# Unit 7: Databases and Big Data

October 13, 2015

References:

- Murrell: Introduction to Data Technologies
- Adler: R in a Nutshell
- Spark Programming Guide

I've also pulled material from a variety of other sources, some mentioned in context below.

Note that for a lot of the demo code I ran the code separately outside of *knitr* and this document because of the time involved in working with large datasets.

## 1 A few preparatory notes

### 1.1 An editorial on 'big data'

Big data is trendy these days.

Personally, I think some of the hype is justified and some is hype. Large datasets allow us to address questions that we can't with smaller datasets, and they allow us to consider more sophisticated (e.g., nonlinear) relationships than we might with a small dataset. But they do not directly help with the problem of correlation not being causation. Having medical data on every American still doesn't tell me if higher salt intake causes hypertension. Internet transaction data does not tell me if one website feature causes increased viewership or sales. One either needs to carry out a designed experiment or think carefully about how to infer causation from observational data. Nor does big data help with the problem that an ad hoc 'sample' is not a statistical sample and does not provide the ability to directly infer properties of a population. A well-chosen smaller dataset may be much more informative than a much larger, more ad hoc dataset. However, having big datasets might allow you to select from the dataset in a way that helps get at causation or in a way that allows you to construct a population-representative sample. Finally, having a big dataset also

allows you to do a large number of statistical analyses and tests, so multiple testing is a big issue. With enough analyses, something will look interesting just by chance in the noise of the data, even if there is no underlying reality to it.

Here's a different way to summarize it.

Different people define the 'big' in big data differently. One definition involves the actual size of the data. Our efforts here will focus on dataset sizes that are large for traditional statistical work but would probably not be thought of as large in some contexts such as Google or the NSA. Another definition of 'big data' has more to do with how pervasive data and empirical analyses backed by data are in society and not necessarily how large the actual dataset size is.

## 1.2 Logistics

One of the main drawbacks with R in working with big data is that all objects are stored in memory, so you can't directly work with datasets that are more than 1-20 Gb or so, depending on the memory on your machine.

Note: in handling big data files, it's best to have the data on the local disk of the machine you are using to reduce traffic and delays from moving data over the network.

## 1.3 What we already know about handling big data!

UNIX operations are generally very fast, so if you can manipulate your data via UNIX commands and piping, that will allow you to do a lot. We've already seen UNIX commands for extracting columns. And various commands such as *grep*, *head*, *tail*, etc. allow you to pick out rows based on certain criteria. As some of you have done in problem sets, one can use *awk* to extract rows. So basic shell scripting may allow you to reduce your data to a more manageable size.

And don't forget simple things. If you have a dataset with 30 columns that takes up 10 Gb but you only need 5 of the columns, get rid of the rest and work with the smaller dataset. Or you might be able to get the same information from a random sample of your large dataset as you would from doing the analysis on the full dataset. Strategies like this will often allow you to stick with the tools you already know.

Also, the example datasets in Section 3 are not good illustrations of this, but as we'll see scattered throughout the Unit and as we saw in Unit 3, there are more compact ways of storing data than in flat text (e.g., csv) files.

# 2 Databases

## 2.1 Overview

A relational database stores data as a set of tables (or relations), which are rather similar to R data frames, in that a table is made up of columns or fields, each containing a single type (numeric, character, date, currency, ...) and rows or records containing the observations for one entity. One principle of databases is that if a category is repeated in a given variable, you can more efficiently store information about each level of the category in a separate table; consider information about people living in a state and information about each state - you don't want to include variables that only vary by state in the table containing information about individuals (at least until you're doing the actual analysis that needs the information in a single table). Or consider students nested within classes nested within schools. Databases are set up to allow for fast querying and merging (called *joins* in database terminology).

You can interact with databases in a variety of database systems (DBMS=database management system) (some systems are *SQLite*, *MySQL*, *postgreSQL*, *Oracle*, *Access*). We'll concentrate on accessing data in a database rather than management of databases. SQL is the *Structured Query Language* and is a special-purpose language for managing databases and making queries. Variations on SQL are used in many different DBMS.

Many DBMS have a client-server model. Clients connect to the server, with some authentication, and make requests. We'll concentrate here on a simple DBMS, *SQLite*, that allows us to just work on our local machine, with the database stored as a single file.

There are often multiple ways to interact with a DBMS, including directly using command line tools provided by the DBMS or via Python or R, among others.

We'll use an SQLite database available on any SCF machine at */server/web/scf/cis.db* as our example database. This is a database of the metadata (authors, titles, years, journal, etc.) for articles published in Statistics journals over the last century. First, let's talk through how one would set up a relational database to store journal article information.

## 2.2 Accessing databases in R

In R, the *DBI* package provides a front-end for manipulating databases from a variety of DBMS (MySQL, SQLite, Oracle, among others). Basically, you tell the package what DBMS is being used on the backend, link to the actual database, and then you can use the syntax in the package.

First we'll connect to the database and get some information on the *schema*, i.e., the structure of the database.

```
library(RSQLite)

## Loading required package:  DBI

fileName <- "/server/web/scf/cis.db"
drv <- dbDriver("SQLite")
db <- dbConnect(drv, dbname = fileName) # using a connection once again!
# con <- dbConnect(SQLite(), dbname = fileName) # alternative

# get information on the database schema
dbListTables(db)

##  [1] "articles"      "authors"       "authorships"   "books"
##  [5] "contacts"      "delayed_jobs"  "isbns"         "issns"
##  [9] "issues"        "journals"      "tag_relations" "taggings"
## [13] "tags"          "volumes"

dbListFields(db, "articles")

##  [1] "id"         "type"       "id_entity"  "id_title"   "title"
##  [6] "year"       "volume"     "number"     "page_start" "page_end"
## [11] "url"        "journal"    "journal_id" "volume_id"  "issue_id"
## [16] "zmath"

dbListFields(db, "authors")

## [1] "id"   "name"

dbListFields(db, "authorships")

## [1] "id"               "id_title"         "author_id"
## [4] "editor"           "sequence"         "publication_id"
## [7] "publication_type"
```

For queries, SQL has statements like:

`SELECT var1, var2, var3 FROM tableX WHERE condition1 AND condition2 ORDER BY var4`

E.g., *condition1* might be `latitude > 80` or `name = 'Breiman'` or `company in ('IBM', 'Apple', 'Dell')`. Now we'll do some queries to pull together information we want. Be-

cause of the relational structure, to extract the titles for a given author, we need to do a series of queries.

```
auth <- dbSendQuery(db, "select * from authorships")
fetch(auth, 5)
```

```
##   id id_title author_id editor sequence publication_id publication_type
## 1  3        2         1      f        0              1        Article\n
## 2  4        3         3      f        0              2        Article\n
## 3  5        4         4      f        0              3        Article\n
## 4  6        5         5      f        0              4        Article\n
## 5  7        6         6      f        0              5        Article\n
```

```
dbClearResult(auth)
```

```
## [1] TRUE
```

```
query <- "select id from authors where name like 'Breiman%'"
a_ids <- dbGetQuery(db, query)


a_ids <- a_ids[ , 1]
a_ids
```

```
## [1]  532 1141
```

```
query <- paste("select id_title from authorships where author_id in (",
               paste(a_ids, collapse = ","), ")")
query
```

```
## [1] "select id_title from authorships where author_id in ( 532,1141 )"
```

```
t_ids <- dbGetQuery(db, query)
t_ids$id_title[1:5]
```

```
## [1]  593 1062 1087 1089 1440
```

```
t_ids <- t_ids[ , 1]
query <- paste("select * from articles where id_title in (",
               paste(t_ids, collapse = ","), ")")
titles <- dbGetQuery(db, query)
head(titles)
```

```
##     id    type  id_entity id_title
## 1  445 Article 1000000073      593
## 2  913 Article 1000000105     1062
## 3  938 Article 1000000105     1087
## 4  940 Article 1000000105     1089
## 5 1863 Article 1000000145     2156
## 6 2287 Article 1000000161     2580
##                                                                    tit
## 1 The individual ergodic theorem of information theory (Corr: V31 p809-8
## 2             The capacities of certain channel classes under random codi
## 3                                  On the completeness of order statisti
## 4              The strong law of large numbers for a class of Markov chai
## 5                                The Poisson tendency in traffic distorti
## 6                                    Consistent estimates and zero-one se
##   year volume number page_start page_end url journal journal_id volume_id
## 1 1957     28      0        809      811                 1748      7998
## 2 1960     31      0        558      567                 1748      9117
## 3 1960     31      0        794      797                 1748      9117
## 4 1960     31      0        801      803                 1748      9117
## 5 1963     34      0        308      311                 1748      9865
## 6 1964     35      0        157      161                 1748     10084
##   issue_id zmath
## 1       74    \n
## 2      106    \n
## 3      106    \n
## 4      106    \n
## 5      146    \n
## 6      162    \n

# do a google scholar check to see that things seem to be ok
```

Note that we were able to insert values from R into the set used to do the selection.

