Unit 7: Databases and Big Data

October 8, 2018

References:

- SCF tutorial on “Working with large datasets in SQL, R, and Python”
- 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, and in some cases the speed with which it is collected. 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 US National Security Agency (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 and data size

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.

The techniques and tools discussed in this Unit (apart from the section on MapReduce/Spark) are designed for datasets in the range of gigabytes to tens of gigabytes, though they may scale to larger if you have a machine with a lot of memory or simply have enough disk space and are willing to wait. If you have 10s of gigabytes of data, you'll be better off if your machine has 10s of GBs of memory, as discussed in this Unit.

If you're scaling to 100s of GBs, terabytes or petabytes, tools such as Spark may be your best bet, or possibly carefully-administered databases.

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, remember that we can often store data more compactly in binary formats than in flat text (e.g., csv) files.

2 Databases

This material is drawn from the tutorial on "Working with large datasets in SQL, R, and Python", though I won't hold you responsible for all of the database/SQL material in that tutorial, only what appears here in this Unit.

2.1 Overview

Basically, standard SQL databases are *relational* databases that are a collection of rectangular format datasets (*tables*, also called *relations*), with each table similar to R or Pandas data frames, in that a table is made up of columns, which are called *fields* or *attributes*, each containing a single *type* (numeric, character, date, currency, enumerated (i.e., categorical), ...) and rows or records containing the observations for one entity. Some of the tables in a given database will generally have fields in common so it makes sense to merge (i.e., join) information from multiple tables. E.g., you might have a database with a table of student information, a table of teacher information and a table of school information, and you might join student information with information about the teacher(s) who taught the students. Databases are set up to allow for fast querying and merging (called joins in database terminology).

Formally, databases are stored on disk, while R and Python store datasets in memory. This would suggest that databases will be slow to access their data but will be able to store more data than can be loaded into an R or Python session. However, databases can be quite fast due in part to disk caching by the operating system as well as careful implementation of good algorithms for database operations. For more information about disk caching see the tutorial.

2.2 Interacting with a database

You can interact with databases in a variety of database systems (*DBMS*=database management system). Some popular systems are SQLite, MySQL, PostgreSQL, Oracle and Microsoft 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 high-level language for managing databases and making queries. Variations on SQL are used in many different DBMS.

Queries are the way that the user gets information (often simply subsets of tables or information merged across tables). The result of an SQL query is in general another table, though in some cases it might have only one row and/or one column.

Many DBMS have a client-server model. Clients connect to the server, with some authentication, and make requests (i.e., queries).

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 concentrate on SQLite (because it is simple to use on a single machine). SQLite is quite nice in terms of being self-contained - there is no server-client model, just a single file on your hard drive that stores the database and to which you can connect to using the SQLite shell, R, Python, etc. However, it does not have some useful functionality that other DBMS have. For example, you can't use ALTER TABLE to modify column types or drop columns.

2.3 Database schema and normalization

To truly leverage the conceptual and computational power of a database you'll want to have your data in a normalized form, which means spreading your data across multiple tables in such a way that you don't repeat information unnecessarily.

The schema is the metadata about the tables in the database and the fields (and their types) in those tables.

Let's consider this using an educational example. Suppose we have a school with multiple teachers teaching multiple classes and multiple students taking multiple classes. If we put this all in one table organized per student, the data might have the following fields:

- student ID
- student grade level
- student name
- class 1
- class 2
- ...
- class n
- grade in class 1
- grade in class 2

- ...
- grade in class n
- teacher ID 1
- teacher ID 2
- ...
- teacher ID n
- teacher name 1
- teacher name 2
- ...
- teacher name n
- teacher department 1
- teacher department 2
- ...
- teacher department n
- teacher age 1
- teacher age 2
- ...
- teacher age n

There are a lot of problems with this.

1. 'n' needs to be the maximum number of classes a student might take. If one ambitious student takes many classes, there will be a lot of empty data slots.
2. All the information about individual teachers (department, age, etc.) is repeated many times, meaning we use more storage than we need to.
3. If we want to look at the data on a per teacher basis, this is very poorly organized for that.

4. If one wants to change certain information (such as the age of a teacher) one needs to do it in many locations, which can result in errors and is inefficient.

It would get even worse if there was a field related to teachers for which a given teacher could have multiple values (e.g., teachers could be in multiple departments). This would lead to even more redundancy - each student-class-teacher combination would be crossed with all of the departments for the teacher (so-called multivalued dependency in database theory).

An alternative organization of the data would be to have each row represent the enrollment of a student in a class.

- student ID
- student name
- class
- grade in class
- student grade level
- teacher ID
- teacher department
- teacher age

This has some advantages relative to our original organization in terms of not having empty data slots, but it doesn't solve the other three issues above.

Instead, a natural way to order this database is with the following tables.

- Student
  - ID
  - name
  - grade_level
- Teacher
  - ID
  - name
  - department

  - age

- Class
  - ID
  - topic
  - class_size
  - teacher_ID
- ClassAssignment
  - student_ID
  - class_ID
  - grade

Then we do queries to pull information from multiple tables. We do the joins based on *keys*, which are the fields in each table that allow us to match rows from different tables.

(That said, if all anticipated uses of a database will end up recombining the same set of tables, we may want to have a denormalized schema in which those tables are actually combined in the database. It is possible to be too pure about normalization! We can also create a virtual table, called a *view*, as discussed later.)

2.3.1 Keys

A *key* is a field or collection of fields that give(s) 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. When information from multiple tables is joined together, the matching of a row from one table to a row in another table is generally done by equating the primary key in one table with a foreign key in a different table.

In our educational example, the primary keys would presumably be: *Student.ID*, *Teacher.ID*, *Class.ID*, and for ClassAssignment two fields: *{ClassAssignment.studentID, ClassAssignment.class_ID}*.

