SciDB User's Guide

SciDB User's Guide

Version 13.1

Table of Contents

List of Figures

Chapter 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 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 operations, timeseries processing, high availability options, 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 (time series) 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 Query Language (AQL) and Array Functional Language (AFL)—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 non-integer dimensions, such as variable-length strings (*alpha*, *beta*, *gamma*, ...) or floating-point values (1.2, 2.76, 4.3, ...). Version 13.1 of SciDB and Paradigm4 support non-int64 dimensions only partially.

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:

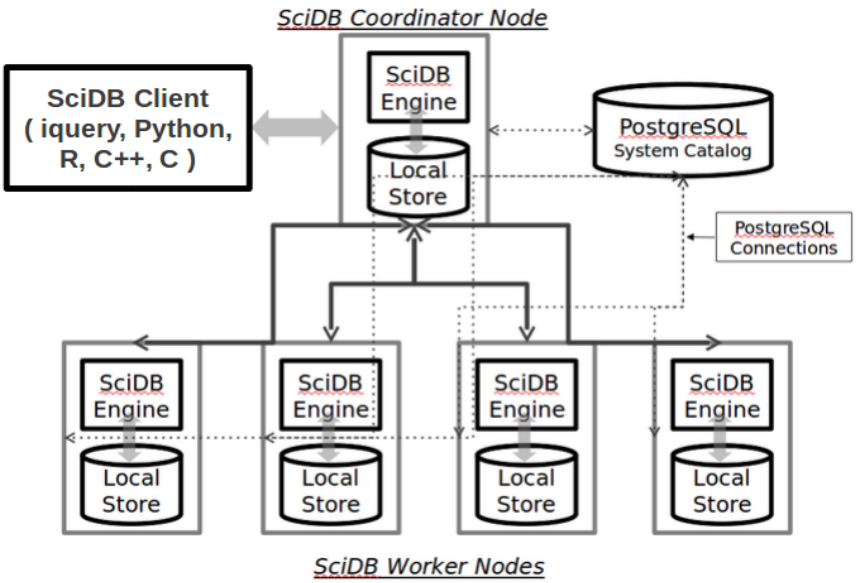
- An array name.
- Array dimensions. The name and size of each dimension must be declared.
- Array attributes of the array. The name and data type of the each attribute must be declared.

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.

1.2. Basic Architecture

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

Figure 1.1. Basic SciDB architecture



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 instance that is responsible for local storage and processing.

External applications, when they connect to a SciDB database, connect to one of the instances in the cluster. While all instances in the SciDB cluster participate in query execution and data storage, one server is the *coordinator* and orchestrates query execution and result fetching. It is the responsibility of the coordinator instance to mediate all communication between the SciDB external client and the entire SciDB database. The rest of the system instances are referred to as *worker* instances 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

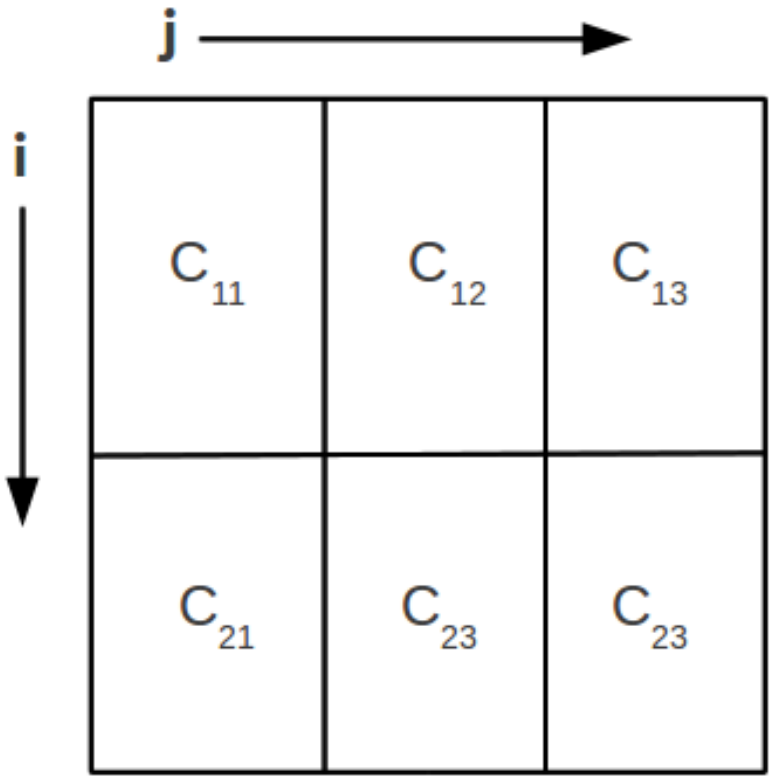
When data is loaded, it is partitioned and stored on each instance of the SciDB database. SciDB uses *chunking*, a partitioning technique for multidimensional arrays where each instance is responsible for stor-

ing and updating a subset of the array locally, and for executing queries that use the locally stored data. By distributing data uniformly across all instances, 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:

Figure 1.2. Chunking diagram



Chunks are stored allocated to instances of the SciDB cluster according to a hash-based scheme.

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 as follows:

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

Array A 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* in both chunks.

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 Array Storage

SciDB arrays consist of array chunk storage and array metadata stored in the system catalog. When arrays are created, updated, or removed, they are done using transactions. Transactions span array storage and the system catalog and ensure consistency of the overall database as queries are executed.

The following sections describe SciDB's instance storage, system catalog, and transaction model.

1.3.1. Instance Storage

Vertical partitioning

Each local SciDB instance divides logical chunks 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 instances—use single-attribute chunks. SciDB uses run-length encoding internally to compress repeated values or commonly occurring patterns typical in scientific applications. Frequently accessed chunks are maintained in an in-memory cache and accelerate query processing by eliminating expensive disk fetches for repeatedly accessed data.

Storage of array versions

SciDB uses a "no overwrite" storage model. No overwrite means that data is never overwritten; each query that stores or updates existing arrays 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.

Storage segments

The local storage manager manages space allocation, placement, and reclamation 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.

Transient storage

SciDB uses temporary data files or "scratch space" 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 last for the lifetime of a query. They are removed upon successful completion or abort of the query.

1.3.2. SciDB System Catalog

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

- Configuration and status information about the SciDB cluster,
- Array-related metadata such as array definitions, array versions, and associations between arrays and other related objects,
- Information about SciDB extensions, such as plug-in libraries containing user-defined objects, which are described in the section "Array Processing."

The system catalog in current versions of SciDB is implemented as PostgreSQL tables. The tables are shared between all SciDB instances within the cluster.

1.3.3. Transaction Model

SciDB combines traditional ACID semantics with versioned, no overwrite array storage. When using versioned arrays, write transactions create new versions of the array—they do not modify pre-existing versions of the array.

The scope of a transaction in SciDB is a single statement. Each statement involves many operations on one or more arrays. Ultimately, the transaction stores the result into a destination array.

SciDB implements array-level locking. Locks are acquired at the beginning of a transaction and are used to protect arrays during queries. Locks are released upon completion of the query. If a query aborts, pending changes are undone at all instances in the system catalog, and the database is returned to a prior consistent state.

1.4. Array Processing

SciDB's query languages provide the basic framework for scalable array processing.

1.4.1. Array Languages

SciDB provides two query language interfaces.

- AQL, the Array Query Language
- AFL, the Array Functional 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.

AQL represents the full set of data management and analytic capabilities including data loading, data selection and projection, aggregation, and joins.

The AQL language includes two classes of queries:

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

AQL statements are handled by the SciDB query compiler which translates and optimizes incoming statements into an execution plan.

SciDB's Array Functional Language (AFL) is a functional language for working with SciDB arrays. AFL *operators* are used to compose queries or statements.

1.4.2. Query Building Blocks

There are four building blocks that you use to control and access your data. These building blocks are:

Operators	SciDB operators, such as join, take one or more SciDB arrays as input and return a SciDB array as output.
Functions	SciDB functions, such as sqrt, take scalar values from literals or SciDB arrays and return a scalar value.
Data types	Data types define the classes of values that SciDB can store and perform operations on.
Aggregates	SciDB aggregates take an arbitrarily large set of values as input and return a scalar value.

Any of these building blocks can be user-defined, that is, users can write new operators, data types, functions, and aggregates.

1.4.3. Pipelined Array Processing

When a SciDB query is issued, it is setup as a pipeline of operators. Operators are responsible for data processing and aggregation as well as intermediate data exchange and data storage.

Execution begins when the client issues a request to fetch a chunk from the result array. Data is then scanned from array storage on all instances and streamed into and out of each operator one chunk at a time. This model of query execution is sometimes referred to as *pull-based* execution and the operators that use this model are called *streaming* operators. Unless required by the data processing algorithm, all SciDB operators are streaming operators. Some operators implement algorithms that require the entire array to be materialized in memory at all instances at once. These are referred to as *materializing* operators.

1.5. Clients and Connectors

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

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 a query using an iterator interface.

1.6. 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*. Optional arguments are shown in square brackets [].

AQL commands are shown in **`FIXED-WIDTH BOLD CAPS`**. When necessary, a line of code may be preceded by the `AQL%` or `AFL%` prompt to show which language the query is issued from.

Chapter 2. SciDB Installation and Administration

SciDB is supported on the following platforms:

- Ubuntu 12.04
- CentOS 6.3
- Red Hat Enterprise Linux (RHEL) 6.3

Note

SciDB requires a 64-bit (x86_64 or amd64) platform.

A complete SciDB installation is a multi-database environment that includes the core SciDB engine, Postgres, the open source SQL database engine which is used for system catalog data, as well as ScaLAPACK/MPI which is used as a computational engine for dense linear algebra.

To report issues, contact support@scidb.org.

2.1. Terminology

The following terms are used to describe the SciDB installation and administration process:

Single server	A configuration that consists of a single machine with a processor that may contain multiple cores, memory and attached storage. A single server may be virtual or physical.
Virtual server	A server that shares hardware rather than having dedicated hardware.
Coordinator server	In a configuration that has multiple servers, exactly one server functions as the coordinator, and contains the *coordinator instance*.
SciDB instance	An independent SciDB group of processes, that is, a single running SciDB. There may be a many-to-one mapping between SciDB instances and a server.
Coordinator instance	A SciDB instance that resides on the coordinator server. There is a single coordinator instance for a SciDB cluster. A coordinator instance coordinates query activity in addition to participating in query execution.
Worker instance	A SciDB instance that only participates in query execution.
Cluster	A group of one or more single servers connected by TCP/IP, working together as a single system. A cluster can be a private grid or a public cloud.
SciDB cluster	A collection of SciDB instances (one coordinator and zero or more worker instances) form a SciDB cluster.

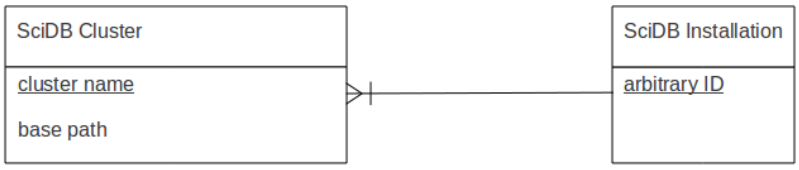
2.2. Visualizing a Completed SciDB Installation

This section describes the pieces of a completed SciDB installation.

2.2.1. The SciDB Installation and Its Clusters

The chapter describes the process for creating one SciDB Installation, which consists of one or more SciDB clusters. The following conceptual data model diagram expresses this graphically.

Figure 2.1. SciDB Cluster



The figure asserts the following: There are SciDB installations. Each SciDB installation has an arbitrary ID and SciDB clusters. There are SciDB clusters. Each SciDB cluster has a cluster name, a base path, and belongs to one SciDB installation. Within any SciDB installation, Each SciDB cluster name is unique.

Note

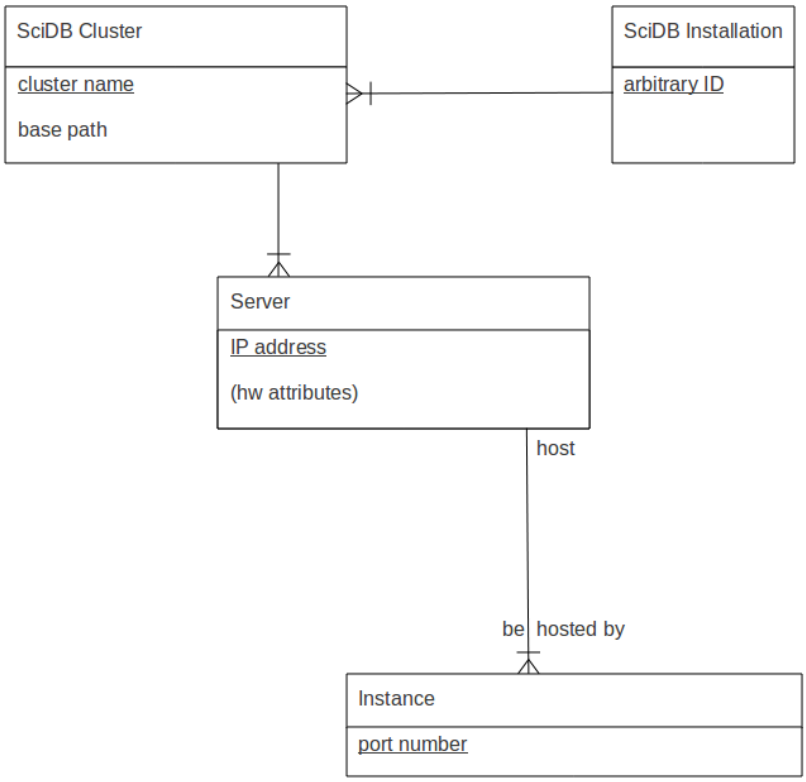
The data models shown in this chapter are purely conceptual. They do not describe any physical data characteristics or connote any particular meta-model such as relational, object-oriented, etc. For a complete description of this notation, see Mastering Data Modeling: A User-Driver Approach by John Carlis et al. (Addison-Wesley, 2001)

2.2.2. The SciDB Cluster, Servers, and Instances

Each SciDB cluster consists of one or more servers, each of which can host one or more server instances.

Like this:

Figure 2.2. SciDB Servers and Instances



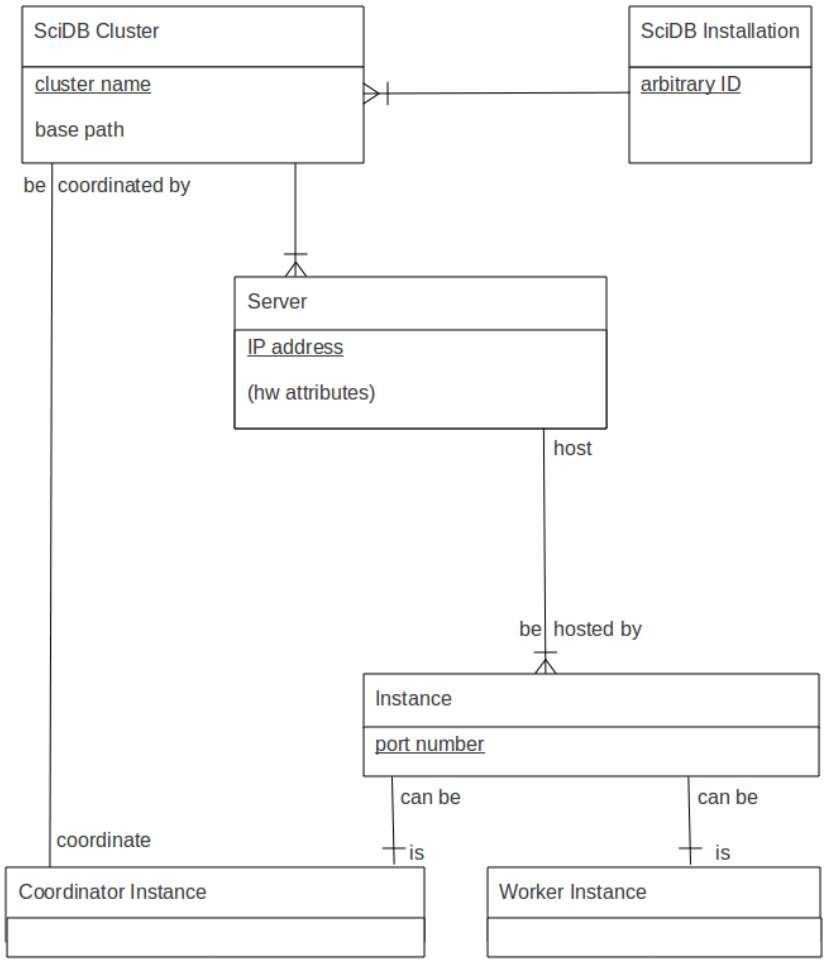
The figure asserts the following: There are SciDB installations. Each SciDB installation has an arbitrary ID and SciDB clusters. There are SciDB clusters. Each SciDB cluster has a cluster name, a base path, and belongs to one SciDB Installation. Within any SciDB installation, each SciDB cluster name is unique. There are servers. Each server has an IP address, belongs to one SciDB cluster, and can host multiple instances. There are instances. Each instance is hosted by one server. Each instance has a port number. Each instance has a unique combination of server and port number. (That is, two instances can have the same port number, but only if they are hosted by different servers.)

2.2.3. The Coordinator Instance and Worker Instances

Every instance is either a coordinator instance, a worker instance, or both. Each SciDB cluster will have exactly one coordinator instance.

Like this:

Figure 2.3. SciDB Instances: Coordinator and Workers



Note

A worker instance can also serve as the coordinator instance. Such an instance is typically referred to as the coordinator instance, but it can share in the workload of SciDB queries like any other worker instance.

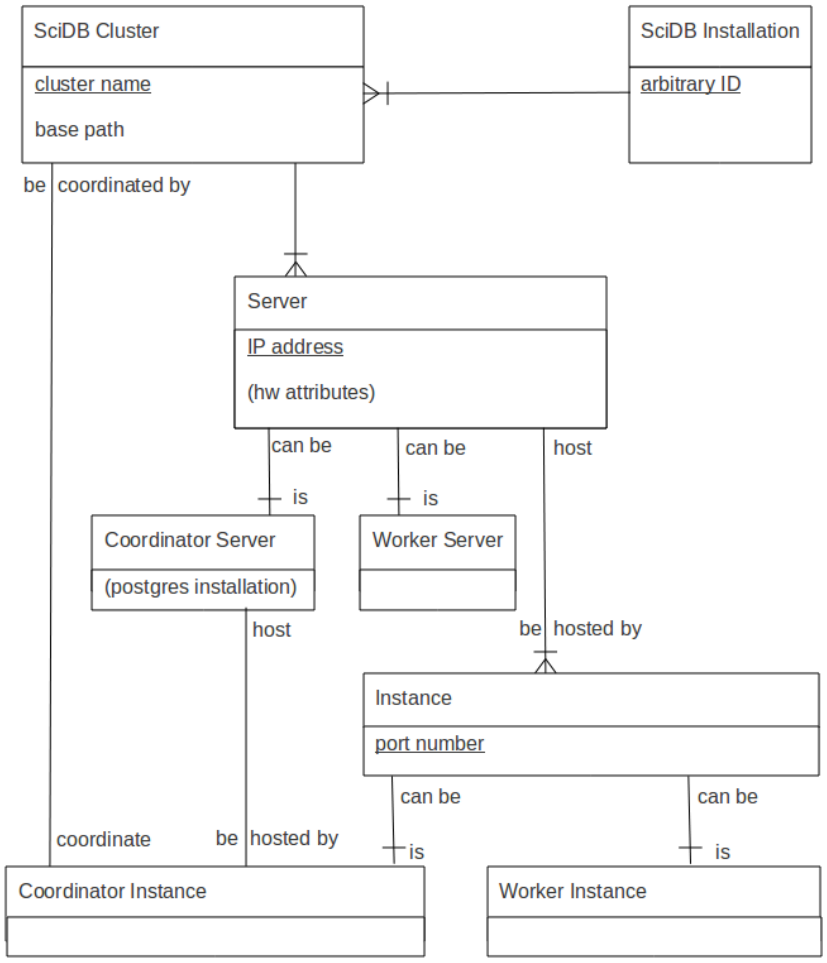
The figure asserts the following: There are SciDB installations. Each SciDB installation has an arbitrary ID and SciDB clusters. There are SciDB clusters. Each SciDB cluster has a cluster name, a base path, and belongs to one SciDB installation. Within any SciDB installation, each SciDB cluster name is unique. There are servers. Each server has an IP address, belongs to one SciDB cluster, and can host multiple instances. There are instances. Each instance is hosted by one server. Each instance has a port number. Each instance has a unique combination of server and port number. (That is, two instances can have the same port number, but only if they are hosted by different servers.) An instance can be a coordinator instance, a worker instance, or both. Every coordinator instance is an instance. Every worker instance is an instance. Each coordinator instance coordinates the work of one SciDB cluster, and each SciDB cluster can be coordinated by one coordinator instance.

2.2.4. The Coordinator Server and Worker Servers

Whichever server hosts the coordinator instance is known as the coordinator server. The other servers as known as worker servers. In most contexts, the distinction between the coordinator server and worker servers is immaterial, and they are collectively known as servers. The installation instructions in this chapter direct you to install Postgres on the coordinator server. Although the coordinator server must host the coordinator instance, it can also host other (worker) instances.

Like this:

Figure 2.4. SciDB Servers: Coordinator and Workers



The figure asserts the following: There are SciDB installations. Each SciDB installation has an arbitrary ID and SciDB clusters. There are SciDB clusters. Each SciDB cluster has a cluster name, a base path, and belongs to one SciDB installation. Within any SciDB installation, each SciDB cluster name is unique. There are servers. Each server has an IP address, belongs to one SciDB cluster, and can host multiple instances. There are instances. Each instance is hosted by one server. Each instance has a port number. Each instance has a unique combination of server and port number. (That is, two instances can have the same port number, but only if they are hosted by different servers.) An instance can be a coordinator instance, a

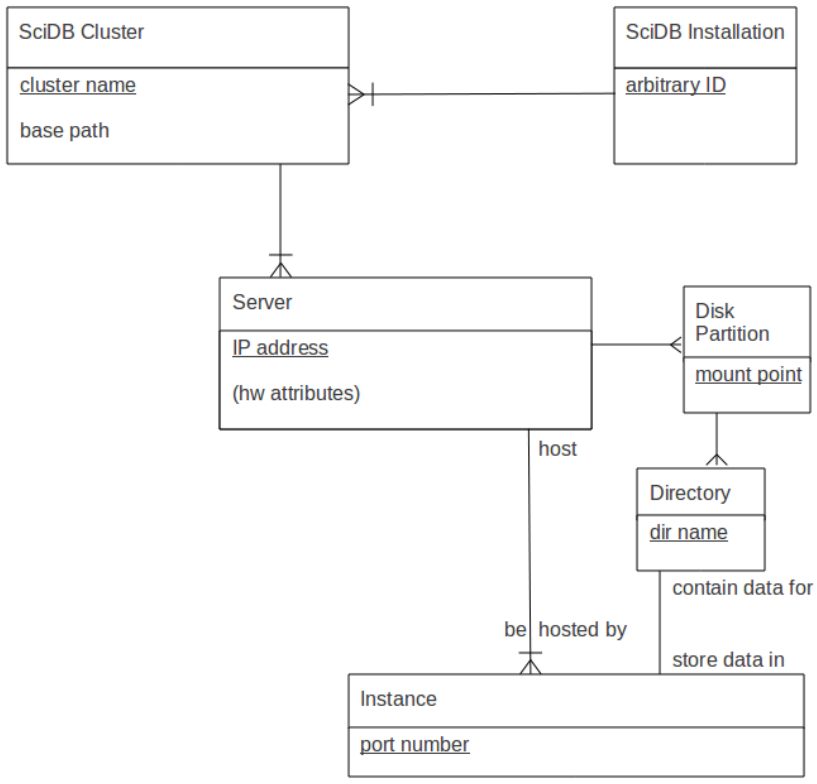
worker instance, or both. Each coordinator instance coordinates the work of one SciDB cluster, and each SciDB cluster can be coordinated by one coordinator instance. A server can be a coordinator server or a worker server. The coordinator server hosts the coordinator instance (and because every coordinator instance is an instance, might also host other instances). The coordinator server will ultimately host the Postgres installation.

2.2.5. Disk Partitions and Directories

When two or more instances are hosted by the same server, each of those instances requires its own directory to store SciDB database files. Those directories could be on the same disk partition.

Like this:

Figure 2.5. Disk Partitions and Directories



The figure asserts the following: There are SciDB installations. Each SciDB installation has an arbitrary ID and SciDB clusters. There are SciDB clusters. Each SciDB cluster has a cluster name, a base path, and belongs to one SciDB installation. Within any SciDB installation, each SciDB cluster name is unique. There are servers. Each server has an IP address, belongs to one SciDB cluster, and can host multiple instances. There are instances. Each instance is hosted by one server. Each instance has a port number. Each instance has a unique combination of server and port number. (That is, two instances can have the same port number, but only if they are hosted by different servers.) There are disk partitions. Each disk partition belongs to one server, and has a unique mount point and can have multiple directories. Each

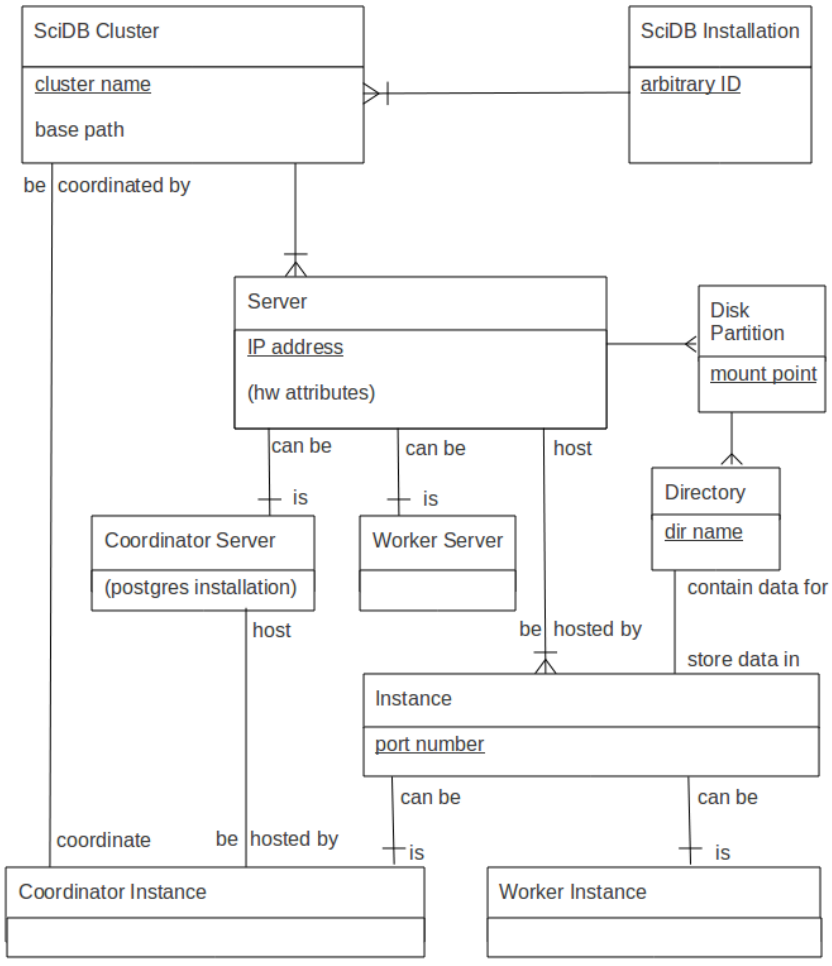
directory in on one disk partition, has a directory name, and can contain data for exactly one instance. Each instance stores its SciDB data on one directory.

2.2.6. Complete SciDB Installation

The complete data model described in the preceding sections is shown below. The model can help you visualize important distinctions that arise during the installation process. For example, some steps of the installation apply to all servers, whereas others apply only to the coordinator server. Likewise, some steps apply only to the coordinator instance, whereas other steps apply to all instances.

The complete model is here:

Figure 2.6. SciDB Installation



Note

Remember, this is merely a conceptual model to help you understand the building blocks of the installation process. It is a conceptual model—not intended to connote any physical data layout or any particular meta-model such as relational or object-oriented. For that matter, it presents a

somewhat simplified view of a SciDB configuration, which is all you need to begin the installation process.

2.3. Preparing the Platform

If you are upgrading from a previous version of SciDB, you can skip this section, and continue with Section 2.4, “Installing Packages”.

Before installing SciDB, the following steps are required to prepare a cluster for running SciDB. Many of these steps must be executed as the Linux 'root' account (creating the scidb account, preparing the system catalog, configuring a local data directory for use by SciDB, installing SciDB software packages).

Depending on your operating system, you may need to perform the following tasks:

- Ubuntu does not enable root by default. To enable the root account on Ubuntu, run the following command:

  ```
  sudo passwd
  ```

  Sudo will prompt you for your password, and then ask you to supply a new password for root.

- For CentOS and RHEL, you should create a user account (**adduser**), and provide sudo access for this account. For details, see your OS documentation.

2.3.1. scidb Account

Note

This step must be performed by root, or an account with equivalent administrative privileges. The scidb account must be available on all servers in the cluster.

First, you need to create the **scidb** account. All SciDB processes as well as data and log files are created and owned by this ID. On a cluster, each server must have this account. This is also the account that initializes, starts, and stops the SciDB cluster.

The scidb user account must have password-less SSH access configured between the coordinator server and all servers in the SciDB cluster. How to configure this access is described later in the chapter.

2.3.2. Configure Storage

Note

This action must be performed by the root account on each server in the cluster. Once created the base path should be owned by the **scidb** account.

On each server, select a location on the local file system to place the data directory for each instance. This must be accessible via the same path name on every server.

The recommended configuration is 1 disk and 4 CPU cores for each SciDB instance. This is easily achieved in a virtual server configuration. For physical server configuration, please explore the SciDB forum at www.scidb.org/forum for guidance on creating an optimal configuration.

2.3.3. Remote Execution Configuration (SSH)

Note

This step must be performed by the scidb user account.

SciDB uses `ssh` for execution of management commands such as start and stop within a cluster. This is why the **scidb** user account should have password-less `ssh` access from the coordinator to the workers and from the coordinator to itself.

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

1. Create a key:

   ```
   ssh-keygen
   ```

   By default, this creates a key file with a public/private key pair in ~/.ssh/id_rsa.pub and ~/.ssh/id_rsa. Optionally, a key file name may be specified. If a non-default filename is used for the key pair, it must be listed in the SciDB configuration file for use by scidb.py.

   ```
   scidb@monolith1:~/.ssh$ ssh-keygen
   Generating public/private rsa key pair.
   Enter file in which to save the key (/home/scidb/.ssh/id_rsa):
   Enter passphrase (empty for no passphrase):
   Enter same passphrase again:
   Your identification has been saved in id_rsa.
   Your public key has been saved in id_rsa.pub.
   ...
   ```

2. Copy the key to the localhost (or coordinator) and to each worker to authorize ssh clients connecting to it:

   ```
   ssh-copy-id -i ~/.ssh/id_rsa.pub scidb@worker
   ssh-copy-id -i ~/.ssh/id_rsa.pub scidb@localhost
   ```

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

   ```
   ssh scidb@worker
   ```

Note

If you are running SciDB on CentOS or RHEL, you may need to perform additional configuration steps. See the next section for details.

2.3.4. Additional Configuration on CentOS and RHEL

This section describes two additional configuration steps that are necessary if you are running SciDB on CentOS or RHEL:

- Configure iptables
- Configure SELinux

2.3.4.1. Configure iptables

If you are running on CentOS or RHEL, you need to set up iptables, in order for a multi-server configuration of SciDB to function properly.

The iptables program controls a host-based firewall for Linux operating systems. For SciDB, you should perform one of the following actions:

- Disable iptables. To disable iptables, use either of the following commands:

```
chkconfig iptables off # do not start iptables during boot
service iptables stop # stop iptables
```

  Or,

- Grant connections for the SciDB ports and for the Postgres port.

To grant connections, perform the following steps:

1. Use **sudo**, or login as root to edit the `/etc/sysconfig/iptables` file.
2. On the coordinator instance, grant access for the Postgres port, 5432. Insert the following line before the final line (COMMIT) of the iptables file:

```
-A INPUT -m state --state NEW -m tcp -p tcp --dport 5432 -j ACCEPT
```

3. On all instances—the coordinator and all workers—you must add a line for each instance port. For example, if you have four instances, on ports 1239, 1240, 1241, and 1242, you would add the following lines:

```
-A INPUT -m state --state NEW -m tcp -p tcp --dport 1239 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 1240 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 1241 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 1242 -j ACCEPT
```

4. Restart the iptables service.

```
sudo service iptables restart
```

Here is an example of the updated iptables file on the coordinator:

```
# Firewall configuration written by system-config-firewall
# Manual customization of this file is not recommended.
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
-A INPUT -p icmp -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT
# Grant connection for Postgres (coordinator only)
-A INPUT -m state --state NEW -m tcp -p tcp --dport 5432 -j ACCEPT
# Grant SciDB instance connections
-A INPUT -m state --state NEW -m tcp -p tcp --dport 1239 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 1240 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 1241 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 1242 -j ACCEPT
```

```
-A INPUT -i lo -j ACCEPT
-A INPUT -j REJECT --reject-with icmp-host-prohibited
-A FORWARD -j REJECT --reject-with icmp-host-prohibited
COMMIT
```

2.3.4.2. Configure SELinux

CentOS and RHEL have SELinux (Security-Enhanced Linux) enabled by default. This can cause problems when you attempt to set up password-less SSH. You can either disable SELinux or configure SELinux to allow password-less ssh to function correctly.

- Configure SELinux so that password-less ssh works. As the scidb user, run the following commands.

  ```
  cd ~
  chmod 700 .ssh
  cd .ssh
  chmod 600 *
  chmod 644 authorized_keys
  chmod 644 known_hosts
  chmod 644 config
  restorecon -R -v ../.ssh
  ```

  Or,

- Disable SELinux. For details, see your operating system documentation, or search the web for instructions. Note that this method may introduce security risks,

2.4. Installing Packages

Packages are available for Ubuntu, CentOS, and RHEL. Note that CentOS and RHEL use the same package.

SciDB currently provides the following packages.

- The `scidb` package contains the server binaries.
- The `libscidbclient` and `libscidbclient-python` packages contain the SciDB client library and the SciDB connector for python applications.
- The `scidb-plugins` package contains plugins that can be added to SciDB to extend its capabilities. This package contains the `dense_linear_algebra` plugin.
- The `scidb-utils` package contains SciDB server utilities.
- The `scidb-dev` contains development header files for developers creating SciDB extensions (types, operators, and aggregates).
- The `scidb-dev` contains development header files for developers creating SciDB extensions (types, operators, and aggregates).
- The `libscidbclient-dbg`, `libscidbclient-python-dbg`, `scidb-dbg`, `scidb-dev-tools-dbg`, `scidb-plugins-dbg`, and `scidb-utils-dbg` packages contain debugging symbols.

2.4.1. Install SciDB on Ubuntu from binary package

This section describes how to download and install SciDB on Ubuntu, using pre-built binary packages.

Note

This step must be performed by the root account. In a cluster, this step must be performed on all servers.

This section describes how to download and install SciDB using pre-built binary packages.

1. Add SciDB's GPG public key by running the following command.

```
wget -O-  http://downloads.paradigm4.com/key | sudo apt-key add -
```

2. Add SciDB's APT repository to the apt configuration file `/etc/apt/sources.list.d/scidb.list`.

```
deb  http://downloads.paradigm4.com/  ubuntu12.04/13.1/
deb-src  http://downloads.paradigm4.com/  ubuntu12.04/13.1/
```

3. Update APT index and install SciDB. If you want the default SciDB installation with all packages installed, you can install the scidb-all (or scidb-all-coord) metapackage. This package includes all of the standard SciDB packages.

```
sudo apt-get update
sudo apt-cache search scidb
sudo apt-get install scidb-all-coord # On the coordinator server
 only
sudo apt-get install scidb-all # On all servers other than the
 coordinator server
sudo apt-get source scidb # If you want the SciDB source
```

If you want the default SciDB installation with all packages installed, you can install the scidb-all (worker) or scidb-all-coord (coordinator) metapackage. These packages include all of the standard SciDB packages required for a complete installation.

Note

The only difference between the scidb-all and scidb-all-coord packages is that scidb-all-coord contains Postgres. If you already have Postgres on your coordinator server, there is no need to install the scidb-all-coord package.

2.4.2. Install SciDB on CentOS / RHEL from binary package

This section describes how to download and install SciDB on CentOS and RHEL, using pre-built binary packages.

Note

This procedure must be performed by the root account. In a cluster, this step must be performed on all servers.

1. Add SciDB's GPG public key by running the following command.

```
wget http://downloads.paradigm4.com/key
rpm --import key
rm -f key
```

2. Add the SciDB repository to the repository folder, by creating the configuration file `/etc/yum.repos.d/scidb.repo`.

3. Add the following lines to the `/etc/yum.repos.d/scidb.repo` file:

```
[scidb]
name=SciDB
baseurl=http://downloads.paradigm4.com/centos6.3/13.1/
gpgcheck=1
```

4. Update packages by running the following command:

```
sudo yum update
```

Note

If you receive an error concerning the boost library, remove boost by running the following command, and then rerun **sudo yum update**.

```
rpm -qa | grep boost | sudo xargs yum remove -y
```

5. Install SciDB. If you want the default SciDB installation with all packages installed, you can install the scidb-all (or scidb-all-coord) metapackage. This package includes all of the standard SciDB packages.

```
sudo yum search scidb
sudo yum install scidb-all-coord # On the coordinator server only
sudo yum install scidb-all # On all servers other than the
 coordinator server
```

If you want the default SciDB installation with all packages installed, you can install the scidb-all (worker) or scidb-all-coord (coordinator) metapackage. These packages include all of the standard SciDB packages required for a complete installation.

Note

The only difference between the scidb-all and scidb-all-coord packages is that scidb-all-coord contains Postgres. If you already have Postgres on your coordinator server, there is no need to install the scidb-all-coord package.

2.4.3. System Catalog Setup

If you are upgrading from a previous version of SciDB, you can skip this section, and continue with Section 2.5, “Configuring SciDB”.

SciDB relies on Postgres as the system catalog. In this section, we describe how to set up and use Postgres **for SciDB**.

Warning

We strongly recommend that you use a separate Postgres installation exclusively for SciDB. The main reason for this is that SciDB needs to store the password for the 'scidb' user in clear text in a configuration file. Thus, if you use an existing Postgres installation for SciDB, you introduce a security risk into the existing Postgres installation.

Note the following:

- SciDB requires Postgres 8.4. This version is typically available on most Linux platforms. If your platform provides multiple versions of Postgres, make sure that version 8.4 is running and using port 5432.
- The installation and configuration of Postgres must be performed by root, or an account with equivalent administrative privileges.
- You install and configure Postgres on the coordinator server only.

By default, Postgres is configured to allow only local access via Unix-domain sockets. In a cluster environment, the Postgres database server needs to be configured to allow access from other instances in the cluster. To do this, perform the following steps:

1. If you have never previously run Postgres on your system, you must perform the following steps:
   a. Run the following command to create the `pg_hba.conf` file (you will edit this file in step 2):

      ```
      sudo service postgresql initdb
      ```

   b. Create a PostgreSQL user on your Linux system. For details, see the Postgres documentation.
2. Configure Postgres to use 'trust' authentication for local and remote connections. To do this, modify the `pg_hba.conf` file (usually at `/etc/postgresql/8.4/main/` or `/var/lib/pgsql/data/`) and include the following settings.

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

   This configuration setting causes Postgres to use 'trust' authentication for all local and remote connections from instances within the 10.0.0.1/8 subnet. Replace the subnet/mask with the correct one for your cluster.

   #### Note

   If you are installing on a Red Hat Enterprise Linux (RHEL) operating system, you must enable trust authentication for *all* entries in the `pg_hba.conf` file.
3. You might need to set the `postgresql.conf` file to have it listen on the relevant port. If you are running a cluster with multiple servers, you will also need to modify the `postgresql.conf` file to allow connections from other instances in the cluster.

   ```
   # - Connection Settings -
   listen_addresses = '*'
   port = 5432
   ```
4. Restart Postgres.

```
sudo /etc/init.d/postgresql restart
```

5. The final step, after you have configured Postgres, is to add it to Linux system services. This means that Postgres will be started automatically on system reboot.

   ```
   sudo /sbin/chkconfig --add postgresql
   ```

   To run the previous command, you need to install the chkconfig package, if it is not already installed on your system.

Warning

This Postgres configuration might pose security issues. When authentication is set to trust PostgreSQL assumes that anyone who can connect to the server is authorized to access the database. To make a more secure installation, you can list specific host IP addresses, user names, and role mappings.

You can read more on the security details of Postgres client-authentication in the Postgres documentation at http://www.postgresql.org/docs/8.3/static/client-authentication.html.

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

```
sudo /etc/init.d/postgresql status
```

2.5. Configuring SciDB

This section introduces the SciDB config.ini and shows how to configure SciDB prior to initialization (usually `/opt/scidb/13.1/etc/config.ini`). Logging configuration is also described.

2.5.1. SciDB Configuration File

Note

The configuration file resides on the coordinator server only.

The standard location of the SciDB configuration file is `/opt/scidb/13.1/etc/config.ini`.

The configuration 'test1' below is an example of the configuration for a single-instance system (coordinator only):

```
[test1]
server-0=localhost,0
db_user=test1user
db_passwd=test1passwd
install_root=/opt/scidb/13.1
metadata=/opt/scidb/13.1/share/scidb/meta.sql
pluginsdir=/opt/scidb/13.1/lib/scidb/plugins
logconf=/opt/scidb/13.1/share/scidb/log4cxx.properties
base-path=/home/scidb/data
base-port=1239
interface=eth0
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=100485760
```

2.5.2. Set Environment Variables

Note

You set the environment variables on the coordinator server only.

The following variables should be set in the scidb user's environment.

```
export SCIDB_VER=13.1
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
```

2.5.3. Cluster Configuration Example

The following SciDB cluster configuration is called 'monolith.' This cluster consists of eight identical virtual servers with the following characteristics:

- x86_64 6-core processor
- 8 GB of RAM
- 1 TB direct attached storage
- 1Gbps Ethernet
- Linux OS from the list of supported distributions.