Now let's see a *join* (by default this is an "*inner join*" – see below) of multiple tables, combined with a query. This allows us to extract the information on Breiman's articles more easily.

```
# alternatively, we can do a query that involves multiple tables
info <- dbGetQuery(db, "select * from articles, authors, authorships where
   authors.name like 'Breiman%' and authors.id = authorships.author_id and
   authorships.id_title = articles.id_title")
# "select * from articles, authors, authorships where authors.name
#  like 'Breiman%' and authors.id = authorships.author_id and
#  authorships.id_title = articles.id_title"
head(info)
```

```
##     id    type  id_entity id_title
## 1  445 Article 1000000073      593
## 2  913 Article 1000000105     1062
## 3  938 Article 1000000105     1087
## 4  940 Article 1000000105     1089
## 5 1863 Article 1000000145     2156
## 6 2287 Article 1000000161     2580
##                                                                        tit
## 1 The individual ergodic theorem of information theory (Corr: V31 p809-81
## 2             The capacities of certain channel classes under random codi
## 3                                  On the completeness of order statisti
## 4              The strong law of large numbers for a class of Markov chai
## 5                                The Poisson tendency in traffic distorti
## 6                                    Consistent estimates and zero-one se
##   year volume number page_start page_end url journal journal_id volume_id
## 1 1957     28      0        809      811                  1748      7998
## 2 1960     31      0        558      567                  1748      9117
## 3 1960     31      0        794      797                  1748      9117
## 4 1960     31      0        801      803                  1748      9117
## 5 1963     34      0        308      311                  1748      9865
## 6 1964     35      0        157      161                  1748     10084
##   issue_id zmath  id           name   id id_title author_id editor
## 1       74    \n 532 Breiman, Leo\n  696      593       532      f
## 2      106    \n 532 Breiman, Leo\n 1355     1062       532      f
## 3      106    \n 532 Breiman, Leo\n 1391     1087       532      f
## 4      106    \n 532 Breiman, Leo\n 1393     1089       532      f
## 5      146    \n 532 Breiman, Leo\n 2847     2156       532      f
```

```
## 6      162    \n 532 Breiman, Leo\n 3413     2580       532      f
##   sequence publication_id publication_type
## 1        0            445        Article\n
## 2        1            913        Article\n
## 3        2            938        Article\n
## 4        0            940        Article\n
## 5        0           1863        Article\n
## 6        0           2287        Article\n
```

Finally, let's see the idea of creating a *view*, which you can think of as a new table, though the DBMS is not actually explicitly constructing such a table.

```
## [1] TRUE
```

```
# finally, we can create a view that amounts to joining the tables
fullAuthorInfo <- dbSendQuery(db, 'create view fullAuthorInfo as select *
     from authors join authorships on authorships.author_id = authors.id')
# 'create view fullAuthorInfo as select * from authors join
#  authorships on authorships.author_id = authors.id'

partialArticleInfo <- dbSendQuery(db, 'create view partialArticleInfo as
     select * from articles join fullAuthorInfo on
     articles.id_title=fullAuthorInfo.id_title')
# 'create view partialArticleInfo as select * from articles join
#  fullAuthorInfo on articles.id_title=fullAuthorInfo.id_title'

fullInfo <- dbSendQuery(db, 'select * from journals join partialArticleInfo
   on journals.id = partialArticleInfo.journal_id')
# 'select * from journals join partialArticleInfo on
#  journals.id = partialArticleInfo.journal_id')
subData <- fetch(fullInfo, 3)
subData
```

```
##     id                                  name articles_count min_year
## 1 1748 The Annals of Mathematical Statistics              0      \\N
## 2  452                          Econometrica              0      \\N
```

```
## 3 1746             The American Statistician                0      \\N
##   max_year                                publisher url mathscinet_id
## 1      \\N  Institute of Mathematical Statistics               \\N
## 2      \\N Blackwell Scientific Publications Ltd               \\N
## 3      \\N      American Statistical Association               \\N
##   english_only electronic_only url_only publisher_society admin_comments
## 1          \\N             \\N      \\N               \\N            \\N
## 2          \\N             \\N      \\N               \\N            \\N
## 3          \\N             \\N      \\N               \\N            \\N
##    core id    type  id_entity id_title
## 1 \\N\n  1 Article 1000000001        2
## 2 \\N\n  2 Article 1000000002        3
## 3 \\N\n  3 Article 1000000003        4
##
## 1 The non-central Wishart distribution and certain problems of multivaria
## 2                  Capital expansion, rate of growth, and employment (Re
## 3
##   year volume number page_start page_end url journal journal_id volume_ic
## 1 1946     17      0        409      431                 1748      4170
## 2 1946     14      0        137      147                  452      2723
## 3 1947      1      0          7       11                 1746       273
##   issue_id zmath id:1                  name id:2 id_title:1 author_id
## 1        2    \n    1      Anderson, T. W.\n    3          2         1
## 2        3    \n    3      Domar, Evsey D.\n    4          3         3
## 3        4    \n    4 Tumbleson, Robert C.\n    5          4         4
##   editor sequence publication_id publication_type
## 1      f        0              1        Article\n
## 2      f        0              2        Article\n
## 3      f        0              3        Article\n
```

```
dbClearResult(fullInfo)
```

```
## [1] TRUE
```

As seen above, you can also use *dbSendQuery()* combined with *fetch()* to pull in a fixed number of records at a time, if you're working with a big database.

## 2.3 Details on joins

A bit more on joins - as we saw with *merge()* in R, there are various possibilities for how to do the merge depending on whether there are rows in one table that are not in another table. In other words, we need to think about whether the relationship between tables is one-to-one, one-to-many, or many-to-many. In database terminology an *inner join* is when you get the rows for which there is data in both tables. A *left outer join* gives all the rows from the first table but only those from the second table that match a row in the first table. A *right outer join* is the reverse, while a *full outer join* returns all rows from both tables. A *cross join* gives the Cartesian product, namely the combination of every row from each table, analogous to *expand.grid()* in R. However a *cross join* with a *where* statement can duplicate the result of an *inner join*:

```
select * from table1 cross join table2 where table1.id = table2.id
select * from table1 join table2 on table1.id = table2.id
```

## 2.4 Keys and indices

A key is a field or collection of fields that gives a unique value for every row/observation. A table in a database should then have a primary key that is the main unique identifier used by the DBMS. Foreign keys are columns in one table that give the value of the primary key in another table.

An index is an ordering of rows based on one or more fields. DBMS use indices to look up values quickly. (Recall our discussion in Unit 4 on looking up values by name vs. index and the benefits of hashing.) So in general you want your tables to have indices. And having indices on the columns used in the matching for a join allows for quick joins. DBMS use indexing to provide sub-linear time lookup, so that lookup is faster than linear time ($O(n)$ when there are $n$ rows), which is what would occur if one had to look at each row sequentially. Lookup may be logarithmic [$O(log(n))$] or constant time [$O(1)$]. A binary search is logarithmic while looking up based on numeric position is $O(1)$.

