SciDB-Py Documentation

Release 0.2

Jake Vanderplas

July 31, 2014

Contents

CHAPTER 1

Installing SciDB-Py

1.1 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 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 1.6-1.7.

Requests tested with version 1.2. Required for using the Shim interface to SciDB.

Pandas (optional) tested with version 0.10. 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.

1.3 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/jakevdp/scidb-py. Install the development package directly from Github with:

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

or download the code and type:

```
python setup.py install
```

CHAPTER 2

Getting Started

See *Installing SciDB-Py* and the *Tutorial* for more information.

CHAPTER 3

Tutorial

SciDB is an open-source database that organizes data in n-dimensional arrays. SciDB features include ACID transactions, parallel processing, distributed storage, efficient sparse array storage, and native parallel linear algebra operations. The ScidbPy package for Python defines a NumPy array-like interface for SciDB arrays in Python. The arrays mimic numpy arrays, but operations on them are performed by the SciDB engine. Data are materialized to Python only when requested. A basic set of array subsetting, arithmetic and utility operations are defined by the package. Additionally, a general query function provides a mechanism for performing arbitrary queries on scidbpy array objects.

3.1 Installation

For details on installation, see *Installing SciDB-Py*.

3.2 Loading the scidbpy package and connecting to SciDB

In order to use SciDB, the Python instance needs an interface to a SciDB server. This is accomplished through the `SciDBInterface` class. `SciDBInterface` is an abstract base class which is designed to be extended with various interface methods: currently the one implemented interface is `SciDBShimInterface`, and interface using the Shim HTTP protocol developed by Paradigm4.

The following example loads the package and defines an object named `sdb` that represents the SciDB interface. The example assumes that the SciDB coordinator is on the computer with host name 'localhost' – adjust the host name as required if SciDB is on a different computer:

```
>>> import numpy as np
>>> from scidbpy import interface, SciDBQueryError, SciDBArray
>>> sdb = interface.SciDBShimInterface('http://localhost:8080')
```

The following examples refer to an interface object named `sdb` similar to the illustration.

3.3 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.

The `SciDBArray` class defines the primary method of interaction between Python and SciDB. `SciDBArray` objects are Python representations of SciDB arrays that mimic numpy arrays in may 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`, and single characters.

3.4 Creating SciDB array objects

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.

3.4.1 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`:

```
>>> 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).

3.4.2 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.

3.4.3 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.

3.4.4 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 `scidbpy`. 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 scidbpy 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.

3.4.5 Scope of scidbpy arrays

The `new_array()` function takes an argument named `persistent`. When `persistent` is set to True, arrays last in SciDB until explicitly removed by a `remove` (AFL) or `DROP` (AQL) query. If `persistent` is set to False, the arrays are removed from SciDB when they fall out of scope in the Python session. 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
>>> X.name in sdb.list_arrays()
True
>>> del X  # remove all references to X
>>> X.name in sdb.list_arrays()
False
```

3.5 Retrieving data from SciDB array objects

A central idea of the package is to program operations on SciDB arrays in a natural Python dialect, computing those operations in SciDB while minimizing data traffic between SciDB 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
```

3.5.1 Tridiagonal Example

Let's consider a more complicated example. Here we'll use the advanced query syntax (discussed below) to efficiently create a 10x10 tridiagonal array, and import its data in both dense and sparse formats:

```
>>> tridiag = sdb.new_array((10, 10))
>>> sdb.query('store( \
...               build_sparse({A}, \
...                 iif({A.d0}={A.d1}, 2, -1), \
...                 {A.d0} <= {A.d1}+1 and {A.d0} >= {A.d1}-1), \
...               {A})',
...            A=tridiag)

>>> # Materialize SciDB array to Python as a numpy array:
>>> tridiag.toarray()
array([[ 2.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [-1.,  2.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0., -1.,  2.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0., -1.,  2.,  1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0., -1.,  2.,  1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0., -1.,  2.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0., -1.,  2.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0., -1.,  2.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0., -1.,  2.,  1.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., -1.,  2.]])

