Intro to HBase via R

Aaron Benz

04/28/2015

# Part I: Intro to HBase

Welcome to a brief introduction to HBase by way of R. This tutorial is aimed at explaining how you can use R through the rhbase package. However, it will use my little custom addition to rhbase which is geared towards using tidyr principals and data.tables/data.frames. Other differences include:

1. Standardized row-key serializer (raw data type)
2. sz and usz functions only apply to the actual data instead of row-key
3. hb.put - wrapper around hb.insert for easier inputting
4. hb.pull - wrapper around hb.scan for easier retrieval

So what will you get out of this? Good question. It is broken down into three parts:

1. Getting HBase, Thrift, and rhbase installed with a brief intro into HBase
2. Inserting Data into HBase + HBase Basic Design/Modelling
3. Retrieving Data from HBase, doing calculations, and inserting calculations

### So what is HBase Anyway?

Hopefully you have some bare knowledge of what HBase is, because after all, you are reading this. By no means will I attempt to explain all of HBase, but here is a brief attempt to summarize the mammoth:

**Wikipedias definition**: HBase is an open source, non-relational, distributed database modeled after Google's BigTable and written in Java. It is developed as part of Apache Software Foundation's Apache Hadoop project and runs on top of HDFS (Hadoop Distributed File system), providing BigTable-like capabilities for Hadoop.

**From** [**Apache HBase's Website**](http://hbase.apache.org/) “Use Apache HBase when you need **random, realtime read/write access** to your Big Data. This project's goal is the hosting of very large tables -- **billions of rows X millions of columns** -- atop clusters of commodity hardware. Apache HBase is an **open-source, distributed, versioned, non-relational database** modeled after Google's BigTable... Just as BigTable leverages the distributed data storage provided by the Google File System, Apache HBase provides BigTable-like capabilities **on top of Hadoop and HDFS**.”

Clearly with those definitions you should know what HBase is now, right? If you're like me, a little more explanation is helpful. To start wrapping your head around NoSQL and HBase, here is a 5-step progression that I have found helpful:

1. A **Key-Value Store**

* At its core, that's all HBase really is. It's a map, a dictionary (python), a hash (ruby), etc...

1. **Distributed**

* HBase sits atop a Hadoop Cluster. Basically what that means from HBase's perspective is that the data in HBase is replicated out to, by default, 3 nodes in a Hadoop Cluster.
* Based on the assumption that servers go down and bad things happen
* A full implementation relies on HDFS, Zookeeper, and your HBase region master(s)

1. **Consistent**

* HBase is an immediately consistent database. That means that HBase guarantees your data to be the exact same on any of the nodes that it is replicated to.
* See CAP Theorem for more information on Consistent, Available, and Partition tolerant databases (Cassandra is a similar NoSQL Columnar Database emphasizing Availability over Consistency)

1. **Sorted**

* The Keys in this key-value store are stored alphabetically
* Designed for faster reads than writes
* Allows the concepts of "scanning" to exist

1. **Multi-Dimensional Key-Value Store** <<<<<<< HEAD

# Remember that it is a key-value store? Well, really, its more like a key-column-value store, or a Map of Maps

* Remember how it is a key-value store. Well, really, its more like a key-column-value store, or a Map of Maps >>>>>>> 91bf20b9969f42dc30bf1b630c84e825e989b0aa
* This is what is referred to as "columnar" or "wide rows"
* For any given key, it allows you to store any amount of information
* Schema-less - your key-column-value combination can be defined at any time and does not naturally conform to any schema

### HBase Frame of Mind

* First there is a HBase **table**, which is exactly what you would think it is... a table <<<<<<< HEAD
* Within a table are **column families**, which are basically just a subgroup of your table. Best practice is to limit the number and size of these. So, if you are new to HBase, just specify one column family, which is all that is necessary in many cases.
* All data is then accessed via a **rowkey**, which is essentially your indexing mechanism (enter row-key or range of row-keys, and BLAM, data). This is also your key in "key-column-value" as depicted earlier
* Within a given row, there can be potentially millions of columns. This is the concept of **wide rows**. Although it can certainly be used for many things, time-series data is a good use case as it allows you to store time values as column names, and then the variable value correlating to a particular timestamp is in a cell (the variable name would be in the row key). This concept is often hard to grasp the first time, so I have provided some visuals to help explain it. Many people's breakthrough on this concept is often when they realize that values are/can be stored as columns.
* **Schemaless**. You do not need to add columns in advance, you can simply do it on the fly. However, you should keep record or develop a scheme of how you are storing data as the actual retrieval will be made very difficult if you have no idea whats in there.

