SciDB

Wusuo Liu and Chia Ying Lee

5/3/2017

# Introduction

**Chia: add brief intro**

Full documentation of SciDB is provided [here](https://paradigm4.atlassian.net/wiki/display/ESD/SciDB+Documentation).

The scidb R package is available on [CRAN](https://cran.r-project.org/package=scidb), along with [documentation](https://cran.r-project.org/web/packages/scidb/vignettes/scidb.pdf).

## SciDB Data Structures and Query Languages

**Charlie: Add explanation of data structures**

## AQL and AFL

SciDB provides query languages AQL, for querying the database, and AFL, for performing function calls on database objects. They are accessed from the unix terminal.

$ iquery  
AQL% set lang AFL;  
AFL%

## scidb R package

The scidb R package provides an interface to R, allowing interaction with the database from R, primarily using the iquery function.

## Basics of SciDB R package

First, load the library, and connect to the SciDB database. By default, it connects to the localhost.

library(scidb)  
dbConnect <- scidbconnect()

The datasets in the SciDB database can be listed using the following command.

(existingArray <- iquery(db = dbConnect, query = "list('arrays')", return = T))

## No name uaid aid  
## 1 0 M 45 45  
## 2 1 ratings 34 34  
## 3 2 RATINGS 36 36  
## 4 3 ratings\_correlationMatrix 72 72  
## 5 4 ratings\_crossjoin 69 69  
## 6 5 ratings\_matrix 31 31  
## 7 6 RATINGS\_MATRIX 38 38  
## 8 7 ratings\_matrix\_centered 60 60  
## 9 8 ratings\_matrix\_MxMT 56 56  
## 10 9 ratings\_singularValues 80 80  
## 11 10 ratingsSvdSdiag 82 82  
## 12 11 ratings\_userAvg 62 62  
## 13 12 ratings\_userMeans 48 48  
## 14 13 ratings\_userMSS 67 67  
## 15 14 sig 78 78  
## 16 15 tmpDb 17 17  
## 17 16 tmp\_ratings\_userMeans 50 50  
## 18 17 tmp\_ratings\_userMeans\_matrix 54 54  
## schema  
## 1 M<v:double> [row=0:3,1000,0,col=0:0,1000,0]  
## 2 ratings<userId:int64,movieID:int64,rating:double NOT NULL,timestamp:int64> [i=0:\*,1000000,0]  
## 3 RATINGS<userId:int64,movieID:int64,rating:double NOT NULL,timestamp:int64> [i=0:\*,1000000,0]  
## 4 ratings\_correlationMatrix<correlation:double> [userId=0:671,1000,0,userId2=0:671,1000,0]  
## 5 ratings\_crossjoin<multiply:double NOT NULL,rating\_avg:double,rating\_avg\_2:double,rating\_MSS:double,rating\_MSS\_2:double> [userId=0:671,1000,0,userId2=0:671,1000,0]  
## 6 ratings\_matrix<rating:double NOT NULL> [userId=0:671,1000,0,movieID=0:163949,1000,0]  
## 7 RATINGS\_MATRIX<rating:double NOT NULL> [userId=0:138493,1000,0,movieID=0:131262,1000,0]  
## 8 ratings\_matrix\_centered<rating\_centered:double> [userId=0:671,1000,0,movieID=0:163949,1000,0]  
## 9 ratings\_matrix\_MxMT<multiply:double NOT NULL> [userId=0:671,1000,0,userId2=0:671,1000,0]  
## 10 ratings\_singularValues<i:double,sigma:double> [i\_1=1:672,10000,0]  
## 11 ratingsSvdSdiag<sigma:double> [i\_2=0:671,1000,0,j=0:671,1000,0]  
## 12 ratings\_userAvg<rating\_avg:double> [userId=0:671,1000,0]  
## 13 ratings\_userMeans<rating\_avg:double> [userId=0:671,1000,0]  
## 14 ratings\_userMSS<rating\_MSS:double> [userId=0:671,1000,0]  
## 15 sig<i:double,sigma:double> [i\_1=1:672,10000,0]  
## 16 tmpDb<val:double> [i=0:99,100,0,j=0:99,100,0]  
## 17 tmp\_ratings\_userMeans<rating\_avg:double,movieID:int64 NOT NULL> [userId=0:671,1000,0]  
## 18 tmp\_ratings\_userMeans\_matrix<rating\_avg:double> [userId=0:671,1000,0,movieID=0:0,1000,0]  
## availability temporary  
## 1 TRUE FALSE  
## 2 TRUE FALSE  
## 3 TRUE FALSE  
## 4 TRUE FALSE  
## 5 TRUE FALSE  
## 6 TRUE FALSE  
## 7 TRUE FALSE  
## 8 TRUE FALSE  
## 9 TRUE FALSE  
## 10 TRUE FALSE  
## 11 TRUE FALSE  
## 12 TRUE FALSE  
## 13 TRUE FALSE  
## 14 TRUE FALSE  
## 15 TRUE FALSE  
## 16 TRUE FALSE  
## 17 TRUE FALSE  
## 18 TRUE FALSE