Some examples of foreign keys would be:

- student_ID as the foreign key in ClassAssignment for joining with Student on Student.ID
- teacher_ID as the foreign key in Class for joining with Teacher based on Teacher.ID
- class_ID as the foreign key in ClassAssignment for joining with Class based on Class.ID

2.3.2 Queries that join data across multiple tables

Suppose we want a result that has the grades of all students in 9th grade. For this we need information from the Student table (to determine grade level) and information from the ClassAssignment table (to determine the class grade). More specifically we need a query that joins *Student* with *ClassAssignment* based on *Student.ID* and *ClassAssignment.student_ID* and filters the rows based on *Student.grade_level*:

```
    SELECT Student.ID, grade FROM Student, ClassAssignment WHERE
    Student.ID = ClassAssignment.student_ID and Student.grade_level
= 9;
```

Note that the query is a *join* (specifically an *inner join*), which is like *merge()* in R. We don't specifically use the JOIN keyword, but one could do these queries explicitly using JOIN, as we'll see later.

2.4 Stack Overflow metadata example

I've obtained data from Stack Overflow, the popular website for asking coding questions, and placed it into a normalized database. The SQLite version has metadata (i.e., it lacks the actual text of the questions and answers) on all of the questions and answers posted in 2016.

We'll explore SQL functionality using this example database.

Now let's consider the Stack Overflow data. Each question may have multiple answers and each question may have multiple (topic) tags.

If we tried to put this into a single table, the fields could look like this if we have one row per question:

- question ID
- ID of user submitting question
- question title
- tag 1
- tag 2
- ...
- tag n
- answer 1 ID

- ID of user submitting answer 1
- age of user submitting answer 1
- name of user submitting answer 1
- answer 2 ID
- ID of user submitting answer 2
- age of user submitting answer 2
- name of user submitting answer 2
- ...

or like this if we have one row per question-answer pair:

- question ID
- ID of user submitting question
- question title
- tag 1
- tag 2
- ...
- tag n
- answer ID
- ID of user submitting answer
- age of user submitting answer
- name of user submitting answer

As we've discussed neither of those schema is particularly desirable.

Challenge: How would you devise a schema to normalize the data. I.e., what set of tables do you think we should create?

You can view one reasonable schema in the file *normalized_example.png*. The lines between tables indicate the relationship of foreign keys in one table to primary keys in another table. The schema in the actual databases of Stack Overflow data we'll use in this tutorial is similar to but not identical to that.

You can download a copy of the SQLite version of the Stack Overflow 2016 database from http://www.stat.berkeley.edu/share/paciorek/stackoverflow-2016.db.

2.5 Accessing databases in R

The *DBI* package provides a front-end for manipulating databases from a variety of DBMS (SQLite, MySQL, PostgreSQL, among others). Basically, you tell the package what DBMS is being used on the back-end, link to the actual database, and then you can use the standard functions in the package regardless of the back-end. This is a similar style to how one uses *foreach* for parallelization.

With SQLite, R processes make calls against the stand-alone SQLite database (.db) file, so there are no SQLite-specific processes. With a client-server DBMS like PostgreSQL, R processes call out to separate Postgres processes; these are started from the overall Postgres background process

You can access and navigate an SQLite database from R as follows.

```
library(RSQLite)
drv <- dbDriver("SQLite")
dir <- '../data' # relative or absolute path to where the .db file is
dbFilename <- 'stackoverflow-2016.db'
db <- dbConnect(drv, dbname = file.path(dir, dbFilename))
# simple query to get 5 rows from a table
dbGetQuery(db, "select * from questions limit 5")
```

```
##   questionid        creationdate score viewcount
## 1   34552550 2016-01-01 00:00:03     0       108
## 2   34552551 2016-01-01 00:00:07     1       151
## 3   34552552 2016-01-01 00:00:39     2      1942
## 4   34552554 2016-01-01 00:00:50     0       153
## 5   34552555 2016-01-01 00:00:51    -1        54
##
## 1                                                                Scope
## 2      Rails - Unknown Attribute - Unable to add a new field to a form o
## 3 Selenium Firefox webdriver won't load a blank page after changing Fire
## 4                                                      Android Studio s
## 5                            Java: reference to non-finial local variables
##   ownerid
## 1 5684416
## 2 2457617
## 3 5732525
## 4 5735112
## 5 4646288
```

We can easily see the tables and their fields:

```
dbListTables(db)

## [1] "answers"          "questions"        "questionsAugment"
## [4] "questions_tags"   "users"

dbListFields(db, "questions")

## [1] "questionid"   "creationdate" "score"        "viewcount"
## [5] "title"        "ownerid"

dbListFields(db, "answers")

## [1] "answerid"     "questionid"   "creationdate" "score"
## [5] "ownerid"
```

Here's how to make a basic SQL query. One can either make the query and get the results in one go or make the query and separately fetch the results. Here we've selected the first five rows (and all columns, based on the * wildcard) and brought them into R as a data frame.

```
results <- dbGetQuery(db, 'select * from questions limit 5')
class(results)

## [1] "data.frame"

query <- dbSendQuery(db, "select * from questions")
results2 <- fetch(query, 5)
identical(results, results2)

## [1] TRUE

dbClearResult(query)  # clear to prepare for another query
```

To disconnect from the database:

```
dbDisconnect(db)
```

2.6 Basic SQL for choosing rows and columns from a table

SQL is a declarative language that tells the database system what results you want. The system then parses the SQL syntax and determines how to implement the query.

Here are some examples using the Stack Overflow database.

```
## find the largest viewcounts in the questions table
dbGetQuery(db,
'select distinct viewcount from questions order by viewcount desc limit 10'

##    viewcount
## 1     196469
## 2     174790
## 3     134399
## 4     129874
## 5     129624
## 6     127764
## 7     126752
## 8     112000
## 9     109422
## 10    106995

## now get the questions that are viewed the most
dbGetQuery(db, 'select * from questions where viewcount > 100000')

##    questionid        creationdate score viewcount
## 1    34579099 2016-01-03 16:55:16     8    129624
## 2    34814368 2016-01-15 15:24:36   206    134399
## 3    35062852 2016-01-28 13:28:39   730    112000
## 4    35429801 2016-02-16 10:21:09   400    100125
## 5    35588699 2016-02-23 21:37:06    57    126752
## 6    35890257 2016-03-09 11:25:05    51    129874
## 7    35990995 2016-03-14 15:01:17   104    127764
## 8    36668374 2016-04-16 18:57:19    20    196469
## 9    37280274 2016-05-17 15:21:49    23    106995
## 10   37806538 2016-06-14 08:16:21   223    174790
## 11   37937984 2016-06-21 07:23:00   202    109422
##
## 1                                                              Fatal erro
## 2
## 3
## 4
## 5                                                            Response to
## 6                                                    Android- Error:Execu
## 7
## 8                                                            How to solve
## 9                                                                 "Syntax
## 10 Code signing is required for product type 'Application' in SDK 'iOS 1
## 11
##    ownerid
## 1  3656666
## 2  3319176
## 3  2761509
## 4  5881764
## 5  2896963
## 6  1118886
## 7  1629278
## 8  1707976
## 9  4043633
## 10 1554347
## 11 2670370
```

