SciDB Reference Guide

SciDB Reference Guide

Version 12.3

Table of Contents

List of Tables

Chapter 1. SciDB Aggregate Reference

This chapter lists SciDB aggregates. Aggregates take as input a set of 1 or more values and return a scalar value. SciDB aggregates have the syntax aggregate_call_N where an aggregate call is one of the following:

- `aggregate_name(attribute_name)`
- `aggregate_name(array_name)`
- `aggregate_name(scalar)`

Aggregate calls can occur in AQL and AFL statements as follows:

AQL syntaxes

```
SELECT aggregate_call_1[,aggregate_call_2,...,aggregate_call_N]
FROM array;

SELECT aggregate_call_1[,aggregate_call_2,...,aggregate_call_N]
FROM array GROUP BY dimension1[,dimension2];

SELECT aggregate_call_1[,aggregate_call_2,...,aggregate_call_N]
FROM array WHERE expression;

SELECT aggregate_call_1[,aggregate_call_2,...,aggregate_call_N]
FROM array REGRID dimension_1, dimension_2,...

SELECT aggregate_call_1[,aggregate_call_2,...,aggregate_call_N]
FROM array WINDOW window_dim_1, window_dim_2,...
```

AFL syntaxes

```
aggregate(array, aggregate_call_1
[, aggregate_call_2,... aggregate_call_N]
[,dimension_1, dimension_2,...])

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

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

Name

avg — Average (mean) aggregate

Synopsis

```
SELECT avg(attribute) FROM array;
```

```
aggregate(array,avg(attribute)[,dimension_1,dimension_2,...]
```

Summary

The avg aggregate takes a set of scalar values from an array attribute and returns the average of those values.

Example

This example finds the average of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Find the average of every column of m3x3:

   ```
   aggregate(m3x3,val,j)
   ```

   This query returns:

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

Name

count — Count nonempty elements aggregate

Synopsis

```
SELECT count(attribute) FROM array;

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

Summary

The count aggregate counts nonempty elements of an array's attributes.

Example

This example finds the number of nonempty cells in a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values 1 along the diagonal of m3x3 and leave the remaining cells empty:

   ```
   store(build_sparse(m3x3,i=j,1),m3x3);
   ```

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

   ```
   aggregate(m3x3,count(val));
   ```

   This query returns:

   ```
   [(6)]
   ```

Name

max — Maximum value aggregate

Synopsis

```
SELECT max(attribute) FROM array;
```

```
aggregate(array,max(attribute)[,dimension_1,dimension_2,...]
```

Summary

Example

This example finds the maximum of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Find the maximum value of each column:

   ```
   aggregate(m3x3,max(val),j);
   ```

   This query returns:

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

Name

min — Minimum value aggregate

Synopsis

```
SELECT min(attribute) FROM array;
```

```
aggregate(array,min(attribute)[,dimension_1,dimension_2,...]
```

Summary

Example

This example finds the minimum of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Find the minimum value of every column of m3x3:

   ```
   aggregate(m3x3,min(val),j);
   ```

   This query returns:

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

Name

stdev — Standard deviation aggregate

Synopsis

```
SELECT stdev(attribute) FROM array;
```

```
aggregate(array,stdev(attribute)[,dimension_1,dimension_2,...]
```

Summary

Example

This example finds the standard deviation of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put random values between 1 and 9 into m3x3:

   ```
   store(build(m3x3,random()%10/1.0),m3x3);
   ```

   This query outputs:

   ```
   [
   [(2),(8),(0)],
   [(5),(2),(6)],
   [(2),(0),(2)]
   ]
   ```

3. Find the standard deviation of every column of m3x3:

   ```
   aggregate(m3x3,stdev(val),j);
   ```

   This query returns:

   ```
   [(1.73205),(4.16333),(3.05505)]
   ```

Name

sum — Sum aggregate

Synopsis

```
SELECT sum(attribute) FROM array;
```

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

Summary

The sum aggregate calculates the cumulative sum of a group of values.

Example

This example finds the sum of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Find the sum of each column in m3x3:

   ```
   aggregate(m3x3,sum(val),j)
   ```

   This query returns:

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

Name

var — Variance aggregate

Synopsis

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

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

Summary

The var aggregate returns the variance of a set of values.

Example

This example finds the variance of every column of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put random values between 1 and 9 into m3x3:

   ```
   store(build(m3x3,random()%10/1.0),m3x3);
   ```

   This query returns:

   ```
   [
   [(2),(8),(0)],
   [(5),(2),(6)],
   [(2),(0),(2)]
   ]
   ```

3. Find the variance for each column of m3x3:

   ```
   aggregate(m3x3,var(val),j)
   ```

   This query returns:

   ```
   [(3),(17.3333),(9.33333)]
   ```

Chapter 2. SciDB Function Reference

Table 2.1. List of SciDB Functions

Function Name	Description	Category
%	Remainder	Arithmetic
*	Multiplication	Arithmetic
+	Addition	Arithmetic
-	Subtraction	Arithmetic
/	Division	Arithmetic
<	Less than	Logical
<=	Less than or equal	Logical
<>	Not equal	Logical
=	Equals	Logical
>	Greater than	Logical
>=	Greater than or equal	Logical
abs	Absolute value	Arithmetic
acos	Inverse (arc) cosine in radians	Transcendental
and	Boolean AND	Logical
append_offset	Change time and date by a given amount	Timestamp
apply_offset	Change time and date by a given amount	Timestamp
asin	Inverse (arc) sine in radians	Transcendental
atan	Inverse (arc) tangent in radians	Transcendental
ceil	Round to next-highest integer	Arithmetic
cos	Cosine (input in radians)	Transcendental
exp	Exponential	Transcendental
first	Start of string	Strings
floor	Round to next-lowest integer	Arithmetic
get_offset	Returns time offset in seconds	Timestamp
high	String information	Strings
iif	Inline IF	Logical
is_nan	Returns TRUE is attribute value is NaN	Logical
is_null	Returns TRUE is attribute value is null	Logical
last	End of string	String
length	Get string length	String
log	Base-e logarithm	Transcendental
log10	Base-10 logarithm	Transcendental
low	String query	String
instanceid	Return instance id	Troubleshooting

Function Name	**Description**	**Category**
not	Boolean NOT	Logical
now	Current array version	Timestamp
or	Boolean OR	Logical
pow	Raise to a power	Arithmetic
random	Random number	Arithmetic
regex	Search for regular expression	Strings
sin	Sine (input in radians)	Transcendental
sqrt	Square root	Arithmetic
strchar	Convert string to char	Datatype conversion
strftime	Convert string to datetime	Datatype conversion
strip_offset	disregards OFFSET and returns result as a DATETIME	Timestamp
strlen	Maximum string length	Strings
substr	Select substring	Strings
tan	Tangent (input in radians)	Transcendental
togmt	Switch to GMT from current time zone setting	Timestamp
tznow	Set time zone	Timestamp

Chapter 3. SciDB Data Type Reference

SciDB supports the following data types.

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

Chapter 4. SciDB Operator Reference

This reference guide lists the operators available in SciDB.