The above expression illustrates the basic usage of the iquery function: it is told the database to connect to, and the query to run. Setting return = T will return the scidb arrary to an R data frame, which we have called existingArray. By default, return = F, meaning that the query is performed in the SciDB database, but no output is returned to R.

## Installation

### Method 1: SciDB's VM

Download the virtual machine from [here](insert%20Google%20Docs%20URL). The virtual machine has SciDB pre-installed, and R and RStudio server already set up with scidb R package installed. However, note that the SciDB installation has problems with MPI, preventing the use of some essential functions such as the gesvd for computing SVD. Also, the installed version of scidb R package is not the latest version.

### Method 2: Install from scratch

We installed the SciDB from scratch.

**Charlie: add installation instructions**

### Start the database

$ scidb.py startall mydb

# The Big Data Problem

We use the movie ratings data from [MovieLens](https://grouplens.org/datasets/movielens/).

There are 2 datasets provided,

* Full data set
* Small data set

**Chia: add explanation of data, describe clustering goal**

## Download data

The following script downloads the two datasets from the MovieLens website, and stores them into respective subfolders in /home/scidb/Project.

## Get the small dataset  
data.dir <- "/home/scidb/Project/ml-latest-small" # Note: the full path is specified  
if (!dir.exists(data.dir)) {  
 download.file("http://files.grouplens.org/datasets/movielens/ml-latest-small.zip",   
 destfile = "/home/scidb/Project/ml-latest-small.zip")  
 unzip("/home/scidb/Project/ml-latest-small.zip")  
   
 # System command to strip header (first line) from data file ratings.csv  
 system(sprintf("tail -n +2 %s/ratings.csv > %s/ratings\_noHeader.csv", data.dir, data.dir))  
}  
rm(data.dir)  
  
## Get the full dataset  
data.dir <- "/home/scidb/Project/ml-20m"  
if (!dir.exists(data.dir)) {  
 download.file("http://files.grouplens.org/datasets/movielens/ml-20m.zip",   
 destfile = "/home/scidb/Project/ml-20m.zip")  
 unzip("/home/scidb/Project/ml-20m.zip")  
   
 system(sprintf("tail -n +2 %s/ratings.csv > %s/ratings\_noHeader.csv", data.dir, data.dir))  
}  
rm(data.dir)

## Load Data into SciDB

Next, we load the data into the SciDB database. Because the data array is a 2D array (i.e. a matrix), whereas the data is provided as a flat array (i.e. 1D array), loading the data has two steps. We explain the steps for the small dataset.

1. First, load the CSV file as a 1D array, named ratings. This is done by first creating the array, and then using the load function to read in the CSV file.

iquery(dbConnect, "CREATE ARRAY ratings <userId:int64, movieID:int64, rating:double NOT NULL, timestamp:int64> [i=0:?,1000000,0]")  
# Note 1: It's important to make rating column NOT NULL, to enable the use of spgemm later on.  
# Note 2: The 0:? allows us to create the array without knowing the size of the input file.  
# Note 3: The chunk size is 1000000 and overlap is 0  
  
iquery(dbConnect, "load(ratings, '/home/scidb/Project/ml-latest-small/ratings\_noHeader.csv', -2, 'csv')")

1. The second step is to *redimension* the ratings array from a 1D to 2D array. The columns userId and movieID in the 1D array will become the dimensions of the 2D array.

