Big Data Architecture - SEIS736

Wade Lykkehoy

Student ID: 101217575

[lykk3260@stthomas.edu](mailto:lykk3260@stthomas.edu)

Project Paper

**Project Overview**

For the project, I chose to perform analysis on the logs of visitors to the White House during the Obama Presidential administration. At a high level, the project consisted of reading in the White House visitor log data, performing necessary data cleansing, then producing a variety of descriptive statistics. This follows a pattern I have used frequently when working with dumps of data in a batch manner. I also loaded my personal contact data and matched against the visitor logs.

I ran small data sets locally on a VM and the full data set on the St Thomas cluster. On the cluster, I experimented with configuration options, persistence, and serialization to see effect on run times.

The following sections will cover:

* Data Used – data used in the project, where it came from, structure, and general notes
* Processing the Data - approach, lessons learned, results
* Running on the Cluster - determining run-time parameters
* General Observations and Ponderings
* Appendices – results and scripts used to run locally and on the cluster

**Data Used**

I used two sources of data, White House visitor logs and my personal contacts.

White House Visitor Logs

The White House visitor logs are a set of files containing record of visitors to the White House during the Obama administration. The data consists of 6 files containing a total of approximately 6 million records with a total size of approximately 1.1 Gig. The data was obtained from the Obama White House Archives website at the following URL:

<https://obamawhitehouse.archives.gov/goodgovernment/tools/visitor-records>

The data files contain one row per recorded visitor with comma separated fields and a header row. The record field definition follows. For my analysis, I only required a subset of the fields, which I highlighted in green:

|  |  |  |
| --- | --- | --- |
| **Field #** | **Field Name** | **Description** |
| 1 | NAMELAST | Visitor’s last name |
| 2 | NAMEFIRST | Visitor’s first name |
| 3 | NAMEMID | Visitors middle initial |
| 4 | UIN | Appointment number |
| 5 | BDG NBR | Badge number |
| 6 | Access Type | Type of access to the complex |
| 7 | TOA | Time of arrival |
| 8 | POA | Post of arrival |
| 9 | TOD | Time of departure |
| 10 | POD | Post of departure |
| 11 | APPT\_MADE\_DATE | Date the appointment was made |
| 12 | APPT\_START\_DATE | Date and time for which the appointment was scheduled |
| 13 | APPT\_END\_DATE | Data and time for which the appointment was scheduled to end |
| 14 | APPT\_CANCEL\_DATE | Date the appointment was cancelled, if applicable |
| 15 | Total\_People | The total number of people scheduled for a particular appointment per requestor |
| 16 | LAST\_UPDATEDBY | Identifier of officer that updated the record |
| 17 | POST | Computer used to enter the appointment |
| 18 | LastEntryDate | Date of most recent updated to the appointment |
| 19 | TERMINAL\_SUFFIX | Identifier of officer that entered the appointment |
| 20 | visitee\_namelast | Last name of the visitee |
| 21 | visitee\_namefirst | First name of the visitee |
| 22 | MEETING\_LOC | Building in which the meeting is scheduled |
| 23 | MEETING\_ROOM | Room in which the meeting is scheduled |
| 24 | CALLER\_NAME\_LAST | Last name of the individual that submitted the WAVES request |
| 25 | CALLER\_NAME\_FIRST | First name of the individual that submitted the WAVES request |
| 26 | CALLER\_ROOM | Room from which the appointment was made |
| 27 | Description | Comments entered by the WAVES requestor; typically contains the purpose of the visit |
| 28 | RELEASE\_DATE | No description provided |

Quality of data was unknown. I anticipated needing to do a fair amount of data cleansing.

My Personal Contacts

My personal contacts were exported from Google Contacts as an Outlook CSV formatted file. It contains 464 records with a header row. The file contains a large number of fields, of which I am only interested in two; first name and last name.

Given this is a dump of data which was manually entered into Google contacts by me, quality and completeness was suspect. Here also, I anticipated the need to do data cleansing.

**Processing the Data**

I split the project into 3 processing phases, creating a separate program for each:

1. Visitor log cleansing – cleansing the White House visitor log records to a level that I can do my planned analysis
2. Generating statistics from the visitor logs
3. Cross-referencing to my personal contacts

Visitor Log Cleansing

The source code for the visitor log cleansing may be found in WHVisitors\_Cleanse.scala.

Given the visitor logs were likely a dump from a system in which data was manually entered, the quality of the data was unknown and suspect. My strategy was to do “just enough” data cleansing to allow me to generate the planned descriptive statistics and match against my contacts.

I built an RDD of data, which included steps for basic syntactic / completeness cleansing as follows:

* Read in the log files
* Split the record at commas, converting to lower case and trimming blanks
* Filter out Incomplete records; those where the number of fields is less than the expected 28
* Filter out header rows
* Filter out records with a blank visitor last name
* Filter out records with a blank visitor first name
* Sanitize the date to be in a consistent format; more on this follows.
* Extract only the fields I would need for my analysis

Date fields are, in my experience, more oft than not a pain to deal with. Inconsistent format, sometimes contain both a date and time, inconsistency with 1 vs 2 digit months and days, inconsistency with 2 vs 4 digit years, etc. I created a helper function to “sanitize” the date into a mm/dd/yyyy format, which I used within a map() statement.