Let's lay out the various verbs in SQL. Here's the form of a standard query (though the ORDER BY is often omitted and sorting is computationally expensive):

`SELECT <column(s)> FROM <table> WHERE <condition(s) on column(s)> ORDER BY <column(s)>`

SQL keywords are often written in ALL CAPITALS though I won't necessarily do that here.

And here is a table of some important keywords:

Table 1. Basic SQL keywords.

Keyword	Usage
SELECT	select columns
FROM	which table to operate on
WHERE	filter (choose) rows satisfying certain conditions
LIKE, IN, <, >, ==, etc.	used as part of conditions
ORDER BY	sort based on columns

For comparisons in a WHERE clause, some common syntax for setting conditions includes LIKE (for patterns), =, >, <, >=, <=, !=.

Some other keywords are: DISTINCT, ON, JOIN, GROUP BY, AS, USING, UNION, INTERSECT, SIMILAR TO.

Question: how would we find the youngest users in the database?

2.7 Simple SQL joins

It turns out that the syntax of using multiple tables we've seen can be viewed formally as a table join and could also be implemented using the JOIN keyword.

The syntax generally looks like this (again the WHERE and ORDER BY are optional):

`SELECT <column(s)> FROM <table1> JOIN <table2> ON <columns to match on> WHERE <condition(s) on column(s)> ORDER BY <column(s)>`

Let's see some joins using the different syntax on the Stack Overflow database. In particular let's select only the questions with the tag "python".

Here's a join using similar syntax to what we saw above, without using the JOIN keyword.

```
result1 <- dbGetQuery(db, "select * from questions, questions_tags
        where questions.questionid = questions_tags.questionid and
        tag = 'python'")
```

And here's how we do it with an explicit JOIN:

```
result2 <- dbGetQuery(db, "select * from questions join questions_tags
        on questions.questionid = questions_tags.questionid
        where tag = 'python'")

head(result1)

##   questionid        creationdate score viewcount
## 1   34553559 2016-01-01 04:34:34     3        96
## 2   34556493 2016-01-01 13:22:06     2        30
## 3   34557898 2016-01-01 16:36:04     3       143
## 4   34560088 2016-01-01 21:10:32     1       126
## 5   34560213 2016-01-01 21:25:26     1       127
## 6   34560740 2016-01-01 22:37:36     0       455
##
## 1                                            Python nested loops only wo
## 2                                        bool operator in for Timestamp
## 3                                                       Pairwise haversi
## 4                                                          Stopwatch (ch
## 5 How to set the type of a pyqtSignal (variable of class X) that takes a
## 6                                                Flask: Peewee model_to_
##   ownerid questionid..7    tag
## 1  845642      34553559 python
## 2 4458602      34556493 python
## 3 2927983      34557898 python
## 4 5736692      34560088 python
## 5 5636400      34560213 python
## 6 3262998      34560740 python
```

```
identical(result1, result2)
```

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

Here's a three-way join (using both types of syntax) with some additional use of aliases to abbreviate table names. What does this query ask for?

```
result1 <- dbGetQuery(db, "select * from
        questions Q, questions_tags T, users U where
        Q.questionid = T.questionid and
        Q.ownerid = U.userid and
        tag = 'python' and
        age < 18")

result2 <- dbGetQuery(db, "select * from
        questions Q
        join questions_tags T on Q.questionid = T.questionid
        join users U on Q.ownerid = U.userid
        where tag = 'python' and
        age < 18")

identical(result1, result2)
```

Challenge: Write a query that would return all the answers to questions with the Python tag.

Challenge: Write a query that would return the users who have answered a question with the Python tag.

2.8 Grouping / stratifying

A common pattern of operation is to stratify the dataset, i.e., collect it into mutually exclusive and exhaustive subsets. One would then generally do some operation on each subset. In SQL this is done with the GROUP BY keyword.

Here's a basic example where we count the occurrences of different tags.

```
dbGetQuery(db, "select tag, count(*) as n from questions_tags
                group by tag order by n desc limit 25")

##               tag      n
## 1      javascript 290966
## 2            java 219155
## 3         android 184272
## 4             php 177969
## 5          python 171745
## 6              c# 163637
## 7            html 126851
## 8          jquery 123707
## 9             ios  95722
## 10            css  86470
## 11      angularjs  76951
## 12            c++  76260
## 13          mysql  75458
## 14         swift  61485
## 15           sql  58346
## 16       node.js  52827
## 17             r  48079
## 18        arrays  46739
## 19          json  45250
## 20 ruby-on-rails  39036
## 21    sql-server  37077
## 22             c  36080
## 23       asp.net  35610
## 24         excel  29924
## 25      angular2  28832
```

In general ‘GROUP BY‘ statements will involve some aggregation operation on the subsets. Options include: COUNT, MIN, MAX, AVG, SUM.

Challenge: Write a query that will count the number of answers for each question, returning the most answered questions.

