Contents

[Data Source 2](#_Toc448697796)

[Data Description And Schema 2](#_Toc448697797)

[Data pre-processing required (parsing, filtering) 2](#_Toc448697798)

[Any bad data issues 2](#_Toc448697799)

[MR algorithm 2](#_Toc448697800)

[Description of any other ecosystem or additional tools 2](#_Toc448697801)

[Output description 2](#_Toc448697802)

[How did you verify that your output is correct? 2](#_Toc448697803)

[Performance/scale characteristics 2](#_Toc448697804)

[What would you have done differently if you did this again? 2](#_Toc448697805)

[Conclusions 2](#_Toc448697806)

[Appendix A – Java Source Code for MapReduce 2](#_Toc448697807)

[Appendix B – Spark Scala Source Code 2](#_Toc448697808)

[Appendix C – Job Output History for MapReduce 2](#_Toc448697809)

## Data Source

Checkpoint is one of the premier Tax and Accounting products from Thomson Reuters. The top 100 U.S. CPA firms rely on Checkpoint as their information resource for tax, audit and accounting, international trade, benefits and finance. (<https://tax.thomsonreuters.com/products/brands/checkpoint/advantages/>)

I have joined the web development team last summer (2015). The team has been logging user search logs since 2010. The logs have been in a NAS drive and is not really being used for anything other than tracing errors. Our searches in Checkpoint is passed on an enterprise search platform so it makes sense logging user searches in order to trace errors if the search from the enterprise platform fails. The user searches that I am going to analyze for this project has a date range from 2010-09-01 to 2016-03-06. The zipped logs are about 10 GB.

## Data Description and Schema

The log data that I have uploaded in the cluster is a tab delimited text file that is variable in length.

The schema of the log file is as follows

fileCount+”\_”+sessionId+ “\t” +SearchKeyword+”\t”+practice+”\t”+userType+”\t”+searchType+”\t”+tab location+”\t”+version+”\t”+search results

10\_01BCEA5099D956DCE55F349110EEBF72 1.1031(k) 1 2 TC 60 v1 366

Sample log file entries:

10\_01BCEA5099D956DCE55F349110EEBF72 1.1031(k) 1 2 TC 60 v1 366

10\_01BCEA5099D956DCE55F349110EEBF72 1.1031(k)-1(g)(6)\* 1 2 TC 60 v1 65

10\_01BCEA5099D956DCE55F349110EEBF72 1.1031(k)-1(g)(6)\* & "blanket lien" 1 2 TC 60 v1 1

10\_01BCEA5099D956DCE55F349110EEBF72 1.1031(k)-1(g)(6)\* & lien or "security interest" 1 2 TC 60 v1 14

10\_01BCEA5099D956DCE55F349110EEBF72 1.1031(k)-1(g)(6)\* & pledge\* 1 2 TC 60 v1 59

10\_01BCEA5099D956DCE55F349110EEBF72 1.1031(k)-1(g)(6)\* & pledged 1 2 TC 60 v1 3

10\_082EA801EB377D2C7D7462031124E911 unreimbursed employee expense deduction 99 2 cobalt 1050 v1 1800

10\_0FC604003EFB7BFC6C028E56BB6A7580 950 1 9 cobalt 60 v1 8

10\_0FC604003EFB7BFC6C028E56BB6A7580 gift tax 1 9 cobalt 60 v1 95

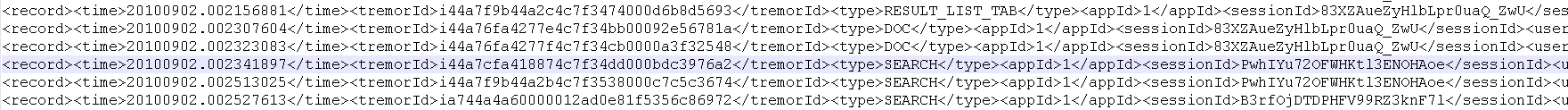
10\_111F2556FACC14BF924072D913B64B75 6013 1 2 cobalt 60 v1 2001

10\_1400EFDD444AE0A7E40549E9E87C0CC7 mortgage insurance premiums 1 2 cobalt 60 v1 2000

## Data pre-processing required (parsing, filtering)

The original log is actually in xml format being logged with other user events like documents viewed. Tab location that were clicked for each user session.

