SciDB Quick Start

SciDB Version 14.3

Table of Contents

1. Introduction

This guide helps new users begin working with SciDB, and R users begin working with SciDB-R. Additionally, we have added a Python interface, similar to the one for R.

SciDB is an all-in-one data management and advanced analytics platform. It provides massively scalable complex analytics inside a next-generation database with data versioning to support the needs of commercial and scientific applications. SciDB organizes data in *n*-dimensional arrays. Its features include ACID transactions, parallel processing, distributed storage, efficient sparse array storage, and native linear algebra operations.

SciDB-R lets you remain an R programmer, but expands R's power to include SciDB's massive-scale data management and analytical capabilities. With SciDB-R, you can do all of the following *from inside an R program*:

- Use SciDB as a storage backend
- Use SciDB to offload large computations to a cluster
- Use SciDB to filter and join data before performing analytics
- Use SciDB to share data among multiple users, all with ACID guarantees
- Use SciDB to perform multidimensional windowing and aggregation
- Use SciDB's massively scalable analytical capabilities, including statistical methods, correlation, and dense and sparse linear algebra operations

SciDB-Py, similar to SciDB-R, lets you remain a Python programmer while including the powerful SciDB database capabilities.

This document describes how to use a Virtual Machine (VM) with both SciDB and the packages for SciDB-R and SciDB-Py. Virtual Machine images described in this document are preloaded with two SciDB-Py demos: Introduction to SciDB-Py and MODIS. Introduction to SciDB-Py gives a brief overview of the Python-scidb interface while MODIS showcases the usage of SciDB-Py to work with a sizeable data set. You can use these images to quickly and easily get started using SciDB and the SciDB package for R or Python.

Note that both the VirtualBox and AMI SciDB installations are small-scale. That is, as configured, they are useful as an introduction to SciDB, and for small amounts of data. They do, however, contain a complete version of SciDB. In fact, for the AMI, you could simply adjust the settings in the config.ini file and then use SciDB on a large-scale EC2 cluster.

All of the information for downloading and installing the individual packages listed above is available on our main website.

In addition, we provide the following documents:

- *SciDB and R* describes the R front-end to SciDB. Link: SciDB and R
- *SciDB User's Guide* provides a complete guide to SciDB. Link: Download SciDB Releases

For more information:

- Consult https://www.virtualbox.org/ for more information about VirtualBox.
- See the R project page on github for help from the user community about the SciDB package for R.
- See the Comprehensive R Archive Network (CRAN) for details about R.

If you have any questions or concerns, contact `<support@scidb.org>` for help getting started.

2. Download and start the image

Paradigm4 provides both a VirtualBox image and an Amazon Machine Image. Use whichever you prefer.

The provided VMs include SciDB 14.3 and the latest R and Python packages. All packages are available with SSL/TLS-encryption. A list of available web-based tools is reported to you when you log in.

Additionally, each image contains the following:

- authenticated web tools: RStudio, IPython Notebook, and a simple SciDB web console
- a brief text document describing the VM configuration process in the `/home/scidb` directory
- a reduced-sized MODIS example, loaded and runnable from both Python and R (as well as some other examples)

To log onto the VirtualBox image or AMI, from either the command prompt or from one of the web interfaces, use the following credentials:

- User name `scidb`
- Password: `paradigm4`

2.1. Use a VirtualBox Image

The VirtualBox Image is running on CentOS 6.

1. Download the 14.3 image provided by Paradigm4 in the following folder:

http://downloads.paradigm4.com/QuickStart/

Download the .ova file to a partition with at least 8GB of available space.

2. Open VirtualBox and click **File** | **Import Appliance** and select the downloaded file.

3. Start the VirtualBox image.

By default, the VM is configured to use "bridged" networking. Bridged networking presents the VM as a new computer to whatever network the host machine is connected to, with DHCP network configuration. The VM may or may not be able to obtain an IP address with this approach.

If the VM can obtain IP addresses, you will see them listed after you start the image. The start screen lists the following information:

```
Welcome to the SciDB VM!

A simple SciDB web interface is available by directing your browser to:
https://192.168.56.101:8083
:
:
```

Note that your URL may be different. If you see valid IP addresses, you can continue to section 4.

Note

Note: You can always log in to a text console in the VM and use SciDB from the command prompt, whether or not the networking is configured correctly.

However, if the VM is not able to obtain IP addresses using bridged networking, you will not see the welcome message. In this case, perform the following steps to set up a Host-only Network for VirtualBox.

1. Power off the VM.

2. From the VirtualBox menu, select **File** | **Preferences**.

3. Select **Network** and add a Host-only Network if you do not already have one. You can use all of the default settings.

4. Click **OK**.

5. Select your SciDB VM, and click **Settings**.

6. Select **Network**, and for `Adapter 1`, change from **Bridged Adapter** to **Host-only Adapter**.

7. The name will be populated with the Host-only Network adapter that you added in step 3. Click **OK** and then restart the VM.

SciDB is configured to start as a service inside the VM. When SciDB startup is successful, three access methods are presented by the VM in a startup text message:

- Access RStudio from a web browser on your computer.
- Access IPython Notebook from a web browser on your computer.
- Access a simple SciDB web console from a web browser on your computer.
- Log onto a VM shell and work from the command line.

