SciDB and R

paradigm4

data-driven discovery

Contents

1 Introduction

SciDB is an open-source database that organizes data in n-dimensional arrays. SciDB features include ACID transactions, parallel processing, distributed storage, efficient sparse array storage, and native linear algebra operations. The `scidb` package for R provides two ways to interact with SciDB from R:

1. By running SciDB queries from R, optionally transferring data through data frames or data frame iterators.
2. Through several SciDB array object classes for R. The arrays mimic standard R arrays and data frames, but operations on them are performed by the SciDB engine. Data are materialized to R only when requested.

In some cases, R scripts and packages may be used with little or no modification with `scidb` arrays, allowing SciDB to power large-scale parallel R computation. This vignette illustrates using SciDB from R by example. For more detailed information on the functions described in this vignette, see the manual pages in the package.

2 Connecting to SciDB and Running Queries

This section outlines the most basic interaction between R and SciDB: running queries and transferring one-dimensional SciDB arrays between R and SciDB through R data frames.

2.1 Connecting to SciDB

The `scidbconnect` function establishes a connection to a simple HTTP network service called shim running on a SciDB coordinator instance (see Section 9.3). The function may be safely called multiple times. Once a connection is established, connection information is maintained until a different connection is established or the R session ends.

The network interface optionally supports TLS encryption and user authentication. Users are defined by the operating system users on the SciDB coordinator instance.

Connect to localhost by default on unencrypted port 8080:

```
> library("scidb")
> scidbconnect()
```

Connect to SciDB on an encrypted port 8083 with example authentication:

```
> scidbconnect(host="localhost", port=8083, username="scidbuser", password="test")
```

The shim service can be configured to support either open/unencrypted or encrypted/authenticated ports, or both. We recommend using only encrypted/authenticated sessions when communicating with SciDB over public networks.

2.2 Listing and removing SciDB arrays

The `scidblist` function lists SciDB objects (arrays, instances, etc.), optionally showing detailed schema information for arrays. Returned results may be filtered using regular expression-style syntax.

The `scidbremove` function removes a SciDB array, or optionally a set of SciDB arrays defined by regular expression. The function accepts a vector of array names, resulting in the removal of all the specified arrays. Combine this feature with the regular expression filtering output of `scidblist` to remove sets of arrays matching the filter.

2.3 Running SciDB queries

The `iquery` function executes SciDB queries using either the SciDB array functional language (AFL) or declarative array query language (AQL) syntax. When AFL is used, the `iquery` function optionally returns query results in an R data frame if the argument `return=TRUE` is specified. Returned output is similar to output obtained by the SciDB `iquery` command-line program with the `-olcsv+` option. The `iquery` function does not return anything by default.

Query results returned by the `iquery` function are internally presented to R using a generic CSV format, providing very flexible support for many data types. (The n-dimensional array class described in the next section uses a binary data exchange method between R and SciDB.) Note that, although R and SciDB have a number of common data types, each system contains types not supported by the other. Thus, conversion errors may arise. The `iquery` function is designed to reasonably minimize such issues and simplify basic data transfer between the systems. Data types common to R and SciDB include double-precision numeric, character string, logical, and 32-bit integers. They `iquery` function supports standard R `read.table` parameter options to facilitate type conversion.

The following code example illustrates basic use of `iquery`.

```
> library("scidb")
> scidbconnect()
> # Manuall build a 1-d SciDB array named "P:"
> iquery("store(build(<x:double>[i=0:99,100,0],asin(1)*i/25),P)")
> # Return a query as a data.frame, manually specifying returned column classes:
> S <- iquery("apply(P,y,sin(x))",return=TRUE,
+              colClasses=c("integer","double","complex"))
> head(S, n=3)

  i          x              y
1 0 0.00000000 0.00000000+0i
2 1 0.06283185 0.06279052+0i
3 2 0.12566371 0.12533323+0i
```

