SciDB User's Guide

SciDB User's Guide, version 11.12

Copyright (C) 2008-2012, SciDB, Inc.

SciDB is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License, version 3, as published by the Free Software Foundation.

SciDB is distributed "AS-IS" AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License at http://www.gnu.org/licenses/ for the complete license terms.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1 Introduction to SciDB

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 and provenance to support the needs of commercial and scientific applications. SciDB is an open source software platform that runs on a grid of commodity hardware or in a cloud.

Paradigm4 Enterprise SciDB with Paradigm4 Extensions is an enterprise distribution of SciDB with additional linear algebra, high availability, and client connector features.

Unlike conventional relational databases designed around a row or column-oriented table data model, SciDB is an array database. The native array data model provides compact data storage and high performance operations on ordered data such as spatial (location-based) data, temporal (timeseries) data, and matrix-based data for linear algebra operations.

This document is a User's Guide, written for scientists and developers in various application areas who want to use SciDB as their scalable data management and analytic platform.

This chapter introduces the key technical concepts in SciDB -- its array data model, basic system architecture including distributed data management, salient features of the local storage manager, and the system catalog. It also provides an introduction to SciDB's array languages -- Array Functional Language (AFL) and Array Query Language (AQL) -- and an overview of transactions in SciDB.

1.1 Array Data Model

SciDB uses multidimensional arrays as its basic storage and processing unit. A user creates a SciDB array by specifying *dimensions* and *attributes* of the array.

Dimensions

An n-dimensional SciDB array has dimensions d1, d2, ..., dn. The size of the dimension is the number of ordered values in that dimension. For example, a 2-dimensional array may have dimensions *i* and *j*, each with values (1, 2, 3, ..., 10) and (1, 2, ..., 30) respectively.

Basic array dimensions are 64-bit integers. SciDB also supports arrays with one or more noninteger dimensions, such as variable-length strings (*alpha*, *beta*, *gamma*, ...) or floating-point values (1.2, 2.76, 4.3, ...).

When the total number of values or cardinality of a dimension is known in advance, the SciDB array can be declared with a *bounded* dimension. However, in many cases, the cardinality of the dimension may not be known at array creation time. In such cases, the SciDB array can be declared with an *unbounded* dimension.

Attributes

Each combination of dimension values identifies a cell or element of the array, which can hold multiple data values called attributes (a1, a2, ..., am). Each data value is referred to as an *attribute*, and belongs to one of the supported datatypes in SciDB.

At array creation time, the user must specify:

- An array name.
- An array type.
- Array dimensions. The name and size of each dimension must be declared.
- Array attributes of the array. The name and datatype of the each attribute must be declared.

Dimensions vs. Attributes

An important part of SciDB database design is selecting which values will be dimensions and which will be attributes. Dimensions form a *coordinate* system for the array. Adding dimensions to an array generally improves the performance of many types of queries by speeding up access to array data. Hence, the choice of dimensions depends on the types of queries expected to be run. Some guidelines for choosing dimensions:

- Dimensions provide selectivity and efficient access to sub-arrays of data within an array. Any coordinate along which such selection queries must be performed constitute a good choice of dimension.

- Array aggregation operators including group-by, window, or grid aggregates specify *coordinates* along which grouping must be performed. Such values must be present as dimensions of the array. For spatial and temporal applications, the space or time dimension is often a good choice of dimension.

- In the case of 2-dimensional arrays common in linear algebra applications, rows represent observations and columns represent variables, factors, or components. Matrix operations such as multiply, covariance, inverse, and best-fit linear equation solution require a 2-dimensional array structure.

In the absence of these factors, choosing to represent values as attributes is generally a good idea. However, SciDB offers the flexibility to transform data from one array definition to another even after it has been loaded. This step is referred to as *redimensioning* the array and is especially useful when the same data set must be used for different types of analytic queries. Redimensioning is used to transform attributes to dimensions and vice-versa.

Once you have created a SciDB database and defined the arrays, you must prepare and load data into it. Loaded data is then available to be accessed and queried using SciDB's built-in analytics capabilities. Unlike traditional relational database environments where complex analytics is separate from data storage and data must be transferred between the two environments, SciDB provides both storage and analytics in a single integrated analytic database.

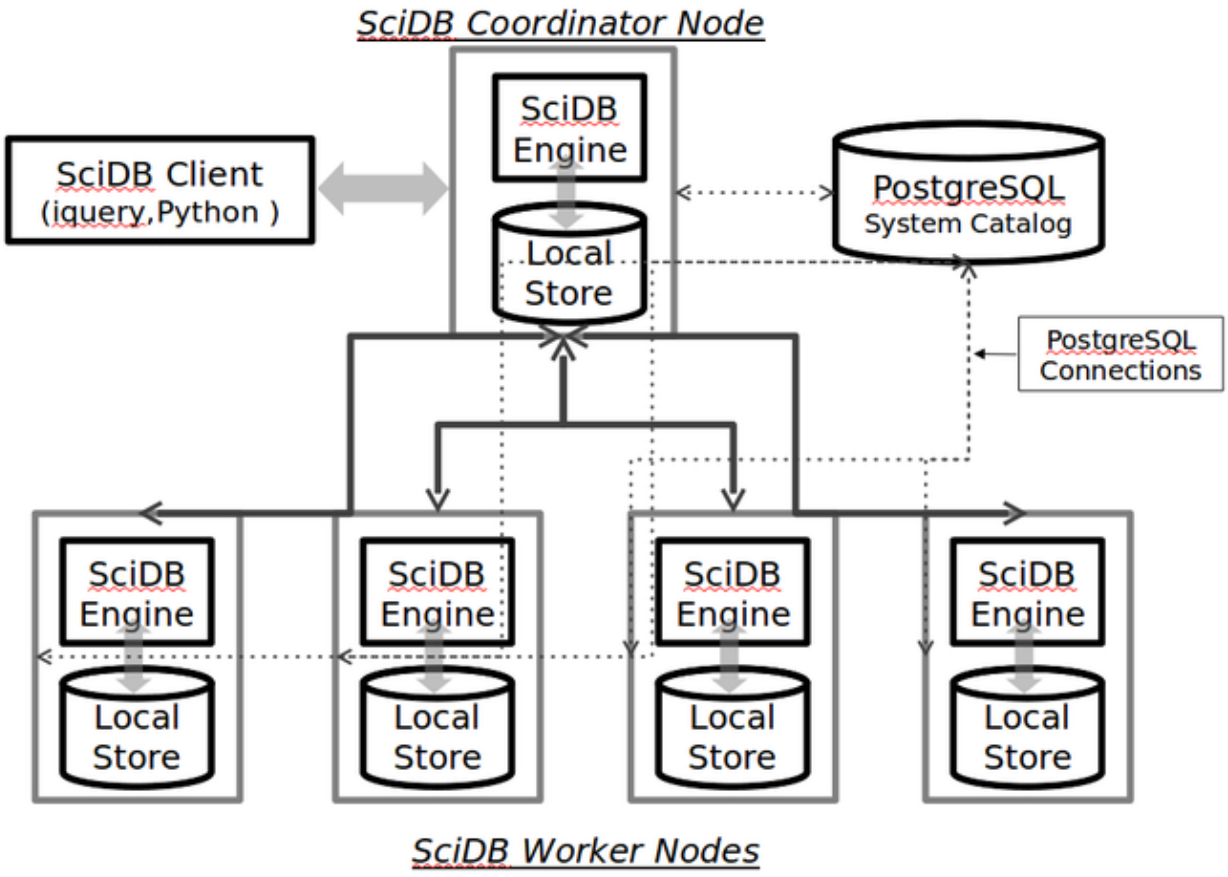
1.2 Basic Architecture

SciDB uses a *shared-nothing* architecture is shown in the illustration below.

SciDB is deployed on a cluster of servers, each with processing, memory, and local storage, interconnected using a standard ethernet and TCP/IP network. Each physical server hosts a SciDB node that is responsible for local storage and processing.

External applications, when they connect to a SciDB database, connect to one of the nodes in the cluster. While all nodes in the SciDB cluster participate in query execution and data storage, one node is the special *coordinator* and orchestrates query execution and result fetching. It is the responsibility of the coordinator node to mediate all communication between the SciDB external client and the entire SciDB database. The rest

of the system nodes are referred to as worker nodes and work on behalf of the coordinator for query processing.



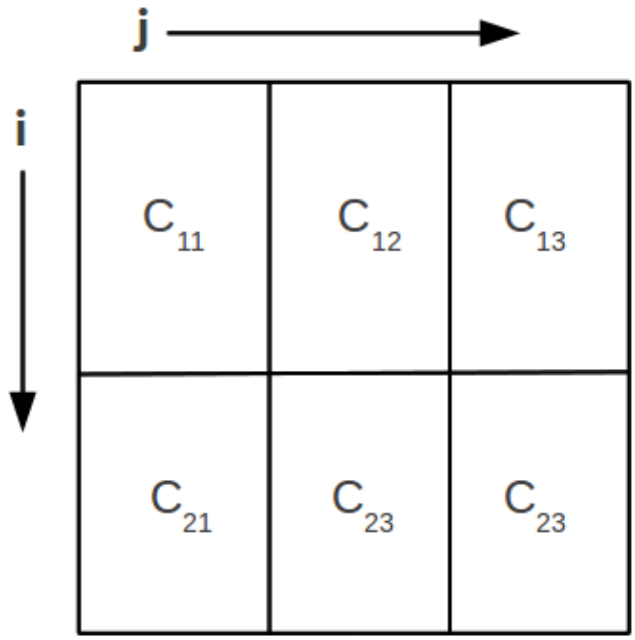
SciDB's scale-out architecture is ideally suited for hardware grids as well as clouds, where additional severs may be added to scale the total capacity.

1.2.1 Chunking and Scalability

When data is loaded, it is partitioned and stored on each node of the SciDB database. SciDB uses *chunking*, a partitioning technique for multi-dimensional arrays where each node is responsible for storing and updating a subset of the array locally, and for executing queries that use the locally stored data. By distributing data uniformly across all nodes, SciDB is able to deliver scalable performance on computationally or I/O intensive analytic operations on very large data sets.

The details of chunking are shown in this section. Remember that you do not need to manage chunk distribution beyond specifying chunk size.

Chunking is specified for each array as follows. Each dimension of an array is divided into chunks. For example, an array with dimensions `i` and `j`, where `i` is of length 10 and chunk size 5 and `j` is of length 30 and chunk size 10 would be chunked as follows:



An array A<a: int32>[i=1:10,5,0, j=1:30,10,0] is chunked as follows. C_{ij} refers to the chunks of the array A as shown below.

C_{11} -> `i`=1:5, `j`=1:10
C_{12} -> `i`=1:5, `j`=10:20
C_{13} -> `i`=1:5, `j`=20:30

C_{21} -> `i`=6:10, `j`=1:10
C_{22} -> `i`=6:10, `j`=10:20
C_{23} -> `i`=6:10, `j`=20:30

Chunks are arranged in row-major order in this example, and stored within the cluster using a round-robin distribution as follows. Suppose a cluster has nodes 1 through 4, the placement of data is shown below.

C_{11} -> node 1
C_{12} -> node 2
C_{13} -> node 3

C_{21} -> node 4
C_{22} -> node 1
C_{23} -> node 2

This scheme is generalized to arrays with more dimensions by arranging the chunks in left-to-right dimension order.

Once you have created a SciDB database and defined the arrays, you must prepare and load data into it. Loaded data is then available to be accessed and queried using the built-in analytics capabilities. Unlike traditional relational database environments where complex analytics is separate from data storage and data must be transferred between the two environments, SciDB provides both in a single integrated analytic database.

1.2.2 Chunk Overlap

It is sometimes advantageous to have neighboring chunks of an array overlap with each other. Overlap is specified for each dimension of an array. For example, consider an array A: follows:

```
A<a: int32>[i=1:10,5,1, j=1:30,10,5]
```

Array has has two dimensions, `i` and `j`. Dimension `i` is of length 10, chunk size 5, and had chunk overlap 1. Dimension `j` has length 30, chunk size 10, and chunk overlap 5. This overlap causes SciDB to store adjoining cells in each dimension from the *overlap area* along with the chunk.

Some advantages of chunk overlap are:

- Speeding up nearest-neighbor queries, where each chunk may need access to a few elements from its neighboring chunks,
- Detecting data clusters or data features that straddle more than one chunk.

SciDB supports operators that can be used to add or change the chunk overlap within an existing array.

1.3 SciDB storage management

1.3.1 Local Storage

Each local node further divides logical chunk of an array into per-attribute chunks, a technique referred to as vertical partitioning. All basic array processing steps -- storage, query processing, and data transfer between nodes -- use single-attribute chunks

SciDB uses run-length encoding internally to compress repeated values or commonly occurring patterns typical in big data applications. Frequently accessed chunks are maintained in the storage manager cache and accelerate query processing by eliminating expensive disk fetches for repeatedly accessed data.

1.3.2 Versions and Delta Encoding

SciDB uses a "no overwrite" storage model. No overwrite means that data is never overwritten; each store() or update query writes a new full chunk or a new *delta chunk*. Delta chunks are calculated by differencing the new version with the prior version and only storing the difference. The SciDB storage manager stores "reverse" deltas -- this means that the most recent version is maintained as a full chunk, and prior versions are maintained as a list or chain of reverse deltas. The delta chain is stored in the "reserve" portion of each chunk, an additional area over and above the total size of the chunk. If the reserve area for the chunk fills up, a new chunk is allocated within the same segment or a new segment and linked into the delta chain.

1.3.3 Automatic Storage Allocation and Reclaim

The local storage manager manages space allocation, placement, and reclaim within the local storage manager using *segments*. A storage segment is a contiguous portion of the storage file reserved for successive chunks of the same array. This is designed to optimize queries issued on a very large array to use sequential disk I/O and hence maximize the rate of data transfer during a query.

Segments also serve as the unit of storage reclaim, so that as array chunks are created, written, and ultimately removed, a segment is reclaimed and reallocated for new chunks or arrays once all its member chunks have been removed. This allows for reuse of storage space.

In addition to the main persistent data store, SciDB also uses temporary data files or "scratch space" used during query execution. This is specified during initialization and start-up as the `tmp-path` configuration setting. Temporary files are managed using the operating system's *tempfile* mechanism. Data written to tempfile only stay for the lifetime of a query. They are removed upon successful completion or abort of the query.

1.4 System Catalog

SciDB relies on a centralized system catalog that is a repository of the following information:

- Array-related metadata such as array definitions, array versions, and associations between arrays and other related objects,
- Cluster configuration information,
- SciDB extensions such as plug-in libraries that encapsulate user-defined types, functions, and operators.

1.5 Array Languages

SciDB provides two query language interfaces.

- AFL, the array functional language, and
- AQL, the array query language

1.5.1 Array Functional Language

AFL represents the full set of data management and analytic capabilities available in the form of operators, ranging from data loading, basic data selection and projection, aggregates, join and merge, multi-dimensional aggregates, *shape changing* operators to transform data from one array definition to another, as well as complex analytics such as matrix and linear algebra.

SciDB Array Functional Language (AFL) adopts a functional programming language syntax over an underlying algebra of array operators. The general form of an AFL statement looks like this:

```
operator_A ( array_inputs)
```

A unary operator accepts one array input. A binary operator accepts two array inputs.

AFL supports combining multiple operators in a single statement or query. An AFL query should be read from the inside out. The first operator executed is the inner-most operator; the output of the innermost operator then becomes the input for the next-innermost operator.

In the query here, the output of `operator_A ( array_name )` is another logical array that becomes the input to the outermost operator, operator_B. The output of B is then returned to the client issuing the query:

```
operator_B ( operator_A ( array_inputs ), constants and expressions )
```

For example, whenever an operator signature uses an *array* input it implies that the input to the operator can be a stored array (using an array identifier) or the result of another operator. For example, SciDB's `list` operator returns an array containing the string-value names of all arrays in the current SciDB instance:

```
list();
[("A"),("target"),("target_new")]
```

SciDB's `concat` operator can take two arrays as input, but it can also take the output of the list operator as input:

```
concat(list(),list());

[("A"),("target"),("target_new"),
("A"),("target"),("target_new")]
```

While *operators* are the main construct in AFL and used to compose queries or statements, the language also supports *types*, *functions*, and *expressions*.

- Types in AFL are analogous to data types in SQL and may be either basic types which are available in SciDB, or as user-defined types, which may be loaded as extensions into SciDB.
- Functions in AFL are scalar functions. An AFL function is a *cell-level* function and takes as input one or more attribute values or expressions over attribute values from an array cell. SciDB supports a number of built-in functions covering a range of arithmetic, logical, string, time, system, and other categories. SciDB also provides a framework for creating and registering user-defined functions, analogous to SQL databases.
- Expressions combine constants, variables (or array attributes), and functions over attribute values. Expressions are used in special operators such as *apply*, *filter*, and *project* .

1.5.2 Array Query Language

SciDB's 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.

The AQL language includes two classes of queries:

- **Data Definition Language** (DDL) : queries to define arrays and load data.
- **Data Manipulation Language** (DML) : queries to access and operate on array data.

The DDL commands in AQL are very similar to AFL, however AQL uses a declarative SQL-style invocation. As of this release, AQL represents a partial subset of SciDB capabilities available today whereas the AFL interface provides the entire set of operators available in SciDB since AFL covers both user and developer operations.

Both AQL and AFL statements are handled by the SciDB query compiler which translates and optimizes incoming statements into an execution plan. The plan is expressed in terms of physical "operators," many of which have direct analogs in AFL.

1.6 Database Concepts

1.6.1 ACID and Transactions

SciDB supports transactions in the database. Each AQL or AFL statement constitutes a single transaction. SciDB supports updates.

A transaction may involve many scan, projection, and analytic operators on one or more arrays, and ultimately store the result in a new or pre-existing array.

SciDB combines traditional ACID semantics with versioned, append-only array storage. When using versioned arrays, write transactions create new versions of the array -- they do not modify pre-existing versions of the array. Such write and read queries are guaranteed to have ACID semantics as defined below:

Atomicity, or "all or none" semantics. A SciDB update transaction once completed will successfully commit all its updates or none at all. Any partial updates will be rolled back. This includes updates to array storage on each node in the cluster as well as the system catalog. A transaction interrupted by user-initiated abort (ctrl+C of a client session), or an error during execution of a query will result in transaction rollback and no change to array storage or catalog.

Consistency. SciDB uses array-level locking. Update transactions such as AQL UPDATE and AFL store and redimension_store transactions hold an exclusive write lock on the array for the entire duration of the transaction. Update transactions create a new version of the array. Read queries are allowed to proceed concurrently with other reads.

Isolation. If multiple update transactions are made to the same array, only one of them is allowed to proceed, and the other transaction will block until the first has completed. Updates to different arrays can proceed concurrently in SciDB.

Durability. When a write or remove transaction (AFL or AQL transactions involving load, store, redimension_store, or remove) completes successfully, its results are committed to stable storage. A transient power failure or reboot of the node does not result in loss of committed data.

1.7 Clients and connectors

The SciDB software package that you downloaded contains a special command line utility called *iquery* which provides an interactive shell and supports both AFL and AQL. For more information about iquery, see Using the iquery Client.

Client applications connect to SciDB using an appropriate connector package which implements the client-side of the SciDB client-server protocol. Once connected via the connector, the user may issue queries written in either AFL or AQL, and fetch the result of the query using an iterator interface.

1.8 ASCII data formats

SciDB uses ASCII formats for loading and saving (or unloading) data. iquery also uses the same ASCII formats to print results retrieved from the server.

Two formats for data sets are allowed. One format is suitable for dense arrays (cases where the vast majority of the elements in the array contain actual values) and one format is suitable for sparse arrays (cases where many of the elements are absent).

Dense array format is shown here. array A<a: int32, b: string>[i=0:2,1,0, j=0:2,3,0]

```
[(10, 'abc'), (9, 'bac'), (8, 'cab')];
[(1, 'xyz'), (2, 'yzx'), (3, 'zyx')];
[(7, 'bbb'), (14, 'bbb'), (21, 'bbb')]
```

Array data is displayed as a sequence of chunks within a flat file. Each chunk must appear within square braces, `[]`. Chunks are separated by a semi-colon, `;`.

Cells within a chunk appear within parentheses `()`, and attributes within the same cell are separated by commas. Whitespace and carriage returns are ignored.

The same array can be represented visually as follows:

10,abc	9,bac	8,cab
1,xyz	2,yzx	3,zyx
7,bbb	14,bbb	21,bbb

The same data in sparse format has the following format:

```
[{0,0} (10,'abc'), {0,1} (9,'bac'), {0,2} (8,'cab')];
[{1,0} (1,'xyz'), {1,1} (2,'yzx'), {1,2} (3,'zyx')];
[{2,0} (7,'bbb'), {2,1} (14,'bbb'), {2,2} (21,'bbb')]
```

In dense format the dimension values are implicit in the ordering of the attributes. In the sparse format, the dimension indices, delimited by curly braces {}, are explicitly included in the ASCII file.

More details on ASCII external file formats are in the chapter Loading Data into a SciDB Array.

1.9 Document Roadmap

The remainder of the chapters in this document are organized as follows.

- Chapter 2, SciDB Installation and Administration, is intended for database and systems administrators who want to offer SciDB as a service to application developers.
- Chapter 3, Getting Started with SciDB Development, describes the `iquery` command-line client for SciDB and helps the beginner-level user get started with SciDB.

- Chapters 4 through 14 cover array-oriented tasks such as creating arrays, loading data, and querying content using SciDB's Array Functional Language (AFL).
- Chapter 15, Array Query Language Reference (AQL), covers SciDB's Array Query Language (AQL).
- Chapter 16, SciDB Plugins: Extending SciDB Functionality, shows how to code user-defined objects as SciDB plugins.
- Chapter 17, Connectors, describes client connectors for SciDB.
- Two appendices offer alphabetical lists of AFL operators and SciDB functions.

1.10 Conventions Used in this Document

Code to be typed in verbatim is shown in `fixed-width font`. Code that is to be replaced with an actual string is shown in *italics*.

Arguments to operators and functions are shown as *argument_name:argument_datatype* where appropriate.

Optional arguments are shown in square brackets [].

AFL queries are available through the SciDB command-line client `iquery`. For this document, all code examples that do not start with `iquery` are written as if the iquery client is running in AFL mode. To set iquery to AFL mode, use the following commands:

```
scidb.py startall hostname
iquery
AQL% set lang afl
```

You can then enter queries at the AFL prompt:

`AFL%` *AFL statement*`;`

All AFL statements end with a semicolon. You can also run code examples in this document using standalone iquery statements as:

`iquery -aq "`*single-line code statement*`"`

Note that the semicolon is not required at the end of a quoted query, though it can be present.

1.11 Where to Get More Help

The SciDB forums (http://www.scidb.org/forum/) are an excellent resource for questions that are not answered in this document. In addition, you can request an invitation to the SciDB Developer's List at http://lists.scidb.org/cgi-bin/mailman/listinfo/dev .

2 SciDB Installation and Administration

2.1 Installing SciDB

SciDB binaries are currently available for the following Linux platforms:

- Red Hat Enterprise Linux 5.4
- Fedora 11
- Ubuntu 11.04

For virtual machine based installs, you can use VMWare Player for desktop testing and Citrix XenServer for production use. You can also use Oracle VirtualBox. VirtualBox installations are recommended only for small-scale testing. For full screen mode with Ubuntu on VirtualBox, install the GuestAdditions package and restart your VirtualBox.

2.1.1 Preparing the Platform

2.1.1.1 Linux User Account

Create a Linux user account, **scidb**. This account must have sudo privileges. It is preferable for it to have NOPASSWD permissions as well.

Modify the `/etc/sudoers` file as follows:

```
## Allow root to run any commands anywhere
root    ALL=(ALL)       ALL
scidb   ALL=(ALL)       NOPASSWD: ALL
```

SciDB is started and stopped as user **scidb**. This user account is also the owner of all processes and files created by SciDB.

