Programming for Big Data

Assignment 3

D10126532 John Warde DT230B

1. Tax Underpayment

Part A. Suitable Analyses

After importing the Revenue data into the R environment, I would use the many tools provided by R and its libraries to analyse the data based on the field/column data and graphs to look tax payers that may not have paid all their tax due and thus retrieving lost tax euros.

The scatterplot is very helpful in identifying outliers. For the majority of PAYE workers, the amount of income tax they pay is related to the amount they earn so for example plot(IncomeTax~Salary) should yield outliers but attention would have to be paid to people who also own businesses.

Part B. Data Storage

In the requirements, it suggested that the Revenue data is stored in "one giant multi-gigabyte database table" and therefore I would recommend using an SQL type library, such as the "RMySQL" library, to access the data from the R language. It would be a one step "import" process creating the appropriate SQL to pull the data in chunks. Parallelisation should also be used to achieve greater efficiency but we would need to be careful how we structure the SQL statements that we send to the database server so as not to reprocess the same rows in each of the parallelised processes i.e. use a WHERE clause to process different set of tax payer by their unique RSI identifier.

RHadoop could also be used as it also has APIs to interface directly into SQL databases but from what we have seen so far of the R language there is no indication that we can set up SQL calls when creating an RHadoop job. However, after a short research on the intert it would appear that this option is possible. This option should be benchmarks again the suggested solution above to determine the best option for Revenue.

The other out-of-memory technologies such as the "bigmemory" library are better suited to large log files. To use bigmemory you would first have to export the data from the SQL database table to a flat file, which would take some time, and then use bigmemory to load and process the file in chunks.

2. Stock Performance

In my submission I have created 3 solutions in the one R script file. The first is just using regular r script without any parallelisation. The second using parallelisation with “foreach %dopar%” and the final uses parallelisation and out-of-memory with RSQLite.

For the non-parallel solution:

- Coded the functionality into self-contained re-usable functions as much as possible.
- Chose to read in the numeric data and apply the stock names using factors – allows more flexibility if stocks are added/removed.

For parallel solution,

- In the getAveragesPerStock() function, I filtered the stockData on the current stock code within the parallelised foreach/%dopar% loop because the cluster is on the same machine. If a cluster of physical or virtual machines was available - I would consider taking the following line out of the parallelised loop

  dfForStock <- dfStock[dfStock$stock==stockNamesAsLevels[i],]

  to possibly allow the transfer of a smaller amount of data to the other machines, however I would benchmark the function before and after to get a clear picture of performance.
- Used cat(“Custom Message: ”, SomeValue, “\n") to track down problems.

For out of memory solution:

- Would consider using the SQL AVG() function to the an average on the database server side rather that bringing all the rows back to R and using the mean() function there. Again, I would benchmark duration again to make a decision on the best option.

  "SELECT AVG(gain) FROM stock_gains WHERE stock = '%s' AND day <= %d ORDER BY day"

- Used cat(“Custom Message: ”, SomeValue, “\n") to track down problems but this caused problems when running code in parallel (%dopar%)

3. R and Hadoop

Since we are looking for averages/means in the data and average/means are associative we can get averages all the chunks of the data we will read and then add up all the averages to get the total mean for the market average and the individual stocks.

```
handleRevenueData = to.dfs(RevenueData)
resultsHandle <-
    mapreduce(
        input = handleRevenueData,
        map = function(., v) {
                    stockName = parseStockName($v)
                    stockGain = parseStockGain($v)
                    keyval(stockName, stockGain)
                    },
        reduce =
              function(k, vv) {
                 total = 0
                 foreach(v in vv){
                    total += v
                 }
                 keyval(k, total / length(vv))
               })
results <- as.data.frame(from.dfs(resultsHandle))
```