Chapter 1

Data Analytics for the Power Grid

2.1 Introduction & Motivation

This is a really nitty-gritty example. What I want to accomplish today is to replicate something of the experience you may have approaching solving Big Data problems on your own for the first time. We are going to take a look at a data set I had not seen before six weeks ago and I will walk you through the approach, challenges, and successes I had. We will carry out some local work and some cloud processing.

I plan to specifically accomplish a few things with this tutorial. First, I want to show what a dirty data set looks like, and how you deal with it. Second, I want to perform some local analysis to detect power excursions and follow up on events. Third, I want to run it with Hadoop to identify an aggregate of events. Fourth, I want to be able to integrate my data set with other data sets to clarify events and discover correlations.

So this is a really hands-on tutorial, and hopefully represents something like what many of you would experience approaching solving Big Data problems for the first time.

What is PMU Data?

PMUs in the Power Grid

A phasor measurement unit measures the electrical waves on a power grid. PMUs are synchronized to a common time source and yield data about the voltage and current phasors, as well as frequency and the rates of change of these quantities. They can be deployed at the grid level, which is where most research has focused, or they can be deployed locally. PMUs can measure up to 2,880 samples per second. They are accurate to the microsecond and are calibrated to a GPS reference site.

The data which we will use were provided by Tim Yardley of the Trustworthy Cyber Infrastructure for the Power Grid center located here at the University of Illinois. These data are from a type of PMU called a frequency disturbance recorder, or FDR. The FDR is a single-phase, 120-V PMU, relatively easy to install. It rapidly samples an outlet's voltage signal 1,440 times a second, then calculates and outputs the voltage angle, frequency, and voltage magnitude at 200 ms intervals. They were deployed at several local homes in 2012 and have been recording more-or-less continuously since then. So these reflect an intra-area, rather than a transmission- or grid-level view, of power fluctuations and responses to demand. They are capable of providing both a real-time and archival view of what the power grid is doing—oscillations, generator trips, faults, and so forth. We're going to run through some hands-on work with the data now.

2.2 Local Analytics

We will start off on our local machines on the virtual machine. Let's start off by examining the PMU data and carrying out some local calculations. We are going to use Python for this analysis; if you don't know it, don't worry—you should be able to follow the development in any case.

Data Cleaning

So let's start by opening up one of the data files. The raw data files are not actually included here, since they are in a binary format that's difficult to work directly with unless you have specialized software. What we have are the raw data converted to the text-based CSV format—comma-separated values—which takes up more space on the disk but is accessible to us. I'll go ahead and upon up this one in `vi` first so you can see the raw format. What we have are time stamps on the left and column fields on the top. Now I'll go ahead and open this same file up in a spreadsheet program and we can see what all of the data together look like.

If I plot the data (which I won't do now since it takes too long, which also highlights one of the first problems in defining Big Data: data sets that are too large to effectively work with using regular tools), then you can see the voltage of the different meters here. Now some of the values are at about the halfway point, which may not by itself clue you off that something is wrong—but what other field would we expect to have values near the halfway point? (frequency) If you examine the open spreadsheet, sure enough you find values outside of the expected fields, and gaps within the fields.

So the first thing I had to figure out in approaching this data set was what to do with the data in the state they're in now. The data are dirty, so I need to clean them. What I mean by that is that although there is a convention for data representation here, I can't rely on that convention being enforced.

Thus data cleaning leads us into our first detour: data validation. This is actually an open question in PMU data analysis right now. You could probably just linearly interpolate values

raw data, filters, cleaning, validation Now we have a problem loading and reading the data because, if you look at the file, the data are dirty. We have to clean the data separately.

From a PNNL report: " 3. Data Validation Schemes a. Most utilities do not employ an on-line data validation scheme beyond what is provided in their PMUs, PDCs, data storage, and visualization applications. b. Data validation is generally limited to one-time or periodic comparison with SCADA readings and sanity checks. c. Most utilities anticipate improvement in data validation through vendors' solutions and a few with in-house data checks. " Most utilities rely on what vendors have provided to validate data—obviously not sufficient to prevent errors in this case. Problem resolution is *ad hoc*, rather than systematic, although records are kept of corrections and problem resolutions. The literature is almost silent regarding data validation techniques, whether on-line or off. One approach has been to utilize dual or triple redundant feeds to minimize the need for interpolation or correction. State estimators can now work in real time to help estimate and correct measurement errors.

