SciDB User's Guide

SciDB User's Guide

Version 12.2

Table of Contents

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

Paradigm4 Enterprise SciDB with Paradigm4 Extensions is an enterprise distribution of SciDB with additional linear algebra operations, 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 (timeseries) data, and matrix-based data for linear algebra operations.

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

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

1.1. Array Data Model

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

Dimensions

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

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

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

Attributes

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

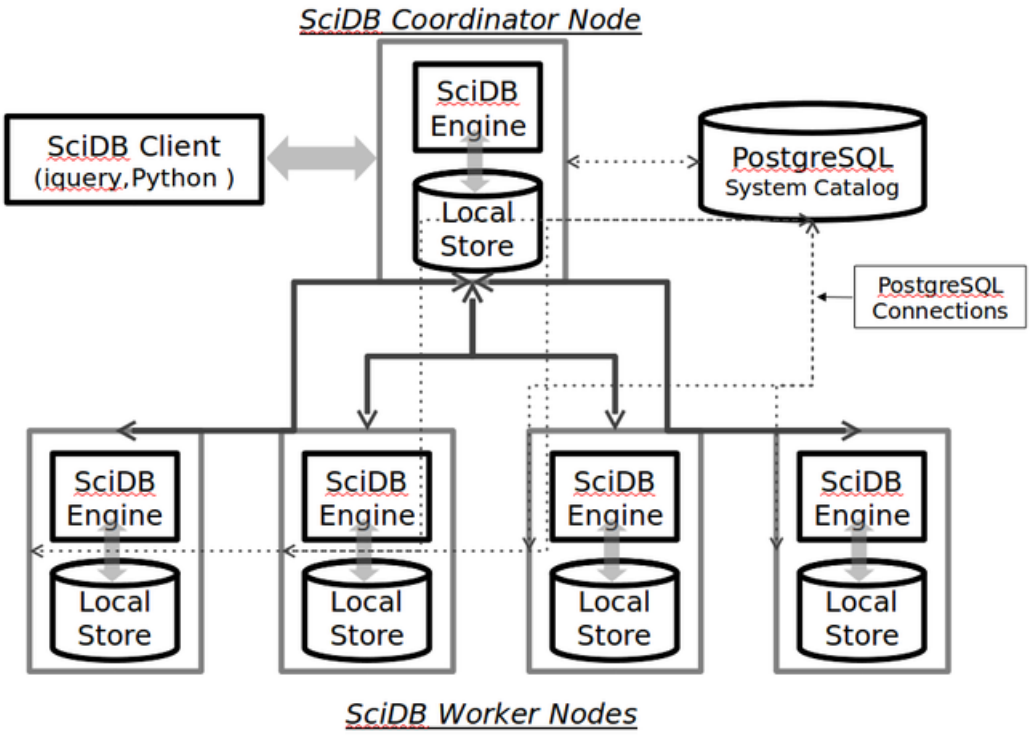
At array creation time, the user must specify:

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

Once you have created a SciDB database and defined the arrays, you must prepare and load data into it. Loaded data is then available to be accessed and queried using SciDB's built-in analytics capabilities.

1.2. Basic Architecture

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



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

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

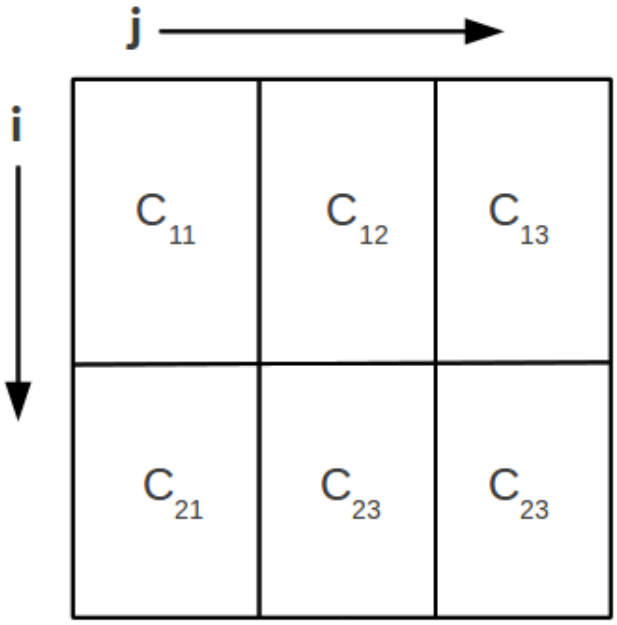
SciDB's scale-out architecture is ideally suited for hardware grids as well as clouds, where additional severs may be added to scale the total capacity.

1.2.1. Chunking and Scalability

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

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

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



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

C_{11} -> server 1

C_{12} -> server 2

C_{13} -> server 3

C_{21} -> server 4

C_{22} -> server 1

C_{23} -> server 2

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

1.2.2. Chunk Overlap

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

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

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

Some advantages of chunk overlap are:

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

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

1.3. SciDB storage management

1.3.1. Local Storage

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

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

1.3.2. Versions and Delta Encoding

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

1.3.3. Automatic Storage Allocation and Reclamation

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

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

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

1.4. Database Concepts

1.4.1. ACID and Transactions

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

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

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

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

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

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

Durability. When a write or remove transaction completes successfully, its results are committed to stable storage. A transient power failure or reboot of the server does not result in loss of committed data.

1.5. System Catalog

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

- Array-related metadata such as array definitions, array versions, and associations between arrays and other related objects,
- Configuration information about the SciDB cluster,
- SciDB extensions, such as plug-in libraries.

1.6. Program Building Blocks

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

Operators	SciDB operators take a SciDB array as input and return a SciDB array as output.
Functions	SciDB functions take a SciDB array attribute or attributes and return a scalar value.
Data types	Data types define the data storage format of an array's attributes and dimensions.
Aggregates	SciDB aggregates take a SciDB array attribute or attributes as input and return a scalar value.

Any of these building blocks can be user-defined. See the chapter on SciDB Plugins for how to create a user-defined datatype (UDT), user-defined operator (UDO), user-defined function (UDF), or user-defined aggregate (UDA).

1.7. Query 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 m 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.

1.8. Clients and Connectors

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

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

1.9. 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 [].

Chapter 2. SciDB Installation and Administration

2.1. Installing SciDB

SciDB binaries are currently available for the following Linux platforms:

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

For virtual machine–based installs, you can use VMWare Player or VBox for desktop testing and Citrix XenServer for production use.

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

Instance	An independent SciDB process, that is, a single runnable copy of SciDB. There may be a many-to-one mapping between SciDB instances and a single server.
Cluster	A group of one or more single servers connected by TCP/IP, working together as a single system, and sharing data. A cluster can be a private grid or a public cloud.
Single server	A configuration that consists of a single machine with a processor that may contain any number of cores, memory and attached storage. A single server may be virtual or physical. A single server is not connected to nor does it share data with any other servers in a cluster.
Virtual server	A server that shares hardware rather than having dedicated hardware.

2.1.1. Preparing the Platform

2.1.1.1. Linux User Account

First, you will need to create a Linux user account, **scidb**. This account will be used to run all SciDB processes and own all files created by SciDB. The **scidb** user account must have superuser privileges. It is also helpful to set up the account for access to the system without password entry.