# Data modeling: **Based off query patterns and stored directly**. Cross-table joins are a BAD thing (Spark can help with this, but that does not mean you should design your data model to do any kind of joins). Essentially you are sacrificing complex querying for huge speed gains.

* Tables are organized by **column families**, which is basically just another container for your data. Best practice is to limit the number and size of these. So, if you are new to HBase, just pretend you only have 1 as in many cases that is all that is necessary.
* All data is then accessed via a **rowkey**, which is essentially your indexing mechanism (enter row-key or range of row-keys, and BLAM, data). This is also your key in "key-column-value" as depicted earlier
* Within a given row, there can be potentially millions of columns. This is the concept of **wide rows**. Although it can certainly be used for many things, time-series data is a good use case as it allows you to store time values as column names, and then the variable value correlating to a particular timestamp is in a cell (the variable name would be in the row key). This concept is often hard to grasp the first time, so I have provided some visuals to help explain it. Many people's breakthrough on this concept is often when they realize that values are/can be stored as columns.
* \*\*Schemaless\*. You do not need to add columns in advance ever, you can simply do it on the fly. However, is advised that you keep record or develop a scheme of how you are storing data as the actual retrieval will be made very difficult if you have no idea whats in there.
* Data modeling: **Based off query patterns and stored directly**. Cross-table joins are a BAD thing (Spark can help with this, but that does not mean your design should implementjoins). Essentially you are sacrificing complex querying for huge speed gains. >>>>>>> 91bf20b9969f42dc30bf1b630c84e825e989b0aa

Hopefully that helped, and if not, there is plenty of information out there about HBase and what it does. Here are a few links:

* [Apache HBase](http://hbase.apache.org/)
* [Wikipedia](http://en.wikipedia.org/wiki/Apache_HBase)
* [Hortonworks](http://hortonworks.com/hadoop/hbase/)

### Installing HBase and rhbase

In order to use this stuff, you have to install HBase, Thrift, and the rhbase package. The basic instructions are found [here](https://github.com/RevolutionAnalytics/RHadoop/wiki/user-rhbase-Home), but if you are trying to get up and running as soon as possible, here are a few helpful hints:

1. Install Thrift following this [guide](http://Thrift.apache.org/docs/install/)
2. Update PKG\_CONFIG\_PATH: export PKG\_CONFIG\_PATH=$PKG\_CONFIG\_PATH:/usr/local/lib/pkgconfig/
3. Verify pkg-config path is correct: pkg-config --cflags Thrift , returns: -I/usr/local/include/Thrift
4. Copy Thrift library sudo cp /usr/local/lib/libThrift-0.8.0.so /usr/lib/

* **Under this implementation I would advise using Thrift 0.8.0 as the latest version might include some bugs as it relates to this build**

1. Install HBase following Apache's quick start [guide](http://hbase.apache.org/book.html#quickstart)
2. Start up HBase and Thrift

* [hbase-root]/bin/start-hbase.sh  
  [hbase-root]/bin/hbase Thrift start

1. Now download and install the package with devtools (or get a tarball copy [here](https://github.com/aaronbenz/rhbase/tree/master/pkg))

### Test it out

Provided HBase, Thrift, and rhbase are now installed correctly and are running, the code below should run successfully.

library(rhbase)  
library(data.table)  
hb.init()

## <pointer: 0x2387900>  
## attr(,"class")  
## [1] "hb.client.connection"

### Understanding Our Sample Data

The data that is being supplied in this tutorial is time-series data taken from airport support vehicles (like a baggage truck) from various airports over a small period of time. The data is stored hierarchically as: Airport\_Day\_VehicleID\_Variable. You can retrieve a list of all of the data simply by loading it from the rhbase package:

data(baggage\_trucks)  
str(baggage\_trucks[1:3])

