

SciDB and



 paradigm4
data-driven discovery

Contents

1	Introduction	3
2	Connecting to SciDB and Running Queries	3
2.1	Connecting to SciDB	3
2.2	Listing and removing SciDB arrays	4
2.3	Running SciDB queries	4
2.4	Iterating over query results	5
3	SciDB Arrays for R Users	6
4	The <code>scidbdf</code> data frame-like class	7
4.1	Examples of <code>scidbdf</code> objects	8
5	The <code>scidb</code> array class	9
5.1	Subsetting and indexing <code>scidb</code> array objects	9
5.1.1	Arithmetic operations and their rules	13
5.2	Persistence of dynamically-allocated <code>scidb</code> arrays	13
5.3	Miscellaneous array functions	14
6	Using SciDB arrays with existing R code	15
7	Package installation	16
7.1	Installing the R package from CRAN	16
7.2	Installing the simple network service for SciDB	17
7.2.1	Installation on RHEL/CentOS 6	17
7.2.2	Installation on Ubuntu 12.04	17
7.3	Error handling	18
7.4	Package options, miscellaneous notes, and software license	18

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 7.2). 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.

Listing 1: Connecting to SciDB

```
library("scidb")
# Connect to localhost by default on unencrypted port 8080:
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. The `iquery` function supports standard R `read.table` parameter options to facilitate type conversion.

Listing 2 illustrates basic use of `iquery`. (Program code is shown in black text and example results are shown in blue in all following example listings.)

Listing 2: Data frame example

```
library("scidb")
scidbconnect()          # Connect to SciDB on localhost
scidblist()             # List SciDB arrays (nothing there yet)

[1] NULL

# Build a 1-D SciDB array named "P:"
iquery("store(build(<x:double>[i=0:99,100,0],asin(1)*i/25),P)")

# Return to R the result of an apply operator:
S = iquery("apply(P,y,sin(x))",return=TRUE)
head(S)
  i      x      y
1 0 0.0000000 0.0000000
2 1 0.0628319 0.0627905
3 2 0.1256640 0.1253330
4 3 0.1884960 0.1873810
5 4 0.2513270 0.2486900
```

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.

Listing 3: Iterating over an iquery result

```
# 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 0.316228
2 2 0.2 0.447214
3 3 0.3 0.547723
4 4 0.4 0.632456
5 5 0.5 0.707107
6 6 0.6 0.774597

nextElem(it)
  i  x  y
1 7 0.7 0.836660
2 8 0.8 0.894427
3 9 0.9 0.948683
4 10 1.0 1.000000

nextElem(it)
Error: StopIteration
```

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. Thus, a one-dimensional SciDB array is conceptually 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.

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 62-bit signed integers (R indices are unsigned positive 31-bit integer or 52-bit integer-valued double values).

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.

In addition to available SciDB missing codes, SciDB double-precision floating point values also provide a value indicating missingness identically to R, and use the identical NA representation that R uses. Unlike R however, other SciDB data types do not define NA (use SciDB NULL instead).

The `scidb` package defines two array classes for R with data backed by SciDB arrays.

4 The `scidbdf` data frame-like class

The `scidbdf` class defines a data frame-like class with data 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). Non-integer row indices are not supported.

Use either the `df2scidb` or `as.scidb` functions 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

Listing 4: SciDB data frame-like 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:  morley
Attributes:
  attribute  type nullable
1      Expt  int32    FALSE
2       Run  int32    FALSE
3      Speed  int32    FALSE
Row dimension:
  No name start length chunk_interval chunk_overlap low high  type
1  0  row      1    100             100             0   1  100 int64

# Materialize the first five rows of X to R (using [] to return results to R):
X[1:5,][]
  Expt Run Speed
0     1   1  850
1     1   2  740
2     1   3  900
3     1   4 1070
4     1   5  930

# Aggregate average speed by experiment using SciDB
aggregate(X, by="Expt", FUN="avg(Speed) as mean")
  Expt mean
XXX CORRECT THIS EXAMPLE
1     1 909.0
2     2 856.0
3     3 845.0
4     4 820.5
5     5 831.5
```

Note that the aggregation function for SciDB arrays has a slightly different syntax than aggregation of data frames; in particular note that the aggregation function is a SciDB expression represented as a character string.

5 The `scidb` array class

Similarly to the data frame-like class, the `scidb` package defines a `scidb` n-dimensional array class for R. 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 presented in Listing ??, represented within SciDB as a 1-D array with five attributes. The following listing 5 illustrates creating a 1-D `scidb` array object in R that refers to the iris data in SciDB, using the `Sepal_Width` attribute. Unlike R, SciDB numeric array indices may be zero or negative (indices begin at zero in the example in Listing 5). Data from `scidb` array objects are not materialized to R until subset with the empty indexing function, `[]`.

