SciDB User Guide

Community Edition

SciDB User Guide: Community Edition

Version 14.3.7410

Table of Contents

List of Figures

List of Tables

Release Notes

These release notes cover features and changes for:

- SciDB Community Edition (CE)
- SciDB Enterprise Edition (EE)
- Paradigm4 Labs (P4Labs), which is a collection of external tools (such as SciDB-R) maintained in the form of GitHub repositories.

New features and important bug fixes are described in Sections 2, 3, and 4. Feature changes made in this release and advance notice of changes planned for the next release (R14.6) are discussed in Section 5. Known limitations are noted in Section 6.

1. Supported Operating Systems

SciDB R14.3 supports three platforms: Ubuntu 12.04, RHEL 6, and CentOS 6.

2. Community Edition Features and Changes

This version includes the following features and changes:

- **Backup/Restore utility:** A new utility is provided to backup and restore SciDB arrays. The functionality is fairly rich. For example, it supports three formats including the OPAQUE format. The utility is **scidb_backup.py**, located in **/opt/scidb/14.3/bin** folder.
- **Macro support:** SciDB 14.3 introduces basic user-defined macros that can be used to extend both the set of SciDB functions and the set of AFL operators. A new operator, **load_module()**, is provided. See Chapter 12, *Macros* for details.
- **Improved storage management:** SciDB 14.3 features better management of disk space, in particular:
  - When SciDB arrays are deleted, their storage space is returned to the file system.
  - When queries are cancelled, disk space used by those queries is reused by subsequent queries.
- **Mixed use of AQL and AFL:** An AQL selection query may be used inside an AFL query wherever a SciDB array is expected. This feature may be useful for people familiar with AQL but want to tap additional functionality provided by AFL. For example, the following query is supported:

  ```
  $ iquery -aq "rank( (select * from build(<v:double>[i=1:5,5,0],i)), v)"
  ```
- **Improved Memory Management:** Certain out-of-memory errors that could occur in the previous release have been fixed.

3. Enterprise Edition Features and Changes

This version includes the following features and changes:

- **SciDB as a service:** SciDB EE 14.3 can now be configured to run as a service. One advantage is that the SciDB service will restart after a system reboot. For details, see Section 5.7 of the *SciDB Enterprise Edition User Guide*.

- **Sparse Matrix Multiply:** The **spgemm()** operator has new functionality and improved performance:
  - **spgemm()** supports optional **'min.+'** and **'max.+'** semiring arithmetic. Such alternative arithmetic permits **spgemm()** to be used for a wide variety of graph algorithms, such as shortest paths and measures of centrality.
  - The performance of **spgemm()** has been improved in the case of matrix-vector multiplication.
- **Shortest paths:** An example script for computing single-source shortest paths (SSSP), using spgemm() and the Bellman-Ford algorithm is given in **/opt/scidb/14.3/bin/ bellman_ford_example.sh**.
- **PageRank:** An example script for computing Google PageRank is given in **/opt/scidb/14.3/bin/ pagerank_example.sh**.
- **Random number generator:** A new operator, **rng_uniform()**, provides a faster, higher quality, pseudo-random number generator than **build(...,random())** and similar variants.

4. Paradigm4 Labs Features and Changes

We make available a collection of external tools in the form of GitHub repositories at http://github.com/paradigm4.

For SciDB 14.3, a non-root install procedure has been added. For details, see http://downloads.paradigm4.com/nri.

5. Deprecation Notes

This section lists important feature changes made in this release and changes forthcoming in future releases. For each change, we also identify in which the release we provided deprecation notice.

Functionality Change	**Deprecation Notice**	**Deprecation Effective**	**Edition**
Changes made in this release (R14.3)			
Removed individual aggregate operators: • **approxdc()** • **avg()** • **count()** • **max()** • **min()** • **stdev()** • **sum()** • **var()** Use **aggregate()** instead.	13.12	14.3	CE
Removed operator **redimension_store()**. Use **store(redimension())** instead.	13.12	14.3	CE

Functionality Change	**Deprecation Notice**	**Deprecation Effective**	**Edition**
Changes made in this release (R14.3)			
Removed operator **cross()**. Use **cross_join()** instead.	13.12	14.3	CE

The behavior of the **show()** operator has changed.
 	
Let **A** be the name of a stored SciDB array. This syntax is no longer supported:
 	
`AFL% show('A as array', 'afl');`
 	
Instead, use the following syntax:
 	
`AFL% show('filter(A, true)','afl');`
	14.3	14.3	CE

Functionality Change	**Deprecation Notice**	**Edition**
Expected changes in the next release (R14.6)		
Will remove the following operators: **corr()**, **covar()** and **covariance()**. Use **pearson()**, **spearman()**, or **kendall()** instead.	14.3	EE
Will remove the following operators: **euclidean()** and **manhattan()**. Use **apply()** instead.	14.3	EE
Will remove attribute type **string_xxx**; use the **string** type.	14.3	CE
Will remove the use of integer data types with the **spgemm()** operator.	14.3	EE
Will remove the **sample()** operator. Use **bernoulli()** instead.	14.3	CE
Will remove the **build_sparse()** operator. Use a combination of **redimension()** and **build()** instead. See the AFL operator reference, build, for details.	13.12	CE
Will remove the **NOT EMPTY** constraint in array schemas. All SciDB arrays will be empty-able.	13.11	CE
Will disallow the use of "impure" operators inside other operators. By impure, we mean operators that have side effects. The following operators fall into this class, and thus should be used only at the top-level of a query. • **create_array()**, **store()**, **load()**, **input()**, **rename()**, **remove()**, and **save()**, which modify arrays. • **load_library()**, **unload_library()**, **load_module()**, and **setopt()**, which modify how queries execute.	14.3	CE
Will disallow the use of the following parameters in **config.ini**: • **chunk-segment-size** • **repart-seq-scan-threshold**	14.3	CE

Functionality Change	**Deprecation Notice**	**Edition**
Expected changes in the next release (R14.6)		
• **repart-algorithm** • **repart-dense-open-once** • **repart-disable-tile-mode** • **rle-chunk-format** • **parallel-sort** • **save-ram**		
Will enforce case sensitivity for all identifiers, including operators, functions, types, attribute names, dimensions, and array names.	14.3	CE
The default chunk size will be set to 1 million cells (presently defaults to 500 thousand cells).	14.3	CE

6. Known Limitations

- The **dimensions()** operator sometimes reports incorrect start, length, low and high values for unbounded arrays. Correct values may be obtained using **apply()** to translate a dimension into an attribute, followed by a min / max aggregate.
- The difference between the largest and the smallest dimension value in an array must not exceed (2^{61} - 1).
- The **gemm()** and **gesvd()** operators produce incorrect results when the input matrices are sparse.
- The **adddim()** and **deldim()** operators may return incorrect results, or cause a system crash, when used in certain queries.
- Whenever a query requires filtering on dimension values, **filter()** may not be the fastest approach. In the vast majority of cases, **between()**, **subarray()**, **cross_join()**, or a combination of them, performs better. For more information about using **cross_join()** as a dimensions filter, see http://www.scidb.org/forum/viewtopic.php?f=18&t=1204.

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. 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.

1.2. 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.

1.2.1. Dimensions

An n-dimensional SciDB array has dimensions *d1*, *d2*, ..., *dn*. Array dimensions are 64-bit integers. 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.

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.

1.2.2. 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. Dimensions have names, upper and lower bounds, sizes, and overlaps. You must specify the dimension name—all other characteristics are optional, and have default values.
- Array attributes. 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.

1.2.3. Array Indexing

SciDB uses array indexing—your dimensions are essentially indices. This splits the data into fixed-size chunks. When you need to retrieve data, you specify the coordinates and SciDB can very quickly determine the chunk that contains the requested data. In other databases, this functionality comes at a cost: you need to create an index and maintain it—and the maintenance over time can become very costly, in terms of the need to re-index as the data set grows.

Another advantage of SciDB array indexing is that your data is clustered: data that logically belongs together is stored close together on disk.

1.3. Terminology

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

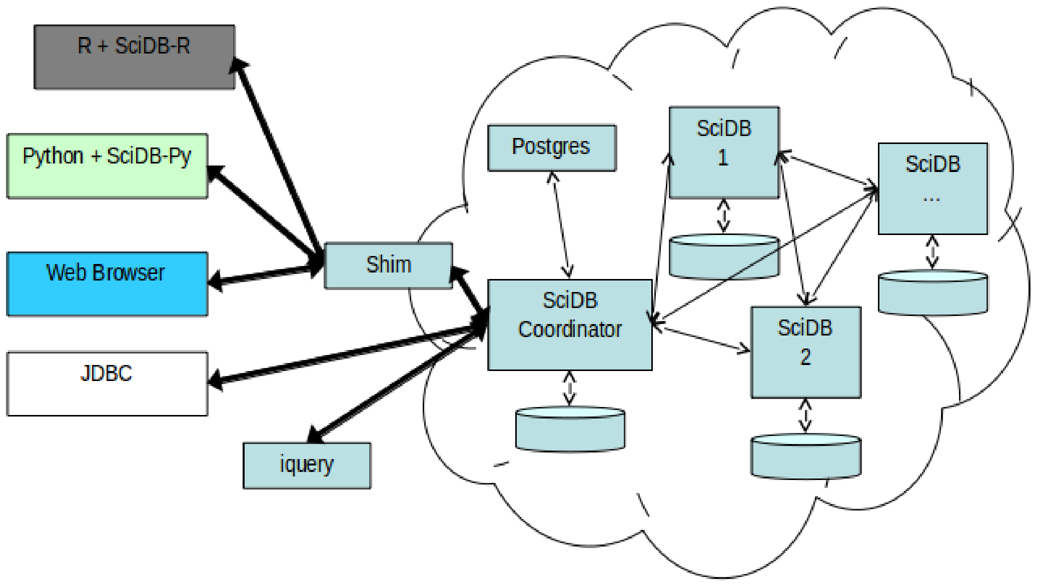
Array	A data structure that is used to store SciDB data. An array has any number of dimensions and attributes. Comparing with the relational model, arrays act like tables, array cells act like rows, dimensions and attributes both act like columns, while the set of all dimensions acts like the primary key. For more details, see Section 1.2, “Array Data Model”.
Matrix	A SciDB array that has two dimensions and one numeric attribute. Linear algebra is performed on matrices.
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.

1.4. Basic Architecture

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

Figure 1.1. SciDB Shared-Nothing Cluster 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 (or instances) 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 instance 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 cluster. 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.

The following statements describe relationships between the constituents of a SciDB installation.

- A SciDB *cluster* consists of one or more servers, each of which can host one or more *SciDB instances*.
- 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 two types of instances: *coordinator instances* and *worker instances*. A cluster has exactly one coordinator instance, and zero or more worker instances.

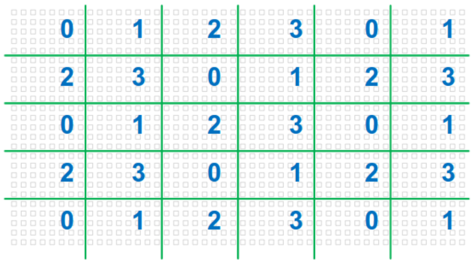
- Note the difference between a server and an instance: the server may contain several instances. The server is a computer (or VM), while the instances areLinux processes.
- When two or more instances are hosted by the same server, each of those instances requires its own directory to store SciDB database files. In such cases, the directories for different instances may be mounted on different drives.
- Postgres communicates with all of the instances, but only resides on the coordinator.
- SciDB supports the following clients: shim (to connect to web browsers), SciDB-R and SciDB-Py (to connect to R and Python, respectively), and iquery. Note that only iquery is described in this guide.

1.5. Chunking

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 storing and updating a subset of the array locally, and for executing queries that use the locally stored data.

By distributing data uniformly across all instances, SciDB is able to deliver scalable performance on computationally or I/O intensive analytic operations on very large data sets. SciDB distributes the data automatically—you do not need to manage chunk distribution beyond specifying chunk size.

Data is split into regular, rectilinear chunks, and distributed between instances. In this diagram, we assume a 4-instance cluster, and the numbers represent the instances, from 0 to 3.



Chunking data in this way has several advantages:

- Data that are close together are stored close together.
- Coordinates are implied, and thus do not need to be stored.
- You get a light-weight indexing method for free.
- For linear algebra and multi-dimensional windowing operations, this mechanism provides good performance.

Additionally, you can specify a *chunk overlap*. SciDB then stores some cells in neighboring chunks. For details, see Section 5.5.1, “Chunk Overlap”.

1.6. 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.

1.6.1. 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 arrays	SciDB stores arrays into individual files. Each instance contains a **datastore** folder. For each array, SciDB creates one file (per instance), using the array ID as the file name. As chunks are added to the array, the system file grows accordingly. Also, each time the system writes to an array, new versions are created, and each version consumes some amount of disk space. If the array is deleted, the system file is removed, and all the space is returned. Empty space can creep into the data files if a transaction is canceled, but SciDB keeps metadata that allows for reuse of this space.
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.
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 *temp_file* mechanism. Data written to the temporary file lasts only for the lifetime of a query. It is removed upon successful completion or abort of the query.

1.6.2. 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.

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

1.6.3. 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.

1.7. Array Processing

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

1.7.1. 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.

1.7.2. 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.

1.7.3. 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.

1.8. Clients and Connectors

The SciDB software package that you downloaded contains a 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.

Chapter 2. SciDB Installation and Administration

SciDB is supported on the following platforms:

- Ubuntu 12.04
- CentOS 6
- Red Hat Enterprise Linux (RHEL) 6

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.

Note

SciDB installation must be done by an account with root privileges.

To report issues, contact `support@scidb.org`.

Installation and Administration is broken down into the following topics:

- Quick Install
- Upgrading SciDB
- Preparing the Platform—if you are doing a new, manual installation, start here.
- Installing Packages
- Configuring SciDB
- Initializing and Starting SciDB
- Tuning your SciDB Installation

2.1. Quick Install

Paradigm4 provides several tools that make it possible to perform a fairly simple installation of SciDB. Additionally, if you have purchased the Enterprise Edition, you can also install it using the following scripts.

See the following for details:

- Cluster install script, located at https://github.com/Paradigm4/deployment.
- Alternatively, you can install without root privileges—on CentOS/RHEL only—by using the following script: http://downloads.paradigm4.com/nri.
- Configurator (generates a config.ini file from your interactive input): https://github.com/Paradigm4/configurator.
- A how-to video on installing SciDB is available here: http://www.paradigm4.com/video.

See the remainder of the chapter for more details about the installation process and files.

2.2. Upgrading SciDB

This section describes how to upgrade your version of SciDB. It contains the following information:

- The upgrade procedure: Section 2.2.1, "Upgrade Procedure"
- The details of the backup and restore script (which is the suggested method for backing up and restoring your arrays): Section 2.2.2, "SciDB Backup Script"
- The details of how to manually backup and restore your arrays: Section 2.2.3, "(OPTIONAL) Manually Backup and Restore Your Arrays"

2.2.1. Upgrade Procedure

For the upgrade procedure, we use the following conventions:

- To represent your current version: **scidbOldVersion**.
- To represent the new version that you are installing: **scidbNewVersion**.

For example, if you are upgrading from 13.12 to 14.3:

- **scidbOldVersion** = 13.12
- **scidbNewVersion** = 14.3

The preferred way to upgrade SciDB is to use the supplied backup script, `scidb_backup.py`. This file is in the /opt/scidb/14.3/bin folder. Perform the following tasks:

1. Backup your data before upgrading—upgrading to 14.3 cannot be done in-place. From the **scidb** user account, run the backup script. As an example, the following command creates a folder, **./mySavedArrays**, and save all of your arrays, using the OPAQUE format, into that folder:

   ```
   $ /opt/scidb/14.3/bin/scidb_backup.py -s opaque mySavedArrays
   ```

   For usage details of the backup script, see Section 2.2.2, "SciDB Backup Script".

2. Stop SciDB.

3. If you want to keep your current SciDB configuration file, make a backup copy of it somewhere. By default, the file is in the following location:

   ```
   opt/scidb/scidbOldVersion/etc/config.ini
   ```

4. **OPTIONAL.** You can run multiple versions of SciDB, if you would like: for details, see Section 3.4, “Running multiple versions of SciDB”. However, you can remove the previous version of SciDB if you only need to run the latest version of SciDB. If you installed to the default location, run the following command to remove the old version:

```
sudo rm -Rf /opt/scidb/scidbOldVersion/
```

5. **Upgrade.** Perform the following steps to upgrade to the latest version of SciDB.

Note

In a cluster, this step must be performed on all servers.

a. Append the repository URL to the correct file:

- **Ubuntu:** Append the following lines to the `/etc/apt/sources.list.d/scidb.list` file:

```
deb  http://downloads.paradigm4.com/  ubuntu12.04/14.3/
deb-src  http://downloads.paradigm4.com/  ubuntu12.04/14.3/
```

- **CentOS / RHEL:** Append the following line to the `/etc/yum.repos.d/scidb.repo` file:

```
baseurl=http://downloads.paradigm4.com/centos6.3/14.3/
```

b. Run the update and install commands:

- **Ubuntu:**

```
sudo apt-get update
sudo apt-get install scidb-14.3-all
```

- **CentOS / RHEL:**

```
sudo yum clean all
sudo yum update
sudo yum install scidb-14.3-all
```

For more details, see Section 2.4, “Installing Packages”.

c. Update your environment variables to refer to the new version of SciDB. Make sure to change your SciDB Version variable from its current value to the **scidbNewVersion**. Old version:

```
export SCIDB_VER=scidbOldVersion # This should be set in your environment.
```

Set to new version:

```
export SCIDB_VER=14.3
```

For details, see Section 2.6.1, “Set Environment Variables”.

d. Set up your configuration file, including setting the `enable-catalog-upgrade` parameter to `true`. Make sure to change all occurrences of **scidbOldVersion** to **scidbNewVersion**. After you edit the file, copy it to the `/opt/scidb/14.3/etc/` folder.

Typically, the following settings are affected:

```
install_root=/opt/scidb/13.12
pluginsdir=/opt/scidb/13.12/lib/scidb/plugins
logconf=/opt/scidb/13.12/share/scidb/log4cxx.properties
```

Also, make sure to add the following setting to the configuration file (if it does not already exist):

```
enable-catalog-upgrade=true
```

For more details about the configuration file, see Section 2.6, “Configuring SciDB”.

e. Restart SciDB. Initialization is recommended to avoid errors that you would otherwise receive, based on incompatibilities between your "old" database and your "new" SciDB version. Remember, though, that initializing the SciDB database destroys all the data in that database.

Note

If you get an error that says SciDB needs to upgrade the system catalog, and that the action failed, make sure to set the `enable-catalog-upgrade` parameter to **true**, as detailed in the previous step.

6. **Restore.** After you have the new version of SciDB running, restore any data that you backed up in step 1. For example, if you used the command in step 1 to backup your arrays, you would run the following command to restore them:

```
$ /opt/scidb/14.3/bin/scidb_backup.py -r opaque mySavedArrays
```

2.2.2. SciDB Backup Script

This section describes the scidb backup and restore script, `scidb_backup.py`. Note the following:

- You must run the script from the **scidb** user account on the coordinator host.
- The script saves or restores SciDB arrays into individual files (one per array), into a folder that you specify.
- By default, the save is done onto the coordinator only—you can perform a parallel save by specifying the `-p` option.
- If you use the `--parallel` option, the script saves arrays into multiple sub-folders, one per SciDB instance. For example, if you have a 2-server, 4-instance cluster, and you specify the backup folder, the following folders are created:
  - On the coordinator: **backup.0** and **backup.1**
  - On the second server: **backup.1** and **backup.2**
- You must use the same options when restoring as backup up. For example, if you backup using the text format, and the parallel option, then you must use the text format and parallel option when you restore your arrays.
- By default, all arrays are saved. Use the `-f` option to filter the arrays to save. You specify a Python regex pattern against which to match the array names to save.
- Use the `-h` option to display usage details for the script.

The script takes the following arguments:

- **-h**: Displays the usage details.

- **-s** *format*: The save format. The value must be one of the following: **opaque**, **binary**, **text**.
- **-r** *format*: The restore format. The value must match the format used to save the arrays.
- **-d**: Deletes all of the array data, on all hosts, for the base folder specified.
- **--port** *port*: The port that the SciDB cluster is running on. The default is 1239, so this parameter is optional if you are running your cluster on the default port.
- **-a**: Saves all versions of the arrays. By default, only the latest version of each array is saved.
- **-f** *pattern*: A regular expression pattern. The pattern is matched against, and only array names that match are saved.
- **-p**: Specifies a parallel save or restore. If you use this option when saving, you must also use it when restoring.
- **-z**: Compresses data files (using gzip) when saving array data, and then uncompresses the data when restoring.

For example, to perform a parallel, binary save of all versions of arrays that start with 'A', and also compress the data, you could run the following command:

```
$ scidb_backup.py -s binary -p -a -f "A.*" -z myBackup_A
```

2.2.3. (OPTIONAL) Manually Backup and Restore Your Arrays

If you choose not to use the backup/restore script, this section describes the manual ways you can backup and restore your arrays. Note the following:

- The OPAQUE save/load format is the most efficient way to back up your array data. It is guaranteed to work between consecutive versions. For example, if you save your arrays (in Opaque format) from version 13.12, you can use Opaque load to load them directly into version 14.3, without needing to perform a redimension. Make sure to reload the saved array into the a SciDB installation with identical database configuration parameters.
- Next best is to use binary save/load. This is faster than using text files, but has some intermediate steps (you need to unpack before saving, and redimension after restoring). There is a script that automates this process for all of your arrays.
- You can use the SciDB text format to save and restore arrays between versions. It is the slowest method, but on the other hand, you can backup and restore multidimensional arrays without needing to perform any intermediate steps.

2.2.3.1. Manually Backup Your Arrays

If you are upgrading to 14.3 from an earlier version than 13.12 (e.g. 13.9 or 13.6), and backing up manually, then we recommend that you use binary save/load. The following steps walk through saving and then restoring a small example array.

1. Assume you have the following array in SciDB, in version 13.12.

   ```
   AFL% show(notFlat)
   ```

```
notFlat

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

[elevation=0:999,100,0,
elapsedTime=0:15,16,0]
```

```
{elevation,elapsedTime} exposure,measuredIntensity
{21,6} 'Low',35
{29,2} 'Medium',89
{51,4} 'Medium',null
{89,1} 'Medium',95
{191,3} 'High',99
{222,5} 'Low',41
{238,5} 'High',97
{288,3} 'Medium',null
{320,1} 'High',100
{378,0} 'Low',100
{390,7} 'Low',29
{490,3} 'Low',60
{549,2} 'Low',71
{583,0} 'Medium',100
{629,0} 'High',100
{632,7} 'High',null
{670,5} 'Medium',80
{787,7} 'Medium',77
{800,4} 'Low',50
{834,6} 'High',null
{838,1} 'Low',85
{850,6} 'Medium',78
{890,2} 'High',99
{999,4} 'High',98
```

2. We need to unpack this data into a 1-dimensional array, and then save it.

```
AFL% save(unpack(notFlat,myDim), 'notFlat_saved.bin', -2, '(int64,int64,string,
 int64 null)');
```

   This saves the array data into a 1-D binary array on the coordinator instance.

3. To restore, you will need to recreate the 1-D intermediate array. The following query returns the schema we need.

```
AFL% show('unpack(notFlat,myDim)','afl');
```

```

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

[myDim=0:*,16000,0]
```

2.2.3.2. Manually Restore Your Arrays

This section provides guidelines on restoring your array data. It continues the example begun in the previous section, Section 2.2.2, “SciDB Backup Script”.

After you upgrade, perform the following steps to reload the array that you saved in the previous section.

1. Create the intermediate 1-D array, using the schema we displayed earlier.

```
AFL% CREATE ARRAY restored_1D_Array <elevation:int64,
      elapsedTime:int64,exposure:string,measuredIntensity:int64 NULL DEFAULT null>
     [myDim=0:*,16000,0]
```

2. Now, we can load the saved data into our new 1-D array.

```
AFL% load(restored_1D_Array, 'notFlat_saved.bin', -2,
      '(int64, int64, string, int64 null)');
```

3. Next, recreate the target array, making sure to match the schema of the original.

```
AFL% create array restoredFinal<exposure:string,measuredIntensity:int64
      NULL DEFAULT null> [elevation=0:999,100,0,elapsedTime=0:15,16,0];
```

4. Finally, redimension the 1-D array into the target array.

```
AFL% store(redimension(restored_1D_Array, restoredFinal),restoredFinal);
```

```
{elevation,elapsedTime} exposure,measuredIntensity
{21,6} 'Low',35
{29,2} 'Medium',89
{51,4} 'Medium',null
{89,1} 'Medium',95
{191,3} 'High',99
{222,5} 'Low',41
{238,5} 'High',97
{288,3} 'Medium',null
{320,1} 'High',100
{378,0} 'Low',100
{390,7} 'Low',29
{490,3} 'Low',60
{549,2} 'Low',71
{583,0} 'Medium',100
{629,0} 'High',100
{632,7} 'High',null
{670,5} 'Medium',80
{787,7} 'Medium',77
{800,4} 'Low',50
{834,6} 'High',null
{838,1} 'Low',85
{850,6} 'Medium',78
{890,2} 'High',99
{999,4} 'High',98
```

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. If you need to install **sudo**, execute the following command (as root):

  ```
  yum install sudo
  ```

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 a user with sudo privileges 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, **on the coordinator server only**.

SciDB uses `ssh` for execution of management commands such as start and stop within a cluster. This is why the **scidb** user account needs to have `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 all instances—the coordinator and to each worker—to authorize ssh clients connecting to SciDB:

```
ssh-copy-id -i ~/.ssh/id_rsa.pub scidb@localhost
ssh-copy-id -i ~/.ssh/id_rsa.pub scidb@0.0.0.0
ssh-copy-id -i ~/.ssh/id_rsa.pub scidb@127.0.0.1
ssh-copy-id -i ~/.ssh/id_rsa.pub scidb@worker
```

3. Login to the localhost (and 0.0.0.0 and 127.0.0.1) and each remote host. Note that no password is required now (though you still need to approve each host by responding 'yes' at the prompt):

```
ssh scidb@localhost
ssh scidb@0.0.0.0
ssh scidb@127.0.0.1
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:

- Disable iptables
- Configure SELinux

Note

These configuration steps must be performed on all servers—the coordinator and all workers.

2.3.4.1. Disable iptables

If you are running on CentOS or RHEL, you need to disable 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 must disable iptables. To disable iptables, use either of the following commands:

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

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. For details, see your operating system documentation, or search the web for instructions.

Note

Disabling SELinux may introduce security risks.

For more information on SELinux, see http://wiki.centos.org/HowTos/SELinux.

2.4. Installing Packages

Packages are available for Ubuntu, CentOS, and RHEL. Note that CentOS and RHEL use the same package. For a list of packages, see Section 2.8.1, “SciDB Packages”.

In order to install SciDB packages, you need to have **wget**.

- On CentOS or RHEL:

  ```
  sudo yum install wget
  ```

- On Ubuntu:

  ```
  sudo apt-get install wget
  ```

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 a user with sudo privileges. 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/  ubuntu12.04/14.3/
   deb-src  http://downloads.paradigm4.com/  ubuntu12.04/14.3/
   ```

   If the `/etc/apt/sources.list.d/scidb.list` file does not exist, create it (using sudo).

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-14.3-all-coord # On the coordinator server only
sudo apt-get install scidb-14.3-all # On all servers other than the coordinator
 server
sudo apt-get source scidb # If you want the SciDB source
```

Note

The only difference between the scidb-14.3-all and scidb-14.3-all-coord packages is that the latter has a dependence on PostgreSQL 8.4. Installing scidb-14.3-all-coord will also install PostgreSQL 8.4, if it is not already installed on your system. If you already have PostgreSQL 8.4 on your coordinator server, there is no particular need to use the scidb-14.3-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. The CentOS / RHEL packages work with CentOS 6.3, 6.4, and 6.5.

Note

This procedure must be performed by a user with sudo privileges. 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
   sudo 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/centos6.3/14.3/
   gpgcheck=1
   ```

4. If you are upgrading from a previous version of SciDB, run the following command:

   ```
   sudo yum clean all
   ```

5. Update packages by running the following command:

   ```
   sudo yum update
   ```

6. Install SciDB. If you want the default SciDB installation with all packages installed, you can install the scidb-14.3-all (or scidb-14.3-all-coord) meta-package. This package includes all of the standard SciDB packages.

   ```
   sudo yum search scidb
   sudo yum install scidb-14.3-all-coord # On the coordinator server only
   sudo yum install scidb-14.3-all # On all servers other than the coordinator server
   ```

Note

The only difference between the scidb-14.3-all and scidb-14.3-all-coord packages is that the latter has a dependence on PostgreSQL 8.4. Installing scidb-14.3-all-coord will also install PostgreSQL 8.4, if it is not already installed on your system. If you already have PostgreSQL 8.4 on your coordinator server, there is no particular need to use the scidb-14.3-all-coord package.

2.5. Set Up Postgres as the SciDB System Catalog

If you are upgrading from a previous version of SciDB, you can skip this section, and continue with Section 2.6, "Configuring SciDB". Also, this section only applies to the coordinator server: if you are installing SciDB to a cluster, you can skip this section for all of your worker servers.

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, be aware that SciDB assumes that version 8.4 is running and using port 5432. To change, set the optional configuration parameter, `pg-port`.
- 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. **For CentOS/RHEL only:** If you have never previously run Postgres on your system, you must create the Postgres configuration file, `pg_hba.conf`, by running the following command:

   ```
   sudo service postgresql initdb
   ```

   Note that for Ubuntu, the configuration file is created automatically when you install the SciDB repository.

