Wrangling ***Big Data*** is one of the best features of the R programming language, which boasts a ***Big Data Ecosystem*** that contains fast in-memory tools (e.g. data.table) and distributed computational tools (sparklyr). With the NEW dtplyr package, data scientists with dplyr experience *gain* the benefits of data.table backend. ***We saw a 3X speed boost for dplyr!***

**Table of Contents**

1. The 30-Second Summary
2. Big Data Ecosystem
3. Enter dtplyr: Boost dplyr with data.table backend
4. Case Study – Wrangling 4.6M Rows of Financial Data
5. The 3X Speedup – Time Comparisons
6. Conclusions and Learning Recommendations
7. Additional Big Data Guidelines and Packages
8. Recognizing the Developers

**1.0 The 30-Second Summary**

We reviewed the ***latest advance in big data*** – The NEW dtplyr package, which is an interface to the high performance data.table library.

**Pros**

* A ***3X speed boost*** on the data joining and wrangling operations on a 4.6M ROw data set. The data wrangling operatiosn were performed in 6 seconds with dtplyr vs 18 seconds with dplyr.
* Performs ***inplace operations*** (:=), which vastly accelerates big data computations
* Shows the data.table translation (this is really cool!)

**Cons**

* **For pure speed**, you will need to learn all of data.table’s features including managing keys for fast lookups.
* In most cases, data.table **will be faster** than dtplyr because of overhead in the dtplyr translation process. However, we saw the difference to be very minimal.
* dtplyr is in ***experimental status*** currently –

**What Should You Learn?**

***Just starting out?*** Our recommendation is to learn dplyr first, then learn data.table, using dtplyr to bridge the gap

* Begin with dplyr, which has ***easy-to-learn syntax*** and works well for datasets of 1M Rows+.
* Learn data.table as you become comfortable in R. data.table is ***great for pure speed*** on data sets 50M Rows+. It has a different “bracketed” syntax that is streamlined but more complex for beginners. However, it has features like fast keyed subsetting and optimization for rolling joins that are out of the scope of this article.
* Use dtplyr as a translation tool to help ***bridge the gap*** between dplyr and data.table.

**2.0 Big Data Ecosystem**

R has an amazing ecosystem of tools designed for ***wrangling Big Data***. The 3 most popular tools are dplyr, data.table, and sparklyr. We’ve trained hundreds of students on big data, and our students ***most common Big Data question is, “Which tool to use and when?”***



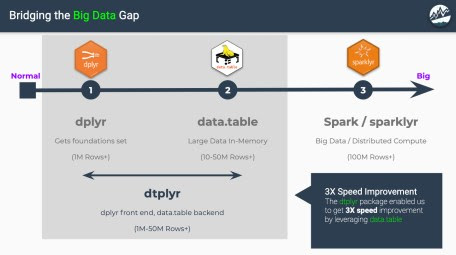
Big Data: Data Wrangling Tools By Dataset Size

The graphic provides *rough guidelines* on when to use which tools by dataset row size.

1. [**dplyr (website)**](https://dplyr.tidyverse.org/) – Used for in-memory calculations. Syntax design and execution emphasizes readability over performance. Very good in most situations.
2. [**data.table (website)**](https://github.com/Rdatatable/data.table/wiki) – Used for higher in-memory performance. Modifies data inplace for huge speed gains. Easily wrangles data in the range of 10M-50M+ rows.
3. [**sparklyr (website)**](https://spark.rstudio.com/) – Distribute work across nodes (clusters) and performs work in parallel. Best used on big data (100M+ Rows).

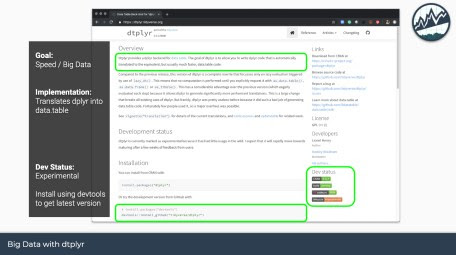
**3.0 Enter dtplyr: Boost dplyr with data.table backend**

We now have a 4th tool that boosts dplyr using data.table as its backend. The good news is that if you are already familiar with dplyr, you don’t need to learn much to get the gains of data.table!



dtplyr: Bridging the Big Data Gap

The [dtplyr package](https://dtplyr.tidyverse.org/) is a new front-end that wraps the High Performance data.table R package. I say new, but dtplyr has actually been around for over 2 years. However, the *implementation recently underwent a complete overhaul* vastly improving the functionality. Let’s check out the goals the package from the dtplyr website: <https://dtplyr.tidyverse.org/>.



dtplyr for Big Data

Here’s what you need to know:

* **Goal**: Increase speed of working with big data when using dplyr syntax
* **Implementation**: The dtplyr package enables the user to write dplyr code. Internally the package translates the code to data.table syntax. When run, the user gains the faster performance of data.table while being able to write the more readable dplyr code.
* **Dev Status**: The package is still ***experimental***. This means that developers are still in the process of testing the package out, reporting bugs, and improving via feature requests.