## List of 3  
## $ JFK\_20140306\_1CKPH7747ZK277944\_gear :Classes 'data.table' and 'data.frame': 126 obs. of 2 variables:  
## ..$ date\_time: num [1:126] 1.39e+09 1.39e+09 1.39e+09 1.39e+09 1.39e+09 ...  
## ..$ gear : num [1:126] 1 2 3 4 3 2 1 2 3 4 ...  
## ..- attr(\*, ".internal.selfref")=<externalptr>   
## $ JFK\_20140306\_1CKPH7747ZK277944\_rpm :Classes 'data.table' and 'data.frame': 13770 obs. of 2 variables:  
## ..$ date\_time: num [1:13770] 1.39e+09 1.39e+09 1.39e+09 1.39e+09 1.39e+09 ...  
## ..$ rpm : num [1:13770] 1012 1229 1460 1758 1706 ...  
## ..- attr(\*, ".internal.selfref")=<externalptr>   
## $ JFK\_20140306\_1CKPH7747ZK277944\_speed:Classes 'data.table' and 'data.frame': 13793 obs. of 2 variables:  
## ..$ date\_time: num [1:13793] 1.39e+09 1.39e+09 1.39e+09 1.39e+09 1.39e+09 ...  
## ..$ speed : num [1:13793] 16.9 17.3 17.6 17.7 18.1 ...  
## ..- attr(\*, ".internal.selfref")=<externalptr>

The three variables are ***gear***, ***speed*** (mph), and **rpm** (revolutions per minute of engine). With these variables, the goal is to calculate the fuel rate of a vehicle. For more information about the data use ?baggage\_trucks.

Note: Credit to Spencer Herath for creating the sample data set of imaginary trucks whizzing around an invisible airport.

# Part II: Getting Data Into HBase with R

Ok, now it's time to talk about actually putting some stuffing in the elephant.

### HBase Table Design

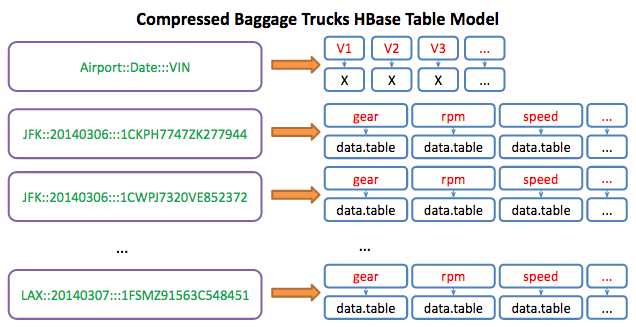
<<<<<<< HEAD It's important that the design of the HBase table suits the desired query pattern(s). A NoSQL Columnar frame of mind is always ***Design Your Tables For Your Query Pattern***. Unlike a relational store, each table that is built is traditionally designed for one single type of query pattern (document stores like Solr or Elastic Search can offer a backwards indexing solution. Overall, this can make a data modeling experience "simpler" in concept). To recap slightly, this frame of mind implies that: ======= It's important that the HBase table is designed to fit your query pattern(s) exactly. A NoSQL Columnar frame of mind is always ***Design Your Tables For Your Query Pattern***. Unlike a relational store, each table that is built is traditionally designed for one single type of query pattern (document stores like Solr or Elastic Search can offer a backwards indexing solution. Overall, this can make a data modeling experience "simpler" in concept). A good way to think about NoSQL is that **in one fetch, you want to retrieve all of the data neccessary to make a business decision**. To recap slightly, this frame of mind implies that: >>>>>>> 91bf20b9969f42dc30bf1b630c84e825e989b0aa

* Joins are BAD
* Avoid lookups as much as possible
* Do all data transformations on the incoming stream versus the outgoing

Remember, the goal is to get the data out and delivered as soon as possible, so take special precaution to insure that the data and the query patterns are designed for the end format, not the raw format.