2.9 Getting unique results (DISTINCT)

A useful SQL keyword is DISTINCT, which allows you to eliminate duplicate rows from any table (or remove duplicate values when one only has a single column or set of values).

```
tagNames <- dbGetQuery(db, "select distinct tag from questions_tags")
head(tagNames)

##          tag
## 1         c#
## 2      razor
## 3      flags
## 4 javascript
## 5       rxjs
## 6    node.js

dbGetQuery(db, "select count(distinct tag) from questions_tags")

##   count(distinct tag)
## 1              41006
```

2.10 Indexes

An index is an ordering of rows based on one or more fields. DBMS use indexes to look up values quickly, either when filtering (if the index is involved in the WHERE condition) or when doing joins (if the index is involved in the JOIN condition). So in general you want your tables to have indexes.

DBMS use indexing to provide sub-linear time lookup. Without indexes, a database needs to scan through every row sequentially, which is called linear time lookup – if there are n rows, the lookup is O(n) in computational cost. With indexes, lookup may be logarithmic – O(log(n)) – (if using tree-based indexes) or constant time – O(1) – (if using hash-based indexes). A binary tree-based search is logarithmic; at each step through the tree you can eliminate half of the possibilities.

Here's how we create an index, with some time comparison for a simple query.

```
system.time(dbGetQuery(db,
  "select * from questions where viewcount > 10000"))   # 10 seconds
system.time(dbGetQuery(db,
  "create index count_index on questions (viewcount)")) # 19 seconds
system.time(dbGetQuery(db,
  "select * from questions where viewcount > 10000"))   # 3 seconds
```

In other contexts, an index can save huge amounts of time. So if you're working with a database and speed is important, check to see if there are indexes.

That being said, using indexes in a lookup is not always advantageous, as discussed in the tutorial.

2.11 Temporary tables and views

You can think of a view as a temporary table that is the result of a query and can be used in subsequent queries. In any given query you can use both views and tables. The advantage is that they provide modularity in our querying. For example, if a given operation (portion of a query) is needed repeatedly, one could abstract that as a view and then make use of that view.

Suppose we always want the age and displayname of owners of questions to be readily available. Once we have the view we can query it like a regular table.

```
## note there is a creationdate in users too, hence disambiguation
dbGetQuery(db, "create view questionsAugment as select
                questionid, questions.creationdate, score, viewcount,
                title, ownerid, age, displayname
                from questions join users
                on questions.ownerid = users.userid")

## Error in result_create(conn@ptr, statement):  table questionsAugment
already exists

## don't be confused by the "data frame with 0 columns and 0 rows"
## message -- it just means that nothing is returned to R;
## the view HAS been created

dbGetQuery(db, "select * from questionsAugment where age < 15 limit 5")
```

One use of a view would be to create a mega table that stores all the information from multiple tables in the (unnormalized) form you might have if you simply had one data frame in R or Python.

2.12 Creating database tables

One can create tables from within the ‘sqlite‘ command line interfaces (discussed in the tutorial), but often one would do this from R or Python. Here’s the syntax from R.

```
## Option 1: pass directly from CSV to database
dbWriteTable(conn = db, name = "student", value = "student.csv",
             row.names = FALSE, header = TRUE)

## Option 2: pass from data in an R data frame
## create data frame 'student' in some fashion
#student <- data.frame(...)
#student <- read.csv(...)
dbWriteTable(conn = db, name = "student", value = student,
             row.names = FALSE, append = FALSE)
```

2.13 More on joins

We've seen a bunch of joins but haven't discussed the full taxonomy of types of joins. There are various possibilities for how to do a join depending on whether there are rows in one table that do not match any rows in another table.

Inner joins: In database terminology an inner join is when the result has a row for each match of a row in one table with the rows in the second table, where the matching is done on the columns you indicate. If a row in one table corresponds to more than one row in another table, you get all of the matching rows in the second table, with the information from the first table duplicated for each of the resulting rows. For example in the Stack Overflow data, an inner join of questions and answers would pair each question with each of the answers to that question. However, questions without any answers or (if this were possible) answers without a corresponding question would not be part of the result.

Outer joins: Outer joins add additional rows from one table that do not match any rows from the other table as follows. 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 converse, while a *full outer join* includes at least one copy of all rows from both tables. So a left outer join of the Stack Overflow questions and answers tables would, in addition to the matched questions and their answers, include a row for each question without any answers, as would a full outer join. In this case there should be no answers that do not correspond to question, so a right outer join should be the same as an inner join.

Cross joins: A cross join gives the Cartesian product of the two tables, namely the pairwise combination of every row from each table, analogous to *expand.grid()* in R. I.e., take a row from the first table and pair it with each row from the second table, then repeat that for all rows from the first table. Since cross joins pair each row in one table with all the rows in another table, the resulting table can be quite large (the product of the number of rows in the two tables). In the Stack Overflow database, a cross join would pair each question with every answer in the database, regardless of whether the answer is an answer to that question.

Simply listing two or more tables separated by commas as we saw earlier is the same as a *cross join*. Alternatively, listing two or more tables separated by commas, followed by conditions that equate rows in one table to rows in another is the same as an *inner join*.

In general, inner joins can be seen as a form of cross join followed by a condition that enforces matching between the rows of the table. More broadly, here are four equivalent joins that all perform the equivalent of an inner join:

```
## explicit inner join:
select * from table1 join table2 on table1.id = table2.id
## non-explicit join without JOIN
select * from table1, table2 where table1.id = table2.id
## cross-join followed by matching
select * from table1 cross join table2 where table1.id = table2.id
## explicit inner join with 'using'
select * from table1 join table2 using(id)
```

Challenge: Create a view with one row for every question-tag pair, including questions without any tags.

Challenge: Write a query that would return the displaynames of all of the users who have *never* posted a question. The NULL keyword will come in handy – it's like 'NA' in R. Hint: NULLs should be produced if you do an outer join.

2.14 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/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, *unit7-bigData.R*, for the bash code.) Note that it's much smaller when compressed (1.7 Gb) or if stored in a binary format. You can download a zipped version of the full CSV from http://www.stat.berkeley.edu/share/paciorek/AirlineDataAll.csv.zip.

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)
dir = '/tmp'
fileName <- file.path(dir, '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 (which is how data.table refers to a database-style *index*) 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,CRSArrTim
##[2,] Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,CRSArrTim
##[3,] Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,CRSArrTim
##      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. From my fairly quick look through the *data.table* documentation I don't see a way to do the subsetting based on ranges of values (e.g., flights 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

The *ff* package stores datasets in columnar format, with one file per column, on disk, so is not limited by memory. It then provides fast access to the dataset from R.