2. Configure Postgres to use 'md5' authentication for local and remote TCP/IP 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.

   ```
   host    all     all     127.0.0.1/32      md5
   host    all     all     ::1/128           md5
   host    all     all     W.X.Y.Z/N         md5
   ```

This configuration setting causes Postgres to use 'md5' authentication for all local and remote connections from instances within the `W.X.Y.Z/N` subnet in CIDR notation. Replace the subnet/mask with the correct one for your cluster network (by using the Linux **ifconfig** tool or consulting your system administrator).

Warning

Although the 'md5' authentication does provide some level of security, this Postgres configuration might still pose security issues. 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/8.4/static/client-authentication.html.

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
sudo /sbin/chkconfig postgresql on
```

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

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

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

2.6. Configuring SciDB

This section introduces the SciDB config.ini and shows how to configure SciDB prior to initialization (usually `/opt/scidb/14.3/etc/config.ini`). Logging configuration is also described. You create and maintain the configuration file on the coordinator server only.

Note

SciDB allows you to assign entire hard disks to its data directories, using parameters in the configuration file. When you do this, the mounted directories **must be entirely empty**. This includes any system files added during routine formatting. If SciDB encounters any non-SciDB files in its directories, it generates errors and cannot run.

For increased performance, you must set some configuration parameters that are dependent upon your particular hardware setup and needs. To increase your performance, see Appendix A, *Tuning your SciDB Installation*.

2.6.1. 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=14.3
export PATH="$PATH:/opt/scidb/$SCIDB_VER/bin:/opt/scidb/$SCIDB_VER/share/scidb"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/opt/scidb/$SCIDB_VER/lib"
```

2.6.2. SciDB Configuration File Example

Note

The configuration file resides on the coordinator server only.

The standard location of the SciDB configuration file is `/opt/scidb/14.3/etc/config.ini`. You can also use the Configurator tool, which generates a config.ini file from your interactive input. It is located here: https://github.com/Paradigm4/configurator.

The configuration 'test1' below is an example of the configuration for a single-node, four-instance system:

```
[test1]
server-0=localhost,3
db_user=test1user
db_passwd=test1passwd
install_root=/opt/scidb/14.3
metadata=/opt/scidb/14.3/share/scidb/meta.sql
pluginsdir=/opt/scidb/14.3/lib/scidb/plugins
logconf=/opt/scidb/14.3/share/scidb/log4cxx.properties
base-path=/home/scidb/data
base-port=1239
network-buffer=1024
mem-array-threshold=1024
smgr-cache-size=1024
execution-threads=16
result-prefetch-queue-size=4
result-prefetch-threads=4
```

For an example of the configuration for a SciDB cluster, and for more details about the configuration parameters, see Section 2.8, "Installation Reference".

2.7. Initializing and Starting SciDB

You initialize and start SciDB from the coordinator server.

Run the following commands to initialize and start a SciDB database:

Note

For all of the following commands, replace *`scidb_cluster_name`* with the name of your actual cluster.

1. 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.

   ```
   cd /tmp && sudo -u postgres /opt/scidb/14.3/bin/scidb.py
    init_syscat scidb_cluster_name
   ```

2. Run the following command as the 'scidb' user.

   ```
   scidb.py initall scidb_cluster_name
   ```

   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.

   If the `scidb.py initall` command fails with an error, inspect the *`<base-path>`*/000/0/scidb.log file. If the log indicates a PostgreSQL authentication error, modify the `pg_hba.conf` file, and change the authentication parameters set in Section 2.5, “Set Up Postgres as the SciDB System Catalog” from 'md5' to 'trust' or the value recommended by your system administrator.

3. Again, as the 'scidb' user, start the set of local SciDB instances specified in your config.ini file.

   ```
   scidb.py startall scidb_cluster_name
   ```

Note

Initialization may fail if you are using the Lustre file system. To correct the problem, use the `-o localflock` option when mounting the file system.

You can also execute the following commands (as the 'scidb' user).

- To report the status of the various instances:

  ```
  scidb.py status scidb_cluster_name
  ```

- To stop all SciDB instances:

  ```
  scidb.py stopall scidb_cluster_name
  ```

For more details about the scidb.py script, see Section 2.8.6, “SciDB Python Script Parameters”.

2.8. Installation Reference

This section contains reference details about available SciDB packages, configuration files and scripts:

- SciDB Packages
- SciDB Configuration Parameters
- Cluster Configuration Example
- Logging Configuration
- SciDB Logs
- SciDB Python Script Parameters

2.8.1. SciDB Packages

SciDB currently provides the following packages.

- The `scidb` package contains the server binaries.
- The `libscidbclient-14.3` and `libscidbclient-14.3-python` packages contain the SciDB client library and the SciDB connector for python applications.
- The `scidb-14.3-plugins` package contains plugins that can be added to SciDB to extend its capabilities. This package contains the `dense_linear_algebra` plugin.
- The `scidb-14.3-utils` package contains SciDB server utilities.
- The `scidb-14.3-dev` contains development header files for developers creating SciDB extensions (types, operators, and aggregates).
- The `libscidbclient-14.3-dbg`, `libscidbclient-14.3-python-dbg`, `scidb-14.3-dbg`, `scidb-14.3-dev-tools-dbg`, `scidb-14.3-plugins-dbg`, and `scidb-14.3-utils-dbg` packages contain debugging symbols.

2.8.2. SciDB Configuration Parameters

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 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]* To avoid possible problems, we recommend that you use all lowercase for the cluster name.
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	User name to use in the catalog connection string. This example uses *test1user* To avoid possible problems, we recommend that you use all lowercase for the `db_user` parameter.
db_passwd	Password to use in the catalog connection string. This example uses*test1passwd* To avoid possible problems, we recommend that you use all lowercase for the `db_passwd` parameter.
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.
requests	The maximum number of client query requests queued for execution. Any requests in excess of the limit are returned to the client with an error. The default value is 1,000.

The following table describes the cluster configuration file parameters 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. The advantage to using the data-dir-prefix parameter is that you can arbitrarily assign physical storage to SciDB instances. 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.
	
	If a server has multiple storage disks, and you want to assign more than one instance to each disk, then you must set the data-dir-prefix parameter for the instances on that server.

pg-port (optional)	The listening port of Postgres—the port on which Postgres accepts incoming connections. Default: 5432.
redundancy (optional)	Indicates the number of replications of array data. The value of this parameter specifies the number of **extra** copies of array data to store. Setting this parameter to a positive value is the first step in creating a fault tolerant SciDB cluster. Default: 0 (meaning only one copy of data is stored). The number of copies of your data equals 1 + the redundancy value. Maximum value for the redundancy parameter is one less than the number of instances in your SciDB cluster.
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. If you do not use the default name and path for the key, then you must set a value for this parameter.
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. By default, each SciDB instance spawns two Linux processes; one for the instance itself, and one as a watchdog. The watchdog process detects if the instance process fails. If it detects a failure, then it forks a new, replacement process, which brings the instance back online.

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.
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*.

2.8.3. Cluster Configuration Example

The following SciDB cluster configuration is called 'salty.' This cluster consists of 2 identical servers. Each server has the the following characteristics:

- x86_64 4-core processor
- 2 physical disks: each disk has 1 TB direct attached storage
- 8 GB of RAM
- 1Gbps Ethernet
- Linux OS from the list of supported distributions.

The following configuration file applies to such a cluster called `salty`.

```
[salty]
server-0=p4xen1.local.paradigm4.com,3
```

```
server-1=p4xen2.local.paradigm4.com,4
db_user=salty
db_passwd=salty
install_root=/opt/scidb/14.3
pluginsdir=/opt/scidb/14.3/lib/scidb/plugins
logconf=/opt/scidb/14.3/share/scidb/log4cxx.properties
base-path=/home/scidb/DB-salty
# note that this cluster does not use the default base-port, 1239,
# so you may need to open port 1600 on your firewall.
base-port=1600

# Specifying values for the data-dir-prefix
# parameter is optional. See below for details.

# server-0
# Port numbers: 1600 - 1603
data-dir-prefix-0-0=/datadisk1/salty.000.0
data-dir-prefix-0-1=/datadisk2/salty.000.1
data-dir-prefix-0-2=/datadisk1/salty.000.2
data-dir-prefix-0-3=/datadisk2/salty.000.3

# server-1
# Port numbers: 1601 - 1604
data-dir-prefix-1-1=/datadisk1/salty.001.1
data-dir-prefix-1-2=/datadisk2/salty.001.2
data-dir-prefix-1-3=/datadisk1/salty.001.3
data-dir-prefix-1-4=/datadisk2/salty.001.4
```

Note the following about this cluster:

- Each datadisk*n* is mounted on a physical disk. Each SciDB server uses two physical disks and has four instances. Thus each instance has its own processor and shares a disk with one other instance.
- This SciDB cluster has a total of 8 instances, split evenly amongst the 8 cores and 4 disks.
- The names of the disk are local per server; thus datadisk1 on server-0 is a different disk than datadisk1 on server-1.
- The port numbers for each instance on a server are incremental. The port numbers in the above example are listed in the comments for each server. Thus, instance number 3 on each server uses the same port, 1603. Note that only the coordinator instance uses the base port, 1600.
- In this example, we specified data-dir-prefix values for each instance in the cluster. Once you create the folders, the **scidb.py initall** command creates symbolic links from the base-path to the directories that you specified for the data-dir-prefix parameter in the config.ini file. Note the following:
  - Specifying values for data-dir-prefix is optional. If you do not specify a data-dir-prefix value for an instance, then SciDB creates a folder using the default naming scheme—this will be a sub-folder of the base-path.
  - If you **do** specify a data-dir-prefix value for an instance, then you must ensure that the specified folder exists, and that it is completely empty. Otherwise, you will get errors when you try to start SciDB.

2.8.4. 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
```

2.8.5. 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

2.8.6. SciDB Python Script Parameters

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

```
scidb.py command scidb_cluster_name 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 '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 *`scidb_cluster_name`* 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/14.3/etc/config.ini`. If you want to use a custom configuration file for a particular SciDB cluster, use the *`conffile`* argument.

Chapter 3. Getting Started with SciDB Development

The `iquery` executable is the basic command-line tool for communicating with SciDB. `iquery` is the default SciDB client used to issue AQL and AFL commands. Start the `iquery` client by typing `iquery` at the command line when a SciDB session is active:

```
$ scidb.py startall scidb_cluster_name
$ 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 and AFL statements end with a semicolon (;). If you forget the terminating semicolon, `iquery` presents a prompt (**CON>**) where you can enter the terminating semicolon and run the query.

3.1. Using the iquery Client

To see the internal `iquery` commands reference, type `help` at the AQL or AFL 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. This is the default output format.
dense	Ideal for viewing 2-dimensional arrays. 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**
`-w [ --precision ]` *`precision`*	Precision for printing floating point numbers. Default is 6.
`-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 where result data will be stored.
`-o [ --format ]` *`format`*	Output format. See the previous table for available options. Default is 'auto'.
`-v [ --verbose ]`	Print the debugging information. Disabled by default. Useful for displaying data such as number of chunks and chunk size.
`-t [ --timer ]`	Query setup time (in seconds).
`-n [ --no-fetch ]`	Skip data fetching. Disabled by default. Useful when the result array is large, and you do not want to have the results displayed on screen.
`-a [ --afl ]`	Switch to AFL query language mode. Default is AQL.
`-u [ --plugins ]` *`path`*	Path to the plugins directory.
`-h [ --help ]`	Show help.
`-V [ --version ]`	Show SciDB version information.
`ignore-errors`	Ignore execution errors in batch mode.

Note the following:

- The `iquery` interface is case sensitive.
- When `iquery` outputs string attributes, the strings are wrapped in single quotes.
- The maximum length of a query is 16MB.

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+, dcsv, lcsv+, sparse, and lsparse. The default format is dcsv.
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. Running iquery from any computer

You can run the iquery command from any computer, not just the SciDB coordinator server. This section describes how to do so.

Note the following:

- Your version of iquery must match the version of SciDB. For example, if your SciDB installation is at version 13.12, you must download and install iquery version 13.12.
- Your client OS does not need to match the OS of the SciDB coordinator server. For example, you can install the iquery client on computer running Ubuntu, and access a SciDB coordinator server that is running CentOS.

Perform the following steps from the computer on which you wish to run iquery:

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

   - On Ubuntu:

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

   - On CentOS or RHEL:

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

2. Add the SciDB repository to the repository folder, by creating the configuration file:

   - On Ubuntu, create the file `/etc/apt/sources.list.d/scidb.list` and add the following line:

     ```
     deb  http://downloads.paradigm4.com/  ubuntu12.04/14.3/
     ```

   - On CentOS or RHEL, create the `/etc/yum.repos.d/scidb.repo` file and add the following lines:

     ```
     [scidb]
     name=SciDB
     baseurl=http://downloads.paradigm4.com/centos6.3/14.3/
     gpgcheck=1
     ```

3. Run the appropriate update and package install commands for your OS:

   - On Ubuntu:

     ```
     sudo apt-get update
     sudo apt-get install scidb-14.3-utils
     ```

   - On CentOS or RHEL:

     ```
     sudo yum update
     sudo yum install scidb-14.3-utils
     ```

4. Set the SciDB environment variables in the user environment.

   ```
   export SCIDB_VER=14.3
   export PATH="$PATH:/opt/scidb/$SCIDB_VER/bin:/opt/scidb/$SCIDB_VER/share/scidb"
   export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/opt/scidb/$SCIDB_VER/lib"
   ```

5. When you run iquery, make sure to specify the correct host (the computer running the SciDB coordinator node). For example, if your SciDB coordinator server is **shared-c6-c2-vm1** (and is running on the default `base-port` of 1239), you could start an iquery session by running the following command:

   ```
   $ iquery --host shared-c6-c2-vm1
   ```

3.4. Running multiple versions of SciDB

You can run multiple versions of SciDB on the same SciDB server. This section discusses some guidelines for doing this.

- You can use the default name for all SciDB configuration files: **config.ini**. Since they each reside in **/opt/scidb/***`version`***/etc**, they can coexist on the same server.

- Edit the following configuration parameters:
  - The name of the SciDB database must be different for each configuration.
  - The **db_user** and **db_passwd** values should be different for each configuration.
  - The **base-path** values must be set to different folders.
  - The **base-port** parameter must be different for each configuration. **Tip:** do not use the default port (1239); using non-default values for both configurations makes it harder to accidentally use the wrong configuration, as you will need to always explicitly specify a port.
- You can create an alias or shell script as a shortcut for invoking the appropriate iquery command.

As an example, suppose that we want to run SciDB 13.3 and 13.6 on the same server. Consider the following simple config.ini files and aliases:

- The following config.ini is in /opt/scidb/13.3/etc:

```
[db_one]
server-0=localhost,0
db_user=one_133
db_passwd=one_133
install_root=/opt/scidb/13.3
pluginsdir=/opt/scidb/13.3/lib/scidb/plugins
logconf=/opt/scidb/13.3/share/scidb/log4cxx.properties
base-path=/home/scidb/one-13.3
base-port=1259
```

- The following config.ini is in /opt/scidb/13.6/etc:

```
[db_two]
server-0=localhost,0
db_user=one_136
db_passwd=one_136
install_root=/opt/scidb/13.6
pluginsdir=/opt/scidb/13.6/lib/scidb/plugins
logconf=/opt/scidb/13.6/share/scidb/log4cxx.properties
base-path=/home/scidb/two-13.6
base-port=1279
```

- You could create the following aliases (add them to your .bashrc or .bash_aliases file):

```
alias iquery13.3=/opt/scidb/13.3/bin/iquery -p 1259
alias iquery13.6=/opt/scidb/13.6/bin/iquery -p 1279
```

  You could then type **iquery13.3** or **iquery13.6** at the command line to invoke the corresponding iquery client.

3.5. 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 Chapter 5, *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 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 the default format (**dcsv**) to a file called `/tmp/random_values.dcsv`:

   ```
   iquery -r /tmp/random_values.dcsv -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 `tests/harness/testcases/data/doc` 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
      < scidb_root/tests/harness/testcases/data/doc/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.

   After running this command, you will have the following SciDB-formatted text file, **datafile.scidb**:

   ```
   0}[
   ("Truck", 23.5),
   ("Sedan", 48.7),
   ("SUV", 19.6),
   ("Convertible", 26.8)
   ]
   ```

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')"
   ```

```
type,mpg
'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)"
```

```
{x} type,mpg,kpl
{0} 'Truck',23.5,9.98985
{1} 'Sedan',48.7,20.7024
{2} 'SUV',19.6,8.33196
{3} '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 < scidb_root/tests/harness/testcases/data/doc/
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)"
```

```
{x} make,type,price,type,mpg,kpl
{0} 'Handa','Truck',26700,'Truck',23.5,9.98985
{1} 'Tolona','Sedan',31000,'Sedan',48.7,20.7024
{2} 'Gerrd',' SUV',42000,'SUV',19.6,8.33196
{3} '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)"
```

```
{x} mpg,kpl
{0} 23.5,9.98985
{1} 48.7,20.7024
{2} 19.6,8.33196
{3} 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 -aq "join(storage,project(target_new,mpg,kpl))"
```

```
{x} make,type,price,mpg,kpl
{0} 'Handa','Truck',26700,23.5,9.98985
{1} 'Tolona','Sedan',31000,48.7,20.7024
{2} 'Gerrd',' SUV',42000,19.6,8.33196
{3} 'Maudi','Convertible',45000,26.8,11.3927
```

3.6. Reserved Words

In SciDB, there are keywords that are designated as reserved. You cannot use these words as identifiers (such as array names, dimension names, or attribute names). The lists are language-dependent.

The following words are reserved when using AFL.

```
and          array    as
asc          between  case
compression  create   default
desc         else     empty
end          false    if
is           not      null
or           reserve  then
true         when
```

All the AFL reserved words are also reserved words in AQL. In addition, AQL also has the following set of reserved words.

```
all         by           cancel
cross       current      drop
errors      fixed        following
from        group        insert
instance    instances    into
join        library      load
on          order        over
partition   preceding    query
redimension regrid       rename
save        select       set
shadow      start        step
thin        to           unbound
unload      update       values
variable    where        window
```

To avoid possible errors, we recommend that you avoid using **all** of the reserved words as identifiers—no matter which language you use. Some SciDB utilities may use both AQL and AFL, "under the covers." Thus, any AQL or AFL reserved word identifier could cause errors.

Chapter 4. SciDB Data Types and Casting

This chapter discusses the built-in SciDB data types, their behavior, and how to cast one type to another.

4.1. Standard Data Types

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; default is the empty 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

4.2. Temporal Data Types

This section lists the acceptable formats for the SciDB datatypes datetime and datetimetz.

These tokens represent portions of the date and time:

- **MON:** three-character month name: Jan, Feb, Mar, and so on. The three-character month name is case insensitive—any combination of lowercase and uppercase letters is acceptable.
- **mm:** month number: 01 for January, 02 for February, and so on. Note that you can omit the leading 0.
- **yyyy:** 4-digit year. If you specify a 2-digit year, SciDB prepends '20'. So, for years in the twenty first century, you need only use 2-digits to represent the year.
- **hour:** hour of the day. Can be 12- or 24-hour time

- **min:** minutes
- **sec:** seconds
- **frac:** fractional portion of a second; you can specify as many digits as you like
- **AMPM:** For one of the acceptable syntaxes, you must specify 'AM' or 'PM', to indicate the period of the day.

Table 4.1. Acceptable formats for the datetime datatype

Date/Time Syntax	**Example**
mm/dd/yyyy hour:min:sec	11/25/2009 16:11:19
dd.mm.yyyy hour:min:sec	25.11.2009 16:11:19
yyyy-mm-dd hour:min:sec	2009-11-25 16:11:19
yyyy-mm-dd hour.min.sec	2009-11-25 16.11.19
yyyy-mm-dd hour:min:sec.frac	2009-11-25 16:11:19.76
yyyy-mm-dd hour.min.sec.frac	2009-11-25 16.11.19.76
mm/dd/yyyy hour:min	11/25/2009 16:11
dd.mm.yy hour:min	25.11.2009 16:11
yyyy-mm-dd hour:min	2009-11-25 16:11
ddMONyyyy:hour:min:sec	25Nov2009:16:11:19
dd-MON-yyyy hour.min.sec AMPM	25-Nov-2009 4.11.19 PM
Date Syntax (no time element)	
yyyy-mm-dd	2009-11-25
mm/dd/yyyy	25.11.2009
dd.mm.yyyy	25.11.2009

For the datetimetz datatype, you can use any of the following syntaxes, where the offset (OFF) is between -13:59 and +13:59.

Table 4.2. Acceptable formats for the datetimetz datatype

Date/Time Syntax with offset	**Example**
mm/dd/yyyy hour:min:sec OFF	11/25/2009 16:11:19 +10:00
dd.mm.yyyy hour:min:sec OFF	25.11.2009 16:11:19 -9:15
yyyy-mm-dd hour:min:sec OFF	2009-11-25 16:11:19 +01:10
yyyy-mm-dd hour.min.sec OFF	2009-11-25 16.11.19 -5:22
yyyy-mm-dd hour:min:sec.frac OFF	2009-11-25 16:11:19.76 +6:10
yyyy-mm-dd hour.min.sec.frac OFF	2009-11-25 16.11.19.76 -11:05
dd-MON-yyyy hour.min.sec AMPM OFF	25-Nov-2009 4.11.19 PM +00:30

The following examples illustrate how to specify date/time values in SciDB.

1. Create arrays to hold date/time information.

   ```
   AFL% create array datetime1 <dt:datetime>[i=0:*,100,0];
   ```

```
AFL% create array datetime2 <dtz:datetimetz>[i=0:*,100,0];
```

2. Load array `datetime1` with values in the SciDB-formatted file, **dates**:

```
$ cat dates
```

```
[
 ("11/25/2009 16:11:19"),
 ("25.11.2009 16:11:19"),
 ("2009-11-25 16:11:19.7612"),
 ("2009-11-25 16.11.19.76"),
 ("2009-11-25 16.11.19"),
 ("2009-11-25 16:11:19"),
 ("11/25/2009 16:11"),
 ("25.11.2009 16:11"),
 ("2009-11-25 16:11"),
 ("25Nov2009:16:11:19"),
 ("25-Nov-2009 4.11.19 PM"),
 ("25-Nov-2009 4.11.19 AM"),
 ("2009-11-25"),
 ("11/25/2009"),
 ("25.11.2009"),
]
```

```
%AFL load(datetime1,'./dates');
```

```
i,dt
0,'2009-11-25 16:11:19'
1,'2009-11-25 16:11:19'
2,'2009-11-25 16:11:19'
3,'2009-11-25 16:11:19'
4,'2009-11-25 16:11:19'
5,'2009-11-25 16:11:19'
6,'2009-11-25 16:11:00'
7,'2009-11-25 16:11:00'
8,'2009-11-25 16:11:00'
9,'2009-11-25 16:11:19'
10,'2009-11-25 16:11:19'
11,'2009-11-25 04:11:19'
12,'2009-11-25 00:00:00'
13,'2009-11-25 00:00:00'
14,'2009-11-25 00:00:00'
```

3. Load array `datetime2` with values in the SciDB-formatted file, **dates_TZ** (containing date/time values with offsets):

```
$ cat dates_TZ
```

```
[
 ("11/25/2009 16:11:19 +10:01"),
 ("25.11.2009 16:11:19 +09:02"),
 ("2009-11-25 16:11:19.76 +08:03"),
 ("2009-11-25 16.11.19.76 +07:04"),
 ("2009-11-25 16.11.19 +06:05"),
 ("2009-11-25 16:11:19 +05:06"),
 ("25-Nov-2009 4.11.19 PM +00:11")
]
```

```
%AFL load(datetime1,'./dates_TZ');
```

```
i,dtz
0,'2009-11-25 16:11:19 +10:01'
1,'2009-11-25 16:11:19 +09:02'
2,'2009-11-25 16:11:19 +08:03'
3,'2009-11-25 16:11:19 +07:04'
4,'2009-11-25 16:11:19 +06:05'
```

```
5,'2009-11-25 16:11:19 +05:06'
6,'2009-11-25 16:11:19 +00:11'
```

4.3. Special Data Types

This section discusses some special values, as well as their semantics.

4.3.1. Special Values for Attributes

SciDB attributes and cells support the following types of values:

- **Ordinary values.** These lie within a defined range of representable values for the attribute type. For example, "123" is an ordinary value for the data type int64.
- **Null (missing) values.** These match the semantics of NULL in a relational database.

  Null values can contain a missing reason code. This allows applications to optionally specify multiple types of null values and treat each type differently. If you are loading data, you can specify a null value with a missing reason code by using "?*n*", where *n* is an integer from 0 to 127. There are two SciDB functions that are associated with missing value codes: `missing()` and `missing_reason()`. For more details, see Section 6.6, “Loading Data with Missing Values”.

  For clarity, throughout the document the term "null" is equivalent to "?0", that is, a missing value with missing reason code 0. All other missing values will be specified with the corresponding missing value code, e.g. "missing value code 100" will be represented by "?100".
- **Empty cells.** This is purely a SciDB concept. Typically, when you create an array that is very sparse, most of its cells will be empty. In most cases, SciDB treats empty cells like they simply do not exist. Consider:
  - Empty cells are not considered when evaluating count(*).
  - Empty cells are disregarded during join, sort, and unpack operations.
  - Empty cells do not participate in aggregations.
  - Some operators, such as the Linear Algebra operators, interpret empty cells as 0.

  Note that the SciDB concept of emptiness is only for entire cells, not for individual attributes. For more details, see Section 6.7, “Loading Empty Cells”.

Additionally, there are some floating-point values that behave specially. The following values are applicable to the SciDB data types double and float.

- **Infinity.** During calculations, it is possible to divide by 0 and store the result into an attribute value. SciDB represents this as either **inf** or **-inf**, depending on the sign of the dividend.
- **NaN.** "Not a Number." This behaves the same as the IEEE floating point special value of the same name. It is a floating-point value that does not actually represent a floating-point number. When arithmetic is performed, and one of the arguments is NaN, NaN is always the result. NaN cannot take part in join operations. The following operations can return NaN:
  - Operations with a NaN as at least one operand.
  - Indeterminate forms (such as divisions 0/0 and inf/inf).

- Real operations with complex results:
  - ■ square root of a negative number
  - ■ logarithm of a negative number
  - ■ inverse sine or cosine of a number that is less than -1 or greater than +1.

4.3.2. Semantics of Special Values

This section describes the behavior of the SciDB-specific special values **empty** and **missing**, and IEEE floating-point special value **NaN**, as well as **inf**/**-inf**,

Note the following representations:

- **?m** denotes missing with a non-negative integer *m* representing the missing code.
- Null is defined as **?0**, a special case of **missing**.
- **op()** denotes a unary or binary expression or function, such as **x + y**, **x < y**, or **sin(x)**.
- An *aggregator* is a function or operator that computes aggregates, such as **count()**.
- A *floating point value* is a SciDB attribute of type `float` or `double`. In this section, it is restricted to "normal" floating-point values (such 1.4, -3.75, and so on), plus **NaN**, **+inf**, and **-inf**, but does not include **empty** nor **missing**.

When **op()** is not an aggregator:

- If at least one operand is **empty**, the result is **empty**.
- If at least one of x or y is **missing**, but neither is **empty**: **op(x,y)** returns **?0**.
- If the value for a unary operator is **missing** but not **empty**, **op(?m)** returns **?m**.
- If at least one of the floating point values x or y is **NaN**:
  - When **op(x,y)** is a comparison expression: x {==,<,>,<=,>=} y returns FALSE, and x <> y returns TRUE
  - When **op(x,y)** is not a comparison expression: **op(x,y)** and **op(NaN)** both return **NaN**.

For aggregators:

- **Empty** is ignored during aggregation. That is, empty cells do not participate in aggregation.
- For details on how aggregates handle **missing**, see the documentation for the individual aggregate.
- **NaN** is included during aggregation. For instance, the sum of a set of double numbers is **NaN**, if at least one of the numbers is **NaN**.

Whenever possible, SciDB computations adhere to the IEEE Standard for Floating-Point Arithmetic (IEEE 754). Exceptions are documented in the appropriate place for the specific operator or function.

For the **sort()** operator, note the following sort order regarding special values:

```
?m < NaN < x
```

where:

- '<' indicates "comes before"
- For **missing** values, the sort order is based on the value of the missing code. Thus, **?0** comes before **?10** which comes before **?72**.
- x represents any floating point number, other than **NaN**.

4.4. Casting Between Data Types

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. 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 to derive an attribute of the correct type.

1. Create an array that contains an integer attribute.

   ```
   AFL% store(build(<a1: int32>[i=0:0,1,0],2),A);
   ```

   ```
   [(2)]
   ```

2. Convert the integer values to doubles.

   ```
   AFL% apply(A, a2, double(a1));
   ```

   ```
   {i} a1,a2
   {0} 2,2
   ```

   This generates a new attribute, a2, with double data type from a1.

3. A numeric data type can also be converted to string, which returns a UTF-8 encoded string.

   ```
   AFL% apply(A, a2, string(a1));
   ```

   ```
   {i} a1,a2
   {0} 2,'2'
   ```

4.5. Casting Between Temporal Data Types

SciDB includes functions for converting 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:

```
AFL% create array T<td: datetime>[i=0:0,1,0];
```

```
AFL% store(build(T, now()), T);
```

```
[('2014-04-08 20:52:14')]
```

```
AFL% apply(T, dst, append_offset(td, 3600));
```

```
{i} td,dst
{0} '2014-04-08 20:52:14','2014-04-08 20:52:14 +01:00'
```

To append an offset and apply it to the time, use the `apply_offset` function, which converts a UTC datetime to local datetime by subtracting the offset timezone value, and stores the result as datetimetz including the offset timezone value. The offset must be expressed in seconds.