Listing 5: A 1-d SciDB array object using one of several available attributes

```
x = scidb("iris", attribute="Sepal_Width", data.frame=FALSE)
dim(x)
[1] 150

x[99:103] []
[1] 2.5 2.8 3.3 2.7 3.0

attributes(x)      # List all available attributes in the SciDB array
[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,...]`.)

- Array length may exceed 2^{31} elements.
- `scidb` array objects are limited to double-precision and 32-bit signed integer numeric, logical, and single-byte character (`char`) element data types.

Listing 6 illustrates basic integer indexing operations on a sparse 3-D SciDB array.

Listing 6: Basic `scidb` subarray indexing

```
scidbremove("A", error=invisible)

# Create a small, sparse 3-d array:
iquery("store(build_sparse(<val:double>[i=0:9,10,0,j=0:9,5,0,k=0:9,2,0],k,k<99
    and (j=1 or j=3 or j=5 or j=7)),A)")
A = scidb("A")

dim(A)
[1] 10 10 10

# Indexing operations return new SciDB arrays:
A[0:3,2:3,5:8]
A reference to a 4x2x4 dimensional SciDB array

# But their data can be materialized into an R array with []:
A[0:3,2:3,5] [,drop=FALSE]
, , 1

      [,1] [,2]
[1,]   NA    5
[2,]   NA    5
[3,]   NA    5
[4,]   NA    5
```

String-valued SciDB non-integer dimensions are supported by `scidb` arrays, with some limitations illustrated in the examples below. The example in Listing 7 illustrates indexing by string values as well as mixed indexing by string and integer.

Listing 7: Integer and string subarray indexing

```
scidbremove(c("A","N"), error=warning)
iquery("store(build_sparse(<val:double>[i=0:9,5,0,j=0:9,5,0],i,i=j),A)")
iquery("create array N<val:double>[x(string)=10,10,0,y(string)=10,10,0]")
iquery("redimension_store(apply(A,x,'x'+string(i),y,'y'+string(j)),N)")
n = scidb("N")

str(n)
SciDB array name:  N      attribute in use:  val
All attributes:  val
Array dimensions:
  No name start length chunk_interval chunk_overlap low high  type
1  0    x      0     10             10             0  0    9 string
2  1    y      0     10             10             0  0    9 string
dimnames(n)
[[1]]
[1] "x0" "x1" "x2" "x3" "x4" "x5" "x6" "x7" "x8" "x9"

[[2]]
[1] "y0" "y1" "y2" "y3" "y4" "y5" "y6" "y7" "y8" "y9"
# Empty cells in SciDB are replaced by a default value, NA below.
n[]
      y
x    y0 y1 y2 y3 y4 y5 y6 y7 y8 y9
x0   0 NA NA NA NA NA NA NA NA NA
x1  NA  1 NA NA NA NA NA NA NA NA
x2  NA NA  2 NA NA NA NA NA NA NA
x3  NA NA NA  3 NA NA NA NA NA NA
x4  NA NA NA NA  4 NA NA NA NA NA
x5  NA NA NA NA NA  5 NA NA NA NA
x6  NA NA NA NA NA NA  6 NA NA NA
x7  NA NA NA NA NA NA NA  7 NA NA
x8  NA NA NA NA NA NA NA NA  8 NA
x9  NA NA NA NA NA NA NA NA NA  9

# Note that the package drops dimension labels in some cases, for example
# when arbitrary indices are selected (a warning is thrown to alert the user):
n[c("x2","x3","x1","x5"),][[
      y
x    y1 y2 y3 y5
0  NA  2 NA NA
1  NA NA  3 NA
2   1 NA NA NA
3  NA NA NA  5
Warning message: Dimension labels were dropped.
```

The **between** function may be used to specify indexing range intervals for integer or string values. It's useful for specifying subarrays of very large arrays efficiently. The **between** function may be

used to specify numeric or string dimension intervals. Listing 8 illustrates its use, using the same example arrays used in Listing 7.

Listing 8: Using between to specify subarrays

```
N[between('x3','x7'), ][]
      y
x    y3 y4 y5 y6 y7
x3   3 NA NA NA NA
x4  NA  4 NA NA NA
x5  NA NA  5 NA NA
x6  NA NA NA  6 NA
x7  NA NA NA NA  7
```

Listing 9 shows a more interesting 2-d array example that compares matrix arithmetic in R and SciDB.

Listing 9: Matrix arithmetic in R and SciDB

