Unit 8: Databases and Big Data

October 22, 2019

References:

- Tutorial on parallel processing using Python's Dask and R's future: https://github.com/berkeley-scf/tutorial-dask-future
- Spark Programming Guide
- SCF tutorial on "Working with large datasets in SQL, R, and Python"
- Murrell: Introduction to Data Technologies
- Adler: R in a Nutshell

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, though I guess it's not quite the buzzword/buzzphrase that it was a few years ago.

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.

Finally, for many applications, storing large datasets in a standard database will work well. We'll see databases later in this Unit.

2 Hadoop, MapReduce, Spark, and Dask

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.

Two commonly-used tools for doing this are Spark and Python's Dask package. We'll cover Dask.

2.1 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

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.

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 *lapply/sapply/apply* commands are base R's version of a map operation.

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. Given the popularity of Spark, I'm not sure how much usage these approaches currently see. 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.) While Spark is more user-friendly than Hadoop, there are also some things that can make it hard to use. Setting up a Spark cluster also involves a bit of work, Spark can be hard to configure for optimal performance, and Spark calculations have a tendency to fail (often involving memory issues) in ways that are hard for users to debug.

2.2 Spark

Note for 2019: in past years we covered the use of Spark for processing big datasets. This year we'll cover similar functionality in Python's Dask package. I've kept this section in case anyone is interested in learning more about Spark, but we won't cover it in class this year.

2.2.1 Overview

We'll focus on Spark rather than Hadoop for the speed reasons described above and because I think Spark provides a nicer 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)

2.2.2 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.

2.2.3 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 .
```

2.2.4 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"
```

2.2.5 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.

2.2.6 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 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 datase
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.
```

2.2.7 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.

2.2.8 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.

2.2.9 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).

```
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.

2.2.10 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 PySpa
## 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

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

subset.take(5)
# [Row(date=20081022, hits=17, hour=230000, lang=u'en', site=u'Media:En-Bara

langSummary = sqlContext.sql("SELECT lang, count(*) as n FROM wikiHits GROU
results = langSummary.collect()
# [Row(lang=u'en', n=3417350075), Row(lang=u'de', n=829077196), Row(lang=u'
```

2.2.11 Analysis results

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

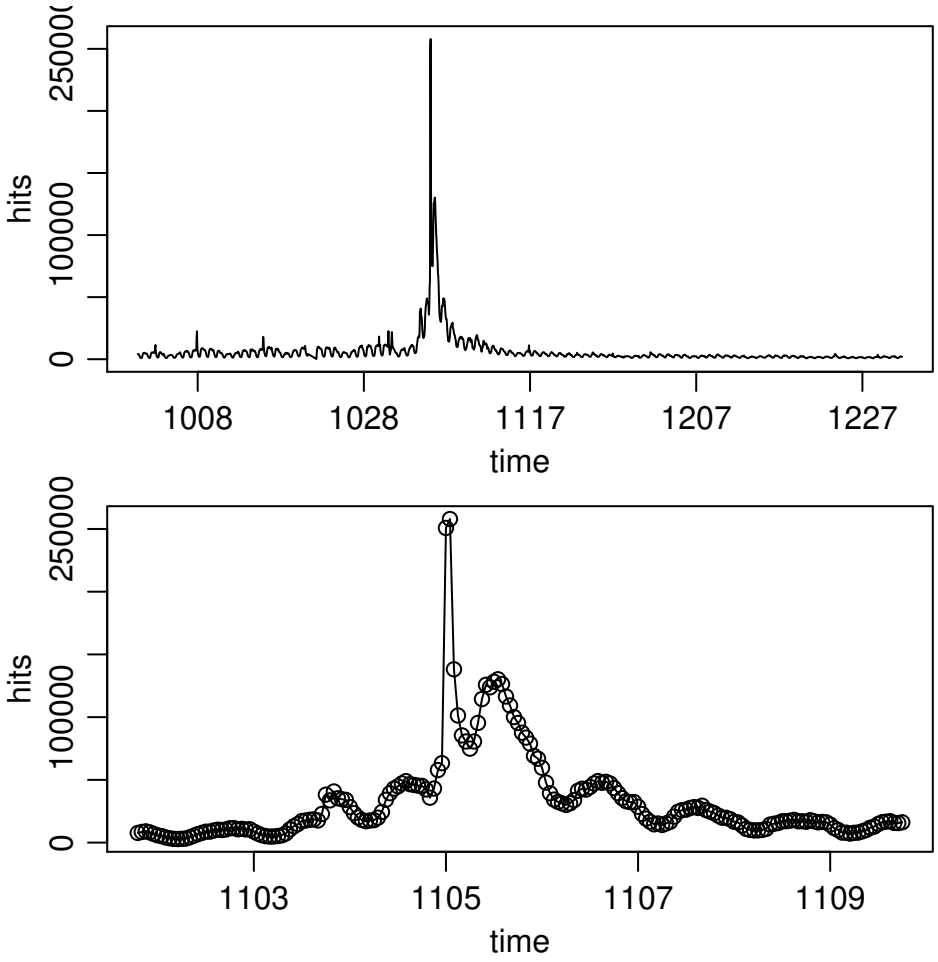

Figure 1. Obama Wikipedia traffic results.

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

2.2.12 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 that example, submitting the job using *sbatch*: `sbatch name_of_job_script.sh`.

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.

2.2.13 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.

2.2.14 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
}
```

```
### 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

#####################################################
# 3: Databases
#####################################################

### 3.5 Accessing databases in R
```