Name

analyze — Analyze load file for chunk size

Synopsis

```
SELECT * FROM analyze(array[, attribute1, attribute2, ...])
```

Summary

The analyze operator helps you decide how to set chunk size when you are creating a SciDB array. The analyze() operator takes as input a source array and a list of attributes in that source array. The operator computes the range of values (the maximum and minimum values), population count, and an estimate of the number of distinct values in the attribute (or combination of attributes).

Example

This example runs analyze on a 2-attribute, 1-dimensional array. The example uses the file doc/user/examples/num_data.scidb, shown here:

```
{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),
(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)
]
```

1. Create an array analyze_array with 1 unbounded dimension:

   ```
   AQL% CREATE ARRAY analyze_array
   <val1:double,val2:double> [line=0:*,10,0];
   ```

2. Load the file num_data.scidb into analyze_array:

   ```
   AQL% LOAD analyze_array
   FROM 'path/doc/user/examples/num_data.scidb';
   ```

3. Analyze the array for chunk sizes:

```
analyze(analyze_array);
```

This query returns:

```
[("val","5.65393e+07","1.8548e+09",20,20),("val2","1","2",2,20)]
```

The output array contains one attribute for every attribute in the input. Each attribute of the output contains the attribute name, maximum value, minimum value, number of distinct elements, and total number of elements.

Name

apply — Apply expression to compute new attribute values

Synopsis

```
SELECT * FROM apply(array,new_attribute1,expression1
[,new_attribute2,expression2]);
```

```
store(apply(array,new_attribute1,expression1[,new_attribute2,expression2]),target_
```

Summary

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

Example

This example computes new attributes for an existing array.

1. Create an array called distance with an attribute called miles:

   ```
   CREATE ARRAY distance <miles:double> [i=0:9,10,0];
   ```

2. Store values of 100–1000 into the array:

   ```
   store(build(distance,i+100.0),distance);
   ```

3. Apply the expression 1.6 * miles to distance and name the result kilometers:

   ```
   apply(distance,kilometers,1.6*miles);
   ```

   This query returns:

   ```
   [(100,160),
   (200,320),
   (300,480),
   (400,640),
   (500,800),
   (600,960),
   (700,1120),
   (800,1280),
   (900,1440),
   (1000,1600)]
   ```

Name

attribute_rename — Rename an array attribute

Synopsis

```
SELECT * FROM attribute_rename(array,old_attribute,new_attribute);
```

Summary

Changes an attribute name in an array.

Example

1. Create an array called array1 with an attribute called val:

   ```
   CREATE ARRAY array1 <val:double>[i];
   ```

2. Rename val to val2:

   ```
   attribute_rename(array1,val,val2);
   ```

Name

attributes — List array attributes

Synopsis

```
SELECT * FROM attributes(array);
```

Summary

The attributes operator lists all the attributes of an array. The output returns the attribute name, the attribute data type, and a Boolean flag representing whether or not the attribute can be null.

Name

avg — Average (mean) value

Synopsis

```
SELECT * FROM avg(array,attribute[,dimension1,dimension2,...]);
```

Summary

The avg operator finds the average value of an array attribute.

Note

The avg operator provides the same functionality as the avg aggregate, but has a different syntax. See the avg aggregate reference page.

Example

This example finds the average value along the second dimension of a 4×4 matrix.

1. Create an array named avg_array:

   ```
   CREATE ARRAY avg_array<val:double>[i=0:3,4,0,j=0:3,4,0];
   ```

2. Store values of 0–15 in avg_array:

   ```
   store(build(avg_array,i*4+j),avg_array);
   ```

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

3. Find the average value along the dimension j:

   ```
   avg(avg_array,val,j);
   ```

   This query outputs:

   ```
   [(1.5),(5.5),(9.5),(13.5)]
   ```

Name

bernoulli — Select random array cells

Synopsis

```
SELECT * FROM bernoulli(array,probability[, seed]);
```

Summary

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

Example

This example select cells at random from a 4×4 matrix, and uses a seed value to select the same cells in successive trials.

1. Create an array called bernoulli_array:

   ```
   CREATE ARRAY bernoulli_array<val:double>[i=0:3,4,0,j=0:3,4,0];
   ```

2. Store values of 0–15 in bernoulli_array:

   ```
   store(build(bernoulli_array,i*4+j),bernoulli_array);
   ```

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

3. Select cells at random with a probability of .5 that a cell will be included. Each successive call to bernoulli will return different results.

   ```
   AFL% bernoulli(bernoulli_array,.5);
   ```

   ```
   [
   [(),(1),(),(3)],
   [(4),(),(),()],
   [(),(9),(),(11)],
   [(12),(),(14),(15)]
   ]
   ```

   ```
   bernoulli(bernoulli_array,.5);
   ```

   ```
   [
   [(),(1),(),(3)],
   [(),(5),(6),()],
   [(),(9),(),(11)],
   [(),(13),(14),()]]
   ```

4. To reproduce earlier results, use a seed value. Seeds must be an integer on the interval [0, INT_MAX].

```
bernoulli(bernoulli_array,.5,1);
```

```
[
[(),(),(2),()],
[(),(),(6),(7)],
[(8),(9),(),(11)],
[(12),(),(),()]
]
```

```
AFL% bernoulli(bernoulli_array,.5,1);
```

```
[
[(),(),(2),()],
[(),(),(6),(7)],
[(8),(9),(),(11)],
[(12),(),(),()]
]
```

Name

between — Select array data from specified region

Synopsis

```
SELECT * FROM between(array,low_coord1,low_coord2,...,high_coord1,high_coord2);
```

Summary

The between operator accepts an input array and a set of coordinates specifying a region within the array. The number of coordinate pairs in the input must be equal to the number of dimensions in the array. The output is an array of the same shape as input, where all cells outside of the given region are marked empty.

Example

This example selects 4 elements from a 16-element array.

1. Create a 4×4 array called between_array:

   ```
   CREATE ARRAY between_array <val:double>[i=0:3,4,0,j=0:3,4,0];
   ```

2.
   ```
   store(build(between_array,i*4+j),between_array);
   ```

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

3. Select all values from the last two rows and last two columns from between_array:

   ```
   between(between_array,2,2,3,3);
   ```

   This query outputs:

   ```
   [
   [(),(),(),()],
   [(),(),(),()],
   [(),(),(10),(11)],
   [(),(),(14),(15)]
   ]
   ```

Name

build — Assign values to array attribute

Synopsis

```
SELECT * INTO target_array FROM build(source_array,expression);

store(build(source_array,expression);
```

Summary

The build operator proceeds through source_array, cell by cell, using the value of expression to compute the value of each cell. The expression argument can be any combination of SciDB functions applied to scalars or SciDB array attributes.

Example

Create an array of all ones:

```
build(<val:double>[i=0:3,4,0,j=0:3,4,0],1);
```

Create an identity matrix:

```
build(<val:double>[i=0:3,4,0,j=0:3,4,0],iif(i=j,1,0));
```

Build an array of monotonically increasing values:

```
build(<val:double>[i=0:3,4,0,j=0:3,4,0],i*4+j);
```

To store the result from a build operator, create an array and use the store operator with the build operator.

```
CREATE ARRAY identity_matrix <val:double>[i=0:3,4,0,j=0:3,4,0];
store(build(identity_matrix,iif(i=j,1,0)),identity_matrix);
```

Limitations

- The build operator can only take arrays with one attribute.
- The build operator can only take arrays with bounded dimensions.