Here is a sample of the original log files.



Here is a sample user search log (substituted X for some fields – identifiable data)

from this.

<record><time>20100902.002341897</time><tremorId>XXXXXXXXXX</tremorId><type>SEARCH</type><appId>1</appId><sessionId>PwhIYu72OFWHKtl3ENOHAoe</sessionId><userId>XXXXXX</userId><userType>9</userType><searchResultId>XXXXXXXXXX</searchResultId><search><searchType>TC</searchType><keywords>e.g.-4A(b)(2)(ii)(C)</keywords><resultCount>0</resultCount><practiceArea>1</practiceArea><tabLoc>60</tabLoc><collections><collectionId>CODE</collectionId><collectionId>CODEHIST</collectionId><collectionId>FTREGS</collectionId><collectionId>ADVREGS</collectionId><collectionId>TD</collectionId><collectionId>NOTICE</collectionId><collectionId>REPREGS</collectionId><collectionId>REPPREGS</collectionId><collectionId>COMREP</collectionId></collections></search></record>

The uploaded log file in the cluster was parsed from these logs.

## Any bad data issues

TopKeywords MapReduce

The search keywords come in 2 formats: Natural language format and Terms and Connector format. I have to account for both when counting and sorting the keywords. There are also keywords that have multiple double quotes and multiple trailing and leading spaces inside or outside the multiple double quotes.

Analysis Map Reduce

Integer division is giving me an error ArithmeticException: / by zero. Changed this to a double precision.

Double precision has to me to minimum of 4 decimal places for the analysis otherwise it will not be able to sort the analysis result properly.

## MR algorithm

* All associated files and sources can be found in this link
* Wrote Both MapReduce and Spark Scala version for the analysis of the data for this project
* Hadoop MapReduce Source Code (Appendix A)- <https://github.com/petabyte/MapReduceProject>

Spark Scala - <https://github.com/petabyte/SparkScalaProject>

* Know the Top 10 search keywords for the last 5-6 years
* What do Tax professional using Checkpoint search the most during Tax Season (Mar - Apr)? Top 10 Search keywords during Tax Season (Mar - Apr)
* Comparative Analysis of Searches in the month of Mar-Apr against searches for the whole year.
  + For the comparative analysis (<https://en.wikipedia.org/wiki/Chi-squared_test>)

I decided that I would apply a Chi-Squared to quantify how much users prefer the Mar-Apr keywords over all the keywords being searched in Checkpoint

To Apply the Chi-Squared technique I will use this formula:

Preference = kT / kTotal

Where:

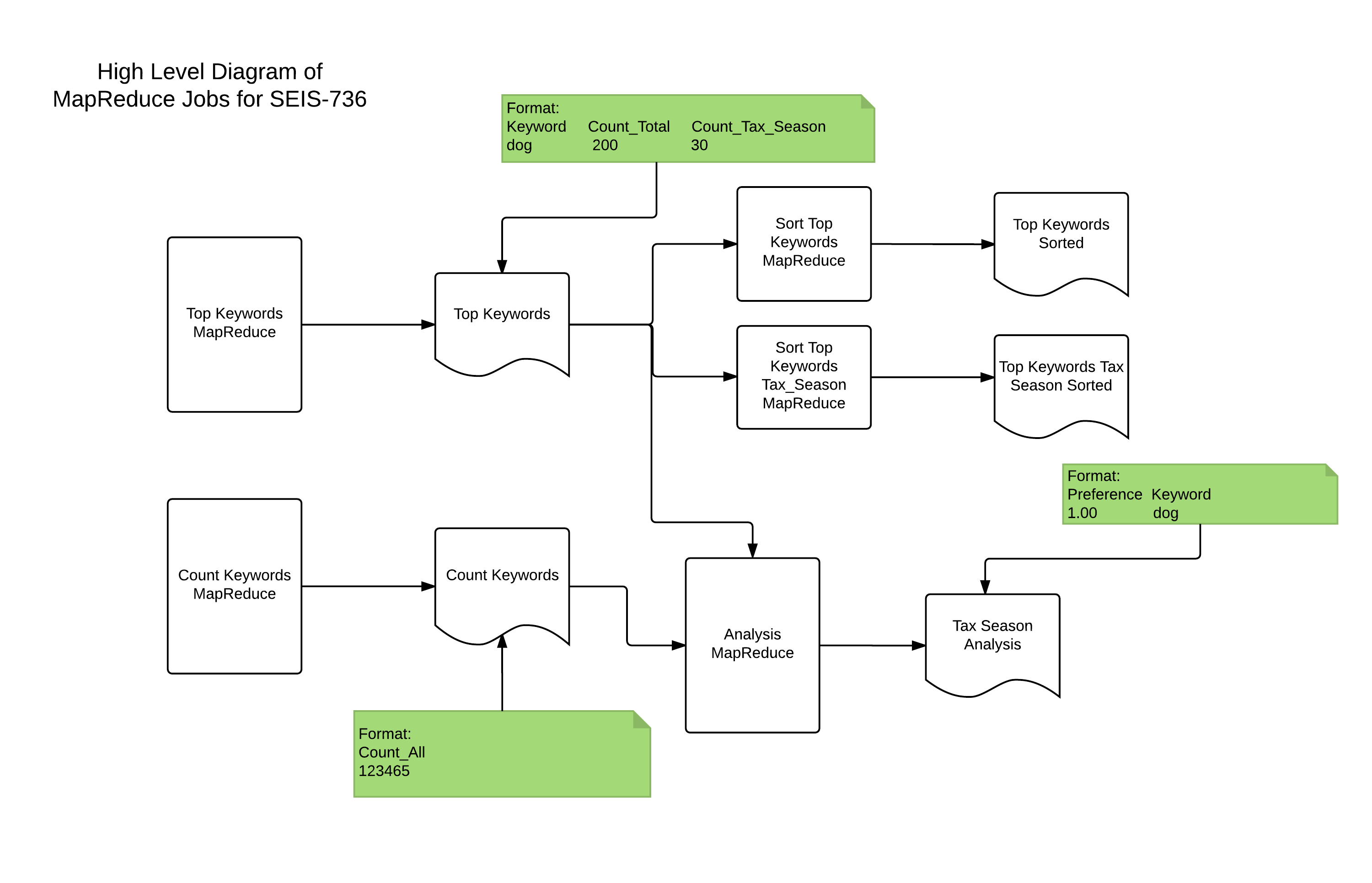
kT = (Count of Search Keyword Tax Season – Total Count for Keyword)^2 / Total Count for Keyword

This is the Chi-square value of the Keyword that we found during tax season compared to the Total Count for the Keyword

kTotal = (Total Count for Keyword – Total Count of All Keywords)^2 / Total Count of All Keywords

This is the Chi-square value of the Total Count of Keyword compared the total number of all keywords

Preference value of closest to 1.0 means that Search is preferred during Tax Season. It is the ratio of the chi squared test of the keyword during the tax season against the overall chi squared test of the keyword against the total number of keywords. This means if a keyword is being searched during tax season mostly most of its count would be the same as the total count for all time.



