962851  0.0  0.000000  0.000000
12  0.0  1.716359  0.0  1.735649  1.860697
13  0.0  0.000000  0.0  0.000000  2.904488
14  0.0  0.000000  0.0  0.000000  3.942438
[10x5 SparseFrame of type '<class 'float64'>'
 with 10 stored elements in Compressed Sparse Row format]

>>> sf.multiply(np.arange(5), axis='columns')
```

(continues on next page)

(continued from previous page)

```
      A         B    C         D         E
10  0.0  0.000000  0.0  0.000000  0.000000
11  0.0  0.962851  0.0  0.000000  0.000000
12  0.0  0.858180  0.0  2.603473  3.721393
13  0.0  0.000000  0.0  0.000000  3.872651
14  0.0  0.000000  0.0  0.000000  3.942438
[10x5 SparseFrame of type '<class 'float64'>'
 with 10 stored elements in Compressed Sparse Row format]
```

2.2.4 Joining

By default SparseFrames are joined on their indexes:

```
>>> sf2 = sparsity.SparseFrame(np.random.rand(3, 2), index=[9, 10, 11], columns=['X',
↪'Y'])
>>> sf2
           X         Y
9   0.182890  0.061269
10  0.039956  0.595605
11  0.407291  0.496680
[3x2 SparseFrame of type '<class 'float64'>'
 with 6 stored elements in Compressed Sparse Row format]

>>> sf.join(sf2)
      A         B    C         D         E         X         Y
9   0.0  0.000000  0.0  0.000000  0.000000  0.182890  0.061269
10  0.0  0.000000  0.0  0.000000  0.000000  0.039956  0.595605
11  0.0  0.962851  0.0  0.000000  0.000000  0.407291  0.496680
12  0.0  0.858180  0.0  0.867824  0.930348  0.000000  0.000000
13  0.0  0.000000  0.0  0.000000  0.968163  0.000000  0.000000
[11x7 SparseFrame of type '<class 'float64'>'
 with 16 stored elements in Compressed Sparse Row format]
```

You can also join on columns:

```
>>> sf3 = sparsity.SparseFrame(np.random.rand(3, 2), index=[97, 98, 99], columns=['E',
↪ 'F'])
>>> sf3
           E         F
97  0.738614  0.958507
98  0.868556  0.230316
99  0.322914  0.587337
[3x2 SparseFrame of type '<class 'float64'>'
 with 6 stored elements in Compressed Sparse Row format]

>>> sf.join(sf3, axis=0).iloc[-5:]
      A    B         C         D         E         F
18  0.0  0.0  0.000000  0.000000  0.000000  0.000000
19  0.0  0.0  0.846591  0.988766  0.000000  0.000000
97  0.0  0.0  0.000000  0.000000  0.738614  0.958507
98  0.0  0.0  0.000000  0.000000  0.868556  0.230316
99  0.0  0.0  0.000000  0.000000  0.322914  0.587337
[5x6 SparseFrame of type '<class 'float64'>'
 with 8 stored elements in Compressed Sparse Row format]
```

2.2.5 Groupby

Groupby-sum operation is optimized for sparse case:

```
>>> df = pd.DataFrame({'X': [1, 1, 1, 0],
...                    'Y': [0, 1, 0, 1],
...                    'gr': ['a', 'a', 'b', 'b'],
...                    'day': [10, 11, 11, 12]})
>>> df = df.set_index(['day', 'gr'])
>>> sf4 = sparsity.SparseFrame(df)
>>> sf4
          X    Y
day gr
10  a   1.0  0.0
11  a   1.0  1.0
    b   1.0  0.0
12  b   0.0  1.0
[4x2 SparseFrame of type '<class 'float64'>'
 with 5 stored elements in Compressed Sparse Row format]

>>> sf4.groupby_sum(level=1)
     X    Y
a  2.0  1.0
b  1.0  1.0
[2x2 SparseFrame of type '<class 'float64'>'
 with 4 stored elements in Compressed Sparse Row format]
```

Operations other then sum can also be applied:

```
>>> sf4.groupby_agg(level=1, agg_func=lambda x: x.mean(axis=0))
     X    Y
a  1.0  0.5
b  0.5  0.5
[2x2 SparseFrame of type '<class 'float64'>'
 with 4 stored elements in Compressed Sparse Row format]
```

2.3 SparseFrame API

SparseFrame(data[, index, columns])	Two dimensional, size-mutable, homogenous tabular data structure with labeled axes (rows and columns).
SparseFrame.add(other[, how, fill_value])	Aligned addition.
SparseFrame.assign(**kwargs)	Assign new columns.
SparseFrame.axes	
SparseFrame.columns	Return column labels
SparseFrame.concat(tables[, axis])	Concat a collection of SparseFrames along given axis.
SparseFrame.copy(*args[, deep])	Copy frame
SparseFrame.drop(labels[, axis])	Drop label(s) from given axis.
SparseFrame.dropna()	Drop nans from index.
SparseFrame.fillna(value)	Replace NaN values in explicitly stored data with *value*.
SparseFrame.groupby_agg([by, level, agg_func])	Aggregate data using callable.
SparseFrame.groupby_sum([by, level])	Optimized sparse groupby sum aggregation.

Continued on next page

Table 1 – continued from previous page

SparseFrame.head([n])	Return rows from the top of the table.
SparseFrame.index	Return index labels
SparseFrame.join(other[, axis, how, level])	Join two tables along their indices.
SparseFrame.max(*args, **kwargs)	Find maximum element(s).
SparseFrame.mean(*args, **kwargs)	Calculate mean(s).
SparseFrame.min(*args, **kwargs)	Find minimum element(s)
SparseFrame.multiply(other[, axis])	Multiply SparseFrame row-wise or column-wise.
SparseFrame.nnz()	Get the count of explicitly stored values (nonzeros).
SparseFrame.read_npz(filename[, storage_options])	Read from numpy npz format.
SparseFrame.reindex([labels, index, ...])	Conform SparseFrame to new index.
SparseFrame.reindex_axis(labels[, axis, ...])	Conform SparseFrame to new index.
SparseFrame.rename(columns[, inplace])	Rename columns by applying a callable to every column name.
SparseFrame.set_index([column, idx, level, ...])	Set index from array, column or existing multi-index level.
SparseFrame.sort_index()	Sort table along index.
SparseFrame.sum(*args, **kwargs)	Sum elements.
SparseFrame.take(idx[, axis])	Return data at integer locations.
SparseFrame.to_npz(filename[, block_size, ...])	Save to numpy npz format.
SparseFrame.toarray()	Return dense np.array representation.
SparseFrame.todense([pandas])	Return dense representation.
SparseFrame.values	CSR Matrix represenation of frame
SparseFrame.vstack(frames)	Vertical stacking given collection of SparseFrames.

2.4 Dask SparseFrame API

SparseFrame
SparseFrame.assign
SparseFrame.compute
SparseFrame.columns
SparseFrame.get_partition
SparseFrame.index
SparseFrame.join
SparseFrame.known_divisions
SparseFrame.map_partitions
SparseFrame.npartitions
SparseFrame.persist
SparseFrame.repartition
SparseFrame.set_index
SparseFrame.rename
SparseFrame.set_index
SparseFrame.sort_index
SparseFrame.to_delayed
SparseFrame.to_npz

2.5 Reference

2.5.1 sparsity package

Submodules

sparsity.indexing.**get_indexers_list**()

class sparsity.io.**LocalFileSystem**
Bases: object

open()
Open file and return a stream. Raise IOError upon failure.

file is either a text or byte string giving the name (and the path if the file isn't in the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless closefd is set to False.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to 'r' which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), 'x' for creating and writing to a new file, and 'a' for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent: locale.getpreferredencoding(False) is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)
'U'	universal newline mode (deprecated)

The default mode is 'rt' (open for reading text). For binary random access, the mode 'w+b' opens and truncates the file to 0 bytes, while 'r+b' opens the file without truncation. The 'x' mode implies 'w' and raises an *FileExistsError* if the file already exists.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is appended to the mode argument), the contents of the file are returned as strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