The following configuration file applies to such a cluster called `monolith` and is explained in the following section.

```
[monolith]
# server-id=IP, number of worker instances
server-0=10.0.20.231,0
server-1=10.0.20.232,1
server-2=10.0.20.233,1
server-3=10.0.20.234,1
server-4=10.0.20.235,1
server-5=10.0.20.236,1
server-6=10.0.20.237,1
server-7=10.0.20.238,1
db_user=monolith
db_passwd=monolith
install_root=/opt/scidb/13.1
metadata=/opt/scidb/13.1/share/scidb/meta.sql
```

```
pluginsdir=/opt/scidb/13.1/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` with examples which must be customized and copied to `config.ini`.

The following table describes the basic configuration file settings:

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]*
server-N, w	The host name or IP address of server N, where N = 0, 1, 2, ..., followed by a comma, followed by the number of worker instances `w` to launch on the server. The coordinator is always on server-0 and launches at least one instance that serves as database coordinator.
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	Full path to the SciDB installation directory.
metadata	Full path to the SciDB metadata definition file.
pluginsdir	Full path to the SciDB plugins directory that contains all server plugins.
logconf	Full path to the **log4xx** logging configuration file.

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. Each SciDB instance uses an enumerated data directory below the base-path. The `list('instances')` command shows all instances and their data directories for a running SciDB cluster.
base-port (optional)	Base port number. Connections to the coordinator (and therefore to the system) are via this port number, while worker instances communicate via base-port + instance number. The default port number for the SciDB coordinator is 1239.
data-dir-prefix (optional)	The SciDB administrator can provide file system directories for reference to multiple disks that are connected to a single server. For example, if a server has 4 disks and 8 instances, your configuration could be as follows: data-dir-prefix-0-0=/datadisk1/myserver.000.0 data-dir-prefix-0-1=/datadisk2/myserver.000.1 data-dir-prefix-0-2=/datadisk3/myserver.000.2 data-dir-prefix-0-3=/datadisk4/myserver.000.3 data-dir-prefix-0-4=/datadisk1/myserver.000.4 data-dir-prefix-0-5=/datadisk2/myserver.000.5 data-dir-prefix-0-6=/datadisk3/myserver.000.6 data-dir-prefix-0-7=/datadisk4/myserver.000.7

	You do not need to specify this parameter for each instance. For any instance that you omit, SciDB creates a folder using the default naming scheme.
interface	Ethernet interface that SciDB uses.
pg-port (optional)	The listening port of Postgres—the port on which Postgres accepts incoming connections. Default: 5432.
ssh-port (optional)	The port that ssh uses for communications within the cluster. Default:22.
key-file-list (optional)	Comma-separated list of filenames that include keys for ssh authentication. Default: /home/scidb/.ssh/id_rsa and id_dsa.
tmp-path (optional)	Full path to temporary directory. Default is a directory within `base-path`.
no-watchdog (optional)	Set this to true if you do not want automatic restart of the SciDB server on a software crash. Default: false.

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. This setting, when true, is a hint to write temporary data directly to disk files without caching them, thereby saving memory at the cost of performance. Default: False or 'Off'.
mem-array-threshold (optional)	Maximum size in MB of temporary data to be cached in memory, before writing to temporary disk files. Default: 1024 MB. Note that even if save-ram is true, some parts of the system may still ignore the hint, and use the cache.
smgr-cache-size (optional)	Size of memory in MB allocated to the global shared cache of array chunks. Only chunks belonging to stored arrays are written to this cache. Default: 256 MB
max-memory-limit (optional)	The hard-limit maximum amount of memory in MB that the SciDB instance shall be allowed to allocate. If the instance requests more memory from the operating system the allocation will fail with an exception. Default: No limit.
merge-sort-buffer (optional)	Size of memory buffer used in merge sort, in MB. Default: 512 MB.
chunk-segment-size (optional)	Size of a storage segment in bytes. Set to a nonzero value to enable reuse of storage from removed arrays. If this number is set to a nonzero value, it must be large enough to contain the largest chunk of the array. If set to zero, no space reuse or storage reclamation is done. Default value: 85MB. Maximum size: 2,147,483,647 B (= 2GB).
execution-threads (optional)	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: 4.
operator-threads (optional)	Number of threads used per operator per query. Limit the number of threads allocated per (multi-threaded) operator in a query. If operator-threads is unspecified, SciDB automatically detects the number of CPU cores and uses that value. If you are running multiple instances on each server, operator-threads must be set lower than the number of CPU cores since multiple SciDB instances share the same set of CPU cores. Default: Number of CPU cores.
result-prefetch-threads (optional)	Per-query threads available for prefetch. Default: 4.

result-prefetch-queue-size (optional)	Per-query number of result chunks to prefetch. Default: 4.
small-memalloc-size (optional)	Small allocation threshold size in bytes for glibc malloc. All memory allocations larger than this size will be treated as "large" and pass through to Linux mmap. M_MMAP_THRESHOLD setting for malloc. Default: 268,435,456 bytes (256 MB)
large-memalloc-limit (optional)	Threshold limit on the maximum number of simultaneous large allocations for glibc malloc. M_MMAP_MAX setting for malloc. Default: 65,536.

In the example above, `db_user` is set to *test1user* and `db_passwd` is set to *test1passwd*.

2.5.4. SciDB Configuration Parameters

The following table describes the basic configuration file settings for SciDB:

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]*
server-N, w	The host name or IP address of server N, where N = 0, 1, 2, ..., followed by a comma, followed by the number of worker instances `w` to launch on the server. The coordinator is always on server-0 and launches at least one instance that serves as database coordinator.
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	Full path to the SciDB installation directory.
metadata	Full path to the SciDB metadata definition file.
pluginsdir	Full path to the SciDB plugins directory that contains all server plugins.
logconf	Full path to the **log4xx** logging configuration file.

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. Each SciDB instance uses an enumerated data directory below the base-path. The `list('instances')` command shows all instances and their data directories for a running SciDB cluster.
base-port (optional)	Base port number. Connections to the coordinator (and therefore to the system) are via this port number, while worker instances communicate via base-port + instance number. The default port number for the SciDB coordinator is 1239.
data-dir-prefix (optional)	The SciDB administrator can provide file system directories for reference to multiple disks that are connected to a single server. For example, if a server has 4 disks and 8 instances, your configuration could be as follows: data-dir-prefix-0-0=/datadisk1/myserver.000.0 data-dir-prefix-0-1=/datadisk2/myserver.000.1

	data-dir-prefix-0-2=/datadisk3/myserver.000.2 data-dir-prefix-0-3=/datadisk4/myserver.000.3 data-dir-prefix-0-4=/datadisk1/myserver.000.4 data-dir-prefix-0-5=/datadisk2/myserver.000.5 data-dir-prefix-0-6=/datadisk3/myserver.000.6 data-dir-prefix-0-7=/datadisk4/myserver.000.7 You do not need to specify this parameter for each instance. For any instance that you omit, SciDB creates a folder using the default naming scheme.
interface	Ethernet interface that SciDB uses.
pg-port (optional)	The listening port of Postgres—the port on which Postgres accepts incoming connections. Default: 5432.
ssh-port (optional)	The port that ssh uses for communications within the cluster. Default:22.
key-file-list (optional)	Comma-separated list of filenames that include keys for ssh authentication. Default: /home/scidb/.ssh/id_rsa and id_dsa.
tmp-path (optional)	Full path to temporary directory. Default is a directory within `base-path`.
no-watchdog (optional)	Set this to true if you do not want automatic restart of the SciDB server on a software crash. Default: false.

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. This setting, when true, is a hint to write temporary data directly to disk files without caching them, thereby saving memory at the cost of performance. Default: False or 'Off'.
mem-array-threshold (optional)	Maximum size in MB of temporary data to be cached in memory, before writing to temporary disk files. Default: 1024 MB. Note that even if save-ram is true, some parts of the system may still ignore the hint, and use the cache.
smgr-cache-size (optional)	Size of memory in MB allocated to the global shared cache of array chunks. Only chunks belonging to stored arrays are written to this cache. Default: 256 MB
max-memory-limit (optional)	The hard-limit maximum amount of memory in MB that the SciDB instance shall be allowed to allocate. If the instance requests more memory from the operating system the allocation will fail with an exception. Default: No limit.
merge-sort-buffer (optional)	Size of memory buffer used in merge sort, in MB. Default: 512 MB.
chunk-segment-size (optional)	Size of a storage segment in bytes. Set to a nonzero value to enable reuse of storage from removed arrays. If this number is set to a nonzero value, it must be large enough to contain the largest chunk of the array. If set to zero, no space reuse or storage reclamation is done. Default value: 85MB. Maximum size: 2,147,483,647 B (= 2GB).
execution-threads (optional)	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: 4.
operator-threads (optional)	Number of threads used per operator per query. Limit the number of threads allocated per (multi-threaded) operator in a query. If opera-

	tor-threads is unspecified, SciDB automatically detects the number of CPU cores and uses that value. If you are running multiple instances on each server, operator-threads must be set lower than the number of CPU cores since multiple SciDB instances share the same set of CPU cores. Default: Number of CPU cores.
result-prefetch-threads (optional)	Per-query threads available for prefetch. Default: 4.
result-prefetch-queue-size (optional)	Per-query number of result chunks to prefetch. Default: 4.
small-memalloc-size (optional)	Small allocation threshold size in bytes for glibc malloc. All memory allocations larger than this size will be treated as "large" and pass through to Linux mmap. M_MMAP_THRESHOLD setting for malloc. Default: 268,435,456 bytes (256 MB)
large-memalloc-limit (optional)	Threshold limit on the maximum number of simultaneous large allocations for glibc malloc. M_MMAP_MAX setting for malloc. Default: 65,536.

In the example above, `db_user` is set to *test1user* and `db_passwd` is set to *test1passwd*.

2.5.5. Logging Configuration

- x86 6-core processor

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

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=ERROR, 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
```

2.6. Tuning your SciDB Installation

This section provides general guidelines about maximizing the performance of SciDB on your system. Below are suggestions meant to guide you in choosing the right combination of settings for SciDB configuration parameters for your SciDB installation.

For the default values of the configuration parameters described in this section, see Section 2.5.4, “SciDB Configuration Parameters”.

2.6.1. Configuring Memory Usage

SciDB provides the following parameters for configuring the usage of RAM:

- *merge-sort-buffer* (megabytes). The maximum amount of memory that the **sort()** operator can consume, per thread. Note that each thread of the operator will consume up to this amount. The number of threads in **sort()** is controlled by the parallel-sort and *result-prefetch-queue-size parameters.*

- *smgr-cache-size* (megabytes). The amount of memory that the storage manager cache may use. This is a global cache that is used by all queries and stays occupied when the system is quiescent. This cache is populated with chunks of persistent arrays that were recently read or recorded. On systems with a very large amount of memory, setting this parameter to a large value will allow one to essentially run SciDB read queries "out of memory."

- *mem-array-threshold* (megabytes). The amount of memory that the temporary array cache may use. This applies to operators that work by creating temporary materialized arrays (aggregates, some repartitions, variable_window, redimension, others). If the cache is too small to hold all temporary materialized data, some of these temporary results are flushed to temporary files on disk.

  All running queries with materialized temporary arrays share this cache. Notice that the *tmp-path* configuration parameter controls the location of the temporary disk storage. It is important to make sure that this location is not mapped into memory (for example via RAM disk or the **tmpfs** utility).

- *network-buffer* (megabytes): Roughly, the amount of memory that the SciDB instance may use to receive data from other instances via the network. To be precise, the sender instance send out this much data prior to pausing and waiting for the receiver instances to consume the data and respond.

- *max-memory-limit* (megabytes): The hard-limit maximum amount of memory that the SciDB instance is allowed to consume. If the instance requests more memory from the operating system—this can happen for several reasons—the allocation will fail with an exception.

When setting values for these parameters, keep in mind the following guidelines:

```
(MAX_NUMBER_OF_QUERIES * network-buffer +
  MAX_NUMBER_OF_QUERIES * merge-sort-buffer * parallel-sort +
  mem-array-threshold + smgr-cache-size ) < max-memory-limit
```

and

```
(MAX_NUMBER_OF_QUERIES * network-buffer +
  MAX_NUMBER_OF_QUERIES * merge-sort-buffer * parallel-sort +
  mem-array-threshold + smgr-cache-size) *
 (number of instances on host ) <= 75% of RAM
```

where MAX_NUMBER_OF_QUERIES is the maximum number of concurrent queries allowed in the system. See more on MAX_NUMBER_OF_QUERIES below.

2.6.2. Configuring CPU Usage

SciDB provides the following parameters for configuring the usage of your CPUs (aka "cores"):

- *execution-threads* (number of threads): Controls the number of threads allocated to handling client requests. This number is closely related to the maximum number of queries that SciDB can run concurrently. In fact, note the following relationship:

  ```
  execution-threads = MAX_NUMBER_OF_QUERIES + 2
  ```

  Usually, each running query uses one of these threads for execution.

- *result-prefetch-threads* (number of threads): Controls the total number of threads available to all queries together. This parameter can be used to adjust the level of parallelism within a query (in addition to the main execution thread). Any given query is not guaranteed to have access to all of the threads because it may be competing with other running queries.
- *result-prefetch-queue-size* (number of threads): The maximum number of threads that a given query can attempt to use.

When setting values for these parameters, keep in mind the following guidelines:

```
result-prefetch-queue-size * MAX_NUMBER_OF_QUERIES =
  result-prefetch-threads
```

and

```
(execution-threads + result-prefetch-threads) * (number of instances
on host ) ~= (number of CPU cores on host) + 2
```

The last relationship may not be true in some cases, depending on the work load. For example, if the work load is very CPU-intensive and not much IO is involved, the number of threads should be slightly larger than the number of cores. However, if there is a mix of CPU and IO in the work load, increasing *result-prefetch-threads* may be beneficial.

The best values for *execution-threads* and *result-prefetch-threads* should be determined empirically.

2.6.3. Configuring Disk Usage

SciDB provides the following *chunk-segment-size* (megabytes) parameter for configuring the usage of your disks.

If this is 0, then chunks are stored contiguously and densely in the storage file. In this mode, storage is not reclaimed. In this case, the `remove()` operator does not have any effect on disk usage.

If this is set to a non-zero value, the storage file is split evenly into segments of the specified size. In this mode, when an array is removed, all segments that belong to this array are marked available for reuse—and can be reused by other arrays in the future. In this mode, a segment may only contain data from exactly one array, which means that each array will occupy at least one full segment on disk.

Currently, an array chunk size may not exceed the segment size. We recommend a value of between 64 and 128 MB, subject to the above restriction.

The general guideline: In a multi-disk host system, all SciDB instances on the host should be equally spread across all available disks. When selecting the number of SciDB instances per host, having several instances per disk may be useful in increasing the disk utilization.

2.6.4. Configuration Example

This section suggests configuration settings for a small, multi-disk SciDB installation.

Note

Some of the parameters mentioned in this example are not discussed in the preceding sections. For details, see Section 2.5.4, “SciDB Configuration Parameters”.

Suppose that we have a cluster with homogeneous motherboards, each motherboard has 16GB RAM and 3 disks. In this case, it is natural to use 3 SciDB instances per motherboard.

We will leave 1GB of RAM for the OS. We use the following settings:

- smgr-cache-size = 1024
- mem-array-threshold = 1024
- merge-sort-buffer = 128
- network-buffer = 512
- replication-send-queue-size = 500
- replication-receive-queue-size = 500
- max-memory-limit = 5000

In this case, we use 1GB for the SMGR cache. We also use 1GB for the memory array cache.

For the send and receive queue sizes, we assume that the average message size is about 1MB, and allocate 1GB total to the replication queue—which is only used when executing stored queries.

We allocate 512MB for other network usage.

With these settings, the system uses about 1GB of RAM when it is at rest, and somewhere between 3-3.5GB footprint while running queries.

The other 2-1.5GB is "breathing room" for various temporary results, operator and user code overhead, and so on.

By setting the max-memory-limit to 5000, SciDB does not allow this system to use more than 5000MB of memory per instance. Note that when the system is running a query using multiple threads, it is fair to expect that each thread has one or several array chunks in memory, which adds to the memory footprint.

Now suppose further that our motherboard with 3 instances has 24CPU cores. We want each instance to use 8 cores. CPU resources are more elastic, so we do not need to leave a core "for the operating system."

Suppose we want to support up to 2 concurrent queries, up to 4 threads per query. Our parameters look like this:

- execution-threads = 4
- operator-threads = 4
- result-prefetch-threads = 8
- result-prefetch-queue-size = 4

In this case, when more than 2 queries are submitted to SciDB, the system begins to execute the first two, and places the rest on the queue. In general, the value of *`operator-threads`* should always equal the value of *`result-prefetch-queue-size`*.

2.7. Initializing and Starting SciDB

2.7.1. 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:

`init_syscat`	Initialize the SciDB system catalog. **Warning**: This will remove any existing SciDB arrays from the current namespace and recreate the system catalog entries for this database. This must be run as user 'root' or as user 'Postgres' since it requires administrative privileges to the Postgres database.
`initall` and `initall-force`	Initialize the SciDB instances. This initializes SciDB on coordinator and worker instances and register them in the system catalog. **Warning**: This will delete all data and log files corresponding to these instances. initall-force forces initialization without prompting the user.
`startall`	Start the SciDB database service.
`stopall`	Stop the SciDB database service.
`status`	Show the status of the SciDB service.
`dbginfo`	Collect debugging information by getting all logs, cores, and install files.
`dbginfo-lt`	Collect only stack and log information for debugging.
`version`	Display SciDB version number information.
`check-pids`	List process IDs of SciDB on all instances.
`check-version`	Display the SciDB version information on each instance. This is useful in verifying that all instances in the cluster are running the same version.

The *db* argument is the name of the SciDB cluster you want to create or get information about.

The configuration file is set by default to `/opt/scidb/13.1/etc/config.ini`. If you want to use a custom configuration file for a particular SciDB cluster, use the *conffile* argument.

Run the following command to initialize SciDB on the server. If the SciDB user has sudo privileges, everything will be done automatically (otherwise see Section 2.4.3, “System Catalog Setup” for additional Postgres configuration steps):

First, initialize the system catalog database in Postgres. This must be done as user Postgres, therefore, this command must be run as user Postgres, or as superuser as follows.

```
sudo -u postgres /opt/scidb/13.1/bin/scidb.py init_syscat db
```

The remainder of the commands are runs as user 'scidb.'

```
scidb.py initall db
```

Warning

This will reinitialize the SciDB database. Any arrays that you have created in previous SciDB sessions will be removed and corresponding storage files will be deleted.

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

```
scidb.py startall db
```

This will report the status of the various instances:

```
scidb.py status db
```

This will stop all SciDB instances:

```
scidb.py stopall db
```

2.7.2. SciDB Logs

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`* for the server *`M`* instance *`N`*. Consider an installation with four servers, S1, S2, S3, and S4, and 8 instances per server.

- S1 is the coordinator server, and contains 1 coordinator instance and 7 worker instances in the following directories: *`base-path`*/000/0, *`base-path`*/000/1, *`base-path`*/000/2, ... *`base-path`*/000/7
- S2 is server, and contains 8 worker instances in the following directories: *`base-path`*/001/1, *`base-path`*/001/2, ... *`base-path`*/001/8
- S3 is server, and contains 8 worker instances in the following directories: *`base-path`*/002/1, *`base-path`*/002/2, ... *`base-path`*/002/8
- S4 is server, and contains 8 worker instances in the following directories: *`base-path`*/003/1, *`base-path`*/003/2, ... *`base-path`*/003/8

2.8. SciDB on Amazon EC2

Amazon Elastic Compute Cloud (EC2) is a central part of Amazon.com's cloud computing platform. A public Amazon Machine Image (AMI) from Paradigm4 is available on EC2 as `SciDB_12_10-Ubuntu-10_04-Sep-14-2012 (ami-80e950e9)`. This AMI has been pre-installed and pre-configured with release 13.1 of SciDB.

Note

> You need to have an Amazon Web Services account, and there is a small hourly fee associated with using the Amazon EC2 service.

1. Use Amazon Web Services (AWS) to create and set up a virtual Amazon Machine Instance. For details, see http://aws.amazon.com.

   You can use the `ami-80e950e9` image as a starting point.

2. AWS provides a configuration wizard for creating your image. You should be able to accept most of the defaults. Some parameters to note:

   - For the **Availability Zone**, we recommend that you choose `us-east-1a`.
   - You should create a key pair if you do not already have one.
   - You should create a security group that uses SSH.

3. Start your AMI and connect to it. On the details page for your AMI, not the URL. For example, if your image URL is e, ec2-23-20-246-232.compute-1.amazonaws.com, you could issue the following command in a terminal:

```
ssh -i ~/.ssh/myKeyFile.pem -o StrictHostKeyChecking=no
 ubuntu@ec2-23-20-246-232.compute-1.amazonaws.com
```

4. Execute the following commands to initialize local storage:

```
./osinit.sh
./pginit.sh
```

5. Execute the following command to get the latest SciDB repository:

```
sudo apt-get update
```

6. Login as user scidb (password scidb123) and execute the following script to initialize and start SciDB.

```
./dbinit.sh
```

Your SciDB installation is now ready to use. As user 'scidb', you can now load data, run queries and use your SciDB installation.

```
scidb@ip-10-191-93-50:~$ iquery -aq "list('instances')"

No,name,port,instance_id,online_since,instance_path
0,"127.0.0.1",1239,0,"2012-09-12 18:52:28","/mnt/xldb/000/0"
1,"127.0.0.1",1240,1,"2012-09-12 18:52:28","/mnt/xldb/000/1"
```

Chapter 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 dbname
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 lang` 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 instance, passes in a query, and prints out the coordinator instance'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` do this:

```
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
```

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+, dcsv, dense, lcsv+, sparse, and lsparse. These options are described in the following table:

Output Option	**Description**
auto (default)	SciDB array format.
csv	Comma-separated values.
csv+	Comma-separated values with dimension indices.
dcsv	Format used in most doc examples. Visually distinguishes dimensions from attributes.
dense	Ideal for viewing 2-dimensional matrices. Displays empty cells as parentheses. Not recommended for very sparse arrays.
lcsv+	Comma-separated values with dimension indices and a boolean flag attribute EmptyTag showing if a cell is empty.
lsparse	Sparse SciDB array format and a boolean flag attribute EmptyTag showing if a cell is empty.
sparse	Sparse SciDB array format.

To see a list of the `iquery` switches and their descriptions, type `iquery -h` or `iquery --help` at the command line. The switches are explained in the following table:

iquery Switch Option	**Description**
`-c [ --host ]` *host_name*	Host of one of the cluster instances. Default is 'localhost'.
`-p [ --port ]` *port_number*	Port for connection. Default is 1239.
`-q [ --query ]` *query*	Query to be executed.
`-f [ --query-file ]` *input_filename*	File with query to be executed.
`-r [ --result ]` *target_filename*	Filename with result array data.
`-o [ --format ]` *format*	Output format: auto, csv, csv+, lcsv+, sparse, lsparse. Default is 'auto'.
`-v [ --verbose ]`	Print the debugging information. Disabled by default.
`-t [ --timer ]`	Query setup time (in seconds).
`-n [ --no-fetch ]`	Skip data fetching. Disabled by default.
`-a [ --afl ]`	Switch to AFL query language mode. Default is AQL.
`-u [ --plugins ]`*path*	Path to the plugins directory.
`-h [ --help ]`	Show help.

iquery Switch Option	Description
`-V [ --version ]`	Show version information.
`ignore-errors`	Ignore execution errors in batch mode.

The `iquery` interface is case sensitive.

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 instance. Default is `localhost`.
port	Port for connection. Default is 1239.
afl	Start the session with the 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+, sparse, and lsparse.
plugins	Path to the plugins directory.

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

```
{
"host":"myhostname",
"port":9999,
"afl":true,
"timer":false,
"verbose":false,
"format":"csv+"
}
```

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 section demonstrates how to use iquery to perform simple array tasks like:

- Create a SciDB array
- Prepare an ASCII 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.

The are more detailed examples on creating a SciDB array in the chapter "Creating and Removing SciDB Arrays."

The following example creates an array, generates random numbers and stores them in the array, and saves the array data into a csv-formatted file.

1. 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)"
   ```

2. Save the 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)"
   ```

The following example creates an array, loads existing csv data into the array, performs simple conversions on the data, joins two arrays with related data set, and eliminates redundant data from the result.

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

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

2. Starting from a csv file, prepare a file to load into a SciDB array. Use the file *datafile.csv*, which is contained in the `doc/user/examples/` directory of your SciDB installation:

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

3. Convert the file to SciDB format with the command csv2scidb:

   ```
   csv2scidb -p SN -s 1 < doc/user/examples/datafile.csv
    > output_path/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.

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

   ```
   iquery -aq "load(target, 'output_path/datafile.scidb')"
   [("Truck",23.5),("Sedan",48.7),
   ("SUV",19.6),("Convertible",26.8)]
   ```

   You will need to use the full pathname for *`output_path`*. For example, if the file `datafile.scidb` is located in `/home/username/files`, you should use the string `'/home/username/files/datafile.csv'` for the load function argument.

5. 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, '/output_path/datafile.scidb')"
```

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

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

   ```
    [("Truck",23.5,9.98985),("Sedan",48.7,20.7024),
    ("SUV",19.6,8.33196),("Convertible",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)"
   ```

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

   ```
   Make,Type,Price
   Handa,Truck,26700
   Tolona,Sedan,31000
   Gerrd, SUV,42000
   Maudi,Convertible,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 < doc/user/examples/datafile_price.csv
    > output_path/datafile_price.scidb
   ```

   Create an array called storage:

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

   Load the `datafile_price.scidb` file into storage:

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

8. 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)"
   [("Handa","Truck",26700,"Truck",23.5,9.98985),
   ("Tolona","Sedan",31000,"Sedan",48.7,20.7024),
   ("Gerrd"," SUV",42000,"SUV",19.6,8.33196),
   ("Maudi","Convertible",45000,"Convertible",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.

9. 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 you do not specify are not included in the output.

10. Use the `join` and `project` operators to put the car data together. Use csv as the query output format:

```
iquery -o csv -aq "join(storage,project(target_new,mpg,kpl))"
make,type,price,mpg,kpl
"Handa","Truck",26700,23.5,9.98985
"Tolona","Sedan",31000,48.7,20.7024
"Gerrd"," SUV",42000,19.6,8.33196
"Maudi","Convertible",45000,26.8,11.3927
```

Chapter 4. Creating and Removing SciDB Arrays

SciDB organizes data as a collection of multidimensional arrays. Just as the relational table is the basis of relational algebra and SQL, the multidimensional array is the basis for SciDB.

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 *attributes* and *dimensions*.
  1. Each *attribute* contains data being stored in the array's cells. A cell can contain multiple attributes.
  2. 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*.

4.1. Create an Array

The AQL **`CREATE ARRAY`** statement creates a new array and specifies the array schema. The syntax of the **`CREATE ARRAY`** statement for a bounded array is:

```
CREATE ARRAY array_name <attributes> [dimensions]
```

The arguments for the **`CREATE ARRAY`** statement are as follows:

`array_name` The array name that uniquely identifies the array in the database. The maximum length of an array name is 1024 bytes. Array names may only contain the alphanumeric characters and underscores (_).

`attributes` The array attributes contain the actual data. You specify an attribute with:

- *Attribute name*: Name of an attribute. The maximum length of an attribute name is 1024 bytes. No two attributes in the same array can share a name. Attribute names may only contain alphanumeric characters and underscores (_).
- *Attribute type*: Type identifier. One of the data types supported by SciDB. Use the `list('types')` command to see the list of available data types.
- NULL (optional): Users can specify 'NULL' to indicate attributes that are allowed to contain null values. If this keyword is not used, all attributes must be non null, i.e., they cannot be assigned the special null value. If the user does not specify a value for such an attribute, SciDB will automatically substitute a default value.
- DEFAULT (optional): Allows the user to specify the value to be automatically substituted when a non NULL attribute lacks a value. If unspecified substitution uses system defaults for various types (0 for numeric types and "" for string). Note that if the attribute is declared as NULL, this clause is ignored.

dimensions Dimensions form the coordinate system for the array. The number of dimensions in an array is the number of coordinates or *indices* needed to specify an array cell. You specify dimensions with:

- *Dimension name*: Each dimension has a name. Just like attributes, each dimension must be named, and dimension names cannot be repeated in the same array. The maximum length of a dimension name is 1024 bytes. Optionally, you may want to create a non-integer dimension. In this case, you will need to specify the dimension data type in the name argument like this: *dimension_name*(*dimension_datatype*). Note that version 13.1 of SciDB and Paradigm4 support non-integer dimensions only partially. Dimension names may only contain alphanumeric characters and underscores (_).
- *Dimension start*: The starting coordinate of a dimension. The default data type is 64-bit integer. If you created a non-integer dimension, this argument is omitted.
- *Dimension end* or *: The ending coordinate of a dimension, or * if unbounded. The default data type is 64-bit integer for bounded dimensions.
- *Dimension chunk size*: Number of elements per chunk.
- *Dimension chunk overlap*: Number of overlapping dimension-index values for adjacent chunks.

The AQL **CREATE ARRAY** statement creates an array with specified name and schema. This statement creates an array:

```
AQL% CREATE ARRAY A <x: double, err: double> [i=0:99,10,0,
 j=0:99,10,0];
```

The array this statement created has:

- Array name A
- An array schema with:
  1. Two attributes: one with name x and type double and one with name err and type double
  2. Two dimensions: one with name i, starting coordinate 0, ending coordinate 99, chunk size 10, and chunk overlap 0; one with name j, starting coordinate 0, ending coordinate 99, chunk size 10, and chunk overlap 0.

This statement creates a different array:

```
AQL% CREATE ARRAY B <val:double>[sample(string)=6,6,0];
```

Array B has one attribute named val of type double and one dimension named sample of type string. Dimension sample has length 6, chunk size 6, and chunk overlap 0.

To delete an array with AQL, use the **DROP ARRAY** statement:

```
AQL% DROP ARRAY A;
```

4.2. Array Attributes

A SciDB array must have at least one attribute. The attributes of the array are used to store individual data values in array cells.

For example, you may want to create a product database. A 1-dimensional array can represent a simple product database where each cell has a string attribute called `name`, a numerical attribute called `price`, and a date-time attribute called `sold`:

```
AQL% CREATE ARRAY products <name:string,price:float default
 float(100.0),sold:datetime> [i=0:*,10,0];
```

Attributes are by default set to not null. To allow an attribute to have value NULL, add NULL to the attribute data type declaration:

```
AQL%
CREATE ARRAY product_null <name:string NULL,price:float
 NULL,sold:datetime NULL>
[i=0:*,10,0];
```

This allows the attribute to store NULL values at data load.

An attribute takes on a default value of 0 when no other value is provided. To set a default value other than 0, set the DEFAULT value of the attribute. For example, this code will set the default value of `price` to 100 if no value is provided:

```
AQL% CREATE ARRAY product_dflt <name:string, price:float default
 float(100.0), sold:datetime> [i=0:*,10,0];
```

4.2.1. NULL and Default Attribute Values

SciDB offers functionality to work with missing data. Consider the data set m4x4_missing.txt, shown here:

```
[
[(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 attribute val2 is missing for the elements at coordinates `{x=1, y=0}` and `{x=3, y=1}`, and the cell at `{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, `{x=2, y=2}`. By default, SciDB will leave empty cells unchanged and replace NULL attributes with 0:

```
AFL% CREATE ARRAY m4x4_missing
 <val1:double,val2:int32>[x=0:3,4,0,y=0:3,4,0];
```

```
AFL% load(m4x4_missing, '../examples/m4x4_missing.txt');
```

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

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

```
AFL% CREATE ARRAY m4x4_missing <val1:double,val2:int32 default
 5468>[x=0:3,4,0,y=0:3,4,0];
```

```
AFL% load(m4x4_missing, '../examples/m4x4_missing.txt');
```

```
[
[(0,100),(1,99),(2,98),(3,97)],
[(4,5468),(5,95),(6,94),(7,93)],
[(8,92),(9,91),(),(11,89)],
[(12,88),(13,5468),(14,86),(15,85)]
]
```

4.2.2. Codes for Missing Data

In addition to simple single-valued NULL substitution described in the previous section, SciDB also supports multi-valued NULLs using the notion of *missing reason codes*. Missing reason codes allow an application to optionally specify multiple types of NULLs and treat each type differently.

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".

The examples below show how to represent missing data in the load file. A question mark (?) 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) ...
```

Use the substitute operator to substitute different values for each type of NULL. For more information on NULL substitution, see the substitute operator reference.

4.2.3. Functions for Missing Values

There is a function, `missing_reason`, that returns the missing reason code for array data. For example, consider an array with the following data:

```
[(47),(null),(-21.1),(?100),(?50)]
```

Run `missing_reason` against the values in the array, and compare against the original values:

```
AFL% apply(A,MRcode, missing_reason(val));
```

```
{i} val,MRcode
{0} 47,-1
```

```
{1} null,0
{2} -21.1,-1
{3} ?100,100
{4} ?50,50
```

As illustrated by this example, we can see that `missing_reason` returns integer values as follows:

- For ordinary attribute values, it returns -1.
- For the standard null value, it returns 0.
- For null values that contain a missing reason code, it returns the code.

There is another function, `missing`, that returns the missing reason code from an integer array. Suppose, for example, that you have an array that contains integers where the values represent missing reason codes:

```
[(-1),(47),(23),(0),(127),(-1)]
```

Now run `missing` against the values in the array:

```
AFL% apply(arrayB,MRcode, missing(val));
```

```
{i} val,MRcode
{0} -1,<void>
{1} 47,?47
{2} 23,?23
{3} 0,null
{4} 127,?127
{5} -1,<void>
```

We can see that the function, `missing(`*x*`)`, returns values as follows:

- If x= -1, it returns <void>.
- If x= 0, it returns null.
- If x is an integer, it returns the missing reason code that corresponds to x.

4.3. Array Dimensions

A SciDB array must have at least one dimension. Dimensions form the coordinate system for a SciDB array. There are several special types of dimensions: dimensions with overlapping chunks, unbounded dimensions, and non-integer dimensions.

Note

The dimension size is determined by the range from the dimension start to end, so 0:99 and 1:100 would create the same dimension size.

4.3.1. 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` with the following schema:

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

Array A has has two dimensions, i and j. Dimension i has size 10, chunk size 5, and chunk overlap 1. Dimension j has size 30, chunk size 10, and chunk overlap 5. SciDB stores cells from the chunk overlap area in both of the neighboring chunks.

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.

4.3.2. Unbounded Dimensions

An array dimension can be created as an unbounded dimension by declaring the high boundary as '*'. When the high boundary is set as * the array boundaries are dynamically updated as new data is added to the array. This is useful when the dimension size is not known at **CREATE ARRAY** time. For example, this statement creates an array named open with two dimensions:

- Bounded dimension I of size 10, chunk size 10, and chunk overlap 0
- Unbounded dimension J of size *, chunk size 10, and chunk overlap 0.

```
AQL% CREATE ARRAY open <val:double>[I=0:9,10,0,J=0:*,10,0];
```

4.3.3. Non-integer Dimensions and Mapping Arrays

Note

Version 13.1 of SciDB only partially supports non-integer dimensions.

Basic arrays in SciDB use the int64 data type for dimensions. SciDB also supports arrays with non-integer 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. Such arrays are useful when you are transforming data into multidimensional format where some dimensions represent factors or categories.

For example, the array winners has a non-integer dimension named ID:

```
AQL% CREATE ARRAY winners <person:string, time:double>
 [year=1996:2008,1000,0, event(string)=3,1000,0]
```

```
AFL% show(winners)
```

```
winners

< person:string,
time:double >

[year=1996:2008,1000,0,
event(string)=3,1000,0]
```

Each of the dimensions year and event is a special one-dimensional SciDB array mapping each distinct value in the dimension to an integer coordinate.

```
AQL% SELECT * FROM winners:event;
```

```
{no} value
{0} "dash"
{1} "marathon"
{2} "steeplechase"
```

The attributes of the array `person` and `time` are:

```
AQL% SELECT * FROM winners;
```

```
{year,event} person,time
{1996,"dash"} "Bailey",9.84
{1996,"marathon"} "Thugwane",7956
{1996,"steeplechase"} "Keter",487.12
{2000,"dash"} "Greene",9.87
{2000,"marathon"} "Abera",7811
{2000,"steeplechase"} "Kosgei",503.17
{2004,"dash"} "Gatlin",9.85
{2004,"marathon"} "Baldini",7855
{2004,"steeplechase"} "Kemboi",485.81
{2008,"dash"} "Bolt",9.69
{2008,"marathon"} "Wanjiru",7596
{2008,"steeplechase"} "Kipruto",490.34
```

4.4. Changing Array Names

An array name is its unique identifier. You can use the AQL **`SELECT ... INTO`** statement to copy an array into another array with a new name.

This means that both `winners` and `OlympicWinners` are distinct arrays in the database. To change an array name use the `rename` command:

```
AFL% rename(winners, OlympicWinners);
```

You can use the `cast` command to change the name of the array, array attributes, and array dimensions. Unlike `rename`, the `cast` operator returns a new array with a few differences in the array schema relative to the input array. A single cast can be used to rename multiple items at once, for example, one or more attribute names and/or one or more dimension names. The input array and template array must have the same numbers and types of attributes and the same numbers and types of dimensions.

```
AFL% show(OlympicWinners)
```

```
OlympicWinners

< person:string,
time:double >

[year=1996:2008,1000,0,
event(string)=3,1000,0]
```

This query creates an array `winnerGrid` with integer dimensions that has renamed attributes `LastName` and `elapsedTime` and dimensions `Year` and `Event`.

```
AQL% SELECT * INTO winnerGrid FROM cast(OlympicWinners, < LastName:
 string, elapsedTime: double>
[x=1996:2008,1000,0,y=0:*,1000,0] );
```

```
{x,y} LastName,elapsedTime
{1996,0} "Bailey",9.84
{1996,1} "Thugwane",7956
{1996,2} "Keter",487.12
{2000,0} "Greene",9.87
{2000,1} "Abera",7811
{2000,2} "Kosgei",503.17
{2004,0} "Gatlin",9.85
{2004,1} "Baldini",7855
{2004,2} "Kemboi",485.81
{2008,0} "Bolt",9.69
{2008,1} "Wanjiru",7596
{2008,2} "Kipruto",490.34
```

```
winnerGrid

< LastName:string,
elapsedTime:double >

[x=1996:2008,1000,0,
y=0:2,1000,0]
```

4.5. Database Design

4.5.1. Selecting Dimensions and 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 are:

- Dimensions provide selectivity and efficient access to array data. Any coordinate along which selection queries must be performed constitutes a good choice of dimension. If you want to select data subject to certain criteria (for example, all products of price greater than $100 whose brand name is longer than six letters that were sold before 01/01/2010) you may want to design your database such that the coordinates for those parameters are defined by dimensions.

- 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 a good choice for a 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 are often performed on a 2-dimensional array structure.

These factors demand—or at least strongly encourage—that you choose to express certain variables as dimensions. In the absence of these factors, you can represent variables as either dimensions of attributes (although every array must have at least one attribute and at least one dimension). 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. Redimensioning an array is explained in Chapter 9, *Changing Array Schemas: Transforming Your SciDB Array*.

4.5.2. Chunk Size Selection

The selection of chunk size in a dimension plays an important role in how well you can query your data. If a chunk size is too large or too small, it will negatively impact performance.

To optimize performance of your SciDB array, you want each chunk to contain roughly 10 to 20 MB of data. So, for example, if your data set consists entirely of double-precision numbers, you would want a chunk size that contains somewhere between 500,000 and 1 million elements (assuming 8 bytes for every double-precision number).

When a multi-attribute SciDB array is stored, the array attributes are stored in different chunks, a process known as *vertical partitioning*. This is a consideration when you are choosing a chunk size. The size of an individual cell, or the number of attributes per cell, does not determine the total chunk size. Rather, the number of cells in the chunk is the number to use for determining chunk size. For arrays where every dimension has a fixed number of cells and every cell has a value you can do a straightforward calculation to find the correct chunk size.

When the density of the data in a data set is highly skewed, that is, when the data is not evenly distributed along array dimensions, the calculation of chunk size becomes more difficult. The calculation is particularly difficult when it isn't known at array creation time how skewed the data is. In this case, you may want to use the *repartitioning* functionality of SciDB to change the chunk size as necessary. Repartitioning an array is explained in Chapter 9, *Changing Array Schemas: Transforming Your SciDB Array*.

Chapter 5. Loading Data

A key part of setting up your SciDB array is loading your data. SciDB supports several load techniques, which together accommodate a wide range of scenarios for moving data into SciDB.

This chapter first presents the overview of loading data into SciDB: the basic principles and general steps that apply to all load techniques. Then it describes each load technique in turn. Finally this chapter presents some detailed information (such as handling errors during load) that applies to all the techniques.

5.1. Overview of Moving Data Into SciDB

You typically load data into SciDB one array at a time. In most situations, if you need three arrays, you will perform three separate loads. Regardless of the specific data-loading technique you use, the general steps for moving data into SciDB are as follows:

1. Visualize the shape of the data as you want it to appear in a SciDB array.

   Remember, the goal of loading data into SciDB is to make it available for array processing. Before you load the data, you should assess your analytical needs to determine what arrays you will need. You also must determine for each array what variables it will include and which of those variables will be dimensions and which will be attributes. For more information about creating arrays, see Section 4.1, "Create an Array".

   Depending on the data-loading technique you choose, this preliminary assessment might or might not include determining chunk sizes and chunk overlaps.

2. Prepare the data files for loading into SciDB.

   Depending on the specific technique you are using, this can mean creating a binary file, a single file in SciDB format, or a CSV file.

3. Load the data into SciDB.

   In all cases, this will mean invoking the LOAD command, either explicitly or indirectly through the loadcsv.py shell command. Different techniques may require different command options and syntax.