To create the account, modify the `/etc/sudoers` file as follows:

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

2.1.1.2. Postgres Installation and Configuration

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

On Ubuntu, you can use apt-get to install the postgresql-contrib package:

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

On Red Hat and Fedora, you can use `yum` :

```
sudo yum install postgresql-contrib
```

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

1. Modify the `pg_hba.conf` file (usually at `/etc/postgresql/8.4/main/` or `/var/lib/pgsql/data/`) by adding the following line:

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

2. In the `pg_hba.conf` file, change all instances of 'ident' to 'trust' (assuming your local network is 10.x.x.x).

3. Restart Postgres.

Warning

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

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

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

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

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

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

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

Note

- Red Hat Enterprise Linux 5.4 comes with PostgreSQL 8.1. We recommend upgrading to version 8.4.7.

- Add Postgres startup scripts to the Linux initialization scripts to start Postgres automatically after a reboot.
- If your **scidb** user does *not* have *sudo* privileges, have your administrator use the following procedure to intialize Postgres:

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

     ```
     root$ sudo -u postgres
           /opt/scidb/12.3/bin/scidb-prepare-db.sh
     ```

2.1.1.3. Remote Execution Configuration (ssh)

SciDB uses ssh for remote execution of cluster management commands. This is why the **scidb** user account should have no-password ssh access from the coordinator to the workers and from the coordinator to itself.

The python-crypto (64-bit) and python-paramiko packages are required for SciDB on Red Hat 5.4. These packages are ssh packages in python. You can install the python ssh client packages as follows:

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

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

1. Create key:

   ```
   ssh-keygen
   ```

2. Copy the key to the localhost (or coordinator) and to each worker:

   ```
   ssh-copy-id scidb@worker
   ssh-copy-id scidb@localhost
   ```

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

   ```
   ssh scidb@worker
   ```

2.1.1.4. Shared file system

To run SciDB in a cluster , export the `/opt/scidb` directory on the coordinator using NFS or samba. To do this, configure the export and restart the nfs service like this:

```
# Configure the export
/opt/scidb *(ro,no_root_squash,sync)

# Restart the nfs service
sudo /etc/init.d/nfs restart
```

Mount this on all workers using the same directory path (`/opt/scidb`) as the mount point. Add this line to the `/etc/fstab` file to mount the shared file system on each worker:

```
# SciDB coordinator mount point
```

```
10.0.0.1:/opt   /opt   nfs    ro,rsize=8192,wsize=8192,timeo=14,intr
```

The coordinators and workers access binaries, shared libraries, plugins, configuration files from `/opt/scidb`.

2.1.2. Install SciDB from binary package

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

2.1.2.1. Ubuntu

Install

1. Install the libscidbclient package:

   ```
   sudo dpkg  -i libscidbclient.*.deb
   ```

   You may want to install the optional debug symbols package:

   ```
   sudo dpkg -i libscidbclient.*.deb
   ```

2. Install the SciDB package:

   ```
   sudo dpkg -i scidb.*.deb
   ```

   You may want to install the optional debug symbols package:

   ```
   sudo dpkg -i scidb-dbg.*.deb
   ```

Note

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

```
# Fails due to unmet dependencies
sudo dpkg -i scidb.*.deb

# Installs dependencies
sudo apt-get -f install

# Succeeds now
sudo dpkg -i scidb-RelWithDebInfo-12.3.*.deb
```

Uninstall

Uninstall SciDB as follows:

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

2.1.2.2. Red Hat and Fedora

Install:

1. Install the libscidbclient package:

   ```
   sudo rpm --force -ivh libscidbclient-RelWithDebInfo-12.3.*.rpm
   ```

   You may want to install the optional debug symbols package:

   ```
   sudo rpm --force -ivh
   libscidbclient-dbg.*.rpm
   ```

2. Next, install the SciDB server package:

   ```
   sudo rpm --force -ivh scidb-12.3.*.rpm
   ```

   You may want to install the optional debug symbols package:

   ```
   sudo rpm --force -ivh scidb-dbg.*.rpm
   ```

Uninstall:

To uninstall SciDB, do the following:

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

2.1.2.3. Environment Variables

Now you need to configure the environment of the **scidb** user account. The following lines should be added to the user's shell configuration file (often `.profile` or `.bashrc`):

```
export SCIDB_VER=12.3
export PATH=/opt/scidb/$SCIDB_VER/bin:/opt/scidb/$SCIDB_VER/share/scidb:$PATH
export LD_LIBRARY_PATH=/opt/scidb/$SCIDB_VER/lib:$LD_LIBRARY_PATH
```

2.2. Configuring SciDB

This chapter demonstrates how to configure SciDB prior to initialization, including checking that the PostgreSQL DBMS is running, that the SciDB configuration file (usually `/opt/scidb/12.3/etc/config.ini`) is set up, and that logging is configured.

2.2.1. SciDB Configuration File

You need to create a configuration file for SciDB. By default, it is named `config.ini` and it resides in the `etc` sub-directory of the installation tree. (By default it is `/opt/scidb/12.3/etc/config.ini`.) The configuration file can have multiple sections, one per service instance.

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

```
[test1]
instance-0=localhost,0
db_user=test1user
db_passwd=test1passwd
install_root=/opt/scidb/12.3
metadata=/opt/scidb/12.3/share/scidb/meta.sql
pluginsdir=/opt/scidb/12.3/lib/scidb/plugins
logconf=/opt/scidb/12.3/share/scidb/log4cxx.properties
base-path=/home/scidb/data
base-port=1239
interface=eth0
no-watchdog=true
redundancy=1
merge-sort-buffer=1024
network-buffer=1024
mem-array-threshold=1024
smgr-cache-size=1024
execution-threads=16
result-prefetch-queue-size=4
result-prefetch-threads=4
chunk-segment-size=10485760
```

2.2.2. Cluster Configuration Example

The following oSciDB cluster configuration is called 'monolith'. This cluster consists of eight identical virtual servers:

- x86 6-core processor
- 8 GB of RAM
- 1 TB direct attached storage
- 1Gbps Ethernet
- RHEL 5.4

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

```
[monolith]
# server-id=IP, number of worker instances
server-0=10.0.20.231,0
server-1=10.0.20.232,1
server-2=10.0.20.233,1
server-3=10.0.20.234,1
server-4=10.0.20.235,1
server-5=10.0.20.236,1
server-6=10.0.20.237,1
server-7=10.0.20.238,1
db_user=monolith
db_password=monolith
install_root=/opt/scidb/12.3
metadata=/opt/scidb/12.3/share/scidb/meta.sql
pluginsdir=/opt/scidb/12.3/lib/scidb/plugins
```

```
logconf=/opt/scidb/log4cxx.properties.trace
base-path=/data/monolith_data
base-port=1239
interface=eth0
```

The install package contains a sample configuration file, `sample_config.ini`, with examples.

The following table describes the basic configuration file settings:

Basic Configuration	
Key	**Value**
Cluster name	Name of the SciDB cluster. The cluster name must appear as a section heading in the config.ini file, e.g., *[cluster1]*
server-N	The host name or IP address used by server N and the number of worker instances on it. Server 0 always has the coordinator running as instance 0, and may have additional worker instances running as well.
db_user	Username to use in the catalog connection string. This example uses *test1user*
db_passwd	Password to use in the catalog connection string. This example uses *test1passwd*
install_root	Path name of install root.
metadata	Metadata definition file.
pluginsdir	The folder or directory in which plugins are stored.
logconf	**log4xx** configuration file.

