SciDB-Py Documentation

Release 15.12.0

SciDB-Py Developers

March 21, 2016

Contents

1 Contents **3**
- 1.1 Whats New . . . 3
- 1.2 Installing SciDB-Py . . . 5
- 1.3 Basic Use . . . 6
- 1.4 Demos and Other Topics . . . 25
- 1.5 API Documentation . . . 25

2 Indices and tables **95**

Python Module Index **97**

SciDB-Py is a Python interface to the SciDB, the massively scalable array-oriented database. SciDB features include ACID transactions, parallel processing, distributed storage, efficient sparse array storage, and native parallel linear algebra operations.

The SciDB-Py package provides an intuitive NumPy-like interface to SciDB, so that users can leverage powerful distributed, data-parallel scientific computing from the comfort of their Python interpreter:

```
from scidbpy import connect
sdb = connect()               # connect to the database
x = sdb.random((1000, 1000)) # 2D array of random numbers
y = (x ** 2 + 3).sum()        # NumPy syntax, computed in the database
```

CHAPTER 1

Contents

1.1 Whats New

1.1.1 15.12 (Released March 21, 2016)

Highlights

- now supports SciDB15.12; use branch *scidb15.7* for use with SciDB15.7

1.1.2 14.10 (Released November 4, 2014)

Highlights

- Support for performing `groupby()` on non-integer attributes
- Added a pandas-like `merge()` method to perform database-style joins
- Experimental support for *compressed transfer*, and for *efficient downloading* of dense arrays
- Added *robust versions* of many AFL operators, that perform array preprocessing (chunk alignment, schema matching, etc.) automatically
- Several new methods: `unique()`, `percentile()`, `isel()`, `hstack()`, `vstack()`, `concatenate()`, `any()`, `all()`, `remove()`, `ls()`, `collapse()`
- Ability to index arrays using *integer arrays*

API Changes in 14.10

- Slicing an array by a boolean array now produces a sparse result, preserving the location of the selected cells. The collapse method converts this array to the 1D dense array previously returned by boolean masking.
- The `merge()` method was changed from a direct AFL call to a high-level join operator. Use robust.merge for the old behavior

1.1.3 14.8 (Released August 22, 2014)

Highlights

- Support for *authenticated and encrypted* connections to ScIDB
- Fixed a bug where uploading large arrays using *from_data* resulted in scrambled cell locations in the database
- Proper treatment of elementwise arithmetic on sparse arrays

1.1.4 14.7 (Released August 1, 2014)

Highlights

- Wider support for *wrapping* all of SciDB's built-in datatypes, including strings, datetimes, and nullable values:

```
>>> x
SciDBArray('py1101328071989_00045<f0:string> [i0=0:2,1000,0]')
>>> x.toarray()
array([u'abc', u'def', u'ghi'], dtype=object)
```

- A *groupby()* method for performing aggregation over groups:

```
x.groupby('gender').aggregate('mean(age)')
```

- *Boolean comparison and filtering of arrays*:

```
>>> x = sdb.random((5,5))
>>> (x > 0.7).toarray()
array([[ True,  True, False, False, False],
       [ True,  True,  True, False, False],
       [False, False, False,  True,  True],
       [False, False,  True, False, False],
       [False,  True,  True, False, False]], dtype=bool)
>>> x[x>0.7].toarray()
array([ 0.83500619,  0.95791602,  0.94745933,  0.89868099,  0.97664716,
        0.7045693 ,  0.88949448,  0.88112397,  0.73766701,  0.94612052])
```

- Pandas-like *syntax* for accessing and defining new array attributes:

```
array['b'] = 'sin(f0+3)'
array['b'].toarray()
```

- *Lazy evaluation* of arrays. Computation for most array operations are deferred until needed.
- AFL queries now return lazy SciDBArrays instead of special class instances, which makes it easy to mix SciDBArray methods with raw AFL calls:

```
>>> sdb.afl.build('<x:float>[i=0:5,10,0]', 'i').max()[0]
```

- A cleaner syntax for *chaining* several AFL calls at once. The following two lines are equivalent:

```
f = sdb.afl
f.subarray(f.project(f.apply(x, 'f2', 'f*2'), 'f2'), 0, 5)

x.apply('f2', 'f*2').project('f2').subarray(0, 5)
```

- New element-wise operators: sqrt, floor, ceil, isnan
- A *cumulate()* method for performing cumulative aggregation over arrays

- Numerous bugfixes.

1.2 Installing SciDB-Py

1.2.1 Software prerequisites

The scidbpy package requires at least:

1. An available SciDB installation
2. The Shim network interface to SciDB

We assume an existing installation of SciDB is available. Binary SciDB packages (for Ubuntu 12.04 and RHEL/CentOS 6) and source code are available from http://scidb.org. The examples in this tutorial assume that SciDB is running on a computer with host name "localhost," at port 8080. If SciDB is not running on localhost, adjust the name accordingly.

The scidbpy package requires installation of a simple HTTP network service called "shim" on the computer that SciDB coordinator is installed on. The network service only needs to be installed on the SciDB computer, not on client computers that connect to SciDB from Python. It's available in packaged binary form for supported SciDB operating systems, and as source code which can be compiled and deployed on any SciDB installation. See http://github.com/paradigm4/shim for source code and installation instructions.

1.2.2 Python Prerequisites

SciDB-Py requires Python 2.6-2.7 or 3.3, as well as NumPy and Requests. Some (optional) functionality requires SciPy and Pandas. Following are a description of these requirements:

NumPy tested with version 1.9.

Requests tested with version 2.7. Required for using the Shim interface to SciDB.

Pandas (optional) tested with version 0.15. Required only for importing/exporting SciDB arrays as Pandas Dataframe objects.

SciPy (optional) tested with versions 0.10-0.12. Required only for importing/exporting SciDB arrays as SciPy sparse matrices.

1.2.3 SciDB-Py Package Installation

The latest release of `scidb-py` can be installed from the Python package index:

```
pip install scidb-py
```

The development version can be found on github at http://github.com/paradigm4/scidb-py. Install the development package directly from Github with:

```
pip install git+http://github.com/paradigm4/scidb-py.git
```

or download the code and type:

```
python setup.py install
```

1.3 Basic Use

The primary data structure in SciDB-Py is the *SciDBArray*. This object defines a NumPy array-like interface for SciDB arrays in Python. While *SciDBArray* syntax mimics NumPy, array operations are converted to SciDB queries which are executed by the database. Data are materialized to Python only when requested. A basic set of array subsetting, arithmetic and utility operations are defined by the package. Additionally, SciDB-Py provides several utilities for compising SciDB queries more explicitly.

1.3.1 Introduction to SciDB arrays

SciDB arrays are composed of *cells*. Each cell may contain one or more values referred to as *attributes*. The data types and number of attributes are consistent across all cells within one array. All the attribute values within a cell may be left undefined, in which case the cell is called empty. Arrays with empty cells are referred to as sparse arrays in the SciDB documentation.

Individual attribute values may also be explicitly marked missing with one of several possible SciDB null codes.

Cells are arranged by an integer coordinate system into n-dimensional arrays. SciDB uses signed 64-bit integers for coordinates. Each coordinate axis is typically referred to as a dimension in the SciDB documentation. SciDB is limited in theory to about 100 dimensions, but in practice that limit is typically much lower (up to say, 10 dimensions or so). While the default SciDB array origin is usually zero, SciDB arrays may use any signed 64-bit integer origin.

SciDBArray objects are Python representations of SciDB arrays that mimic numpy arrays in many ways. *SciDBArray* array objects are limited to the following SciDB array attribute data types: `bool`, `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `uint8`, `unit16`, `uint32`, `uint64`, characters, and strings.

1.3.2 Loading the scidbpy package and connecting to SciDB

In order to use SciDB, Python needs an interface to a SciDB server. This is accomlished through the *connect()* function.

connect takes an optional URL specifying the location of the SciDB coordinator node (running Shim; see *Installing SciDB-Py*). If no URL is provided, it looks for a `SCIDB_URL` environment variable, and then defaults to `http://localhost:8080`.

The following snippet imports SciDB-Py and establishes a connection with the database – adjust the host name as required if SciDB is on a different computer:

```
>>> import numpy as np
>>> from scidbpy import connect
>>> sdb = connect('http://localhost:8080')
```

Throughout this documentation, the `sdb` variable is used to refer to the connection to a SciDB instance, as above.

1.3.3 Authenticated and Encrypted Connections

Shim supports two modes of user authentication (PAM and Digest Authentication) and one form of encryption (SSL over HTTPS – see the Shim API documentation page for more details). The *connect()* function will apply sensible defaults if you provide a username and password:

- Digest Authentication is used if the host URL begins with `http://`
- PAM authentication and SSL are used if the host URL begins with `https://`

Under normal circumnstances you shouldn't need to worry about the details of authentication. However, note that the `sdb` object returned form `connect` includes *`login()`* and *`logout()`* methods for manually managing PAM-authenticated sessions (Digest authentication is applied for every request, and doesn't require a separate login step).

1.3.4 Creating arrays

The following sections illustrate a number of ways to create *`SciDBArray`* objects. The examples assume that an `sdb` interface object has already been set up.

From a numpy array

Perhaps the simplest approach to creating an arbitrary *`SciDBArray`* object is to upload a numpy array into SciDB with the `from_array()` function. Although this approach is very convenient, it is not really suitable for very big arrays (which might exceed memory availability in a single computer, for example). In such cases, consider other options described below.

The following example creates a *`SciDBArray`* object named `Xsdb` from a small 5x4 numpy array named `X`.

```
from scidbpy import connect
sdb = connect()

X = np.random.random((5, 4))
Xsdb = sdb.from_array(X)
```

The package takes care of naming the SciDB array in this example (use `Xsdb.name` to see the SciDB array name).

From a scipy sparse matrix

In a similar way, a *`SciDBArray`* can be created from a scipy sparse matrix. For example:

```
from scipy.sparse import coo_matrix
X = np.random.random((10, 10))
X[X < 0.9] = 0  # make array sparse
Xcoo = coo_matrix(X)
Xsdb = sdb.from_sparse(Xcoo)
```

This operation is most efficient for matrices stored in coordinate form (`coo_matrix`). Other sparse formats will be internally converted to COO form in the process of transferring the data.

Convenience array creation functions

Many standard numpy functions for creating special arrays are supported. These include:

`zeros()` to create an array full of zeros:

```
# Create a 10x10 array of double-precision zeros:
A = sdb.zeros((10, 10))
```

`ones()` to create an array full of ones:

```
# Create a 10x10 array of 64-bit signed integer ones:
A = sdb.ones((10, 10), dtype='int64')
```

`random()` to create an array of uniformly distributed random floating-point values:

```
# Create a 10x10 array of numbers between -1 and 2 (inclusive)
#    sampled from a uniform random distribution.
A = sdb.random((10, 10), lower=-1, upper=2)
```

randint() to create an array of uniformly distributed random integers:

```
# Create a 10x10 array of uniform random integers between 0 and 10
#  (inclusive of 0, non-inclusive of 10)
A = sdb.randint((10, 10), lower=0, upper=10)
```

arange() to create and array with evenly-spaced values given a step size:

```
# Create a vector of ten integers, counting up from zero
A = sdb.arange(10)
```

linspace() to create an array with evenly spaced values between supplied bounds:

```
# Create a vector of 5 equally spaced numbers between 1 and 10,
# including the endpoints:
A = sdb.linspace(1, 10, 5)
```

identity() to create a sparse or dense identity matrix:

```
# Create a 10x10 sparse, double-precision-valued identity matrix:
A = sdb.identity(10, dtype='double', sparse=True)
```

These functions should be familiar to anyone who has used NumPy, and the syntax of each function closely follows its NumPy counterpart. In each case, the array is defined and created directly in the SciDB server, and the resulting Python object is simply a wrapper of the native SciDB array. Because of this, the functions outlined here and in the following sections can be more efficient ways to generate large SciDB arrays than copying data from a numpy array.

Note: SciDB does not yet have a way to set a random seed, prohibiting reproducible results involving the random number generator.

From an existing SciDB array

Finally, *SciDBArray* objects may be created from existing SciDB arrays, so long as the data type restrictions outlined above are met. (It usually makes sense to load large data sets into SciDB externally from the Python package, using the SciDB parallel bulk loader or similar facility.)

The following example uses the `query()` function to build and store a small 10x5 SciDB array named "A" independently of Python. We then create a *SciDBArray* object from the SciDB array with the `wrap_array()` function, passing the name of the array identifier on the SciDB server:

```
# remove A if it already exists
if "A" in sdb.list_arrays():
    sdb.query("remove(A)")

# create an array named 'A' on the server
sdb.query("store(build(<v:double>[i=1:10,10,0,j=1:5,5,0],i+j),A)")

# create a Python object pointing to this array
A = sdb.wrap_array("A")
```

Note that there are some restrictions on the types of arrays which can be wrapped by `SciDB-Py`. The array data must be of a compatible type, and have integer indices. Also, arrays with indices that don't start at zero may not behave as expected for item access and slicing, discussed below.

Note also that many functions in the SciDB-Py package work on single-attribute arrays. When a *SciDBArray* object refers to a SciDB array with more than one attribute, only the first listed attribute is used.

Persistence of SciDB-Py arrays

Each array has a *persistent* attribute. When `persistent` is set to `True`, arrays remain in SciDB until explicitly removed by a `remove` query. If `persistent` is set to False, the arrays are removed when the `SciDBInterface.reap()` or *SciDBArray.reap()* methods are invoked. (Note that *interface.SciDBInterface.reap()* is automatically invoked when Python exits).

Arrays defined from an existing SciDB array using the *wrap_array()* argument are always persistent, while all other array creation routines set `persistent=False` by default:

```
X = sdb.random(10, persistent=False)  # default
print(X.name in sdb.list_arrays())  # True
X.reap()
print(X.name in sdb.list_arrays())  # False
```

When *connect()* is used as a context manager, non-persistent arrays are reaped at the end of the context block:

```
with connect(url) as sdb:
    X = sdb.random(10)
# deleted here
```

1.3.5 Accessing array data

Converting arrays to other data structures

SciDB-Py is designed to perform operations on SciDB arrays in a natural Python dialect, computing those operations in SciDB while minimizing data traffic between the database and Python. However, it is useful to materialize SciDB array data to Python, for example to obtain and plot results.

SciDBArray objects provide several functions that materialize array data to Python:

toarray() can be used to populate a `numpy` array from an *N*-dimensional array with any number of attributes:

```
>>> A = sdb.linspace(0, 10, 5)
>>> A.toarray()
array([  0. ,   2.5,   5. ,   7.5,  10. ])

>>> B = sdb.join(sdb.linspace(0, 8, 5), sdb.arange(5, dtype=int))
>>> B.toarray()
array([(0.0, 0), (2.0, 1), (4.0, 2), (6.0, 3), (8.0, 4)],
          dtype=[('f0', '<f8'), ('f0_2', '<i8')])
```

tosparse() can be used to populate a SciPy sparse matrix from a 2-dimensional array with a single attribute:

```
>>> I = sdb.identity(5, sparse=True)
```

```
>>> I.tosparse(sparse_fmt='dia')
    <5x5 sparse matrix of type '<type 'numpy.float64'>'
        with 5 stored elements (1 diagonals) in DIAgonal format>
```

tosparse() will also work with 1-dimensional arrays or multi-dimensional arrays; in this case the result cannot be exported to a SciPy sparse format, but will be returned as a Numpy record array listing the indices and values.

todataframe() can be used to populate a Pandas dataframe from a 1-dimensional array with any number of attributes:

```
>>> B = sdb.join(sdb.linspace(0, 8, 5, dtype='<A:double>'),
                 sdb.arange(1, 6, dtype='<B:int32>'),
                 sdb.ones(5, dtype='<C:float>'))
>>> B.todataframe()
   A  B  C
0  0  1  1
1  2  2  1
2  4  3  1
3  6  4  1
4  8  5  1
```

These methods are discussed in greater detail in *Downloading SciDBArrays*.

Element Access

Single elements of *SciDBArray* objects can be referenced with the standard numpy indexing syntax. These single elements are returned by value:

```
>>> x = sdb.arange(12).reshape((3,4))
>>> x[1, 2]
6
```

Note that element assignment (e.g. `x[0, 0] = 4`)is not supported.

Subarrays and Slice Syntax

SciDBArrays support NumPy's slice syntax for extracting subregions:

```
>>> x = sdb.arange(30).reshape((6, 5))
>>> x.toarray()
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]])
>>> x[0:2].toarray() # the first 2 rows
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> x[:, 1:3].toarray()  # the second 2 columns
array([[ 1,  2],
       [ 6,  7],
       [11, 12],
       [16, 17],
       [21, 22],
       [26, 27]])
>>> x[::2].toarray()  # every other row
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24]])
```

Some of NumPy's "Fancy Indexing" operations, like indexing with a boolean array, are also supported; see *Comparing and Filtering Arrays*. You can also index arrays using integer arrays:

```
>>> x = sdb.arange(100) * 5
>>> y = sdb.from_array(np.array([ 3,  3,  5, 10, 30, 20,  5]))
>>> x[y].toarray()
array([ 15,  15,  25,  50, 150, 100,  25])
```

Slicing by dimension name

The *isel()* method allows you to index into arrays by dimension name instead of position:

```
>>> x = sdb.arange(30).reshape((6, 5))
>>> x.schema
'<f0:int64> [i0=0:5,1000,0,i1=0:4,1000,0]'
>>> x.isel(i1=2).toarray()   # same as x[:, 2]
array([ 2,  7, 12, 17, 22, 27])
```

Attribute access

You can access specific attributes of an array by passing their names in the brackets. You can also add new attributes by providing a SciDB expression:

```
>>> x = sdb.arange(4)
>>> x.att_names
['f0']
# extract the f0 attribute
>>> x['f0'].toarray()
array([0, 1, 2, 3])

# add a new attribute, and access it
>>> x['y'] = 'sin(f0 * 3)'
>>> x['y'].toarray()
array([ 0.        ,  0.14112001, -0.2794155 ,  0.41211849])

# multi-attribute access
>>> x[['y', 'f0']].toarray()
array([(0.0, 0), (0.1411200080598672, 1), (-0.27941549819892586, 2),
       (0.4121184852417566, 3)],
      dtype=[('y', '<f8'), ('f0', '<i8')])
```

1.3.6 Basic Math on SciDB array objects

Operations on *SciDBArray* objects generally return new *SciDBArray* objects. The general idea is to promote function composition involving *SciDBArray* objects without moving data between SciDB and Python.

The `scidbpy` package provides quite a few common operations including subsetting, pointwise application of scalar functions, aggregations, and pointwise and matrix arithmetic.

Standard numpy attributes like `shape`, `ndim` and `size` are defined for *SciDBArray* objects:

```
>>> X = sdb.random((5, 10))
>>> X.shape
(5, 10)
>>> X.size
50
>>> X.ndim
2
```

Many SciDB-specific attributes are also defined, including `chunk_size`, `chunk_overlap`, and `sdbtype`,

```
>>> X.chunk_size
[1000, 1000]
>>> X.chunk_overlap
[0, 0]
>>> X.sdbtype
sdbtype('<f0:double>')
```

SciDBArrays also contain a `datashape` object, which encapsulates much of the interface between Python and SciDB data, including the full array schema:

```
>>> Xds = X.datashape
>>> Xds.schema
'<f0:double> [i0=0:4,1000,0,i1=0:9,1000,0]'
```

Scalar functions of SciDBArray objects (aggregations)

The package exposes the following aggregations:

Name	Description
`min()`	minimum value
`max()`	maximum value
`sum()`	sum of values
`var()`	variance of values
`stdev()`	standard deviation of values
`std()`	standard deviation of values
`avg()`	average/mean of values
`mean()`	average/mean of values
`count()`	count of nonempty cells
`approxdc()`	fast estimate of the number of distinct values

Examples: Minimum Aggregates

Each operation can be computed across the entire array, or across specified dimensions by passing the index or indices of the desired dimensions. For example:

```
>>> np.random.seed(0)
>>> X = sdb.from_array(np.random.random((5, 3)))
>>> X.toarray()
array([[ 0.5488135 ,  0.71518937,  0.60276338],
       [ 0.54488318,  0.4236548 ,  0.64589411],
       [ 0.43758721,  0.891773  ,  0.96366276],
       [ 0.38344152,  0.79172504,  0.52889492],
       [ 0.56804456,  0.92559664,  0.07103606]])
```

Here we'll find the minimum of all values in the array. The returned result is a new SciDBArray, so we select the first element:

```
>>> X.min()[0]
0.071036058197886942
```

Like numpy, passing index 0 gives us the minimum within every column:

```
>>> X.min(0).toarray()
array([ 0.38344152,  0.4236548 ,  0.07103606])
```

Passing index 1 gives us the minimum within every row:

```
>>> X.min(1).toarray()
array([ 0.5488135 ,  0.4236548 ,  0.43758721,  0.38344152,  0.07103606])
```

Note that the convention for specifying aggregate indices here is designed to match numpy, and is *opposite the convention used within SciDB*. To recover SciDB-style aggregates, you can use the `scidb_syntax` flag:

```
>>> X.min(1, scidb_syntax=True).toarray()
array([ 0.38344152,  0.4236548 ,  0.07103606])
```

Further Examples

These operations return new `SciDBArray` objects consisting of scalar values. Here are a few examples that materialize their results to Python:

```
>>> tridiag.count()[0]
28
>>> tridiag.sum()[0]
20.0
>>> tridiag.var()[0]
1.6190476190476193
```

Note that a count of nonempty cells is also directly available from the `nonempty()` function:

```
>>> tridiag.nonempty()
28
```

A related function is `nonnull()`, which counts the number of nonempty cells which do not contain a null value. In this case, the result is the same as `nonempty()`:

```
>>> tridiag.nonnull()
28
```

Pointwise application of scalar functions

The package exposes SciDB scalar-valued scalar functions that can be applied element-wise to SciDB arrays:

Function	Description
`sin()`	Trigonometric sine
`asin()`	Trigonometric arc-sine / inverse sine
`cos()`	Trigonometric cosine
`acos()`	Trigonometric arc-cosine / inverse cosine
`tan()`	Trigonometric tangent
`atan()`	Trigonometric arc-tangent / inverse tagent
`exp()`	Natural exponent
`log()`	Natural logarithm
`log10()`	Base-10 logarithm
`sqrt()`	Square root
`ceil()`	Ceiling function
`floor()`	Floor function
`is_nan()`	Test for NaN values

All trigonometric functions assume arguments are given in radians. Here is a simple example that compares a computation in SciDB with a local one (using the ‘tridiag‘ array defined in the last examples):

```
>>> sin_tri = sdb.sin(tridiag)
>>> np.linalg.norm(sin_tri.toarray() - np.sin(tridiag.toarray()))
0.0
```

Shape and layout functions

Arrays may be transposed and their data re-arranged into new shapes with the usual `transpose()` and `reshape()` functions:

```
>>> tri_reshape = tridiag.reshape((20,5))
>>> tri_reshape.shape
(20, 5)
>>> tri_reshape.transpose().shape
(5, 20)
>>> tri_reshape.T.shape  # shortcut for transpose
(5, 20)
```

Arithmetic

The package defines elementwise operations on all arrays and linear algebra operations on matrices and vectors. Scalar multiplication is supported.