2.1.1.2 Postgres Installation and Configuration

SciDB has been tested with Postgres 8.4.X. A suitable version of Postgres (8.4.6 or 8.4.7) is typically available on most Linux platforms.

Install the postgresql-contrib package:

```
sudo apt-get install postgresql-contrib
```

You can use `yum` for Red Hat and Fedora.

By default, Postgres is configured to allow only local access via Unix-domain sockets. In a clustered environment, the Postgres DBMS needs to be configured to allow access from the different nodes in the system. You can fix this by modifying the `pg_hba.conf` file (usually at `/etc/postgresql/8.4/main/` or `/var/lib/pgsql/data/`).

Add the following line to the file:

```
host    all         all         10.0.0.1/8            trust
```

Change all instances of 'ident' to 'trust':

(assuming your local network is 10.x.x.x) and restart Postgres. This configuration might pose security issues, though. You can read _here on the security details to make a more secure installation, by listing specific host IP addresses, user names, and role mappings.

You might also need to set the `postgresql.conf` file to have it listen on the relevant port and ip address, as it might be turned off or limited to `localhost` by default.

If you are running a cluster on multiple servers, you will also need to modify the `postgresql.conf` file to allow connections:

```
# - Connection Settings -
listen_addresses = '*'
```

You can verify that a PostgreSQL instance is running on the coordinator with the `status` command"

```
sudo /etc/init.d/postgresql-8.4 status
sudo /etc/init.d/postgresql-8.4 start
```

Add postgres startup scripts to the Linux init scripts to start Postgres automatically after a reboot.

NOTES:

- Red Hat 5.4 includes support for postgresql 8.1. We recommend upgrading to version 8.4.7.
- Configure Postgres to start automatically during system boot-up. You can do this by modifying the appropriate rc scripts or by using the chkconfig command.
- The python-crypto (64-bit) and python-paramiko packages are required for SciDB on Red Hat 5.4.

2.1.1.3 Remote Execution Configuration (ssh)

SciDB uses ssh for remote execution of cluster management commands. The Linux user account used by SciDB must have password-less ssh access from the coordinator to the workers, as well as from the coordinator to itself.

There are several methods to configure password-less ssh between nodes. We recommend the following simple method.

1. Create key.

   ```
   ssh-keygen
   ```

2. Copy key to each worker node. Also copy key to the localhost (or coordinator).

   ```
   ssh-copy-id scidb@worker

   ssh-copy-id scidb@localhost
   ```

3. Login to remote host. Note that no password is required now.

```
ssh scidb@worker
```

2.1.1.4 Shared file system

To run SciDB in a cluster environment, export /opt/scidb directory on the coordinator node using NFS or samba. Mount this on all worker nodes using the same directory path (/opt/scidb) as the mount point. The coordinator and worker nodes access binaries, shared libraries, plugins, configuration files from /opt/scidb.

2.1.2 Install SciDB from binary package

If you are installing a downloaded pre-built binary package, you can install it using `dpkg` for Ubuntu and `rpm` or `yum` for Red Hat. We currently provide packages for Ubuntu and RPMs for Red Hat and Fedora.

2.1.2.1 Ubuntu

Install

1. First, you should download and install the python-paramiko and python-crypto packages if they haven't been installed before:

   ```
   sudo apt-get install python-paramiko
   sudo apt-get install python-crypto
   ```

2. Next, install the libscidbclient package (and debug symbols, if desired):

```
sudo dpkg  -i libscidbclient-RelWithDebInfo-11.12.0.nnnn-final-Ubuntu-11.04-amd64.deb
(optional) sudo dpkg -i libscidbclient-dbg-RelWithDbgInfo-11.12.0.nnnn-final-Ubuntu-11.04-amd64.de
```

3. Then install the SciDB package (and debug symbols, if desired):

   ```
   sudo dpkg -i scidb-RelWithDebInfo-11.12.0.nnnn-final-Ubuntu-11.04-amd64.deb
   (optional) sudo dpkg -i scidb-dbg-RelWithDebInfo-11.12.0.nnnn-final-Ubuntu-11.04-amd64.deb
   ```

Note: dpkg does not resolve dependencies and you may need to manually install the dependencies or use apt-get to resolve any unmet dependencies on the system. This could happen on either the libscidbclient or SciDB package install. Example:

```
sudo dpkg -i scidb-RelWithDebInfo-11.12....deb    # Fails due to unmet dependencies
sudo apt-get -f install                           # installs missing dependencies
sudo dpkg -i scidb-RelWithDebInfo-11.12....deb    # Succeeds now
```

Uninstall

1. Uninstall the package using this.

   ```
   sudo dpkg -r scidb-dbg
   sudo dpkg -r scidb

   sudo dpkg -r libscidbclient-dbg
   sudo dpkg -r libscidbclient
   ```

2.1.2.2 Red Hat and Fedora

You can use `rpm`, `yum` or other tools to install on Red Hat. First, you need to install the libscidbclient package (and debug symbols, if desired):

Install Install the SciDB client package (debug symbols package is optional)

```
sudo rpm --force -ivh libscidbclient-RelWithDebInfo-11.12*.rpm
(optional) sudo rpm --force -ivh libscidbclient-dbg-RelWithDebInfo-11.12*.rpm
```

Next, install the SciDB server package (and debug symbols, if desired).

```
sudo rpm --force -ivh scidb-11.12*.rpm
(optional) sudo rpm --force -ivh scidb-dbg-*.rpm
```

Uninstall

```
sudo rpm -e scidb-dbg
sudo rpm -e scidb

sudo rpm -e libscidbclient-dbg
sudo rpm -e libscidbclient
```

2.1.2.3 Environment Variables

To start and run SciDB, configure the environment of the user account of the SciDB process (we suggest creating a special user `scidb` for this purpose). The following lines should be added to the user's shell configuration file (often `.profile` or `.bashrc`):

```
export SCIDB_VER=11.12
export PATH=/opt/scidb/$SCIDB_VER/bin:/opt/scidb/$SCIDB_VER/share/scidb:$PATH
export LD_LIBRARY_PATH=/opt/scidb/$SCIDB_VER/lib:$LD_LIBRARY_PATH
```

2.1.3 Build from source

If you have chosen to start from sources, please follow the directions in this section. If you have successfully installed SciDB from a binary distribution, you can skip to the next section.

The following software packages are required to build SciDB from sources.

- cmake (2.8.3 or newer)
- boost (1.42. Newer boost versions are not currently supported.)
- protobuf6
- libpqxx (3.0 or higher)
- flex (2.5.35 or newer)
- bison (2.4 or higher)
- log4cxx
- apr
- apr-util
- cppunit
- readline 6
- bz2-dev

- swig (2.0 or higher)
- paramiko (for Python)
- crypto (for Python)
- subversion
- doxygen (optional)

On Ubuntu, install these packages using:

```
sudo apt-get update

sudo apt-get install -y build-essential cmake libboost1.42-all-dev \
postgresql-8.4 libpqxx-3.0 libpqxx3-dev libprotobuf6 libprotobuf-dev \
protobuf-compiler doxygen flex bison  \
liblog4cxx10 liblog4cxx10-dev libcppunit-1.12-1 libcppunit-dev \
libbz2-dev postgresql-contrib  \
subversion libreadline6-dev libreadline6 \
python-paramiko python-crypto
```

Download, build and install swig from _ sources.

Download the latest scidb source package.

```
wget https://p4-releases.s3.amazonaws.com/scidb-11.12.0.xxxx.tgz
tar xzf scidb...tgz
cd scidb-11.12

cmake .
make
sudo make install
```

To start and run SciDB, you need to configure the environment of the user who will run the SciDB instances as follows. The following lines should be added to your shell's configuration file. For example if your shell is Unix bash, this goes in the .bashrc file, or the .bash_profile:

```
export SCIDB_VER=11.12
export PATH=/opt/scidb/$SCIDB_VER/bin:/opt/scidb/$SCIDB_VER/share/scidb:$PATH
export LD_LIBRARY_PATH=/opt/scidb/$SCIDB_VER/lib:$LD_LIBRARY_PATH
```

2.2 Configuring SciDB

This chapter demonstrates how to configure SciDB prior to initialization, including checking that the PostgreSQL DBMS is running, that the SciDB configuration file (usually `/opt/scidb/11.12/etc/config.ini`) is set up, and that logging is configured.

2.2.1 SciDB Configuration File

You need to create a configuration file for SciDB. By default, it is named `config.ini` and it resides in the `etc` sub-directory of the installation tree. (By default it is `/opt/scidb/11.12/etc/config.ini`.) The configuration file can have multiple sections, one per service instance.

The configuration 'test1' below is an example of the configuration for a single node, single instance system (coordinator only).

```
[test1]
node-0=localhost,0
db_user=test1user
db_passwd=test1passwd
install_root=/opt/scidb/11.12
metadata=/opt/scidb/11.12/share/scidb/meta.sql
pluginsdir=/opt/scidb/11.12/lib/scidb/plugins
logconf=/opt/scidb/11.12/share/scidb/log4cxx.properties
base-path=/home/scidb/data
base-port=1239
interface=eth0
no-watchdog=true
redundancy=2
merge-sort-buffer=1024
network-buffer=1024
mem-array-threshold=1024
smgr-cache-size=1024
execution-threads=16
result-prefetch-queue-size=4
result-prefetch-threads=4
chunk-segment-size=10485760
```

2.2.2 Cluster Configuration Example

The following example of a cluster configuration is called 'monolith'. This cluster consists of eight systems/nodes of equivalent specifications and running the same version of Red Hat Enterprise Linux:

- Intel Xeon X5660 2.8GHz six-core processor
- 4GB of RAM
- 1TB 6.0Gb/sec SATA hard drive
- 1Gbps network connectivity
- RHEL 5.4

In this example, the 'monolith' cluster is running one instance of SciDB on each physical node. Each node is specified by its IP address and the number of 'worker' instances to be run on that node. A node is essentially a physical machine and an instance refers to SciDB workers/processes on each system. The coordinator is a special instance of worker node; in addition to its duties of a worker, it performs additional cluster management and coordination duties. The coordinator node is not counted as a 'worker' in the `config.ini` and it is assumed to always be on node-0.

We have also specified some of the most common configuration options such as where data will be stored on disk (base_path), which port to use (base-port), which network interface SciDB should use (interface), etc.

```
[monolith]
# node-id=IP, number of worker nodes
node-0=10.0.20.231,0
node-1=10.0.20.232,1
node-2=10.0.20.233,1
node-3=10.0.20.234,1
node-4=10.0.20.235,1
node-5=10.0.20.236,1
node-6=10.0.20.237,1
node-7=10.0.20.238,1
db_user=monolith
db_passwd=monolith
install_root=/opt/scidb/11.12
metadata=/opt/scidb/11.12/
```

```
share/scidb/meta.sql
pluginsdir=/opt/scidb/11.12/lib/scidb/plugins
logconf=/opt/scidb/log4cxx.properties.trace
base-path=/data/monolith_data
base-port=1239
interface=eth0
```

The install package contains a sample configuration file, `sample_config.ini`, which is used for SciDB testing. The sample file is an example of a configuration file containing multiple configurations, one for a single instance and one for a dual instance, but on the same node.

The following table describes the basic configuration file contents and how to set them:

Basic Configuration	
Key	**Value**
cluster name	Name of the SciDB cluster. The cluster name must appear as a section heading in the config.ini file, e.g., *[cluster1]*
node-N	The host name or IP address used by node N and the number of worker instances on it. Node 0 always has the coordinator node running as instance 0, and may have additional worker instances running as well.
db_user	Username to use in the catalog connection string. This example uses *test1user*
db_passwd	Password to use in the catalog connection string. This example uses *test1passwd*
install_root	Path name of install root. Must be the same on all nodes. When using a multi-node environment, configuring this using NFS will reduce the number of installations needed.
metadata	Metadata definition file. The recommended NFS configuration makes this visible under the same path name on master and worker processes.
pluginsdir	The folder or directory in which plugins are stored. Must be visible under the same path name to all workers.
logconf	Config file for SciDB log. Edit this to set a different filename or log level (default file name is *INFO* and default file name is scidb.log.

The following table describes the cluster configuration file contents and how to set them:

Cluster Configuration	
Key	**Value**
base-path	The root data directory for each SciDB instance. Note that this directory will be the same for all nodes (all instances read from the same config.ini). When each SciDB node is initialized, it creates it's own sub-directories for it's local data. E.g., if the base is `/scidb`, then `/scidb/000/0` will hold the data and logs for the coordinator, while `/scidb/001/1` will hold data for the first worker instance on the first worker node.
base-port	base port number. Connections to the coordinator (and therefore to the system) are via this number, while worker instances communicate at base-port + instance number. The default number that `iquery` expects is 1239.
interface	Ethernet interface used for SciDB used on all nodes - master and workers. Used to bind SciDB to the correct local interface.
ssh-port	(optional) the port which ssh uses for communications within the cluster. The default value is 22.
key-file-list	(optional) a comma-separated list of filenames that include keys for ssh authentication.
tmp-path	(optional) a directory to use as temporary space.

no-watchdog	default=false, Set this to true if you do not want automatic restart of the SciDB node on software crash.

The following table describes the configuration file elements for tuning your system performance:

Performance Configuration	
Key	**Value**
save-ram	(optional) 'True', 'true', 'on' or 'On' will enable this option. Off by default.
merge-sort-buffer	(optional) Size of memory buffer used in merge sort. Default is 512 MB.
mem-array-threshold	(optional) Memory footprint for temporary arrays. Default is 1024 MB.
chunk-reserve	Percentage of chunk preallocated to store chunk deltas. Default is 10%. Setting this parameter to 0 disables the delta mechanism.
chunk-segment-size	Size in bytes of a storage segment. A storage segment is a unit of allocation and reclaim used by storage manager. (1) Place chunks belonging to the same array within the same segment for sequential reads, (2) Reclaim space from removed arrays. If set to zero, then segments are not used and storage is allocated chunk at a time, and space will not be reclaimed.
execution-threads	Size of thread pool available for query execution. Shared pool of threads used by all queries for network IO and some query execution tasks. Default is 4.
result-prefetch-threads	Per-query number of result chunks to prefetch. Default is 4.
result-prefetch-queue-size	Per-query number of result chunks to prefetch. Default is 4.
smgr-cache-size	(optional) Size of buffer cache. Default is 256 MB

In our example shown above, the db_user field is set to *test1user* and db_passwd is set to *test1passwd*. Due to a current limitation in the system, use of the same user name with different cluster configurations might cause problems, so it is preferable to generate unique user names for each cluster configuration. These are Postgres user names, not system users. The Postgres database created is the same as the section header, *test1* .

2.2.3 Logging Configuration

SciDB uses Apache's log4cxx (http://logging.apache.org/log4cxx/) for logging.

The logging configuration file, specified by the `logconf` variable in `config.ini`, contains the following Apache log4cxx logger settings:

```
###
# Levels: TRACE < DEBUG < INFO < WARN < ERROR < FATAL
###
log4j.rootLogger=DEBUG, file

log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=scidb.log
log4j.appender.file.MaxFileSize=10000KB
log4j.appender.file.MaxBackupIndex=2
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d [%t] [%-5p]: %m%n
```

2.2.4 System Catalog Initialization

This step is required only if user `scidb` does **not** have *sudo* privileges. If this is the case, ask your system administrator to run this script as the `postgres` user:

```
sudo -u postgres /opt/scidb/11.12/bin/scidb-prepare-db.sh
```

This script does the following:

1. creates a new role or account (say *test1user*) with password (say *test1passwd*)
2. creates a database for testing scidb (say *test1*) using the role created in the first step
3. creates a schema in that newly created Postgres database to hold the SciDB catalog data.

2.3 Initializing and Starting SciDB

2.3.1 The scidb.py Script

To begin a SciDB session, use the `scidb.py` script. In a standard SciDB build, this script is located at:

`/opt/scidb/`*version.number*`/bin`

The syntax for the `scidb.py` script is:

`scidb.py` *command db conffile*

The options for the *command* argument are:

initall	Initialize the Postgres database. **Warning**: This will remove any existing SciDB arrays from the current namespace.
startall	Start a SciDB instance
stopall	Stop the current SciDB instance
status	Show the status of the current SciDB instance
dbginfo	Collect debugging information by getting all logs, cores, and install files
dbginfo-lt	Show stack and log information from debug
version	Show SciDB version number

The `db` argument is the name of the SciDB instance you want to create.

The configuration file is set by default to `/opt/scidb/11.12/etc/config.ini`. If you want to use a custom configuration file for a particular SciDB instance, use the `conffile` argument.

Run the following command to initialize SciDB on all nodes. If the SciDB user has sudo privileges, everything will be done automatically (otherwise see the previous section for additional Postgres configuration steps):

```
scidb.py initall test1
```

Note: This will reinitialize the SciDB database. Any arrays that you have created in previous SciDB sessions will be removed and the memory recovered.

To start the set of local SciDB instances specified in your config.ini file, use the following command:

```
scidb.py startall test1
```

This will report the status of the various nodes:

```
scidb.py status test1
```

and this will shut it down:

```
scidb.py stopall test1
```

SciDB logs are written to the file `scidb.log` in the appropriate directories for each instance: <base-path>/000/0 for the coordinator and <base-path>/M/N the worker node M instance N.

2.4 Upgrading SciDB

Note: test1 in the following examples refers to the name of the SciDB database. All of the following steps are performed as Linux user **scidb**.

- Shutdown SciDB on all nodes in the cluster.

  ```
  scidb.py stopall test1
  ```

- Download and install the latest SciDB package using the standard package manager on your platform (rpm or dpkg).

  If you are installing a downloaded pre-built binary package, you can install it using `dpkg` for Ubuntu and `rpm` or `yum` for Red Hat. We currently provide packages for Ubuntu and RPMs for Red Hat and Fedora.

2.4.1 Ubuntu

1. First, install the libscidbclient package (and debug symbols, if desired):

   ```
   sudo dpkg -i libscidbclient-RelWithDbgInfo-11-12*.deb
   (optional) sudo dpkg -i libscidbclient-dbg-RelWithDbgInfo-11-12*.deb
   ```

2. Then install the SciDB package (and debug symbols, if desired):

   ```
   sudo dpkg -i scidb-RelWithDebInfo-11.12-*.deb
   (optional) sudo dpkg -i scidb-dbg-RelWithDebInfo-11.12-*.deb
   ```

2.4.2 Red Hat and Fedora

1. First, you need to install the libscidbclient package (and debug symbols, if desired):

   ```
   sudo rpm --force -Uvh libscidbclient-RelWithDebInfo-11.12*.rpm
   (optional) sudo rpm --force -Uvh libscidbclient-dbg-RelWithDebInfo-11.12*.rpm
   ```

2. Next, install the SciDB server package (and debug symbols, if desired).

   ```
   sudo rpm --force -Uvh scidb-11.12*.rpm
   (optional) sudo rpm --force -Uvh scidb-dbg-*.rpm
   ```

- Copy over the previous config.ini from your earlier version.

```
cp /opt/scidb/11.12/etc/config.ini /opt/scidb/11.12/etc/config.ini
```

2.4.3 Additional Steps

- Modify the config.ini file that you just copied. Change all references to your previous version to the new version (ex: install_root=/opt/scidb/11.12)

- Edit your environment and update PATH and LD_LIBRARY_PATH.

```
export SCIDB_VER=11.12
export PATH=/opt/scidb/$SCIDB_VER/bin:/opt/scidb/$SCIDB_VER/share/scidb:$PATH
export LD_LIBRARY_PATH=/opt/scidb/$SCIDB_VER/lib:$LD_LIBRARY_PATH
```

- **NOTE:** SciDB 11.12 does not accept storage files from earlier versions. You must re-initialize and re-load data.

```
which scidb.py # Make sure you are running 11.12
scidb.py initall test1
scidb.py startall test1
scidb.py status test1
```

3 Getting Started with SciDB Development

3.1 Using the iquery Client

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. Start the `iquery` client by typing `iquery` at the command line when a SciDB session is active:

```
scidb.py startall scidb_server
iquery
```

By default, `iquery` opens an AQL command prompt:

```
AQL%
```

You can then enter AQL queries at the command prompt. To switch to AFL queries, use the set command:

```
AQL% set lang afl;
```

AQL statements end with a semicolon (;).

To see the internal `iquery` commands reference type `help` at the prompt:

```
AQL% help;
set             - List current options
set lang afl    - Set AFL as querying language
set lang aql    - Set AQL as querying language
set fetch       - Start retrieving query results
set no fetch    - Stop retrieving query results
set timer       - Start reporting query setup time
set no timer    - Stop reporting query setup time
set verbose     - Start reporting details from engine
set no verbose - Stop reporting details from engine
quit or exit    - End iquery session
```

You can pass an AQL query directly to `iquery` from the command line using the -q flag:

```
iquery -q "my AQL statement"
```

You can also pass a file containing an AQL query to `iquery` with the -f flag:

```
iquery -f my_input_filename
```

AQL is the default language for `iquery`. To switch to AFL, use the -a flag:

```
iquery -aq "my AFL statement"
```

Each invocation of `iquery` connects to the SciDB coordinator node, passes in a query, and prints out the coordinator node's response. `iquery` connects by default to SciDB on port 1239. If you use a port number that is not the default, specify it using the "-p" option with `iquery`. For example, to use port 9999 to run an AFL query contained in the file `my_filename`

```
iquery -af my_input_filename -p 9999
```

The query result will be printed to stdout. Use -r flag to redirect the output to a file:

```
iquery -r my_output_filename -af my_input_filename
```

By default, the output will be formatted as a SciDB array. The output array may be formatted as sparse, in which case the block number for a cell will be printed, or dense, in which case the attributes will be printed.

To change the output format, use the -o flag:

```
iquery -o csv -r my_output_filename.csv -af my_input_filename
```

Available options for output format are csv, csv+, lcsv, lcsv+, sparse, and lsparse.

To see a list of the `iquery` switches and their descriptions, use `iquery -h` or `iquery --help`:

```
iquery -h
Available options::
  -c [ --host ] arg       Host of one of the cluster nodes. Default is
                          'localhost'.
  -p [ --port ] arg       Port for connection. Default is 1239.
  -q [ --query ] arg      Query to be executed.
  -f [ --query-file ] arg File with query to be executed.
  -r [ --result ] arg     Filename with result array data.
  -o [ --format ] arg     Output format: auto, csv, csv+, lcsv+, sparse,
                          lsparse. Default is 'auto'.
  -v [ --verbose ]        Print debug info. Disabled by default.
  -t [ --timer ]          Query setup time.
  -n [ --no-fetch ]       Skip data fetching. Disabled by default.
  -a [ --afl ]            Switch to AFL query language mode. AQL by default.
  -u [ --plugins ] arg    Path to the plugins directory.
  -h [ --help ]           Show help.
  -V [ --version ]        Show version info.
  --ignore-errors         Ignore execution errors in batch mode.
```

The `iquery` interface is case sensitive.

Example:

- Create an array called random_numbers with:
  - ♦ 2 dimensions, x = 9 and y = 10
  - ♦ one double attribute called num
  - ♦ random numerical values in each cell

```
iquery -aq "store(build(<num:double>[x=0:8,1,0, y=0:9,1,0], random()),random_numbers)"
```

- Save the numerical values in random_numbers in csv format to a file called /tmp/random_values.csv:

```
iquery -o csv -r /tmp/random_values.csv -aq "scan(random_numbers)"
```

3.2 iquery Configuration

You can use a configuration file to save and restore your `iquery` configuration. The file is stored in ~/.config/scidb/iquery.conf. Once you have created this file it will load automatically the next time you start `iquery`. The allowed options are:

host	Host name for the cluster node. Default is `localhost`.
port	Port for connection. Default is 1239.
afl	Start AFL command line
timer	Report query run-time (in seconds).
verbose	Print debug information.
format	Set the format of query output. Options are csv, csv+, lcsv, lcsv+, sparse, and lsparse.
plugins	Path to the plugins directory

For example, your iquery.conf file might look like this:

```
{
"host":"mynodename",
"port":9999,
"afl":true,
"timer":false,
"verbose":false,
"format":"csv+",
"plugins":"./plugins"
}
```

The opening and closing braces at the beginning and end of the file must be present and each entry (except the last one) should be followed by a comma.