Name

build_sparse — Assign values to attributes of a sparse array

Synopsis

```
SELECT * INTO target_array FROM build_sparse(source_array,expression,boolean_expre

store(build_sparse(source_array,expression,boolean_expression));
```

Summary

The build_sparse operator takes as input an array or anonymous schema, an expression that defines a scalar value, and an expression that defines a Boolean value. The output of build_sparse contains an array with the same schema as the input array or anonymous schema, the value specified by *`expression`* wherever *`boolean_expression`* evaluates to true, and empty cells wherever *`boolean_expression`* evaluates to false.

Example

Build a sparse-formatted identity matrix where the cells that would be occupied by 0 are empty:

```
build_sparse(<val:double>[i=0:3,4,0,j=0:3,4,0],1,i=j);
```

This query outputs:

```
[[{0,0}(1),{1,1}(1),{2,2}(1),{3,3}(1)]]
```

Name

cancel — Cancel a query

Synopsis

```
cancel(query_id);
```

This operator is designed for internal use.

Summary

Cancel a currently running query by query id.

The query id can be obtained from the SciDB log or via the list() command. SciDB maintains query context information for each completed and in-progress query in the server. If the user issues a "ctrl-C" or abort from the client, the query is cancelled and its context is removed from the server.

Name

cast — Change attribute and dimension names

Synopsis

```
SELECT * INTO target_array FROM cast(source_array, schema);

store(cast(source_array,schema),target_array);
```

Summary

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

Example

This example changes the name of an array attribute and an array dimension. The arrays must be compatible; that is, they must have the same number of dimensions and attributes, and the attributes and dimensions must be of the same type.

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

   ```
   CREATE ARRAY source <val:double>[i=0:9,10,0];
   ```

2. Use an anonymous schema to change the attribute name to num_val and the dimension name to x. Store the result in an array called target:

   ```
   store(cast(source, <num_val:double>[x=0:9,10,0]),target);
   ```

   This is useful when you are joining arrays and want to avoid naming conflicts. For example, doing a cross_join on source and target will create an array with two attributes, val and num_val, and two dimensions, i and x:

   ```
   store(cross_join(source,target),new_array);
   show(new_array);
   ```

   ```
   [("new_array<val:double,num_val:double> [i=0:9,10,0,x=0:9,10,0]")]
   ```

Name

concat — Concatenate two arrays

Synopsis

```
SELECT * INTO target_array FROM concat(left_array,right_array);

store(concat(left_array,right_array),target_array);
```

Summary

The concat operator concatenates two arrays with the name number of dimensions. Concatenation is performed by the left-most dimension. All other dimensions of the input arrays must match. The left-most dimension of both arrays must have a fixed size (not unbounded) and same chunk size and overlap. Both inputs must have the same attributes.

Example

This example concatenates a 4×3 array and a 1×3 array.

1. Create a 4×3 array left_array containing value 1 in all cells:

```
create array left_array <val:double>[i=0:3,1,0,j=0:3,1,0];
store(build(left_array,1),left_array);
```

2. Create a 1×3 array right_array containing value 0 in all cells:

```
create array right_array <val:double>[i=0:1,1,0,j=0:2,1,0];
store(build(right_array,0),right_array);
```

3. Concatentate left_array and right_array and store the result in concat_array:

```
store(concat(left_array,right_array),concat_array);
```

This produces an array concat_array with contents and schema as follows:

```
show(concat_array);scan(concat_array);
```

```
show(concat_array);scan(concat_array);
[("concat_array<val:double> [i=0:5,1,0,j=0:2,1,0]")]
[[(1)]];[[(1)]];[[(1)]];[[(1)]];[[(1)]];[[(1)]];
[[(1)]];[[(1)]];[[(1)]];[[(1)]];[[(1)]];[[(1)]];
[[(0)]];[[(0)]];[[(0)]];[[(0)]];[[(0)]];[[(0)]]
```

Name

count — Count nonempty cells

Synopsis

```
SELECT * FROM count(array);
```

Summary

The count operator counts nonempty cells of the input array. When dimensions are provided they are used to do a group-by and a count per resulting group is returned.

Note

The count operator provides the same functionality as the count aggregate. See the count aggregate page.

Example

This example finds the average value along the second dimension of a 4×4 array where some cells are empty.

1. Create an array named source_array:

```
CREATE ARRAY source_array<val:double>[i=0:3,4,0,j=0:3,4,0];
```

2. Store values of 0–15 in source_array:

```
store(build(source_array,i*4+j),source_array);
```

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

3. Use between to create some empty cells in source_array and store the result in count_array:

```
store(between(source_array,1,1,1,2),count_array);
```

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

4. Find the count of nonempty element in count_array:

```
count(count_array);
```

This query outputs:

```
[(2)]
```

5. Count the nonempty elements along the dimensions of count_array:

```
count(count_array,i);
```

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

```
count(count_array,j);
```

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

Name

cross — Cross-product join

Synopsis

```
SELECT * INTO m3xtarget_array FROM cross(left_array,right_array);
```

Summary

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

Example

This example returns the cross-join of a 3×3 array with a vector of length 2.

1. Create a 3×3 array m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

3. Create a vector of length 2 containing values 101 and 102:

   ```
   store(build(<val:double>[i=0:1,1,0],i+101),vector);
   ```

4. Find the cross of m3x3 and vector:

   ```
   store(cross(m3x3,vector),cross_array);
   ```

   This query returns:

   ```
   [
   [[(0,101)],[(1,101)],[(2,101)]],
   [[(3,101)],[(4,101)],[(5,101)]],
   [[(6,101)],[(7,101)],[(8,101)]]];

   [[[(0,102)],[(1,102)],[(2,102)]],
   [[(3,102)],[(4,102)],[(5,102)]],
   [[(6,102)],[(7,102)],[(8,102)]]]
   ```

   The array cross_array has schema:

   ```
   show(cross_array);
   ```

   ```
   [("cross_array<val:double,val_2:double> [i=0:2,3,0,j=0:2,3,0,i_2=0:1,1,0]")]
   ```

Name

cross_join — Cross-product join with equality predicates

Synopsis

```
SELECT * INTO target_array FROM cross_join(left_array,right_array,left_dim1,right_

store(cross_join(left_array,right_array,left_dim1,right_dim_1,...),tagrget_array);
```

Summary

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

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

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

Example

This example returns the cross-join of a 3×3 array with a vector of length 3.

1. Create an array called left_array:

   ```
   create array left_array<val: double>[i=0:2,3,0, j=0:2,3,0];
   ```

2. Store values of 0–8 into left array:

   ```
   store(build(left_array,i*3+j),left_array);
   ```

3. Create an array called right_array:

   ```
   create array right_array<val: double>[k=1:3,3,0];
   ```

4. Store values of 101–103 into right_array:

   ```
   store(build(right_array,k+101),right_array);
   ```

5. Perform a cross-join on left_array and right_array along dimension j of left_array:

   ```
   cross_join(left_array,right_array,j,k);
   ```

   This query outputs:

   ```
   [
   [(0,101),(1,102),(2,103)],
   [(3,101),(4,102),(5,103)],
   [(6,101),(7,102),(8,103)]
   ]
   ```

Name

deldim — Reduce array dimensionality

Synopsis

```
SELECT * FROM deldim(array);
```

Summary

The deldim operator deletes the left-most dimension from the array. Deleted dimension must have size = 1. See the adddim operator reference page for an example.