<<<<<<< HEAD Currently the data in its raw format is **organized by variable, by vehicle, and by date.** Because this is archived data, additionally compression to a binary blob(byte array) is achievable. If properly done, it will drastically reduce the size of the data set and immensely increase the speed at which it is retrieved (because the dataset sits in just one blob as opposed to many cells). However, the approach should be cautious of the memory limitations that might exist; the size and number of blobs has to be manageable. That is, data needs to be retrieved in manageable partitions that contain all of the necessary variables to perform a fuel usage calculation. With those concepts in mind, the query pattern should be able to take on specific airports and date ranges, along with whatever variables that are desired. Thus: ======= Currently the data in its raw format is **by variable by vehicle by date.** Additionally, because this is archived data, additionally compression to a binary blob (byte array) is achievable, drastically reducing the size of the data set while immensely increasing the speed at which it is retrieved (because its just in one blob as opposed to many cells). However, the approach should be cautious of the memory limitations that might exist; the size and number of blobs has to be manageable. That is, data needs to be retrieved in manageable partitions that contain all of the necessary variables to perform a fuel usage calculation. With those concepts in mind, the query pattern should be able to take on specific airports and date ranges, along with whatever variables that are desired. Thus: >>>>>>> 91bf20b9969f42dc30bf1b630c84e825e989b0aa

* row-key = airport::day::vin
* column = variable
* value = specific data.table



##### Create HBase Table

Now that the data model is defined, a table must be created to use it. In this case, we create a table called Test with a column family called test:

## [1] TRUE

hostLoc = '127.0.0.1' #Give your server IP  
port = 9090 #Default port for Thrift service  
hb.init()

## <pointer: 0x383beb0>  
## attr(,"class")  
## [1] "hb.client.connection"

hb.list.tables()

## $Test\_Text  
## maxversions compression inmemory bloomfiltertype bloomfiltervecsize  
## test: 3 NONE FALSE NONE 0  
## bloomfilternbhashes blockcache timetolive  
## test: 0 FALSE 2147483647

TABLE\_NAME = "Test"  
COLUMN\_FAMILY = "test"  
hb.new.table(TABLE\_NAME, COLUMN\_FAMILY)

## [1] TRUE

### Input Data into HBase

Now that the HBase table is created, all that is left is actually putting the baggage\_trucks data into HBase. This can be done with the convenient hb.put function:

require(magrittr,quietly = T)  
require(tidyr,quietly = T,warn.conflicts = F)  
data(baggage\_trucks)  
dt\_names <- names(baggage\_trucks) %>%  
 data.frame() %>%  
 tidyr::separate(".",c("airport","date","vin","variable"))  
dt\_names <- dt\_names %>% unite(rowkey,airport,date,vin, sep = "::")  
head(dt\_names)

## rowkey variable  
## 1 JFK::20140306::1CKPH7747ZK277944 gear  
## 2 JFK::20140306::1CKPH7747ZK277944 rpm  
## 3 JFK::20140306::1CKPH7747ZK277944 speed  
## 4 JFK::20140306::1CWPJ7320VE852372 gear  
## 5 JFK::20140306::1CWPJ7320VE852372 rpm  
## 6 JFK::20140306::1CWPJ7320VE852372 speed

#### How hb.put Works

The design of hb.put is meant to be relatively simple and flexible. For a given table and column family, hb.put allows the creation of a list of "columns" and a list of "values" for each "row-key." This option is very useful and designed for inputting multiple columns into the same row-key (an uncompressed time-series use case). Additionally, hb.put allows the insertion of data using a 1-1-1 ratio like this example will do.

hb.put(table\_name = TABLE\_NAME,column\_family = COLUMN\_FAMILY, rowkey = dt\_names$rowkey, column = dt\_names$variable,value = baggage\_trucks)

## [1] TRUE

And just like that, data is in HBase. Before proceeding further, it is important to understand how data was put into HBase, as this is actually a modification from the original rhbase package. The row-keys are turned into byte arrays using the charToRaw method, essentially turning the character row-key into a raw binary data type. The data.tables are turned into byte arrays using R's native serializer. If you would like to use your own serializer for the actual data, input the data as raw (maybe because its already serialized), etc..., simply specify sz = "raw", "character", or custom function in hb.put, OR specify it in the original hb.init function. Note: the row-key serializer is not editable at the moment. The change to this branch was separating the serialization method for the row-keys from the values.

#### Examples Retrieving Data

Now that data is inserted, here are some brief examples worth reviewing to understand how data can be retrieved from HBase with this package.