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

Cluster Configuration	
Key	**Value**
base-path	The root data directory for each SciDB instance. Each SciDB instance initializes its data directory within the base-path. Path `scidb/00`*n*`/1` will be the path for instance *n*.
base-port	Base port number. Connections to the coordinator (and therefore to the system) are via this number, while worker instances communicate at base-port + instance number. The default number that `iquery` expects is 1239.
interface	Ethernet interface that SciDB must use.
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: None.
tmp-path (optional)	The directory to use as temporary space.
no-watchdog (optional)	Set this to true if you do not want automatic restart of the SciDB server on a software crash. Default: false.

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

Performance Configuration	
Key	**Value**

save-ram (optional)	'True', 'true', 'on' or 'On' will enable this option. Off by default. This allows you to store temporary data in memory. It is not advisable to do this; it is better to store temporary data in files.
merge-sort-buffer (optional)	Size of memory buffer used in merge sort. Default: 512 MB.
mem-array-threshold (optional)	Maximum memory used for temporary arrays. Default: 1024 MB.
chunk-reserve (optional)	Percentage of chunk preallocated to store chunk deltas. Setting this parameter to 0 disables the delta mechanism. Default: 10%.
chunk-segment-size (optional)	Size in bytes of a storage segment. A storage segment is a unit of allocation and reclamation used by storage manager. If set to zero, no space reuse or storage reclamation is done.
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.
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.
smgr-cache-size (optional)	Size of buffer cache. Default: 256 MB

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

2.2.3. Logging Configuration

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

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

```
###
# Levels: TRACE < DEBUG < INFO < WARN < ERROR < FATAL
###

log4j.rootLogger=DEBUG, file

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

2.3. Initializing and Starting SciDB

2.3.1. The scidb.py Script

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

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

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

```
scidb.py command db conffile
```

The options for the *command* argument are:

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

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

The configuration file is set by default to `/opt/scidb/12.3/etc/config.ini`. If you want to use a custom configuration file for a particular SciDB cluster, use the *conffile* argument.

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

```
scidb.py initall test1
```

Warning

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

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

```
scidb.py startall test1
```

This will report the status of the various instances:

```
scidb.py status test1
```

This will stop all SciDB instances:

```
scidb.py stopall test1
```

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

2.4. Upgrading SciDB

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

- Shutdown SciDB:

  ```
  scidb.py stopall test1
  ```

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

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

2.4.1. Ubuntu

1. First, upgrade the libscidbclient package :

   ```
   sudo dpkg -i libscidbclient.*.deb
   ```

   You may want to install the optional debug symbols:

   ```
   sudo dpkg -i libscidbclient-dbg.*.deb
   ```

2. Then install the SciDB package:

   ```
   sudo dpkg -i scidb-RelWithDebInfo-12.3.deb
   ```

   You may want to install the optional debug symbols:

   ```
   sudo dpkg -i scidb-dbg-RelWithDebInfo-12.3.deb
   ```

2.4.2. Red Hat and Fedora

1. First, you need to install the libscidbclient package:

   ```
   sudo rpm --force -Uvh libscidbclient-RelWithDebInfo-12.3.*.rpm
   ```

   If you prefer, you can install with debug symbols:

   ```
   sudo rpm --force -Uvh libscidbclient-dbg-RelWithDebInfo-12.3.*.rpm
   ```

2. Next, install the SciDB server package:

   ```
   sudo rpm --force -Uvh scidb-12.3.*.rpm
   ```

   If you prefer, you can install debug symbols:

   ```
   sudo rpm --force -Uvh scidb-dbg-.*.rpm
   ```

3. Copy over the previous `config.ini` from your earlier version:

   ```
   cp /opt/scidb/12.3/etc/config.ini /opt/scidb/11.12/etc/config.ini
   ```

2.4.3. Additional Steps

- Modify the config.ini file that you just copied. Change all references to your previous version to the new version (ex: `install_root=/opt/scidb/12.3`)

- Edit your environment and update PATH and LD_LIBRARY_PATH:

  ```
  export SCIDB_VER=12.3
  export PATH=/opt/scidb/$SCIDB_VER/bin:/opt/scidb/$SCIDB_VER/share/scidb:$PA\
  TH
  ```

```
export LD_LIBRARY_PATH=/opt/scidb/$SCIDB_VER/lib:$LD_LIBRARY_PATH
```

- **NOTE:** SciDB 12.3 does not accept storage files from earlier versions. You must reinitialize and reload data:

```
which scidb.py # Make sure you are running 12.3
scidb.py initall test1
scidb.py startall test1
scidb.py status test
```

Chapter 3. Getting Started with SciDB Development

3.1. Using the iquery Client

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

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

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

```
AQL%
```

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

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

AQL statements end with a semicolon (;).

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

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

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

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

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

```
iquery -f my_input_filename
```

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

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

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

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

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

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

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

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

Available options for output format are csv, csv+, lcsv+, sparse, and lsparse. These options are descirbed in the following table:

Output Option	**Description**
auto (default)	SciDB array format
csv	Comma-separated values
csv+	Comma-separated values with dimension indices
lcsv+	Comma-separated values with dimension indices and a boolean flag attribute EmptyTag showing if a cell is empty
sparse	Sparse SciDB array format
lsparse	Sparse SciDB array format and a boolean flag attribute EmptyTag showing if a cell is empty

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

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

The `iquery` interface is case sensitive.

3.2. iquery Configuration

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

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

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

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

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

3.3. Example iquery session

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

1. Create an array called random_numbers with:
   - 2 dimensions, x = 9 and y = 10
   - One double attribute called `num`
   - Random numerical values in each cell

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

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

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

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

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

```
iquery -aq "create array target <name: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:*

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

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

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

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

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

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

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, '/tmp/datafile.scidb')"
```

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

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

```
 [("CRV",23.5,9.98985),("Prius",48.7,20.7024),
 ("Explorer",19.6,8.33196),("Q5",26.8,11.3927)]
```

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

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

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

```
Make,Model,Price
Honda,CRV,26700
```

```
Toyota,Prius,31000
Ford, Explorer,42000
Audi,Q5,45000
```

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

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

Create an array called storage:

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

Load the datafile_price.scidb file into storage:

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

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)"
[("Honda","CRV",26700,"CRV",23.5,9.98985),
("Toyota","Prius",31000,"Prius",48.7,20.7024),
("Ford"," Explorer",42000,"Explorer",19.6,8.33196),
("Audi","Q5",45000,"Q5",26.8,11.3927)]
```

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

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

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

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

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

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

Chapter 4. Creating and Removing SciDB Arrays

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

A SciDB database is organized into arrays that have:

- A *name*. Each array in a SciDB database has an identifier that distinguishes it from all other arrays in the same database.
- A *schema*, which is the array structure. The schema contains array *attributes* and *dimensions*.
  1. Each *attribute* contains data being stored in the array's cells. A cell can contain multiple attributes.
  2. Each *dimension* consists of a list of index values. At the most basic level the dimension of an array is represented using 64-bit unsigned integers. The number of index values in a dimension is referred to as the dimension's *size*.