Name

dimensions — List array dimensions

Synopsis

```
SELECT * FROM dimensions(array);
```

Summary

The argument to the dimensions operator is the name of the array. It returns an array with the following attributes: dimension-name, dimension start-index, dimension-length, chunk size, chunk overlap, low-boundary-index, high-boundary-index, datatype.

Example

This example creates an array with one unbounded dimension and one string-type dimension:

```
CREATE ARRAY unbound_string_dim
<val:double>[i=0:*,10,0,j(string)=10,10,0];
dimensions(unbound_string_dim);
```

This code outputs:

```
[("i",0,4611686018427387903,10,0,4611686018427387903,
-4611686018427387903,"int64"),
("j",0,10,10,0,4611686018427387903,-4611686018427387903,
"string")]
```

Name

diskinfo — Internal debugging: Check disk capacity

Synopsis

```
diskinfo()
```

This operator is designed for internal use.

Summary

Get information about storage space. Returns an array with the following attributes:

- used
- available
- clusterSize
- nFreeClusters
- • nSegments

Name

echo — Print string

Synopsis

```
echo(string)
```

This operator is designed for internal use.

Summary

Accepts a string and returns a single-element array containing the string.

Name

explain_logical, explain_physical — Show query plan

Synopsis

```
explain_logical( query: string, language: string )
explain_physical( query: string, language: string )
```

This operator is designed for internal use.

Summary

The operators explain_logical and explain_physical can be used to emit a human-readable plan string for a particular query without running the query itself. SciDB first constructs a logical plan, optimizes it and then translates it into a physical plan.

Name

filter — Select subset of data by boolean expression

Synopsis

```
SELECT * FROM filter(array,expression);
```

Summary

The filter operator filters out data in the array based on an expression over the attribute and dimension values. The filter operator returns an array the with the same schema as the input array but marks all cells in the input that do not satisfy the predicate expression 'empty'.

Example

This example filters an array to remove outlying values.

1. Create an array m4x4:

   ```
   CREATE ARRAY m4x4<val:double>[i=0:3,4,0,j=0:3,4,0];
   ```

2. Put values between 0 and 15 into the nondiagonal elements of m4x4 and values greater than 100 into the diagonal elements:

   ```
   store(build(m4x4,iif(i=j,100+i,i*4+j)),m4x4);
   ```

   ```
   [
   [(100),(1),(2),(3)],
   [(4),(101),(6),(7)],
   [(8),(9),(102),(11)],
   [(12),(13),(14),(103)]
   ]
   ```

3. Filter all values of 100 or greater out of m4x4:

   ```
   filter(m4x4,val<100);
   ```

   This query outputs:

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

Name

help — Operator signature

Synopsis

```
SELECT * FROM help(operator_name);
```

Summary

Accepts an operator name and returns an array containing a human-readable signature for that operator.

Example

This example returns the signature of the multiply operator.

```
help('multiply');
```

Name

input — Read a system file

Synopsis

```
SELECT * INTO target_array FROM input(source_array,filename,[,instance_id]);

store(input(target_array,filename),target_array[,instance_id]);
```

Summary

Input works exactly the same way as load, except it does NOT store the data unless the INTO clause or the AFL store operator is present. The instance_id argument allows you to select which SciDB instance you want to input into. To see a list of SciDB instances running on your system, type scidb.py status hostname athte Unix command-line.

Example

This example reads a csv file from the examples directory.

```
input(m4x4,'path/trunk/doc/user/examples/m4x4_missing.txt);
```

Name

inverse — Matrix inverse

Synopsis

```
SELECT * FROM inverse(array);
```

Summary

The inverse operator produces the matrix inverse of a square matrix. The input matrix must be invertible, i.e., the determinant of the matrix must be nonzero.

Example

This example find the matrix inverse of a 3×3 matrix.

1. Create a matrix m3x3:

   ```
   CREATE ARRAY <val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 1 and 2 into m3x3 to represent a nonsingular matrix:

   ```
   store(build(m3x3,iif(i=j,1,2)),m3x3);
   ```

   This query outputs:

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

3.
   ```
   inverse(m3x3);
   ```

   This query outputs:

   ```
   [
   [(-0.6),(0.4),(0.4)],
   [(0.4),(-0.6),(0.4)],
   [(0.4),(0.4),(-0.6)]
   ]
   ```

Name

join — Join two arrays

Synopsis

```
SELECT * INTO target_array FROM join(left_array,right_array);

store(join(left_array,right_array),target_array)
```

Summary

Join combines the attributes of two input arrays at matching dimension values. The join result has the same dimension names as the first input. The left and right arrays must have the same shape. If a cell in either the left or right array is empty, the corresponding cell in the result is also empty.

Example

This example concatenates two 3×3 arrays.

1. Create a 3×3 array left_array containing value 1 in all cells:

   ```
   create array left_array <val:double>[i=0:2,3,0,j=0:2,3,0];
   store(build(left_array,1),left_array);
   ```

2. Create a 3×3 array right_array containing value 0 in all cells:

   ```
   create array right_array <val:double>[i=0:2,3,0,j=0:2,3,0];
   store(build(right_array,0),right_array);
   ```

3. Join left_array and right_array:

   ```
   join(left_array,right_array);
   ```

   This produces an array concat_array with contents and schema as follows:

   ```
   show(concat_array);scan(concat_array);
   ```

   ```
   show(concat_array);scan(concat_array);
   [("concat_array<val:double> [i=0:5,1,0,j=0:2,1,0]")]
   [[(1)]];[[(1)]];[[(1)]];[[(1)]];[[(1)]];[[(1)]];
   [[(1)]];[[(1)]];[[(1)]];[[(1)]];[[(1)]];[[(1)]];
   [[(0)]];[[(0)]];[[(0)]];[[(0)]];[[(0)]];[[(0)]]
   ```

Name

list — List contents of SciDB namspace

Synopsis

```
SELECT * FROM list(element)
```

Summary

The list operator allows you to get a list of elements in the current SciDB instance. The input is one of the following strings:

aggregates	Show all operators that take as input a SciDB array and return a scalar.
arrays	Show all functions. Each function will be listed with its available dataypes and the library in functions which it resides.
functions	Show all libraries that are loaded in the current SciDB instance.
instances	Show all SciDB instances. Each instance will be listed with its port, id number, and time-and-date stamps for when it came online.
libraries	Show all libraries that are loaded in the current SciDB session.
operators	Show all operators and the libraries in which they reside.
types	Show all the dataypes the SciDB supports.
queries	Show all active queries. Each active query will have an id, a time and date when it was queries initiated, an error code, whether it generated any errors, and a status (boolean flag where TRUE means that the query is idle).

Name

load_library — Load a plugin

Synopsis

```
load_library(library_name);
```

Summary

Load a SciDB plugin. The act of loading a plugin shared library first registers the library in the SciDB system catalogs. Then it opens and examines the shared library to store its contents with SciDB's internal extension management subsystem. Shared library module which are registered with the SciDB instance will be loaded at system start time.

Example