1. Retrieving only from 03/06/2014 onward for LAX and for just the "Speed" variable:

hb.pull(TABLE\_NAME, COLUMN\_FAMILY, start = "LAX::20140306", end = "LAXa", columns = "speed", batchsize = 100)

## rowkey column\_family column values  
## 1: LAX::20140306::1CHMP60486H283129 test speed <data.table>  
## 2: LAX::20140306::1FAJL35763D392894 test speed <data.table>  
## 3: LAX::20140306::1FJAL1998VS238734 test speed <data.table>  
## 4: LAX::20140306::1FSMZ91563C548451 test speed <data.table>  
## 5: LAX::20140307::1CHMP60486H283129 test speed <data.table>  
## 6: LAX::20140307::1FAJL35763D392894 test speed <data.table>  
## 7: LAX::20140307::1FJAL1998VS238734 test speed <data.table>  
## 8: LAX::20140307::1FSMZ91563C548451 test speed <data.table>

1. Retrieving everything between the start of 03/07/2014 and the start of 03/08/2014:

hb.pull(TABLE\_NAME, COLUMN\_FAMILY, start = "LAX::20140307", end = "LAX::20140308",batchsize = 100)

## rowkey column\_family column values  
## 1: LAX::20140307::1CHMP60486H283129 test gear <data.table>  
## 2: LAX::20140307::1CHMP60486H283129 test rpm <data.table>  
## 3: LAX::20140307::1CHMP60486H283129 test speed <data.table>  
## 4: LAX::20140307::1FAJL35763D392894 test gear <data.table>  
## 5: LAX::20140307::1FAJL35763D392894 test rpm <data.table>  
## 6: LAX::20140307::1FAJL35763D392894 test speed <data.table>  
## 7: LAX::20140307::1FJAL1998VS238734 test gear <data.table>  
## 8: LAX::20140307::1FJAL1998VS238734 test rpm <data.table>  
## 9: LAX::20140307::1FJAL1998VS238734 test speed <data.table>  
## 10: LAX::20140307::1FSMZ91563C548451 test gear <data.table>  
## 11: LAX::20140307::1FSMZ91563C548451 test rpm <data.table>  
## 12: LAX::20140307::1FSMZ91563C548451 test speed <data.table>

# Part III: Retrieve and Store

Ok, so now that data is in HBase, lets:

1. **Retrieve Data with rhbase**
2. **Manipulate Data using Tidyr + data.table + timeseriesr**
3. **Perform Calculations with timeseriesr**
4. **Store Our Results using rhbase**

## Retrieving Data with rhbase

From the HBase input tutorial, we stored data.tables via byte arrays in HBase (from hbase\_input document). But what about getting it out? By using hb.pull, we will be able to pull our desired HBase data.

Going back to the use case at hand, our goal is to measure the total fuel consumption of these trucks present in the data. To do so, we need to call gear, rpm, and speed (the three variables we put in HBase) and apply a custom fuel calculator in R. However, as mentioned previously, we need to be careful about how we bring data in, as too much data at one time could easily lead to memory problems.

Based on how we modeled our data, we are going to **retrieve ALL variables for ALL VINs by EACH day for EACH airport**. We can do this in HBase with the scan function. Recalling the description of HBase, all of the keys or row-keys are sorted alphabetically. This means that all of the data for 1 airport is stored next to each other. Additionally, because of our row-key, all of the data is sorted by Airport by Date. A scan operation will allow us to get back all of the data for one airport for one day in essentially one iop. That is, with one "scan" we can get all of the data because its located next to each other. So how can we do this in R?

We begin with a list of our airports:

airports <- c("JFK","LAX")

Then we create a list of all the dates we want, which in this case is between the 6th and 8th of March 2014.

start\_dates <- paste0("2014030", 6:7)  
end\_dates <- paste0("2014030", 7:8)

Now lets create a function that will allow us to pull back all of the variables one VIN one Day at a time (this is done to demonstrate responsible memory management.)

library(data.table)  
rk\_generator <- function(start, end, ...){  
 function(x){  
 data.table(start = paste(x,start, ...), end = paste(x,end,...))  
 }  
}  
  
march <- rk\_generator(start\_dates, end\_dates, sep = "::")

The functional march allows us to have all of the time stamps we want for each airport. For example:

march(airports[1])

## start end  
## 1: JFK::20140306 JFK::20140307  
## 2: JFK::20140307 JFK::20140308

Don't you just love how easy R is? This output will feed our function to call HBase for each day for each airport.