4.1. Create an Array

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

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

The arguments for the CREATE ARRAY statements are as follows:

`array_name`
: The array name that uniquely identifies the array in the database. The maximum length of an array name is 1024 bytes. Array names may not contain the characters `@` or `:` as these characters are reserved for special array types generated by other SciDB operators.

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

  - *Attribute name*: Name of an attribute. The maximum length of an attribute name is 1024 bytes. No two attributes in the same array can share a name.
  - *Attribute type*: Type identifier. One of the data types supported by SciDB. Use the `list('types')` command to see the list of available data types.
  - NULL (optional): If unspecified, all attributes are 'NOT NULL', i.e. they must have a value. Optionally, users can specify 'NULL' to indicate attributes that are allowed to contain null values.
  - DEFAULT (optional): Specify the value to use when an attribute is NULL or absent.

dimensions

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

- *Dimension name*: Each dimension has a name. Just like attributes, each dimension must be named, and dimension names cannot be repeated in the same array. The maximum length of a dimension name is 1024 bytes.
- *Dimension start*: The starting coordinate of a dimension. The default data type is 64-bit integer.
- *Dimension end* or *: The ending coordinate of a dimension, or * if unbounded. The default data type is 64-bit integer for bounded dimensions.
- *Dimension chunk size*: Number of elements per chunk.
- *Dimension chunk overlap*: Number of overlapping cells from a neighboring chunk.

The AQL `CREATE ARRAY` statement creates an array with specified name and schema.

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

This statement creates an array with:

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

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

```
DROP ARRAY A;
```

To create an array with AFL, use the `create_array` operator:

```
AFL% create_array(A,<val:double,err:double>[i=0:99,10,0,j=0:99,10,0]);
```

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

```
AFL% remove(A);
```

4.2. Array Attributes

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

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

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

Attributes are by default set to NOT NULL:

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

```
[("products<name:string NOT NULL,
price:float NOT NULL,sold:datetime NOT NULL> [i=0:*,10,0]")]
```

To allow an attribute to have value NULL, add NULL to the attribute data type declaration:

```
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:

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

4.3. Array Dimensions

A SciDB array must have at least one dimension. Dimensions form the coordinate system for a SciDB array.

Note

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

4.3.1. Chunk Overlap

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

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

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

Some advantages of chunk overlap are:

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

4.3.2. Unbounded Dimensions

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

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

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

4.3.3. Noninteger Dimensions and Mapping Arrays

Basic arrays in SciDB use the int64 data type for dimensions. SciDB also supports arrays with noninteger dimensions. These arrays map dimension *values* of a declared type to an internal int64-array *position*. Mapping is done through special mapping arrays internal to SciDB. Such arrays are useful when you are transforming data into multidimensional format where some dimensions represent factors or categories.

For example, the array D has a noninteger dimension named ID:

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

```
[("D <val:int64 NOT NULL,empty_indicator:indicator NOT NULL>
[ID(string)=10,5,0]")]
```

The dimension indices of ID are:

```
AQL% SELECT * FROM D:ID;
```

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

The values of the attribute `val` of `D` are:

```
AQL% SELECT * FROM D;
```

```
[(0),(90),(2),(6),(12),(20),(30),(42),(56),(72)]
```

Note

In the current version of SciDB, it is not possible to load data directly from an external file into a mapping array.

4.4. Changing Array Names

An array name is used to identify an array in the current SciDB namespace. You can use the AQL `SELECT ... INTO` statement to rename an array.

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

This means that both A and new_A are in the current SciDB namespace. To change an array name and remove the old array name from the current SciDB namespace, use the rename command:

```
AFL% rename(new_A, A_backup);
```

You can use the cast command to change the name of the array, array attributes, and array dimensions. 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 arrays should have the same numbers and types of attributes and the same numbers and types of dimensions.

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

[("A<x:double NOT NULL,err:double NOT NULL> [i=0:99,10,0,j=0:99,10,0]")]
```

This query creates an array new_A with attributes val1 and val2 and dimensions x and y:

```
AQL% SELECT * INTO new_A
FROM cast(A,<val1:double,val2:double>
[x=0:99,10,0,y=0:99,10,0]")];
```

4.5. Database Design

4.5.1. Selecting Dimensions and Attributes

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

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

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

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

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

4.5.2. Chunk Size Selection

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

To optimize performance of your SciDB array, you want chunks to contain on order of 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 multiattribute 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.

4.5.3. Balancing Data Distribution

When the density of the data in a data set is highly skew, 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 the data skewness isn't known at array creation time. In this case, you may want to use SciDB's *repartitioning* functionality to change the chunk size as necessary.

Chapter 5. Loading Data:

The key part of setting up your SciDB array is loading data. This chapter explains how to use the SciDB LOAD statement and how to prepare your data for loading.

5.1. The LOAD Statement

The AQL LOAD statement loads formatted data into an existing SciDB array. The syntax of the LOAD statement is:

```
LOAD array_name FROM load_file_path ;
```

The *array_name* argument is the SciDB array into which the data will be loaded. The *load_file_path* argument is the file path of the formatted data to be loaded.

Your data file must be formatted so that the chunks are loaded in the correct order. For example, if you have a 16-by-16 array divided into ordered 4-by-4 chunks, you will need to order the data in the load file appropriately. Chunks in the SciDB array are ordered as follows:

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

So the chunks in the load file must appear in the following order:

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

5.2. Preparing a 1-Dimensional Load File

This section describes how to load data from a flat, csv-formatted file into a SciDB array. Consider the 20-line, csv-formatted file `num_data.csv`, the first few lines of which are shown here:

```
val,err
1.48306e+09,1
5.80814e+08,1
1.51079e+09,1
1.16154e+09,1
1.42655e+09,1
1.06341e+09,1
```

To prepare this file for loading into a SciDB array, use the command `csv2scidb`. The `csv2scidb` command takes multicolumn csv data and transforms it into 1-dimensional arrays with one attribute for every comma-delimited column. 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:

```
-v version of tool
-i PATH input file
-o PATH output file
-a PATH appended output file
-c INT length of chunk
-f INT starting chunk number
-d char delimeter - default ,
-p STR type pattern - N number, S string, s nullable-string, C char
-s N skip N lines at the beginning of the file
```

This code will transform `num_data.csv` to SciDB load file format:

```
csv2scidb -s 1 -c 10 -p N < num_data.csv > num_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, and provide that information as attribute names. The -c flag specifies a chunk size of 10. The -p flag specifies the type of data you are loading. Possible values are N (number), S (string), s (nullable string), and C (char).

The file `num_data.scidb` looks like this:

```
{0}[
(1.48306e+09,1),
(5.80814e+08,1),
(1.51079e+09,1),
(1.16154e+09,1),
(1.42655e+09,1),
(1.06341e+09,1),
(4.9253e+08,1),
(5.6065e+08,1),
(1.60886e+08,2),
(1.37844e+09,1)
];
{10}[
(4.08495e+08,1),
(5.65393e+07,1),
(1.47646e+09,1),
(9.52609e+08,1),
(1.8548e+09,1),
(1.42396e+09,1),
(1.75107e+09,1),
(1.52007e+09,1),
(5.4882e+08,1),
(7.28928e+08,1)
];
```

The {0} and {10} are the chunk number for the succeeding data. These are called *chunk headers*. The square braces show the beginning and end of the array chunk. The parentheses show the cells of the array. There are commas between attributes in cells and cells in the chunk.

To create an array for this data, create an array with 1-dimension. The chunk size and overlap is not terribly important for an array of this size and dimensionality. Here, the chunk size is half of the dimension size and the chunk overlap 0. The original data set had two column headers of val and err, so you can name the attributes `val` and `err`:

```
CREATE ARRAY num_data <val:double,err:double>[i=0:19,20,0];
```

To load the data into the array `num_data`, use a `LOAD` statement:

```
LOAD num_data FROM '/tmp/num_data.scidb';
```

5.3. Data with Special 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 '/tmp/v4.scidb';
```

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

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 '/tmp/v4.scidb';
```

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

Load files may also contain null values.