```
load_library('librational')
```

Name

lookup — Select array cells by dimension index

Synopsis

```
SELECT * FROM lookup(pattern_array,source_array);
```

Summary

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

Example

This example selects a row from a 2-dimensional array.

1. Create an vector of ones called indices1:

   ```
   store(build(<val1:double>[i=0:3,4,0],1),indices1);
   ```

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

2. Create a vector with values between 0 and 3 called indices2:

   ```
   store(build(<val1:double>[i=0:3,4,0],i),indices2);
   ```

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

3. Join indices1 and indices2 into a two-attribute array called pattern_array:

   ```
   store(join(indices1,indices2),pattern_array);
   ```

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

4. Create a 2-dimensional array called source_array with values between 100 and 115:

   ```
   store(build(<val:double>[i=0:3,4,0,j=0:3,4,0],i*4+j+100),source_array);
   ```

   ```
   [
   [(100),(101),(102),(103)],
   [(104),(105),(106),(107)],
   [(108),(109),(110),(111)],
   [(112),(113),(114),(115)]
   ]
   ```

5. Use lookup to use the dimension coordinates array pattern_array to return the second row of source_array:

   ```
   lookup(pattern_array,source_array);
   ```

   This query outputs:

   ```
   [(104),(105),(106),(107)]
   ```

Name

max — Select maximum value

Synopsis

```
SELECT * FROM max(array[,attribute][,dimension1,dimension2,...])
```

Summary

The max operator calculate maximum of the specified attribute in the array. Result is an array with single element containing maximum of specified attribute.

Note

The max operator provides the same functionality as the max aggregate. See the max aggregate reference page for more information.

Example

This example find the maximum value of each row of a 2-dimensional array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

```
CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
```

2. Store values of 0–8 in m3x3:

```
store(build(m3x3,i*3+j),m3x3);
```

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

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

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

This query returns:

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

Name

merge — Merge two arrays

Synopsis

```
SELECT * INTO target_array FROM merge(left_array,right_array);
```

Summary

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

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

Example

This example merges two sparse arrays.

1. Create a sparse array left_array containing value 1 in the first row:

   ```
   store(build_sparse(<val:double>[i=0:2,3,0,j=0:2,3,0],1,i=0),left_array);
   ```

   This query outputs:

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

2. Create a sparse identity matrix called right_array

   ```
   store(build_sparse(<val:double>[i=0:2,3,0,j=0:2,3,0],1,i=j),right_array);
   ```

   This query outputs:

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

3. Merge left_array and right_array:

   ```
   merge(left_array,right_array);
   ```

   This query outputs:

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

Name

min — Select minimum value

Synopsis

```
SELECT * FROM min(array,attribute[,dimension_1,dimension_2,...]);
```

Summary

The min operator selects the minimum value from an array attribute.

Note

The min operator provides the same functionality as the min aggregate. See the min aggregate reference page for more information.

Example

This example finds the minimum value of each row of a 2-dimensional array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

   ```
   CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Store values of 0–8 in m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

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

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

   This query returns:

   ```
   [(0),(3),(6)]
   ```

Name

multiply — Matrix multiplication

Synopsis

```
SELECT * FROM multiply(left_array,right_array);
```

Summary

The multiply operator performs matrix multiplication on two input matrices and returns a result matrix.

Both inputs must be compatible for the multiply operation, and both must have a single attribute. To be compatible, two matrices must have the same size of 'inner' dimension and same chunk size along that dimension.

Example

This example multiplies a 3×2 array and a 2×3 array.

1. Create a 3×2 array lhs:

   ```
   store(build(<val:double>[i=0:2,3,0,j=0:1,2,0],(i+1)*3+j),lhs);
   ```

   This query outputs:

   ```
   [
   [(3),(4)],
   [(6),(7)],
   [(9),(10)]
   ]
   ```

2. Create a 2×3 array rhs:

   ```
   store(build(<val:double>[i=0:1,2,0,j=0:2,3,0],(i+1)*3-j),rhs);
   ```

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

3. Multiply lhs and rhs.

   ```
   multiply(lhs,rhs)
   ```

   This query returns:

   ```
   [
   [(33),(26),(19)],
   [(60),(47),(34)],
   [(87),(68),(49)]
   ]
   ```

Name

normalize — Divide each element of a vector by the square root of the sum of squares of the elements

Synopsis

```
SELECT * FROM normalize(array,attribute);
```

Summary

The normalize operator scales the values of a vector.

Example

Scale a vector whose values are between 1 and 10.

```
store(build(<val:double>[i=0:9,10,0],(i+1)),unscaled);
[(1),(2),(3),(4),(5),(6),(7),(8),(9),(10)]
normalize(unscaled);
[(0.0509647),(0.101929),(0.152894),(0.203859),(0.254824),(0.305788),(0.356753),(0.
```

Limitations

The normalize operator can only take 1-dimensional arrays.

Name

project — Select array attributes

Synopsis

```
SELECT * INTO target_array FROM project(source_array, attribute1, attribute2,...);

store(project(source_array,attribute1,attribute2,...),target_array);
```

Summary

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

Example

This example takes an array with 3 attributes and returns an array with 2 attributes.

1. Create an array source_array:

   ```
   store(build(<val1:double>[i=0:4,5,0],1),source_array);
   ```

   This query outputs:

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

2. Create an attribute val2 and store val1 and val2 in the array:

   ```
   store(apply(source_array,val2,i+1),two_attr);
   ```

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

3. Create an attribute called val3 and store val1, val2, and val3 in an array three_attr:

   ```
   store(apply(two_attr,val3,sin(val2/1.0)),three_attr);
   ```

   ```
   [(1,1,0.841471),(1,2,0.909297),(1,3,0.14112),(1,4,-0.756802),(1,5,-0.958924)]
   ```

4. Project attribute val3 and val2:

   ```
   project(three_attr,val3,val2);
   ```

   This query outputs:

   ```
   [(0.841471,1),(0.909297,2),(0.14112,3),(-0.756802,4),(-0.958924,5)]
   ```

Name

redimension — Change attributes to dimensions

Synopsis

```
AFL% redimension(source_array,target_array)
```

Summary

The redimension operator changes attributes to dimensions. The input and target arrays must have compatible schemas, and both commands determine the list of transformations (attribute to dimension) by matching names in the attribute and dimension lists of the two arrays.

Example

This example redimensions a 2-attribute, 1-dimensional array into a 2-dimensional, 1-attribute array. This example uses the data set expo_example.txt, shown here:

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

1. Create an array named expo, with one cell for every row in expo_example.txt

   ```
   CREATE ARRAY expo
   <s:string,p:string,val:double>
   [i=1:25,5,0]
   ```

2. Convert the file expo_example.txt to SciDB format. You will need to exit your iquery session or do this in a new terminal window because the csv2scidb tool is run at the command line.

```
csv2scidb -p SSN -s N < expo_example.txt > expo_example.scidb
```

3. Load the data expo_example.txt into expo:

```
LOAD expo FROM '/doc/examples/expo_example.txt';
```

4. Create an array with a noninteger dimension to be the redimension target:

```
CREATE ARRAY Dsp
<val:double>
[s(string)=5,5,0, p(string)=5,5,0];
```

5. Redimension expo into Dsp:

```
redimension(expo, Dsp);
```

   This query returns:

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

Name

redimension_store — Transform attributes to dimensions

Synopsis

```
AFL% redimension_store(source_array,target_array);
```

Summary

redimension_store converts array attributes to dimensions. The redimension_store operator updates the target_array and creates additional mapping arrays if necessary.

