Distributed Computing Patterns in R

Whit Armstrong
armstrong.whit@gmail.com

KLS Diversified Asset Management

May 17, 2013

Messaging patterns

- Messaging patterns are ways of combining sockets to communicate effectively.
- In a messaging pattern each socket has a defined role and fulfills the responsibilities of that role.
- ZMQ offers several built-in messaging patterns which make it easy to rapidly design a distributed application:
  - Request-reply, which connects a set of clients to a set of services.
  - Pub-sub, which connects a set of publishers to a set of subscribers.
  - Pipeline, which connects nodes in a fan-out/fan-in pattern that can have multiple steps and loops.
  - Exclusive pair, which connects two sockets exclusively.

What does ZMQ give us?

- ZMQ is a highly specialized networking toolkit.
- It implements the basics of socket communications while letting the user focus on the application.
- Very complex messaging patterns can be built on top of these simple ZMQ sockets (Paranoid Pirate, Majordomo, Binary Star, Suicidal Snail, etc.).
- I highly recommend reading "The Guide" before writing your own apps.
- `http://zguide.zeromq.org/page:all`

Request / Reply example

- Req / Rep is the most basic message pattern.
- Both the request socket and reply socket are synchronous.
- The reply socket can only service one request at a time, however, many clients may connect to it and queue requests.

Request / Reply, Server

```
require(rzmq)

ctx <- init.context()
responder <- init.socket(ctx, "ZMQ_REP")
bind.socket(responder, "tcp://*:5555")

while (1) {
    req <- receive.socket(responder)
    send.socket(responder, "World")
}
```

Request / Reply, Client

```
require(rzmq)

requester <- init.socket(ctx, "ZMQ_REQ")
connect.socket(requester, "tcp://localhost:5555")

for (request.number in 1:5) {
    print(paste("Sending Hello", request.number))
    send.socket(requester, "Hello")
    reply <- receive.socket(requester)
    print(paste("Received:", reply, "number", request.number))
}

## [1] "Sending Hello 1"
## [1] "Received: World number 1"
## [1] "Sending Hello 2"
## [1] "Received: World number 2"
## [1] "Sending Hello 3"
## [1] "Received: World number 3"
## [1] "Sending Hello 4"
## [1] "Received: World number 4"
## [1] "Sending Hello 5"
## [1] "Received: World number 5"
```

Request / Reply server as remote procedure call

```
require(rzmq)

ctx <- init.context()
responder <- init.socket(ctx, "ZMQ_REP")
bind.socket(responder, "tcp://*:5557")

while (1) {
    req <- receive.socket(responder)
    send.socket(responder, req * req)
}
```

Request / Reply client as remote procedure call

```
require(rzmq)

requester <- init.socket(ctx, "ZMQ_REQ")
connect.socket(requester, "tcp://localhost:5557")

x <- 10
send.socket(requester, x)
reply <- receive.socket(requester)
all.equal(x * x, reply)

## [1] TRUE

print(reply)

## [1] 100
```

Request / Reply client – rpc server with user function

```
require(rzmq)

ctx <- init.context()
responder <- init.socket(ctx, "ZMQ_REP")
bind.socket(responder, "tcp://*:5558")

while (1) {
    msg <- receive.socket(responder)
    fun <- msg$fun
    args <- msg$args
    result <- do.call(fun, args)
    send.socket(responder, result)
}
```

Request / Reply client – rpc client with user function

```
require(rzmq)

requester <- init.socket(ctx, "ZMQ_REQ")
connect.socket(requester, "tcp://localhost:5558")

foo <- function(x) {
    x * pi
}
req <- list(fun = foo, args = list(x = 100))
send.socket(requester, req)
reply <- receive.socket(requester)
print(reply)

## [1] 314.2
```