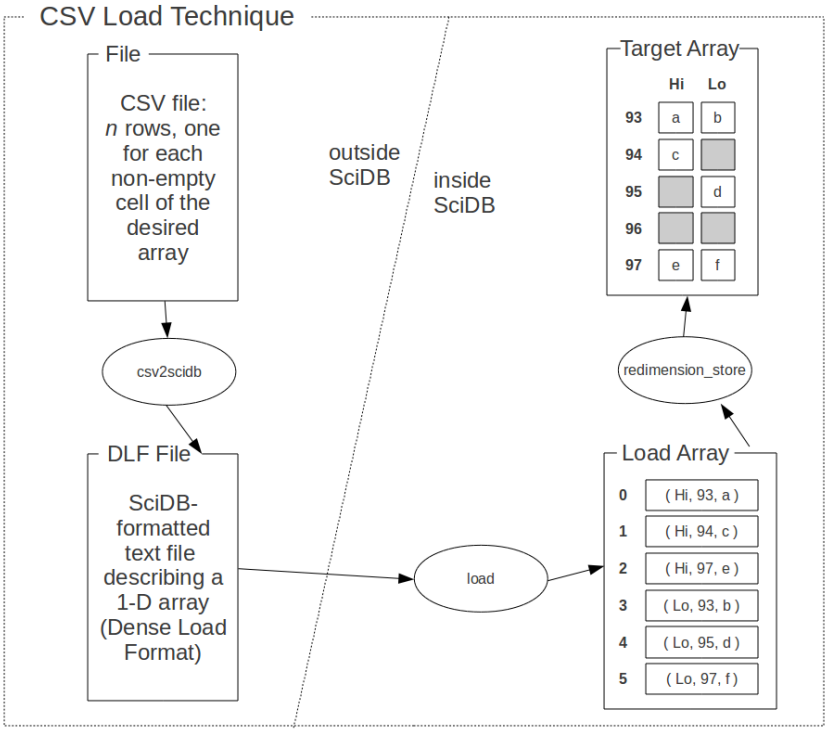
4. Rearrange the loaded data into the target array; the multi-dimensional array that supports your analytics.

   For some loading techniques, such rearrangement will typically involve the redimension_store operator and possibly involve the analyze operator. The program loadcsv.py, which is the linchpin of the parallel load technique, can perform this step for you.

5.2. Loading CSV Data

The CSV loading technique starts from a file in comma-separated-value (CSV) format, translates it into a SciDB-formatted text file describing a one-dimensional array, loads that file into a 1-dimensional array in SciDB, and rearranges that 1-dimensional array into the multi-dimensional shape you need to support your querying and analytics. The following figure presents an overview:

Figure 5.1. Overview of data load



Obviously, the CSV loading technique commends itself to situations in which your external application produces a CSV file. If you have a CSV file, you will use either the CSV loading technique or the parallel loading technique described elsewhere in this chapter. But if you can control the format that the external application uses to produce the data, you might choose to produce a CSV file and to use CSV loading technique in the following situations:

- For loading small arrays, such as arrays that will be lookup arrays or utility arrays that will be combined with other, larger arrays.
- For loading data into an intermediate SciDB array before you have determined the chunk sizes for the dimensions of the target array.

5.2.1. Visualize the Target Array

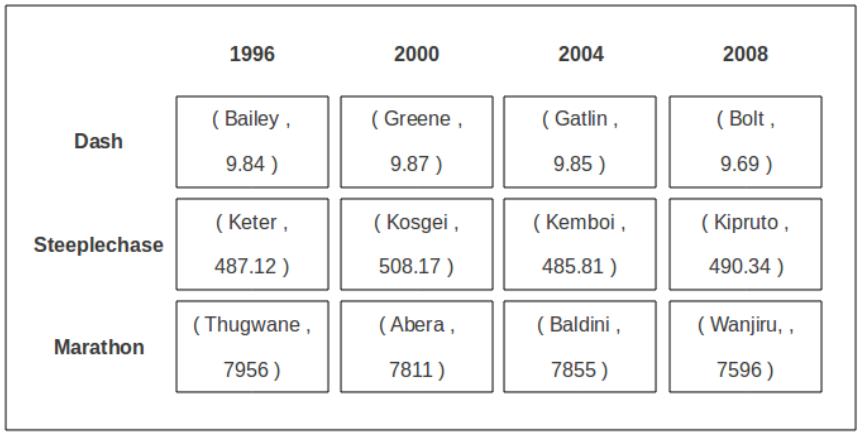
When using the CSV loading technique, visualizing the desired SciDB array means the following:

- Determine the attributes for the array, including the attribute name, datatype, whether it allows null values, and whether it has a default value to be used to replace null values.
- Determine the dimensions of the target array, including each dimension's name and datatype.

When using the CSV load technique, you can postpone contemplating each dimension's chunk size until after you have loaded the data into the intermediate 1-D array. This lets you use the analyze operator on that array to learn some simple statistics about the loaded data that can help you choose chunk sizes and chunk overlaps for each dimension of the target array.

For example, suppose you want an array with two dimensions and two attributes, like this:

Figure 5.2. Example of 2-dimensional array with 2 attributes



The dimensions are "year" and "event." The attributes are "person" and "time." The top right cell indicates, for example, that in 2008 Bolt won the dash in 9.69 seconds.

This simple, 12-cell array will be the target array used to illustrate steps of the CSV load technique.

5.2.2. Prepare the Load File

The CSV loading technique starts with a comma-separated-value (CSV) file. Each row of the file describes one cell of the target array, including its dimension values. Because the target array has four variables (two dimensions and two attributes), each row of the CSV file will have four values. Because the target array has twelve non-empty cells, the CSV file will have 12 rows of data, like this:

```
event,year,person,time
dash,1996,Bailey,9.84
dash,2000,Greene,9.87
dash,2004,Gatlin,9.85
dash,2008,Bolt,9.69
steeplechase,1996,Keter,487.12
steeplechase,2000,Kosgei,503.17
steeplechase,2004,Kemboi,485.81
steeplechase,2008,Kipruto,490.34
marathon,1996,Thugwane,7956
marathon,2000,Abera,7811
marathon,2004,Baldini,7855
marathon,2008,Wanjiru,7596
```

To include a null value for an attribute, you have several choices:

- Leave the field empty: no space, no tab, no ASCII character whatsoever.
- Use a question mark—with nothing else—in place of the value. (By contrast, if the value you want to load is the question mark character itself, put it in quotation marks.
- Use a question mark immediately followed by an integer between 0 and 127 (inclusive). The integer you use is the "missing reason code" for the null value.

After you create the CSV file, you must convert it to the SciDB dense load format. For that, use the `csv2scidb` shell command. The `csv2scidb` command takes multicolumn CSV data and transforms it into a format that the SciDB loader will recognize as a 1-dimensional array with one attribute for every column of the original CSV file. The syntax of `csv2scidb` is:

```
csv2scidb [options]  < input-file  > output-file
```

Note

`csv2scidb` is accessed directly at the command-line and not through the `iquery` client.

To see the options for `csv2scidb`, type `csv2scidb --help` at the command line. The options for csv2scidb are:

```
csv2scidb: Utility for conversion of CSV file to SciDB input text
 format
 Usage: csv2scidb [options] [ < input-file ] [ > output-file ]
 Default: -f 0 -c 1000000 -q
 Options:
   -v version of utility
   -i PATH input file
   -o PATH output file
   -a PATH appended output file
   -c INT length of chunk
   -f INT starting coordinate
   -n INT number of instances
   -d char delimiter - default ,
   -p STR type pattern - N number, S string, s nullable-string,
      C char, c nullable-char
   -q Quote the input line exactly, simply wrap it in ()
   -s N skip N lines at the beginning of the file
   -h  prints this helpful error message
  You cannot specify both -q and a set of -p options.
```

This command will transform `olympic_data.csv` to SciDB load file format:

```
csv2scidb -s 1 -p SNSN < .../examples/olympic_data.csv > .../examples/
olympic_data.scidb
```

The -s flag specifies the number of lines to skip at the beginning of the file. Since the file has a header, you can strip that line with "-s 1". The -p flag specifies the types of data in the columns you are transforming. Possible values are N (number), S (string), s (nullable string), and C (char).

The file `olympic_data.scidb` looks like this:

```
$ cat ../examples/olympic_data.scidb
```

```
{0}[
("dash",1996,"Bailey",9.84),
("dash",2000,"Greene",9.87),
("dash",2004,"Gatlin",9.85),
("dash",2008,"Bolt",9.69),
("steeplechase",1996,"Keter",487.12),
```

```
("steeplechase",2000,"Kosgei",503.17),
("steeplechase",2004,"Kemboi",485.81),
("steeplechase",2008,"Kipruto",490.34),
("marathon",1996,"Thugwane",7956),
("marathon",2000,"Abera",7811),
("marathon",2004,"Baldini",7855),
("marathon",2008,"Wanjiru",7596)
]
```

The square braces show the beginning and end of the array dimension. The parentheses enclose the individual cells of the array. There are commas between attributes in cells and between cells in the array.

5.2.3. Load the Data

After you prepare the file in the SciDB dense-load format, you are almost ready to load the data into SciDB. But first you must create an array to serve as the load array. The array must have one dimension and N attributes, where N is the number of columns in the original CSV file. For the olympic data, the array that you create must have four attributes, like this:

```
AQL% CREATE ARRAY winnersFlat < event:string, year:int64,
 person:string, time:double > [i=0:*,1000000,0];
```

Within the preceding CREATE ARRAY statement, notice the following:

- The attribute names: Even if you plan to delete the 1-dimensional load array after you create the target 2-dimensional, 2-attribute array—the attribute names matter. You should name the attributes as you expect to name the corresponding attribute and dimensions in the array you will ultimately create to support your analytics.
- The order of attributes: You must declare the attributes in the same left-to-right order as the values that appear on each line of the CSV file.
- The dimension name: The dimension name ("i" in this case) is uninteresting. You can use any name, because that dimension does not correspond to any variable from your data set and that dimension will not appear in any form in the target array. Remember that within the load array, every variable of your data appears as an attribute. These variables are not rearranged into attributes and dimensions until the last step of the procedure. (Although the dimension name is uninteresting, its values will correspond to the corresponding row number in the CSV file.)
- The chunk size (in this case, 1000000) for the dimension: Even though you are likely to use the winnersFlat array only briefly and perhaps delete it after you populate the target array, the chunk size matters because it can affect performance of the load and of the next step: the redimension_store.
- The chunk overlap (in this case, 0) for the dimension: If you are using the load array briefly—only as the target of the load operation and as the source of the subsequent redimension-store operation—then there is no need for chunks in the load array to overlap at all.

For more information about chunk size and overlap, see the Basic Architecture section.

After you create the target array, you can populate it with data using the `LOAD` statement:

```
AQL% LOAD winnersFlat FROM '../examples/olympic_data.scidb';
```

The data file paths in the AFL and AQL commands are relative to the working directory of the server.

5.2.4. Rearrange As Necessary

After you establish the load array, you can use SciDB features to translate it into the target array whose shape accommodates your analytical needs. Of course, you should have the basic shape of the target array in mind from the outset—perhaps even before you create the CSV file.

There are, however, some characteristics of arrays beyond these basics. These include the chunk size and chunk overlap value of each dimension. Before you choose values for these parameters, you can use the SciDB analyze operator to learn some simple statistics about the data in the array you just loaded. Here is the command to analyze the array winnersFlat:

```
AQL% SELECT * FROM analyze(winnersFlat);
```

```
{attribute_number}
 attribute_name,min,max,distinct_count,non_null_count
{0} "event","dash","steeplechase",3,12
{1} "person","Abera","Wanjiru",12,12
{2} "time","9.69","7956",12,12
{3} "year","1996","2008",4,12
```

For the simple example presented here, the simple statistics reveal little of interest. For large arrays however, the data can be illuminating and can influence your decisions about chunk size and chunk overlap.

For more information about chunk size and overlap, see the Basic Architecture section. For more information about the analyze operator, see analyze in the AFL Operator Reference.

The following command creates the target array:

```
AQL% CREATE ARRAY winners <person:string, time:double>
[year=1996:2008,1000,0, event(string)=3,1000,0]
```

The result of that command is an array that can accommodate the data about olympic winners. To populate this array with the data, use the following command:

```
AFL% redimension_store(winnersFlat,winners);
```

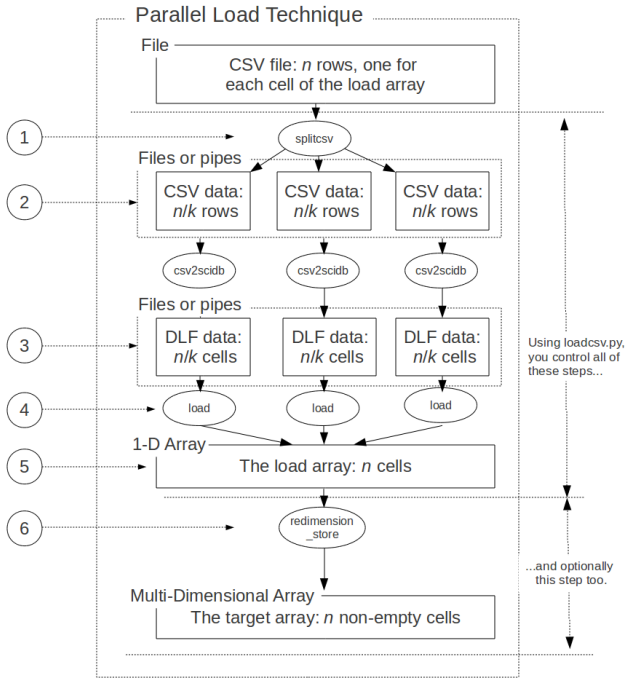
The result of this command is the desired array; you have completed the CSV load procedure.

5.3. Parallel Load

Like the CSV load technique, the parallel technique starts from a single CSV file. However, there are significant differences that can yield much faster load performance. Parallel load separates the CSV file into multiple files and distributes those files among the server instances in your SciDB cluster, allowing those instances to work in parallel on the load. In addition, parallel load can transfer data through pipes (rather than through materialized intermediate files), which also improves performance.

The following figure presents an overview:

Figure 5.3. Parallel load technique



In the figure, notice the following:

1. The program loadcsv.py performs a number of steps, starting with partitioning the original CSV file into k distinct subsets, where k is the number of SciDB instances in the cluster.
2. The k subsets of the original CSV file can be files or pipes. Pipes are faster, but you can use files to troubleshoot your load processes.
3. The program loadcsv.py converts each of the individual CSV files into data that conforms to the SciDB dense file format. Here too, the data can be expressed as files or pipes.
4. The program invokes the SciDB load k times, once for each dense-load-format file. Each of these load operations runs on a separate SciDB instance in your cluster.
5. The resulting 1-dimensional array is equivalent to the load array that would be produced by the (non-parallel) CSV load technique.
6. You can run redimension_store explicitly, or you can instruct loadcsv.py to do it for you with the -A command line switch.

The parallel loading technique is recommended for situations in which your external application produces a very large CSV file.

5.3.1. Visualize the Target Array

When using the parallel loading technique, visualizing the target SciDB array means the following:

- Determine the attributes for the array, including attribute name, datatype, whether it allows null values, and whether it has a default value to be used to replace null values.

- Determine the dimensions of the array, including each dimension's name and datatype.

As with the CSV load technique, you can postpone contemplating each dimension's chunk size until after you have loaded the data into the intermediate 1-D array. This lets you use the analyze operator on that array to learn some simple statistics about the loaded data that can help you choose chunk sizes and chunk overlaps for each dimension of the multi-dimensional array you desire.

5.3.2. Load the Data

The linchpin of the parallel load technique is the program loadcsv.py. Its primary input is a single CSV file and its primary result is a 1-D SciDB array: the load array. Besides its primary input, you can specify additional parameters to the program with command-line switches. Likewise, you can use switches to control the by-products of the parallel load operation.

The syntax of `loadcsv.py` is:

```
loadcsv.py [options]
```

Note

`loadcsv.py` is accessed directly at the command-line and not through the `iquery` client. Furthermore, you must run loadcsv.py from the SciDB administrator account.

Note that you can also load binary data in parallel. Currently, there is no binary analogue for the loadcsv.py program. You must use the load operator directly, and specify a parameter that instructs SciDB to perform the load in parallel. For details, see the load operator reference topic.

To see the options for `loadcsv.py`, type `loadcsv.py --help` at the command line. The options for loadcsv.py are:

Option	Details
`-h, --help`	Show this help message and exit
`-d DB_ADDRESS`	SciDB coordinator host name or IP address
`-p DB_PORT`	SciDB coordinator port. Default=1239
`-r DB_ROOT`	SciDB installation root folder. Default=/opt/scidb/*sciDB_version*
`-i INPUT_FILE`	CSV input file. Default=stdin
`-n SKIP`	Number of lines to skip. Default=0.
`-t TYPE_PATTERN`	CSV field types pattern: : N number, S string, s nullable string, C char. For example: "NNsCS"
`-D DELIMITER`	Delimiter. Default is a comma (,).
`-f STARTING_COORDINATE`	Starting coordinate. Default=0.
`-c CHUNK_SIZE`	Chunk size. Default=500,000
`-o OUTPUT_BASE`	Output file base name
`-m`	Create intermediate CSV files (not FIFOs)
`-l`	Leave intermediate CSV files
`-M`	Create intermediate dense-load-format (DLF) files (not FIFOs)
`-L`	Leave intermediate DLF files (not FIFOs)

-P SSH_PORT	SSH Port. Default is your system default.
-u SSH_USERNAME	SSH username
-k	SSH key/identity file
-b	SSH bypass strict host key checking
-a LOAD_NAME	Load array name
-s LOAD_SCHEMA	Load array schema
-w SHADOW_NAME	Shadow array name
-e ERRORS_ALLOWED	Number of load errors allowed per instance. Default=0.
-x	Remove load and shadow arrays before loading (if they exist)
-A TARGET_NAME	Target array name
-S TARGET_SCHEMA	Target array schema
-X	Remove target array before loading (if it exists)
-v	Display verbose messages
-V	Display SciDB version information
-q	Quiet mode

The command-line switches work in combination to control these aspects of the parallel load process:

- The operation of csv2scidb

  Remember, loadcsv.py invokes csv2scidb, a utility that itself requires some switches. On the loadcsv.py command line, you use the -i switch to indicate the location of the input CSV file, -n to indicate the number of lines at the top of the CSV file to be skipped, -t to indicate the CSV field-type pattern, -f to indicate the starting dimension index, and -D to indicate the character delimiter.

- Location of SciDB instance, its data directory, and its attendant utilities.

  The loadcsv.py program needs to know details about the SciDB installation. You supply these details with -d, which indicates the host name or IP address of the coordinator instance, -p, which indicates the port number on which the coordinator instance is listening, and -r, which indicates the root installation folder containing the utilities csv2scidb, iquery, and splitcsv. (The utility splitcsv partitions the input CSV file into separate files to be distributed among the SciDB instances for parallel loading.)

- SSH connectivity

  The loadcsv.py program connects to the SciDB cluster through SSH. The program requires that each node in the cluster has SSH configured on the same port; use -P to indicate that port. Use -u to indicate the SSH user name. Use -k to supply the SSH Key/Identify file used to authenticate the the SSH user on the remote node. (loadcsv.py requires that password-less SSH is configured for every node in the cluster.)

  In certain situations, SSH authentication presents a confirmation step in the user interface. Use the -b switch to bypass this step. If you do not want to use -b (because it weakens security), you can instead manually connect through SSH to each node in the cluster before you use loadcsv.py.

- Characteristics and handling of the 1-dimensional load array.

  The loadcsv.py program can operate on an existing 1-dimensional array, create a new one, or even delete an existing one before creating a new one. You control this behavior with the switches -c, -a, -s, and -x. The switch -c controls the chunk size of the load array. Use -a to supply the name of the array. Use -s to supply a schema definition if you want loadcsv.py to create the load array for you. Use -x to empower

loadcsv.py to delete any existing array before creating the new one you described with the -a and -s switches. The -x switch is a safeguard to ensure that you do not inadvertently delete a 1-dimensional array that you need. The -x switch is meaningless if you do not supply both -a and -s.

- Characteristics and handling of the multi-dimensional target array.

  The loadcsv.py program can populate a multidimensional array to support your analytics. It can populate an existing array, create and populate a new one, or even delete an existing one before creating and populating a new one. You control this behavior with the switches -A, -S, and -X. Use -A to supply the name of the array. Use -S to supply a schema definition if you want loadcsv.py to create the target array for you. Use -x to empower loadcsv.py to delete any existing array before creating the new one you described with the -A and -S switches. The -X switch is a safeguard to ensure that you do not inadvertently delete a multi-dimensional array that you need. The -X switch is meaningless if you do not supply both -A and -S.

- Handling of load errors

  A later section of this chapter describes SciDB mechanisms for handling errors during load. These mechanisms include both a maximum error count you supply and a shadow array, which accommodates error messages that occur on specific cell locations of the 1-dimensional load array. When using loadcsv.py, you use the -e switch to establish the maximum error count (per SciDB instance working on the load) and -w to give the name of the shadow array.

- Location of pipes or intermediate files, and the optional persistence of intermediate files

  The program loadcsv.py can distribute the partitioned CSV data via files or via pipes. Pipes provide superior performance, but you can use files if you want. To request files, use -m. To request that such files be retained after the load operation (typically for debugging purposes), use -l.

  Likewise, the program can use either pipes or files to accommodate the output of the csv2scidb program—the SciDB-readable files to be loaded into the destination. By default, the dense-load-format result format will be set to a pipe. To request files, use -M. To request that such files be retained after the load operation (again, typically for debugging), use -L.

  Whether you use pipes or files, you can control the location of the output of the splitcsv utility—the utility that partitions the original CSV file. Use the -o switch. The parameter you supply with -o indicates the base name of each part of the partitioned output. For example, if the command line includes -o '/tmp/base', the various files or pipes on the individual server instances would be named:

  /tmp/base_0000

  /tmp/base_0001

  etc.

- Control of verbose/quiet mode for progress and status reporting.

  Use -v for verbose mode, -q for quiet mode, -h for help, and -V to show SciDB version information.

This command will load data from aData.csv, which contains one header row and three numeric columns, into the existing array aFlat:

```
loadcsv.py -n 1 -t NNN
           -a 'aFlat'
           -i './aData.csv'
```

This command will create the array aFlat and load data from aData.csv into it.

```
loadcsv.py -n 1 -t NNN
           -a 'aFlat'
           -s '<row:int64,col:int64,val:int64 null>
               [csvRow=0:*,500000,0]'
           -i './aData.csv'
```

This command loads data into the existing array aFlat using files instead of pipes:

```
loadcsv.py -n 1 -t NNN
           -a 'aFlat'
           -i './aData.csv'
           -o '/home/scidb/aData'
           -m -M
```

This command also uses files instead of pipes, and retains those files after the load operation:

```
loadcsv.py -n 1 -t NNN
           -a 'aFlat'
           -i './aData.csv'
           -o '/home/scidb/aData'
           -m -l -M -L
```

This command loads data into aFlat, and uses a shadow array and a maximum error count to handle load errors gracefully.

```
loadcsv.py -n 1 -t NNN
           -a 'aFlat'
           -i './aData.csv'
           -o '/home/scidb/aData'
           -e 10
           -w 'aFlatshadow'
```

5.3.3. Rearrange As Necessary

After you establish the 1-dimensional load array in SciDB, you can to translate it into the desired array whose shape accommodates your analytical needs: the target multi-dimensional array. Because you are using loadcsv.py, you have two choices for accomplishing this step. You can use redimension_store after loadcsv.py populates the load array. This step is identical to the analogous step described in the section on the CSV load technique.

Alternatively, you can instruct loadcsv.py to transform the 1-dimensional load array into the target multi-dimensional array. To achieve this, use the -A switch and optionally the -S and -X switches.

The following command loads data from the CSV file (aData.csv) into the existing 1-dimensional load array (aFlat) and rearranges that data into the existing target multi-dimensional array (aFinal).

```
loadcsv.py -n 1 -t NNN
           -a 'aFlat'
           -i './aData.csv'
           -o '/home/scidb/aData'
           -A 'aFinal'
```

The following command loads data from the CSV file into the load array, creates the target multidimensional array, and rearranges the data from the load array into the target array.

```
loadcsv.py -n 1 -t NNN
```

```
           -a 'aFlat'
           -i './aData.csv'
           -o '/home/scidb/aData'
           -A 'aFinal'
           -S '<val:int64 null>
               [row=1:*,1000,0,col=1:*,1000,0]'
```

The following command loads data from the CSV file into the load array, establishes a shadow array for the load operations, and rearranges the data from the load array into the target array.

```
loadcsv.py -n 1 -t NNN
           -a 'aFlat'
           -i './aData.csv'
           -o '/home/scidb/aData'
           -A 'aFinal'
           -w 'aFlatShadow
```

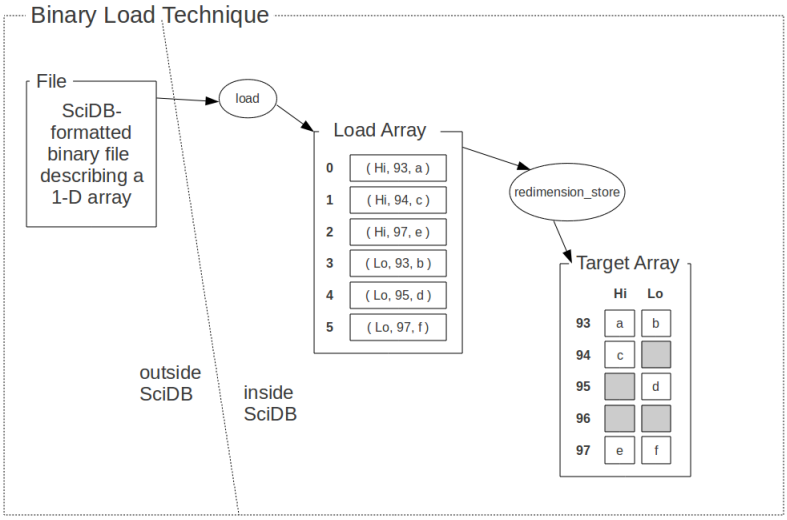
Note

The shadow array corresponds to the 1-dimensional load array, not the multidimensional target array.

5.4. Loading Binary Data

The binary loading technique starts from a binary file, loads it into a 1-dimensional array in SciDB, and rearranges that 1-dimensional array into the multidimensional shape you need to support your querying and analytics. The following figure summarizes.

Figure 5.4. Binary load technique



Obviously, the binary loading technique commends itself to situations in which your external application can produce a binary file. But if you can control the format that the external application uses to produce the data, you might choose to produce a binary file and to use the binary loading technique for loading

large arrays when you do not want to encounter the overhead involved in parsing CSV files and SciDB-formatted text files. For example, avoiding this overhead is especially desirable if your data includes many variables whose data type is double.

5.4.1. Visualize the Target Array

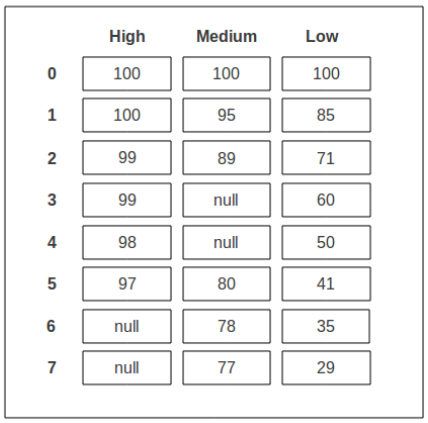
When using the binary loading technique, visualizing the desired multi-dimensional array means the following:

- Determine the attributes for the array, including attribute name, datatype, whether it allows null values, and whether it has a default value to be used to replace null values.
- Determine the dimensions of the array, including each dimension's name and datatype.

When using the binary load technique, you can postpone contemplating each dimension's chunk size until after you have loaded the data into the load array. This lets you use the analyze operator on that array to learn some simple statistics about the loaded data that can help you choose chunk sizes and chunk overlaps for each dimension of the multi-dimensional array you desire.

For example, suppose you want an array with two dimensions and one attribute, like this:

Figure 5.5. Visualizing the target array



	High	Medium	Low
0	100	100	100
1	100	95	85
2	99	89	71
3	99	null	60
4	98	null	50
5	97	80	41
6	null	78	35
7	null	77	29

The dimensions are "exposure" (with values High, Medium, and Low) and "elapsedTime" (with values from 0 to 7 seconds). The sole attribute is "measuredIntensity." The bottom right cell indicates, for example, that seven seconds after low exposure, the measured intensity is 29. Note that the desired array includes some null values for the measuredIntensity attribute.

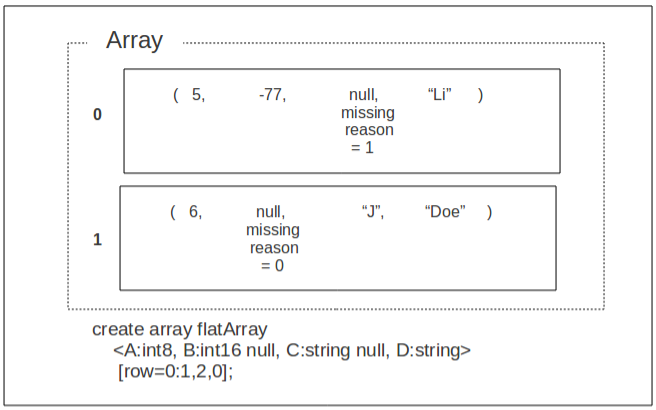
This simple, 24-cell array will be the target array used to illustrate steps of the binary load technique.

5.4.2. Prepare the Binary Load File

A SciDB binary load file represents a 1-dimensional SciDB array. The 1-dimensional array is dense; it has no empty cells (although it can have null values for nullable attributes). The binary load file represents each cell of the 1-dimensional array in turn; within each cell, the file represents each attribute in turn.

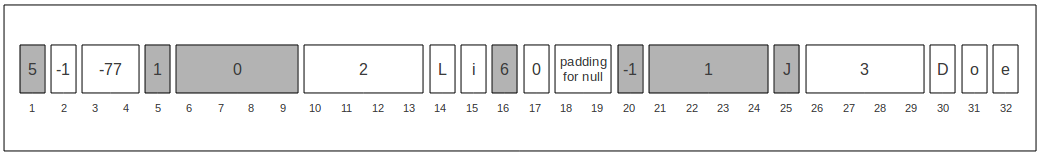
The following two figures illustrate. The first figure shows a very simple array: 1 dimension, four attributes, and only two cells. The figure also shows the AQL statement that created the array, revealing which attributes allow null values.

Figure 5.6. Example array created using binary load



The next figure represents the layout of this array within the corresponding binary load file.

Figure 5.7. Binary load file



The figure illustrates the following characteristics of a binary load file:

- Each cell of the array is represented in contiguous bytes. (But remember, some programs that create binary files will pad certain values so they align on word boundaries. This figure does not show such values. You can use the SKIP keyword to skip over such padding.)
- There are no end-of-cell delimiters. The first byte of the representation of the first attribute value of cell N begins immediately after the the last byte of the last attribute of cell N-1.
- A fixed-length data type that allows null values will always consume one more byte than the datatype requires, regardless of whether the actual value is null or non-null. For example, an int8 will require 2 bytes and an int64 will require 9 bytes. (In the figure, see bytes 2-4 or 17-19.)
- A fixed-length data type that disallows null values will always consume exactly as many bytes as that datatype requires. For example, an int8 will consume 1 byte and an int64 will consume 8 bytes. (See byte 1 or 16.)
- A string data type that disallows nulls is always preceded by four bytes indicating the string length. (See bytes 10-13 or 26-29.)
- A string data type that allows nulls is always preceded by five bytes: a null byte indicating whether a value is present and four bytes indicating the string length. For values that are null, the string length will be zero. (See bytes 5-9 or 20-24.)
- The length of a null string is recorded as zero. (See bytes 5-9.)

- If a nullable attribute contains a non-null value, the preceding null byte is -1. (See byte 2 or 20.)
- If a nullable attribute contains a null value, the preceding null byte will contain the missing reason code, which must be between 0 and 127. (See byte 5 or 17.)
- The file does not contain index values for the dimension of the array to be populated by the LOAD command. The command reads the file sequentially and creates the cells of the array accordingly. The first cell is assigned the first index value of the dimension, and each successive cell receives the next index value.

Storage for a given type is assumed to be in the x86_64 endian format.

Each value in the file must conform to a datatype recognized by SciDB. This includes native types, types defined in SciDB extensions, and user-defined types. For a complete list of the types recognized by your installation of SciDB, use the following AQL:

```
AQL% SELECT * FROM list('types');
```

5.4.3. Load the Data

After you prepare the file in the SciDB-recognized binary format, you are almost ready to load the data into SciDB. But first you must create the load array. The array must have one dimension and N attributes, where N is the number of variables (attributes and dimensions) in the target array. For the simple example about measured intensity after exposure, the array you create must have three attributes, like this:

```
AQL% CREATE ARRAY intensityFlat
 < exposure:string, elapsedTime:int64, measuredIntensity:int64 null >
 [i=0:*,1000000,0];
```

Within the preceding CREATE ARRAY statement, notice the following:

- The attribute names—Although the array intensityFlat is merely the load array—one that you might even delete after you create and populate the target 2-dimensional, 1-attribute array—the attribute names matter. You should name the attributes as you expect to name the corresponding attribute and dimensions in the array you will ultimately create to support your analytics.
- The order of the attributes—You should declare the attributes in the same left-to-right order as the values that appear in each record of the binary file.
- The null declaration for the measuredIntensity attribute—This is needed because the data includes some null values for that attribute.
- The dimension name—The dimension name ("i" in this case) is uninteresting. You can use any name, because that dimension does not correspond to any variable from your data set and that dimension will not appear in any form in the final array you eventually create. Remember, the binary load procedure loads the data into a 1-dimensional array where every variable of your data appears as an attribute. These variables are not rearranged into attributes and dimensions until the last step of the procedure.
- The chunk size (in this case, 1000000) for the dimension—Even though you might use the intensityFlat array only briefly and delete it after you establish and populate the target array, the chunk size of the load array matters because it can affect performance of the load and of the next step: the redimension_store. The chunk size you choose for the load array has no effect on the chunk sizes you will eventually choose for the exposure and elapsedTime dimensions of the target array.
- The chunk overlap (in this case, 0) for the dimension—For the intermediate array that exists only as the target of a load and as the source of a subsequent redimension_store, there is no need for chunks to overlap at all.

For more information about chunk size and overlap, see the Basic Architecture section.

After you create the load array, you can populate it with data using the LOAD statement:

```
AQL% LOAD intensityFlat FROM '../examples/intensity_data.bin'
     AS '(string,
          int64,
          int64 null)';
```

The file path of `intensity_data.bin` is relative to the SciDB server's working directory on the coordinator instance.

Notice the format string—the quoted text following the AS keyword. The LOAD command uses the format string as a guide for interpreting the contents of the binary file.

5.4.4. Loading Binary String Data

You must null-terminate strings when loading binary data. Also, count the terminating null character as part of the length of the string. For example, consider the following partial data file for a string attribute that does not allow nulls:

```
05 00 00 00 48 69 67 68 00 ...
07 00 00 00 4d 65 64 69 75 6D 00 ...
```

The first four bytes for each record contain the size of the string—5 bytes and 7 bytes respectively. The next bytes contain the string data:

```
05 00 00 00 48 69 67 68 00 ...
            H  i  g  h  \0
07 00 00 00 4d 65 64 69 75 6D 00 ...
            M  e  d  i  u  m  \0
```

The following example shows the string lengths for strings in the intensityFlat array:

```
AQL% SELECT exposure, strlen(exposure) FROM
```

```
{i} exposure,expr
{0} "High",4
{1} "High",4
{2} "High",4
{3} "High",4
{4} "High",4
{5} "High",4
{6} "High",4
{7} "High",4
{8} "Medium",6
{9} "Medium",6
{10} "Medium",6
{11} "Medium",6
{12} "Medium",6
{13} "Medium",6
{14} "Medium",6
{15} "Medium",6
{16} "Low",3
```

```
{17} "Low",3
{18} "Low",3
{19} "Low",3
{20} "Low",3
{21} "Low",3
{22} "Low",3
{23} "Low",3
```

5.4.5. Rearrange As Necessary

After you populate the load array, you can use SciDB features to translate it into the desired array whose shape accommodates your analytical needs. Of course, you should have the basic shape of the target array in mind from the outset—perhaps even before you created the binary file.

There are, however, some characteristics of arrays beyond these basics. These include the chunk size and chunk overlap value of each dimension. Before you choose values for these parameters, you can use the SciDB analyze operator to learn some simple statistics about the data in the 1-dimensional array you loaded. Here is the command to analyze the array intensityFlat:

```
AQL% SELECT * FROM analyze(intensityFlat)
```

```
{attribute_number}
 attribute_name,min,max,distinct_count,non_null_count
{0} "elapsedTime","0","7",8,24
{1} "exposure","High","Medium",3,24
{2} "measuredIntensity","29","100",16,20
```

Of course, for the simple example presented here, the simple statistics reveal little of interest. For large arrays however, the data can be illuminating and can influence your decisions about chunk size and chunk overlap. For more information about chunk size and overlap, see the Basic Architecture section in Introduction to SciDB. For more information about the analyze operator, see the analyze section in SciDB Operator Reference.

The following command creates the desired 2-dimensional, 1-attribute array:

```
AQL% CREATE ARRAY
            intensity
           <measuredIntensity:int64 null>
           [exposure(string)=3,3,0,
            elapsedTime=0:40000,10000,0];
```

The result of that command is an array that can accommodate the data about measured intensity after exposure. To populate this array with the data, use the following command:

```
AFL% redimension_store(intensityFlat,intensity);
```

The result of this command is the desired array; you have completed the binary load procedure.

5.4.6. Skipping Fields and Field Padding During Binary Load

During binary load, you can instruct the loader to skip some data in the file. This is useful when you want exclude entire fields from the load operation, and when you want to skip over some padded bytes that have been added to a field by the application that produced the binary file.

For skipping entire fields: From a binary file with N attributes, you can load a 1-dimensional SciDB array that has M attributes, where M < N. You do this with the SKIP keyword. Compare the following three pairs of AQL statements, which create and populate arrays excluding zero, one, and two fields of the same load file.

The first pair of statements includes all fields:

```
AQL% CREATE ARRAY
       intensityFlat
         < exposure:string,
           elapsedTime:int64,
           measuredIntensity:int64 null >
       [i=0:*,1000000,0];
```

```
AQL% LOAD intensityFlat
     FROM '../examples/intensity_data.bin'
     AS   '(string,
            int64,
            int64 null)';
```

The second pair of statements excludes a string field:

```
AQL% CREATE ARRAY intensityFlat_NoExposure
 < elapsedTime:int64, measuredIntensity:int64 null >
 [i=0:*,1000000,0];
```

```
AQL% LOAD intensityFlat_NoExposure
     FROM '../examples/intensity_data.bin'
     AS   '(skip,
            int64,
            int64 null)';
```

The third pair of statements excludes two int64 fields, one of which allows null values:

```
AQL% CREATE ARRAY  intensityFlat_NoTime_NoMeasurement
 < exposure:string >
 [i=0:*,1000000,0];
```

```
AQL% LOAD intensityFlat_NoTime_NoMeasurement
     FROM '../examples/intensity_data.bin'
     AS   '(string,
            skip(8),
            skip(8) null)';
```

The preceding pairs of AQL statements illustrate the following characteristics of the SKIP keyword:

- For variable-length fields, you can use the SKIP keyword without a number of bytes.
- For fixed-length fields, you can use the SKIP keyword with a number of bytes in parentheses.
- To skip a field that contains null values, use the NULL keyword after the SKIP keyword.

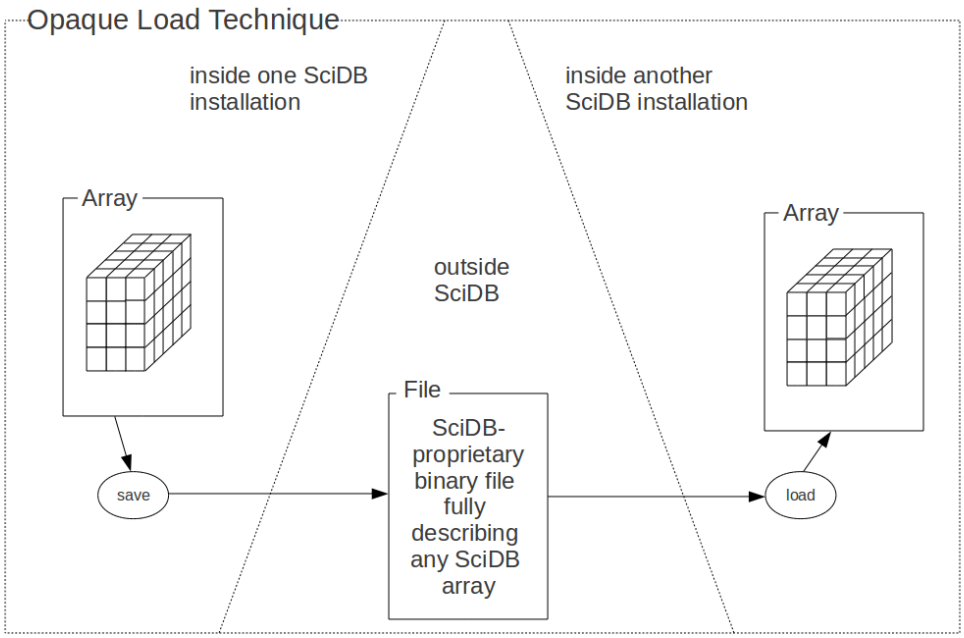
Note

When writing field values into a file, some programming languages will always align field values to start on 32-bit word boundaries.

5.5. Transferring Data From One SciDB Installation to Another

The data-loading technique that transfers array data from one SciDB installation to another is called the "opaque" technique, so named because the intermediate file format is not user-programmable. The opaque technique starts from any array in one SciDB installation, produces an external file, and loads that file into another SciDB installation—establishing an array that had the same dimensions, attributes, dimension indexes, and attribute values as the original array from the source installation. The following figure presents an overview.

Figure 5.8. Overview of opaque load technique



The opaque data-loading technique is recommended in the following situations:

- The source of the data is an existing SciDB array (rather than a CSV file or binary file).
- You want to use a simple procedure that requires few commands and few intermediate results
- You want to avoid the responsibility for ensuring that your load file is in the correct format.
- The array you are transferring includes a dimension whose datatype is not int64.

5.5.1. Visualize the Desired Array

When using the opaque loading technique, visualizing the SciDB array you want in the destination SciDB installation can be easy because the desired array—or something very close to it—already exists in the

source installation. In the most straightforward case, you can transfer the current version of the source array to the destination array without modification: the array in the destination installation will match the source array in all of the following ways:

- Dimensions: Same dimension names, datatypes, upper and lower bounds, index values, and the same order of dimensions.
- Attributes: Same attribute names and datatypes, and the same order of attributes within cells.
- Cells: Same cell values.
- Chunks: For each dimension, the same values for the chunk size and chunk overlap parameters.