```
AFL% create array T1 <t:datetimetz>[i=0:0,1,0];
```

```
AFL% store(project(apply(T,dst,apply_offset(td,3600)),dst),T1);
```

```
{i} t
{0} '2014-04-08 21:52:14 +01:00'
```

To return the datetime portion of a datetimetz value, use the `strip_offset` function:

```
AFL% apply(T1,dst,strip_offset(t));
```

```
{i} t,dst
{0} '2014-04-08 21:52:14 +01:00','2014-04-08 21:52:14'
```

To apply the offset to the datetime and return a GMT datetime, use the `togmt` function:

```
AFL% apply(T1,dst,togmt(t))
```

```
{i} t,dst
{0} '2014-04-08 21:52:14 +01:00','2014-04-08 20:52:14'
```

For more details on the SciDB temporal functions, see Temporal functions.

Chapter 5. 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 cells of the array. 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 *size* of the dimension.

5.1. SciDB Array Schema Syntax

The following items provide a precise syntax of a SciDB array schema. A schema can be used in a variety of operators, such as `cast()`, `build()`, `redimension()`, and so on. Here, we discuss schemas within the context of creating arrays.

```
create_array_statement ::= CREATE ARRAY array_name schema
schema                 ::= < attributes > [ dimensions ]
```

To create an array, you specify the key words **CREATE ARRAY**, followed by a name and a schema. The schema comprises a list of attributes and a list of dimensions.

Note

The **CREATE ARRAY** keywords are allowed in both AFL and AQL.

The array attributes describe the types of data contained in each cell of the array. Each attribute is defined as follows:

```
attribute ::= attribute_name : attribute_type nullable default
nullable  ::=
                NULL
              | NOT NULL
              |
default   ::=
                DEFAULT default_value
              |
```

Each attribute consists of:

- **A name:** The array name uniquely identifies the array in the database. The maximum length of an array name is 1024 bytes. Array names may contain only the alphanumeric characters and underscores (_).
- **Its data type:** One of the data types supported by SciDB (can also be a user-defined type). Use the `list('types')` command to see the list of available data types.

- **Nulls allowed (Optional):** Users can specify 'NULL' to indicate attributes that are allowed to contain null values. If this keyword is not used, all values must be non null, that is, 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.
- **A default value (Optional):** Users can specify the value to be automatically substituted when no value has been explicitly supplied for a non NULL attribute. If unspecified, substitution uses system defaults for various types (0 for numeric types and "" for string). Note that if the attribute is declared as allowing nulls, this clause is ignored.

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. Each dimension is defined as follows:

```
dimension    ::=   dimension_name
                 | dimension_name = expression : dimension_hi , expression , expression
dimension_hi ::=   expression
                 | *
```

Each dimension consists of:

- **A name.** If you specify only the name, SciDB uses defaults for the chunk size and overlap, and makes the dimension unbounded. If not using defaults, you must specify the remaining dimension characteristics.

  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. Dimension names may only contain alphanumeric characters and underscores (_)
- **A starting value:** An expression for the dimension start value.
- **An ending value:** Either an expression or an asterisk (*) can be supplied as the dimension end value. An asterisk indicates the dimension has no set size (referred to as an *unbounded dimension*). Together, the starting and ending values define the range of possible values that the dimension coordinate can take. This range includes both the starting and ending values themselves. For example, [1,1000] defines a dimension size of 1000.
- **A chunk size:** The number of elements per chunk. An expression for the chunk size. Note that this functions as a maximum value, as arrays can be sparse, and thus contain many fewer values than the number specified here.
- **A chunk overlap:** The number of overlapping dimension-index values for adjacent chunksAn expression for the chunk overlap.

Expressions must evaluate to a scalar value. For example, an expression might be **100**, or **500 * 4**, and so on. Additionally, expressions can consist of environment variables—so long as they get evaluated in the shell, and then passed into SciDB. For details, see Section 5.5.4, “Environment Variables for Dimensions”.

As an example, this AQL statement creates an array:

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

The created array consists of the following:

- Array name, `A`
- An array schema with the following characteristics:
  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.

5.2. Creating a sparse array

Sparse arrays contain empty cells, as well as cells that contain data. In SciDB, you can use a combination of the operators redimension, apply, and build to create a sparse array. This section contains a few examples.

In this example, we create an array that has a random value every 50 cells (all other cells are empty).

1. Create an array to hold the eventual dimension index.

```
AFL% store(build(<index:int64>[i=0:9,10,0], i*50), A);
```

```
[(0),(50),(100),(150),(200),(250),(300),(350),(400),(450)]
```

2. Create an array to hold the values.

```
AFL% store(build(<val:double>[i=0:9,10,0], random()%2000/2000.0), B);

```

```
[(0.9505),(0.1395),(0.68),(0.462),(0.504),(0.168),(0.1205),(0.0505),(0.5955),
(0.842)]
```

3. Combine `A` and `B` into a single array.

```
AFL% store(join(A,B), test);
```

```
{i} index,val
{0} 0,0.9505
{1} 50,0.1395
{2} 100,0.68
{3} 150,0.462
{4} 200,0.504
{5} 250,0.168
{6} 300,0.1205
{7} 350,0.0505
{8} 400,0.5955
{9} 450,0.842
```

4. Finally, redimension `test` into a sparse array.

```
AFL% redimension(test, <val:double>[index=0:*, 50,0]);
```

```
{index} val
{0} 0.9505
{50} 0.1395
{100} 0.68
{150} 0.462
{200} 0.504
{250} 0.168
{300} 0.1205
{350} 0.0505
{400} 0.5955
{450} 0.842
```

Suppose you want to create a 3x3, 2-dimensional array, where you store the value 5.5 in all cells where the sum of the two dimensions is greater than 4.

```
If i+j > 4, store 5.5; else empty
```

The array looks like this:

```
[
[(),(),()],
[(),(),(5.5)],
[(),(5.5),(5.5)]
]
```

This query creates the array:

```
AFL% redimension(
         apply
            (build(<v:double>[i1=1:3,3,0,j1=1:3,3,0],5.5),
         i, iif(i1+j1>4,i1,null), j, iif(i1+j1>4,j1,null)),
       <v:double>[i=1:3,3,0,j=1:3,3,0]);
```

```
{i,j} v
{2,3} 5.5
{3,2} 5.5
{3,3} 5.5
```

We first build an array where all cells contain the value **5.5**. Then we apply the condition that we want—the sum of the two dimensions must be greater than 4. This has the effect of nullifying all the dimension values that we don't want in the final, sparse array. When we use the `redimension` operator, the correct set of dimensions is created to hold the attribute value.

In general, assume you want to build a sparse array with the following characteristics:

- a **schema**, consisting of a single attribute (the `build` operator takes exactly one attribute), and one or more dimensions,
- a **value**, to place into all non-empty cells, and
- a **condition**, used to determine which cells are empty and which are filled with **val**.

To build such an array, construct the following query:

```
redimension
(
   apply
   (
      build(schema_with_altered_dim_names, value),
      dim1, iff(condition, dim1_altered, null),
      ...
   ),
   schema
)
```

5.3. Deleting an Array

You can remove an array and its attendant schema definition from the SciDB database. You can do this using either AQL or AFL.

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

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

To delete an array with AFL, use the `remove` operator:

```
AFL% remove(default_1)
```

If you want to remove some versions of your arrays to save space, see Section 9.3, “Array Versions”.

5.4. 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];
```

5.4.1. NULL and Default Attribute Values

SciDB offers functionality to work with missing data. This functionality includes special handling for empty cells, null values, and default values.

Consider the data set m4x4_missing.scidb, located in the /tmp folder and 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 empty. You can tell SciDB how you want to handle the missing data with various array options.

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, '/tmp/m4x4_missing.scidb');
```

```
[
[(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, '/tmp/m4x4_missing.scidb');
```

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

5.4.2. 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.

Additionally, you can specify a missing reason code for the default value of an attribute. For example, the following statement specifies a missing reason code of 30 as the default value for the `val3` attribute.

```
AFL% create array D<val1:char default 'a', val2:int32 default 1,
     val3:int64 null default missing(30)> [i=0:9,10,0];
```

```
[('D<val1:char DEFAULT \'a\',val2:int32 DEFAULT 1,val3:int64 NULL DEFAULT ?30>
 [i=0:9,10,0]')]
```

5.4.3. 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.

5.5. 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 and unbounded dimensions. Additionally, you can use expressions to define dimension size, or let SciDB assign default values.

Note

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

5.5.1. 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. Specifying an overlap causes SciDB to store adjoining cells in each dimension from the *overlap area* in 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.

5.5.2. 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];
```

5.5.3. Dimension Defaults

You can specify only the dimension name, and SciDB will create defaults for the starting index, chunk size, and chunk overlap. The dimension is created as an unbounded dimension (that is, there is no ending index specified).

This statement uses default values for the dimensions (default starting index, chunk size, and overlap):

```
AQL% CREATE ARRAY default_1 <val:double>[i];
```

```
AFL% show(default_1)
```

```
default_1

< val:double >

[i=0:*,500000,0]
```

Array `default_1` has one dimension named `i`, and was created with the defaults for all of the dimension values. The default chunk size is set based on the number of dimensions that do not have a specified chunk size. SciDB attempts to set the total chunk size to approximately 500,000, which is the product of all the dimensions.

Here is an example where we create three dimensions, two using the defaults:

```
AQL% CREATE ARRAY default_2 <val:double>[i=0:999,200,0, j,k];
```

```
AFL% show(default_2)
```

```
default_2

< val:double >

[i=0:999,200,0,
j=0:*,50,0,
k=0:*,50,0]
```

Array `default_2` has three dimensions: multiply the chunk size for each of the dimensions, and the product is 500,000: 50x50x200.

5.5.4. Environment Variables for Dimensions

You may encounter cases where the characteristics of your dimensions are based on environment variables. In these cases, you can use expressions for the dimension characteristics.

This works only when you are passing queries to SciDB through the Linux shell—the shell parses the environment variables and supplies SciDB with a scalar expression that it can evaluate. SciDB itself knows nothing about the Linux environment variables.

The expressions must evaluate to constants outside of SciDB: environment variables meet this requirement.

This example sets some environment variables, and then uses them to calculate the characteristics for the dimension.

```
$ export FIRST=10
$ export MULTIPLE=50
$ export CHNKSIZE=100
```

```
$ iquery -aq "create array expr_1 <v:double> [row=$FIRST:$FIRST+$MULTIPLE*$CHNKSIZE,
$CHNKSIZE,0]"
```

```
$ iquery -aq "show(expr_1)"
```

```
expr_1 < v:double > [row=10:5010,100,0]
```

5.6. 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.

```
AFL% CREATE ARRAY winners <person:string, time:double, event:string>
 [year=1996:2008,1000,0];
```

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

```
winners

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

[year=1996:2008,1000,0]
```

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,
event:string >

[year=1996:2008,1000,0]
```

This query creates an array `winnerGrid` that has renamed attributes `LastName` and `elapsedTime` and dimension `Year`.

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

```
{x} LastName,elapsedTime,event
{1996} 'Bailey',9.84,'dash'
{2000} 'Greene',9.87,'dash'
{2004} 'Gatlin',9.85,'dash'
{2008} 'Bolt',9.69,'dash'
```

```
winnerGrid

< LastName:string,
elapsedTime:double,
event:string >

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

5.7. Database Design

This section discusses some general guidelines around choosing the dimensions, attributes, and chunk sizes for your SciDB arrays. This section contains the following information:

- Selecting Dimensions and Attributes
- Selecting Chunk Sizes

5.7.1. 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.

- Variables that constitute a key are good candidates for being dimensions. Dependent variables are good candidates for being attributes. Variables l that have low distinct counts are usually better as attributes than as dimensions. Variables that are ordered make good dimensions.

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 10, *Changing Array Schemas*.

5.7.2. Selecting Chunk Size

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 10, *Changing Array Schemas*.

Chapter 6. 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.

This chapter contains the following sections:

6.1. 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 5.1, "SciDB Array Schema Syntax".

   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.

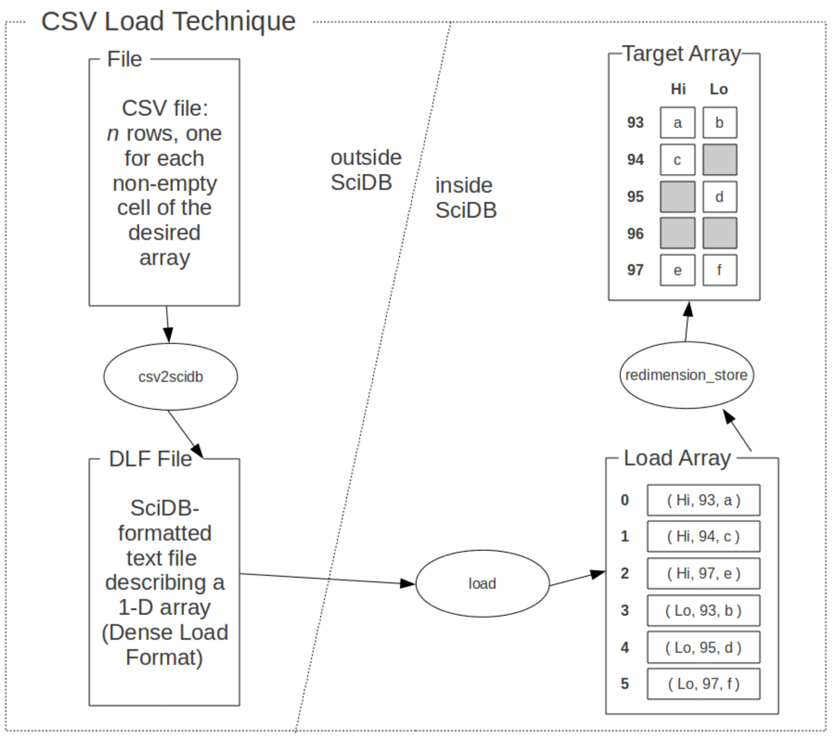
4. Rearrange the loaded data into the target array; the multi-dimensional array that supports your analytics.

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.

6.2. 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 6.1. 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.

6.2.1. Visualize the Target Array

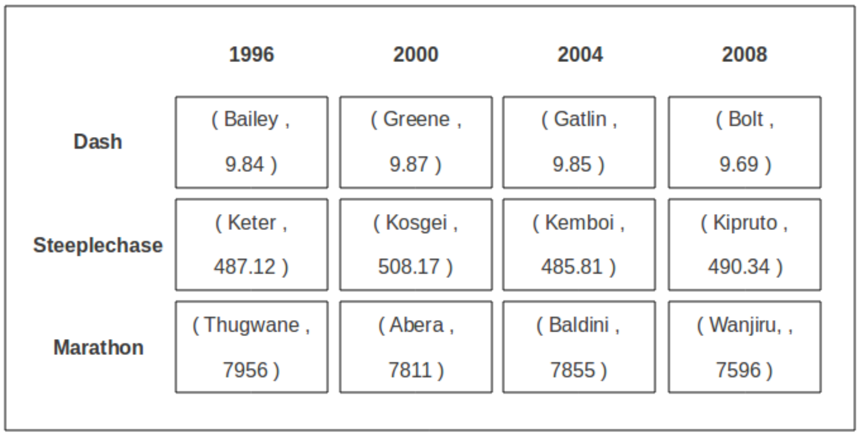
When using the CSV loading technique, visualizing the desired SciDB array means the following:

- 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 6.2. Example of 2-dimensional array with 2 attributes



	1996	2000	2004	2008
Dash	(Bailey , 9.84)	(Greene , 9.87)	(Gatlin , 9.85)	(Bolt , 9.69)
Steeplechase	(Keter , 487.12)	(Kosgei , 508.17)	(Kemboi , 485.81)	(Kipruto , 490.34)
Marathon	(Thugwane , 7956)	(Abera , 7811)	(Baldini , 7855)	(Wanjiru, , 7596)

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.

6.2.2. 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: Convert CSV file to SciDB input text format.
Usage:   csv2scidb [options] [ < input-file ] [ > output-file ]
Default: -f 0 -c 1000000 -q
Options:
  -v        version information
  -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: defaults to ,
  -p STR    type pattern: N number, S string, s nullable-string,
            C char, c nullable-char
  -q        quote the input line exactly by wrapping it in ()
  -s N      skip N lines at the beginning of the file
  -h        prints this helpful message

Note: the -q and -p options are mutually exclusive.
```

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

```
csv2scidb -s 1 -p SNSN < $DOC_DATA/olympic_data.csv > $DOC_DATA/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 $DOC_DATA/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.

6.2.3. 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 '../tests/harness/testcases/data/doc/olympic_data.scidb';
```

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

6.2.4. 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 query creates the target array:

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

The result of that query is an array that can accommodate the data about Olympic winners.

We then use the operators `uniq()` and `index_lookup()` to create an index array:

```
AFL% create array event_index <event:string>[event_id=0:*,10,0];
```

```
AFL% store(uniq(sort(project(winnersFlat,event)),'chunk_size=10'),event_index);
```

```
{event_id} event
{0} 'dash'
{1} 'marathon'
{2} 'steeplechase'
```

To populate this array with the data, run the following query:

```
AFL% store(redimension
        (project
           (index_lookup(winnersFlat,event_index,winnersFlat.event, event_id),
         year,person,time,event_id),winners),
       winners);
```

```
{year,event_id} person,time
{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
```

The AQL equivalent query uses the **`SELECT ... INTO`** syntax:

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

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

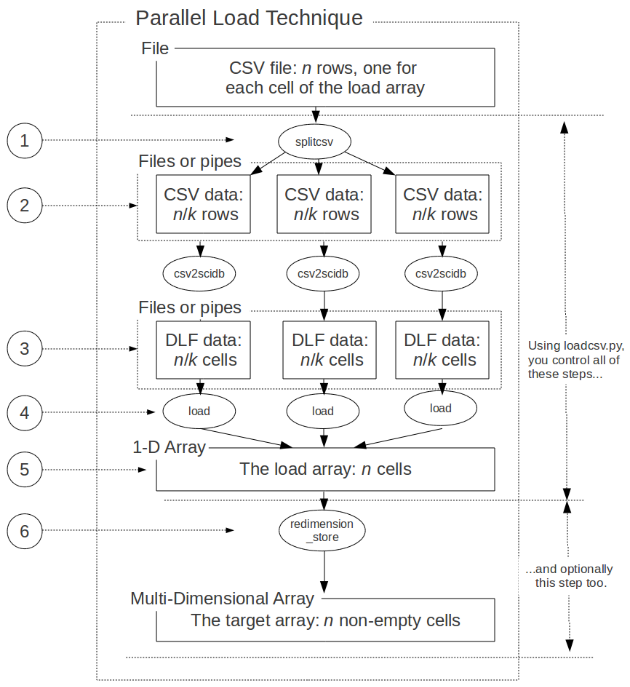
```
{year,event_id} person,time
{1996,0} 'Bailey',9.84
{1996,1} 'Keter',487.12
{1996,2} 'Thugwane',7956
{2000,0} 'Greene',9.87
{2000,1} 'Kosgei',503.17
{2000,2} 'Abera',7811
{2004,0} 'Gatlin',9.85
{2004,1} 'Kemboi',485.81
{2004,2} 'Baldini',7855
{2008,0} 'Bolt',9.69
{2008,1} 'Kipruto',490.34
{2008,2} 'Wanjiru',7596
```

6.3. Loading in Parallel

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 6.3. 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.

6.3.1. 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, data type, 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 data type.

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.

6.3.2. 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 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'
```

6.3.3. 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.

6.3.4. Loading HDFS Data

HDFS is a distributed file system written in Java for the Hadoop framework. HDFS can store CSV files, but they are not directly visible to Linux systems. To load HDFS data into SciDB, you can circumvent this restriction by piping the contents of the CSV file directly into the SciDB script for parallel loading of CSV files, **loadcsv.py**.

On a normal Linux file system, you would execute a command similar to the following:

```
$ cat test.csv | loadcsv.py option_list
```

The analogous command on an HDFS file system is the following:

```
$ hadoop dfs -cat test.csv | loadcsv.py option_list
```

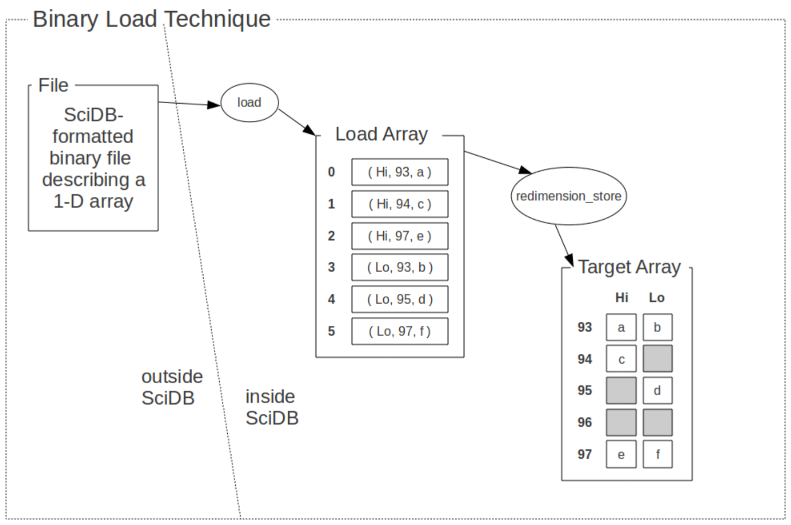
Note the following:

- `hadoop dfs -cat` is the Hadoop command to list a file to STDOUT.
- The options for loadcsv.py are described in Section 6.3.2, “Load the Data”.

6.4. 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 6.4. 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.

6.4.1. Visualize the Target Array

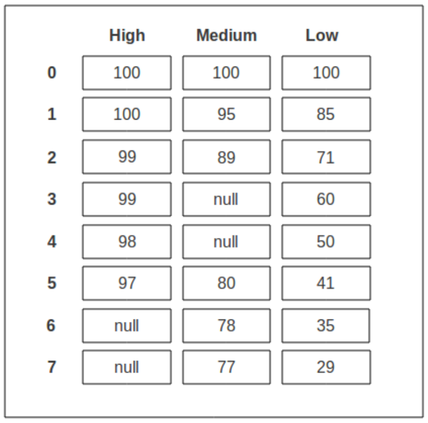
When using the binary loading technique, visualizing the desired multi-dimensional array means the following:

- 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 6.5. 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.

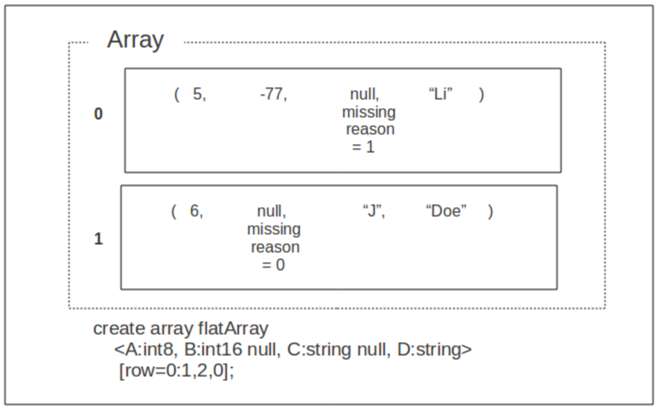
This simple, 24-cell array will be the target array used to illustrate steps of the binary load technique.

6.4.2. 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.

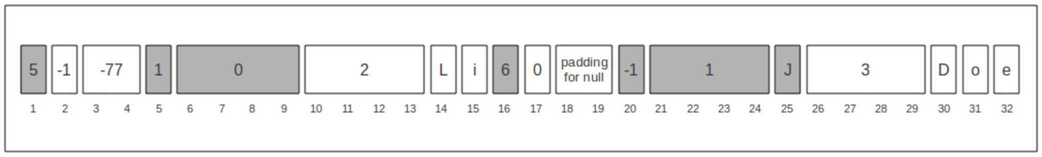
Figure 6.6. Example array created using binary load


Array
0
(5, -77, null, missing reason = 1 "Li")
1
(6, null, missing reason = 0 "J", "Doe")

```
create array flatArray
   <A:int8, B:int16 null, C:string null, D:string>
    [row=0:1,2,0];
```

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

Figure 6.7. 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 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 data type 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 data type 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 data type 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');
```

6.4.3. 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 '../tests/harness/testcases/data/doc/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.

6.4.4. 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 intensityFlat;
```

```
{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
```

6.4.5. 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.

Currently, only data that is of type int64 can be converted into a SciDB dimension. We can create an index array for the exposure values, and then store the IDs into the redimensioned array. The following query creates the exposure_index array:

```
AFL% store(uniq(sort(project(intensityFlat,exposure)),'chunk_size=10'),exposure_index);
```

```
{exposure_id} exposure
{0} 'High'
{1} 'Low'
{2} 'Medium'
```

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

```
AFL% CREATE ARRAY intensity
      <measuredIntensity:int64 null>
      [elapsedTime=0:40000,10000,0, exposure_id=0:2,3,0];
```

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

```
AFL% store(redimension
         (project
           (index_lookup
               (intensityFlat,exposure_index,intensityFlat.exposure, exposure_id),
         elapsedTime,measuredIntensity,exposure_id),
      intensity),intensity);
```

The AQL equivalent query uses the **`SELECT ... INTO`** syntax:

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

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

6.4.6. 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 '../tests/harness/testcases/data/doc/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 '../tests/harness/testcases/data/doc/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 '../tests/harness/testcases/data/doc/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.

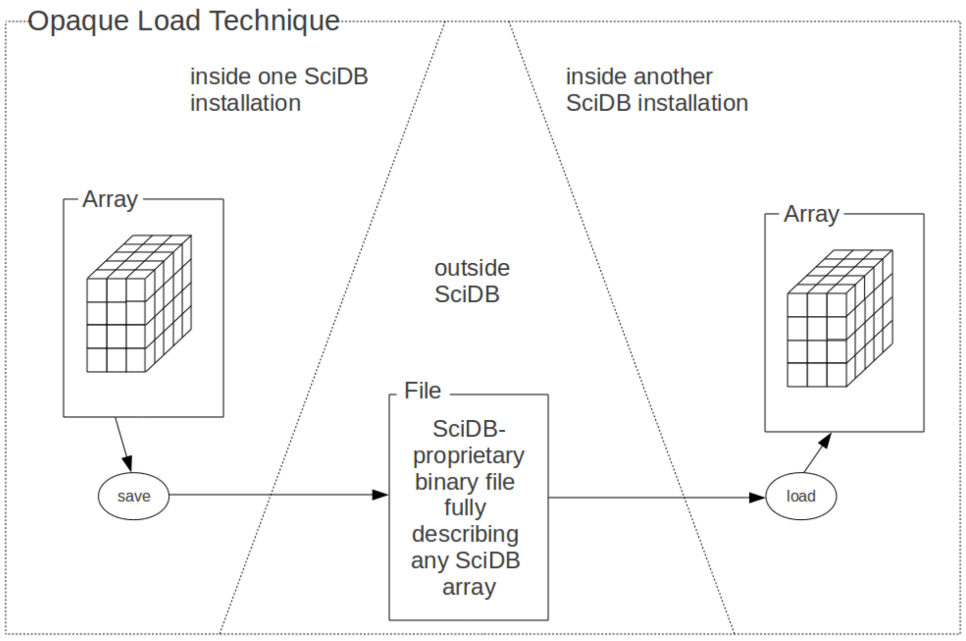
Note

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

6.5. Transferring Data Between SciDB Installations

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 6.8. Overview of opaque load technique


Opaque Load Technique
inside one SciDB installation
inside another SciDB installation
Array
outside SciDB
Array
File
SciDB-proprietary binary file fully describing any SciDB array
save
load

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.

Note that the opaque format is *not generally compatible between versions of SciDB*. Data saved using the opaque format should only be re-loaded into the same SciDB database version with identical database configuration parameters.

6.5.1. 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.

6.5.2. 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
```

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.

6.5.3. 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 >

[elapsedTime=0:40000,10000,0,
exposure_id=0:2,3,0]
```

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 >
         [duration=0:40000,10000,0,
          exposure_id=0:3,3,0]
```

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 '../tests/harness/testcases/data/doc/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.

6.6. Loading 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 '../tests/harness/testcases/data/doc/v4.scidb'
```

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

The out-of-the-box default value for each datatype is described in Chapter 4, *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 '../tests/harness/testcases/data/doc/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 '../tests/harness/testcases/data/doc/v4_null.scidb';
```

6.7. Loading 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 '../tests/harness/testcases/data/doc/v4.scidb'
```

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

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

6.8. 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 $DOC_DATA/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 '../tests/harness/testcases/data/doc/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 7. Basic Array Tasks

This chapter contains the following sections:

7.1. 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.

7.1.1. 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.