3.3 Example iquery session

This example demonstrates how to use iquery to perform simple array tasks:

- Create a SciDB array
- Prepare an ASCII load file in the SciDB *dense* load file format
- Load data from that file into the array.
- Execute basic queries on the array
- Join two arrays containing related data

1. Create an array, `target`, in which you are going to place the values from the csv file:

```
iquery -aq "create array target <name:string, mpg:double>[x=0:*,1,0]"
```

2. Starting from a csv file, prepare a SciDB load file.

datafile.csv:

```
Model,MPG
CRV, 23.5
Prius, 48.7
Explorer, 19.6
Q5, 26.8
```

Convert the file to SciDB format with the command csv2scidb:

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

Note: `csv2scidb` is a separate data preparation utility provided with SciDB. To see all options available for csv2scidb, type `csv2scidb --help` at the command line.

3. Use the load command to load the SciDB-formatted file you just created into `target`:

```
iquery -aq "load(target, '/tmp/datafile.scidb')"
[("CRV",23.5),("Prius",48.7),("Explorer",19.6),("Q5",26.8)]
```

By default, iquery always re-reads or retrieves the data that has just written to the array. To suppress the print to screen when you use the load command, use the -n flag in iquery:

```
iquery -naq "load(target, '/tmp/datafile.scidb')"
```

4. Now, suppose you want to convert miles-per-gallon to kilometers per liter. Use the apply function to perform a calculation on the attribute values `mpg`:

```
iquery -aq "apply(target,kpl,mpg*.4251)"
[("CRV",23.5,9.98985),("Prius",48.7,20.7024),("Explorer",19.6,8.33196),("Q5",26.8,11.3927)]
```

Note that this does not update `target`. Instead, SciDB creates an result array with the new calculated attribute `kpl`. To create an array containing the kpl attribute, use the `store` command:

```
iquery -aq "store(apply(target,kpl,mpg*.4251),target_new)"
```

5. Suppose you have a related data file, datafile_price.csv:

```
Make,Model,Price
Honda,CRV,26700
Toyota,Prius,31000
Ford, Explorer,42000
Audi,Q5,45000
```

You want to add the data on price and make to your array. Use csv2scidb to convert the file to SciDB data format:

```
csv2scidb -p SSN -s 1 < /tmp/datafile_price.csv > /tmp/datafile_price.scidb
```

Create an array called storage:

```
iquery -aq "create array storage <make:string, model:string, price:int64>[x=0:*,1,0]"
```

Load the datafile_price.scidb file into storage:

```
iquery -naq "load(storage, '/tmp/datafile_price.scidb')"
```

6. Now, you want to combine the data in these two files so that each entry has a make, and model, a price, an mpg, and a kpl. You can join the arrays, with the `join` operator:

```
iquery -aq "join(storage,target_new)"
[("Honda","CRV",26700,"CRV",23.5,9.98985),
```

```
("Toyota","Prius",31000,"Prius",48.7,20.7024),
("Ford"," Explorer",42000,"Explorer",19.6,8.33196),
("Audi","Q5",45000,"Q5",26.8,11.3927)]
```

Note that attributes 2 and 4 are identical. Before you store the combined data in an array, you want to get rid of duplicated data.

7. You can use the project operator to specify attributes in a specific order:

```
iquery -aq project(target_new,mpg,kpl)
[(23.5,9.98985),(48.7,20.7024),(19.6,8.33196),(26.8,11.3927)]
```

Attributes that are not specified are not included in the output.

Now use the `join` and `project` operators to put the car data together. For easier reading, use csv as the query output format:

```
iquery -o csv -aq "join(storage,project(target_new,mpg,kpl))"
make,model,price,mpg,kpl
"Honda","CRV",26700,23.5,9.98985
"Toyota","Prius",31000,48.7,20.7024
"Ford"," Explorer",42000,19.6,8.33196
"Audi","Q5",45000,26.8,11.3927
```

---

4 Creating and Removing SciDB Arrays

SciDB stores data as a collection of chunked multi-dimensional nested arrays. Just as a relational table is the basic data structure of relational algebra and SQL, SciDB uses multi-dimensional arrays as the basis for linear algebra and complex analytics.

A SciDB database is organized into arrays that have:

- A *name*. Each array in a SciDB database has an identifier that distinguishes it from all other arrays in the same database.
- A *schema*, which is the array structure. The schema contains array dimensions and attributes.
- We use the term *shape* in reference to an array's collection of dimensions. To have the same or compatible shapes, two arrays must have the same number and order of dimensions and the corresponding dimensions in the two arrays must have the same type and size.
- Each *dimension* consists of a list of index values. At the most basic level the dimension of an array is represented using 64-bit unsigned integers. The number of index values in a dimension is referred to as the dimension's size (or occasionally its extent). SciDB support arrays with noninteger dimensions by mapping noninteger values into the basic unsigned 64-bit integer dimension type.
- *Cell* or *element*. Each combination of dimension values identifies a single element or cell in the array. Each cell may be empty or is occupied by one or more attributes.
- *Attribute*. Each cell contains a list of named, typed attributes.

4.1 Creating and removing arrays

Arrays are created and removed using the *create array* and *remove* commands described below.

4.2 create array

Summary: Create a SciDB array

The CREATE ARRAY statement is used to a create new array. The statement specifies the array name and the array schema. The array schema is a description of the array properties which includes the array *shape* (the number of dimensions and their sizes), and the array *attributes* (the data items that appear in each cell).

Signature:

```
CREATE [array_type] ARRAY
    array_name
    < attribute_name : type_name [ NULL | NOT NULL ] [, ...] >
    [ dimension_name = start: end|*, chunk_size, chunk_overlap [, ...] ]
```

SciDB arrays have the following elements.

Array Feature	Optional	Description

array_name	No	The string name of the array. The array name uniquely identifies the array in the SciDB instance. You cannot use the same array name twice in one SciDB instance. Array names should not contain the characters `@` or `:` as these characters are reserved for mapping arrays.
EMPTY flag	Yes	By default, all arrays have all cells present. Users can optionally specify 'EMPTY', in which case some cells may be omitted.
attribute_name	No	Name of an attribute. No two attributes in the same array can share a name.
type_name	No	Type identifier. One of the data types supported by SciDB. Use the `list('types')` command to see the list of available data types.
NULL flag	Yes	By default, all attributes are 'NOT NULL', i.e. they will have a value. Optionally, users can specify 'NULL' to indicate attributes that are allowed to contain null values.
dimension_name	No	Each dimension has a name. Just like attributes, each dimension must be named, and dimension names cannot be repeated in the same array
start (integer)	No	The starting coordinate of a dimension
end (integer)	No	The ending coordinate of a dimension, or * if unknown
chunk_size	No	The length of the data chunk along a dimension.
chunk_overlap	No	The length of overlap along a dimension. Overlap specifies the extended chunk, including additional cells from adjoining chunks in the array that are colocated with a given chunk. Like chunk sizes, overlap is an internal storage management concept, and does not change the result of an operator. In this release, some operators rely on overlap, as described later in this document.

4.3 Basic examples

This sections contains several examples of creating arrays using the CREATE ARRAY statement.

Example One:

Create a 2-dimensional array that has:

1. Dimension names 'x' and 'y',
2. Equal length dimensions ranging between 0 and 49 (having length 50)
3. Chunks each of size 10 in each dimension
4. No chunk overlap
5. One integer attribute named 'Val'

```
CREATE ARRAY Example_One < Val: int32 > [ X=0:49,10,0, Y=0:49,10,0]
```

Example Two:

Create a 3-dimensional array with dimension indices

1. having names X, Y and Z,
2. with values for the X and Y dimensions ranging between 0 and 9 (length 10) and the Z dimension ranging over 0:99 (length = 100)
3. stored entirely within a single chunk of size 10x10x100.
4. with no overlap.

and two attributes A and B being a string and a double-precision value:

```
CREATE ARRAY Example_Three < A: string, B: double > [ X=0:9,10,0, Y=0:9,10,0, Z=0:99,100,0 ]
```

Example Three:

The following is a 2-dimensional array with dimensions

1. having names I and J, with I ranging over 0 to 99 and J over 0 to 199, and
2. broken into *chunks* of size 10x20

The array's cells contain a pair of attribute values, A and B, having types int32 and float.

```
CREATE ARRAY Example_Four <A:int32, B:float> [I=0:99,10,0, J=0:199,20,0]
```

Example Four:

The following is a 2-dimensional array with dimensions

1. having names I and J, with I ranging over 0 to 9 and J over 0 to 9, and
2. fitting into a single *chunk* of size 10x10

Our example has two attributes, A and B, having types int32 and double.

```
CREATE ARRAY Example_Five < A: int32, B: double > [ I=0:9,10,0, J=0:9,10,0 ]
```

4.4 show

Summary: The show operator can be used to review the schema of an array that is stored within SciDB.

Signature:

```
show( array )
```

Example:

```
create empty array A <a:int32> [x=0:2,3,0];

show(A);
[("A<a:int32 NOT NULL,empty_indicator:indicator NOT NULL> [x=0:2,3,0]")]
```

4.5 Empty arrays

The EMPTY keyword in the CREATE statement indicates that some cells may be absent.

```
CREATE EMPTY ARRAY A_10 <x:double null, y:double null>[i=1:10,5,0]
```

Note that EMPTY is fundamentally different than NULL. EMPTY is a property of an entire cell and indicates that the entire cell has been omitted, i.e. is not present. In contrast, NULL is a property of a cell value. A cell that contains one or more null values is not considered empty.

If the array is not designated EMPTY at creation time, then unspecified cells within existing chunks assume the default value (which in the current version is 0 for numerical attributes and empty string for string and character attributes). However, chunks that are missing entirely are always treated as regions of empty cells.

4.6 Dimensions

4.6.1 Unbounded Dimensions

An array dimension can be created as an unbounded dimension by declaring the high boundary as open using '*'. Examples of unbounded dimensions include *I=0:*,10,0* or *J=-7:*,100,0.* In the first example, *I* has a low boundary of 0 and an open high boundary and *J* has a low boundary of -7 and an open high boundary.

A regular array does not allow chunks to be loaded into chunk addresses that fall outside the region defined by the create array definition. However, with an array having unbounded dimensions, chunks can be incrementally loaded into the array into new chunks of the array and there is no limit on the array dimensions, other than storage resources available in the system. The array boundaries are dynamically updated as new data is added to it.

```
CREATE ARRAY open_array <a:int64>[x=0:*,5,0]
```

An example of incremental loading into an unbounded array is described in the *load* section. Loading data into an unbounded array causes new array versions to be created.

4.6.2 noninteger dimensions and Mapping arrays

Regular arrays in SciDB use the int64 data type for dimensions. SciDB also supports arrays with noninteger dimensions. These arrays map dimension *values* of a declared type to an internal int64 array *position*. Mapping is done through special mapping arrays internal to SciDB.

Below is an example of an array with a noninteger dimension:

```
CREATE ARRAY non_int_array <a:int64>[ID(string)=10,5,0]
```

This command will create an array with a noninteger dimension, named ID, that has 10 unique string dimension values mapped internally to positions 0,.., 9, and uses chunks of 5 elements each, with no overlap.

The most common way to store data into such an array is to use redimension_store. This AFL command is described in detail in a later chapter.

However, any output array that has a schema compatible with this array can be stored into it.

```
CREATE ARRAY another_non_int_array <a:int64>[ID(string)=10,5,0]
store(another_non_int_array,non_int_array)
```

SciDB stores the mapping from value to position for these special arrays (with noninteger dimensions) using a mapping array. The name of the mapping array is "array@ID:dimname". This mapping array is used by SciDB array operators to translate from dimension value to an integer dimension position in the array. The mapping array is fully replicated at every node in a cluster SciDB configuration.

For example, when data is stored into the above array, we see the following new mapping array created for it. This array maps each string *identifier* to a corresponding integer array *position.*

```
 show("non_int_array@1:ID");
[("non_int_array@1:ID<value:string NOT NULL> [no=0:9,10,0]")]
```

If the base array is created as a standard versioned array, the corresponding mapping arrays are also versioned. Hence, the mapping array shown above corresponds to array version 1, and array version 2 of *non_int_array* has mapping array *non_int_array@2:ID* .

4.7 remove

Summary: Remove an array from a SciDB instance

Syntax:
`remove(`*array_name*`);`

The remove command removes a SciDB array and all of its versions from the SciDB namespace. The remove operation is preserved even if there is a node failure. This means that if a remove operation throws an error, the array will still be removed.

---

5 Loading Data into a SciDB Array

The load, save, input and substitute commands are used while loading and unloading data from SciDB into external load files in ASCII format.

5.1 load

SciDB supports the load operator for loading data into an existing SciDB array.

Signature:

load (*array_name : array_identifier*, *data_file : string* [, *nodeid : int*])

array_name is the array to be loaded. *data_file* is the path to a file to use -- absolute or relative to the working directory of the SciDB server. For a description of *nodeid*, see below section on parallel load.

SciDB data load files must be organized according to the schema of the target array.

Given an array defined as follows:

```
CREATE ARRAY Load_Example < Val: int32 > [X=1:100,25,0, Y=1:100,25,0]
```

Load files divide the data to be loaded into chunks. Load file chunks correspond to array chunks as defined by the CREATE ARRAY statement. So, for the example 2-dimensional array with the following chunk layout

```
C11 C12 C13 C14
C21 C22 C23 C24
C31 C32 C33 C34
C41 C42 C43 C44
```

The *chunks* in the load file must appear in the following order:

```
C11; C12; C13; C14; C21; C22; ...; C41; C42; C43; C44
```

Each chunk may also be prefixed with an optional chunk header that lists the dimension values of the starting element in that chunk. If this chunk header is not present, the file is assumed to contain all of the chunks in the order described above.

5.1.1 Load formats

SciDB supports two chunk formats in load files corresponding to *dense* and *sparse* data sets. The sparse format is more efficient when a majority of the cells in the array are absent and do not contain attribute data. A SciDB array loaded from a data file that has the sparse load format generates sparse chunks in array storage, whereas one loaded from a dense file format creates chunks that use the dense storage format.

5.1.2 Dense load format

In the following example we illustrate how a multi-chunk, dense array is created and loaded.

```
CREATE ARRAY Two_Dim<a: int32, c: char>[I=0:7,4,0, J=0:7,4,0]
load(Two_Dim, '/tmp/2d-mc.txt')
```

In the dense representation, the dimension values for each cell are implicit in the representation. Data is divided up into chunks, with each chunk enclosed within a '[ ]' and separated by a semi-colon.

Cells within each chunk must appear in *left to right* dimension order (e.g., row-major order for a two-dimensional array, or generalized appropriately to higher dimensions). Each cell contains a comma-separated list of attribute values placed within (). This 'dense' representation can denote empty cells using '()'.

Contents of /tmp/2d-mc.txt:

```
[
[ (0, 'A'), (1, 'B'), (2, 'C'), (3, 'D')],
[ (8, 'I'), (9, 'J'), (10, 'K'), (11, 'L')],
[ (16, 'Q'), (17, 'R'), (18, 'S'), (19, 'T')],
[ (24, 'Y'), (25, 'Z'), (26, 'A'), (27, 'B')]
];
[
[ (4, 'E'), (5, 'F'), (6, 'G'), (7, 'H')],
[ (12, 'M'), (13, 'N'), (14, 'O'), (15, 'P')],
[ (20, 'U'), (21, 'V'), (22, 'W'), (23, 'X')],
[ (28, 'C'), (29, 'D'), (30, 'E'), (31, 'F')]
];
[
[ (32, 'G'), (33, 'H'), (34, 'I'), (35, 'J')],
[ (40, 'O'), (41, 'P'), (42, 'Q'), (43, 'R')],
[ (48, 'W'), (49, 'X'), (50, 'Y'), (51, 'Z')],
[ (56, 'E'), (57, 'F'), (58, 'G'), (59, 'H')]
];
[
[ (36, 'K'), (37, 'L'), (38, 'M'), (39, 'N')],
[ (44, 'S'), (45, 'T'), (46, 'U'), (47, 'V')],
[ (52, 'A'), (53, 'B'), (54, 'C'), (55, 'D')],
[ (60, 'I'), (61, 'J'), (62, 'K'), (63, 'L')]
]
```

For the simpler case of a one-dimensional array, the dense load file format looks like this:

```
[ (36, 'K'), (37, 'L'), (38, 'M'), (39, 'N')];
[ (44, 'S'), (45, 'T'), (46, 'U'), (47, 'V')];
[ (52, 'A'), (53, 'B'), (54, 'C'), (55, 'D')];
[ (60, 'I'), (61, 'J'), (62, 'K'), (63, 'L')]
```

5.1.3 Sparse load format

The sparse load format allows a large number of cells can be unspecified.

The sparse load format lists the data by chunks--there is a semi-colon between the chunks--and within each chunk the data is organized as a list of comma-separated cells, where each cell includes the coordinates and the attributes of the cell.

For example:

```
create array sparse_example<a:double> [x=0:8,2,0,y=0:3,2,0]
```

This array is a 2-D array contained within several 2x2 chunks. The following is an example file '/tmp/sparse_load.txt':

```
cat  /tmp/sparse_load.txt
[[
{0,0} (11)
{1,0} (21)
{0,1} (12)
]];
[[
{0,2} (13)
]];
[[
{2,0} (31)
{3,0} (41)
{2,1} (32)
{3,1} (42)
]];
[[
{2,2} (33)
{3,3} (44)
]];
[[
{7,0} (81)
{6,1} (72)
{7,1} (82)
]];
[[
{6,2} (73)
{7,2} (83)
{7,3} (84)
]];
[[
{8,0} (91)
]];
[[
{8,2} (93)
{8,3} (94)
]]
```

You would use the following load syntax to load this file into the `sparse_example` array:

```
 load(sparse_example, '/tmp/sparse_load.txt')
```

Notes:

1. The last chunk in the load file should **not** have a ';' chunk delimiter.
2. The SciDB loader ignores additional newline or space characters in the load file.
3. Choosing a particular load file format has no bearing on how data is stored internally in SciDB.
4. In addition to storing the data, load and store operators also return the data back to the client (or next operator in the query). When the data is voluminous, this may not be desired and the output of the query should be suppressed. For instance, the iquery executable that accompanies SciDB includes the "-n" option for this purpose.

5. Note that unspecified cells can have various interpretations, depending on how the array is declared. See the section on empty arrays.

5.1.4 Optimizing Chunk Size to Load Format

The size and density of the load file determines the optimum chunk size for its storage array. Chunks should contain on order of 500,000 elements. The number of bits in an element also affects the optimum chunk size. Chunks that have long strings (greater than 30 characters) may need to set the chunk size to less than 500,000 elements for optimum performance.

5.1.5 Datatype Support

SciDB supports the following datatypes for attributes and dimensions. Not all SciDB functions support all datatypes. To see if a function supports a particular datatype, use the `list('functions')` operator.

Data type	Description
binary	Machine-readable binary file
bool	Boolean TRUE (1) or FALSE (0)
char	Single-character
datetime	Date and time
datetimetz	Timezone
double	Double precision decimal
float	Floating-point number
indicator	Datatype indicator
int8	Signed 8-bit integer
int16	Signed 16-bit integer
int32	Signed 32-bit integer
int64	Signed 64-bit integer
string	Character string
uint8	Unsigned 8-bit integer
uint16	Unsigned 16-bit integer
uint32	Unsigned 32-bit integer
uint64	Unsigned 64-bit integer
void	Return nothing

5.1.5.1 Date and Time Offsets

SciDB offers timestamp datatypes so that you can customize date and time formats.

```
create array time_and_date <mytime:datetime>[i=0:0,1,0];
store(build(time_and_date,now()),time_and_date);

[("2011-12-13 18:41:14")]
```

The time zone data type uses offsets indicating a relative time to the date and time to The `datetimetz` formats are:

```
YYYY/MM/DD hh:mm:ss +/-hh:mm
DD.MM.YYYY hh:mm::ss +/-hh:mm
YYYY-MM-DD hh:mm:ss.fff +/-hh:mm
YYYY-MM-DD hh.mm.ss.fff +/-hh:mm
```

You can cast datetime to datetimetz by appending an offset:

```
apply(time_and_date,dst,append_offset(mytime,3600));

[("2011-12-13 18:41:14","2011-12-13 19:41:14 +01:00")]
```

To append an offset and apply it to the time, use apply_offset:

```
create array timedate_and_timezone <mytime:datetime, myzone:datetimetz>[i=0:1,1,0];
store(apply(time_and_date,myzone,apply_offset(mytime,3600)),timedate_and_timezone);

[("2011-12-13 18:41:14","2011-12-13 19:41:14 +01:00")]
```

To cast a datetimetz to datetime, use the strip_offset function:

```
apply(timedate_and_timezone,dst,strip_offset(myzone));

[("2011-12-13 18:41:14",
"2011-12-13 19:41:14 +01:00",
"2011-12-13 19:41:14")]
```

To remove offsets applied to a datetime and cast to datetime, use the togmt function:

```
apply(timedate_and_timezone,dst,togmt(myzone));

[("2011-12-13 18:41:14",
"2011-12-13 19:41:14 +01:00",
"2011-12-13 18:41:14")]
```

To return just the offset from a datetimetz, use the get_offset function:

```
apply(timedate_and_timezone,seconds,get_offset(myzone));

[("2011-12-13 18:41:14",
"2011-12-13 19:41:14 +01:00",
3600)]
```

To return the current time with the current offset, use the tznow function:

```
apply(timedate_and_timezone,est,tznow());

[("2011-12-13 18:41:14",
"2011-12-13 19:41:14 +01:00",
"2011-12-13 17:07:59 -05:00")]
```

You can compare timestamps with the relational operators =, <>, > , <, <=, >=. SciDB compares times after the offset is applied. So "2010-10-10, 13:00:00 +1:00" is equal to "2010-10-10, 12:00:00 +0:00".

5.1.6 Missing Data

Suppose you have a numerical data set that is missing some values:

```
less m4x4_missing.txt

[
[(0,100),(1,99),(2,98),(3,97)],
[(4),(5,95),(6,94),(7,93)],
[(8,92),(9,91),(),(11,89)],
[(12,88),(13),(14,86),(15,85)]
]
```

The array m4x4_missing has two issues: the second values in the cells (1,0) and (3,1) are missing, and cell (2,2) is completely empty. You can tell SciDB how you want to handle the missing data with various array options.

First, consider the case of the completely empty cell, (2,2). By default, SciDB will add zeros to empty cells. If you want SciDB to substitute zeros in all empty cells at load time, create the array with the desired numerical data attributes and load the data set:

```
create array m4x4_missing <val1:double,val2:int32>[x=0:3,4,0,y=0:3,4,0];
load(m4x4_missing,'/tmp/m4x4_missing.txt');

[
[(0,100),(1,99),(2,98),(3,97)],
[(4,0),(5,95),(6,94),(7,93)],
[(8,92),(9,91),(0,0),(11,89)],
[(12,88),(13,0),(14,86),(15,85)]
]
```

This means that cell (2,2) has values (0,0). If the missing data is a string, it will be replaced with an empty string:

```
remove(m4x4_missing);
create array m4x4_missing <val1:string,val2:string> [x=0:3,4,0,y=0:3,4,0];
load(m4x4_missing,
'/tmp/m4x4_missing');

[
[("0","100"),("1","99"),("2","98"),("3","97")],
[("4",""),("5","95"),("6","94"),("7","93")],
[("8","92"),("9","91"),("",""),("11","89")],
[("12","88"),("13",""),("14","86"),("15","85")]
]
```

To change the default value, that is, the value the SciDB subtitutes for the missing data, set the default clause of the attribute option:

```
create array m4x4_missing <val1:double default=3.14159,val2:int32 default 5468>[x=0:3,4,0,y=0:3,4,0];
```

If you want to preserve missing data as empty cells, use the `empty` option for the create array statement:

```
remove(m4x4_missing);
create empty array m4x4_missing <val1:double,val2:int32> [x=0:3,4,0,y=0:3,4,0];
```

```
load(m4x4_missing, '/tmp/m4x4_missing');

[
[(0,100),(1,99),(2,98),(3,97)],
[(4,0),(5,95),(6,94),(7,93)],
[(8,92),(9,91),(),(11,89)],
[(12,88),(13,0),(14,86),(15,85)]
]
remove(m4x4_missing);
create empty array m4x4_missing <val1:string,val2:string> [x=0:3,4,0,y=0:3,4,0];
load(m4x4_missing, '/tmp/m4x4_missing');

[
[("0","100"),("1","99"),("2","98"),("3","97")],
[("4",""),("5","95"),("6","94"),("7","93")],
[("8","92"),("9","91"),(),("11","89")],
[("12","88"),("13",""),("14","86"),("15","85")]
]
```

In this case, cell (2,2) is left empty.