2.4 Iterating over query results

The `iquery` function returns query results into a single R data frame by default. Large results expected to contain lots of rows may be iterated over by setting the `iterative=TRUE` argument. When `iquery` is used with the `iterative=TRUE` setting, it returns an iterator that iterates over chunks of rows of the result data frame. Iterators are defined by the `iterators` package. Their data may be directly accessed with the `nextElem` method, or indirectly with `foreach`. See the `iterators` and `foreach` packages for many examples and further documentation of their use.

```
> # Build a small 1-d SciDB test array:
> iquery("store(build(<x:double>[i=1:10,10,0],i/10.0),A)")
> # Return the result of a SciDB apply operator in an R iterator with a
> # chunk size of at most 7 rows at a time:
> it <- iquery("apply(A,y,sqrt(x))", return=TRUE, iterative=TRUE, n=7)
> nextElem(it)

  i   x         y
1 1 0.1 0.3162278
2 2 0.2 0.4472136
3 3 0.3 0.5477226
4 4 0.4 0.6324555
5 5 0.5 0.7071068
6 6 0.6 0.7745967

> nextElem(it)

   i   x         y
1  7 0.7 0.8366600
2  8 0.8 0.8944272
3  9 0.9 0.9486833
4 10 1.0 1.0000000
```

3 SciDB Arrays for R Users

Data are organized by SciDB in n-dimensional sparse arrays. "Sparse" in SciDB arrays means that array elements may be left undefined, and such array elements are omitted from computations. Note that this interpretation of sparse differs in a subtle way from that used by sparse matrices defined by R's Matrix package (whose sparse matrix elements are implicitly zero).

The elements of a SciDB array, called *cells*, contain one or more *attributes* (similar to R variables). The number and data types of attributes are uniform across all cells in an array, and SciDB stores data for each attribute separately. Thus, a one-dimensional SciDB array is conceptually very similar to a data frame in R: the SciDB dimension index corresponds to data frame row index, and SciDB attributes to data frame columns. Higher-dimensional arrays in SciDB don't correspond directly to objects in R; the `scidb` n-dimensional array class described below is limited to working with one attribute at a time to more closely emulate standard R arrays.

The integer coordinate systems used to index SciDB arrays are similar to R, except that SciDB integer indices may be zero or negative, and are represented by 63-bit signed integers (R indices use either unsigned positive 31-bit integer or positive 52-bit integer-valued double values). This means that SciDB arrays can't always be indexed by R–in practice R users should avoid creating SciDB arrays with coordinate systems that extend past R's limits.

SciDB attribute values within a cell may be explicitly marked missing, indicated by a special SciDB missing code also referred to as a NULL code. SciDB internally supports a large number of possible missing codes. All SciDB missing code values are mapped to `NA` values in R.

The `scidb` package defines two array classes for R with data backed by SciDB arrays: the `scidbdf` class for data frame-like objects, and the `scidb` class for matrix and general array objects.

4 The `scidbdf` data frame-like class

The `scidbdf` class defines a data frame-like class backed by one-dimensional SciDB arrays. Like data frames, the columns represent variables of distinct types and the rows represent observations. Each attribute in the backing SciDB array represents a column in the `scidbdf` object. The `scidbdf` object elements are read-only (the backing SciDB array may be manually updated, for example using the iquery function).

Use the `as.scidb` function to create new SciDB arrays and corresponding `scidbdf` R objects by copying R data frames into SciDB. The `types` and `nullable` options may be used to explicitly specify the SciDB type and nullability values of each data frame column. See the R help page for `df2scidb` for more information.

The `scidb` function returns an R `scidbdf` or `scidb` object representation of an existing SciDB array.

Objects of class `scidbdf` obey a subset of R indexing operations. Columns may be selected by numeric index or attribute name, but the short-hand R $-style variable selection notation is not supported. Rows may only be selected by integer.

Subsets of `scidbdf` objects are returned as new `scidbdf` objects of the appropriate size (dimension, number of attributes/columns). The package uses the special empty-bracket notation, `[]`, to indicate that data should be materialized to R as an R data frame. Illustrations are provided in the examples.