7.1.2. 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()` operator returns an array result containing an array's schema. To see the entire schema, use a **SELECT** statement that contains the `show` operator:

```
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.

7.2. 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`:

```
AFL% show(m3x3)
```

```
m3x3

< a:double >

[i=1:3,3,0,
j=1:3,3,0]
```

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

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)]
]
```

7.3. 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),(),(),(),(),()]
```

7.4. 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.

7.5. 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 8. SciDB Basic Analytic Capabilities

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.

8.1. 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 15, *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:int32 >

[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:int32,
attr2:int32 >

[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)]
]
```

8.1.1. 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:

```
m4x4_max_min

< max:int32 NULL DEFAULT null,
min:int32 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 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`:

```
AFL% store(build(m4x4_null,iif(i=2,0,null)),m4x4_null);
```

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

The rules for aggregates concerning missing values and empty cells:

- All aggregate operators and functions exclude empty cells.
- Except for `count(*)`, aggregate operators and functions exclude missing values.
- `count(*)` includes missing values.

8.1.2. 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:int32 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
```

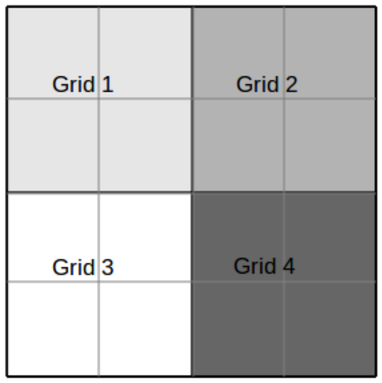
The AFL `aggregate` operator takes dimension arguments to support group-by functionality. This query selects the largest value from each column `y` from the array `m4x4` using AFL:

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

```
{y} attr1_max
{0} 12
{1} 13
{2} 14
{3} 15
```

8.1.3. 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:int32 NULL DEFAULT null,
min_1:int32 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:

```
AFL% regrid(m4x4, 2, 2, max(attr1), min(attr1));
```

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

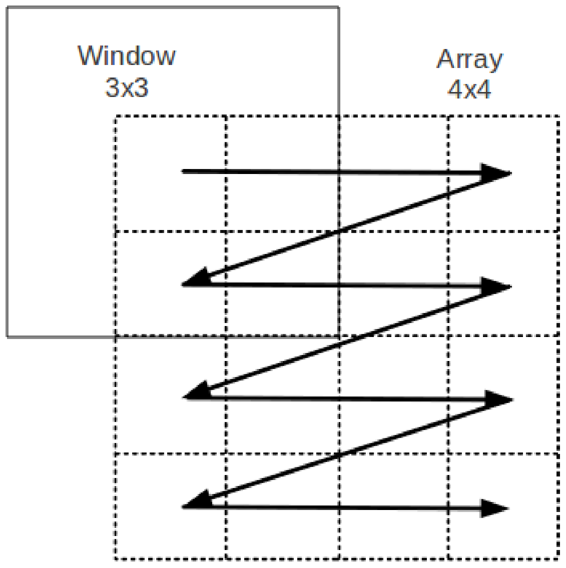
8.1.4. Window Aggregates

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:int64 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),()]
]
```

8.1.5. 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 operator supports this type of usage. The redimension operator is used to transform a source array into a result array with different 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 an array describing some recent Olympic champions, and you want to produce an array that shows the gold-medal count for each country.

The schema of the array, **winners**, appears below:

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

```
winners

< event:string,
person:string,
year:int64,
time:double,
country:string,
country_id:int64 >

[i=0:*,144,0]
```

To examine the data in the **winners** array, use the following AFL statement:

```
AFL% scan(winners);
```

```
{i} event,person,year,time,country,country_id
{0} 'dash','Bailey',1996,9.84,'Canada',0
{1} 'dash','Greene',2000,9.87,'USA',5
{2} 'dash','Gatlin',2004,9.85,'USA',5
{3} 'dash','Bolt',2008,9.69,'Jamaica',3
{4} 'steeplechase','Keter',1996,487.12,'Kenya',4
{5} 'steeplechase','Kosgei',2000,503.17,'Kenya',4
{6} 'steeplechase','Kemboi',2004,485.81,'Kenya',4
{7} 'steeplechase','Kipruto',2008,490.34,'Kenya',4
{8} 'marathon','Thugwane',1996,7956,'USA',5
{9} 'marathon','Abera',2000,7811,'Ethiopia',1
{10} 'marathon','Baldini',2004,7855,'Italy',2
{11} 'marathon','Wanjiru',2008,7596,'Kenya',4
```

To create the schema for the desired array, execute the following statement:

```
AFL% CREATE ARRAY perCountryMedalCount <country:string, medalCount: uint64 null>
 [country_id=0:*,20,0];
```

Notice that the sole dimension is the country ID (**country_id**), 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% store(redimension(winners, perCountryMedalCount, count(*) as
 medalCount),perCountryMedalCount);
```

```
{country_id} country,medalCount
{0} 'Canada',1
{1} 'Ethiopia',1
{2} 'Italy',1
```

```
{3} 'Jamaica',1
{4} 'Kenya',5
{5} 'USA',3
```

The result of this redimension operation is the desired array. Notice that the sum of the counts is 12, the number of nonempty cells in the **winners** array.

8.2. Aggregation using Cumulate

The cumulate operator accumulates values along an array dimension and calculates the specified aggregate function.

You can calculate the cumulative sum of an array attribute, along a particular dimension with the cumulate operator. For example, consider the array `m4x4`, with values and schema as follows:

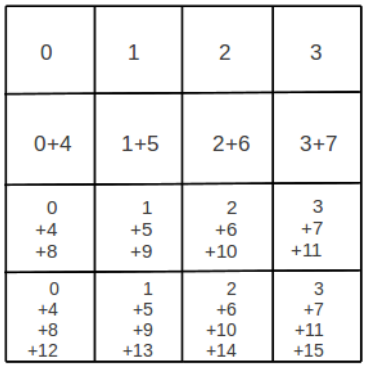
```
AFL% show(m4x4)
```

```
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)]
]
```

The `cumulate` operator accumulates the values along a selected dimension and aggregates those values. For example, if you choose the first dimension, `i`, `cumulate` aggregates the values along `i` as shown in the following figure (assuming the aggregation function is **sum**):



0	1	2	3
0+4	1+5	2+6	3+7
0 +4 +8	1 +5 +9	2 +6 +10	3 +7 +11
0 +4 +8 +12	1 +5 +9 +13	2 +6 +10 +14	3 +7 +11 +15

```
AFL% cumulate(m4x4,sum(val),i);
```

```
[
[(0),(1),(2),(3)],
[(4),(6),(8),(10)],
[(12),(15),(18),(21)],
[(24),(28),(32),(36)]
]
```

If you specify the dimension j, cumulate sums along j:

```
AFL% cumulate(m4x4,sum(val),j);
```

```
[
[(0),(1),(3),(6)],
[(4),(9),(15),(22)],
[(8),(17),(27),(38)],
[(12),(25),(39),(54)]
]
```

If the attribute is not specified, `cumulate` uses the first attribute of the array. If the dimension is not specified, `cumulate` uses the first declared dimension of the array. The `cumulate` operator aggregates along a dimension. This differs from the sum of a **`GROUP BY`** aggregate, which groups all of the values in a dimension and then sums them:

```
SELECT sum(val)
     FROM m4x4
     GROUP BY j;
```

```
[(24),(28),(32),(36)]
```

```
SELECT sum(val)
     FROM m4x4
     GROUP BY i;
```

```
[(6),(22),(38),(54)]
```

8.3. 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.

8.3.1. 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 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. Here are the contents of the array:

```
[
[(13,'sand'),(),(),()],
[(8,'peanut'),(),(6,'pastry'),(7,'doc')],
[(4,'four'),(),(7,'carl'),()],
[(),(0,'livia'),(14,'Apostrophe'),()]
]
```

To sort the array elements by `val`, use the following query:

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

```
{n} val,label
{0} 0,'livia'
{1} 4,'four'
{2} 6,'pastry'
{3} 7,'doc'
{4} 7,'carl'
{5} 8,'peanut'
{6} 13,'sand'
{7} 14,'Apostrophe'
```

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

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

```
{n} val,label
{0} 0,'livia'
{1} 4,'four'
{2} 6,'pastry'
{3} 7,'carl'
{4} 7,'doc'
{5} 8,'peanut'
{6} 13,'sand'
{7} 14,'Apostrophe'
```

To use a descending sort, use the following command:

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

```
{n} val,label
{0} 14,'Apostrophe'
{1} 13,'sand'
{2} 8,'peanut'
{3} 7,'doc'
{4} 7,'carl'
{5} 6,'pastry'
{6} 4,'four'
{7} 0,'livia'
```

8.3.2. 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)]
]
```

8.3.3. 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 9. 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 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.

9.1. 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. For the complete list of compatibility rules for insertion operations, see the insert operator AFL reference topic.

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')]
]
```

9.2. 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)]
]
```

9.3. 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. Use the `version()` operator to see a list of the existing versions for an array:

```
AFL% versions(m4x4);
```

```
{VersionNo} version_id,timestamp
{1} 1,'2014-03-14 17:23:40'
{2} 2,'2014-03-14 17:23:40'
{3} 3,'2014-03-14 17:23:40'
```

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.

9.4. Deleting Array Versions

You can remove array versions, and keep only one version of an array. This can be useful if need to recover disk space.

Let's say you have 5 versions of array **stocks**:

```
AFL% project(versions(stocks), version_id);
```

```
{VersionNo} version_id
{1} 1
{2} 2
{3} 3
{4} 4
{5} 5
```

If you want to keep only the latest version of the array, you can perform the following steps to remove all other versions:

```
AFL% store(stocks,stocks_tmp);
```

```
AFL% remove(stocks);
```

```
AFL% rename(stocks_tmp,stocks);
```

To verify, use the `versions()` operator again:

```
AFL% project(versions(stocks), version_id);
```

```
{VersionNo} version_id
{1} 1
```

To keep one of the intermediate versions of **stocks**, rather than the latest, use the version syntax when specifying the array in the `store()` query. For example, to save the second version of **stocks**, you would run the following query:

```
AFL% store(stocks@2,stocks_tmp);
```

Chapter 10. Changing Array Schemas

This chapter describes several ways to transform SciDB 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.

10.1. 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 one-dimensional SciDB array, **device_probe**:

```
{i} device_id,probe_id,val
{1} 0,0,0.01
{2} 1,0,2.04
{3} 2,0,6.09
{4} 3,0,12.16
{5} 4,0,20.25
{6} 0,1,30.36
{7} 1,1,42.49
{8} 2,1,56.64
{9} 3,1,72.81
{10} 4,1,91
{11} 0,2,111.21
{12} 1,2,133.44
{13} 2,2,157.69
{14} 3,2,183.96
{15} 4,2,212.25
{16} 0,3,242.56
{17} 1,3,274.89
{18} 2,3,309.24
{19} 3,3,345.61
{20} 4,3,384
{21} 0,4,424.41
{22} 1,4,466.84
{23} 2,4,511.29
{24} 3,4,557.76
{25} 4,4,606.25
```

This array has three attributes, two of which are integers and one which is a floating-point number. The dimension name is i, the dimension size is 25, and the chunk size is 5.

When you examine the data, notice that it could be expressed in a 2-dimensional format like this:

	Probe ID= 0	**Probe ID = 1**	**Probe ID = 2**	**Probe ID = 3**	**Probe ID = 4**
Device ID = 0	0.01	30.36	111.21	242.56	424.41
Device ID = 1	2.04	42.49	133.44	274.89	466.84
Device ID = 2	6.09	56.64	157.69	309.24	511.29
Device ID = 3	12.16	72.81	183.96	345.61	557.76
Device ID = 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>[device_id=0:4,5,0, probe_id=0:4,5,0];
```

Each of the dimensions is of size 5, corresponding to a dimension in the 5-by-5 table. Now, you can use the `redimension()` operator to redimension the array device_probe into the array two_dim:

```
AFL% store(redimension(device_probe, two_dim),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 device_id=2 AND probe_id=3;
```

```
[
[(),(),(),(),()],
[(),(),(),(),()],
[(),(),(),(309.24),()],
[(),(),(),(),()],
[(),(),(),(),()]
]
```

10.1.1. Cell Collisions

The redimension operator 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:

- **Include a synthetic dimension.** A synthetic dimension is a vector that consists of the counts for all the candidate cell values for each cell location.
- **Perform aggregation.**
- **Select randomly.** Randomly choose a cell from among the candidate cells.

For details on these techniques, see the redimension operator reference section.

10.1.2. Redimensioning Arrays Containing Null Values

Nullable attributes are handled in a special manner by redimension. If the source array contains null values for the attribute being transformed, these cells will be dropped during the redimension. For example, consider the 1-dimensional array **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=0:2,2,0, val2=0:5,5,0];
```

```
AFL% store(redimension(redim_missing,redim_target),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.

10.2. 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 )
```

10.2.1. 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` operator.

- **Unpacking** a multidimensional array into a 1-dimensional array is performed with the `unpack` operator.
- **Reversing** the cells in an array is performed with the `reverse` operator.
- **Sorting** the cells in an array or within subarrays corresponding to each dimension, is performed with the `sort` operator.

For example, you might want to reshape your array from an *m* -by- *n* array to a 2 *m* -by- *n* /2 array. The `reshape` operator 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` operator 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` operator:

```
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)]
]
```

10.2.2. 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` operator. 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` query:

```
AFL% show(m4x4)
```

```
m4x4

< val:double >

[i=0:3,4,0,
j=0:3,4,0]
```

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

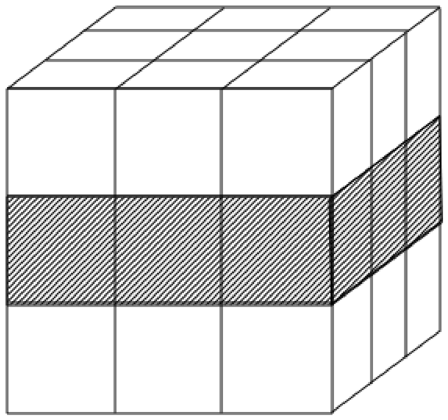
```
[
[(0),(1),(2),(3)],
[(4),(5),(6),(7)],
[(8),(9),(10),(11)],
[(12),(13),(14),(15)]
]
```

```
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 10.1. 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` operator 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` operator 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)]
]
```

10.3. 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` operator:

```
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` operator:

```
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)]
]
```

10.4. Changing Array Dimensions

10.4.1. 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]);
```

10.4.2. 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)
```

```
{device_id,probe_id} val
{0,0} 0.01
{0,1} 30.36
{0,2} 111.21
{0,3} 242.56
{0,4} 424.41
{1,0} 2.04
```

```
{1,1} 42.49
{1,2} 133.44
{1,3} 274.89
{1,4} 466.84
{2,0} 6.09
{2,1} 56.64
{2,2} 157.69
{2,3} 309.24
{2,4} 511.29
{3,0} 12.16
{3,1} 72.81
{3,2} 183.96
{3,3} 345.61
{3,4} 557.76
{4,0} 20.25
{4,1} 91
{4,2} 212.25
{4,3} 384
{4,4} 606.25
```

```
AFL% show(Dsp)
```

```
Dsp

< val:double >

[device_id=0:4,5,0,
probe_id=0:4,5,0]
```

Suppose you want to examine a sample plane from each dimension of the array. You can use the slice operator to select array slices from array `Dsp` :

```
AQL% SELECT * INTO Dsp_slice_0 FROM slice(Dsp, device_id, 0);
```

```
{probe_id} val
{0} 0.01
{1} 30.36
{2} 111.21
{3} 242.56
{4} 424.41
```

```
AQL% SELECT * INTO Dsp_slice_1 FROM slice(Dsp, device_id, 1);
```

```
{probe_id} val
{0} 2.04
{1} 42.49
{2} 133.44
{3} 274.89
{4} 466.84
```

```
AQL% SELECT * INTO Dsp_slice_2 FROM slice(Dsp, device_id, 2);
```

```
{probe_id} val
{0} 6.09
{1} 56.64
{2} 157.69
{3} 309.24
{4} 511.29
```

The slices are 1-dimensional.

```
AFL% show(Dsp_slice_2)
```

```
Dsp_slice_2

< val:double >

[probe_id=0:4,5,0]
```

Concatenating these slices will create a 1-d array:

```
AQL% SELECT * INTO Dsp_1d FROM concat(Dsp_slice_0, Dsp_slice_2);
```

```
{probe_id} 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 >

[probe_id=0:9,5,0]
```

To concatenate these arrays into a 2-dimensional array, you need to add a dimension to both. The `adddim` operator 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,probe_id} val
{0,0} 0.01
{0,1} 30.36
{0,2} 111.21
{0,3} 242.56
{0,4} 424.41
{1,0} 6.09
{1,1} 56.64
{1,2} 157.69
{1,3} 309.24
{1,4} 511.29
```

```
AFL% show(Dsp_new)
```

```
Dsp_new

< val:double >

[d=0:1,1,0,
probe_id=0:4,5,0]
```

Chapter 11. JDBC Driver for SciDB

This chapter describes how to connect to SciDB by using the SciDB driver for JDBC.

After you install the SciDB driver for JDBC, you can connect from your program to your database with a connection URL. This way of connecting to a database is by through JDBC Driver Manager by using the getConnection method of the DriverManager class. The simplest manner of using this method takes a string parameter that contains a URL.

The chapter contains the following sections:

- Installation
- Set the CLASSPATH variable
- Register the driver
- Connection URL
- Sample code

11.1. Installation

Note the following dependencies for the JDBC driver:

- Google Protocol Buffers. For example:
  - On Ubuntu, the package is named `libprotobuf-java`.
  - On CentOS and RHEL, the package is named `protobuf-java.noarch`. Note that the SciDB repository redistributes this package. Thus, if you want to install the package on a CentOS or RHEL client machine, you should create the `etc/yum.repos.d/scidb.repo` file on your client. For details, see the “Install SciDB on CentOS / RHEL from binary package” section in the Installation Chapter.
- java1.6 or higher

Create or choose a directory for the JDBC driver, represented here as $JDBC. Run the following commands to download the SciDB JDBC driver:

```
cd $JDBC
wget http://downloads.paradigm4.com/client/14.3/jdbc/scidb4j.jar
```

11.2. Set the CLASSPATH Variable

The SciDB driver for JDBC jar file and protobuf library dependency must be listed in your CLASSPATH variable. The CLASSPATH variable is the search string that the Java Virtual Machine (JVM) uses to locate the JDBC drivers on your computer.

Set your system CLASSPATH variable to include the following entries:

- On Ubuntu:

  ```
  $JDBC/scidb4j.jar:/usr/share/java/protobuf-java.jar
  ```

- On CentOS/RHEL:

  ```
  $JDBC/scidb4j.jar:/usr/share/java/protobuf.jar
  ```

If the drivers are not listed in your CLASSPATH variable, you receive the following error message when you try to load the driver:

```
java.lang.ClassNotFoundException: org.scidb.jdbc.Driver
```

11.3. Register the Driver

Registering the driver instructs the JDBC Driver Manager which driver to load. When you load a driver by using the **class.forName** function, you must specify the name of the driver. The name for SciDB Driver for JDBC is **org.scidb.jdbc.Driver**.

The following code snippet illustrates how to register the driver:

```
Class.forName("org.scidb.jdbc.Driver");
```

11.4. Connection URL

You must pass your database connection information in the form of a connection URL.

The URL is of the form **jdbc:scidb://*hostname*[:*port*]**. Substitute the host name and port for your SciDB database.

The host value can be an IP address or a host name (assuming that your network resolves host names to IP addresses). You can test this by pinging the host name and verifying that you receive a reply with the correct IP address. The numeric value after the host is the port number on which the database is listening. Make sure to substitute the port number that your database is using. If you do not specify a port, the default value is used.

The following code snippet demonstrates how to specify a connection URL:

```
Connection conn = DriverManager.getConnection("jdbc:scidb://localhost/");
```

11.5. Sample Code

If you downloaded the SciDB source package, the following code is available in the `JDBCExample.java` file.

```
package org.scidb;

import org.scidb.jdbc.IResultSetWrapper;

import java.io.IOException;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

class JDBCExample
{
  public static void main(String [] args) throws IOException
  {
    try
    {
```

```
      Class.forName("org.scidb.jdbc.Driver");
    }
    catch (ClassNotFoundException e)
    {
      System.out.println("Driver is not in the CLASSPATH -> " + e);
    }

    try
    {
      Connection conn = DriverManager.getConnection("jdbc:scidb://localhost/");
      Statement st = conn.createStatement();
      // create array A<a:string>[x=0:2,3,0, y=0:2,3,0];
      // select * into A from
      // array(A, '[["a","b","c"]["d","e","f"]["123","456","789"]]');
      ResultSet res = st.executeQuery("select * from " +
         " array(<a:string>[x=0:2,3,0, y=0:2,3,0]," +
         " '[[\"a\",\"b\",\"c\"][\"d\",\"e\",\"f\"][\"123\",\"456\",\"789\"]]')");
      ResultSetMetaData meta = res.getMetaData();

      System.out.println("Source array name: " + meta.getTableName(0));
      System.out.println(meta.getColumnCount() + " columns:");

      IResultSetWrapper resWrapper =
         res.unwrap(IResultSetWrapper.class);
      for (int i = 1; i <= meta.getColumnCount(); i++)
      {
        System.out.println(meta.getColumnName(i) + " - " + meta.getColumnTypeName(i)
           + " - is attribute:" + resWrapper.isColumnAttribute(i));
      }
      System.out.println("=====");

      System.out.println("x y a");
      System.out.println("-----");
      while(!res.isAfterLast())
      {
        System.out.println(res.getLong("x") + " " + res.getLong("y") + " "
           + res.getString("a"));
        res.next();
      }
    }
    catch (SQLException e)
    {
      System.out.println(e);
    }
  System.exit(0);
  }
}
```

To run this example, make sure SciDB is running, then run the following commands:

```
cd $JDBC
wget http://downloads.paradigm4.com/client/14.3/jdbc/example.jar
java -classpath example.jar:$CLASSPATH org.scidb.JDBCExample
```

If the example runs without error, the output is similar to the following:

```
Source array name: build
3 columns:
x - int64 - is attribute:false
y - int64 - is attribute:false
a - string - is attribute:true
=====
x y a
-----
0 0 a
0 1 b
0 2 c
```

```
1 0 d
1 1 e
1 2 f
2 0 123
2 1 456
2 2 789
```

Chapter 12. Macros

In this version, SciDB introduces some experimental functionality. We may expand this functionality in future versions. Conceptually, this functionality resembles the macro facility offered by the C preprocessor, `cpp`.

This feature allows a user to create what amount to subroutines: a mapping from a name and parameter list to an expression body. The query processor will recognize an application of the macro within a query and expand the macro inline.

This functionality consists of the following two pieces:

- *macro*: an entity that gets expanded before query execution. Wherever the name of the macro is found in the query, it is replaced by the body of the macro.
- *module*: a collection of macros. A module is a text file, and only one module can be loaded into SciDB at a time.

A listing of several useful macros can be found in Appendix E, *Sample Macros*.

12.1. Syntax

The current syntax is as follows:

```
macro-definition  ::= macro-name ( macro-parameters ) = macro-body [where \{ macro-
definitions \}]
```

where:

- `macro-name` is an identifier.
- `macro-parameters` is a (possibly empty) comma-separated list of distinct identifiers.
- `macro-body` is an AFL query that may refer by name to both the macro's parameters and also any locally bound helper macros defined in the optional `where` clause. Such nested macros are only visible within the enclosing macro's body.
- `macro-definitions` is a (possibly empty) semicolon-separated list of macro definitions.

12.2. Usage

You provide macro definitions in a text file (called a *module*) that is accessible to the coordinator instance. The module may include both Postgres style line (`--`) and block (`/* ... */`) comments.

You then load the contents of the text file into SciDB using the operator `load_module()`. It takes one argument: the pathname of a text file containing the macros. SciDB parses its contents, loads them into memory, and makes them available for subsequent queries.

For example, if you have all of your macros in **/home/user1/scidb/expansion.txt**, you would run the following query to load the contents into SciDB:

```
$ iquery -aq "load_module('/home/user1/scidb/expansion.txt')"
```

Note the following:

- The contents stay in memory until you restart SciDB, whereupon you will need to reload them.
- Once loaded, the macros become globally available to all subsequent queries evaluated on the SciDB cluster.
- Any subsequent call of `load_module` invalidates previously loaded macros, even if the particular macro is not defined in the newly loaded file.
- A macro named *m* hides any existing identically named applicable entity in the system, including operators, functions, their user defined counterparts (UDO's and UDF's), and other macros.
- Macros are looked up with a case sensitive search—the name must match exactly for the macro to be expanded.

Caveats (attempting any of these will generate errors):

- Do not define a recursive macro.
- Do not define two macros with the same name in the same scope.
- Do not apply a macro to more or fewer arguments than the macro has parameters.

12.3. Examples

The following example provides a macro that computes the Euclidean distance between the points <x1,y1> and <x2,y2>.

```
distance(x1,y1,x2,y2) = sqrt(sq(x2 - x1) + sq(y2 - y1)) where
{
    sq(x) = pow(x,2)   -- the square of the scalar "x"
}
```

Notice here that `sq` is an example of a nested macro—its definition is not visible anywhere else.

Assuming this macro is specified in a module, and also that the module is loaded into SciDB, you could then run a query like the following:

```
$ iquery -aq "filter(project(apply(join(A,B), dist, distance(A.x, A.y, B.x,
 B.y)),dist),dist < 6)"
```

```
{i,j} dist
{0,0} 1
{0,1} 4.24264
{0,3} 3.16228
{1,0} 1.41421
{1,2} 3
{1,3} 4.12311
{2,0} 2.23607
{2,1} 3
```

The following example illustrates a macro that analyzes the chunks for your SciDB cluster. Once you have loaded the module, you can then access the query as follows:

```
$ iquery -aq "chunk_skew()"
```

```
{iid,aid} name,nchunks,min_ccnt,avg_ccnt,max_ccnt,total_cnt
{0,23} 'big2@1',3,100,100,100,300
{0,25} 'big3@1',25,1000,1000,1000,25000
{1,23} 'big2@1',3,100,100,100,300
{1,25} 'big3@1',25,1000,1000,1000,25000
{2,23} 'big2@1',2,100,100,100,200
```

```
{2,25} 'big3@1',25,1000,1000,1000,25000
{3,23} 'big2@1',2,100,100,100,200
{3,25} 'big3@1',25,1000,1000,1000,25000
```

The **map** macro is a shortcut for the oft-used **project(apply(...))** construction.

```
/** Apply 'expression' to each element of the 'array'. **/
map(array,expression,name) = project(apply(array,name,expression),name);
```

The center macro is useful for centering the columns of a matrix. After you load the module, run the following queries:

```
$ iquery "store(build(<val:double>[row=1:5,5,0,col=1:5,5,0],row),M)"
```

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

```
$ iquery "center(M,val,col)"
```

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

The source for these macros and others is included in Appendix E, *Sample Macros*

Chapter 13. User-Defined Types and Functions

Out of the box, SciDB provides users with a standard set of data types, such as integers, floating-point numbers, and strings. Scientific and large-scale analytic applications often require other data types such as complex numbers, rational numbers, two-dimensional points, or others. Some applications call for specific mathematical functions (such as greatest common factor of two integers or non-uniform random number generation) that SciDB does not provide by default. SciDB's extensibility mechanism allows users to add their own implementation of types and functions to the SciDB engine.

Suppose a SciDB application requires a rational number datatype. Rather than use double precision, the user wants to store an integer-type numerator and denominator pair. As part of the the new type's functionality, users will also want to support basic arithmetic (+, -, *, /) and logical (<, <=, =, >=, >, <>) functionality.

At the level of the AQL query language, the new type can be used as follows:

```
create array rational_example < N : rational > [ I=0:99,10,0, J=0:99,10,0 ]

# Q1:

SELECT COUNT(*)
   FROM rational_example AS R
   WHERE R.N = rational(1,2);

# Q2:

SELECT str(R.N)
  FROM rational_example AS R
  WHERE R.N + rational(1,4) > rational(1,2);
```

So far as a user's queries are concerned, there will be no difference between the way a built-in type and a user-defined type (or function) works. There are, however, a few items to consider:

- All type conversions need to be explicit. SciDB does not support implicit casting.
- Client applications can accommodate only a limited set of types: doubles, integers, and strings. When you write queries (using iquery, for example), the query's result needs to explicitly convert result types into something that the client understands.
- While it is possible to write complex and computationally-expensive UDFs, we recommend that you build UDFs as small, self-contained units of functionality. You can then use SciDB's operator composition functionality to increase the power and complexity of your operations.
- SciDB does not support features such as embedding queries within UDFs, nor plugins that do anything more sophisticated than take a vector of scalar values and return a single scalar type result.

To create and use UDTs and UDFs, read the following topics:

- Process to Implement a Plugin Library
- User-Defined Type Examples
- User-Defined Function Examples
- Plugin Architecture

- How to Load a Plugin
- Steps to Create a SciDB Plugin
- Register Your Functions

13.1. Process to Implement a Plugin Library

The easiest way to implement your own plugin library is to copy the style of the provided examples.

1. Create a new directory parallel to one of the example directories, for example **point1**.
2. In the **~/examples/CMakeLists.txt** file, add a new line with the name of the new directory. For example, if your new directory is "point1," you would add the following line:

   ```
   add_subdirectory("point1")
   ```
3. Rename the C file that you copied to an appropriate name for your library, for example **point1**.
4. Change the contents of the **~/examples/point1/CMakeLists.txt** to get the server to build your new plugin library.
5. Make you modifications to the new source code file, for example **~/examples/point1/point1.cpp**.
6. Using **make**, build **libpoint1.so**. The system places the file into the plugins directory, along with the other example plugin files.

You can now load the library into SciDB, using the `load_library` operator.

13.2. User-Defined Type Examples

The SciDB distribution includes several example extensions in the `~/examples` folder located beneath the SciDB root directory. The following user-defined type (UDT) examples are available:

Example Name	**Description**
complex	Complex number UDT (a + b.i), together with the associated algebraic operations, and equality.
rational	Rational number UDT (int64 numerator and denominator) together with the associated algebra operations and ordering comparisons.
point	Two-dimensional (2-D) point UDT. Double precision X and Y.

There is also a selection of user-defined functions (UDFs) available in the `~/examples/more_math` folder.

13.3. User-Defined Function Examples

There is a selection of user-defined functions (UDFs) available in the `~/examples/more_math` folder.

Example Name	**Description**
isprime	Function to determine whether the input integer is prime.
fact	Function to find the factors of the input integer.

Example Name	Description
lasso	Lasso (Least Absolute Shrinkage and Selection Operator) function, a regularized version of of the least squares method of solving a series of linear equations.
mylog	Function to calculate the logarithm, where you can specify the base. SciDB provides base e and base 10 logarithmic functions; this UDF can be used for other bases.

13.4. Plugin Architecture

The algorithms implemented within the SciDB engine are designed to treat instances of data type values as black box memory segments. For example, all that the SciDB engine knows about the contents of the Complex Number data type is that it is 16 bytes long. The code that needs to know about the contents of these 16 bytes is implemented by the user in their own C/C++ code, which they compile into a shared library.