So if you're working with a database and speed is important, check to see if there are indices.

## 2.5 Creating SQLite database tables from R

I won't do a full demo of this, but the basic syntax for this is as follows. You can read from a CSV to create the table or from an R dataframe. The following assumes you have two tables stored as CSVs, with one table of student info and one table of class info.

```
dbWriteTable(conn = db, name = "student", value = "student.csv",
   row.names = FALSE, header = TRUE)
```

```
dbWriteTable(conn = db, name = "class", value = "class.csv",
   row.names = FALSE, header = TRUE)
# alternatively
student <- read.csv("student.csv") # Read csv files into R
class <- read.csv("class.csv")
# Import data frames into database
dbWriteTable(conn = db, name = "student", value = student,
   row.names = FALSE)
dbWriteTable(conn = db, name = "class", value = class,
   row.names = FALSE)
```

## 2.6 SAS

SAS is quite good at handling large datasets, storing them on disk rather than in memory. I have used SAS in the past for subsetting and merging large datasets. Then I will generally extract the data I need for statistical modeling and do the analysis in R.

Here's an example of some SAS code for reading in a CSV followed by some subsetting and merging and then output.

```
/* we can use a pipe - in this case to remove carriage returns, */
/* presumably because the CSV file was created in Windows */
filename tmp pipe "cat ~/shared/hei/gis/100w4kmgrid.csv | tr -d '\r'";

/* read in one data file */
data grid;
infile tmp
lrecl=500 truncover dsd firstobs=2;
informat gridID x y landMask dataMask;
input gridID x y landMask dataMask;
run ;

filename tmp pipe "cat ~/shared/hei/GOES12/goes/Goes_int4km.csv | tr -d '\r

/* read in second data file */
data match;
infile tmp
```

```
lrecl=500 truncover dsd firstobs=2;
informat goesID gridID areaInt areaPix;
input goesID gridID areaInt areaPix;
run ;

/* need to sort before merging */
proc sort data=grid;
    by gridID;
run;
proc sort data=match;
    by gridID;
run;

/* notice some similarity to SQL */
data merged;
merge match(in=in1) grid(in=in2);
by gridID;  /* key field */
if in1=1;   /* also do some subsetting */
/* only keep certain fields */
keep gridID goesID x y landMask dataMask areaInt areaPix;
run;

/* do some subsetting */
data PA;   /* new dataset */
    set merged;  /* original dataset */
    if x<1900000 and x>1200000 and y<2300000 and y>1900000;
run;


%let filename="~/shared/hei/code/model/GOES-gridMatchPA.csv";
/* output to CSV */
PROC EXPORT DATA= WORK.PA
            OUTFILE= &filename
            DBMS=CSV REPLACE;
RUN;
```

Note that SAS is oriented towards working with data in a “data frame”-style format; i.e., rows

as observations and columns as fields, with different fields of possibly different types. As you can see in the syntax above, the operations concentrate on transforming one dataset into another dataset.

# 3 R and big data

There has been a lot of work in recent years to allow R to work with big datasets.

- The *data.table* package provides for fast operations on large data tables in memory. The *dplyr* package has also been optimized to work quickly on large data tables in memory, including operating on *data.table* objects from the *data.table* package.
- The *ff* and *bigmemory* packages provide the ability to load datasets into R without having them in memory, but rather stored in clever ways on disk that allow for fast access. Metadata is stored in R.
- The *biglm* package provides the ability to fit linear models and GLMs to big datasets, with integration with *ff* and *bigmemory*.
- Finally the *sqldf* package provides the ability to use SQL queries on R dataframes and on-the-fly when reading from CSV files. The latter can help you avoid reading in the entire dataset into memory in R if you just need a subset of it.

In this section we'll use an example of US government data on airline delays (1987-2008) available through the ASA 2009 Data Expo at http://stat-computing.org/dataexpo/2009/the-data.html.

First we'll use UNIX tools to download the individual yearly CSV files and make a single CSV (~12 Gb). See the demo code file for the bash code. Note that it's much smaller when compressed (1.7 Gb) or if stored in a binary format.

## 3.1 Working quickly with big datasets in memory: data.table

In many cases, particularly on a machine with a lot of memory, R might be able to read the dataset into memory but computations with the dataset may be slow.

The *data.table* package provides a lot of functionality for fast manipulation: indexing, merges/joins, assignment, grouping, etc.

Let's read in the airline dataset, specifying the column classes so that *fread()* doesn't have to detect what they are. I'll also use factors since factors are represented numerically. It only takes about 5 minutes to read the data in. We'll see in the next section that this is much faster than with other approaches within R.

```
require(data.table)
fileName <- '/tmp/AirlineDataAll.csv'

dt <- fread(fileName, colClasses=c(rep("numeric", 8), "factor",
                                   "numeric", "factor", rep("numeric", 5),
                                   rep("factor", 2), rep("numeric", 4),
                                   "factor", rep("numeric", 6)))
#Read 123534969 rows and 29 (of 29) columns from
#    11.203 GB file in 00:05:16


class(dt)
# [1] "data.table" "data.frame"
```

Now let's do some basic subsetting. We'll see that setting a key and using binary search can improve lookup speed dramatically.

```
system.time(sfo <- subset(dt, Origin == "SFO"))
## 8.8 seconds
system.time(sfoShort <- subset(dt, Origin == "SFO" & Distance < 1000))
## 12.7 seconds

system.time(setkey(dt, Origin, Distance))
## 33 seconds:
## takes some time, but will speed up later operations
tables()
##      NAME             NROW    MB
##[1,] dt       123,534,969 27334
##[2,] sfo        2,733,910   606
##[3,] sfoShort   1,707,171   379
##      COLS
##[1,] Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,CRSArrTime
##[2,] Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,CRSArrTime
##[3,] Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,CRSArrTime
##      KEY
##[1,] Origin,Distance
```

```
##[2,]
##[3,]
##Total: 28,319MB

## vector scan
system.time(sfo <- subset(dt, Origin == "SFO"))
## 8.5 seconds
system.time(sfoShort <- subset(dt, Origin == "SFO" & Distance < 1000 ))
## 12.4 seconds

## binary search
system.time(sfo <- dt[.('SFO'), ])
## 0.8 seconds
```

Setting a key in *data.table* simply amounts to sorting based on the columns provided, which allows for fast lookup later using binary search algorithms, as seen with the last query. Think about the analogy of looking up by name vs. index that we discussed in Unit 4. From my fairly quick look through the *data.table* documentation I don't see a way to do the subsetting with distance less than 1000 using the specialized functionality of *data.table*.

There's a bunch more to *data.table* and you'll have to learn a modest amount of new syntax, but if you're working with large datasets in memory, it will probably be well worth your while. Plus *data.table* objects are data frames (i.e., they inherit from data frames) so they are compatible with R code that uses dataframes.

## 3.2 Working with big datasets on disk: ff and bigmemory

Note that with our 12 Gb dataset, the data took up 27 Gb of RAM on the SCF server *radagast*. Operations on the dataset would then use up additional RAM. So this would not be feasible on most machines. And of course other datasets might be so big that even *radagast* wouldn't be able to hold them in memory.