```
[
 (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 '/tmp/v4_null.scidb';
```

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

5.4. Parallel Load

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

id Value	Description
0	Coordinator
1,2, ..., N	*id* of the instance that should perform the load where N is the number of instances in the cluster.
–1	All instances in the cluster. The file path is assumed to be the same at all instances. If an instance cannot open the data file, the load will continue after logging a warning.

For parallel load, each instance must be given distinct chunks. To do this, chunks in the load file must be prefixed with a distinct chunk header that lists the starting dimension values of the chunk.

If your SciDB cluster has 4 instances (with identifiers 1, 2, 3, and 4) and there are 20 chunks, you can place chunks 1–5 on instance 1, chunks 6–10 on instance 2, and so on. The following load command will simultaneously load all 20 chunks into the array and complete 4 times faster.

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

5.5. Data Generation

SciDB provides functions for data generation. The simplest way to add data to an array with one attribute is the `build` operator and a SciDB expression.

For example, to store numerical values of 1 to 10 in an array with one attribute and one dimension of size 10, you can do the following:

```
AQL% CREATE ARRAY consecutive_integers
<val:int64>
[x=0:9,10,0];
AQL% SELECT * INTO consecutive_integers FROM
build(consecutive_integers, x+1);
```

This code takes the index values *x* from array consecutive_integers and uses them in the expression *x* + 1. The `build` command then executes the expression. This puts the following numbers into the attribute `val`:

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

You can create additional array attributes and additional data with the `apply` function. For example, this statement generates an additional attribute name `val2` with value `val`+100:

```
AQL% SELECT * FROM apply(consecutive_integers,val2,val+100);
```

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

You can also create new arrays with the generated data by using the **INTO** clause. For example, this statement creates an array called `random_numbers` with an one attribute copied from `consecutive_numbers` attribute `val` and one attribute `val_rand` containing random numbers:

```
AQL% SELECT * INTO random_numbers  FROM
apply(consecutive_integers,val_rand,val*random());
```

```
[(1,1687457050),(2,486200554),(3,341466474),
(4,2037600252),(5,5420696925),(6,1547216142),
(7,12796206535),(8,136216624),(9,16958580471),
(10,10628809700)]
```

If you want to build an array on data you have already generated, you can use the `xgrid` command to expand an existing array. The `xgrid` command expands an array by repeating elements from that array. For example, suppose you have a 3-by-3 array:

```
AFL% show(m3x3);scan(m3x3);
[("m3x3<val:double NOT NULL> [x=0:2,3,0,y=0:2,3,0]")]
[
```

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

You can make individual elements of this array into *n*-by-*n* grids. To make each cell into a 2-by-2 grid, use the following syntax:

```
AFL% xgrid(m3x3,2,2);
[
[(1),(1),(2),(2),(3),(3)],
[(1),(1),(2),(2),(3),(3)],
[(4),(4),(5),(5),(6),(6)],
[(4),(4),(5),(5),(6),(6)],
[(7),(7),(8),(8),(9),(9)],
[(7),(7),(8),(8),(9),(9)]
]
```

5.6. Saving Data from a SciDB Array to a File

You can save all or part of the data that is contained in a SciDB array to a file. You can use a **SELECT** statement with the `save` command to save an entire array:

```
AQL% SELECT * FROM
save(random_numbers,'/tmp/random_number_data.txt');
```

This statement saves a SciDB-formatted file called `random_number_data.txt`.

To save the data to `csv` format, set the `iquery` output option to `csv`:

```
% iquery -o csv -q "SELECT * FROM save(random_numbers,'/tmp/random_number_data.csv
```

```
val,val_rand
1,939618095
2,1011655774
3,3620619210
4,2317057332
5,6137260845
6,10771327980
7,4496569336
8,10364290328
9,2309513805
10,1398261690
```

Note

You will need to enter your iquery statement directly at the command line to change the output option to csv. Type `exit;` at the `AQL%` prompt to stop the current `iquery` session.

5.7. Redimensioning an Array

A common use case for creating and loading SciDB arrays is using data from a data warehouse. This data set may be very large and formatted as a csv file. You can use the csv2scidb utility to convert a csv file

to the 1-dimensional array format and load the file into a SciDB array. Once you have a 1-dimensional SciDB array, you can redimension the array to convert the attributes to dimensions.

For example, suppose you have a csv file like this:

```
s,p,val
"sample-0","probe-0",0.01
"sample-1","probe-0",2.04
"sample-2","probe-0",6.09
"sample-3","probe-0",12.16
"sample-4","probe-0",20.25
"sample-0","probe-1",30.36
"sample-1","probe-1",42.49
"sample-2","probe-1",56.64
"sample-3","probe-1",72.81
"sample-4","probe-1",91
"sample-0","probe-2",111.21
"sample-1","probe-2",133.44
"sample-2","probe-2",157.69
"sample-3","probe-2",183.96
"sample-4","probe-2",212.25
"sample-0","probe-3",242.56
"sample-1","probe-3",274.89
"sample-2","probe-3",309.24
"sample-3","probe-3",345.61
"sample-4","probe-3",384
"sample-0","probe-4",424.41
"sample-1","probe-4",466.84
"sample-2","probe-4",511.29
"sample-3","probe-4",557.76
"sample-4","probe-4",606.25
```

This data has three columns, two of which are stings and one which is a floating-point number. The column headers are 's','p',and 'val'. To load this data set, create a 1-dimensional SciDB array with three attributes and load the data into it. For this example, the array is named expo. The dimension name is i, the dimension size is 25, the chunk size is 5. The attributes are s, of type string, p of type string, and val of type double.

