# SciDB and R

*Using the scidb package for R.*

# Contents

# 1 Introduction

SciDB is an open-source database that organizes data in $n$-dimensional arrays. Many interesting 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, 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.

# 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 Listing and removing SciDB arrays

The `scidblist` function lists SciDB objects (arrays, instances, etc.), optionally showing detailed schema information for arrays.

The `scidbremove` function removes a SciDB array, or optionally a set of SciDB arrays defined by regular expression.

See their respective R help pages for detailed information on available function arguments.

## 2.2 Connecting to SciDB

The `scidbconnect` function establishes a connection to a SciDB coordinator instance. The function may be safely called multiple times and. Once a connection is established, connection information is maintained until a different connection is established or the R session ends.

## 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 share 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.$n$-dimensional. They iquery function can optionally use R read.table options for parsing value types.

Listing 1 illustrates basic use of iquery.

Listing 1: 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

plot(S$x,S$y,xlab="x",ylab="y",col=4)
```

## 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 instead by setting the iterative=TRUE

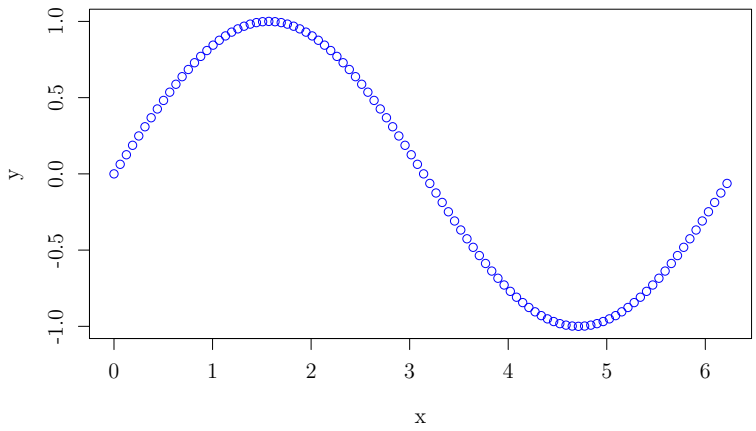


Figure 1: Plot output from Listing 1.

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 2: 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)")

# Reurn 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
```

## 2.5 Copying data frames from R into SciDB

The df2scidb function copies an R data frame into a one-dimensional SciDB array. The data frame rows are used as the array dimension. The data frame columns are mapped to SciDB array attributes, each column becoming one attribute. SciDB attributes and dimensions are discussed in greater detail below in Section 3, and in the SciDB reference documentation. The df2scidb function only supports R numeric, integer, character, and logical data types in the data frame. The function accepts many arguments outlined in detail in the R help page, ?df2scidb.

Listing 3 illustrates copying data from R to SciDB and back to R using the df2scidb and iquery functions.

Listing 3: Data frame example

```
library("scidb")
scidbonnect()                # Connect to SciDB
df2scidb(iris)               # Copy the R 'iris' data frame to SciDB.
scidblist(verbose=TRUE)      # List SciDB arrays and schema.
[1] iris <Sepal_Length:double, Sepal_Width:double,
          Petal_Length:double, Petal_Width:double,
          Species:string>        [row=0:149,150,0]

# Ask for the data back from SciDB:
head(iquery("scan(iris)",return=TRUE))
[1]   row Sepal_Length Sepal_Width Petal_Length Petal_Width Species
  1   0          5.1         3.5          1.4         0.2  setosa
  2   1          4.9         3.0          1.4         0.2  setosa
  3   2          4.7         3.2          1.3         0.2  setosa
  4   3          4.6         3.1          1.5         0.2  setosa
  5   4          5.0         3.6          1.4         0.2  setosa
  6   5          5.4         3.9          1.7         0.4  setosa
```

# 3 SciDB Arrays for R Users

Data are organized by SciDB into $n$-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 from that used by sparse matrices defined by R's Matrix package (whose sparse elements are implicitly zero).