```
//
// This is the struct that describes how to organize the 16 bytes of data that
// makes up an instance of a SciDB Complex UDT.
struct Complex
{
    double re;
    double im;
};

//
// This is the code that takes data from the SciDB engine, performs the addition, and
 deposits
// the return result in an appropriately sized "black box" of bytes. The SciDB engine
 takes
// this return result and stores it, or passes it on to another function.
static void addComplex(const Value** args, Value* res, void*)
{
    Complex& a = *(Complex*)args[0]->data();
    Complex& b = *(Complex*)args[1]->data();
    Complex& c = *(Complex*)res->data();

    c.re = a.re + b.re;
    c.im = a.im + b.im;
}
```

When SciDB parses a query like the one labeled "Q2" above, the SciDB engine checks to ensure that it has been provided with a shared library containing code to perform the addition (rational type, rational type -> rational type) operation.

In the case of the Complex type shown here, SciDB looks for a function named "+" that takes two arguments of the appropriate type (in this case, a pair of Complex number instances). Then, at run time, the SciDB engine assembles the necessary 16-byte "black boxes", invokes the function addComplex(), and deals with the value it computes.

As you can see from the example code above, SciDB uses a `typesystem::Value` class to encapsulate information about all type value instances. The Values class provides a set of methods for getting and setting the "value" of the class for each of the SciDB built-in types; `getType()` and `setType()`. Or, in the case of a typesystem::Value val; instance that contains a string, `val.getString()` and `val.setString()`.

From the perspective of the SciDB engine, all UDFs have the same basic signature:

```
void functionUDF(const Value** args, Value* res, void*)
```

```
{

}
```

Each function must define how many arguments it is to receive. These arguments are obtained from the vector of `typesystem::Value` pointers that makes up the first argument. Each UDF returns a single result. The location where this result is to be placed is passed in by reference in the second argument. The final argument is a pointer to a data structure that conveys information about the state of the engine, and is a means of passing data between repeated calls to the UDF within the same query.

13.5. How to Load a Plugin

All of the example plugins that are included with the SciDB distribution are built at the same time the SciDB core engine is built. However, SciDB does not load unregistered plugins when it starts. To use an example, you need to load it into a SciDB instance. Use the following statements to load the rational example shared library into SciDB.

In AFL, run the following query:

```
AFL% load_library('rational');
```

In AQL, run the following query:

```
AQL% load library 'rational';
```

These queries load a plugin named **librational.so**. The default plugin folder is `/opt/scidb/14.3/lib/scidb/plugins`.

The act of loading a 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.

To unload a shared library, run either of the following:

```
AFL% unload_library('rational');
```

or

```
AQL% unload library 'rational';
```

The unload query unregisters the library in the system catalog. Note, however, that unloading a library does not take effect until the next time you restart SciDB. This is done to protect any running queries that may be using the library at the time the library is unloaded.

13.6. Steps to Create a SciDB Plugin

This section explains the steps needed for creating a new plugin for SciDB.

After you create a new data type, you need to complete the following steps before you can use that type in SciDB.

1. **Constructors.** You need to create User-Defined Functions (UDFs) that construct instances of your new type based on the values of other types. In general, the types you will use as input to these UDFs will be built-in types. For example, it is typical to use a string as a source for a new data type's contents. The following UDF converts a string with a particular format into an instance of a rational number data type:

   ```
   //
   // This is the struct used to store the data inside SciDB.
   ```

```
typedef struct
{
    int64_t num;
    int64_t denom;
} SciDB_Rational;

void str2Rational(const Value** args, Value* res, void*)
{
 int64_t n, d;
    SciDB_Rational* r = (SciDB_Rational*)res->data();

    if (sscanf(args[0]->getString(), "(%"PRIi64"/%"PRIi64")", &n, &d) != 2)
        throw PLUGIN_USER_EXCEPTION("librational",
         SCIDB_SE_UDO, RATIONAL_E_CANT_CONVERT_TO_RATIONAL)
            << args[0]->getString();

 if ((0 == d) && (0 == n))
  d = 1;

 boost::rational<int64_t>rp0(n, d);
 r->num   = rp0.numerator();
 r->denom = rp0.denominator();

}
```

Note that the string to UDT conversion function, `str2Rational`, is particularly important. This conversion function is the UDF used by the `load()` operator to bulk ingest data into SciDB.

2. **Converters to built-in types.** You need UDFs that convert your UDT back into a built-in type, or a number of built-in types. In the case of the complex type, for example, you can either write a UDF that composes the 16 bytes into a string, or else a pair of UDFs that extract the real and imaginary portions of the type.

```
static void reComplex(const Value** args, Value* res, void*)
{
   Complex& a = *(Complex*)args[0]->data();
   res->setDouble(a.re);
}

static void imComplex(const Value** args, Value* res, void*)
{
   Complex& a = *(Complex*)args[0]->data();
   res->setDouble(a.im);
}
```

Remember that SciDB does not perform implicit casting. You need to include these UDFs in any queries that pull these values out of an array.

3. **Common functions.** For some data types, you may need functions to perform arithmetic and comparisons. For others, you may not need all of these functions. For example, it makes sense to support the full set of relational operators for a data type that can be ordered (such as rational numbers). Here are the UDFs for determining order for the rational UDT.

```
void rationalLT(const Value** args, Value* res, void * v)
{
    SciDB_Rational* r0 = (SciDB_Rational*)args[0]->data();
    SciDB_Rational* r1 = (SciDB_Rational*)args[1]->data();

 check_zero ( r0 );
 check_zero ( r1 );

 boost::rational<int64_t>rp0(r0->num, r0->denom);
 boost::rational<int64_t>rp1(r1->num, r1->denom);

 if ( rp0 < rp1 )
```

```
        res->setBool(true);
 else
        res->setBool(false);
}

void rationalEQ(const Value** args, Value* res, void * v)
{
    SciDB_Rational* r0 = (SciDB_Rational*)args[0]->data();
    SciDB_Rational* r1 = (SciDB_Rational*)args[1]->data();

 check_zero ( r0 );
 check_zero ( r1 );

 boost::rational<int64_t>rp0(r0->num, r0->denom);
 boost::rational<int64_t>rp1(r1->num, r1->denom);

 if ( rp0 == rp1 )
        res->setBool(true);
 else
        res->setBool(false);
}
```

For unordered types, such as a data type to represent complex numbers, you need less functionality. For example, for our complex number data type, two values are equal or not equal. SciDB needs only one UDF, to return TRUE when two type values are equal, and FALSE the two values are not equal.

4. **Integrators.** UDFs that are necessary to support the integration of the UDT with other facilities, such as aggregates. For example, `max()` and `min()` use the UDFs that order instance values. If your type has specific requirements `max()` and `min()`, you might need to add these UDFs.

5. **Error Handling.** Your UDFs need to check for errors and exceptions in their code. SciDB provides facilities to report to the SciDB engine that your UDF has encountered an error, and the type of error. Doing this allows the SciDB engine to terminate the query and report some useful status information to the log file.

   Errors and exceptions are thrown using a macro, **USER_EXCEPTION (error_code, description : string)**. See the listing for the function `str2Rational`, in step 1 above. For a full list of the terse error codes that you can throw from within a UDF, consult the **~/include/system/ErrorCodes.h** file.

13.7. Register Your Functions

Once you have implemented your functions, you should register them with the SciDB facilities for extracting information from a shared library. The SciDB install provides a set of C macros to do this.

Macro Name	**Description**	**Example**
REGISTER_TYPE (name, length)	Instructs SciDB to register a new UDT in it's catalogs with the name provided (note that this argument to the macro is not a string) and the length, in bytes, of the type instance values.	REGISTER_TYPE (complex, 16)
REGISTER_FUNCTION (name, input argument types, output argument type, function pointer)	Instructs SciDB to register a new UDF in its catalogs. The new UDF can be called in AQL or AFL using the first argument name (again, not a string), the the function is expected to take a list of argument types as input, and return a value of the type provided. The	REGISTER_FUNCTION (+, ("complex", "complex"), "complex", addComplex);

Macro Name	**Description**	**Example**
	actual reference to the function you want SciDB to call is the last argument to the macro.	
REGISTER_CONVERTER (input type, output type, conversion cost, function pointer)	From time to time SciDB needs to convert types, and it can require UDFs to perform this operation. This macro is how you register conversions.	REGISTER_CONVERTER (string, complex, EXPLICIT_CONVERSION-_COST, string2complex);

Chapter 14. SciDB Linear Algebra Library

The SciDB linear algebra operators accept SciDB matrices as inputs. Linear algebra operators interpret the first declared dimension as rows, and the second dimension as columns.

The Linear Algebra Library provides functionality for matrix operations including:

- Matrix inverse
- Matrix multiply
- Matrix transpose
- Singular Value Decomposition

14.1. Matrix Inverse

The matrix inverse for a square matrix A is a matrix A^{-1} such that $AA^{-1} = I$, where I is the identity matrix. In SciDB, you can calculate the matrix inverse by using the `gemm()` and `gesvd()` operators.

For example, assume we have the following SciDB array, A:

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

Note that A is a real-valued, square, invertible matrix, which is useful in regard to the following discussion.

The SVD of A is:

```
A*V = U*S
```

where U and V are orthonormal matrices, S is a diagonal matrix, and * represents matrix multiplication. And, since A is invertible, we know from the definition of the SVD that the diagonal entries of S are all positive numbers.

Let S^{-1} represent the inverse of S. S^{-1} is the diagonal matrix whose entries are the reciprocals of the diagonal entries of S. We then multiply each side of the above equation by S^{-1}.

```
A * V * S^-1 = U * S * S^-1
             = U * I (the identity matrix)
             = U
```

Let U^T be the transpose of U. Since A is square, U is square, and because it is orthonormal, $U^T * U = I = U * U^T$. Now we multiply each side of the above equation by U^T.

```
A * V * S^-1 * U^T = U * U^T
                   = I
```

Thus, $A * V * S^{-1} * U^T = I$, and therefore the inverse of A is given by $V * S^{-1} * U^T$. The following query computes this:

```
AFL% transpose(
       gemm(
         project(apply(
            cross_join(
               gesvd(A,'left') as X,
               gesvd(A, 'values') as Y, X.i_2, Y.i),
         val, u / sigma),val),
         gesvd(A,'right'),build(A,0))
      );
```

```
[
[(-6),(12),(6),(7)],
[(9),(-17),(-7),(-9)],
[(-5),(10),(4),(5)],
[(1),(-2),(-1),(-1)]
]
```

14.2. Matrix Multiply

We want to compute the standard linear algebra matrix product. For two matrices *A* and *B* where *A* is *m*-by-*n* and *B* is *n*-by-*p*, the product *AB* is an *m*-by-*p* matrix given by:

$$AB(i,k) = \sum_{j=1}^{n} A(i,j)B(j,k)$$

In SciDB, we use the operator `gemm()` to multiply matrices. Note that `gemm()` takes three arguments; for the third argument, we construct a matrix with all of its values as zeros.

Assume we have the following matrices:

```
AFL% show(m2x3)
```

```
m2x3

< val:double >

[row=0:1,32,0,
col=0:2,32,0]
```

```
AFL% scan(m2x3);
```

```
[
[(0.5),(1.5),(2.5)],
[(1),(2),(3)]
]
```

```
AFL% show(m3x2)
```

```
m3x2

< val:double >

[row=0:2,32,0,
col=0:1,32,0]
```

```
AFL% scan(m3x2);
```

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

The matrix multiplication will produce a 2-by-2 matrix, so we need a 2x2 matrix that has all zeros:

```
AFL% store(build(<val:double>[row=0:1,32,0, col=0:1,32,0],0),Zeros);
```

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

The following query performs the matrix multiplication:

```
AFL% gemm(m2x3,m3x2,Zeros);
```

```
[
[(6.5),(15.5)],
[(8),(20)]
]
```

14.3. Matrix Transpose

The transpose of matrix *A*, denoted A^T, is the matrix $A^T(j,i) = A(i,j)$ for all *j* and *i* where $1 \le i \le n$ and $1 \le j \le m$. The `transpose` operator performs matrix transpose:

```
AFL% scan(m3x3);
```

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

```
AFL% transpose(m3x3);
```

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

14.4. Singular Value Decomposition

For a singular matrix *M*, that is, a matrix not invertible either because it is rectangular or because it has a determinant of 0, singular-value decomposition returns the matrix factorization:

$$M = USV^T$$

For $m \times n$ matrix *M*, the left matrix *U* is $m \times m$ and the right matrix V^* is $n \times n$. *U* is orthogonal in the row space ($U^T U = I(m \times m)$) and V^* is orthogonal in the column space ($V^{*T} V = I(n \times n)$).

The matrix Σ is a diagonal matrix containing the singular values of *M*.

For example, consider the 3×2 matrix `m3x2`:

```
AFL% show(m3x2)
```

```
m3x2

< val:double >

[row=0:2,32,0,
col=0:3,32,0]
```

```
AFL% scan(m3x2);
```

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

You can return the left matrix, the right matrix, or a vector containing the singular values:

```
AQL% SELECT * FROM gesvd(m3x2, 'right');
```

```
[
[(0.325251),(0.8759),(0.112699),(0.338098)],
[(0.391754),(0.218207),(-0.282651),(-0.847952)],
[(-0.272166),(0.136083),(0.872783),(-0.38165)]
]
```

```
AQL% SELECT * FROM gesvd(m3x2, 'left');
```

```
[
[(0.569595),(-0.821926),(0)],
[(0.821926),(0.569595),(0)],
[(0),(0),(1)]
]
```

```
AQL% SELECT * FROM gesvd(m3x2, 'values');
```

```
[(5.05411),(2.90792),(0)]
```

Chapter 15. SciDB Aggregate Reference

This chapter lists SciDB aggregates. Aggregates take as input a set of 1 or more values and return a SciDB array.

You can get a list of all of the available aggregates by running the following query:

```
$ iquery -aq "list('aggregates')"
```

15.1. Aggregate Syntax

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]);
```

15.2. Result Naming Convention

Aggregate calls return a value or values as attributes. Each aggregate call has an *`aggregateFunction`*, an *`inputAttribute`*, and a *`resultAttribute`*. The name for the result is *`inputAttribute_aggregateFunction`*. For example, assume an array **A** with an attribute **sales**. The name for the result attribute of **aggregate(A,sum(sales));** is 'sales_sum'.

```
AFL% aggregate(A,sum(sales));
```

```
i,sales_sum
0,14
```

The count aggregate may take * as the input attribute, meaning to count all the items in the group including null items. The result name for count(*) is 'count'.

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

```
i,count
0,10
```

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 [GROUP BY dimension_1,dimension_2,...]
```

```
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. You can optionally specify one or more dimensions to group by.

Getting a distinct count of an attribute is computationally expensive. Additionally, if you have a live and changing data set (there are concurrent updates / inserts / deletes going on) then you have an imprecise answer as soon as you have computed it.

Thus, SciDB offers a an imprecise but much faster "approximate" method. The motivation being that in many cases, an approximate count is just as useful as an exact count, especially when the numbers involved are very large. Often, it is better to get a number that's accurate to within 1-2% that takes 1/10th the time to compute.

- The `approxdc` aggregate does not count null values.
- The `approxdc` aggregate is a hybrid. If the number of distinct values is less than about 3,000, `approxdc` returns a precise count. If the number of distinct values is more than 3,000, the result is accurate to within about 1-2%.
- The `analyze` operator also returns the approximate count—in addition to some other useful information

Example

This example finds the approximate number of distinct integers in an array.

1. Create an array.

   ```
   AFL% create array counting <val:int32>
         [i=0:999,100,0, j=0:999,100,0];
   ```

2. Store values from 0 to 999,999—one million distinct values.

   ```
   AFL% store(build(counting, (i * 1000) + j),counting);
   ```

3. Retrieve an approximate count of the number of distinct values in the array.

   ```
   AFL% aggregate(counting,approxdc(val));
   ```

   ```
   {i} val_ApproxDC
   {0} 997498
   ```

   As you can see, the result, 997498, is within a few percent of the actual number of distinct values in the array.

Now, let's store 2000 different values, and use approxdc to retrieve the actual number of distinct values.

1. Take the mod 2000 to cut down the number of distinct values.

   ```
   AFL% store(build(counting, ((i * 1000) + j) % 2000),counting);
   ```

2. Retrieve an approximate count of the number of distinct values in the array.

   ```
   AFL% aggregate(counting,approxdc(val));
   ```

   ```
   {i} val_ApproxDC
   {0} 2000
   ```

   In this case, notice that the result is exact.

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 [GROUP BY dimension_1,dimension_2,...]
```

```
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 array.

1. Create an array:

   ```
   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);
   ```

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

3. Find the average of every column of m3x3:

   ```
   AFL% aggregate(m3x3,avg(val),j)
   ```

   ```
   [(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 [GROUP BY dimension_1,dimension_2,...]

AFL% aggregate(array,count(attribute)[,dimension_1,dimension_2,...])
```

OR

```
AQL% SELECT count(*) FROM array [GROUP BY dimension_1,dimension_2,...]
```

```
AFL% aggregate(array,count(*) [,dimension_1,dimension_2,...])
```

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 array.

1. Create an array with values 1 along the diagonal of m3x3, and the remaining cells empty.

```
AFL% store(redimension
        (apply
            (build(<v:double>[i1=1:3,3,0,j1=1:3,3,0],1),i,iif(i1=j1,i1,null),
         j,iif(i1=j1,j1,null)),
      <v:double>[i=1:3,3,0,j=1:3,3,0]), m3x3);
```

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

2. Find the number of nonempty cells in the array:

```
AFL% aggregate(m3x3,count(v));
```

```
[(3)]
```

This example finds the number of nonempty and non-null cells from an array that contains some NULL values.

1. Show the contents of array **A**:

```
[
[(),(),()],
[(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, along the first dimension:

```
AQL% SELECT count(value) FROM A GROUP BY row;
```

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

Name

max — Maximum value aggregate

Synopsis

```
AQL% SELECT max(attribute) FROM array [GROUP BY dimension_1,dimension_2,...]
```

```
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. You can optionally specify one or more dimensions to group by.

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 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% aggregate(m3x3,max(val),i);
   ```

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

Name

min — Minimum value aggregate

Synopsis

```
AQL% SELECT min(attribute) FROM array [GROUP BY dimension_1,dimension_2,...]
```

```
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. You can optionally specify one or more dimensions to group by.

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 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);
   ```

   ```
   [
   [(12),(2),(9)],
   [(13),(17),(2)],
   [(2),(15),(9)]
   ]
   ```

3. Select the minimum value of each row of m3x3:

   ```
   AFL% aggregate(m3x3,min(val),i);
   ```

   ```
   [(2),(2),(2)]
   ```

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} 2
   ```

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} 2,12
{0,0,1} 2,2
{0,0,2} 2,9
{0,1,0} 2,13
{0,1,1} 2,17
{0,1,2} 2,2
{0,2,0} 2,2
{0,2,1} 2,15
{0,2,2} 2,9
```

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} 2,12
{0,1} 2,2
{0,2} 2,9
{1,0} 2,13
{1,1} 2,17
{1,2} 2,2
{2,0} 2,2
{2,1} 2,15
{2,2} 2,9
```

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);
```

```
[
[(),(2,2),()],
[(),(),(2,2)],
[(2,2),(),()]
]
```

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);
```

```
[
[(),(2),()],
[(),(),(2)],
[(2),(),()]
]
```

Name

prod — Product aggregate

Synopsis

```
AQL% SELECT prod(attribute) FROM array [GROUP BY dimension_1,dimension_2,...]
```

```
AFL% aggregate(array,prod(attribute)[,dimension_1,dimension_2,...])
```

Summary

The prod aggregate calculates the cumulative product of a group of values.

The product of an empty set is 0. The product of a set that contains only NULL values is also 0. If the set contains NULL and NOT NULL values, the result is the product of all the NOT NULL values.

Example

This example finds the product of every column of a 3×3 array.

1. Create and array, m3x3, and store values of 1–9 into it:

```
AFL% store(build(<val:double>[row=0:2,3,0,col=0:2,3,0],row*3+col+1),m3x3);
```

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

2. Find the product of each column in m3x3:

```
AFL% aggregate(m3x3,prod(val),col)
```

```
[(28),(80),(162)]
```

3. Find the product of all the cells of m3x3:

```
AFL% aggregate(m3x3,prod(val))
```

```
[(362880)]
```

Name

stdev — Aggregate that calculates the sample standard deviation.

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 sample standard deviation of those values. You can optionally specify one or more dimensions to group by.

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 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);
   ```

   ```
   [
   [(0.3),(0.1),(0.7)],
   [(0.8),(0),(0.8)],
   [(0.2),(0.4),(0.4)]
   ]
   ```

3. Select the standard deviation of each row of m3x3:

   ```
   AFL% aggregate(m3x3,stdev(val),i);
   ```

   ```
   [(0.305505),(0.46188),(0.11547)]
   ```

Name

sum — Sum aggregate

Synopsis

```
AQL% SELECT sum(attribute) FROM array [GROUP BY dimension_1,dimension_2,...]
```

```
AFL% aggregate(array,sum(attribute)[,dimension_1,dimension_2,...])
```

Summary

The sum aggregate calculates the cumulative sum of a group of values. You can optionally specify one or more dimensions to group by.

The sum of an empty set is 0. The sum 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 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% aggregate(m3x3,sum(val),j);
   ```

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

4. Sum the values of m3x3 along dimension i. This sums the rows of m3x3:

   ```
   AFL% aggregate(m3x3,sum(val),i);
   ```

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

Name

var — Aggregate that calculates the sample variance.

Synopsis

```
AQL% SELECT var(attribute) FROM array;
```

```
AFL% aggregate(array,var(attribute)[,dimension_1,dimension_2,...])
```

Summary

The `var` aggregate returns the sample variance of a set of values. You can optionally specify one or more dimensions to group by.

The sample variance of an empty set is NULL. The sample 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 array.

1. Create an array:

   ```
   AQL% 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:

   ```
   AFL% store(build(m3x3,random()%10/1.0),m3x3);
   ```

   ```
   [
   [(1),(2),(2)],
   [(2),(9),(0)],
   [(2),(0),(0)]
   ]
   ```

3. Find the variance for each column of m3x3:

   ```
   AFL% aggregate(m3x3,var(val),j)
   ```

   ```
   [(0.333333),(22.3333),(1.33333)]
   ```

Chapter 16. 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);
```

SciDB contains the following categories of functions:

- Algebraic Functions
- Comparison Functions
- Temporal Functions
- Trigonometric and Transcendental Functions
- Miscellaneous functions

You can get a list of all of the available functions by running the following query:

```
$ iquery -aq "list('functions')"
```

Name

Algebraic functions — Perform basic arithmetic in a query expression.

Synopsis

```
SELECT expression1 FROM array WHERE expression2;
```

Summary

These functions 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
ceil	Round to next-highest value	ceil(scalar)	double
floor	Round to next-lowest integer	floor(scalar)	double
pow	Raise to a power	pow(base,exponent)	double
random	Generate random positive integer. The maximum value is the maximum, unsigned, 32-bit integer: 2147483647.	random()	Output default is uint32
sqrt	Square root	sqrt(scalar)	double,float

Example

This example illustrates a few of the algebraic functions.

1. Construct an array that contains a random set of values between 0 and 1.

```
AFL% create array C <val:double> [i=0:4,5,0];
```

```
AFL% store(build(C,random() /  2147483647.0),C);
```

```
{i} val
{0} 0.547088
```

```
{1} 0.554735
{2} 0.751838
{3} 0.935875
{4} 0.481854
```

The largest value that random() can produce is 2,147,483,647. So we divide the output by a double that is equivalent; this returns a double value between 0 and 1.

2. Try out the square root and power functions.

```
AQL% SELECT sqrt(i) AS root, pow(2.0,i) AS power FROM C;
```

```
{i} root,power
{0} 0,1
{1} 1,2
{2} 1.41421,4
{3} 1.73205,8
{4} 2,16
```

3. Try out the "simple" math functions, such as plus and minus.

```
AFL% store(build(<val:double> [i=0:1,2,0, j=0:1,2,0],i*2 +j+1),A);
```

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

```
AFL% store(build(A,-i*4-j -1),B);
```

```
[
[(-1),(-2)],
[(-5),(-6)]
]
```

```
AQL% SELECT A.val + B.val FROM A,B;
```

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

```
AQL% SELECT A.val - B.val FROM A,B;
```

```
[
[(2),(4)],
[(8),(10)]
]
```

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

```
[
[(-1),(-4)],
[(-15),(-24)]
]
```

```
AQL% SELECT A.val / B.val FROM A,B;
```

```
[
[(-1),(-1)],
[(-0.6),(-0.666667)]
]
```

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 (not a number)	is_nan(scalar)	double
is_null	Is NULL	is_null(scalar)	void
not	Boolean NOT	not(scalar)	bool

Example

This example demonstrates the `iif` and `is_null` functions.

1. Create an array that has an attribute with a default value of null.

```
AFL% create array A <val:double NULL default NULL> [i=0:2,3,0, j=0:2,3,0]
```

2. Use the `iif` function to store values along the diagonal of A.

```
AFL% store(build(A, iif(i=j, i, null)),A);
```

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

3. Illustrate the usage for the `is_null` function. We use the `apply` operator to create an attribute, **missing_value**, that holds the result of the `is_null` function.

```
AFL% apply(A,missing_value,is_null(val));
```

```
i,j,val,missing_value
0,0,0,false
0,1,null,true
0,2,null,true
1,0,null,true
1,1,1,false
1,2,null,true
2,0,null,true
2,1,null,true
2,2,2,false
```

As you can see, the attribute, **missing_value**, is true when **val** is null, and false otherwise.

Name

Temporal functions — Perform functions on datetime and datetimetz attributes

Synopsis

```
SELECT expression1 FROM array WHERE expression2;
```

Summary

The following table lists the time functions.

Function Name	Description	Syntax	Returned Data type
+ and -	Adds or subtracts the specified number of seconds to a datetime value.	datetime +/- int64	datetime
-	Subtracts one datetime value from another, returning the difference in seconds.	datetime - datetime	int64
append_offset	Appends a GMT offset, in seconds, to a datetime attribute.	append_offset(datetime,int64)	datetimetz
apply_offset	Converts a UTC datetime to local datetime by subtracting the offset timezone value, and stores the result as datetimetz including the offset timezone value.	apply_offset(datetime,int64)	datetimetz
day_of_week	Returns an integer value representing the day of the week: 0 (Sunday) through 6 (Saturday).	day_of_week(datetime), or day_of_week(datetimetz)	uint8
get_offset	Returns the timezone offset, in seconds.	get_offset(datetimetz)	int64
hour_of_day	Extracts the hour from a datetime attribute.	hour_of_day(datetime), or hour_of_day(datetimetz)	uint8
now	Returns the current date and time.	now()	datetime
strftime	Returns the the formatted date and time, based on the formatting options that you pass to the function. The arguments for the format string are the same as for the Linux strftime function.	strftime(datetime,string)	string
strip_offset	Returns the datetime value, after stripping the offset from a datetimetz argument.	strip_offset(datetimetz)	datetime
togmt	Converts a datetimetz to a datetime, adding in the timezone offset to return the GMT date and time that corresponds to the input.	togmt(datetimetz)	datetime
tznow	Returns the current date and time.	tznow()	datetimetz

Examples

This example illustrates the `day_of_week` and related `hour_of_day` functions.

1. Assume the following array, **dates**.

   ```
   AFL% show(dates);
   ```

   ```
   [('dates<dt:datetime> [i=0:*,10,0]')]
   ```

   ```
   {i} dt
   {0} '2012-04-12 12:14:01'
   {1} '2013-03-11 19:24:58'
   {2} '2009-11-22 16:05:19'
   {3} '1975-01-08 00:00:01'
   {4} '1969-12-31 19:00:01'
   ```

2. Calculate the day of the week for each date in the array.

   ```
   AFL% apply(dates, weekday, day_of_week(dt));
   ```

   ```
   i,dt,weekday
   0,'2012-04-12 12:14:01',4
   1,'2013-03-11 19:24:58',1
   2,'2009-11-22 16:05:19',0
   3,'1975-01-08 00:00:01',3
   4,'1969-12-31 19:00:01',3
   ```

   So, you can see that the first date is a Thursday, the second is a Monday, and the third is a Sunday.