Beyond this most straightforward case, there are cases in which you make slight adjustments to the array, either in the source installation before you create the file in opaque format, or in the destination installation when you create the array that will contain the data loaded from the file. The next two sections elaborate on these cases.

5.5.2. Prepare the File for Opaque Loading

The opaque loading technique can create a file describing the current version of any SciDB array. The following command accomplishes that for the array called "intensity."

```
SAVE
          intensity
     INTO CURRENT INSTANCE
          'intensity_data.opaque'
     AS   'OPAQUE';
```

The keywords `CURRENT INSTANCE` instruct SciDB to create the file in the SciDB working directory on the coordinator node. The keyword `OPAQUE` instructs SciDB to create the file in opaque format. The preceding SAVE statement writes the opaque-formatted file in this location on the coordinator instance of the source installation:

```
base-path/instance-folder/intensity_data.opaque
```

where *`base-path`* is the location of your SciDB working directory (as defined in the SciDB config.ini file), and the *`instance-folder`* is folder for the instance, in this case the coordinator, where the query is being run. For example, if the *`base-path`* is `/home/scidb/scidb-data/`, then the file will be saved as follows:

```
/home/scidb/scidb-data/000/0/intensity_data.opaque
```

You could also save to a different instance, rather than the current one:

```
SAVE
          intensity
     INTO INSTANCE 2
          'intensity_data.opaque'
     AS   'OPAQUE';
```

Assuming *`base-path`* is the same as in the previous save query, then this query would save the data to the following location:

```
/home/scidb/scidb-data/000/2/intensity_data.opaque
```

For details on SciDB paths, see Section 2.7.2, “SciDB Logs”.

If you want the resulting opaque-formatted file to describe something other than the original array—say, a subset of it or an array with an additional attribute—you can modify the array accordingly using various SciDB operators. There are two methods:

- In AQL, you can establish the result array you want, store it, and then use the SAVE command on the newly stored array. The AQL SAVE syntax does not currently support saving non-stored arrays, so you must explicitly store the array you want to SAVE to a load file.
- In AFL, you can establish the result array you want and use that result array as an operand of the SAVE operator. For more information, see the Save Operator Reference topic.

After you have saved the file in the working directory on the coordinator node, you need to move the file to a location where the other SciDB installation can access it when you run the load command there.

5.5.3. Load the Data

Once you have the opaque-formatted file in a location where the destination installation of SciDB can access it, you are almost ready to load the data. But first you must create an array as the target of the load operation. The array you create must match the source array in the following regards:

- Dimensions: The array in the destination installation must have the same number of dimensions as the source array. The left-to-right order of the dimensions must have the same datatypes as the source array. Note that the names of the dimensions need not match the names in the source array.
- Attributes: The array in the destination installation must have the same number of attributes as the source array. The left-to-right order of the attributes must have the same datatypes as the source array. Note that the names of the attributes need not match the names in the source array.

To ensure that you create a target array that is compatible with the to-be-loaded data, you should check the schema of the original array on the source installation. The following statement—run on the source installation of SciDB—reveals the information you need:

```
AFL% show(intensity)
```

```
intensity

< measuredIntensity:int64 NULL DEFAULT null >

[exposure(string)=3,3,0,
elapsedTime=0:40000,10000,0]
```

With that information, you can now create the array in the destination installation of SciDB. The following command creates an array that is compatible with the data in the opaque-formatted load file:

```
AQL% CREATE ARRAY intensityCopy
         < measuredIntensity:int64 NULL >
         [ exposure(string)=3,3,0,
           duration=0:40000,10000,0]
```

Notice that the array differs from the source array in two regards that do not compromise the compatibility with the opaque-formated load file. The source array is called "intensity" but the destination array is called "intensityCopy." A dimension of the source array is called "elapsedTime" but the corresponding dimension of the destination array is called "duration."

Now that the destination array exists, you can load the data into it:

```
AQL% LOAD intensityCopy
     FROM CURRENT INSTANCE '../examples/intensity_data.opaque'
     AS 'OPAQUE';
```

The result of this command is the array in the destination installation of SciDB. You have completed the opaque loading procedure.

5.6. Data with Special Values

This section discusses default attribute values for missing values, and some behavior for empty cells.

5.6.1. Data with Missing Values

Suppose you have a load file that is missing some values, like this file, `v4.scidb`:

```
[
 (0,100),(1,99),(2,),(3,97)
]
```

The load file `v4.scidb` has a missing value in the third cell. If you create an array and load this data set, SciDB will substitute 0 for the missing value:

```
AQL% CREATE ARRAY v4 <val1:int8,val2:int8>[i=0:3,4,0]

AQL% LOAD v4 FROM '../examples/v4.scidb'

[(0,100),(1,99),(2,0),(3,97)]
```

The out-of-the-box default value for each datatype is described in Chapter 10, *SciDB Data Types and Casting*.To change the default value, that is, the value the SciDB substitutes for the missing data, set the DEFAULT attribute option. This code creates an array `v4_dflt` with default attribute value set to 111:

```
AQL% CREATE ARRAY v4_dflt <val1:int8,val2:int8 default 111>[i=0:3,4,0]

AQL% LOAD v4_dflt FROM '../examples/v4.scidb'

[(0,100),(1,99),(2,111),(3,97)]
```

Load files may also contain null values, such as in this file, `v4_null.scidb`:

```
[
 (0,100),(1,99),(2,null),(3,97)
]
```

To preserve null values at load time, add the NULL option to the attribute type:

```
AQL% CREATE ARRAY v4_null <val1:int8,val2:int8 NULL> [i=0:3,4,0];

AQL% LOAD v4_null FROM '../examples/v4_null.scidb';
```

5.6.2. Empty Cells

In addition to missing values for attributes, SciDB arrays may also contain completely empty cells. For example, if we load `v4.scidb` into an array with a dimension that is larger than the supplied data, the array will contain empty cells:

```
AQL% CREATE ARRAY v6_dflt <val1:int8,val2:int8 default 111>
      [i=0:5,6,0]

AQL% LOAD v6_dflt FROM '../examples/v4.scidb'

[(0,100),(1,99),(2,111),(3,97),(),()]
```

Note that the last two cells of the array are empty. If you do not want empty cells, you can use the NOT EMPTY keywords when you create the array:

```
AQL% CREATE NOT EMPTY ARRAY v6_dflt_notEmpty <val1:int8,val2:int8
 default 111>[i=0:5,6,0]

AQL% LOAD v6_dflt_notEmpty FROM '../examples/v4.scidb'

[(0,100),(1,99),(2,111),(3,97),(0,111),(0,111)]
```

Note that for val2, the supplied default value (111) is used, and for val1, the default value of the int8 datatype is used (zero).

5.7. Handling Errors During Load

By default, if an error occurs during load, SciDB displays an error message to stdout and cancels the operation. Because load is designed to work on high volumes of data, the SciDB load facility includes a mechanism by which you can keep track of errors while still loading the error-free values. This mechanism is known as "shadow arrays."

During a load operation, SciDB can populate two arrays:

- The load array is populated with data from the load file.
- The shadow array is populated with error messages that occurred during the load.

The shadow array uses the same dimensions and dimension values as the load array.

For attributes, things are slightly different. If the load array has n attributes, the shadow array has n+1 attributes, as follows:

- For each attribute in the load array, the shadow array includes an identically named attribute. Although the names are identical, the datatypes are not. In the shadow array, each of these n attributes is a string.
- The shadow array includes one additional integer attribute named "row_offset." After the load operation, this attribute is populated for any cell that contains an error message. The value of row_offset describes SciDB's best estimate of the byte of the file on which the error occurs.

After a load operation that is largely successful but produces a few errors, most cells of the shadow array will be empty; such cells correspond to cells in the load array that were loaded without error. Within any non-empty cell in the shadow array, the row_offset contains an integer, and each string attribute is either null (indicating that the corresponding value was loaded into the load array without error) or populated with a message describing the error.

Note that SciDB will create a shadow array automatically for you if you use the `SHADOW ARRAY` keywords in your LOAD statement.

By using shadow arrays, you can achieve a successful load, even if the load file contains some imperfections. Of course, if a file is grossly deformed or incompatible with the load operation, you probably want SciDB to abandon the load operation. In such a case, you can include with the load statement a maximum

number of errors, after which SciDB should abandon the load operation. To specify the maximum number of errors, use the ERRORS keyword.

For example, consider the following CREATE ARRAY statement that establishes a 1-dimensional array to serve as the load array:

```
AQL% CREATE ARRAY  intensityFlat
 < exposure:string, elapsedTime:int64, measuredIntensity:int64 null >
 [i=0:6,1000000,0];
```

Assume that you want to load into this array the data from the following CSV file, which contains some errors:

```
exposure,elapsedTime,measuredIntensity
High,777,100
High,Jack,99
Medium,777,100
Medium,888,95
Medium,Jess,Jill
Low,?,Josh
Low,1888,?
```

As you compare the CSV file with the CREATE ARRAY statement, notice the following:

- The second row contains an error—a text value in a numeric field.
- The fifth row contains two errors—text values in numeric fields.
- The sixth row contains two errors—a null value in the second field (whose corresponding attribute in the CREATE ARRAY statement prohibits nulls) and a text value in the third field.
- The seventh row contains a legitimate null value in the third field.
- All other rows are unremarkable.

The corresponding SciDB-formatted text file—that is, the file that results when you run csv2scidb on this CSV file, is shown below:

```
$ cat '../examples/int4error.scidb'
```

```
{0}[
("High",777,100),
("High",Jack,99),
("Medium",777,100),
("Medium",888,95),
("Medium",Jess,Jill),
("Low",?,Josh),
("Low",1888,?)
]
```

To load this file into SciDB using a shadow array to keep track of load errors, use this AQL statement:

```
AQL% LOAD intensityFlat
      FROM '../examples/int4error.scidb'
      AS   'text'
      ERRORS 99
      SHADOW ARRAY intensityFlatShadow;
```

```
{i} exposure,elapsedTime,measuredIntensity
{0} "High",777,100
{1} "High",0,99
{2} "Medium",777,100
{3} "Medium",888,95
{4} "Medium",0,null
{5} "Low",0,null
{6} "Low",1888,null
```

In the LOAD statement, notice the following:

- The LOAD statement establishes a limit of 99 errors for the load. If the load operation encounters more than 99 errors, SciDB will abandon it.
- The LOAD statement uses a shadow array named intensityFlatShadow to record load errors. If the shadow array you name in the LOAD statement does not already exist, SciDB will create it for you. If the shadow array you name already exists, you must ensure that the shadow array's schema is properly compatible with the schema of the load array: string attributes with names identical to the attributes of the target array (string or otherwise) plus an int64 attribute named row_offset.

One result of this LOAD statement is the array intensityFlatShadow. To examine its schema definition, use the show operator:

```
AFL% show(intensityFlatShadow)
```

```
intensityFlatShadow

< exposure:string NULL DEFAULT null,
elapsedTime:string NULL DEFAULT null,
measuredIntensity:string NULL DEFAULT null,
row_offset:int64 >

[i=0:6,1000000,0]
```

Notice that the shadow array includes one string attribute for every attribute (string or otherwise) in the target array. Notice also the integer row_offset attribute. Furthermore, notice that the dimension declaration—in this case, just the single dimension named i—matches the dimension declaration in the load array in all regards: bounds, chunk size, and chunk overlap.

Another result of the preceding LOAD statement is the set of populated values of the shadow array. To examine these values, use this AQL statement:

```
AQL% SELECT * FROM intensityFlatShadow;
```

```
{i} exposure,elapsedTime,measuredIntensity,row_offset
{1} null,"Failed to parse string",null,35
{4} null,"Failed to parse string","Failed to parse string",94
{5} null,"Assigning NULL to non-nullable attribute","Failed to parse
 string",110
```

The data in the intensityFlatShadow array includes three non-empty cells, indicating the following:

- One row (the second) produced an error in the second field. The error occurred approximately 35 bytes from the start of the file.

- One row (the fifth) produced two errors, in the second and third fields. The first of these errors occurred approximately 94 bytes from the start of the file.
- One row (the sixth) produced two errors, in the second and third fields. The first of these errors occurred approximately 110 bytes from the start of the file.
- All other rows were loaded successfully.

And of course, the other result of the LOAD command is data loaded into the load array. To examine that data, use this AQL statement:

```
AQL% SELECT * FROM intensityFlat;
```

```
{i} exposure,elapsedTime,measuredIntensity
{0} "High",777,100
{1} "High",0,99
{2} "Medium",777,100
{3} "Medium",888,95
{4} "Medium",0,null
{5} "Low",0,null
{6} "Low",1888,null
```

The data in the intensityFlat array indicates the following:

- The second row has two correct values (High and 99) in the first and third attributes. The second attribute, whose incoming value generated an error, has been populated with the default value (0) for that field. SciDB inserted the default value because that attribute does not allow nulls.
- The fifth row has one correct value (Medium) in the first attribute. The second attribute has been populated with the default value for that attribute. By contrast, the third attribute, which also generated an error, has been set to null because that attribute allows null values.
- The sixth row has one correct value (Low) in the first attribute. The second attribute has been populated with the default value for that attribute because that attribute does not allow nulls. By contrast, the third attribute, which also generated an error, has been set to null because that attribute allows null values.
- All other rows were loaded successfully. This includes the last row, whose null value in the third attribute was represented in the original CSV file.

After a load operation that produced some errors, you can create an array that combines the error messages in the shadow array with the problematic cells in the load array. For example, the following AQL statement accomplishes this with intensityFlat and intensityFlatShadow:

```
AQL% SELECT
       intensityFlat.exposure
                       AS exp,
       intensityFlatShadow.exposure
                       AS expMSG,
       intensityFlat.elapsedTime
                       AS elTime,
       intensityFlatShadow.elapsedTime
                       AS elTimeMSG,
       intensityFlat.measuredIntensity
                       AS Intensity,
       intensityFlatShadow.measuredIntensity
                       AS IntensityMSG,
```

```
       row_offset
    INTO
       intensityFlatBadCells
    FROM
       intensityFlat,
       intensityFlatShadow;
```

You can examine the result of this AQL statement as follows:

```
AQL% SELECT * FROM intensityFlatBadCells;
```

```
{i} exp,expMSG,elTime,elTimeMSG,Intensity,IntensityMSG,row_offset
{1} "High",null,0,"Failed to parse string",99,null,35
{4} "Medium",null,0,"Failed to parse string",null,"Failed to parse
 string",94
{5} "Low",null,0,"Assigning NULL to non-nullable
 attribute",null,"Failed to parse string",110
```

The query result shows the usefulness of the array intensityFlatBadCells. The array contains one non-empty cell for each problematic cell of the load operation. Within the array intensityFlatBadCells, the attributes are arranged in consecutive pairs, where each pair consists of a value from the load array, and an indication of whether that value was successfully loaded. For example, the third non-empty cell of intensityFlatBadCells indicates the following:

- The value in the first attribute of the cell in the load array ("Medium") was successfully loaded because the error message corresponding to that attribute in the shadow array is null.
- The value in the second attribute of the cell in the load array was not successfully loaded because the error message is not null. The value (0) is the applicable default value for that attribute.
- The value in the third attribute of the cell in the load array was not successfully loaded because the error message is not null. The value itself is null because that attribute of the target array allows nulls.
- SciDB estimates that the problems loading values for this cell begin at or near byte number 110 of the load file.

As you can see, an array like intensityFlatBadCells constitutes a useful report on the results of a load operation. Whenever you perform a load operation using a shadow array, you can combine the shadow array with the target array to make an array like intensityFlatBadCells. Thereafter, you can use that array to help you investigate the problems that occurred during the load. How you choose to remedy or otherwise respond to those problems depends on the nature of your data and the data-quality policies of your organization.

Chapter 6. Basic Array Tasks

6.1. Selecting Data From an Array

The AQL Data Manipulation Language (DML) provides queries to access and operate on array data. The basis for selecting data from a SciDB array is the AQL **SELECT** statement with **INTO**, **FROM**, and **WHERE** clauses.

6.1.1. SELECT Statement Syntax

The syntax of the **SELECT** statement is:

```
SELECT expression
       [INTO target_array]
       FROM array_expression | source_array
       [WHERE expression]
```

The arguments for the statement are:

expression	**SELECT** *expression* can select individual attributes and dimensions, as well as constants and expressions. The wildcard character * means select all attributes.
target_array	The **INTO** clause can create an array to store the output of the query. The target array may also be a pre-existing array in the current SciDB namespace.
array_expression | *source_array*	The **FROM** clause takes a SciDB array or array expression as argument. The *array_expression* argument is an expression or subquery that returns an array result. The *source_array* is an array that has been created and stored in the current SciDB namespace.
expression	The *expression* argument of the **WHERE** clause allows to you specify parameters that filter the query.

6.1.2. The SELECT Statement

AQL expressions in the **SELECT** list or the **WHERE** clause are standard expressions over the attributes and dimensions of the array. The simplest **SELECT** statement is **SELECT** *, which selects all data from a specified array or array expression. Consider two arrays, A and B:

```
AQL% CREATE ARRAY A <val_a:double>[i=0:9,10,0];
```

```
AQL% CREATE ARRAY B <val_b:double>[j=0:9,10,0];
```

These arrays contain data. To see all the data stored in the array, you can use the following **SELECT** statement:

```
AQL% SELECT * FROM A;
```

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

```
AQL% SELECT * FROM B;
```

```
[(101),(102),(103),(104),(105),(106),(107),(108),(109),(110)]
```

The `show` command returns an array result containing an array's schema. To see the entire schema, use a **`SELECT`** * statement with the `show` command:

```
AQL% SELECT * FROM show(A);

[("A<val_a:double> [i=0:9,10,0]")]

AQL% SELECT * FROM show(B);

[("B<val_b:double> [j=0:9,10,0]")]
```

To refine the result of the **`SELECT`** statement, use an argument that specifies part of an array result. **`SELECT`** can take array dimensions or attributes as arguments:

```
AQL% SELECT j FROM B;

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

AQL% SELECT val_b FROM B;

[(101),(102),(103),(104),(105),(106),(107),(108),(109),(110)]
```

The **`SELECT`** statement can also take an expression as an argument. For example, you can scale attribute values by a certain amount:

The **`WHERE`** clause can also use built-in functions to create expressions. For example, you can choose just the middle three cells of array B with the greater-than and less-than functions with the `and` operator:

```
AQL% SELECT j FROM B WHERE j > 3 AND j < 7;

[(),(),(),(),(4),(5),(6),(),(),()]
```

You can also select an expression of the attribute values for the middle three cells of B by providing an expression for the argument of both **`SELECT`** and **`WHERE`**. For example, this statement returns the square root of the middle three cells of array `B`:

```
AQL% SELECT sqrt(val_b) FROM B WHERE j>3 AND j<7;

[(),(),(),(),(10.247),(10.2956),(10.3441),(),(),()]
```

The **`FROM`** clause can take an array or any operation that outputs an array as an argument. The **`INTO`** clause stores the output of a query.

6.2. Array Joins

A join combines two or more arrays (typically as a preprocessing step for subsequent operations). The simplest type of join is an *inner join*. An inner join performs an attribute-attribute join on every cell in two source arrays. An inner join can be performed for two arrays with the same number of dimensions, same dimension starting coordinates, and same chunk size.

The syntax of a inner join statement is:

```
SELECT expression FROM src_array1, src_array2
```

The inner join of arrays A and B joins the attributes:

```
AQL% SELECT * FROM A,B;
```

```
[(1,101),(2,102),(3,103),(4,104),(5,105),(6,106),(7,107),(8,108),
(9,109),(10,110)]
```

This query will store the attribute-attribute join of A and B in array C:

```
AQL% SELECT * INTO C FROM A, B;
```

```
[(1,101),(2,102),(3,103),(4,104),(5,105),(6,106),(7,107),(8,108),
(9,109),(10,110)]
```

The target array C has schema:

```
AFL% show(C)
```

```
C

< val_a:double,
val_b:double >

[i=0:9,10,0]
```

The attributes maintain the names from A and B; the dimension takes the name from the first array of the join operation.

Arrays do not need to have the same number of attributes to be compatible. As long as the dimension starting indices, chunk sizes, and chunk overlaps are the same, the arrays can be joined. For example, you can join the two-attribute array C with the one-attribute array B:

```
AQL% SELECT * INTO D FROM C,B;
```

```
[(1,101,101),(2,102,102),(3,103,103),(4,104,104),(5,105,105),
(6,106,106),(7,107,107),(8,108,108),(9,109,109),(10,110,110)]
```

This produces array D with the following schema, as returned by the show operator:

```
AQL% SELECT * FROM show(D);
```

```
[("D<val_a:double,val_b:double,val_b_2:double> [i=0:9,10,0]")]
```

Since C and B shared an attribute name, `val_b`, the array D contains a renamed attribute, `val_b_2`.

If two arrays have an attribute with the same name, you can select the attributes to use with array dot notation:

```
AQL% SELECT  C.val_b + D.val_b FROM C,D;
```

```
[(202),(204),(206),(208),(210),(212),(214),(216),(218),(220)]
```

If you are using attributes that have the same fully qualified name, for example, you are joining an array with itself, you can use an alias to rename an array for the particular query. This query aliases array A to a1 and a2, then uses dot notation to use the attribute `val_a` in two different expressions.

```
AQL% SELECT a1.val_a,a2.val_a+2 FROM A AS a1,A AS a2;
```

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

The aliasing rules apply to other join operators as well, including merge, cross, and cross_join. See the section "Aliases" below for more information on aliases.

The **JOIN ... ON** predicate calculates the multidimensional join of two arrays after applying the constraints specified in the **ON** clause. The **ON** clause lists one or more constraints in the form of equality predicates on dimensions or attributes. The syntax is:

```
SELECT expression [INTO target_array]
     FROM array_expression | source_array
    [ JOIN expression | attribute ]
     ON dimension | attribute
```

A dimension-dimension equality predicate matches two compatible dimensions, one from each input. The result of this join is an array with higher number of dimensions—combining the dimensions of both its inputs, less the matched dimensions. If no predicate is specified, the result is the full cross product array.

An attribute predicate in the **ON** clause is used to filter the output of the multidimensional array.

For example, consider a 2-dimensional array `m3x3`schema and attributes values:

```
AQL% CREATE ARRAY m3x3<a:double> [i=1:3,3,0,j=1:3,3,0];
```

```
Query was executed successfully
```

Now consider also a 1-dimensional array `vector3` whose schema and attributes are:

```
AFL% show(vector3)
```

```
vector3

< b:double >

[k=1:3,3,0]
```

```
AFL% scan(vector3);
```

```
[(21),(20.5),(20.3333)]
```

A dimension join returns a 2-dimensional array with coordinates `{i,j}` in which the cell at coordinate `{i,j}` combines the cell at `{i,j}` of `m3x3` with the cell at coordinate `{k=j}` of `vector3`:

```
AQL% SELECT * FROM m3x3 JOIN vector3 ON m3x3.j = vector3.k;
```

```
[[(0,21),(1,20.5),(2,20.3333)],[(3,21),(4,20.5),(5,20.3333)],[(6,21),
(7,20.5),(8,20.3333)]]
```

6.3. Aliases

AQL provides a way to refer to arrays and array attributes in a 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 or attribute name, followed by the alias. Future references to the array can then use the alias. Once an alias has been assigned, all attributes and dimensions of the array can use the fully qualified name using the dotted naming convention.

```
AQL%  SELECT data.i*10 FROM A AS data WHERE A.i < 5;
```

```
[(0),(10),(20),(30),(40),(),(),(),(),()]
```

6.4. Nested Subqueries

You can nest AQL queries to refine query results.

For example, you can nest **SELECT** statements before a **WHERE** clause to select a subset of the query output.

```
AQL% SELECT pow(c,2) FROM
(SELECT A.val_a + B.val_b AS c FROM A,B) WHERE i > 5;
```

```
[(),(),(),(),(),(),(12996),(13456),(13924),(14400)]
```

This query does the following:

1. Sums two attributes from two different arrays and stores the output in an alias,
2. Selects the cells with indices greater than 5, and
3. Squares the result.

6.5. Data Sampling

SciDB provides operations to sample array data. The `bernoulli` operator allows you to select a subset of array cells based upon a given probability. For example, you can use the `bernoulli` operator to randomly sample data from an array one element at a time. The syntax of `bernoulli` is:

```
bernoulli(array, probability:double  [, seed:int64])
```

The `sample` command allows you to randomly sample data one array chunk at a time:

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

The probability is a double between 0 and 1. The commands work by generating a random number for each cell or chunk in the array and scaling it to the probability. If the random number is within the probability, the cell/chunk is included. Both commands allow you to produce repeatable results by seeding the random number generator. All calls to the random number generator with the same seed produce the same random number. The seed must be a 64-bit integer.

Chapter 7. Performing Simple Analytics

This chapter provides an overview of the basic analytic capabilities of SciDB. SciDB provides commands to group data from an array and calculate summaries over those groups. These commands are called *aggregates*.

In addition, SciDB also provides scalable operators to calculate *order statistics* of array data -- these include *rank*, *avg_rank*, and *quantile*, as well as the operator *sort* which rearranges array data and returns a vector of sorted items.

7.1. Aggregates

SciDB offers the following aggregate methods that calculate summaries over groups of values in an array.

Aggregate	Definition
`approxdc`	Approximate count of distinct values
`avg`	Average value
`count`	Number of nonempty elements (array count) and non-null elements (attribute count).
`max`	Largest value
`min`	Smallest value
`sum`	Sum of all elements
`stdev`	Standard deviation
`var`	Variance

These aggregates appear within the context of one of the following SciDB operators or query types. We classify these aggregating operators based on how they divide data within the input array into subgroups.

- A *Grand aggregate* computes an aggregate over an entire array or an arbitrary subset of an array specified via filtering or other data preparation.
- A *Group-by aggregate* computes summaries by grouping array data by dimension value.
- A *Grid aggregate* computes summaries for non-overlapping grids of the input array. Hence each group or grid is a multidimensional subarray of the input array.
- A *Window aggregate* computes summaries over a moving window in an array. SciDB supports two types of window operators: fixed boundary windows and variable boundary windows. Variable boundary windows are identified by the VARIABLE WINDOW clause in AQL and their size depends on the number of nonempty elements. These window aggregates are described in depth later in this chapter.

We describe these different types of array aggregates in more detail in the following sections, as well as in Chapter 11, *SciDB Aggregate Reference*.

Most examples in this chapter use the following example arrays: `m4x4` and `m4x4_2attr`, which have the following schemas and contain the following values:

`m4x4`

```
< attr1:double >
```

```
[x=0:3,4,0,
y=0:3,4,0]
```

```
AFL% scan(m4x4);
```

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

```
m4x4_2attr

< attr1:double,
attr2:double >

[x=0:3,4,0,
y=0:3,4,0]
```

```
AFL% scan(m4x4_2attr);
```

```
[
[(0,0),(1,2),(2,4),(3,6)],
[(4,8),(5,10),(6,12),(7,14)],
[(8,16),(9,18),(10,20),(11,22)],
[(12,24),(13,26),(14,28),(15,30)]
]
```

7.1.1. Grand Aggregates

A grand aggregate in SciDB calculates aggregates or summaries of attributes across an entire array or across an arbitrary subset of an array you specify via filtering or other preparation with array operators.. You calculate grand aggregates with the AQL **SELECT** statement conforming to this syntax:

```
AQL% SELECT aggregate(attribute)[,aggregate(attribute)]...
[ INTO dst-array]
FROM src-array | array-expression
[WHERE where-expression]
```

The output is a SciDB array with one attribute named for each summary type in the query, whose dimensions are determined by the size and shape of the result.

For example, to select the maximum and the minimum values of the attribute `attr1` of the array `m4x4`:

```
AQL% SELECT max(attr1),min(attr1) FROM m4x4;
```

```
[(15,0)]
```

You can store the output of a query into a destination array, `m4x4_max_min` with the **INTO** clause:

```
AQL% SELECT max(attr1),min(attr1) INTO m4x4_max_min FROM m4x4;
```

```
[(15,0)]
```

The destination array `m4x4_max_min` has the following schema:

```
not empty m4x4_max_min

< max:double NULL DEFAULT null,
min_1:double NULL DEFAULT null >

[i=0:0,1,0]
```

To select the maximum value from the attribute attr1 of `m4x4_2attr` and the minimum value from the attribute `attr2` of `m4x4_2attr`:

```
AQL% SELECT max(attr2), min(attr2) FROM m4x4_2attr;

[(30,0)]
```

Note

In the special case of a one-attribute array, you may omit the attribute name. For example, to select the maximum value from the attribute `attr1` of the array `m4x4`, use the AQL **SELECT** statement:

```
AQL% SELECT max(m4x4);

{i} attr1_max
{0} 15
```

The AFL `aggregate` operator also computes grand aggregates. To select the maximum value from the attribute attr1 of `m4x4_2attr` and the minimum value from the attribute `attr2` of `m4x4_2attr`:

```
AFL% aggregate(m4x4_2attr, max(attr2),min(attr1));

[(30,0)]
```

In most cases, SciDB aggregates exclude null-valued data. For example, consider the following array `m4x4_null`:

```
[
[(null),(null),(null),(null)],
[(null),(null),(null),(null)],
[(0),(0),(0),(0)],
[(null),(null),(null),(null)]
]
```

The commands `count(attr1)` and `count(*)` return different results because the first ignores null values, while the second does not:

```
AQL% SELECT count(attr1) AS a, count(*) AS b FROM m4x4_null;

{i} a,b
{0} 4,16
```

7.1.2. Group-By Aggregates

Group-by aggregates group array data by array dimensions and summarize the data in those groups.

AQL **GROUP BY** aggregates take a list of dimensions as the grouping criteria and compute the aggregate function for each group. The result is an array containing only the dimensions specified in the **GROUP BY** clause and a single attribute per specified aggregate call. The syntax of the **SELECT** statement for a group-by aggregate is:

```
SELECT expression1 [,expression2]...
  [ INTO dst-array ]
  FROM src-array | array-expression
  [ WHERE where-expression ]
  GROUP BY dimension1 [,dimension2]... ;
```

AQL expressions in the SELECT list are expressions containing attributes or dimensions of the array (also referred to as variables of the array), scalar functions and aggregates. For example, this query selects the largest value of `attr1` from each row of `m4x4`:

```
AQL% SELECT max(attr1) FROM m4x4 GROUP BY x;
```

```
{x} max
{0} 3
{1} 7
{2} 11
{3} 15
```

The output has the following schema:

```
< max:double NULL DEFAULT null >

[x=0:3,4,0]
```

Note

> You will notice that the new attributes generated by applying the aggregates have special suffixes, for example, `min_1` and `max_1` . This is done when calculating aggregates to keep attribute names unique especially during intermediate stages of array processing.

This query selects the maximum value of `attr1` from each column of array `m4x4`

```
AQL% SELECT max(attr1) FROM m4x4 GROUP BY y;
```

```
{y} max
{0} 12
{1} 13
{2} 14
{3} 15
```

The AFL `aggregate` operator takes dimension arguments to support group-by functionality. This query selects the largest value from each column `y` from the array `m4x4` using AFL:

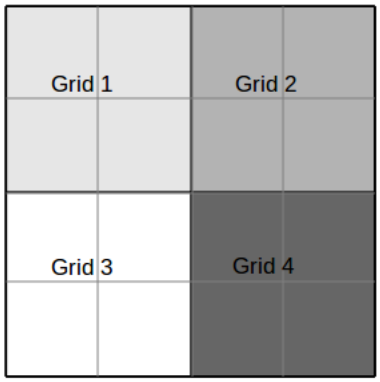
```
AFL% aggregate(m4x4, max(attr1), y);
```

```
{y} attr1_max
{0} 12
{1} 13
```

```
{2} 14
{3} 15
```

7.1.3. Grid Aggregates

A grid aggregate selects non-overlapping subarrays from an existing array and calculates an aggregate of each subarray. For example, if you have a 4x4 array, you can create 4 non-overlapping 2x2 regions and calculate an aggregate for those regions. The array `m4x4` would be divided into 2x2 grids as follows:



The syntax of a grid aggregate statement is:

```
AQL% SELECT aggregate(attribute) [,aggregate(attribute)] ...
 INTO dst-array
 FROM src-array | array-expression
 WHERE where-expression
 REGRID AS
        ( PARTITION BY  dimension1 dimension1-size
          [, dimension2 dimension2-size]... ) ;
```

For example, this statement finds the maximum and minimum values for each of the four grids in the previous figure:

```
AQL%
SELECT max(attr1), min(attr1)
FROM m4x4
REGRID AS (PARTITION BY x 2, y 2);
```

```
[
[(5,0),(7,2)],
[(13,8),(15,10)]
]
```

This output has schema:

```

< max:double NULL DEFAULT null,
min_1:double NULL DEFAULT null >

[x=0:1,4,0,
y=0:1,4,0]
```

In AFL, you can use the `regrid` operator to get the same result:

```
[
[(5,0),(7,2)],
[(13,8),(15,10)]
]
```

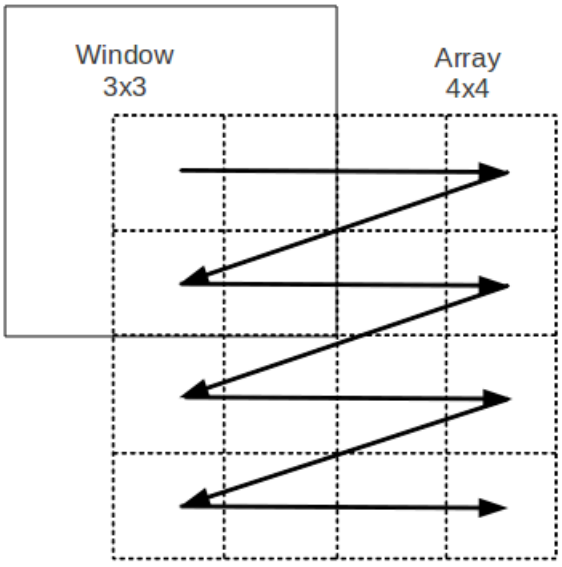
7.1.4. Window Aggregates

Window aggregates allow you to specify groups with a moving window. The window is defined by a size in each dimension. The window centroid starts at the first array element. The moving window starts at the first element of the array and moves in stride-major order from the lowest to highest value in each dimension. The AQL syntax for window aggregates is:

```
AQL% SELECT aggregate (attribute)[, aggregate (attribute)]...
    INTO dst-array
    FROM src-array | array-expression
    WHERE where-expression
    FIXED | VARIABLE WINDOW AS
    (PARTITION BY dimension1 dim1-low PRECEDING AND dim1-
high FOLLOWING
                [, dimension2 dim2-low PRECEDING  AND dim2-
high FOLLOWING ]... );
```

SciDB supports two types of window aggregates, identified by the keywords FIXED WINDOW and VARIABLE WINDOW as shown in the synopsis above. Both types of window aggregates calculate an aggregate over a window surrounding each array element. A fixed boundary window aggregate uses an exact size for each of its dimensions. Each dimension specifies both the number of preceding values and the number of following values relative to the center. Window dimension sizes include empty cells. SciDB supports multi-dimensional windows, hence, to calculate a fixed window query on a 3-dimensional array, one must define a window with 3 dimensions.

For example, you can use fixed window to calculate a running sum for a 3x3 window on array `m4x4`.



In AQL, you would use this statement:

```
AQL%
    SELECT sum(attr1)
    FROM m4x4
    FIXED WINDOW AS
    (PARTITION BY x 1 PRECEDING AND 1 FOLLOWING,
    y 1 PRECEDING AND 1 FOLLOWING);
```

```
[
[(10),(18),(24),(18)],
[(27),(45),(54),(39)],
[(51),(81),(90),(63)],
[(42),(66),(72),(50)]
]
```

The output has the following schema:

```
< sum:double NULL DEFAULT null >

[x=0:3,4,0,
y=0:3,4,0]
```

In AFL, you can use the `window` operator to achieve the same result:

```
AFL% window (m4x4,1,1,1,1,sum(attr1));
```

```
[
[(10),(18),(24),(18)],
[(27),(45),(54),(39)],
[(51),(81),(90),(63)],
[(42),(66),(72),(50)]
]
```

In contrast, the boundary of a variable window can vary since the window size includes only nonempty values. Both the number of preceding (nonempty) values and following (nonempty) values relative to the center must appear in the query. SciDB supports only a one-dimensional variable window operator, and this dimension appears in the query. This special dimension defines the "axis" of this type of window along which the window boundary is calculated and along which the window center moves during the query.

One can think of the (one dimensional) variable window aggregate to be a special case where all the unspecified dimensions have unit length. The following examples show how to use variable windows.

Consider the array `m4x4_empty`:

```
AFL% CREATE ARRAY m4x4_empty<val: double, label: string
 NULL>[i=0:3,4,0, j=0:3,4,0];
```

```
AFL% scan(m4x4_empty);
```

```
[
[(0,null),(),(),()],
[(4,null),(),(6,null),(7,null)],
[(8,null),(),(),()],
[(),(13,null),(14,null),()]
```

```
]
```

The following variable window aggregate query along dimension `i` is shown here. This query uses a window with one value preceding and one value following the window center after excluding empty cells.

```
AQL%
SELECT sum(val)
FROM m4x4_empty
VARIABLE WINDOW AS
(PARTITION BY
i 1 PRECEDING AND 1 FOLLOWING);
```

```
{i,j} sum
{0,0} 4
{1,0} 12
{1,2} 20
{1,3} 7
{2,0} 12
{3,1} 13
{3,2} 20
```

In AFL, you can specify the same query as follows:

```
AFL% variable_window(m4x4_empty, i, 1, 1, sum(val));
```

```
[
[(4),(),(),()],
[(12),(),(20),(7)],
[(12),(),(),()],
[(),(13),(20),()]
]
```

7.1.5. Aggregation During Redimension

The preceding sections of this chapter describe those features of SciDB that were designed exclusively to calculate aggregates. You can also calculate aggregates "in-line" as part of other data management or rearrangement steps: the redimension and redimension_store operators support this type of usage. Those operators are used to transform a source array into a result array with difference schema or dimensions. In those cases, aggregates can be useful in summarizing multiple elements from the source array that are mapped to a single element in the destination array.

For example, suppose you have a 2-dimensional array describing some recent olympic champions, and you want to produce a 1-dimensional array that shows the gold-medal count for each country.

The schema of the original 2-dimensional array (named "winners") appears below:

```
AFL% show(winners)
```

```
winners

< person:string,
country:string,
time:double >

[year=1996:2008,13,0,
event(string)=3,3,0]
```

To examine the data in the "winners" array, use the following AQL statement with the -olcsv+ format:

```
AQL% SELECT * FROM winners;
```

```
{year,event} person,country,time
{1996,"dash"} "Bailey","Canada",9.84
{1996,"marathon"} "Thugwane","USA",7956
{1996,"steeplechase"} "Keter","Kenya",487.12
{2000,"dash"} "Greene","USA",9.87
{2000,"marathon"} "Abera","Ethiopia",7811
{2000,"steeplechase"} "Kosgei","Kenya",503.17
{2004,"dash"} "Gatlin","USA",9.85
{2004,"marathon"} "Baldini","Italy",7855
{2004,"steeplechase"} "Kemboi","Kenya",485.81
{2008,"dash"} "Bolt","Jamaica",9.69
{2008,"marathon"} "Wanjiru","Kenya",7596
{2008,"steeplechase"} "Kipruto","Kenya",490.34
```

To create the schema for the desired array, using the following AQL statement:

```
AQL% CREATE ARRAY perCountryMedalCount <medalCount: uint64 null>
 [country(string)=20,20,0];
```

Notice that the sole dimension is country, which is an attribute (not a dimension) of the original "winners" array. To populate the desired array with data, use the following AFL statement:

```
AFL% redimension_store (winners, perCountryMedalCount, count(*) as
 medalCount)
```

```
{country} medalCount
{"Canada"} 1
{"Ethiopia"} 1
{"Italy"} 1
{"Jamaica"} 1
{"Kenya"} 5
{"USA"} 3
```

The result of this redimension_store operation is the desired array. You can examine the contents with the following AQL statement:

```
AQL% SELECT * FROM perCountryMedalCount;
```

```
{country} medalCount
{"Canada"} 1
{"Ethiopia"} 1
{"Italy"} 1
{"Jamaica"} 1
{"Kenya"} 5
{"USA"} 3
```

Notice that the sum of the counts is 12—the number of nonempty cells in the "winners" array.

7.2. Order Statistics

SciDB offers the following ordering and ranking capabilities.

- Sort, which can be used to return a sorted vector of all array elements.
- Rank and avg_rank. These methods can be used to assign an ordinal rank to each element of the array. These can be calculated for the entire array, or on a per dimension basis.
- Quantiles. These can be calculated for the entire array, or on a per dimension basis.

7.2.1. Sort

The sort operator takes as input a multi-dimensional array and produces a one-dimensional vector of elements sorted by the specified attribute.

If you specify multiple attributes, all attributes are used for sorting. Array elements are sorted first by the first attribute, and then for each value of the the first attribute, they are sorted by the second attribute, and so on for each specified attribute.

If multiple elements have the same attribute value—for an attribute that is not specified in the sort operator—the sort operator selects an arbitrary ordering of elements when producing the sorted output.

The result array contains only non-empty cells from the source array. Consider the array `m4x4_empty`, which is a 2-dimensional array with some empty cells. Each element contains two attributes. To sort the array elements by `val`, use the following query:

```
AQL%
SELECT *
FROM sort(m4x4_empty, val);
```

```
{n} val,label
{0} 0,null
{1} 4,null
{2} 6,null
{3} 7,null
{4} 8,null
{5} 13,null
{6} 14,null
```

To sort the array by val, and then by label, use the following query.

```
AQL%
SELECT *
FROM sort(m4x4_empty, label);
```

```
{n} val,label
{0} 0,null
{1} 4,null
{2} 6,null
{3} 7,null
{4} 8,null
{5} 13,null
{6} 14,null
```

To use a descending sort, use the following command:

```
AQL%
SELECT *
FROM sort(m4x4_empty, label desc);
```

```
{n} val,label
{0} 0,null
{1} 4,null
{2} 6,null
{3} 7,null
{4} 8,null
{5} 13,null
{6} 14,null
```

7.2.2. Ranking Methods

The `rank` and `avg_rank` operators rank the elements of an array or within subgroups of an array.

Consider the following examples using the `m4x4_double` array.