Realistic example – c++ server

```cpp
#include <string>
#include <iostream>
#include <stdexcept>
#include <unistd.h>
#include <zmq.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
#include <order.pb.h>
#include <fill.pb.h>
using namespace boost::posix_time;
using std::cout; using std::endl;

int main () {
  zmq::context_t context(1);
  zmq::socket_t socket (context, ZMQ_REP);
  socket.bind ("tcp://*:5559");

  while (true) {
    //  wait for order
    zmq::message_t request;
    socket.recv(&request);

    tutorial::Order o;
    o.ParseFromArray(request.data(), request.size());

    std::string symbol(o.symbol());
    double price(o.price());
    int size(o.size());

    // send fill to client
    tutorial::Fill f;
    f.set_timestamp(to_simple_string(microsec_clock::universal_time()));
    f.set_symbol(symbol); f.set_price(price); f.set_size(size);

    zmq::message_t reply (f.ByteSize());
    if(!f.SerializeToArray(reply.data(),reply.size())) {
      throw std::logic_error("unable to SerializeToArray.");
    }
    socket.send(reply);
  }
  return 0;
}
```

Realistic example – R client

```
broker <- init.socket(ctx, "ZMQ_REQ")
connect.socket(broker, "tcp://*:5559")

## read the proto file
readProtoFiles(files = c("code/proto.example/order.proto", "code/proto.example/fill.proto"))

aapl.order <- new(tutorial.Order, symbol = "AAPL", price = 420.5, size = 100L)
aapl.bytes <- serialize(aapl.order, NULL)

## send order
send.socket(broker, aapl.bytes, serialize = FALSE)
## pull back fill information
aapl.fill.bytes <- receive.socket(broker, unserialize = FALSE)
aapl.fill <- tutorial.Fill$read(aapl.fill.bytes)
writeLines(as.character(aapl.fill))

## timestamp: "2013-May-16 17:33:41.619589"
## symbol: "AAPL"
## price: 420.5
## size: 100


esgr.order <- new(tutorial.Order, symbol = "ESGR", price = 130.9, size = 1000L)
esgr.bytes <- serialize(esgr.order, NULL)

## send order
send.socket(broker, esgr.bytes, serialize = FALSE)
## pull back fill information
esgr.fill.bytes <- receive.socket(broker, unserialize = FALSE)
esgr.fill <- tutorial.Fill$read(esgr.fill.bytes)
writeLines(as.character(esgr.fill))

## timestamp: "2013-May-16 17:33:41.627151"
## symbol: "ESGR"
## price: 130.9
## size: 1000
```

Pub / Sub example

- Pub / Sub is a more interesting pattern.
- The Pub socket is asynchronous, but the sub socket is synchronous.

Pub / Sub, Server

```
require(rzmq)

context = init.context()
pub.socket = init.socket(context, "ZMQ_PUB")
bind.socket(pub.socket, "tcp://*:5556")

node.names <- c("2yr", "5yr", "10yr")
usd.base.curve <- structure(rep(2, length(node.names)), names = node.names)
eur.base.curve <- structure(rep(1, length(node.names)), names = node.names)

while (1) {
    ## updates to USD swaps
    new.usd.curve <- usd.base.curve + rnorm(length(usd.base.curve))/100
    send.raw.string(pub.socket, "USD-SWAPS", send.more = TRUE)
    send.socket(pub.socket, new.usd.curve)

    ## updates to EUR swaps
    new.eur.curve <- eur.base.curve + rnorm(length(eur.base.curve))/100
    send.raw.string(pub.socket, "EUR-SWAPS", send.more = TRUE)
    send.socket(pub.socket, new.eur.curve)
}
```