To continue, see Section 3, “Connect to R, Python or SciDB”.

2.2. Use an Amazon Machine Image

We have created an Amazon Machine Image (AMI) that contains a saved Virtual Machine with SciDB and R installed. You can use Amazon's EC2 service to instantly launch this machine, log into it and use the SciDB-R package. It requires an Amazon Web Services account and Amazon will charge you a small hourly fee for using their EC2 service. Consult http://aws.amazon.com/ for more information.

The operating system for the AMI is Ubuntu 12.04.

To work with the AMI, perform the following steps:

1. Create the AMI image.
    a. If you do not already have an Amazon AWS account, create one at http://aws.amazon.com/.
    b. On the Amazon Web Services page, select **My Account/Console** > **AWS Management Console**.
    c. Click **EC2**.
    d. Make sure that your region is set to **US East / N. Virginia**. If you need to change your region, use the drop-down menu from the top navigation bar, next to your username.
    e. Click **Instances**, then **Launch Instance**.
    f. From the **Quick Start** menu, select **Community AMIs**.
    g. In the search box, enter `ami-9f132cf6` (the name of the SciDB image) into the text field and hit **Enter**. This should return the details for the ami-9f132cf6 image.
    h. Click **Select**.
    i. Change the instance type and storage. For exploring some of sample data that comes pre-loaded in the AMI, the default image type and storage is inadequate.
        - From the screen **Step 2: Choose an Instance Type**: click **All instance types** menu, and select **m3.2xlarge**.
        - From the screen **Step 4: Add Storage**: change the size from the default (16 GB) to 200.
    j. Click **Review and Launch**. You can edit your AMI. You need to open some ports, and you may want to use key pairs:
        - **Edit Security Groups:** For your security group, create rules permitting incoming TCP connections for ports 22 (SSH), 8083 (SciDB web console), 8787 (RStudio), and 8888 (IPython). The listed ports use SSL-encrypted communication and require user authentication. For each port, in the **Source** field, select **Anywhere** (which allows connections from IP address).
        - **Edit Tags: (Optional)** If desired, you may create a Key Pair or use an existing one. This will allow logging into the instance as the **scidb** user using PKI rather than a password.

          If you do not create any tags, you will be asked to **Proceed without a key pair**, and acknowledge that users can only connect to this AMI if they know a valid user name and password.
2. Once you have configured your image, launch and log onto the image.
    a. Click **Launch**. You are asked to confirm your key pair (or to proceed without one), and then the instance is launched.
    b. Click **View Instances** to return to your AWS console page, where your instances are listed.
    c. Wait a few minutes for the image to come fully online. When you see `running` in the **State** column, click the Instance.

d. The instance details are displayed in the the lower portion of the page. Below the instance name is the public DNS for the instance. For example:

```
ec2-54-242-36-232.compute-1.amazonaws.com
```

Note that this is an example: your public DNS will be different.

e. Open a terminal window from your computer to SSH into the AMI as the scidb user.

```
ssh scidb@ec2-54-242-36-232.compute-1.amazonaws.com
```

Again, the URL is an example only. When asked for the scidb password, enter **paradigm4**.

Note

Note: The standard EC2 approach of using a PEM key pair to log in as the scidb user is always supported as well.

3. The supplied database is **mydb**, and should be running when you log on. The database is configured as a 4-instance cluster. You can, of course, log on and modify the configuration to suit your needs, by performing the following steps:

   1. Stop the database:

      ```
      scidb.py stopall mydb
      ```

   2. Edit the `/opt/scidb/14.3/etc/config.ini` file to reflect the desired configuration.

   3. Run the following command to initialize the database:

      ```
      scidb.py initall-force mydb
      ```

      WARNING: This command will delete all of the data from the database.

   4. Run the following command to start the database:

      ```
      scidb.py startall mydb
      ```

4. In addition to using SciDB from the command line, you can access a GUI for SciDB or for RStudio or IPython Notebook. Point your browser to one of the following URLs (assuming the same "base-URL" listed in step 2.d):

   - To use SciDB: https://ec2-54-242-36-232.compute-1.amazonaws.com:8083
   - To use SciDB-R: http://ec2-54-242-36-232.compute-1.amazonaws.com:8787
   - To use SciDB-Py: https://ec2-54-242-36-232.compute-1.amazonaws.com:8888

   When asked to log on, remember that the username is **scidb** and the password is **paradigm4**.

Note

Note: If you have problems connecting to the running AMI, reboot the AMI machine through the Amazon Web interface.

To continue, see Section 3, “Connect to R, Python or SciDB”.

3. Connect to R, Python or SciDB

Once you have downloaded the VirtualBox image, or created your own AMI from the template, you can proceed to connect to SciDB, IPython Notebook or RStudio.