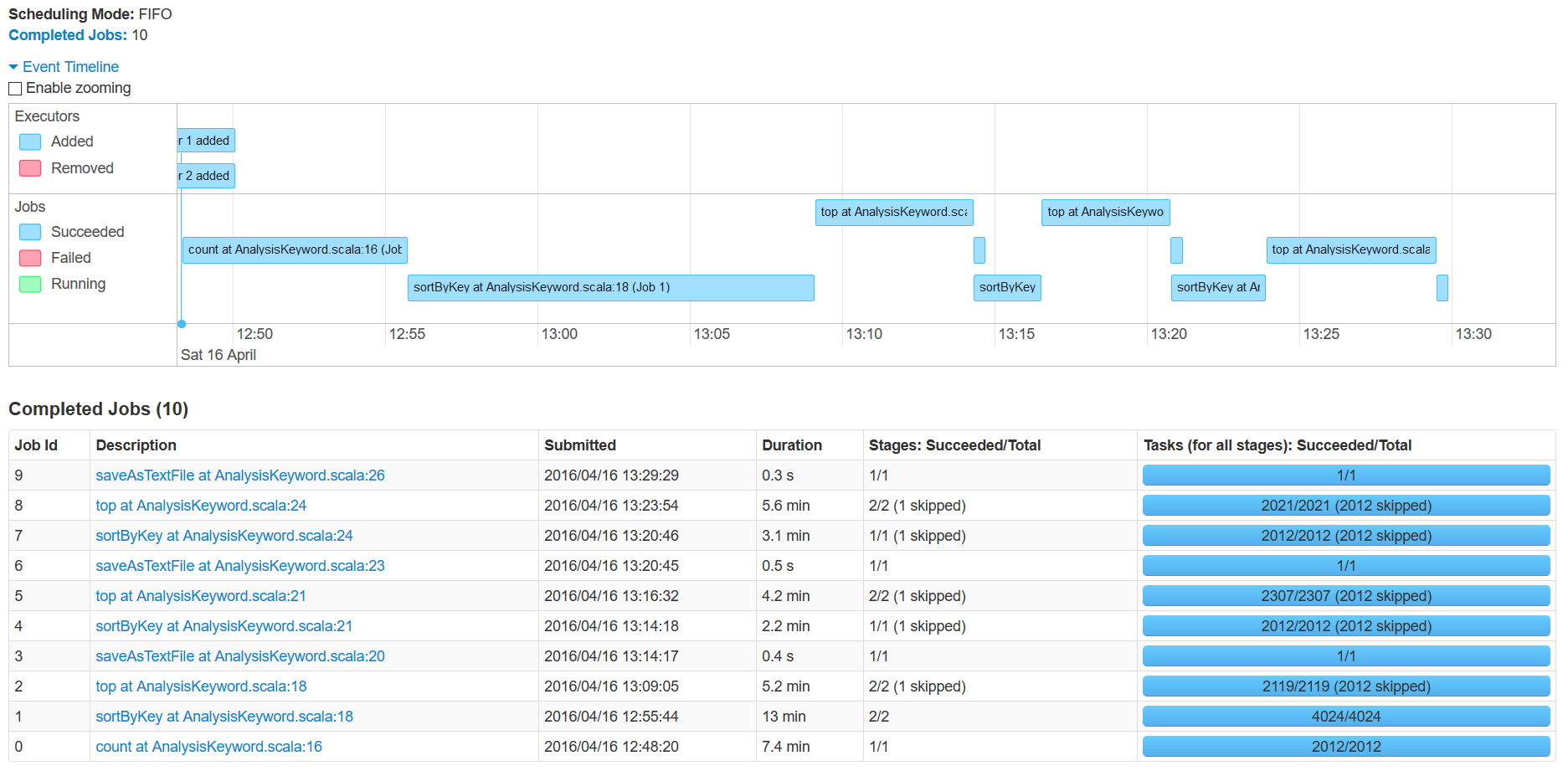
Figure B Map Reduce Schedule

Figure A Spark Scala Job Schedule

## Description of any other ecosystem or additional tools

After the MapReduce jobs I have imported the data in R to visualize (as you will see in the graph in the conclusions section). This is good for presenting the results rather than just having a tab delimited file.