4.1 Examples of `scidbdf` objects

```
> library("scidb")
> scidbconnect()
> # Copy the Michelson-Morley experiment data to SciDB,
> # returning a scidbdf object
> X <- as.scidb(morley, name="morely")
> str(X)


SciDB array name:  morely
SciDB array schema:  array<Expt:int32,Run:int32,Speed:int32> [row=1:100,100,0]
Attributes:
  attribute  type nullable
1      Expt int32    FALSE
2       Run int32    FALSE
3     Speed int32    FALSE
Row dimension:
  name  type start length chunk_interval chunk_overlap low high
1  row int64     1    100            100             0  NA   NA

> # Materialize the first four rows of X to R (using [] to return results to R):
> X[1:4, c("Run","Speed")][]


  Run Speed
1   1   850
2   2   740
3   3   900
4   4  1070
```

5 The scidb array class

Similarly to the data frame-like class, the package defines the scidb array class for R that represents vectors, matrices and general n-dimensional arrays. Array objects defined by the scidb class behave in some ways like standard R arrays. But their data reside in SciDB and most operations on them are computed by SciDB.

The scidb array class supports working with a single array attribute at a time to conform to R arrays (which generally support a single value per cell). Consider the iris data, represented as a data frame within SciDB as a 1-d array with five attributes. The next example illustrates creating a scidbdf data frame-like object in R referring to the iris data and also a 1-d scidb array object in R referring to the same data using the Petal_Width attribute. Note that, unlike R, SciDB numeric array indices may be zero or negative and generally use zero as the default origin. Data from scidb array objects are not materialized to R until extracted with the empty indexing function, [].

```
> # Upload the iris data to SciDB storing it in an array named `iris`
> # and create a data frame-like object `df` that refers to it:
> df <- as.scidb(iris,name="iris")
> dim(df)

[1] 150   5

> # Now create a vector-like object using the same data in SciDB, but
> # referring to only one of the available variables (Petal_Width):
> x <- scidb("iris", attribute="Petal_Width", data.frame=FALSE)
> length(x)

[1] 150

> x[99:103][]

[1] 1.1 1.3 2.5 1.9 2.1

> # List all available attributes in the SciDB array:
> x@attributes

[1] "Sepal_Length" "Sepal_Width"  "Petal_Length" "Petal_Width"  "Species"
```

5.1 Subsetting and indexing scidb array objects

SciDB arrays act in many ways like regular arrays in R. Rectilinear subarrays may be defined by ranges of integer indices. Subarrays of `scidb` array objects are returned as new `scidb` array objects of the appropriate size.

Despite the similarities, there are differences between regular R and `scidb` array object indexing. In particular:

- The empty indexing function, `[]` applied to a `scidb` object materializes its array data as an R array. If the data exceed a return size threshold, an iterator over the array indices and data will be returned instead. The package option `options("scidb.max.array.elements")` controls the threshold.
- Index ranges follow SciDB convention. Arrays may have non-positive integer indices. In particular, note that the starting SciDB integer index is arbitrary, but often zero. (By contrast, the upper left corner of R arrays is always indexed by [1,1,...].)
- `scidb` array objects are limited to double-precision and 32-bit signed integer numeric, logical, and single-byte character (char) element data types.

The following example illustrates basic subsetting operations on a sparse matrix. Note that the default SciDB index is zero-based.

```
> scidbremove("A", error=invisible)
> # Manually create an example sparse 2-d array in SciDB:
> iquery("store(build_sparse(<val:double>[i=0:9,10,0,j=0:9,5,0],i*j,i<=j),A)")
> A <- scidb("A")
> # Indexing operations return new SciDB array objects:
> A[0:5,2:4]

A reference to a  6x3 SciDB array

> # But their data can be materialized into an R array with []:
> A[0:5,2:4][]

6 x 3 sparse Matrix of class "dgCMatrix"

[1,] 0 0  0
[2,] 2 3  4
[3,] 4 6  8
[4,] . 9 12
[5,] . . 16
[6,] . .  .
```

5.2 Arithmetic operations

The `scidb` array class supports a few standard linear algebra operations for dense and sparse matrices and vectors.

The example below shows a dense matrix example that compares matrix arithmetic in R and SciDB. Like previous data frame examples, use `as.scidb` to export R matrices and vectors to SciDB arrays.