3.1. Connect to RStudio or IPython Notebook

Depending on whether you are using a VirtualBox image or an Amazon Machine image, you can access RStudio:

- VirtualBox Image: you should have noted the URL for RStudio when you started the VM. For example, **http://192.168.56.101:8787**
- AMI: use the public DNS for the AMI, and append **:8787** for the port. For example: **ec2-54-242-36-232.compute-1.amazonaws.com:8787**

You can start by running the following short example:

```
library("scidb")
scidbconnect()
iquery("store(build(<v:double>[i=0:9,10,0,j=0:9,10,0],random()%100),B)")
scidblist()
```

To access IPython Notebook:

- VirtualBox Image: you should have noted the URL for Python when you started the VM. For example, **https://192.168.56.101:8888**
- AMI: use the public DNS for the AMI, and append **:8888** for the port. For example: **ec2-54-242-36-232.compute-1.amazonaws.com:8888**

3.2. Connect to SciDB

You can connect to a SciDB Web console, or use SciDB in a terminal shell.

Access a SciDB Web Console

You can access a SciDB web console to view logs and run queries from your browser. Depending on whether you are using a VirtualBox image or an Amazon Machine image, do one of the following:

- VirtualBox Image: you should have noted the SciDB web URL when you started the VM. For example, **https://192.168.56.101:8083**
- AMI: connect to a secure HTTP service running on the public DNS for the AMI on port 8083. For example: **https://ec2-54-242-36-232.compute-1.amazonaws.com:8083**

The URL home page displays a brief summary of the running SciDB configuration and a button that you can press to view the SciDB coordinator log.

Select **Interactive Query** from the menu along the top of the page to view a basic interactive query screen. You can perform the following actions:

- Enter a valid AFL query in the text box and press **Execute query and return result** to run the query and view its output,
- Enter a query and press **Execute query** to run the query without viewing output. Note that non-operators like **remove** and DDL operators like **rename** only work with the **Execute query** button.
- Press **Upload file** for a simplified CSV file uploading dialog.

Note

If you are experiencing problems with the SciDB web console, try logging out (click **Logout**) and then log back in with your credentials.

Log on to the VM Shell

Start the VirtualBox image, or SSH into the AMI. At the command prompt, enter your credentials:

- User name: `scidb`
- Password: `paradigm4`

For more information on SciDB, see the *SciDB User's Guide*.

Note

Tip: if you are using the VirtualBox image from the command prompt, and you are having trouble cutting and pasting information, you can SSH into the VirtualBox image. For example, if the URL for the image (displayed upon starting the image) is **https://192.168.56.101:8083**, you could open a Linux prompt and type the following command:

```
$ ssh scidb@192.168.56.101
```

When prompted for the password, enter `paradigm4`. You can now work from your Linux desktop.

4. Sample Data

The virtual machine image contains some sample data. The **XLDB_Examples** folder contains scripts that were used to create and load the sample arrays.

5. Terminology

The following terms are useful when learning about SciDB.

Arrays
: SciDB uses multidimensional arrays as its basic storage and processing unit. An array has any number of dimensions and attributes (see below). A user creates a SciDB array by specifying dimensions and attributes of the array.

Dimensions
: Dimensions form the coordinate system. An n-dimensional SciDB array has dimensions d_1, d_2, ..., d_n. The size of the dimension is the number of ordered values in that dimension.

Attributes
: Each combination of dimension values identifies a cell or element of an array, which can hold multiple data values called attributes (a_1, a_2, ..., a_m). Each data value is referred to as an attribute, and belongs to one of the supported data types in SciDB.

AFL (Array Functional Language)
: AFL is a functional language for working with SciDB arrays. AFL operators are used to compose queries or statements. AFL contains operators for performing both data definition and data manipulation. All queries in this guide are composed of AFL operators.

AQL (Array Query Language)
: AQL is a high-level declarative language for working with SciDB arrays. It is similar to the SQL language for relational databases, but uses an array-based data model and a more comprehensive analytical query set compared with standard relational databases.

iquery
: The iquery executable is the basic command-line tool for communicating with SciDB. iquery is the default SciDB client used to issue AQL and AFL commands.

An important part of SciDB database design is selecting which values will be *dimensions* and which will be *attributes*. You can get guidance on database design in the **Database Design** section of the **Creating and Removing Arrays** chapter in the full documentation.

A few observations to keep in mind when you are defining your arrays:

- Joins of arrays are performed along dimensions.
- Operators such as **slice** and **subarray** use dimensions to quickly select subregions.
- Group-by clauses work over dimensions.
- Aggregates work over attributes.
- Filtering over dimensions is faster than filtering over attributes.

6. Learning SciDB

To get acquainted with SciDB, you will perform the following tasks.

1. Create a simple array and retrieve some information about the array.
2. Load data into an array. There are several ways to do this. We describe a few of them.
3. Examine ways to select sub-regions from an array.
4. Walk through an advanced example that combines several operators into a single query.

Note that all the example queries end with a semicolon. If you are in an iquery session from a Linux command prompt, you need to end queries with a semicolon. However, if you are using the SciDB web interface, do not end queries with a semicolon—you will get an error if you do so.

6.1. Creating an Array

Note

Note: In this guide, the output of queries is formatted to fit the page nicely. When you run your queries, the output will be the same—but the format of the output will be different. SciDB offers many formatting options. One useful one is the **dcsv** format, illustrated below.