I then had a thought, since I was invoking my own function during a pass through the data records via map(), if I added more to that function I could reduce the passes through the data and gain efficiency. I started creating a super-duper do-all function where I was cramming all data cleansing work. Then I realized I was not thinking in a Spark manner, I was thinking in an old-school serial processing manner. I abandoned the super-duper function and went with multiple map() and .filter() calls, knowing Spark has sophisticated optimization techniques and would optimize based the DAG.

I printed out a count of records before and after the cleansing, along with 20 records for visual inspection.

As a litmus test for whether the data makes sense, I created frequencies of the length of the values in the following fields and wrote out to both stdout and to a file (please see Appendix A - Field Value Length Frequencies):

* Visitor last name
* Visitor first name
* Visitee last name
* Visitee first name
* Description

Printing information to stdout worked very nicely to see the process status and results when running locally (I had previously turned off the INFO messages generated by Spark). However, when running on the cluster, unless I told Spark to run the driver on the local node, not useful for tracking processing status. After processing was complete, I was able to collect the logs and look through them to see what happened. Something I would do differently next time is prefix my prints with something unique such that they could easily be filtered out with a tool such as grep. Dynamic status is an area I would like to explore further as from my experience, having some type of status information, either in text or feeding into a dashboard, is a common pattern particularly when processing streaming data.

Last step for this phase/program was to save the cleansed data for the next phases/programs to use. I first tried saving as object files. However, when I attempted to read it back in, I received errors which I was not able to resolve. So, I went down the path of saving as a text file. First try with that, I got the following in the files:

[Ljava.lang.String;@2f19b9ba

[Ljava.lang.String;@5d244857

[Ljava.lang.String;@70eda3a6

Not what I expected and not very useful. With a little tinkering in the spark shell, I found using map() to create an RDD consisting of a comma separated string, then saving that as a text file produced files that looked like what I wanted:

cleansedData.map(rec => rec.mkString(","))

.saveAsTextFile(cleansedLogDir)

adamopoulos,stella,,05/01/2015,,,class visit

brosman,muriel,,05/01/2015,,,class visit

brumfield,avery,,05/01/2015,,,class visit

My victory soon turned into defeat. When loading the data back in, records which had no value for the last field, such as the following, were problematic.

yalamanchili,winston,,05/01/2015,,,

When read back in, the array that was created for these records, did not contain elements for the trailing fields. Thus the above record would result in an array of only 4 elements rather than the expected 6. The JVM got rather cranky when I indexed the 5th and 6th elements. With a bit of tinkering, I found creating a string with an explicit blank character after the comma worked.

cleansedData.map(rec => rec.mkString(", "))

.saveAsTextFile(cleansedLogDir)

adamopoulos, stella, , 05/01/2015, , , class visit

yalamanchili, winston, , 05/01/2015, , ,

Though when I read the records back in, I did have to trim them after splitting.

val theData = sc.textFile(cleansedLogDir)

.map(line => line.split(",").map(\_.trim))

This feels like a “hack” and something I would like to explore further.

Generating Statistics from the Logs

The source code for generating statistics from the cleansed visitor logs may be found in WHVisitors\_GenStats.scala.

Now that I had clean (or “clean enough”) visitor log data that I could read back in correctly, time to generate some statistics.

The basic process/program flow was:

* Read in the saved cleansed data, splitting on commas and removing blanks, creating an RDD.
* Create an RDD with the frequency of unique values in field Description and write to a file.
* Create an RDD with the Frequency of visitees, based on their combined last name and first name and write to a file.
* Create an RDD with the count of visitors to the President, which were identified by looking for “potus” in either the visitee last name or first name, and write to a file.
* Create an RDD with the Frequency of visitors to the President, based on their combined last name and first name, and write to a file.
* Create an RDD with the count of visitors to the First Lady, which were identified by looking for “flotus” in either the visitee last name or first name, and write to a file.
* Create an RDD with the frequency of visitors to the First Lady, and write to a file.
* Create an RDD with the percent of visitors on a tour, which were identified by looking for “tour” in the description, and write to a file.
* Create an RDD with the count of tour visitors by month and year and write to a file.
* Create an RDD with the count of tour visitors by season; spring, summer, winter, fall, and write to a file.

The frequencies were straight-forward to create by using the map() function to create a key from one or more fields, and “1” as the value, then using reduceByKey() to get counts. I saved the RDD using saveAsTextFile(). Note I did not have any issues here writing out the RDD as I did with the full records in the cleanse part. I believe this was because here I had simpler pair/value RDDs.

When I ran the program, I got one file per partition for each of the frequencies. What I really wanted was just one file for each, with the entire frequency dump. Thus when calling reduceByKey(), I added an explicit value of 1 for the number of partitions to produce.