Used standalone java to parse and clean the log files from the original format for me to be able to upload to the cluster.

Shell script using the hdfs to load the log files.

Used DistributedCache to read the Total Count of Keywords when doing the Comparative Analysis.

Used Spark Scala to run the same MapReduce Algorithm for analysis and comparison.

## Output description

As you can see in Figure B Map Reduce Job Dependencies. The whole process has 5 output.

1.Top Keywords – This contains the total count for each keyword for all time and counts during tax season (Mar - Apr)

2. Count Keywords – This contains the total count of all the keywords parsed. This will be used as a Distributed Cache File in the Analysis Map Reduce Job.

3.Top Keywords Sorted – This is a sorted file in descending count sort order for Keywords for all time.

4. Top Keywords Tax Season Sorted – This is a sorted file in descending count sort order for Keywords during Tax Season.

5. Tax Season Analysis – This is the analysis output where the Chi-squared test technique is applied for keywords between counts during Tax Season vs counts for All Time. The file is sorted in descending chi value order.

For the Spark Scala job there were only 3 output files that I have to use. I did not need an output file for the Count Keywords and the unsorted Top Keywords and unsorted Top Keywords during Tax Season.

## How did you verify that your output is correct?

Created a small test file with expected values where I did an initial run of all the MapReduce Job. This is under testLogFile under my user name in hdfs.

Wrote MRUnit Test for all map reduce jobs with expected input and output values.

Wrote the logic of the Java MapReduce Jobs into Spark Scala and got the same results. This is discussed in the conclusion section.

## Performance/scale characteristics

I choose to run all the map reduce jobs with 1 reducer. If this needs to be more performant I would employ having multiple reduce jobs for the Top Keywords because we it will still need to be sorted. Please see Appendix C for the Job History capture from the cluster.

1. Top Keywords MapReduce– This map reduce output the count for each keywords found in the logs and the count for the keywords during the tax season.

* ran 4 mins 26 sec
* average map time 6 sec
* average shuffle time 58 sec
* average merge time 4 sec
* average reduce time 55 sec

1. Count Keywords MapReduce - This map reduce processed all the keywords to be counted. The total number of keywords 73,512,845 words.

* ran 4 mins, 4 sec
* average map time 6 sec
* average shuffle time 1 min , 3 secs
* average merge time 7 sec
* average map time 42 sec

1. Sort Top Keywords MapReduce – This job sorts the keywords in descending order based on count.

* ran 1 mins, 3 sec
* average map time 12 sec
* average shuffle time 11 sec
* average merge time 7 sec
* average reduce time 26 sec

1. Sort Top Keywords Tax Season MapReduce – This job sorts the keywords in descending order based on count just during the Tax Season (Map - Apr)

* ran 1 mins, 3 sec
* average map time 11 sec
* average shuffle time 12 sec
* average merge time 7 sec
* average reduce time 26 sec

1. Analysis Map Reduce – This is where the chi-Value comparison logic happens and sorts the computed double value in descending order.

* ran 1 mins, 16 sec
* average map time 28 sec
* average shuffle time 3 sec
* average merge time 1 sec
* average reduce time 34 sec

The Spark Scala Job was run with these parameters – 2 executors having 512M memory each

spark-submit \

--class AnalysisKeyword \

--master yarn-cluster \

--executor-memory 512M \

--num-executors 2 \

./SparkScalaProject-1.0.jar

The spark application submitted was divided into 10 jobs and 14 stages. Tried to minimize the stages by using cache() on some of the RDDs but the whole process run ~40.7 minutes which is much slower than the Java Hadoop MapReduce version.