**4.0 Case Study – Wrangling 4.6M Rows (375MB) of Financial Data**

Let’s try out the new and improved dtplyr + data.table combination on a large-ish data set.

**4.1 Bad Loans Cost Millions (and Data Sets are MASSIVE)**



Loan defaults cost organization millions. Further, the datasets are massive. This is a task where data.table and dtplyr will be needed as part of the ***preprocessing steps*** prior to building a Machine Learning Model.

**4.2 Fannie Mae Data Set**

The data used in the tutorial can be downloaded from [Fannie Mae’s website](https://loanperformancedata.fanniemae.com/lppub/index.html). We will just be using the ***2018 Q1 Acquisition and Performance data set***.



A few quick points:

* The *2018 Q1 Performance Data Set* we will use is 4.6M rows, enough to send Excel to a grinding hault, crashing your computer in the process.
* For dplyr, it’s actually do-able at 4.6M rows. However, if we were to do the full 25GB, we’d definitely want to use data.table to speed things up.
* We’ll do a series of common data manipulation operations including ***joins and grouped time series calculation*** to determine which loans become delinquent in the next 3 months.

**4.3 Install and Load Libraries**

In this tutorial, we’ll use the latest Development Version of dtplyr installed using devtools. All other packages used can be used by installing with install.packages().

devtools::install\_github("tidyverse/dtplyr")

Next, we’ll load the the following libraries with library():

* data.table: High-performance data wrangling
* dtplyr: Interface between dplyr and data.table
* tidyverse: Loads dplyr and several other useful R packages
* vroom: Fast reading of delimited files (e.g. csv) with vroom()
* tictoc: Simple timing operations
* knitr: Use the kable() function for nice HTML tables

# Load data.table & dtplyr interface

library(data.table)

library(dtplyr)

# Core Tidyverse - Loads dplyr

library(tidyverse)

# Fast reading of delimited files (e.g. csv)

library(vroom) # vroom()

# Timing operations

library(tictoc)

# Table Printing

library(knitr) # kable()

**4.4 Read the Data**

We’ll read the data. The column-types are going to be pre-specified to assist in the loading process. The vroom() function does the heavy lifting.

First, I’ll setup the paths to the two files I’ll be reading:

1. **Acquisitions\_2018Q1.txt** – Meta-data about each loan
2. **Performance\_2018Q1.txt** – Time series data set with loan performance characteristics over time

For me, the files are stored in a folder called 2019-08-15-dtplyr. Your paths may be different depending on where the files are stored.

# Paths to files

path\_acq <- "2019-08-15-dtplyr/Acquisition\_2018Q1.txt"

path\_perf <- "2019-08-15-dtplyr/Performance\_2018Q1.txt"

**Read the Loan Acquisition Data**

Note we specify the columns and types to improve the speed of reading the columns.

# Loan Acquisitions Data

col\_types\_acq <- list(

loan\_id = col\_factor(),

original\_channel = col\_factor(NULL),

seller\_name = col\_factor(NULL),

original\_interest\_rate = col\_double(),

original\_upb = col\_integer(),

original\_loan\_term = col\_integer(),

original\_date = col\_date("%m/%Y"),

first\_pay\_date = col\_date("%m/%Y"),

original\_ltv = col\_double(),

original\_cltv = col\_double(),

number\_of\_borrowers = col\_double(),

original\_dti = col\_double(),

original\_borrower\_credit\_score = col\_double(),

first\_time\_home\_buyer = col\_factor(NULL),

loan\_purpose = col\_factor(NULL),

property\_type = col\_factor(NULL),

number\_of\_units = col\_integer(),

occupancy\_status = col\_factor(NULL),

property\_state = col\_factor(NULL),

zip = col\_integer(),

primary\_mortgage\_insurance\_percent = col\_double(),

product\_type = col\_factor(NULL),

original\_coborrower\_credit\_score = col\_double(),

mortgage\_insurance\_type = col\_double(),

relocation\_mortgage\_indicator = col\_factor(NULL))

acquisition\_data <- vroom(

file = path\_acq,

delim = "|",

col\_names = names(col\_types\_acq),

col\_types = col\_types\_acq,

na = c("", "NA", "NULL"))