We generally use `as.scidb` for convenience–it's far from the most efficient way to import data into SciDB. For very large data, use the SciDB bulk data load utility as outlined in the SciDB documentation instead.

```
> v <- as.scidb(matrix(rnorm(25),5))
> str(v)

SciDB array name:  R_array1d1815f040541...
SciDB array schema:  array<val:double> [i=0:4,5,0,j=0:4,5,0]
attribute in use:  val
All attributes:  val
Array dimensions:
  name  type start length chunk_interval chunk_overlap low high
1    i int64     0      5              5             0  NA   NA
2    j int64     0      5              5             0  NA   NA

> crossprod( v[] )              # Compute t(v) %*% v using R

           [,1]      [,2]       [,3]       [,4]       [,5]
[1,]  1.9828463  2.907710  0.4369621  0.1687685 -0.3393235
[2,]  2.9077099  7.994399  1.2859649  3.5795483 -3.2595679
[3,]  0.4369621  1.285965  4.4088661 -1.0451336  0.5036469
[4,]  0.1687685  3.579548 -1.0451336  5.5147411 -3.3275643
[5,] -0.3393235 -3.259568  0.5036469 -3.3275643  2.6672657

> # Now compute using SciDB, and materialize the result to R:
> crossprod(v)[]

           [,1]      [,2]       [,3]       [,4]       [,5]
[1,]  1.9828463  2.907710  0.4369621  0.1687685 -0.3393235
[2,]  2.9077099  7.994399  1.2859649  3.5795483 -3.2595679
[3,]  0.4369621  1.285965  4.4088661 -1.0451336  0.5036469
[4,]  0.1687685  3.579548 -1.0451336  5.5147411 -3.3275643
[5,] -0.3393235 -3.259568  0.5036469 -3.3275643  2.6672657
```

Basic matrix/vector arithmetic operations on SciDB arrays (addition, subtraction, matrix and matrix vector products, scalar products, `crossprod` and `tcrossprod`) use standard R syntax. You can mix R and SciDB matrices and vectors and the `scidb` package will try to do the right thing by assigning R data to temporary SciDB arrays conforming to required database schema. The next example shows an example of computations that mix `scidb` array objects with R vectors.

```
> # Build a 5x5 SciDB matrix, assigned to R variable 'A':
> A <- build("random() % 10", dim=c(5,5))
> # Compute a matrix-vector product against an R-generated vector (in SciDB):
> y <- A %*% rnorm(5)
> # Return the computed result to R
> y[]
```

```
          [,1]
[1,]  1.415892
[2,] -7.698924
[3,] -7.662604
[4,] -8.685752
[5,] -4.907824
```

Expression	Operation	Operands	Output
A `%*%` B	Matrix multiplication	A, B Conformable SciDB arrays or R matrices/vectors	SciDB array
$A \pm B$	Matrix summation/difference	A, B SciDB arrays or R matrices/vectors	SciDB array
`crossprod(`A,B`)`	Cross product `t(A) %*% B`	A, B SciDB arrays or R matrices/vectors	SciDB array
`tcrossprod(`A,B`)`	Cross product `A %*% t(B)`	A, B SciDB arrays or R matrices/vectors	SciDB array
A `*/` B	Elementwise product/quotient	A, B Conformable SciDB arrays or R matrices/vectors	SciDB array
α `*/` A	Scalar multiplication/division	SciDB array A, scalar α	SciDB array
`t(`A`)`	Transpose	SciDB array A	SciDB array
`sin(`A`)`	Sine function, also other trig functions	SciDB array A	SciDB array
`log(`$A, base$`)`	Logarithm function	SciDB array A, numeric base	SciDB array
`diff(`$A, lag = 1$`)`	Finite differences	SciDB array A, integer lag	SciDB array
$A[\text{range}, \text{range}, \ldots]$	Subarray	SciDB array A	SciDB Array
$A[\,]$	Materialize	SciDB array A	R array
`svd(`A`)`	Singular value decomposition	Dense SciDB array A	SciDB arrays
`svd(`A, nu`)`	Truncated SVD	Sparse or dense SciDB array A	SciDB arrays