'U' mode is deprecated and will raise an exception in future versions of Python. It has no effect in Python 3. Use newline to control universal newlines mode.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on *io.DEFAULT_BUFFER_SIZE*. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which isatty() returns True) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be passed. See the codecs module for the list of supported encodings.

errors is an optional string that specifies how encoding errors are to be handled—this argument should not be used in binary mode. Pass 'strict' to raise a ValueError exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) See the documentation for codecs.register or run 'help(codecs.Codec)' for a list of the permitted encoding error strings.

newline controls how universal newlines works (it only applies to text mode). It can be None, '', 'n', 'r', and 'rn'. It works as follows:

- On input, if newline is None, universal newlines mode is enabled. Lines in the input can end in 'n', 'r', or 'rn', and these are translated into 'n' before being returned to the caller. If it is '', universal newline mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- On output, if newline is None, any 'n' characters written are translated to the system default line separator, os.linesep. If newline is '' or 'n', no translation takes place. If newline is any of the other legal values, any 'n' characters written are translated to the given string.

If closefd is False, the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be True in that case.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). *opener* must return an open file descriptor (passing os.open as *opener* results in functionality similar to passing None).

open() returns a file object whose type depends on the mode, and through which the standard file operations such as reading and writing are performed. When open() is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a TextIOWrapper. When used to open a file in a binary mode, the returned class varies: in read binary mode, it returns a BufferedReader; in write binary and append binary modes, it returns a BufferedWriter, and in read/write mode, it returns a BufferedRandom.

It is also possible to use a string or bytearray as a file for both reading and writing. For strings StringIO can be used like a file opened in a text mode, and for bytes a BytesIO can be used like a file opened in a binary mode.

`sparsity.io.`**`path2str`**(*arg*)

Convert arg into its string representation.

This is only done if arg is subclass of PurePath

`sparsity.io.`**`read_npz`**(*filename*, *storage_options=None*)

Read from a npz file.

Parameters

- **`filename`** (*`str`*) – path to file.
- **`storage_options`** (*`dict`*) – (optional) storage options for external filesystems.

Returns **sf**

Return type sp.SparseFrame

sparsity.io.**to_npz**(*sf*, *filename*, *block_size=None*, *storage_options=None*)
Write to npz file format.

Parameters

- **sf** (*sp.SparseFrame*) – sparse frame to store.
- **filename** (*str*) – path to write to.
- **block_size** (*int*) – block size in bytes when sending data to external filesystem. Default is 100MB.
- **storage_options** (*dict*) – (optional) storage options for external filesystems.

Returns **sf**

Return type *SparseFrame*

class sparsity.sparse_frame.**SparseFrame**(*data*, *index=None*, *columns=None*, ***kwargs*)
Bases: object

Two dimensional, size-mutable, homogenous tabular data structure with labeled axes (rows and columns). It adds pandas indexing abilities to a compressed row sparse frame based on scipy.sparse.csr_matrix. This makes indexing along the first axis extremely efficient and cheap. Indexing along the second axis should be avoided if possible though.

For a distributed implementation see sparsity.dask.SparseFrame.

add(*other*, *how='outer'*, *fill_value=0*, ***kwargs*)
Aligned addition. Adds two tables by aligning them first.

Parameters

- **other** (*sparsity.SparseFrame*) – Another SparseFrame.
- **how** (*str*) – How to join frames along their indexes. Default is 'outer' which makes the result contain labels from both frames.