## Pull Data, Merge, and Calculate

1. Bring in some data for 1 day
2. Merge the data together to do a proper fuel calculation of gal\_per\_hr, average speed, and time in use
3. Visualize some of the results:

a\_day <- march(airports[1])[1]

OK, time to bring in some data:

library(rhbase)  
library(magrittr)  
hb.init()

## <pointer: 0x3736230>  
## attr(,"class")  
## [1] "hb.client.connection"

data <- hb.pull("Test","test",start = a\_day[[1]], end = a\_day[[2]], columns = c("gear","rpm","speed"))  
data[1:6]

## rowkey column\_family column values  
## 1: JFK::20140306::1CKPH7747ZK277944 test gear <data.table>  
## 2: JFK::20140306::1CKPH7747ZK277944 test rpm <data.table>  
## 3: JFK::20140306::1CKPH7747ZK277944 test speed <data.table>  
## 4: JFK::20140306::1CWPJ7320VE852372 test gear <data.table>  
## 5: JFK::20140306::1CWPJ7320VE852372 test rpm <data.table>  
## 6: JFK::20140306::1CWPJ7320VE852372 test speed <data.table>

And just like that, boom, data!!!!!

## Manipulate Data with Tidyr + data.table + timeseriesr

OK, lets do something with that stuff. Our goal is to combine the gear, rpm, and speed data.tables by VIN. To do this, we will: 1. Split the row key to make the values meaningful with tidyr 2. Combine data.tables with VIN in mind 3. Clean up merged data

1. Split with tidyr:

data <- data %>%  
 tidyr::separate(col = "rowkey", into = c("airport","day","vin"))  
data[1:5]

## airport day vin column\_family column values  
## 1: JFK 20140306 1CKPH7747ZK277944 test gear <data.table>  
## 2: JFK 20140306 1CKPH7747ZK277944 test rpm <data.table>  
## 3: JFK 20140306 1CKPH7747ZK277944 test speed <data.table>  
## 4: JFK 20140306 1CWPJ7320VE852372 test gear <data.table>  
## 5: JFK 20140306 1CWPJ7320VE852372 test rpm <data.table>

1. Combine by VIN + column with rbindlist

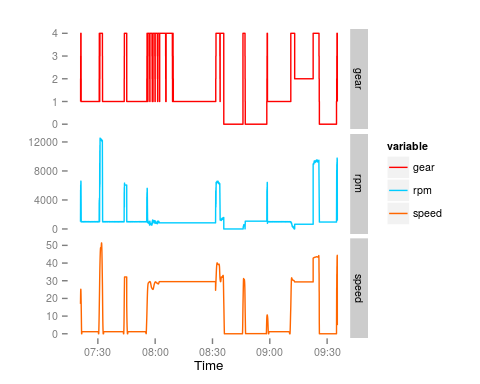
#rbind by variable  
setkeyv(data, c("vin", "column"))  
  
merge\_em <- function(values){  
 if(length(values)<=1) return(values)  
 out <- values[[1]]  
 for(i in 2:length(values)){  
 out <- merge(out,values[[i]],all=T,by = "date\_time")  
 }  
 out  
}  
data2 <- data[,list("rbinded" = list(merge\_em(values))),by=c("vin","day","airport")] #data.table functionality  
data2$rbinded[[1]] %>% setDT

1. Clean up our data with timeseriesr. Essentially, because our timestamps for each variable were not guaranteed to match, we probably (and do) have NA values in each data set. This use of dtreplace will take any NA values and replace them with the last observation.

data2$rbinded <- data2$rbinded %>% lapply(timeseriesr::dtreplace) %>% lapply(setDT)#fills in missing NAs  
data2$rbinded[[1]]

## date\_time gear rpm speed  
## 1: 1394112055 1 1012.492 16.937217  
## 2: 1394112055 1 1228.810 17.264578  
## 3: 1394112055 1 1459.631 17.585472  
## 4: 1394112056 1 1757.546 17.743490  
## 5: 1394112056 1 1705.891 18.054664  
## ---   
## 14081: 1394120120 1 3762.915 19.516140  
## 14082: 1394120120 1 2902.979 14.085581  
## 14083: 1394120120 1 2602.196 11.170207  
## 14084: 1394120121 1 1898.852 4.939276  
## 14085: 1394120121 1 1150.547 4.939276