If we need to work with a dataset in R but the dataset won’t fit in memory, 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 <- file.path(dir, '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 = file.path(dir, '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(file.path(dir, '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.ff()*.

Note that a copy of an *ff* object does not appear to actually copy any data, but merely create another name referring to the same data object.

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(file.path(dir, '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 B
# CAK CCR CHS CID CLE CLT CMH CMI COS CPR CRP CRW CVG DAB DAL DAY DCA DEN D
# EUG EVV EWR FAI FAR FAT FLG FLL FOE FSD GCN GEG GJT GRR GSO GSP GTF HNL H
# ICT ILG ILM IND ISP JAN JAX JFK KOA LBB LEX LGA LGB LIH LIT LMT LNK MAF M
# MFR MHT MIA MKE MLB MLI MOB MRY MSN MSP MSY OGG OKC OMA ONT ORD ORF PBI P
# ...

# let's do some basic tabulation
DestTable <- sort(table.ff(dat$Dest), decreasing = TRUE)
# table is a generic, so shouldn't need explicit table.ff,
# unless dat$Dest is not see as an ff object

# 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

# 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

# why do I need to call min.ff and max.ff rather than min/max?
# tmp <- clone(dat$DepDelay) # make an explicit copy
```

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

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)
dir = '/tmp'
fileName <- file.path(dir, 'AirlineDataAll.csv')
# 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

You should already be familiar with using *dplyr.* One very nice feature is that with *dplyr* one can work with data stored in the *data.table* format, in external databases, and in Spark. There is also an extension to dplyr that allows for dplyr operations to be done in parallel.

```r
library(dplyr)

## with database
dir <- '../data' # relative or absolute path to where the .db file is
dbFilename <- 'stackoverflow-2016.db'

db <- src_sqlite(file.path(dir, dbFilename))
questions <- tbl(db, "questions")
questions

## with data.table
dir <- '/tmp'
fileName <- file.path(dir, '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 'flights'

flights %>% group_by(UniqueCarrier) %>%
summarize(mnDelay = 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)

dir = '/tmp'
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 = file.path(dir, 'signif.csv'),
     sep = ',', col.names = FALSE,  row.names = FALSE,
     append = TRUE, quote = FALSE)
}



fileName <- file.path(dir, '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 = file.path(dir, 'signif'))
```

```
system.time(ffload(file.path(dir, '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)   # 0.385 seconds
system.time(sMat %*% vec) # 0.015 seconds
```

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, also published as Kleiner et al. in Journal of the Royal Statistical Society (2014) 76:795.

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

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. In the last 20 years, focus has shifted to technologies for processing large datasets that are distributed across multiple machines, but can be manipulated as if they are one dataset.

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 implemented in *dplyr* is the split-apply-combine strategy (http://www.jstatsoft.org/v40/i

Note that the idea of concepts of map and reduce are core concepts in functional programming (and that we said R was a functional programming language). The various apply commands are a version of a map operation in base R.

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. Given the popularity of Spark, I'm not sure how much usage these approaches currently see. 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 (and generally, Spark will be reading the data initially from 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

6.3.1 Overview

We'll focus on Spark rather than Hadoop for the speed reasons described above and because I think Spark provides a very nice environment/interface in which to work. Plus it comes out of the (former) AmpLab here at Berkeley. We'll start with the Python interface to Spark and then see a bit of the *sparklyr* R package for interfacing with Spark.

More details on Spark are in the Spark programming guide.

Some key aspects of Spark:

- Spark can read/write from various locations, but a standard location is the HDFS, with read/write done in parallel across the cores of the Spark cluster.

- The basic data structure in Spark is a *Resilient Distributed Dataset (RDD)*, which is basically a distributed dataset of individual units, often individual rows loaded from text files.

- RDDs are stored in chunks called *partitions*, stored on the different nodes of the cluster (either in memory or if necessary on disk).

- Spark has a core set of methods that can be applied to RDDs to do operations such as filtering/subsetting, transformation/mapping, reduction, and others.
- The operations are done in parallel on the different partitions of the data
- Some operations such as reduction generally involve a *shuffle*, moving data between nodes of the cluster. This is costly.
- Recent versions of Spark have a distributed *DataFrame* data structure and the ability to run SQL queries on the data.

Question: what do you think are the tradeoffs involved in determining the number of partitions to use?

Note that some headaches with Spark include:

- whether and how to set the amount of memory available for Spark workers (executor memory) and the Spark master process (driver memory)
- hard-to-diagnose failures (including out-of-memory issues)

6.3.2 Getting started

We'll use Spark on Savio. You can also use Spark on NSF's XSEDE Bridges supercomputer (among other XSEDE resources), and via commercial cloud computing providers, as well as on your laptop (but obviously only to experiment with small datasets). The demo works with a dataset of Wikipedia traffic, ~110 GB of zipped data (~500 GB unzipped) from October-December 2008, though for in-class presentation we'll work with a much smaller set of 1 day of data.

The Wikipedia traffic are available through Amazon Web Services storage. The steps to get it are:

1. Start an AWS EC2 virtual machine that mounts the data onto the VM
2. Install Globus on the VM
3. Transfer the data to Savio via Globus

Details on how I did this are in *get_wikipedia_data.sh*. The resulting data are available to you in */global/scratch/paciorek/wikistats_full/raw* on Savio.

6.3.3 Storing data for use in Spark

In many Spark contexts, the data would be stored in a distributed fashion across the hard drives attached to different nodes of a cluster (i.e., in the HDFS).

On Savio, Spark is set up to just use the scratch file system, so one would NOT run the code here, but I'm including it to give a sense for what it's like to work with HDFS. First we would need to get the data from the standard filesystem to 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.