3. Extract the hour for each datetime value in the array.

   ```
   AFL% apply(dates, time_hour, hour_of_day(dt));
   ```

   ```
   i,dt,time_hour
   0,'2012-04-12 12:14:01',12
   1,'2013-03-11 19:24:58',19
   2,'2009-11-22 16:05:19',16
   3,'1975-01-08 00:00:01',0
   4,'1969-12-31 19:00:01',19
   ```

Using the dates array that we created in the previous example, illustrate the `strftime` function.

- Get the date as **Month, day, year**.

  ```
  AFL% apply(dates,date,strftime(dt,'%B %d, %Y'));
  ```

  ```
  {i} dt,date
  {0} '2012-04-12 12:14:01','April 12, 2012'
  {1} '2013-03-11 19:24:58','March 11, 2013'
  {2} '2009-11-22 16:05:19','November 22, 2009'
  {3} '1975-01-08 00:00:01','January 08, 1975'
  {4} '1969-12-31 19:00:01','December 31, 1969'
  ```

- Get the time as **hour:minute:second AM/PM**.

  ```
  AFL% apply(dates,date,strftime(dt,'%l:%M:%S %p'));
  ```

  ```
  {i} dt,date
  {0} '2012-04-12 12:14:01','12:14:01 PM'
  {1} '2013-03-11 19:24:58',' 7:24:58 PM'
  {2} '2009-11-22 16:05:19',' 4:05:19 PM'
  {3} '1975-01-08 00:00:01','12:00:01 AM'
  {4} '1969-12-31 19:00:01',' 7:00:01 PM'
  ```

  There is a shorthand for this format, **%r**:

```
AFL% apply(dates,date,strftime(dt,'%r'));
```

```
{i} dt,date
{0} '2012-04-12 12:14:01','12:14:01 PM'
{1} '2013-03-11 19:24:58','07:24:58 PM'
{2} '2009-11-22 16:05:19','04:05:19 PM'
{3} '1975-01-08 00:00:01','12:00:01 AM'
{4} '1969-12-31 19:00:01','07:00:01 PM'
```

- Get the date in the ISO 8601 date format.

```
AFL% apply(dates,date,strftime(dt,'%F'));
```

```
{i} dt,date
{0} '2012-04-12 12:14:01','2012-04-12'
{1} '2013-03-11 19:24:58','2013-03-11'
{2} '2009-11-22 16:05:19','2009-11-22'
{3} '1975-01-08 00:00:01','1975-01-08'
{4} '1969-12-31 19:00:01','1969-12-31'
```

This example illustrates offsets for time zones. It uses the same array as the previous example, array, **dates**.

1. Apply an offset for each value in the array, **dates**.

```
AFL% store(project(apply(dates,dt_plusI,append_offset(dt,3600*(i
+1))),dt_plusI),datesTZ);
```

```
{i} dt_plusI
{0} '2012-04-12 12:14:01 +01:00'
{1} '2013-03-11 19:24:58 +02:00'
{2} '2009-11-22 16:05:19 +03:00'
{3} '1975-01-08 00:00:01 +04:00'
{4} '1969-12-31 19:00:01 +05:00'
```

We store the result array into **datesTZ**. The **datesTZ** array contains datetimetz values, where the offset added is based on the index of the value: one hour (3600 seconds) is added for the first value, two hours for the second, and so on.

2. Illustrate the SciDB functions that process timezone offset values.

   a. Use get_offset to return the offsets as integers.

```
AFL% apply(datesTZ,offset_integer,get_offset(dt_plusI));
```

```
{i} dt_plusI,offset_integer
{0} '2012-04-12 12:14:01 +01:00',3600
{1} '2013-03-11 19:24:58 +02:00',7200
{2} '2009-11-22 16:05:19 +03:00',10800
{3} '1975-01-08 00:00:01 +04:00',14400
{4} '1969-12-31 19:00:01 +05:00',18000
```

   b. Use strip_offset to strip the offsets, and return the datetime value, without the timezone offset.

```
AFL% apply(datesTZ,date_stripped,strip_offset(dt_plusI));
```

```
{i} dt_plusI,date_stripped
{0} '2012-04-12 12:14:01 +01:00','2012-04-12 12:14:01'
{1} '2013-03-11 19:24:58 +02:00','2013-03-11 19:24:58'
{2} '2009-11-22 16:05:19 +03:00','2009-11-22 16:05:19'
{3} '1975-01-08 00:00:01 +04:00','1975-01-08 00:00:01'
{4} '1969-12-31 19:00:01 +05:00','1969-12-31 19:00:01'
```

3. Get the date and time, as a datetimetz value, and then convert it to a datetime value that adds the GMT offset.

```
AFL% store(build(<current:datetimetz>[i=0:0,1,0],tznow()),nowDate);
```

```
[('2013-12-02 17:06:58 -05:00')]
```

The **nowDate** array has the current date and time. Now use the `apply` operator to return the GMT date and time.

```
AFL%  project(apply(nowDate,GMTdate,togmt(current)),GMTdate);
```

```
[('2013-12-02 22:06:58')]
```

This example illustrates the difference between `append_offset` and `apply_offset`.

1. Get the time and date as a datetime value, and store it to an array.

```
AFL% store(build(<current:datetime>[i=0:0,1,0],now()),currentDate);
```

```
[('2013-12-02 22:06:59')]
```

2. Assume that we know that our time is GMT -5 hours. We can use the `append_offset` function to convert our datetime value into the correct datetimetz value.

```
AFL% apply(currentDate, currentGMT, append_offset(current, (-5*3600)));
```

```
{i} current,currentGMT
{0} '2013-12-02 22:06:59','2013-12-02 22:06:59 -05:00'
```

3. Now, assume that we have a datetime value that we know is a GMT representation, but we want to convert it to being correct for our time zone, which happens to be GMT -5 hours. Here, we can use the `apply_offset` function to convert our datetime value into the correct datetimetz value.

```
AFL% apply(currentDate, currentGMT, apply_offset(current, (-5*3600)));
```

```
{i} current,currentGMT
{0} '2013-12-02 22:06:59','2013-12-02 17:06:59 -05:00'
```

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)]
```

Name

Miscellaneous functions — SciDB functions that do not fit into any of the other categories.

Synopsis

```
SELECT expression1 FROM array WHERE expression2;
```

Summary

These uncategorized functions perform useful tasks in SciDB.

Function Name	Description	Syntax	Data Types
first_index	Returns the initial dimension value of the supplied dimension. This function takes two arguments: • Array name as a string • Dimension name as a string (optional if the array has only one dimension)	first_index ('array') or first_index ('array','dimension')	string

high	Returns the maximum dimension value where the cell is non-empty. This function takes two arguments:
	• Array name as a string
	• Dimension name as a string (optional if the array has only one dimension)
		high('array')
		or
		high('array','dimension')
			string

instanceid	Returns the instance ID (int64) of the SciDB instance invoked. Usually, SciDB arrays are spread across some or all of the instances in the SciDB cluster. Thus, you can use the instanceid function to determine where the chunks of an array are stored.	instanceid()	None
last_index	Returns the final dimension value of the supplied dimension. This function takes two arguments: • Array name as a string • Dimension name as a string (optional if the array has only one dimension)	last_index ('array') or last_index ('array','dimension')	string

length	Returns the size of the supplied dimension. This function takes two arguments:
	• Array name as a string, and optionally,
	• Dimension name as a string (optional if the array has only one dimension)
		length('array')
		or
		length('array','dimension')
			string

Function Name	Description	Syntax	Data Types
	Since `length` is a function, it can be used with the `apply` operator in calculations where the size of the dimension is needed.		

low	Returns the minimum dimension value where the cell is non-empty. This function takes two arguments:
	• Array name as a string
	• Dimension name as a string (optional if the array has only one dimension)
		low('array')
		
		or
		low('array','dimension')
			string

missing	Returns the missing reason code from an integer array.	missing(scalar integer)	int32
missing_reason	Returns the missing reason code for array data.	missing_reason(mr_code)	all datatypes
regex	Regular expression matching function. Uses the standard regular expression language from the **boost** library. Returns a boolean.	regex('expr1','expr2')	string
strlen	Returns the string length as an int32.	strlen(string)	string

Examples

These examples illustrates the miscellaneous functions.

Use the `instanceid` function to check the distribution of chunks for array A1 over the instances. This example assumes that your SciDB cluster contains four instances.

1. Construct an array that contains 1 million random integers from 0 to 999.

   ```
   create array A1 <val:int32> [i=0:999,100,0, j=0:999,100,0];
   store(build(A1,random()/999),A1);
   ```

   The dimension sizes are both 1000; thus the array contains 1 million values, split into 100 chunks.

2. We can filter on the instance ID, and then count the number of elements on each instance.

   ```
   count(filter(apply(A1,instanceid, instanceid()),instanceid=0));
   ```

   ```
   [(260000)]
   ```

   ```
   count(filter(apply(A1,instanceid, instanceid()),instanceid=1));
   ```

   ```
   [(260000)]
   ```

   ```
   count(filter(apply(A1,instanceid, instanceid()),instanceid=2));
   ```

   ```
   [(260000)]
   ```

   ```
   count(filter(apply(A1,instanceid, instanceid()),instanceid=3));
   ```

   ```
   [(260000)]
   ```

Use the `strlen` function to return the lengths of strings in an array.

1. Assume we have the following array, Names:

```
AFL% show(Names)
```

```
Names

< firstnames:string,
lastnames:string >

[i=0:2,3,0,
j=0:1,2,0]
```

```
AFL% scan(Names);
```

```
[
[('Bill','Clinton'),('Anne','Rice')],
[('Joe','Pantoliano'),('Steve','Jobs')],
[('Burt','Reynolds'),('Ronald','Reagan')]
]
```

2. We can select the cells where the first name and last name contain the same number of characters.

```
AFL% filter(apply(Names,first,strlen(firstnames),
        last,strlen(lastnames)),first=last);
```

```
{i,j} firstnames,lastnames,first,last
{0,1} 'Anne','Rice',4,4
{2,1} 'Ronald','Reagan',6,6
```

Use the `regex` function to filter a list of strings, based on a regular expression.

1. Store the list of functions in an array.

```
AFL% store(list('functions'),SciDB_functions);
```

2. We use a regular expression to return all the elements that contains the string, 'datetimetz'.

```
AFL% filter(project(SciDB_functions,name,profile),
      regex(profile,'(.*)datetimetz(.*)'));
```

```
name,profile
'<','bool <(datetimetz,datetimetz)'
'<=','bool <=(datetimetz,datetimetz)'
'<>','bool <>(datetimetz,datetimetz)'
'=','bool =(datetimetz,datetimetz)'
'>','bool >(datetimetz,datetimetz)'
'>=','bool >=(datetimetz,datetimetz)'
'append_offset','datetimetz append_offset(datetime,int64)'
'apply_offset','datetimetz apply_offset(datetime,int64)'
'day_of_week','uint8 day_of_week(datetimetz)'
'get_offset','int64 get_offset(datetimetz)'
'hour_of_day','uint8 hour_of_day(datetimetz)'
'strip_offset','datetime strip_offset(datetimetz)'
'togmt','datetime togmt(datetimetz)'
'tznow','datetimetz tznow()'
```

Chapter 17. SciDB AFL Operator Categories

The AFL operators divide up into several areas:

- Aggregate: operators that involve an aggregate function call.
- Combine: operators that take two arrays as arguments, and return one result array.
- Compute: operators that compute or generate new data.
- Input/Output (I/O): operators that interact with the outside world.
- Math: operators that perform mathematical manipulations on the input array.
- Metadata: operators that read and write metadata.
- Meta-transformations: operators that change the properties of intermediate arrays.
- Read/Save: operators that save data to persistent arrays and read it back.
- Rearrange: operators that perform complex transformations.
- Winnow: operators that return a subset of the input array.

You can get a list of all of the available operators by running the following query:

```
$ iquery -aq "list('operators')"
```

17.1. Aggregate

The following SciDB operators perform aggregation.

Operator Name	Description
`aggregate`	Takes as input a set of 1 or more values and returns a SciDB array. The following aggregates are provided: • approxdc—approximate count • avg—average (mean) • count • max—maximum value • min—minimum value • prod—cumulative product • stdev—standard deviation • sum—cumulative sum • var—variance

Operator Name	Description
`analyze`	Produces a 1-dimensional result array where each cell describes some simple statistics about the values in one attribute of a stored array.
`cumulate`	Calculates a running aggregate along a single dimension of the input array.
`regrid`	Produces a result array that aggregates over non-overlapping subarrays of the input array.
`variable_window`	Selects nonempty cells from a variable size, 1-dimensional window.
`window`	Computes aggregates over moving window of an array.

17.2. Combine

The following SciDB operators take two input arrays and combine them to produce a result array.

Operator Name	Description
`concat`	Returns a result array that consists of all of the cells of the first array, followed by all of the cells of the second array.
`cross_join`	Returns a result array that is a cross-product join with equality predicates.
`join`	Returns a result array that combines the attributes of two input arrays at matching dimension values.
`merge`	Produces a result array by merging the data from two input arrays.
`substitute`	Returns a result array with a specified value substituted for null values in an array.

17.3. Compute

The following SciDB operators compute or generate new data.

Operator Name	Description
`apply`	Returns new attributes, calculated from supplied parameters.
`build`	Create (but not persist) an array from a schema, using an expression to fill the cells of the array. Use with the `store` operator to persist the array.

17.4. Input / Output

The following SciDB operators interact with the outside world.

Operator Name	Description
`input`	Reads in a system file, without saving it to an array.
`load`	Load data to an array from a file.
`load_library`	Loads a plugin.
`save`	Saves array data to a file.
`unload_library`	Unloads a plugin (does not take effect until the next time you restart SciDB).

17.5. Math

The following SciDB operators perform math on an input array, or linear algebra on a matrix.

Operator Name	Description
avg_rank	Ranks the elements of an array. Averages the rank for the tied values.
gemm	Produces ***AB*** + ***C***, given three matrix arrays ***A***, ***B***, and ***C***.
gesvd	Produces a result matrix containing any one of the three components of the singular value decomposition of a general matrix.
normalize	Produces a result array that scales the values of a vector.
quantile	Produces a set of quantiles of the specified array.
rank	Ranks the elements of an array. Tied values are all assigned the identical, integer rank.

17.6. Metadata

The following SciDB operators read and write metadata.

Operator Name	Description
attributes	Produces a 1-dimensional result array where each cell describes one attribute of a stored array.
CREATE ARRAY	Creates—and persists—an empty array, based on the supplied list of attributes and dimensions. There are many examples throughout the documentation.
dimensions	Lists details about the dimensions for the specified array.
help	Displays the operator signature for the specified operator.

list('arrays')
list('functions')
list('aggregates')
list('operators')
	Lists arrays / functions / aggregates / operators of SciDB namespace.

remove	Removes an array and its attendant schema definition from the SciDB database.
rename	Changes the array name.
show	Produces a result array whose contents describe the schema of an array you supply. Use **list('arrays')** to return the schema for all arrays.

17.7. Meta Transformations

The following SciDB operators change the properties of mid-query, intermediate arrays, without changing the underlying data.

Operator Name	Description
adddim	Adds a stub dimension.
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.

Operator Name	Description
`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.
`deldim`	Removes a dimension of size 1.
`repart`	Produces a result array similar to a source array, but with different chunk sizes, different chunk overlaps, or both.

17.8. Read / Save

The following SciDB operators save data to persistent arrays and read it back.

Operator Name	Description
`allversions`	Returns a result array containing all versions of the stored array.
`insert`	Insert values from a source array into a target array.
`scan`	Produces a result array that is equivalent to a stored array. That is, the scan operator reads a stored array.
`store`	Stores query output into a SciDB array.
`versions`	Shows versions for the specified array.

17.9. Rearrange

The following SciDB operators allow users to rearrange arrays. None of these operators change the input array. In most cases, the result array is not stored in the database. The exception is redimension_store, which stores the result array in the database (potentially overwriting the source array).

Operator Name	Description
`redimension`	Can change some or all of the array variables from dimensions to attributes or vice versa, and optionally calculate aggregates. To store the result of a redimension operation, use `store(redimension(...))`.
`reshape`	Produces a result array with the same cells as a given array, but a different shape.
`reverse`	Produces a result array that has the original values reversed in each array dimension.
`sort`	Produces a 1-dimensional result array by sorting non-empty cells of a source array.
`transpose`	Produces a result array that is the transpose of the input array.
`unpack`	Produces a one-dimensional result array from the data in a multi-dimensional source array.
`xgrid`	Multiplies the size of each dimension by an integer scale that you supply.

17.10. Winnow

The following SciDB operators return a subset of the input array.

Operator Name	**Description**
`bernoulli`	Produces a result array whose set consists of random array cells from the input array.
`between`	Produces a result array from a specified, contiguous region of a source array, and with empty cells everywhere else.
`filter`	Produces a result array, filtering out elements based on a supplied boolean expression.
`lookup`	Produces a result array, selecting array cells by supplied dimension index.
`project`	Produces a result array with the same dimensions as—but a subset of the attributes of—a source array.
`sample`	Produces a result array by selecting random chunks of a source array.
`slice`	Produces a result array that is a subset of the source array derived by holding one or more dimension values constant.
`subarray`	Produces a result array by selecting a contiguous area of cells.
`thin`	Produces a result array by selecting regularly spaced elements of the input array in each dimension.

Chapter 18. 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.

To see a categorized listing of the AFL operators, see Chapter 17, *SciDB AFL Operator Categories*.

You can get a list of all of the available operators by running the following query:

```
$ iquery -aq "list('operators')"
```

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

aggregate — The aggregate operator takes as input a set of 1 or more values and returns a SciDB array.

Synopsis

```
aggregate(array, aggregate_call_1
[, aggregate_call_2,... aggregate_call_N]
[,dimension_1, dimension_2,...]);
```

Summary

SciDB aggregates are functions of several values that produce a scalar value. They are used together with the `aggregate`, `redimension_store`, and `redimension` operators to compute data aggregations, optionally grouped along a specified dimension or dimensions.

The aggregate operator takes the following arguments:

- **Array name**: the aggregation is performed on data from the named array.
- **Aggregating functions**: this is the aggregation to be performed. You can list one or more aggregating functions. For a list of SciDB aggregates, see Aggregates.
- **Dimensions** (optional): most aggregates allow aggregation along one or more dimensions. You can list zero or more dimensions along which the aggregation is performed.

The aggregate operator returns a SciDB array, which contains the aggregation result as a new attribute. The name of this attribute is constructed as follows: *`inputAttribute_aggregateFunction`* , where *`inputAttribute`* is the name of the attribute being aggregated on, and *`aggregateFunction`* is the name of the aggregate function called. For example, assume an array **A** with an attribute **sales**. The name for the result attribute of **aggregate(A,sum(sales));** is 'sales_sum'.

Example

This example creates a sparse array and then performs some aggregation on it.

1. Create an array containing values along the diagonal, and empty cells everywhere else.

```
AFL% store(redimension (
         apply (
             build(<val:double>[i1=0:3,4,0, j1=0:3,4,0], i1),
         i, iif(i1=j1, i1,null), j, iif(i1=j1,i1,null)),
      <val:double>[i=0:3,4,0, j=0:3,4,0]),A);
```

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

2. Use the aggregate operator to count the number of non-empty cells.

```
AFL% aggregate(A, count(val));
```

```
val_count
4
```

3. Use the aggregate operator to calculate the sum of all cells in the array.

```
AFL% aggregate(A, sum(val));
```

```
val_sum
6
```

Name

allversions — Returns a result array containing all versions of the stored 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 array, 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 to return all three versions of m3x3:

   ```
   AFL% allversions(m3x3);
   ```

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

5. View the schema for the result array.

```
AFL% show('allversions(m3x3)','afl');
```

```
{i} schema
{0} 'm3x3<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,
   person:string,
   year:int64,
   country:string,
   time:double >

   [i=0:11,12,0]
   ```

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} 'person','string',false
   ```

```
{2} 'year','int64',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',6,12
{1} 'event','dash','steeplechase',3,12
{2} 'person','Abera','Wanjiru',12,12
{3} 'time','9.69','7956',12,12
{4} 'year','1996','2008',4,12
```

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','9.69','7956',12,12
{1} 'year','1996','2008',4,12
```

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.

To persist the new attribute names when you use `attribute_rename`, you must store the results to a new array—if you store to the original array, the original attribute names are left in place.

You can achieve the same functionality (and more) by using the `cast` operator. For details, see cast.

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 an array, and renames one attribute and excludes one attribute from the results.

1. Show the source_array:

```
{i} year,event,person,country,time
{0} 1996,'dash','Bailey','Canada',9.84
{1} 2000,'dash','Greene','USA',9.87
{2} 2000,'steeplechase','Kosgei','Kenya',503.17
{3} 1996,'marathon','Thugwane','Kenya',7956
{4} 2000,'marathon','Abera','Ethiopia',7811
{5} 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% store(attribute_rename(project(winnersFlat,year, event, country, time),
      time,time_in_seconds),result);
```

```
{i} year,event,country,time_in_seconds
{0} 1996,'dash','Canada',9.84
{1} 2000,'dash','USA',9.87
{2} 2000,'steeplechase','Kenya',503.17
{3} 1996,'marathon','Kenya',7956
{4} 2000,'marathon','Ethiopia',7811
{5} 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 output cell includes the following information:

- **No:** the sequence number of the attribute as it appears in the input array
- **name:** the attribute name
- **type_id:** the attribute data type
- **nullable:** a boolean flag representing whether or not the attribute can be null

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:

```
{i} event,person,year,country,time
{0} 'dash','Bailey',1996,'Canada',9.84
{1} 'dash','Greene',2000,'USA',9.87
{2} 'dash','Gatlin',2004,'USA',9.85
{3} 'dash','Bolt',2008,'Jamaica',9.69
{4} 'steeplechase','Keter',1996,'Kenya',487.12
{5} 'steeplechase','Kosgei',2000,'Kenya',503.17
{6} 'steeplechase','Kemboi',2004,'Kenya',485.81
{7} 'steeplechase','Kipruto',2008,'Kenya',490.34
{8} 'marathon','Thugwane',1996,'USA',7956
{9} 'marathon','Abera',2000,'Ethiopia',7811
{10} 'marathon','Baldini',2004,'Italy',7855
{11} 'marathon','Wanjiru',2008,'Kenya',7596
```

2. Create a result array describing the attributes of the named array:

```
AFL% attributes(winnersFlat);
```

```
{No} name,type_id,nullable
{0} 'event','string',false
{1} 'person','string',false
{2} 'year','int64',true
{3} 'country','string',true
{4} 'time','double',false
```

3. Create a result array describing the nullable attributes of the named array:

```
AFL% filter(attributes(winnersFlat),nullable=true);
```

```
{No} name,type_id,nullable
{2} 'year','int64',true
{3} 'country','string',true
```

Name

avg_rank — Rank elements of an array.

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 an array 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);
   ```

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

3. Rank the elements in rank_array by dimension i:

   ```
   AFL% avg_rank(rank_array,val,i);
   ```

   ```
   [
   [(6,3.5),(5,2),(6,3.5),(2,1)],
   [(0,1),(3,2),(4,3),(6,4)],
   [(6,4),(3,2),(4,3),(0,1)],
   [(3,2),(4,3),(5,4),(1,1)]
   ]
   ```

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. Use the optional, integer *`seed`* parameter to reproduce results; each run using the same seed returns identical results.

Example

This example select cells at random from a 5×5 array, 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);
   ```

   ```
   [
   [(),(2),(3),(),(5)],
   [(),(),(8),(),()],
   [(11),(12),(),(14),()],
   [(),(17),(18),(19),(20)],
   [(),(22),(),(24),()]
   ]
   ```

   ```
   AFL% bernoulli(bernoulli_array,0.5);
   ```

   ```
   [
   [(),(2),(3),(4),(5)],
   [(),(7),(),(),(10)],
   [(),(12),(),(14),(15)],
   [(16),(17),(),(),()],
   [(),(22),(),(),()]
   ]
   ```

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 [, true]);
```

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.

You can also supply a SciDB-text-formatted string as a literal to the `build()` operator. In this case, you can build an array with an arbitrary set of attribute values. An example is provided below.

Limitations

- The build operator can only take arrays with one attribute.
- The build operator can only take arrays with bounded dimensions.

Examples

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 array 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)]
]
```

In this example, we supply a SciDB-text-formatted string as an input to the `build()` operator.

```
AFL% create array A <v:double NULL> [r=0:2,3,0, c=0:3,4,0];
```

```
AFL% build(A, '[[(null),(), 1.25, -.45],[(?45), 6, 11.12, 7], [9,8,8, 4.1]]',true);
```

```
[
[(null),(),(1.25),(-0.45)],
[(?45),(6),(11.12),(7)],
[(9),(8),(8),(4.1)]
]
```

Name

build_sparse — Produces a sparse result array and assigns values to its non-empty cells.

Deprecated. The `build_sparse` operator may be removed in the future. We recommend that you use `redimension(build())` instead.

Synopsis

```
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.

Examples

In this example, we build a sparse array, and then a dense array. In the sparse array, only the diagonal elements are present, while in the dense array, only the diagonal elements have non-NULL values.

Here we create and store the sparse array 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 array 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)]
]
```

The remainder of the examples illustrate how to construct a sparse array using the `build()` operator—these are example you can use when the `build_sparse()` operator is removed from SciDB.

In this example, we create an array that has a random value every 50 cells (all other cells are empty).

1. Create an array to hold the eventual dimension index.

   ```
   AFL% store(build(<index:int64>[i=0:9,10,0], i*50), A);
   ```

   ```
   [(0),(50),(100),(150),(200),(250),(300),(350),(400),(450)]
   ```

2. Create an array to hold the values.

   ```
   AFL% store(build(<val:double>[i=0:9,10,0], random()%2000/2000.0), B);
   ```

   ```
   [(0.554),(0.7775),(0.198),(0.0285),(0.084),(0.5875),(0.5375),(0.037),(0.2465),
   (0.717)]
   ```

3. Combine A and B into a single array.

   ```
   AFL% store(join(A,B), test);
   ```

   ```
   {i} index,val
   {0} 0,0.554
   {1} 50,0.7775
   {2} 100,0.198
   {3} 150,0.0285
   {4} 200,0.084
   {5} 250,0.5875
   {6} 300,0.5375
   {7} 350,0.037
   {8} 400,0.2465
   {9} 450,0.717
   ```

4. Finally, redimension the array into a sparse array.

   ```
   AFL% redimension(test, <val:double>[index=0:*, 10,0]);
   ```

   ```
   {index} val
   {0} 0.554
   {50} 0.7775
   {100} 0.198
   {150} 0.0285
   {200} 0.084
   {250} 0.5875
   {300} 0.5375
   {350} 0.037
   {400} 0.2465
   ```

```
    {450} 0.717
```

Suppose you want to create a 3x3, 2-dimensional array, where you store the value 5.5 in all cells where the sum of the two dimensions is greater than 4.

```
If i+j > 4, store 5.5; else empty
```

The array looks like this:

```
[
[(),(),()],
[(),(),(5.5)],
[(),(5.5),(5.5)]
]
```

This query creates the array:

```
AFL% redimension(apply(build(<v:double>[i1=1:3,3,0,j1=1:3,3,0],5.5),
         i,iif(i1+j1>4,i1,null), j,iif(i1+j1>4,j1,null)),
         <v:double>[i=1:3,3,0,j=1:3,3,0]);
```

```
{i,j} v
{2,3} 5.5
{3,2} 5.5
{3,3} 5.5
```

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 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.

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]
   ```

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."

   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]
          ),
          winningTimeRecast
   ```

```
   );
```

```
{year} time_in_seconds
{1996} 9.84
{2000} 9.87
{2008} 7596
```

3. Show the schema of the new array:

```
AFL% show(winningTimeRecast)
```

```
winningTimeRecast

< time_in_seconds:double NULL DEFAULT null >

[year=1996:2008,1,0]
```

Name

concat — Returns a result array that consists of all of the cells of the first array, followed by all of the cells of the second array.

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 3×2 matrix and a 2×2 matrix.

1. Create a 3×2 matrix.

```
AFL% store(build(<val:double>[row=0:2,6,0, col=0:1,2,0],row*2+col),left_array);
```

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

2. Create a 2×2array.

```
AFL% store(build(<val:double>[row=0:1,6,0, col=0:1,2,0],pow( double(row*2+col
+1),2.0)),right_array);
```

```
[
[(1),(4)],
[(9),(16)]
]
```

3. Concatenate left_array and right_array.

```
AFL% concat(left_array,right_array);
```

```
[
[(0),(1)],
[(2),(3)],
[(4),(5)],
[(1),(4)],
[(9),(16)]
]
```

4. The schema of the result looks like this:

```
{i} schema
{0} 'left_array@1right_array@1<val:double> [row=0:4,6,0,col=0:1,2,0]'
```

Name

consume — Ensures that all data in the input array has been scanned.

Synopsis

```
consume(array[,numAttributesToScan]);
```

Summary

Causes all data to be scanned. The optional parameter, *`numAttributesToScan`*, determines the number of attributes to scan as a group.

- Setting *`numAttributesToScan`* to 1 results in a "vertical" scan; that is, all chunks of the current attribute are scanned before moving onto the next attribute. This is the default behavior.
- Setting *`numAttributesToScan`* to the maximum (the number of attributes in the input array) results in a "horizontal" scan; that is, chunk *`i`* of each attribute is scanned before moving onto chunk *`i`*+1.
- Setting *`numAttributesToScan`* to a value between 1 and the maximum results in a hybrid behavior: the number of attributes specified are scanned completely, and then the next group is scanned completely, and so on. For example, if your input array has 10 attributes, and you set *`numAttributesToScan`* to 2, the first 2 attributes are scanned completely, then the next 2, and so on until the final 2 are scanned.