In addition to completely empty cells, SciDB arrays can handle individual missing attribute values within nonempty cells. To substitute 0 for missing numerical attribute values, create the array and load the data:

```
remove(m4x4_missing);
create array m4x4_missing <val1:double,val2:int32>[x=0:3,4,0,y=0:3,4,0];
load(m4x4_missing,'/tmp/m4x4_missing');

[
[(0,100),(1,99),(2,98),(3,97)],
[(4,0),(5,95),(6,94),(7,93)],
[(8,92),(9,91),(0,0),(11,89)],
[(12,88),(13,0),(14,86),(15,85)]
]
```

In the above array, the val2 of cell (1,0) and cell (3,1) are set to 0 during the load process. To set the value to NULL, set the NULL flag of the attribute:

```
remove(m4x4_missing);
create array m4x4_missing <val1:double null,val2:int32 null> [x=0:3,4,0,y=0:3,4,0];
load(m4x4_missing,'/tmp/m4x4_missing');

[[(0,100),(1,99),(2,98),(3,97)],
[(4,null),(5,95),(6,94),(7,93)],
[(8,92),(9,91),(),(11,89)],
[(12,88),(13,null),(14,86),(15,85)]]
```

Missing data may also be specified with NA. For example,

```
less /tmp/na_test.txt

[
[(NA,100),(1,99),(2,98),(3,97)],
[(4,5),(5,95),(6,94),(7,93)],
[(8,92),(9,91),(12,12),(11,89)],
[(12,88),(7,13),(14,86),(15,85)]
]
create array na_test <val1:double default 3.14159, val2:double default 3.14159> [x=0:3,4,0, y=0:3,4,0];
```

```
load(na_test,'/tmp/na_test.txt');

[
[(NA,100),(1,99),(2,98),(3,97)],
[(4,5),(5,95),(6,94),(7,93)],
[(8,92),(9,91),(12,12),(11,89)],
[(12,88),(7,13),(14,86),(15,85)]
]
```

NA may appear in cells that are missing values. NA will not be replaced with the default value at load.

```
less /tmp/na_test_missing.txt

[
[(NA,100),(1,99),(2,98),(3,97)],
[(,NA),(5,95),(6,94),(7,93)],
[(NA,),(9,91),(12,12),(11,89)],
[(NA),(7,13),(14,86),(15,85)]
]

load(na_test,'/tmp/na_test_missing.txt');

[
[(NA,100),(1,99),(2,98),(3,97)],
[(3.14159,NA),(5,95),(6,94),(7,93)],
[(NA,3.14159),(9,91),(12,12),(11,89)],
[(NA,3.14159),(7,13),(14,86),(15,85)]
]
```

If you apply a function to a SciDB attribute that is NA SciDB will return NA:

```
apply(na_test_missing,val3,sin(val1));

[
[(NA,100,NA),(1,99,0.841471),(2,98,0.909297),(3,97,0.14112)],
[(3.14159,NA,2.65359e-06),(5,95,-0.958924),(6,94,-0.279415),(7,93,0.656987)],
[(NA,3.14159,NA),(9,91,0.412118),(12,12,-0.536573),(11,89,-0.99999)],
[(NA,3.14159,NA),(7,13,0.656987),(14,86,0.990607),(15,85,0.650288)]
]
```

See the next section, "Null attribute format" for more information on NULL attributes and default clauses.

5.1.7 NULL attribute format

Both the dense and sparse array formats can include special codes for NULL attributes. For example, if a faulty instrument occasionally fails to report a reading, that attribute could be represented in a SciDB array as NULL. If an erroneous instrument reports readings that are out of valid bounds for an attribute, that may also be represented as NULL.

NULL must be represented using the token 'null' or '?' in place of the attribute value.

In addition, NULL values can be tagged with a "missing reason code" to help a SciDB application distinguish among different types of null values -- for example, assigning a unique code to the following types of errors: "instrument error", "cloud cover", or "not enough data for statistically significant result". Or, in the case of financial market data, data may be missing because "market closed", "trading halted", or "data feed down".

Missing reason codes allow an application to optionally treat each kind of null as a special case, e.g. to supply or calculate a context-sensitive default value.

The examples below show how to represent missing data in the load file. *?* or *null* represent null values, and *?2* represents null value with a reason code of *2*.

```
[[ ( 10, 4.5, "My String", 'C'), (10, 5.1, ?1, 'D'), (?2, 5.1, "Another String", ?) ...

or

[[ ( 10, 4.5, "My String", 'C'), (10, 5.1, ?1, 'D'), (?2, 5.1, "Another String", null) ...
```

The `substitute` operator described later in this chapter can be used to replace missing values with user-defined values looked up from another SciDB array.

The default clause of the NULL flag allows you to set a custom value for missing data:

```
less /tmp/default_value.txt
[(1),(),(3)]
create array D <a:int32 null default 7> [x=0:2,3,0];
load(D,'/tmp/default_value.txt');
[(1),(7),(3)]
```

You can set the default value to value that is legal in the datatype of the attribute. You can also use a function.

```
remove(D);
create array D <a: double default sin(3*3.14/2)> [x=0:2,3,0];
load(D,'/tmp/default_value.txt);
[(1),(-0.999997),(3)]
```

5.1.8 Loading an unbounded array

An unbounded array is one declared to have one or more unbounded dimensions. With an unbounded dimension there is no declared limit on the *high* value. SciDB allows an array to be expanded along unbounded dimensions and new chunks can be appended to an unbounded array after the initial load beyond the current maximum size of the dimension.

```
CREATE ARRAY open_array <a:int64>[x=0:*,5,0]
```

Load data in three phases into *open_array* described in the create statement above.

```
load(open_array, '/tmp/load_1.txt')
load(open_array, '/tmp/load_2.txt')
load(open_array, '/tmp/load_3.txt')
```

The load files are shown below and described in more detail in the sections on `input` and `load`. Initial load from /tmp/load_1.txt

```
[(0), (1), (2), (3), (4)];[(5), (6), (7), (8), (9)]
```

Additional loads:

```
/tmp/load_2.txt
{10}[(15), (16), (17), (18), (19)]
/tmp/load_3.txt
{15}[(10), (11), (12), (13), (14)]
```

5.1.9 Loading in parallel

The optional *nodeid* parameter instructs the load command to open and load data from a particular instance of SciDB. Possible *nodeid* values are:

nodeid	Description
0	Coordinator
1..N	Node id of a node in the cluster, obtained form the system catalog table *node*
-1	All nodes in the cluster

Specifying "-1" for the nodeid causes a parallel load to all nodes in the cluster. The file path of the load file must be the same on all nodes in the cluster, but each node must be given distinct chunks to load. If multiple nodes attempt to load the same chunk during the, the load command will fail. Conversely, if any particular node cannot open the file, the load will continue after logging a warning.

If your SciDB installation has 4 nodes (say, node1, node2, node3, and node4) and the load must load 20 chunks, you can place chunks 1-5 on node1, chunks 6-10 on node2, and so on. The following load command will simultaneously load all 20 chunks into the array in roughly 1/4th the time compared to a coordinator load.

```
load (Array, '/tmp/load.data', -1);
```

5.1.10 Loading through a Unix pipe

All the above methods of loading SciDB can also be done via a Unix pipe (instead of load files). Pipes avoid creating and storing SciDB load files from source files.

5.1.11 csv2scidb

CSV is a common data exchange format. Many systems produce data in CSV format. SciDB provides a utility called *csv2scidb* to convert a csv file into a 1-dimensional array. This array must later be transformed using other SciDB operators such as redimension and redimension_store discussed later in this document.

Consider the csv file /tmp/observations.csv:

```
String_One,15354,01-01-2005 10:11:32,31.7257
String_One,15354,01-01-2005 10:11:35,404.0464
String_One,15354,01-01-2005 10:11:38,926.4216
String_One,,01-01-2005 10:11:41,16.7285
...
```

csv2scidb can be used to re-format this file into a dense SciDB load file ready to be loaded into the 1-dimensional array *Raw_Load*.

```
iquery -anq "create array Raw_Load<s: string, dt: dateTime, r: double>[1:*, 134340,0]"
mkfifo /tmp/load_pipe
cat /tmp/observations.csv  | awk -F, '{print $1","$3","$4}'
   | csv2scidb -c 134340 -p SSN > /tmp/load_pipe &
```

```
iquery -anq "load (Raw_Load, '/tmp/load_pipe')"
```

The following arguments to csv2scidb are shown here. *-c* is used to specify the chunk size, and *-p* is used to specify a parse format for the file (S = string, N = numerical, C = char). See *csv2scidb --help* for more details on usage.

5.2 input

Summary: Input is used to read a file from the filesystem, interpret the contents as cells in an array and return the result. Input has the exact same signature as load.

Signature:

input (*array_name : array_identifier*, *data_file : string* [, *nodeid : int*])

Input works exactly the same way as load, except it does NOT store the data. The store operator can be used to persist the output of the `input` operator. This means that:

```
store ( input (a, 'somefile' ), a)
```

is the same as

```
load ( a, 'somefile' )
```

and

```
store ( input (a, 'somefile' ), b)
```

5.3 substitute

Substitute null values in the input array, using their missing reason code, if using the "?code" format, as index in the second (substitution) array.

Signature:

`substitute`(*array*, *mask: array*)

For example, array foo contains two nulls (which is equivalent to ?0) and one ?1:

```
create array foo <val:int32 null>[x=0:8,1,0];
load(foo,'/tmp/foo.txt');
scan(foo);

[(1),(2),(null),(4),(5),(null),(7),(?1)]
```

You can use substitute to replace ?0 with 20 and ?1 with 30, for instance. We build an array where 0 maps to 20 and 1 maps to 30:

```
build(<val:double> [x=0:1,2,0], iif( x = 0, 20, 30));

[(20),(30)]
```

You can use this build expression as part of substitute directly by using an anonymous schema:

```
substitute(foo,build(<val:int32> [x=0:1,2,0], iif( x = 0, 20, 30)));

[(1),(2),(20),(4),(5),(20),(7),(30)]
```

The `substitute` operator does not store the new values or create a new version of `foo`:

```
list('arrays');

[("foo"),("foo@1")]

scan(foo);

[(1),(2),(null),(4),(5),(null),(7),(?1)]
```

Both input arguments to `substitute` must be single-attribute arrays with the same attribute type.

5.4 save

Summary: Unload array data to a file

Use the save operator to unload the data to an external file on the file system. The format of the file is the same as the load file format described above. If the internal chunk storage format of the array is dense, this file is unloaded into a dense file format. And if the array chunk storage format is sparse, the save output is in sparse file format.

Signature:

```
save( ''array'', ''file_path : string'' )
```

Array name refers to the array to be unloaded and saved. File path is the path to a file to use -- if a relative path name is used this is assumed to be relative to the working directory of the SciDB server.

In a cluster configuration, save operates as follows. For each node in a cluster *save* creates a file with the same file name containing only the chunks stored on that node. The entire array data is obtained by concatenating saved files from all nodes.

---

6 Updates and Versioning

SciDB uses a "no overwrite" storage model - i.e., data is never overwritten, instead each store or update query writes a new version of the array. Versions are stored efficiently within the storage manager to minimize redundant storage. SciDB uses copy-on-write and delta encoding to store only data that has changed between versions, often resulting in efficient internal storage.

6.1 store

Summary: Update a SciDB array

The AFL `store` command is a write operator, that is, one of the AFL commands that can update an array. Each execution of `store` causes a new version of the array to be created. When an array is removed, so are all its versions.

store() can be used to save the resultant output array into an existing/new array. It can also be used to duplicate an array (by using the name of the source array in the first parameter and target_array in the second parameter).

Signature:

> `store`(*input*, *array_name*)

The *input* argument can be:

1. a previously created array, or
2. a sub-query result

The *array_name* argument can be:

1. name of a pre-existing array, or
2. name of a non-existent array. In this case, the array takes the schema of the input array or sub-query result.

Note: SciDB does not allow the array type (EMPTY vs. non-EMPTY) to be altered after array creation. However, array data can be copied from a non-EMPTY array to an EMPTY array if the array dimensions and attributes are compatible.

The example below shows the contents of file /tmp/dense2.txt used in the AFL examples below.

```
{0,3}
[
[(140),  (150),  (160) ],
[(240),  (250),  (260) ],
[(340),  (350),  (360) ]
];
{3,0}
[
[ (410),  (420),  (430) ],
[ (510),  (520),  (530) ],
[ (610),  (620),  (630) ]
```

```
]
```

The input file shown above contains 2 chunks at {0,3} and {3,0} that can be used to update a previously loaded array using:

```
store(merge(input(updarr, '../tests/basic/data/dense2.txt'), updarr), updarr)
```

This creates a new version of the *updarr* array.

6.2 Array Versions

When an array is updated, a new array version is created. You can refer to any version or dimension of an array with the following syntax:

`array_name` [@ `version` | `datetime`] [: `dimension_name`]

For example, suppose you create an array and do two consecutive load commands:

```
CREATE ARRAY A_versions <x:double, y:double>[i=1:10,5,0];
load(A_versions, '/tmp/dataset.txt')

[(1,100),(1,99),(2,98),(3,97),(4,17),
(5,95),(6,94),(7,93),(8,92),(9,91)]

load(A_versions, '/tmp/dataset2.txt');

[(100,100),(99,99),(98,98),(97,97),(17,17),
(5,95),(6,94),(7,93),(8,92),(9,91)]
```

The two calls to `load` will create two versions of the array. To see a listing of an array's versions, use the `version` operator:

```
versions(A_versions);

[(1,"2011-12-13 15:13:18"),
(2,"2011-12-13 15:13:26")]
```

You can use an operator on a prior version of an array using a version qualifier appended to the @ symbol:

```
scan(A_versions@1);
[(1,100),(1,99),(2,98),(3,97),(4,17),
(5,95),(6,94),(7,93),(8,92),(9,91)]

scan(A_versions@2);
[(100,100),(99,99),(98,98),(97,97),(17,17),
(5,95),(6,94),(7,93),(8,92),(9,91)]
```

Array versions can also be identified with a timestamp.

```
scan(A_versions@now());

[(100,100),(99,99),(98,98),(97,97),(17,17),
(5,95),(6,94),(7,93),(8,92),(9,91)]
```

SciDB timestamp format is "YYYY-MM-DD HH:MM:SS".

```
scan(A_versions@datetime('2011-12-13 15:13:26'));

[(100,100),(99,99),(98,98),(97,97),(17,17),
(5,95),(6,94),(7,93),(8,92),(9,91)]
```

By default, the array name without a version identifier refers to the latest version of the array.

```
scan(A_versions)

[(100,100),(99,99),(98,98),(97,97),(17,17),
(5,95),(6,94),(7,93),(8,92),(9,91)]
```

7 Basic Array Operations: Viewing and Manipulating Your Data

7.1 apply

Summary: Compute new attribute values

Use the apply operator to compute new values from attributes and indexes of input arrays. The value(s) computed in the apply are appended to the attributes in the input array.

Signature:

```
apply( array, new_attribute : attribute_identifier, expr_to_apply : expression )
apply( array, new_attribute1 : attribute_identifier1, expr_to_apply1 : expression1
    [, new_attribute2 : attribute_identifier2, expr_to_apply2 : expression2] ...)
```

For example, the query below will compute new attributes C and D, whose values are given by the expressions A + 20 and A + 30. The resulting array will have all the old attributes, as well as the new attributes C and D.

```
apply ( subarray ( my_array, 4, 4, 6, 6 ), C, A + 20, D, A + 30 );
```

Queries with more than one apply statement can also nest them. This is important if one apply statement uses the output of a previous apply statement as input:

```
apply ( apply ( subarray ( my_array, 4, 4, 6, 6 ), C, A + 20 ), D, C * 3);
```

7.2 attribute_rename

Summary: Change attribute name

Works similarly to cast but only changes the name of an attribute. The new name is used in the result array, and the source array is not changed.

Signature:

attribute_rename (*array*, *old_name : attribute_identifier*, *new_name : attribute_identifier*)

Example:

```
attribute_rename(m3x3,val,foo)
```

7.3 build

Return a dense single-attribute array, assigning values to its attributes using an expression. Expressions can refer to one of the functions supported in SciDB, and to dimension values.

Signature:

`build`(*array : array_identifier* | *anonymous_schema* , *expression*)

The build operator's first argument is:

- the name of an array that is to be used as the template for the operator's result, or
- an array schema ("anonymous_schema") to be used as the template for the operator result

`build` proceeds through the array, cell by cell, using the value of the *exp : expression* to compute the value of each cell. The expression can use any of the array's dimension values. On a cluster configuration, build works in parallel on all nodes.

Example 1: Create an identity matrix called Build_Example_INT:

```
CREATE ARRAY Build_Example_INT < A: int32 > [ X=0:4,5,0, Y=0:4,5,0 ];
build (Build_Example_INT, iif(X=Y, 1, 0));

[
[(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)]
]
```

Example 2: Build an array of monotonically increasing values from an anonymous schema:

```
build (<val:int64> [x=1:4,4,0,y=1:4,4,0], x*4+y-5);

[
[(0),(1),(2),(3)],
[(4),(5),(6),(7)],
[(8),(9),(10),(11)],
[(12),(13),(14),(15)]
]
```

Note: The `build` operator does not store data and does not change the given array; it only uses the given shape to generate and return a result. To store the result of the build operator you will need to use the `store` operator.

Note: "The input array or anonymous schema input to `build` should contain only a single attribute (although multiple dimensions are allowed). Use the `join` operator with the `build` operator to build multi-attribute arrays."

Note: If an array is declared as EMPTY it cannot be used as an input to build() operator. EMPTY arrays implicitly include an attribute named "empty_indicator" and so are multi-attribute.

7.4 build_sparse

`build_sparse` works similarly to `build` but returns an EMPTY type array and accepts a second expression that must be boolean.

Signature:

build_sparse(*array : array_identifier* | *anonymous_schema*, *exp : expression*, *bexp : expression*)

The `build_sparse` operator's first argument is:

- the name of an array that is to be used as the template for the operator's result, or
- an array schema ("anonymous_schema") to be used as the template for the operator result.

The output of build_sparse contains empty cells wherever *bexp* evaluates to false. Both expressions can access any of the array coordinates.

Example: Build a diagonal matrix called `sparse_diagonal`:

```
CREATE ARRAY sparse_diagonal <a1: double> [x=0:2,3,0, y=0:2,3,0]
build_sparse(sparse_diagonal, 1.0*x+100.0*y, x=y)
```

Build the same matrix template with an anonymous schema:

```
build_sparse(<a1: double> [x=0:2,3,0, y=0:2,3,0], 1.0*x+100.0*y, x=y)
```

7.5 cast

Summary: Change the attribute or dimension names

The `cast` operator allows renaming an array or any of its attributes and dimensions. A single cast invocation can be used to rename multiple items at once -- one or more attribute names and/or one or more dimension names. The input array and template arrays should have the same numbers and types of attributes and the same numbers and types of dimensions.

Note that the template-array need not have actual data and is only used as a template from which attribute names and dimension names are used. Note that the input array is not modified, rather, a new array is created as a result of the cast operator. To store this newly created array use the store operator.

Signature:

cast (*array*, *template : array_identifier* | *anonymous_schema*)

The `cast` operator's second argument is:

- the name of an array and its schema to be used as the template for the operator's result, or
- an array schema ("anonymous_schema") to be used as the template for the operator result

Example 1:

1. Create an array called source with a attribute called val:

   ```
   create array source <val:double> [x=0:5,3,0];
   ```
2. Use an anonymous schema to change the attribute name to value,and the dimension name to i. Store the result in an array called target:

   ```
   store(cast(source, <value:int64>[i=0:5,3,0]),target);
   ```

`target` has the new names while `source` maintains the old:

```
show(target);

[("target<value:double NOT NULL> [i=0:5,3,0]")]

show(source);

[("source<val:double NOT NULL> [x=0:5,3,0]")]
```

Example 2: One important application of `cast` is the resolution of naming conflicts:

```
store(build (<a:int64> [x=0:10,1,0], x), vector);

[(0),(1),(2),(3),(4),(5),(6),(7),(8),(9),(10)]

store(build (<a:int64> [x=0:10,1,0], -x), vector2);

[(0),(-1),(-2),(-3),(-4),(-5),(-6),(-7),(-8),(-9),(-10)]

apply(join(vector,vector2), b, a+a);
-- ERROR: a is ambiguous

apply(join(vector, cast (vector2, <b:int64>[x=0:10,1,0])), c, a+b);

[(0,0,0),(1,-1,0),(2,-2,0),(3,-3,0),(4,-4,0),(5,-5,0),(6,-6,0),(7,-7,0),(8,-8,0),(9,-9,0),(10,-10,0)]
```

7.6 project

Summary: Select array attributes

Project the input array on the specified attributes, in the specified order. Attributes that are not specified are excluded from the output.

Signature:

project (*array*, *attribute1 : attribute_identifier* [, *attribute-2 : attribute_identifier* ...])

Example:

```
 project( subarray ( my_array, 4, 4, 6, 6 ), A);

(44)  (45)  (46)
(54)  (55)  (56)
(64)  (65)  (66)
```

```
 project( apply ( subarray ( my_array, 4, 4, 6, 6 ), C, A + 20 ), C);
(64.0)  (65.0)  (66.0)
(74.0)  (75.0)  (76.0)
(84.0)  (85.0)  (86.0)
```

7.7 rename

Summary: Rename an array

Signature:

rename(old_name : array_identifier, new_name : array_identifier)

The first argument is the name of the array to be renamed, and the second argument is the new array name.

Example:

```
store ( build (< A: int32 > [ X=0:4,5,0, Y=0:4,5,0 ], iif(X=Y,1,0)), rename_example);
list ('arrays');
rename (rename_example, new_name);
project (new_name, A);

[[(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)]]
```

Notes:

1. Once a rename is done, the old array name can be re-used. Any mapping arrays are also renamed.
2. The `rename` operator should not be used concurrently with another query on the same array. Perform the rename first, then execute other queries on a renamed array.

7.8 scan

Summary: Print attribute values

Scan returns the contents of the array whose name is passed in as an argument.

Signature:

scan(*array_identifier*)

Example:

```
scan(my_array);
```

The 'scan' operator expects an array identifier as its first argument, and returns an array as a result of operation. This operator is used to get a complete listing of the contents of an array stored in the SciDB database. To see a particular version of an array, append the version number to the array name:

```
scan(my_array@1);
```

8 Data Sampling Operators

8.1 bernoulli

Summary: Select cells at random

The bernoulli operator evaluates each cell by generating a random number and seeing if it lies in the range (0, probability). If it does, the cell is included.

Signature

> `bernoulli`(*array* | *array_operator*, *probability:double* [, *seed:int64*])

The first parameter can be an array name or an array operator (operator which outputs an array as result).

The sampling probability is the probability of inclusion. To repeat results, use a seed value for the underlying random number generator.