* Additionally, to be able to use subsequent functions, we need to specify the size of each dimension.

print(u\_max <- iquery(dbConnect, "aggregate(ratings, max(userId))", return = T))  
print(m\_max <- iquery(dbConnect, "aggregate(ratings, max(movieID))", return = T))

The userIds range from 1 to 671. The movieIDs range from 1 to 163949. Now, we redimension `ratings` array so that userId is the first dimension (rows) and movieID is the second dimension (columns).

iquery(dbConnect,   
 "store(redimension(ratings, <rating:double NOT NULL>[userId=0:671,?,0,movieID=0:163949,?,0]), ratings\_matrix)")  
# Note: We start the indices from 0, not 1, to enable the use of spgemm later on.

The `store` function stores the resulting array into a new array named `ratings\_matrix`.

Note that even though ratings\_matrix is a huge matrix (671 x 163949), it is very sparse.

print(n\_ratings <- iquery(dbConnect, "aggregate(ratings, count(\*))", return = T))

There are 100004 ratings available. All other entries of the matrix are zero, and not explicitly stored in the database. (In fact, according to the MovieLens data description, only 9000 movies are rated by the 700 users.)

Below is the corresponding code for loading the full dataset into SciDB. The names of the data arrays are RATINGS and RATINGS\_MATRIX.

iquery(dbConnect, "CREATE ARRAY RATINGS <userId:int64, movieID:int64, rating:double NOT NULL, timestamp:int64> [i=0:?,1000000,0]")  
  
iquery(dbConnect, "load(RATINGS, '/home/scidb/Project/ml-20m/ratings\_noHeader.csv', -2, 'csv')")  
  
U\_max <- iquery(dbConnect, "aggregate(RATINGS, max(userId))", return = T)$userId\_max  
M\_max <- iquery(dbConnect, "aggregate(RATINGS, max(movieID))", return = T)$movieID\_max  
  
query <- sprintf("redimension(RATINGS, <rating:double NOT NULL>[userId=0:%d,?,0,movieID=0:%d,?,0])",  
 U\_max, M\_max)  
iquery(dbConnect, sprintf("store(%s, RATINGS\_MATRIX)", query)

## Essential statistics using SciDB

We show some essential statistical methods using SciDB functions.

What is the average rating of each user?

userAvg <- iquery(dbConnect, "aggregate(ratings\_matrix, avg(rating), userId)", return = T)  
head(userAvg)

## userId rating\_avg  
## 1 1 2.550000  
## 2 2 3.486842  
## 3 3 3.568627  
## 4 4 4.348039  
## 5 5 3.910000  
## 6 6 3.261364

Similarly, we can obtain the the average average rating of each movie.

movieAvg <- iquery(dbConnect, "aggregate(ratings\_matrix, avg(rating), movieID)", return = T)  
head(movieAvg)

## movieID rating\_avg  
## 1 1 3.872470  
## 2 2 3.401869  
## 3 3 3.161017  
## 4 4 2.384615  
## 5 5 3.267857  
## 6 6 3.884615

A note of caution, the averages are computed only on non-zero ratings.

Other aggregation functions include sum, min, max, stdev, etc.

# Clustering Users in the Small Dataset

Using the small dataset, we'll cluster users in 2 ways:

1. Compute pairwise correlation (over movieIDs) between users. Apply hierarchical clustering on the correlation matrix.
2. Compute the SVD of the ratings matrix and keep the top 10 singular values. Apply kmeans clustering on the top 10 singular component coefficients.

The correlation matrix and SVD will be computed using SciDB functionality, while kmeans and hierarchical clustering will be computed using base R functionality.

## Clustering using the correlation matrix

Now, let's try to compute the correlation between users. Since userId is in rows, we just have to normalize the rows and matrix multiply .

The SciDB function for sparse matix multiply is spgemm, in the linear\_algebra library.

iquery(dbConnect, "load\_library('linear\_algebra')")

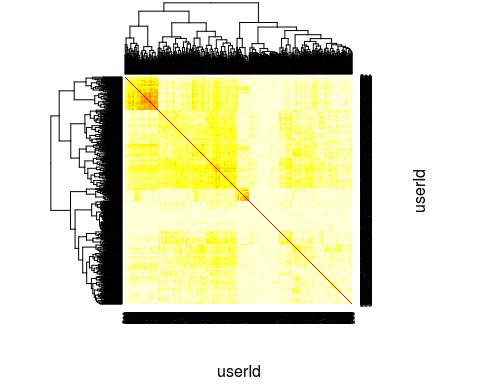
iquery(dbConnect, "store(spgemm(ratings\_matrix, transpose(ratings\_matrix)), ratings\_matrix\_MxMT)")