Element-wise sums and products:

```
>>> np.random.seed(1)
>>> X = sdb.from_array(np.random.random((10, 10)))
>>> Y = sdb.from_array(np.random.random((10, 10)))
>>> S = X + Y
>>> D = X - Y
>>> M = 2 * X
>>> (S + D - M).sum()[0]
-1.1102230246251565e-16
```

We can combine operations as well:

```
>>> Z = 0.5 * (X + X.T)
```

There are also linear algebra operations (matrix-matrix product, matrix-vector product) using the `dot()` function:

```
>>> XY = sdb.dot(X, Y)
>>> XY1 = sdb.dot(X, Y[:,1])
>>> XTX = sdb.dot(X.T, X)
```

Broadcasting

Numpy broadcasting conventions are generally followed in operations involving differently-sized `SciDBArray` objects. Consider the following example that centers a matrix by subtracting its column average from each column.

First we create a test array with 5 columns:

```
>>> np.random.seed(0)
>>> X = sdb.from_array(np.random.random((10, 5)))
```

Now create a vector of column means:

```
>>> xcolmean = X.mean(0)
>>> xcolmean.shape
(5,)
```

Subtract these means from the columns – this is a broadcasting operation:

```
>>> XC = X - xcolmean
```

To check that the columns are now centered, we compute the column mean of `XC`:

```
>>> XC.mean(1).toarray()
array([ -2.22044605e-17,   4.44089210e-17,  -1.11022302e-17,
         1.11022302e-16,  -3.33066907e-17])
```

The broadcasting operation which creates `XC` is implemented using a join operation along dimension 1.

Lazy Evaluation

When possible, SciDB-Py defers actual database computation until data are needed. It does this by using **lazy arrays**, which are references to as-yet unevaluated SciDB queries. Many array methods actually return lazy arrays:

```
>>> x = sdb.random((3,4))
>>> x.name  # an array in the database
'py1102522658694_00001'
>>> y = x.mean(0)
>>> y.name  # not yet in the database
'aggregate(py1102522658694_00001,avg(f0),i1)'
```

Note that y's name doesn't refer to an array in the database, but rather a query on x. Lazy arrays can also be identified by their non-null *query* attribute:

```
>>> y.query
'aggregate(py1102522658694_00001,avg(f0),i1)'
>>> x.query is None
True
```

Calling `eval()` forces lazy-arrays to be evaluated (it has no effect on non-lazy arrays):

```
>>> y.eval()
>>> y.name
'py1102522658694_00014'
```

In most cases you don't need to worry about whether an array is lazy or not – lazy arrays have all the same methods as regular arrays, and normally the difference is transparent to the user. However, lazy arrays can be more efficient with regard to compound queries. Consider an equation like the law of cosines:

```
c2 = a ** 2 + b ** 2 - 2 * a * b * sdb.cos(C)
```

This equation involves creating 7 intermediate data products:

- `t1 = a ** 2`
- `t2 = b ** 2`
- `t3 = 2 * a`
- `t4 = t3 * b`
- `t5 = sdb.cos(C)`
- `t6 = t4 * t5`
- `t7 = t1 + t2`
- `c2 = t7 - t6`

If `a`, `b`, and `C` are large SciDBArrays, this involves many round-trip communiciations to the databse, several passes over the data, and the storage of 7 arrays. Lazy arrays reduce this overhead by representing some of these temporary

arrays as unevaluated sub-queries. Passing larger queries to SciDB at once also gives the database more opportunity to optimize the final query, performing the computation in fewer passes over the data.

In some situations it's necessary or more efficient to force evaluation of lazy arrays (often places where an array appears several times in a complex query). Some SciDB-Py methods perform this evaluation internally. You should also consider calling `eval()` on lazy arrays if you think the unevaluated queries are becoming too cumbersome.

1.3.7 Aggregation and Join Operations

SciDB-Py provides several high-level functions to perform database-style joins and aggregations. The syntax of these functions are modeled after Pandas.

Groupby

The *groupby()* operation allows you to partition an array into groups based on the value in one or more columns, and then perform operations on each group separately:

```
In [2]: import pandas as pd
In [3]: df = pd.DataFrame({'x': [1, 1, 1, 2, 2, 1, 3], 'y':[1, 2, 3, 4, 5, 6, 7]})
In [4]: x = sdb.from_dataframe(df)
In [5]: x.groupby('x').aggregate('sum(y)').todataframe()
Out[5]:
     x_cat  y_sum  x
idx
0        0     12  1
1        1      9  2
2        2      7  3
```

groupby() takes one or more attribute or dimension names as inputs, and returns an intermediate *GroupBy* object. Calling *aggregate()* on this object aggregates over groups.

The argument to groupby can be either:

- A string, interpreted as a SciDB aggregate command
- A dict, mapping output attribute names to SciDB Aggregation commands

For example:

```
In [17]: x.groupby('x').aggregate({'y_sum':'sum(y)', 'y_max':'max(y)'}).todataframe()
Out[17]:
     x_cat  y_sum  y_max  x
idx
0        0     12      6  1
1        1      9      5  2
2        2      7      7  3
```

Grouping on attributes

When attribute names are used to group arrays, they are first lexicographically sorted and converted into categorical dimensions. This kind of grouping is more expensive than a grouping on dimension names.

Aggregate

The *aggregate()* method behaves similarly to GroupBy, but provides a syntax more akin to R. It takes a *by* argument to support grouping on dimension values. However, unlike GroupBy, the names passed to *by must* be dimensions

Database-style joins

The *merge()* method mimics the Pandas merge function, to perform database-style joins on two arrays. When joining two arrays, they are first aligned along common values of one or more *join dimensions*. Then, the attributes of each array are concatenated.

Like GroupBy, *merge()* automatically computes categorical dimensions to support joining on attribute names:

```
In [34]: x = sdb.arange(5)
In [35]: x['f1'] = 'f0 * 10'
In [36]: y = sdb.arange(6) * 3
In [37]: x.todataframe()
Out[37]:
    f0  f1
i0
0    0   0
1    1  10
2    2  20
3    3  30
4    4  40

In [44]: y.todataframe()
Out[44]:
     x
i0
0    0
1    3
2    6
3    9
4   12
5   15

In [45]: sdb.merge(x, y).todataframe()
Out[45]:
    f0  f1   x
i0
0    0   0   0
1    1  10   3
2    2  20   6
3    3  30   9
4    4  40  12

In [46]: sdb.merge(x, y, left_on='f0', right_on='x').todataframe()
Out[46]:
                  f0  f1  x
i0_x f0_cat i0_y
0    0      0      0   0  0
3    3      1      3  30  3
```

Note: Merges are currently restricted to inner joins

Note: Prior to SciDB-Py v14.10, the *merge* function performed a direct AFL merge() call. It now performs the higher-level function described above.

1.3.8 Comparing and Filtering Arrays

SciDB-Py provides support for comparing and filtering SciDB arrays, using NumPy-like syntax.

The standard inequality operators perform element-wise inequality testing between SciDB arrays, NumPy arrays, and scalars:

```
In [1]: from scidbpy import connect

In [2]: sdb = connect()

In [3]: x = sdb.arange(5)

In [4]: x.toarray()
Out[4]: array([0, 1, 2, 3, 4])

In [5]: (x < 2).toarray()
Out[5]: array([ True,  True, False, False, False], dtype=bool)

In [6]: y = np.array([5, 0, 3, 2, 1])

In [7]: (x < y).toarray()
Out[7]: array([ True, False,  True, False, False], dtype=bool)

In [13]: z = sdb.from_array(y)

In [14]: (x < z).toarray()
Out[14]: array([ True, False,  True, False, False], dtype=bool)
```

Array broadcasting is not currently performed when comparing two arrays – they must have identical shapes.

As with NumPy arrays, boolean SciDB arrays can be used as masks:

```
In [4]: r = sdb.random((3,4))

In [5]: r.toarray()
Out[5]:
array([[ 0.72039148,  0.6497302 ,  0.84122248,  0.87304017],
       [ 0.14896572,  0.71237498,  0.21999935,  0.14793879],
       [ 0.69345283,  0.18611741,  0.43660223,  0.06478555]])

In [6]: r[r > 0.5].toarray()
Out[6]:
array([[ 0.72039148,  0.6497302 ,  0.84122248,  0.87304017],
       [ 0.        ,  0.71237498,  0.        ,  0.        ],
       [ 0.69345283,  0.        ,  0.        ,  0.        ]])
```

Note that this masking behavior is different than NumPy – NumPy collapses the input array when masking, returning a 1D result of unmasked items. To reproduce this behavior in SciDB-Py, use the *collapse()* method:

```
In [9]: r[r > 0.5].collapse().toarray()
Out[9]:
array([ 0.72039148,  0.6497302 ,  0.84122248,  0.87304017,  0.71237498,
        0.69345283])
```

SciDB-Py behaves this way in order to retain the location of unmasked items, which is often useful information. For example, we can see these locations when using *todataframe()*:

```
In [10]: r[r > 0.5].todataframe()
Out[10]:
              f0
i0 i1
0  0   0.720391
   1   0.649730
   2   0.841222
   3   0.873040
1  1   0.712375
2  0   0.693453
```

Note: SciDB-Py's masking behavior was changed in version 12.10. Prior to this, SciDB-Py collapsed results like NumPy

Extracting values along a particular axis

Use the *SciDBArray.compress()* method to extract row or column subsets of an array. For example, to extract all rows where the sum across all columns exceeds a threshold:

```
In [3]: x = sdb.random((3,4))

In [4]: x.toarray()
Out[4]:
array([[ 0.10111977,  0.5511177 ,  0.49532397,  0.4213646 ],
       [ 0.3812068 ,  0.97679566,  0.20473656,  0.40256096],
       [ 0.2387294 ,  0.88714084,  0.01064819,  0.48275173]])

In [5]: x.mean(1).toarray()
Out[5]: array([ 0.39223151,  0.491325  ,  0.40481754])

In [6]: x.compress(x.mean(1) > 0.4, axis=0).toarray()
Out[6]:
array([[ 0.3812068 ,  0.97679566,  0.20473656,  0.40256096],
       [ 0.2387294 ,  0.88714084,  0.01064819,  0.48275173]])
```

Aggregation based on masked values

A future version of SciDB-Py will provide a groupby operator, allowing comparisons to be used to compute group-wise aggregates:

```
sdb.groupby(x, x < 0.5).sum()
```

Until that method is added, you can perform the same computation with two aggregate calls:

```
mask = x < 0.5
x[mask].sum()
x[~mask].sum()
```

Comparison with SciDB-R

See this page for SciDB-R's syntax for comparing and filtering arrays.

1.3.9 The SciDB Query Interface

`scidbpy` provides python wrappers for many useful SciDB operations, but the SciDB AFL query language can provide even more customization of operations (For more information on SciDB's AFL and AQL languages, see the SciDB Manual). The `query()` function provides a useful interface for generating raw queries by exploiting Python's String Formatting syntax. Through automatic insertion of the server-side identifiers of SciDB arrays, attributes, and dimensions, the query interface makes constructing complicated queries very convenient.

The general approach first creates a new *SciDBArray* object and then issues a query to populate data. For example, to build an array of zeros similar to the result of the `zeros()` function shown above, the query can be constructed in the following way:

```
>>> # first define an empty array to hold the result
>>> zeros = sdb.new_array(shape=(5, 5), dtype='double')
>>> # now execute a query to fill the array
>>> sdb.query('store(build({A}, 0), {A})', A=zeros)
>>> zeros.toarray()
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

The result is that `zeros` is a 10x10 array filled with zeros. Here the format statement `{A}` is replaced by the name of the desired array on the SciDB server.

We can use this interface to quickly build more complex arrays. For example, to create an identity matrix similar to the result of the `identity()` function shown above, we add a boolean check:

```
>>> ident = sdb.new_array((5, 5), dtype='double')
>>> sdb.query('store(build({A}, iif({A.d0}={A.d1}, 1, 0)), {A})',
...           A=ident)
>>> ident.toarray()
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

Here the substitutions `{A.d0}` and `{A.d1}` are replaced by the first and second dimension names of the array referenced by `A`.

Things can become even more complicated. The following example creates a 5x5 tridiagonal array, similar to the one used in the above examples:

```
>>> tridiag = sdb.new_array((5, 5))
>>> sdb.query('store(build({A}, \
...               iif({A.d0}={A.d1}, 2, iif({A.d0} <= {A.d1}+1 and {A.d0} >= {A.d1}-1, -1, 0))), {A}
>>> tridiag.toarray()
array([[ 2., -1.,  0.,  0.,  0.],
       [-1.,  2., -1.,  0.,  0.],
       [ 0., -1.,  2., -1.,  0.],
       [ 0.,  0., -1.,  2., -1.],
       [ 0.,  0.,  0., -1.,  2.]])
```

The query builds a tridiagonal array with 2 on the diagonal and -1 on the sub- and super-diagonals. This shows how the query-formatting syntax provided by the scidbpy package can be used to generate extremely powerful AFL queries.

The full replacement syntax is outlined in the documentation of the `query()` function. It is a useful way to help streamline the process of writing SciDB queries if and when it becomes necessary.

Working with the Array Functional Language

In addtion to the *SciDBArray* class and `query` interface, SciDB-Py provides a *direct binding* to SciDB's Array Functional Language, or AFL.

The AFL consists of approximately 100 functions to perform array analysis. You can access these operators through the `afl` attribute of a SciDB instance. More information on these operators can be found on the SciDB documentation.

AFL operators accept and return *SciDBArray* objects, and thus can be used interchangeably with methods on the *SciDBArray* class itself.

Usage Example

```
>>> from scidbpy import connect
>>> sdb = connect()
>>> x = sdb.random((3, 4))
>>> afl = sdb.afl
>>> # the number of elements greater than 0.5
>>> cts = afl.aggregate(afl.filter(x, 'f0 > 0.5'), 'count(*)')
>>> cts.query
'aggregate(filter(py1100918363281_00001,f0 > 0.5),count(*))'
>>> cts.toarray()
array([8], dtype=uint64)
```

Note that each AFL operator includes documentation from the official SciDB manual:

```
In [37]: afl.filter?
Type:        function
String form: <function filter at 0x104eb1c08>
File:        /Users/beaumont/scidbpy/scidbpy/afl.py
Definition:  afl.filter(*args)
Docstring:
filter( srcArray, expression )

Produces a result array by filtering out (mark as empty) the cells in the source array     for which

Parameters
----------

    - srcArray: a source array with srcAttrs and srcDims.
    - expression: an expression which takes a cell in the source array
      as input and evaluates to either True or False.
```

Note: Some AFL functions, like `list()`, expect single-quoted strings as input. These quotes must be explicitly provided (e.g. `afl.list("'arrays'")`).

Chaining AFL Operators

Many SciDB queries involve nesting several AFL calls, with the result of an inner call included as the first argument of an outer call. For example:

```
afl.filter(afl.project(afl.apply(x, 'y', 'f0+1'), 'y'), 'y > 3')
```

SciDB-py includes some syntactic sugar for building queries like this: for AFL operators whose names don't collide with another *SciDBArray* method, `x.operator(...)` is equivalent to `afl.operator(x, ...)`. Thus, the above query can be re-written as:

```
x.apply('y', 'f0+1').project('y').filter('y > 3')
```

These two syntaxes are equivalent.

Warning: This syntax only applies to AFL operators that don't collide with a *SciDBArray* method name.

1.3.10 Downloading SciDBArrays

It is often advantageous to convert small arrays into "normal" python data structures, for further analysis in python. SciDB-Py provides several conversion routines:

- *SciDBArray.toarray()* converts the array to a NumPy array
- *SciDBArray.todataframe()* converts the array to a Pandas DataFrame
- *SciDBArray.tosparse()* converts the array to a SciPy sparse array

SciDB supports a wide variety of data types and array schemas, including several concepts that don't have obvious analogs in NumPy. These include:

- Nonzero array origins
- Unbounded array dimensions
- Null values for all datatypes (including integers)

The exact shape and datatype of the result of *SciDBArray.toarray()* depends on these details. Below we outline the various possibilities, starting with the easiest cases.

Single attribute, non-sparse, non-nullable array

An array with a single non-nullable attribute is converted into a NumPy array of equivalent datatype:

SciDB datatype	NumPy datatype (typecode)
bool	bool ('<b1')
int8	int8 ('<b')
uint8	uint8 ('<B')
int16	int16 ('<h')
uint16	uint16 ('<H')
int32	int32 ('<i')
uint32	uint32 ('<I')
int64	int64 ('<l')
uint64	uint64 ('<L')
float	float32 ('<float32')
double	double ('<d')
char	S1 ('c')
datetime	datetime64 ('<M8[s]')
datetimetz	datetime64 ('<M8[s]')
string	object

Note that strings are converted into python object arrays, since NumPy string arrrays are otherwise required to have the same string length in each element

Single attribute, nullable array

NumPy does not support the notion of missing values for datatypes like integers and boolean. Thus, when downloading an array with a nullable attribute, these datatypes are "promoted" to a datatype with a dedicated missing value holder:

SciDB datatype	Null-promoted datatype
bool	double
int8	double
uint8	double
int16	double
uint16	double
int32	double
uint32	double
int64	double
uint64	double
float	float32
double	double
char	S1 ('c')
datetime	datetime64 ('<M8[s]')
datetimetz	datetime64 ('<M8[s]')
string	object

In the NumPy array, each masked element is assigned the default null value for its datatype:

NumPy Datatype	Default masked value
float, double	NaN
char	'\0'
object	None
datetime	NaT

Another way to deal with missing values is to substitute a manually-defined missing value. This converts the array to a non-nullable array:

```
>>> from scidbpy import connect
>>> sdb = connect()

>>> x = sdb.afl.build('<a:int8 NULL>[i=0:5,10,0]', 'iif(i>0, i, null)')
>>> x.toarray()
array([ nan,   1.,   2.,   3.,   4.,   5.])

>>> x.substitute(-1).toarray()
array([-1,  1,  2,  3,  4,  5], dtype=int8)
```

SciDB allows several different "missing-data" codes to be assigned to a masked cell. At the moment SciDB-Py doesn't distinguish between these: either a cell has data, or it is considered masked.

Arrays with empty cells

In addition to masked values, SciDBArrays can have empty cells. These cells are treated as zero-valued when converting to a NumPy array. The zero-value for non-numeric datatypes is determined from the NumPy.zeros function:

```
>>> x = sdb.afl.build('<a:int8>[i=0:3,10,0]', 10)
>>> x = x.redimension('<a:int8>[i=0:5,10,0]')
>>> x.toarray()
array([10, 10, 10, 10,  0,  0], dtype=int8)
```

```
>>> x = sdb.afl.build('<a:char>[i=0:3,10,0]', "'a'")
>>> x = x.redimension('<a:char>[i=0:5,10,0]')
>>> x.toarray()
array(['a', 'a', 'a', 'a', '', ''], dtype='|S1')
```

Nonzero Origins

SciDBArray.toarray() shifts any non-zero origin to the 0-position in the NumPy array:

```
>>> x = sdb.afl.build('<a:int8>[i=5:10,10,0]', 'i')
>>> x.toarray()
array([ 5,  6,  7,  8,  9, 10], dtype=int8)
```

The original array indices can be extracted using the unpack operator:

```
>>> x.unpack('_idx').toarray()
array([(5, 5), (6, 6), (7, 7), (8, 8), (9, 9), (10, 10)],
      dtype=[('i', '<i8'), ('a', 'i1')])
```

Unbound Arrays

When a SciDBArray is unbound, the resulting NumPy array is truncated to the region containing data.

Multiattribute arrays

Arrays with multiple attributes are handled analogously to single-attribute arrays as discussed above. However, the output is returned as a NumPy record array, with record labels matching the SciDB attribute labels.