8.2 between

Summary: Select a subset of data within a specified region

The between operator accepts an input array and a set of coordinates specifying a region within the array. The output is an array of the same shape as input, where all cells outside of the given region are marked empty.

Signature:

> `between`(*array*, *low-boundary-coordinate-1: value*, ..., *low-boundary-coordinate-N: value*, *high-boundary-coordinate-1: value*, ..., *high-boundary-coordinate-N: value*)

Note that between is very similar to subarray, except that between does not change the shape of the array, and does not change the starting coordinates.

For example, consider the following array of integers:

```
less /tmp/fbf.txt
[
[(0),(1),(4),(5)],
[(2),(3),(6),(7)],
[(8),(9),(12),(13)],
[(10),(11),(14),(15)]
]
create array fbf <val1:int64>[x=0:3,4,0, y=0:3,4,0];
load(fbf, '/tmp/fbf.txt');

[
[(0),(1),(4),(5)],
[(2),(3),(6),(7)],
[(8),(9),(12),(13)],
[(10),(11),(14),(15)]
]
```

Between can be used to set all cells outside of the (1,1)->(2,2) region to empty:

```
between(fbf,1,1,2,2);

[
[(),(),(),()],
[(),(3),(6),()],
[(),(9),(12),()],
[(),(),(),()]
]
```

8.3 filter

Summary: Select subset of data by boolean expression

The filter operator 'filters' out data in the array based on an expression over the attribute and dimension values. Suppose we want to filter out all the data from my_array, where the value of attribute A doesn't equal 75. We would use the following query:

```
filter( my_array, A <> 75);
```

The filter operator marks all cells in the input which do not satisfy the predicate expression to 'empty'.

Signature:

filter (*array*, *condition : expression*)

where 'condition' is a boolean expression.

The following examples illustrate both the functionality of the filter operator, and also how operators in AFL can be combined into query expressions.

```
filter ( subarray ( my_array, 4, 4, 6, 6), A > 46 );

(0,0.0,false)   (0,0.0,false)  (0,0.0,false)
(54,54.0,true)  (55,55.0,true)  (56,56.0,true)
(64,64.0,true)  (65,65.0,true)  (66,66.0,true)
```

```
filter ( subarray ( my_array, 4, 4, 6, 6), A > 46 and A < 56);

(0,0.0,false)   (0,0.0,false)   (0,0.0,false)
(54,54.0,true)  (55,55.0,true)  (56,56.0,true)
(0,0.0,false)   (0,0.0,false)  (0,0.0,false)
```

The output of the filter is an array with the EMPTY attribute.

8.4 lookup

Summary: Pattern selection

Lookup maps elements from the second array using the attributes of the first array as coordinates into the second array. The result array has the same shape as first array and the same attributes as second array.

Signature:

lookup (*pattern : array*, *source: array*)

Example: suppose the array foo has coordinates 0 to 7 and the values 1 to 8:

```
scan(foo);

[(1),(2),(3),(4),(5),(6),(7),(8)]
```

Suppose the array bar ranges from -10 to 10 and has values from 40 to 60:

```
scan(bar);

[(40),(41),(42),(43),(44),(45),(46),(47),(48),(49),
 (50),(51),(52),(53),(54),(55),(56),(57),(58),(59),(60)]
```

Then lookup(foo,bar) returns 8 values, corresponding to bar positions from 1 to 8 :

```
lookup(foo,bar);

[(51),(52),(53),(54),(55),(56),(57),(58)]
```

8.5 sample

Summary: Select random chunk

In sample, the selection is chunk-based. We evaluate each chunk for inclusion based on the probability.

Users often want samples with repeatable results. To achieve this, SciDB uses a *seed* value for the underlying random number generator.

Signature:

sample(*array*, *probability : double* [, *seed : int64*])

Example:

```
bernoulli(m4x4,0.3);

[
[(0),(1),(0),(3)],
[(0),(0),(0),(0)],
[(8),(0),(0),(0)],
[(12),(13),(0),(0)]
]
```

9 Sorting, Windowing, and Aggregating: Grouping Your Data

9.1 aggregate

Summary: Group elements and find statistical properties of the group

The aggregate operator takes an array as input, groups the array by the specified dimension and computes a given statistic for each group. The statistics available are:

Name	Operation Performed
avg	Average value
count	Number of nonempty elements
max	Largest value
min	Smallest value
sum	Sum of all elements
stdev	Standard deviation
var	Variance

The aggregate operator returns a scalar value for each statistic.

Signature:
`aggregate`(*array, aggregate_name_1(attribute)*
[, *aggregate_name_2(attribute),... aggregate_name_N(attribute)*]
[, *dimension_1, dimension_2,... dimension_M*])

Example:

1. Create a 3-by-3 array called m3x3:

```
store ( build ( <val:double> [x=0:2,3,0,y=0:2,3,0], x*3+y), m3x3);

[
[(0),(1),(2)],
[(3),(4),(5)],
[(6),(7),(8)]
]
```

2. Find the sums of each column:

```
aggregate(m3x3,sum(val),y);

[(9),(12),(15)]
```

3. Find the average value, number of nonempty elements, largest element, smallest element, sum of all elements, variance, and standard deviation:

```
aggregate(m3x3,avg(val),count(val),max(val),min(val),sum(val),var(val),stdev(val));

[(4,9,8,0,36,7.5,2.73861)]
```

You can also use the built-in aggregate operations avg, count, max, min, sum, var, and stdev.

9.1.1 avg

Summary: Arithmetic mean

Calculate the average value of the specified attribute in the array. The result is an array with single element containing average value. If the input array contains only one attribute, then attribute name can be omitted.

Signature:

avg(*array* [, *attribute-name : attribute_identifier* [, *dimension1 : dimension_identifier* [, *dimension2 : dimension_identifier*]]])

The first argument is the array to be aggregated over. The second argument is the name of the attribute to use. Additional dimension arguments are optional. If present, the list of dimensions specified in the avg operator is used to perform a group-by average. The result is organized as an array with the remaining dimensions from the source array, after grouping has been performed based on the group-by dimensions.

Example: The following command returns the average value of a over all elements of array.

```
avg(m3x3, val);

[(4)]
```

This example calculates the average value of an attribute after grouping over all values of the dimension x. The second example below results in a one-dimensional array whose dimension is the remaining dimension y from the input.

```
avg(m3x3, val, y);

[(3),(4),(5)]
```

9.1.2 count

Summary: Cell count

Counts non-empty cells of the input array. When dimensions are provided they are used to do a group-by and a count per resulting group is returned.

Signature:

count(*array* [, *dimension1 : dimension_identifier* [, *dimension2 : dimension_identifier*]])

Example:

```
count(m3x3);

[(9)]

count(m3x3,x);

[(3),(3),(3)]
```

Note that cells of value 0 or null are not considered empty:

```
create array A <a:int32 null> [x=0:0,1,0];
count(A);

[(0)]

store(build(A,x),A);
count(A);

[(1)]

store(build(A,null),A);
count(A);

[(1)]
```

This is different than the behavior of the aggregate operator with the count option. count(array_name) is a shorthand for aggregate(array_name, count(*)). For example:

1. Create an array with a single attribute and three cells:

   ```
   create empty array A <a:int32 null default 0> [x=1:3,3,0];
   ```
2. Put "1" in cell 1, "null" in cell 2, and make cell 3 empty:

   ```
   store(build_sparse(A, iif(x=1,1,null), x<>3),A);

   [{1}(1),{2}(null)]
   ```
3. Return the count of nonempty cells:

   ```
   aggregate(A, count(*));

   [(2)]
   ```
4. Return the number of values where attribute is not null:

   ```
   aggregate(A, count(a));

   [(1)]
   ```

9.1.3 max

Summary: Maximum value

Calculate maximum of the specified attribute in the array. Result is an array with single element containing maximum of specified attribute.

If input array contains only one attribute, then attribute name can be omitted. Again, if dimensions are provided, they are used to produce groups and the maximum of each group is returned.

Signature:

max(*array* [, *attribute-name : attribute_identifier* [, *dimension1 : dimension_identifier* [, *dimension2 : dimension_identifier*]]])

Example:

```
max(m3x3);

[(8)]

max(m3x3,val,x);

[(2),(5),(8)]
```

9.1.4 min

Summary: Minimum value

Calculates the minimum value of the specified attribute in the array. Result is an array with single element containing minimum of specified attribute. If input array contains only one attribute, then attribute name can be omitted. If an attribute list is specified, the result is an array with the remaining dimensions from the source array, and the minimum is evaluated over groups, where each group is the set of all elements matching the group by dimension(s).

Signature:

min(*array* [, *attribute-name : attribute_identifier* [, *dimension1 : dimension_identifier* [, *dimension2 : dimension_identifier*]]])

Example:

```
min(m3x3);

[(0)]

min(m3x3,val,y);

[(0),(1),(2)]
```

9.1.5 sum

Summary: Sum attribute values

Calculate sum of the specified attribute in the array. Result is an array with single element containing sum of specified attribute. If input array contains only one attribute, then attribute name can be omitted. Again, similar group by semantics as with the other aggregates, if a dimension list is present.

Signature:

sum(*array* [, *attribute-name : attribute_identifier* [, *dimension1 : dimension_identifier* [, *dimension2 : dimension_identifier*]]])

Example:

```
sum(m3x3);

[(36)]
```

```
sum(m3x3,val,x);

[(3),(12),(21)]
```

9.1.6 var

Summary: Variance of attribute values

Calculate the variance of the specified attribute in the array. Result is an array with single element containing the variance of specified attribute. If input array contains only one attribute, then attribute name can be omitted. Again, similar group by semantics as with the other aggregates is supported, if a dimension list is present.

Signature:

var(*array* [, *attribute-name : attribute_identifier* [, *dimension1 : dimension_identifier* [, *dimension2 : dimension_identifier*]]])

Example:

```
var(m3x3);

[(7.5)]

var(m3x3,val,x);

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

Variance accepts numeric attributes only. The result is the sample variance of the attribute values.

9.1.7 stdev

Summary: Standard deviation of attribute values

Calculate the standard deviation of the specified attribute in the array. Result is an array with single element containing the standard deviation of specified attribute. If input array contains only one attribute, then attribute name can be omitted. Again, similar group by semantics as with the other aggregates is supported, if a dimension list is present.

Signature:

stdev(*array* [, *attribute-name : attribute_identifier* [, *dimension1 : dimension_identifier* [, *dimension2 : dimension_identifier*]]])

Example:

```
stdev(m3x3);

[(2.73861)]

stdev(m3x3,val,y);
```

```
[(3),(3),(3)]
```

Standard deviation accepts numeric attributes only. The result is the population standard deviation of the attribute values.

9.2 regrid

Summary: Compute aggregates for a sub-grid

Partition (divide) the cells in the input array into blocks, and for each block, apply a specific aggregate operation over the value(s) of some attribute in each block.

`regrid` does not allow grids to span array chunks and requires the chunk size to be a multiple of the grid size in each dimension.

Signature:

regrid (array, grid_1, grid_2,.. grid_N, aggregate_call_1 [, aggregate_call_2,...aggregate_call_N])

The first input to the regrid operator can be:

1. An array.
2. An array operator, that is, an operator that outputs a SciDB array.

The number of grids to be input should be equal to the number of dimensions of the array.

Example:

```
store ( build ( <val:double> [x=0:3,4,0,y=0:3,4,0], x*4+y), m4x4);

[
[(0),(1),(2),(3)],
[(4),(5),(6),(7)],
[(8),(9),(10),(11)],
[(12),(13),(14),(15)]
]

regrid(m4x4, 2,2, sum(val));

[[(10),(18)],[(42),(50)]]

regrid(m4x4, 2,2, sum(val),max(val));

[[(10,5),(18,7)],[(42,13),(50,15)]]
```

9.3 sort

Summary: Sort by attribute value

Sort a one-dimensional array by one or more attributes. The sort attributes are specified using a 1-based attribute number.

Signature:

sort (array, attribute_name [desc] [, ...])

The first argument can be:

1. An array
2. An array operator, that is, an operator that outputs a SciDB array.

The attribute name can optionally be followed by `desc` to sort in descending order. Default is ascending order.

Example:

```
less /tmp/sd.txt
[(1,"a"),(2,"z"),(0,"b"),(3,"x"),(2,"f"),(6,"o"),(1.5,"e"),(2.7,"l")]

create array sd <num:double,label:string>[x=0:8,8,0];
load(sd,'/tmp/sd.txt');

[(1,"a"),(2,"z"),(0,"b"),(3,"x"),(2,"f"),(6,"o"),(1.5,"e"),(2.7,"l")]

sort(sd,num);

[(0,"b"),(1,"a"),(1.5,"e"),(2,"z"),(2,"f"),(2.7,"l"),(3,"x"),(6,"o")]
```

To sort in descending order, use the desc argument:

```
sort(sd,num desc);

[(6,"o"),(3,"x"),(2.7,"l"),(2,"z"),(2,"f"),(1.5,"e"),(1,"a"),(0,"b")]

sort(sd,label desc);

[(2,"z"),(3,"x"),(6,"o"),(2.7,"l"),(2,"f"),(1.5,"e"),(0,"b"),(1,"a")]
```

9.4 window

Summary: Compute aggregates over a window

Compute one or more aggregates of any of an array's attributes over a moving window.

Signature:

```
window(array, grid1, grid2,.. gridN, aggregate_call1 [, aggregate_call2,...aggregate_callN])
```

Each `aggregate_call` argument consists of a call to an aggregate and an optional alias. For example:

```
sum(val) as output
```

will sum the attribute called `val` and place it in `output`.

Example:

```
create array m4x4 <val1:double,val2:int32> [x=0:3,4,0,y=0:3,4,0];
load(m4x4, '/tmp/m4x4_2attr');
[
[(0,100),(1,99),(2,98),(3,97)],
[(4,96),(5,95),(6,94),(7,93)],
[(8,92),(9,91),(10,90),(11,89)],
[(12,88),(13,87),(14,86),(15,85)]
]
window(m4x4,2,2, max(val1),sum(val2));
[
[(0,100),(1,199),(2,197),(3,195)],
[(4,196),(5,390),(6,386),(7,382)],
[(8,188),(9,374),(10,370),(11,366)],
[(12,180),(13,358),(14,354),(15,350)]]
```

The resulting schema for the output of `window` is:

```
 <val1_sum:double NULL, val2_max:int32 NULL>[x=0:3,4,0, y=0:3,4,0]
```

You can also use an alias with the aggregate call:

```
window(m4x4,2,2, max(val1) as z, sum(val2) as w);
```

The window is defined by a size in each dimension, for example, the window in the above example is 3-by-3.

The starting position of the window centroid is the first element of the array. At the edges of the array the window aggregate only contains elements that are included in the array. The centroid of the window moves in stride-major order from lowest to highest value in each dimension. The output array has the same shape (no overlap) as the input array. Each element of the output array contains the aggregates computed over the corresponding window location over the input array.

SciDB uses array overlap to perform the window operation on each chunk locally. If necessary a repart operator is implicitly included in the execution plan.

---

10 Combining Arrays: Putting the Pieces of Your Database Together

10.1 concat

Summary: Concatenate two arrays

Array should have the name number of dimensions. Concatenation is performed by the left-most dimension. All other dimensions of the input arrays must match. The left-most dimension of both arrays must have a fixed size (not unbounded) and same chunking schema. Both inputs must have the same attributes.

The inputs to concat can be:

1. An existing array.
2. An array operator, that is, an operator that outputs a SciDB array.

Signature:

`concat`(*left: array*, *right : array*)

Example:

```
store(build(<val:int64>[x=0:3,100,0,y=0:4,100,0],x*5+y),four_by_five);

[
[(0),(1),(2),(3),(4)],
[(5),(6),(7),(8),(9)],
[(10),(11),(12),(13),(14)],
[(15),(16),(17),(18),(19)]
]

store(build(<val:int64>[x=0:1,100,0,y=0:4,100,0],20+x*2+y),two_by_five);
[
[(20),(21),(22),(23),(24)],
[(22),(23),(24),(25),(26)]
]

concat(four_by_five,two_by_five);

[
[(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)],
[(22),(23),(24),(25),(26)]
]
```

Note that concat can be combined with transpose, adddim, deldim, subarray, slice and other ops for more complex transformations.

10.2 cross

Summary: Cross-product join

The inputs to cross can be:

1. An existing array.
2. An array operator, that is, an operator that outputs a SciDB array.

Calculates the full cross product join of two arrays, say A (m-dimensional array) and B (n-dimensional array) such that the result is an m+n dimensional array in which each cell is computed as the concatenation of the attribute lists from corresponding cells in arrays A and B. For example, consider a 2-dimensional array A with dimensions i, j, and a 1-dimensional array B with dimension k. The cell at coordinate position {i, j, k} of the output is computed as the concatenation of cells {i, j} of A with cell at coordinate {k} of B.

Signature:

cross(*left : array*, *right : array*)

Example:

```
create array a<a1: double>[i=1:3,3,0, j=1:3,3,0];
create array b<b1: double>[k=1:2,2,0];

store(build(a, i+j), a);

[[(2),(3),(4)],
[(3),(4),(5)],
[(4),(5),(6)]]

store(build(b, 1.0/k), b);

[(1),(0.5)]

cross(a, b);

[[
[(2,1),(2,0.5)],
[(3,1),(3,0.5)],
[(4,1),(4,0.5)]
],
[
[(3,1),(3,0.5)],
[(4,1),(4,0.5)],
[(5,1),(5,0.5)]
],
[
[(4,1),(4,0.5)],
[(5,1),(5,0.5)],
[(6,1),(6,0.5)]
]]
```

The output is a 3-dimensional array with the following schema:

```
<a1:double NOT NULL, b1:double NOT NULL>[i=1:3,3,0, j=1:3,3,0, k=1:2,2,0]
```

10.3 cross_join

Summary: Cross-product join with equality predicates

Calculates the cross product join of two arrays, say A (m-dimensional array) and B (n-dimensional array) with equality predicates applied to pairs of dimensions, one from each input. Predicates can only be computed along dimension pairs that are aligned in their type, size, and chunking.

Assume p such predicates in the cross_join, then the result is an m+n-p dimensional array in which each cell is computed by concatenating the attributes as follows:

For a 2-dimensional array A with dimensions i, j, and a 1-dimensional array B with dimension k, cross_join(A, B, j, k) results in a 2-dimensional array with coordinates {i, j} in which the cell at coordinate position {i, j} of the output is computed as the concatenation of cells {i, j} of A with cell at coordinate {k=j} of B.

Signature:

cross_join(*left : array*, *right : array*, *left_dim_1: dimension_identifier*, *right_dim_1 : dimension_identifier* [, *left_dim_2: dimension_identifier*, *right_dim_2: dimension_identifier*,...])

Example:

```
create array a<a1: double>[i=1:3,3,0, j=1:3,3,0];
create array b<b1: double>[k=1:3,3,0];

store(build(a, i+j), a);

[[(2),(3),(4)],
[(3),(4),(5)],
[(4),(5),(6)]]

store(build(b, 1.0/k), b);

[(1),(0.5), (0.333)]

cross_join(a, b, j, k);

[
[(2,1),(3,0.5),(4,0.333333)],
[(3,1),(4,0.5),(5,0.333333)],
[(4,1),(5,0.5),(6,0.333333)]
]
```

The result has schema:

```
<a:double NOT NULL, b:double NOT NULL>[i=1:3,3,0, j=1:3,3,0]
```

10.4 join

Summary: Join two arrays

Join combines the attributes of two input arrays at matching dimension values.

Signature:

join (*left : array*, *right : array*)

The following example illustrates a join between two different subarray operators applied to the same underlying array. Note that join combines attributes from cells at dimension addresses on its immediate input arrays; not in the original inputs:

```
 subarray ( my_array AS M1, 4, 4, 6, 6 );

(44,44.0)  (45,45.0)  (46,46.0)
(54,54.0)  (55,55.0)  (56,56.0)
(64,64.0)  (65,65.0)  (66,66.0)
```

```
 subarray ( my_array AS M2, 6, 6, 8, 8 );

(66,66.0)  (67,67.0)  (68,68.0)
(76,76.0)  (77,77.0)  (78,78.0)
(86,86.0)  (87,87.0)  (88,88.0)
```

```
join ( subarray ( my_array AS M1, 4, 4, 6, 6 ),
  subarray ( my_array AS M2, 6, 6, 8, 8 ) );

(44,44.0,66,66.0)  (45,45.0,67,67.0)  (46,46.0,68,68.0)
(54,54.0,76,76.0)  (55,55.0,77,77.0)  (56,56.0,78,78.0)
64,64.0,86,86.0)  (65,65.0,87,87.0)  (66,66.0,88,88.0)
```

The join result has the same dimension names as the first input. The left and right arrays must have the same shape. If a cell in either the left or right array is empty, the corresponding cell in the result is also empty.

10.5 merge

Summary: Merge two arrays.

The inputs to merge can be:

1. An existing array.
2. An array operator, that is, an operator that outputs a SciDB array.

The two input arrays should have the same shape as one another: that is, the same attribute list and dimensions.

Merge combines elements from the input array the following way: for each cell in the two inputs, if the cell of first (left) array is *non-empty*, then the attributes from that cell are selected and placed in the output. If the cell in the first array is marked as *empty*, then the attributes of the corresponding cell in the second array are taken. If the cell is *empty* in both input arrays, the output's cell is set to *empty*.

Signature:

merge(*left : array*, *right : array*)

Example:

```
scan(vec1);

[[(1,true)],[()],[(3,true)],[()],[(5,true)]]

scan(vec2);

[[()],[(5,true)],[()],[(7,true)],[()]]

merge(vec1,vec2);

[[(1,true)],[(5,true)],[(3,true)],[(7,true)],[(5,true)]]
```

11 Array Schemas: Change the Shape and Size of Your Arrays

In the case of some shape-changing operators that require a fully qualified array schema, SciDB supports two ways to specify an array schema.

- You can create a new array with the desired output schema and refer to it for the changed shape:

```
show(src);
[("a1<a:double NOT NULL> [i=1:3,3,0,j=1:3,3,0]")]
reshape(src, foo)
```

- You can specify a schema without a creating a named array. This is called an *anonymous schema*:

```
reshape(src, <a: double>[k=1:9,9,0]);
```

11.1 adddim and deldim

Summary: Adding and deleting dimensions of an array

Change the structure of an array by adding or deleting a dimension.

The adddim operator prepends the existing dimensions for an array with a new dimension, whose name is supplied as the second argument to the operator.

The new dimension will have start = 0, length = 1, chunkSize = 1, overlap = 0.

The `deldim` operator deletes the left-most dimension from the array. Deleted dimension must have size = 1.

Signature:

```
adddim( array, new_dimension : dimension_name )
deldim( array )
```

Examples:

```
adddim(matrix, timestamp);
deldim(subarray(matrix, 1, 100, 1, 200));
```

11.2 redimension and redimension_store

Summary: Transform attributes to dimensions

The `redimension` and `redimension_store` operators convert array attributes to dimensions. You can redimension the array and apply aggregates to duplicate cells.

Signature:

redimension(*array*, *target : array_identfier*)
redimension_store(*array*, *target : array_identfier*)

redimension_store (source_array, target_array [, (aggregate_call_1 as alias_1) ,... (aggregate_call_N as alias_N)])

`redimension` does not create or update array storage or metadata, and returns the transformed array result. redimension also only works when transforming int64 attributes into dimensions.

`redimension_store` updates the target_array storage and creates additional mapping arrays if necessary.

For both variants of redimension, the target array must have the EMPTY flag set.

The input and target arrays must have compatible schemas, and both commands determine the list of transformations (attribute to dimension) by matching names in the attribute and dimension lists of the two arrays. It is possible to omit certain attributes and dimensions from the target array.

Note: redimension supports noninteger dimensions while redimension_store does not.

Example 1: Count the even and odd values in an array.

1. Create an array that has a single attribute "a".