# Average user rating  
query\_userAvg <- "apply(aggregate(ratings\_matrix, sum(rating), userId), rating\_avg, rating\_sum/163949.0)"  
iquery(dbConnect, sprintf("store(project(%s, rating\_avg), ratings\_userAvg)", query\_userAvg))  
  
# Mean sum of square ratings  
query\_rating\_sq <- "apply(ratings\_matrix, rating\_sq, rating \* rating)"  
query\_rating\_MSS <- sprintf("apply(aggregate(%s, sum(rating\_sq), userId), rating\_MSS, rating\_sq\_sum/163949.0)",  
 query\_rating\_sq)  
iquery(dbConnect, sprintf("store(project(%s, rating\_MSS), ratings\_userMSS)", query\_rating\_MSS))  
  
# Cross join with ratings matrix  
query\_crossjoin <- "cross\_join(ratings\_matrix\_MxMT as A, ratings\_userAvg as B, A.userId, B.userId)"  
query\_crossjoin <- sprintf("cross\_join(%s as A, ratings\_userAvg as B, A.userId2, B.userId)",   
 query\_crossjoin)  
query\_crossjoin <- sprintf("cross\_join(%s as A, ratings\_userMSS as B, A.userId, B.userId)",  
 query\_crossjoin)  
query\_crossjoin <- sprintf("cross\_join(%s as A, ratings\_userMSS as B, A.userId2, B.userId)",  
 query\_crossjoin)  
iquery(dbConnect, sprintf("store(%s, ratings\_crossjoin)", query\_crossjoin))  
  
# Compute correlation matrix from M\*M^T  
query <- "apply(ratings\_crossjoin, correlation,   
(multiply/163949.0 - rating\_avg \* rating\_avg\_2) /   
sqrt((rating\_MSS - pow(rating\_avg, 2)) \* (rating\_MSS\_2 - pow(rating\_avg\_2, 2))))"  
  
iquery(dbConnect,   
 sprintf("store(project(%s, correlation), ratings\_correlationMatrix)", query))

correlationMatrix <- iquery(dbConnect,   
 "filter(ratings\_correlationMatrix, TRUE)",   
 return = T)  
  
library(Matrix, quietly = T)  
correlationMatrix <- spMatrix(nrow = 671, ncol = 671,   
 i = correlationMatrix$userId,   
 j = correlationMatrix$userId2,  
 x = correlationMatrix$correlation)

We visualize the correlation matrix using the heatmap function in R. (Actually, we'll visualize 1-correlationMatrix, which is the dissimilarity matrix.) This function also automatically computes the dendrogram and reorders the rows and columns to produce a visual clustering of the userIds.

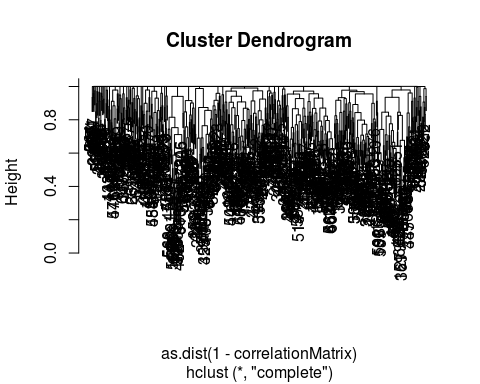
heatmap(as.matrix(1 - correlationMatrix), symm = T, xlab = "userId", ylab = "userId")



### Hierarchical clustering

We now perform hierarchical clustering on the dissimilarity matrix 1-correlationMatrix.

hc <- hclust(as.dist(1 - correlationMatrix))  
plot(hc)



## Clustering using Principle Component Analysis and the SVD

Next, we perform the SVD on the ratings matrix to obtain the top princple components of the ratings data.

SciDB provides the gesvd function in the dense\_linear\_algebra library, for computing the SVD of a dense matrix. To use this function, first load the library.

iquery(dbConnect, "load\_library('dense\_linear\_algebra')")

Next, we center the data at 0, a requirement of the PCA.