You can redimension the array and apply aggregates to duplicate cells.

The input and target arrays must have compatible schemas, and both commands determine the list of transformations (attribute to dimension) by matching names in the attribute and dimension lists of the two arrays.

Example

This example redimensions a 2-attribute, 1-dimensional array into a 2-dimensional, 1-attribute array. This example uses the data set expo_example.txt, shown here:

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

1. Create an array named expo, with one cell for every row in expo_example.txt

   ```
   CREATE ARRAY expo
   ```

```
<s:string,p:string,val:double>
[i=1:25,5,0]
```

2. Convert the file expo_example.txt to SciDB format. You will need to exit your iquery session or do this in a new terminal window because the csv2scidb tool is run at the command line.

   ```
   csv2scidb -p SSN -s N < expo_example.txt > expo_example.scidb
   ```

3. Load the data expo_example.txt into expo:

   ```
   LOAD expo FROM '/doc/examples/expo_example.txt';
   ```

4. Create an array with a noninteger dimension to be the redimension target:

   ```
   CREATE ARRAY Dsp
   <val:double>
   [s(string)=5,5,0, p(string)=5,5,0];
   ```

5. Redimension expo and store the result in Dsp:

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

   This query returns:

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

Name

reduce_distro — Reduce the distribution of a replicated array

Synopsis

```
AFL% reduce_distro( array, partitioning_schema: integer )
```

Summary

Internal only.

Name

regrid — Select nonoverlapping subarrays

Synopsis

```
SELECT * FROM regrid(array,grid_1, grid_2,...,grid_N,
aggregate_call_1 [, aggregate_call_2,...,aggregate_call_N])
```

Summary

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

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

Example

This example divides a 4×4 array into 4 equal partitions and calculates the average of each one. This process is known as *spatial averaging*.

1. Create an array m4x4:

   ```
   CREATE ARRAY m4x4 <val:double> [i=0:3,4,0,j=0:3,4,0];
   ```

2.
   ```
   store(build (m4x4, i*4+j), m4x4);
   ```

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

3. Regrid m4x4 into four partitions and find the average of each partition.

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

   This query outputs:

   ```
   [[(2.5),(4.5)],[(10.5),(12.5)]]
   ```

Name

remove — Remove an array from the SciDB namespace

Synopsis

```
AFL% remove(array);
```

Summary

The AFL remove operator works like the AQL DROP ARRAY statement.

Example

Name

rename — Change array name

Synopsis

```
AFL% rename(array,new_array)
```

Summary

The AFL rename operator work similarly to the AQL statement SELECT * INTO except that the old array name can be reused immediately with the rename operator. The rename operator is akin to using the Unix mv command, whereas SELECT * INTO is akin to the Unix cp command.

Example

Name

repart — Change array chunk sizes

Synopsis

```
SELECT
```

Summary

Example

Name

reshape — Change dimension sizes and array shape

Synopsis

```
SELECT * FROM reshape(array,array|array_schema);
```

Summary

The reshape operator changes the shape of an array to the rank and dimensions of a given array or a given array schema. The the reshape command inputs must have the same number of total cells and cell attributes.

Example

This example reshapes a 4×4 array into a 2×8 array.

1. Create an array called m4x4:

   ```
   CREATE ARRAY m4X4 <val:double>[i=0:3,4,0,j=0:3,4,0];
   ```

2. Store values of 0–15 in m4x4:

   ```
   store(build(m4x4,i*4+j),m4x4);
   ```

   This query outputs:

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

3. Reshape m4x4 as 2-by-8:

   ```
   reshape(m4x4,<val:double>[i=0:7,8,0,j=0:1,2,0]);
   ```

   This query returns:

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

Name

reverse — Reverse values in each array dimension

Synopsis

```
SELECT * INTO target_array FROM source_array;
```

Summary

The reverse operator reverses all the values of each dimension in an array.

Example

This example reverses a 3×3 matrix.

1. Create a 3×3 array m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Reverse the values in m3x3:

   ```
   reverse(m3x3);
   ```

   This query outputs:

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

Name

sample — Select random array chunks

Synopsis

```
SELECT * FROM sample(array,probability);
```

Summary

Example

This example selects random chunks from a 1-dimensional 3-chunk array.

1. Create a 1-dimensional array with dimension size of 6 and chunk size of 2:

   ```
   CREATE ARRAY vector1<val:double>[i=0:5,2,0];
   ```

2. Put values of 0–5 into vector1:

   ```
   store(build(vector1,i),vector1);
   ```

3. Sample chunks from the array with the probability of 1/3 that a chunk is included>

   ```
   sample(vector1,.3);
   ```

Name

save — Save array data to a file

Synopsis

```
AFL% save(array,filepath)
```

Summary

The AFL save operator works like the AQL SAVE clause. It saves the data from the cells of a SciDB array into a file.

Example

This example creates a a matrix with two attributes and saves the cell values to a file.

1. Create a 2-dimensional array containing values 100–108:

   ```
   store(build(<val:double>[i=0:2,3,0,j=0:2,3,0],i*3+j+100),array1);
   ```

2. Create a 2-dimensional array containing values 200–208:

   ```
   store(build(<val:double>[i=0:2,3,0,j=0:2,3,0],i*3+j+200),array2);
   ```

3. Join array1 and array2 and store the output in an array storage_array:

   ```
   store(join(array1,array2),storage_array);
   ```

   This query outputs:

   ```
   [
   [(100,200),(101,201),(102,202)],
   [(103,203),(104,204),(105,205)],
   [(106,206),(107,207),(108,208)]
   ]
   ```

4. Save the contents of storage_array to a file.

   ```
   save(storage_array,'/tmp/storage_array.txt');
   ```

   The contents of storage_array.txt are:

   ```
   {0,0}
   [
   [(100,200),(101,201),(102,202)],
   [(103,203),(104,204),(105,205)],
   [(106,206),(107,207),(108,208)]
   ]
   ```

Name

scan — Display cell values

Synopsis

```
SELECT * FROM scan(array);
```

Summary

The scan operator displays to contents of each cell in an array. You can use scan with a WHERE clause to view subsets of large arrays. The output of the scan operator is an array the same size as the input operator. To supress display of empty cells, set the iquery -o sparse option.

Example

This example selects the second row from an array and shows the cell values in that row.

1. Create a 3×3 array m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

3. Use scan in a FROM clause to display the middle row of m3x3:

   ```
   SELECT val FROM scan(m3x3) WHERE i=1;
   ```

   This query outputs:

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

4. You can supress the empty cells in the output by setting the iquery output to sparse:

   ```
   quit;
   iquery -o sparse
   AQL% SELECT val FROM scan(m3x3) WHERE i=1;
   {1,0}[[{1,0}(3),{1,1}(4),{1,2}(5)]]
   ```

Name

setopt — Set/get configuration option value at runtime.

Synopsis

```
setopt ( option-name [ , new-option-value ] )
```

This operator is designed for internal use.