Efficient Transfer for Dense Arrays

SciDB arrays are internally sparse. To preserve the location of nonempty cells when downloading, SciDB-Py has to explicitly transfer the multidimensional index of each nonempty cell, along with the values of that cell. This adds processing and bandwidth overhead.

SciDB-Py v14.10 introduced a new **dense transfer** option for more efficient downloading of arrays with no empty cells. If you specify `method=dense` to methods like *toarray()*, SciDB-Py will avoid transferring indices.

It is up to the user to set `method=dense`, as well as to verify that the array is fully dense (has no empty cells).

Note: "method=dense" is an experimental feature of SciDB-Py v14.10. Please report any bugs.

Compressed Transfer

Starting in v14.10, SciDB-Py has experimental support for gzipped-compressed transfer. This requires a version of Shim more recent than Nov 3, 2014.

To explicitly enable compression for a particular transfer, specify `compression=1-9` in a method like *toarray()*. `compression=1` corresponds to fast compression, "compression=9" corresponds to best compression.

To implicitly enable compression for all transfers, set the *default_compression* attribute of the `SciDBInterface` to 1-9:

```
sdb.default_compression = 1
sdb.zeros(10).toarray()  # implicitly uses toarray(compression=1)
```

1.4 Demos and Other Topics

- correlation
- Airline groupby demo
- Distance-weighted population demo

1.5 API Documentation

1.5.1 API Reference

This is the list of classes and functions available in SciDB-py.

SciDB Array Class

class scidbpy.**SciDBArray**(*datashape*, *interface*, *name*, *persistent=False*)
SciDBArray class

It is not recommended to instantiate this class directly; use a convenience routine from SciDBInterface.

Attributes

T	Permute the dimensions of an array.
afl	An alias to the AFL namespace
att_names	
chunk_overlap	
chunk_size	
datashape	
dim_names	
dtype	
natt	
ndim	
persistent	Controls whether the array is deleted when
query	
schema	Return the array schema
sdbslice	
sdbtype	
shape	
size	

Methods

aggregate(*args, **kwargs)	Perform one or more aggregations over an array.
alias([name])	Return an alias of the array, optionally with a new name
all()	Returns whether all elements of each attribute are true.
any()	Returns whether any elements of each attribute are true.
approxdc([index, scidb_syntax])	Return the number of distinct values of the array or along an axis.
as_temp([name])	Create a SciDB TEMP array, stored in RAM
att(a)	Return the attribute name of the array.
attribute(a)	Return the attribute name of the array.
attribute_rename(*args)	Rename a set of attributes
avg([index, scidb_syntax])	Return the average of the array or the average along an axis.
collapse()	Flatten and remove all the empty cells.
compress(mask[, axis])	Extract a subset of entries along a given axis,
contains_nulls([attr])	Return True if the array contains null values.
contents(**kwargs)	Return a string representation of the array contents
copy([new_name, persistent])	Make a copy of the array in the database
count([index, scidb_syntax])	Return the count of the array or the count along an axis.
cumprod([axis])	Return the cumulative product over the array.
cumsum([axis])	Return the cumulative sum over the array.
cumulate(expression[, dimension])	Compute running operations along data (e.g., cumulative sums)
dimension(d)	Return the dimension name of the array
dimension_rename(*args)	Rename a set of dimensions
eval([out, store])	If the array is backed by an unevaluated query,
from_query(interface, query)	Build a lazily-evaulated SciDB array from a query string
groupby(by)	Build a groupby object from this array
head([n])	Extract and download the first few elements in the array
index_lookup(idx_array, attribute[, ...])	Wrapper around the index_lookup AFL call.
isel(**kwargs)	Select a subset of the array by dimension name
issparse()	Check whether array is sparse.
max([index, scidb_syntax])	Return the maximum of the array or the maximum along an axis.
mean([index, scidb_syntax])	Return the average of the array or the average along an axis.
min([index, scidb_syntax])	Return the minimum of the array or the minimum along an axis.
nonempty()	Return the number of nonempty elements in the array.
nonnull([attr])	Return the number of non-empty and non-null values.
reap([ignore])	Delete this object from the database if it isn't persistent.
regrid(size[, aggregate])	Regrid the array using the specified aggregate
relabel(renames)	relabel the attributes or dimensions in an array.
rename(new_name[, persistent])	Rename the array in the database, optionally making the new array persistent.
reshape(shape, **kwargs)	Reshape data into a new array
std([index, scidb_syntax])	Return the standard deviation of the array or along an axis.
stdev([index, scidb_syntax])	Return the standard deviation of the array or along an axis.
substitute(value)	Reshape data into a new array, substituting a default for any nulls.
sum([index, scidb_syntax])	Return the sum of the array or the sum along an axis.
toarray(**kwargs)	Transfer data from database and store in a numpy array.
todataframe(**kwargs)	Transfer array from database and store in a local Pandas dataframe
tolist(**kwargs)	Download the array as a (nested) python list
tosparse([sparse_fmt])	Transfer array from database and store in a local sparse array.
transpose(*axes)	Permute the dimensions of an array.
unpack([name])	Unpack with automatic dimension name disambiguation
var([index, scidb_syntax])	Return the variance of the array or the variance along an axis.

T

Permute the dimensions of an array.

Parameters **axes** : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

Returns **out** : ndarray

Copy of *a*, with axes suitably permuted.

afl

An alias to the AFL namespace

aggregate(**args*, ***kwargs*)

Perform one or more aggregations over an array.

Parameters ***args: One or more SciDB aggregate expressions**

Aggregations to perform, like 'sum(value) as x'

by :

See also:

groupby

Examples

x = sdb.arange(10).reshape((5, 2)) x.aggregate('count(*)').toarray() x.aggregate('max(f0)', by='i0').toarray()

alias(*name=None*)

Return an alias of the array, optionally with a new name

all()

Returns whether all elements of each attribute are true.

Returns **all** : SciDBArray

boolean array

any()

Returns whether any elements of each attribute are true.

Returns **any** : SciDBArray

boolean array

approxdc(*index=None*, *scidb_syntax=False*)

Return the number of distinct values of the array or along an axis.

The distinct count is an estimate only.

Parameters **index** : int, optional

Axis along which to operate. By default, flattened input is used.

scidb_syntax : bool, optional (default=False)

If False, index follows the numpy convention (i.e., the array is collapsed over the index'th axis). If True, index follows the SciDB convention (i.e., the array is collapsed over all axes *except* index)

Returns A SciDB array

as_temp(*name=None*)
Create a SciDB TEMP array, stored in RAM

Returns **temp** : SciDBArray

A new array, stored in-memory in the database

att(*a*)
Return the attribute name of the array.

Parameters **a** : int

Index of the attribute to lookup

attribute(*a*)
Return the attribute name of the array.

Parameters **a** : int

Index of the attribute to lookup

attribute_rename(**args*)
Rename a set of attributes

Parameters **args** : (old_name, new_name, ...)

0 or more rename pairs

Returns **renamed** : SciDBArray

The new array

avg(*index=None*, *scidb_syntax=False*)
Return the average of the array or the average along an axis.

Parameters **index** : int, optional

Axis along which to operate. By default, flattened input is used.

scidb_syntax : bool, optional (default=False)

If False, index follows the numpy convention (i.e., the array is collapsed over the index'th axis). If True, index follows the SciDB convention (i.e., the array is collapsed over all axes *except* index)

Returns A SciDB array

collapse()
Flatten and remove all the empty cells.

Returns **collapsed** : SciDBArray

A new 1D dense array, containing all of the nonempty cells in this array.

compress(*mask*, *axis=0*)
Extract a subset of entries along a given axis, where an input mask array is non-null

Parameters **array** : SciDBArray

The array to filter

mask : SciDBArray

A 1-dimensional SciDBArray, whose non-null values indicate the entries to retain

axis : int

The axis of array along which to apply the mask. The shape of array along this axis must be the length of mask

contains_nulls(*attr=None*)
Return True if the array contains null values.

Parameters **attr** : None, int, or array_like

the attribute index/indices to check. If None, then check all.

Returns **contains_nulls** : boolean

contents(***kwargs*)
Return a string representation of the array contents

copy(*new_name=None*, *persistent=False*)
Make a copy of the array in the database

Parameters **new_name** : string (optional)

if specifiedmust be a valid array name which does not already exist in the database.

persistent : boolean (optional)

specify whether the new array is persistent (default=False)

Returns **copy** : SciDBArray

return a copy of the original array

count(*index=None*, *scidb_syntax=False*)
Return the count of the array or the count along an axis.

The count is equal to the number of nonnull elements.

Parameters **index** : int, optional

Axis along which to operate. By default, flattened input is used.

scidb_syntax : bool, optional (default=False)

If False, index follows the numpy convention (i.e., the array is collapsed over the index'th axis). If True, index follows the SciDB convention (i.e., the array is collapsed over all axes *except* index)

Returns A SciDB array

cumprod(*axis=None*)
Return the cumulative product over the array.

Parameters **axis** : int, optional

The axis to multiply over. The default multiplies over the flattened array

Returns **prods** : SciDBArray

A new array, with the same shape (but flattened if axis=None)

See also:

cumsum, `cumulate`

cumsum(*axis=None*)
Return the cumulative sum over the array.

Parameters **axis** : int, optional

The axis to sum over. The default sums over the flattened array

Returns **sums** : SciDBArray

A new array, with the same shape (but flattened if axis=None)

See also:

cumprod, cumulate

cumulate(*expression*, *dimension=0*)

Compute running operations along data (e.g., cumulative sums)

Parameters **expression: str**

A valid SciDB expression

dimension : int or str (optional, default=0)

Which dimension to accumulate over

Returns **arr** : SciDBArray

A new array of the same shape.

See also:

cumsum, *cumprod*

Examples

```
>>> x = sdb.arange(12).reshape((3, 4))
>>> x.cumulate('sum(f0)').toarray()
array([[ 0,  1,  2,  3],
       [ 4,  6,  8, 10],
       [12, 15, 18, 21]])
```

dimension(*d*)

Return the dimension name of the array

Parameters **d** : int

The index of the dimension to lookup

dimension_rename(**args*)

Rename a set of dimensions

Parameters **args** : (old_name, new_name, ...)

0 or more rename pairs

Returns **renamed** : SciDBArray

The new array

eval(*out=None*, *store=True*, ***kwargs*)

If the array is backed by an unevaluated query, evaluate the query and store the result in the database

This changes array.name from a query string to a stored array name. Calling eval() on an array that is already backed by a stored array does nothing.

Parameters **out** : SciDBArray (optional)

An optional pre-existing array to store the evaluation into.

classmethod **from_query**(*interface*, *query*)
Build a lazily-evaulated SciDB array from a query string

Parameters **interface** : SciDBInterface

The database connection to use

query : str

The query string to wrap

Returns **array** : SciDBArray

groupby(*by*)
Build a groupby object from this array

Parameters **by** : string or list of strings

Names of attributes and dimensions to group by

Returns **groups** : *scidbpy.aggregation.GroupBy* instance

An object that can be used, e.g., to perform aggregations over each group. See *scidbpy.aggregation.GroupBy* documentation for more information.

head(*n=5*)
Extract and download the first few elements in the array

Parameters **n** : int (optional, default=5)

The number of elements to retrieve

Returns **head** : SciDBArray

The first N elements in the array, downloaded as a Pandas dataframe (if pandas is installed) or a Numpy array

index_lookup(*idx_array*, *attribute*, *output_attribute=u'idx'*)
Wrapper around the index_lookup AFL call.

This automatically wraps the array name and attribute in an alias, as is required by AFL

Parameters **idx_array** : SciDBArray

A single-attribute array of unique values to lookup

attribute : string

The name of an attribute in this array

output_attribute : string

The attribute of the output containing the indices

Returns **indexed** : SciDBArray

The current array appended with an index attribute

isel(***kwargs*)
Select a subset of the array by dimension name

Parameters **kwargs** : dimension names -> slice descrption

What to select from the array

Returns **subarray** : SciDBArray

The array subset

Examples

x = sdb.arange(20).reshape((4, 5)) print(x.schema) # <f0:int64> [i0=0:3,1000,0,i1=0:4,1000,0] x.isel(i0=0) # x[0] x.isel(i1=2) # x[:, 2] x.isel(i1=slice(2,4)).toarray() # x[:, 2:4]

issparse()
Check whether array is sparse.

max(*index=None*, *scidb_syntax=False*)
Return the maximum of the array or the maximum along an axis.

Parameters **index** : int, optional

Axis along which to operate. By default, flattened input is used.

scidb_syntax : bool, optional (default=False)

If False, index follows the numpy convention (i.e., the array is collapsed over the index'th axis). If True, index follows the SciDB convention (i.e., the array is collapsed over all axes *except* index)

Returns A SciDB array

mean(*index=None*, *scidb_syntax=False*)
Return the average of the array or the average along an axis.

Parameters **index** : int, optional

Axis along which to operate. By default, flattened input is used.

scidb_syntax : bool, optional (default=False)

If False, index follows the numpy convention (i.e., the array is collapsed over the index'th axis). If True, index follows the SciDB convention (i.e., the array is collapsed over all axes *except* index)

Returns A SciDB array

Notes

Identical to *SciDBArray.avg()*

min(*index=None*, *scidb_syntax=False*)
Return the minimum of the array or the minimum along an axis.

Parameters **index** : int, optional

Axis along which to operate. By default, flattened input is used.

scidb_syntax : bool, optional (default=False)

If False, index follows the numpy convention (i.e., the array is collapsed over the index'th axis). If True, index follows the SciDB convention (i.e., the array is collapsed over all axes *except* index)

Returns A SciDB array

nonempty()
Return the number of nonempty elements in the array.

Nonempty refers to the sparsity of an array, and thus includes in the count elements with values which are set to NULL.

See also:

nonnull

nonnull (*attr=0*)

Return the number of non-empty and non-null values.

This query must be done for each attribute: the default is the first attribute.

Parameters **attr** : None, int or array_like

the attribute or attributes to query. If None, then query all attributes.

Returns **nonnull** : array_like

the nonnull count for each attribute. The returned value is the same shape as the input `attr`.

See also:

nonempty

persistent

Controls whether the array is deleted when the database is reaped

reap (*ignore=False*)

Delete this object from the database if it isn't persistent.

Parameters **ignore** : bool (default False)

If False and the array is persistent, then reap raises an error If True and the array is persistent, reap does nothing

Raises **SciDBForbidden if ''persistent=True'' and ''ignore=False'**

regrid (*size*, *aggregate=u'avg'*)

Regrid the array using the specified aggregate

Parameters **size** : int or tuple of ints

Specify the size of the regridding along each dimension. If a single integer, then use the same regridding along each dimension.

aggregate : string

specify the aggregation function to use when creating the new grid. Default is 'avg'. Possible values are: ['avg', 'sum', 'min', 'max', 'count', 'stdev', 'var', 'approxdc']

Returns **A** : scidbarray

The re-gridded version of the array. The size of dimension i is ceil(self.shape[i] / size[i])

relabel (*renames*)

relabel the attributes or dimensions in an array.

Parameters **renames: dict**

A dictionary mapping old names to new names

Returns **renamed** : SciDBArray

A new array

rename (*new_name*, *persistent=False*)

Rename the array in the database, optionally making the new array persistent.

Parameters **new_name** : string

must be a valid array name which does not already exist in the database.

persistent : boolean (optional)

specify whether the new array is persistent (default=False)

Returns **self** : SciDBArray

return a pointer to self

reshape(*shape*, ***kwargs*)

Reshape data into a new array

Parameters **shape** : tuple or int

The shape of the new array. Must be compatible with the current shape

****kwargs :**

additional keyword arguments will be passed to SciDBDatashape

Returns **arr** : SciDBArray

new array of the specified shape

schema

Return the array schema

std(*index=None*, *scidb_syntax=False*)

Return the standard deviation of the array or along an axis.

Parameters **index** : int, optional

Axis along which to operate. By default, flattened input is used.

scidb_syntax : bool, optional (default=False)

If False, index follows the numpy convention (i.e., the array is collapsed over the index'th axis). If True, index follows the SciDB convention (i.e., the array is collapsed over all axes *except* index)

Returns A SciDB array

Notes

Identical to *SciDBArray.stdev()*

stdev(*index=None*, *scidb_syntax=False*)

Return the standard deviation of the array or along an axis.

Parameters **index** : int, optional

Axis along which to operate. By default, flattened input is used.

scidb_syntax : bool, optional (default=False)

If False, index follows the numpy convention (i.e., the array is collapsed over the index'th axis). If True, index follows the SciDB convention (i.e., the array is collapsed over all axes *except* index)

Returns A SciDB array

substitute(*value*)

Reshape data into a new array, substituting a default for any nulls.

Parameters **value** : value to replace nulls (required)

Returns **arr** : SciDBArray

new non-nullable array

Notes

This is currently limited to single-attribute arrays. Use the raw AFL substutute operator for multi-attribute arrays

sum(*index=None*, *scidb_syntax=False*)
Return the sum of the array or the sum along an axis.

Parameters **index** : int, optional

Axis along which to operate. By default, flattened input is used.

scidb_syntax : bool, optional (default=False)

If False, index follows the numpy convention (i.e., the array is collapsed over the index'th axis). If True, index follows the SciDB convention (i.e., the array is collapsed over all axes *except* index)

Returns A SciDB array

toarray(***kwargs*)
Transfer data from database and store in a numpy array.

Parameters **compression** : None, 'auto' or 1-9

Whether to use compression. None disables compression. 'auto' uses the *default_compression* attribute on the SciDB interface object. 1-9 uses gzip compression at the specified level (1=fast, 9=best)

method : 'sparse' or 'dense' (optional, default sparse)

Whether the array to download is sparse or dense.

'sparse' works with all SciDB arrays, but is slower (it computes and transfers array indices for each cell). It is the default.

'dense' transfer only works for bound arrays with no empty cells. It is faster, since it doesn't compute or transfer indices.

transfer_bytes : DEPRECATED

Unused

Returns **arr** : np.ndarray

The dense array containing the data.

Notes

If the array is backed by a query, the query is evaluated and stored in the database

todataframe(***kwargs*)
Transfer array from database and store in a local Pandas dataframe

The array dimensions are assigned to the index of the output.

Parameters **compression** : 'auto', None, or [1-9]

Whether and how to compress the transfer.

Returns **arr** : pd.DataFrame

The dataframe object containing the data in the array.

tolist (***kwargs*)
Download the array as a (nested) python list

tosparse (*sparse_fmt=u'recarray'*, ***kwargs*)
Transfer array from database and store in a local sparse array.

Parameters **sparse_fmt** : string or None

Specify the sparse format to use. Available formats are: - 'recarray' : a record array containing the indices and

values for each data point. This is valid for arrays of any dimension and with any number of attributes.

- ['coo'|'csc'|'csr'|'dok'|'lil'] : a scipy sparse matrix. These are valid only for 2-dimensional arrays with a single attribute.

compression : 'auto', None, or [1-9]

Whether to use compression. None disables compression. 'auto' uses the value from the SciDBInterface's default_compression attribute. 1-9 specifies a gzip-compression level (1=fast, 9=best)

transfer_bytes : deprecated

Unused

Returns **arr** : ndarray or sparse matrix

The sparse representation of the data

transpose (**axes*)
Permute the dimensions of an array.

Parameters **axes** : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

Returns **out** : ndarray

Copy of *a*, with axes suitably permuted.

unpack (*name=u'idx'*)
Unpack with automatic dimension name disambiguation

Unpacking flattens an array to 1D, converting all old dimensions to attributes

Parameters **name** : str (optional, default 'idx')

The name of the new dimension. Will be disambiguated

var (*index=None*, *scidb_syntax=False*)
Return the variance of the array or the variance along an axis.

Parameters **index** : int, optional

Axis along which to operate. By default, flattened input is used.

scidb_syntax : bool, optional (default=False)

If False, index follows the numpy convention (i.e., the array is collapsed over the index'th axis). If True, index follows the SciDB convention (i.e., the array is collapsed over all axes *except* index)

Returns A SciDB array

SciDB Interface

`scidbpy.interface.`**`connect`**(*url=None*, *username=None*, *password=None*)

Connect to a SciDB instance.

Parameters **url** : str (optional)

Connection URL. If not provided, will fall back to the SCIDB_URL environment variable (if present), or http://127.0.0.1:8080. MUST begin with http or https. Username and password are mandatory with https.

username : str (optional)

SciDB username, for authenticated communication. Defaults to the value of the SCIDB_USER environment variable. If that doesn't exist, unauthetnicated communication is used.

password : str (optional)

SciDB password, for authenticated communication. Defaults to the value of the SCIDB_PASSWORD environment variable. If that doesn't exist, unauthetnicated communication is used

Returns A SciDBShimInterface connection to the database.

Base Class

class `scidbpy.interface.`**`SciDBInterface`**

Attributes

`afl`	
`default_compression`	The default compression to use when downloading data

Methods

`acos`(A)	Element-wise trigonometric inverse cosine
`approxdc`(A[, index, scidb_syntax])	Array or axis unique element estimate.
`arange`([start,] stop[, step,][, dtype])	Return evenly spaced values within a given interval.
`asin`(A)	Element-wise trigonometric inverse sine
`atan`(A)	Element-wise trigonometric inverse tangent
`avg`(A[, index, scidb_syntax])	Array or axis average.
`ceil`(A)	Element-wise ceiling function
`concatenate`(arrays[, axis])	Concatenate several arrays along a particular dimension.