You can rank the elements of `m4x4_double` by dimension with the rank operator. For example, this query returns an array where the second attribute of each cell is the rank of the element for dimension j (columns):

```
AFL% CREATE ARRAY m4x4_double < val:double >[i=0:3,4,0,j=0:3,4,0];
```

```
m4x4_double

< val:double >

[i=0:3,4,0,
j=0:3,4,0]
```

```
AFL% scan(m4x4_double);
```

```
[
[(0),(10.0977),(10.9116),(1.69344)],
[(9.08163),(11.5071),(3.35299),(7.88384)],
[(11.8723),(4.94542),(6.52825),(11.9999)],
[(6.43888),(5.042),(11.8873),(7.80345)]
]
```

```
AQL% SELECT * FROM rank(m4x4_double,val,j);
```

```
[
[(0,1),(10.0977,3),(10.9116,3),(1.69344,1)],
[(9.08163,3),(11.5071,4),(3.35299,1),(7.88384,3)],
[(11.8723,4),(4.94542,1),(6.52825,2),(11.9999,4)],
[(6.43888,2),(5.042,2),(11.8873,4),(7.80345,2)]
]
```

The operators rank and avg_rank offer different ways to handle ties. For example, consider the array `m4x4_floor`:

```
AQL%
SELECT floor(val)
INTO m4x4_floor
FROM m4x4_double;
```

```
[[(0),(10),(10),(1)],[(9),(11),(3),(7)],[(11),(4),(6),(11)],[(6),(5),
(11),(7)]]
```

Ranking by dimension j produces ties. There are two cells with value 7 in the last column. The `rank` operator assigns the tied values the same rank. The `avg_rank` operator assigns the average of the rank positions occupied by the tied values.

```
AQL%
SELECT *
FROM rank(m4x4_floor,expr,j);
```

```
[
[(0,1),(10,3),(10,3),(1,1)],
[(9,3),(11,4),(3,1),(7,2)],
[(11,4),(4,1),(6,2),(11,4)],
[(6,2),(5,2),(11,4),(7,2)]
]
```

```
AQL%
SELECT *
FROM avg_rank(m4x4_floor,expr,j);
```

```
[
[(0,1),(10,3),(10,3),(1,1)],
[(9,3),(11,4),(3,1),(7,2.5)],
[(11,4),(4,1),(6,2),(11,4)],
[(6,2),(5,2),(11,4),(7,2.5)]
]
```

7.2.3. Calculating Quantiles

The `quantile` operator calculates quantiles over array attributes. A *q*-quantile is a point taken at a specified interval on a sorted data set that divides the data set into q subsets. The 2-quantile is the *median*, that is, the numerical value separating the lower half and upper half of the data set. For example, consider the data set represented by the array `m4x4_floor`:

```
AFL% show(m4x4_floor)
```

```
m4x4_floor

< expr:int64 >

[i=0:3,4,0,
j=0:3,4,0]
```

```
AFL% scan(m4x4_floor);
```

```
[
[(0),(10),(10),(1)],
[(9),(11),(3),(7)],
[(11),(4),(6),(11)],
[(6),(5),(11),(7)]
]
```

The lowest value in `m4x4_floor` is 0, the median value is 7, and the highest value is 11.

```
AQL%
SELECT *
```

```
FROM quantile(m4x4_floor,2);
```

```
{quantile} percentage,expr_quantile
{0} 0,0
{1} 0.5,7
{2} 1,11
```

The result of the 8-quantile for `m4x4_floor` is shown below.

```
AQL%
SELECT *
FROM quantile(m4x4_floor,8);
```

```
{quantile} percentage,expr_quantile
{0} 0,0
{1} 0.125,1
{2} 0.25,4
{3} 0.375,6
{4} 0.5,7
{5} 0.625,9
{6} 0.75,10
{7} 0.875,11
{8} 1,11
```

Chapter 8. Updating Arrays

When you use AQL or AFL to manipulate the contents of SciDB, the operators and statements you use can have several effects:

- Most statements produce a result array without changing the original array. For example, the AFL `filter()` operator produces a result array based on data from an array you supply, but it does not change the supplied array in any way.
- Some statements change the metadata of an array in place, without changing the data and without producing a result array. For example, the the AFL `rename()` operator changes the name of an array, but does not produce a result array (which means that you cannot use the `rename()` operator as an operand in any other AFL operator.)
- Some statements change the data of an array in place, and simultaneously produce a result array that you can use as an operand of another AQL operator. For example, the AFL `insert()` operator modifies the contents—i,e., the data rather than the schema definition—of an array, and also produces a result array that reflects the contents of the stored array after the insertion operation.

This chapter describes some AQL statements that fall into the third category—that is, AQL statements that perform write-in-place updates to stored array data.

When you modify the contents of an array, SciDB uses a "no overwrite" storage model. No overwrite means that data in an array can be updated but previous values can be accessed for as long as the array exists in the SciDB namespace. Every time you update data in a stored array, SciDB creates a new array version, much like source control systems for software development.

This chapter describes the following AQL statements that perform in-place updates:

- The AQL UPDATE ... SET statement lets you update the values of attributes within cells that already exist in an array. The new values come from an expression you supply. The UPDATE ... SET statement is designed for "point" updates or selective updates; it is especially useful after a large data set has been imported and some values contain errors that you want to correct.
- The AQL INSERT INTO statement lets you update attribute values and insert new cells into an existing array. The new values come from another array with a compatible schema. The INSERT INTO statement is designed for bulk or batch updates of new data to be appended to existing data, such as including daily incremental feeds of financial data. The INSERT INTO statement has both add and update semantics. That is, if a cell already exists, you can use INSERT to update its values, and if a cell does not yet exist, INSERT INTO will create a new cell and populate it with attribute values.

8.1. The INSERT INTO statement

The AQL INSERT statement can modify an array's contents by changing values in existing cells, inserting values in empty cells, or both.

```
AQL% INSERT
         INTO named_arrray
         select_statement | array_literal ;
```

The most straightforward AQL INSERT statement simply inserts the contents of one array into another. The following statement inserts the contents of A into B:

```
AQL% insert
```

```
          INTO B
          SELECT * FROM A
```

Although the syntax is straightforward, the operation of this statement deserves elaboration. First, array A and B must have compatible schemas. For the INSERT operations, compatibility includes the same number of dimensions and attributes, same data-types and null/not-null setting on each corresponding pair of attributes, and restrictions on dimension starting indexes, chunk sizes, and chunk overlaps. In addition, the current release of SciDB requires that every dimension of either array must have datatype int64. The complete list of compatibility rules for insertion operations appears in the chapter called SciDB AFL Operator Reference, in the section on the insert operator.

Here is the schema for array A:

```
AFL% show(A)
```

```
A

< value:string NULL DEFAULT null >

[row=1:3,3,0,
col=1:3,3,0]
```

And here is the schema for array B. Note that A and B are insert-compatible:

```
AFL% show(B)
```

```
B

< value:string NULL DEFAULT null >

[row=1:3,3,0,
col=1:3,3,0]
```

Provided the two arrays are schema compatible, the insert operator writes values into individual cells of the target array according to the following rules:

- If the corresponding cell location of the source array is empty, the insert operator does not change anything in the target array. At that cell location of the target array, an empty cell would remain empty, null values would remain null, and other values would remain unchanged.
- If the corresponding cell location of the source array is non-empty, the insert operator changes the corresponding cell of the target array to match the value of the source. Note that this means that null values in the source can overwrite non-null values in the target. Note that it also means that if the cell location of the target array was initially empty, it will be non-empty after the insert operation.

Continuing with the preceding example, here are the contents of A and B before the insert operation:

```
AQL% SELECT * FROM A
```

```
[
[(),(),()],
[(null),(null),(null)],
[("a7"),("a8"),("a9")]
]
```

```
AQL% SELECT * FROM B
```

```
[
[(),(null),("b3")],
[(),(null),("b6")],
[(),(null),("b9")]
]
```

And here is the result of the insert operation:

```
AQL% insert
          INTO B
          SELECT * FROM A
```

```
[
[(),(null),("b3")],
[(null),(null),(null)],
[("a7"),("a8"),("a9")]
]
```

Compare the original and modified versions of array B and note the following:

- Where A contained empty cells, the corresponding cells of B are unchanged. See row 1 of the output.
- Where A contained non-empty cells, the corresponding cells of B are changed. This includes replacing non-null values of B with null values from the corresponding cells of A. (See cell [2,3].)
- The count of non-empty cells in B has increased. (See the cells at [2,1] and [3,1].)

Although the source and target arrays must be compatible, you can still insert values into one array from a seemingly incompatible array with some judicious projecting. For example, consider array C, which has two attributes:

```
AFL% show(C)
```

```
C

< value:string NULL DEFAULT null,
value2:string NULL DEFAULT null >

[row=1:3,3,0,
col=1:3,3,0]
```

```
AQL% SELECT * FROM C
```

```
[
[("c1","c111"),(),("c3","c333")],
[(),("c5","c555"),()],
[("c7","c777"),(),("c9","c999")]
]
```

Although C is not insert-compatible with B (because B has fewer attributes), you can insert values from C into B by projecting to exclude one of C's attributes from the source of the insert statement, as follows:

```
AQL% insert
          INTO B
          SELECT value FROM C
```

```
[
[("c1"),(null),("c3")],
[(),("c5"),("b6")],
[("c7"),(null),("c9")]
]
```

Note that to be insert-compatible, two arrays must have the same number of attributes and dimensions, but the attributes and dimensions do not need to have the same names. The insert operator aligns dimensions from the respective arrays in left-to-right order, and aligns attributes from the respective arrays in the same way. The names of the attributes and dimensions are immaterial. For example, following statement inserts data from an attribute named value2 into array B, whose sole attribute is named value:

```
AQL% insert
          INTO B
          SELECT value2 FROM C
```

```
[
[("c111"),(null),("c333")],
[(),("c555"),("b6")],
[("c777"),(null),("c999")]
]
```

When supplying the array to be inserted in the source array, you are not limited to a select statement. Alternative syntax lets you use an array literal, as in the following command:

```
AQL% insert
          INTO B
          '[
           [()()(333333333)]
           [()(555555555)()]
           [(777777777)()()]
           ]'
```

```
[
[(),(null),("333333333")],
[(),("555555555"),("b6")],
[("777777777"),(null),("b9")]
]
```

8.2. The UPDATE ... SET statement

To update data in an existing SciDB array, use the statement:

```
AQL% UPDATE array SET "attr = expr", ... [ WHERE condition ];
```

Consider the following 2-dimensional array, `m4x4`:

```
AFL% store(build(<val:double>[i=0:3,4,0,j=0:3,4,0],i*4+j),m4x4);
```

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

To change every value in `val` to its additive inverse, run the following:

```
AQL% UPDATE m4x4 SET val=-val;
```

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

Use the **WHERE** clause to choose attributes based on conditions. For example, you can select only cells with absolute values greater than 5 to set their multiplicative inverse:

```
AQL% UPDATE m4x4 SET val=-pow(val,-1) WHERE abs(val) > 5;
```

```
[
[(0),(-1),(-2),(-3)],
[(-4),(-5),(0.166667),(0.142857)],
[(0.125),(0.111111),(0.1),(0.0909091)],
[(0.0833333),(0.0769231),(0.0714286),(0.0666667)]
]
```

8.3. Array Versions

When an array is updated, a new array version is created. SciDB stores the array versions. For example, in the previous section, SciDB stored every version of m4x4 created by the **UPDATE** command. You can see these versions with `versions`:

```
AFL% versions(m4x4);
```

```
{VersionNo} version_id,timestamp
{1} 1,"2013-02-05 16:18:53"
{2} 2,"2013-02-05 16:18:53"
{3} 3,"2013-02-05 16:18:53"
```

You can see the contents of any previous version of the array by using the version number:

```
AFL% scan(m4x4@1);
```

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

Or the array timestamp:

```
AQL% SELECT * FROM scan(m4x4@datetime('2012-11-19 1:20:50'));
```

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

You can use the array version name in any query. The unqualified name of the array always refers to the most recent version as of the start of the query.

Chapter 9. Changing Array Schemas: Transforming Your SciDB Array

This chapter describes several ways to transform arrays:

- Change the dimensions of an array.
- Rearrange the data in the array using the following operators: reshape, unpack reverse and sort.
- Reduce an array by selecting some subset of its data by using the following operators: subarray, slice, thin.
- Change the attributes of an array.
- Change the dimensions of an array.

9.1. Redimensioning an Array

A common use case for creating and loading SciDB arrays is using data from a data warehouse. This data set may be very large and formatted as a csv file. You can use the csv2scidb utility to convert a csv file to the 1-dimensional array format and load the file into a SciDB array. Once you have a 1-dimensional SciDB array, you can redimension the array to convert the attributes to dimensions.

For example, suppose you have a csv file like this:

```
d,p,val
"device-0","probe-0",0.01
"device-1","probe-0",2.04
"device-2","probe-0",6.09
"device-3","probe-0",12.16
"device-4","probe-0",20.25
"device-0","probe-1",30.36
"device-1","probe-1",42.49
"device-2","probe-1",56.64
"device-3","probe-1",72.81
"device-4","probe-1",91
"device-0","probe-2",111.21
"device-1","probe-2",133.44
"device-2","probe-2",157.69
"device-3","probe-2",183.96
"device-4","probe-2",212.25
"device-0","probe-3",242.56
"device-1","probe-3",274.89
"device-2","probe-3",309.24
"device-3","probe-3",345.61
"device-4","probe-3",384
"device-0","probe-4",424.41
"device-1","probe-4",466.84
"device-2","probe-4",511.29
"device-3","probe-4",557.76
"device-4","probe-4",606.25
```

This data has three columns, two of which are strings and one which is a floating-point number. The column headers are 'd','p',and 'val'. To load this data set, create a 1-dimensional SciDB array with three attributes and load the data into it. For this example, the array is named expo. The dimension name is i, the dimension size is 25, the chunk size is 5. The attributes are s, of type string, p of type string, and val of type double.

```
AFL% CREATE ARRAY device_probe <d:string, p:string, val:double>
 [i=1:25,5,0];
```

When you examine the data, notice that it could be expressed in a 2-dimensional format like this:

	probe-0	probe-1	probe-2	probe-3	probe-4
device-0	0.01	30.36	111.21	242.56	424.41
device-1	2.04	42.49	133.44	274.89	466.84
device-2	6.09	56.64	157.69	309.24	511.29
device-3	12.16	72.81	183.96	345.61	557.76
device-4	20.25	91	212.25	384	606.25

SciDB allows you to redimension the data so that you can store it in this 2-dimensional format. First, create an array with 2 dimensions:

```
AFL% create array two_dim<val:double>[d(string)=5,5,0,
 p(string)=5,5,0];
```

Each of the dimensions is of size 5, corresponding to a dimension in the 5-by-5 table. Now, you can use the redimension_store operator to redimension the array device_probe into the array two_dim:

```
AFL% redimension_store(device_probe, two_dim);
```

```
[
[(0.01),(30.36),(111.21),(242.56),(424.41)],
[(2.04),(42.49),(133.44),(274.89),(466.84)],
[(6.09),(56.64),(157.69),(309.24),(511.29)],
[(12.16),(72.81),(183.96),(345.61),(557.76)],
[(20.25),(91),(212.25),(384),(606.25)]
]
```

Now the data is stored so that device and probe numbers are the dimensions of the array. This means that you can use the dimension indices to select data from the array. For example, to select the second device from the third probe, use the dimension indices:

```
AQL% SELECT val FROM two_dim WHERE d='device-2' AND p='probe-3';
```

```
[
[(),(),(),(),()],
[(),(),(),(),()],
[(),(),(),(309.24),()],
[(),(),(),(),()],
[(),(),(),(),()]
]
```

9.1.1. Cell Collisions

The redimension and redimension_store operators can yield result arrays with fewer cells than the source array. This can occur when there are "cell collisions." A cell collision occurs when a single cell location of the result array has more than one corresponding cell in the source array.

There are three techniques for handling cell collisions:

- synthetic dimensions
- aggregation
- randomly choose a cell from among the candidate cells

For details on these techniques, see the redimension and redimension_store operator reference sections.

9.1.2. Redimensioning Arrays Containing Null Values

Nullable attributes are handled in a special manner by redimension_store. If the source array contains null values for the attribute being transformed, these cells will be dropped during the redimension_store. For example, consider the 1-dimensional array redim_missing:

```
AFL% CREATE ARRAY redim_missing< val1:string,val2:string
 NULL,val3:double >[i=0:9,10,0];
```

```
AFL% show(redim_missing)
```

```
redim_missing

< val1:string,
val2:string NULL DEFAULT null,
val3:double >

[i=0:9,10,0]
```

```
AFL% scan(redim_missing);
```

```
{i} val1,val2,val3
{0} "0","0",1
{1} "0","1",0.540302
{2} "0","2",-0.416147
{3} "0","3",-0.989992
{4} "0","4",-0.653644
{5} "1",null,0.7
{6} "1","1",0.841471
{7} "1","2",0.909297
{8} "1","3",0.14112
{9} "1","4",-0.756802
```

Suppose you want to change the first two attributes into dimension indices and store the third attribute in the resulting 2-dimensional array. Create an array redim_target to store the redimension results:

```
AFL% CREATE ARRAY redim_target <val3:double>
 [val1(string)=2,2,0,val2(string)=5,5,0];
```

```
AFL% redimension_store(redim_missing,redim_target);
```

```
[
[(1),(0.540302),(-0.416147),(-0.989992),(-0.653644)],
[(),(0.841471),(0.909297),(0.14112),(-0.756802)]
```

```
]
```

If it is important to preserve cells with NULL attribute values, you must first use the `substitute` operator to convert NULL values into non-NULL values. This procedure is described in the substitute reference topic.

9.2. Array Transformations

Once you have created, loaded, and redimensioned a SciDB array, you may want to change some aspect of that array. SciDB offers functionality to transform the variables of an array in several ways (attributes and dimensions).

The array transformation operations produce a result array with a new schema. They do not modify the source array. Array transformation operations have the signature:

```
AQL% SELECT  *
      FROM  operation( source_array , parameters )
```

This query outputs a SciDB array. To store that array result, use the **INTO** clause:

```
AQL% SELECT  *
      INTO result_array
      FROM  operation( source_array , parameters )
```

9.2.1. Rearranging Array Data

SciDB offers functionality to rearrange an array data:

- **Reshaping** an array by changing the dimension sizes. This is performed with the `reshape` command.
- **Unpacking** a multidimensional array into a 1-dimensional array is performed with the `unpack` command.
- **Reversing** the cells in an array is performed with the `reverse` command.
- **Sorting** the cells in an array or within subarrays corresponding to each dimension, is performed with the `sort` command.

For example, you might want to reshape your array from an m -by- n array to a $2m$ -by- $n/2$ array. The `reshape` command allows you to transform an array into another compatible schema. Consider a 4×4 array, `m4x4` , with contents and schema as follows:

```
AFL% CREATE ARRAY m4x4 <val:double>[i=0:3,4,0,j=0:3,4,0];
```

```
AFL% scan(m4x4);
```

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

```
]
```

```
AFL% show(m4x4)
```

```
m4x4

< val:double >

[i=0:3,4,0,
j=0:3,4,0]
```

As long as the two array schemas have the same number of cells, you can use reshape to transform one schema into the other. A 4×4 array has 16 cells, so you can use any schema with 16 cells, such as 8×2, as the new schema:

A special case of reshaping is unpacking a multidimensional array to a 1-dimensional result array. When you unpack an array, the coordinates of the array cells are stored in the attributes to the result array. This is particularly useful if you are planning to save your data to csv format. Unpacking also excludes all empty cells from the result array.

The `unpack` command takes the second and higher dimensions of an array and transforms them into attributes along the first dimension. The result array consists of the dimension values of the input array with the attribute values from the corresponding cells appended. So, an attribute value `val` that was in row 1, column 3 of a 2-dimensional array will be transformed into a cell with attribute values 1,3, `val`. For example, a 2-dimensional, 1-attribute array will output a 1-dimensional, 3-attribute array as follows:

```
AFL% CREATE ARRAY m3x3  < val:double >[i=0:2,3,0,j=0:2,3,0];
```

```
AQL% SELECT * INTO m1 FROM unpack(m3x3,k);
```

```
[(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)]
```

```
m1

< i:int64,
j:int64,
val:double >

[k=0:*,9,0]
```

You can reverse the ordering of the data in each dimension of an array with the `reverse` command:

```
m3x3

< val:double >

[i=0:2,3,0,
j=0:2,3,0]
```

```
AQL% SELECT * FROM m3x3;
```

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

```
]
```

```
AQL% SELECT * FROM reverse(m3x3);
```

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

9.2.2. SciDB Array Reducing Operators

One common array task is selecting subsets of an array. SciDB allows you to reduce an array to contiguous subsets of the array cells or noncontiguous subsets of the array's cells.

- A **subarray** is a contiguous block of cells from an array. This action is performed by the `subarray` operator.
- An array **slice** is a subset of the array defined by planes of the array. This action is performed by the `slice` operator.
- A dimension can be winnowed or **thinned** by selecting data at intervals along its entirety. This action is performed by the `thin` operator.

You can select part of an existing array into another array with the `subarray` command. For example, you can select a 2-by-2 array of the last two values from each dimension of the array `m4x4` with the following `subarray` command:

```
AFL% show(m4x4)
```

```
m4x4

< val:double >

[i=0:3,4,0,
j=0:3,4,0]
```

```
AFL% scan(m4x4);
```

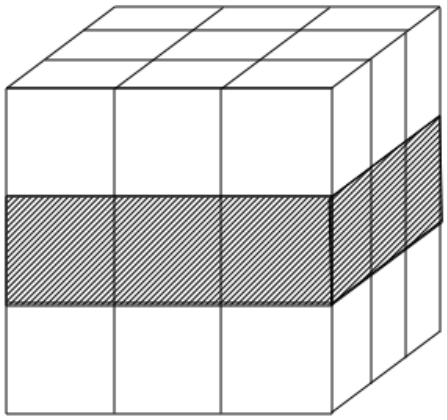
```
[
[(0),(1),(2),(3)],
[(4),(5),(6),(7)],
[(8),(9),(10),(11)],
[(12),(13),(14),(15)]
]
```

```
AQL% SELECT * FROM subarray(m4x4, 2, 2, 3, 4);
```

```
[
[(10),(11)],
[(14),(15)]
]
```

If you have a 3-dimensional array, you might want to select just a flat 2-dimensional slice, as like the cross-hatched section of this image.

Figure 9.1. Select a 2-d slice from a 3-d array



For example, you can select the data in a horizontal slice in the middle of a 3-dimensional array `m3x3x3` by using the `slice` command and specifying the value for dimension `k`:

```
AFL% show(m3x3x3)
```

```
m3x3x3

< val:double >

[i=0:2,3,0,
j=0:2,3,0,
k=0:2,3,0]
```

```
AFL% scan(m3x3x3);
```

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

```
AFL% slice(m3x3x3,k,1);
```

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

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

You may want to sample data uniformly across an entire dimension. The `thin` command selects elements from given array dimensions at defined intervals. For example, you can select every other element from every other row:

```
AFL% show(m4x4)
```

```
m4x4

< val:double >

[i=0:3,4,0,
j=0:3,4,0]
```

```
AFL% scan(m4x4);
```

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

```
AFL% thin(m4x4, 0, 2, 0, 2);
```

```
[
[(0),(2)],
[(8),(10)]
]
```

Let's look at a single array to compare the three SciDB reduction operators.

1. Create a 16×16 array, square_array:

   ```
   AFL% create array square_array
    <val:int64>[i=0:15,16,0,j=0:15,16,0];
   ```

2. Put values of 0–255 into square_array:

   ```
   AFL% store(build(square_array,i*16+j),square_array);
   ```

3. Select a 4x4 **subarray** from the interior of square_array:

   ```
   AFL% subarray(square_array,5,5,8,8);
   ```

   ```
   [
   [(85),(86),(87),(88)],
   [(101),(102),(103),(104)],
   [(117),(118),(119),(120)],
   [(133),(134),(135),(136)]
   ]
   ```

4. **Slice** the sixth column (j=5), and then the tenth row (i=9) from square_array:

   ```
   AFL% slice(square_array,j,5);
   ```

```
[(5),(21),(37),(53),(69),(85),(101),(117),(133),(149),(165),(181),
(197),(213),(229),(245)]
```

```
AFL% slice(square_array,i,9);
```

```
[(144),(145),(146),(147),(148),(149),(150),(151),(152),(153),
(154),(155),(156),(157),(158),(159)]
```

5. Use the **thin** operator to uniformly sample data from square_array:

```
AFL% thin(square_array,0,4,0,8);
```

```
[
[(0),(8)],
[(64),(72)],
[(128),(136)],
[(192),(200)]
]
```

```
AFL% thin(square_array,0,8,0,2);
```

```
[
[(0),(2),(4),(6),(8),(10),(12),(14)],
[(128),(130),(132),(134),(136),(138),(140),(142)]
]
```

```
AFL% thin(square_array,2,8,1,2);
```

```
[
[(33),(35),(37),(39),(41),(43),(45),(47)],
[(161),(163),(165),(167),(169),(171),(173),(175)]
]
```

9.3. Changing Array Attributes

An array's attributes contain the data stored in the array. You can transform attributes by

- Changing the name of the attribute.
- Adding an attribute.
- Changing the order of attributes in a cell.
- Deleting an attribute.

You can change the name of an attribute with the `attribute_rename` command:

```
AQL% SELECT * INTO m3x3_new FROM attribute_rename(m3x3,val,val2);
```

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

```
AFL% show(m3x3_new)
```

```
m3x3_new

< val2:double >

[i=0:2,3,0,
j=0:2,3,0]
```

You can add attributes to an existing array with the `apply` command:

```
AQL% SELECT * INTO m3x3_new_attr FROM apply(m3x3,val2,val
+10,val3,pow(val,2));
```

```
[
[(0,10,0),(1,11,1),(2,12,4)],
[(3,13,9),(4,14,16),(5,15,25)],
[(6,16,36),(7,17,49),(8,18,64)]
]
```

```
AFL% show(m3x3_new_attr)
```

```
m3x3_new_attr

< val:double,
val2:double,
val3:double >

[i=0:2,3,0,
j=0:2,3,0]
```

You can select a subset of an array's attributes and return them using the following statement.

```
AQL% SELECT val2, val3 FROM m3x3_new_attr;
```

```
[
[(10,0),(11,1),(12,4)],
[(13,9),(14,16),(15,25)],
[(16,36),(17,49),(18,64)]
]
```

9.4. Changing Array Dimensions

9.4.1. Changing Chunk Size

If you have created an array with a particular chunk size and then later find that you need a different chunk size, you can use the `repart` operator to change the chunk size. For example, suppose you have an array that is 1000-by-1000 with chunk size 100 in each dimension:

```
AFL% show(chunks);
```

```
[("chunks<val1:double,val2:double> [i=0:999,100,0,j=0:999,100,0]")]
```

You can repartition the chunks to be 10 along one dimension and 1000 in the other:

```
AQL% SELECT * INTO chunks_part
    FROM repart(chunks,<val1:double,val2:double>
```

```
    [i=0:999,10,0,j=0:999,1000,0]);
```

```
AFL% show(chunks_part);
```

```
[("chunks_part<val1:double,val2:double>
 [i=0:999,10,0,j=0:999,1000,0]")]
```

Repartitioning is also important if you want the change the chunk overlap to speed up nearest-neighbor or window aggregate queries.

```
AQL% SELECT * INTO chunks_overlap
    FROM repart(chunks,<val1:double,val2:double>
     [i=0:999,100,10,j=0:999,100,10]);
```

9.4.2. Appending a Dimension

You may need to append dimensions to existing arrays, particularly when you want to do more complicated transformations to your array. This example demonstrates how you can take slices from an existing array and then reassemble them into a array with a different schema. Consider the following 2-dimensional array:

```
AFL% scan(Dsp)
```

```
{d,t} val
{"device-0","probe-0"} 0.01
{"device-0","probe-1"} 30.36
{"device-0","probe-2"} 111.21
{"device-0","probe-3"} 242.56
{"device-0","probe-4"} 424.41
{"device-1","probe-0"} 2.04
{"device-1","probe-1"} 42.49
{"device-1","probe-2"} 133.44
{"device-1","probe-3"} 274.89
{"device-1","probe-4"} 466.84
{"device-2","probe-0"} 6.09
{"device-2","probe-1"} 56.64
{"device-2","probe-2"} 157.69
{"device-2","probe-3"} 309.24
{"device-2","probe-4"} 511.29
{"device-3","probe-0"} 12.16
{"device-3","probe-1"} 72.81
{"device-3","probe-2"} 183.96
{"device-3","probe-3"} 345.61
{"device-3","probe-4"} 557.76
{"device-4","probe-0"} 20.25
{"device-4","probe-1"} 91
{"device-4","probe-2"} 212.25
{"device-4","probe-3"} 384
{"device-4","probe-4"} 606.25
```

```
AFL% show(Dsp)
```

```
Dsp

< val:double >
```

```
[d(string)=5,5,0,
t(string)=5,5,0]
```

Suppose you want to examine a sample plane from each dimension of the array. You can use the slice command to select array slices from array `Dsp` :

```
AQL% SELECT * INTO Dsp_slice_0 FROM slice(Dsp, d, 'device-0');
```

```
{t} val
{"probe-0"} 0.01
{"probe-1"} 30.36
{"probe-2"} 111.21
{"probe-3"} 242.56
{"probe-4"} 424.41
```

```
AQL% SELECT * INTO Dsp_slice_1 FROM slice(Dsp, d, 'device-1');
```

```
{t} val
{"probe-0"} 2.04
{"probe-1"} 42.49
{"probe-2"} 133.44
{"probe-3"} 274.89
{"probe-4"} 466.84
```

```
AQL% SELECT * INTO Dsp_slice_2 FROM slice(Dsp, d, 'device-2');
```

```
{t} val
{"probe-0"} 6.09
{"probe-1"} 56.64
{"probe-2"} 157.69
{"probe-3"} 309.24
{"probe-4"} 511.29
```

The slices are 1-dimensional.

```
AFL% show(Dsp_slice_2)
```

```
Dsp_slice_2

< val:double >

[t(string)=5,5,0]
```

Concatenating these slices will create a 1-d array:

```
AQL% SELECT * INTO Dsp_1d FROM concat(Dsp_slice_0, Dsp_slice_2);
```

```
{t} val
{0} 0.01
{1} 30.36
{2} 111.21
{3} 242.56
{4} 424.41
{5} 6.09
{6} 56.64
{7} 157.69
```

```
{8} 309.24
{9} 511.29
```

```
AFL% show(Dsp_1d)
```

```
Dsp_1d

< val:double >

[t=0:9,5,0]
```

To concatenate these arrays into a 2-dimensional array, you need to add a dimension to both. The `adddim` command will add a stub dimension to the array to increase its dimensionality.

```
AQL% SELECT * INTO Dsp_new FROM concat(adddim(Dsp_slice_0, d),
 adddim(Dsp_slice_2, d));
```

```
{d,t} val
{0,"probe-0"} 0.01
{0,"probe-1"} 30.36
{0,"probe-2"} 111.21
{0,"probe-3"} 242.56
{0,"probe-4"} 424.41
{1,"probe-0"} 6.09
{1,"probe-1"} 56.64
{1,"probe-2"} 157.69
{1,"probe-3"} 309.24
{1,"probe-4"} 511.29
```

```
AFL% show(Dsp_new)
```

```
Dsp_new

< val:double >

[d=0:1,1,0,
t(string)=5,5,0]
```

Chapter 10. SciDB Data Types and Casting

SciDB supports the following data types. You can access this list by using `list('types')` at the AFL command line.

Data Type	Default Value	Description
bool	false	Boolean TRUE (1) or FALSE (0)
char	\0	Single-character
datetime	1970-01-01 00:00:00	Date and time
datetimetz	1970-01-01 00:00:00 -00:00	Date and time with timezone offset.
double	0	Double-precision decimal
float	0	Floating-point number
int8	0	Signed 8-bit integer
int16	0	Signed 16-bit integer
int32	0	Signed 32-bit integer
int64	0	Signed 64-bit integer
string	""	Variable length character string
uint8	0	Unsigned 8-bit integer
uint16	0	Unsigned 16-bit integer
uint32	0	Unsigned 32-bit integer
uint64	0	Unsigned 64-bit integer

Attribute values in SciDB can be cast or converted from one data type to another. SciDB permits type conversions between numerical data types (for example, from int8 to int32 or int8 to double). SciDB also supports the conversion of numeric data types to non-numeric data types, such as string.

Attribute type conversion can be requested explicitly using the following syntax. For example, if you have an integer data type and would like to use an operator only defined to accept double data type attributes, you can use the following conversion step to derive an attribute of the correct type.

```
CREATE ARRAY A <a1: int32>[i=0:0,1,0];
store(build(A,2),A);

apply(A, a2, double(a1));
apply(A, a2, string(a1));
```

This generates a new attribute a2 with double data type from a1. A numeric data type can also be converted to string, which returns a UTF-8 encoded string.

SciDB includes special functions for the conversion between temporal data types.

The following examples demonstrate how to use conversion functions between the datetime data type and the datetime with timezone, datetimetz data type. The date time with time zone data type, datetimetz, uses a timezone offset relative to GMT. You can cast datetime to datetimetz by appending an offset using the following example:

```
create array T<td: datetime>[i=0:0,1,0];
store(build(T, now()), T);
apply(T, dst, append_offset(td, 3600));
```

To append an offset and apply it to the time, use the apply_offset() function, which adds a timezone offset to the datetime. The offset must be expressed in seconds.

```
create array T1 <t:datetime>[i=0:1,1,0];
store(apply(T1,dst,apply_offset(t,3600)),T1);
```

To return the datetime portion of a datetimetz value, use the strip_offset function:

```
apply(T1,dst,strip_offset(dst));
```

To apply the offset to the datetime and return a GMT datetime, use the togmt function:

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

Chapter 11. SciDB Aggregate Reference

This chapter lists SciDB aggregates. Aggregates take as input a set of 1 or more values and return a scalar value. SciDB aggregates have the syntax aggregate_call_N where an aggregate call is one of the following:

- `aggregate_name(attribute_name)`
- `aggregate_name(expression)`

  Note: the `aggregate_name(expression)` syntax exists only in AQL.

Aggregate calls can occur in AQL and AFL statements as follows:

AQL syntaxes

```
SELECT aggregate_call_1[,aggregate_call_2,...,aggregate_call_N]
FROM array;

SELECT aggregate_call_1[,aggregate_call_2,...,aggregate_call_N]
FROM array GROUP BY dimension1[,dimension2];

SELECT aggregate_call_1[,aggregate_call_2,...,aggregate_call_N]
FROM array WHERE expression;

SELECT aggregate(attribute) [,aggregate(attribute)] ...
 INTO dst-array
 FROM src-array | array-expression
 WHERE where-expression
 REGRID AS
        ( PARTITION BY  dimension1 dimension1-size
          [, dimension2 dimension2-size]... ) ;

SELECT aggregate (attribute)[, aggregate (attribute)]...
    INTO dst-array
    FROM src-array | array-expression
    WHERE where-expression
    FIXED | VARIABLE WINDOW AS
    (PARTITION BY dimension1 dim1-low PRECEDING AND dim1-
high FOLLOWING
                [, dimension2 dim2-low PRECEDING  AND dim2-
high FOLLOWING ]... );
```

AFL syntaxes

```
AFL% aggregate(array, aggregate_call_1
[, aggregate_call_2,... aggregate_call_N]
[,dimension_1, dimension_2,...])


AFL% window(array,
dim_1_low,dim_1_high,
[dim_2_low,dim_2_high,]...
```

```
aggregate_1[,aggrgegate_2, ...]

AFL% variable_window(array,
dim_low,dim_high,
aggregate_1[,aggrgegate_2, ...]

AFL% regrid(array,grid_1,grid_2,...,grid_N,
aggregate_call_1[,aggregate_call_2,...,aggregate_call_N]);
```

Name

approxdc — Produces a result array containing approximate counts of the number of distinct values of an attribute.

Synopsis

```
AQL% SELECT approxdc(attribute) FROM array;
```

```
AFL% aggregate(array,approxdc(attribute)[,dimension_1,dimension_2,...]
```

Summary

The approxdc aggregate takes a set of values from an array attribute and returns an approximate count of the number of distinct values present.

The approxdc aggregate does not count null values.

Example

These examples find the approximate number of distinct words in an array or in various subsets of the array.

1. Show the array schema:

```
AFL% show(wordOfConversation)
```

```
wordOfConversation

< wordID:int64 >

[languageID=1:1000,100,0,
conversationID=1:5000000,1000,0,
timeOffsetInSeconds=1:10000,1000,100]
```

2. Show the approximate count of distinct words in the array:

```
AQL% SELECT approxdc(wordID)
       FROM wordOfConversation;
```

3. For each language represented in the array, show the approximate count of distinct words used in all conversations in that language:

```
AQL% SELECT approxdc(wordID)
       FROM wordOfConversation
       GROUP BY languageID;
```

4. For each conversation represented in the array, show the approximate count of distinct words used:

```
AQL% SELECT approxdc(wordID)
       FROM wordOfConversation
       GROUP BY conversationID;
```

Name

avg — Average (mean) aggregate

Synopsis

```
AQL% SELECT avg(attribute) FROM array;
```

```
AFL% aggregate(array,avg(attribute)[,dimension_1,dimension_2,...]
```

Summary

The avg aggregate takes a set of scalar values from an array attribute and returns the average of those values.

The average of an empty set is NULL. The avg of a set that contains only NULL values is also NULL. If the set contains NULL and NOT NULL values, the avg result is an average of the NOT NULL values only.

Example

This example finds the average of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Find the average of every column of m3x3:

   ```
   aggregate(m3x3,val,j)
   ```

   This query returns:

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

Name

count — Returns a count of non-empty cells, or attributes that are not null.

Synopsis

```
AQL% SELECT count(attribute) FROM array;

AFL% aggregate(array,count(attribute))
```

OR

```
AQL% SELECT count(*) FROM array;

AFL% aggregate(array,count(*))
```

Summary

If you use the count(*attribute*) syntax, the count aggregate counts everything except the following:

- empty cells
- attribute values that are NULL.

If you use the count(*) syntax, the count aggregate returns a count of the all the cells present (both NULL and NOT NULL).