```
create array A <a:double>[x=0:5,3,0,y=0:5,3,0];
store(build(A, x*3+y), A);
```

2. Apply a synthetic attribute "even_or_odd" that is 0 when a is even and 1 when a is odd. Turn the even_or_odd attribute into a dimension with the `redimension` command and perform a `count` aggregate:

```
redimension(apply(A, even_or_odd, iif(int64(a)%2=0,0,1)),
  <count:uint64 null,empty_tag:indicator>[even_or_odd=0:1,2,0], count(a) as count);
```

SciDB returns:

```
[{0}(18),{1}(18)]
```

This tells you that there are 18 even and 18 odd values in the array.

Note: the target array schema must contain the empty_tag attribute and the datatype of the new `count` attribute must match the output of the aggregate (uint64 and nullable).

Example 2: Maintain one of the original dimensions and use the aggregates `sum` and `avg` to find the sum and average of the even and odd elements:

```
redimension(apply(A, even_or_odd, iif(int64(a)%2=0,0,1)),
  <sum:double null, avg:double null, empty_tag:indicator>[x=0:5,3,0,even_or_odd=0:1,2,0],
  sum(a) as sum, avg(a) as avg);
```

SciDB returns:

```
[
[{0,0}(6,2),{0,1}(9,3),{1,0}(18,6),{1,1}(15,5),{2,0}(24,8),{2,1}(27,9)]
];
[
[{3,0}(36,12),{3,1}(33,11),{4,0}(42,14),{4,1}(45,15),{5,0}(54,18),{5,1}(51,17)]
```

```
]
```

This means that for x=0, the sum of the even numbers is 6 and the average is 2; the sum of the odd numbers is 9 and average is 3.

Example 3: Instead of creating a schema, the target schema can come from a pre-existing array:

```
create empty array dense_redim <a:double>[x=0:5,3,0]
redimension(dense,dense_redim)
```

In this case, we don't use any aggregates and condense values. In the original array, there are 6 values for x=0, and the result of the query at x=0 will contain an unspecified value, chosen from one of the 6 possible candidates. This command does not change any data for the array dense_redim - just reads the schema from it.

The two statements

```
create empty array dense_redim <a:double>[x=0:5,3,0];
```

and

```
create array dense_redim <a:double, empty_tag:indicator>[x=0:5,3,0];
```

are equivalent.

11.3 repart

Summary: Change array chunking

Change partitioning (chunking) of the array. Target array must have the same attributes and dimensions, but chunk size may be different. Repart returns an array whose attributes are taken from the input array, with the dimensions of the target.

Signature:

repart(*array*, *target : array_identifier* | *anonymous_schema*)

The input to `repart` can be:

1. An array
2. A subquery that outputs a SciDB array

Example:

1. Create a 4-by-4 array with chunk size of 1 called `source`:

   ```
   create array source <val:double> [x=0:3,1,0,y=0:3,1,0];
   ```
2. Add numerical values to source:

   ```
   store(build(source,x*3+y),source);
   ```
3. Repartition the array into 2-by-2 chunks and store the result in an array called `target`:

```
store(repart(source, <values:double> [x=0:3,2,0, y=0:3,2,0]),target);
show(target);

[("target<val:double NOT NULL> [x=0:3,2,0,y=0:3,2,0]")]
```

11.4 reshape

Summary: Change array shape

Change the shape of an array to that of another array or array schema.

Signature:

reshape(*array*, *target : array_identifier* | *anonymous_schema*)

The first input to `reshape` can be:

- An array
- A subquery that outputs a SciDB array

The `reshape` operator's second argument is:

- the name of an array and its schema to be used as the template for the operator's result, or
- an array schema ("anonymous_schema") to be used as the template for the operator result

NOTES:

1. The target array (or anonymous array schema) should have the same number of attributes as the input array.
2. The arrays must have fixed size dimensions. That is, reshape for unbounded arrays is not supported.
3. Size of the input and target array should be the same. That is, both arrays should have the same number of *cells*. For example, it is possible to reshape 2x2x2 array to 4x2, but not 3x3 to 2x2.

Example:

1. Create a 3-by-4 array called `source`:

   ```
   create array source <val:double> [x=0:2,1,0,y=0:3,1,0];
   ```
2. Add numerical values to source:

   ```
   store(build(source,x*3+y),source)
   ```
3. Reshape the array to 4-by-3 and store the result in an array called `target`:

   ```
   store(reshape(source, <values:double> [x=0:3,1,0, y=0:2,1,0]),target)
   show(target);

   [("target<val:double NOT NULL> [x=0:3,1,0,y=0:2,1,0]")]
   ```

11.5 reverse

Summary: Invert the elements of an array by reversing the values of each dimension.

Signature:

reverse(*array*)

Assume that dense is a 2-dimensional array with the following definition and contents:

```
CREATE ARRAY dense <a: int32>[x=0:5,3,0, y=0:5,3,0]

scan(dense)
[
[(66),(65),(64)],
[(56),(55),(54)],
[(46),(45),(44)]
];
[
[(63),(62),(61)],
[(53),(52),(51)],
[(43),(42),(41)]
];
[
[(36),(35),(34)],
[(26),(25),(24)],
[(16),(15),(14)]
];
[
[(33),(32),(31)],
[(23),(22),(21)],
[(13),(12),(11)]
]
```

reverse returns an array in which each dimension is the reverse of the corresponding dimension of the source array.

```
reverse(dense)
[
[(11),(12),(13)],
[(21),(22),(23)],
[(31),(32),(33)]
];
[
[(14),(15),(16)],
[(24),(25),(26)],
[(34),(35),(36)]
];
[
[(41),(42),(43)],
[(51),(52),(53)],
[(61),(62),(63)]
];
[
[(44),(45),(46)],
[(54),(55),(56)],
[(64),(65),(66)]
]
```

11.6 slice

Summary: Subplane of an array

Get a slice of the array. The result is a slice of the input array corresponding to the given coordinate value(s). Number of dimensions of the result array is equal to the number of dimensions of input array minus number of specified dimension, and the coordinate value should be a valid dimension value of the input array.

Signature:

slice(*array*, *dimension1 : dimension_identifier*, *coordinate1 : value* [,*dimensionN : dimension_identifier*, *coordinate-value-N : value*])

The first input to `slice` can be:

- An array
- A subquery that outputs a SciDB array

Example:

```
scan(m3x3);

[
[(0),(1),(2)],
[(3),(4),(5)],
[(6),(7),(8)]
]

slice(m3x3,x,1);

[(3),(4),(5)]
```

11.7 subarray

Summary:Select by dimension range

Subarray selects a block of cells from an input array. The result is an array whose shape is defined by the 'bounding box' specified by the subarray.

Signature:

subarray (*array*, *low-boundary-coordinate-1: value* , ..., *low-boundary-coordinate-N: value*, *high-boundary-coordinate-1: value*, ..., *high-boundary-coordinate-N: value*)

The first argument to the subarray operator is the array from which the block is to be sampled. This argument can be:

- An array
- A subquery that outputs a SciDB array

The second parameter section specifies which block is to be extracted. There are as many pairs of array index

values in the parameter block as the *input : array* has dimensions. For each dimension, the operator requires the minimum index value for each dimension first, followed by the maximum index value for each dimension. In other words, if the *input : array* has two dimensions, then the subarray operator requires four start and end values.

Example:

Suppose 'my_array' is a 2-dimensional array with two integer attributes and dimensions x and y.

```
create array my_array <val1:int64, val2:int64> [x=0:4,5,0,y=0:4,5,0];
load(my_array, '/tmp/my_array.txt');

[
[(1,3),(5,7),(9,11),(13,15),(17,19)],
[(21,23),(25,27),(29,31),(33,35),(37,39)],
[(41,43),(45,47),(49,51),(53,55),(57,59)],
[(61,63),(65,67),(69,71),(73,75),(77,79)],
[(81,83),(85,87),(89,91),(93,95),(97,99)]
]
```

The first input to the subarray operator is the source array. The rest of the inputs are the indexes of the subarray window. First, the lower bound indexes for all dimensions in the input array, are given, then the upper ones. The result of subarray is an array of smaller or equal size, whose dimensions always start at 0 and span only the length of the subarray region.

You can use the subarray operator as shown here:

```
subarray( my_array, 1, 1, 2, 2)
```

This query will return a block of the input array where the initial (top-left) cell is [1,1] and the final (bottom-right) cell is at [2, 2].

```
(11,11.0)  (12,12.0)
(21,21.0)  (22,22.0)
```

This query will return a block of the input array where the initial (top-left) cell is [4,4] and the final (bottom-right) cell is at [6,6].

```
 subarray ( my_array, 4, 4, 6, 6);

(44,44.0)  (45,45.0)  (46,46.0)
(54,54.0)  (55,55.0)  (56,56.0)
(64,64.0)  (65,65.0)  (66,66.0)
```

```
 subarray ( my_array, 4, 3, 6, 7);

(43,43.0)  (44,44.0)  (45,45.0)  (46,46.0)  (47,47.0)
(53,53.0)  (54,54.0)  (55,55.0)  (56,56.0)  (57,57.0)
(63,63.0)  (64,64.0)  (65,65.0)  (66,66.0)  (67,67.0)
```

11.8 thin

Summary: Select elements from an array dimension

Signature:

```
thin( ''array'', ''start-1'', ''step-1'', ''start-2'', ''step-2'', ... )
```

Select regularly spaced elements of the array in each dimension. The selection criteria are specified by the starting dimension value *start-i* and the number of cells to skip using *step-i* for each dimension of the input array.

Example:

1. Create a 2-dimensional array:

   ```
   create array A <a:int32> [x=0:8,6,0,y=0:8,8,1];
   ```
2. Store numerical values in the array:

   ```
   store(build(A,(x*8+y),A));
   ```
3. Select every other element from the first dimension:

   ```
   thin(A,1,3,0,2);
   ```

Note: The position specified by *start-1* must be within the actual, rather than the relative position of the array indices. For example, if the range of dimension x were 10:18, the thin command would throw an out-of-range error.

11.9 unpack

Summary: Change multidimensional array to single dimension

Unpack array into a single-dimensional array creating new attributes to represent source array dimension values. Result array has a single zero-based dimension and arguments combining attributes of the input array. The name for the new single dimension is passed to the operator as the second argument.

Signature:

`unpack(` *array*, *attribute_name* `)`

Example:

```
show(m3x3);

[("m3x3<val:double NOT NULL> [x=0:2,3,0,y=0:2,3,0]")]

scan(m3x3);

[
[(0),(1),(2)],
[(3),(4),(5)],
[(6),(7),(8)]
]

unpack(m3x3,i);

[(0,0,0),(0,1,1),(0,2,2),(1,0,3),(1,1,4),
```

```
(1,2,5),(2,0,6),(2,1,7),(2,2,8)]
```

The format for the output of the unpack operator is to begin by enumerating the dimensions of the input, and then to append the attribute values. Chunk size -- ie., the number of elements per chunk is preserved from input to output. The resulting one-dimensional array has chunk dimension equal to the count of cells in each input chunk.

NOTE: unpack only supports arrays whose size is evenly divisible by chunk size in every dimension. You cannot unpack an array A <a:int32> [x=0:100,30,0]; but you can unpack an array A <a:int32> [x=0:100,20,0].

11.10 xgrid

Summary: Expand single element to grid

Use this operator to scale an input array by repeating cells of the original array specified number of times. This operator can be considered as the inverse of the regrid operator. While regrid splits an input array into hypercubes and calculates an aggregate function over them, xgrid produces a hypercube from the single element of the input array by repeating that element.

Signature:

xgrid(*array*, *scale-1*, ..., *scale-N*)

The first argument to xgrid can be

1. An array.
2. An array operator, that is, an operator that outputs a SciDB array.

Example:

```
scan(m3x3);
[
[(0),(1),(2)],
[(3),(4),(5)],
[(6),(7),(8)]
]

xgrid(m3x3,2,2);

[
[(0),(0),(1),(1),(2),(2)],
[(0),(0),(1),(1),(2),(2)],
[(3),(3),(4),(4),(5),(5)],
[(3),(3),(4),(4),(5),(5)],
[(6),(6),(7),(7),(8),(8)],
[(6),(6),(7),(7),(8),(8)]
]
```

12 Matrix Algebra

12.1 inverse

The `inverse` operator produces the matrix inverse of a square matrix. The input matrix must be invertible, i.e., the determinant of the matrix must be nonzero. The inverse is calculated using LU decomposition executed on a single node (the coordinator).

Signature:

```
inverse(array)
```

Example:

1. Create a 2-dimensional array `inv_33`.

   ```
   create array inv_33 < val : double > [ I=1:3,3,0, J=1:3,3,0 ];
   store(build(inv_33, 1.0/(I+J-1)), inv_33);
   scan(inv_33);
   ```
2. Find the matrix inverse of `inv_33`:

   ```
   inverse(inv_33);
   ```
3. Remove `inv_33`:

   ```
   remove(inv_33);
   ```

NOTES:

1. `inverse`() accepts a square matrix with double data type attribute only. Other numeric attribute data types are not supported.
2. `inverse`() is not a parallel algorithm. For parallel implementations of matrix inverse please refer to the `solve`() and `svd`() operators.

See Also:

- Matrix Inverse at Wolfram's Mathworld

12.2 multiply

The multiply operator performs matrix multiplication on two input matrices and returns a result matrix.

Signature:

```
multiply(array, array)
```

Both inputs must be compatible for the multiply operation, and both must have a single attribute. To be compatible, two matrices must have the same size of 'inner' dimension and same chunk size along that dimension.

Example:

1. Create a 10-by-6 array `lhs` with one attribute of type double:

```
create array lhs <val: double> [ I=1:10,5,0, J=1:6,3,0 ];
store(build(lhs, (I*3)+J), lhs);
```

2. Create a 6-by-15 array:

```
create array rhs <val: double> [ I=1:6,3,0, J=1:15,5,0 ];
store(build(rhs, (I*3)+J), rhs);
```

3. Multiply the two arrays. The result is 10-by-15:

```
multiply(lhs, rhs);
```

4. Remove the arrays:

```
remove(lhs);
remove(rhs);
```

NOTES:

1. *multiply* supports all numeric data types (double, float, and all signed and unsigned integer data types).
2. If two input arrays contain multiple attributes the attribute to be used in the multiply operator must be obtained using project().
3. Matrix multiply currently does not accept matrices with NULL attributes. Empty cells in the inputs are treated as 0.
4. If two input matrices do not have compatible chunk sizes, the AFL repart() operator must be used to produce compatible matrices prior to multiplication.
5. multiply() does not create metadata or storage objects unless the result is added in memory with the `store`() operator.

12.3 normalize

Divide each element of a vector by the square root of the sum of squares of the elements.

Signature: normalize(*array*, *attribute : attribute_identifier*)

Example:

```
scan(v10);

[(1),(2),(3),(4),(5),(6),(7),(8),(9),(10)]

normalize(v10);

[(0.0509647),(0.101929),(0.152894),(0.203859),(0.254824),
   (0.305788),(0.356753),(0.407718),(0.458682),(0.509647)]
```

12.4 transpose

Performs the linear algebraic matrix transpose.

Signature:

```
transpose (array)
```

The transpose operator accepts an array which may contain any number of attributes and dimensions. Attributes may be of any type. If the array contains dimensions d1, d2, d3, ..., dn the result contains the dimensions in reverse order dn, ..., d3, d2, d1.

Example 1:

1. Create an array E with two dimensions i and j and int32 attribute x:

```
CREATE ARRAY E <x: int32> [ i=0:2,3,0, j=0:2,3,0 ];
store(build(E, i), E);
scan(E);
```

2. Transpose the array:

```
transpose(E);
```

3. Remove the array E:

```
remove(E);
```

Example 2:

1. Create a 2-by-3 array:

```
CREATE ARRAY E2x3 <x: int32 > [ i=0:1,2,0, j=0:2,3,0 ];
store(build(E2x3, x+1), E2x3);
scan (E2x3);
```

2. Transpose the array. The resulting array is 3-by-2:

```
transpose(E2x3);
```

---

13 Namespace Operators: Get Information About Your SciDB Database

AFL provides a number of mechanisms for getting information about the database it is managing. These facilities all interrogate the SciDB internal meta-data, and return an array object which can be subject to further query expressions.

13.1 attributes

Summary: Array attributes

Get a list of array attributes. Returns an array with the following attributes: name, type, and boolean flag. The boolean flag represents nullability of the attribute and is *True* if the attribute can be set to NULL.

Signature:

```
attributes ( array_identifier )
```

The argument to the attributes operator is the name of the array.

Examples:

```
CREATE ARRAY Attr_Example_One < A: int32, B: string > [ I=0:9,10,0 ];
attributes ( Attr_Example_One );

[("A","int32",false),("B","string",false)]
```

```
CREATE ARRAY Attr_Example_Two < A: string, B: double, C: int32 > [ I=0:9,10,0 ];
attributes ( Attr_Example_Two );

[("A","string",false),("B","double",false),("C","int32",false)]
```

Attributes are only available for stored arrays. Result arrays returned by an AFL command may not be used as input to this `attributes` operator.

13.2 dimensions

Summary: List array dimensions.

Signature:

dimensions (*array_identifier*)

The argument to the dimensions operator is the name of the array. It returns an array with the following attributes: name, start, length, chunk size, chunk overlap, low-boundary, high-boundary.

Examples:

```
create array three_dimensions <val:int64> [X=0:9,10,0, Y=0:9,10,0, Z=0:99,10,0];
```

```
store(build(three_dimensions,X+1/1.0),three_dimensions);

dimensions (three_dimensions);

[("X",0,10,10,0,0,9,"int64"),("Y",0,10,10,0,0,9,"int64"),("Z",0,100,10,0,0,99,"int64")]
```

For a regular array defined using a create array statement with fixed dimensions the *start* and *length* are set to the values specified in the CREATE ARRAY statement.

The low and high boundaries represent the current occupancy of cells within the array. Low-boundary represents the lowest dimension value occupied by any element in the array and high-boundary represents the highest dimension value occupied by an array element.

NOTE:

- The length field for an unbounded dimension is set to the maximal signed 64-bit integer.
- The low and high boundaries for a newly created array that does not yet have any data is also set to undefined values recorded as special constants (MAXINT64 - 1) and -(MAXINT64-1) respectively. These constants are interpreted as undefined for array dimension values into which no data has been loaded.

13.3 help

Summary: Operator signature

Accepts an operator name and returns an array containing a human-readable signature for that operator.

Signature: help (*operator_name: string*)

Example:

```
help('multiply');

[("Operator: multiply
Usage: multiply(<input>, <input>)")]
```

13.4 list

Summary: List contents of SciDB namspace

The `list` operator allows you to get a list of elements in the current SciDB instance.

Signature:

```
list ( element : string )
```

Arguments:

The *element : string* parameter value is one of the following:

`aggregates`	Show all operators that take as input a SciDB array and return a scalar.

`arrays`	Show all arrays created in the current SciDB instance.
`functions`	Show all functions. Each function will be listed with its available dataypes and the library in which it resides.
`libraries`	Show all libraries that are loaded in the current SciDB instance.
`nodes`	Show all nodes. Each node will be listed with its port, id number, and time and date stamp for when it came online.
`operators`	Show all operators and the library in which they reside.
`types`	Show all the dataypes the SciDB supports.
`queries`	Show all active queries. Each active query will have an id, a time and date when it was initiated, an error code, whether it generated any errors, and a status (boolean flag where TRUE means that the query is idle).

If called without any parameters, `list` will return a list of all arrays created in SciDB. Arrays persist across SciDB sessions until you re-initialize the database (`scidb.py initall` *hostname*).

Example:

```
list();

[("Reads"),("Load_and_Append"),("Load_Two"),("Example_One"),("Example_Four"),("Build_Example_INT")]

list('arrays');

[("Reads"),("Load_and_Append"),("Load_Two"),("Example_One"),("Example_Four"),("Build_Example_INT")]
```

The `list` operator returns a SciDB array, so you can be use it as the input to other data sampling operators. The following query returns all of the functions in SciDB named 'regex', together with their function signatures:

```
filter(list ('functions'), name='regex');
```

The following query uses the SciDB *regex()* function to filter the functions to find only those that take as input a pair of strings:

```
filter (list ('functions'), regex(profile, '(.*)(string,string)(.*)'));
```

And finally, the following query filters out only functions that take a pair of strings as input and compute some boolean result (like "regex"). Then it sorts that result by the function's name (the first attribute produced by *list('functions')*), and finally projects out only the function's signature:

```
project (
    sort (
        filter (
            list ('functions'),
            regex(profile, '(.*)bool(.*)(string,string)(.*)')
        ),
    1),
    profile
);
```

13.5 show

Summary: Array format

The `show` operator combines the information an array's attributes and dimensions. The result is formated as it would appear in a `create array ...` statement.

Signature:

show (*array_identifier*)

Example:

```
CREATE ARRAY Attr_Example_Two < A: string, B: double, C: int32 > [ I=0:9,10,0 ];
show ( Attr_Example_Two );

[("Attr_Example_Two 8<A:string NOT NULL,B:double NOT NULL,C:int32 NOT NULL> [I=0:9,10,0]")]
```

13.6 versions

Summary: Array versions

Get a list of array versions for an array.

Signature:

versions (*array_identifier*)

Example:

```
versions(updarr);

[(1,"12/31/10 11:03:10"),(2,"12/31/10 12:22:17"),(3,"12/31/10 12:22:19")]
```

The output of the versions command is a list of versions, each of which has a version ID and a datestamp which is the date and time of creation of that version.

---

14 Internal Commands

The commands in this chapter are for internal debugging purposes. They are not guaranteed or tested for general customer use.

14.1 cancel

Summary: Cancel a currently running query by query id.

Signature: cancel(*query_id : integer*)

Example:

```
cancel(12345);
```

The query id can be obtained from the scidb log or via the list() command. SciDB maintains query context information for each completed and in-progress query in the server. If the user issues a "ctrl-C" or abort from the client, the query is cancelled and its context is removed from the server.

In-progress queries are aborted by issuing a cancel() command with the query ID. Context information is also removed by this command.

This query context can also be removed for completed queries by issuing a cancel() command.

Query context information maintained by the server is not persistent and is lost on server restart.

14.2 diskinfo

Summary: Check disk capacity

Get information about storage space. Returns an array with the following attributes:

- used
- available
- clusterSize
- nFreeClusters
- nSegments

Signature

```
diskinfo()
```

14.3 echo

Summary: Print a string

Accepts a string and returns a single-element array containing the string.

Signature:

echo(*string*)

Example:

```
echo('Hello world');

[("Hello world")]
```

14.4 explain_logical/explain_physical

Summary: Show query plan

The operators explain_logical and explain_physical can be used to emit a human-readable plan string for a particular query without running the query itself. SciDB first constructs a logical plan, optimizes it and then translates it into a physical plan.

Signature:

explain_logical(*query: string*, *language: string*) explain_physical (*query: string*, *language: string*)

where language corresponds to the language of the query string and is either *afl* or *aql*.

Example:

```
explain_physical('store(join(a, b), c)','afl')
```

Note that the output of the explain operators is a single-element array that contains the plan string. The plan text format is intended for SciDB developers and experts and is subject to change in the future.

14.5 reduce_distro

Summary: Reduce the distribution of a replicated array

Given an array that is distributed with psReplication - quickly reduce the distribution to a different schema by masking certain chunks. For internal use.

Signature:

reduce_distro(*array*, *partitioning_schema: integer*)

Example:

```
reduce_distro(\"target@1:label\",2);

[("a"),("b"),("e"),("f"),("l"),("o"),("x"),("z")]
```

14.6 setopt

Set/get configuration option value at runtime. Option value should be specified as string. If new value is not specified, then values of this configuration option at all nodes are printed. If new value is specified, then value of option is updated at all odes and result array contains old and new values of the option at all nodes.

Signature