Pub / Sub, USD Client

```
require(rzmq)

subscriber = init.socket(ctx, "ZMQ_SUB")
connect.socket(subscriber, "tcp://localhost:5556")
topic <- "USD-SWAPS"
subscribe(subscriber, topic)

i <- 0
while (i < 5) {
    ## throw away the topic msg
    res.topic <- receive.string(subscriber)
    if (get.rcvmore(subscriber)) {
        res <- receive.socket(subscriber)
        print(res)
    }
    i <- i + 1
}

##   2yr   5yr  10yr
## 1.989 1.996 1.992
##   2yr   5yr  10yr
## 2.006 2.005 1.996
##   2yr   5yr  10yr
## 2.001 1.992 2.003
##   2yr   5yr  10yr
## 2.005 1.997 1.998
##   2yr   5yr  10yr
## 1.998 2.010 2.006
```

Pub / Sub, EUR Client

```
require(rzmq)

subscriber = init.socket(ctx, "ZMQ_SUB")
connect.socket(subscriber, "tcp://localhost:5556")
topic <- "EUR-SWAPS"
subscribe(subscriber, topic)
i <- 0
while (i < 5) {
    ## throw away the topic msg
    res.topic <- receive.string(subscriber)
    if (get.rcvmore(subscriber)) {
        res <- receive.socket(subscriber)
        print(res)
    }
    i <- i + 1
}
```

```
##     2yr     5yr    10yr
## 0.9991 1.0146 0.9962
##     2yr     5yr    10yr
## 1.0268 0.9912 1.0090
##    2yr    5yr   10yr
## 1.001 1.001 1.000
##     2yr     5yr    10yr
## 1.0048 1.0010 0.9837
##     2yr     5yr    10yr
## 1.0075 0.9881 0.9972
```

Obligatory deathstar example

```
require(deathstar, quietly = TRUE)

estimatePi <- function(seed) {
    set.seed(seed)
    numDraws <- 10000
    r <- 0.5
    x <- runif(numDraws, min = -r, max = r)
    y <- runif(numDraws, min = -r, max = r)
    inCircle <- ifelse((x^2 + y^2)^0.5 < r, 1, 0)
    sum(inCircle)/length(inCircle) * 4
}

cluster <- c("localhost")
run.time <- system.time(ans <- zmq.cluster.lapply(cluster = cluster, as.list(1:1000),
    estimatePi))

print(mean(unlist(ans)))

## [1] 3.142

print(run.time)

##    user  system elapsed
##   1.276   0.816   6.575

print(attr(ans, "execution.report"))

##              jobs.completed
## krypton:9297             84
## krypton:9300             83
## krypton:9306             83
## krypton:9308             83
## krypton:9311             83
## krypton:9314             83
## krypton:9318             84
## krypton:9325             83
## krypton:9329             84
## krypton:9332             83
## krypton:9377             84
## krypton:9380             83
```

doDeathstar foreach example

```
require(doDeathstar, quietly = TRUE)
registerDoDeathstar("localhost")

z <- foreach(i = 1:100) %dopar% {
    set.seed(i)
    numDraws <- 10000
    r <- 0.5
    x <- runif(numDraws, min = -r, max = r)
    y <- runif(numDraws, min = -r, max = r)
    inCircle <- ifelse((x^2 + y^2)^0.5 < r, 1, 0)
    sum(inCircle)/length(inCircle) * 4
}

print(mean(unlist(z)))

## [1] 3.142
```

Thanks for listening!

Many people contributed ideas and helped debug work in progress as the rzmq package was being developed.

Bryan Lewis for collaborating and planning this talk with me.
JD Long for my excessive reuse of the estimatePi example.
Kurt Hornik for putting up with my packaging.
John Laing for finding bugs in my code.
Prof Brian Ripley for just being himself.