### 3.2.1 ff

Now we can read the data into R using the *ff* package, in particular reading in as an *ffdf* object. Note the arguments are similar to those for *read.{table,csv}()*. *read.table.ffdf()* reads the data in chunks.

```
require(ff)
require(ffbase)

# I put the data file on local disk on the machine I am using
# (/tmp on radagast)
# it's good to test with a small subset before
# doing the full operations
fileName <- '/tmp/test.csv'
dat <- read.csv.ffdf(file = fileName, header = TRUE,
     colClasses = c('integer', rep('factor', 3),
     rep('integer', 4), 'factor', 'integer', 'factor',
     rep('integer', 5), 'factor','factor', rep('integer', 4),
     'factor', rep('integer', 6)))


fileName <- '/tmp/AirlineDataAll.csv'
system.time(  dat <- read.csv.ffdf(file = fileName, header = TRUE,
    colClasses = c('integer', rep('factor', 3), rep('integer', 4),
    'factor', 'integer', 'factor', rep('integer', 5), 'factor',
    'factor', rep('integer', 4), 'factor', rep('integer', 6))) )
## takes about 22 minutes

system.time(ffsave(dat, file = '/tmp/AirlineDataAll'))
## takes 11 minutes
## file is saved (in a binary format) as AirlineDataAll.ffData
## with metadata in AirlineDataAll.RData

rm(dat) # pretend we are in a new R session

system.time(ffload('/tmp/AirlineDataAll'))
# this is much quicker:
# 107 seconds
```

In the above operations, we wrote a copy of the file in the ff binary format that can be read more quickly back into R than the original reading of the CSV using *ffsave()* and *ffload()*. Also note the reduced size of the binary format file compared to the original CSV. It's good to be aware of where

the binary ff file is stored given that for large datasets, it will be large. With *ff* (I think *bigmemory* is different in how it handles this) it appears to be stored in */tmp* in an R temporary directory. Note that as we work with large files we need to be more aware of the filesystem, making sure in this case that */tmp* has enough space.

Let's look at the *ff* and *ffbase* packages to see what functions are available using `library(help=ff)`. Notice that there is an *merge.ffdf()*.

Note that a copy of an *ff* object appears to be a shallow copy.

Next let's do a bit of exploration of the dataset. Of course in a real analysis we'd do a lot more and some of this would take some time.

```
ffload('/tmp/AirlineDataAll')
# [1] "tmp/RtmpU5Uw6z/ffdf4e684aecd7c4.ff" "tmp/RtmpU5Uw6z/ffdf4e687fb73a88
# [3] "tmp/RtmpU5Uw6z/ffdf4e6862b1033f.ff" "tmp/RtmpU5Uw6z/ffdf4e6820053932
# [5] "tmp/RtmpU5Uw6z/ffdf4e681e7d2235.ff" "tmp/RtmpU5Uw6z/ffdf4e686aa01c8.
# ...

dat$Dest
# ff (closed) integer length=123534969 (123534969) levels: BUR LAS LAX OAK
# ABE ABQ ACV ALB ALO AMA ANC ATL AUS AVP AZO BDL BFL BGR BHM BIL BLI BNA BO
# CAK CCR CHS CID CLE CLT CMH CMI COS CPR CRP CRW CVG DAB DAL DAY DCA DEN DF
# EUG EVV EWR FAI FAR FAT FLG FLL FOE FSD GCN GEG GJT GRR GSO GSP GTF HNL HO
# ICT ILG ILM IND ISP JAN JAX JFK KOA LBB LEX LGA LGB LIH LIT LMT LNK MAF MI
# MFR MHT MIA MKE MLB MLI MOB MRY MSN MSP MSY OGG OKC OMA ONT ORD ORF PBI PH
# ...

# let's do some basic tabulation
DestTable <- sort(table.ff(dat$Dest), decreasing = TRUE)
# why do I need to call table.ff() and not table()?

# takes a while

#    ORD     ATL     DFW     LAX     PHX     DEN     DTW     IAH     MSP

# 6638035 6094186 5745593 4086930 3497764 3335222 2997138 2889971 2765191 2

#    STL     EWR     LAS     CLT     LGA     BOS     PHL     PIT     SLC
```

```
#  2720250 2708414 2629198 2553157 2292800 2287186 2162968 2079567 2004414 1

# looks right - the busiest airports are ORD (O'Hare in Chicago) and ATL (A

dat$DepDelay[1:50]
#opening ff /tmp/RtmpU5Uw6z/ffdf4e682d8cd893.ff
#  [1] 11 -1 11 -1 19 -2 -2  1 14 -1  5 16 17  1 21  3 13 -1 87 19 31 17 32
# [26] 29 26 15  5 54  0 25 -2  0 12 14 -1  2  1 16 15 44 20 15  3 21 -1  0

min.ff(dat$DepDelay, na.rm = TRUE)
# [1] -1410
max.ff(dat$DepDelay, na.rm = TRUE)
# [1] 2601

# tmp <- clone(dat$DepDelay) # make a deep copy
```

Let's review our understanding of S3 methods. Why did I need to call *table.ff()* rather than just simply calling *table()* on the ff object?

A note of caution. Debugging code involving *ff* can be a hassle because the size gets in the way in various ways. Until you're familiar with the various operations on ff objects, you'd be wise to try to run your code on a small test dataset loaded in as an ff object. Also, we want to be sure that the operations we use keep any resulting large objects in the *ff* format and use *ff* methods and not standard R functions.

### 3.2.2 bigmemory

The *bigmemory* package is an alternative way to work with datasets in R that are kept stored on disk rather than read entirely into memory. *bigmemory* provides a *big.matrix* class, so it appears to be limited to datasets with a single type for all the variables. However, one nice feature is that one can use *big.matrix* objects with *foreach* (one of R's parallelization tools, to be discussed soon) without passing a copy of the matrix to each worker. Rather the workers can access the matrix stored on disk.

### 3.2.3 sqldf

The *sqldf* package provides the ability to use SQL queries on data frames (via *sqldf()*) as well as to filter an input CSV via an SQL query (via *read.csv.sql()*), with only the result of the subsetting

put in memory in R. The full input data can be stored temporarily in an SQLite database on disk.

```
require(sqldf)
# read in file, with temporary database in memory
system.time(sfo <- read.csv.sql(fn,
      sql = "select * from file where Origin = 'SFO'",
      dbname=NULL, header = TRUE))
# read in file, with temporary database on disk
system.time(sfo <- read.csv.sql(fn,
      sql = "select * from file where Origin = 'SFO'",
      dbname=tempfile(), header = TRUE))
```

## 3.3 dplyr package

The *dplyr* package is the successor to the *plyr* package, providing plyr type functionality for data frames with enhancements for working with large tables and accessing databases (among other things). With *dplyr* one can work with data stored in the *data.table* format and in external databases.

```
# with database
cis <- src_sqlite("/tmp/cis.db")
authors <- tbl(cis, "authors")
authors

# with data.table
fileName <- '/tmp/AirlineDataAll.csv'
flights <- tbl_dt(fread(fileName, colClasses=c(rep("numeric", 8), "factor",
                              "numeric", "factor", rep("numeric", 5),
                              rep("factor", 2), rep("numeric", 4),
                              "factor", rep("numeric", 6))))

# now use dplyr functionality on 'authors' or 'flights'
# example analysis
summarize(group_by(flights, UniqueCarrier), mean(DepDelay, na.rm=TRUE))

# Source: local data table [29 x 2]
```

```
#
#   UniqueCarrier mean(DepDelay, na.rm = TRUE)
#1             PS                      8.928104
#2             TW                      7.658251
#3             UA                      9.667930
#4             WN                      9.077167
#5             EA                      8.674051
#6             HP                      8.107790
#7             NW                      6.007974
#8         PA (1)                      5.532442
#9             PI                      9.560336
#10            CO                      7.695967
#..           ...                           ...
```

## 3.4 Fitting models to big datasets: biglm

The *biglm* package provides the ability to fit large linear models and GLMs. *ffbase* has a *bigglm.ffdf()* function that builds on *biglm* for use with *ffdf* objects. Let's fit a basic model on the airline data. Note that we'll also fit the same model on the dataset when we use Spark at the end of the Unit.

```
require(ffbase)
require(biglm)

datUse <- subset(dat, ArrDelay < 60*12 & ArrDelay > (-30) &
                 !is.na(ArrDelay) & !is.na(Distance) & !is.na(DayOfWeek))
datUse$Distance <- datUse$Distance / 1000  # helps stabilize numerics
# 119971791 records

# any concern about my model?
system.time(mod <- bigglm(ArrDelay ~ Distance + DayOfWeek, data = datUse))
# 542.149  11.248 550.779
summary(mod)

coef <- summary(mod)$mat[,1]
```