Table 1: SciDB Array Class Operations

5.3 Apply, sweep and other array functions

R's `apply` function applies a function along margins of an array or matrix. A verion of `apply` limited to SciDB aggregation functions is available for `scidb` and `scidbdf` objects. The SciDB aggregation function is supplied as a string and references the array attributes. Here is an example:

```
> # Create an example matrix and compute its column means:
> A <- matrix(as.double(1:20),nrow=5)
> apply(A,2,mean)

[1]  3  8 13 18

> # Make a copy of the matrix in SciDB and assign it to the scidb object X:
> X <- as.scidb(A)
> str(X)

SciDB array name:  R_array1d181050fd251...
SciDB array schema:  array<val:double> [i=0:4,5,0,j=0:3,4,0]
attribute in use:  val
All attributes:  val
Array dimensions:
  name  type start length chunk_interval chunk_overlap low high
1    i int64     0      5              5             0  NA   NA
2    j int64     0      4              4             0  NA   NA

> # Compute its column means, using a SciDB aggregation expression:
> m <- apply(X,2,"avg(val)")
> m[]

[1]  3  8 13 18
```

A version of R's `sweep` function works similarly. Next we subtract the column means from each column of the SciDB matrix in the last example.

```
> sweep(X,MARGIN=2,STATS=m)[]

     [,1] [,2] [,3] [,4]
[1,]   -2   -2   -2   -2
[2,]   -1   -1   -1   -1
[3,]    0    0    0    0
[4,]    1    1    1    1
[5,]    2    2    2    2
```

More general aggregation and data manipulation functions are outlined in the next section.

The `diff` function works similarly to R's usual one for computing finite differences along vectors and matrices.

Use the `cumulate` function to compute running operations along data, for example cumulative sums. The operation to be performed must be a valud SciDB aggregation function expressed as a character string. Here is a simple example:

```
> x <- as.scidb(iris)
> y <- cumulate(x, "sum(Petal_Width)")
> print(head(y, n=10))

 [1] 0.2 0.4 0.6 0.8 1.0 1.4 1.7 1.9 2.1 2.2
```

The `count` function applied to a `scidb` array object returns the count of non-empty cells in in the backing SciDB array.

`crossprod` and `tcrossprod` are defined for `scidb` array objects and mixtures of `scidb` and matrices.

The `image` function displays a heatmap of a regrid of a 2-d `scidb` array object, and returns the regridded array to R. The `grid=c(m,n)` function parameter specifies the regrid window sizes in each array dimension, and defaults to the array chunk sizes. The regrid aggregation function may be specified using the `op` function argument, and by default averages the array values over the regrid windows.

6 Data manipulation functions

The package defines a number of common SciDB data manipulation functions for SciDB array objects.

6.1 Conditional subset operations

The package supports array subset filter operations along dimension indices. Use the `subset` function to filter array contents by a boolean expression similarly to the standard R `subset` function. Under the hood, this function uses the SciDB `filter` operator–the function name `subset` more closely matches standard R syntax.

The `subset` function requires two arguments, a SciDB array reference and a valid SciDB logical expression represented as a string. Here is a simple example:

```
> df <- as.scidb(iris)
> subset(df,"Petal_Width > 2.4")[]
```

```
    Sepal_Length Sepal_Width Petal_Length Petal_Width   Species
101          6.3         3.3          6.0         2.5 virginica
110          7.2         3.6          6.1         2.5 virginica
145          6.7         3.3          5.7         2.5 virginica
```