```
## DO NOT RUN THIS CODE ON SAVIO ##
## data for Spark on Savio is stored in scratch ##

hadoop fs -ls /
hadoop fs -ls /user
hadoop fs -mkdir /user/paciorek/data
hadoop fs -mkdir /user/paciorek/data/wikistats
hadoop fs -mkdir /user/paciorek/data/wikistats/raw
hadoop fs -mkdir /user/paciorek/data/wikistats/dated

hadoop fs -copyFromLocal /global/scratch/paciorek/wikistats/raw/* \
       /user/paciorek/data/wikistats/raw

# check files on the HDFS, e.g.:
hadoop fs -ls /user/paciorek/data/wikistats/raw

## now do some processing with Spark, e.g., preprocess.{sh,py}

# after processing can retrieve data from HDFS as needed
hadoop fs -copyToLocal /user/paciorek/data/wikistats/dated .
```

6.3.4 Using Spark on Savio

Here are the steps to use Spark on Savio. We'll demo using an interactive job (the *srun* line here) but one could include the last three commands in the SLURM job script.

```
tmux new -s spark  ## to get back in if disconnected: tmux a -t spark

## having some trouble with ic_stat243 and 4 nodes; check again
srun -A ic_stat243 -p savio2 --nodes=4 -t 1:00:00 --pty bash
module load java spark/2.1.0 python/3.5
source /global/home/groups/allhands/bin/spark_helper.sh
spark-start
## note the environment variables created
env | grep SPARK


spark-submit --master $SPARK_URL  $SPARK_DIR/examples/src/main/python/pi.py
```

First we'll load Python; then we can use Spark via the Python interface interactively. We'll see how to submit batch jobs later.

```
# PySpark using Python 3.5 (Spark 2.1.0 doesn't support Python 3.6)
# HASHSEED business has to do ensuring consistency across Python sessions
pyspark --master $SPARK_URL --conf "spark.executorEnv.PYTHONHASHSEED=321"
--executor-memory 60G
```

6.3.5 Preprocessing the Wikipedia traffic data

At this point, one complication is that the date-time information on the Wikipedia traffic is embedded in the file names. We'd like that information to be fields in the data files. This is done by running the code in *preprocess_wikipedia.py* in the Python interface to Spark (pyspark). Note that trying to use multiple nodes and to repartition in various ways caused various errors I was unable to diagnose, but the code as is should work albeit somewhat slowly. The resulting data are available to you in */global/scratch/paciorek/wikistats_full/dated*. These are the data you will use for PS6.

In principle one could run *preprocess_wikipedia.py* as a batch submission, but I was having problems getting that to run successfully.

6.3.6 Spark in action: processing the Wikipedia traffic data

Now we'll do some basic manipulations with the Wikipedia dataset, with the goal of analyzing traffic to Barack Obama's sites during the time around his election as president in 2008. Here are the steps we'll follow:

- Count the number of lines/observations in our dataset.
- Filter to get only the Barack Obama sites.
- Map step that creates key-value pairs from each record/observation/row.
- Reduce step that counts the number of views by hour and language, so hour-day-lang will serve as the key.
- Map step to prepare the data so it can be output in a nice format.

Note that Spark uses *lazy evaluation*. Actual computation only happens when one asks for a result to be returned or output written to disk.

First we'll see how we read in the data and filter to the observations (lines / rows) of interest.

```
dir = ’/global/scratch/paciorek/wikistats’
### read data and do some checks ###
## ’sc’ is the SparkContext management object, created via PySpark
## if you simply start Python, without invoking PySpark,
## you would need to create the SparkContext object yourself
lines = sc.textFile(dir + ’/’ + ’dated’)
lines.getNumPartitions()  # 16800 (480 input files) for full dataset
# note delayed evaluation
lines.count()  # 9467817626 for full dataset
# watch the UI and watch wwall as computation progresses
testLines = lines.take(10)
testLines[0]
testLines[9]
### filter to sites of interest ###
import re
from operator import import add
def find(line, regex = "Barack_Obama", language = None):
    vals = line.split(’ ’)
    if len(vals) < 6:
        return(False)
    tmp = re.search(regex, vals[3])
    if tmp is None or (language != None and vals[2] != language):
        return(False)
    else:
        return(True)
lines.filter(find).take(100) # pretty quick

# not clear if should repartition; will likely have small partitions if not
# obama = lines.filter(find).repartition(480) # ~ 18 minutes for full dataset (but remember lazy evaluation)
obama = lines.filter(find)  # use this for demo in section
obama.count()  # 433k observations for full dataset
```

Now let’s use the mapReduce paradigm to get the aggregate statistics we want.

```
### map-reduce step to sum hits across date-time-language triplets ###

def stratify(line):
    # create key-value pairs where:
    #   key = date-time-language
    #   value = number of website hits
    vals = line.split(’ ’)
    return(vals[0] + ’-’ + vals[1] + ’-’ + vals[2], int(vals[4]))
# sum number of hits for each date-time-language value
counts = obama.map(stratify).reduceByKey(add)  # 5 minutes
# 128889 for full dataset
### map step to prepare output ###
def transform(vals):
    # split key info back into separate fields
    key = vals[0].split(’-’)
    return(",".join((key[0], key[1], key[2], str(vals[1]))))
### output to file ###
# have one partition because one file per partition is written out
outputDir = dir + ’/’ + ’obama-counts’
counts.map(transform).repartition(1).saveAsTextFile(outputDir) # 5 sec.
```

6.3.7 Spark monitoring

There are various interfaces to monitor Spark and the HDFS.

- http://<master_url>:8080 – general information about the Spark cluster
- http://<master_url>:4040 – information about the Spark tasks being executed
- http://<master_url>:50070 – information about the HDFS

When one runs *spark-start* on Savio, it mentions some log files. If you look in the log file for the master, you should see a line that says "Bound MasterWebUI to 0.0.0.0 and started at http://10.0.5.93:8080" that indicates what the <master_url> is (here it is 10.0.5.93). We need to connect to that URL to view the web UI.

On Savio, to view the interfaces in a web browser, you need to start a remote desktop (VNC) session, following these instructions: https://research-it.berkeley.edu/services/high-performance-computing/using-brc-visualization-node-realvnc; I suggest using the VNC add-on to the Chrome browser. Once you have a window onto Savio in your VNC session, start a browser from the terminal windows by entering: `/global/scratch/kmuriki/otterbrowser <master_url>:8080`, e.g. 10.0.5.93:8080.

6.3.8 Spark operations

Let's consider some of the core methods we used.

- *filter()*: create a subset
- *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 (order across records doesn't matter) and commutative (order of arguments doesn't matter) and take 2 arguments and return 1, all so that they can be done in parallel in a straightforward way.
- *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 divided into the specified number of partitions

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, obama,* and *counts* are all RDDs. However the result of *collect()* is just a standard Python object.

6.3.9 Nonstandard reduction

Finding the median of a set of values is 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.

As an example we could find the median page size by language (this is not a particularly interesting/useful computation in this dataset, but I wanted to illustrate how this would work).