Elastic Computing with R and Redis

http://goo.gl/G9VAA

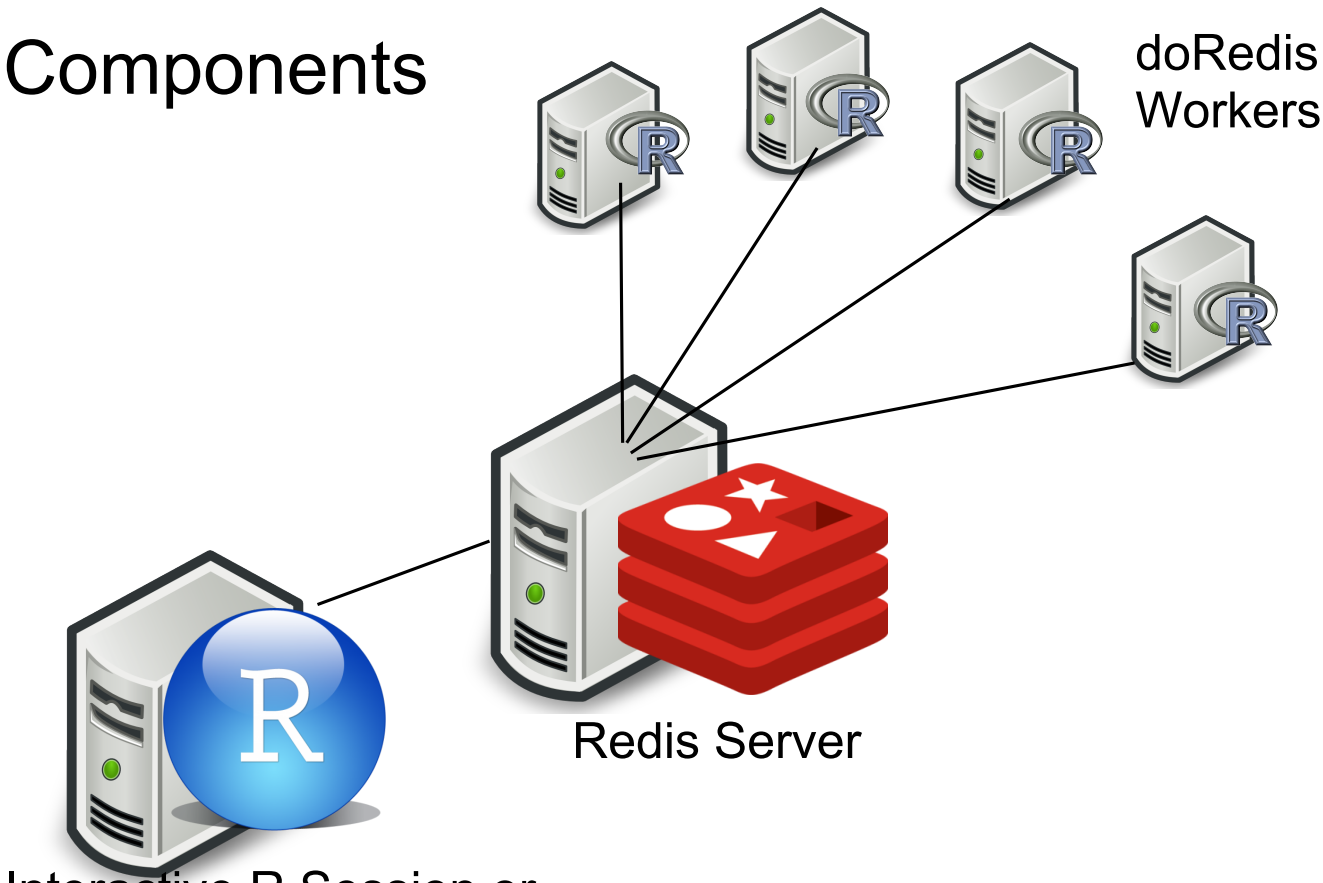
What does "elastic" mean?

- Computational resources can be added or removed at any time.
- Running computations benefit from added resources automatically.
- Computations on de-allocated resources are rescheduled automatically.

Why Elastic?