Co

Table 1.4 – continued from previous page

cos(A)	Element-wise trigonometric cosine
count(A[, index, scidb_syntax])	Array or axis count.
cross_join(A, B, *dims)	Perform a cross-join on arrays A and B.
dot(A, B)	Compute the matrix product of A and B
dstack(arrays)	Stack arrays in sequence depth wise (along the third axis).
exp(A)	Element-wise natural exponent
floor(A)	Element-wise floor function
from_array(A[, instance_id, chunk_size])	Initialize a scidb array from a numpy array
from_dataframe(A[, instance_id])	Initialize a scidb array from a pandas dataframe
from_sparse(A[, instance_id])	Initialize a scidb array from a sparse array
hstack(arrays)	Stack arrays in sequence horizontally (column wise).
identity(n[, dtype, sparse])	Return a 2-dimensional square identity matrix of size n
isnan(A)	Element-wise nan test function
join(*args)	Perform a series of array joins on the arguments and return the result.
linspace(start, stop[, num, endpoint, retstep])	Return evenly spaced numbers over a specified interval.
list_arrays()	List the arrays currently in the database
log(A)	Element-wise natural logarithm
log10(A)	Element-wise base-10 logarithm
ls([pattern])	List the arrays in the database, optionally matching to a pattern
max(A[, index, scidb_syntax])	Array or axis maximum.
mean(A[, index, scidb_syntax])	Array or axis mean.
merge(left, right[, on, left_on, right_on, ...])	Perform a Pandas-like join on two SciDBArrays.
min(A[, index, scidb_syntax])	Array or axis minimum.
new_array([shape, dtype, persistent, name])	Create a new array, either instantiating it in SciDB or simply reserving the name
normalize(A)	
ones(shape[, dtype])	Return an array of ones
percentile(a, q[, att])	Compute the qth percentile of the data along the specified axis
query(query, *args, **kwargs)	Perform a query on the database.
randint(shape[, dtype, lower, upper, persistent])	Return an array of random integers between lower and upper
random(shape[, dtype, lower, upper, persistent])	Return an array of random floats between lower and upper
reap()	Reap all arrays created via new_array
remove(array)	Remove an array from the database
sin(A)	Element-wise trigonometric sine
sqrt(A)	Element-wise square root
std(A[, index, scidb_syntax])	Array or axis standard deviation.
stdev(A[, index, scidb_syntax])	Array or axis standard deviation.
substitute(A, value)	Replace null values in an array
sum(A[, index, scidb_syntax])	Array or axis sum.
svd(A[, return_U, return_S, return_VT])	Compute the Singular Value Decomposition of the array A:
tan(A)	Element-wise trigonometric tangent
toarray(A[, transfer_bytes])	Convert a SciDB array to a numpy array
todataframe(A[, transfer_bytes])	Convert a SciDB array to a pandas dataframe
tosparse(A[, sparse_fmt, transfer_bytes])	Convert a SciDB array to a sparse representation
unique(x[, is_sorted])	Store the unique elements of an array in a new array
var(A[, index, scidb_syntax])	Array or axis variance.
vstack(arrays)	Stack arrays in sequence vertically (column wise).
wrap_array(scidbname[, persistent])	Create a new SciDBArray object that references an existing SciDB
zeros(shape[, dtype])	Return an array of zeros

acos(*A*)

Element-wise trigonometric inverse cosine

approxdc(*A*, *index=None*, *scidb_syntax=False*)
Array or axis unique element estimate.

see `SciDBArray.approxdc()`

arange([*start*,] *stop*[, *step*,] *dtype=None*, ***kwargs*)
Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the behavior is equivalent to the Python range function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use `linspace` for these cases.

Parameters **start** : number, optional

Start of interval. The interval includes this value. The default start value is 0.

stop : number

End of interval. The interval does not include this value, except in some cases where *step* is not an integer and floating point round-off affects the length of *out*.

step : number, optional

Spacing between values. For any output *out*, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If *step* is specified, *start* must also be given.

dtype : dtype

The type of the output array. If *dtype* is not given, it is inferred from the type of the input arguments.

****kwargs :**

Additional arguments are passed to SciDBDatashape when creating the output array.

Returns **arange** : SciDBArray

Array of evenly spaced values.

For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of *out* being greater than *stop*.

asin(*A*)
Element-wise trigonometric inverse sine

atan(*A*)
Element-wise trigonometric inverse tangent

avg(*A*, *index=None*, *scidb_syntax=False*)
Array or axis average.

see `SciDBArray.avg()`

ceil(*A*)
Element-wise ceiling function

concatenate(*arrays*, *axis=0*)
Concatenate several arrays along a particular dimension.

This behaves like numpy's concatenate function when the input array dimensions are > the concatenation axis. It behaves differently than numpy when the array dimensions are less than the concatenation axis, in the following way:

Concatenating 1D arrays along axis=0 behaves like numpy's vstack. Concatenating 1D arrays along axis=1 behaves like numpy's hstack. Concatenating 1D or 2D arrays along axis=2 behaves like dstack.

Parameters **arrays** : Sequence of SciDBArrays

The arrays to concatenate

axis : int, optional (default 0)

The dimension to join on. Array shapes must match along all dimensions except this axis.

Returns **stacked** : SciDBArray

A stacked array

See also:

hstack, *vstack*, *dstack*

cos (*A*)
Element-wise trigonometric cosine

count (*A*, *index=None*, *scidb_syntax=False*)
Array or axis count.

see `SciDBArray.count()`

cross_join (*A*, *B*, **dims*)
Perform a cross-join on arrays A and B.

Parameters **A, B** : SciDBArray

***dims** : tuples

The remaining arguments are tuples of dimension indices which should be joined.

default_compression
The default compression to use when downloading data

dot (*A*, *B*)
Compute the matrix product of A and B

Parameters **A** : SciDBArray

A must be a two-dimensional matrix of shape (n, p)

B : SciDBArray

B must be a two-dimensional matrix of shape (p, m)

Returns **C** : SciDBArray

The wrapper of the SciDB Array, of shape (n, m), consisting of the matrix product of A and B

dstack (*arrays*)
Stack arrays in sequence depth wise (along the third axis).

Parameters **arrays** : Sequence of SciDBArrays

The arrays to join. All arrays must have the same shape along all but the third dimension.

Returns **stacked** : SciDBArray

The array formed by stacking the given arrays.

See also:

hstack, *vstack*, *concatenate*

exp (*A*)
Element-wise natural exponent

floor (*A*)
Element-wise floor function

from_array (*A*, *instance_id=0*, *chunk_size=1000*, ***kwargs*)
Initialize a scidb array from a numpy array

Parameters **A** : array_like (numpy array or sparse array)

input array from which the scidb array will be created

instance_id : integer

the instance ID used in loading (default=0; see SciDB documentation)

chunk_size : integer or list of integers

The chunk size of the uploaded SciDBArray. Default=1000

****kwargs :**

Additional keyword arguments are passed to new_array()

Returns **arr** : SciDBArray

SciDB Array object built from the input array

from_dataframe (*A*, *instance_id=0*, ***kwargs*)
Initialize a scidb array from a pandas dataframe

Parameters **A** : pandas dataframe

data from which the scidb array will be created.

instance_id : integer

the instance ID used in loading (default=0; see SciDB documentation)

****kwargs :**

Additional keyword arguments are passed to new_array()

Returns **arr** : SciDBArray

SciDB Array object built from the input array

from_sparse (*A*, *instance_id=0*, ***kwargs*)
Initialize a scidb array from a sparse array

Parameters **A** : sparse array

sparse input array from which the scidb array will be created. Note that this array will internally be converted to COO format.

instance_id : integer

the instance ID used in loading (default=0; see SciDB documentation)

****kwargs :**

Additional keyword arguments are passed to new_array()

Returns **arr** : SciDBArray

SciDB Array object built from the input array

hstack(*arrays*)

Stack arrays in sequence horizontally (column wise).

Parameters **arrays** : Sequence of SciDBArrays

The arrays to join. All arrays must have the same shape along all but the second dimension.

Returns **stacked** : SciDBArray

The array formed by stacking the given arrays.

See also:

vstack, *dstack*, *concatenate*

identity(*n*, *dtype=u'double'*, *sparse=False*, ***kwargs*)

Return a 2-dimensional square identity matrix of size n

Parameters **n** : integer

the number of rows and columns in the matrix

dtype : string or list

The data type of the array

sparse : boolean

specify whether to create a sparse array (default=False)

****kwargs :**

Additional keyword arguments are passed to SciDBDataShape.

Returns **arr** : SciDBArray

A SciDBArray containint an [n x n] identity matrix

isnan(*A*)

Element-wise nan test function

join(**args*)

Perform a series of array joins on the arguments and return the result.

linspace(*start*, *stop*, *num=50*, *endpoint=True*, *retstep=False*, ***kwargs*)

Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

The endpoint of the interval can optionally be excluded.

Parameters **start** : scalar

The starting value of the sequence.

stop : scalar

The end value of the sequence, unless *endpoint* is set to False. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that *stop* is excluded. Note that the step size changes when *endpoint* is False.

num : int, optional

Number of samples to generate. Default is 50.

endpoint : bool, optional

If True, *stop* is the last sample. Otherwise, it is not included. Default is True.

retstep : bool, optional

If True, return (*samples*, *step*), where *step* is the spacing between samples.

****kwargs :**

additional keyword arguments are passed to SciDBDataShape

Returns samples : SciDBArray

There are *num* equally spaced samples in the closed interval `[start, stop]` or the half-open interval `[start, stop)` (depending on whether *endpoint* is True or False).

step : float (only if *retstep* is True)

Size of spacing between samples.

list_arrays()
List the arrays currently in the database Returns ——- array_list : dictionary

A mapping of array name -> schema

log(*A*)
Element-wise natural logarithm

log10(*A*)
Element-wise base-10 logarithm

ls(*pattern=u'*'*)
List the arrays in the database, optionally matching to a pattern

Parameters pattern : String (optional)

A glob-style pattern string. If present, only arrays whose names match the pattern are displayed. '*' matches any string, '?' matches any character

Returns result : list

A list of SciDB array names

max(*A*, *index=None*, *scidb_syntax=False*)
Array or axis maximum.

see `SciDBArray.max()`

mean(*A*, *index=None*, *scidb_syntax=False*)
Array or axis mean.

see `SciDBArray.mean()`

static merge(*left*, *right*, *on=None*, *left_on=None*, *right_on=None*, *how=u'inner'*, *suffixes=(u'_x'*, *u'_y')*)
Perform a Pandas-like join on two SciDBArrays.

Parameters left : SciDBArray

The left array to join on

right : SciDBArray

The right array to join on

on : None, string, or list of strings

The names of dimensions or attributes to join on. Either on or both *left_on* and *right_on* must be supplied. If on is supplied, the specified names must exist in both left and right

left_on : None, string, or list of strings

The names of dimensions or attributes in the left array to join on. If provided, then right_on must also be provided, and have as many elements as left_on

right_on : None, string, or list of strings

The name of dimensions or attributes in the right array to join on. See notes above for left_join

how : 'inner' | 'left' | 'right' | 'outer'

The kind of join to perform. Currently, only 'inner' is supported.

suffixes : tuple of two strings

The suffix to add to array dimensions or attributes which are duplicated in left and right.

Returns joined : SciDBArray

The joined array.

Notes

When joining on attributes, a categorical index is computed for each array. This index will appear as a dimension in the output.

This function builds an AFL join or cross join query, performing preprocessing on the inputs as necessary to match chunk sizes, avoid name collisions, etc.

If neither on, left_on, or right_on are provided, then the join defaults to the overlapping dimension names.

min(*A*, *index=None*, *scidb_syntax=False*)
Array or axis minimum.

see `SciDBArray.min()`

new_array(*shape=None*, *dtype=u'double'*, *persistent=False*, *name=None*, ***kwargs*)
Create a new array, either instantiating it in SciDB or simply reserving the name for use in a later query.

Parameters shape : int or tuple (optional)

The shape of the array to create. If not specified, no array will be created and a name will simply be reserved for later use. WARNING: if shape=None and persistent=False, an error will result when the array goes out of scope, unless the name is used to create an array on the server.

dtype : string (optional)

the datatype of the array. This is only referenced if *shape* is specified. Default is 'double'.

persistent : boolean (optional)

whether the created array should be persistent, i.e. survive in SciDB past when the object wrapper goes out of scope. Default is False.

name : str (optional)

The name to give the array in the databse. If present, persistent will be set to True.

****kwargs** : (optional)

If *shape* is specified, additional keyword arguments are passed to SciDBDataShape. Otherwise, these will not be referenced.

Returns

——-

arr : SciDBArray

wrapper of the new SciDB array instance.

ones(*shape*, *dtype=u'double'*, ***kwargs*)

Return an array of ones

Parameters **shape** : tuple or int

The shape of the array

dtype : string or list

The data type of the array

****kwargs :**

Additional keyword arguments are passed to SciDBDataShape.

Returns arr: SciDBArray

A SciDBArray consisting of all ones.

percentile(*a*, *q*, *att=None*)

Compute the qth percentile of the data along the specified axis

Parameters **a** : SciDBArray

Input array

q : float in the range [0, 100] or a sequence of floats

The percentiles to compute

att : str, optional

The array attribute to compute percentiles for. Defaults to the first attribute

Returns **qs** : SciDBArray

An array with as many elements as q, listing the data value at each percentile

query(*query*, **args*, ***kwargs*)

Perform a query on the database.

This wraps a query constructor which allows the creation of sophisticated SciDB queries which act on arrays wrapped by SciDBArray objects. See Notes below for details.

Parameters **query** : string

The query string, with curly-braces to indicate insertions

***args, **kwargs :**

Values to be inserted (see below).

randint(*shape*, *dtype=u'uint32'*, *lower=0*, *upper=2147483647*, *persistent=False*, ***kwargs*)

Return an array of random integers between lower and upper

Parameters **shape** : tuple or int

The shape of the array

dtype : string or list

The data type of the array

lower : float

The lower bound of the random sample (default=0)

upper : float

The upper bound of the random sample (default=2147483647)

persistent : bool

Whether the array is persistent (default=False)

****kwargs :**

Additional keyword arguments are passed to SciDBDataShape.

Returns arr: SciDBArray

A SciDBArray consisting of random integers, uniformly distributed between *lower* and *upper*.

random(*shape*, *dtype=u'double'*, *lower=0*, *upper=1*, *persistent=False*, ***kwargs*)

Return an array of random floats between lower and upper

Parameters **shape** : tuple or int

The shape of the array

dtype : string or list

The data type of the array

lower : float

The lower bound of the random sample (default=0)

upper : float

The upper bound of the random sample (default=1)

persistent : bool

Whether the new array is persistent (default=False)

****kwargs :**

Additional keyword arguments are passed to SciDBDataShape.

Returns arr: SciDBArray

A SciDBArray consisting of random floating point numbers, uniformly distributed between *lower* and *upper*.

reap()

Reap all arrays created via new_array

remove(*array*)

Remove an array from the database

This removes the array even if its persistent property is True!

Parameters **array** : str or SciDBArray

The array (or name of array) to remove

See also:

reap, `SciDBArray.reap`

sin(*A*)
Element-wise trigonometric sine

sqrt(*A*)
Element-wise square root

std(*A*, *index=None*, *scidb_syntax=False*)
Array or axis standard deviation.

see `SciDBArray.std()`

stdev(*A*, *index=None*, *scidb_syntax=False*)
Array or axis standard deviation.

see `SciDBArray.stdev()`

substitute(*A*, *value*)
Replace null values in an array

See `SciDBArray.substitute()`

sum(*A*, *index=None*, *scidb_syntax=False*)
Array or axis sum.

see `SciDBArray.sum()`

svd(*A*, *return_U=True*, *return_S=True*, *return_VT=True*)
Compute the Singular Value Decomposition of the array A:

A = U.S.V^T

Parameters **A** : SciDBArray

The array for which the SVD will be computed. It should be a 2-dimensional array with a single value per cell. Currently, the svd routine requires non-overlapping chunks of size 32.

return_U, return_S, return_VT : boolean

if any is True, then return the associated array. All are True by default

Returns **[U], [S], [VT]** : SciDBArrays

Arrays storing the singular values and vectors of A.

tan(*A*)
Element-wise trigonometric tangent

toarray(*A*, *transfer_bytes=True*)
Convert a SciDB array to a numpy array

todataframe(*A*, *transfer_bytes=True*)
Convert a SciDB array to a pandas dataframe

tosparse(*A*, *sparse_fmt=u'recarray'*, *transfer_bytes=True*)
Convert a SciDB array to a sparse representation

unique(*x*, *is_sorted=False*)
Store the unique elements of an array in a new array

Parameters **x** : SciDBArray

The array to compute unique elements of.

is_sorted : bool

Whether the array is pre-sorted. If True, x must be a 1D array.

Returns **u** : SciDBArray

The unique elements of x

var(*A*, *index=None*, *scidb_syntax=False*)
Array or axis variance.

see `SciDBArray.var()`

vstack(*arrays*)
Stack arrays in sequence vertically (column wise).

Parameters **arrays** : Sequence of SciDBArrays

The arrays to join. All arrays must have the same shape along all but the first dimension.

Returns **stacked** : SciDBArray

The array formed by stacking the given arrays.

See also:

hstack, *dstack*, *concatenate*

wrap_array(*scidbname*, *persistent=True*)
Create a new SciDBArray object that references an existing SciDB array

Parameters **scidbname** : string

Wrap an existing scidb array referred to by *scidbname*. The SciDB array object persistent value will be set to True, and the object shape, datashape and data type values will be determined by the SciDB array.

persistent : boolean

If True (default) then array will not be deleted when this variable goes out of scope. Warning: if persistent is set to False, data could be lost!

zeros(*shape*, *dtype=u'double'*, ***kwargs*)
Return an array of zeros

Parameters **shape** : tuple or int

The shape of the array

dtype : string or list

The data type of the array

****kwargs :**

Additional keyword arguments are passed to SciDBDataShape.

Returns arr: SciDBArray

A SciDBArray consisting of all zeros.

Shim Interface

class `scidbpy.interface.`**SciDBShimInterface**(*hostname*, *user=None*, *password=None*, *pam=None*, *digest=None*)
HTTP interface to SciDB via shim [1]_

Parameters **hostname** : string

A URL pointing to a running shim/SciDB session

user : string (optional)

A username, for authentication

password : string (optional)

A password, for authentication

pam : bool (optional)

Whether to use PAM authentication. If *True*, then user and password are required. If *None*, will be guessed based on hostname and password values

digest : bool (optional)

Whether to use Digest authentication. If *True*, then user and password are required. If *None*, will be guessed based on hostname and password values.

[1] https://github.com/Paradigm4/shim

Attributes

afl	
default_compression	The default compression to use when downloading data

Methods

acos(A)	Element-wise trigonometric inverse cosine
approxdc(A[, index, scidb_syntax])	Array or axis unique element estimate.
arange([start,] stop[, step,][, dtype])	Return evenly spaced values within a given interval.
asin(A)	Element-wise trigonometric inverse sine
atan(A)	Element-wise trigonometric inverse tangent
avg(A[, index, scidb_syntax])	Array or axis average.
ceil(A)	Element-wise ceiling function
concatenate(arrays[, axis])	Concatenate several arrays along a particular dimension.
cos(A)	Element-wise trigonometric cosine
count(A[, index, scidb_syntax])	Array or axis count.
cross_join(A, B, *dims)	Perform a cross-join on arrays A and B.
dot(A, B)	Compute the matrix product of A and B
dstack(arrays)	Stack arrays in sequence depth wise (along the third axis).
exp(A)	Element-wise natural exponent
floor(A)	Element-wise floor function
from_array(A[, instance_id, chunk_size])	Initialize a scidb array from a numpy array
from_dataframe(A[, instance_id])	Initialize a scidb array from a pandas dataframe
from_sparse(A[, instance_id])	Initialize a scidb array from a sparse array
hstack(arrays)	Stack arrays in sequence horizontally (column wise).
identity(n[, dtype, sparse])	Return a 2-dimensional square identity matrix of size n
isnan(A)	Element-wise nan test function
join(*args)	Perform a series of array joins on the arguments and return the result.
linspace(start, stop[, num, endpoint, retstep])	Return evenly spaced numbers over a specified interval.

Co

Table 1.6 – continued from previous page

list_arrays()	List the arrays currently in the database
log(A)	Element-wise natural logarithm
log10(A)	Element-wise base-10 logarithm
login(user, password)	Login using PAM authentication (e.g., over HTTPS)
logout()	Logout from PAM authentication (e.g., over HTTPS)
ls([pattern])	List the arrays in the database, optionally matching to a pattern
max(A[, index, scidb_syntax])	Array or axis maximum.
mean(A[, index, scidb_syntax])	Array or axis mean.
merge(left, right[, on, left_on, right_on, ...])	Perform a Pandas-like join on two SciDBArrays.
min(A[, index, scidb_syntax])	Array or axis minimum.
new_array([shape, dtype, persistent, name])	Create a new array, either instantiating it in SciDB or simply reserving the name
normalize(A)	
ones(shape[, dtype])	Return an array of ones
percentile(a, q[, att])	Compute the qth percentile of the data along the specified axis
query(query, *args, **kwargs)	Perform a query on the database.
randint(shape[, dtype, lower, upper, persistent])	Return an array of random integers between lower and upper
random(shape[, dtype, lower, upper, persistent])	Return an array of random floats between lower and upper
reap()	Reap all arrays created via new_array
remove(array)	Remove an array from the database
sin(A)	Element-wise trigonometric sine
sqrt(A)	Element-wise square root
std(A[, index, scidb_syntax])	Array or axis standard deviation.
stdev(A[, index, scidb_syntax])	Array or axis standard deviation.
substitute(A, value)	Replace null values in an array
sum(A[, index, scidb_syntax])	Array or axis sum.
svd(A[, return_U, return_S, return_VT])	Compute the Singular Value Decomposition of the array A:
tan(A)	Element-wise trigonometric tangent
toarray(A[, transfer_bytes])	Convert a SciDB array to a numpy array
todataframe(A[, transfer_bytes])	Convert a SciDB array to a pandas dataframe
tosparse(A[, sparse_fmt, transfer_bytes])	Convert a SciDB array to a sparse representation
unique(x[, is_sorted])	Store the unique elements of an array in a new array
var(A[, index, scidb_syntax])	Array or axis variance.
vstack(arrays)	Stack arrays in sequence vertically (column wise).
wrap_array(scidbname[, persistent])	Create a new SciDBArray object that references an existing SciDB
zeros(shape[, dtype])	Return an array of zeros

acos(*A*)
Element-wise trigonometric inverse cosine

approxdc(*A*, *index=None*, *scidb_syntax=False*)
Array or axis unique element estimate.

see `SciDBArray.approxdc()`

arange([*start*], *stop*[, *step*], *dtype=None*, ***kwargs*)
Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the behavior is equivalent to the Python range function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use `linspace` for these cases.