Summary

Set/get configuration option value at runtime. Option value should be specified as string. If new value is not specified, then values of this configuration option at all instances are printed. If new value is specified, then value of option is updated at all instances and result array contains old and new values of the option at all instances.

Name

show — Show array schema

Synopsis

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

Summary

The show operator returns an array's schema. This is useful if you are changing array dimensions with nested statements.

Example

Show the schema that results from several nested operations:

```
store(subarray(build(
 <val:double>[i=0:2,3,0,j=0:3,4,0,k=0:4,5,0],
 j+k),1,1,2,2,3,3),output_array);
show(output_array);
```

The schema of output_array is:

```
[("output_array<val:double> [i=0:1,3,0,j=0:2,4,0,k=0:1,5,0]")]
```

Name

slice — Select subset of array along a plane

Synopsis

```
SELECT * FROM slice(array,dimension1,index1[dimension2,index2,...]);
```

Summary

The slice operator takes a sample of cells along a specified plane of an array. The result is a slice of the input array corresponding to the given coordinate value(s). Number of dimensions of the result array is equal to the number of dimensions of input array minus number of specified dimension, and the coordinate value should be a valid dimension value of the input array.

Example

This example selects the middle column from a 3×3 array.

1. Create a 3×3 array m3x3:

   ```
   CREATE ARRAY m3x3<val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Select the middle column of m3x3:

   ```
   slice(m3x3,j,1);
   ```

   This query outputs:

   ```
   [(1),(4),(7)]
   ```

Name

sort — Sort by attribute value

Synopsis

```
SELECT * FROM sort(array,attribute,option);
```

Summary

Sort a one-dimensional array by one or more attributes. The sort attributes are specified using a 1-based attribute number. The default is ascending order. Set the option argument to desc to sort in descending order.

Example

Sort a set of random values from lowest to highest:

```
sort(build(<val:double>[i=0:9,10,0],random()%10),val);
```

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

Name

stdev — Standard deviation

Synopsis

```
SELECT * FROM stdev(array,attribute,dimension1,dimension2,...)
```

Summary

Note

The stdev operator provides the same functionality as the stdev aggregate. See the stdev aggregate reference page for more information.

Example

This example finds the standard deviation of each row of a 2-dimensional array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

   ```
   CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Store values of random values between 0 and 1 in m3x3:

   ```
   store(build(m3x3,random()%9/10.0),m3x3);
   ```

   ```
   [
   [(0.5),(0.6),(0)],
   [(0.8),(0.8),(0.4)],
   [(0.1),(0.8),(0.6)]
   ]
   ```

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

   ```
   var(m3x3,val,i);
   ```

   This query returns:

   ```
   [(0.321455),(0.23094),(0.360555)]
   ```

Name

store — Store query output in a SciDB array

Synopsis

```
store(operator(operator_args),target_array);
```

Summary

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

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

Note

The AFL store operator provides the same functionality as the AQL SELECT * INTO ... FROM ... statement.

Example

Build and store a 2-dimensional, 1-attribute matrix of zeros:

```
store(build(<val_double>[i=0:2,3,0,j=0:2,3,0],0),zeros_array);
```

You can change the name of the array zeros_array to ones_array and the cell values to 1 with a store statement:

```
store(build(zeros_array,1),ones_array);
```

Build and store a 2-dimensional, 1-attribute matrix of random numbers between 1 and 10:

```
store(build(random_array,random()%10),random_array);
```

```
[
[(6),(8),(3)],
[(6),(5),(1)],
[(6),(1),(3)]
]
```

You can update the array with a different set of random numbers by re-running the store statement:

```
store(build(random_array,random()%10),random_array);
```

```
[
[(4),(5),(6)],
[(5),(4),(6)],
[(8),(4),(2)]
]
```

Name

subarray — Select contiguous area of cells

Synopsis

```
SELECT * FROM subarray(array,boundary_coord_1,boundary_coord_2,...)
```

Summary

Subarray selects a block of cells from an input array. The result is an array whose shape is defined by the boundary coordinates specified by the subarray arguments. A boundary coordinate pair must be specified for every dimension of the input array.

Example

This example selects the values from the last two columns and the last two rows of a 4×4 matrix.

1. Create an array called m4x4:

   ```
   CREATE ARRAY m4X4 <val:double>[i=0:3,4,0,j=0:3,4,0];
   ```

2. Store values of 0–15 in m4x4:

   ```
   store(build(m4x4,i*4+j),m4x4);
   ```

   This query outputs:

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

3. Return an array containing the cells that were in both the last two columns and the last two rows on m4x4:

   ```
   subarray(m4x4,2,2,3,3);
   ```

   This query returns:

   ```
   [
   [(10),(11)],
   [(14),(15)]
   ]
   ```

Name

substitute — Substitute new value for null values in an array

Synopsis

```
SELECT * FROM substitute(null_array,substitute_array);
```

Summary

Substitute null values in one array with non-null values from another array. The arrays must have the same dimension start index.

Example

This example replaces all null values in an array with zero.

1. Create an array m4x4_null with a nullable attribute:

   ```
   CREATE ARRAY m4x4_null <val:double null>[i=0:3,4,0,j=0:3,4,0];
   ```

2. Store null in the second row of m4x4_null and 100 in all the other cells:

   ```
   store(build(m4x4_null,iif(i=1,null,100)),m4x4_null);
   ```

3. Create a single-cell array called substitute_array

   ```
   CREATE ARRAY substitute_array <missing:double>[i=0:0,1,0];
   ```

4. Put value 0 into substitute_array:

   ```
   store(build(substitute_array,0),substitute_array);
   ```

5. Use the substitute operator to replace the null-valued cells in m4x4_null with 0-valued cells:

   ```
   substitute(m4x4_null,substitute_array);
   ```

   This query outputs:

   ```
   [
   [(100),(100),(100),(100)],
   [(0),(0),(0),(0)],
   [(100),(100),(100),(100)],
   [(100),(100),(100),(100)]
   ]
   ```

Limitation

The substitute operator can only take single-attribute arrays as inputs.

Name

sum — Sum attribute values

Synopsis

```
SELECT * FROM sum(array,attribute[,dimension])
```

Summary

Note

The sum operator offers the same functionality as the sum aggregate. See sum aggregate reference page for more information.

Example

This example sums the columns and rows of a 3×3 array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

```
CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
```

2. Store values of 0–8 in m3x3:

```
store(build(m3x3,i*3+j),m3x3);
```

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

3. Sum the values of m3x3 along dimension j. This sums the columns of m3x3:

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

This query outputs:

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

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

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

This query outputs:

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

Name

thin — Select data from an array dimension at fixed intervals

Synopsis

```
SELECT * FROM thin(array,start_1,step_1,start_2,step_2,...);
```

Summary

The thin operator selects regularly spaced elements of the array in each dimension. The selection criteria are specified by the starting dimension value `start_1` and the number of cells to skip using `step_1` for each dimension of the input array. The dimension chunk size must be evenly divisible by the step size.