- Bursty/intermittent computational workloads
- Periodic resource availability
- Resource contention and dynamic reallocation

Components


doRedis
Workers
Redis Server
Interactive R Session or
Main Program

Toplogy

The components can:

- all be on a single computer
- all be on separate computers
- a mix of the above
- connected by intra- or inter-networks (deparmental network, EC2, Azure, etc.)

doRedis and EC2

Ready to roll AMI available. Linux magic is in the

`redis-worker-installer.sh`

file distributed with the package (a generic doRedis service for any LSB system).

Windows version also available from https://github.com/bwlewis/doRedisWindowsService.

EC2 Example I: Start doRedis workers

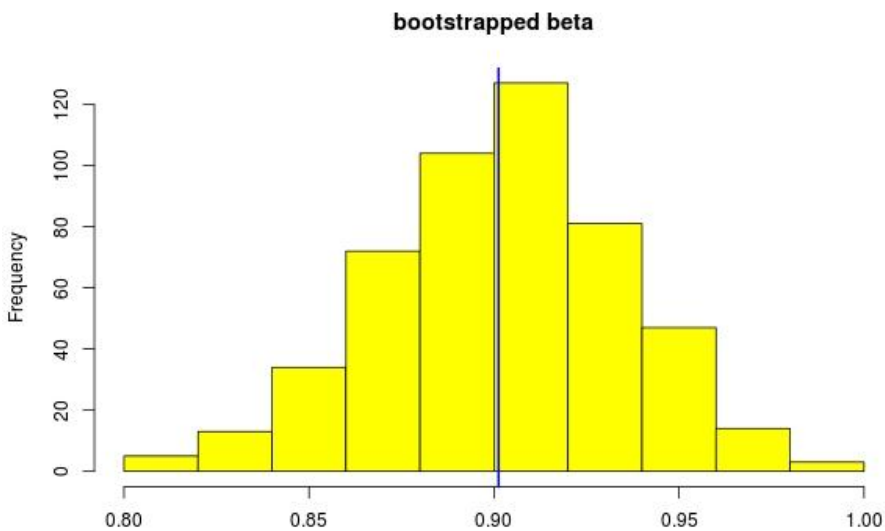
- Launch *one* new instance--this can serve as the Redis host and as a worker node.
- Obtain the IP address of the new instance.
- Additional instances may be specified at any time by supplying EC2 user-data:

```
host: <ip address of redis>
queue: <job queue name>
port: <redis port if not std.>
```

EC2 Example II: Example program

```
library("doRedis"); library("quantmod")

SP500 <- getSymbols("^GSPC",auto.assign=FALSE)
GOOG  <- getSymbols("GOOG",auto.assign=FALSE)
GOOG  <- diff(log(GOOG[,6])); SP500 <- diff(log(SP500[,6]))

# Estimate beta from the data:
beta = coef(lm(GOOG ~ SP500))[2]

# Bootstrap to get a sense of variation:
n    <- length(GOOG)
registerDoRedis(queue="RJOBS", host="HOST")
b <- foreach(j=1:5000,.combine=c,.packages="xts") %dopar% {
       i <- sample(n,n,replace=TRUE)
       coef(lm(GOOG[i] ~ SP500[i]))[2]
}

hist(b,col="yellow",main="bootstrapped beta",xlab="")
abline(v=beta,col="blue",lwd=2)
```

Example program output



This example is from Pat Burns' website: http://www.burns-stat.com/

doRedis tips and tricks

Redis server configuration (redis.conf)

- Comment out the bind line to listen on all interfaces:

```
# bind 127.0.0.1
```

- Set the timeout to zero to let workers wait indefinitely:

```
timeout 0
```

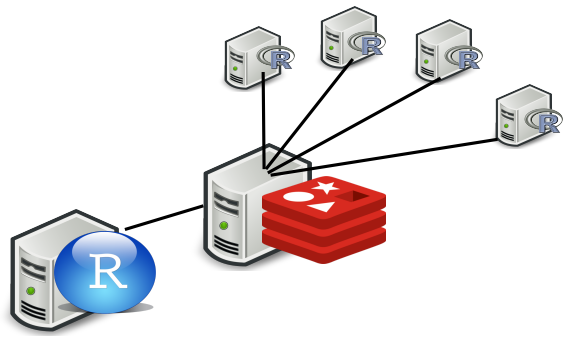
- chunkSize option
  Preferred number of loop iterations per job

- redisWorker iter and timeout options
  Number of jobs to execute before exiting/time to wait before exiting when queue is removed.