Parameters **start** : number, optional

Start of interval. The interval includes this value. The default start value is 0.

stop : number

End of interval. The interval does not include this value, except in some cases where *step* is not an integer and floating point round-off affects the length of *out*.

step : number, optional

Spacing between values. For any output *out*, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If *step* is specified, *start* must also be given.

dtype : dtype

The type of the output array. If *dtype* is not given, it is inferred from the type of the input arguments.

****kwargs :**

Additional arguments are passed to SciDBDatashape when creating the output array.

Returns **arange** : SciDBArray

Array of evenly spaced values.

For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of *out* being greater than *stop*.

asin(*A*)
Element-wise trigonometric inverse sine

atan(*A*)
Element-wise trigonometric inverse tangent

avg(*A*, *index=None*, *scidb_syntax=False*)
Array or axis average.

see `SciDBArray.avg()`

ceil(*A*)
Element-wise ceiling function

concatenate(*arrays*, *axis=0*)
Concatenate several arrays along a particular dimension.

This behaves like numpy's concatenate function when the input array dimensions are > the concatenation axis. It behaves differently than numpy when the array dimensions are less than the concatenation axis, in the following way:

Concatenating 1D arrays along axis=0 behaves like numpy's vstack. Concatenating 1D arrays along axis=1 behaves like numpy's hstack. Concatenating 1D or 2D arrays along axis=2 behaves like dstack.

Parameters **arrays** : Sequence of SciDBArrays

The arrays to concatenate

axis : int, optional (default 0)

The dimension to join on. Array shapes must match along all dimensions except this axis.

Returns **stacked** : SciDBArray

A stacked array

See also:

hstack, *vstack*, *dstack*

cos (*A*)
Element-wise trigonometric cosine

count (*A*, *index=None*, *scidb_syntax=False*)
Array or axis count.

see `SciDBArray.count()`

cross_join (*A*, *B*, **dims*)
Perform a cross-join on arrays A and B.

Parameters **A, B** : SciDBArray

***dims** : tuples

The remaining arguments are tuples of dimension indices which should be joined.

default_compression
The default compression to use when downloading data

dot (*A*, *B*)
Compute the matrix product of A and B

Parameters **A** : SciDBArray

A must be a two-dimensional matrix of shape (n, p)

B : SciDBArray

B must be a two-dimensional matrix of shape (p, m)

Returns **C** : SciDBArray

The wrapper of the SciDB Array, of shape (n, m), consisting of the matrix product of A and B

dstack (*arrays*)
Stack arrays in sequence depth wise (along the third axis).

Parameters **arrays** : Sequence of SciDBArrays

The arrays to join. All arrays must have the same shape along all but the third dimension.

Returns **stacked** : SciDBArray

The array formed by stacking the given arrays.

See also:

hstack, *vstack*, *concatenate*

exp (*A*)
Element-wise natural exponent

floor (*A*)
Element-wise floor function

from_array (*A*, *instance_id=0*, *chunk_size=1000*, ***kwargs*)
Initialize a scidb array from a numpy array

Parameters **A** : array_like (numpy array or sparse array)

input array from which the scidb array will be created

instance_id : integer

the instance ID used in loading (default=0; see SciDB documentation)

chunk_size : integer or list of integers

The chunk size of the uploaded SciDBArray. Default=1000

****kwargs :**

Additional keyword arguments are passed to new_array()

Returns arr : SciDBArray

SciDB Array object built from the input array

from_dataframe(*A*, *instance_id=0*, ***kwargs*)
Initialize a scidb array from a pandas dataframe

Parameters A : pandas dataframe

data from which the scidb array will be created.

instance_id : integer

the instance ID used in loading (default=0; see SciDB documentation)

****kwargs :**

Additional keyword arguments are passed to new_array()

Returns arr : SciDBArray

SciDB Array object built from the input array

from_sparse(*A*, *instance_id=0*, ***kwargs*)
Initialize a scidb array from a sparse array

Parameters A : sparse array

sparse input array from which the scidb array will be created. Note that this array will internally be converted to COO format.

instance_id : integer

the instance ID used in loading (default=0; see SciDB documentation)

****kwargs :**

Additional keyword arguments are passed to new_array()

Returns arr : SciDBArray

SciDB Array object built from the input array

hstack(*arrays*)
Stack arrays in sequence horizontally (column wise).

Parameters arrays : Sequence of SciDBArrays

The arrays to join. All arrays must have the same shape along all but the second dimension.

Returns stacked : SciDBArray

The array formed by stacking the given arrays.

See also:

vstack, *dstack*, *concatenate*

identity (*n*, *dtype=u'double'*, *sparse=False*, ***kwargs*)
Return a 2-dimensional square identity matrix of size n

Parameters **n** : integer

the number of rows and columns in the matrix

dtype : string or list

The data type of the array

sparse : boolean

specify whether to create a sparse array (default=False)

****kwargs :**

Additional keyword arguments are passed to SciDBDataShape.

Returns **arr** : SciDBArray

A SciDBArray containint an [n x n] identity matrix

isnan (*A*)
Element-wise nan test function

join (**args*)
Perform a series of array joins on the arguments and return the result.

linspace (*start*, *stop*, *num=50*, *endpoint=True*, *retstep=False*, ***kwargs*)
Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

The endpoint of the interval can optionally be excluded.

Parameters **start** : scalar

The starting value of the sequence.

stop : scalar

The end value of the sequence, unless *endpoint* is set to False. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that *stop* is excluded. Note that the step size changes when *endpoint* is False.

num : int, optional

Number of samples to generate. Default is 50.

endpoint : bool, optional

If True, *stop* is the last sample. Otherwise, it is not included. Default is True.

retstep : bool, optional

If True, return (*samples*, *step*), where *step* is the spacing between samples.

****kwargs :**

additional keyword arguments are passed to SciDBDataShape

Returns **samples** : SciDBArray

There are *num* equally spaced samples in the closed interval `[start, stop]` or the half-open interval `[start, stop)` (depending on whether *endpoint* is True or False).

step : float (only if *retstep* is True)

Size of spacing between samples.

list_arrays()
List the arrays currently in the database Returns ——- array_list : dictionary

A mapping of array name -> schema

log(*A*)
Element-wise natural logarithm

log10(*A*)
Element-wise base-10 logarithm

login(*user*, *password*)
Login using PAM authentication (e.g., over HTTPS)

logout()
Logout from PAM authentication (e.g., over HTTPS)

ls(*pattern=u'*'*)
List the arrays in the database, optionally matching to a pattern

Parameters pattern : String (optional)

A glob-style pattern string. If present, only arrays whose names match the pattern are displayed. '*' matches any string, '?' matches any character

Returns result : list

A list of SciDB array names

max(*A*, *index=None*, *scidb_syntax=False*)
Array or axis maximum.

see `SciDBArray.max()`

mean(*A*, *index=None*, *scidb_syntax=False*)
Array or axis mean.

see `SciDBArray.mean()`

merge(*left*, *right*, *on=None*, *left_on=None*, *right_on=None*, *how=u'inner'*, *suffixes=(u'_x', u'_y')*)
Perform a Pandas-like join on two SciDBArrays.

Parameters left : SciDBArray

The left array to join on

right : SciDBArray

The right array to join on

on : None, string, or list of strings

The names of dimensions or attributes to join on. Either on or both *left_on* and *right_on* must be supplied. If on is supplied, the specified names must exist in both left and right

left_on : None, string, or list of strings

The names of dimensions or attributes in the left array to join on. If provided, then right_on must also be provided, and have as many elements as left_on

right_on : None, string, or list of strings

The name of dimensions or attributes in the right array to join on. See notes above for left_join

how : 'inner' | 'left' | 'right' | 'outer'

The kind of join to perform. Currently, only 'inner' is supported.

suffixes : tuple of two strings

The suffix to add to array dimensions or attributes which are duplicated in left and right.

Returns joined : SciDBArray

The joined array.

Notes

When joining on attributes, a categorical index is computed for each array. This index will appear as a dimension in the output.

This function builds an AFL join or cross join query, performing preprocessing on the inputs as necessary to match chunk sizes, avoid name collisions, etc.

If neither on, left_on, or right_on are provided, then the join defaults to the overlapping dimension names.

min(*A*, *index=None*, *scidb_syntax=False*)
Array or axis minimum.

see `SciDBArray.min()`

new_array(*shape=None*, *dtype=u'double'*, *persistent=False*, *name=None*, ***kwargs*)
Create a new array, either instantiating it in SciDB or simply reserving the name for use in a later query.

Parameters shape : int or tuple (optional)

The shape of the array to create. If not specified, no array will be created and a name will simply be reserved for later use. WARNING: if shape=None and persistent=False, an error will result when the array goes out of scope, unless the name is used to create an array on the server.

dtype : string (optional)

the datatype of the array. This is only referenced if *shape* is specified. Default is 'double'.

persistent : boolean (optional)

whether the created array should be persistent, i.e. survive in SciDB past when the object wrapper goes out of scope. Default is False.

name : str (optional)

The name to give the array in the databse. If present, persistent will be set to True.

****kwargs** : (optional)

If *shape* is specified, additional keyword arguments are passed to SciDBDataShape. Otherwise, these will not be referenced.

Returns

———-

arr : SciDBArray

wrapper of the new SciDB array instance.

ones(*shape*, *dtype=u'double'*, ***kwargs*)
Return an array of ones

Parameters **shape** : tuple or int

The shape of the array

dtype : string or list

The data type of the array

****kwargs :**

Additional keyword arguments are passed to SciDBDataShape.

Returns arr: SciDBArray

A SciDBArray consisting of all ones.

percentile(*a*, *q*, *att=None*)
Compute the qth percentile of the data along the specified axis

Parameters **a** : SciDBArray

Input array

q : float in the range [0, 100] or a sequence of floats

The percentiles to compute

att : str, optional

The array attribute to compute percentiles for. Defaults to the first attribute

Returns **qs** : SciDBArray

An array with as many elements as q, listing the data value at each percentile

query(*query*, **args*, ***kwargs*)
Perform a query on the database.

This wraps a query constructor which allows the creation of sophisticated SciDB queries which act on arrays wrapped by SciDBArray objects. See Notes below for details.

Parameters **query** : string

The query string, with curly-braces to indicate insertions

***args, **kwargs :**

Values to be inserted (see below).

randint(*shape*, *dtype=u'uint32'*, *lower=0*, *upper=2147483647*, *persistent=False*, ***kwargs*)
Return an array of random integers between lower and upper

Parameters **shape** : tuple or int

The shape of the array

dtype : string or list

The data type of the array

lower : float

The lower bound of the random sample (default=0)

upper : float

The upper bound of the random sample (default=2147483647)

persistent : bool

Whether the array is persistent (default=False)

****kwargs :**

Additional keyword arguments are passed to SciDBDataShape.

Returns arr: SciDBArray

A SciDBArray consisting of random integers, uniformly distributed between *lower* and *upper*.

random(*shape*, *dtype=u'double'*, *lower=0*, *upper=1*, *persistent=False*, ***kwargs*)

Return an array of random floats between lower and upper

Parameters **shape** : tuple or int

The shape of the array

dtype : string or list

The data type of the array

lower : float

The lower bound of the random sample (default=0)

upper : float

The upper bound of the random sample (default=1)

persistent : bool

Whether the new array is persistent (default=False)

****kwargs :**

Additional keyword arguments are passed to SciDBDataShape.

Returns arr: SciDBArray

A SciDBArray consisting of random floating point numbers, uniformly distributed between *lower* and *upper*.

reap()

Reap all arrays created via new_array

remove(*array*)

Remove an array from the database

This removes the array even if its persistent property is True!

Parameters **array** : str or SciDBArray

The array (or name of array) to remove

See also:

reap, `SciDBArray.reap`

sin(*A*)

Element-wise trigonometric sine

sqrt(*A*)

Element-wise square root

std(*A*, *index=None*, *scidb_syntax=False*)
Array or axis standard deviation.

see `SciDBArray.std()`

stdev(*A*, *index=None*, *scidb_syntax=False*)
Array or axis standard deviation.

see `SciDBArray.stdev()`

substitute(*A*, *value*)
Replace null values in an array

See `SciDBArray.substitute()`

sum(*A*, *index=None*, *scidb_syntax=False*)
Array or axis sum.

see `SciDBArray.sum()`

svd(*A*, *return_U=True*, *return_S=True*, *return_VT=True*)
Compute the Singular Value Decomposition of the array A:

A = U.S.V^T

Parameters **A** : SciDBArray

The array for which the SVD will be computed. It should be a 2-dimensional array with a single value per cell. Currently, the svd routine requires non-overlapping chunks of size 32.

return_U, return_S, return_VT : boolean

if any is True, then return the associated array. All are True by default

Returns **[U], [S], [VT]** : SciDBArrays

Arrays storing the singular values and vectors of A.

tan(*A*)
Element-wise trigonometric tangent

toarray(*A*, *transfer_bytes=True*)
Convert a SciDB array to a numpy array

todataframe(*A*, *transfer_bytes=True*)
Convert a SciDB array to a pandas dataframe

tosparse(*A*, *sparse_fmt=u'recarray'*, *transfer_bytes=True*)
Convert a SciDB array to a sparse representation

unique(*x*, *is_sorted=False*)
Store the unique elements of an array in a new array

Parameters **x** : SciDBArray

The array to compute unique elements of.

is_sorted : bool

Whether the array is pre-sorted. If True, x must be a 1D array.

Returns **u** : SciDBArray

The unique elements of x

var (*A*, *index=None*, *scidb_syntax=False*)
Array or axis variance.

see `SciDBArray.var()`

vstack (*arrays*)
Stack arrays in sequence vertically (column wise).

Parameters **arrays** : Sequence of SciDBArrays

The arrays to join. All arrays must have the same shape along all but the first dimension.

Returns **stacked** : SciDBArray

The array formed by stacking the given arrays.

See also:

hstack, *dstack*, *concatenate*

wrap_array (*scidbname*, *persistent=True*)
Create a new SciDBArray object that references an existing SciDB array

Parameters **scidbname** : string

Wrap an existing scidb array referred to by *scidbname*. The SciDB array object persistent value will be set to True, and the object shape, datashape and data type values will be determined by the SciDB array.

persistent : boolean

If True (default) then array will not be deleted when this variable goes out of scope. Warning: if persistent is set to False, data could be lost!

zeros (*shape*, *dtype=u'double'*, ***kwargs*)
Return an array of zeros

Parameters **shape** : tuple or int

The shape of the array

dtype : string or list

The data type of the array

****kwargs :**

Additional keyword arguments are passed to SciDBDataShape.

Returns arr: SciDBArray

A SciDBArray consisting of all zeros.

Visualization and Analysis

class `scidbpy.aggregation.GroupBy` (*array*, *by*, *columns=None*)
Perform a GroupBy operation on an array

The interface of this class mimics a subset of the functionality of Pandas' groupby.

Notes

GroupBy items can be names of attributes or dimensions, or a single-attribute array whose shape matches the input.

For each non-unsigned integer attribute used in a groupby, a new categorical index dimension is created.

Examples

```
>>> x = sdb.afl.build('<a:int32>[i=0:100,1000,0]', 'iif(i > 50, 1, 0)')
>>> y = sdb.afl.build('<b:int32>[i=0:100,1000,0]', 'i % 30')
>>> z = sdb.join(x, y)
>>> grp = z.groupby('a')
>>> grp.aggregate('sum(b)').todataframe()
   a  b_sum
0  0    645
1  1    715
```

Multiple aggregation functions can be provided with a dict:

```
>>> grp.aggregate({'s':'sum(b)', 'm':'max(b)'}).todataframe()
   a    s   m
0  0  645  29
1  1  715  29
```

Methods

aggregate(mappings[, unpack])	Peform an aggregation over each group
approxdc()	Compute the approxdc of all attributes in each group
avg()	Compute the avg of all attributes in each group
count()	Compute the count of all attributes in each group
max()	Compute the max of all attributes in each group
min()	Compute the min of all attributes in each group
stdev()	Compute the stdev of all attributes in each group
sum()	Compute the sum of all attributes in each group
var()	Compute the var of all attributes in each group

aggregate (*mappings*, *unpack=True*)

Peform an aggregation over each group

Parameters **mappings** : string or dictionary

If a string, a single SciDB expression to apply to each group If a dict, mapping several attribute names to expression strings

unpack : bool (optional)

If True (the default), the result will be unpacked into a dense 1D array. If False, the result will be dimensioned by each groupby item.

Returns **agg** : SciDBArray

A new SciDBArray, obtained by applying the aggregations to the groups of the input array.

approxdc()
Compute the approxdc of all attributes in each group

avg()
Compute the avg of all attributes in each group

count()
Compute the count of all attributes in each group

max()
Compute the max of all attributes in each group

min()
Compute the min of all attributes in each group

stdev()
Compute the stdev of all attributes in each group

sum()
Compute the sum of all attributes in each group

var()
Compute the var of all attributes in each group

scidbpy.aggregation.**histogram**(*X*, *bins=10*, *att=None*, *range=None*, *plot=False*, ***kwargs*)
Build a 1D histogram from a SciDBArray.

Parameters **X** : SciDBArray

The array to compute a histogram for

att : str (optional)

The attribute of the array to consider. Defaults to the first attribute.

bins : int (optional)

The number of bins

range : [min, max] (optional)

The lower and upper limits of the histogram. Defaults to data limits.

plot : bool

If True, plot the results with matplotlib

histtype : 'bar' | 'step' (default='bar')

If plotting, the kind of hisogram to draw. See matplotlib.hist for more details.

kwargs : optional

Additional keywords passed to matplotlib

Returns (counts, edges [, artists])

- edges is a NumPy array of edge locations (length=bins+1)
- counts is the number of data betwen [edges[i], edges[i+1]] (length=bins)
- artists is a list of the matplotlib artists created if *plot=True*

1.5.2 AFL Operator Reference

This is a list of functions in SciDB-Py's AFL binding. They are all accessible under the `afl` attribute from the interface object returned by `connect`, eg:

```
from scidbpy import connect
sdb = connect()
afl = sdb.afl
afl.adddim(...)
```

See *Working with the Array Functional Language* for more information.

List of Operators

`scidbpy.`**`project`**`()`
Produces a result array that includes some attributes of the source array.

```
project( srcArray {, selectedAttr}+ )
```

Parameters

- srcArray: the source array with srcAttrs and srcDims.
- a list of at least one selectedAttrs from the source array.

`scidbpy.`**`create_array`**`()`
Creates an array with the given name and schema and adds it to the database.

```
create_array ( array_name, array_schema , temp [, load_array , cells ] )

or

CREATE ['TEMP'] ARRAY array_name  array_schema [ [ [cells] ] USING load_array ]
```

Parameters

- array_name: an identifier that names the new array.
- array_schema: a multidimensional array schema that describes the rank and shape of the array to be created, as well as the types of each its attributes.
- temp: a boolean flag, true for a temporary array, false for a db array.
- load_array an existing database array whose values are to be used to determine sensible choices for those details of the target dimensions that were elided.
- cells the desired number of logical cells per chunk (default is 1M)

`scidbpy.`**`sort`**`()`
Produces a 1D array by sorting the non-empty cells of a source array.

```
sort( srcArray {, attr [asc | desc]}* {, chunkSize}? )
```

Parameters

- srcArray: the source array with srcAttrs and srcDim.
- attr: the list of attributes to sort by. If no attribute is provided, the first attribute will be used.

- asc | desc: whether ascending or descending order of the attribute should be used. The default is asc.
- chunkSize: the size of a chunk in the result array. If not provided, 1M will be used.

Notes Assuming null < NaN < other values

scidbpy.**consume**()

Accesses each cell of an input array, if possible, by extracting tiles and iterating over tiles. numAttrsToScanAtOnce determines the number of attributes to scan as a group. Setting this value to '1' will result in a 'vertical' scan—all chunks of the current attribute will be scanned before moving on to the next attribute. Setting this value to the number of attributes will result in a 'horizontal' scan—chunk i of every attribute will be scanned before moving on to chunk i+1

```
consume( array [, numAttrsToScanAtOnce] )
```

Parameters

- array: the array to consume
- **numAttrsToScanAtOnce: optional 'stride' of the scan, default is 1** Output array (an empty array):

<]

scidbpy.**index_lookup**()

The input_array may have any attributes or dimensions. The index_array must have a single dimension and a single non-nullable attribute. The index array data must be sorted, unique values with no empty cells between them (though it does not necessarily need to be populated to the upper bound). The third argument must correctly refer to one of the attributes of the input array - the looked-up attribute. This attribute must have the same datatype as the only attribute of the index array. The comparison '<' function must be registered in SciDB for this datatype.