1. Now lets see what this baby looks like:



#### Perform our calculations with timeseriesr

3.Now that we have our data in memory, lets do our calculation! Below is a function to calculate gal\_per\_hr using our three variables (rpm, gear, and speed)

#I basically estimated this by looking at some relations between acceleration, vehicle weight, torque, and all of the other things you see below on the magical world wide web  
   
#calculates Fuel Usage by row, returns array of fuel usage  
#engine power = torque \* rpm / 5252  
# The constant 5252 is the rounded value of (33,000 ft?lbf/min)/(2?? rad/rev).  
#fuel usage: if Power < 0, alpha  
# : alpha + min(Pmax, engine power)  
# alpha = constant idle fuel consumption rate (estimate of fuel sued to maintain engine operation)  
# Pt = power = min(Pmax, torque + inertia(mass \* radius^2))  
gal\_per\_hr = function(engine\_gear, engine\_rpm, time, speed, alpha = .7, mass = 10000, gear\_ratio = 1.2, Pmax = 200, efficiency\_parameter = .02, acceleration\_parameter = .01, wheel\_radius){  
 torque <- c(diff(engine\_rpm),0)  
 torque[torque<0] <- 0  
 torque <- torque \* engine\_gear \* gear\_ratio  
 Pt <- torque \* engine\_rpm / (33000/2/pi)  
 Pt[Pt>200] <- 200  
 engine\_power <- t  
 acceleration <- c(-diff(speed),0) / c(-diff(time),1)   
 #Pi = Mv(kg) \* acceleration \* velocity /1000  
 fuel <- alpha + efficiency\_parameter \* Pt + acceleration\_parameter \* acceleration \* mass\*.45359/1000 \* speed  
 fuel[fuel < alpha] <- alpha  
 fuel[is.nan(fuel)] <- alpha  
   
 return(fuel)  
}

Actually perform that operation

data2$rbinded <- lapply(data2$rbinded,function(i){  
 i$gal\_per\_sec <- gal\_per\_hr(engine\_gear = i$gear, engine\_rpm = i$rpm, speed = i$speed, time = i$date\_time)/60/60 #currently in hours, but want it to be in seconds as the time column is in seconds  
 i %>% setDT  
 })  
data2$rbinded[[1]]

## date\_time gear rpm speed gal\_per\_sec  
## 1: 1394112055 1 1012.492 16.937217 0.0008217549  
## 2: 1394112055 1 1228.810 17.264578 0.0009034893  
## 3: 1394112055 1 1459.631 17.585472 0.0010965335  
## 4: 1394112056 1 1757.546 17.743490 0.0005422796  
## 5: 1394112056 1 1705.891 18.054664 0.0014310010  
## ---   
## 14081: 1394120120 1 3762.915 19.516140 0.0001944444  
## 14082: 1394120120 1 2902.979 14.085581 0.0001944444  
## 14083: 1394120120 1 2602.196 11.170207 0.0001944444  
## 14084: 1394120121 1 1898.852 4.939276 0.0001944444  
## 14085: 1394120121 1 1150.547 4.939276 0.0001944444

Now we have done our main calculation. But we want to know the total amount of gallons each truck burned per day. We are going to use calc\_area, which is a timeseriesr function that calculates the area under the curve. It uses a Riemann left sum approach, but other methods may be added in the future.

#calculates area under the curve as a total  
data2$gallons <- data2$rbinded %>%  
 sapply(function(x) timeseriesr::calc\_area(time\_date = x$date\_time, value = x$gal\_per\_sec))  
#calculates the total time range in hours (data is recorded in seconds)  
data2$hours <- data2$rbinded %>%  
 sapply(function(x) (max(x$date\_time) - min(x$date\_time))/60/60)  
#calculates the average speed  
data2$mph <- data2$rbinded %>%  
 sapply(function(x) mean(x$speed))  
data2[,gal\_per\_hr := gallons/hours]