2.3 Using Dask for big data processing

Unit 7 on parallelization gives an overview of using Dask in similar fashion to how we used R's *future* package for flexible parallelization on different kinds of computational resources (in particular, parallelizing across multiple cores on one machine versus parallelizing across multiple cores across multiple machines/ndoes).

Here we'll see the use of Dask to work with distributed datasets. Dask can process datasets (potentially very large ones) by parallelizing operations across subsets of the data using multiple cores on one or more machines.

Like Spark, Dask automatically reads data from files in parallel and operates on chunks (shards) of the full dataset in parallel. There are two big advantages of this:

- You can do calculations (including reading from disk) in parallel because each worker will work on a piece of the data.
- When the data is split across machines, you can use the memory of multiple machines to handle much larger datasets than would be possible in memory on one machine. That said, Dask processes the data in chunks, so one often doesn't need a lot of memory, even just on one machine.

While reading from disk in parallel is a good goal, if all the data are on one hard drive, there are limitations on the speed of reading the data from disk because of having multiple processes all trying to access the disk at once. Supercomputing systems will generally have parallel file systems that truly support parallel reading (and writing, i.e., *parallel I/O*). Hadoop/Spark deal with this by distributing across multiple disks, generally one disk per machine/node.

Because computations are done in external compiled code (e.g., via *numpy*) it's effective to use the threaded scheduler when operating on one node to avoid having to copy and move the data.

2.3.1 Dask dataframes (pandas)

Dask dataframes are Pandas-like dataframes where each dataframe is split into groups of rows, stored as smaller Pandas dataframes.

One can do a lot of the kinds of computations that you would do on a Pandas dataframe on a Dask dataframe, but many operations are not possible. See here. here.

By default dataframes are handled by the threads scheduler.

Here's an example of reading from a dataset of flight delays (about 11 GB data). You can get the data here.

```
import dask
dask.config.set(scheduler='threads', num_workers = 4)
import dask.dataframe as ddf
path = '/scratch/users/paciorek/243/AirlineData/csvs/'
air = ddf.read_csv(path + '*.csv.bz2',
      compression = 'bz2',
      encoding = 'latin1', # (unexpected) latin1 value(s) in TailNum field
      dtype = {'Distance': 'float64', 'CRSElapsedTime': 'float64',
      'TailNum': 'object', 'CancellationCode': 'object'})
# specify dtypes so Pandas doesn't complain about column type heterogeneity
```

Dask will reads the data in parallel from the various .csv.bz2 files (unzipping on the fly), but note the caveat in the previous section about the possibilities for truly parallel I/O.

However, recall delayed evaluation in Dask – the reading is delayed until *compute()* is called. For that matter, the various other calculations (max, groupby, mean) are only done after *compute()* is called.

```
air.DepDelay.max().compute()   # this takes a while
sub = air[(air.UniqueCarrier == 'UA') & (air.Origin == 'SFO')]
byDest = sub.groupby('Dest').DepDelay.mean()
byDest.compute()               # this takes a while too
```

You should see this:

```
Dest
ACV 26.200000
BFL 1.000000
BOI 12.855069
BOS 9.316795
CLE 4.000000
...
```

2.3.2 Dask bags

Bags are like lists but there is no particular ordering, so it doesn't make sense to ask for the i'th element.