The operator will create a new attribute, named input_attribute_name_index by default, or using the provided name, which will be the new last non-empty-tag attribute in the output array. The output attribute will be of type int64 nullable and will contain the respective coordinate of the corresponding input_attribute in index_array. If the corresponding input_attribute is null, or if no value for input_attribute exists in the index_array, the output attribute at that position shall be set to null. The output attribute shall be returned along all the input attributes in a fashion similar to the apply() operator. The operator uses some memory to cache a part of the index_array for fast lookup of values. By default, the size of this cache is limited to MEM_ARRAY_THRESHOLD. Note this is in addition to the memory already consumed by cached MemArrays as the operator is running. If a larger or smaller limit is desired, the 'memory_limit' parameter may be used. It is provided in units of mebibytes and must be at least 1. The operator may be further optimized to reduce memory footprint, optimized with a more clever data distribution pattern and/or extended to use multiple index arrays at the same time.

```
 index_lookup (input_array, index_array,
input_array.attribute_name [,output_attribute_name]
[,'memory_limit=MEMORY_LIMIT'])
```

parameters

input_array <..., input_attribute: type,... > [*] index_array <index_attribute: type not null>

[dimension=0:any,any,any] input_attribute –the name of the input attribute [output_attribute_name] –the name for the output attribute if

desired ['memory_limit=MEMORY_LIMIT'] –the memory limit to use MB)

examples

```
 index_lookup(stock_trades, stock_symbols, stock_trades.ticker)
 index_lookup(stock_trades, stock_symbols, stock_trades.ticker,
ticker_id, 'memory_limit=1024')
```

scidbpy.**setopt**()
Gets/Sets a config option at runtime.

```
setopt( option [, newValue] )
```

Parameters

- option: the config option.
- newValue: an optional new value for the config option. If provided, the option is set. Either way, the option value(s) is returned.

scidbpy.**merge**()
Combines elements from the input arrays the following way: for each cell in the two inputs, if the cell of leftArray is not empty, the attributes from that cell are selected and placed in the output array; otherwise, the attributes from the corresponding cell in rightArray are taken. The two arrays should have the same attribute list, number of dimensions, and dimension start index. If the dimensions are not the same size, the output array uses the larger of the two.

```
merge( leftArray, rightArray )
```

Parameters

- leftArray: the left-hand-side array.
- rightArray: the right-hand-side array.

scidbpy.**store**()
Stores an array to the database. Each execution of store() causes a new version of the array to be created.

```
store( srcArray, outputArray )
```

Parameters

- srcArray: the source array with srcAttrs and srcDim.
- outputArray: an existing array in the database, with the same schema as srcArray.

scidbpy.**subarray**()
Produces a result array from a specified, contiguous region of a source array.

```
subarray( srcArray {, lowCoord}+ {, highCoord}+ )
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- the low coordinates
- the high coordinates

Examples

```
    - Given array A <quantity: uint64, sales:double> [year, item] =
       year, item, quantity, sales
       2011, 2, 7, 31.64
       2011, 3, 6, 19.98
       2012, 1, 5, 41.65
       2012, 2, 9, 40.68
       2012, 3, 8, 26.64
    - subarray(A, 2011, 1, 2012, 2) <quantity: uint64, sales:double>
      [year, item] =
       year, item, quantity, sales
       0, 1, 7, 31.64
       1, 0, 5, 41.65
       1, 1, 9, 40.68
```

Notes

- Almost the same as between(). The only difference is that the dimensions are 'cropped'.

scidbpy.**transpose**()
Produces an array with the same data in srcArray but with the list of dimensions reversd.

```
    transpose( srcArray )
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.

scidbpy.**rank**()
Computes the rankings of an array, based on the ordering of attr (within each group as specified by the list of groupbyDims, if provided). If groupbyDims is not specified, global ordering will be performed. If attr is not specified, the first attribute will be used.

```
    rank( srcArray [, attr {, groupbyDim}*] )
```

Parameters

- srcArray: the source array with srcAttrs and srcDims.
- attr: which attribute to sort on. The default is the first attribute.
- groupbyDim: if provided, the ordering will be performed among the records in the same group.

scidbpy.**avg_rank**()
Ranks the array elements, where each element is ranked as the average of the upper bound (UB) and lower bound (LB) rankings. The LB ranking of an element E is the number of elements less than E, plus 1. The UB ranking of an element E is the number of elements less than or equal to E, plus 1.

```
    avg_rank( srcArray [, attr {, groupbyDim}*] )
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- 0 or 1 attribute to rank with. If no attribute is provided, the first attribute is used.
- an optional list of groupbyDims used to group the elements, such that the rankings are calculated within each group. If no groupbyDim is provided, the whole array is treated as one group.

Examples

```
- Given array A <quantity: uint64, sales:double> [year, item] =
   year, item, quantity, sales
   2011, 2, 7, 31.64
   2011, 3, 6, 19.98
   2012, 1, 5, 41.65
   2012, 2, 9, 40.68
   2012, 3, 8, 26.64
- avg_rank(A, sales, year) <sales:double, sales_rank: uint64>
  [year, item] =
   year, item, sales, sales_rank
   2011, 2, 31.64, 2
   2011, 3, 19.98, 1
   2012, 1, 41.65, 3
   2012, 2, 40.68, 2
   2012, 3, 26.64, 1
```

Notes

- For any element with a distinct value, its UB ranking and LB ranking are equal.

scidbpy.**quantile**()

Computes the quantiles of an array, based on the ordering of attr (within each group as specified by groupbyDim, if specified). If groupbyDim is not specified, global ordering will be performed. If attr is not specified, the first attribute will be used.

```
quantile( srcArray, numQuantiles [, attr {, groupbyDim}*] )
```

Parameters

- srcArray: the source array with srcAttrs and srcDims.
- numQuantiles: the number of quantiles.
- attr: which attribute to sort on. The default is the first attribute.
- groupbyDim: if provided, the ordering will be performed among the records in the same group.

Examples

```
- Given array A <v:int64> [i=0:5,3,0] =
   i, v
   0, 0
   1, 1
   2, 2
   3, 3
   4, 4
   5, 5
- quantile(A, 2) <percentage, v_quantile>[quantile=0:2,3,0] =
   {quantile} percentage, v_quantile
   {0} 0, 0
   {1} 0.5, 2
   {2} 1, 5
```

scidbpy.**list**()

Produces a result array and loads data from a given file, and optionally stores to shadowArray. The available things to list include:

•aggregates: show all the aggregate operators.

•arrays: show all the arrays.

•chunk descriptors: show all the chunk descriptors.

•chunk map: show the chunk map.

•functions: show all the functions.

•instances: show all SciDB instances.

•libraries: show all the libraries that are loaded in the current SciDB session.

•operators: show all the operators and the libraries in which they reside.

•types: show all the datatypes that SciDB supports.

•queries: show all the active queries.

•datastores: show information about each datastore

•counters: (undocumented) dump info from performance counters

```
list( what='arrays', showSystem=false )
```

Parameters

- what: what to list.
- showSystem: whether to show systems information.

scidbpy.**input**()

Produces a result array and loads data from a given file, and optionally stores to shadowArray.

```
input( schemaArray | schema, filename, instance=-2, format='',
   maxErrors=0, shadowArray='', isStrict=false )
```

Parameters

- schemaArray | schema: the array schema.
- filename: where to load data from.
- instance: which instance; default is -2. ??
- format: ??
- maxErrors: ??
- shadowArray: if provided, the result array will be written to it.
- isStrict if true, enables the data integrity checks such as for data collisions and out-of-order input chunks, defualt=false.

Notes

- [comment from author] Must be called as INPUT('existing_array_name', '/path/to/file/on/instance'). ?? schema not allowed??
- This really needs to be modified by the author.

scidbpy.**apply**()

Produces a result array with new attributes and computes values for them.

```
apply(srcArray {, newAttr, expression}+)
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- 1 or more pairs of a new attribute and the expression to compute the values for the attribute.

Examples

```
- Given array A <quantity: uint64, sales:double> [year, item] =
   year, item, quantity, sales
   2011, 2, 7, 31.64
   2011, 3, 6, 19.98
   2012, 1, 5, 41.65
   2012, 2, 9, 40.68
   2012, 3, 8, 26.64
- apply(A, unitprice, sales/quantity) <quantity: uint64, sales:
  double, unitprice: double> [year, item] =
   year, item, quantity, sales, unitprice
   2011, 2, 7, 31.64, 4.52
   2011, 3, 6, 19.98, 3.33
   2012, 1, 5, 41.65, 8.33
   2012, 2, 9, 40.68, 4.52
   2012, 3, 8, 26.64, 3.33
```

scidbpy.**xgrid**()

Produces a result array by 'scaling up' the source array. Within each dimension, the operator duplicates each cell a specified number of times before moving to the next cell. A scale must be provided for every dimension.

```
xgrid( srcArray {, scale}+ )
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- scale: for each dimension, a scale is provided telling how much larger the dimension should grow.

Examples

```
- Given array A <quantity: uint64, sales:double> [year, item] =
   year, item, quantity, sales
   2011, 2, 7, 31.64
   2011, 3, 6, 19.98
   2012, 1, 5, 41.65
   2012, 2, 9, 40.68
   2012, 3, 8, 26.64
- xgrid(A, 1, 2) <quantity: uint64, sales:double> [year, item] =
   year, item, quantity, sales
   2011, 3, 7, 31.64
   2011, 4, 7, 31.64
   2011, 5, 6, 19.98
   2011, 6, 6, 19.98
   2012, 1, 5, 41.65
   2012, 2, 5, 41.65
   2012, 3, 9, 40.68
   2012, 4, 9, 40.68
```

```
        2012, 5, 8, 26.64
        2012, 6, 8, 26.64
```

scidbpy.**filter**()

The filter operator returns an array the with the same schema as the input array. The result is identical to the input except that those cells for which the expression evaluates either false or null are marked as being empty.

```
    filter( srcArray, expression )
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- expression: an expression which takes a cell in the source array as input and evaluates to either True or False.

scidbpy.**cross_between**()

Produces a result array by cutting out data in one of the rectangular ranges specified in rangesArray.

```
    cross_between( srcArray, rangesArray )
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- rangesArray: an array with (|srcDims| * 2) attributes all having type int64.

Examples

```
    - Given array A <quantity: uint64, sales:double> [year, item] =
       year, item, quantity, sales
       2011, 2, 7, 31.64
       2011, 3, 6, 19.98
       2012, 1, 5, 41.65
       2012, 2, 9, 40.68
       2012, 3, 8, 26.64
    - Given array R <year_low, item_low, year_high, item_high>[i] =
       i, year_low, item_low, year_high, item_high
       0, 2011, 3, 2011, 3
       1, 2012, 1, 2012, 2
    - cross_between(A, R) <quantity: uint64, sales:double> [year, item]
      =
       year, item, quantity, sales
       2011, 3, 6, 19.98
       2012, 1, 5, 41.65
       2012, 2, 9, 40.68
```

Notes

- Similar to between().
- The operator only works if the size of the rangesArray is very small.

scidbpy.**between**()

Produces a result array from a specified, contiguous region of a source array.

```
    between( srcArray {, lowCoord}+ {, highCoord}+ )
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- the low coordinates
- the high coordinates

Examples

```
- Given array A <quantity: uint64, sales:double> [year, item] =
   year, item, quantity, sales
   2011, 2, 7, 31.64
   2011, 3, 6, 19.98
   2012, 1, 5, 41.65
   2012, 2, 9, 40.68
   2012, 3, 8, 26.64
- between(A, 2011, 1, 2012, 2) <quantity: uint64, sales:double>
  [year, item] =
   year, item, quantity, sales
   2011, 2, 7, 31.64
   2012, 1, 5, 41.65
   2012, 2, 9, 40.68
```

Notes

- Almost the same as subarray. The only difference is that the dimensions retain the original start/end/boundaries.

scidbpy.**cast**()

Produces a result array with data from srcArray but with the provided schema. There are three primary purposes:

•To change names of attributes or dimensions.

•To change types of attributes

•To change a non-integer dimension to an integer dimension.

•To change a nulls-disallowed attribute to a nulls-allowed attribute.

```
cast( srcArray, schemaArray | schema )
```

Parameters

- srcArray: a source array.
- schemaArray | schema: an array or a schema, from which attrs and dims will be used by the output array.

Examples

```
- Given array A <quantity: uint64, sales:double> [year, item] =
   year, item, quantity, sales
   2011, 2, 7, 31.64
   2011, 3, 6, 19.98
   2012, 1, 5, 41.65
   2012, 2, 9, 40.68
   2012, 3, 8, 26.64
- cast(A, <q:uint64, s:double>[y=2011:2012,2,0, i=1:3,3,0])
  <q:uint64, s:double> [y, i] =
   y, i, q, s
   2011, 2, 7, 31.64
   2011, 3, 6, 19.98
```

```
        2012, 1, 5, 41.65
        2012, 2, 9, 40.68
        2012, 3, 8, 26.64
```

scidbpy.**cancel**()
Cancels a query by ID.

```
    cancel( queryId )
```

Parameters

- queryId: the query ID that can be obtained from the SciDB log or via the list() command.

Notes

- This operator is designed for internal use.

scidbpy.**_diskinfo**()
Checks disk usage.

```
    diskinfo()
```

Notes

- For internal usage.

scidbpy.**slice**()
Produces a 'slice' of the source array, by holding zero or more dimension values constant. The result array does not include the dimensions that are used for slicing.

```
    slice( srcArray {, dim, dimValue}* )
```

Parameters

- srcArray: the source array with srcAttrs and srcDims.
- dim: one of the dimensions to be used for slicing.
- dimValue: the constant value in the dimension to slice.

scidbpy.**_explain_logical**()
Produces a single-element array containing the logical query plan.

```
    explain_logical( query , language = 'aql' )
```

Parameters

- query: a query string.
- language: the language string; either 'aql' or 'afl'; default is 'aql'

Notes

- For internal usage.

scidbpy.**unpack**()
Unpacks a multi-dimensional array into a single-dimensional array, creating new attributes to represent the dimensions in the source array.

```
    unpack( srcArray, newDim )
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- newDim: the name of the dimension in the result 1D array.

scidbpy.**variable_window**()

Produces a result array with the same dimensions as the source array, where each cell stores some aggregates calculated over a 1D window covering the current cell. The window has fixed number of non-empty elements. For instance, when rightEdge is 1, the window extends to the right-hand side however number of coordinatesthat are needed, to cover the next larger non-empty cell.

```
    variable_window( srcArray, dim, leftEdge, rightEdge {,
       AGGREGATE_CALL}+ )
        AGGREGATE_CALL := AGGREGATE_FUNC(inputAttr) [as resultName]
        AGGREGATE_FUNC := approxdc | avg | count | max | min | sum | stdev
       | var | some_use_defined_aggregate_function
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- dim: along which dimension is the window defined.
- leftEdge: how many cells to the left of the current cell are included in the window.
- rightEdge: how many cells to the right of the current cell are included in the window.
- 1 or more aggregate calls. Each aggregate call has an AGGREGATE_FUNC, an inputAttr and a resultName. The default resultName is inputAttr followed by '_' and then AGGREGATE_FUNC.

Examples

```
    - Given array A <quantity: uint64, sales:double> [year, item] =
       year, item, quantity, sales
       2011, 2, 7, 31.64
       2011, 3, 6, 19.98
       2012, 1, 5, 41.65
       2012, 2, 9, 40.68
       2012, 3, 8, 26.64
    - variable_window(A, item, 1, 0, sum(quantity)) <quantity_sum:
      uint64> [year, item] =
       year, item, quantity_sum
       2011, 2, 7
       2011, 3, 13
       2012, 1, 5
       2012, 2, 14
       2012, 3, 17
```

Notes

- For a dense array, this is a special case of window().
- For the aggregate function approxdc(), the attribute name is currently non-conventional. It is xxx_ApproxDC instead of xxx_approxdc. Should change.

scidbpy.**_reduce_distro**()

Makes a replicated array appear as if it has the required partitioningSchema.

```
reduce_distro( replicatedArray, partitioningSchema )
```

Parameters

- replicatedArray: an source array which is replicated across all the instances.
- partitioningSchema: the desired partitioning schema.

scidbpy.**cross_join**()

Calculates the cross product of two arrays, with 0 or more equality conditions on the dimensions. Assume p pairs of equality conditions exist. The result is an (m+n-p) dimensional array. From the coordinates of each cell in the result array, a single cell in leftArray and a single cell in rightArray can be located. The cell in the result array contains the concatenation of the attributes from the two source cells. If a pair of join dimensions have different lengths, the result array uses the smaller of the two.

```
cross_join( leftArray, rightArray {, attrLeft, attrRight}* )
```

Parameters

- leftArray: the left-side source array with leftAttrs and leftDims.
- rightArray: the right-side source array with rightAttrs and rightDims.
- 0 or more pairs of an attribute from leftArray and an attribute from rightArray.

Examples

```
- Given array A <quantity: uint64, sales:double> [year, item] =
   year, item, quantity, sales
   2011, 2, 7, 31.64
   2011, 3, 6, 19.98
   2012, 1, 5, 41.65
   2012, 2, 9, 40.68
   2012, 3, 8, 26.64
- Given array B <v:uint64> [k] =
   k, v
   1, 10
   2, 20
   3, 30
   4, 40
   5, 50
- cross_join(A, B, item, k) <quantity: uint64, sales:double,
  v:uint64> [year, item] =
   year, item, quantity, sales, v
   2011, 2, 7, 31.64, 20
   2011, 3, 6, 19.98, 30
   2012, 1, 5, 41.65, 10
   2012, 2, 9, 40.68, 20
   2012, 3, 8, 26.64, 30
```

Notes

- Joining non-integer dimensions does not work.

scidbpy.**help**()

Produces a single-element array containing the help information for an operator.

```
help( operator )
```

Parameters

- operator: the name of an operator.

scidbpy.**rename**()
Changes the name of an array.

```
rename( oldArray, newArray )
```

Parameters

- oldArray: an existing array.
- newArray: the new name of the array.

scidbpy.**insert**()
Inserts all data from left array into the persistent targetArray. targetArray must exist with matching dimensions and attributes. targetArray must also be mutable. The operator shall create a new version of targetArray that contains all data of the array that would have been received by merge(sourceArray, targetArrayName). In other words, new data is inserted between old data and overwrites any overlapping old values. The resulting array is then returned.

```
insert( sourceArray, targetArrayName )
```

Parameters

- sourceArray the array or query that provides inserted data
- targetArrayName: the name of the persistent array inserted into

Notes Some might wonder - if this returns the same result as merge(sourceArray, targetArrayName), then why not use store(merge())? The answer is that 1. this runs a lot faster - it does not perform a full scan of

targetArray

2. this also generates less chunk headers

scidbpy.**remove_versions**()
Removes all versions of targetArray that are older than oldestVersionToSave

```
remove_versions( targetArray, oldestVersionToSave )
```

Parameters

- targetArray: the array which is targeted.
- oldestVersionToSave: the version, prior to which all versions will be removed.

scidbpy.**remove**()
Drops an array.

```
remove( arrayToRemove )
```

Parameters

- arrayToRemove: the array to drop.

scidbpy.**reshape**()

Produces a result array containing the same cells as, but a different shape from, the source array.

```
reshape( srcArray, schema )
```

Parameters

- srcArray: the source array with srcAttrs and srcDims.
- schema: the desired schema, with the same attributes as srcAttrs, but with different size and/or number of dimensions. The restriction is that the product of the dimension sizes is equal to the number of cells in srcArray.

scidbpy.**repart**()

Produces a result array similar to the source array, but with different chunk sizes, different chunk overlaps, or both.

```
repart( srcArray, schema )
```

Parameters

- srcArray: the source array with srcAttrs and srcDims.
- schema: the desired schema.

scidbpy.**redimension**()

Produces a array using some or all of the variables of a source array, potentially changing some or all of those variables from dimensions to attributes or vice versa, and optionally calculating aggregates to be included in the new array.

```
redimension( srcArray, schemaArray | schema , isStrict=true | {,
   AGGREGATE_CALL}* )
    AGGREGATE_CALL := AGGREGATE_FUNC(inputAttr) [as resultName]
    AGGREGATE_FUNC := approxdc | avg | count | max | min | sum | stdev
   | var | some_use_defined_aggregate_function
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- schemaArray | schema: an array or schema from which outputAttrs and outputDims can be acquired. All the dimensions in outputDims must exist either in srcAttrs or in srcDims, with one exception. One new dimension called the synthetic dimension is allowed. All the attributes in outputAttrs, which is not the result of an aggregate, must exist either in srcAttrs or in srcDims.
- isStrict if true, enables the data integrity checks such as for data collisions and out-of-order input chunks, defualt=false. In case of aggregates, isStrict requires that the aggreates be specified for all source array attributes which are also attributes in the new array. In case of synthetic dimension, isStrict has no effect.
- 0 or more aggregate calls. Each aggregate call has an AGGREGATE_FUNC, an inputAttr and a resultName. The default resultName is inputAttr followed by '_' and then AGGREGATE_FUNC. The resultNames must already exist in outputAttrs.

Notes

- The synthetic dimension cannot co-exist with aggregates. That is, if there exists at least one aggregate call, the synthetic dimension must not exist.

- When multiple values are 'redimensioned' into the same cell in the output array, the collision handling depends on the schema: (a) If there exists a synthetic dimension, all the values are retained in a vector along the synthetic dimension. (b) Otherwise, for an aggregate attribute, the aggregate result of the values is stored. (c) Otherwise, an arbitrary value is picked and the rest are discarded.
- Current redimension() does not support Non-integer dimensions or data larger than memory.

scidbpy.**join**()
Combines the attributes of two arrays at matching dimension values. The two arrays must have the same dimension start coordinates, the same chunk size, and the same chunk overlap. The join result has the same dimension names as the first input. The cell in the result array contains the concatenation of the attributes from the two source cells. If a pair of join dimensions have different lengths, the result array uses the smaller of the two.