```
iquery("store(build(<x:double>[i=1:5,5,0,j=1:5,5,0],double(i)/double(j)),V)")
v = scidb("V") # v is an R representation of a SciDB array

str(v)
SciDB array name: V      attribute in use: x
All attributes: x
Array dimensions:
  No name start length chunk_interval chunk_overlap low high type
1  0     i      1      5              5              0  1   5 int64
2  1     j      1      5              5              0  1   5 int64

t(v[]) %*% v[] # Compute V %*% V using R
      [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 55.00000 27.50000 18.33333 13.75000 11.00000
[2,] 27.50000 13.75000  9.16667  6.87500  5.50000
[3,] 18.33333  9.16667  6.11111  4.58333  3.66667
[4,] 13.75000  6.87500  4.58333  3.43750  2.75000
[5,] 11.00000  5.50000  3.66667  2.75000  2.20000

# Now compute using SciDB, and materialize the result to R:
(t(v) %*% v)[]
      [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 55.00000 27.50000 18.33333 13.75000 11.00000
[2,] 27.50000 13.75000  9.16667  6.87500  5.50000
[3,] 18.33333  9.16667  6.11111  4.58333  3.66667
[4,] 13.75000  6.87500  4.58333  3.43750  2.75000
[5,] 11.00000  5.50000  3.66667  2.75000  2.20000
```

Linear algebra operations like the cross product in Listing 9 store their results in new dynamically-

named SciDB arrays. One may always find the SciDB name for a `scidb` array object from the array object's `@name` slot.

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. Listing 10 shows an example of computations that mix `scidb` array objects with R vectors.

Listing 10: Mixed R and SciDB array arithmetic

```
iquery("store(build(<x:double>[i=0:4,5,0,j=0:4,5,0],double(i+1)/double(j+1)),U)
      ")
u = scidb("U")          # u is an R representation of a SciDB array
set.seed(1)
x = cbind(rnorm(5))     # An R column vector

y = u %*% x             # Computed by SciDB, returning a SciDB array object

y[,drop=FALSE]          # Return the computed result to R
      [,1]
[1,] -0.3484533
[2,] -0.6969065
[3,] -1.0453598
[4,] -1.3938131
[5,] -1.7422663
```

Although the examples may seem trivial, the simple linear algebra capability shown in Listings 9 and 10 enable quite a lot of interesting computation. Later sections illustrate using this idea to overload more substantial functions in existing R packages.

5.1.1 Arithmetic operations and their rules

The `scidb` class supports the operations shown in Table 1.

The subarray/materialize operations `[]` support the standard `drop` argument and the argument `default=x`, where `x` is the default scalar value to use to fill-in sparse array values when materialized to R. The use of `default` overrides the related global

5.2 Persistence of dynamically-allocated `scidb` arrays

Previous examples illustrate that new `scidb` arrays may be created after some R operations. For example, the subarray of a `scidb` array is a new `scidb` array. SciDB arrays created as the result of

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
<code>crossprod(A,B)</code>	Cross product $\mathbf{t}(A) \% \% B$	A, B SciDB arrays or R matrices/vectors	SciDB array
<code>tcrossprod(A,B)</code>	Cross product $A \% \% \mathbf{t}(B)$	A, B SciDB arrays or R matrices/vectors	SciDB array
$A * B$	Elementwise multiplication	A, B Conformable SciDB arrays or R matrices/vectors	SciDB array
$\alpha * A$	Scalar multiplication	SciDB array A , scalar α	SciDB array
$\mathbf{t}(A)$	Transpose	SciDB array A	SciDB array
$A[\text{range}, \text{range}, \dots]$	Subarray	SciDB array A	SciDB Array
$A[]$	Materialize	SciDB array	R array

Table 1: SciDB Array Class Operations

R operations do not persist by default—they are removed from SciDB when their corresponding R objects are deleted in R. Consider the example in Listing 11, shown with debugging turned on.

Listing 11: Non-persistence of intermediate arrays