The `consume()` operator does not return any information.

The purpose of this operator is to allow accurate timing of queries. For example, some queries set up a "delegate array," and instantly return a handle to it; the actual code that generates the data in the delegate array is not executed until the client pulls the data. If you are timing such a query, your results may be misleading.

Using the `consume()` operator ensures that all data is pulled, thus more accurately representing the amount of time it takes to execute your query.

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.

Note the following:

- **cross_join** performs an inner join on selected dimensions.
- Dimensions are not matched by order (as in **join**) nor by name, but by pairings that you explicitly provide.
- Array operands can have unmatched dimensions.
- Matching dimensions must have the same:
  - Chunk size
  - Chunk overlap
  - Data type

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

cumulate — Produces a result array containing a running aggregate along some flux vector (a single dimension of the input array).

Synopsis

```
cumulate(array, aggregate_call[,aggregate_call2,...] [, dimension]])
```

Summary

Calculates a running aggregate along some flux vector (a single dimension of the input array). The calculation is always done along one particular axis (dimension) of the array. The aggregate call must be one of the SciDB or P4 aggregates, or a user-defined aggregate function.

If the dimension is not specified, `cumulate` uses the first declared dimension of the array.

Limitation

For now, cumulate cannot handle an input array that has any non-zero chunk overlaps.

Examples

This example calculates the cumulative sum along the second dimension of a 2-dimensional, 1 -attribute array.

1. Create a matrix and store values into it:

```
AFL% store(build(<val:double>[row=0:2,3,0,col=0:3,4,0],row*4+col+1),A);
```

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

2. Calculate the cumulative sum along the first dimension:

```
AFL% cumulate(A,sum(val),row);
```

```
[
[(1),(2),(3),(4)],
[(6),(8),(10),(12)],
[(15),(18),(21),(24)]
]
```

3. Calculate the cumulative product along the first dimension:

```
AFL% cumulate(A,prod(val),row);
```

```
[
[(1),(2),(3),(4)],
[(5),(12),(21),(32)],
[(45),(120),(231),(384)]
]
```

4. Calculate the cumulative average along the second dimension:

```
AFL% cumulate(A,avg(val),col);
```

```
[
[(1),(1.5),(2),(2.5)],
[(5),(5.5),(6),(6.5)],
[(9),(9.5),(10),(10.5)]
]
```

5. Calculate the the average, min, and max values along the first dimension:

```
AFL% cumulate(A,avg(val),min(val),max(val),row);
```

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

This example uses an array that contains empty cells and null values.

1. Assume we have the following array, **B**.

```
AFL% scan(B)
```

```
[
[(1.25,null),(),(-7,-7)],
[(),(),(4.2,1)],
[(4.5,null),(11,9),(2.6,null)],
[(),(1.7,6.5),(6.2,null)]
]
```

2. Calculate the cumulative average along the second dimension, for both attributes.

```
AFL% cumulate(B,avg(val1),avg(val2),j);
```

```
[
[(1.25,null),(),(-2.875,-7)],
[(),(),(4.2,1)],
[(4.5,null),(7.75,9),(6.03333,9)],
[(),(1.7,6.5),(3.95,6.5)]
]
```

3. Calculate the variance for the first attribute, along the first dimension.

```
AFL% cumulate(B,var(val1));
```

```
[
[(null),(),(null)],
[(),(),(62.72)],
[(5.28125),(null),(36.6933)],
[(),(43.245),(34.28)]
]
```

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>[row=0:2000,10,0,column=1:10,10,0];
```

```
AFL% dimensions(array1);
```

```
{No} name,start,length,chunk_interval,chunk_overlap,low,high,type
{0} 'row',0,2001,10,0,4611686018427387903,-4611686018427387903,'int64'
{1} 'column',1,10,10,0,4611686018427387903,-4611686018427387903,'int64'
```

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'.

Examples

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),()]
   ]
   ```

This example uses a regular expression to filter the list of available SciDB operators, and return the operators that contain the letter **z**.

```
AFL% filter(list('operators'),regex(name,'(.*)z(.*)'));
```

```
name,library
'analyze','scidb'
'materialize','scidb'
'normalize','scidb'
```

Name

gemm — Produces ***AB*** + ***C***, given three matrix arrays ***A***, ***B***, and ***C***.

Synopsis

```
gemm(A_matrix, B_matrix, C_matrix);
```

Library

The `gemm` operator resides in the Dense Linear Algebra library. Run the following query to load this library:

```
AFL% load_library('dense_linear_algebra');
```

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

Limitations

- The first attribute of all three arrays must be of type `double`. All other attributes are ignored.
- The chunks of the input matrices must be square, and must have a chunk interval between 32 and 1024.
- 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 overlap must be zero.

Example

This example creates three matrices and calculates the gemm.

1. Ensure that the `dense_linear_algebra` library is loaded.

   ```
   AFL% load_library('dense_linear_algebra');
   ```

2. 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)]
]
```

3. 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 dense matrix.

Synopsis

```
gesvd(input_matrix, factor);
```

Library

The `gesvd` operator resides in the Dense Linear Algebra library. Run the following query to load this library:

```
AFL% load_library('dense_linear_algebra');
```

Summary

For a matrix *M*, GESVD returns the matrix factorization:

$$M = USV^T$$

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** $\times n$.

Limitations

The input matrix must have the following characteristics:

- It must be dense: that is, it cannot have any empty cells.
- The first attribute must be of type `double`. All other attributes are ignored.
- The chunks of the input array must be square, and must have a chunk interval between 32 and 1024.
- 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.

Example

This example creates a matrix and calculates its singular value decomposition.

1. Ensure that the `dense_linear_algebra` library is loaded.

   ```
   AFL% load_library('dense_linear_algebra');
   ```

2. 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);
   ```

   ```
   [
   [(0.866025),(-0.5)],
   [(0.5),(0.866025)]
   ]
   ```

3. 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);
   ```

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

4. Construct a rotation matrix, C, 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))),C);
   ```

   ```
   [
   [(0.866025),(0.5)],
   [(-0.5),(0.866025)]
   ]
   ```

5. Multiply the matrices together. The product becomes the input to the `gesvd` operator.

   ```
   AFL% store(gemm(gemm(A,B, build(A,0)),C, build(A,0)),product);
   ```

   ```
   [
   [(1.75),(0.433013)],
   [(0.433013),(1.25)]
   ]
   ```

6. Calculate the U, S, and VT decompositions.

   ```
   AFL% gesvd(product,'U');
   ```

   ```
   [
   [(-0.866025),(-0.5)],
   [(-0.5),(0.866025)]
   ]
   ```

   ```
   AFL% gesvd(product,'S');
   ```

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

   ```
   AFL% gesvd(product,'VT');
   ```

   ```
   [
   [(-0.866025),(-0.5)],
   ```

```
[(-0.5),(0.866025)]
]
```

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 — Displays the operator signature for the specified operator.

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 `show` operator.

```
AFL% help('show');
```

```
[('Operator: show
Usage: show(<array name | anonymous schema | query string [, \'aql\' | \'afl\']>)')]
```

Name

index_lookup — Uses an attribute from input array **A** to look up the coordinate value from another input array **B**.

Synopsis

```
index_lookup (input_array, index_array, input_array.attribute_name
    [,output_attribute_name] [,'index_sorted=true | false']
    [,'memory_limit=MEMORY_LIMIT']);
```

Summary

The required parameters are as follows:

- The *`input_array`* may have any attributes or dimensions.
- The *`index_array`* must have a single dimension and a single non-nullable attribute. The index array data must contain unique values. Empty cells are permitted, either between populated cells, or beyond all populated cells.
- The input attribute, *`input_array.attribute_name`*, must correctly refer to one of the attributes of the input array—the attribute to be looked up. This attribute must have the same data type as the sole attribute of the index array. The comparison < (less than) function must be registered in SciDB for this data type.

The optional parameters are as follows:

- Use *`output_attribute_name`* if you want to provide a name for the output attribute. **index_lookup** returns all of the attributes in the *`input_array`*, followed by the newly created output attribute. The new attribute is named **input_attribute_name_index** by default, or takes the optional, provided name.
- `index_sorted` is **false** by default. If you set this to **true**, the index array must be sorted, and it must be "dense," that is, empty cells are not permitted between the populated cells.
- Set the `memory_limit` parameter to increase or decrease the size of the memory cache. **index_lookup** uses memory to cache a part of the *`index_array`* for fast lookup of values. By default, the size of this cache is limited to **MEM_ARRAY_THRESHOLD**. Note that this is in addition to the memory already consumed by cached MemArrays as the operator is running. Note that the *`MEMORY_LIMIT`* value represents the number of megabytes for the cache, and must be at greater than or equal to 1, if entered.

Note the following characteristics for the output attribute:

- It will have data type int64 and be nullable.
- It will contain the respective coordinates of the corresponding input attribute in the *`index_array`*.
- If the corresponding input attribute is null, or if no value for the input attribute exists in the *`index_array`*, the output attribute at that position is set to null.

Example

This example shows how to return the index value for the stock symbol attribute of a small array populated with stock trading data.

1. Assume we have the following small array containing a data for a few stock trades.

```
AFL% show(trades)
```

```
trades

< symbol:string,
ms:int64,
volume:uint64,
price:double >

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

```
{i} symbol,ms,volume,price
{0} 'BAC',34665774,900,12.7
{1} 'BAC',36774769,11300,19.7
{2} 'AAPL',56512800,100,438.7
{3} 'C',55403661,100,46.5
{4} 'BAC',56395968,100,18.6
{5} 'ZNGA',30741156,500,7.1
{6} 'C',56377439,200,44.85
{7} 'MSFT',40979647,100,36.65
{8} 'FB',40515039,100,27.9
{9} 'JPM',39816561,100,55.5
```

2. Use the **uniq()** operator to create an index of the Stock symbols.

```
AFL% store(uniq(sort(project(trades,symbol))),stock_symbols);
```

```
{i} symbol
{0} 'AAPL'
{1} 'BAC'
{2} 'C'
{3} 'FB'
{4} 'JPM'
{5} 'MSFT'
{6} 'ZNGA'
```

3. Use **index_lookup()** to return the index value for each row of the **trades** array.

```
AFL% index_lookup(trades, stock_symbols, trades.symbol, index);
```

```
{i} symbol,ms,volume,price,index
{0} 'BAC',34665774,900,12.7,1
{1} 'BAC',36774769,11300,19.7,1
{2} 'AAPL',56512800,100,438.7,0
{3} 'C',55403661,100,46.5,2
{4} 'BAC',56395968,100,18.6,1
{5} 'ZNGA',30741156,500,7.1,6
{6} 'C',56377439,200,44.85,2
{7} 'MSFT',40979647,100,36.65,5
{8} 'FB',40515039,100,27.9,3
{9} 'JPM',39816561,100,55.5,4
```

Name

input — Populate a temporary array with data read from a file.

Synopsis

```
input(existing_array|anonymous_schema,input_file, instance_id, format[,[max_errors,
 shadow_array]])
```

Summary

The `input` operator interprets its arguments similarly to the way the `load` operator does, but rather than storing its result in the database it returns a temporary array.

The `input` operator takes the following parameters:

- *existing_array* | *anonymous_schema*: You can specify an existing array or provide an array schema for matching the data in the file to be input.
- *input_file*: the complete path to the file that contains the source data to be loaded.
- *instance_id*: **Optional.** Specifies the instance or instances for performing the input. If pieces of the input file exist on several nodes, you can perform the input in parallel.

  The default is to load all data from the coordinator instance. The value must be one of the following:

  - **-2**: Load all data using the coordinator instance of the query. This is the default.
  - **-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*: **Optional.** The default is to input data from a SciDB-formatted text file. The format string depends on the type of file that you are loading.
  - **Binary load:** If you are loading binary data, the load `operator` uses the format string as a guide for interpreting the contents of the binary file.
  - **OPAQUE load:** the string must be **opaque** or **OPAQUE**; you must have previously saved the array data in the OPAQUE format.
  - **SciDB-formatted text load:** If your text file is in SciDB format, use the string **text** or **TEXT**. This is the default.
- *max_errors*: **Optional.** Specifies 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*: **Optional.** Specifies a "shadow array"—SciDB writes this array into the array name that you provide. This is a mechanism by which you can keep track of errors while still loading the error-free values. The schema of shadow array is the same as output array, except that each attribute becomes a string data type. The shadow array also contains an additional attribute: **row_offset**. The row_offset attribute contains a position in the file where an error was detected.

Example

This example reads a SciDB-formatted file from the examples directory.

```
AFL% create array m4x4 <val1:int32, val2:int32>[i=0:3,4,0, j=0:3,4,0];
```

```
AFL% input(m4x4,'../tests/harness/testcases/data/doc/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)]
]
```

But note that the m4x4 array is empty—the data was not stored during the execution of the `input` operation.

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

```
[[]]
```

Name

insert — Insert values from a source array into a target array.

Synopsis

```
insert(source_array,target_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 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. The `merge` operator is also similar, except that merge is not a write operator (that is, merge does not store the result of the operation).

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.

   ```
   AFL% scan(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.

   ```
   AFL% scan(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,'2014-03-14 17:22:23'
   ```

```
{2} 2,'2014-03-14 17:22:23'
```

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

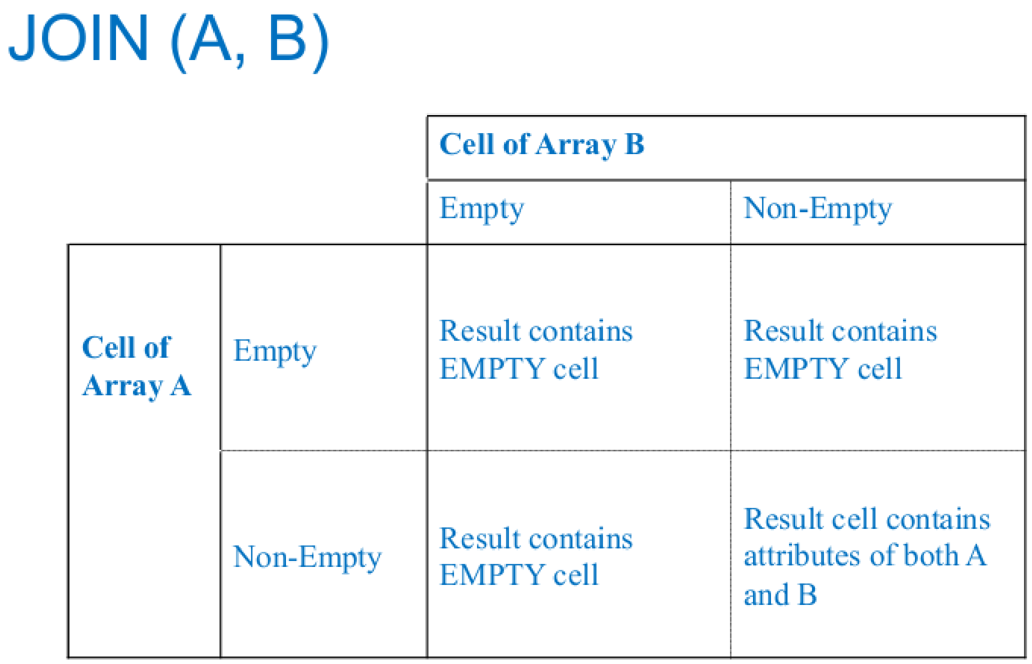
join — Join two arrays

Synopsis

```
join(left_array,right_array);
```

Summary

The following table shows the relationship between input and output cells.



JOIN (A, B)

		Cell of Array B	
		Empty	Non-Empty
Cell of Array A	Empty	Result contains EMPTY cell	Result contains EMPTY cell
	Non-Empty	Result contains EMPTY cell	Result cell contains attributes of both A and B

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 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.

Note the following:

- **join** performs an inner join on dimension values.
- Dimensions are not matched by name, but by their order (first of *`left_array`* with first of *`right_array`*, and so on).
- During the operation, compatibility of dimensions means the following:
  - Same number of dimensions
  - Same dimension boundaries for each corresponding pair
  - Same chunk and overlap values for each corresponding pair
  - Same starting index value for each corresponding pair

- In the result array, the dimension names match the dimension names of *`left_array`*.
- If the matching dimensions are not the same size, the smaller one appears in the result array.

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 — Lists 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, and the SciDB version information. For each instance, this parameter returns the SciDB version information, and the version information for each library that is loaded for that instance.
operators	Show all operators and the libraries in which they reside.
types	Show all the data types 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).

Invoking the list operator without an argument, `list()`, is equivalent to passing in the 'arrays' string. That is, the following statements are equivalent:

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

Example

This example shows sample output from the `list` operator.

1. List all arrays:

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

```
name,id,schema,availability
'vector0',210,'vector0<val:double> [i=0:4,5,0]',true
'vector1',212,'vector1<val:string> [i=0:4,5,0]',true
```

2. List all versions of all arrays:

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

```
name,id,schema,availability
'vector0',210,'vector0<val:double> [i=0:4,5,0]',true
```

```
'vector0@1',211,'vector0@1<val:double> [i=0:4,5,0]',true
'vector1',212,'vector1<val:string> [i=0:4,5,0]',true
'vector1@1',213,'vector1@1<val:string> [i=0:4,5,0]',true
```

3. List all SciDB aggregates:

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

```
name
'ApproxDC'
'avg'
'count'
'max'
'min'
'prod'
'stdev'
'sum'
'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`.

Inputs

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*: **Optional.** Specifies the instance or instances for performing the load. The default is to load all data from the coordinator instance. The value must be one of the following:
  - **-2**: Load all data using the coordinator instance of the query. This is the default.
  - **-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*: **Optional.** The default is to load data from a SciDB-formatted text file. The format string depends on the type of file that you are loading.
  - **Binary load:** If you are loading binary data, the load `operator` uses the format string as a guide for interpreting the contents of the binary file.
  - **OPAQUE load:** the string must be **opaque** or **OPAQUE**; you must have previously saved the array data in the OPAQUE format.
  - **SciDB-formatted text load:** If your text file is in SciDB format, use the string **text** or **TEXT**. This is the default.
- *max_errors*: **Optional.** Specifies 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*: **Optional.** Specifies a "shadow array"—SciDB writes this array into the array name that you provide. This is a mechanism by which you can keep track of errors while still loading the error-free values. The schema of the shadow array is the same as output array, except that each attribute becomes a string data type. The shadow array also contains an additional attribute: **row_offset**. The row_offset attribute contains a position in the file where an error was detected.

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,'../tests/harness/testcases/data/doc/
intensity_data.bin',-2,
   '(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',5,97
{6} 'High',6,null
{7} 'High',7,null
{8} 'Medium',0,100
{9} 'Medium',1,95
{10} 'Medium',2,89
{11} 'Medium',3,null
{12} 'Medium',4,null
{13} 'Medium',5,80
{14} 'Medium',6,78
{15} 'Medium',7,77
{16} 'Low',0,100
{17} 'Low',1,85
{18} 'Low',2,71
{19} 'Low',3,60
{20} 'Low',4,50
{21} 'Low',5,41
{22} 'Low',6,35
{23} '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,'../tests/harness/testcases/data/doc/
intensity_data.bin',-2,
   '(string, int64, int64)',99,shadowArray);
```

```
{i} exposure,elapsedTime,measuredIntensity
{0} 'High',0,25855
{1} '',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
```

Name

load_module — Load a module text file

Synopsis

```
load_module('module_pathname');
```

Summary

Load a SciDB module file. A module is text file that contains a collection of macros. Only one module can be loaded into SciDB at a time.

With respect to modules, note the following:

- The contents stay in memory until you restart SciDB, whereupon you will need to reload them.
- Once loaded, the macros become globally available to all subsequent queries evaluated on the SciDB cluster.
- Any call of `load_module` invalidates previously loaded macros, even if the particular macro is not defined in the newly loaded file.
- A macro named *m* hides any existing identically named applicable entity in the system, including operators, functions, and their user defined counterparts (UDO's and UDF's), and other macros.

Example

This example loads the example module, **module_1.txt**, where *`scidb_trunk`* is the base path to the top-level SciDB source folder.

```
load_module('/scidb_trunk/tests/harness/testcases/data/module_1.txt');
```

Name

load_library — Load a plugin

Synopsis

```
load_library('library_name');
```

Summary

Load a SciDB plugin. The act of loading a 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 libraries 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

This example loads the example plugin, **librational.so**.

```
load_library('rational');
```

Name

lookup — Produces a result array, selecting array cells by supplied 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. Run `lookup`, using 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

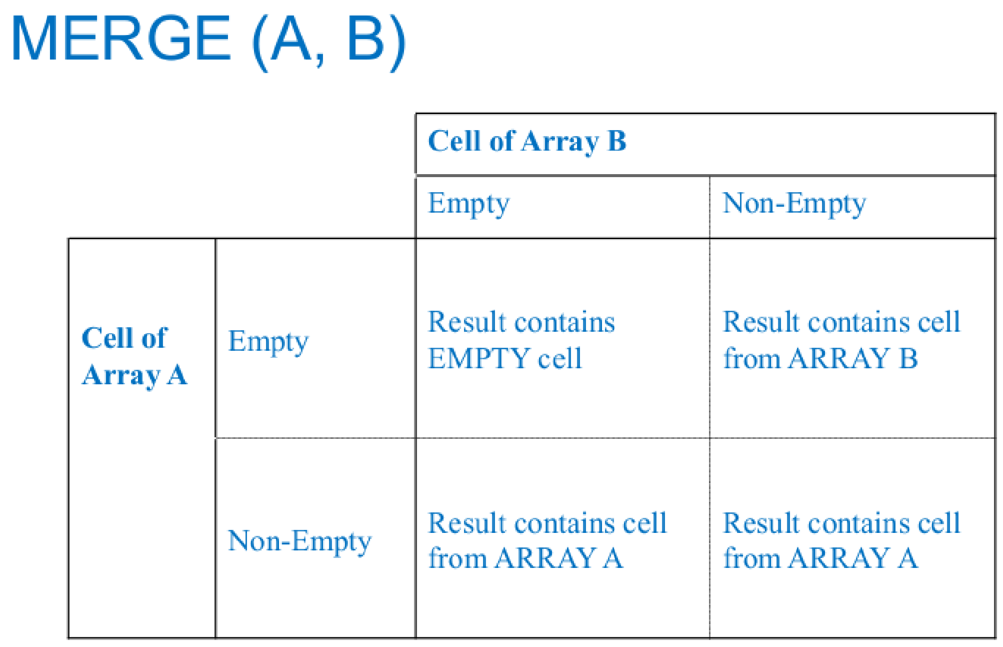
merge — Produces a result array by merging the data from two other arrays.

Synopsis

```
merge(left_array,right_array);
```

Summary

The following table shows the relationship between input and output cells.



MERGE (A, B)

		Cell of Array B	
		Empty	Non-Empty
Cell of Array A	Empty	Result contains EMPTY cell	Result contains cell from ARRAY B
	Non-Empty	Result contains cell from ARRAY A	Result contains cell from ARRAY A

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.

For each cell, `merge` combines elements from the input arrays the following way:

- If the cell of the first 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 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 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 (this is the default datatype for SciDB dimensions).

Example

This example merges two sparse arrays.

1. We have an array, **left_array** as follows:

```
[
[(1.1),(1.2),(1.3),(1.4),(1.5),(1.6)],
[(),(),(),(),(),()],
[(),(),(),(),(),()]
]
```

2. We have a sparse identity array called **right_array**.

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

3. Merge left_array and right_array:

```
AFL% merge(left_array,right_array);
```

```
[
[(1.1),(1.2),(1.3),(1.4),(1.5),(1.6)],
[(),(1),(),(),(),()],
[(),(),(1),(),(),()]
]
```

Name

mpi_init — Initializes the MPI infrastructure.

Synopsis

```
mpi_init();
```

Summary

The **mpi_init()** operator takes no arguments, and returns an empty array. It spawns MPI slave processes, gets their handshakes, and tells the slaves to exit in order to initialize (and test) the MPI infrastructure.

All MPI-based operators (e.g. **gemm()** and **gesvd()**, including `mpi_init()` are serialized with respect to each other. That is, they do not execute concurrently.

mpi_init() also has the side-effect of cleaning any MPI-related OS resources left behind by previous MPI-based queries (such as files, processes, IPC objects, an so on). **mpi_init()** is not required to be run, but it can be used in case of unexpected failures to return the MPI infrastructure to a clean state.

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','USA',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','USA',7956
   2000,'dash','USA',9.87
   2000,'steeplechase','Kenya',503.17
   2000,'marathon','Ethiopia',7811
   2008,'marathon','Kenya',7596
   ```

Name

quantile — Returns the quantiles of the specified 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.

Inputs

The `quantile` operator takes the following arguments:

- **array**: a source array with one or more attributes and one or more dimensions.
- **q-num**: the number of quantiles.
- **attribute_*n***: An optional attribute to use for the quantiles. If no attributes are specified, the first one is used.
- **dimension_*n***: An optional list of dimensions to group by.

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][, dimension_1, dimension_2,...]]);
```

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.

Inputs

The `rank` operator takes the following arguments:

- **array**: a source array with one or more attributes and one or more dimensions.
- **attribute_*n***: An optional attribute on which to sort. If no attributes are specified, the first one is used.
- **dimension_*n***: An optional list of dimensions to group by. If no dimensions are specified, the ordering is global across the entire array.

Example

This example ranks a 4×4 array by dimension.

1. Assume that you have the following 4×4 array called rank_array:

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

2. Rank the elements in rank_array by dimension i:

```
AFL% rank(rank_array,val,i);
```

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

Notice that the first value for each cell is the value of the **val** attribute, and the second value is the rank. Ties are marked in bold.

3. Rank all the elements of the array:

```
AFL% rank(rank_array,val);
```

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

Notice that the four occurrences of '9' are all ranked the same, as '13', which is the highest rank in the data set.

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_call_n (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.

Inputs

The `redimension` operator takes the following arguments:

- **source_array**: a source array with one or more attributes and one or more dimensions.
- **template_array | schema_definition**: an array or schema from which the output attributes and dimensions can be acquired. All the dimensions in the target must exist either in the source array attributes or dimensions, with one exception. One new, synthetic dimension is allowed. All the target attributes (other than those that are the results of aggregate calls) must exist either in the source array attributes or dimensions.
- **aggregate_call_*n***: Zero or more aggregate calls.

Limitations

- Each attribute or dimension to be changed must be of type int64.
- 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 can allow null values, but cells for which the attribute is null will not be included in the output.
- 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
             < $DOC_DATA/raw.csv >
               $DOC_DATA/raw.scidb
   ```

3. Load the data:

   ```
   AQL% LOAD raw FROM '../tests/harness/testcases/data/doc/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

regrid — Selects non-overlapping subarrays and performs aggregation on them.

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. For each block, regrid applies a specific aggregate operation over the value(s) of some attribute in each block.

Note that the chunk size does not have to be a multiple of the grid size, and the grids may span array chunks. However, processing is most efficient if chunk size is a multiple of the grid size, and grids do not span array chunks.

Inputs

The `regrid` operator takes the following arguments:

- **array**: a source array with one or more attributes and one or more dimensions.
- **grid_*n***: A list of block sizes, one per dimension.
- **aggregate_call_*n***: One or more aggregate calls.

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) as Average);
   ```

   ```
   [[(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.

Examples

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',731,'source<val:double> [i=0:9,10,0]',true
```

```
AFL% rename(source,target);
```

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

```
name,id,schema,availability
'target',731,'target<val:double> [i=0:9,10,0]',true
```

Unlike many operators, rename does not create a new array version; rather it renames all existing versions of the specified array.

1. Create several versions of an array.

   ```
   AFL% store(build(<val:int64> [i=0:3,4,0,j=0:1,2,0],i*4+j),A);
   ```

   ```
   [
   [(0),(1)],
   [(4),(5)],
   [(8),(9)],
   [(12),(13)]
   ]
   ```

   ```
   AFL% store(build(A,i*3+j),A);
   ```

   ```
   [
   [(0),(1)],
   [(3),(4)],
   [(6),(7)],
   [(9),(10)]
   ]
   ```

   ```
   AFL% store(build(A,i*2+j+1),A);
   ```

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

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

2. Show the details of all the versions for array **A**.

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

```
name,id,schema,availability
'A',733,'A<val:int64> [i=0:3,4,0,j=0:1,2,0]',true
'A@1',734,'A@1<val:int64> [i=0:3,4,0,j=0:1,2,0]',true
'A@2',735,'A@2<val:int64> [i=0:3,4,0,j=0:1,2,0]',true
'A@3',736,'A@3<val:int64> [i=0:3,4,0,j=0:1,2,0]',true
```

3. Rename array **A**.

```
AFL% rename(A,Octagon);
```

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

```
name,id,schema,availability
'Octagon',733,'Octagon<val:int64> [i=0:3,4,0,j=0:1,2,0]',true
'Octagon@1',734,'Octagon@1<val:int64> [i=0:3,4,0,j=0:1,2,0]',true
'Octagon@2',735,'Octagon@2<val:int64> [i=0:3,4,0,j=0:1,2,0]',true
'Octagon@3',736,'Octagon@3<val:int64> [i=0:3,4,0,j=0:1,2,0]',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.

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 illustrate 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 — Returns a result array, where the values in each array dimension are reversed.

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 1–12 into m3x4:

   ```
   AFL% store(build(m3x4,i*4+j+1),m3x4);
   ```

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

3. Reverse the values in m3x4:

   ```
   AFL% reverse(m3x4);
   ```

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

   Compare the `transpose` operator to reverse:

   ```
   AFL% transpose(m3x4);
   ```

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

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]))
   ```

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

Name

sample — Produces a result array by selecting random chunks of a source array.