```
join( leftArray, rightArray )
```

Parameters

- leftArray: the left-side source array with leftAttrs and leftDims.
- rightArray: the right-side source array with rightAttrs and rightDims.

Notes

- join() is a special case of cross_join() with all pairs of dimensions given.

scidbpy.**unload_library**()
Unloads a SciDB plugin.

```
unload_library( library )
```

Parameters

- library: the name of the library to unload.

Notes

- This operator is the reverse of load_library().

scidbpy.**versions**()
Lists all versions of an array in the database.

```
versions( srcArray )
```

Parameters

- srcArray: a source array.

scidbpy.**save**()
Saves the data in an array to a file.

```
save( srcArray, file, instanceId = -2, format = 'store' )
```

Parameters

- srcArray: the source array to save from.
- file: the file to save to.

- instanceId: positive number means an instance ID on which file will be saved. -1 means to save file on every instance. -2 - on coordinator.
- format: ArrayWriter format in which file will be stored

Notes n/a Must be called as SAVE('existing_array_name', '/path/to/file/on/instance')

scidbpy.**_save_old**()
Saves the data in an array to a file.

```
save( srcArray, file, instanceId = -2, format = 'store' )
```

Parameters

- srcArray: the source array to save from.
- file: the file to save to.
- instanceId: positive number means an instance ID on which file will be saved. -1 means to save file on every instance. -2 - on coordinator.
- format: ArrayWriter format in which file will be stored

Notes n/a Must be called as SAVE('existing_array_name', '/path/to/file/on/instance')

scidbpy.**_sg**()
SCATTER/GATHER distributes array chunks over the instances of a cluster. The result array is returned. It is the only operator that uses the network manager. Typically this operator is inserted by the optimizer into the physical plan.

```
sg( srcArray, partitionSchema, instanceId=-1, outputArray='',
   isStrict=false, offsetVector=null)
```

Parameters

- srcArray: the source array, with srcAttrs and srcDims.
- **partitionSchema:** 0 = psReplication, 1 = psHashPartitioned, 2 = psLocalInstance, 3 = psByRow, 4 = psByCol, 5 = psUndefined.
- **instanceId:** -2 = to coordinator (same with 0), -1 = all instances participate, 0..#instances-1 = to a particular instance. [TO-DO: The usage of instanceId, in calculating which instance a

  chunk should go to, requires further documentation.]
- outputArray: if not empty, the result will be stored into this array
- isStrict if true, enables the data integrity checks such as for data collisions and out-of-order input chunks, defualt=false.
- **offsetVector: a vector of #dimensions values.** To calculate which instance a chunk belongs, the chunkPos is

  augmented with the offset vector before calculation.

scidbpy.**bernoulli**()
Evaluates whether to include a cell in the result array by generating a random number and checks if it is less than probability.

```
bernoulli( srcArray, probability [, seed] )
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- probability: the probability threshold, in [0..1]
- an optional seed for the random number generator.

Examples

```
    - Given array A <quantity: uint64, sales:double> [year, item] =
       year, item, quantity, sales
       2011, 2, 7, 31.64
       2011, 3, 6, 19.98
       2012, 1, 5, 41.65
       2012, 2, 9, 40.68
       2012, 3, 8, 26.64
    - bernoulli(A, 0.5, 100) <quantity: uint64, sales:double> [year,
      item] =
       year, item, quantity, sales
       2011, 3, 6, 19.98
       2012, 1, 5, 41.65
       2012, 3, 8, 26.64
```

scidbpy.**_explain_physical**()
Produces a single-element array containing the physical query plan.

```
    explain_physical( query , language = 'aql' )
```

Parameters

- query: a query string.
- language: the language string; either 'aql' or 'afl'; default is 'aql'

Notes

- For internal usage.

scidbpy.**scan**()
Produces a result array that is equivalent to a stored array.

```
    scan( srcArray [, ifTrim] )
```

Parameters

- srcArray: the array to scan, with srcAttrs and srcDims.
- ifTrim: whether to turn an unbounded array to a bounded array. Default value is false.

scidbpy.**load_library**()
Loads a SciDB plugin.

```
    load_library( library )
```

Parameters

- library: the name of the library to load.

Notes

- A library may be unloaded using unload_library()

scidbpy.**unfold**()

Complicated input data are often loaded into table-like 1-d multi- attribute arrays. Sometimes we want to assemble uniformly-typed subsets of the array attributes into a matrix, for example to compute correlations or regressions. unfold will transform the input array into a 2-d matrix whose columns correspond to the input array attributes. The output matrix row dimension will have a chunk size equal to the input array, and column chunk size equal to the number of columns.

```
unfold( array )
```

Parameters

- array: the array to consume

Examples

```
unfold(apply(build(<v:double>[i=0:9,3,0],i),w,i+0.5))
```

scidbpy.**dimensions**()

List the dimensions of the source array.

```
dimensions( srcArray )
```

Parameters

- srcArray: a source array.

scidbpy.**show**()

Shows the schema of an array.

```
show( schemaArray | schema | queryString [, 'aql' | 'afl'] )
```

Parameters

- schemaArray | schema | queryString: an array where the schema is used, the schema itself or arbitrary query string

o

scidbpy.**substitute**()

Produces a result array the same as srcArray, but with null values (of selected attributes) substituted using the values in substituteArray.

```
substitute( srcArray, substituteArray {, attr}* )
```

Parameters

- srcArray: a source array with srcAttrs and srcDims, that may contain null values.
- substituteArray: the array from which the values may be used to substitute the null values in srcArray. It must have a single dimension which starts at 0, and a single attribute.
- An optional list of attributes to substitute. The default is to substitute all nullable attributes.

scidbpy.**attributes**()

Produces a 1D result array where each cell describes one attribute of the source array.

```
attributes( srcArray )
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.

Examples

```
- Given array A <quantity: uint64, sales:double> [year, item] =
   year, item, quantity, sales
   2011, 2, 7, 31.64
   2011, 3, 6, 19.98
   2012, 1, 5, 41.65
   2012, 2, 9, 40.68
   2012, 2, 8, 26.64
- attributes(A) <name:string, type_id:string, nullable:bool> [No] =
   No, name, type_id, nullable
   0, 'quantity', 'uint64', false
   1, 'sales', 'double', false
```

scidbpy.**window**()

Produces a result array with the same size and dimensions as the source array, where each ouput cell stores some aggregate calculated over a window around the corresponding cell in the source array. A pair of window specification values (leftEdge, rightEdge) must exist for every dimension in the source and output array.

```
window( srcArray {, leftEdge, rightEdge}+ {, AGGREGATE_CALL}+ [,
   METHOD ] )
    AGGREGATE_CALL := AGGREGATE_FUNC(inputAttr) [as resultName]
    AGGREGATE_FUNC := approxdc | avg | count | max | min | sum | stdev
   | var | some_use_defined_aggregate_function
    METHOD := 'materialize' | 'probe'
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- leftEdge: how many cells to the left of the current cell (in one dimension) are included in the window.
- rightEdge: how many cells to the right of the current cell (in one dimension) are included in the window.
- 1 or more aggregate calls. Each aggregate call has an AGGREGATE_FUNC, an inputAttr and a resultName. The default resultName is inputAttr followed by '_' and then AGGREGATE_FUNC. For instance, the default resultName for sum(sales) is 'sales_sum'. The count aggregate may take * as the input attribute, meaning to count all the items in the group including null items. The default resultName for count(*) is 'count'.
- An optional final argument that specifies how the operator is to perform its calculation. At the moment, we support two internal algorithms: 'materialize' (which materializes an entire source chunk before computing the output windows) and 'probe' (which probes the source array for the data in each window). In general, materializing the input is a more efficient strategy, but when we're using thin(...) in conjunction with window(...), we're often better off using probes, rather than materilization. This is a decision that the optimizer needs to make.

Examples

```
- Given array A <quantity: uint64, sales:double> [year, item] =
   year, item, quantity, sales
   2011, 2, 7, 31.64
```

```
        2011, 3, 6, 19.98
        2012, 1, 5, 41.65
        2012, 2, 9, 40.68
        2012, 3, 8, 26.64
    - window(A, 0, 0, 1, 0, sum(quantity)) <quantity_sum: uint64>
      [year, item] =
        year, item, quantity_sum
        2011, 2, 7
        2011, 3, 13
        2012, 1, 5
        2012, 2, 14
        2012, 3, 17
```

scidbpy.**regrid**()

Partitions the cells in the source array into blocks (with the given blockSize in each dimension), and for each block, calculates the required aggregates.

```
    regrid( srcArray {, blockSize}+ {, AGGREGATE_CALL}+ {, chunkSize}*
       )
        AGGREGATE_CALL := AGGREGATE_FUNC(inputAttr) [as resultName]
        AGGREGATE_FUNC := approxdc | avg | count | max | min | sum | stdev
       | var | some_use_defined_aggregate_function
```

Parameters

- srcArray: the source array with srcAttrs and srcDims.
- A list of blockSizes, one for each dimension.
- 1 or more aggregate calls. Each aggregate call has an AGGREGATE_FUNC, an inputAttr and a resultName. The default resultName is inputAttr followed by '_' and then AGGREGATE_FUNC. For instance, the default resultName for sum(sales) is 'sales_sum'. The count aggregate may take * as the input attribute, meaning to count all the items in the group including null items. The default resultName for count(*) is 'count'.
- 0 or numDims chunk sizes. If no chunk size is given, the chunk sizes from the input dims will be used. If at least one chunk size is given, the number of chunk sizes must be equal to the number of dimensions, and the specified chunk sizes will be used.

Notes

- Regrid does not allow a block to span chunks. So for every dimension, the chunk interval needs to be a multiple of the block size.

scidbpy.**aggregate**()

Calculates aggregates over groups of values in an array, given the aggregate types and attributes to aggregate on.

```
    aggregate( srcArray {, AGGREGATE_CALL}+ {, groupbyDim}* {,
       chunkSize}* )
        AGGREGATE_CALL := AGGREGATE_FUNC(inputAttr) [as resultName]
        AGGREGATE_FUNC := approxdc | avg | count | max | min | sum | stdev
       | var | some_use_defined_aggregate_function
```

Parameters

- srcArray: a source array with srcAttrs and srcDims.
- 1 or more aggregate calls. Each aggregate call has an AGGREGATE_FUNC, an inputAttr and a resultName. The default resultName is inputAttr followed by '_' and then AGGREGATE_FUNC. For instance, the default resultName for sum(sales) is 'sales_sum'. The

count aggregate may take * as the input attribute, meaning to count all the items in the group including null items. The default resultName for count(*) is 'count'.

- 0 or more dimensions that together determines the grouping criteria.
- 0 or numGroupbyDims chunk sizes. If no chunk size is given, the groupby dims will inherit chunk sizes from the input array. If at least one chunk size is given, the number of chunk sizes must be equal to the number of groupby dimensions, and the groupby dimensions will use the specified chunk sizes.

Examples

```
- Given array A <quantity: uint64, sales:double> [year, item] =
   year, item, quantity, sales
   2011, 2, 7, 31.64
   2011, 3, 6, 19.98
   2012, 1, 5, 41.65
   2012, 2, 9, 40.68
   2012, 3, 8, 26.64
- aggregate(A, count(*), max(quantity), sum(sales), year) <count:
  uint64, quantity_max: uint64, sales_sum: double> [year] =
   year, count, quantity_max, sales_sum
   2011, 2, 7, 51.62
   2012, 3, 9, 108.97
```

Notes

- All the aggregate functions ignore null values, except count(*).

scidbpy.**cumulate**()

Calculates a running aggregate over some aggregate along some fluxVector (a single dimension of the inputArray).

```
cumulate ( inputArray {, AGGREGATE_ALL}+ [, aggrDim] )
    AGGREGATE_CALL := AGGREGATE_FUNC ( inputAttribute ) [ AS aliasName
   ]
    AGGREGATE_FUNC := approxdc | avg | count | max | min | sum | stdev
   | var | some_use_defined_aggregate_function
```

Parameters

- inputArray: an input array
- 1 or more aggregate calls.
- aggrDim: the name of a dimension along with aggregates are computed. Default is the first dimension.

Examples

```
  input:          cumulate(input, sum(v) as sum_v, count(*) as cnt, I)
 +-I->
J|     00   01   02   03       00       01       02       03
 V   +----+----+----+----+        +--------+--------+--------+--------+
 00  | 01 |    | 02 |    |   00   | (1, 1) |        | (3, 2) |        |
     +----+----+----+----+        +--------+--------+--------+--------+
 01  |    | 03 |    | 04 |   01   |        | (3, 1) |        | (7, 2) |
     +----+----+----+----+        +--------+--------+--------+--------+
 02  | 05 |    | 06 |    |   02   | (5, 1) |        | (11, 2)|        |
```

```
          +----+----+----+----+        +--------+--------+--------+--------+
     03   |    | 07 |    | 08 |   03   |        | (7, 1) |        | (15, 2)|
          +----+----+----+----+        +--------+--------+--------+--------+
```

Notes

- For now, cumulate does NOT handle input array that have overlaps.

scidbpy.**uniq**()

The input array must have a single attribute of any type and a single dimension. The data in the input array must be sorted and dense. The operator is built to accept the output produced by sort() with a single attribute. The output array shall have the same attribute with the dimension i starting at 0 and chunk size of 1 million. An optional chunk_size parameter may be used to set a different output chunk size. Data is compared using a simple bitwise comparison of underlying memory. Null values are discarded from the output.

```
    uniq (input_array [,'chunk_size=CHUNK_SIZE'] )
```

Parameters

array <single_attribute: INPUT_ATTRIBUTE_TYPE> [single_dimension=*]

Examples

```
     uniq (sorted_array)
     store ( uniq ( sort ( project (big_array, string_attribute) ),
    'chuk_size=100000'), string_attribute_index )
```

scidbpy.**_materialize**()

Produces a materialized version of an source array.

```
    materialize( srcArray, format )
```

Parameters

- srcArray: the sourcce array with srcDims and srcAttrs.
- format: uint32, the materialize format.

1.5.3 Robust AFL Operators

The functions in this module closely correspond to AFL calls, but they preprocess arrays as needed to satisfy any requirements that SciDB imposes on the schemas of arrays used as AFL arguments.

As an example, the AFL merge function requires that its two input arrays have the same attribute list, number of dimensions, and dimension start index. The merge() function performs this preprocessing as needed.

scidbpy.robust.**join**(*a*, *b*)

Robust AFL join operation

Parameters **a** : SciDBArray

Left array

b : SciDBArray

Right array

Returns **result** : SciDBArray

join(a, b)

Notes

Performs any of the following steps if needed:

- Broadcast A and B to equal shapes
- Align array origins
- Match chunk sizes and overlaps

scidbpy.robust.**merge**(*a*, *b*)

Robust AFL merge operation

Parameters **a** : SciDBArray

Left array

b : SciDBArray

Right array

Returns **result** : SciDBArray

merge(a, b)

Notes

Performs any of the following steps if needed:

- Broadcast A and B to equal shapes
- Match attribute names, if unambiguous
- Align array origins

scidbpy.robust.**gemm**(*a*, *b*, *c*)

Robust AFL gemm operation

Performs a * b + c

Redimensions inputs if necessary

Parameters **a** : SciDBArray

First array

b : SciDBArray

Second array

c : SciDBArray

Third array

Returns **result** : SciDBArray

a * b + c

scidbpy.robust.**cumulate**(*array*, **args*)

Robust AFL cumulate call

Parameters **array** : Array to cumulate

***args: Additional arguments to pass to AFL cumulate()**

Returns cumulate(array, *args)

Notes

Re-chunks array to have no chunk overlap, if needed

scidbpy.robust.**reshape**(*array*, **args*)
Robust AFL reshape

scidbpy.robust.**gesvd**(*array*, **args*)
Robust AFL svd call

Parameters **array** : SciDBArray

***args** : Subsequent arguments to SVD AFL call

Notes

Rechunks array if needed by AFL

scidbpy.robust.**thin**(*array*, **args*)
Robust AFL thin call

Parameters **array** : SciDBArray

The array to thin

args: sequence of ints

sequence of start, step for each dimension

Notes

The array is redimensioned if necessary, so that the chunk size is a multiple of the thin steps

scidbpy.robust.**cross_join**(*a*, *b*, **dims*)
Robust AFL cross_join

Parameters **a** : SciDBArray

The left array in the join

b : SciDBArray

The right array in the join

***dims** : Pairs of dimension names

The dimensions to join along

Notes

Arrays will be rechunked as needed for the cross join to run

scidbpy.robust.**uniq**(*a*, *is_sorted=False*)
Robust AFL uniq operator

Parameters **a** : SciDBArray

Array to compute unique elements of. Must have a single attribute

is_sorted: bool

Whether the array is pre_sorted

Returns **u** : SciDBArray

The unique elements of A

Notes

Will flatten and/or sort the input as necessar

1.5.4 Schema Manipulation Utilities

The *scidbpy.schema_utils* module contains functions useful for manipulating array schemas. Many native SciDB functions require that the schemas of input arrays obey certain properties, like having identical chunk sizes. The routines in this module help to preprocess arrays to satisfy these requirements.

The functions in this module are designed to return their inputs unchanged, if no modification is necessary. This saves you from having to pre-check whether a given preprocessing step is necessary.

See *Robust AFL Operators* for a collection of SciDB-Py analogs to AFL functions, which perform necessary array preprocessing automatically.

Functions

scidbpy.schema_utils.**as_column_vector**(*array*)
Convert a 1D array into a 2D array with a single column

scidbpy.schema_utils.**as_row_vector**(*array*)
Convert a 1D array into a 2D array with a single row

scidbpy.schema_utils.**as_same_dimension**(**arrays*)
Coerce arrays into the same shape if possible, or raise a ValueError

Parameters ***arrays**

One or more arrays to test

Returns **new_arrays** : tuple of SciDBArrays

Raises **ValueError**

if arrays have mismatched dimensions, and cannot be coerced into the same shape.

Notes

Currently this function only checkes for mismatched dimensions, it is unable to fix them.

scidbpy.schema_utils.**assert_schema**(*arrays*, *zero_indexed=False*, *bounded=False*, *same_attributes=False*, *same_dimension=False*)
Check that a set of arrays obeys a set of criteria on their schemas.

Parameters **arrays** : tuple of SciDBArrays

The arrays to check

zero_indexed : boolean, optional (default False)

If True, check at all arrays have origins at 0

bounded : boolean, optional (default False)

If True, check that all arrays are bounded (ie don't have * in the dimension schema)

same_attributes : boolean, optional (default False)

If True, check that all arrays have identical attribute names, order, datatypes, and nullability

same_dimension : boolean, optional (default True)

If True, check that all arrays have the same dimensionality

Returns **arrays** : tuple of SciDBArrays

The input

Raises **ValueError** : If any test fails

scidbpy.schema_utils.**assert_single_attribute**(*array*)

Raise a ValueError if an array has multiple attributes

Parameters **array** : SciDBArray

The array to test

Returns **array** : SciDBArray

The input array

Raises **ValueError**

if array has multiple attributes

scidbpy.schema_utils.**boundify**(*array*, *trim=False*)

Redimension an array as needed so that no dimension is unbound (ie ending with *)

Parameters **array** : SciDBArray

The array to bound

Returns **array** : SciDBArray

A (possibly redimensioned) version of array

Notes

This forces evaluation of lazy arrays

scidbpy.schema_utils.**cast_to_integer**(*array*, *attributes*)

Cast a set of attributes in an array to integer datatypes.

This is a useful preprocessing step before redimensioning attributes as dimensions

scidbpy.schema_utils.**change_axis_schema**(*datashape*, *axis*, *start=None*, *stop=None*, *chunk=None*, *overlap=None*, *name=None*)

Create a new DataShape by modifying the parameters of one axis

Parameters **datashape** : SciDBDataShape

The template data shape

axis : int

Which axis to modify

stop : int (optional)

New axis upper bound

chunk : int (optional)

New chunk size

overlap : int (optional)

New chunk overlap

name : str (optional)

New dimension name

Returns **new_schema** : SciDBDataShape

The new schema, obtained by overriding the input parameters of the template datashape along the specified axis

scidbpy.schema_utils.**coerced_shape**(*array*)

Return an array shape, even if the array is unbound.