- **fill_value** (*float*) – Fill value if other frame is not exactly the same shape. For sparse data the only sensible fill value is 0. Passing any other value will result in a ValueError.

Returns **added**

Return type sparsity.SparseFrame

assign(***kwargs*)
Assign new columns.

Parameters **kwargs** (*dict*) – Mapping from column name to values. Values must be of correct shape to be inserted successfully.

Returns **assigned**

Return type *SparseFrame*

axes

columns
Return column labels

Returns **index**

Return type pd.Index

classmethod concat(*tables*, *axis=0*)
Concat a collection of SparseFrames along given axis.

Uses join internally so it might not be very efficient.

Parameters

- **tables** (*list*) – a list of SparseFrames.
- **axis** – which axis to concatenate along.

copy(**args*, *deep=True*, ***kwargs*)
Copy frame

Parameters

- **args** – are passed to indizes and values copy methods
- **deep** (*bool*) – if true (default) data will be copied as well.
- **kwargs** – are passed to indizes and values copy methods

Returns **copy**

Return type *SparseFrame*

data
Return data matrix

Returns **data**

Return type scipy.spar.csr_matrix

drop(*labels*, *axis=1*)
Drop label(s) from given axis.

Currently works only for columns.

Parameters

- **labels** (*array-like*) – labels to drop from the columns
- **axis** (*int*) – only columns are supported atm.

Returns **df**

Return type *SparseFrame*

drop_duplicate_idx(***kwargs*)
Drop rows with duplicated index.

Parameters **kwargs** – kwds are passed to pd.Index.duplicated

Returns **dropeed**

Return type *SparseFrame*

dropna()
Drop nans from index.

fillna(*value*)
Replace NaN values in explicitly stored data with *value*.

Parameters **value** (*scalar*) – Value to use to fill holes. value must be of same dtype as the underlying SparseFrame's data. If 0 is chosen new matrix will have these values eliminated.

Returns **filled**

Return type *SparseFrame*

groupby_agg(*by=None*, *level=None*, *agg_func=None*)
Aggregate data using callable.

The *by* and *level* arguments are mutually exclusive.

Parameters

- **by** (*array-like, string*) – grouping array or grouping column name
- **level** (*int*) – which level from index to use if multiindex
- **agg_func** (*callable*) – Function which will be applied to groups. Must accept a SparseFrame and needs to return a vector of shape (1, n_cols).

Returns **sf** – aggregated result

Return type *SparseFrame*

groupby_sum(*by=None*, *level=0*)
Optimized sparse groupby sum aggregation.

Simple operation using sparse matrix multiplication. Expects result to be sparse as well.

The by and level arguments are mutually exclusive.

Parameters

- **by** (*np.ndarray (optional)*) – Alternative index.
- **level** (*int*) – Level of (multi-)index to group on.

Returns **df** – Grouped by and summed SparseFrame.

Return type sparsity.SparseFrame

head(*n=1*)
Return rows from the top of the table.

Parameters **n** (*int*) – how many rows to return, default is 1

Returns **head**

Return type *SparseFrame*

iloc
partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.

index
Return index labels

Returns **index**

Return type pd.Index

join(*other*, *axis=1*, *how='outer'*, *level=None*)
Join two tables along their indices.

Parameters

- **other** (*sparsity.SparseTable*) – another SparseFrame
- **axis** (*int*) – along which axis to join
- **how** (*str*) – one of 'inner', 'outer', 'left', 'right'
- **level** (*int*) – if axis is MultiIndex, join using this level

Returns **joined**

Return type sparsity.SparseFrame

loc
partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.

max (**args*, ***kwargs*)
Find maximum element(s).

mean (**args*, ***kwargs*)
Calculate mean(s).

min (**args*, ***kwargs*)
Find minimum element(s)

multiply (*other*, *axis='columns'*)
Multiply SparseFrame row-wise or column-wise.

Parameters

- **other** (*array-like*) – Vector of numbers to multiply columns/rows by.
- **axis** (*int | str*) –
  - 1 or 'columns' to multiply column-wise (default)
  - 0 or 'index' to multiply row-wise

nnz ()
Get the count of explicitly stored values (nonzeros).