Example

This example selects every

1. Create an array m6x6:

   ```
   CREATE ARRAY m6x6 <val:double>[i=0:5,6,0,j=0:5,6,0];
   ```

2. Put values of 1–35 into m6x6:

   ```
   store(build(m6x6,i*6+j),m6x6);
   ```

   ```
   [
   [(0),(1),(2),(3),(4),(5)],
   [(6),(7),(8),(9),(10),(11)],
   [(12),(13),(14),(15),(16),(17)],
   [(18),(19),(20),(21),(22),(23)],
   [(24),(25),(26),(27),(28),(29)],
   [(30),(31),(32),(33),(34),(35)]
   ]
   ```

3. Select every other column of m6x6, starting at the first column;

   ```
   thin(m6x6,0,1,0,2);
   ```

   This query outputs:

   ```
   [
   [(0),(2),(4)],
   [(6),(8),(10)],
   [(12),(14),(16)],
   [(18),(20),(22)],
   [(24),(26),(28)],
   [(30),(32),(34)]]
   ```

4. Select every other row from m6x6, starting at the first row;

   ```
   thin(m6x6,0,1,0,2);
   ```

   This query outputs:

   ```
   [
   ```

```
[(0),(1),(2),(3),(4),(5)],
[(12),(13),(14),(15),(16),(17)],
[(24),(25),(26),(27),(28),(29)]
]
```

5. Select every other value from m6x6, starting at the second column;

   ```
   thin(m6x6,1,2,1,2);
   ```

   This query outputs:

   ```
   [
   [(7),(9),(11)],
   [(19),(21),(23)],
   [(31),(33),(35)]
   ]
   ```

Name

transpose — Matrix transpose

Synopsis

```
SELECT * FROM transpose(array)
```

Summary

The transpose operator accepts an array which may contain any number of attributes and dimensions. Attributes may be of any type. If the array contains dimensions d1, d2, d3, ..., dn the result contains the dimensions in reverse order dn, ..., d3, d2, d1.

Example

This example transposes a 3×3 matrix.

1. Create a 1-attribute, 2-dimensional array called m3x3:

   ```
   CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Store values of 0–8 in m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Transpose m3x3:

   ```
   transpose(m3x3);
   ```

   This query outputs:

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

Name

unload_library — Unload a plugin

Synopsis

```
unload_library('library_name')
```

Summary

Unload a plug-in from the current SciDB instance.

Note

The unload_library operator provides the same functionality as the AQL UNLOAD LIBRARY '*library_name*' statement.

Example

This example loads and unloads the example plug-in librational.so.

```
load_library('librational')
unload_library ('librational')
```

The file extension is not included in the library name.

Name

unpack — Transform multidimensional array to single dimension

Synopsis

```
SELECT * INTO vector FROM unpack(array,attribute_name)
```

Summary

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

Example

This example takes 2-dimensional, 1-attribute array and outputs a 1-dimensional, 3-attribute array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

   ```
   CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Store values of 0–8 in m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

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

3. Create a new attribute called val2 containing values 100–108 and store the resulting array as m3x3_2attr:

   ```
   store(apply(m3x3,val2,val+100),m3x3_2attr);
   ```

   This query outputs:

   ```
   [
   [(0,100),(1,101),(2,102)],
   [(3,103),(4,104),(5,105)],
   [(6,106),(7,107),(8,108)]
   ]
   ```

4. Unpack m3x3_2attr into a 1-dimensional array.

   This query outputs:

   ```
   [
   (0,0,0,100),
   (0,1,1,101),
   (0,2,2,102),
   ```

```
(1,0,3,103),
(1,1,4,104),
(1,2,5,105),
(2,0,6,106),
(2,1,7,107),
(2,2,8,108)
]
```

The first two values in each cell are the dimensional indices, and the second two are the attribute values.

Name

var — Variance

Synopsis

```
SELECT * FROM var(array,attribute)
```

Summary

The var operator returns the variance of a set of values taken from an array attribute.

Note

The var operator provides the same functionality as the var aggregate. See the var aggregate reference page for more information.

Example

This example finds the variance of each row of a 2-dimensional array.

1. Create a 1-attribute, 2-dimensional array called m3x3:

   ```
   CREATE ARRAY m3x3 <val:double>[i=0:2,3,0,j=0:2,3,0];
   ```

2. Store values of random values between 0 and 1 in m3x3:

   ```
   store(build(m3x3,random()%9/10.0),m3x3);
   ```

   ```
   [
   [(0.5),(0.6),(0)],
   [(0.8),(0.8),(0.4)],
   [(0.1),(0.8),(0.6)]
   ]
   ```

3. Select the variance of each row of m3x3:

   ```
   var(m3x3,val,i);
   ```

   This query returns:

   ```
   [(0.103333),(0.0533333),(0.13)]
   ```

Name

versions — Show array versions

Synopsis

```
SELECT * FROM versions(array);
```

Summary

The versions operator lists all versions of an array in the SciDB namespace. The output of the versions command is a list of versions, each of which has a version ID and a datestamp which is the date and time of creation of that version.

Example

This example creates an array, updates it twice, and then returns the first version of the array.

1. Create an array called m1:

   ```
   CREATE ARRAY m1 <val:double>[i=0:9,10,0];
   ```

2. Store 1 in each cell of m1:

   ```
   store(build(m1,1),m1);
   ```

3. Update every cell to have value 100:

   ```
   store(build(m1,100),m1);
   ```

4. Use the versions command to see the two versions of m1 that you created:

   ```
   versions(m1);
   ```

5. Use the scan operator and the '@1' array name extension to display the first version of m1.

   ```
   scan(m1@1);
   ```

   This query outputs:

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

Name

window — Compute aggregates over moving window

Synopsis

```
SELECT * FROM window(array,grid_1,grid_2,...,grid_N,aggregate_call_1[,aggrgegate_c
```

Summary

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

Note

The AFL window operator provides the same functionality as the AQL SELECT ... FROM ... WINDOW statement. See the User's Guide chapter on Aggregates for more information.

Example

This example calculates a running sum for a 3×3 window on a 4×4 array.

1. Create an array called m4x4:

   ```
   CREATE ARRAY m4X4 <val:double>[i=0:3,4,0,j=0:3,4,0];
   ```

2. Store values of 0–15 in m4x4:

   ```
   store(build(m4x4,i*4+j),m4x4);
   ```

   This query outputs:

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

3. Return the maximum and minimum values on a moving 3×3 window on m4x4:

   ```
   window(m4x4,3,3,max(val),min(val));
   ```

   This query returns:

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

Name

xgrid — Expand single array element to grid

Synopsis

```
SELECT * FROM xgrid(array,scale_1[,scale_2,..., scale_N])
```

Summary

The xgrid operators scales an input array by repeating cells of the original array specified number of times in a contiguous subregion. xgrid takes one *scale* argument for every dimension in *array*. The output array has the same number of dimensions and attributes as the input array.

Example

This example scales each cell of a 2-dimensional array into a 2×2 subarray.

1. Create an array called m3x3:

   ```
   CREATE ARRAY m3x3 <val:double> [i=0:2,3,0,j=0:2,3,0];
   ```

2. Put values of 0–8 into m3x3:

   ```
   store(build(m3x3,i*3+j),m3x3);
   ```

3. Expand each cell of m3x3 into a 2×2 subgrid. Store the resulting array as m6x6:

   ```
   store(xgrid(m3x3,2,2),m6x6);
   ```

   This query returns:

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