```
setopt ( option-name [ , new-option-value ] )
```

Example

```
setopt('threads', '4')
```

14.7 sg

Summary: Scatter or Gather an array.

The `sg` operator is used internally to redistribute or repartition the elements of an array between SciDB processing nodes. `sg` by itself does not perform any computation and its use in user queries can lead to inefficiencies and is discouraged.

`sg` redistributes chunks of the input array across cluster nodes according to a specified policy. sg is not a user-visible operator, but it is heavily used by the system internally. It is a physical operator for data movement within the system and users are not expected to invoke it directly. For example, the scidb optimizer inserts an sg operator after load to redistribute array data using round robin distribution. The optimizer may also insert sg operators to assist with aggregations, operators that don't preserve shape or operators that don't preserve distribution.

Signature

```
sg ( ''array,'' [ ''partitioning schema'' [, ''node id'' [, ''result array name''
   [, ''store_flag'' [, ''offset_x'', ''offset_y'', ... ] ] ] ] ])
```

Arguments:

Argument name	Description	Comments
input_array	array to be redistributed.	May also be the output of another operator like scan, filter, etc.
partitioning schema	integer constant specifying partitioning schema.	Possible values are shown in the table below.
node number	a node number to send all chunks.	Used only with psLocalNode. Ignored for other partitioning schemas.
result array	a name of result array.	If omitted, the result of sg is nameless and temporary with query duration.
store_flag	a boolean indicator used to specify whether sg should store the result array.	By default, true if result array name is given. If store_flag is true, the result of sg is persisted as a new array with given name. If store_flag is false, the result is a temporary named array that can

		be referenced elsewhere in the query.
offset	a distribution offset vector.	For internal use.

The partition schema is one of the following types.

partition schema	Description	Comment
0	psReplication	replicate array on every node of cluster
1	psRoundRobin	distribute chunks by round robin
2	psLocalNode	send every local chunk to specified node
3	psByRow	distribute every chunk to node according to chunk coordinate of dimension 0
4	psByCol	distribute every chunk to node according to chunk coordinate of dimension 1

Examples:

Create a query to distribute array by round robin onto every node:

```
sg(seq_info, 1, -1, seq_info_sg);
```

Create a query to gather whole array on node 0 in memory array (will not be stored on disk):

```
sg(seq_info_sg, 2, 0);
```

---

15 AQL: Array Query Language Reference

The AQL language includes two classes of queries:

- **Data Definition Language** (DDL) : queries to define arrays and load data.
- **Data Manipulation Language** (DML) : queries to access and operate on array data.

This document provides reference material for AQL.

15.1 DDL

15.1.1 CREATE ARRAY

The CREATE ARRAY command creates an array with specified name and schema.

The rest of this document will refer to the following CREATE ARRAY example:

```
CREATE ARRAY A <x: double NOT NULL, err: double> [i=0:99,10,0, j=0:99,10,0];
```

This creates a 100-by-100 array with a chunk size of 10 in each dimension and no chunk overlap.

15.1.2 LOAD

Loading data into arrays is done using the load command.

The syntax for LOAD is:

LOAD *array_name : string* FROM *file_path : string*;

Example:

```
LOAD A FROM '/tmp/A.txt';
```

The data file formats supported by SciDB are documented in the AFL document. The file path 'A.txt' is assumed to be a path name relative to the coordinator's working directory. For this document, the file A.txt is a 100-by-100 file with two numerical values per cell.

Note: AQL only supports coordinator load (corresponding to nodeid=0 in the AFL *load()* command). See the section on *load()* in the AFL documentation.

15.1.3 DROP

Deleting arrays is done with the `drop` command.

DROP ARRAY array_name;

For example to drop array A, run the following command:

```
DROP ARRAY A;
```

The array and all its versions are dropped. Any associated mapping arrays created for noninteger dimensions are also dropped.

Note: Many AQL commands do not create or accept arrays with noninteger dimensions.

15.2 DML Query Syntax

15.2.1 SELECT attributes FROM

As in SQL, AQL SELECT is used to select one or more attributes from an array.

Syntax: SELECT "expr_list | *" [INTO name] FROM "array_name | array_result"

For example:

```
SELECT * FROM A;
```

This is a basic query the selects all attributes from the whole array A. SELECT lists can also reference individual attributes and dimensions, as well as constants and expressions. Expressions are discussed later in this document.

```
SELECT err FROM A;
```

This is a SELECT query that selects the attribute `err` from A. To store the value of the attribute `err` in and array called C:

```
SELECT err INTO C FROM A;
```

This query selects the attribute `err` from the array A and stores the result in array C. If the array C does not exist, an array with the appropriate schema is created for it. If C exists, its attribute values are updated.

If an array exists in SciDB, but its schema has different chunking and overlap specifications, or some dimensions in the schema are declared as noninteger dimensions, then the repart() or redimension() is applied to the result before storing it into the destination array.

In place of a table *A* or a join result, SELECT can also be used to select data from an AFL operator result. For a full list of AFL operators, please refer to the "SciDB Array Functional Language Reference" document. Shown below is such an example with the "show" operator which return an array containing the schema of *A*.

Example:

```
SELECT * FROM show(A);
```

15.2.2 WHERE Clause

A WHERE clause is used to filter cells from the input array.

Syntax: SELECT "expr_list | *" [INTO name] FROM "array_name | array_result" [WHERE expr]

For example:

```
SELECT * FROM A WHERE err < 2;
```

The above query selects all cells in the array whose error is less than 2. The shape of the array is preserved by the WHERE clause. The user will see a result with the same shape as A, except the cells that don?t match the condition of the WHERE clause will be empty.

Note that the where clause can be used to check if a particular attribute is NULL. This is done with the 'IS NULL' and 'IS NOT NULL' construct.

```
SELECT x FROM A WHERE x IS NOT null;
```

SciSB supports SQL semantics for null values. Hence, any comparison or use of null values in expressions will evaluate to NULL, except for the ?is null? or 'is not null' predicates. For a description of NULL values in SciDB please refer to the section on "CREATE ARRAY" in the AFL document.

In contrast, EMPTY elements of an array are not evaluated by the WHERE clause.

15.2.3 AQL Expressions

AQL expressions in the SELECT list or the WHERE clause are standard expressions over the attributes and dimensions of the array. They can also refer to special built-in functions supported by SciDB. The full list of built-in functions is available in the Alphabetical List of Functions appendix to this document.

```
SELECT x FROM A WHERE sin(x)<0;
```

returns all values of the attribute x (in radians) where sin(x) is negative.

```
SELECT sqrt(x) FROM A WHERE sin(x)<0;
```

returns the square root of all values of the attribute x for which sin(x) is negative.

15.2.4 Natural JOIN

A join combines two or more arrays typically as a preprocessing step for subsequent operations. The following example illustrates SciDB JOIN.

Syntax:

SELECT "expr_list | *" [INTO name] FROM joinexpr [WHERE expr]

joinexpr := expr ("," expr))*

Consider an array B, that has the same shape as array A declared earlier:

```
CREATE ARRAY B <y: double, err2: double> [i=0:99,10,0, j=0:99,10,0];
```

The following query returns an array by joining arrays A and B.

```
SELECT * FROM A, B;
```

This join is a natural join of array A with B. This is only possible with two arrays with matching shape. The arrays must have the same number of dimensions, matching dimension start and endpoints, and same chunk size. But dimension names need not match. The dimensions of the resulting array are the same dimensions as the inputs but with the dimension names from the first array.

The resulting array also has combined attributes of all inputs. Each cell in the resulting array in this example has 4 attributes, x, err from A and y, err2 from B. Natural join is useful when you need to merge two arrays with the same shape in order to perform further processing on their attributes.

The following example joins two arrays in order to compute the sum of two attributes for each element.

```
SELECT A.x + B.y FROM A,B;
```

15.2.5 JOIN ON

AQL JOIN ON is used to join two compatible arrays.

The JOIN ON returns an array that is the cross product join of its inputs. Only JOIN ON equality predicates are supported. Predicates must use compatible dimensions from the input arrays -- size, types and chunking.

Each cell in the output array contains all elements from the join inputs from corresponding dimension values.

AQL currently supports dimension-dimension, attribute-attribute and attribute-dimension JOINs.

15.2.5.1 Dimension-dimension join

In a dimension-dimension join, two or more arrays can be joined on dimensions. This is translated by AQL into AFL cross-joins. JOIN ON dimension predicates are compiled into the dimension pairs list for cross-join. Any aliasing required is performed by the AQL-to-AFL compiler.

Example: Consider the following example 3-by-3 arrays `v1` and `k`.

```
CREATE array v1 <a:double, b:int64> [x=1:3,3,0, y=1:3,3,0];
CREATE ARRAY k <c: string, d:double> [x=1:3,3,0, y=1:3,3,0];

scan(v1)
[
[(1,1),(1,2),(1,3)],
[(2,1),(2,2),(2,3)],
[(3,1),(3,2),(3,3)]
]

scan(k)
[
[("addr_11",11),("addr_12",12),("addr_13",13)],
[("addr_21",21),("addr_22",22),("addr_23",23)],
[("addr_31",31),("addr_32",32),("addr_33",33)]
]
```

Performing a join on the first dimension creates a 3-by-3-by-3 array:

```
set lang AQL;
DROP array res_dim;
SELECT * INTO res_dim FROM v1 JOIN k ON v1.x = k.x;

[
[
[(1,1,"addr_11",11),(1,1,"addr_12",12),(1,1,"addr_13",13)],
[(1,2,"addr_11",11),(1,2,"addr_12",12),(1,2,"addr_13",13)],
[(1,3,"addr_11",11),(1,3,"addr_12",12),(1,3,"addr_13",13)]
],
[
[(2,1,"addr_21",21),(2,1,"addr_22",22),(2,1,"addr_23",23)],
[(2,2,"addr_21",21),(2,2,"addr_22",22),(2,2,"addr_23",23)],
[(2,3,"addr_21",21),(2,3,"addr_22",22),(2,3,"addr_23",23)]
],
[
[(3,1,"addr_31",31),(3,1,"addr_32",32),(3,1,"addr_33",33)],
[(3,2,"addr_31",31),(3,2,"addr_32",32),(3,2,"addr_33",33)],
[(3,3,"addr_31",31),(3,3,"addr_32",32),(3,3,"addr_33",33)]
]
]
```

The output of the join on statement above is an array with the following schema:

```
<a:double NOT NULL,b:int64 NOT NULL,c:string NOT NULL,d:double NOT NULL>
     [x=1:3,3,0,y=1:3,3,0,y_2=1:3,3,0]
```

The new generated dimension name is y_2.

15.2.5.2 Attribute-attribute JOIN

The JOIN ON predicate may have a list of equality predicates based on array attributes. If this is done, the inputs are first transformed into temporary stored arrays using the redimension_store() operator and then joined using AFL cross_join(). Any additional arrays created to execute the JOIN are removed at the end of query execution.

The following shows an example of attribute-attribute join:

```
SELECT * INTO res_attr FROM v1 JOIN k ON v1.a = k.d;
```

The above query produces a result with the following schema:

```
<a:double NOT NULL,b:int64 NOT NULL,c:string NOT NULL,
     empty_indicator:indicator NOT NULL> [x=1:3,3,0,y=1:3,3,0]
```

```
SELECT * FROM vector3 A JOIN vector3 B ON A.a = B.b and A.b = B.a;
```

produces a result with the following schema:

```
<a:int64 NOT NULL, b:int64 NOT NULL, empty_indicator:indicator NOT NULL>
[i=*:*,10,0, i_2=*:*,10,0]
```

15.2.5.3 Dimension-Attribute Join

A dimension-attribute join can be performed between attributes and dimensions that have the same data type.

```
SELECT * INTO result FROM v1 JOIN k ON v1.b = k.x;
[
[
[(2,1,"addr_11"),(),()],
[(),(3,2,"addr_22"),()],
[(),(),(4,3,"addr_33")]
],
[
[(),(3,2,"addr_22"),()],
[(),(),()],[(),(),()]],
[
[(),(),(4,3,"addr_33")],
[(),(),()],[(),(),()]
]
]
```

15.2.6 Aggregates

AQL supports the following built-in aggregate functions:

Name	Operation Performed
avg	Average value
count	Number of nonempty elements
max	Largest value
min	Smallest value
sum	Sum of all the elements
stdev	Standard deviation
var	Variance

The scalar aggregate functions in AQL take an input array and produce a scalar result. The value for the result is the aggregate function applied to the entire array.

For example:

```
SELECT max (SELECT a1 FROM a);
```

Each array aggregate function accepts one attribute of appropriate type. For example, to compute the maximum value of an attribute, that attribute type must support inequality and equality functions. So, for the max aggregate, the data types that can be used are

> integers (int8, int16, int32, int64, uint8, uint16, uint32, uint64) and floating point (double,float) types.

count here is an aggregate function that takes an array as input and returns a scalar value in its output.

Below is a query that counts all the measurements of array A:

```
SELECT count(A);
```

Note: While all aggregates can be calculated in this manner over the entire array typically on one attribute, the *count* function is an exception. In the AFL version, you can use a dimension name as an optional input into the count function.

- When `count()` is used without any arguments, the result is a

count of the number of non-empty elements in the array.

- When `count` is supplied

with a dimension name, count counts the number of non-null occurrences of that attribute within the array.

15.2.7 GROUP BY Aggregates

AQL aggregates can also be used with the GROUP BY clause. The syntax for a group by aggregate is as follows.

SELECT "namedExprList | *" [INFO name] FROM joinexpr [WHERE expr] [groupBy];

groupBy := "group by" dimensionList "as" pathExpr

dimensionList := dim ("," dim)*

AQL GROUP BY takes a list of dimensions as a "group by" list, groups the array by the specified dimensions and computes the aggregate function for each group. The result of aggregate is an array with same dimensions as the source array without the group by dimensions. The result array has a single attribute which is the aggregate result per group.

The result of a "GROUP BY x, y" is an array with only x, y dimensions and a new attribute which is the corresponding partition. This partition can be accessed by sub-queries or expressions in the SELECT aggregate clause. The following examples explain the GROUP BY aggregate.

15.2.7.1 Example 1. count group by

Suppose we want to first group measurements by the first dimension, then calculate the count over each of the groups, the AQL query will look like this:

```
SELECT count(part)
FROM A
GROUP BY i AS part;
```

The group by clause here creates a nested array for each group and names the nested array *part.* Part is a one-dimensional array corresponding to a given value of dimension *i* and attributes of the input array *A*.

15.2.7.2 Example 2. group by avg over an attribute

Now suppose we need to compute the average measurement from A, grouped by i we issue the following AQL query which selects the attribute to be used as input to the average. AQL also supports standard SQL-like syntax for computing the aggregate over array groupings. In the current release of SciDB, the AQL GROUP BY must provide a name for the partition attribute created in the GROUP BY aggregate. See example below.

```
SELECT avg(x)
FROM A
GROUP BY i AS T;
```

15.2.8 Nested sub-queries

In this release nested queries can either be simple aggregate queries in the SELECT clause, or nested queries in the FROM clause. Other kinds of nested queries will be supported in forthcoming releases

In the example below, we select partitions whose average value is greater than 10:

```
SELECT * FROM
(
SELECT avg (SELECT x FROM part)
FROM A
GROUP BY i AS part
)
WHERE avg > 10;
```

15.2.9 Updates and Versions

AQL arrays that are updatable can be updated using the following command:

UPDATE array SET "attr = expr", ... [WHERE condition];

For example, to add 1.0 to each measurement in array A and add 0.05 to each error, we can write the following query:

```
UPDATE A SET x = x+1, err = err + 0.05;
```

Update can be used with a WHERE clause. To increase the measurement only if the error is smaller than 0.05:

```
UPDATE A SET x = x+1 WHERE err < 0.05;
```

Note that the UPDATE clause can be used to update non-null elements using the following example:

```
SELECT * FROM A;
[(), (2), (3)]
```

```
UPDATE A SET a = a + 1 WHERE a IS NOT NULL;
[(), (3), (4)]
```

```
UPDATE A SET a = 22222 WHERE a IS NULL;
[(22222),(3),(4)]
```

For a definition of NULLs in the SciDB array data model please refer to the AFL documentation for this release.

SciDB saves all versions of the data, so the user has an option to query data as it was in the database at a specific time. This is done by appending a @ datetime prefix after the name of the array. For example, the following query returns data as of November 11 2010:

```
SELECT * FROM A @ datetime('2011-12-15 22:33:12');
```

If multiple versions have the same datetime, the most recent version, i.e., the one with the highest version number is returned. The timestamp of a version is the time on the SciDB server, which is by default reported in UTC timezone.

To set the time zone, use the datetimetz argument.

For convenience, SciDB includes a now() function, so one can write a query to retrieve all data that was in the array 1000 miliseconds prior to current date and time:

```
SELECT * FROM A @ now() - 1000;
```

Any expression yielding a datetime value is supported after the @ operator.

15.2.10 Aliases

AQL provides a way to refer to arrays in the query via aliases. These are useful when using the same array repeatedly in an AQL statement, or when abbreviating a long array name. Aliases are created by adding an "as" to the array name, followed by the alias. Future references to the array can then use the alias.

Example:

```
SELECT * FROM vector3 AS A;
```

Once an alias has been assigned, all attributes and dimensions of the array can use the fully qualified name using the dotted naming convention.

```
SELECT A.x + 10 from vector3 AS A;
```

For example:

```
SELECT pow(sin(a1), 2) as p1, pow(cos(a1), 2) as p2 FROM A;
```

16 SciDB Plugins: Extending SciDB Functionality

User-defined extensions to SciDB functionality are referred to as *plugins*. SciDB supports user-defined functions (UDFs), user-defined types (UDTs), and user-defined operators (UDOs).

16.1 Extensibility: Types and Functions

Out of the box, SciDB provides users with a standard set of data types; integer, float/double, and string. Scientific and large-scale analytic applications often require other data types such as complex numbers, rational numbers, two-dimensional points, or others. Some applications call for specific mathematical functions (such as greatest common factor of two integers or non-uniform random number generation) that SciDB does not provide by default. SciDB's extensibility mechanism allows users to add their own implementation of types and functions to the SciDB engine.

Suppose a SciDB application requires a rational number datatype. Rather than use double precision, the user wants to store an integer-type numerator and denominator pair. As part of the the new type's functionality users will also want to support basic arithmetic (+, -, *, /) and logical (<, <=, =, >=, >, <>) functionality.

At the level of the AQL query language, the new type can be used as follows:

```
create array rational_example < N : rational > [ I=0:99,10,0, J=0:99,10,0 ]

# Q1:

SELECT COUNT(*)
   FROM rational_example AS R
 WHERE R.N = rational(1,2);

# Q2:

SELECT str(R.N)
  FROM rational_example AS R
 WHERE R.N + rational(1,4) > rational(1,2);
```

So far as a user's queries are concerned, there will be no difference between the way a built-in type and a user-defined type (or function) works. There are, however, a couple of things to be aware of:

1. All type conversions need to be explicit. SciDB does not (yet) support implicit casting.
2. Client applications can only accommodate a limited set of types: doubles, integers, and strings. When you write queries (using iquery, say) the query's result needs to explicitly convert result types into something that the client understands.
3. While it is quite possible to write complex and computationally expensive UDFs (we include an example of a prime number factorization) it's generally a better practice to build UDFs as small, self-contained units of functionality and then to combine them using the SciDB query language's facilities.
4. We do not (yet) support features like embedding queries within UDFs, or plugins that do anything more sophisticated than take a vector of scalar values and return a single scalar type result.

16.2 SciDB Plugin Examples

SciDB includes multiple example extensions in the ~/examples folder located beneath the SciDB root directory. These examples are:

Name	Description
complex	Complex Number UDT (a + b.i), together with the associated algebraic operations, and equality.
rational	Rational Number UDT (int64 numerator and denominator) together with the associated algebra operations and ordering comparisons
point	2-D Point UDT. Double precision X and Y
more_math	A selection of user-defined functions which perform useful mathematical operations.

16.3 SciDB Plugins Architecture

The basic architecture of a SciDB Plugin works as follows. The algorithms implemented within the SciDB engine are designed to treat instances of data type values as *black box* memory segments. For example, all that the SciDB engine "knows" about the contents of the Complex Number data type is that it is 16 bytes long. The code that needs to know about the contents of these 16 bytes is implemented by the user in their own C/C++, which they compile into a shared library.

At run-time, the SciDB engine dynamically links this shared library and calls the functions it contains to perform the operations specified in the query.

For example, the user-written 'C' code to add two complex numbers looks like this:

```
//
// This is the struct that describes how the 16 bytes of data that makes up an instance
// of a SciDB Complex UDT is organized.
struct Complex
{
    double re;
    double im;
};

//
// This is the code that takes data from the SciDB engine, performs the addition, and deposits
// the return result in an appropriately sized "black box" of bytes. The SciDB engine takes
// this return result and stores it, or passes it on to another function.
static void addComplex(const Value** args, Value* res, void*)
{
    Complex& a = *(Complex*)args[0]->data();
    Complex& b = *(Complex*)args[1]->data();
    Complex& c = *(Complex*)res->data();

    c.re = a.re + b.re;
    c.im = a.im + b.im;
}
```

When it parses a query like the one labeled "Q2" above, the SciDB engine checks to ensure that it had been provided with a shared library containing code to perform the plus (Type, Type) -> Type operation. In this case, SciDB would look for a function named "+" that took two arguments of the appropriate type (in this case, a pair of complex number instances). Then at run time the SciDB engine would assemble the necessary 16-byte "black boxes", invoke the function 'C' addComplex(), and deal with the value it computed.

16.4 User-Defined Functions: How SciDB Provides Datatype Instances

As you can see from the example code above, SciDB uses a typesystem::Value class to encapsulate information about all type value instances. The Values class provides a set of methods for getting and setting the "value" of the class for each of the SciDB built-in types; getType() and setType(), or more explicitly (in the case of a typesystem::Value val; instance that contains a string) val.getString() and val.setString().

From the perspective of the SciDB engine, all UDFs have the same basic signature:

```
void functionUDF(const Value** args, Value* res, void*)
{

}
```

Each function must "know" how many arguments it is to receive. These arguments are obtained from the vector of typesystem::Value pointers that makes up the first argument. Each UDF (currently) returns a single result and the location where this result is to be placed is passed in by reference in the second argument. The final argument is a pointer to a data structure that conveys information about the state of the engine, and is a means of passing data between repeated calls to the UDF within the same query.

16.5 Loading a Plugin

Each of these example plugins included with the SciDB distribution is built (by default) at the time we build the core engine. However, SciDB does not load unregistered plugins when it starts up. To use one of the examples you need to load it into the SciDB instance. The following figure illustrates how to load shared libraries containing plugins into SciDB using first, the AFL interface, and second, our AQL query language.

```
--
-- AFL load_library() operation
load_library ( 'librational.so' )

--
-- AQL 'load library' syntax
load library 'librational.so';
```

The act of loading a plugin shared library first registers the library in the SciDB system catalogs. Then it opens and examines the shared library to store its contents with SciDB's internal extension management subsystem. Shared library module which are registered with the SciDB instance will be loaded at system start time.

If you want to unload library run:

```
--
-- AFL unload_library() operation
unload_library('libpoint1')

--
-- AQL 'unload library' syntax
unload library 'libpoint1'
```

This command will unregister the library in the system catalog. The library will not be loaded on consecutive restart, but it might not be unloaded immediately because some queries can be using it.

16.6 Tutorial: Creating SciDB Plugins

This section explains the steps needed for creating a new plugin for SciDB.

16.6.1 Designing your UDT

Your UDT will need the following kinds of UDFs.

1. UDFs that construct instances of your new type based on the values of other types. In general the types you will use as input to these UDFs will be built in types. For example, it is typical to use a string as a source for a new data type's contents. For example, the following UDF converts a string with a particular format into an instance of a rational number UDT.