Performance reports assume four nines of availability, but in practice you're lucky to get one: 35–90% availability is typical. This system, if you want to dig into the data from 2012, became much more reliable and the errors are intermittent rather than persistent now.

For instance, `data/2014/04/01-APR-2014 13_46_11.300_TO_02-APR-2014 07_20_33.400.csv` contains different status flags than we normally expect (`'Old'`). The PMU `_STATIONA-PM1` also has a 2 Hz sampling rate rather than the 10 Hz rate the other data sources use.

Proper data validation—or rather, best practices—is actually an open question in PMU data analysis right now.

Now in this case, because we have several samples per second, it may be all right to simply omit the offending lines. But in any case, I need to code up a tool separately to clean these data first.

Event Detection and Statistics

There are two components to the analysis at this stage: event detection and statistics. Event detection consists of scanning the data for threshold-level deviations or behaviors. Statistics then is

For intra-area observation, there are a few kinds of events we may be interested in:

- Power excursions, indicated by sudden changes in voltage.
- Long- and short-term oscillations, indicated by frequency changes.
- Drift in voltage over an interval, indicated by comparisons of minima and maxima.

We would also like to be able to compare events across PMUs and data stream types as well—this is straightforward because of the high sampling rate, although the data are gappy.

```
def main():
    # Set default time and range.
    time = datetime(1900, 1, 1)
    time_range = [datetime(1900, 1, 1), timedelta(0, 0, 0)]

    # Retrieve the list of all files.
    data_files = get_file_names()

    # Get the date and time spans covered by the files.
    data_spans_con = get_data_spans_con(data_files)
    data_spans     = get_data_spans(data_files)
    if (time == datetime(1900, 1, 1)):
        time = datetime(2013, 7, 7, 23, 45, 0)

    # Look up a date range.
    if (time_range[1] == timedelta(0, 0, 0)):
        time_range = [time, timedelta(12, 0, 0)]
   print "Identifying files for date range %s to %s."%(time_range[0].strftime('%d-%b-%y %H-%M-%S')
    data_file_range = get_files_for_time_span(data_spans, data_files, time_range)

    # Load data from the files and store fields as well.
```

```
print "Loading data for the following files now."
data_raw = []
fields   = []
for file in data_file_range:
    print ' %s'%file[8:]
    (df, flds) = get_raw_data_for_file(file)
    data_raw.append(df)
    fields.append(flds)

# Filter data by 11-point median filter.
print "Filtering data for those files now."
data = []
for (i, d) in enumerate(data_raw):
    data.append(filter_extreme_values(d, fields[i], 100))

# Write filtered data out.
print "Writing filtered data out."
for (i, file) in enumerate(data_file_range):
    print ' %s'%file[8:]
    for datum in data:
        datum.to_csv(get_data_dir()+file, index=False)

# Scan filtered data for power excursions.
for (i, file) in enumerate(data_file_range):
    print ' %s'%file[8:]
    for datum in data:
        fil = find_excursions(datum, fields[i])
print fil

import matplotlib.pyplot as plt
plt.plot(data[0]['_FNET1042-PM1:UTKV'][::100])
plt.plot(fil['_FNET1042-PM1:UTKV'][::100])
plt.show()
```

Incidentally, we run into a bit of trouble here because `_STATIONA-PM1` has a 2 Hz sampling rate. This leads to a lot of empty values in its fields which makes it difficult to calculate rolling median and standard deviation values on the same basis. One way we could deal with this is to calculate and store the expected sampling rate if we wanted to make this more robust. We will neglect this problem since it is irrelevant to the larger picture right now.

The sudden addition or removal of large amounts of load or generation in a power system leads to changes in frequency. For example, a generator trip causes a decline in frequency, whereas load shedding results in an increase in frequency. The change in frequency is proportional to the size of the tripped generator or the amount of load shed. These changes propagate in both space and time throughout the grid.

2.3 Cloud Analytics

Amazon Web Services Setup

Overview of architecture

Recollect the cloud architecture overview yesterday. One of the reasons that is so relevant to Big Data analysis is that it facilitates a sophisticated system for producing a scalable number of machines with a given software stack and running certain programs in tandem.