6.2 Database joins

The package overloads the R `merge` function, enabling a number of database join-like operations on SciDB array objects. Use the R `help("merge",package="scidb")` function for detailed help. Here is an example that performs an inner join on array attributes:

```
> authors <- data.frame(
+             surname = c("Tukey", "Venables", "Tierney", "Ripley", "McNeil"),
+             nationality = c("US", "Australia", "US", "UK", "Australia"),
+             deceased = c("yes", rep("no", 4)),
+             stringsAsFactors=FALSE)
> books <- data.frame(
+           name = c("Tukey", "Venables", "Tierney",
+                    "Ripley", "Ripley", "McNeil", "R Core"),
+           title = c("Exploratory Data Analysis",
+                     "Modern Applied Statistics ...",
+                     "LISP-STAT", "Spatial Statistics", "Stochastic Simulation",
+                     "Interactive Data Analysis", "An Introduction to R"),
+           other.author = c(NA, "Ripley", NA, NA, NA, NA, "Venables & Smith"),
+           stringsAsFactors=FALSE)
> a <- as.scidb(authors)
> b <- as.scidb(books)
> merge(a,b,by=list("surname","name"))[,c(4:6,8:9)][]
```

```
   surname nationality deceased                         title other_author
0   McNeil   Australia       no     Interactive Data Analysis         <NA>
1   Ripley          UK       no            Spatial Statistics         <NA>
2   Ripley          UK       no         Stochastic Simulation         <NA>
3  Tierney          US       no                     LISP-STAT         <NA>
4    Tukey          US      yes     Exploratory Data Analysis         <NA>
5 Venables   Australia       no Modern Applied Statistics ...       Ripley
```

```
> # cf. The standard R data.frame merge function:
> # merge(authors,books,by.x="surname",by.y="name")
```

The `merge` implementation has some limitations outlined in its man page. Joins on attributes are presently the most limited cases, and most joins are limited to inner joins at the moment. Use `merge` on array dimensions for the most flexibility.

6.3 Aggregation

The `aggregate` function is overloaded to perform aggregation operations on SciDB array objects. Aggregation may be defined over array dimensions and/or array attributes very similarly to standard R aggregation syntax, except that aggregation functions must be valid SciDB aggregate expressions, represented as strings.

```
> # Upload the iris data to SciDB and create a data frame-like object
> # that refers to it:
> df <- as.scidb(iris)
> aggregate(df, by="Species", FUN="avg(Petal_Length), stdev(Petal_Width)")[]
```

```
  Species_index Petal_Length_avg Petal_Width_stdev    Species
0             0            1.462         0.1053856     setosa
1             1            4.260         0.1977527 versicolor
2             2            5.552         0.2746501  virginica
```

6.4 Binding new variables to an array

The SciDB package defines the `bind` function to add variables to arrays similar to the R `cbind` function for data frames. However, `bind` can also operate on higher-dimensional arrays. The example below adds a variable named ‘prod‘ to the SciDB array `df` defined in the last example.

```
> y  <- bind(df, "prod", "Petal_Length * Petal_Width")
> head(y, n=3)
```

```
  Sepal_Length Sepal_Width Petal_Length Petal_Width Species prod
1          5.1         3.5          1.4         0.2  setosa 0.28
2          4.9         3.0          1.4         0.2  setosa 0.28
3          4.7         3.2          1.3         0.2  setosa 0.26
```

6.5 Sorting, enumerating factors

Use the `sort` function to sort on a subset of dimensions and/or attribute of a SciDB array object, creating a new sorted array.

Use the `unique` function to return a SciDB array that removes duplicate elements of a single-attribute SciDB input array.

Use the `index_lookup` function along with `unique` to bind a new variable that enumerates unique values of a variable similarly to the R `factor` function.

Examples follow. Note that we use the SciDB `project` function in one example. `project` presents an alternative syntax to the functionally equivalent column subset selection of variables using brackets.

```
> x <- as.scidb(iris)
> # Sort x by Petal_Width and Species
> a <- sort(x, attributes=c("Petal_Width","Species"))
> head(a, n=3)

  Sepal_Length Sepal_Width Petal_Length Petal_Width Species
1          4.9         3.1          1.5         0.1  setosa
2          4.9         3.6          1.4         0.1  setosa
3          4.8         3.0          1.4         0.1  setosa

> # Find unique values of Species:
> unique(a[,"Species"])[]

     Species
0     setosa
1 versicolor
2  virginica

> # Add a new variable that enumerates factor levels of Species:
> head(index_lookup(a, unique(project(a,"Species")), "Species"))

  Sepal_Length Sepal_Width Petal_Length Petal_Width Species Species_index
1          4.9         3.1          1.5         0.1  setosa             0
2          4.9         3.6          1.4         0.1  setosa             0
3          4.8         3.0          1.4         0.1  setosa             0
4          4.3         3.0          1.1         0.1  setosa             0
5          5.2         4.1          1.5         0.1  setosa             0
6          4.9         3.0          1.4         0.2  setosa             0
```