Examples

This example finds the number of nonempty cells in a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   AFL% CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values 1 along the diagonal of m3x3 and leave the remaining cells empty:

   ```
   AFL% store(build_sparse(m3x3,1,i=j),m3x3);

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

3. Find the number of nonempty cells in the matrix:

   ```
   AFL% aggregate(m3x3,count(val));

   [(3)]
   ```

This example finds the number of nonempty and non-null cells from a matrix that contains some NULL values.

1. Show the contents of count_array:

```
[
[(),(),()],
[(null),(null),(null)],
[("a7"),("a8"),("a9")]
]
```

2. Count the number of nonempty cells:

```
AQL% SELECT count(*) FROM A;

[(6)]
```

3. Count the number of nonempty and non-null cells for the `value` attribute:

```
AQL% SELECT count(value) FROM A;

[(3)]
```

Name

max — Maximum value aggregate

Synopsis

```
AQL% SELECT max(attribute) FROM array;
```

```
AFL% aggregate(array,max(attribute)[,dimension_1,dimension_2,...]
```

Summary

The max aggregate takes a set of scalar values from an array attribute and returns the maximum value.

The maximum value of an empty set is NULL. The max of a set that contains only NULL values is also NULL. If the set contains NULL and NOT NULL values, the max aggregate considers only NOT NULL values.

Example

This example finds the maximum of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Find the maximum value of each column:

   ```
   aggregate(m3x3,max(val),j);
   ```

   This query returns:

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

Name

min — Minimum value aggregate

Synopsis

```
AQL% SELECT min(attribute) FROM array;
```

```
AFL% aggregate(array,min(attribute)[,dimension_1,dimension_2,...]
```

Summary

The min aggregate takes a set of scalar values from an array attribute and returns the minimum value.

The minimum value of an empty set is NULL. The min of a set that contains only NULL values is also NULL. If the set contains NULL and NOT NULL values, the min aggregate considers only NOT NULL values.

Example

This example finds the minimum of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Find the minimum value of every column of m3x3:

   ```
   aggregate(m3x3,min(val),j);
   ```

   This query returns:

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

Name

stdev — Standard deviation aggregate

Synopsis

```
AQL% SELECT stdev(attribute) FROM array;

AFL% aggregate(array,stdev(attribute)[,dimension_1,dimension_2,...])
```

Summary

The stdev aggregate takes a set of scalar values from an array attribute and returns the standard deviation of those values.

The standard deviation of an empty set is NULL. The standard deviation of a set that contains only NULL values is also NULL. If the set contains NULL and NOT NULL values, the stdev aggregate considers only NOT NULL values.

Example

This example finds the standard deviation of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put random values between 1 and 9 into m3x3:

   ```
   store(build(m3x3,random()%10/1.0),m3x3);
   ```

   This query outputs:

   ```
   [
   [(2),(8),(0)],
   [(5),(2),(6)],
   [(2),(0),(2)]
   ]
   ```

3. Find the standard deviation of every column of m3x3:

   ```
   aggregate(m3x3,stdev(val),j);
   ```

   This query returns:

   ```
   [(1.73205),(4.16333),(3.05505)]
   ```

Name

sum — Sum aggregate

Synopsis

```
AQL% SELECT sum(attribute) FROM array;
```

```
AFL% aggregate(array,sum(attribute)[,dimension_1,dimension_2,...])
```

Summary

The sum aggregate calculates the cumulative sum of a group of values.

The sum of an empty set is 0. The standard deviation of a set that contains only NULL values is also 0. If the set contains NULL and NOT NULL values, the result is the sum of all the NOT NULL values.

Example

This example finds the sum of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Find the sum of each column in m3x3:

   ```
   aggregate(m3x3,sum(val),j)
   ```

   This query returns:

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

Name

var — Variance aggregate

Synopsis

```
AQL% SELECT var(attribute) FROM array;

AFL% aggregate(array,var(attribute)[,dimension_1,dimension_2,...])
```

Summary

The var aggregate returns the variance of a set of values.

The variance of an empty set is NULL. The variance of a set that contains only NULL values is also NULL. If the set contains NULL and NOT NULL values, the var aggregate considers only NOT NULL values.

Example

This example finds the variance of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put random values between 1 and 9 into m3x3:

   ```
   store(build(m3x3,random()%10/1.0),m3x3);
   ```

   This query returns:

   ```
   [
   [(2),(8),(0)],
   [(5),(2),(6)],
   [(2),(0),(2)]
   ]
   ```

3. Find the variance for each column of m3x3:

   ```
   aggregate(m3x3,var(val),j)
   ```

   This query returns:

   ```
   [(3),(17.3333),(9.33333)]
   ```

Chapter 12. SciDB Function Reference

This chapter lists the SciDB functions that are available for use in SciDB expressions. Expressions can be used in the following types of syntaxes:

AQL Syntax:

```
SELECT expression FROM array;

SELECT expression1 FROM array WHERE expression2;
```

AFL Syntax:

```
operator(array,expression);
```

Name

Algebraic functions — Perform basic arithmetic in a query expression

Synopsis

```
SELECT expression1 FROM array WHERE expression2;
```

Summary

These functions perform perform basic arithmetic.

Function Name	Description	Syntax	Datatypes
%	Remainder	scalar%scalar	int8, int16, int32, int64, uint8, uint16, uint32, uint64
*	Multiplication	scalar*scalar	double, float, int8, int16, int32, int64, uint8, uint16, uint32, uint64
+	Addition	scalar + scalar	(datetime, int64), double, float, string, int8, int16, int32, int64, uint8, uint16, uint32, uint64
-	Subtraction or additive inverse	scalar-scalar -scalar	(datetime, int64), datetime, double, float, string, int8, int16, int32, int64, uint8, uint16, uint32, uint64
/	Division	scalar/scalar	double, float, string, int8, int16, int32, int64, uint8, uint16, uint32, uint64
abs	Absolute value	abs(scalar)	double, int32
floor	Round to next-lowest integer	floor(scalar)	double
pow	Raise to a power	pow(base,exponent)	double
random()	Generate random positive integer	random()	Output default is uint32
sqrt	Square root	sqrt(scalar)	double,float

Name

Comparison functions — Compare scalar values

Synopsis

```
SELECT expression1 FROM array WHERE expression2;
```

Summary

These functions compare scalar values. Comparison operations result in a value of 1 (TRUE), 0 (FALSE), or NULL.

Function Name	Description	Syntax	Datatypes
<	Less than	scalar < scalar	bool, char, datetime, datetimetz, double, float, int8, int16, int32, int64, uint8, uint16, uint32, uint64, string
<=	Less than or equal	scalar*scalar	bool, char, datetime, datetimetz, double, float, int8, int16, int32, int64, uint8, uint16, uint32, uint64, string
<>	Not equal	scalar <> scalar	bool, char, datetime, datetimetz, double, float, int8, int16, int32, int64, uint8, uint16, uint32, uint64, string
=	Equal	scalar = scalar	bool, char, datetime, datetimetz, double, float, int8, int16, int32, int64, uint8, uint16, uint32, uint64, string
>	Greater than	scalar > scalar	bool, char, datetime, datetimetz, double, float, int8, int16, int32, int64, uint8, uint16, uint32, uint64, string
>=	Greater than or equal	scalar >= scalar	bool, char, datetime, datetimetz, double, float, int8, int16, int32, int64, uint8, uint16, uint32, uint64, string
iif	Inline if	iif(expression, if_true, otherwise)	bool, char, datetime, datetimetz, double, float, int8, int16, int32, int64, uint8, uint16, uint32, uint64, string
is_nan		is_nan(scalar)	double
is_null		is_null(scalar)	void
not	Boolean NOT	not(scalar)	bool

Name

Transcendental functions — Perform mathematical functions in a query expression

Synopsis

```
SELECT expression1 FROM array WHERE expression2;
```

Summary

These functions perform non-algebraic functions including trigonometry and logarithmic functions.

Function Name	Description	Syntax	Datatypes
acos	Inverse (arc) cosine in radians	acos(scalar)	double,float
asin	Inverse (arc) sine in radians	asin(scalar)	double,float
atan	Inverse (arc) tangent in radians	atan(scalar)	double,float
cos	Cosine (input in radians)	cos(scalar)	double,float
exp	Exponential	exp(scalar)	double,float
log	Base-*e* logarithm	log(scalar)	double,float
log10	Base-10 logarithm	log10(scalar)	double,float
sin	Sine (input in radians)	sin(scalar)	double,float
tan	Tangent (input in radians)	tan(scalar)	double,float

Examples

This example calculates the sine, cosine, and tangent of a set of values.

1. Create a 1-dimensional array, `trig_1`, and store values of 0, π/3, 2π/3, π, 4/3π, and 5/3π.

   ```
   AFL% store(build(trig_1,(2.0/3.0)*acos(0)*x),trig_1);

   [(0),(1.0472),(2.0944),(3.14159),(4.18879),(5.23599)]
   ```

2. Calculate the trigonometric functions for array `values`.

   ```
   AQL% SELECT cos(val) FROM trig_1;

   [(1),(0.5),(-0.5),(-1),(-0.5),(0.5)]

   AQL% SELECT sin(val) FROM trig_1;

   [(0),(0.866025),(0.866025),(-8.74228e-08),(-0.866025),(-0.866025)]

   AQL% SELECT tan(val) FROM trig_1;

   [(0),(1.73205),(-1.73205),(8.74228e-08),(1.73205),(-1.73205)]
   ```

This example calculates the arcsine, arccosine, and arctangent of a set of values.

1. Create a 1-dimensional array, `trig_2`, and store values between 0 and 1.

```
AFL% store(build(trig_2,1.0/(x+1)),trig_2);

[(1),(0.5),(0.333333),(0.25),(0.2),(0.166667)]
```

2. Calculate the inverse trigonometric functions for array `values`.

```
AQL% SELECT acos(val) FROM trig_2;

[(0),(1.0472),(1.23096),(1.31812),(1.36944),(1.40335)]

AQL% SELECT asin(val) FROM trig_2;

[(1.5708),(0.523599),(0.339837),(0.25268),(0.201358),(0.167448)]

AQL% SELECT atan(val) FROM trig_2;

[(0.785398),(0.463648),(0.321751),(0.244979),(0.197396),
(0.165149)]
```

This example calculates the exp, log, and natural log of a set of values.

1. Calculate the exponential function (e^x) for a set of values.

```
AFL% store(build(logs,(1.7*(x+0.01))),logs);

[(0.017),(1.717),(3.417),(5.117),(6.817),(8.517)]

AQL% SELECT exp(val) FROM logs;

[(1.01715),(5.5678),(30.4778),(166.834),(913.241),(4999.04)]
```

2. Calculate the log and natural log for a set of values.

```
AFL% store(build(logs, pow(10,x)),logs);

[(1),(10),(100),(1000),(10000),(100000)]

AQL% SELECT log(val) FROM logs;

[(0),(2.30259),(4.60517),(6.90776),(9.21034),(11.5129)]

AQL% SELECT log10(val) FROM logs;

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

Chapter 13. SciDB AFL Operator Reference

This reference guide lists the operators available in SciDB's Array Functional Language (AFL). Operators can be used in several ways in SciDB queries.

- Operators can be used in AQL in **`FROM`** clauses.
- Operators can be used at the AFL command line or, in some cases, nested with other AFL operators.

Operator syntaxes generally follow this pattern:

```
operator(array|array_expression|anonymous_schema,arguments);
```

The first argument to an operator is generally an array that you have previously created and stored in your current SciDB namespace. However, in many cases, the first argument may also be a SciDB operator. The output of the nested operator serves as the input for the outer operator. This is called an *array expression*.

```
operator_1(operator_2(array,arguments_2),arguments_1);
```

Not all SciDB operators can take another operator as input. These exceptions are noted in the Synopsis section of the operator's reference page. An operator argument that is specified as *`array`* can also be an array expression. An operator argument that is specified as *`named_array`* can only be an array that you have previously created and stored.

In addition, some operators can take an array schema as input instead of a named array or array expression. This is called an *anonymous schema*. An operator that can take an anonymous schema instead of an array will be indicated in the arguments of the Synopsis section.

Name

adddim — Produces a result array with one more dimension than a given source array.

Synopsis

```
adddim(array,new_dimension);
```

Summary

The adddim operator adds a stub, integer dimension to an array to increase its dimensionality by 1. The datatype of the new dimension is int64. The size of the new dimension is 1.

The cardinality of the added dimension is limited to 1—that is, you can only increase a 1-dimensional array to 2 dimensions, a 2-dimensional array to 3 dimensions, and so on. If you need more redimensioning capabilities, use the redimension operator.

SciDB does not have an analogous operator for adding an attribute. To add an attribute to an array, use the apply operator.

Example

This example creates a 2-dimensional array from 1-dimensional arrays.

1. Create a vector of zeros:

```
AFL% store(build(<val:double>[i=0:4,5,0],0),vector0);
```

```
{i} val
{0} 0
{1} 0
{2} 0
{3} 0
{4} 0
```

2. Create a vector of ones:

```
AFL% store(build(<val:double>[j=0:4,5,0],1),vector1);
```

```
{j} val
{0} 1
{1} 1
{2} 1
{3} 1
{4} 1
```

3. Concatenate these vectors without increasing their dimensionality. Note that the output is 1-dimensional:

```
AFL% concat(vector0,vector1);
```

```
{i} val
{0} 0
{1} 0
{2} 0
```

```
{3} 0
{4} 0
{5} 1
{6} 1
{7} 1
{8} 1
{9} 1
```

4. Use adddim to add a dimension to both vectors and then concatenate them. The result will have two dimensions:

```
AFL% concat(adddim(vector0,x),adddim(vector1,y));
```

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

Name

allversions — Return a result array containing all versions of an existing array.

Synopsis

```
allversions(named_array)
```

Summary

The allversions operator takes all versions of an array and returns a result array that combines all versions of *named_arrary* into one array. The resulting array has a dimension called VersionNo that has indices 1–*final_array_version* appended to the left-most dimension. The argument *named_array* must be an array that was previously created and stored in the SciDB namespace.

Example

This example creates a 3×3 matrix, updates it, and then uses allversions to combine all previous versions of the array.

1. Create array m3x3 and load zeros into it:

```
AFL% store(build(<val:double>[i=0:2,3,0,j=0:2,3,0],0),m3x3);
```

```
[
[(0),(0),(0)],
[(0),(0),(0)],
[(0),(0),(0)]
]
```

2. Update m3x3 with 100 in every cell:

```
AFL% store(build(m3x3,100),m3x3);
```

```
[
[(100),(100),(100)],
[(100),(100),(100)],
[(100),(100),(100)]
]
```

3. Update m3x3 with 200 in every cell:

```
AFL% store(build(m3x3,200),m3x3);
```

```
[
[(200),(200),(200)],
[(200),(200),(200)],
[(200),(200),(200)]
]
```

4. Use allversions in a store statement to store an array containing all three versions of m3x3:

```
AFL% store(allversions(m3x3),m3x3_versions);
```

```
[
```

```
[
[(0),(0),(0)],
[(0),(0),(0)],
[(0),(0),(0)]
],
[
[(100),(100),(100)],
[(100),(100),(100)],
[(100),(100),(100)]
],
[
[(200),(200),(200)],
[(200),(200),(200)],
[(200),(200),(200)]
]
]
```

The array m3x3_versions has the following schema:

```
{i} schema
{0} "m3x3_versions<val:double>
 [VersionNo=1:3,1,0,i=0:2,3,0,j=0:2,3,0]"
```

Name

analyze — Produces a 1-dimensional result array where each cell describes some simple statistics about the values in one attribute of a stored array.

Synopsis

```
analyze(array[, attribute1, attribute2, ...]);
```

Summary

The analyze operator helps you characterize the contents of an array. Each cell in the result array includes the following attributes:

- attribute_number: An index for the one-dimensional result array.
- atttribute_name: The name of an attribute from the source array.
- min: The lowest value for the attribute in the source array.
- max: The highest value for the attribute in the source array.
- distinct_count: An estimate of the number of different values appearing in the source array.
- non_null_count: The number of cells in the array with non-null values for the attribute.

You can use the analyze operator to characterize some or all of the attributes in an array. To characterize some, name them explicitly with the attribute parameter. To characterize all attributes, you can name them all explicitly, or you can omit the attribute parameter entirely.

You can use the analyze operator in the FROM clause of an AQL SELECT statement, as a stand-alone operator in a AFL statement, or as an operand within other SciDB operators.

Example

This example first shows a 1-dimensional array, then analyzes its attributes, then analyzes only its numeric attributes.

1. Show the schema for the 1-dimensional source array:

   ```
   AFL% show(winnersFlat)
   ```

   ```
   winnersFlat

   < event:string,
   year:int64,
   person:string,
   country:string,
   time:double >

   [i=0:*,12,0]
   ```

2. Create a result array describing the attributes of the source array:

   ```
   AFL% attributes(winnersFlat);
   ```

```
{No} name,type_id,nullable
{0} "event","string",false
{1} "year","int64",false
{2} "person","string",false
{3} "country","string",false
{4} "time","double",false
```

3. Create a result array characterizing the values of the attributes of the source array:

```
AFL% analyze(winnersFlat);
```

```
{attribute_number}
 attribute_name,min,max,distinct_count,non_null_count
{0} "country","Canada","USA",5,6
{1} "event","dash","steeplechase",3,6
{2} "person","Abera","Wanjiru",6,6
{3} "time","9.84","7956",6,6
{4} "year","1996","2008",3,6
```

4. Create a result array characterizing the values of the numeric attributes (time and year) of the source array:

```
AFL% analyze(winnersFlat,year,time);
```

```
{attribute_number}
 attribute_name,min,max,distinct_count,non_null_count
{0} "time","9.84","7956",6,6
{1} "year","1996","2008",3,6
```

Name

apply — Produces a result array similar to a source array, but with additional attributes whose values are calculated from parameters you supply.

Synopsis

```
apply(source_array,new_attribute1,expression1
              [,new_attribute2,expression2]...);
```

Summary

Use the apply operator to produce a result array with new attributes and to compute values for them. The new array includes all attributes present in the source array, plus the newly created attributes. The newly created attribute(s) must not have the same name as any of the existing attributes of the source array.

Examples

This example produces a result array similar to an existing array (called distance), but with an additional attribute (called kilometers).

1. Create an array called distance with an attribute called miles:

   ```
   AFL% CREATE ARRAY distance <miles:double> [i=0:9,10,0];
   ```

2. Store values of 100–1000 into the array:

   ```
   AFL% store(build(distance,i*100.0),distance);
   ```

3. Apply the expression 1.6 * miles to distance and name the result kilometers:

   ```
   AFL% apply(distance,kilometers,1.6*miles);
   ```

   ```
   {i} miles,kilometers
   {0} 0,0
   {1} 100,160
   {2} 200,320
   {3} 300,480
   {4} 400,640
   {5} 500,800
   {6} 600,960
   {7} 700,1120
   {8} 800,1280
   {9} 900,1440
   ```

This example combines the array operator and the xgrid operator to produce a result array that is an enlarged version of an existing array. The enlargement includes more cells (via xgrid) and an additional attribute called val_2 (via apply).

1. Create a 1-dimensional array called vector:

   ```
   AFL% CREATE ARRAY vector <val:double>[i=0:9,10,0];
   ```

2. Put values of 1–10 into vector and store the result:

```
AFL% store(build(vector,i+1),vector);
```

3. Use the xgrid operator to expand `vector` and the apply operator to add an attribute whose values contain the additive inverse of the dimension index:

```
AFL% apply(xgrid(vector,2),val_2,-i);
```

```
{i} val,val_2
{0} 1,0
{1} 1,-1
{2} 2,-2
{3} 2,-3
{4} 3,-4
{5} 3,-5
{6} 4,-6
{7} 4,-7
{8} 5,-8
{9} 5,-9
{10} 6,-10
{11} 6,-11
{12} 7,-12
{13} 7,-13
{14} 8,-14
{15} 8,-15
{16} 9,-16
{17} 9,-17
{18} 10,-18
{19} 10,-19
```

This example uses the apply operator and a data type function to produce a result array whose attribute values have been cast to a new datatype.

1. Create an array called integer with an int64 attribute:

```
AFL% store(build(<val:int64>[i=0:9,10,0],i+1),A);
```

2. Use apply to apply the data conversion function `double` to the attribute val.

```
AFL% apply(A,val_2,double(val));
```

```
{i} val,val_2
{0} 1,1
{1} 2,2
{2} 3,3
{3} 4,4
{4} 5,5
{5} 6,6
{6} 7,7
{7} 8,8
{8} 9,9
{9} 10,10
```

Name

attribute_rename — Produces a result array with the same dimensions, attributes, and cell values as a source array, but with one or more of the attributes renamed.

Synopsis

```
attribute_rename(array,old_attribute1,new_attribute1
[, old_attribute2,new_attribute2]...);
```

Summary

The attribute_rename operator produces a result array that is nearly identical to a source array, except that one or more attributes are renamed.

You can use the attribute_rename operator in the FROM clause of an AQL SELECT statement, as a stand-alone operator in a AFL statement, or as an operand within other SciDB operators.

Example

This example takes a 2-D array with three attributes and produces an identical array, except the third attribute is renamed.

1. Show the source_array:

```
AFL% scan(champions)
```

```
{year,event} person,country,time
{1996,"dash"} "Bailey","Canada",9.84
{1996,"marathon"} "Thugwane","RSA",7956
{2000,"dash"} "Greene","USA",9.87
{2000,"marathon"} "Abera","Ethiopia",7811
{2000,"steeplechase"} "Kosgei","Kenya",503.17
{2008,"marathon"} "Wanjiru","Kenya",7596
```

2. Use the project operator to exclude the `person` attribute, and the attribute_rename operator to rename the `time` attribute:

```
AFL% attribute_rename(project(champions,country,time)
   ,time,time_in_seconds);
```

```
{year,event} country,time_in_seconds
{1996,"dash"} "Canada",9.84
{1996,"marathon"} "RSA",7956
{2000,"dash"} "USA",9.87
{2000,"marathon"} "Ethiopia",7811
{2000,"steeplechase"} "Kenya",503.17
{2008,"marathon"} "Kenya",7596
```

Name

attributes — Produces a 1-dimensional result array where each cell describes one attribute of a stored array.

Synopsis

```
attributes(named_array);
```

Summary

The attributes operator produces a result array where each cell describes an attribute of the named array. Each cell includes the attribute name, the attribute data type, a Boolean flag representing whether or not the attribute can be null, and the sequence number of the attribute as it appears in the named array. The argument *named_array* must be an array that was previously created and stored in SciDB.

You can use the attributes operator in the FROM clause of an AQL SELECT statement, as a stand-alone operator in a AFL statement, or as an operand within other SciDB operators.

Example

This example first shows a stored array, then creates a result array describing its attributes, then creates a result array describing each of its nullable attributes.

1. Show the source_array:

```
AFL% scan(champions)
```

```
{year,event} person,country,time
{1996,"dash"} "Bailey","Canada",9.84
{1996,"marathon"} "Thugwane","RSA",7956
{2000,"dash"} "Greene","USA",9.87
{2000,"marathon"} "Abera","Ethiopia",7811
{2000,"steeplechase"} "Kosgei","Kenya",503.17
{2008,"marathon"} "Wanjiru","Kenya",7596
```

2. Create a result array describing the attributes of the named array:

```
AFL% attributes(champions);
```

```
{No} name,type_id,nullable
{0} "person","string",false
{1} "country","string",false
{2} "time","double",true
```

3. Create a result array describing the nullable attributes of the named array:

```
AQL% SELECT * FROM attributes(champions) WHERE nullable = true;
```

```
{No} name,type_id,nullable
{2} "time","double",true
```

Name

avg — Average (mean) value

Synopsis

```
avg(array,attribute[,dimension_1,dimension_2,...]
```

Summary

The avg operator finds the average value of an array attribute.

Note

The avg operator provides the same functionality as the avg aggregate, but has a different syntax. For details, see the avg aggregate reference.

Example

This example finds the average value along the second dimension of a 4×4 matrix.

1. Create an array named avg_array:

```
AQL% CREATE ARRAY avg_array<val:double>[i=0:3,4,0,j=0:3,4,0];
```

2. Store values of 0–15 in avg_array:

```
AFL% store(build(avg_array,i*4+j),avg_array);
```

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

3. Find the average value along the dimension j:

```
AFL% avg(avg_array,val,j);
```

```
{i} val_avg
{0} 1.5
{1} 5.5
{2} 9.5
{3} 13.5
```

Name

avg_rank — Rank elements of a matrix

Synopsis

```
avg_rank (array, [attribute [, dimension_1 [, dimension_2... ]]])
```

Summary

The `avg_rank` operator ranks array elements and calculates average rank as the average of the upper bound (UB) and lower bound (LB) rankings. The LB ranking of A, same as returned by rank, is the number of elements less than A, plus 1. The UB ranking of A is the number of elements less than or equal to A, plus 1. avg_rank returns the average of the UB and LB ranking for each element.

If no duplicates are present, then for each element the UB rank is the same as the LB rank and avg_rank returns exactly the same result as rank.

Example

This example calculates ranks along the columns of a matrix where there are ties within columns.

1. Create a 4×4 array called rank:

```
AFL% create array rank_array <val:double>[i=0:3,4,0,j=0:3,4,0];
```

2. Put random values of 0–6 into rank:

```
AFL% store(build(rank_array,random()%7/1.0),rank_array);
```

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

3. Rank the elements in rank_array by dimension i:

```
AFL% avg_rank(rank_array,val,i);
```

```
[
[(4,2),(2,1),(6,4),(5,3)],
[(4,2.5),(2,1),(4,2.5),(5,4)],
[(0,1),(1,2.5),(3,4),(1,2.5)],
[(5,3.5),(0,1.5),(5,3.5),(0,1.5)]
]
```

Name

bernoulli — Select random array cells

Synopsis

```
bernoulli(array,probability[, seed]);
```

Summary

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.

Example

This example select cells at random from a 5×5 matrix, and uses a seed value to select the same cells in successive trials.

1. Create an array called bernoulli_array:

```
AFL% CREATE ARRAY
 bernoulli_array<val:double>[i=0:4,5,0,j=0:4,5,0];
```

2. Store values of 1–25 in bernoulli_array:

```
AFL% store(build(bernoulli_array,i*5+1+j),bernoulli_array);
```

```
[
[(1),(2),(3),(4),(5)],
[(6),(7),(8),(9),(10)],
[(11),(12),(13),(14),(15)],
[(16),(17),(18),(19),(20)],
[(21),(22),(23),(24),(25)]
]
```

3. Select cells at random with a probability of .5 that a cell will be included. Each successive call to bernoulli will return different results.

```
AFL% bernoulli(bernoulli_array,0.5);
```

```
[
[(),(),(),(),()],
[(),(),(),(9),(10)],
[(),(12),(),(14),(15)],
[(),(),(18),(19),()],
[(21),(22),(),(),()]
]
```

```
AFL% bernoulli(bernoulli_array,0.5);
```

```
[
[(),(2),(),(),(5)],
[(6),(),(),(9),()],
[(),(),(13),(14),(15)],
```

```
[(16),(17),(18),(),()],
[(21),(),(23),(),()]
]
```

4. To reproduce earlier results, use a seed value. Seeds must be an integer on the interval [0, INT_MAX].

```
AFL% bernoulli(bernoulli_array,0.5,15);
```

```
[
[(),(2),(),(),()],
[(6),(),(8),(9),(10)],
[(),(),(),(14),()],
[(16),(),(),(19),(20)],
[(21),(22),(),(),()]
]
```

```
AFL% bernoulli(bernoulli_array,0.5,15);
```

```
[
[(),(2),(),(),()],
[(6),(),(8),(9),(10)],
[(),(),(),(14),()],
[(16),(),(),(19),(20)],
[(21),(22),(),(),()]
]
```

Name

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

Synopsis

```
between(array,low_coord1[,low_coord2,...],
              high_coord1[,high_coord2,...]);
```

Summary

The between operator accepts an input array and a set of coordinates specifying a region within the array. The number of coordinate pairs in the input must be equal to the number of dimensions in the array. The output is an array of the same shape as input, where all cells outside of the given region are marked empty.

The subarray operator is similar, except that it returns the specified region only. For details, see the subarray operator reference.

You can use the between operator in the FROM clause of an AQL SELECT statement, as a stand-alone operator in a AFL statement, or as an operand within other SciDB operators.

Example

This example selects 4 elements from a 16-element array.

1. Create a 4×4 array called between_array:

```
AQL% CREATE ARRAY between_array <val:double>[i=0:3,4,0,j=0:3,4,0];
```

2.
```
AFL% store(build(between_array,i*4+j),between_array);
```

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

3. Select all values from the last two rows and last two columns from between_array:

```
AFL% between(between_array,2,2,3,3);
```

```
[
[(),(),(),()],
[(),(),(),()],
[(),(),(10),(11)],
[(),(),(14),(15)]
]
```

Name

build — Produces a result set that is a new single-attribute array populated with values.

Synopsis

```
build(template_array|schema_definition,expression);
```

Summary

The build operator produces a result array with the same shape as the template array, but with attribute values equal to the value of *expression*. The expression argument can be any combination of SciDB functions applied to constants or SciDB attributes. The template array or schema definition must have exactly one attribute.

Limitations

- The build operator can only take arrays with one attribute.
- The build operator can only take arrays with bounded dimensions.

Example

This query creates a 4×4 array of ones from a schema definition:

```
AFL% build(<val:double>[i=0:3,4,0,j=0:3,4,0],1);
```

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

This query creates a 4×4 identity matrix from a schema definition:

```
AFL% build(<val:double>[i=0:3,4,0,j=0:3,4,0],iif(i=j,1,0));
```

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

This query creates a 4×4 array of monotonically increasing values:

```
AFL% build(<val:double>[i=0:3,4,0,j=0:3,4,0],i*4+j);
```

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

```
[(12),(13),(14),(15)]
]
```

Remember, the build operator produces a result array and does not modify the template array. To store the result from a build operator, create an array and use the store operator with the build operator. This query creates an array called identity_matrix and then stores the result of the build operator:

```
AFL% CREATE ARRAY identity_matrix <val:double>[i=0:3,4,0,j=0:3,4,0];

AFL% store(build(identity_matrix,iif(i=j,1,0)),identity_matrix);

AFL% scan(identity_matrix);

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

Name

build_sparse — Produces a sparse result array and assigns values to its non-empty cells.

Synopsis

```
AFL% build_sparse(template_array|schema_definition,
                  expression,boolean_expression);
```

Summary

In SciDB, a sparse array is an array that allows empty cells. SciDB dense arrays do not allow empty cells. You can use `build_sparse` to create arrays with empty cells—but you can also build dense arrays with the `build_sparse` operator.

That is, you can you use build_sparse to create an array, and fill every cell with a value; in this case, you have used the `build_sparse` operator to create a dense array.

- A **sparse** array may (but does not have to) contain empty cells.
- A **dense** array cannot contain empty cells.

Note

NULL, 0, and empty are distinct. That is, a dense array can have cells that contain 0 or NULL, but cannot contain empty cells.

The build_sparse operator takes as input a template_array or schema definition, an expression that defines a scalar value, and an expression that defines a Boolean value. The argument *`template_array`* must be an array that was previously created and stored in SciDB. The output of build_sparse is a result array with the same schema as the template array or schema definition, the value specified by *`expression`* wherever *`boolean_expression`* evaluates to true, and empty cells wherever *`boolean_expression`* evaluates to false.

Limitations

- The `build_sparse` operator can only take arrays with one attribute.
- The `build_sparse` operator can only take arrays with bounded dimensions.

Example

In this example, we build a sparse identity matrix, and then a dense identity matrix. In the sparse matrix, only the diagonal elements are present, while in the dense matrix, only the diagonal elements have non-NULL values.

Here we create and store the sparse identity matrix in `m3x3_sparse`:

```
AFL% store(build_sparse(<val:double>[i=0:3,4,0,j=0:3,4,0],1,i=j),
       m3x3_sparse);
```

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

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

Here we create and store the sparse identity matrix in `m3x3_dense`:

```
AFL% store(build(<val:double
 null>[i=0:3,4,0,j=0:3,4,0],iif(i=j,1,NULL)),m3x3_dense);
```

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

Name

cast — Produces a result array with the same dimensions, attributes, and cells as a source array—but with other differences, which can include different names for its dimensions or attributes, different nulls-allowed status for an attribute, and different datatypes for its dimensions.

Synopsis

```
cast(array,template_array|schema_definition);
```

Summary

The cast operator has three primary uses:

- To change names of attributes or dimensions
- To change a non-integer dimension to an integer dimension
- To change a nulls-disallowed attribute to a nulls-allowed attribute

A single cast invocation can be used to make all of these changes at once. The input array and template arrays should have the same numbers and types of attributes.

The cells of the result array have the same attribute values as the corresponding cells of the source array.

If you use cast to change a non-integer dimension to an integer dimension, the new dimension will have one value for every value in the source dimension. The values will be consecutive integers in the range you supply in the schema_definition parameter or in the schema definition of the template array.

You can use the cast operator in the FROM clause of an AQL SELECT statement, as a stand-alone operator in a AFL statement, or as an operand within other SciDB operators.

Examples

This example shows the schema of an array, combines the recast and store operators to change some characteristics of the array, and shows the schema of the new array.

1. Show the schema of an existing array:

```
AFL% show(winningTime)
```

```
winningTime

< time:double >

[year=1996:2008,1,0,
event(string)=3,1,0]
```

2. Use the cast operator to produce a result array similar to the existing array, but with these changes:
   - The attribute in the new array allows null values.
   - The attribute in the new array is called "time_in_seconds."
   - The string dimension "event" of the source array has been replaced with an int64 dimension "event-Code" in the new array.

In this example, the cast operator appears as an operand of the store operator, yielding a named array "winningTimeRecast":

```
AFL% store
   (
   cast
      (
      winningTime,
         <time_in_seconds:double null>
         [year=1996:2008,1,0, eventCode=0:2,1,0]
       ),
       winningTimeRecast
   );
```

```
{year,eventCode} time_in_seconds
{1996,0} 9.84
{1996,1} 7956
{2000,0} 9.87
{2000,1} 7811
{2000,2} 503.17
{2008,1} 7596
```

3. Show the schema of the new array:

```
AFL% show(winningTimeRecast)
```

```
winningTimeRecast

< time_in_seconds:double NULL >

[year=1996:2008,1,0,
eventCode=0:2,1,0]
```

Name

concat — Concatenate two arrays

Synopsis

```
concat(left_array,right_array);
```

Summary

The concat operator concatenates two arrays with the same 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 chunk size and overlap. Both inputs must have the same attributes.

Example

This example concatenates a 4×3 array and a 2×3 array.

1. Create a 4×3 array left_array containing value 1 in all cells:

   ```
   AFL% create array left_array <val:double>[i=0:3,6,0,j=0:2,3,0];
   ```

   ```
   AFL% store(build(left_array,1),left_array);
   ```

2. Create a 2×3 array right_array containing value 0 in all cells:

   ```
   AFL% create array right_array <val:double>[i=0:1,6,0,j=0:2,3,0];
   ```

   ```
   AFL% store(build(right_array,0),right_array);
   ```

3. Concatenate left_array and right_array:

   ```
   AFL% create array concat_array <val:double>[i=0:5,6,0,j=0:2,3,0];
   ```

   ```
   AFL% store(concat(left_array,right_array),concat_array);
   ```

   This produces an array concat_array with contents and schema as follows:

   ```
   [("concat_array<val:double> [i=0:5,6,0,j=0:2,3,0]")]
   ```

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

Name

count — Returns a count of non-empty cells.

Synopsis

```
count(array[ ,dimension_1,dimension_2,...])
```

Summary

The count operator counts nonempty 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. When dimensions are provided, they are used to do a group-by, and a count per resulting group is returned.

Note

The count aggregate provides similar functionality to the count operator. For details, see the count aggregate reference.

Example

This example finds the element count value along the first and second dimension of a 4×4 array where some cells are empty.

1. Create an array named source_array:

```
AFL% CREATE ARRAY source_array<val:double>[i=0:3,4,0,j=0:3,4,0];
```

2. Store values of 0–15 in source_array:

```
AFL% store(build(source_array,i*4+j),source_array);
```

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

3. Use between to create some empty cells in source_array and store the result in count_array:

```
AFL% store(between(source_array,1,1,1,2),count_array);
```

```
[
[(),(),(),()],
[(),(5),(6),()],
[(),(),(),()],
[(),(),(),()]
]
```

4. Find the count of nonempty elements in count_array:

```
AFL% count(count_array);
```

```
[(2)]
```

5. Count the nonempty elements along the dimensions of count_array:

```
AFL% count(count_array,i);
```

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

```
AFL% count(count_array,j);
```

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

6. If we specify both dimensions, the operator returns an array of the original size, with a single value that represents whether or not each cell is empty:

```
AFL% count(count_array,i,j);
```

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

Name

cross — Produces a result array whose set of cells consists of every possible combination of a cell from one source array and a cell from another source array.

Synopsis

```
cross(left_array,right_array);
```

Summary

The cross operator calculates the full cross product of two arrays. If the left array has m dimensions and the right array has n dimensions, the result array will have m + n dimensions. If the left array has x non-empty cells and the right array has y non-empty cells, the result array will have x × y non-empty cells.

For example, consider a 2-dimensional array A with dimensions i, j, and a 1-dimensional array B with dimension k. Within the result array produced by cross(A,B), the cell at coordinate position {i, j, k} is the concatenation of cell {i, j} of A with cell {k} of B.

If the two source arrays have a pair of like-named variables (i.e., attributes or dimensions), the result array will include one variable with that name and another variable with that name plus the suffix "_2."

If a cell of one source array is empty, all associated cells in the result array will be empty.

Examples

This example returns the cross product of two 1-dimensional arrays, each of which includes only one attribute.

1. Show the first array:

```
AFL% scan(letters);
```

```
{i} letter
{0} "A"
{1} "B"
{2} "C"
```

2. Show the second array:

```
AFL% scan(numbers);
```

```
{i} number
{0} 0
{1} 1
{2} 2
{3} 3
{4} 4
```

3. Generate the cross product:

```
AFL% cross(numbers,letters);
```

```
[
[(0,"A"),(0,"B"),(0,"C")],
```

```
[(1,"A"),(1,"B"),(1,"C")],
[(2,"A"),(2,"B"),(2,"C")],
[(3,"A"),(3,"B"),(3,"C")],
[(4,"A"),(4,"B"),(4,"C")]
]
```

This example returns the cross product of two 1-dimensional arrays, one of which has multiple attributes.

1. Show the first array:

```
AFL% scan(lettersPlus);
```

```
{i} letter,type
{0} "A","vowel"
{1} "B","consonant"
{2} "C","consonant"
```

2. Generate the cross product with the array called "numbers":

```
AFL% cross(numbers,lettersPlus);
```

```
[
[(0,"A","vowel"),(0,"B","consonant"),(0,"C","consonant")],
[(1,"A","vowel"),(1,"B","consonant"),(1,"C","consonant")],
[(2,"A","vowel"),(2,"B","consonant"),(2,"C","consonant")],
[(3,"A","vowel"),(3,"B","consonant"),(3,"C","consonant")],
[(4,"A","vowel"),(4,"B","consonant"),(4,"C","consonant")]
]
```

This example returns the cross product of two 1-dimensional arrays, one of which includes some null-valued attributes.

1. Show the array containing some null values (where the even numbers would be):

```
AFL% scan(oddNumbers);
```

```
{i} number
{0} null
{1} 1
{2} null
{3} 3
{4} null
```

2. Generate the cross product with the letters array:

```
AFL% cross(oddNumbers,letters);
```

```
[
[(null,"A"),(null,"B"),(null,"C")],
[(1,"A"),(1,"B"),(1,"C")],
[(null,"A"),(null,"B"),(null,"C")],
[(3,"A"),(3,"B"),(3,"C")],
[(null,"A"),(null,"B"),(null,"C")]
]
```

This example returns the cross product of two 1-dimensional arrays, one of which includes some empty cells.

1. Show the array containing some empty cells (where the odd numbers would be):

```
AFL% scan(evenNumbers);
```

```
{i} number
{0} 0
{2} 2
{4} 4
```

2. Generate the cross product with the letters array:

```
AFL% cross(evenNumbers,letters);
```

```
[
[(0,"A"),(0,"B"),(0,"C")],
[(),(),()],
[(2,"A"),(2,"B"),(2,"C")],
[(),(),()],
[(4,"A"),(4,"B"),(4,"C")]
]
```

Name

cross_join — Cross-product join with equality predicates

Synopsis

```
cross_join(left_array,right_array,left_dim1,right_dim1,...);
```

Summary

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.

If the join dimensions are different lengths, the cross-join will return the smaller dimension for the join points.

Note

The `cross_join` operator is sensitive to the order of arguments: always pass the larger array as the first argument.

Example

This example returns the cross-join of a 3×3 array with a vector of length 3.

1. Create an array called left_array:

   ```
   AFL% CREATE ARRAY left_array<val:double>[i=0:2,3,0, j=0:2,3,0];
   ```

2. Store values of 0–8 into left array:

   ```
   AFL% store(build(left_array,i*3+j),left_array);

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

3. Create an array called right_array:

   ```
   AFL% CREATE ARRAY right_array<val:double>[k=0:5,3,0];
   ```

4. Store values of 101–106 into right_array:

   ```
   AFL% store(build(right_array,k+101),right_array);

   [(101),(102),(103),(104),(105),(106)]
   ```

5. Perform a cross-join on left_array and right_array along dimension j of left_array:

```
AFL% cross_join(left_array,right_array,j,k);
```

```
[
[(0,101),(1,102),(2,103)],
[(3,101),(4,102),(5,103)],
[(6,101),(7,102),(8,103)]
]
```

Name

deldim — Produces a result array with one fewer dimension than a given source array.

Synopsis

```
deldim(array);
```

Summary

The deldim operator deletes the left-most dimension from the array. The to-be-deleted dimension must have size = 1. See the adddim operator reference for details of creating a dimension of size = 1.

You can use the deldim operator in the FROM clause of an AQL SELECT statement, as a stand-alone operator in a AFL statement, or as an operand within other SciDB operators.

Name

dimensions — List array dimensions

Synopsis

```
dimensions(named_array);
```

Summary

The argument to the dimensions operator is the name of an array. It returns an array with one row per dimension, and the following attributes for each dimension:

- name
- start index
- length
- chunk size
- chunk overlap
- low boundary index
- high boundary index
- data type

The argument *named_array* must be an array that was previously created and stored in the SciDB namespace.

Example

This example creates an array with one integer dimension and one string-type dimension:

```
AFL% CREATE ARRAY array1
 <val:double>[i=0:2000,10,0,j(string)=10,10,0];
```

```
AFL% dimensions(array1);
```

```
{No} name,start,length,chunk_interval,chunk_overlap,low,high,type
{0} "i",0,2001,10,0,4611686018427387903,-4611686018427387903,"int64"
{1} "j",0,10,10,0,4611686018427387903,-4611686018427387903,"string"
```

Name

filter — Produces a result array, filtering out elements based on a supplied boolean expression

Synopsis

```
filter(array,expression);
```

Summary

The filter operator filters out data in an array based on an expression over the attribute and dimension values. The filter operator returns an array the with the same schema as the input array but marks all cells in the input that do not satisfy the predicate expression 'empty'.

Example

This example filters an array to remove outlying values.

1. Create a 4×4 array:

   ```
   AFL% CREATE ARRAY m4x4<val:double>[i=0:3,4,0,j=0:3,4,0];
   ```

2. Put values between 0 and 15 into the non-diagonal elements of m4x4 and values greater than 100 into the diagonal elements:

   ```
   AFL% store(build(m4x4,iif(i=j,100+i,i*4+j)),m4x4);
   ```

   ```
   [
   [(100),(1),(2),(3)],
   [(4),(101),(6),(7)],
   [(8),(9),(102),(11)],
   [(12),(13),(14),(103)]
   ]
   ```

3. Filter all values of 100 or greater out of m4x4:

   ```
   AFL% filter(m4x4,val<100);
   ```

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

Name

gemm — Produces ***AB*** + ***C***, given three matrix arrays ***A***, ***B***, and ***C***.

Synopsis

```
gemm(A_matrix, B_matrix, C_matrix);
```

Summary

The gemm operator multiplies matrices ***A*** and ***B*** then adds ***C***. Matrices ***A*** and ***B*** must must be compatible for matrix multiplication and addition.

Given that ***A*** has dimensions $m \times n$, then ***B*** must have dimensions $n \times p$. That is, the number of columns of ***A*** must equal the number of rows of ***B***. ***C*** must have dimensions $m \times p$. The result has the same dimensions as ***C***.

For more information, see the following web pages:

- Matrix multiplication in general: http://en.wikipedia.org/wiki/Matrix_multiplication
- GEMM: http://en.wikipedia.org/wiki/General_Matrix_Multiply

To use the `gemm` operator, you must first load the `dense_linear_algebra` library.

Limitations

- The first attribute of all three arrays must be of type `double`. All other attributes are ignored.
- Each dimension of each matrix must have the following characteristics:
  - Currently, the starting index must be zero.
  - The ending index cannot be '*'.
  - Currently, the chunk size must be 32.
  - Currently, the chunk overlap must be zero.
- Currently, `gemm` only accepts square matrices as arguments, that is m≡n≡p.
- Currently, `gemm` is implemented only on the Ubuntu operating system.

Example

This example creates three matrices and calculates the gemm.

1. Create matrices ***A***, ***B***, and ***C***.

```
AFL% CREATE ARRAY A <val:double> [i=0:1,32,0,j=0:1,32,0];

AFL% store(build (A, i*2 + j + 1), A);

[
```

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

```
AFL%
 store(build(<val:double>[i=0:1,32,0,j=0:1,32,0],iif(i=j,1,0)),B);
```

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

```
AFL% CREATE ARRAY C <val:double> [i=0:1,32,0,j=0:1,32,0];
```

```
AFL% store(build (C, 1), C);
```

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

2. Perform the gemm calculation.

```
AFL% gemm(A,B,C);
```

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

Name

gesvd — Produces a result matrix containing any one of the three components of the singular value decomposition of a general matrix.

Synopsis

```
gesvd(input_matrix, factor);
```

Summary

For a matrix *M*, GESVD returns the matrix factorization:

$$M = U\Sigma V^*$$

The `gesvd` operator produces a singular value decomposition (SVD) of the input matrix and returns one of the three factors. For more details of the SVD, see http://en.wikipedia.org/wiki/Singular_value_decomposition.

The *`factor`* must be one of the following values:

- `'U'` (or `'left'`): the matrix of left-singular vectors.
- `'S'` (or `'values'`): a vector that contains the singular values in decreasing numerical order.
- `'VT'` (or `'right'`): the transpose of the matrix of right-singular vectors.