```python
import numpy as np
def findShortLines(line):
    vals = line.split(’ ’)
    if len(vals) < 6:
        return(False)
    else:
        return(True)
def computeKeyValue(line):
    vals = line.split(’ ’)
    # key is language, val is page size
    return(vals[2], int(vals[5]))
def medianFun(input):
    # input[1] is an iterable object containing the page sizes for one key
    # this list comprehension syntax creates a list from the iterable object
    med = np.median([val for val in input[1]])
    # input[0] is the key
    # return a tuple of the key and the median for that key
    return((input[0], med))
output = lines.filter(findShortLines).map(computeKeyValue).groupByKey()
medianResults = output.map(medianFun).collect()
```

Note that because we need to aggregate all the data by key before doing the reduction on the full data in each key (which is actually just a ’map’ operation in this case once the data are already grouped by key), this is much slower than a reduce operation like max or mean.

6.3.10 Spark DataFrames and SQL queries

In recent versions of Spark, one can work with more structured data objects than RDDs. Spark now provides *DataFrames*, which are collections of row and behave like distributed versions of R or Pandas dataframes. DataFrames seem to be taking the place of RDDs, at least for general, high-level use. They can also be queried using SQL syntax.

Here’s some example code for using DataFrames.

```
### read the data in and process to create an RDD of Rows ###
dir = ’/global/scratch/paciorek/wikistats’
lines = sc.textFile(dir + ’/’ + ’dated’)
### create DataFrame and do some operations on it ###
def remove_partial_lines(line):
    vals = line.split(’ ’)
    if len(vals) < 6:
        return(False)
    else:
        return(True)
def create_df_row(line):
    p = line.split(’ ’)
    return(int(p[0]), int(p[1]), p[2], p[3], int(p[4]), int(p[5]))
tmp = lines.filter(remove_partial_lines).map(create_df_row)
## ’sqlContext’ is the Spark sqlContext management object, created
via PySpark
## if you simply start Python without invoking PySpark,
## you would need to create the sqlContext object yourself
df = sqlContext.createDataFrame(tmp, schema = ["date", "hour", "lang",
"site", "hits", "size"])
df.printSchema()
## note similarity to dplyr and R/Pandas dataframes
df.select(’site’).show()
df.filter(df[’lang’] == ’en’).show()
df.groupBy(’lang’).count().show()
```

And here’s how we use SQL with a DataFrame:

```
### use SQL with a DataFrame ###
df.registerTempTable("wikiHits")  # name of ’SQL’ table is ’wikiHits’
subset = sqlContext.sql("SELECT * FROM wikiHits WHERE lang = ’en’ AND
site LIKE ’%Barack_Obama%’ ")
subset.take(5)
# [Row(date=20081022, hits=17, hour=230000, lang=u’en’, site=u’Media:En-Bar
size=145491), Row(date=20081026, hits=41, hour=220000, lang=u’en’,
site=u’Public_image_of_Barack_Obama’, size=1256906), Row(date=20081112,
hits=8, hour=30000, lang=u’en’, site=u’Electoral_history_of_Barack_Obama’,
size=141176), Row(date=20081104, hits=13890, hour=110000, lang=u’en’,
site=u’Barack_Obama’, size=2291741206), Row(date=20081104, hits=6,
hour=110000, lang=u’en’, site=u’Barack_Obama%2C_Sr.’, size=181699)]
langSummary = sqlContext.sql("SELECT lang, count(*) as n FROM wikiHits
GROUP BY lang ORDER BY n desc limit 20") # 38 minutes
results = langSummary.collect()
# [Row(lang=u’en’, n=3417350075), Row(lang=u’de’, n=829077196), Row(lang=u’
n=734184910), Row(lang=u’fr’, n=466133260), Row(lang=u’es’, n=425416044),
Row(lang=u’pl’, n=357776377), Row(lang=u’commons.m’, n=304076760),
Row(lang=u’it’, n=300714967), Row(lang=u’ru’, n=256713029), Row(lang=u’pt’,
n=212763619), Row(lang=u’nl’, n=194924152), Row(lang=u’sv’, n=105719504),
Row(lang=u’zh’, n=98061095), Row(lang=u’en.d’, n=81624098), Row(lang=u’fi’,
n=80693318), Row(lang=u’tr’, n=73408542), Row(lang=u’cs’, n=64173281),
Row(lang=u’no’, n=48592766), Row(lang=u’he’, n=46986735), Row(lang=u’ar’,
n=46968973)]
```

6.3.11 Analysis results

The file *obama_plot.R* does some manipulations to plot the hits as a function of time, shown here:

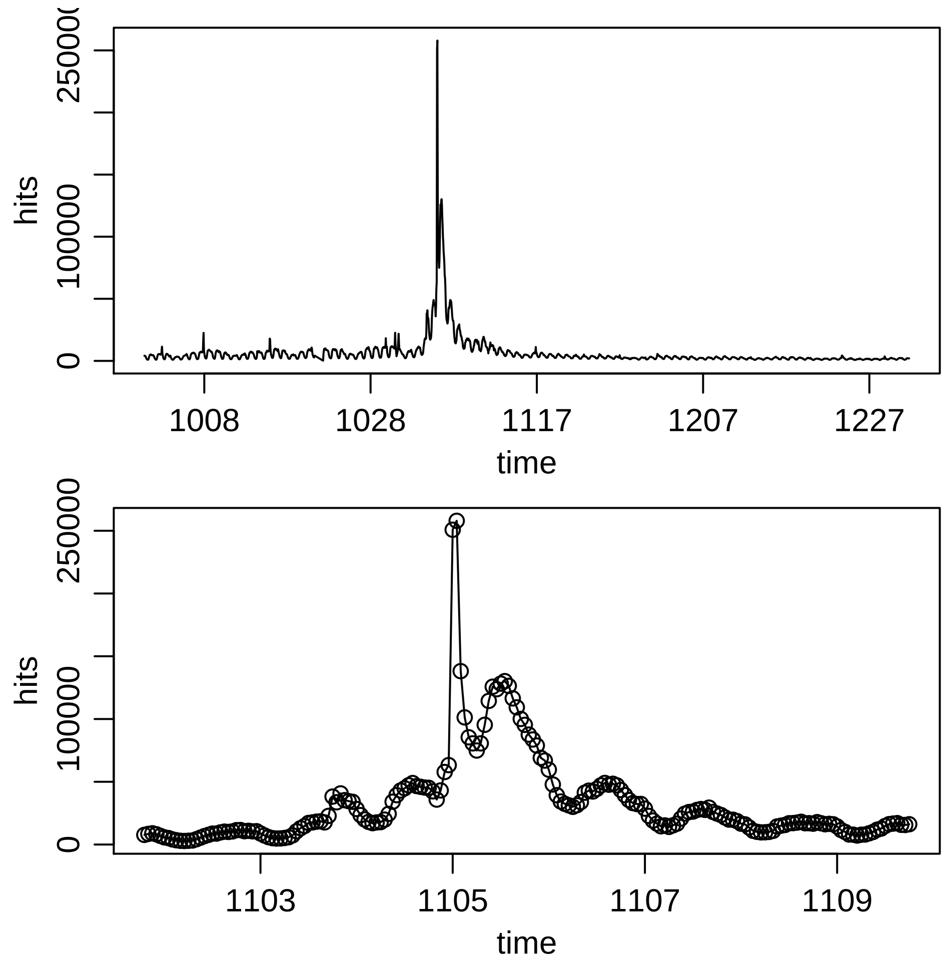

Figure 1. Obama Wikipedia traffic results.

So there you have it – from big data (500 GB unzipped) to knowledge (a 17 KB file of plots).

6.3.12 Other 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.