The basic way to create an array in SciDB is to use the CREATE ARRAY statement.

1. From a command prompt, start iquery with AFL as the language.

   ```
   $ iquery -a
   ```

   Optionally, you can start iquery with the dcsv format option: this will display output one cell per line, with some other useful information.

   ```
   $ iquery -o dcsv -a
   ```

   Either command opens an AFL command prompt.

   ```
   AFL%
   ```

2. Use the CREATE ARRAY statement to create an array.

   ```
   AFL% CREATE ARRAY test <val:double>[i=0:4,6,0, j=0:4,6,0];
   ```

   This creates a 5x5 array, with a single, double precision floating-point attribute. Note the following about the CREATE ARRAY statement:

   - The first argument in the name for the array, in this case `test`.
   - The next argument is the attribute list, contained within '<>'. Here we have only a single attribute, so we specify its name and data type.

- The third argument is the dimension list, contained within '[]'.
- For each dimension, we specify its name, lower bound, upper bound, chunk size, and chunk overlap.
- The first dimension is `i`, starting at 0 and ranging to 4, setting the dimension size to 5. The chunk size for both dimensions is set to 6. In this guide, we always use 0 for the value of the chunk overlap. For details about chunk overlap, see the "Array Dimensions" section of the *SciDB User's Guide*.
- In this case, the second dimension, `j`, has the same values as the first dimension, but that does not need to be the case.

6.2. Getting Array Information

After you create an array, you can retrieve information about the array.

- List all of the existing arrays.

```
AFL% list('arrays');
```

```
name,id,schema,availability
'test',79,'test<val:double> [i=0:4,6,0,j=0:4,6,0]',true
```

- Retrieve the schema for a particular array.

```
AFL% show(test);
```

```
[('test<val:double> [i=0:4,6,0,j=0:4,6,0]')]
```

- Retrieve the contents for a particular array.

```
AFL% scan(test);
```

```
[[]]
```

As you can see, `test` is currently empty.

6.3. Populating an Array with Data

SciDB offers several ways to get data into an array.

- Populate the array with the same value in every cell
- Populate the array using an expression
- Populate the array from a file

6.3.1. Constant Values

This section describes how to populate an array with a single value. We use the **build** operator, which takes two arguments:

- A **schema**, which is the list of attributes and dimensions, with their details. Note that for the **build** operator, you must specify *exactly* one attribute.
- An **expression**, which specifies the values for the attribute.

1. Populate an array. The following query fills a 5x5 array with 1's.

```
AFL% build(<val:double>[i=0:4,6,0, j=0:4,6,0],1);
```

```
[
[(1),(1),(1),(1),(1)],
```

```
[(1),(1),(1),(1),(1)],
[(1),(1),(1),(1),(1)],
[(1),(1),(1),(1),(1)],
[(1),(1),(1),(1),(1)]
]
```

Note that this query does not store the result anywhere—that is, SciDB does not create a new array filled with 1's.

2. To store the result of our previous query, we need to use the **store** operator.

```
AFL% store(build(<val:double>[i=0:4,6,0, j=0:4,6,0],1),test);
```

```
[
[(1),(1),(1),(1),(1)],
[(1),(1),(1),(1),(1)],
[(1),(1),(1),(1),(1)],
[(1),(1),(1),(1),(1)],
[(1),(1),(1),(1),(1)]
]
```

This stores the output from the **build** operator into the array, `test`.

6.3.2. Calculated Values

This section describes how to use expressions to populate an array with values.

1. SciDB has a random function, **random()**, that is useful for populating an array with randomly-generated values. Here, we create a 3x4 array, and fill it with random numbers between 10 and 99.

```
AFL% store(build(<randomVal:int64>[i=0:2,3,0, j=0:3,4,0],random()%90+10),random_100);
```

```
[
[(56),(19),(47),(44)],
[(34),(95),(62),(97)],
[(84),(33),(55),(87)]
]
```

Note that in this query, our attribute is of type int64 (an integer data type). Of course, your output will differ, as the random() function produces random output.

2. This example uses the **iif** operator—inline if—to populate an array with two different values.

```
AFL% store(build(<val:double>[i=0:4,6,0, j=0:4,6,0],iif(i=j,1,0)),test);
```

```
[
[(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)]
]
```

Note that we have created a 5x5 identity array. The **iif** operator takes three arguments:

- An expression to evaluate. In this case, the expression is i=j.
- A value to store into a cell if the expression is true.
- A value to store into a cell if the expression is false.

So, in this example, when i=j, we store 1, and in all other cells, we store 0.

3. You can nest the **iif** operator to get more control over the values to add.

```
AFL% store(build(<val:double>[i=0:4,6,0, j=0:4,6,0],iif(i>j,1,iif(i=1,7,0))),test);
```

```
[
```

```
[(0),(0),(0),(0),(0)],
[(1),(7),(7),(7),(7)],
[(1),(1),(0),(0),(0)],
[(1),(1),(1),(0),(0)],
[(1),(1),(1),(1),(0)]
]
```

In this example, if i > j, we add a 1 to the array, and if i <= j, SciDB evaluates the nested **iif** function, and adds a 7 or 0 to the array, depending on whether the second expression is true.