We are particularly concerned with three pieces of Amazon Web Services: Elastic Cloud Compute (EC2), Elastic MapReduce (EMR), and Simple Storage Service (S3). EC2 is a scalable provider of compute servers. Basically, you request a number of virtual machines to run an identical software stack, and then spin those up and execute your code on them, or host web pages or services or interfaces. EMR is a service on EC2 specifically for running Apache Hadoop. We'll take a look at how EMR works in a minute. S3 is an online hosting service which also allows you to mount data directly to the cloud instances of EC2 and EMR. We are using it to host the virtual machine for this workshop, for instance, as well as the data which we will operate on momentarily.

Starting instances

This VM has a dedicated Hadoop user account, a recommended practice to separate the Hadoop installation from other software applications and user accounts on the same machine.

```
$ sudo login hduser
```

The password for everything is `cse-user1`.

Start up nodes—then `jps` to verify that running.

```
hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.4.0.jar
```

```
hdfs dfs -rmr /user/hduser/output*
```

```
hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.4.0.jar
```

```
$ hdfs dfs -cat /user/hduser/output/part-r-00000
38        <value>
38        <property>
38        <name>
31        <description>
7         <configuration>
```

1. Format the filesystem:

   ```
   $ hdfs namenode -format
   ```

2. Start NameNode daemon and DataNode daemon:

   ```
   $ start-all.sh
   ```

3. Browse the web interface for the NameNode, available at `http://localhost:50070/`. 4. Make the HDFS directories required to execute MapReduce jobs:

```
$ hdfs dfs −mkdir /user
$ hdfs dfs −mkdir /user/hduser
$ hdfs dfs −mkdir /user/hduser/input
```

5. Copy the input files into the distributed filesystem:

```
$ hdfs dfs −copyFromLocal $HADOOP_HOME/etc/hadoop /user/hduser/input
```

6. Run an example, one which here counts the number of instances of each string matching the regular expression:

```
$ hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop−mapreduce−examples −2.4.0.
  grep /user/hduser/input/hadoop /user/hduser/output '<[a−z.]+>'
```

7. Examine the output files. a. Copy output files from the distributed filesystem to the local filesystem and examine them:

```
$ hdfs dfs −get output output
$ cat output/*
```

b. View output files on the distributed filesystem:

```
$ hdfs dfs −cat output/*
```

8. When done, stop the daemons with:

```
$ stop−dfs.sh
```

We need to mount and access our data, as well as write out the results of our Hadoop calculation.

Processing with Hadoop Streaming

How Hadoop streaming works

Apache Hadoop is an open-source software framework used to process large data sets at scale on commodity hardware. There are several pieces of Hadoop, including file storage and resource management, but we're interested in the MapReduce algorithm, which implements . Hadoop is typically invoked either through Java or at a higher level through specialty languages like Pig and Hive. Hadoop Streaming is a way of extending access to the MapReduce algorithm to any programming language by using the standard input and standard output streams to control and map input and output key/value pairs. Hadoop Streaming is invoked as

To understand the logistics of this, consider two scripts, `mapper.sh` and `reducer.sh` which can be executed in a pipe chain thus:

```
cat inputFile | mapper.sh | reducer.sh > outputFile
```

To execute properly, we clearly require each script to read incoming data directly from `stdin` and write the results of the process to `stdout`. The main difference from Hadoop Streaming is that this version is local and serial, with all data piped through a single process chain. Hadoop Streaming, in contrast, will partition and deliver the data in chunks to each process automatically.

Let's examine the same

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
    -input myInputDirs \
    -output myOutputDir \
    -mapper /bin/cat \
    -reducer /bin/wc
```

In this case, `myInputDirs` contains the

There are some subtleties to the convention used: the prefix of a line up to the first tab character is the key and the rest of the line is the value. This can be customized somewhat, but among other things it means that the data structures we relied on locally (such as the Python `pandas` DataFrame) aren't effectively utilized here.

For PMU data, we need to specify two things in the key: the data header and the time stamp. (Note that this increases the intermediate data size enormously if all events were included.) The value will contain the peak standard deviation of the threshold event and its index in the main data set.

In principle, we should be able to draft the analysis code thus:

```
cat data/2012/12/20-DEC-2012\ 05_32_25.600_TO_20-DEC-2012\ 20_04_04.000.csv | ./map
```

The shell scripts `map.sh` and `reduce.sh` simply execute Python scripts:

```
python2 pmu_map.py
```

```
python2 pmu_reduce.py
```

Actually, in many cases you may only be interested in the `Map()` process; for instance, we will map the PMU value inputs to threshold events in the voltage fluctuation. In this case, we can either directly process these data with a `Reduce()` process, or we can write the data to disk and use them at a later time in another program. For a large data set, the main idea would be to quickly and efficiently yield the threshold events either live or for postmortem analysis. These values can then be broken out separately and studied for internal and external correlations with other data. For instance, we can check to see if threshold events occur across more than one data source within a short period of time. We can also integrate these data with other data sets, which we will do in a moment. (Recall as well that Hadoop is more than just MapReduce, although that's what we're using in this demonstration.)

```
$ hadoop-daemon.sh start namenode
$ hadoop-daemon.sh start datanode
$ jps
$ hadoop fs -mkdir /user
$ hadoop fs -mkdir /user/pmu
$ hadoop fs -copyFromLocal pmu_*.py /user/pmu #more explicit than put
```

Types of events to query

The MapReduce algorithm is only well-adapted to describe certain categories of parallel problems, ones which require very little interthread communication. The `Map()` operation filters and sorts the data in preparation for the `Reduce()` procedure, which carries out some summary process. We are thus limited in the range of queries we can make in the data (or, alternatively, we can iterate the `Map()` and `Reduce()` phases to achieve more complicated analyses). So what kind of queries on these date would MapReduce be favorable to carry out?

- Power excursions
- Times when x sources of data are active
- Ranges over which data exist

Running queries

I have set up a couple of Python scripts using the same code base as `load_pmu_csv` which scan the code for power excursions, defined (as before) as sufficiently large variations in the standard deviation. The `Map()` function in this case converts the raw input CSV data to a list of standard deviations over each sampling window. In many cases, that may be sufficient information, and you wouldn't need to reduce. (Run it like this.) But if we want to process it further, then we also have `pmu_reduce.py`. The `Reduce()` function scans the input for threshold values, extracts the time stamps, and returns the time stamp and peak standard deviation.

Let's run this and see what we get for the entire data set.

Interpreting results

(come back to this if takes too long)

2.4 Integrating Heterogeneous Data Sets

Let's go back to our local machines. Next, we will integrate other data sets to see how that might work.

Geography

I have available anonymized latitude and longitude coordinates of the sites. This can allow us to visualize changes in the grid behavior over time and space at the several sites.

One question to think about is how to integrate data which are discrete at different points. But in this case it is fairly apparent how to do it—imagine some other problems: working with genomic and pathology data; or working with epidemic data, incubation times, and air and land travel routes and rates; or working with real-time traffic data and police and ambulance vehicle locations, and then modeling a disaster scenario.

Meteorology

We can jump over to NOAA and check, but their normals hourly category doesn't include the relevant time spans for us. http://www.ncdc.noaa.gov/cdo-web/datasets We have a few options sampled at a high frequency and available over our time spans—we'll use the exposure that Weather Underground provides. (Show full METARS.) http://www.wunderground.com/history/airport/KCMI/2014/5/21/DailyHistory.html?&theprefset=SH
The data are natively stored in this rather cryptic format, but this site actually extracts it and exposes a CSV interface we can use to just download the data directly into a DataFrame. http://www.wunderground.com/history/airport/K

So we have data sampled at an extremely high and regular frequency which needs to be integrated with data only occasionally sampled and then at irregular intervals. You could also be concerned about the curve fitting techique used for the climate data: what I have written simply linearly interpolates.

Other data sets

https://data.cityofchicago.org/Transportation/Chicago-Traffic-Tracker-Congestion-Estimates-by-Re/t2qc-9pjd

2.5 Conclusion

One of the major differences between a research project and a deployment is the shift to using and integrating real-time data to anticipate or analyze events online, rather than retrospectively.

Another difference between everything I've shown you just now and many systems is that none of this has used a traditional relational database for access—SQL. We've just read in raw data and processed it—we haven't had to deal with querying a separate database, which is often a factor. Then you have to consider how to store the heterogeneous data on different time and space grid points.