The elements of a SciDB array, called *cells*, contain one or more *attributes*, or 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: 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 coördinate systems used to index SciDB arrays are similar to those used by R. Like R, array

dimension indices may be integer or non-integer. Unlike R, SciDB integer indices may be zero or negative, and are represented by 62-bit signed integers.

SciDB attribute values within a cell may be explicitly marked missing, indicated by a special SciDB missing code. SciDB internally supports a large number of possible missing codes. Presently, all SciDB missing code values are mapped to NA values in R.

Double-precision floating point values in SciDB also provide a value indicating missing values identically to R, and use the identical `NA` representation that R uses. Other SciDB data types do not define `NA`. Use of `NA` with other than double-precision floating point values is undefined.

# 4 The scidbdf data.frame-like class

# 5 The scidb array class

The `scidb` package defines a `scidb` array class for R. Array objects defined by `scidb` 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 in order to conform to general R arrays (which have difficulty supporting general multi-valued elements). Consider the iris data presented in Listing 3, represented within SciDB as a 1-D array with five attributes. Listing 4 illustrates creating a `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 4). Data from `scidb` array objects are not materialized to R until subset with the empty indexing function, `[]`.

Listing 4: A 1-d SciDB array object using one of several available attributes

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

x[99:103][]
row
 99 100 101 102 103
2.8 3.3 2.7 3.0 2.9

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 similarly to regular arrays in R. Rectilinear subarrays may be defined by ranges of integer indices or by lists of specific integer and string values, if the SciDB backing array supports that.

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

- Proper subarrays of scidb arrays are new scidb arrays.
- 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.
- 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. (The upper-left corner of R arrays is always indexed by [1,1,...].)
- Array length may exceed $2^{31}$ elements.
- Empty sparse array dimensions among materialized subarray indices are omitted from the resulting array in R. That means materialized subarray dimensions may be smaller than the requested range.
- scidb array objects are limited to double-precision and 32-bit signed integer numeric, logical, and single-byte character string data.

Listing 5 illustrates basic integer indexing operations on a sparse 3-D SciDB array.

Listing 5: Basic scidb subarray indexing

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

# 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")

# Indexing operations return new SciDB arrays:
A[0:3,2:3,5:8]
Reference to the 4x1x4 dimensional SciDB array.attribute array64624eb3f52d.val

# But, their data can be materialized into an R array with []:
A[0:3,2:3,5:8] []
   k
i   5 6 7 8
  0 5 6 7 8
  1 5 6 7 8
  2 5 6 7 8
  3 5 6 7 8
```

String-valued SciDB non-integer dimensions are supported by scidb arrays. The example in Listing 6 illustrates indexing by string values as well as mixed indexing by string and integer.

Listing 6: 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

# It's OK to mix/match NID and integer indexing. Note also that
# totally empty dimensions are omitted from the result.
P = N[c("x1","x2","x5"),1:8]
P[]
    y
x     1  2  5
  x1  1 NA NA
  x2 NA  2 NA
  x5 NA NA  5
```

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 7 illustrates its use, using the same example arrays used in Listing 6.

Listing 7: 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 8 shows a more interesting 2-d array example that compares matrix arithmetic in R and SciDB.

Listing 8: 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

V[] %*%  V[]                    # Compute V %*% V by R
   j
i    1    2        3    4 5
  1  5  2.5 1.666667 1.25 1
  2 10  5.0 3.333333 2.50 2
  3 15  7.5 5.000000 3.75 3
  4 20 10.0 6.666667 5.00 4
  5 25 12.5 8.333333 6.25 5

# Now compute by SciDB; a dynamically-generated SciDB array holds the result:
VV = V %*% V
VV[]
   j
i    1    2        3    4 5
  1  5  2.5 1.666667 1.25 1
  2 10  5.0 3.333333 2.50 2
  3 15  7.5 5.000000 3.75 3
  4 20 10.0 6.666667 5.00 4
  5 25 12.5 8.333333 6.25 5
```

Linear algebra operations like the cross product in Listing 8 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 (addition, subtraction, matrix and matrix vector products, scalar products) are directly with R syntax for scidb array objects. 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 chunk partition schemes. Listing 9 shows an example of computations that mix scidb array objects with R vectors.

Listing 9: 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
              j
i             0
  0 -0.3484533
  1 -0.6969065
  2 -1.0453598
  3 -1.3938131
  4 -1.7422663

# Compare with same computation computed by R:
U[] %*% x
i          [,1]
  0 -0.3484533
  1 -0.6969065
  2 -1.0453598
  3 -1.3938131
  4 -1.7422663
```