6.3.3. Load from a File

In many cases, your data exists in a file on disk. Here we describe how to load from a CSV-formatted file into a SciDB array. There are several other approaches to loading data from disk files, and they are discussed in the *SciDB User's Guide*.

Suppose that you have a CSV file, `/tmp/datafile.csv`, with the following content:

```
Type,MPG
Truck, 23.5
Sedan, 48.7
SUV, 19.6
Convertible, 26.8
```

We will use the SciDB utility, **csv2scidb**, to convert this file into the SciDB format, and then load it into an array.

1. Create a SciDB array to hold the data.

   ```
   AFL% create array Aprime <type:string, mpg:double> [x=0:*,10,0];
   ```

2. Convert the data into the SciDB format. Note that you run **csv2scidb** from your Linux terminal prompt, not from inside an iquery session.

   ```
   $ csv2scidb -p SN -s 1 < /tmp/datafile.csv > /tmp/datafile.scidb
   ```

   The **csv2scidb** utility is described in the *SciDB User's Guide*.

3. Now load the SciDB-formatted file into your array. For the **load** query, you need to specify the full system path name for the file that you are loading.

   ```
   AFL% load(Aprime, '/tmp/datafile.scidb');
   ```

   ```
   {x} type,mpg
   {0} "Truck",23.5
   {1} "Sedan",48.7
   {2} "SUV",19.6
   {3} "Convertible",26.8
   ```

6.4. Selecting Subsets of an Array

One common task is selecting subsets of an array. SciDB allows you to reduce matrices to contiguous or noncontiguous subsets of its cells.

Let's look at a single array to compare three ways to reduce an array: `subarray`, `slice`, and `thin`.

1. We will use the array that we created earlier, **test**, and fill it with values.

   ```
   AFL% store(build(<val:double>[i=0:4,6,0, j=0:4,6,0], i*5 +j+1),test);
   ```

   ```
   [
   [(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)]
   ]
   ```

2. Select a 3x3 sub array from the interior of the array:

```
AFL% subarray(test,1,1,3,3);
```

```
{i,j} val
{0,0} 7
{0,1} 8
{0,2} 9
{1,0} 12
{1,1} 13
{1,2} 14
{2,0} 17
{2,1} 18
{2,2} 19
```

3. Slice the third column (j=2), and then the second row (i=1):

```
AFL% slice(test,j,2);
```

```
[(3),(8),(13),(18),(23)]
```

```
AFL% slice(test,i,1);
```

```
[(6),(7),(8),(9),(10)]
```

4. Use the `thin` operator to uniformly sample data from the array.

```
AFL% thin(test,0,2,0,2);
```

```
[
[(1),(3),(5)],
[(11),(13),(15)],
[(21),(23),(25)]
]
```

```
AFL% thin(test,1,3,1,2);
```

```
[
[(7),(9)],
[(22),(24)]
]
```

   The thin operator selects elements from given array dimensions at defined intervals.

6.5. The Power of Operator Composition

You can use combinations of operations on SciDB data. This allows you to view and analyze data in a nearly endless variety of ways.

For example, let's look at a query to list the SciDB functions:

```
AFL% list('functions');
```

If you run this query, it returns several hundred elements. Here are the first few items:

```
{No} name,profile,deterministic,library
{0} "%","double %(double,double)",true,"scidb"
{1} "%","int16 %(int16,int16)",true,"scidb"
{2} "%","int32 %(int32,int32)",true,"scidb"
{3} "%","int64 %(int64,int64)",true,"scidb"
{4} "%","int8 %(int8,int8)",true,"scidb"
{5} "%","uint16 %(uint16,uint16)",true,"scidb"
{6} "%","uint32 %(uint32,uint32)",true,"scidb"
{7} "%","uint64 %(uint64,uint64)",true,"scidb"
{8} "%","uint8 %(uint8,uint8)",true,"scidb"
```

Note that the remainder function, %, is overloaded—it can be used on most of the numeric data types. Several of the other SciDB functions are also overloaded.

Now let's say that you want only the names of functions, and you want them sorted, and you only want one record per function name. One way to do this is as follows:

1. Run the following query to create an array:

```
AFL% store(sort(project(filter(list('functions'),
   library='scidb'),name)),functionsArray);
```

   Let's break down the actions in this query:

   - Starting from the inside—which is where the SciDB engine starts—we have `list('functions')`, which lists information about all functions.
   - We `filter` the list of functions to return only the ones in SciDB itself (not in any add-on libraries).
   - Next, we `project` only the name of the functions—this is similar to **SELECT name** from SQL.
   - We then `sort` the list and `store` it to a SciDB array.