- set.seed.worker function
  Fine control over worker RNG state--see also the doRNG package and others.

setChunkSize, setExport, setPackages implement global ways to set some options, useful with plyR and others...

Caveat!



- Distributing data to workers through Redis...
  - Can be a bottleneck.
  - Redis largest value allowed is 512MB.

One solution: Access big data from **within** parallel jobs if possible. Easy to set this up to happen just once per worker even if many jobs are processed.

Revised example program

```
library("doRedis");

n    <- length(GOOG)
registerDoRedis(queue="RJOBS", host="HOST")

b <- foreach(j=1:5000,.combine=c, .packages="quantmod") %
dopar% {

  if(!exists("GOOG",envir=globalenv())) {
    S <- getSymbols("^GSPC",auto.assign=FALSE)
    G  <- getSymbols("GOOG",auto.assign=FALSE)
    assign("GOOG",diff(log(GOOG[,6])),envir=globalenv())
    assign("SP500",diff(log(SP500[,6])),envir=globalenv())
  }

  i <- sample(n,n,replace=TRUE)
  coef(lm(GOOG[i] ~ SP500[i]))[2]
}
```

foreach tips and tricks

Nesting (parallel loop unrolling)

```
library("doRedis")
registerDoRedis("RJOBS")
startLocalWorkers(n=1,queue="RJOBS")

# Use %:% to nest foreach loops. This trivial example
creates
# one set of 15 tasks:

foreach(x=0:2) %:%
  foreach(y=1:5,.combine=c) %dopar% { x+y }

[[1]]
[1] 1 2 3 4 5

[[2]]
[1] 2 3 4 5 6

[[3]]
[1] 3 4 5 6 7
```

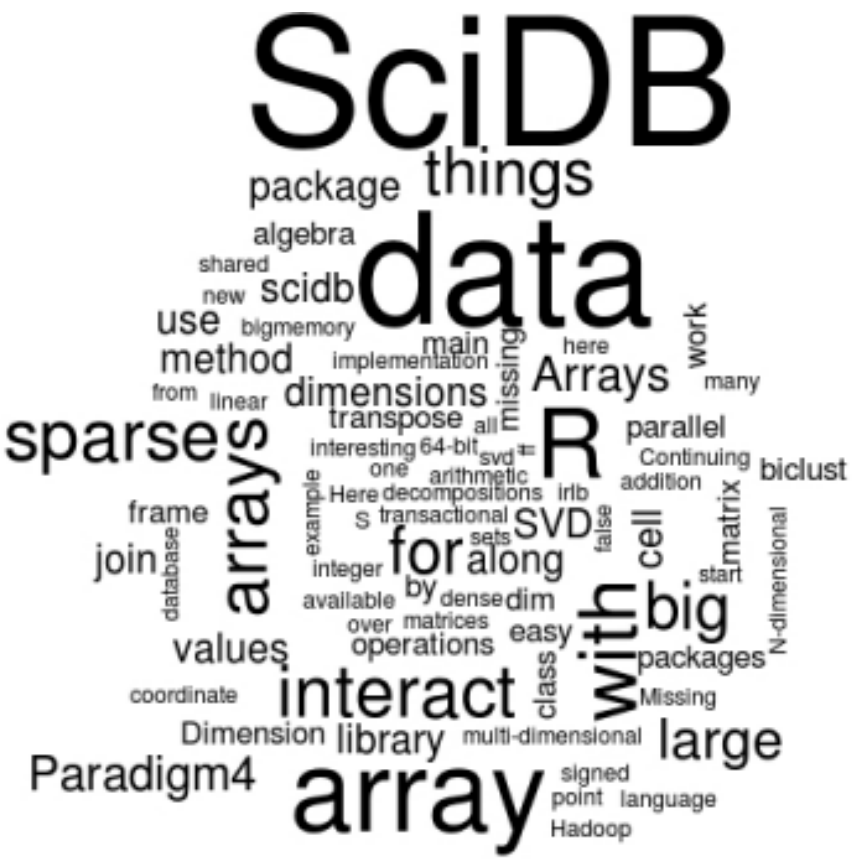
Parallel list comprehensions

```
# Use %:% and when to form list comprehensions. Conditions
# are evaluated in parallel, which can be an advantage
# if there is a huge amount of data to evaluate.


foreach(x=0:2) %:%
  foreach(y=1:5,.combine=c) %:%
    when(x<y) %dopar% {x+y}

[[1]]
[1] 1 2 3 4 5

[[2]]
[1] 3 4 5 6

[[3]]
[1] 5 6 7
```