## vin day airport rbinded gallons hours  
## 1: 1CKPH7747ZK277944 20140306 JFK <data.table> 3.1177370 2.240531  
## 2: 1CWPJ7320VE852372 20140306 JFK <data.table> 0.9673207 1.033856  
## 3: 1FEMY6958XU502984 20140306 JFK <data.table> 1.5951636 1.757431  
## 4: 1FVTK43691G456340 20140306 JFK <data.table> 3.6247181 3.852247  
## 5: 1TULD4346YD544661 20140306 JFK <data.table> 1.6412674 1.803878  
## mph gal\_per\_hr  
## 1: 9.026001 1.3915173  
## 2: 10.746364 0.9356439  
## 3: 14.793853 0.9076681  
## 4: 13.923766 0.9409360  
## 5: 9.326923 0.9098551

\*\*We have now calculated the total amount of gallons that each truck burned, the total hours it ran, miles per hour, and its average gallons per hour.\* Now, lets put all of this back into HBase to move onto the day/airport.

## Store Our Results Using rhbase

We are going to store all of the information that we collected back into the same HBase table for later use. That includes: 1. rbinded data.table (because we might want to reuse it later) 2. gallons 3. total hours in operation 4. gal\_per\_hr 5. average mph

To do this we need to slightly reorganize our table (hopefully in the future I/someone will add a small wrapper to take care of this) to fit the hb.put standards. Nonetheless, the tidyr package allows us to do this with ease.

#First, create rowkey with unite  
data3 <- tidyr::unite(data2, "rowkey",airport:vin,sep = "::")  
  
#Second, reoganize data with gather  
data3 <- tidyr::gather(data3, column, value, -rowkey,convert = T)

## Warning in melt.data.table(data, measure.vars = gather\_cols, variable.name  
## = key\_col, : All 'measure.vars are NOT of the SAME type. By order of  
## hierarchy, the molten data value column will be of type 'list'. Therefore  
## all measure variables that are not of type 'list' will be coerced to.  
## Check the DETAILS section of ?melt.data.table for more on coercion.

data3[c(1,6,11,16,21)]

## rowkey column value  
## 1: JFK::20140306::1CKPH7747ZK277944 rbinded <data.table>  
## 2: JFK::20140306::1CKPH7747ZK277944 gallons 3.117737  
## 3: JFK::20140306::1CKPH7747ZK277944 hours 2.240531  
## 4: JFK::20140306::1CKPH7747ZK277944 mph 9.026001  
## 5: JFK::20140306::1CKPH7747ZK277944 gal\_per\_hr 1.391517

Great! Now that we have it in the format we want, lets put it back in HBase

#Assuming the hb.init connection is still valid  
hb.put("Test","test",rowkey = data3$rowkey, column = data3$column, value = data3$value)

## [1] TRUE

And just to test it out, lets see what happens when we pull back one of the new columns we added:

hb.pull("Test","test",c("gal\_per\_hr","rbinded"))

## rowkey column\_family column values  
## 1: JFK::20140306::1CKPH7747ZK277944 test gal\_per\_hr 1.391517  
## 2: JFK::20140306::1CKPH7747ZK277944 test rbinded <data.table>  
## 3: JFK::20140306::1CWPJ7320VE852372 test gal\_per\_hr 0.9356439  
## 4: JFK::20140306::1CWPJ7320VE852372 test rbinded <data.table>  
## 5: JFK::20140306::1FEMY6958XU502984 test gal\_per\_hr 0.9076681  
## 6: JFK::20140306::1FEMY6958XU502984 test rbinded <data.table>  
## 7: JFK::20140306::1FVTK43691G456340 test gal\_per\_hr 0.940936  
## 8: JFK::20140306::1FVTK43691G456340 test rbinded <data.table>  
## 9: JFK::20140306::1TULD4346YD544661 test gal\_per\_hr 0.9098551  
## 10: JFK::20140306::1TULD4346YD544661 test rbinded <data.table>

**Pretty cool right?**

What is great about this is that you can easily parallelize this operation across multiple cores/nodes/clusters because we broke it down into intervals (airport and date). That is, our data model and query pattern were designed for easily doing a fork and join operation. Check out the parallel package or even the rmr2 package for more details of how to do that.

If you have any suggestions, comments, or questions please feel free to contact me. Also if you would like to further contribute to the rhbase fork or timeseriesr package, I welcome all aboard.