2. Let's take a look at the schema of `functionsArray`:

```
AFL% show(functionsArray);
```

```
[('functionsArray<name:string> [n=0:*,220,0]')]
```

3. We will use the `uniq()` operator to remove duplicate values. `uniq()` works on a sorted, one-dimensional array, which we have.

```
AFL% uniq(functionsArray);
```

```
{i} name
{0} '%'
{1} '*'
{2} '+'
{3} '-'
{4} '/'
{5} '<'
{6} '<='
{7} '<>'
{8} '='
{9} '>'
{10} '>='
{11} 'abs'
{12} 'acos'
{13} 'and'
{14} 'append_offset'
{15} 'apply_offset'
{16} 'asin'
{17} 'atan'
{18} 'ceil'
{19} 'cos'
{20} 'day_of_week'
{21} 'exp'
{22} 'first_index'
{23} 'floor'
{24} 'format'
{25} 'get_offset'
{26} 'high'
{27} 'hour_of_day'
{28} 'iif'
{29} 'instanceid'
{30} 'is_nan'
{31} 'is_null'
{32} 'last_index'
{33} 'length'
{34} 'log'
{35} 'log10'
{36} 'low'
```

```
{37} 'max'
{38} 'min'
{39} 'missing'
{40} 'missing_reason'
{41} 'not'
{42} 'now'
{43} 'or'
{44} 'pow'
{45} 'random'
{46} 'regex'
{47} 'sin'
{48} 'sqrt'
{49} 'strchar'
{50} 'strftime'
{51} 'strip_offset'
{52} 'strlen'
{53} 'substr'
{54} 'tan'
{55} 'togmt'
{56} 'tznow'
```

7. Quick Reference

This section presents a cheat sheet that covers the most useful SciDB operators. The examples are run from inside an AFL iquery session (run **iquery -a** from your Linux command prompt).

7.1. Community Edition

The operators described in this section are all available in the community edition of SciDB.

CREATE ARRAY: Create an array

```
CREATE ARRAY:
create array ARRAY_NAME schema
schema := <ATTNAME:type [null],...> [DIMNAME=min:max,chunk,olap,...]
```

```
AFL% create array test_array <val:double> [x=1:3,1,0, y=1:3,1,0];
```

The previous query generates a "toy" array. Real data should have about one million cells per chunk, like so:

```
AFL% create array one_dimensional <val:double, v2:string> [i=0:*,1000000,0];
```

```
AFL% create array dense_matrix <val:double> [x=0:9999,1000,0, y=0:9999,1000,0];
```

```
AFL% create array sparser_matrix <val:double> [x=0:*,100000,0, y=0:*,100000,0];
```

STORE, BUILD: Store and build

```
store(input, ARRAY_NAME )
build(schema, expression)
```

```
AFL% store(build(<val:double>[x=1:3,1,0,y=1:3,1,0], x+y), test_array);
```

The following query fills the array with random values (from 0 to 99,999):

```
AFL% store(build(<val:double>[x=1:3,1,0,y=1:3,1,0],random()/100000.0), test_array);
```

The following query is equivalent to the previous query—since we already defined **test_array**, we can use the array name as a shorthand for the required schema:

```
AFL% store(build(test_array, random()/100000.0), test_array);
```

LIST: List and metadata

```
list('arrays' [,true])
list('functions')
list('aggregates')
```

```
list('operators')
list('types')
list('queries')
show(array)
dimensions(array)
attributes(array)
```

RENAME, REMOVE, INSERT, SCAN, ALLVERSIONS: Array management

```
rename(ARRAY_NAME, NEW_ARRAY_NAME)
remove(ARRAY_NAME)
insert(input, ARRAY_NAME) --merge input with last version of ARRAY_NAME, unlike store()
scan(ARRAY_NAME@version)
allversions(ARRAY_NAME)
```

CANCEL: Cancel a query

```
cancel(query_id) --obtain query_id by running list('queries')
```

LOAD, SAVE: Load and save to external files

```
load(array, '/path/to/file', [INSTANCE | -1], 'format')
save(array, '/path/to/file', [INSTANCE | -1], 'format')
```

If the path is relative, it is relative to each instance's data directory. The **format** can be '**opaque**', '**binary**' or blank for default text. Using **-1** means "try to load from all instances", using an integer value for **INSTANCE** means load from that instance only. For example, the following query saves to instance 0 (the coordinator instance) using the opaque format:

```
save(test_data, '/tmp/test', 0, 'opaque')
```

The following query does a distributed save, in binary format, for a 1-dimensional, dense array that has 3 attributes (of type int32, int64, and double):

```
save(test_data, 'test_data_piece.scidb', -1, '(int32,int64,double)')
```

The **loadcsv.py** program is the recommended way to load CSV files into a 1-dimensional array. See **redimension()** below for details on converting arrays into multidimensional form.

FILTER: Return only cells that satisfy a boolean predicate

```
filter(input, boolean expression)
```

```
AFL% filter(test_array, x>=3 and val<10000);
```

APPLY: Compute a scalar expression for each array cell

```
apply(input, NEW_ATTNAME, expression, [, NEW_ATTNAME2, expression2,...])
```

```
AFL% apply(test_array, v_2, val*x, v_sqrt, sqrt(val));
```

```
AFL% filter(apply(test_array, v2, iif(x = y, null, val)), x=y or x=3);
```

PROJECT: Return only specified attributes / reorder attributes

```
project(input, ATTNAME [, ATTNAME_2,...])
```

```
AFL% project(apply(test_array, v_2, val*x, v_sqrt, sqrt(val)), v_2);
```