```
iquery("store(build(<x:double>[i=0:4,5,0,j=0:4,5,0],double(i+1)/double(j+1)),U)
")
u = scidb("U")
V = u[1:3,1:5]      # The subarray V is a new SciDB array
options(scidb.debug=TRUE)
rm(V)
gc()                # Force R to run garbage collection
remove(array64626e31022a)
```

The debugging message in Listing 11 illustrates that the temporary SciDB array that V represented was removed from SciDB when R garbage collection was run. In order to set any SciDB array associated with a `scidb` array object as persistent, set the array `@gc$remove` setting to `FALSE`—for example, `V@gc$remove=FALSE` in the above example.

5.3 Miscellaneous array functions

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

The `filter` function may be used to apply arbitrary SciDB filter logic to array attributes. Simple filtering comparisons against scalars may be directly specified with the usual comparison symbols, `<`, `>`, `<=`, `>=`, `==`, `!=`. The result of the `filter` function and the simple binary comparison operations is a new `scidb` object containing the filtered values.

6 Using SciDB arrays with existing R code

This section illustrates using SciDB together with R and standard R packages from CRAN to compute solutions to large-scale problems. R functions that rely on linear algebra and aggregation operations may be adapted to use SciDB arrays in place of native R vectors and matrices in order to benefit from the large-scale parallel computing capabilities of SciDB.

The truncated singular value distribution (TSVD) is an important, widely used analysis method. Truncated SVD lies at the heart of principle components and other analysis methods.

We use the `irlba` package from CRAN, <http://cran.r-project.org/web/packages/irlba/index.html> to efficiently compute a truncated SVD. The IRLB algorithm used by the package relies on mostly matrix–vector products, and is well-suited for use with SciDB. In fact, we can use SciDB matrices with the `irlba` package without modifying the package at all.

The `irlba` package includes an option for user-defined matrix-vector products between a matrix `A` and a vector `x`, that is the R computation `A %*% x`, and for computation of `t(A) %*% x`. Because matrix vector and transpose operations are defined for the `scidb` array class, we don't technically need to use the user-defined option in the `irlba` package. However, by using the option, we can greatly improve efficiency by avoiding explicitly forming the matrix transpose by computing `t(t(x) %*% A)` instead of `t(A) %*% x`. Listing 12 illustrates this.

Listing 12: Efficient matrix vector product for IRLBA

```
# Let A be a scidb matrix
# Let x be a numeric vector
# Compute A %*% x if transpose=FALSE
# Compute t(A) %*% x if transpose=TRUE
# Return an R numeric vector.
matmul = function(A, x, transpose=FALSE)
{
  if(transpose)
  {
    return(t(crossprod(x,A))[,drop=FALSE]))
  }
  (A %*% x)[,drop=FALSE]
}
```

After defining the custom matrix or transpose matrix product in Listing 12, we can load and use the `irlba` package with SciDB arrays. Listing 13 illustrates computation of a few largest singular values and associated singular vectors of a $50,000 \times 50,000$ matrix with random entries (consuming about 18 GB). That problem large enough that it can't be computed easily in R—the matrix is too large to even represent in R version 2 (although that changes soon in R version 3).

Listing 13: Example large truncated SVD computation

```
library("irlba")
library("scidb")
scidbconnect()

# Create a 50,000 x 50,000 matrix filled with random-valued entries:
iquery(
  "store(build(<x:double>[i=0:49999,1000,0,j=0:49999,1000,0],double(random())
    /10000000000),A)"
)
a = scidb("A")
dim(a)
[1] 50000 50000

# Compute the three largest singular values and corresponding vectors;
S = irlba(a, nu=3, nv=3, matmul=matmul)
```

After a while, the algorithm returns the truncated SVD in the variable `S`. The result obtained is comparable to what the `svd(A, nu=3, nv=3)` command would have produced, if it could handle the large matrix. Further optimizations are possible, but this simple example shows that it can be easy to get large-scale computation working without rewriting R code.

7 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.

7.1 Installing the R package from CRAN

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

Listing 14: Installing the R package from CRAN

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


7.2 Installing the simple network 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).

7.2.1 Installation on RHEL/CentOS 6

Listing 15: Installing the simple HTTP service on RHEL

```
# Install with:
wget http://illposed.net/shim-13.2-1.x86_64.rpm
rpm -i shim-13.2-1.x86_64.rpm

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

7.2.2 Installation on Ubuntu 12.04

Listing 16: Installing the simple HTTP service on Ubuntu

```
# Install with:
wget http://illposed.net/shim_13.2_amd64.deb
sudo gdebi shim_13.2_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.

7.3 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.

7.4 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.

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

Table 2: Package options

Miscellaneous notes follow:

- 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 $+\text{--Inf}$. 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.

Copyright (C) 2008–2013 SciDB, Inc.

The SciDB package for R is free software: you can redistribute it and/or modify it under the terms of the AFFERO GNU General Public License as published by the Free Software Foundation.

The SciDB package for R is distributed "AS-IS" AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. See the AFFERO GNU General Public License for the complete license terms.

You should have received a copy of the AFFERO GNU General Public License along with the package. If not, see [<http://www.gnu.org/licenses/agpl-3.0.html>](http://www.gnu.org/licenses/agpl-3.0.html)