In viewing the frequencies, I thought about sorting. What I desired was to sort descending by value to place the items with highest frequency counts at the top. Then within a particular count value, sort ascending by key. Function sortByKey() would get me part way there. Function sortBy() was what I needed. I constructed a complex key consisting of a tuple with 0 – value as the first member and the key as the second:

.sortBy(kvpair => ((0 - kvpair.\_2), kvpair.\_1))

Counts were also straight forward to create. I used filter() to select out the records of interest, then count() to get the count. I desired to write these counts out to a file and pondered how to do that easily. I ended up creating a simple pair/value RDD and saving to a text file. Worked great and was a single, simple line of code:

sc.parallelize(Array(("POTUS", count\_cp)), 1)

.saveAsTextFile(saveDir\_cp)

In this process/program, I experimented with explicitly persisting RDDs. I persisted the cleansed data RDD after loading and splitting as that would be used as the base for subsequent frequency and count generation. I also persisted an RDD where I filtered out records that had the word “tour” in the descriptions. In timing trials on the cluster, I saw no difference. Both with and without persisting ran in just a little over 1 minute (I was the only process running on the cluster at the time). I suspect the reason I did not see any difference is that my RDDs were small enough to be cached in memory and Spark was smart enough to do just that throughout the processing.

I tried reducing the number of executors and cores to see what the effect would be on run time:

|  |  |  |
| --- | --- | --- |
| Cores | Executors | Run Time  (avg over 3 runs) |
| 5 | 17 | 1:05 |
| 5 | 8 | 1:03 |
| 2 | 17 | 1:04 |

No noticeable difference in run time. At first I was puzzled by this as I expected as the cores or executors decreased, the run time would increase due to reducing the number of processes that would run in parallel. I took a step back and thought through how Spark would process my data. I had 12 data files feeding into this phase/program, each no larger than 45 meg. Thus, I would expect Spark to create one partition per input file, resulting in 12 parallel processes. I added a print statement to confirm the number of partitions; it was indeed 12. So, the maximum number of parallel processes I was achieving was 12. Having more than 12 total cores would not improve performance and I was actually wasting quite a large amount of compute power.

Curious to see if I would get a performance boost by explicitly setting the number of partitions to a value greater than 12, I modified the textFile() call to specify 24 partitions; two for each input file.

val theData = sc.textFile(cleansedLogDir, 24)

.map(line => line.split(",").map(\_.trim))

I did see a slight decrease in run time to 0:56. I suspect not seeing a larger decrease in run time is telling me that the overhead of Spark is sizable relative to my application’s execution time. I also noticed that rather than 24 partitions, I had 30. Interesting.

To be somewhat “evil”, I ran specifying 1 core and 1 executor. Run time increased quite a bit to 1:46. I expected an increase, though larger. I suspect the reason that it was not larger is again, Spark’s overhead relative to my applications execution time.

I was curious to see if enabling Kryo serialization would have any affect on run time. I added setting the configuration for serialization:

val sparkConf = new SparkConf()

sparkConf.set("spark.serializer",

"org.apache.spark.serializer.KryoSerializer")

val sc = new SparkContext(sparkConf)

I expected to see very little to no difference as the amount of data being shuffled within my program was relatively small. Test runs showed no noticeable difference in run time.

The frequency of unique values in field Description brought out some interesting reasons for visiting (some noted in Appendix B). It would be fun to do further analysis. It appears Obama’s had numerous folks over for bowling!

I was surprised with the relatively small count of visitors to the First Lady, 27,575. I would have expected more.

I was also surprised by the number of unique visitors to the President, 159,612. That is a lot of visitors. Perhaps many of the visits that were recorded were larger meetings where the President was present, vs 1-on-1 meetings.

The percent of visitors to the White House who were on a tour (having the word “tour” in the description) was approximately 61%. That does not seem out of line with what I would expect.

When I ran the count of visitors by month and year, I noticed a couple things in the result:

* A record where a bad date got through the data sanitizing from the cleansing step. I really dislike dates… Given it was only 1, I did not spend any additional time tracking it down.
* I had no visitor counts for May and June of 2013, and unusually small counts for April, July, August, September, and October of 2013. It would be interesting to look for those months in the raw data to see if it is truly missing for that period of time or there was an unusually high percent of bad records, which were dropped during cleansing.

Cross Referencing to my Personal Contacts

The source code for cross referencing to my contacts may be found in WHVisitors\_XrefContacts.scala.

In this process/program, I matched my personal contacts against the visitor logs. As a key, I used first name and last name. Under normal circumstances, we would want more to match on, however for this exercise it would be sufficient.

I first read in my personal contacts and performed some cleansing as follows:

* Read in the contact data, splitting on commas, converting to lower case, and removing blanks.
* Filter out records that do not at least contain the first 3 fields; first name, middle name, and last name
* Filter out header rows
* Filter out records with a blank last name
* Filter out records with a blank first name
* Create a value with the first and last name concatenated together

This produced a simple RDD with elements of the form:

[lastname, firstname]

I then read in the cleansed log data, also creating a simple RDD of values in the exact same format as for my contacts.