SUBARRAY, BETWEEN, SLICE: Fast filtering on dimensions

```
subarray(input, x_begin, y_begin,... x_end, y_end...)
between(input, x_begin, y_begin,... x_end, y_end...)
slice(input, DIMNAME, value, [DIMNAME2, value2...])
```

```
AFL% between(test_array, 1,null,2,null);
```

The **subarray()** operator resets all array dimensions to start at 0 (for the result array).

```
AFL% subarray(test_array, 1,1,2,2);
```

The **slice()** operator collapses out all data in the sliced-on dimensions.

```
AFL% slice(test_array, x, 2);
```

CROSS_JOIN: Combine two arrays, aligning cells with equal dimension values

```
cross_join(input, input2, [input1.DIMNAME, input2.DIMNAME...])
```

```
AFL% store(filter(build(<val2:double> [x2=1:3,1,0], x2), x2<>2), test_strip);
```

Always place the smaller array as the second argument. Joined dimensions must have the same start coordinate and chunk size.

```
AFL% cross_join(test_array, test_strip, test_array.x, test_strip.x2);
```

For matrix centering using **cross_join()**, subtract the column means:

```
AFL% project(
        apply(
          cross_join(
            test_array as A, aggregate(test_array, avg(val) as av, y) as B, A.y, B.y
          ),
          d, A.val-B.av
        ),
        d
    );
```

For edge cases, there are other operators: **cross()** and **join()**.

MERGE: Union-like combination of two arrays

```
merge(input1, input2)
```

Dimensions and attributes must match, and the first argument gets priority.

```
AFL% merge(filter(test_array, x=y), build(test_array,0));
```

REPART: Change chunk sizes

```
repart(input, schema)
```

Used as a glue between aggregates, joins, and so on. Here, we move all the data into one chunk:

```
AFL% apply(repart(test_array, <v:double>[a=1:3,3,0,b=1:3,3,0]), iid, instanceid());
```

AGGREGATE: Compute summary statistics

```
aggregate(input, aggregate(ATTNAME)[as ALIAS] [, aggregate2...] [DIM,...])
```

```
AFL% aggregate(test_array, avg(val), sum(val));
```

The following query does a group by on dimension x:

```
AFL% aggregate(test_array, avg(val), sum(val), x);
```

count(*) returns the count of non-empty cells; **count(ATTNAME)** counts non-nulls:

```
AFL% aggregate(apply(test_array, v2, iif(x = y, null, val)), count(*), count(v2));
```

REGRID: Apply aggregates to fixed non-overlapping windows

```
regrid(input, INTERVAL_X, INTERVAL_Y,...
   aggregate(ATTNAME) [as ALIAS] [,aggregate2...])
```

Shrink our 3x3 matrix into a 2x2 matrix. The result at {2,2} is an aggregate of only 1 cell:

```
AFL% regrid(test_array, 2,2, avg(val), count(*));
```

WINDOW: Apply aggregates over a moving window

```
window(input, NUM_PRECEDING_X, NUM_FOLLOWING_X, NUM_PRECEDING_Y...,
   aggregate(ATTNAME) [as ALIAS] [,aggregate2...])
```

An average of a 3x3 window around each non-empty cell (empties ignored):

```
AFL% window(test_array, 1,1,1,1,avg(val));
```

```
AFL% window(test_array, 0,0,0,1,avg(val), count(*));
```

VARIABLE_WINDOW: Apply aggregates over a moving window that skips empty cells

```
variable_window(input, DIM, NUM_PRECEDING, NUM_FOLLOWING,
   aggregate(ATTNAME) [as ALIAS] [,aggregate2...])
```

One dimensional windows only.

```
AFL% variable_window(test_array, y, 3, 0, sum(val));
```

CUMULATE: Apply a cumulative aggregate along a specified dimension

```
cumulate (input, aggregate(ATTNAME) [as ALIAS] [, aggregate2...] [, DIM])
```

Like a variable window with ALL preceding, and 0 following.

```
AFL% cumulate(test_array, sum(val), count(*), y);
```

REDIMENSION: Promote attributes to dimensions, and vice versa

```
redimension(input, schema [, aggregate(ATTNAME) [as ALIAS],...])
```

Pay close attention to result chunk sizing—aim for an average of 1 million cells per chunk.

```
AFL% store( redimension( C, B), B);
```

The following query returns the count of cells on each instance.

```
AFL% redimension(
       apply(test_array, iid, int64(instanceid())),
       <count:uint64 null> [iid=0:*,10,0],
       count(*) as count
);
```

UNPACK, SORT: Unpack and sort

```
unpack(input, NEW_DIMNAME, [chunk_size])
sort(input, [, ATTNAME [asc|desc],..], [chunk_size])
```

Unpack flattens the array down to a single dimension, converting all existing dimensions to new attributes. Sort returns a sorted one-dimensional array of all attributes.

```
AFL% sort(test_array, val desc);
```

UNIQ: Select unique elements from a sorted array

```
uniq(input [,'chunk_size=SIZE'])
```

Input must be one-dimensional, dense, sorted, and contain a single attribute.

```
AFL% uniq(sort(test_array), 'chunk_size=100');
```

```
AFL% store(uniq(sort(project(trades_flat, symbol))), stock_symbols_index);
```

INDEX_LOOKUP: Use attributes of an array to lookup coordinates in another

```
index_lookup(input, input2, input.ATTNAME [, NEW_ATTNAME])
```

The array input2 is used as the index. It must be one-dimensional, with a single attribute. The operator looks up values of input.ATT in input2 and applies a new attribute: the int64 position of each item in input2, or null if not found.