Here are the results. Day 1 is Monday, so that's the baseline category for the ANOVA-like part of the model.

```
Large data regression model: bigglm(DepDelay ~ Distance + DayOfWeek, data =
Sample size =  119971791
               Coef    (95%     CI)     SE p
(Intercept)  6.3662  6.3504  6.3820 0.0079 0
Distance     0.7638  0.7538  0.7737 0.0050 0
DayOfWeek2  -0.6996 -0.7197 -0.6794 0.0101 0
DayOfWeek3   0.3928  0.3727  0.4129 0.0101 0
DayOfWeek4   2.2247  2.2046  2.2449 0.0101 0
DayOfWeek5   2.8867  2.8666  2.9068 0.0101 0
DayOfWeek6  -2.4273 -2.4481 -2.4064 0.0104 0
DayOfWeek7  -0.1362 -0.1566 -0.1158 0.0102 0
```

Of course as good statisticians/data analysts we want to do careful assessment of our model, consideration of alternative models, etc. This is going to be harder to do with large datasets than with more manageable ones. However, one possibility is to do the diagnostic work on subsamples of the data.

Now let's consider the fact that very small substantive effects can be highly statistically significant when estimated from a large dataset. In this analysis the data are generated from $Y \sim \mathcal{N}(0 + 0.001x, 1)$, so the $R^2$ is essentially zero.

```
n <- 150000000  # n*4*8/1e6 Mb of RAM (~5 Gb)
# but turns out to be 11 Gb as a text file
nChunks <- 100
chunkSize <- n/nChunks

set.seed(0)

for(p in 1:nChunks) {
  x1 <- runif(chunkSize)
  x2 <- runif(chunkSize)
  x3 <- runif(chunkSize)
  y <- rnorm(chunkSize, .001*x1, 1)
  write.table(cbind(y,x1,x2,x3), file = '/tmp/signif.csv',
     sep = ',', col.names = FALSE,  row.names = FALSE,
```

```
      append = TRUE, quote = FALSE)
}


fileName <- '/tmp/signif.csv'
system.time(  dat <- read.csv.ffdf(file = fileName,
   header = FALSE, colClasses = rep('numeric', 4)))
# 922.213  18.265 951.204 -- timing is on an older machine than radagast

names(dat) <- c('y', 'x1','x2', 'x3')
ffsave(dat, file = '/tmp/signif')


system.time(ffload('/tmp/signif'))
# 52.323   7.856  60.802  -- timing is on an older machine

system.time(mod <- bigglm(y ~ x1 + x2 + x3, data = dat))
#  1957.358    8.900 1966.644  -- timing is on an older machine

options(digits = 12)
summary(mod)


# R^2 on a subset (why can it be negative?)
coefs <- summary(mod)$mat[,1]
wh <- 1:1000000
1 - sum((dat$y[wh] - coefs[1] + coefs[2]*dat$x1[wh] +
  coefs[3]*dat$x2[wh] + coefs[4]*dat$x3[wh])^2) /
  sum((dat$y[wh] - mean(dat$y[wh]))^2)
```

Here are the results:

```
Large data regression model: bigglm(y ~ x1 + x2 + x3, data = dat)
Sample size = 1.5e+08
               Coef         (95%       CI)          SE         p
(Intercept) -0.0001437 -0.0006601 0.0003727 0.0002582 0.5777919
x1           0.0013703  0.0008047 0.0019360 0.0002828 0.0000013
```

```
x2           0.0002371 -0.0003286 0.0008028 0.0002828 0.4018565
x3          -0.0002620 -0.0008277 0.0003037 0.0002829 0.3542728
### and here is the R^2 calculation (why can it be negative?)
[1] -1.111046828e-06
```

So, do I care the result is highly significant? Perhaps if I'm hunting the Higgs boson... As you have hopefully seen in statistics courses, statistical significance $\neq$ practical significance.

# 4 Sparsity

A lot of statistical methods are based on sparse matrices. These include:

- Matrices representing the neighborhood structure (i.e., conditional dependence structure) of networks/graphs.
- Matrices representing autoregressive models (neighborhood structure for temporal and spatial data)
- A statistical method called the *lasso* is used in high-dimensional contexts to give sparse results (sparse parameter vector estimates, sparse covariance matrix estimates)
- There are many others (I've been lazy here in not coming up with a comprehensive list, but trust me!)

When storing and manipulating sparse matrices, there is no need to store the zeros, nor to do any computation with elements that are zero. A few of you exploited sparse matrices in PS4.

R, Matlab and Python all have functionality for storing and computing with sparse matrices. We'll see this a bit more in the linear algebra unit.

```
require(spam)
mat = matrix(rnorm(1e8), 1e4)
mat[mat > (-2)] <- 0
sMat <- as.spam(mat)
print(object.size(mat), units = 'Mb')

## 762.9 Mb

print(object.size(sMat), units = 'Mb')

## 26 Mb
```

```
vec <- rnorm(1e4)
system.time(mat %*% vec)

##    user  system elapsed
##   0.385   0.000   0.385

system.time(sMat %*% vec)

##    user  system elapsed
##   0.015   0.000   0.015
```

Here's a blog post describing the use of sparse matrix manipulations for analysis of the Netflix Prize data.

# 5 Using statistical concepts to deal with computational bottlenecks

As statisticians, we have a variety of statistical/probabilistic tools that can aid in dealing with big data.

1. Usually we take samples because we cannot collect data on the entire population. But we can just as well take a sample because we don't have the ability to process the data from the entire population. We can use standard uncertainty estimates to tell us how close to the true quantity we are likely to be. And we can always take a bigger sample if we're not happy with the amount of uncertainty.

2. There are a variety of ideas out there for making use of sampling to address big data challenges. One idea (due in part to Prof. Michael Jordan here in Statistics/EECS) is to compute estimates on many (relatively small) bootstrap samples from the data (cleverly creating a reduced-form version of the entire dataset from each bootstrap sample) and then combine the estimates across the samples. Here's the arXiv paper on this topic.

3. Randomized algorithms: there has been a lot of attention recently to algorithms that make use of randomization. E.g., in optimizing a likelihood, you might choose the next step in the optimization based on random subset of the data rather than the full data. Or in a regression context you might choose a subset of rows of the design matrix (the matrix of covariates) and corresponding observations, weighted based on the statistical leverage [recall the discussion

of regression diagnostics in a regression course] of the observations. Here's another arXiv paper that provides some ideas in this area.

# 6 Hadoop, MapReduce, and Spark

Here we'll talk about a fairly recent development in parallel computing. Traditionally, high-performance computing (HPC) has concentrated on techniques and tools for message passing such as MPI and on developing efficient algorithms to use these techniques.

## 6.1 Overview

A basic paradigm for working with big datasets is the *MapReduce* paradigm. The basic idea is to store the data in a distributed fashion across multiple nodes and try to do the computation in pieces on the data on each node. Results can also be stored in a distributed fashion.

A key benefit of this is that if you can't fit your dataset on disk on one machine you can on a cluster of machines. And your processing of the dataset can happen in parallel. This is the basic idea of *MapReduce*.

The basic steps of *MapReduce* are as follows:

- read individual data objects (e.g., records/lines from CSVs or individual data files)
- map: create key-value pairs using the inputs (more formally, the map step takes a key-value pair and returns a new key-value pair)
- reduce - for each key, do an operation on the associated values and create a result - i.e., aggregate within the values assigned to each key
- write out the {key,result} pair

A similar paradigm that is being implemented in some R packages by Hadley Wickham is the split-apply-combine strategy (http://www.jstatsoft.org/v40/i01/paper).