```
SELECT * FROM show(expo);
```

```
[("expo<s:string NOT NULL,p:string NOT NULL,
val:double NOT NULL> [i=1:25,5,0]")]
```

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

	probe-0	probe-1	probe-2	probe-3	probe-4
sample-0	0.01	30.36	111.21	242.56	424.41
sample-1	2.04	42.49	133.44	274.89	466.84
sample-2	6.09	56.64	157.69	309.24	511.29
sample-3	12.16	72.81	183.96	345.61	557.76
sample-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 empty array Dsp
<val:double>
[s(string)=5,5,0, p(string)=5,5,0];
```

Each of the dimensions is of size 5, corresponding to a dimension in the 5-by-5 table. Now, you can use the redimension_store operator to redimension the array expo into the array Dsp:

```
AFL% redimension_store(expo, Dsp);
```

```
[
[(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 sample 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 sample from the third probe, use the dimension indices:

```
AQL% SELECT val FROM Dsp  WHERE s='sample-2' and p='probe-3';
```

Redimensioning is a powerful tool when you want to do array aggregation along the coordinate axes of a data set. For example, you can find the average value of a sample for each probe. This would be equivalent to finding the average of every row in the table:

```
create empty array Ds
<av:double NULL>[s(string)=5,5,0];
redimension_store(expo, Ds, true, avg(val) as av);
```

Or, you can find the average value of all the samples for a single probe. This would be equivalent to finding the average of every column in the table:

```
AFL% create empty array Dp
<av:double NULL>[p(string)=5,5,0];
AFL% redimension_store(expo, Dp, true, avg(val) as av);
```

Chapter 6. Basic Array Tasks

6.1. Selecting Data From an Array

AQL's 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. The syntax of the `SELECT` statement is:

```
SELECT list | *
[INTO target_array]
FROM array_expression | source_array
[WHERE operator]
```

The arguments for the statement are:

list | *	`SELECT` *list* 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
array_expression | *source_array*	The `FROM` clause takes a SciDB array as argument. The *array_expression* argument is an expression or subquery that returns an array result. The *source_array* is an array in the current SciDB namespace from which data is being selected.
operator	The expression argument of the `WHERE` clause allows to you specify parameter that filter the query.

6.1.1. 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 result. 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 in the array, you can use a `SELECT *` statement with the `scan` command. The `scan(A)` command returns a SciDB array result containing the values of the array data in A. By using `scan(A)` with a `SELECT *` statement, the query will return the entire array result of `scan(A)`:

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

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

AQL% SELECT * FROM scan(B);

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

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

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

[("A<val_a:double NOT NULL> [i=0:9,10,0]")]

AQL% SELECT * FROM show(B);

[("B<val_b:double NOT NULL> [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:

```
SELECT j FROM B;
SELECT val_b FROM B;
```

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

```
AQL% SELECT val_b/10 FROM B;

[(10.1),(10.2),(10.3),(10.4),(10.5),
(10.6),(10.7),(10.8),(10.9),(11)]
```

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:

```
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:

```
SELECT sqrt(val_b) FROM B WHERE j>3 and j<7;

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

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

6.2. Array Joins

A join combines two or more arrays typically as a preprocessing step for subsequent operations. The simplest type of join is for two arrays with the same number of dimensions, same dimension starting and ending coordinates, and same chunk size.

The syntax of a simple join statement is:

```
SELECT expressionINTO target_arrayFROM src_array
```

The natural join of these arrays joins the attributes:

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

This join produces:

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

You can store the output using the `INTO` clause. For example, this code will store the attribute-attribute join of A and B in array C:

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

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

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

This produces array D with the following schema:

```
[("D<val_a:double NOT NULL,
val_b:double NOT NULL,
val_b_2:double NOT NULL>
[i=0:9,10,0]")]
```

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;
```

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

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

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

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

```
[("m3x3<a:double NOT NULL> [i=1:3,3,0,j=1:3,3,0]")]
[[(4),(5),(6)],[(7),(8),(9)],[(10),(11),(12)]]
```

And a 1-dimensional array `vector3`schema and attribute values:

```
[("vector3<b:double NOT NULL> [k=1:3,3,0]")]
[(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;
```

```
[[(4,21),(5,20.5),(6,20.3333)],
[(7,21),(8,20.5),(9,20.3333)],
[(10,21),(11,20.5),(12,20.3333)]]
```

6.3. Aliases

AQL provides a way to refer to arrays and array attributes in a query via aliases. These are useful when using the same array repeatedly in an AQL statement, or when abbreviating a long array name. Aliases are created by adding an "as" to the array or attribute name, followed by the alias. Future references to the array can then use the alias. Once an alias has been assigned, all attributes and dimensions of the array can use the fully qualified name using the dotted naming convention.

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

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

6.4. Nested Subqueries

You can nest AQL queries to refine query results.

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

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.

```
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)]
```

6.5. Data Sampling

SciDB provides operations to sample array data. The `bernoulli` command 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. Seeds must be a 64-bit integer.

Chapter 7. Aggregates

SciDB supports commands to group data from an array and calculate summaries over those groups. These commands are called *aggregates*. SciDB provides the following types of aggregates based on how data is grouped:

- *Grand aggregates* compute aggregates over entire arrays.
- *Group-by aggregates* compute summaries by grouping array data by dimension values.
- *Grid aggregates* compute summaries for nonoverlapping subarrays.
- *Window aggregates* compute summaries over a moving window in an array.

This chapter uses example arrays `m4x4` and `m4x4_2attr`, which have the following schemas and contain the following values:

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

```
[("m4x4<attr1:double NOT NULL> [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)]
]
```

```
AFL%  show(m4x4_2attr);
```

```
[("m4x4_2attr<attr1:double NOT NULL,attr2:double NOT NULL>
[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)]
]
```

SciDB offers the following built-in aggregates.

Aggregate Function	Definition
`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

7.1. Grand Aggregates

Grand aggregates in SciDB calculate aggregates or summaries of attributes across an entire array. The syntax of the **SELECT** statement with built-in summary functions is:

```
SELECT function(attribute),function(attribute),...
INTO dst-array
FROM src-array | array-expression
WHERE where-expression
```

The output is a SciDB array with one attribute named for the summary type in the query and array dimensions 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 an **INTO** clause:

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

The destination array `m4x4_max_min` has schema:

```
[("m4x4_max_min<max:double NULL,min_1:double NULL> [i=0:0,1,0]")]
```

To select the maximum value from the attribute val of `m4x4_2attr` and the minimum value from the attribute `val2` of `m4x4_2attr`:

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

[(30,0)]
```

Note

In the special case of a one-attribute array, you can 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);

[(15)]
```

The AFL `aggregate` operator also computes grand aggregates. To select the maximum value from the attribute val of `m4x4_2attr` and the minimum value from the attribute `val2` of `m4x4_2attr`:

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

```
[(30,0)]
```

SciDB functions exclude null-valued data. For example, consider the following array `m4x4_null`:

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

The syntaxes `count(attr1)` and `count(*)` return different results:

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

```
[(4,16)]
```

One syntax, `count(attr1)`, shows only cells that have values that are not NULL. The other syntax, `count(*)`, shows cells whose values are NULL and cells whose values are not NULL.