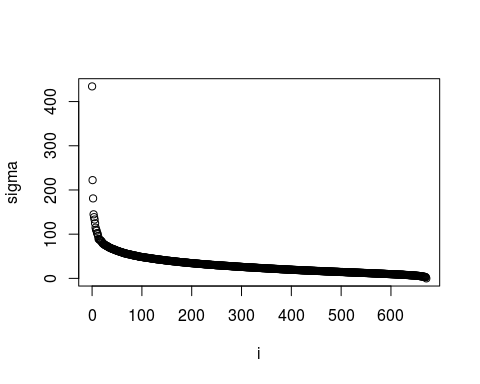
query\_userSum <- "aggregate(ratings\_matrix, avg(rating), movieID)"  
query\_crossjoin <- sprintf("cross\_join(ratings\_matrix as A, %s as B, A.movieID, B.movieID)",   
 query\_userSum)  
query\_center <- sprintf("project(apply(%s, rating\_centered, rating-rating\_avg), rating\_centered)",   
 query\_crossjoin)  
  
ratings\_matrix\_centered <- iquery(dbConnect,   
 sprintf("store(%s, ratings\_matrix\_centered)", query\_center),   
 return = T)

Now, we are ready to compute the singular values of the ratings matrix.

ratings\_singularValues <- iquery(dbConnect,   
 "store(gesvd(ratings\_matrix\_centered, 'S'), ratings\_singularValues)",   
 return = T)  
save(ratings\_singularValues, file="/home/scidb/Project/ratings\_singularValues.Rdata")

By inspecting the plot of the singular values, we see that the top 3 singular values capture the most variation in the data.

load("/home/scidb/Project/ratings\_singularValues.Rdata")  
  
## For logistical reasons, we have saved the SVD results as Rdata format.   
## Equivalently, the singular values can be retrieved from the SciDB database  
## using the following command:  
# ratings\_singularValues <- iquery(dbConnect, "filter(ratings\_singularValues, TRUE)", return = T)  
  
plot(ratings\_singularValues)



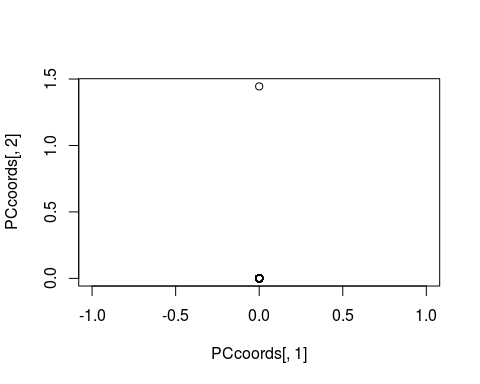
We also need the left and right singular vectors. We'll get SciDB to compute and store the singular vectors, and then extract only the first 10 singular vectors into R.

ratingsSvdU <- iquery(dbConnect, "store(gesvd(ratings\_matrix\_centered, 'U'), ratingsSvdU)", return = T)  
ratingsSvdU[[1]]=as.integer(ratingsSvdU[[1]])  
ratingsSvdU[[2]]=as.integer(ratingsSvdU[[2]])  
save(ratingsSvdU,file="/home/scidb/Project/ratingsSvdU.Rdata")  
  
ratingsSvdVt <- iquery(dbConnect, "store(gesvd(ratings\_matrix\_centered, 'VT'), ratingsSvdVt)", return = T)  
save(ratingsSvdVt,file="/home/scidb/Project/ratingsSvdVt.Rdata")

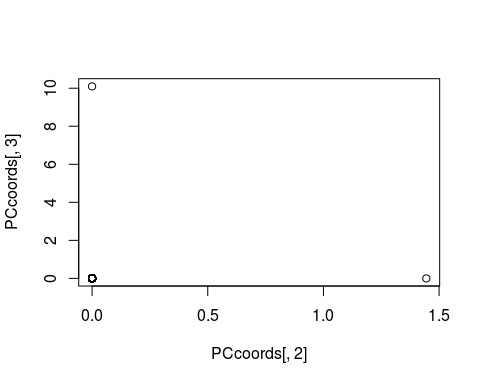
query <- "apply(apply(ratings\_singularValues, i\_2, i\_1-1), j, i\_2)" # Please check if i or i\_1 is the dimension name, and whether it starts at 0 or 1?  
query <- sprintf("redimension(%s, <sigma:double>[i\_2=0:671,?,0,j=0:671,?,0])", query)   
iquery(dbConnect, sprintf("store(%s, ratingsSvdSdiag)", query))  
iquery(dbConnect, "store(spgemm(ratingsSvdU, ratingsSvdSdiag), ratingsSvdUxS)")

load("/home/scidb/Project/ratingsSvdU.Rdata")  
  
uMat <- spMatrix(nrow = 672, ncol = 672,   
 i = ratingsSvdU$userId + 1,   
 j = ratingsSvdU$i + 1,   
 x = ratingsSvdU$u)  
sigMat <- Diagonal(x = c(0,ratings\_singularValues$sigma))  
# The +1 is due to difference in 0-indexing in SciDB and 1-indexing in R  
  