*Hadoop* is an infrastructure for enabling MapReduce across a network of machines. The basic idea is to hide the complexity of distributing the calculations and collecting results. Hadoop includes a file system for distributed storage (HDFS), where each piece of information is stored redundantly (on multiple machines). Calculations can then be done in a parallel fashion, often on data in place on each machine thereby limiting the amount of communication that has to be done over the network. Hadoop also monitors completion of tasks and if a node fails, it will redo the relevant tasks on another node. Hadoop is based on Java but there are projects that allow R to interact with Hadoop, in particular *RHadoop* and *RHipe*. *Rhadoop* provides the *rmr*, *rhdfs*, and *rhbase* packages.

For more details on *RHadoop* see Adler and http://blog.revolutionanalytics.com/2011/09/mapreduce-hadoop-r.html.

Setting up a Hadoop cluster can be tricky. Hopefully if you're in a position to need to use Hadoop, it will be set up for you and you will be interacting with it as a user/data analyst.

Ok, so what is Spark? You can think of Spark as in-memory Hadoop. Spark allows one to treat the memory across multiple nodes as a big pool of memory. So just as *data.table* was faster than *ff* because we kept everything in memory, Spark should be faster than Hadoop when the data will fit in the collective memory of multiple nodes. In cases where it does not, Spark will make use of the HDFS.

## 6.2 MapReduce and RHadoop

Let's see some examples of the MapReduce approach using R syntax of the sort one would use with *RHadoop*. While we'll use R syntax in the second piece of code below, the basic idea of what the map and reduce functions are is not specific to R. Note that using Hadoop with R may be rather slower than actually writing Java code for Hadoop.

First, let's consider a basic word-counting example. Suppose we have many, many individual text documents distributed as individual files in the HDFS. Here's pseudo code from Wikipedia. Here in the map function, the input {key,value} pair is the name of a document and the words in the document and the output {key, value} pairs are each word and the value 1. Then the reduce function takes each key (i.e., each word) and counts up the number of ones. The output {key, value} pair from the reduce step is the word and the count for that word.

```
function map(String name, String document):
// name (key): document name
// document (value): document contents
   for each word w in document:
      return (w, 1)

function reduce(String word, Iterator partialCounts):
// word (key): a word
// partialCounts (values): a list of aggregated partial counts
sum = 0
for each pc in partialCounts:
   sum += pc
return (word, sum)
```

Now let's consider an example where we calculate mean and standard deviation for the income of individuals in each state. Assume we have a large collection of CSVs, with each row containing information on an individual. *mapreduce()* and *keyval()* are functions in the *RHadoop* package. I'll assume we've written a separate helper function, *my_readline()*, that manipulates individual lines from the CSVs.

```
library(rmr)

mymap <- function(k, v) {
   record <- my_readline(v)
   key <- record[['state']]
   value <- record[['income']]
   keyval(key, value)
}

myreduce <- function(k, v){
   keyval(k, c(length(v), mean(v), sd(v)))
}

incomeResults <- mapreduce(
   input = "incomeData",
   map = mymap,
   reduce = myreduce,
   combine = NULL,
   input.format = 'csv',
   output.format = 'csv')

from.dfs(incomeResults, format = 'csv', structured = TRUE)
```

A few additional comments. In our map function, we could exclude values or transform them in some way, including producing multiple records from a single record. And in our reduce function, we can do more complicated analysis. So one can actually do fairly sophisticated things within what may seem like a restrictive paradigm. But we are constrained such that in the map step, each record needs to be treated independently and in the reduce step each key needs to be treated independently. This allows for the parallelization.

## 6.3 Spark

We'll focus on Spark rather than Hadoop for the speed reasons described above and because I think Spark provides a very nice environment in which to work. Plus it comes out of the AmpLab here at Berkeley. One downside is we'll have to know a bit of Python to use it.

### 6.3.1 Getting set up on Spark and the HDFS

We'll use Spark on an Amazon EC2 virtual cluster. Thankfully, Spark provides a Python-based script for setting up such a cluster. Occasionally the setup process goes awry but usually it's pretty easy. We need our Amazon authentication keys as well as public-private keypair for SSH. **Make sure you don't hard code your Amazon key information into any public file (including Github public repositories) - hackers will find the keys and use them to spin up instances, probably to mine bitcoin or send Spam; this happened in this class in 2014.**

We start by downloading the Spark package as a .tgz file (choosing the "source code" option) and untarring/zipping it. This all works from the VM. Also, on the SCF it's available on the Linux machines at */usr/local/src/pd/spark-1.4.0/spark-1.4.0/ec2*.

```
export SPARK_VERSION=1.5.1
export CLUSTER_SIZE=12  # number of slave nodes
export mycluster=sparkvm-paciorek # need unique name relative to other users

# I unzipped the Spark tarball to /usr/lib/spark via sudo on BCE
cd /usr/lib/spark/ec2

# set Amazon secret keys (manually or in my case by querying them elsewhere)
#AWS_ACCESS_KEY_ID=blah
AWS_ACCESS_KEY_ID=$(grep -i "^AWS_ACCESS_KEY_ID" ~/stat243-fall-2015-credent
#AWS_SECRET_ACCESS_KEY=blahhhh
AWS_SECRET_ACCESS_KEY=$(grep -i "^AWS_SECRET_ACCESS_KEY" ~/stat243-fall-201

### DO NOT HARD CODE YOUR AMAZON SECRET KEY INFORMATION INTO ANY PUBLIC FIL

# start cluster
./spark-ec2 -k chris_paciorek@yahoo.com:stat243-fall-2015 -i ~/.ssh/stat243-
 --region=us-west-2 -s ${CLUSTER_SIZE} -v ${SPARK_VERSION} launch ${mycluste
```

```
# login to cluster
# as root
./spark-ec2 -k ec2star -i ~/.ssh/stat243-fall-2015-ssh_key.pem --region=us-w
   login ${mycluster}
# or you can ssh in directly if you know the URL
# ssh -i ~/.ssh/stat243-fall-2015-ssh_key.pem root@ec2-54-71-204-234.us-west


# you can check your nodes via the EC2 management console

# to logon to one of the slaves, look at /root/ephemeral-hdfs/conf/slaves
# and ssh to that address
ssh `head -n 1 /root/ephemeral-hdfs/conf/slaves`

# We can view system status through a web browser interface

# on master node of the EC2 cluster, do:
MASTER_IP=`cat /root/ephemeral-hdfs/conf/masters`
echo ${MASTER_IP}
# Point a browser on your own machine to the result of the next command
# you'll see info about the "Spark Master", i.e., the cluster overall
echo "http://${MASTER_IP}:8080/"
# Point a browser on your own machine to the result of the next command
# you'll see info about the "Spark Stages", i.e., the status of Spark tasks
echo "http://${MASTER_IP}:4040/"
# Point a browser on your own machine to the result of the next command
# you'll see info about the HDFS"
echo "http://${MASTER_IP}:50070/"

# when you are done and want to shutdown the cluster:
#  IMPORTANT to avoid extra charges!!!
./spark-ec2 --region=us-west-2 --delete-groups destroy ${mycluster}
```

Next let's get the airline dataset onto the master node and then into the HDFS. Note that the file system commands are like standard UNIX commands, but you need to do `hadoop fs -` in front of the command. At the end of this chunk we'll start the Python interface for Spark.

```
export PATH=$PATH:/root/ephemeral-hdfs/bin/

hadoop fs -mkdir /data
hadoop fs -mkdir /data/airline

df -h
mkdir /mnt/airline
scp paciorek@smeagol.berkeley.edu:/scratch/users/paciorek/243/AirlineData/[1
   /mnt/airline
# for in-class demo:
# scp paciorek@smeagol.berkeley.edu:/scratch/users/paciorek/243/AirlineData/

hadoop fs -copyFromLocal /mnt/airline/*bz2 /data/airline

# check files on the HDFS, e.g.:
hadoop fs -ls /data/airline

# get numpy installed
# there is a glitch in the EC2 setup that Spark provides -- numpy is not ins
yum install python27-pip python27-devel
pip-2.7 install 'numpy==1.9.2'  # 1.10.1 has an issue with a warning in medi
/root/spark-ec2/copy-dir /usr/local/lib64/python2.7/site-packages/numpy

# pyspark is in /root/spark/bin
export PATH=${PATH}:/root/spark/bin
# start Spark's Python interface as interactive session
pyspark
```

### 6.3.2 Using Spark for pre-processing

Now we'll do some basic manipulations with the airline dataset. We'll count the number of lines/observations in our dataset. Then we'll do a map-reduce calculation that involves counting the number of flights by airline, so airline will serve as the key.