7.2. Group-By Aggregates

Group-by aggregates allow you to 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 with same dimensions as the source array (without the group-by dimensions) and a single attribute per aggregate requested. The syntax of the SELECT statement for a group-by aggregate is:

```
SELECT function(attribute), function(attribute), ...
  INTO dst-array
  FROM src-array | array-expression
  WHERE where-expression
  GROUP BY dimension, dimension, ...;
```

For example, this query selects the maximum value from the attribute `val` of array `m4x4` grouped by dimension `x`:

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

This query outputs:

```
[(3),(7),(11),(15)]
```

which has schema:

```
<max:double NULL> [x=0:3,4,0]
```

This query selects the maximum values from attribute `attr1` of array `m4x4` grouped by dimension `y`:

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

```
[(12),(13),(14),(15)]
```

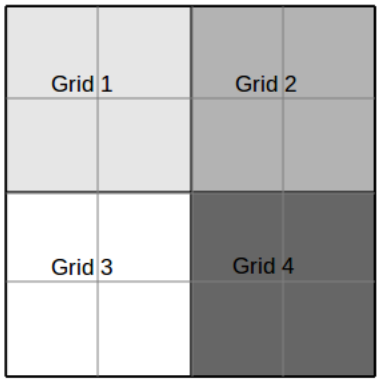
The AFL `aggregate` operator takes dimension arguments to support group-by functionality. This query selects the maximum values from the dimension `y` and attribute `val` from the array `m4x4` using AFL:

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

```
[(12),(13),(14),(15)]
```

7.3. Grid Aggregates

A grid aggregate selects nonoverlapping subarrays from an existing array and calculates an aggregate of each subarray. For example, if you have a 4x4 array, you can create 4 nonoverlapping 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:

```
SELECT function(attribute), function(attribute), ...
INTO dst-array
FROM src-array | array-expression
WHERE where-expression
REGRID dimension1-size, 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 2,2;
```

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

This output has schema:

```
<max:double NULL,min_1:double NULL> [x=0:1,2,0,y=0:1,2,0]
```

In AFL, you can use the `regrid` operator:

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

```
[
[(5,0),(7,2)],
```

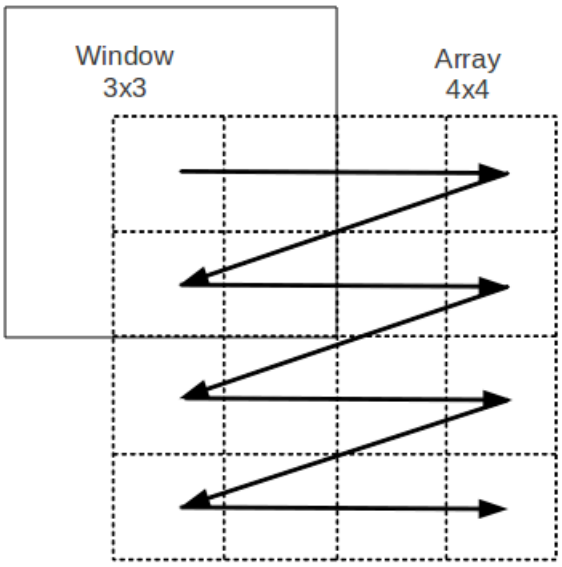
```
[(13,8),(15,10)]
]
```

7.4. 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 grouping starts at the first element of the array and moves in stride-major order from the lowest to highest value in each dimension. The syntax of a window aggregate statement is:

```
SELECT function(attribute), function(attribute), ...
INTO dst-array
FROM src-array | array-expression
WHERE where-expression
WINDOW dimension1-size, dimension2-size, ...;
```

For example, you can use a 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
WINDOW 3,3;
```

Which returns values:

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

with schema:

```
<sum:double NULL> [x=0:3,4,0,y=0:3,4,0]
```

Since the window centroid starts at cell {0,0}, the region of the window that is outside the array boundary is not counted in the aggregation.

In AFL, you would use the `window` operator:

```
AFL% window(m4x4,3,3,sum(attr1));
```

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

Chapter 8. Updating Your Data

SciDB uses a "no overwrite" storage model. No overwrite means that data in an array can be updated but previous values can be accessed as long as the array exists in the SciDB namespace. Every time you update data in an array, SciDB creates a new array version, much like source control systems for software development.

8.1. The UPDATE ... SET statement

To update data in an existing SciDB array, use the statement:

```
UPDATE array SET "attr = expr", ... [ WHERE condition ];
```

Consider the following 2-dimensional array, `m4x4`:

```
[("m4x4<val:double NOT NULL> [x=0:3,4,0,y=0:3,4,0]")]
[
[(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:

```
UPDATE m4x4 SET val=-val;
```

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

The **WHERE** clause lets you choose attributes based on conditions. For example, you can select just cells with absolute values greater than 5 to set to their multiplicative inverse:

```
UPDATE m4x4 SET val=pow(val,-1) WHERE abs(val) > 5;
```

```
[
[(0),(-1),(-2),(-3)],
[(-4),(-5),(-0.166667),(-0.142857)],
[(-0.125),(-0.111111),(-0.1),(-0.0909091)],
[(-0.0833333),(-0.0769231),(-0.0714286),(-0.0666667)]
]
```

8.2. Array Versions

When an array is updated, a new array version is created. SciDB stores the array versions. For example, in the previous section, SciDB stored every version of m4x4 created by the **UPDATE** command. You can see these versions with `versions`:

```
AQL% SELECT * FROM versions(m4x4);
```

```
[(1,"2012-02-03 17:20:50"),
(2,"2012-02-06 14:51:20"),
(3,"2012-02-06 14:52:33")]
```

You can see the contents of any previous version of the array by using the version number:

```
AQL% SELECT * FROM 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('2012-02-03 17:20:50'));
[
[(0),(1),(2),(3)],
[(4),(5),(6),(7)],
[(8),(9),(10),(11)],
[(12),(13),(14),(15)]
]
```

You can use the array version name in any query. The unqualified name of the array always refers to the most recent version as of the start of the query.

Chapter 9. Changing Array Schemas: Transforming Your SciDB Array

9.1. Array Transformations

Once you have created and loaded a SciDB array, you may want to change some aspect of that array. SciDB offers functionality to transform the elements of the array schema (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:

```
SELECT * FROM operation(source_array,parameters)
```

This query outputs a SciDB array. To store that array result, use the `INTO` clause:

```
SELECT * INTO result_array FROM operation(source_array,parameters)
```

9.1.1. Rearranging Array Data

SciDB offers functionality to rearrange an array data:

- **Reshaping** an array by changing the dimension sizes. is performed with the `reshape` command.
- **Unpacking** a multidimensional array into a 1-dimensional array is performed with the `unpack` command.
- **Reversing** the cells in a dimension is performed with the `reverse` command.

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` command allows you to transform an array into another compatible schema. Consider a 4-by-4 array, `m4x4`:

```
AFL% show(m4x4);scan(m4x4);
[("m4x4<val:double NOT NULL> [i=0:3,4,0,j=0:3,4,0]")]
[
[(0),(1),(2),(3)],
[(4),(5),(6),(7)],
[(8),(9),(10),(11)],
[(12),(13),(14),(15)]
]
```

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-by-4 array has 16 cells, so you can use any schema with 16 cells, such as 8-by-2, as the new schema:

```
AQL% SELECT * INTO m8x2 FROM
reshape(m4x4,<val:double>
[i2=0:7,8,0,j2=0:1,2,0]);
```

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

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

A special case of reshaping is unpacking a multidimensional array to a 1-dimensional 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 is you are planning to save your data to csv format.