On CRAN

development version at:

https://github.com/bwlewis/doRedis



http://goo.gl/Hn0Ro

Bryan Lewis, Paradigm4
blewis@paradigm4.com

Free and open-source array database

Sparse/dense, multi-dimensional arrays

Distributed storage, parallel processing

Excels at parallel sparse/dense linear algebra

ACID, data replication, versioned data

The package defines two main ways to interact with SciDB:

1. Iterable data frame interface using SciDB query language directly

2. **N-dimensional sparse/dense array class for R backed by SciDB arrays**

```
library("scidb")
scidbconnect(host="localhost")


# An example reference to a SciDB matrix:
A <- scidb("A")
dim(A)
[1] 50000 50000
```

Subarrays return new SciDB array objects

```
A[c(0,49000,171), 5:8]
```

```
Reference to a 3x4 SciDB array
```

Use `[]` to materialize data to R

```
A[c(0,49000,171), 5:8][]
```

```
           [,1]       [,2]       [,3]       [,4]
[1,]  0.9820799 -0.4563357 -1.2947495 -0.8085465
[2,] -1.5090126  0.1547963 -0.2435732 -0.1836875
[3,]  1.3296710 -1.5006536 -0.5980172  0.3752186
```

Arithmetic

```
X <- A %*% A[,1:5]
dim(X)
```

```
[1] 50000     5
```

Mixed **SciDB** and **R** object arithmetic

```
Z <- A[c(0,49000,171), 5:7]

(0.5*(Z + t(Z)) %*% rnorm(3))[, drop=FALSE]

          [,1]
[1,]  3.707263
[2,] -2.833560
[3,]  3.518370
```

Basic aggregation (scidbdf class)

```
A <- as.scidb(iris)
Warning message:
In df2scidb :Attribute names have been changed

aggregate(A, Petal_Length ~ Species, "avg
(Petal_Length) as mean")

     Species  mean
1     setosa 1.462
2 versicolor 4.260
3  virginica 5.552
```

It is sometimes possible to use SciDB arrays in R packages with little modification.

```
library("biclust")
library("s4vd")
data(lung)
A <- lung
x <- biclust(A, method=BCssvd, K=1)

# Now with SciDB arrays:
library("s4vdp4")
X <- as.scidb(A)
x1 <- biclust(X, method=BCssvd, K=1)

# Compare the results:
sqrt( x@info$res[[1]]$u - x1@info$res[[1]]$u) )
             [,1]
[1,] 5.202109e-16
```

SVD and principal components

```
S <- svd(A, nu=3, nv=3)
dim(S)
[1]     4 50000 50000

# Result is a 3-D array containing U,
  S (sparse), and V
```

Virtual machines and EC2 images ready to roll (including Rstudio) available from:

www.scidb.org

R package on CRAN and development version at:

github.com/Paradigm4