Note that all of the various operations are OOP methods applied to either the SparkContext management object or to a Spark dataset, called a Resilient Distributed Dataset (RDD). Here *lines* and *output* are both RDDs. However the result of *collect()* is just a standard Python object.

In the last step, let's compare how long it took to grab the SFO subset relative to the performance of R earlier in this Unit.

```
from operator import add
import numpy as np

lines = sc.textFile('/data/airline').cache()
numLines = lines.count()

# particularly for in-class demo - good to repartition the 3 files to more p
# lines = lines.repartition(96).cache()

# mapper
def stratify(line):
    vals = line.split(',')
    return(vals[16], 1)

result = lines.map(stratify).reduceByKey(add).collect()
# reducer is simply the addition function

# >>> result
#[(u'Origin', 22), (u'CIC', 7281), (u'LAN', 67897), (u'LNY', 289), (u'DAB',

# this counting by key could have been done
# more easily using countByKey()

vals = [x[1] for x in result]
sum(vals) == numLines  # a bit of a check
# True
[x[1] for x in result if x[0] == "SFO"]  # SFO result
# [2733910]

# if don't collect, can grab a few results
output = lines.map(stratify).reduceByKey(add)
output.take(5)
#[(u'Origin', 22), (u'CIC', 7281), (u'LAN', 67897), (u'LNY', 289), (u'DAB',
```

```
# also, you can have interim results stored as objects
mapped = lines.map(stratify)
result = mapped.reduceByKey(add).collect()


lines.filter(lambda line: "SFO" in line.split(',')[16]).saveAsTextFile('/dat

## make sure it's all in one chunk for easier manipulation on master
lines.filter(lambda line: "SFO" in line.split(',')[16]).repartition(1).saveA
#lines.filter(lambda line: "SFO" in line.split(',')[16]).repartition(1).
#saveAsTextFile('/data/airline-SFO2')
```

Let's consider some of the core methods we used. The Spark programming guide discusses these and a number of others.

- *map()*: take an RDD and apply a function to each element, returning an RDD
- *reduce()* and *reduceByKey()*: take an RDD and apply a reduction operation to the elements, doing the reduction stratified by the key values for *reduceByKey()*. Reduction functions need to be associative and commutative and take 2 arguments and return 1, all so that they can be done in parallel in a straightforward way.
- *filter()*: create a subset
- *collect()*: collect results back to the master
- *cache()*: tell Spark to keep the RDD in memory for later use
- *repartition()*: rework the RDD so it is in the specified number of chunks

Question: how many chunks do you think we want the RDD split into? What might the tradeoffs be?

Here's an example where we don't have a simple commutative/associative reducer function. Instead we group all the observations for each key into a so-called iterable object. Then our second map function treats each key as an element, iterating over the observations grouped within each key.

```
def computeKeyValue(line):
    vals = line.split(',')
    # key is carrier-month-origin-destination
    keyVals = '-'.join([vals[x] for x in [8,1,16,17]])
    if vals[0] == 'Year':
        return('0', [0,0,1,1])
    cnt1 = 1
    cnt2 = 1
    # 14 and 15 are arrival and departure delays
    if vals[14] == 'NA':
        vals[14] = '0'
        cnt1 = 0
    if vals[15] == 'NA':
        vals[15] = '0'
        cnt2 = 0
    return(keyVals, [int(vals[14]), int(vals[15]), cnt1, cnt2])


def medianFun(input):
    if len(input) == 2:  # input[0] should be key and input[1] set of values
        if len(input[1]) > 0:
            # iterate over set of values
            # input[1][i][0] is arrival delay
            # input[1][i][1] is departure delay
            m1 = np.median([val[0] for val in input[1] if val[2] == 1])
            m2 = np.median([val[1] for val in input[1] if val[3] == 1])
            return((input[0], m1, m2)) # m1, m2))
        else:
            return((input[0], -999, -999))
    else:
        return((input[0], -9999, -9999))


output = lines.map(computeKeyValue).groupByKey().cache()
medianResults = output.map(medianFun).collect()
medianResults[0:5]
```

```
# [(u'DL-8-PHL-LAX', 85.0, 108.0), (u'OO-12-IAH-CLL', -6.0, 0.0), (u'AA-4-LA
```

### 6.3.3 Using Spark for fitting models

Here we'll see the use of Spark to fit basic regression models in two ways. Warning: there may well be better algorithms to use and there may be better ways to implement these algorithms in Spark. But these work and give you the idea of how you can implement fitting within the constraints of a map-reduce paradigm.

Note that my first step is to repartition the data for better computational efficiency. Instead of having the data split into 22 year-specific chunks that vary in size (which is how things are initially because of the initial file structure), I'm going to split into a larger number of equal-size chunks to get better load-balancing.

**Linear regression via sufficient statistics** In the first algorithm we actually compute the sufficient statistics, which are simply $X^\top X$ and $X^\top Y$. Because the number of predictors is small, these are miniscule compared to the size of the dataset. This code has two ways of computing the matrices. The first treats each line as an observation and sums the $X_i^\top X_i$ and $X_i Y_i$ values across all observations. The second uses a map function that can operate on an entire partition, iterating through the elements of the partition, and computing $X_k^\top X_k$ and $X_k^\top Y_k$ for each partition, $k$. The second way is rather faster.

```
lines = sc.textFile('/data/airline')

def screen(vals):
    vals = vals.split(',')
    return(vals[0] != 'Year' and vals[14] != 'NA' and
           vals[18] != 'NA' and vals[3] != 'NA' and
           float(vals[14]) < 720 and float(vals[14]) > (-30) )
# 0 field is Year
# 14 field is ArrDelay
# 18 field is Distance
# 3 field is DayOfWeek

lines = lines.filter(screen).repartition(192).cache()
# 192 is a multiple of the total number of cores: 24 (12 nodes * 2 cores/no
```

```
n = lines.count()

import numpy as np
from operator import add

P = 8
bc = sc.broadcast(P)

########################
# calc xtx and xty
########################
def crossprod(line):
    vals = line.split(',')
    y = float(vals[14])
    dist = float(vals[18])
    dayOfWeek = int(vals[3])
    xVec = np.array([0.0] * P)
    xVec[0] = 1.0
    xVec[1] = float(dist)/1000
    if dayOfWeek > 1:
        xVec[dayOfWeek] = 1.0
    xtx = np.outer(xVec, xVec)
    xty = xVec * y
    return(np.c_[xtx, xty])

xtxy = lines.map(crossprod).reduce(add)
# 11 minutes

# now just solve system of linear equations!!

########################
# calc xtx and xty w/ mapPartitions
########################

# dealing with x matrix via mapPartitions
```

```
def readPointBatch(iterator):
    strs = list(iterator)
    matrix = np.zeros((len(strs), P+1))
    for i in xrange(len(strs)):
        vals = strs[i].split(',')
        dist = float(vals[18])
        dayOfWeek = int(vals[3])
        xVec = np.array([0.0] * (P+1))
        xVec[8] = float(vals[14]) # y
        xVec[0] = 1.0  # int
        xVec[1] = float(dist) / 1000
        if(dayOfWeek > 1):
            xVec[dayOfWeek] = 1.0
        matrix[i] = xVec
    return([matrix.T.dot(matrix)])

xtxyBatched = lines.mapPartitions(readPointBatch).reduce(add)
# 160 seconds

mle = np.linalg.solve(xtxy[0:P,0:P], xtxy[0:P,P])
```

**Linear regression via cyclic coordinate descent** In the second algorithm we pretend that we don't want to compute $X^\top X$, mimicing the situation in which we have too many predictors to either store $X^\top X$ or to do linear algebra on $X^\top X$. Instead we'll have a vector of starting values for $\beta$. Then we'll cycle through each element of $\beta$ and optimize that element, holding the others constant. The individual update steps look like this:

$$\beta_p^{t+1} = \frac{\sum_i r_i x_i}{\sum_i x_{ip}^2}$$

where $r_i$ the residual not including the $p$th predictor: $\sum_i \left(y_i - \sum_{j \neq p} x_{ij} \beta_j^t\right)$ and based on the current parameter values.