7 Array promises and garbage collection

Most opreations on SciDB array objects return array promises–new SciDB array objects that have not been fully evaluated by SciDB yet, but that promise to return values when asked. SciDB array promises are essentially just SciDB query expressions together with a schema that the resulting output array will have once it's been evaluated. In such cases, the `name` slot of a `scidb` or `scidbdf` promise object shows the SciDB query expresion.

Occasionally, promise objects must be explicitly evaluated and stored as an intermediate result in SciDB. Such dynamically allocated arrays use a naming convention that begins with "R_array" and end with a unique numeric identifier determined by the current SciDB session.

Ephemeral intermediate arrays are connected to R's garbage collector and autotically deleted from the SciDB catalog when they are no longer needed by R. Users can disconnect SciDB array objects from R's garbage collector (making stored arrays persistent in SciDB), by setting `@gc$remove = FALSE`.

Sometimes efficiencies can be gained by explicitly evaluating a promise object and storing it to a SciDB array. The package provides the `scidbeval` function to let users decide to explicitly evaluate an expression. Many data manipulation functions also have an `eval` argument for the same purpose.

SciDB arrays keep references to other SciB array dependencies in a list within an environment, preventing them from automatic garbage collection so that the promise can be evaluated. The environment is slot is named `@gc` in the array objects and the dependency list is `@gc$depend`.

Here is an example:

```
> # Create a 5x4 SciDB matrix named 'test' in the database. We set `gc=TRUE`
> # so that the SciDB array `test` will be deleted by R's garbage collector
> # for us automatically:
> A <- as.scidb(matrix(rnorm(20),nrow=5), name="test", gc=TRUE)
> A@name

[1] "test"


> # Assign the third row of the transpose to a scidb object B:
> B <- t(A)[3,]
> # B is a promise object--it's name is an (unevaluated) SciDB query expression
> # that depends on A. But it has a valid output schema, and we also see that
> # it depends ultimately on the SciDB array `test`:
> B@name

[1] "subarray(between(transpose(test),3,0,3,4),3,0,3,4)"

> B@schema

[1] "array<val:double> [j=0:0,4,0,i=0:4,5,0]"
```

```
> B@gc$depend[[1]]@gc$depend[[1]]@name

[1] "test"

> # We can use `scidbeval` to force B's evaluation and storage inside SciDB:
> C <- scidbeval(B)
> # C has the same schema as B, but is now evaluated and stored inside SciDB
> # using an automatically-generated name. Unlike B, C does not depend on
> # the `test` array any more.
> C@name

[1] "R_array1d18f141ac01101270513297"

> C@schema

[1] "array<val:double> [j=0:0,4,0,i=0:4,5,0]"
```

8 Examples

8.1 K-means clustering

8.2 Correlations and graphs

9 Package installation

Installation proceeds in two steps: installing the R package on any computer that has a network connection to a SciDB database, and installing a simple network service on the SciDB database coordinator computer.

9.1 Installing the R package from sources on GitHub

The `scidb` package source is maintained in the SciDBR GitHub repository. That's where the most up-to-date version of the package is available. Released versions of the package posted to CRAN are updated much less frequently, approximatley semiannually. A git tag indicates each CRAN release version of the package in the source code repository.

The wonderful `devtools` R package makes installation of source packages from GitHub nearly as simple as installation from CRAN.