PCcoords <- uMat \* sigMat[1:(nrow(sigMat)-1), 1:(nrow(sigMat)-1)]  
  
# ratings\_U10 <- iquery(dbConnect, "subarray(ratings\_U, null, 0, null, 4)", return = T)  
# ratings\_VT10 <- iquery(dbConnect, "subarray(ratings\_VT, null, 0, null, 4)", return = T)

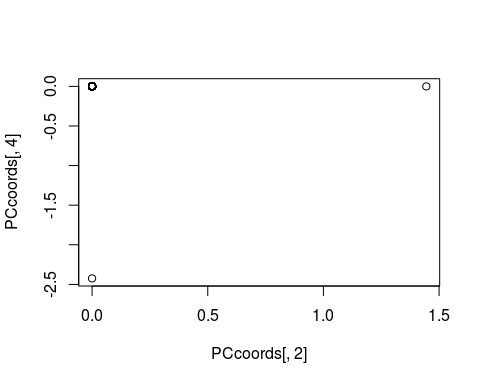
plot(PCcoords[, 1], PCcoords[, 2])



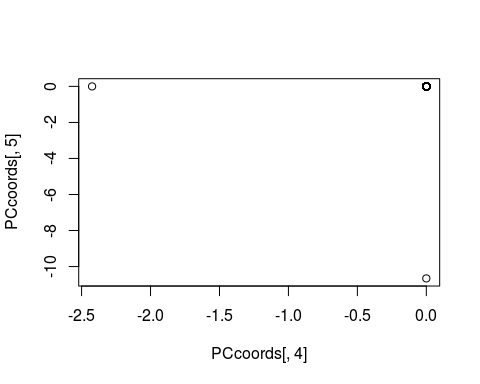
plot(PCcoords[, 2], PCcoords[, 3])



plot(PCcoords[, 2], PCcoords[, 4])

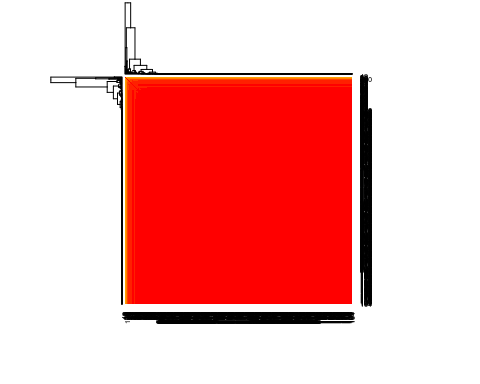


plot(PCcoords[, 4], PCcoords[, 5])

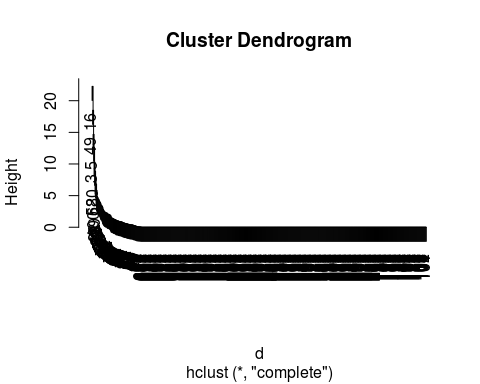


### kmeans clustering

d <- dist(PCcoords[, 1:100])  
heatmap(as.matrix(d), symm = T)



hclustTree <- hclust(d)  
plot(hclustTree)



# Big Data Computations with the Full Dataset

The computations in the previous section extend to using the full dataset. However, due to limited RAM and hard disk space on our laptops, we are unable to run the identical computations for the full dataset.

We can still show a tractable example: finding the pairwise correlations between movies.

system.time(  
 iquery(dbConnect,   
 "store(spgemm(transpose(RATINGS\_MATRIX), RATINGS\_MATRIX), RATINGS\_MATRIX\_MxMT)")  
)