>>> # Materialize SciDB array to Python as a scipy.sparse array:
>>> tridiag.tosparse('csr')
<10x10 sparse matrix of type '<type 'numpy.float64'>'
        with 28 stored elements in Compressed Sparse Row format>
```

3.6 Operations 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]'
```

3.6.1 Element Access

Single elements of `SciDBArray` objects can be referenced with the standard numpy indexing syntax. These single elements are returned by value. Here we'll use the `tridiag` array created above:

```
>>> tridiag[1, 1]
2
>>> tridiag[0, 1]
-1
```

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

3.6.2 Subarrays

Rectilinear subarrays are also selected with standard numpy syntax. Subarrays of `SciDBArray` objects are new `SciDBArray` objects. Consider the example sparse tridiagonal array used previously:

```
>>> # Define a 3x10 subarray (returned as a new SciDBArray object)
>>> X = tridiag[2:5,:]
>>> X.toarray()
array([[ 0., -1.,  2.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0., -1.,  2.,  1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0., -1.,  2.,  1.,  0.,  0.,  0.,  0.]])
```

Tuple-based indexing (also known as *fancy indexing*) is not yet supported.

Note that subarray indexing of `SciDBArray` objects follows numpy convention. SciDB arrays with negative-valued coordinate indices should be translated to a coordinate system with a nonnegative origin.

3.6.3 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 (using the `tridiag` array defined previously):

```
>>> 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
```

3.6.4 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

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
```

3.6.5 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)
```

3.6.6 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)
```

3.6.7 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(1)
>>> 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.

3.7 Advanced Usage: the SciDB Query Interface

`scidbpy` provides python wrappers for many useful SciDB operations, but the SciDB AFL and AQL query languages 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 sparse tridiagonal array, similar to the one used in the above examples:

```
>>> tridiag = sdb.new_array((5, 5))
>>> sdb.query('store( \
...              build_sparse({A}, \
...                iif({A.d0}={A.d1}, 2, -1), \
...                {A.d0} <= {A.d1}+1 and {A.d0} >= {A.d1}-1), \
...              {A})',
...           A=tridiag)
>>> 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 sparse 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.

3.8 Example applications

3.8.1 Covariance and correlation matrices

We can use SciDB's distributed parallel linear algebra operations to compute covariance and correlation matrices without too much difficulty. The following example computes the covariance and correlation between the columns of the matrices X and Y. We break the example up into a few parts for clarity.

Part 1, set up some example matrices:

```
>>> # Create a small test array with 5 columns:
>>> X = sdb.from_array(np.random.random((10, 5)))

>>> # Create a second small test array with 10 columns:
>>> Y = sdb.from_array(np.random.random((10, 10)))
```

Part 2, center the example matrices:

```
>>> # Subtract the column means from X using broadcasting:
>>> XC = X - X.mean(0)

>>> # Similarly subtract the column means from Y:
>>> YC = Y - Y.mean(0)
```

Part 3, compute the covariance matrix:

```
>>> COV = sdb.dot(XC.T, YC) / (X.shape[0] - 1)
```

Part 4, compute the correlation matrix:

```
>>> # Column vector with column standard deviations of X matrix:
>>> xsd = X.std(1).reshape((5, 1))

>>> # Row vector with column standard deviations of Y matrix:
>>> ysd = Y.std(1).reshape((1, 10))

>>> # Their outer product:
>>> outersd = sdb.dot(xsd, ysd)

>>> COR = COV / outersd
>>> COR.toarray()
array([[ 0.66403867, -0.3750696 ,  0.07049322,  0.3543565 ,  0.19460585,
         0.12932699, -0.32011222,  0.54153159,  0.08729989,  0.73609071],
       [-0.1918192 , -0.09265041, -0.44833276, -0.41902633,  0.47258236,
        -0.36989832,  0.39182811,  0.29622453,  0.13236683,  0.18712435],
       [ 0.63690931, -0.15126441,  0.48608902,  0.47874834, -0.32307991,
         0.10882704, -0.1950527 ,  0.23340459, -0.06955862,  0.76675295],
       [-0.12190756, -0.21477533, -0.54087076, -0.2134786 ,  0.71007003,
        -0.08317723, -0.15595565,  0.08533781,  0.12180002, -0.16792813],
       [ 0.39885069, -0.4231043 , -0.00321296, -0.23295093,  0.48057586,
        -0.22329337, -0.39134802,  0.67833575,  0.49968504, -0.05949198]])
```

The overhead of working interactively with SciDB can make these examples run somewhat slowly for small problems. But the same code shown here can be applied to arbitrarily large matrices, and those computations can run in parallel across a cluster.

CHAPTER 4

Code Reference

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

4.1 SciDB Array Class

4.2 SciDB Interface

4.2.1 Base Class

4.2.2 Shim Interface

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*