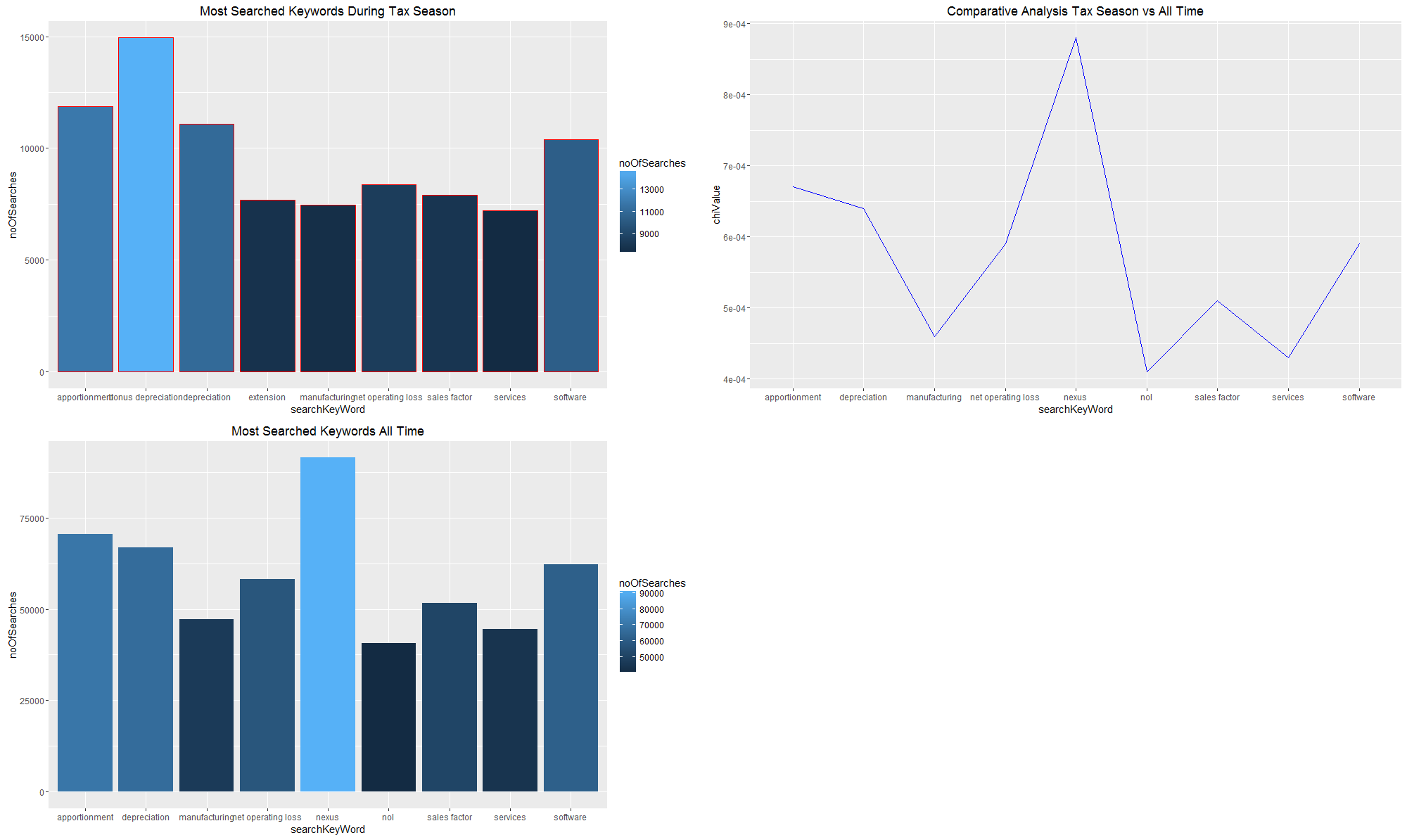
## What would you have done differently if you did this again?

Besides from starting early I would have written the job in Spark right away if I have known Spark.