classmethod read_npz (*filename*, *storage_options=None*)
Read from numpy npz format.

Reads the sparse frame from a npz archive. Supports reading npz archives from remote locations with GCSFS and S3FS.

Parameters

- **filename** (*str*) – path or uri to location
- **storage_options** (*dict*) – further options for the underlying filesystem

Returns sf

Return type *SparseFrame*

reindex (*labels=None*, *index=None*, *columns=None*, *axis=None*, **args*, ***kwargs*)
Conform SparseFrame to new index.

Missing values will be filled with zeroes.

Parameters

- **labels** (*array-like*) – New labels / index to conform the axis specified by 'axis' to.
- **columns** (*index,*) – New labels / index to conform to. Preferably an Index object to avoid duplicating data
- **axis** (*int*) – Axis to target. Can be either (0, 1).
- **kwargs** (*args,*) – Will be passed to reindex_axis.

Returns reindexed

Return type *SparseFrame*

reindex_axis(*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=0*)
Conform SparseFrame to new index.

Missing values will be filled with zeros.

Parameters

- **labels** (*array-like*) – New labels / index to conform the axis specified by 'axis' to.
- **axis** (*int*) – Axis to target. Can be either (0, 1).
- **method** (*None*) – unsupported
- **level** (*None*) – unsupported
- **copy** (*None*) – unsupported
- **limit** (*None*) – unsupported
- **fill_value** (*None*) – unsupported

Returns **reindexed**

Return type *SparseFrame*

rename(*columns*, *inplace=False*)
Rename columns by applying a callable to every column name.

Parameters

- **columns** (*callable*) – a callable that will accepts a column element and returns the new column label.
- **inplace** (*bool*) – if true the operation will be executed inplace

Returns **renamed**

Return type SparseFrame | None

set_index(*column=None*, *idx=None*, *level=None*, *inplace=False*)
Set index from array, column or existing multi-index level.

Parameters

- **column** (*str*) – set index from existing column in data.
- **idx** (*pd.Index, np.array*) – Set the index directly with a pandas index object or array
- **level** (*int*) – set index from a multiindex level. useful for groupbys.
- **inplace** (*bool*) – perform data transformation inplace

Returns **sf** – the transformed sparse frame or None if inplace was True

Return type sp.SparseFrame | None

sort_index()
Sort table along index.

Returns **sorted**

Return type sparsity.SparseFrame

sum(**args*, ***kwargs*)
Sum elements.

take(*idx*, *axis=0*, ***kwargs*)
Return data at integer locations.

Parameters

- **idx** (*array-like | int*) – array of integer locations
- **axis** – which axis to index
- **kwargs** – not used

Returns **indexed** – reindexed sparse frame

Return type *SparseFrame*

to_npz (*filename*, *block_size=None*, *storage_options=None*)
Save to numpy npz format.

Parameters

- **filename** (*str*) – path to local file ot s3 path starting with *s3://*
- **block_size** (*int*) – block size in bytes only has effect if writing to remote storage if set to None defaults to 100MB
- **storage_options** (*dict*) – additional parameters to pass to FileSystem class; only useful when writing to remote storages

toarray()
Return dense np.array representation.

todense (*pandas=True*)
Return dense representation.

Parameters **pandas** (*bool*) – If true returns a pandas DataFrame (default), else a numpy array is returned.

Returns **dense** – dense representation

Return type pd.DataFrame | np.ndarray

values
CSR Matrix represenation of frame

classmethod vstack (*frames*)
Vertical stacking given collection of SparseFrames.

sparsity.sparse_frame.**sparse_one_hot** (*df*, *column=None*, *categories=None*, *dtype='f8'*, *index_col=None*, *order=None*, *prefixes=False*, *ignore_cat_order_mismatch=False*)
One-hot encode specified columns of a pandas.DataFrame. Returns a SparseFrame.

See the documentation of sparsity.dask.reshape.one_hot_encode().

2.5.2 sparsity.dask sub-package

Submodules

CHAPTER 3

Attention

Please enjoy with carefulness as it is a new project and might still contain some bugs.

Python Module Index

S

Index

T

V