```
spark-submit --master $SPARK_URL $SPARK_DIR/examples/src/main/python/pi.py
```

The file *example_spark_job.sh* is an example SLURM job submission script that runs the PySpark code in *test_batch.py*. If you want to run a Spark job as a batch submission to the scheduler you can follow this example.

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 http://apache-spark-user-list.1001560.n3.nabble.com/Scala-vs-Python-performance-differences-td4247.html on that.

6.3.13 R interfaces to Spark

Both *SparkR* (from the Spark folks) and *sparklyr* (from the RStudio folks) allow you to interact with Spark-based data from R. There are some limitations to what you can do (both in what is possible and in what will execute with reasonable speed), so for heavy use of Spark you may want to use Python or even the Scala or Java interfaces. We'll focus on *sparklyr*.

With *sparklyr*, you can:

- use *dplyr* functionality
- use distributed apply computations via *spark_apply()*.

There are some limitations though:

- the *dplyr* functionality translates operations to SQL so there are limited operations one can do, particularly in terms of computations on a given row of data.
- *spark_apply()* appears to run very slowly, presumably because data is being serialized back and forth between R and Java data structures.

6.3.14 sparklyr example

Here's some example code that works on Savio. One important note is that if you don't adjust the memory, you'll get obscure Java errors that occur because Spark runs out of memory, and this is only clear if you look in the right log files in the directory $SPARK_LOG_DIR.

```
## see unit8-bigData.sh for starting Spark
## also invoke:
## module load r r-packages

## local installation on your own computer
if(!require(sparklyr)) {
    install.packages("sparklyr")
    # spark_install() ## if spark not already installed
}
```

```r
### connect to Spark ###

## need to increase memory otherwise get hard-to-interpret Java
## errors due to running out of memory; total memory on the node is 64 GB
conf <- spark_config()
conf$spark.driver.memory <- "8G"
conf$spark.executor.memory <- "50G"

# sc <- spark_connect(master = "local")  # if doing on laptop
sc <- spark_connect(master = Sys.getenv("SPARK_URL"),
                    config = conf)  # non-local

### read data in ###

cols <- c(date = 'numeric', hour = 'numeric', lang = 'character',
          page = 'character', hits = 'numeric', size = 'numeric')


## takes a while even with only 1.4 GB (zipped) input data (100 sec.)
wiki <- spark_read_csv(sc, "wikistats",
                       "/global/scratch/paciorek/wikistats/dated",
                       header = FALSE, delimiter = ' ',
                       columns = cols, infer_schema = FALSE)

head(wiki)
class(wiki)
dim(wiki)   # not all operations work on a spark dataframe

### some dplyr operations on the Spark dataset ###

library(dplyr)

wiki_en <- wiki %>% filter(lang == "en")
head(wiki_en)

table <- wiki %>% group_by(lang) %>% summarize(count = n()) %>%
    arrange(desc(count))
## note the lazy evaluation: need to look at table to get computation to ru
table
dim(table)
class(table)

### distributed apply ###

## need to use spark_apply to carry out arbitrary R code
## the function transforms a dataframe partition into a dataframe
## see help(spark_apply)
##
## however this is _very_ slow, probably because it involves
## serializing objects between java and R
wiki_plus <- spark_apply(wiki, function(data) {
    data$obama = stringr::str_detect(data$page, "Barack_Obama")
    data
}, columns = c(colnames(wiki), 'obama'))

obama <- collect(wiki_plus %>% filter(obama))

### SQL queries ###

library(DBI)
## reference the Spark table (see spark_read_csv arguments)
## not the R tbl_spark interface object
wiki_en2 <- dbGetQuery(sc,
            "SELECT * FROM wikistats WHERE lang = 'en' LIMIT 10")
wiki_en2
```