You can think of operations on Dask bags as being like parallel map operations on lists in Python or R.

By default bags are handled via the multiprocessing scheduler.

Let's see some basic operations on a large dataset of Wikipedia log files. You can get a subset of the Wikipedia data here.

Here we again read the data in (which Dask will do in parallel):

```
import dask.multiprocessing
dask.config.set(scheduler='processes', num_workers = 4)  # multiprocessing
import dask.bag as db
path = '/scratch/users/paciorek/wikistats/dated_2017/'
wiki = db.read_text(path + 'part-0000*gz')
```

Here we'll just count the number of records.

```
import time
t0 = time.time()
wiki.count().compute()
time.time() - t0   # 136 sec.
```

And here is a more realistic example of filtering (subsetting).

```
import re
def find(line, regex = "Obama", language = "en"):
    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)


wiki.filter(find).count().compute()
obama = wiki.filter(find).compute()
obama[0:5]
```

Note that it is quite inefficient to do the *find()* (and implicitly reading the data in) and then compute on top of that intermediate result in two separate calls to *compute()*. Rather, we should

set up the code so that all the operations are set up before a single call to *compute()*. More on this in Section 6 of the Dask content in the Dask/future tutorial.

2.3.3 Dask arrays (numpy)

Dask arrays are numpy-like arrays where each array is split up by both rows and columns into smaller numpy arrays.

One can do a lot of the kinds of computations that you would do on a numpy array on a Dask array, but many operations are not possible. See here.

By default arrays are handled via the threads scheduler.

Non-distributed arrays Let's first see operations on a single node, using a single 13 GB 2-d array. Again, Dask uses lazy evaluation, so creation of the array doesn't happen until an operation requiring output is done.

```
import dask
dask.config.set(scheduler = 'threads', num_workers = 4)
import dask.array as da
x = da.random.normal(0, 1, size=(40000,40000), chunks=(10000, 10000))
# square 10k x 10k chunks
mycalc = da.mean(x, axis = 1)  # by row
import time
t0 = time.time()
rs = mycalc.compute()
time.time() - t0  # 41 sec.
```

For a row-based operation, we would presumably only want to chunk things up by row, but this doesn't seem to actually make a difference, presumably because the mean calculation can be done in pieces and only a small number of summary statistics moved between workers.

```
import dask
dask.config.set(scheduler='threads', num_workers = 4)
import dask.array as da
# x = da.from_array(x, chunks=(2500, 40000))  # adjust chunk size of existi
x = da.random.normal(0, 1, size=(40000,40000), chunks=(2500, 40000))
mycalc = da.mean(x, axis = 1)  # row means
import time
```

```
t0 = time.time()
rs = mycalc.compute()
time.time() - t0   # 42 sec.
```

Of course, given the lazy evaluation, this timing comparison is not just timing the actual row mean calculations.

But this doesn't really clarify the story...

```
import dask
dask.config.set(scheduler='threads', num_workers = 4)
import dask.array as da
import numpy as np
import time
t0 = time.time()
x = np.random.normal(0, 1, size=(40000,40000))
time.time() - t0   # 110 sec.
# for some reason the from_array and da.mean calculations are not done lazil
t0 = time.time()
dx = da.from_array(x, chunks=(2500, 40000))
time.time() - t0   # 27 sec.
t0 = time.time()
mycalc = da.mean(x, axis = 1)  # what is this doing given .compute() also ta
time.time() - t0   # 28 sec.
t0 = time.time()
rs = mycalc.compute()
time.time() - t0   # 21 sec.
```

Dask will avoid storing all the chunks in memory. (It appears to just generate them on the fly.) Here we have an 80 GB array but we never use more than a few GB of memory (based on 'top' or 'free -h').

```
import dask
dask.config.set(scheduler='threads', num_workers = 4)
import dask.array as da
x = da.random.normal(0, 1, size=(100000,100000), chunks=(10000, 10000))
mycalc = da.mean(x, axis = 1)  # row means
import time
```

```
t0 = time.time()
rs = mycalc.compute()
time.time() - t0   # 205 sec.
rs[0:5]
```

Distributed arrays This should be straightforward based on using Dask distributed. However, one would want to be careful about creating arrays by distributing the data from a single Python process as that would involve copying between machines.

3 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.

3.1 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.

3.2 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.

3.3 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. We'll list some in class:

1. ???

2. ???

3. ???

4. ???

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.)

3.3.1 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