Although the examples may seem trivial, the simple linear algebra capability shown in Listings 8 and 9 allow us to achieve quite a lot. Later sections illustrate using this idea to overload more substantial functions in existing R packages.

## 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 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 10, shown with debugging turned on.

Listing 10: 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 10 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 Sparse array implicit values and indexing notes

This section outlines additional differences between sparse `scidb` array objects and sparse R matrices introduced by the `scidb` package for practical reasons.

Subarrays of `scidb` array objects materialized to R by subsetting operations are returned as dense R arrays. We made this choice because sparse SciDB arrays don't correspond directly to sparse R matrices (SciDB arrays can be higher dimensional and contain different data types, for example).

Indexing operations on `scidb` array objects differ from those for sparse R matrices in a few ways for practical reasons. In particular:

- Returned sparse subarrays are bounded by their data extent–the returned subarray can be smaller than the requested index range. In practice, this allows SciDB to quickly return data from huge sparse arrays.
- Dense arrays are returned to R with implicit values filled in. Users have control over the choice of implied value.
- Subarrays are always returned in sorted index order.
- Repeated indices are not returned in subarrays.

The last two items are a result of technical issues and will go away in the next major version of the package.

The `scidb` package defines a default implicit value of zero for SciDB arrays in the package option `scidb.default.value`. Users may set this to any value. When there is a type mismatch between the default value and the array data type, `NA` is returned. Alternatively, indexing options include a default value parameter that can be used in place of setting a global default. Listing 11 illustrates

its use. It is important to note that the default value setting only affects data returned to R and not back-end SciDB computations. Treatment of sparsity in SciDB varies considerably by SciDB operator–consult the SciDB documentation for details.

Each of the sparse indexing issues discussed above are illustrated in Listings 11 and 12.

Listing 11: Sparse array indexing and default values

```
iquery("store(build_sparse(<v:double>[i=1:5,5,0,j=1:5,5,0],i,i=j or i=5), A)")

# The returned subarray shows i, j dimension labels corresponding to SciDB.
# The default numeric sparse array value 0 is used by default:
A[1:5, 1:5, default=0][]
   j
i   1 2 3 4 5
  1 1 0 0 0 0
  2 0 2 0 0 0
  3 0 0 3 0 0
  4 0 0 0 4 0
  5 5 5 5 5 5

# A different default value is easy to specify:
A[1:5, 1:5, default=NA][]
   j
i    1  2  3  4  5
  1  1 NA NA NA NA
  2 NA  2 NA NA NA
  3 NA NA  3 NA NA
  4 NA NA NA  4 NA
  5  5  5  5  5  5

# Subarray selections are bounded by their data extent (the 1st row is
# omitted from the result because it contains no data in the SciDB array):
A[1:3, 2:3, default=NA][]
   j
i    2  3
  2  2 NA
  3 NA  3

# See how this differs from default R indexing:
A[1:5, 1:5, default=NA][]  [1:3, 2:3]
   j
i    2  3
  1 NA NA
  2  2 NA
  3 NA  3
```

Listing 12: More SciDB sparse array indexing notes

```
# Returned subarrays from SciDB always have sorted indices (for now):
A[c(3,1,5),][]
   j
i   1 3 2 4 5
  1 1 0 0 0 0
  3 0 0 3 0 0
  5 5 5 5 5 5

# And, repeated indices in returned subarrays are omitted (for now):
A[c(3,1,1,5,][]
   j
i   1 3 2 4 5
  1 1 0 0 0 0
  3 0 0 3 0 0
  5 5 5 5 5 5
```

### 5.4 Miscellaneous array functions

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

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 can often be easily adapted to use SciDB arrays in place of native R vectors and matrices to benefit from the large-scale parallel computing capabilities of SciDB. An important method used in many analyses is the truncated singular value distribution (SVD). 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 13 illustrates this.