The `unpack` command takes the second and higher dimensions of an array and transforms them into attributes along the first dimension. The result array consists of the dimension values of the input array with the attribute values from the corresponding cells appended. So, an attribute value `val` that was in row 1, column 3 of a 2-dimensional array will be transformed into a cell with attribute values 1,3,`val`. For example, a 2-dimensional, 1-attribute array will output a 1-dimensional, 3-attribute array as follows:

```
AQL% SELECT * FROM show(m3x3);
```

```
[("m3x3<val:double NOT NULL> [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),
(0,0,0),
(0,0,0),
(0,0,0)]
```

```
AQL% SELECT * FROM show(m1);
```

```
[("m1<i:int64 NOT NULL,
j:int64 NOT NULL,
val:double NOT NULL>
[val1=0:15,4,0]")]
```

You can reverse the ordering of the data in each dimension of an array with the `reverse` command:

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

```
[("m3x3<val:double NOT NULL> [i=0:2,3,0,j=0:2,3,0]")]
[[(0),(1),(2)],[(3),(4),(5)],[(6),(7),(8)]]
```

```
AQL% SELECT * FROM reverse(m3x3);
```

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

9.1.2. Reduce an Array

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` command.
- An array **slice** is a subset of the array defined by planes of the array. This action is performed by the `slice` command.
- A dimension can be winnowed or **thinned** by selecting data at intervals along its entirety. This action is performed by the `thin` command.

You can select part of an existing array into another array with the `subarray` command. For example, you can select a 2-by-2 array of the last two values from each dimension of the array `m4x4` with the following `subarray` command:

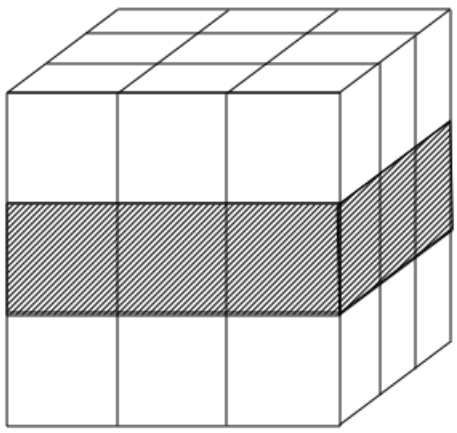
```
AFL% show(m4x4);scan(m4x4);
[("m4x4<val:double NOT NULL>
[i=0:3,4,0,j=0:3,4,0]")]
```

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

```
AQL% SELECT * FROM subarray(m4x4,2,2,3,3);
```

```
[
[(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:



For example, you can select the data in a horizontal slice in the middle of a 3-dimensional array `m3x3x3` by using the `slice` command and specifying the value for dimension `k`:

```
AFL% show(m3x3x3);scan(m3x3x3);
[("m3x3x3<val:double NOT NULL>
[i=0:2,3,0,j=0:2,3,0,k=0:2,3,0]")]
```

```
[
[[(0),(1),(2)],
[(4),(5),(6)],
[(8),(9),(10)]
],
[[(7),(8),(9)],
[(11),(12),(13)],
[(15),(16),(17)]
],
[
[(14),(15),(16)],
[(18),(19),(20)],
[(22),(23),(24)]
]
]
```

```
AFL% slice(m3x3x3,k,1);
```

```
[
[(1),(5),(9)],
[(8),(12),(16)],
[(15),(19),(23)]
]
```

You may want to sample data uniformly across an entire dimension. The `thin` command selects elements from given array dimensions at defined intervals. For example, you can select every other element from every other row:

```
AFL% show(m4x4);scan(m4x4);
[("m4x4<val:double NOT NULL>
[i=0:3,4,0,j=0:3,4,0]")]
```

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

```
AQL% SELECT * FROM thin(m4x4,1,2,0,2);
```

```
[[(4),(6)],
[(12),(14)]]
```

9.2. Changing Array Attributes

An array's attributes contain the data stored in the array. You can transform attributes by

- Changing the name of the attribute.
- Adding an attribute.
- Changing the order of attributes in a cell.
- Deleting an attribute.

You can change the name of an attribute with the `attribute_rename` command:

```
AQL% SELECT * INTO m3x3_new FROM
attribute_rename(m3x3,val,val2);
```

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

```
AQL% SELECT * FROM show(m3x3_new);
```

```
[("m3x3_new<val2:double NOT NULL> [i=0:2,3,0,j=0:2,3,0]")]
```

You can add attributes to an existing array with the `apply` command:

```
AQL% SELECT * INTO m3x3_new_attr
FROM apply(m3x3,val2,val+10,val3,pow(val,2));
```

```
[
[(0,10,0),(1,11,1),(2,12,4)],
[(3,13,9),(4,14,16),(5,15,25)],
[(6,16,36),(7,17,49),(8,18,64)]
]
```

```
AQL% SELECT * FROM show(m3x3_new_attr);
```

```
[("m3x3_new_attr
<val:double NOT NULL,val2:double NOT NULL,val3:double NOT NULL>
[i=0:2,3,0,j=0:2,3,0]")]
```

You can select a subset of an array's attributes and return them in any order with the `project` command.

```
AQL% SELECT * FROM project(m3x3_new_attr,val3,val2);
```

```
[
[(0,10),(1,11),(4,12)],
[(9,13),(16,14),(25,15)],
[(36,16),(49,17),(64,18)]
]
```

9.3. Changing Array Dimensions

9.3.1. 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` command to change the chunk size. For example, suppose you have an array that is 1000-by-1000 with chunk size 100 in each dimension:

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

```
[("chunks<val1:double NOT NULL,val2:double NOT NULL>
[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]);
AQL% SELECT * FROM show(chunks_part);
```

```
[("chunks_part<val1:double NOT NULL,
val2:double NOT NULL>
[i=0:999,10,0,j=0:999,1000,0]")]
```

Repartitioning is also important if you want the change the chunk overlap to speed up nearest-neighbor or window aggregate queries.

```
AQL% SELECT * INTO chunks_overlap
FROM repart(chunks,<val1:double,val2:double>
[i=0:999,100,10,j=0:999,100,10]);
```

9.3.2. Appending a Dimension

You may need to append dimensions to existing arrays, particularly when you want to do more complicated tranformations 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% show(Dsp);scan(Dsp);
```

```
[("Dsp<val:double NOT NULL,
empty_indicator:indicator NOT NULL>
[s(string)=5,5,0,p(string)=5,5,0]")]

[
[(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)]
]
```

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

```
AQL% SELECT * INTO Dsp_slice_0
FROM slice(Dsp,s,'sample-0');

AQL% SELECT * INTO Dsp_slice_1
FROM slice(Dsp,s,'sample-1');

AQL% SELECT * INTO Dsp_slice_2
FROM slice(Dsp,s,'sample-2');
```

The slices are 1-dimensional.

```
AQL% SELECT * FROM show(Dsp_slice_0);
```

```
[("Dsp_slice_0
<val:double NOT NULL,
empty_indicator:indicator NOT NULL>
[p(string)=5,5,0]")]
```

Concatenating these slices will create a 1-d array:

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

```
[("Dsp_1d<val:double NOT NULL,
empty_indicator:indicator NOT NULL>
[p=0:9,5,0]")]
```

To concatenate these arrays into a 2-dimensional array, you need to add a dimension to both. The `adddim` command will add a stub dimension to the array to increase its dimensionality.

```
AQL% SELECT * INTO Dsp_new FROM
concat(adddim(Dsp_slice_0, s),
adddim(Dsp_slice_2, s));
AQL% SELECT * FROM show(Dsp_new);
```

```
[("Dsp_new<val:double NOT NULL,
empty_indicator:indicator NOT NULL>
[s=0:1,1,0,p(string)=5,5,0]")]
```