Now I could do the cross-reference. Initially, I thought I would use join() or one of the variants. Once I got to this point and thought through what I was doing, I realized those are meant to work on key/value pair RDDs. Hmmm, I just had simple RDDs with single values. After searching and reading, I landed on the intersection() function. I added a sort and a dump of the resulting RDD to a file (please see Appendix C – White House Visitor Names Matching Personal Contacts).

I expect some matches due to common names in both sources. I was quite surprised that there were 111 matches. While many of the matches were on common names, some of the matches were on names that I would consider rather unique. Next time I see some of those individuals, I will ask them if they were the ones who visited the White House.

**Running on the Cluster**

I processed subsets of the data locally on my VM and the full dataset on the St Thomas cluster. For cluster configuration parameters, I followed advice given by Mark Grover and Ted Malaska in a recorded presentation they gave at the 2016 Spark Summit, titled “Top 5 Mistakes When Writing Spark Applications” (<https://databricks.com/session/top-5-mistakes-when-writing-spark-applications>), which is:

* Between 4 and 6 cores per executor generally gives the best HDFS throughput.
* Reserve 1 core per node for Hadoop/Yarn daemons.
* Reserve 1 Gig of RAM per node Hadoop/Yarn daemons and miscellaneous overhead.
* When specifying executor memory, keep in mind that Yarn adds an additional 7% to what you ask for.
* The application master will run in one of the executors, thus reserve one for that.

Applying this advice to the St Thomas cluster in a manner that would max out the resources:

Total available cores in the cluster:

16 nodes \* 11 cores per (reserving 1 core per node) = 176 total cores

Let’s go with 5 cores (mid-way between the 4 and 6 recommended) per executor. Calculating the maximum number of executors:

176 total cores / 5 cores per executor = 35.2 executors (round down to 35)

35 executors / 16 nodes = 2.2 executors per node (round down to 2)

Memory per executor:

Memory per node = 64 Gig – 1 Gig for Hadoop/Yarn = 63 Gig

63 Gig memory / 2 executors per node = 31.5 Gig per executor

31.5 Gig – 7% for Yarn overhead = 29.29 Gig per executor (round down to 29)

Giving the following:

--executor-cores 5

--num-executors 35

--executor-memory 29g

Since I want to be nice to other users of the cluster, I went with half the number of executors. Also, given my dataset is just over 1 gig in size and even if I were to persist multiple RDD’s in memory, 29 Gig seems excessive. I cut that down to 16 Gig. I soon found that on the St Thomas cluster, there is a maximum memory limit of 16 Gig. So, I reduced down to 12 Gig.

--executor-cores 5

--num-executors 17

--executor-memory 12g

Please see Appendix D – Script to Run Locally and Appendix E – Script to Run on Cluster for the complete set of options used.

**General Observations and Ponderings**

I continue to be impressed with Spark and the ease by which you can create programs which take advantage of parallel processing.

One of the “mindset” aspects I had to overcome in this project was allowing myself to be ok with chaining together several small operations rather than cram as much logic into each operation as I could. Letting Spark do what it is meant to do, which is optimizing execution based on the resulting DAG.

I initially developed the code in the Spark Shell, then pasted into a Scala program. A couple things I quickly encountered. The Spark Shell lets you re-use names to your heart’s content. Within a Scala program, you cannot do that. Another is that for readability, you can split long statements across lines in a Scala program. However, you cannot then just copy and paste back into the Spark shell. Multiple resources I had seen advocated one of the benefits around the Spark ecosystem is being able to paste code back and forth. However, to do that, you do need to follow some best practices/guidelines.

I used Eclipse for developing the Scala programs. I started with the provided example and basically just plugged my code in. One challenge I ran into is that Eclipse is configured with an absolute path for source files rather than a relative path using the project name (it points to the pre-loaded training exercises folder). Until I figured this out and changed it, I kept banging my head against the wall as to why each project I created had all source files in it. And when I deleted any source file, it deleted from all projects. Perhaps an item to note in instructions for those who will use Eclipse for their projects.

I would also like to explore good patterns and best practices for debugging Spark programs via print traces or other means. As you mentioned in class, and I certainly found, when running on a cluster, print gets rather lost in the log files. I have also found from past experience, print traces in a concurrent parallel environment may not tell the entire story.

With the tinkering I did with the num-executors and executor-cores parameters in the GenStats process/program, I can see to maximize performance, you need to spend time thinking about your application and how Spark will run it. Just throwing more executors or cores at your program may not be beneficial. I am somewhat surprised Spark does not have some kind of “tuning advisor” functionality/capability to make suggestions on run-time parameters and application code based on the analysis and RDD optimization that it performs plus run-time monitoring. Perhaps there is an opportunity here for some enterprising individual.

I ended up creating several Bash helper scripts for running the applications, loading data into HDFS, copying files from my local development environment to the cluster, and retrieving output (created files) from HDFS. It seems there are several patterns that an IDE or other tool could/should help you manage.