Synopsis

```
sample(array,probability[,seed]);
```

Summary

The sample operator selects chunks from an array at random, subject to a probability you supply. Use the optional, integer *seed* parameter to reproduce results; each run using the same seed returns identical results.

To select random cells, as opposed to random chunks, use the bernoulli operator.

Example

This example selects random chunks from a 1-dimensional 8-chunk array.

1. Create a 2-dimensional array with dimension sizes of 6, and chunk sizes of 2 and 3:

   ```
   AFL% CREATE ARRAY vector1<val:double>[i=0:5,2,0, j=0:5,3,0];
   ```

2. Put values of 0–35 into vector1:

   ```
   AFL% store(build(vector1,i*6+j),vector1);
   ```

   ```
   [
   [(0),(1),(2)],
   [(6),(7),(8)]
   ];
   [
   [(3),(4),(5)],
   [(9),(10),(11)]
   ];
   [
   [(12),(13),(14)],
   [(18),(19),(20)]
   ];
   [
   [(15),(16),(17)],
   [(21),(22),(23)]
   ];
   [
   [(24),(25),(26)],
   [(30),(31),(32)]
   ];
   [
   [(27),(28),(29)],
   [(33),(34),(35)]
   ]
   ```

3. Sample chunks from the array with the probability of .30 that a chunk is included:

   ```
   AFL% sample(vector1,0.3);
   ```

   ```
   {2,0}[[(12),(13),(14)],[(18),(19),(20)]];[[(15),(16),(17)],[(21),(22),(23)]]
   ```

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*: **Optional.** Specifies the instance for the source array data. The default is to save all data onto the coordinator instance. 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*: **Optional.** The format string depends on the how you want to save the data. The default format is SciDB-formatted text. Note that you must include the *instance_id* parameter if you want to specify an output format.
  - **Binary save**: The format string must match the datatypes for the list of attributes. For example, if your array has three attributes, a string, a double, and an int64 that allows nulls, your format string would be, **'(string, double, int64 null)'**.
  - **OPAQUE save:** the string must be **opaque** or **OPAQUE**.
  - **Text save:** you can save the data into a variety of different text formats. For details, see Output Options. If you want to load the data back into SciDB at some point, the best text option is **'dense'**, as this format can be loaded back without needing to edit the data before loading. Note that saving floating point values into a text format can be lossy, since the storage format on disk is binary.

Examples

This example creates an array 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,'dense');
```

```
{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
```

This example saves a 2-dimensional array into the SciDB text format.

1. Assume you have the following array, saveExample:

```
AFL% show(saveExample)
```

```
saveExample

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

[i=0:24,5,0]
```

```
[('High',0,100,165),('High',1,100,285),('High',2,99,90),('High',3,99,398),
('High',4,98,177),('High',5,97,275),('High',6,null,465),('High',7,null,24),
('Medium',0,100,195),('Medium',1,95,127),('Medium',2,89,801),('Medium',3,null,967),
('Medium',4,null,789),('Medium',5,80,234),('Medium',6,78,978),('Medium',7,77,785),
('Low',0,100,601),('Low',1,85,133),('Low',2,71,177),('Low',3,60,238),
('Low',4,50,731),('Low',5,41,429),('Low',6,35,393),('Low',7,29,704),()]
```

2. You could save this array as a binary file, distributed across all instances, using the following query:

```
AFL% save(saveExample,'example.bin',-1,'(string, int64, int64 null, int64)');
```

Name

scan — Produces a result array that is equivalent to a stored array. That is, `scan` 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);
   ```

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

Name

show — Produces a result array whose contents describe the schema of an array you supply.

Synopsis

```
show(named_array);
```

or

```
show('query'[,'afl']);
```

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 also use `show` to return the schema for a query. This can be useful to preview the schema for a query, instead of having to first save the result of the query to an array. Use the optional string, **'afl'**, if the query is written in AFL.

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.

Examples

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.

```
AFL% show(champions);
```

```
{i} schema
{0} 'champions<person:string,country:string,time:double>
 [year=1996:2008,13,0,event_id=0:3,3,0]'
```

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_id} person
{1996,0} 'Bailey'
{1996,1} 'Thugwane'
{1996,2} 'Keter'
{2000,0} 'Greene'
{2000,1} 'Abera'
{2000,2} 'Kosgei'
{2004,0} 'Gatlin'
{2004,1} 'Baldini'
{2004,2} 'Kemboi'
{2008,0} 'Bolt'
{2008,1} 'Wanjiru'
{2008,2} '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_id=0:3,3,0]'
```

This example illustrates the usage of `show` to return schema for queries.

1. Create two arrays, and fill them with some data.

   ```
   AFL% store(build(<val1:double> [i=0:4,32,0, j=0:0,32,0], i),A);
   ```

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

   ```
   AFL% store(build(<val2:double> [i=0:0,32,0, j=0:4,32,0], 10-j),B);
   ```

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

   ```
   AFL% store(build(<val3:double> [i=0:4,32,0, j=0:4,32,0], 0),C);
   ```

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

2. View the schema of the result of multiplying A and B (using the `gemm` operator).

   ```
   AFL% show('gemm(A,B,C)','afl');
   ```

   ```
   [('GEMM<gemm:double> [i=0:4,32,0,j=0:4,32,0]')]
   ```

3. You can use the show operator to view the output type of an aggregate.

   ```
   AFL% show('select sum(val1) from A');
   ```

   ```
   [('not empty A@1<sum:double NULL DEFAULT null> [i=0:0,1,0]')]
   ```

   ```
   AFL% show('select stdev(val1) from A');
   ```

   ```
   [('not empty A@1<stdev:double NULL DEFAULT null> [i=0:0,1,0]')]
   ```

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 and fill it with values of 0–8:

```
AFL% store(build(<val:double>[i=0:2,3,0,j=0:2,3,0],i*3+j),m3x3);
```

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

2. Select the middle column of m3x3:

```
AFL% slice(m3x3,j,1);
```

```
{i} val
{0} 1
{1} 4
{2} 7
```

You can accomplish the same task using the between operator, as shown in the following query:

```
AFL% redimension(between(m3x3,0,1,2,1),<val:double>[i=0:2,3,0]);
```

```
{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 ]]...
  [, 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 named **n**, and 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.

To sort by dimensions, you can use the `apply` operator to return a dimension as an attribute, and then sort.

Inputs

The `sort` operator takes the following arguments:

- **array**: a source array with one or more attributes and one or more dimensions.
- **attribute**: Zero or more attributes. If no attribute is provided, the first attribute in the source array is used.
- **chunk_size**: An optional chunk size for the result array.

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_id} person,country,time
{1996,0} 'Bailey','Canada',9.84
{1996,1} 'Thugwane','USA',7956
{1996,2} 'Keter','Kenya',487.12
{2000,0} 'Greene','USA',9.87
{2000,1} 'Abera','Ethiopia',7811
{2000,2} 'Kosgei','Kenya',503.17
{2004,0} 'Gatlin','USA',9.85
{2004,1} 'Baldini','Italy',7855
{2004,2} 'Kemboi','Kenya',485.81
{2008,0} 'Bolt','Jamaica',9.69
{2008,1} 'Wanjiru','Kenya',7596
{2008,2} '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(project(winnersFlat,country, year, event), 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'
```

To illustrate how to sort by a dimension, consider the following example.

1. Create a small, two dimensional array.

```
AFL% store(build(<val:double>[i=0:2,3,0, j=0:2,3,0],i%3+j),A);
```

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

2. Use the apply operator to return the first dimension, and then sort by it.

```
AFL% sort(apply(A,dim,i),dim);
```

```
{n} val,dim
{0} 0,0
{1} 1,0
{2} 2,0
{3} 1,1
{4} 2,1
{5} 3,1
{6} 2,2
{7} 3,2
{8} 4,2
```

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:

```
[(0),(1),(2),(null),(4)]
```

2. Sort by number (ascending):

```
AFL% sort(numbers, number asc);
```

```
[(null),(0),(1),(2),(4)]
```

3. Sort by number (descending):

```
AFL% sort(numbers, number desc);
```

```
[(4),(2),(1),(0),(null)]
```

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.

You can also use `store()` to reclaim storage space to Linux file system. As you update an array, new versions are created, and more space is used to store the array versions. When an array is deleted, all the file space is returned to the operating system. If, however, you want to keep the array, but do not care about previous versions, you can copy the array, and then delete the original.

Examples

Build and store a 2-dimensional, 1-attribute array of zeros:

```
AFL% store(build(<val: double>[i=0:2,3,0,j=0:2,3,0],0),zeros_array);
```

You can create a new array from the existing array's schema, store cell values of 1 with a store statement:

```
AFL% store(build(zeros_array,1),ones_array);
```

Build and store a 2-dimensional, 1-attribute array 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);
```

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

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);
```

```
[
[(8),(9),(3),(1)],
[(4),(9),(0),(0)],
[(7),(4),(9),(0)],
[(0),(7),(9),(3)]
]
```

Remove all versions of an array, except for the current one, to reclaim storage.

1. Assume you have an array that has several versions.

   ```
   AFL% aggregate(versions(A),count(*));
   ```

   ```
   {i} count
   {0} 5
   ```

2. Copy **A** to a new array, **B**:

   ```
   AFL% store(A,B);
   ```

3. Remove **A**.

   ```
   AFL% remove(A);
   ```

4. Optionally, change the array name back to the original.

   ```
   AFL% rename(B,A);
   ```

5. Confirm that now, there is only one version of the array.

   ```
   AFL% aggregate(versions(A),count(*));
   ```

   ```
   {i} count
   {0} 1
   ```

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 the result array non-nullable.

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.

   ```
   AFL% load(m4x4_null, '../tests/harness/testcases/data/doc/
   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

thin — Produces a result array by selecting 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_n*** and the number of cells to skip using ***step_n*** for each dimension of the input array. Note that a step of 1 means to select everything.

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 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. Transpose m3x3:

   ```
   AFL% transpose(m3x3);
   ```

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

   Compare the `reverse` operator to transpose:

   ```
   AFL% reverse(m3x3);
   ```

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

Name

uniq — Returns a result array with duplicate values removed.

Synopsis

```
uniq(array[,'chunk_size=chunk_size']);
```

Summary

The uniq operator takes as input a one-dimensional, sorted array, and returns an array with all duplicate values removed. It is analogous to the Unix **uniq** command.

Note the following:

- The input array must have a single attribute of any type and a single dimension.
- For **uniq()** to work correctly, the data in the input array must be sorted. The operator was designed to accept the output produced by **sort()** (assuming the input has a single attribute).
- The result array has the same attribute as the input array, and its sole dimension is named **i**, starting at 0, and with a chunk size of one million (1,000,000). You can control the chunk size of the resulting array by setting the optional parameter to a specific value, `chunk_size`.
- Data is compared using a simple bitwise comparison of underlying memory.
- Null values are discarded from the output.

Examples

This example sorts a one-dimensional array, and then removes the duplicate values.

1. Assume the following 2-dimensional array is loaded into SciDB:

```
AFL% scan(A);
```

```
[
[(1.15),(4.54),(1.54),(1.83)],
[(4.14),(4.99),(3.56),(1.15)],
[(1.54),(null),(0.61),(3.99)],
[(4.14),(3.14),(3.56),(null)]
]
```

2. Now, use **sort()** to prepare the array.

```
AFL% store(sort(A),A_sorted);
```

```
[(null),(null),(0.61),(1.15),(1.15),(1.54),(1.54),(1.83),(3.14),(3.56),(3.56),
(3.99),(4.14),(4.14),(4.54),(4.99)]
```

3. Remove nulls and duplicate values using **uniq()**, and set the chunk size of the result array to 10.

```
AFL% uniq(A_sorted,'chunk_size=10');
```

```
[(0.61),(1.15),(1.54),(1.83),(3.14),(3.56),(3.99),(4.14),(4.54),(4.99)]
```

This example projects an attribute from an array that contains multiple attributes, and then removes duplicate values.

1. Assume the following array, **winnersFlat**, is loaded into SciDB:

```
i,event,person,year,country,time
0,'dash','Bailey',1996,'Canada',9.84
1,'dash','Greene',2000,'USA',9.87
2,'dash','Gatlin',2004,'USA',9.85
3,'dash','Bolt',2008,'Jamaica',9.69
4,'steeplechase','Keter',1996,'Kenya',487.12
5,'steeplechase','Kosgei',2000,'Kenya',503.17
6,'steeplechase','Kemboi',2004,'Kenya',485.81
7,'steeplechase','Kipruto',2008,'Kenya',490.34
8,'marathon','Thugwane',1996,'USA',7956
9,'marathon','Abera',2000,'Ethiopia',7811
10,'marathon','Baldini',2004,'Italy',7855
11,'marathon','Wanjiru',2008,'Kenya',7596
```

2. Project the year, and view the sorted, unique values.

```
AFL% uniq(sort(project(winnersFlat,year)),'chunk_size=4');
```

```
[(1996),(2000),(2004),(2008)]
```

3. **uniq()** works on string attributes as well.

```
AFL% uniq(sort(project(winnersFlat,country)),'chunk_size=4');
```

```
[('Canada'),('Ethiopia'),('Italy'),('Jamaica'),('Kenya'),('USA'),(),()]
```

Name

unload_library — Unload a plugin

Synopsis

```
unload_library('library_name')
```

Summary

Unload a plug-in from the current SciDB instance. The unload_library operator provides the same functionality as the AQL UNLOAD LIBRARY '*library_name*' statement.

Note

The unload_library operation does not take effect until the next time you restart SciDB.

Example

This example loads and unloads the example plugin, librational.so.

```
load_library('rational')
unload_library ('rational')
```

The 'lib' prefix and the file extension are not included in the library name when you use the unload_library operator.

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.

One use case for unpack is for saving a multidimensional array into a binary backup file. The way to do this is to first unpack the array, and then use the `save` operator to perform a binary save. You perform a binary save similarly to the way you perform a binary load—see Section 6.4, “Loading Binary Data” for details.

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.

```
[('A<value:string NULL DEFAULT null> [row=1:3,3,0,col=1:3,3,0]')]
```

```
[
[(),(),()],
[(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.

This example shows how to use unpack to backup an array into a binary file.

1. Assume we have the following array, **Names**:

```
AFL% show(Names)
```

```
Names

< firstnames:string,
lastnames:string >

[i=0:2,3,0,
j=0:1,2,0]
```

```
AFL% scan(Names);
```

```
[
[('Bill','Clinton'),('Anne','Rice')],
[('Joe','Pantoliano'),('Steve','Jobs')],
[('Burt','Reynolds'),('Ronald','Reagan')]
]
```

2. Unpack **Names** into a 1-dimensional array, **namesFlat**.

```
AFL% store(unpack(Names,x),namesFlat);
```

```
{x} i,j,firstnames,lastnames
{0} 0,0,'Bill','Clinton'
{1} 0,1,'Anne','Rice'
{2} 1,0,'Joe','Pantoliano'
```

```
{3} 1,1,'Steve','Jobs'
{4} 2,0,'Burt','Reynolds'
{5} 2,1,'Ronald','Reagan'
```

3. View the schema for **namesFlat**, so that we can create the query for the binary save.

```
AFL% show(namesFlat)
```

```
namesFlat

< i:int64,
j:int64,
firstnames:string,
lastnames:string >

[x=0:*,6,0]
```

4. So we have two int64 types, followed by two string types. Now we perform the binary save.

```
AFL% save(namesFlat,'/tmp/namesFlat.bin',0, '(int64,int64,string,string)');
```

   This writes the binary file, **namesFlat.bin**, to the /tmp folder.

Name

variable_window — Calculates aggregate values over nonempty cells from a variable size, 1-dimensional window.

Synopsis

```
variable_window(array, dimension,left_edge, right_edge,
   aggregate_1(attr_name_1)[,aggrgegate_2(attr_name_2), ...])
```

Summary

The `variable_window` command aggregates along a 1-dimensional window of variable length. The window is defined by the left and right edges and excludes empty cells.

Inputs

The `variable_window` operator takes the following arguments:

- **array**: a source array with one or more attributes and one or more dimensions.
- **dimension**: the dimension along which the window is defined.
- **left_edge**: the number of cells to the left of the current cell to include in the window.
- **right_edge**: the number of cells to the right of the current cell to include in the window.
- **aggregate_***n*(*`attr_name`*): one or more aggregate calls, over the specified attribute, *`attr_name`*,

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,'2013-05-31 14:41:45'
   {2} 2,'2013-05-31 14:41:46'
   ```

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 — Produces a result array where each output cell is some aggregate calculated over a window around the corresponding cell in the source array.

Synopsis

```
window(array, dim_1_low,dim_1_high, [dim_2_low,dim_2_high,]...
   aggregate_1(attr_name_1)[,aggrgegate_2(attr_name_2), ...])
```

Summary

Computes one or more aggregates of any of an array's attributes over a moving window. The result array has the same size and dimensions as the source array.

Note

The AFL window operator provides the same functionality as the AQL **`SELECT`** ... **`FROM`** ... **`WINDOW`** statement.

Inputs

The `window` operator takes the following arguments:

- **array**: a source array with one or more attributes and one or more dimensions.
- **dim_*n*_low**: for each dimension, 1 ... n, this argument specifies the number of cells to the left of the current cell to include in the window.
- **dim_*n*_high**: for each dimension, 1 ... n, this argument specifies the number of cells to the right of the current cell to include in the window.
- **aggregate_*n*(*`attr_name`*)**: one or more aggregate calls, over the specified attribute, *`attr_name`*,

Example

This example calculates a running maximum and minimum 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. Tuning your SciDB Installation

This chapter 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.

The default values of the configuration parameters are described in the "SciDB Configuration Parameters" section of the Installation chapter.

A.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 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.

Note

These parameters are per-instance. For example, if you set *`smgr-cache-size`* to 25 Gigabytes, you are allowing the storage manager cache *on each instance* to use up to 25 Gigabytes of memory.

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

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

and

```
(MAX_NUMBER_OF_QUERIES *
  max(network-buffer, merge-sort-buffer * result-prefetch-threads)
  + 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.

A.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.

A.3. Optimizing Configuration Parameters

Beyond using the configuration parameters for optimizing your SciDB installation, this section provides a few useful guidelines.

- The max-memory-limit parameter

- How to identify unreleased locks
- Determine optimal chunk size
- Useful Linux commands

A.3.1. Max-memory-limit

You should **always** set the *`max-memory-limit`* parameter.

If this parameter is not set, you will not receive "out of memory" notifications. If a query runs up against the limit, and then tries to use more memory than is available, Linux kills the process.

Note that if you are using the Paradigm4 add-ons to SciDB, the `system` plugin contains code that helps you detect failures sooner.

A.3.2. Unreleased Locks

It is possible for SciDB to get into a state where a database lock was never released. This can happen if a write query fails in some way. Now, SciDB is in a state where the next time a query attempts to write to that array, the write will fail.

If a query appears to be running longer than anticipated, you can check the array to see if it is in this state. You can use the `list` operator to check the availability of all of your arrays.

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

```
name,id,schema,availability
'A',23,'A<val:double> [i=0:2,32,0,j=0:1,32,0]',false
```

Here, you can see that the availability of array A is false. This can indicate that there is an unreleased lock for this array.

To recover from this situation, you need to restart SciDB.

A.3.3. Chunk Size

The chunk size for your array can have a negative impact on memory usage in either of the following ways:

- **Chunk size is too big.** If you have a chunk that you thought would be sparser than it turned out to be, the chunk may be too big for the available memory.
- **Too many small chunks.** If you have lots of nearly empty chunks, then the sheer number of chunks may be too large for the available memory, since the chunk map needs to fit in memory.

You can use the following query (listed on the SciDB forum, as well) to analyze your chunks.

```
project(
 cross_join(
  redimension(
   apply(filter(list('chunk map'), inst=instn and attid=0), iid, int64(inst), aid,
 int64(arrid)),
   <nchunks:uint64 null,
    min_ccnt:uint32 null,
    avg_ccnt:double null,
    max_ccnt:uint32 null,
    total_cnt: uint64 null>
   [iid = 0:*,1000,0, aid= 0:*,1000,0],
   count(*) as nchunks,
   min(nelem) as min_ccnt,
   avg(nelem) as avg_ccnt,
   max(nelem) as max_ccnt,
   sum(nelem) as total_cnt
  ) as A,
  redimension(
   apply( list('arrays', true), aid, int64(id)),
   <name: string null>
   [aid = 0:*,1000,0]
  ) as B,
  A.aid, B.aid
 ),
 name, nchunks, min_ccnt, avg_ccnt, max_ccnt, total_cnt
);
```

Output looks like this:

```
 iid,aid,name,nchunks,min_ccnt,avg_ccnt,max_ccnt,total_cnt
 0,79,'flat@1',5,1000000,1e+06,1000000,5000000
 0,81,'matrix@1',5,999546,1.00017e+06,1001470,5000865
 0,85,'svd_result@1',3000,10000,10000,10000,30000000
 1,79,'flat@1',5,1000000,1e+06,1000000,5000000
 1,81,'matrix@1',5,998209,999826,1000741,4999128
 1,85,'svd_result@1',3002,3,9993.34,10000,30000013
```

- **iid** is the Instance ID; this example is using two instances
- **aid** is the versioned array ID; this example contains 3 arrays
- **nchunks** is the number of chunks for the given array on that instance
- **ccnt** is the chunk count—the number of cells per chunk
- **min_ccnt**, **avg_ccnt**, and **max_ccnt**; the query collects min, max and average of **ccnt**.
- **total_cnt** is the total number of cells, for the specified array and instance

This query can help detect cross-chunk skew and cross-instance skew.

A.3.4. Useful Linux Commands

The **ulimit** command provides control over the resources available to the shell and to processes started by it. We recommend that you always set this limit to **unlimited** by issuing the following command:

```
sudo ulimit -c unlimited
```

This allows Linux to dump core files in the event of a crash. You can then use these dumps to diagnose the problem, or send us the details so that we can identify the issue.

Also, in the event of a crash, run **dmesg** on each SciDB server. You use the **dmesg** command to write out the kernel messages in Linux. This will provide additional debugging information.

Before a crash, you can run the Linux utility **pstack**. It presents a well-formatted, multi-threaded picture of the running processes.

A.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 the "SciDB Configuration Parameters" section in the Installation chapter.

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`*.

Appendix B. Troubleshooting

This appendix contains troubleshooting information.

B.1. MPI Issues

MPICH is a high performance and widely portable implementation of the Message Passing Interface (MPI) standard. SciDB depends upon **mpich2-1.2** being installed and configured. This section provides a checklist for ensuring that MPICH is communicating with SciDB.

- If you encounter any errors or problems related to MPI, try running `mpi_init()`. For details, see mpi_init.
- SSH connectivity must be set up for the scidb user from the coordinator to 0.0.0.0, 127.0.0.1, localhost, and all the workers. For each worker:
  - Make sure that you copy the authorization key, as described in section “Remote Execution Configuration (SSH)” in the Installation chapter.
  - Log in once: ssh scidb@<*`worker`*>. At the following prompt, answer **yes**.

    ```
    Are you sure you want to continue connecting (yes/no)?
    ```

  - Confirm it works by running: scidb@<*`worker`*> again. This must take you directly to a shell prompt on the worker. Otherwise, SSH is not yet set up on this worker.
- DNS must be configured on all SciDB servers. In particular, verify that the worker instances are able to resolve the coordinator host name to the correct IP.
- If a host has a static IP, make sure to replace **127.0.1.1** in the **/etc/hosts** file with the static IP. In general, the **/etc/hosts** file should not contain multiple IPs mapped to the same name. For example, consider the following section of the **/etc/hosts** file (for a machine with hostname **test-u1204-c2-vm1**):

  ```
  127.0.0.1      localhost
  127.0.1.1      test-u1204-c2-vm1
  ```

  The second line can confuse MPI. You should comment it out or enter the actual static IP for **test-u1204-c2-vm1**. The practice of using **127.0.1.1** is specific to Ubuntu—for details, see the Ubuntu documentation (link to Debian Reference, chapter 10: http://qref.sourceforge.net/quick/ch-gateway.en.html).
- Configuration with multiple Network Interface Cards (NICs), such as eth0, eth1, and so on, has not been tested. We recommend that you disable all but one of your NICs. If you cannot disable all but one of your NICs, make sure that the DNS names resolve to the correct IPs.
- Make sure there is enough shared memory available. On each SciDB server, run the following command to see your shared memory usage:

  ```
  $ df -h /dev/shm
  ```

  Your output should look similar to the following:

  ```
  Filesystem      Size  Used Avail Use% Mounted on
  none            3.8G  320K  3.8G   1% /run/shm
  ```

  Your available shared memory needs to be at least 512 MB * #instances_per_host. You can change the size of shared memory by adding a line to the `/etc/fstab` file:

```
# shared memory device
none    /dev/shm     tmpfs   defaults,size=48G     0 0
```

Running out of the shared memory from `/dev/shm` usually manifests itself by the SciDB process being killed with the **SIGBUS** signal as reported in the SciDB error log (**scidb-stderr.log**):

```
2013-5-13 23:17:10 (ppid=23581): Started.
2013-5-14 0:2:0 (ppid=23581): SciDB child (pid=23604) terminated by signal = 7, core
 dumped
```

If you have run out of shared memory, you can increase the value for the `max-memory-limit` parameter in your SciDB config.ini file.

B.2. Configuration Issues

The Troubleshooting appendix contains a section on some guidelines for optimizing the SciDB configuration parameters. For details, see Section A.3, “Optimizing Configuration Parameters”.

Also, when setting up the config.ini file, we recommend that you use absolute IP addresses (rather than **localhost**) whenever possible, especially on multi-server SciDB installations.

B.3. Out Of Memory Issues

If you encounter any out-of-memory issues, try changing the value for the `max-memory-limit` configuration parameter. For details, see Section A.1, “Configuring Memory Usage”.

B.4. Other Issues

If you notice odd behavior or any performance degradation, try restarting your SciDB cluster.

From the scidb user account, run the following commands from your Linux prompt to restart:

```
$ scidb.py stopall scidb_cluster_name
```

After the cluster stops, start it again:

```
$ scidb.py startall scidb_cluster_name
```

In the previous commands, replace *`scidb_cluster_name`* with the actual name of your SciDB cluster.

Appendix C. Licenses

This appendix contains details for the following third-party licenses:

- ScaLAPACK
- bsdiff
- JsonCpp
- MPICH2

C.1. Affero GPL

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

SciDB is distributed "AS-IS" AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. See the Affero General Public License at http://www.gnu.org/licenses/agpl-3.0.html for the complete license terms.

C.2. ScaLAPACK

C.3. bsdiff

```
Copyright 2003-2005 Colin Percival
All rights reserved
```

C.4. JsonCpp

C.5. MPICH2

Appendix D. 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)

Appendix E. Sample Macros

This appendix contains the listing for macros that are discussed in the User Guide.

```
/** Analyze your chunk sizes  **/
chunk_skew() =
    project(
     cross_join(

      redimension(
       apply(
        filter(
         list('chunk map'),
         inst=instn and attid=0),
        iid, int64(inst),
        aid, int64(arrid)),
       <nchunks   : uint64 null,
        min_ccnt  : uint32 null,
        avg_ccnt  : double null,
        max_ccnt  : uint32 null,
        total_cnt : uint64 null>
        [iid = 0:*,1000,0,aid= 0:*,1000,0],
       count(*)   as nchunks,
       min(nelem) as min_ccnt,
       avg(nelem) as avg_ccnt,
       max(nelem) as max_ccnt,
       sum(nelem) as total_cnt
      ) as A,

      redimension(
       apply( list('arrays', true), aid, int64(id)),
       <name: string null>
       [aid = 0:*,1000,0]
      ) as B,
      A.aid, B.aid
     ),
     name, nchunks, min_ccnt, avg_ccnt, max_ccnt, total_cnt
    );

/** Return the Euclidean distance between the two points (x1,y1) and (x2,y2). **/
distance(x1,y1,x2,y2) = sqrt(sq(x2-x1) + sq(y2-y1)) where
{
    sq(x) = x * x;   -- the square of the scalar "x"
};

/**
 *  Return the number of non empty cells in the array 'A'. A simple alias for
 *  the now deprecated count() aggregate.
*/
cnt(A) = aggregate(A,count(*));

/** Apply 'expression' to each element of the 'array'. **/
map(array,expression,name) = project(apply(array,name,expression),name);


/** Return the number of cells in which 'A.a' and 'B.b' differ. **/
difference(A,B,a,b) = cnt(nonzero(map(join(A,B),a - b,_diff)))
where
{
    cnt(A)       = aggregate(A,count(*));
    nonzero(A)   = filter(A, _diff <> 0);
    map(A,e,n)   = project(apply(A,n,e),n);
};

/** Centers the columns 'c' of the real valued 'a'  matrix 'M'. **/
```

```
center(M,a,c) = zero(map(cj(M,a,c),centered,value - mean)) where
{
    cj(M,a,c) = cross_join(cast(M,            <value:double null>[i,j]) as input,
                  cast(aggregate(M,avg(a),c), <mean :double null>[j])   as means,
                  input.j,
                  means.j);
    zero(A) = substitute(A,build(<a:double>[i=0:0,1,0],0));
};
```

Index