If the array is unbound, the shape is guaranteed to contain the data

Parameters **array** : SciDBArray

The array to lookup the shape for

Returns **shape** : tuple of ints

The shape

scidbpy.schema_utils.**disambiguate**(**arrays*)

Process a list of arrays with calls to cast as needed, to avoid any name collisions in dimensions or attributes

The first array is guaranteed *not* to be modified

Parameters ***arrays**

One or more arrays to process

Returns **arrays** : tuple of SciDBArrays

The (possibly recasted) inputs. None of the dimensions or attribute names match.

scidbpy.schema_utils.**expand**(**arrays*)

Grow arrays to equal shape, without truncating any data

Parameters ***arrays**

One or more SciDBArrays

Returns **arrays** : tuple of SciDBArrays

The input arrays, redimensioned as needed so they all have the same domain.

scidbpy.schema_utils.**left_dimension_pad**(*array*, *n*)

Add dummy dimensions as needed to an array, so that it is at least n-dimensional.

Parameters **array** : SciDBArray

The array to pad

n : int

The minimum dimensionality of the output

Returns **array** : SciDBArray

A version of the input, with extra dimensions added before the old dimensions.

scidbpy.schema_utils.**limits**(*array*, *names*)

Compute the lower/upper bounds for a set of attributes

Parameters **array** : SciDBArray

The array to consider

names : list of strings

Names of attributes to consider

Returns **limits** : dict mapping name->(lo, hi)

Contains the minimum and maximum value for each attribute

Notes

This performs a full scan of the array

scidbpy.schema_utils.**match_attribute_names**(**arrays*)

Rename attributes in a set of arrays, so that all arrays have the same names of attributes

Parameters ***arrays**

one or more SciDBArrays

Returns **arrays** : tuple of SciDBArrays

All output arrays have the same attribute names

Raises **ValueError** : if arrays aren't conformable

Notes

An array's attributes will be renamed to match an attribute name in the first array, if the association is unambiguous. For example, consider two arrays with attribute schemas `<a:int32, b:float>` and `<a:int32, c:float>`. The attribute `c` will be renamed to `b`, since the datatypes match and there is no other `b` attribute.

scidbpy.schema_utils.**match_chunk_permuted**(*src*, *target*, *indices*, *match_bounds=False*)

Match chunks along a set of dimension pairs.

Parameters **src** : SciDBArray

The array to modify

target: SciDBArray

The array to match

indices: A list of tuples

Each tuple (i,j) indicates that dimension *j* of src should have the same chunk properties as dimension *i* of target

match_bounds : bool (optional, default False)

If true, match the dimension boundaries as well

Returns **new_src, new_target** : tuple of SciDBArrays

A (possibly redimensioned) version of the inputs

scidbpy.schema_utils.**match_chunks**(**arrays*)
Redimension arrays so they have identical chunk sizes and overlaps

It is assumed that all input arrays have the same dimensionality. If needed, use *as_same_dimension()* to ensure this.

Parameters ***arrays**

One or more arrays to match

Returns **arrays** : Tuple of SciDBArrays

The chunk sizes and overlaps will be matched to the first input.

See also:

match_chunk_permuted to match chunks along particular pairs of dimensions

scidbpy.schema_utils.**match_dimensions**(*A*, *B*, *dims*)
Match the dimension bounds along a list of dimensions in 2 arrays.

Parameters **A** : SciDBArray

First array

B : SciDBArray

Second array

dims : list of pairs of integers

For each (i,j) pair, indicates that A[i] should have same dimension boundaries as B[j]

Returns **Anew, Bnew** : SciDBArrays

(Possibly redimensioned) versions of A and B

scidbpy.schema_utils.**match_size**(**arrays*)
Resize all arrays in a list to the size of the first array. This requires that all arrays span a subset of the first array's domain.

Parameters ***arrays**

One or more SciDBArrays

Returns **arrays** : tuple of SciDBArrays

The (possibly redimensioned) inputs. All arrays are resized to match the first array

Raises **ValueError** : If any arrays have a domain that is not a subset

of the first array's domain.

scidbpy.schema_utils.**rechunk**(*array*, *chunk_size=None*, *chunk_overlap=None*)
Change the chunk size and/or overlap

Parameters **array** : SciDBArray

The array to sanitize

chunk_size : int or list of ints (optional)

The new chunk_size. Defaults to old chunk_size

chunk_overlap : int or list of ints (optional)

The new chunk overlap. Defaults to old chunk overlap

Returns **array** : SciDBArray

A (possibly redimensioned) version of the input

scidbpy.schema_utils.**redimension**(*array*, *dimensions*, *attributes*, *dim_boundaries=None*)

Redimension an array as needed, swapping and dropping attributes as needed.

Parameters **array: SciDBArray**

The array to redimension

dimensions : list of strings

The dimensions or attributes in array that should be dimensions

attributes : list of strings

The dimensions or attributes in array that should be attributes

dim_boundaries : dict (optional)

A dictionary mapping dimension names to boundary tuples (lo, hi) Specifies the dimension bounds for attributes promoted to dimensions. If not provided, will default to (0,*). WARNING: this will fail if promiting negatively-valued attributes to dimensions.

Returns **result** : SciDBArray

A new version of array, redimensioned as needed to ensure proper dimension/attribute schema.

Notes

- Only integer attributes can be listed as dimensions
- If an attribute or dimension in the original array is not explicitly provided as an input, it is dropped
- If no attributes are specified, a new dummy attribute is added to ensure a valid schema.

scidbpy.schema_utils.**right_dimension_pad**(*array*, *n*)

Add dummy dimensions as needed to an array, so that it is at least n-dimensional.

Parameters **array** : SciDBArray

The array to pad

n : int

The minimum dimensionality of the output

Returns **array** : SciDBArray

A version of the input, with extra dimensions added after the old dimensions.

scidbpy.schema_utils.**to_attributes**(*array*, **dimensions*)

Ensure that a set of attributes or dimensions are attributes

Parameters **array** : SciDBArray

The array to promote

dimensions : one or more strings

Dimension names to demote. Attribute labels are ignored

Returns **demoted** : SciDBArray

A new array

scidbpy.schema_utils.**to_dimensions**(*array*, **attributes*)
Ensure that a set of attributes or dimensions are dimensions

Parameters **array** : SciDBArray

The array to promote

attributes : one or more strings

Attribute names to promote. Dimension labels are ignored

Returns **promoted** : SciDBArray

A new array

scidbpy.schema_utils.**zero_indexed**(*array*)
Redimension an array so all lower coordinates are at 0

Raises **ValueError** : if any array has dimensions starting below zero.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

S

scidbpy, 25
scidbpy.robust, 84
scidbpy.schema_utils, 87

Symbols

_diskinfo() (in module scidbpy), 72
_explain_logical() (in module scidbpy), 72
_explain_physical() (in module scidbpy), 79
_materialize() (in module scidbpy), 84
_reduce_distro() (in module scidbpy), 73
_save_old() (in module scidbpy), 78
_sg() (in module scidbpy), 78

A

acos() (scidbpy.interface.SciDBInterface method), 38
acos() (scidbpy.interface.SciDBShimInterface method), 50
afl (scidbpy.SciDBArray attribute), 27
aggregate() (in module scidbpy), 82
aggregate() (scidbpy.aggregation.GroupBy method), 61
aggregate() (scidbpy.SciDBArray method), 27
alias() (scidbpy.SciDBArray method), 27
all() (scidbpy.SciDBArray method), 27
any() (scidbpy.SciDBArray method), 27
apply() (in module scidbpy), 68
approxdc() (scidbpy.aggregation.GroupBy method), 61
approxdc() (scidbpy.interface.SciDBInterface method), 39
approxdc() (scidbpy.interface.SciDBShimInterface method), 50
approxdc() (scidbpy.SciDBArray method), 27
arange() (scidbpy.interface.SciDBInterface method), 39
arange() (scidbpy.interface.SciDBShimInterface method), 50
as_column_vector() (in module scidbpy.schema_utils), 87
as_row_vector() (in module scidbpy.schema_utils), 87
as_same_dimension() (in module scidbpy.schema_utils), 87
as_temp() (scidbpy.SciDBArray method), 28
asin() (scidbpy.interface.SciDBInterface method), 39
asin() (scidbpy.interface.SciDBShimInterface method), 51
assert_schema() (in module scidbpy.schema_utils), 87
assert_single_attribute() (in module scidbpy.schema_utils), 88
atan() (scidbpy.interface.SciDBInterface method), 39
atan() (scidbpy.interface.SciDBShimInterface method), 51
att() (scidbpy.SciDBArray method), 28
attribute() (scidbpy.SciDBArray method), 28
attribute_rename() (scidbpy.SciDBArray method), 28
attributes() (in module scidbpy), 80
avg() (scidbpy.aggregation.GroupBy method), 62
avg() (scidbpy.interface.SciDBInterface method), 39
avg() (scidbpy.interface.SciDBShimInterface method), 51
avg() (scidbpy.SciDBArray method), 28
avg_rank() (in module scidbpy), 66

B

bernoulli() (in module scidbpy), 78
between() (in module scidbpy), 70
boundify() (in module scidbpy.schema_utils), 88

C

cancel() (in module scidbpy), 72
cast() (in module scidbpy), 71
cast_to_integer() (in module scidbpy.schema_utils), 88
ceil() (scidbpy.interface.SciDBInterface method), 39
ceil() (scidbpy.interface.SciDBShimInterface method), 51
change_axis_schema() (in module scidbpy.schema_utils), 88
coerced_shape() (in module scidbpy.schema_utils), 89
collapse() (scidbpy.SciDBArray method), 28
compress() (scidbpy.SciDBArray method), 28
concatenate() (scidbpy.interface.SciDBInterface method), 39
concatenate() (scidbpy.interface.SciDBShimInterface method), 51
connect() (in module scidbpy.interface), 37
consume() (in module scidbpy), 64
contains_nulls() (scidbpy.SciDBArray method), 29
contents() (scidbpy.SciDBArray method), 29
copy() (scidbpy.SciDBArray method), 29

cos() (scidbpy.interface.SciDBInterface method), 40
cos() (scidbpy.interface.SciDBShimInterface method), 52
count() (scidbpy.aggregation.GroupBy method), 62
count() (scidbpy.interface.SciDBInterface method), 40
count() (scidbpy.interface.SciDBShimInterface method), 52
count() (scidbpy.SciDBArray method), 29
create_array() (in module scidbpy), 63
cross_between() (in module scidbpy), 70
cross_join() (in module scidbpy), 74
cross_join() (in module scidbpy.robust), 86
cross_join() (scidbpy.interface.SciDBInterface method), 40
cross_join() (scidbpy.interface.SciDBShimInterface method), 52
cumprod() (scidbpy.SciDBArray method), 29
cumsum() (scidbpy.SciDBArray method), 29
cumulate() (in module scidbpy), 83
cumulate() (in module scidbpy.robust), 85
cumulate() (scidbpy.SciDBArray method), 30

D

default_compression (scidbpy.interface.SciDBInterface attribute), 40
default_compression (scidbpy.interface.SciDBShimInterface attribute), 52
dimension() (scidbpy.SciDBArray method), 30
dimension_rename() (scidbpy.SciDBArray method), 30
dimensions() (in module scidbpy), 80
disambiguate() (in module scidbpy.schema_utils), 89
dot() (scidbpy.interface.SciDBInterface method), 40
dot() (scidbpy.interface.SciDBShimInterface method), 52
dstack() (scidbpy.interface.SciDBInterface method), 40
dstack() (scidbpy.interface.SciDBShimInterface method), 52

E

eval() (scidbpy.SciDBArray method), 30
exp() (scidbpy.interface.SciDBInterface method), 41
exp() (scidbpy.interface.SciDBShimInterface method), 52
expand() (in module scidbpy.schema_utils), 89

F

filter() (in module scidbpy), 70
floor() (scidbpy.interface.SciDBInterface method), 41
floor() (scidbpy.interface.SciDBShimInterface method), 52
from_array() (scidbpy.interface.SciDBInterface method), 41
from_array() (scidbpy.interface.SciDBShimInterface method), 52
from_dataframe() (scidbpy.interface.SciDBInterface method), 41
from_dataframe() (scidbpy.interface.SciDBShimInterface method), 53
from_query() (scidbpy.SciDBArray class method), 30
from_sparse() (scidbpy.interface.SciDBInterface method), 41
from_sparse() (scidbpy.interface.SciDBShimInterface method), 53

G

gemm() (in module scidbpy.robust), 85
gesvd() (in module scidbpy.robust), 86
GroupBy (class in scidbpy.aggregation), 60
groupby() (scidbpy.SciDBArray method), 31

H

head() (scidbpy.SciDBArray method), 31
help() (in module scidbpy), 74
histogram() (in module scidbpy.aggregation), 62
hstack() (scidbpy.interface.SciDBInterface method), 42
hstack() (scidbpy.interface.SciDBShimInterface method), 53

I

identity() (scidbpy.interface.SciDBInterface method), 42
identity() (scidbpy.interface.SciDBShimInterface method), 54
index_lookup() (in module scidbpy), 64
index_lookup() (scidbpy.SciDBArray method), 31
input() (in module scidbpy), 68
insert() (in module scidbpy), 75
isel() (scidbpy.SciDBArray method), 31
isnan() (scidbpy.interface.SciDBInterface method), 42
isnan() (scidbpy.interface.SciDBShimInterface method), 54
issparse() (scidbpy.SciDBArray method), 32

J

join() (in module scidbpy), 77
join() (in module scidbpy.robust), 84
join() (scidbpy.interface.SciDBInterface method), 42
join() (scidbpy.interface.SciDBShimInterface method), 54

L

left_dimension_pad() (in module scidbpy.schema_utils), 89
limits() (in module scidbpy.schema_utils), 90
linspace() (scidbpy.interface.SciDBInterface method), 42
linspace() (scidbpy.interface.SciDBShimInterface method), 54
list() (in module scidbpy), 67
list_arrays() (scidbpy.interface.SciDBInterface method), 43

list_arrays() (scidbpy.interface.SciDBShimInterface method), 55
load_library() (in module scidbpy), 79
log() (scidbpy.interface.SciDBInterface method), 43
log() (scidbpy.interface.SciDBShimInterface method), 55
log10() (scidbpy.interface.SciDBInterface method), 43
log10() (scidbpy.interface.SciDBShimInterface method), 55
login() (scidbpy.interface.SciDBShimInterface method), 55
logout() (scidbpy.interface.SciDBShimInterface method), 55
ls() (scidbpy.interface.SciDBInterface method), 43
ls() (scidbpy.interface.SciDBShimInterface method), 55

M

match_attribute_names() (in module scidbpy.schema_utils), 90
match_chunk_permuted() (in module scidbpy.schema_utils), 90
match_chunks() (in module scidbpy.schema_utils), 91
match_dimensions() (in module scidbpy.schema_utils), 91
match_size() (in module scidbpy.schema_utils), 91
max() (scidbpy.aggregation.GroupBy method), 62
max() (scidbpy.interface.SciDBInterface method), 43
max() (scidbpy.interface.SciDBShimInterface method), 55
max() (scidbpy.SciDBArray method), 32
mean() (scidbpy.interface.SciDBInterface method), 43
mean() (scidbpy.interface.SciDBShimInterface method), 55
mean() (scidbpy.SciDBArray method), 32
merge() (in module scidbpy), 65
merge() (in module scidbpy.robust), 85
merge() (scidbpy.interface.SciDBInterface static method), 43
merge() (scidbpy.interface.SciDBShimInterface method), 55
min() (scidbpy.aggregation.GroupBy method), 62
min() (scidbpy.interface.SciDBInterface method), 44
min() (scidbpy.interface.SciDBShimInterface method), 56
min() (scidbpy.SciDBArray method), 32

N

new_array() (scidbpy.interface.SciDBInterface method), 44
new_array() (scidbpy.interface.SciDBShimInterface method), 56
nonempty() (scidbpy.SciDBArray method), 32
nonnull() (scidbpy.SciDBArray method), 33

O

ones() (scidbpy.interface.SciDBInterface method), 45
ones() (scidbpy.interface.SciDBShimInterface method), 57

P

percentile() (scidbpy.interface.SciDBInterface method), 45
percentile() (scidbpy.interface.SciDBShimInterface method), 57
persistent (scidbpy.SciDBArray attribute), 33
project() (in module scidbpy), 63

Q

quantile() (in module scidbpy), 67
query() (scidbpy.interface.SciDBInterface method), 45
query() (scidbpy.interface.SciDBShimInterface method), 57

R

randint() (scidbpy.interface.SciDBInterface method), 45
randint() (scidbpy.interface.SciDBShimInterface method), 57
random() (scidbpy.interface.SciDBInterface method), 46
random() (scidbpy.interface.SciDBShimInterface method), 58
rank() (in module scidbpy), 66
reap() (scidbpy.interface.SciDBInterface method), 46
reap() (scidbpy.interface.SciDBShimInterface method), 58
reap() (scidbpy.SciDBArray method), 33
rechunk() (in module scidbpy.schema_utils), 91
redimension() (in module scidbpy), 76
redimension() (in module scidbpy.schema_utils), 92
regrid() (in module scidbpy), 82
regrid() (scidbpy.SciDBArray method), 33
relabel() (scidbpy.SciDBArray method), 33
remove() (in module scidbpy), 75
remove() (scidbpy.interface.SciDBInterface method), 46
remove() (scidbpy.interface.SciDBShimInterface method), 58
remove_versions() (in module scidbpy), 75
rename() (in module scidbpy), 75
rename() (scidbpy.SciDBArray method), 33
repart() (in module scidbpy), 76
reshape() (in module scidbpy), 76
reshape() (in module scidbpy.robust), 86
reshape() (scidbpy.SciDBArray method), 34
right_dimension_pad() (in module scidbpy.schema_utils), 92

S

save() (in module scidbpy), 77

scan() (in module scidbpy), 79
schema (scidbpy.SciDBArray attribute), 34
SciDBArray (class in scidbpy), 25
SciDBInterface (class in scidbpy.interface), 37
scidbpy (module), 25
scidbpy.robust (module), 84
scidbpy.schema_utils (module), 87
SciDBShimInterface (class in scidbpy.interface), 48
setopt() (in module scidbpy), 65
show() (in module scidbpy), 80
sin() (scidbpy.interface.SciDBInterface method), 47
sin() (scidbpy.interface.SciDBShimInterface method), 58
slice() (in module scidbpy), 72
sort() (in module scidbpy), 63
sqrt() (scidbpy.interface.SciDBInterface method), 47
sqrt() (scidbpy.interface.SciDBShimInterface method), 58
std() (scidbpy.interface.SciDBInterface method), 47
std() (scidbpy.interface.SciDBShimInterface method), 58
std() (scidbpy.SciDBArray method), 34
stdev() (scidbpy.aggregation.GroupBy method), 62
stdev() (scidbpy.interface.SciDBInterface method), 47
stdev() (scidbpy.interface.SciDBShimInterface method), 59
stdev() (scidbpy.SciDBArray method), 34
store() (in module scidbpy), 65
subarray() (in module scidbpy), 65
substitute() (in module scidbpy), 80
substitute() (scidbpy.interface.SciDBInterface method), 47
substitute() (scidbpy.interface.SciDBShimInterface method), 59
substitute() (scidbpy.SciDBArray method), 34
sum() (scidbpy.aggregation.GroupBy method), 62
sum() (scidbpy.interface.SciDBInterface method), 47
sum() (scidbpy.interface.SciDBShimInterface method), 59
sum() (scidbpy.SciDBArray method), 35
svd() (scidbpy.interface.SciDBInterface method), 47
svd() (scidbpy.interface.SciDBShimInterface method), 59

T

T (scidbpy.SciDBArray attribute), 27
tan() (scidbpy.interface.SciDBInterface method), 47
tan() (scidbpy.interface.SciDBShimInterface method), 59
thin() (in module scidbpy.robust), 86
to_attributes() (in module scidbpy.schema_utils), 92
to_dimensions() (in module scidbpy.schema_utils), 93
toarray() (scidbpy.interface.SciDBInterface method), 47
toarray() (scidbpy.interface.SciDBShimInterface method), 59
toarray() (scidbpy.SciDBArray method), 35
todataframe() (scidbpy.interface.SciDBInterface method), 47
todataframe() (scidbpy.interface.SciDBShimInterface method), 59
todataframe() (scidbpy.SciDBArray method), 35
tolist() (scidbpy.SciDBArray method), 36
tosparse() (scidbpy.interface.SciDBInterface method), 47
tosparse() (scidbpy.interface.SciDBShimInterface method), 59
tosparse() (scidbpy.SciDBArray method), 36
transpose() (in module scidbpy), 66
transpose() (scidbpy.SciDBArray method), 36

U

unfold() (in module scidbpy), 80
uniq() (in module scidbpy), 84
uniq() (in module scidbpy.robust), 86
unique() (scidbpy.interface.SciDBInterface method), 47
unique() (scidbpy.interface.SciDBShimInterface method), 59
unload_library() (in module scidbpy), 77
unpack() (in module scidbpy), 72
unpack() (scidbpy.SciDBArray method), 36

V

var() (scidbpy.aggregation.GroupBy method), 62
var() (scidbpy.interface.SciDBInterface method), 48
var() (scidbpy.interface.SciDBShimInterface method), 59
var() (scidbpy.SciDBArray method), 36
variable_window() (in module scidbpy), 73
versions() (in module scidbpy), 77
vstack() (scidbpy.interface.SciDBInterface method), 48
vstack() (scidbpy.interface.SciDBShimInterface method), 60

W

window() (in module scidbpy), 81
wrap_array() (scidbpy.interface.SciDBInterface method), 48
wrap_array() (scidbpy.interface.SciDBShimInterface method), 60

X

xgrid() (in module scidbpy), 69

Z

zero_indexed() (in module scidbpy.schema_utils), 93
zeros() (scidbpy.interface.SciDBInterface method), 48
zeros() (scidbpy.interface.SciDBShimInterface method), 60