Here we use the stock_symbols_index constructed above to compute number of trades and max price grouped by stock symbol:

```
AFL% redimension(
       index_lookup(
          trades_flat, stock_symbols_index, trades_flat.symbol, symbol_id
       ),
       <symbol:string, num_trades:uint64 null, high:double null>
       [symbol_id=0:*,1000,0],
       count(*) as num_trades, max(price) as high
     );
```

Find all elements in test_a that are not in test_b:

```
AFL% store(build(<val:int64> [x=1:1000,100,0], random()%5000), test_a);
```

```
AFL% store(build(<val:int64> [x=1:1000,100,0], random()%5000), test_b);
```

```
AFL% filter(index_lookup(test_a, test_b, test_a.val, idx), idx is null);
```

CAST: Change names of attributes or dimensions or make attributes nullable

```
cast(input, schema)
```

This is useful as a kind of mid-query glue, providing a way to rename attributes and dimensions so they can be subsequently referenced in expressions without ambiguity.

```
AFL% create array foo <v:double> [x=1:10,10,0];
AFL% create array foo2 <v:double> [x=1:10,10,0];
...
AFL% apply( join(foo, cast(foo2, <v2:double>[x2=1:10,10,0])), z, x2*(v-v2)...);
```

SUBSTITUTE: Replace null values

```
substitute(input, input2 [, ATTNAME, ATTNAME2,..])
```

SciDB nulls are numeric integer codes, with the common "null" being 0, followed by 1,2, and so on. **substitute()** works by using the missing code from the attribute of input as a coordinate into input2. The missing values in an input attribute are then replaced with the values of input2 at the corresponding position. The attribute is also marked as non-nullable.

The most common case is to substitute null with 0:

```
AFL% substitute(test, build (<val:double>[x=0:0,1,0], 0), test.attr);
```

Another common case is to substitute nulls with empty strings:

```
AFL% substitute(test, build (<val:string>[x=0:0,1,0], ''), test.attr, test.attr2);
```

GEMM: Multiplication of dense matrices

```
gemm(input1, input2, input3)
```

Returns input1 * input2 + input3. Inputs must have square chunk sizes of at least 32x32 and at most 1024x1024.

```
AFL% gemm(left, right, build(<val:double>[x=0:9,32,0,y=0:9,32,0], 0));
```

GESVD: Singular value decomposition of a dense matrix

```
gesvd(input, 'left|values|right')
```

Input must have square chunk size of at least 32x32 and at most 1024x1024. Specify the factor to return as either 'left', 'values', or 'right'.

```
AFL% gesvd(matrix, 'values');
```

7.2. Paradigm4 Extensions

The following operators are available in the proprietary, enterprise version of SciDB.

SPGEMM: Multiplication of sparse matrices

```
spgemm(input1, input2 [, 'ring_spec'])
```

Where ring_spec is one of 'min.+', 'max.+' or 'count-mults' (the default). The second dimension of input1 must match the first dimension of input2 in length and chunk size.

```
AFL% spgemm(left, right, 'min.+');
```

TSVD: Truncated singular value decomposition of sparse matrices

```
tsvd(input, inputT, n [, tol [, maxit [, initialVector [, left, right]]]])
```

Both input and inputT (transpose) must be chunked such that the chunk size along the second dimension includes the dimension entirely. n is the desired number of singular values to compute. For other options, see the documentation.

```
AFL% store(build(<v:double>[i=0:19,5,0,j=0:19,20,0],sin(i-j)),X);
AFL% store(redimension(X, <v:double>[j=0:19,10,0,i=0:19,20,0]), XT);
AFL% tsvd(X,XT,5,0.001,10);
```

GLM: Generalized linear model

```
glm(input, response, weights, 'distribution', 'link')
```

Input is a dense matrix of reals, response and weights are 1-dimensional arrays that match the number of rows in input. The distribution function is one of 'gaussian', 'poisson', 'binomial', or 'gamma'. The link function is one of 'identity', 'log', 'inverse', 'logit', or 'probit'.

```
AFL% store(build(<v:double>[i=1:5000,1000,0,j=1:50,50,0],(random()%1000)/1000.0),X);
AFL% store(build(<v:double>[i=1:5000,1000,0],random()%2),y);
AFL% glm(X,y,build(<v:double>[i=1:5000,1000,0],1),'binomial','logit');
```

KENDALL, PEARSON, SPEARMAN: Correlation metrics

```
kendall(input, input2)
pearson(input, input2 [, 'NaN handling policy'])
spearman(input, input2)
```

input1 and input2 must be RxC matrices of reals. Computes the distance metric of the columns of input1 against all columns of input2, returning a CxC result. These operators should be used with dense data only.