If the input matrix is an $m \times n$ matrix, and letting **MIN** = min(*m*,*n*), then dim(***U***) is $m \times$ **MIN**, dim(***S***) is **MIN**, and dim(***VT***) is **MIN** $\times n$.

To use the `gesvd` operator, you must first load the `dense_linear_algebra` library.

Limitations

The input matrix must have the following characteristics:

- The first attribute must be of type `double`. All other attributes are ignored.
- Each dimension must have the following characteristics:
  - Currently, the starting index must be zero.
  - The ending index cannot be '*'.
  - Currently, the chunk size must be 32.
  - Currently, the chunk overlap must be zero.
- Currently, `gesvd` is implemented only on the Ubuntu operating system.

Example

This example creates a matrix and calculates its singular value decomposition.

1. Construct a rotation matrix, A, that rotates by π/6.

```
AFL% store(build(<val:double>[i=0:1,32,0,j=0:1,32,0],
    iif(i=j,sqrt(3)/2, iif(i=1,0.5,-0.5))),A);
```

2. Construct a scaling matrix, B, that distorts by a factor of 2.

```
AFL% store(build(<val:double>[i=0:1,32,0,j=0:1,32,0],iif(i!=j, 0,
 iif(i=0,2,1))),B);
```

3. Construct a rotation matrix, C, that rotates by $-\pi/6$.

```
AFL% store(build(<val:double>[i=0:1,32,0,j=0:1,32,0],
    iif(i=j,sqrt(3)/2, iif(i=1,-0.5,0.5))),C);
```

4. Multiply the matrices together. The product becomes the input to the `gesvd` operator.

```
AFL% store(multiply(multiply(A,B),C),product);
```

5. Calculate the U, S, and VT decompositions.

```
AFL% gesvd(product,'U');
```

```
AFL% gesvd(product,'S');
```

```
AFL% gesvd(product,'VT');
```

Note that the factors match the matrices that we used to construct the input to `gesvd`, except for the signs of some of the matrix values. This happens because SVD in only unique up to the sign of the matrix entries.

Name

help — Operator signature

Synopsis

```
help('operator_name');
```

Summary

Accepts an operator name and returns an array containing a human-readable signature for that operator.

Example

This example returns the signature of the multiply operator.

```
help('multiply');
```

```
[("Operator: multiply
Usage: multiply(<input>, <input>, ...)")]
```

Name

insert — Insert values from a source array into a target array.

Synopsis

```
insert(array,named_array);
```

Summary

The insert operator has two effects. One effect is to update the target array by inserting values from the source array. This means that the `insert` operator is a write operator, like the `store` and `redimension_store` operators.

The other effect is to produce a result array identical to the updated target array. This means that you can use the insert operator as an operand of other SciDB operators, although using the `insert` operator inside any other write operator (such as `store` or `redimension_store`) can yield unpredictable results.

The source array and target array must be compatible. For the insert operator, compatible means the following:

- The source and target arrays must have the same number of attributes.

  Note: To the insert operator, attribute names are immaterial; the attribute names in the source and target arrays need not match. Rather, the first attribute of the source corresponds to the first attribute of the target; the second to the second, and so on.

- In the left-to-right ordering of attributes in each array, each pair of corresponding attributes must have the same datatype and the same null/not null setting.

- The source and target arrays must have the same number of dimensions.

  Note: Here too, the names are immaterial. Source dimensions and target dimensions correspond based on the left-to-right order of dimensions.

- In the left-to-right ordering of dimensions in each array, each pair of corresponding dimensions must have the same chunk size, chunk overlap, and dimension starting index.

For each cell location of the target array, the insert operator writes values according to the following rules:

- If the corresponding cell location of the source array is empty, the insert operator does not write anything. At that cell location of the target array, an empty cell would remain empty, null values would remain null, and other values would remain unchanged.

- If the corresponding cell location of the source array is non-empty, the insert operator changes the corresponding cell of the target array to match the value of the source. This has the following effects:

  - Null values in the source can overwrite non-null values in the target.

  - If the cell location of the target array was initially empty, it will be non-empty after the insert operation.

Note

The AFL insert operator provides the same functionality as the AQL **`INSERT INTO`** statement.

Limitations

- In both the source and the target array, each dimension must have datatype int64.
- For each corresponding pair of dimensions in the source and target:
  - If the target dimension is bounded, the source dimension must also be bounded and the respective upper bounds must be equal.
  - If the target dimension is unbounded, the source dimension can be unbounded.
  - If the target dimension is unbounded and the source dimension is bounded, the source dimension's chunk size must divide the source dimension's size evenly.

Examples

These examples show the behavior of the insert() operator.

1. Show array A. Note that row 1 has only empty cells, row 2 has only null values, and row 3 has only non-null values.

```
AQL% SELECT * FROM A
```

```
[
[(),(),()],
[(null),(null),(null)],
[("a7"),("a8"),("a9")]
]
```

2. Show array B. Note that column 1 has only empty cells, column 2 has only null values, and column 3 has only non-null values.

```
AQL% SELECT * FROM B
```

```
[
[(),(null),("b3")],
[(),(null),("b6")],
[(),(null),("b9")]
]
```

3. Insert values from A into B.

```
AFL% insert(A,B)
```

```
[
[(),(null),("b3")],
[(null),(null),(null)],
[("a7"),("a8"),("a9")]
]
```

4. Show the versions of B. Version 1 is the original version of B. Version 2 is the version resulting from the insert operation of the previous step.

```
AQL% SELECT * FROM versions(B)
```

```
{VersionNo} version_id,timestamp
{1} 1,"2012-12-17 18:39:09"
{2} 2,"2012-12-17 18:39:09"
```

5. Insert values from B (the original version of B—unchanged by the previous insert operation) into A

```
AFL% insert (B@1, A)
```

```
[
[(),(null),("b3")],
[(null),(null),("b6")],
[("a7"),(null),("b9")]
]
```

Name

join — Join two arrays

Synopsis

```
join(left_array,right_array);
```

Summary

Join combines the attributes of two input arrays at matching dimension values. The two arrays must have the same dimension start coordinates, the same chunk size, and the same chunk overlap. The join result has the same dimension names as the first input. If the the left-hand and right-hand arrays do not have the same dimension size, join will return an array with the same dimensions as the smaller input array. If a cell in either the left or right array is empty, the corresponding cell in the result is also empty.

Example

This example joins two arrays with different dimension lengths.

1. Create a 3×3 array left_array containing value 1 in all cells:

```
AFL% create array left_array <val:double>[i=0:2,3,0,j=0:2,3,0];
```

```
AFL% store(build(left_array,1),left_array);
```

2. Create a 3×6 array right_array containing value 0 in all cells:

```
AFL% create array right_array <val:double>[i=0:2,3,0,j=0:5,3,0];
```

```
AFL% store(build(right_array,0),right_array);
```

3. Join left_array and right_array:

```
AFL% store(join(left_array,right_array),result_array);
```

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

```
AFL% show(result_array)
```

```
result_array

< val:double,
val_2:double >

[i=0:2,3,0,
j=0:2,3,0]
```

Name

list — List contents of SciDB namespace

Synopsis

```
list(element[,version_flag])
```

Summary

The list operator allows you to get a list of elements in the current SciDB instance. The input is one of the following strings:

aggregates	Show all operators that take as input a SciDB array and return a scalar.
arrays	Show all arrays. If *version_flag* is true, the operator lists all versions of each array. Thus, if an array has n versions, the output will include n+1 rows for that array.
functions	Show all functions and the libraries in which they reside.
instances	Show all SciDB instances. Each instance will be listed with its port, id number, and time and date stamps for when it came online.
libraries	Show all libraries that are loaded in the current SciDB session.
operators	Show all operators and the libraries in which they reside.
types	Show all the datatypes the SciDB supports.
queries	Show all active queries. Each active query will have an id, a time and date when it was queries initiated, an error code, whether it generated any errors, and a status (boolean flag where TRUE means that the query is idle).

Example

This example shows sample output from the `list` operator.

1. List all arrays:

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

```
{No} name,id,schema,availability
{0} "C",3221,"C<val:double> [i=0:1,32,0,j=0:1,32,0]",true
{1} "product",3223,"product<val:double>
 [i=0:1,32,0,j=0:1,32,0]",true
{2} "vector0",3440,"vector0<val:double> [i=0:4,5,0]",true
{3} "vector1",3442,"vector1<val:string> [i=0:4,5,0]",true
```

2. List all versions of all arrays:

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

```
{No} name,id,schema,availability
{0} "C",3221,"C<val:double> [i=0:1,32,0,j=0:1,32,0]",true
{1} "C@1",3222,"C@1<val:double> [i=0:1,32,0,j=0:1,32,0]",true
{2} "product",3223,"product<val:double>
 [i=0:1,32,0,j=0:1,32,0]",true
```

```
{3} "product@1",3224,"product@1<val:double>
 [i=0:1,32,0,j=0:1,32,0]",true
{4} "vector0",3440,"vector0<val:double> [i=0:4,5,0]",true
{5} "vector0@1",3441,"vector0@1<val:double> [i=0:4,5,0]",true
{6} "vector1",3442,"vector1<val:string> [i=0:4,5,0]",true
{7} "vector1@1",3443,"vector1@1<val:string> [i=0:4,5,0]",true
```

3. List all SciDB aggregates:

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

```
{No} name
{0} "ApproxDC"
{1} "avg"
{2} "count"
{3} "first"
{4} "last"
{5} "mad"
{6} "max"
{7} "median"
{8} "min"
{9} "stdev"
{10} "sum"
{11} "top_five"
{12} "var"
```

Name

load — Load data to an array from a file.

Synopsis

```
load(output_array,input_file, instance_id, format[,[max_errors,
 shadow_array]])
```

Summary

The AFL load operator loads data from *input_file* into the cells of a SciDB array, *output_array*. The load operator takes the following arguments:

- *output_array* — The name of a SciDB array to hold the data.
- *input_file* — The complete path to the file that contains the source data for the array.
- *instance_id* — Specifies the instance or instances for performing the load. The value must be one of the following:
  - **-2**: Load all data using the coordinator instance of the query.
  - **-1**: Initiate the load from all instances. That is, the load is distributed to all instances, and the data is loaded concurrently.
  - **0**, **1**, ...: Load all data using the specified instance ID.
- *format* — A string that specifies the format for the incoming data. The LOAD command uses the format string as a guide for interpreting the contents of the binary file.
- *max_errors* — An optional parameter to specify the limit of errors before the operator will fail. The default value is 0, meaning that if any errors are encountered, the operation will fail.
- *shadow_array* — An optional parameter for specifying a "shadow array." This is a mechanism by which you can keep track of errors while still loading the error-free values.

Example

This example loads data from a binary file.

1. Create an array to hold the array data:

```
AFL% CREATE ARRAY intensityFlat
   < exposure:string, elapsedTime:int64,
   measuredIntensity:int64 null > [i=0:*,5,0];
```

2. Load the data, specifying a maximum of 99 errors, and saving information to a shadow array:

```
AFL% load(intensityFlat,'../examples/intensity_data.bin',-1,
   '(string, int64, int64 null)',99,shadowArray);
```

```
{i} exposure,elapsedTime,measuredIntensity
{0} "High",0,100
{1} "High",1,100
```

```
{2} "High",2,99
{3} "High",3,99
{4} "High",4,98
{5} "High",0,100
{6} "High",1,100
{7} "High",2,99
{8} "High",3,99
{9} "High",4,98
{10} "High",0,100
{11} "High",1,100
{12} "High",2,99
{13} "High",3,99
{14} "High",4,98
{15} "High",5,97
{16} "High",6,null
{17} "High",7,null
{18} "Medium",0,100
{19} "Medium",1,95
{20} "High",5,97
{21} "High",6,null
{22} "High",7,null
{23} "Medium",0,100
{24} "Medium",1,95
{25} "High",5,97
{26} "High",6,null
{27} "High",7,null
{28} "Medium",0,100
{29} "Medium",1,95
{30} "Medium",2,89
{31} "Medium",3,null
{32} "Medium",4,null
{33} "Medium",5,80
{34} "Medium",6,78
{35} "Medium",2,89
{36} "Medium",3,null
{37} "Medium",4,null
{38} "Medium",5,80
{39} "Medium",6,78
{40} "Medium",2,89
{41} "Medium",3,null
{42} "Medium",4,null
{43} "Medium",5,80
{44} "Medium",6,78
{45} "Medium",7,77
{46} "Low",0,100
{47} "Low",1,85
{48} "Low",2,71
{49} "Low",3,60
{50} "Medium",7,77
{51} "Low",0,100
{52} "Low",1,85
{53} "Low",2,71
{54} "Low",3,60
{55} "Medium",7,77
```

```
{56} "Low",0,100
{57} "Low",1,85
{58} "Low",2,71
{59} "Low",3,60
{60} "Low",4,50
{61} "Low",5,41
{62} "Low",6,35
{63} "Low",7,29
{65} "Low",4,50
{66} "Low",5,41
{67} "Low",6,35
{68} "Low",7,29
{70} "Low",4,50
{71} "Low",5,41
{72} "Low",6,35
{73} "Low",7,29
```

3. Now, let's see what happens if the measuredIntensity attribute does not allow nulls:

```
AFL% remove(intensityFlat);
```

```
AFL% CREATE ARRAY intensityFlat
< exposure:string, elapsedTime:int64, measuredIntensity:int64 >
[i=0:*,5,0];
```

```
AFL% load(intensityFlat,'../examples/intensity_data.bin',-1,
   '(string, int64, int64)',99,shadowArray);
```

```
{i} exposure,elapsedTime,measuredIntensity
{0} "High",0,25855
{1} "",0,0
{5} "High",0,25855
{6} "",0,0
{10} "High",0,25855
{11} "",0,0
```

4. View the data in shadowArray to get details on why the file did not load correctly:

```
AFL% scan(shadowArray)
```

```
{i} exposure,elapsedTime,measuredIntensity,row_offset
{1} "Failed to read file: 0","Failed to read file: 0","Failed to
 read file: 0",25
{6} "Failed to read file: 0","Failed to read file: 0","Failed to
 read file: 0",25
{11} "Failed to read file: 0","Failed to read file: 0","Failed to
 read file: 0",25
```

Name

load_library — Load a plugin

Synopsis

```
load_library(library_name);
```

Summary

Load a SciDB plugin. 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.

To unload a library, see the unload_library operator reference.

Example

```
load_library('librational')
```

Name

lookup — Select array cells by dimension index

Synopsis

```
lookup(pattern_array,source_array);
```

Summary

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 *pattern_array* and the same attributes as *source_array*.

Example

This example selects a row from a 2-dimensional array.

1. Create an vector of ones called indices1:

   ```
   AFL% store(build(<val1:double>[i=0:3,4,0],1),indices1);

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

2. Create a vector with values between 0 and 3 called indices2:

   ```
   AFL% store(build(<val1:double>[i=0:3,4,0],i),indices2);

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

3. Join indices1 and indices2 into a two-attribute array called pattern_array:

   ```
   AFL% store(join(indices1,indices2),pattern_array);

   {i} val1,val1_2
   {0} 1,0
   {1} 1,1
   {2} 1,2
   {3} 1,3
   ```

4. Create a 2-dimensional array called source_array with values between 100 and 115:

   ```
   AFL% store(build(<val:double>[i=0:3,4,0,j=0:3,4,0],i*4+j
   +100),source_array);


   [
   [(100),(101),(102),(103)],
   [(104),(105),(106),(107)],
   [(108),(109),(110),(111)],
   [(112),(113),(114),(115)]
   ]
   ```

5. Use lookup to use the dimension coordinates array pattern_array to return the second row of source_array:

```
AFL% lookup(pattern_array,source_array);

[(104),(105),(106),(107)]
```

Name

max — Select maximum value

Synopsis

```
max(array,attribute[,dimension_1,dimension_2,...])
```

Summary

The max operator calculates the maximum of the specified attribute in the array. Result is an array with single element containing the maximum of the specified attribute.

Note

The max operator provides the same functionality as the max aggregate. For details, see the max aggregate reference.

Example

This example find the maximum value of each row of a 2-dimensional array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

```
AFL% CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
```

2. Store values of 0–8 in m3x3:

```
AFL% store(build(m3x3,i*3+j),m3x3);
```

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

3. Select the maximum value of each row of m3x3:

```
AFL% max(m3x3,val,i);
```

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

Name

merge — Produces a result array by merging the data from two other arrays.

Synopsis

```
merge(left_array,right_array);
```

Summary

The *left_array* and *right_array* must be compatible. For the merge operator, compatible means the following:

- The arrays must have the same number of attributes.

  Note: To the merge operator, attribute names are immaterial; the attribute names in the *left_array* and *right_array* need not match. Rather, the first attribute of the *left_array* corresponds to the first attribute of the *right_array*; the second to the second, and so on.

- In the ordering of attributes in each array, each pair of corresponding attributes must have the same datatype and the same null/not null setting.

- The *left_array* and *right_array* must have the same number of dimensions.

  Note: Here too, the names are immaterial. Dimensions of the operand arrays correspond based on the left-to-right order of dimensions.

- In the ordering of dimensions in each array, each pair of corresponding dimensions must have the same chunk size, chunk overlap, and dimension starting index.

Merge combines elements from the input arrays the following way: for each cell in the two inputs, if the cell of first (left) array is not 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.

If the dimensions are not the same size, merge will return an output array the same size as the larger input array.

Limitations

In both *left_array* and *right_array*, each dimension must have datatype int64.

Example

This example merges two sparse arrays.

1. Create a sparse array left_array and store value 1 in the first row:

   ```
   AFL% CREATE ARRAY left_array <val:double>[i=0:2,3,0,j=0:5,6,0];
   ```

   ```
   AFL% store(build_sparse(left_array,1,i=0),left_array);
   ```

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

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

2. Create a sparse identity matrix called right_array

```
AFL% CREATE ARRAY right_array <val:double>[i=0:2,3,0,j=0:2,6,0];
```

```
AFL% store(build_sparse(right_array,1,i=j),right_array);
```

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

3. Merge left_array and right_array:

```
AFL% merge(left_array,right_array);
```

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

Name

min — Select minimum value

Synopsis

```
min(array,attribute[,dimension_1,dimension_2,...])
```

Summary

The min operator selects the minimum value from an array attribute.

Note

The min operator provides the same functionality as the min aggregate. For details, see the min aggregate reference.

Example

This example finds the minimum value of each row of a 2-dimensional array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

```
AFL% CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
```

2. Store random values in m3x3:

```
AFL% store(build(m3x3,random()%20+1),m3x3);
```

```
[
[(11),(7),(11)],
[(9),(1),(19)],
[(7),(4),(20)]
]
```

3. Select the minimum value of each row of m3x3:

```
AFL% min(m3x3,val,i);
```

```
[(7),(1),(4)]
```

Continuing on with this example, we can retrieve the minimum value, along with its position in the array, by using the following query:

```
AFL% project(filter(deldim(cross_join(aggregate(m3x3,min(val) as
 m),m3x3)),val=m),val);
```

This query does the following:

1. Find the minimum value, and alias it so that we can use it in a cross join:

```
AFL% aggregate(m3x3,min(val) as m)
```

```
{i} m
```

```
{0} 1
```

2. Cross join the minimum value with the original array, so that cell {i,j}x becomes {i,i,j}(m=min,x):

```
AFL% cross_join(aggregate(m3x3,min(val) as m),m3x3)
```

```
{i,i,j} m,val
{0,0,0} 1,11
{0,0,1} 1,7
{0,0,2} 1,11
{0,1,0} 1,9
{0,1,1} 1,1
{0,1,2} 1,19
{0,2,0} 1,7
{0,2,1} 1,4
{0,2,2} 1,20
```

3. Remove the extra dimension added in the previous step:

```
AFL% deldim(cross_join(aggregate(m3x3,min(val) as m),m3x3))
```

```
{i,j} m,val
{0,0} 1,11
{0,1} 1,7
{0,2} 1,11
{1,0} 1,9
{1,1} 1,1
{1,2} 1,19
{2,0} 1,7
{2,1} 1,4
{2,2} 1,20
```

4. Filter (select) the cells that have the actual minimum value:

```
AFL% filter(deldim(cross_join(aggregate(m3x3,min(val) as
 m),m3x3)), val=m);
```

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

5. Of the correct cells, show only `val`, not the alias, `m`:

```
AFL% project(filter(deldim(cross_join(aggregate(m3x3,min(val) as
 m),m3x3)), val=m),val);
```

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

Name

multiply — Produces a result array via matrix multiplication of two input arrays

Synopsis

```
AFL%  multiply(left_array,right_array);
```

Summary

The multiply operator performs matrix multiplication on two input matrices and returns a result matrix.

Both inputs must be compatible for the multiply operation, and both must have a single numeric attribute.

If m, n, and p represent integers, to be compatible for multiplication, two matrices must have dimensions as follows: if *left_array* has dimensions $m \times n$, then *right_array* must have dimensions $n \times p$, and the result array will have dimensions $m \times p$.

Example

This example multiplies a 3×2 array and a 2×3 array.

1. Create a 3×2 array:

```
AFL% store(build(<val:double>[i=0:2,3,0,j=0:1,2,0],(i
+1)*3+j),left);
```

```
[
[(3),(4)],
[(6),(7)],
[(9),(10)]
]
```

2. Create a 2×3 array:

```
AFL% store(build(<val:double>[i=0:1,2,0,j=0:2,3,0],(i+1)*3-
j),right);
```

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

3. Multiply left and right.

```
AFL% multiply(left, right)
```

```
[
[(33),(26),(19)],
[(60),(47),(34)],
[(87),(68),(49)]
]
```

Name

normalize — Produces a result array that scales the values of a vector.

Synopsis

```
normalize(array);
```

Summary

The normalize operator divides each element of a 1-attribute vector by the square root of the sum of squares.

Limitations

The normalize operator can only take 1-dimensional, 1-attribute arrays.

Example

Scale a vector whose values are between 1 and 10.

1. Create a 1x10 array.

```
AFL% store(build(<val:double>[i=0:9,10,0],(i+1)),unscaled);
```

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

2. Normalize the vector.

```
AFL% normalize(unscaled)
```

```
[(0.0509647),(0.101929),(0.152894),(0.203859),(0.254824),
(0.305788),(0.356753),(0.407718),(0.458682),(0.509647)]
```

Name

project — Produces a result array with the same dimensions as—but a subset of the attributes of—a source array.

Synopsis

```
project(source_array,attribute[,attribute]...);
```

Summary

The project operator produces a result array that includes some attributes of a source array but excludes others. You indicate which attributes to include by supplying their names with the attribute parameters. In the result array, the attributes will appear in the order you name them as parameters.

You can use the project operator in the FROM clause of an AQL SELECT statement, as a stand-alone operator in an AFL statement, or as an operand within other SciDB operators.

Example

This example takes a 1-dimensional array with 5 attributes, excludes one of the attributes, and reorders the remaining four.

1. Show the source_array:

```
AFL% scan(champions)
```

```
event,year,person,country,time
"dash",1996,"Bailey","Canada",9.84
"marathon",1996,"Thugwane","RSA",7956
"dash",2000,"Greene","USA",9.87
"steeplechase",2000,"Kosgei","Kenya",503.17
"marathon",2000,"Abera","Ethiopia",7811
"marathon",2008,"Wanjiru","Kenya",7596
```

2. Use the project operator to exclude the `person` attribute, and to reorder the remaining attributes so that `year` is first:

```
AFL% project(champions,year,event,country,time)
```

```
year,event,country,time
1996,"dash","Canada",9.84
1996,"marathon","RSA",7956
2000,"dash","USA",9.87
2000,"steeplechase","Kenya",503.17
2000,"marathon","Ethiopia",7811
2008,"marathon","Kenya",7596
```

Name

quantile — Quantile of an array

Synopsis

```
quantile(srcArray,q-num[,attribute[,dimension]])
```

Summary

A q-quantile is a point taken at a specified interval on a sorted data set that divides the data set into q subsets. The quantiles are the data values marking the boundaries between consecutive subsets.

You specify the source array and the number of quantiles. Optionally, you can specify an attribute and a dimension for grouping. If you want to group by a dimension, you *must* specify the attribute.

Note the following:

- The quantile operator returns *q-num*+1 values, which correspond to the lower and upper bounds for each subset.
- The quantile operator returns the same datatype as the attribute.
- The *q-num* argument must be a positive integer. Otherwise sciDB returns an error.

Examples

This example calculates the 2-quantile for a 1-dimensional array.

1. Create a 1-dimensional array called quantile_array:

```
AFL% create array quantile_array <val:int64>[i=0:10,11,0];
```

```
Query was executed successfully
```

2. Put eleven numerical values between 0 and 11 into quantile_array:

```
[(10),(3),(0),(3),(4),(5),(9),(11),(7),(3),(3)]
```

3. Find the 2-quantile of quantile_array:

```
AFL% quantile(quantile_array,2);
```

```
[(0,0),(0.5,4),(1,11)]
```

This example demonstrates the group-by-dimension parameter.

1. We start with a 5x5 array, with a single, integer attribute:

```
{i} schema
{0} "m5x5<val:int32> [i=0:4,5,0,j=0:4,5,0]"
```

```
[
[(16),(13),(22),(7),(13)],
[(11),(19),(23),(21),(24)],
[(16),(21),(15),(7),(16)],
```

```
[(10),(19),(0),(23),(23)],
[(12),(7),(18),(7),(8)]
]
```

2. Find the 2-quantile of the array, and then by the first dimension, and then by the second dimension.

```
AFL% quantile(m5x5,2);
```

```
{quantile} percentage,val_quantile
{0} 0,0
{1} 0.5,16
{2} 1,24
```

```
AFL% quantile(m5x5,2,val,i)
```

```
[
[(0,7),(0.5,13),(1,22)],
[(0,11),(0.5,21),(1,24)],
[(0,7),(0.5,16),(1,21)],
[(0,0),(0.5,19),(1,23)],
[(0,7),(0.5,8),(1,18)]
]
```

```
AFL% quantile(m5x5,2,val,j)
```

```
[
[(0,10),(0.5,12),(1,16)],
[(0,7),(0.5,19),(1,21)],
[(0,0),(0.5,18),(1,23)],
[(0,7),(0.5,7),(1,23)],
[(0,8),(0.5,16),(1,24)]
]
```

Name

rank — Rank array elements

Synopsis

```
rank(array, [attribute [, dimension1 [, dimension2... ]]
   [attribute2 [, dimension...]]])
```

```
rank(array[,attribute1,attribute2,...]
   [,dimension1,dimension2,...]);
```

Summary

Ranking array elements sorts them and assigns an ordinal rank.

The `avg_rank` operator is equivalent to `rank` except for handling ties. The `avg_rank` operator averages the rank for the tied values. For details, see the avg_rank reference.

Example

This example ranks a 4×4 array by dimension.

1. Create a 4×4 array called rank_array:

   ```
   AFL% CREATE ARRAY rank_array <val:double>[i=0:3,4,0,j=0:3,4,0];
   ```

2. Put random values of 0–6 into rank_array:

   ```
   AFL% store(build(rank_array,random()%7/1.0),rank_array);
   ```

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

3. Rank the elements in rank_array by dimension i:

   ```
   AFL% rank(rank_array,val,i);
   ```

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

4. Rank the elements in rank_array by dimension j:

   ```
   AFL% rank(rank_array,val,j);
   ```

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

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

Name

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

Synopsis

```
AFL% redimension(source_array,
                 template_array|schema_definition
                 [, aggregate (source_attribute)
                           [as result_attribute]]...)
```

Summary

The redimension operator produces a result array using data from a source array.

When you use redimension, you must decide two things:

- How will the variables of the source array appear in the result array?
- How will cell collisions be handled?

For each variable of the source array, you have these choices for how to use it in the result array:

- Include the variable without modification (i.e., an attribute remains an attribute and a dimension remains a dimension).

  To include an attribute without modification, simply include in the schema definition of the result array an attribute of the same data type with the same name. The analogous rule holds for including dimensions without modification.

- Include the variable, but convert it (from an attribute to a dimension or vice versa)

  To convert an attribute to a dimension, simply include in the schema definition for the result array a dimension with the same name and datatype as the attribute from the source array. The analogous process holds for converting dimensions to attributes.

- Exclude the variable

  To exclude a variable, simply omit it from the schema definition of the result array. Even if you exclude a variable, its data can still contribute to the result array through aggregates.

Note that a single use of the redimension operator can make all of the above kinds of modifications to the various variables in the source array.

Depending on how you arrange the source array's variables in the target array, the redimension operator might encounter cell collisions. A cell collision occurs when the redimension operator generates multiple candidate cell values for a single cell location of the target array.

If the redimension operator produces a collision at a cell location, SciDB will produce a single cell from all of the candidate cells. The attributes of that cell in the target array will be populated as follows:

- If the target attribute was declared as the value of an aggregate function, the value will be the value of that function calculated over the set of candidate cells for that cell location. For each aggregate you calculate, a nullable attribute to accommodate it must exist in the target array's schema. The attribute must have the appropriate datatype for that aggregate function.

- If the target attribute is simply the value of an attribute from the source array (rather than an aggregate function), the value of the target attribute will be from an arbitrarily chosen candidate cell for that location. If there are several such attributes in the target array, their values will all come from the same candidate cell.

The schema for the result array must accommodate the output of the redimension operator. Specifically:

- If a variable in the source array appears in the result array, the two variables must match in name and data type.
- If the redimension operator uses an aggregate, an attribute for that aggregate value must exist in the result array. The attribute must allow nulls and must be of the appropriate datatype for that aggregate function.
- The result array cannot include any other variables besides variables that appear in the source array, and attributes to accommodate aggregate values.

Limitations

- Each attribute or dimension to be changed must be of type int64.
- The dimensions in the source array must be bounded.
- Except for newly added aggregate values, the variables in the new array must be a subset of the variables in the source array.
- If a dimension of the new array corresponds to an attribute of the source array:
  - The dimension must be large enough to accommodate all distinct values of that attribute present in the source array.
  - The attribute in the source array cannot allow null values.
  - The chunk overlap property of the dimension must equal 0.
- If a dimension of the new array corresponds to a dimension in the source array:
  - The two dimensions must have identical size and identical range (i.e., max and min values).
  - The two dimensions must have identical values for chunk size and chunk overlap.
- If you use aggregates as part of the redimension operator, the destination attributes—the attributes that will contain the aggregate values—must allow null values.

Examples

This example redimensions a raw 1-dimensional array into a 2-dimensional array by transforming two of the attributes into dimensions. This example uses the data set in the file raw.csv shown here:

```
pos,device,val
1,1,1.334
1,2,1.334
1,3,1.334
1,4,1.334
1,5,1.334
2,1,2.445
2,3,2.445
2,4,2.445
```

```
2,5,2.667
3,1,0.998
3,2,1.998
3,3,1.667
3,4,2.335
4,1,2.004
4,2,2.006
4,3,2.889
4,5,2.365
5,1,2.008
5,2,2.119
5,3,2.118
5,4,2.667
5,5,2.556
```

1. Create an array named raw, to accommodate the data shown in the listing above.

```
AQL% CREATE ARRAY raw
      <pos: int64, device: int64, val: float>
      [offset=0:*,5,0];
```

2. Convert the csv file to SciDB format. You will need to exit your iquery session or do this in a new terminal window because the csv2scidb tool is run at the command line.

```
$ csv2scidb -p NNN -s 1
          < ../examples/raw.csv >
            ../examples/raw.scidb
```

3. Load the data:

```
AQL% LOAD raw FROM '../examples/raw.scidb';
```

```
{offset} pos,device,val
{0} 1,1,1.334
{1} 1,2,1.334
{2} 1,3,1.334
{3} 1,4,1.334
{4} 1,5,1.334
{5} 2,1,2.445
{6} 2,3,2.445
{7} 2,4,2.445
{8} 2,5,2.667
{9} 3,1,0.998
{10} 3,2,1.998
{11} 3,3,1.667
{12} 3,4,2.335
{13} 4,1,2.004
{14} 4,2,2.006
{15} 4,3,2.889
{16} 4,5,2.365
{17} 5,1,2.008
```

```
{18} 5,2,2.119
{19} 5,3,2.118
{20} 5,4,2.667
{21} 5,5,2.556
```

4. Create an array with dimensions device and pos to be the redimension target:

```
AQL% CREATE ARRAY A
      <val: float>
      [device=1:5,5,0, pos=1:5,5,0];
```

5. Redimension the source array raw into result array A:

```
AFL% redimension(raw, A);
```

```
{device,pos} val
{1,1} 1.334
{1,2} 2.445
{1,3} 0.998
{1,4} 2.004
{1,5} 2.008
{2,1} 1.334
{2,3} 1.998
{2,4} 2.006
{2,5} 2.119
{3,1} 1.334
{3,2} 2.445
{3,3} 1.667
{3,4} 2.889
{3,5} 2.118
{4,1} 1.334
{4,2} 2.445
{4,3} 2.335
{4,5} 2.667
{5,1} 1.334
{5,2} 2.667
{5,4} 2.365
{5,5} 2.556
```

6. Redimension the source array raw into result array A and store the result. Remember, the redimension operator produces a result array, but does not store the result.

```
AFL% store(redimension(raw, A),A);
```

This example redimensions a 2-dimensional source array into a 1-dimensional result array with aggregates. The result array has one cell for each row of the source array.

1. Show the schema of the 2-dimensional source array:

```
AFL% show(A);
```

```
{i} schema
```

```
{0} "A<val:float> [device=1:5,5,0,pos=1:5,5,0]"
```

2. Show the contents of the 2-dimensional source array:

```
AFL% scan(A);
```

```
{device,pos} val
{1,1} 1.334
{1,2} 2.445
{1,3} 0.998
{1,4} 2.004
{1,5} 2.008
{2,1} 1.334
{2,3} 1.998
{2,4} 2.006
{2,5} 2.119
{3,1} 1.334
{3,2} 2.445
{3,3} 1.667
{3,4} 2.889
{3,5} 2.118
{4,1} 1.334
{4,2} 2.445
{4,3} 2.335
{4,5} 2.667
{5,1} 1.334
{5,2} 2.667
{5,4} 2.365
{5,5} 2.556
```

3. Create an array for the 1-dimensional result; note that the attributes are declared to allow null values:

```
AQL% CREATE ARRAY Position
         <minVal:float null,
          avgVal:double null,
          maxVal:float null>
         [pos=1:5,5,0];
```

4. Use the redimension operator to produce the 1-dimensional result array that includes aggregates:

```
AFL% redimension(A,Position,
            min(val) as minVal,
            avg(val) as avgVal,
            max(val) as maxVal);
```

```
{pos} minVal,avgVal,maxVal
{1} 1.334,1.334,1.334
{2} 2.445,2.5005,2.667
{3} 0.998,1.7495,2.335
{4} 2.004,2.316,2.889
{5} 2.008,2.2936,2.667
```

Name

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

Synopsis

```
redimension_store(source_array,named_target_array
                  [, aggregate (source_attribute)
                            [as  result_attribute]]...);
```

Summary

The redimension_store operator produces a stored result array using data from a source array. Many of the considerations and details are the same as for the redimension operator, the main difference being that redimension_store saves the results to the *named_target_array*. For details, see the redimension reference documentation.

The argument *named_target_array* must be an array that was previously created in SciDB. The target array's schema must accommodate the output of the redimension_store operator. Specifically:

- If a variable in the source array appears in the target array, the two variables must match in name and data type.
- If the redimension_store operator uses an aggregate, an attribute for that aggregate value must exist in the target array. The attribute must allow nulls and must be of the appropriate datatype for that aggregate function.
- If the target array includes a synthetic dimension, the datatype of that dimension must be int64.
- The target array cannot include any other variables besides variables that appear in the source array, attributes to accommodate aggregate values, and one synthetic dimension.
- The target array can include a synthetic dimension OR a set of attributes for aggregate values, but not both. In other words, you can choose to handle collisions by calculating aggregates or by maintaining a vector with all the candidate cell values for each cell location, but not both.

If cell collisions occur, but the target array includes neither a synthetic dimension nor aggregates, SciDB will arbitrarily choose a cell from the set of candidate cells.

Example One

This example creates a two-dimension, two-attribute array and then uses the redimension_store operator to populate it with data from a one-dimension, four-attribute array.

1. Show the data in the source array.

```
{csvRow} patientID,elapsedTime,pulse,ppm
{0} 1,0,72,10
{1} 1,10,75,434
{2} 1,20,77,676
{3} 1,25,76,721
{4} 1,30,77,744
{5} 1,60,82,654
{6} 1,120,68,377
```

```
{7} 1,300,70,89
{8} 2,0,86,20
{9} 2,10,86,544
{10} 2,20,87,689
{11} 2,25,90,804
{12} 2,30,85,922
{13} 2,60,81,1067
{14} 2,120,79,866
{15} 2,300,79,645
{16} 3,0,68,17
{17} 3,10,68,333
{18} 3,20,65,444
{19} 3,25,70,606
{20} 3,30,70,673
{21} 3,60,77,624
{22} 3,120,78,508
{23} 3,300,78,212
```

2. Create the target array.

```
AQL% CREATE ARRAY doseData
      <pulse:int64,ppm:int64>
      [patientID=1:100,100,0,elapsedTime=0:999,1000,0]
```

3. Use redimension_store to populate the target array with data from the source array.

```
AFL% redimension_store(ddFlat,doseData)
```

```
{patientID,elapsedTime} pulse,ppm
{1,0} 72,10
{1,10} 75,434
{1,20} 77,676
{1,25} 76,721
{1,30} 77,744
{1,60} 82,654
{1,120} 68,377
{1,300} 70,89
{2,0} 86,20
{2,10} 86,544
{2,20} 87,689
{2,25} 90,804
{2,30} 85,922
{2,60} 81,1067
{2,120} 79,866
{2,300} 79,645
{3,0} 68,17
{3,10} 68,333
{3,20} 65,444
{3,25} 70,606
{3,30} 70,673
{3,60} 77,624
{3,120} 78,508
{3,300} 78,212
```

Example Two

This example converts the result array from the previous example from a two-dimension, two-attribute array into a three-dimension, one-attribute array. The resulting array supports queries that assess how pulse and elapsed time affect blood count (in parts per million).

1. Create the target array.

```
AQL% CREATE ARRAY ppmFunction
      <ppm:int64>
      [patientID=1:100,100,0,
       elapsedTime=0:999,1000,0,pulse=0:300,301,0]
```

2. Use redimension_store to populate the target array with data from the source array.

```
AFL% redimension_store(doseData,ppmFunction)
```

```
{patientID,elapsedTime,pulse} ppm
{1,0,72} 10
{1,10,75} 434
{1,20,77} 676
{1,25,76} 721
{1,30,77} 744
{1,60,82} 654
{1,120,68} 377
{1,300,70} 89
{2,0,86} 20
{2,10,86} 544
{2,20,87} 689
{2,25,90} 804
{2,30,85} 922
{2,60,81} 1067
{2,120,79} 866
{2,300,79} 645
{3,0,68} 17
{3,10,68} 333
{3,20,65} 444
{3,25,70} 606
{3,30,70} 673
{3,60,77} 624
{3,120,78} 508
{3,300,78} 212
```

Example Three

This example converts the result array from the previous example into a different array in which one attribute becomes a dimension and one dimension becomes an attribute. The new array supports queries that assess how blood count (in parts per million) and elapsed time affect pulse.

1. Create the target array.

```
AQL% CREATE ARRAY pulseFunction
```

```
      <pulse:int64>
      [patientID=1:100,100,0,
       elapsedTime=0:999,1000,0,
       ppm=0:10000,10001,0]
```

2. Use redimension_store to populate the target array with data from the source array.

```
AFL% redimension_store(doseData,pulseFunction)
```

```
{patientID,elapsedTime,ppm} pulse
{1,0,10} 72
{1,10,434} 75
{1,20,676} 77
{1,25,721} 76
{1,30,744} 77
{1,60,654} 82
{1,120,377} 68
{1,300,89} 70
{2,0,20} 86
{2,10,544} 86
{2,20,689} 87
{2,25,804} 90
{2,30,922} 85
{2,60,1067} 81
{2,120,866} 79
{2,300,645} 79
{3,0,17} 68
{3,10,333} 68
{3,20,444} 65
{3,25,606} 70
{3,30,673} 70
{3,60,624} 77
{3,120,508} 78
{3,300,212} 78
```

Example Four

This example shows three different ways to handle collisions.