Listing 1: Installing the R package from GitHub

```
library("devtools")
install_github("SciDBR", "Paradigm4")
```

9.2 Installing the R package from CRAN

The `scidb` package is available on CRAN. Start an R session and run:

Listing 2: Installing the R package from CRAN

```
install.packages("scidb")
```

9.3 Installing the simple HTTP service for SciDB

The SciDB R package requires installation of a simple open-source HTTP network service called `shim` on the computer that SciDB is installed on. The service needs to be installed only on the SciDB coordinator computer, not on client computers that connect to SciDB from R. It's available in packaged binary form for supported SciDB operating systems, and as source code which can be compiled and deployed on any SciDB installation.

Both installation approaches install the `shim` network service on the SciDB coordinator computer. Installing as a service requires root permission. The compiled source code version requires no special permissions to run.

Installation from binary software packages for SciDB-supported operating systems is easiest. Detailed up-to-date information can be found on Paradigm4's laboratory on Paradigm4's Github repository at https://github.com/Paradigm4/shim/wiki/Installing-shim. We outline installation for each supported operating system below. See our github page for source code. The open source package author, Bryan Lewis, maintains binary packages for SciDB-supported operating systems. They are tied to specific versions of SciDB. The present version is 13.9 (September, 2013).

9.3.1 Installation on RHEL/CentOS 6

Listing 3: Installing the simple HTTP service on RHEL

```
# Install with:
wget http://paradigm4.github.io/shim/shim-13.9-1.x86_64.rpm
rpm -i shim-13.9-1.x86_64.rpm

# (Uninstall, if desired, with:)
yum remove shim
```

9.3.2 Installation on Ubuntu 12.04

Listing 4: Installing the simple HTTP service on Ubuntu

```
# Install with:
wget http://paradigm4.github.io/shim/shim_13.9_amd64.deb
sudo gdebi shim_13.9_amd64.deb

# (Uninstall, if desired, with:)
apt-get remove shim
```

See the Wiki and web pages at https://github.com/Paradigm4/shim/ for up to date package information and source code.

The installed `shim` network service exposes SciDB as a very simple HTTP API. It includes a simple browser-based status and query tool. After installing shim, point your browser to the I.P. address of the SciDB coordinator machine and port 8080, for example: http://localhost:8080 on the coordinator machine itself. Note that this API is not official and may change in the future. Help drive those changes by contributing ideas, code and bugfixes to the project on github, or feel free to discuss the service on the SciDB.org/forum.

9.4 Error handling

SciDB errors are trapped and converted to R errors that can be handled by standard R mechanisms. Some operations might try to return too much data to R, exceeding R's indexing limitations, system memory, or both. The package tries to avoid this kind of error using package options that limit returned data size shown in the next section.

9.5 Package options, miscellaneous notes, and software license

The `scidb` package defines several global package options. Package options may be set and retrieved with the R `options` function, and are listed in Table 2.

Miscellaneous notes follow:

Option	Default value	Description
scidb.debug	NULL	Set to TRUE to display all queries issued to the SciDB engine and other debugging information.
scidb.index.sequence.limit	100 000 000	Maximum allowed scidb array object sequential indexing limit (for larger ranges, use between)
scidb.max.array.elements	100 000 000	Maximum allowed non-empty elements to return in a subsetting operation of a scidb array object.

Table 2: Package options

- R does not support 64-bit integer types. 64-bit signed and unsigned integers smaller than 2^{53} in magnitude will be represented as double-precision floating point numbers. 64-bit integers outside that range appear as $+/-$`Inf`. All other integers (int8, uint8, int16, uint16, etc.) are represented in R by 32-bit signed integers. The uint32 type is not directly supported.
- R doesn't support single-precision floating point numbers. `iquery` results convert single-precision numbers within SciDB to double-precision floating-point numbers in R. Single-precision SciDB numbers are not supported by the `scidb` array class.
- SciDB does not natively support complex numbers. Loading complex numbers directly into SciDB from R is not defined.
- The `iquery` function provides the most flexible mechanism for type conversion between the systems, fully under user control using `read.table` options.
- Allowed array naming conventions vary between R and SciDB. For example, SciDB does not allow decimal points in attribute names. The package may alter names with character substitution to reconcile names when it is reasonable to do so. A warning is emitted whenever an object is automatically renamed in this way.