I used only RDD’s for the project. It would be interesting to go back through and see where DataFrames could be used. It would also be very interesting to explore other Spark add-ons for the analysis and for creating plots. In my other class, we are using R for data analysis and regression. I have been very impressed with its power. It would be very interesting to explore SparkR and the capabilities of R and Spark combined.

I found that I used HUE a lot when developing and testing my code locally on my VM. It was an easy way to quickly view the output files, remove them for re-running, etc.

I see a lot of potential for working with streaming data. My experience is streaming data is becoming (or perhaps has become) more prevalent and important than working with data in batches. Also an aspect I am very interested in learning more about. A suggestion for future versions of the class.

**Appendix A – Field Value Length Frequencies**

Frequency of the length of values for the visitor last name (NAMELAST):

(1,76)

(2,20336)

(3,111457)

(4,529973)

(5,1120031)

(6,1368710)

(7,1118805)

(8,759999)

(9,395933)

(10,198324)

(11,92548)

(12,50048)

(13,31760)

(14,22127)

(15,14000)

(16,8043)

(17,4394)

(18,2054)

(19,1106)

(20,1311)

(21,6)

(22,3)

(23,5)

(25,3)

(28,1)

Frequency of the length of values for the visitor first name (NAMEFIRST):

(1,1969)

(2,6899)

(3,148904)

(4,806281)

(5,1394862)

(6,1392080)

(7,1211081)

(8,507191)

(9,273692)

(10,39279)

(11,53417)

(12,5505)

(13,4165)

(14,2403)

(15,1700)

(16,573)

(17,373)

(18,211)

(19,159)

(20,287)

(21,4)

(22,6)

(23,5)

(24,1)

(25,2)

(26,2)

(27,1)

(29,1)

Frequency of the length of values for the visitee last name (visitee\_namelast):

(0,302218)

(1,26929)

(2,21810)

(3,54430)

(4,189189)

(5,492236)

(6,3849940)

(7,374320)

(8,256279)

(9,160020)

(10,72781)

(11,23589)

(12,14571)

(13,4211)

(14,3224)

(15,2693)

(16,874)

(17,1413)

(18,27)

(19,157)

(20,137)

(24,5)

Frequency of the length of values for the visitee first name (visitee\_namefirst):

(0,140958)

(1,954)

(2,13275)

(3,141665)

(4,423436)

(5,695688)

(6,402734)

(7,329306)

(8,3508169)

(9,71405)

(10,7921)

(11,12620)

(12,56064)

(13,1588)

(14,44358)

(15,504)

(16,56)

(17,44)

(18,17)

(19,264)

(20,27)

Frequency of the length of values for the description (Description).

(0,1786019)

(1,10868)

(2,4263)

(3,1842)

(4,11296)

(5,168611)

(6,78560)

(7,61212)

(8,13766)

(9,9550)

(10,1848316)

(11,184887)

(12,55946)

(13,81813)

(14,297021)

(15,28961)

(16,24280)

(17,45852)

(18,50636)

(19,41994)

(20,32962)

(21,15330)

(22,7023)

(23,10686)

(24,15494)

(25,10706)

(26,14246)

(27,10704)

(28,11230)

(29,7194)

(30,7599)

(31,8251)

(32,25442)

(33,6122)

(34,6619)

(35,5283)

(36,4238)

(37,7901)

(38,5856)

(39,7101)

(40,624235)

(41,46541)

(42,3666)

(43,6826)

(44,8506)

(45,9670)

(46,14714)

(47,8995)

(48,2468)

(49,3946)

(50,13730)

(51,3365)

(52,6005)

(53,6519)

(54,3942)

(55,2233)

(56,2044)

(57,1350)

(58,3306)

(59,1308)

(60,2034)

(61,2936)

(62,1425)

(63,760)

(64,1107)

(65,2053)

(66,871)

(67,2063)

(68,830)

(69,780)

(70,528)

(71,1538)

(72,2027)

(73,1830)

(74,555)

(75,1266)

(76,944)

(77,1425)

(78,6553)

(79,737)

(80,764)

(81,1602)

(82,1236)

(83,375)

(84,525)

(85,119)

(86,220)

(87,840)

(88,642)

(89,884)

(90,185)

(91,724)

(92,438)

(93,801)

(94,48)

(95,311)

(96,373)

(97,147)

(98,730)

(99,276)

(100,490)

(101,1604)

(102,198)

(103,76)

(104,221)

(105,29)

(106,55)

(107,202)

(108,284)

(109,159)

(110,248)

(111,328)

(112,107)

(113,86)

(114,300)

(115,196)

(116,39)

(117,77)

(118,70)

(119,76)

(120,15)

(121,51)

(122,66)

(123,658)

(124,525)

(125,38)

(126,59)

(127,203)

(128,46)

(129,102)

(130,33)

(131,66)

(132,87)

(133,49)

(134,47)

(135,204)

(136,125)

(137,547)

(138,61)

(139,116)