```
//
// This is the struct used to store the data inside SciDB.
typedef struct
{
    int64_t num;
    int64_t denom;
} SciDB_Rational;

void str2Rational(const Value** args, Value* res, void*)
{
        int64_t n, d;
    SciDB_Rational* r = (SciDB_Rational*)res->data();

    if (sscanf(args[0]->getString(), "(%"PRIi64"/%"PRIi64")", &n, &d) != 2)
        throw PLUGIN_USER_EXCEPTION("librational", SCIDB_SE_UDO, RATIONAL_E_CANT_CONVERT_TO_RATIONAL)
            << args[0]->getString();

        if ((0 == d) && (0 == n))
                d = 1;

        boost::rational<int64_t>rp0(n, d);
        r->num   = rp0.numerator();
        r->denom = rp0.denominator();

}
```

Note that the "string to UDT" conversion function is particularly important. type (string) -> type is the UDF used by the load() operation to bulk ingest data into SciDB.

2. UDFs that convert your UDT back into a built-in type, or a number of built-in types. In the case of the complex type, for example, you can either write a UDF that composes the 16 bytes into a string, or else a pair of UDFs that extract the real and imaginary portions of the type.

```
static void reComplex(const Value** args, Value* res, void*)
{
   Complex& a = *(Complex*)args[0]->data();
   res->setDouble(a.re);
}
```

```
static void imComplex(const Value** args, Value* res, void*)
{
   Complex& a = *(Complex*)args[0]->data();
   res->setDouble(a.im);
}
```

Remember that SciDB will not perform implicit casting. You need to include these UDFs in any queries that pull these values out of database.

3. UDFs that perform common type operations, such as simple arithmetic or relational operations will not need to support all datatypes. While it makes sense to support the full set of relational operators for a datatype that can be ordered (such as rational number), a values in the complex domain cannot be ordered (for sorting, say). All SciDB needs for ordering are two UDFs: one to return TRUE when two type values are equal, and a second to return TRUE when one type is less than another.

```
void rationalLT(const Value** args, Value* res, void * v)
{
    SciDB_Rational* r0 = (SciDB_Rational*)args[0]->data();
    SciDB_Rational* r1 = (SciDB_Rational*)args[1]->data();

        check_zero ( r0 );
        check_zero ( r1 );

        boost::rational<int64_t>rp0(r0->num, r0->denom);
        boost::rational<int64_t>rp1(r1->num, r1->denom);

        if ( rp0 < rp1 )
        res->setBool(true);
        else
        res->setBool(false);
}

void rationalEQ(const Value** args, Value* res, void * v)
{
    SciDB_Rational* r0 = (SciDB_Rational*)args[0]->data();
    SciDB_Rational* r1 = (SciDB_Rational*)args[1]->data();

        check_zero ( r0 );
        check_zero ( r1 );

        boost::rational<int64_t>rp0(r0->num, r0->denom);
        boost::rational<int64_t>rp1(r1->num, r1->denom);

        if ( rp0 == rp1 )
        res->setBool(true);
        else
        res->setBool(false);
}
```

4. UDFs that are necessary to support the integration of the UDT with other facilities; aggregates like AVG(), MAX() and MIN(), for example. MAX() and MIN() use the UDFs that order instance values. If your type has a peculiar requirements for MAX() and MIN(), it might be reasonable to add these UDFs.

16.6.2 Exceptions and Error Handling

Your UDFs will often need to check for errors and exceptions in their code. In SciDB, we provide facilities to report to the SciDB engine that your UDF has encountered an error, and what kind of error. Doing this allows the SciDB engine to terminate the query and report some useful status information to the log file. Errors and exceptions are thrown using a macro USER_EXCEPTION(error_code, description : string).

```
    throw PLUGIN_USER_EXCEPTION(<plugin name>, SCIDB_SE_UDO, \
    <plugin error code>) << <args>;
```

For example,

```
void str2Rational(const Value** args, Value* res, void*)
{
        int64_t n, d;
    SciDB_Rational* r = (SciDB_Rational*)res->data();

    if (sscanf(args[0]->getString(), "(%"PRIi64"/%"PRIi64")", &n, &d) != 2)
        throw PLUGIN_USER_EXCEPTION("librational", SCIDB_SE_UDO, RATIONAL_E_CANT_CONVERT_TO_RATIONAL)
            << args[0]->getString();

        if ((0 == d) && (0 == n))
                d = 1;

        boost::rational<int64_t>rp0(n, d);
        r->num   = rp0.numerator();
        r->denom = rp0.denominator();

}
```

For a full list of the terse error codes that you can throw from within a UDF, consult the '~/include/system/ErrorCodes.h' file.

16.6.3 Registering Your 'C' Functions as UDFs

Once you have implemented your functions, you should register them with the SciDB facilities for extracting information from a shared library. The SciDB install provides a set of 'C' macros to do this. These macros are:

Macro Name	*Description*	Example
REGISTER_TYPE (name, length)	Instructs SciDB to register a new UDT in it's catalogs with the name provided (note that this argument to the macro is not a string) and the length, in bytes, of the type instance values.	REGISTER_TYPE (complex, 16)
REGISTER_FUNCTION (name, input argument types, output argument type, function pointer)	Instructs SciDB to register a new UDF in its catalogs. The new UDF can be called in AQL or AFL using the first argument name (again, not a string), the the function is expected to take a list of	REGISTER_FUNCTION(+, ("complex", "complex"), "complex", addComplex);

	argument types as input, and return a value of the type provided. The actual reference to the function you want SciDB to call is the last argument to the macro.	
REGISTER_CONVERTER(input type, output type, conversion cost, function pointer)	From time to time SciDB needs to convert types, and it can require UDFs to perform this operation. This macro is how you register conversions.	REGISTER_CONVERTER(string, complex, EXPLICIT_CONVERSION_COST, string2complex);

16.6.4 A Simple Recipe

The simplest way to implement your own plugin library is to copy the style of the examples.

1. Create a new directory in parallel to the one that implements one of the examples, say the complex type.

In the ~/examples/CMakeLists.txt file, add new line with name of new directory. Let's say the new directory is named "point1"

```
add_subdirectory("complex")   <-- already exists.
add_subdirectory("point1")       <-- the reference to your new plugin directory
```

At this point you will want to rename `point1/complex.cpp` to something more appropriate for the purposes of the library.

2. Change the contents of the new `~/examples/point1/CMakeLists.txt` file to get the server to build your new plugin library.

3. Make your modifications in the new source code file: `~/examples/point1/point.cpp`.

4. Using "make", build "libpoint1.so". It will be placed into the plugins directory folder alongside "libcomplex.so".

5. Load your new library module.

16.7 User-Defined Operators

The most complicated user-defined objects are user-defined operators. Every operator in SciDB is a pair of objects:

- A *logical operator* class, and
- A *physical operator* class.

The main purpose of logical operator is:

- to infer an array schema, and
- to provide information about expected inputs and parameters of the operator.

Ideally, the logical operator is common to every operator of the same class. However, the logical operator can have several implementations called *physical operators*. The main purpose of physical operator to execute operator implementation.

Every operator, logical or physical, can have a *state*. States are created by special factory methods. Every instance of an operator is a new instance of the class. This means that you can add a new field to inherited classes.

16.7.1 Creating a User-Defined Operator

The easiest way to create a new operator is to find the closest built-in operator, copy-and-paste it into a separate folder, and change the existing implementation into the desired implementation.

In the `example/operators` directory in your SciDB build you can find a stub example for creating a plugin with user-defined operators. You can replace the example stubs by the built-in operator implementation that is closest to what you want your new operator to do and then rename internal classes and operator names.

The following sections provide short descriptions of base classes for logical and physical operators and descriptive comments about class members.

16.7.1.1 Logical Operator Example

The logical operator must be inherit from the LogicalOperator class and implement the methods constructor and inferSchema:

```
class LogicalStub : public LogicalOperator
{
public:
    LogicalStub(const std::string& logicalName, const std::string& alias):
        LogicalOperator(logicalName, alias)
    {
        /**
         * See built-in operators implementation for example
         */
    }

    ArrayDesc inferSchema(std::vector<ArrayDesc> schemas, boost::shared_ptr<Query> query)
    {
        /**
         * See built-in operators implementation for example
         */
        return ArrayDesc();
    }

};
```

The constructor contains code for the declaration of possible inputs and parameters. For example, the APPLY operator has the following constructor:

```
   Apply(const std::string& logicalName, const std::string& alias):
       LogicalOperator(logicalName, alias)
   {
       _properties.tile = true;
       ADD_PARAM_INPUT()
       ADD_PARAM_OUT_ATTRIBUTE_NAME("void")//0
```

```
        ADD_PARAM_EXPRESSION("void")        //1
        ADD_PARAM_VARIES()
    }
```

- `properties.tile` is true if operator can work in tile mode.
- `ADD_PARAM_INPUT()` says that operator expects one more input (in this case, an input array).
- `ADD_PARAM_OUT_ATTRIBUTE_NAME("void")` says that the operator will add new attribute with the given data type ("void" means "any").
- `inferSchema` will produce the real data types based on input schema.
- `ADD_PARAM_EXPRESSION("void")` says that operator expect one expression with "any" ("void") data type. You may add other attributes and attribute kinds as well.
- `ADD_PARAM_VARIES()` means that APPLY can have a variable number of parameters. In this case you need to implement one more virtual method `nextVaryParamPlaceholder`. See the APPLY implementation for example.
- `inferSchema` provides the schema for resultant array.

16.7.1.2 Physical operators

Physical operators must inherit from the PhysicalOperator class and implement the `execute` method:

```
class PhysicalStub: public PhysicalOperator
{
public:
    PhysicalStub(const std::string& logicalName, const std::string& physicalName,\
    const Parameters& parameters, const ArrayDesc& schema):
            PhysicalOperator(logicalName, physicalName, parameters, schema)
        {
        }

    shared_ptr<Array> execute(std::vector<shared_ptr<Array> >& inputArrays,\
    shared_ptr<Query> query)
        {
        /**
         * See built-in operators implementation for example
         */
        return shared_ptr<Array>();
        }
};
```

For example, here is the APPLY operator:

```
boost::shared_ptr<Array> execute(vector< boost::shared_ptr<Array> >& inputArrays,\
boost::shared_ptr<Query> query)
{
    assert(inputArrays.size() == 1);
    assert(_parameters.size()%2 == 0);

    vector<shared_ptr<Expression> > expressions(0);

    size_t currentParam = 0;
    for(size_t i =0; i< _schema.getAttributes().size(); i++)
    {
        assert(_parameters[currentParam]->getParamType() == PARAM_ATTRIBUTE_REF);
        assert(_parameters[currentParam+1]->getParamType() == PARAM_PHYSICAL_EXPRESSION);

        string const& schemaAttName = _schema.getAttributes()[i].getName();
        string const& paramAttName = \
```

```
            ((boost::shared_ptr<OperatorParamReference>&)_parameters[currentParam])->\
            getObjectName();

            if(schemaAttName!=paramAttName)
            {
                expressions.push_back\
                ( shared_ptr<Expression> ());
            }
            else
            {
                expressions.push_back(((boost::shared_ptr<OperatorParamPhysicalExpression>&)\
               _parameters[currentParam+1])->getExpression());
                currentParam+=2;
            }

            if(currentParam == _parameters.size())
            {
                for (size_t j = i+1; j< _schema.getAttributes().size(); j++)
                {
                    expressions.push_back( shared_ptr<Expression> () );
                }
                break;
            }
        }

        assert(currentParam == _parameters.size());
        assert(expressions.size() == _schema.getAttributes().size());

        boost::shared_ptr<Array> input = inputArrays[0];
        return boost::shared_ptr<Array>(new ApplyArray(_schema, input, \
        expressions, query, _tileMode));
    }
```

The `execute()` method takes a number of input arrays and query context. It can use all methods of input arrays and perform any evaluations. The result must be a new array instance.

It is also possible to create a pipelined array instance which will perform evaluations only when data will be requested. For example, you may want to evaluate a chunk only when the `getChunk` method is called. ApplyArray in the above code is an example of such an array.

---

17 Connectors

17.1 SciDB Client API for Python

SciDB uses an ODBC/JDBC like interface to connect to the SciDB server and execute commands. This interface is available from multiple computer languages. This page documents the Python version of the SciDB API.

To get access to the API add the following to your Python file:

```
import sys
sys.path.append('/opt/scidb/11.10/lib')  # or location appropriate to your installed version
import scidbapi as scidb
```

This imports the scidbapi module, scidbapi.py, which defines an interface to scidb. This interface is implemented using several Python and C++ libraries beneath it: libscidbpython.py (generated by the SWIG compiler) contains python classes that are proxies for C++ classes, _libscidbpython.so which was also generated by SWIG and provides some of the conversion between the Python API and C++, and libscidbclient.so, which implements the C++ remote client library for SciDB.

You can then list information about the scidbapi module with the Python statement:

```
help(scidb)
```

17.2 Example Python Application

Two sample python applications are provided in the `/opt/scidb/11.12/share/scidb/examples/python` directory of a server installation. These are also located at src/capi/pythonexamples directory of the SciDB sources, available to registered SciDB developers. The example files are:

- README
- sample.py is a program that creates and loads an array, executes a select AQL statement, and drops the array.
- sample2.py takes in a list of afl/aql queries from a file (or files) and executes them. This example shows the use of additional data types and the empty flag for queries with filter predicates.
- simplearray.data is read by sample.py
- sample2.csv is read by sample2.py

NOTE: You may find other contributed examples in the src/capi/pythonexamples; however, they will not be given the level of attention for maintaining correctness as sample.py and sample2.py and may be written in older versions of the API.

NOTE: The python API will probably move to its own directory in the source tree (out of the capi directory) in the near future.

17.3 Example: Connect and Execute a Simple Query

You connect to SciDB using connect() and execute queries using executeQuery(). Connect() takes a server address and the port number for the SciDB coordinator.

```
db = scidb.connect("localhost", 1239)
# connect to the SciDB coordinator.
result = db.executeQuery("drop array simplearray", 'aql')
# execute an AQL query
```

You then iterate over result to obtain the result data. See the section on Array and Chunk Iterators, below.

17.4 Create and Load Queries

executeQuery (statement, type, result, handle)

Arg	Description
statement	Valid AQL or AFL statement.
type	scidb.AQL or scidb.AFL.
result	Each query requires a new QueryResult structure on the client. QueryResult is described in the section below on Data Access.
handle	connection handle

The examples below show how to execute create and load queries.

Create an array *simplearray*.

```
db.executeQuery("CREATE immutable ARRAY simplearray
     < foo:int32, bar:char, baz:string > [row=0:99,10,0, col=0:9,10,0]", "aql")
```

Load data into this array. The data file must be visible on the server's file system. A relative path to the file will be interpreted relative to the working directory of the server. This will be appropriate if the data were saved from the same server. In other cases, it may be more appropriate to use an absolute path to files to be loaded.

```
db.executeQuery("load simplearray from 'simplearray.data'", "AQL")
db.executeQuery("select * from simplearray", "AQL")
```

17.5 Data Access

17.5.1 Query Result

Accessing the schema of the result set is performed through a set of python objects accessible through the QueryResult.

Use the following help commands to get more information

```
help(scidb.swig.QueryResult)
```

Arg	Description
array	Handle to the array object, its iterators and descriptors returned by the server.
queryID	Query ID as known to the server. It is valid after the successful execution of a statement and may not be re-used.
selective	Indicates if a data retrieval command was executed.
executionTime	Execution time of this query.
explainLogical	Logical plan used.
explainPhysical	Physical plan used.

17.5.2 Array, Attribute, and Dimension Descriptors

Additional information such as the dimensions and attributes of the result array are accessible from objects in the array. For more information use help on the following classes:

```
help(scidb.swig.ArrayDesc)
help(scidb.swig.AttributeDesc)
help(scidb.swig.DimensionDesc)
```

17.5.3 Array and Chunk Iterators

The data access API is based on the nested array data model of SciDB.

A SciDB array is returned to the caller as a collection of *chunks* that together represent the array. Array and chunk iterators must be used to scan all cells of the array.

Each attribute of the array can be accessed using a separate set of iterators. Attribute iteration is done at two levels: an outer iteration of chunks of the array and an iteration of cells in a chunk. The array iterator iterates over the chunks in dimension major order, as does the chunk iterator. That is to say that both arrays and chunks are multidimensional and data is returned in first to last dimension order (e.g., row-major order for a 2d array).

The data access API includes the following objects.

```
Array
ArrayDesc
AttributeDesc
DimensionDesc
ConstArrayIterator
ConstChunkIterator
ConstChunk
Coordinates
Value
```

The following example shows how to iterate over all chunks of an array, and all elements of each chunk.

```
    chunkiters = []
    for i in range(attrs.size()):
        nc = -1
        while not iters[i].end():
            nc += 1
```

```
            chunkiter = iters[i].getChunk().getConstIterator(
                 (scidb.swig.ConstChunkIterator.IGNORE_EMPTY_CELLS |
                  scidb.swig.ConstChunkIterator.IGNORE_OVERLAPS))
            print "Chunk iterator %d loaded." % nc

            while not chunkiter.end():
                dataitem = chunkiter.getItem()

                item = scidb.getTypedValue(dataitem, attrs[i].getType())
                # generate the right type of python object

                print "Data: %s" % item

           iters[i].increment_to_next();
```

17.5.4 Items

Each *dataitem* returned by the iterator requires a different internal method to retrieve it. Determining that method requires examining the type of its attribute, which can be found by calling AttributeDesc.getType(). If you are using a built-in type, then there is a utility function that will call the per-type for you and return and object of the correct type. For example:

```
    scidb.getTypedValue(dataItem, attrs[i].getType())
    # attrs is a per-attribute array of AttributeDesc
```

Special methods are available to detect if an element (or array position) has special significance, such as:

```
dataitem.isEmpty()
```

17.6 Cleanup

A query previously started may be canceled using cancelQuery(). See above for a description of the query ID.

```
db.cancelQuery(queryID) # queryID is in the queryResult class
```

And the client can disconnect from the server using

```
db.disconnect()
```

which will also be called when the db object is deleted (typically by garbage collection). In order to be sure the connection resources are recycled for use by the SciDB server, its probably smart to call db.disconnect() explicitly, rather than waiting for garbage collection to delete the db object at some indeterminate point in the future.

17.7 Exception Handling

If the connector encounters an error, or if the server returns an error during query execution an exception is raised to the python application. These exceptions may be handled using the standard python try/except mechanism.

18 Appendix: Alphabetical List of AFL Operators

Operator Name	Description	Category
adddim	Add one dimension	Creating an Array
aggregate	Compute a single result from an array dimension	Sorting, Windowing, and Aggregating
allversions	Show all array versions	Namespace
apply	Compute new values from array attribute and index values	Basic Array Operations
attribute_rename	Change the name of an array attribute	Changing Array Schemas
attributes	List array attributes	Namespace
avg	Average of a set of array indexes or attributes	Sorting, Windowing, and Aggregating
bernoulli	Use Bernoulli sampling to select a set of cells from an array	Data Sampling
between	Select data from a specified region	Sampling
build	Build array data	Basic Array Operations
build_sparse	Build sparse array data	Basic Array Operations
cancel	Cancel a query	Internal
cast	Change attribute or dimension names	Changing Array Schemas
concat	Concatenate two arrays	Combining Arrays
count	Nonempty array cells	Sorting, Aggregating, and Windowing
create array	Create SciDB array	Creating an Array
cross	Cross product join	Combining Arrays
cross_join	Cross product join with equality predicates	Combining Arrays
deldim	Delete one dimension	Changing Array Schemas
dimensions	Show array dimensions	Namespace
diskinfo	Show space available on SciDB disk	Internal
echo	Print a string	Internal
explain_logical	Explain how the logical structure of a query will be enacted	Internal
explain_physical	Explain how the physical structure of a query will be enacted	Internal
filter	Select by Boolean expression	Data Sampling
help	Operator signature	Namespace
input	Read a file from the system	Loading and Updating
inverse	Matrix inverse	Matrix Algebra
join	Combine attributes by dimension value	Combining Arrays
list	List database contents	Namespace
load	Load data into existing array	Loading and Updating
load_library	Load a plugin	System Administration
lookup	Select data by pattern	Data Sampling

max	Maximum of a set of array indexes or attributes	Sorting, Windowing, and Aggregating
merge	Merge array attributes	Combining arrays
min	Minimum of a set of array indexes or attributes	Sorting, Windowing, and Aggregating
multiply	Matrix multiply	Matrix algebra
normalize	Normalize vector values	Matrix algebra
project	Display attribute values	Basic Array Operations
redimension	Transform attributes to dimensions	Changing Array Schemas
redimension_store	Transform attributes to dimensions	Change array schema
reduce_distro	Reduce distribution	Internal
regrid	Compute aggregates for a sub-grid	Sort, Window, and Aggregate
remove	Remove array from SciDB instance	Creating an Array
rename	Rename array	Creating an Array
repart	Change array chunk size	Changing Array Schemas
reshape	Change array dimension size	Changing Array Schemas
reverse	Reverse dimension values	Changing Array Schemas
sample	Select subset of data at random	Data Sampling
save	Save array data to file	Loading and Updating
scan	Print attributes	Basic Array Operations
setopt	Set/get configuration option value at runtime	Internal
sg	Scatter or gather an array	Internal
show	Print array format	Namespace
slice	Select subplane from array	Data Sampling
sort	Sort attribute values	Sorting, Windowing, and Aggregating
sort2	Sort	Internal
stdev	Standard deviation	Sorting, Windowing, and Aggregating
store	Write array to memory	Loading and Updating
subarray	Select by dimension range	Data Sampling
substitute	Substitute null values	Loading and Updating
sum	Sum	Sorting, Windowing, and Aggregating
thin	Select elements from array dimension	Data Sampling
transpose	Transpose array dimension	Changing Array Schemas
unload_library	Remove library from SciDB instance	System Administration
unpack	Reduce multidimensional array to 1 dimension	Changing Array Schemas
var	Variance	Sorting, Windowing, and Aggregating
versions	Show array versions	Namespace
window	Compute window aggregates	Sorting, Windowing, and Aggregating

xgrid	Expand element into grid	Changing Array Schemas

19 Appendix: Alphabetical List of SciDB Functions

Function Name	Description	Category
%	Remainder	Arithmetic
*	Multiplication	Arithmetic
+	Addition	Arithmetic
-	Subtraction	Arithmetic
/	Division	Arithmetic
<	Less than	Logical
<=	Less than or equal	Logical
<>	Not equal	Logical
=	Equals	Logical
>	Greater than	Logical
>=	Greater than or equal	Logical
abs	Absolute value	Arithmetic
acos	Inverse (arc) cosine in radians	Transcendental
and	Boolean AND	Logical
append_offset	Change time and date by a given amount	Timestamp
apply_offset	Change time and date by a given amount	Timestamp
asin	Inverse (arc) sine in radians	Transcendental
atan	Inverse (arc) tangent in radians	Transcendental
ceil	Round to next-highest integer	Arithmetic
cos	Cosine (input in radians)	Transcendental
exp	Exponential	Transcendental
first	Start of string	Strings
floor	Round to next-lowest integer	Arithmetic
get_offset	Returns time offset in seconds	Timestamp
high	String information	Strings
iif	Inline IF	Logical
is_nan	Returns TRUE is attribute value is NaN	Logical
is_null	Returns TRUE is attribute value is null	Logical
last	End of string	String
length	Get string length	String
log	Base-e logarithm	Transcendental
log10	Base-10 logarithm	Transcendental
low	String query	String
nodeid	Return node id	Troubleshooting
not	Boolean NOT	Logical
now	Current array version	Timestamp
or	Boolean OR	Logical
pow	Raise to a power	Arithmetic

random	Random number	Arithmetic
regex	Search for regular expression	Strings
sin	Sine (input in radians)	Transcendental
sqrt	Square root	Arithmetic
strchar	Convert string to char	Datatype conversion
strftime	Convert string to datetime	Datatype conversion
strip_offset	disregards OFFSET and returns result as a DATETIME	Timestamp
strlen	Maximum string length	Strings
substr	Select substring	Strings
tan	Tangent (input in radians)	Transcendental
togmt	Switch to GMT from current time zone setting	Timestamp
tznow	Set time zone	Timestamp