Once again, we'll use *mapPartitions()* to do the calculations we need for each step of the optimization piecewise on each partition (I use *batch* to describe this in the code). Note that the result after 10 iterations here is not the MLE, though most of the day of week effects are all shifted by a common amount relative to the mean when compared to the MLE.

```
def readPointPartition(iterator):
    strs = list(iterator)
    matrix = np.zeros((len(strs), P+1))
    print(len(strs))
    for i in xrange(len(strs)):
        vals = strs[i].split(',')
        dist = float(vals[18])
        dayOfWeek = int(vals[3])
        xVec = np.array([0.0] * (P+1))
        xVec[8] = float(vals[14]) # y
        xVec[0] = 1.0  # int
        xVec[1] = float(dist) / 1000
        if(dayOfWeek > 1):
            xVec[dayOfWeek] = 1.0
        matrix[i] = xVec
    return([matrix])

batches = lines.mapPartitions(readPointPartition).cache()
# 3 min

def denomSumSqPartition(mat):
    return((mat*mat).sum(axis=0))

def getNumPartition(mat):
    beta[p] = 0
    sumXb = mat[:, 0:P].dot(beta)
    return(sum((mat[:,P] - sumXb)*mat[:,p]))

sumx2 = batches.map(denomSumSqPartition).reduce(add)

beta = np.array([0.0] * P)
p = 0

oldBeta = beta.copy() # otherwise a shallow (i.e., pointer) copy!

it = 0
```

```
tol = .001
maxIts = 10
crit = 1e16

while crit > tol and it <= maxIts:
#for it in range(1,6):
    for p in xrange(P):
        # distribute current beta and current coordinate
        bc = sc.broadcast(beta)
        bc = sc.broadcast(p)
        # get numerator as product of residual and X for coordinate
        sumNum = batches.map(getNumPartition).reduce(add)
        beta[p] = sumNum / sumx2[p]
        print("Updated var " + str(p) + " in iteration ", str(it), ".")
    crit = sum(abs(beta - oldBeta))
    oldBeta = beta.copy()
    print("-"*100)
    print(beta)
    print(crit)
    print("-"*100)
    it = it+1

# 7 s per iteration;  ~9 minutes for 10 iterations
beta
#array([ 6.59246803,  0.76054724, -0.92357814,  0.16881708,  2.00073749,
#        2.66270618, -2.65116571, -0.36017589])
```

We've only done linear regression here, but similar coordinate descent computations can be done for GLMs and for Lasso type problems, and probably for other types of models. In fact coordinate descent is a (the?) standard algorithm for Lasso as discussed in this paper, which describes the methods in the *glmnet* package.

**Linear regression via gradient descent** Cycling through each coefficient has the obvious disadvantage of having to cycle through each coefficient, which would become particularly problematic in a model with more predictors. An alternative is to do gradient descent on the entire vector of

coefficients at once. For this problem the update at each step looks like this

$$\beta^{t+1} = \beta^t - \alpha \nabla L(\beta^t)$$

where $\nabla L(\beta^t)$ is the gradient vector of the log-likelihood evaluated at the current value, $\beta^t$. A key issue here, which we'll discuss in detail in the Unit on optimization is the step size, $\alpha$, also known as the learning rate. Here's the code for doing gradient descent on the entire set of coefficients.

```
alpha = .4

def sumVals(mat):
    return(sum(mat[:,P]))

beta = np.array([0.0] * P)

beta[0] = batches.map(sumVals).reduce(add) / n
oldBeta = beta.copy()

bc = sc.broadcast(P)
bc = sc.broadcast(beta)

def getGradBatch(mat):
    sumXb = mat[:, 0:P].dot(beta)
    return( ((sumXb - mat[:,P])*((mat[:, 0:P]).T)).sum(1) )

def ssqObj(mat):
    return ( (pow(mat[:,P] - mat[:, 0:P].dot(beta), 2)).sum() )

objValue = batches.map(ssqObj).reduce(add)

nIts = 100

storeVals = np.zeros((nIts, P+2))
tol = .001
maxIts = 100
crit = 1e16
```

```
while crit > tol and it < maxIts:
    gradVec = batches.map(getGradBatch).reduce(add)
    beta = beta - alpha*gradVec / n
    crit = sum(abs(beta - oldBeta))
    bc = sc.broadcast(beta)
    objValue = batches.map(ssqObj).reduce(add)
    oldBeta = beta.copy()
    storeVals[it, 0] = pow(objValue/n,0.5)
    storeVals[it, 1] = crit
    storeVals[it, 2:(P+2)] = beta
    print("-"*100)
    print(it)
    print(beta)
    print(crit)
    print(pow(objValue/n,0.5))
    print("-"*100)
    it = it + 1

# 15 min
#[ 6.57348292  0.75335604 -0.9251238   0.16222806  1.98565752  2.64468325
# -2.63650861 -0.36507276]
```

Note that I'm recording the value of the objective function to make sure that it is decreasing in every iteration, as one could set the learning rate such that that does not happen.

### 6.3.4 Final comments

**Running a batch Spark job** We can run a Spark job using Python code as a batch script rather than interactively. Here's an example, which computes the value of $\pi$ by Monte Carlo simulation (more on the general technique in the Unit on simulation). Assuming the script is named *piCalc.py*, we would call the script like this: `spark-submit piCalc.py 100000000 1000`

```
import sys
from pyspark import SparkContext
from numpy import random as rand
if __name__ == "__main__":
    sc = SparkContext()
```

```
    # use sys.argv to get arguments
    # for example:
    total_samples = int(sys.argv[1]) if len(sys.argv) > 1 else 1000000
    num_slices = int(sys.argv[2]) if len(sys.argv) > 2 else 2
    samples_per_slice = round(total_samples / num_slices)
    def sample(p):
        rand.seed(p)
        x, y = rand.random(samples_per_slice), rand.random(samples_per_slice
        # x, y = rand.random(samples_per_slice),
        #   rand.random(samples_per_slice)
        return sum(x*x + y*y < 1)

    count = sc.parallelize(xrange(0, num_slices), num_slices).map(sample).re
    #count = sc.parallelize(xrange(0, num_slices), num_slices).
    # map(sample).reduce(lambda a, b: a + b)
    print "Pi is roughly %f" % (4.0 * count / (num_slices*samples_per_slice)
```

This code again uses the idea that it's computationally more efficient to have each operation occur on a batch of data rather than an individual data point. So there are 1000 tasks and the total number of samples is broken up amongst those tasks. In fact, Spark has problems if the number of tasks gets too large.

**Python vs. Scala/Java** Spark is implemented natively in Java and Scala, so all calculations in Python involve taking Java data objects converting them to Python objects, doing the calculation, and then converting back to Java. This process is called serialization and takes time, so the speed when implementing your work in Scala (or Java) may be faster. Here's a small bit of info on that.

**sparkR** Finally, there is an R interface for Spark, but it's pretty new and not as widely used, so I didn't think it worth covering.