(140,119)

(141,234)

(143,1)

(146,1)

(148,4)

(149,1)

(169,7)

(170,1)

(183,6)

(206,182)

(213,2)

(214,1)

**Appendix B – Generated Statistics**

Frequency of unique values in field Description

There were 18,873 unique values; the top 25:

(group tour,1828521)

(,1786019)

(ew tour must enter through sherman park,619319)

(west wing tour,275210)

(tours,163176)

(group tours,130397)

(tours/,65838)

(ew tour must enter through sherman park,43753)

(group tours /,36922)

(group tour/,36318)

(group tours/,34646)

(the event is on the state floor.,19784)

(group tours\*\*,17511)

(holiday open house,16819)

(tours /,16593)

(ww tour,14443)

(holiday reception,14365)

(tours./,13135)

(group tours./,12365)

(f,10854)

(holiday open house/,9394)

(tour,8825)

(all appointee event,8269)

(meeting,6632)

(state arrival - mexico\*\*,6524)

A few that I happened upon which I found “interesting”:

(this is a bowling event.,8)

(bowling guests,2)

(coming for a site visit for kitchen garden project.,2)

(ice sculpture delivery to the eeob,2)

(christmas card display consultation,1)

(ms. weaver was unable to get her flu shot yesterday,1)

(she is riding on af1 to san francisco and needs to get to the oval office for departure.,1)

(tow truck driver,1)

(trick or treat event,1)

(xerox technician will be repairing a copier in staircase,1)

Frequency of visitees, based on their combined last name and first name

There were 21,127 unique values; the top 25:

([office, visitors],3374376)

([, potus],190762)

([potus, ],74486)

([, potus/flotus],51652)

([waves, visitorsoffice],44129)

([, ],28962)

([/, potus],23001)

([lierman, kyle],21824)

([lambrew, jeanne],21272)

([, flotus],20647)

([bryant, ruth],13967)

([burton, collin],13586)

([oneil, olivia],13233)

([thompson, jared],11574)

([doebler, max],11332)

([hetzel, office],10688)

([foster, heather],10245)

([monteiro, paul],10073)

([raghavan, gautam],9519)

([bollinger, chelsea],8673)

([nelson, greg],8496)

([megan, matthew],8493)

([jenkins, brad],8335)

([utech, dan],7966)

([mccullough, victoria],7466)

Count of visitors to the President

(POTUS,295385)

Frequency of visitors to the President

There were 159,612 unique values; the top 25:

([prather, alan],219)

([mottola, annamaria],163)

([pelosi, nancy],134)

([rose, christopher],129)

([wilson, russell],122)

([clay, aaron],117)

([sabo, eric],114)

([boguslaw, robert],107)

([dewey, glenn],106)

([fettig, jason],89)

([hoyer, steny],84)

([rakers, michelle],83)

([schmitt, christopher],77)

([reid, harry],76)

([wilson, peter],76)

([murray, david],72)

([durbin, dick],69)

([ridlington, gregory],68)

([smith, michael],67)

([conyers, john],65)

([colburn, michael],62)

([franke, christopher],62)

([sato, fern],62)

([turnmire, brian],62)

([botelho, marcio],60)

Count of visitors to the First Lady

(FLOTUS,27575)

Frequency of visitors to the First Lady

There were 21,784 unique values; the top 25:

([mottola, annamaria],33)

([prather, alan],29)

([sprow, stephanie],23)

([schmitt, christopher],20)

([wilson, russell],20)

([goslins, rachel],19)

([laackman, allyson],18)

([dewey, glenn],16)

([boguslaw, robert],15)

([clark, lindsey],15)

([fletcher, kathleen],15)

([slaterrigaud, traci],14)

([august, hannah],13)

([kennedy, victoria],13)

([lion, margo],13)

([lynch, robert],13)

([woetzel, damian],13)

([craine, kimber],12)

([hicks, hillary],12)

([lechtenberg, tyler],12)

([parsons, alannah],12)

([rose, christopher],12)

([udall, jill],12)

([witcher, ikea],12)

([sabo, eric],11)

Percent of visitors on a tour

(Percent of visitors on a tour,60.507435)

Count of tour visitors by month and year

(0000\_08,1)

(2009\_02,6)

(2009\_03,65)

(2009\_04,84)

(2009\_05,91)

(2009\_06,129)

(2009\_07,156)

(2009\_08,126)

(2009\_09,16385)

(2009\_10,47851)

(2009\_11,31971)

(2009\_12,49701)

(2010\_01,32126)

(2010\_02,36157)

(2010\_03,59545)

(2010\_04,58616)

(2010\_05,56438)

(2010\_06,70013)

(2010\_07,66467)

(2010\_08,69015)

(2010\_09,35986)

(2010\_10,59484)

(2010\_11,37314)

(2010\_12,64212)

(2011\_01,20604)

(2011\_02,32085)

(2011\_03,65411)

(2011\_04,62835)

(2011\_05,68983)

(2011\_06,67133)

(2011\_07,74547)