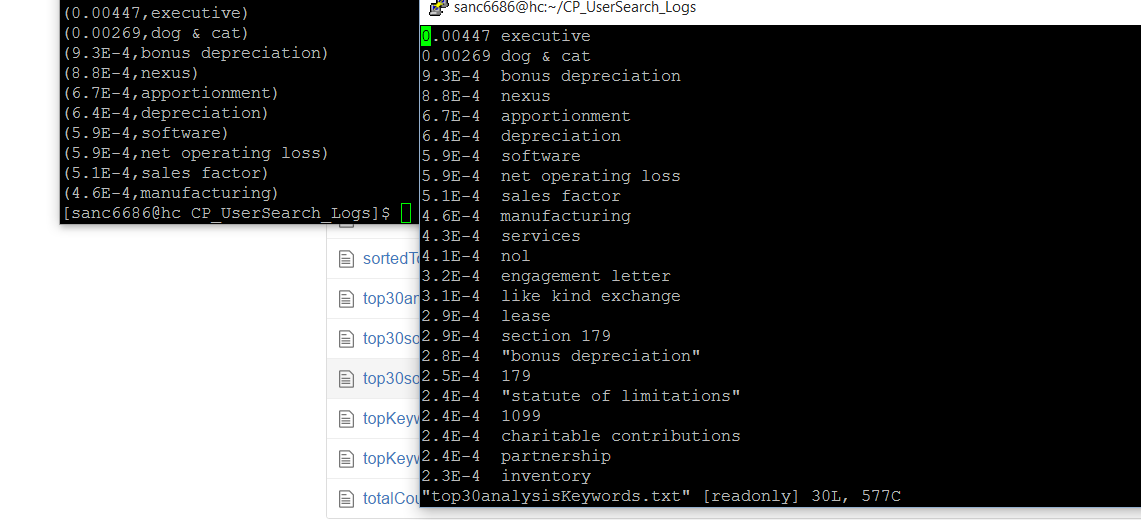
## Conclusions

1. As expected some of the most searched keywords in our system (Checkpoint) are also searched. Here are some of the difference between the Tax Season searches compared to all searches
   1. Bonus depreciation – by definition this is any additional depreciation award over what is normally available. The value of the actual amount of bonus depreciation varies year to year based on 26 USCA Section 179. As a strategy for our website maybe we should provide a quick view or highlight the link to this section during Tax Season
   2. extension – well I was expecting this because there are several different ways to file an extension based on the entity filing the tax return. There are also a lot of special rules for businesses and individuals. Maybe we should also highlight extension rules and features in our website.

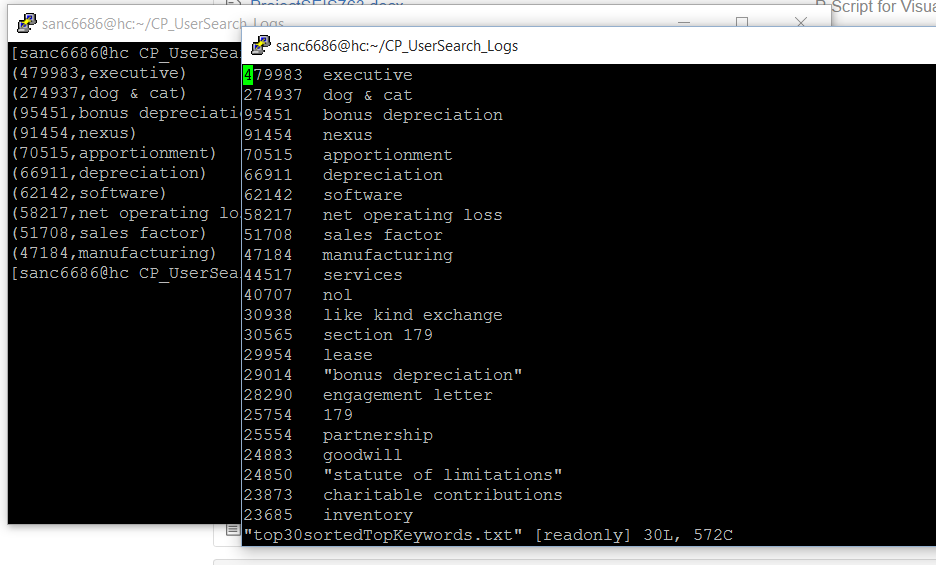
Below is a visualization of the result generated in R