1. Show the source array.

```
{event,year} person,country,time
{"dash",1996} "Bailey","Canada",9.84
{"dash",2000} "Greene","USA",9.87
{"dash",2004} "Gatlin","USA",9.85
{"dash",2008} "Bolt","Jamaica",9.69
{"marathon",1996} "Thugwane","RSA",7956
{"marathon",2000} "Abera","Ethiopia",7811
{"marathon",2004} "Baldini","Italy",7855
{"marathon",2008} "Wanjiru","Kenya",7596
{"steeplechase",1996} "Keter","Kenya",487.12
{"steeplechase",2000} "Kosgei","Kenya",503.17
```

```
{"steeplechase",2004} "Kemboi","Kenya",485.81
{"steeplechase",2008} "Kipruto","Kenya",490.34
```

2. Create an array to accommodate data where each non-empty cell will show a (country,year) pair of a country with at least one victory in that year, along with a count of the number of victories that country had during that year.

```
AQL% CREATE ARRAY victoryCountPerCountryYear
     <victoryCount:uint64 null>
     [year=1996:2008,13,0,country(string)=300,300,0];
```

3. Use redimension_store to populate the array with data. Note that the redimension_store operator encounters a collision (as revealed by the cell with a count greater than 1 in the result).

```
AFL% redimension_store
     (nationWinners,
      victoryCountPerCountryYear,
      count(*) as victoryCount)
```

```
{year,country} victoryCount
{1996,"Canada"} 1
{1996,"Kenya"} 1
{1996,"RSA"} 1
{2000,"Ethiopia"} 1
{2000,"Kenya"} 1
{2000,"USA"} 1
{2004,"Italy"} 1
{2004,"Kenya"} 1
{2004,"USA"} 1
{2008,"Jamaica"} 1
{2008,"Kenya"} 2
```

4. Create an array that includes a synthetic dimension (synD) for collisions. The other (non-synthetic) dimensions define (country,year) pairs. The attributes accommodate all other variables from the original array, nationWinners.

```
AQL% CREATE ARRAY vPer_CY_Synthetic
     <person:string,event:string,time:double>
     [country(string)=300,300,0,year=1996:2008,13,0,synD=1:6,6,0]
```

5. Use redimension_store to populate the array with data. Note that the redimension_store operator encounters a collision (as revealed by the cell whose value for synD is greater than 1).

```
AFL% redimension_store(nationWinners,vPer_CY_Synthetic)
```

```
{country,year,synD} person,event,time
{"Canada",1996,1} "Bailey","dash",9.84
{"Ethiopia",2000,1} "Abera","marathon",7811
{"Italy",2004,1} "Baldini","marathon",7855
{"Jamaica",2008,1} "Bolt","dash",9.69
{"Kenya",1996,1} "Keter","steeplechase",487.12
{"Kenya",2000,1} "Kosgei","steeplechase",503.17
```

```
{"Kenya",2004,1} "Kemboi","steeplechase",485.81
{"Kenya",2008,1} "Wanjiru","marathon",7596
{"Kenya",2008,2} "Kipruto","steeplechase",490.34
{"RSA",1996,1} "Thugwane","marathon",7956
{"USA",2000,1} "Greene","dash",9.87
{"USA",2004,1} "Gatlin","dash",9.85
```

6. Create an array whose dimensions define (country,year) pairs. The attributes accommodate all other variables from the original array, nationWinners. Note that the array schema includes neither a synthetic dimension nor any attribute to accommodate aggregate values.

```
AQL% CREATE ARRAY arbitraryWinner
      <person:string,event:string,time:double>
      [country(string)=300,300,0,year=1996:2008,13,0]
```

7. Use redimension_store to populate the array with data. Note that the redimension_store operator encounters a collision (for Kenya, 2008), and resolves it by arbitrarily choosing one of the Kenyan victories of that year.

```
AFL% redimension_store(nationWinners,arbitraryWinner)
```

```
{country,year} person,event,time
{"Canada",1996} "Bailey","dash",9.84
{"Ethiopia",2000} "Abera","marathon",7811
{"Italy",2004} "Baldini","marathon",7855
{"Jamaica",2008} "Bolt","dash",9.69
{"Kenya",1996} "Keter","steeplechase",487.12
{"Kenya",2000} "Kosgei","steeplechase",503.17
{"Kenya",2004} "Kemboi","steeplechase",485.81
{"Kenya",2008} "Wanjiru","marathon",7596
{"RSA",1996} "Thugwane","marathon",7956
{"USA",2000} "Greene","dash",9.87
{"USA",2004} "Gatlin","dash",9.85
```

Example Five

This example shows that a single synthetic dimension applies, even if the redimension_store operator eliminates multiple dimensions from the source array.

1. Show the source array.

```
AFL% scan(nationWinners)
```

```
{event,year} person,country,time
{"dash",1996} "Bailey","Canada",9.84
{"dash",2000} "Greene","USA",9.87
{"dash",2004} "Gatlin","USA",9.85
{"dash",2008} "Bolt","Jamaica",9.69
{"marathon",1996} "Thugwane","RSA",7956
{"marathon",2000} "Abera","Ethiopia",7811
{"marathon",2004} "Baldini","Italy",7855
{"marathon",2008} "Wanjiru","Kenya",7596
```

```
{"steeplechase",1996} "Keter","Kenya",487.12
{"steeplechase",2000} "Kosgei","Kenya",503.17
{"steeplechase",2004} "Kemboi","Kenya",485.81
{"steeplechase",2008} "Kipruto","Kenya",490.34
```

2. Create an array to accommodate data where each non-empty cell will describe a country with at least one victory in the original data, along with a count of the number of victories that country has.

```
AQL% CREATE ARRAY victoryCountPerCountry
     <victoryCount:uint64 null>
     [country(string)=300,300,0];
```

3. Use redimension_store to populate the array with data. Note that the redimension_store operator encounters collisions (as revealed by the cells with a count greater than 1 in the result).

```
AFL% redimension_store
       (nationWinners,
        victoryCountPerCountry,
        count(*) as victoryCount)
```

```
{country} victoryCount
{"Canada"} 1
{"Ethiopia"} 1
{"Italy"} 1
{"Jamaica"} 1
{"Kenya"} 5
{"RSA"} 1
{"USA"} 2
```

4. Create an array that includes a synthetic dimension (synD) for collisions. The other (non-synthetic) dimension defines countries. The attributes accommodate all other attributes from the original array.

```
AQL% CREATE ARRAY vPer_C_Synthetic
     <person:string,event:string,time:double>
     [country(string)=300,300,0,synD=1:6,6,0]
```

5. Use redimension_store to populate the array with data. Note that the redimension_store operator encounters collisions for two reasons:

   - Event is not a dimension in the target array, and one country (Kenya) has won multiple events (steeplechase and marathon)
   - Year is not a dimension in the target array, and two countries (Kenya and USA) have won in multiple years.

   Also note that despite the two reasons collisions, a single synthetic dimension suffices to accommodate all the cells that constitute the collisions.

```
AFL% redimension_store(nationWinners,vPer_C_Synthetic)
```

```
{country,synD} person,event,time
{"Canada",1} "Bailey","dash",9.84
```

```
{"Ethiopia",1} "Abera","marathon",7811
{"Italy",1} "Baldini","marathon",7855
{"Jamaica",1} "Bolt","dash",9.69
{"Kenya",1} "Wanjiru","marathon",7596
{"Kenya",2} "Keter","steeplechase",487.12
{"Kenya",3} "Kosgei","steeplechase",503.17
{"Kenya",4} "Kemboi","steeplechase",485.81
{"Kenya",5} "Kipruto","steeplechase",490.34
{"RSA",1} "Thugwane","marathon",7956
{"USA",1} "Greene","dash",9.87
{"USA",2} "Gatlin","dash",9.85
```

Example Six

This example shows a collision-handling strategy in which the target array includes some attributes with values from an arbitrarily chosen candidate cell and other attributes whose values are aggregates calculated over the set of candidate cells.

1. Show the source array.

```
{givenName,surname} a,b,c,d
{"Adam","Richards"} 77,5555,-5111,800
{"Adam","Welch"} 44,8888,-8111,300
{"Adam","Zwick"} 22,9999,-9111,100
{"Bill","Welch"} 55,7777,-7111,400
{"Bill","Zwick"} 33,null,-2111,233
{"Cathy","Welch"} 66,6666,-6111,550
```

2. Create an array to be the target of the redimension_store operator. Note that the array excludes one attribute (a) from the source array, includes two other attributes (b and c), and declares another attribute (avgD) to accommodate the value of an aggregate. Notice also that the target array excludes one of the dimensions (givenName) of the source array, so the redimension_store operator will generate collisions.

```
AQL% CREATE ARRAY namesRedimensioned
       <b:int64 null, c:int64, avgD:double null>
       [surname(string)=5,5,0]
```

3. Use redimension_store to populate the target array with data.

```
AFL% redimension_store
      (names,
       namesRedimensioned,
       avg(d) as avgD)
```

```
{surname} b,c,avgD
{"Richards"} 5555,-5111,800
{"Welch"} 8888,-8111,416.667
{"Zwick"} 9999,-9111,166.5
```

The redimension_store operation yields a target array with three cells. The first cell is not the result of a collision (because the source array included only one person with surname "Richards.") The second and third cells are the result of collisions.

Name

regrid — Select non-overlapping subarrays

Synopsis

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

Summary

The regrid operator partitions 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.

Limitations

- The grids may not span array chunks.
- The chunk size must be a multiple of the grid size in each dimension.

Example

This example divides a 4×4 array into 4 equal partitions and calculates the average of each one. This process is known as *spatial averaging*.

1. Create an array m4x4:

   ```
   AFL% CREATE ARRAY m4x4 <val:double> [i=0:3,4,0,j=0:3,4,0];
   ```

2. ```
   AFL% store(build (m4x4, i*4+j), m4x4);
   ```

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

3. Regrid m4x4 into four partitions and find the average of each partition.

   ```
   AFL% regrid(m4x4, 2,2, avg(val));
   ```

   ```
   [[(2.5),(4.5)],[(10.5),(12.5)]]
   ```
   ```

Name

remove — Removes an array and its attendant schema definition from the SciDB database.

Synopsis

```
remove(named_array);
```

Summary

The AFL remove statement works like the AQL **DROP ARRAY** statement; it deletes a named array, including all of its versions and its schema definition, from the SciDB database. The argument *named_array* must be an array that was previously created and stored in SciDB.

Note that `remove` is an AFL statement, but not an operator. Consequently, it does not produce a result array, it cannot appear in the FROM clause of an AQL SELECT statement, and it cannot appear as an operand within AFL operators.

Example

Create an array named source and then remove it:

```
AFL% store(build(<val:double>[i=0:9,10,0],1),source);

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

AFL% remove(source);

Query was executed successfully
```

Name

rename — Change array name

Synopsis

```
rename(named_array,new_array);
```

Summary

The AFL rename operator works like the AQL statement SELECT * INTO except that the old array name can be reused immediately with the rename operator.

The rename operator is akin to using the Unix `mv` (move) command, whereas SELECT * INTO is akin to the Unix `cp` (copy) command. The argument *`named_array`* must be an array that was previously created and stored in the SciDB namespace.

Example

Create an array named source, show its name and schema, rename it, and show its new name and schema. Note that the array ID remains the same.

```
AFL% store(build(<val:double>[i=0:9,10,0],1),source);

AFL% list('arrays');

name,id,schema,availability
"source",4104,"source<val:double> [i=0:9,10,0]",true

AFL% rename(source,target);

AFL% list('arrays');

name,id,schema,availability
"target",4104,"target<val:double> [i=0:9,10,0]",true
```

Name

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

Synopsis

```
AFL% repart(array,template_array|schema_definition)
```

Summary

The repart operator produces a result array similar to a source array, but with different chunk sizes, different chunk overlaps, or both. The new array must conform to the schema of an existing template array or to the schema definition supplied with the operator. The repart operator does not alter the source array.

The new array must have the same attributes and dimensions as the source array.

Example

This example repartitions a 4×4 array with 16 1x1 chunks into a 4x4 array with four 2x2 chunks.

1. Create a 2-dimensional array called source where each dimension uses a chunk size of 1:

   ```
   AFL% CREATE ARRAY source <val:double> [x=0:3,1,0,y=0:3,1,0];
   ```

2. Add values of 0–15 to source:

   ```
   AFL% store(build(source,x*3+y),source);
   ```

   ```
   {x,y} val
   {0,0} 0
   {0,1} 1
   {0,2} 2
   {0,3} 3
   {1,0} 3
   {1,1} 4
   {1,2} 5
   {1,3} 6
   {2,0} 6
   {2,1} 7
   {2,2} 8
   {2,3} 9
   {3,0} 9
   {3,1} 10
   {3,2} 11
   {3,3} 12
   ```

3. Repartition the array into 2-by-2 chunks and store the result in an array called target:

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

   ```
   {x,y} val
   {0,0} 0
   ```

```
{0,1} 1
{1,0} 3
{1,1} 4
{0,2} 2
{0,3} 3
{1,2} 5
{1,3} 6
{2,0} 6
{2,1} 7
{3,0} 9
{3,1} 10
{2,2} 8
{2,3} 9
{3,2} 11
{3,3} 12
```

Name

reshape — Produces a result array with the same cells as a given array, but a different shape.

Synopsis

```
AFL% reshape(source_array,template_array|schema_definition);
```

Summary

The reshape operator produces a result array containing the same cells as—but a different shape from—an existing array.

The new array must have the same number of attributes as the source array. The reshape operator cannot convert attributes to dimensions or vice versa. For that, use redimension or redimension_store.

The new array must have the same number of cells as the source array, but the resulting array can have more or fewer dimensions than the source.

To illustrate, let's look at a 3x4 source array. From a 3x4 source array, reshape can produce a result array of one, two, three, or even more dimensions:

- One dimension: the new array's sole dimension has size 12.
- Two dimensions: The new array can be 1x12, 2x6, 3x4, 4x3, 6x2, or 12x1.
- Three dimensions: The new array can be PxQxR, where P,Q, and R are positive integers whose product equals 12, the number of cells in the source array.
- More dimensions: The new array can have any number of dimensions, as long as the product of the dimension sizes equals the number of cells in the source array. Allowable shapes for a 12-cell source array can include 1x1x2x6 and 1x1x12x1x1.

To indicate the shape of the result array, you have two choices:

- You can refer to an existing array with the template_array parameter. The new array has the same schema as the template array. The template array is not changed by the reshape operator.
- You can declare the schema explicitly with the schema_definition parameter. The examples in this section show this technique.

Note the following:

- The reshape operator does not alter the source array
- The reshape operator does not work for a source array that has a non-zero chunk overlap.

Example

This example reshapes a 3×4 array into various other 12-cell arrays.

1. Create an array called m3x4:

   ```
   AFL% CREATE ARRAY m3x4 <val:int64>[i=0:2,3,0,j=0:3,4,0];
   ```

2. Store values of 1–12 in m3x4:

```
AFL% store(build(m3x4,i*4+j+1),m3x4);
```

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

3. Reshape m3x4 as 6x2:

```
AFL% reshape(m3x4,<val:int64>[i=0:5,6,0,j=0:1,2,0]);
```

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

4. Reshape m3x4 as 2x6:

```
AFL% reshape(m3x4,<val:int64>[i=0:1,2,0,j=0:5,6,0]);
```

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

5. Reshape m3x4 as 3x2x2:

```
AFL% reshape(m3x4,<val:int64>[p=0:2,3,0,q=0:1,2,0,r=0:1,2,0]);
```

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

6. Reshape m3x4 as 12 (a one-dimensional array of size 12):

```
AFL% reshape(m3x4,<val:int64>[p=0:11,12,0]);
```

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

7. Reshape m3x4 as 1x12 (a two-dimensional array where the size of one of the dimensions equals 1):

```
AFL% reshape(m3x4,<val:int64>[p=0:0,1,0,q=0:11,12,0]);
```

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

Name

reverse — Reverse values in each array dimension

Synopsis

```
reverse(source_array);
```

Summary

The reverse operator reverses all the values of each dimension in an array.

Example

This example reverses a 3×4 array.

1. Create a 3×4 array, m3x4:

   ```
   AFL% CREATE ARRAY m3x4<val:double>[i=0:2,3,0,j=0:3,4,0];
   ```

2. Put values of 0–11 into m3x4:

   ```
   AFL% store(build(m3x4,i*4+j),m3x4);
   ```

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

3. Reverse the values in m3x4:

   ```
   AFL% reverse(m3x4);
   ```

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

4. Reshape m3x4 into a 4x3 array, and then reverse the values.

   ```
   AFL% reverse(reshape(m3x4,<val:double>[i=0:3,4,0, j=0:2,3,0]))
   ```

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

Name

sample — Produces a result array by selecting random chunks of a source array.

Synopsis

```
sample(array,probability);
```

Summary

The sample operator selects chunks from an array at random, subject to a probability you supply.

You can use the sample operator in the FROM clause of an AQL SELECT statement, as a stand-alone operator in a AFL statement, or as an operand within other SciDB operators.

Example

This example selects random chunks from a 1-dimensional 8-chunk array.

1. Create a 1-dimensional array with dimension size of 16 and chunk size of 2:

   ```
   AFL% CREATE ARRAY vector1<val:double>[i=0:15,2,0];
   ```

2. Put values of 0–15 into vector1:

   ```
   AFL% store(build(vector1,i),vector1);

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

3. Sample chunks from the array with the probability of .30 that a chunk is included:

   ```
   AFL% sample(vector1,0.3);

   {4}[(4),(5)];{8}[(8),(9)]
   ```

Name

save — Save array data to a file

Synopsis

```
save(src_array,file_path[,instance_id[,format]])
```

Summary

The AFL save operator saves the data from the cells of a SciDB array into a file. By default, it saves the data in SciDB array format: to specify a different output format, use the *format* parameter.

- *src_array* — The source array containing the data that you want to save.
- *file_path* — The complete path to the file to hold the returned data.
- *instance_id* — An optional parameter to specify the instance for the source array data. The value must be one of the following:
  - **-2**: Save all data on the coordinator instance of the query.
  - **-1**: Save data as it is distributed; that is, each instance concurrently saves its own portion of data to file.
  - **0**, **1**, ...: Save all data to the specified instance ID.
- *format* — An optional parameter to specify the output format for the array data. For details, see Output Options. Note that you must include the *instance_id* parameter if you want to specify an output format.

Example

This example creates a a matrix with two attributes and saves the cell values to a file.

1. Create a 2-dimensional array containing values 100–108:

```
AFL% store(build(<val:double>[i=0:2,3,0,j=0:2,3,0],i*3+j
+100),array1);
```

2. Create a 2-dimensional array containing values 200–208:

```
AFL% store(build(<val:double>[i=0:2,3,0,j=0:2,3,0],i*3+j
+200),array2);
```

3. Join array1 and array2 and store the output in an array storage_array:

```
AFL% store(join(array1,array2),storage_array);
```

```
[[(100,200),(101,201),(102,202)],[(103,203),(104,204),(105,205)],
[(106,206),(107,207),(108,208)]]
```

4. Save the contents of storage_array to a file.

```
AFL% save(storage_array,'/tmp/storage_array.txt',-2,'dcsv');
```

```
{i,j} val,val_2
```

```
{0,0} 100,200
{0,1} 101,201
{0,2} 102,202
{1,0} 103,203
{1,1} 104,204
{1,2} 105,205
{2,0} 106,206
{2,1} 107,207
{2,2} 108,208
```

Name

scan — Produces a result array that is equivalent to a stored array. That is, the scan operator reads a stored array.

Synopsis

```
scan(stored_array);
```

Summary

The scan operator reads a stored array from disk. The output of the scan operator is an array the same size as *stored_array*. The argument *stored_array* must be an array that was previously created and stored in SciDB.

The scan operator is most useful for displaying a stored array on stdout from the AFL language.

Example

This example creates, builds, and stores an array, then shows the cell values in that array.

1. Create a 3×3 array m3x3:

   ```
   AFL% CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   AFL% store(build(m3x3,i*3+j),m3x3);
   ```

3. Use scan in an AFL statement to display m3x3:

   ```
   AFL% scan(m3x3);
   ```

   ```
   {i,j} val
   {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
   ```

Name

show — Produces a result array whose contents describe the schema of an array you supply.

Synopsis

```
show(named_array);
```

Summary

The show operator returns an array's schema. The argument *named_array* must be an array that was previously created and stored in SciDB.

You can use the show operator in the FROM clause of an AQL SELECT statement, as a stand-alone operator in a AFL statement, or as an operand within other SciDB operators.

Example

This example shows the schema for an array, creates and stores an abridged version of that array, then shows the schema of the abridged version.

1. Show the schema for the original array

```
AQL% SELECT * FROM show(champions);
```

```
{i} schema
{0} "champions<person:string,country:string,time:double>
 [year=1996:2008,13,0,event(string)=3,3,0]"
```

2. Use the project and store operators to create an abridged array that excludes the country and time attributes:

```
AQL% SELECT * INTO championsAbridged FROM
 project(champions,person);
```

```
{year,event} person
{1996,"dash"} "Bailey"
{1996,"marathon"} "Thugwane"
{1996,"steeplechase"} "Keter"
{2000,"dash"} "Greene"
{2000,"marathon"} "Abera"
{2000,"steeplechase"} "Kosgei"
{2004,"dash"} "Gatlin"
{2004,"marathon"} "Baldini"
{2004,"steeplechase"} "Kemboi"
{2008,"dash"} "Bolt"
{2008,"marathon"} "Wanjiru"
{2008,"steeplechase"} "Kipruto"
```

3. Show the schema for the abridged array:

```
AQL% SELECT * FROM show(championsAbridged);
```

```
{i} schema
```

```
{0} "championsAbridged<person:string>
 [year=1996:2008,13,0,event(string)=3,3,0]"
```

Name

slice — Produces a result array that is a subset of the source array derived by holding one or more dimension values constant

Synopsis

```
slice(array,dimension1,value1[dimension2,value2,...]);
```

Summary

The slice operator produces an m-dimensional result array from an n-dimensional source array where n is greater than m. If m is 3 and n is 2, the operation can be visualized as "slicing" a specific plane of a 3-D array. The number of dimensions of the result array equals the number of dimensions of the source array minus the number of (dimension, value) pairs you provide as parameters. For each (dimension, value) pair you supply, the value must appear in that dimension in the source array.

Example

This example selects the middle column from a 3×3 array.

1. Create a 3×3 array m3x3:

   ```
   AFL% CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];

   Query was executed successfully
   ```

2. Put values of 0–8 into m3x3:

   ```
   AFL% store(build(m3x3,i*3+j),m3x3);
   ```

3. Select the middle column of m3x3:

   ```
   AFL% slice(m3x3,j,1);

   {i} val
   {0} 1
   {1} 4
   {2} 7
   ```

Name

sort — Produces a 1-dimensional result array by sorting non-empty cells of a source array.

Synopsis

```
sort(array,
     attribute [ asc  |  desc ]
  [, attribute [ asc  |  desc ]]...
  [, chunk_size ] );
```

Summary

The sort operator produces a one-dimensional result array, even if the source array has multiple dimensions. The result array contains each non-empty cell of the source array. Note that the result array does not show values of the original dimensions in the source array.

The result array's sole dimension is unbounded.

The sort operator can sort by one or more attributes. The operator first sorts by the first attribute, then by the second, et cetera. Use the optional keyword `asc` or `desc` to control the sort order for each attribute, ascending or descending. The default is ascending.

You can control the chunk size of the resulting array with the optional `chunk_size` parameter.

You can use the sort operator in the FROM clause of an AQL SELECT statement, as a stand-alone operator in a AFL statement, or as an operand within other SciDB operators.

Examples

This example first scans a 2-D array, then sorts it by ascending country, then sorts it by ascending country and descending time.

1. Show the source_array:

```
AFL% scan(champions);
```

```
{year,event} person,country,time
{1996,"dash"} "Bailey","Canada",9.84
{1996,"marathon"} "Thugwane","USA",7956
{1996,"steeplechase"} "Keter","Kenya",487.12
{2000,"dash"} "Greene","USA",9.87
{2000,"marathon"} "Abera","Ethiopia",7811
{2000,"steeplechase"} "Kosgei","Kenya",503.17
{2004,"dash"} "Gatlin","USA",9.85
{2004,"marathon"} "Baldini","Italy",7855
{2004,"steeplechase"} "Kemboi","Kenya",485.81
{2008,"dash"} "Bolt","Jamaica",9.69
{2008,"marathon"} "Wanjiru","Kenya",7596
{2008,"steeplechase"} "Kipruto","Kenya",490.34
```

2. Sort by country (ascending):

```
AFL% sort(champions,country);
```

```
{n} person,country,time
{0} "Bailey","Canada",9.84
{1} "Abera","Ethiopia",7811
{2} "Baldini","Italy",7855
{3} "Bolt","Jamaica",9.69
{4} "Keter","Kenya",487.12
{5} "Kosgei","Kenya",503.17
{6} "Kemboi","Kenya",485.81
{7} "Wanjiru","Kenya",7596
{8} "Kipruto","Kenya",490.34
{9} "Thugwane","USA",7956
{10} "Greene","USA",9.87
{11} "Gatlin","USA",9.85
```

3. Sort by country (ascending), then year (descending), and use a chunk size of 100 for the result array:

```
AFL% sort(winnersAbridged,country, year desc, 100);
```

```
{n} country,year,event
{0} "Canada",1996,"dash"
{1} "Ethiopia",2000,"marathon"
{2} "Italy",2004,"marathon"
{3} "Jamaica",2008,"dash"
{4} "Kenya",2008,"steeplechase"
{5} "Kenya",2008,"marathon"
{6} "Kenya",2004,"steeplechase"
{7} "Kenya",2000,"steeplechase"
{8} "Kenya",1996,"steeplechase"
{9} "USA",2004,"dash"
{10} "USA",2000,"dash"
{11} "USA",1996,"marathon"
```

Note that the result includes 12 cells, corresponding to the 12 non-empty cells of the source array. The sort operator ignores empty cells, of which there are several in the winnersAbridged array. (They empty cells correspond to non-Olympic years included in the dimension "year.")

To illustrate how the sort operator handles null values, this example first scans an array that includes a null value, then sorts the cells in ascending order, then sorts them in descending order.

1. Show the source_array:

```
AFL% scan(numbers);
```

```
{i} number
{0} 0
{1} 1
{2} 2
{3} null
{4} 4
```

2. Sort by number (ascending):

```
AFL% sort(numbers, number asc);
```

```
{n} number
```

```
{0} null
{1} 0
{2} 1
{3} 2
{4} 4
```

3. Sort by number (descending):

```
AFL% sort(numbers, number desc);
```

```
{n} number
{0} 4
{1} 2
{2} 1
{3} 0
{4} null
```

Name

stdev — Standard deviation

Synopsis

```
stdev(array,attribute[,dimension_1,dimension_2,...]
```

Summary

The stdev operator takes a set of scalar values from an array attribute and returns the standard deviation of those values.

Note

The stdev operator provides the same functionality as the stdev aggregate. For details, see the stdev aggregate reference.

Example

This example finds the standard deviation of each row of a 2-dimensional array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

```
AFL% CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
```

2. Store values of random values between 0 and 1 in m3x3:

```
AFL% store(build(m3x3,random()%9/10.0),m3x3);
```

```
[
[(0.8),(0.4),(0.3)],
[(0.2),(0.5),(0.5)],
[(0),(0),(0.5)]
]
```

3. Select the standard deviation of each row of m3x3:

```
AFL% stdev(m3x3,val,i);
```

```
[(0.264575),(0.173205),(0.288675)]
```

Name

store — Store query output in a SciDB array

Synopsis

```
store(operator(operator_args),named_array);
```

Summary

store is a write operator, that is, one of the AFL operations 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 of its versions. The argument *named_array* must be an array that was previously created and stored in the SciDB namespace.

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).

Note

The AFL store operator provides the same functionality as the AQL **SELECT** * **INTO** ... **FROM** ... statement.

Examples

Build and store a 2-dimensional, 1-attribute matrix of zeros:

```
AFL% store(build(<val: double>[i=0:2,3,0,j=0:2,3,0],0),zeros_array);
```

You can change the name of the array zeros_array to ones_array and the cell values to 1 with a store statement:

```
AFL% store(build(zeros_array,1),ones_array);
```

Build and store a 2-dimensional, 1-attribute matrix of random numbers between 1 and 10:

```
AFL% CREATE ARRAY random_array <val:double null>[i=0:3,4,0,j=0:3,4,0];
```

```
AFL% store(build(random_array,random()%10),random_array);
```

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

You can update the array with a different set of random numbers by re-running the store statement:

```
AFL% store(build(random_array,random()%10),random_array);
```

```
[
```

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

Name

subarray — Produce a result array by selecting a contiguous area of cells.

Synopsis

```
subarray(array,low_coord1[,low_coord2,...],
              high_coord1[,high_coord2,...]);
```

Summary

The subarray operator accepts an input array and a set of coordinates specifying a region within the array. The result is an array whose shape is defined by the boundary coordinates specified by the subarray arguments.

Note the following:

- The number of coordinate pairs in the input must be equal to the number of dimensions in the array.
- The dimensions in the result array begin at 0, even if the dimensions in the input array do not.

Example

This example selects the values from the last two columns and the last two rows of a 4×4 array.

1. Create an array called m4x4:

   ```
   AFL% CREATE ARRAY m4x4 <val:double>[i=0:3,4,0,j=0:3,4,0];
   ```

2. Store values of 0–15 in m4x4:

   ```
   AFL% store(build(m4x4,i*4+j),m4x4);
   ```

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

3. Return an array containing the cells that were in both the last two columns and the last two rows on m4x4:

   ```
   AFL% subarray(m4x4,2,2,3,3);
   ```

   ```
   [
   [(10),(11)],
   [(14),(15)]
   ]
   ```

Name

substitute — Returns a result array with a specified value substituted for null values in an array.

Synopsis

```
substitute(nullable_array,substitute_array
    [,attribute_1,attribute_2,...]);
```

Summary

Substitute null values in one array with non-null values from another array. By default, all nullable attributes have their null values substituted. Optionally, you can list specific attributes to participate in the substitution.

The substitute operator renders attributes in *nullable_array* non-nullable. If an attribute has null values, you can use this operator to substitute null values in the array and change the nullability of the attribute in the schema.

Note the following limitations:

- The starting indices for both arrays must be zero.
- The substitute array must have exactly one attribute.

Example

This example replaces all null values in an array with zero, first for one attribute, and then for all attributes.

1. Create an array `m4x4_null` with two nullable attributes.

   ```
   AFL% create array m4x4_null <val1:double null, val2:double
    null>[i=0:3,4,0, j=0:3,4,0];
   ```

2. We have a 2-attribute, 4x4 array, `substitute_example` that we load into `m4x4_null`. Note that some values for each attribute are null.

   ```
   AQL% LOAD m4x4_null FROM '../examples/substitute_example.scidb'
   ```

   ```
   [
   [(1,null),(null,2.5),(),()],
   [(),(6,null),(nan,null),(null,3.14153)],
   [(7.3,0),(null,null),(),(inf,2.225)],
   [(-inf,null),(null,inf),(),(1.3,2.6)]
   ]
   ```

3. Create a single-cell array called `zeros`, and load it with the value 0.

   ```
   AFL% store(build(<subVal:double>[i=0:0,1,0],0),zeros)
   ```

   ```
   [(0)]
   ```

4. Use the substitute operator to replace the null-valued cells first in `val1` and then in `val2` with zeros.

   ```
   AFL% substitute(m4x4_null,zeros, val1);
   ```

```
[
[(1,null),(0,2.5),(),()],
[(),(6,null),(nan,null),(0,3.14153)],
[(7.3,0),(0,null),(),(inf,2.225)],
[(-inf,null),(0,inf),(),(1.3,2.6)]
]
```

```
AFL% substitute(m4x4_null,zeros, val2);
```

```
[
[(1,0),(null,2.5),(),()],
[(),(6,0),(nan,0),(null,3.14153)],
[(7.3,0),(null,0),(),(inf,2.225)],
[(-inf,0),(null,inf),(),(1.3,2.6)]
]
```

5. Now substitute all nulls in both attributes with zeros:

```
AFL% substitute(m4x4_null,zeros);
```

```
[
[(1,0),(0,2.5),(),()],
[(),(6,0),(nan,0),(0,3.14153)],
[(7.3,0),(0,0),(),(inf,2.225)],
[(-inf,0),(0,inf),(),(1.3,2.6)]
]
```

Note that only null values get substituted. Empty cells remain empty, and all other values remain the same.

Name

sum — Sum attribute values

Synopsis

```
sum(array,attribute[,dimension_1,dimension_2,...])
```

Summary

The sum operator calculates the cumulative sum of a group of values.

Note

The sum operator offers the same functionality as the sum aggregate. For details, see the sum aggregate reference.

Example

This example sums the columns and rows of a 3×3 array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

```
AFL% CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
```

2. Store values of 0–8 in m3x3:

```
AFL% store(build(m3x3,i*3+j),m3x3);
```

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

3. Sum the values of m3x3 along dimension j. This sums the columns of m3x3:

```
AFL% sum(m3x3,val,j);
```

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

4. Sum the values of m3x3 along dimension i. This sums the rows of m3x3:

```
AFL% sum(m3x3,val,i);
```

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

Name

thin — Select data from an array dimension at fixed intervals

Synopsis

```
thin(array,start_1,step_1,start_2,step_2,...);
```

Summary

The thin operator selects regularly spaced elements of the array in each dimension. The selection criteria are specified by the starting dimension value *start_1* and the number of cells to skip using *step_1* for each dimension of the input array.

Note the following limitations:

- The starting offsets must be smaller than the step size, that is *start_1* < *step_1*, *start_2* < *step_2*, and so on.
- The dimension chunk size must be evenly divisible by the step size.

Example

This example selects values from a 6×6 array.

1. Create an array m6x6:

   ```
   AFL% CREATE ARRAY m6x6 <val:double>[i=0:5,6,0,j=0:5,6,0];
   ```

2. Put values of 1–35 into m6x6:

   ```
   AFL% store(build(m6x6,i*6+j),m6x6);
   ```

   ```
   [
   [(0),(1),(2),(3),(4),(5)],
   [(6),(7),(8),(9),(10),(11)],
   [(12),(13),(14),(15),(16),(17)],
   [(18),(19),(20),(21),(22),(23)],
   [(24),(25),(26),(27),(28),(29)],
   [(30),(31),(32),(33),(34),(35)]
   ]
   ```

3. Select every other column of m6x6, starting at the first column;

   ```
   AFL% thin(m6x6,0,1,0,2);
   ```

   ```
   [
   [(0),(2),(4)],
   [(6),(8),(10)],
   [(12),(14),(16)],
   [(18),(20),(22)],
   [(24),(26),(28)],
   [(30),(32),(34)]
   ]
   ```

4. Select every other row from m6x6, starting at the first row;

```
AFL% thin(m6x6,0,2,0,1);
```

```
[
[(0),(1),(2),(3),(4),(5)],
[(12),(13),(14),(15),(16),(17)],
[(24),(25),(26),(27),(28),(29)]
]
```

5. Select every other value from m6x6, starting at the second column;

```
AFL% thin(m6x6,1,2,1,2);
```

```
[
[(7),(9),(11)],
[(19),(21),(23)],
[(31),(33),(35)]
]
```

Name

transpose — Array transpose

Synopsis

```
transpose(array)
```

Summary

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 d_1, d_2, d_3, ..., d_n the result contains the dimensions in reverse order d_n, ..., d_3, d_2, d_1.

Example

This example transposes a 3×3 matrix.

1. Create a 1-attribute, 2-dimensional array called m3x3:

   ```
   AFL% CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Store values of 0–8 in m3x3:

   ```
   AFL% store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Transpose m3x3:

   ```
   AFL% transpose(m3x3);
   ```

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

Name

unload_library — Unload a plugin

Synopsis

```
unload_library('library_name')
```

Summary

Unload a plug-in from the current SciDB instance.

Note

The unload_library operator provides the same functionality as the AQL UNLOAD LIBRARY '*library_name*' statement.

To load a library, see the load_library operator reference.

Example

This example loads and unloads the example plugin, librational.so.

```
load_library('librational')
unload_library ('librational')
```

The file extension is not included in the library name.

Name

unpack — Produces a one-dimensional result array from the data in a multi-dimensional source array. Note that the unpack operator excludes all empty cells from the result array.

Synopsis

```
unpack(source_array,dimension_name[,chunk_size]);
```

Summary

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

You can control the chunk size of the resulting array with the optional chunk_size parameter. The default chunk size is 1 million.

Examples

This example takes 2-dimensional, 1-attribute array and outputs a 1-dimensional, 3-attribute array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

```
AQL% CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
```

2. Store values of 0–8 in m3x3:

```
AFL% store(build(m3x3,i*3+j),m3x3);
```

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

3. Create a new attribute called val2 containing values 100–108 and store the resulting array as m3x3_2attr:

```
AFL% store(apply(m3x3,val2,val+100),m3x3_2attr);
```

```
[
[(0,100),(1,101),(2,102)],
[(3,103),(4,104),(5,105)],
[(6,106),(7,107),(8,108)]
]
```

4. Unpack m3x3_2attr into a 1-dimensional array.

```
AFL% unpack(m3x3_2attr, x);
```

```
{x} i,j,val,val2
{0} 0,0,0,100
{1} 0,1,1,101
```

```
{2} 0,2,2,102
{3} 1,0,3,103
{4} 1,1,4,104
{5} 1,2,5,105
{6} 2,0,6,106
{7} 2,1,7,107
{8} 2,2,8,108
```

The first two values in each cell are the dimensions, and the second two are the attribute values.

This example illustrates how empty cells are removed during the unpack process.

1. We use a previously created 3x3 array, A, where row 1 has only empty cells, row 2 has only null values, and row 3 has only non-null values.

```
[
[(),(),()],
[(null),(null),(null)],
[("a7"),("a8"),("a9")]
]
```

2. Unpack array A.

```
AFL% unpack(A, x);
```

```
{x} row,col,value
{0} 2,1,null
{1} 2,2,null
{2} 2,3,null
{3} 3,1,"a7"
{4} 3,2,"a8"
{5} 3,3,"a9"
```

Note that `unpack` has excluded the empty cells from the result array.

Name

var — Variance

Synopsis

```
var(array,attribute[,dimension_1,dimension_2,...])
```

Summary

The `var` operator returns the variance of a set of values taken from an array attribute.

Note

The var operator provides the same functionality as the var aggregate. For details, see the var aggregate reference.

Example

This example finds the variance of each row of a 2-dimensional array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

```
AFL% CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
```

2. Store values of random values between 0 and 9 in m3x3:

```
AFL% store(build(m3x3,random()%10),m3x3);
```

```
[
[(5),(2),(9)],
[(0),(1),(9)],
[(7),(3),(5)]
]
```

3. Select the variance of each row of m3x3:

```
AFL% var(m3x3,val,i);
```

```
[(12.3333),(24.3333),(4)]
```

Name

variable_window — Select nonempty cells from a variable size 1-dimensional window

Synopsis

```
variable_window(array,dimension,left_edge,
     right_edge,aggregate_call)
```

Summary

The `variable_window` command aggregates along a 1-dimensional window of variable length. The window is defined by the left and right edges, however, this type of window aggregate excludes cells that are empty.

Example

This example aggregates the sum along a 1-dimensional variable window that collects one nonempty value preceding and one nonempty value following a cell.

1. Create an array called m4x4 and fill it with increasing integers:

```
AFL% CREATE ARRAY m4x4 <val:double>[i=0:3,4,0,j=0:3,4,0];
```

```
AFL% store(build(m4x4,i*4+j),m4x4);
```

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

2. Use variable_window to select one value preceding and the one value following a cell. The window proceeds along the i dimension and calculates the sum of the windowed values.

```
AFL% variable_window(m4x4,i,1,1,sum(val));
```

```
[
[(4),(6),(8),(10)],
[(12),(15),(18),(21)],
[(24),(27),(30),(33)],
[(20),(22),(24),(26)]
]
```

```
AFL% variable_window(m4x4,j,1,1,sum(val));
```

```
[
[(1),(3),(6),(5)],
[(9),(15),(18),(13)],
[(17),(27),(30),(21)],
[(25),(39),(42),(29)]
]
```

Name

versions — Show array versions

Synopsis

```
versions(named_array);
```

Summary

The versions operator lists all versions of an array in the SciDB namespace. The output of the versions command is a list of versions, each of which has a version ID and a date stamp which is the date and time of creation of that version. The argument *named_array* must be an array that was previously created and stored in the SciDB namespace.

Example

This example creates an array, updates it twice, and then returns the first version of the array.

1. Create an array called m1:

   ```
   AFL% CREATE ARRAY m1 <val:double>[i=0:9,10,0];
   ```

2. Store 1 in each cell of m1:

   ```
   AFL% store(build(m1,1),m1);

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

3. Update every cell to have value 100:

   ```
   AFL% store(build(m1,100),m1);

   [(100),(100),(100),(100),(100),(100),(100),(100),(100),(100)]
   ```

4. Use the versions command to see the two versions of m1 that you created:

   ```
   AFL% versions(m1);

   {VersionNo} version_id,timestamp
   {1} 1,"2012-12-11 17:17:38"
   {2} 2,"2012-12-11 17:17:38"
   ```

5. Use the scan operator and the '@1' array name extension to display the first version of m1.

   ```
   AFL% scan(m1@1);

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

Name

window — Compute aggregates over moving window

Synopsis

```
AFL% window(array, dim_1_low,dim_1_high, [dim_2_low,dim_2_high,]...
   aggregate_1[,aggrgegate_2, ...]
```

Summary

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

Note

The AFL window operator provides the same functionality as the AQL **SELECT** ... **FROM** ... **WINDOW** statement. See the User's Guide chapter on Aggregates for more information.

Example

This example calculates a running sum for a 3×3 window on a 4×4 array. The window is multi-dimensional, with the same number of dimensions as the array, and is specified by a pair of values for each dimension, the "high" and "low" sizes. Each dimension of the window includes one cell for the "center", "high" number of cells above it, and "low" cells below it.

1. Create an array called m4x4:

   ```
   AFL% CREATE ARRAY m4x4 <val:double>[i=0:3,4,0,j=0:3,4,0];
   ```

2. Store values of 0–15 in m4x4:

   ```
   AFL% store(build(m4x4,i*4+j),m4x4);
   ```

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

3. Return the maximum and minimum values on a moving 3×3 window on m4x4: This window specification is a two-dimensional window of size 3x3, whose "center" is at the upper left corner of the 3x3 rectangle.

   ```
   AFL% window(m4x4,0,2,0,2,max(val),min(val));
   ```

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

Name

xgrid — Produces a result array with the same dimensions and attributes as a source array, but with the size of each dimension multiplied by an integer scale you supply.

Synopsis

```
AFL%  xgrid(source_array,scale_1[,scale_2,..., scale_N])
```

Summary

The xgrid operator produces a result array by scaling an input array. Within each dimension, the operator duplicates each cell a specified number of times before moving to the next cell. The xgrid operator takes one *scale* argument for every dimension in *source_array*. The result array has the same number of dimensions and attributes as the input array.

Example

This example scales each cell of a 2-dimensional array into a 2×2 subarray.

1. Create an array called m3x3:

```
AFL% CREATE ARRAY m3x3 <val:double> [i=0:2,3,0,j=0:2,3,0];
```

2. Put values of 0–8 into m3x3:

```
AFL% store(build(m3x3,i*3+j),m3x3);
```

```
{i,j} val
{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
```

3. Expand each cell of m3x3 into a 2×2 sub-grid. Store the resulting array as m6x6:

```
AFL% store(xgrid(m3x3,2,2),m6x6);
```

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

Appendix A. Licenses

ScaLAPACK

Appendix B. Acknowledgments

We gratefully acknowledge the contribution of the people and packages that have paved the way for us in Numerical Linear Algebra:

BLAS

L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, *An Updated Set of Basic Linear Algebra Subprograms (BLAS)*, **ACM Trans. Math. Soft.**, 28-2 (2002), pp. 135--151.

LAPACK

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, ***LAPACK Users' Guide*** (Third Edition), (Philadelphia: Society for Industrial and Applied Mathematics, 1999) (ISBN 0-89871-447-8)

ScaLAPACK

L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, ***ScaLAPACK Users' Guide (Software, Environments and Tools)***, (Philadephia: Society for Industrial and Applied Mathematics, 1987)

MPI

M. Snir, J. Dongarra, J. S. Kowalik, S. Huss-Lederman, S.W. Otto, D. W. Walker, ***MPI: The Complete Reference***, (Vol. I and II), 2nd ed., (Cambridge, MA: The MIT Press; 1998)

Index

Symbols

A

B

C

D

E

F

G

H

I

S

T

U

V

W

X