(2011\_08,64193)

(2011\_09,49158)

(2011\_10,55170)

(2011\_11,42258)

(2011\_12,73865)

(2012\_01,24937)

(2012\_02,32118)

(2012\_03,65080)

(2012\_04,53362)

(2012\_05,59078)

(2012\_06,70507)

(2012\_07,62265)

(2012\_08,67451)

(2012\_09,45657)

(2012\_10,52889)

(2012\_11,43333)

(2012\_12,76826)

(2013\_01,27010)

(2013\_02,35454)

(2013\_03,29417)

(2013\_04,694)

(2013\_07,5)

(2013\_08,89)

(2013\_09,3)

(2013\_10,2)

(2013\_11,21253)

(2013\_12,24053)

(2014\_01,14530)

(2014\_02,12984)

(2014\_03,38561)

(2014\_04,24313)

(2014\_05,51459)

(2014\_06,47620)

(2014\_07,53562)

(2014\_08,7692)

(2014\_09,34221)

(2014\_10,50082)

(2014\_11,34489)

(2014\_12,49508)

(2015\_01,22605)

(2015\_02,29099)

(2015\_03,48831)

(2015\_04,38295)

(2015\_05,54700)

(2015\_06,46761)

(2015\_07,45354)

(2015\_08,19629)

(2015\_09,33771)

(2015\_10,46364)

(2015\_11,40380)

(2015\_12,59803)

(2016\_01,29440)

(2016\_02,28664)

(2016\_03,50123)

(2016\_04,53168)

(2016\_05,49878)

(2016\_06,62200)

(2016\_07,56617)

(2016\_08,45955)

Count of tour visitors by season; spring, summer, winter, fall

(Fall,778021)

(Spring,989027)

(Summer,997487)

(Winter,775787)

**Appendix C – White House Visitor Names Matching Personal Contacts**

[ackerman, natalie]

[adams, gary]

[anderson, dan]

[anderson, jennifer]

[anderson, jevon]

[anderson, mark]

[bandy, john]

[barnes, cindy]

[bernstein, paul]

[bird, james]

[born, john]

[brandt, craig]

[brandt, jeff]

[brown, deb]

[bruns, scott]

[byrnes, jennifer]

[caldwell, keith]

[campbell, austin]

[campbell, kevin]

[carlson, peter]

[chase, kari]

[chung, jane]

[cook, connie]

[cooper, dawn]

[copeland, andrew]

[crouch, john]

[davila, adam]

[dowd, peter]

[erdmann, craig]

[felton, rebecca]

[fischer, christopher]

[fox, charlie]

[fraser, barbara]

[freedman, michelle]

[freeman, francis]

[gale, jacob]

[gilles, brian]

[harman, henry]

[hickey, paul]

[hickman, jon]

[hill, matthew]

[hitt, charles]

[hunt, peter]

[hyde, william]

[jensen, mark]

[johnson, angela]

[johnson, ben]

[johnson, patricia]

[johnson, taylor]

[johnson, thomas]

[johnston, stephanie]

[jones, bryan]

[jones, tami]

[jordan, jon]

[jordan, naomi]

[kelly, jeff]

[knutson, andrew]

[knutson, michael]

[kuper, debra]

[lane, joe]

[lane, lisa]

[larson, matt]

[lee, gene]

[lewis, nicole]

[lindeman, elizabeth]

[lindsey, shirley]

[mckenzie, maureen]

[medina, pedro]

[meyer, polly]

[mills, eric]

[nelson, bill]

[nelson, david]

[nelson, kyle]

[nelson, rick]

[newton, james]

[nielsen, carrie]

[northrup, john]

[olson, karl]

[olson, richard]

[olson, ted]

[paul, margaret]

[perry, matt]

[peterson, mark]

[pettit, paul]

[preston, kirsten]

[ray, damon]

[richardson, faith]

[riley, eric]

[robertson, aaron]

[ross, linda]

[sanders, chris]

[schmidt, jason]

[schwichtenberg, david]

[seifert, sarah]

[sharp, donald]

[short, melissa]

[singh, arun]

[smith, susan]

[stephens, blaine]

[taylor, tracy]

[thomas, will]

[thompson, robbie]

[thompson, travis]

[vento, shannon]

[waltenbaugh, kurt]

[wang, michael]

[weber, carol]

[weber, mary]

[wiebe, john]

[williams, lori]

[zimmer, donna]

**Appendix D – Script to Run Locally**

#!/usr/bin/env bash

if [ $# -ne 2 ]

then

echo "Usage : $0 dataset process"

exit 1

fi

if [[ "$1" == "VSmall" || "$1" == "OneLog" || "$1" == "Full" ]]; then

case "$2" in

Cleanse)

hdfs dfs -rm -r SparkProjects/WHVisitors/Data/$1/CleansedLogs

hdfs dfs -rm -r SparkProjects/WHVisitors/SparkOut/$1/Cleanse

spark-submit \

--class WHVisitors\_Cleanse \

--master local[\*] \

--name "${USER}-WHVisitors\_Cleanse" \

~/SparkProjects/WHVisitors/WHVisitors\_Cleanse.jar \

SparkProjects/WHVisitors/Data/$1/RawLogs \

SparkProjects/WHVisitors/Data/$1/CleansedLogs \

SparkProjects/WHVisitors/SparkOut/$1/Cleanse

hdfs dfs -ls SparkProjects/WHVisitors/Data/$1/CleansedLogs

hdfs dfs -ls SparkProjects/WHVisitors/SparkOut/$1/Cleanse

;;