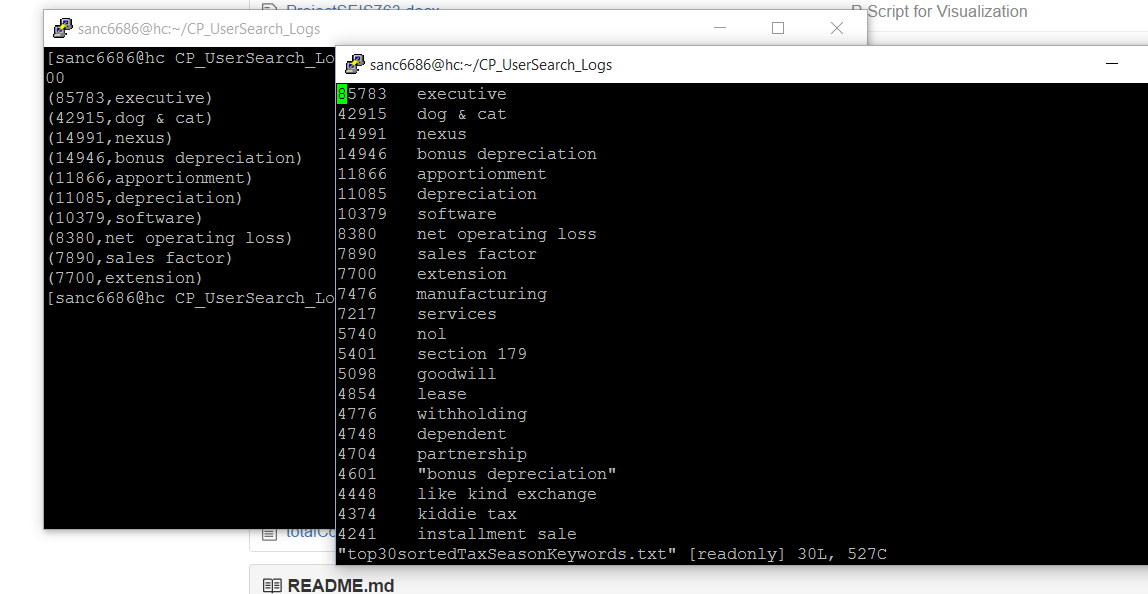
1. After learning Spark Scala in class. I wrote a Spark Scala version of the MapReduce flow for analyzing the data. The MapReduce version has 19 (this is excluding the test classes I have to write <https://github.com/petabyte/MapReduceProject>) classes which I have to write in 3 days during Spring Break. The reason for this is the framework requires you to write at least 3 classes for each MapReduce Job that you want to run. If you want to sort with a custom Comparator either you can add that to an existing MapReduce Job or create a new MapReduce Job. The Spark Scala version I only wrote 5 files (including test classes). The Spark Scala version is easier to manage and I don’t have to run succeeding jobs one by one because Spark automatically splits the transformations to jobs and stages.
2. Comparison of the Analysis Result – Spark Scala on the left – Hadoop Map Reduce on the right



1. Comparison of the Top Keywords of All Time – Spark Scala on the left - Hadoop MapReduce on the right



1. Comparison of the Top Keywords during Tax Season – Spark Scala on the left -Hadoop MapReduce on the right



## Appendix A – Java Source Code for MapReduce

## Appendix B – Spark Scala Source Code

## Appendix C – Job Output History for MapReduce