Listing 13: 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)
{
  rowChunkSize = A@D$chunk_interval[1]    # SciDB array row chunk size
  colChunkSize = A@D$chunk_interval[2]    # SciDB array column chunk size
  f1 = tempfile()                         # Temporary files (small)
  f2 = tempfile()
  writeBin(as.vector(x),f1)
  if(transpose) {
    iquery( sprintf("create array _xxx<val:double>[i=0:0,1,0,j=0:%s,%s,0]",
       length(x)-1,rowChunkSize) )
    iquery( sprintf("load(_xxx,'%s',0,'(double )')",f1) )
    iquery( sprintf("save(multiply(_xxx, %s), '%s', 0,'(double)')", A@name, f2)
       )
    ans = cbind(readBin(f2,n=ncol(A),what="double"))
  } else {
    iquery( sprintf("create array _xxx<val:double>[i=0:%s,%s,0,j=0:0,1,0]",
       length(x)-1,colChunkSize) )
    iquery( sprintf("load(_xxx,'%s',0,'(double )')",f1) )
    iquery( sprintf("save(multiply(%s, _xxx), '%s', 0,'(double)')", A@name, f2)
       )
    ans = cbind(readBin(f2,n=nrow(A),what="double"))
  }
  iquery( "remove(_xxx)" )
  unlink(f1)
  unlink(f2)
  ans
}
```

After defining the custom matrix or transpose matrix product in Listing 13, we can load and use the `irlba` package with SciDB arrays. Listing 14 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 is large enough that it can't be computed in R–the matrix is too large

to represent in R. (Note that if a SciDB array named A already exists, either delete it before running the following example, or change the name used in the example.)

Listing 14: 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())
     /1000000000),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 few minutes or so, depending on the system SciDB is running on, SciDB returns the truncated SVD to R 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.

# 7 Package Installation

The version of the SciDB R package discussed in this vignette must be installed on a computer with SciDB installed. If SciDB is installed in a networked cluster configuration, the R package should be installed on the same computer as the SciDB coördinator instance. This limitation is related to use of the coördinator node file system to upload data into SciDB and will be removed in the next major version release of the package. The R package is presently limited to 64-bit Linux operating systems that SciDB supports as a consequence of this requirement.

The package is presently distributed as a standard R source package. The installation process is outlined below.

1. Install SciDB
2. Install the SciDB client packages, including for example at least the `libscidbclient_12.10.0-1_amd64.deb` and `scidb-dev_12.10.0-1_all.deb` packages (or analogous RPM packages for RedHat systems).
3. Install R on your SciDB coördinator computer.
4. Obtain the `scidb_0.1-0.tar.gz` source package from Paradigm4.

5. Run `R CMD INSTALL scidb_0.1-0.tar.gz` from the command line, or `install.packages("scidb_0.1-0.tar.gz",repos=NULL)` from within R.

6. Enjoy SciDB!

## 7.1 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.

## 7.2 Package options

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 7.2.

| 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.default.value | 0 | Default value for returned subarrays of sparse numeric matrices. |
| scidb.max.array.elements | 100 000 000 | Maximum allowd non-empty elements to return in a subsetting operation of a scidb array object. |

Table 1: Package options

## 7.3 Miscellaneous notes

- The `scidb` package must run on the same computer that the SciDB coördinator is installed on–remote connection is not presently supported (but will be in the next major version).

- R does not support 64-bit integer types. Integers smaller than $2^{53}$ in magnitude will be represented as double-precision floating point numbers. Integers outside that range appear as $+/-$`Inf`.

- R only supports 32-bit integers. Lower resolution SciDB integers (for example, 8-bit integers) will be represented by 32-bit integers in R.

- 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 presently not defined.

- 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.