GenStats)

hdfs dfs -rm -r SparkProjects/WHVisitors/SparkOut/$1/GenStats

spark-submit \

--class WHVisitors\_GenStats \

--master local[\*] \

--name "${USER}-WHVisitors\_GenStats" \

~/SparkProjects/WHVisitors/WHVisitors\_GenStats.jar \

SparkProjects/WHVisitors/Data/$1/CleansedLogs \

SparkProjects/WHVisitors/SparkOut/$1/GenStats

hdfs dfs -ls SparkProjects/WHVisitors/SparkOut/$1/GenStats

;;

XrefContacts)

hdfs dfs -rm -r SparkProjects/WHVisitors/SparkOut/$1/XrefContacts

spark-submit \

--class WHVisitors\_XrefContacts \

--master local[\*] \

--name "${USER}-WHVisitors\_XrefContacts" \

~/SparkProjects/WHVisitors/WHVisitors\_XrefContacts.jar \

SparkProjects/WHVisitors/Data/$1/CleansedLogs \

SparkProjects/WHVisitors/Data/$1/Contacts \

SparkProjects/WHVisitors/SparkOut/$1/XrefContacts

hdfs dfs -ls SparkProjects/WHVisitors/SparkOut/$1/XrefContacts

;;

\*)

echo "$0 : ERROR : Unknown process $2"

exit 1

;;

esac

else

echo "$0 : ERROR : Unknown dataset $1"

exit 1

fi

exit 0

**Appendix E – Script to Run on Cluster**

#!/usr/bin/env bash

# St Thomas cluster specs:

# Physical nodes: 16; max for submit 15

# Cores per node: 12; max for submit 11

# Mem per node : ??

if [ $# -ne 2 ]

then

echo "Usage : $0 dataset process"

exit 1

fi

if [[ "$1" == "VSmall" || "$1" == "OneLog" || "$1" == "Full" ]]; then

case "$2" in

Cleanse)

hdfs dfs -rm -r SparkProjects/WHVisitors/Data/$1/CleansedLogs

hdfs dfs -rm -r SparkProjects/WHVisitors/SparkOut/$1/Cleanse

spark-submit \

--class WHVisitors\_Cleanse \

--master yarn-cluster \

--executor-cores 5 \

--num-executors 17 \

--executor-memory 12g \

--name "${USER}-WHVisitors\_Cleanse" \

~/SparkProjects/WHVisitors/WHVisitors\_Cleanse.jar \

SparkProjects/WHVisitors/Data/$1/RawLogs \

SparkProjects/WHVisitors/Data/$1/CleansedLogs \

SparkProjects/WHVisitors/SparkOut/$1/Cleanse

hdfs dfs -ls SparkProjects/WHVisitors/Data/$1/CleansedLogs

hdfs dfs -ls SparkProjects/WHVisitors/SparkOut/$1/Cleanse

;;

GenStats)

hdfs dfs -rm -r SparkProjects/WHVisitors/SparkOut/$1/GenStats

spark-submit \

--class WHVisitors\_GenStats \

--master yarn-cluster \

--executor-cores 5 \

--num-executors 17 \

--executor-memory 12g \

--name "${USER}-WHVisitors\_GenStats" \

~/SparkProjects/WHVisitors/WHVisitors\_GenStats.jar \

SparkProjects/WHVisitors/Data/$1/CleansedLogs \

SparkProjects/WHVisitors/SparkOut/$1/GenStats

hdfs dfs -ls SparkProjects/WHVisitors/SparkOut/$1/GenStats

;;

XrefContacts)

hdfs dfs -rm -r SparkProjects/WHVisitors/SparkOut/$1/XrefContacts

spark-submit \

--class WHVisitors\_XrefContacts \

--master yarn-cluster \

--executor-cores 5 \

--num-executors 17 \

--executor-memory 12g \

--name "${USER}-WHVisitors\_XrefContacts" \

~/SparkProjects/WHVisitors/WHVisitors\_XrefContacts.jar \

SparkProjects/WHVisitors/Data/$1/CleansedLogs \

SparkProjects/WHVisitors/Data/$1/Contacts \

SparkProjects/WHVisitors/SparkOut/$1/XrefContacts

hdfs dfs -ls SparkProjects/WHVisitors/SparkOut/$1/XrefContacts

;;

\*)

echo "$0 : ERROR : Unknown process $2"

exit 1

;;

esac

else

echo "$0 : ERROR : Unknown dataset $1"

exit 1

fi

exit 0