The loan acquisition data contains information about the owner of the loan.

acquisition\_data %>% head() %>% kable()

| **loan\_id** | **original\_channel** | **seller\_name** | **original\_interest\_rate** | **original\_upb** | **original\_loan\_term** | **original\_date** | **first\_pay\_date** | **original\_ltv** | **original\_cltv** | **number\_of\_borrowers** | **original\_dti** | **original\_borrower\_credit\_score** | **first\_time\_home\_buyer** | **loan\_purpose** | **property\_type** | **number\_of\_units** | **occupancy\_status** | **property\_state** | **zip** | **primary\_mortgage\_insurance\_percent** | **product\_type** | **original\_coborrower\_credit\_score** | **mortgage\_insurance\_type** | **relocation\_mortgage\_indicator** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 100001040173 | R | QUICKEN LOANS INC. | 4.250 | 453000 | 360 | 2018-01-01 | 2018-03-01 | 65 | 65 | 1 | 28 | 791 | N | C | PU | 1 | P | OH | 430 | NA | FRM | NA | NA | N |
| 100002370993 | C | WELLS FARGO BANK, N.A. | 4.250 | 266000 | 360 | 2018-01-01 | 2018-03-01 | 80 | 80 | 2 | 41 | 736 | N | R | PU | 1 | P | IN | 467 | NA | FRM | 793 | NA | N |
| 100005405807 | R | PMTT4 | 3.990 | 233000 | 360 | 2017-12-01 | 2018-01-01 | 79 | 79 | 2 | 48 | 696 | N | R | SF | 1 | P | CA | 936 | NA | FRM | 665 | NA | N |
| 100008071646 | R | OTHER | 4.250 | 184000 | 360 | 2018-01-01 | 2018-03-01 | 80 | 80 | 1 | 48 | 767 | Y | P | PU | 1 | P | FL | 336 | NA | FRM | NA | NA | N |
| 100010739040 | R | OTHER | 4.250 | 242000 | 360 | 2018-02-01 | 2018-04-01 | 49 | 49 | 1 | 22 | 727 | N | R | SF | 1 | P | CA | 906 | NA | FRM | NA | NA | N |
| 100012691523 | R | OTHER | 5.375 | 180000 | 360 | 2018-01-01 | 2018-03-01 | 80 | 80 | 1 | 14 | 690 | N | C | PU | 1 | P | OK | 730 | NA | FRM | NA | NA | N |

Get the size of the acquisitions data set: 426K rows by 25 columns. Not that bad, but this is meta-data for the loan. ***The dataset we are worried about is the next one.***

dim(acquisition\_data)

## [1] 426207 25

**Read the Loan Performance Data**

# Loan Performance Data

col\_types\_perf = list(

loan\_id = col\_factor(),

monthly\_reporting\_period = col\_date("%m/%d/%Y"),

servicer\_name = col\_factor(NULL),

current\_interest\_rate = col\_double(),

current\_upb = col\_double(),

loan\_age = col\_double(),

remaining\_months\_to\_legal\_maturity = col\_double(),

adj\_remaining\_months\_to\_maturity = col\_double(),

maturity\_date = col\_date("%m/%Y"),

msa = col\_double(),

current\_loan\_delinquency\_status = col\_double(),

modification\_flag = col\_factor(NULL),

zero\_balance\_code = col\_factor(NULL),

zero\_balance\_effective\_date = col\_date("%m/%Y"),

last\_paid\_installment\_date = col\_date("%m/%d/%Y"),

foreclosed\_after = col\_date("%m/%d/%Y"),

disposition\_date = col\_date("%m/%d/%Y"),

foreclosure\_costs = col\_double(),

prop\_preservation\_and\_repair\_costs = col\_double(),

asset\_recovery\_costs = col\_double(),

misc\_holding\_expenses = col\_double(),

holding\_taxes = col\_double(),

net\_sale\_proceeds = col\_double(),

credit\_enhancement\_proceeds = col\_double(),

repurchase\_make\_whole\_proceeds = col\_double(),

other\_foreclosure\_proceeds = col\_double(),

non\_interest\_bearing\_upb = col\_double(),

principal\_forgiveness\_upb = col\_double(),

repurchase\_make\_whole\_proceeds\_flag = col\_factor(NULL),

foreclosure\_principal\_write\_off\_amount = col\_double(),

servicing\_activity\_indicator = col\_factor(NULL))

performance\_data <- vroom(

file = path\_perf,

delim = "|",

col\_names = names(col\_types\_perf),

col\_types = col\_types\_perf,

na = c("", "NA", "NULL"))

Let’s inspect the data. We can see that this is a time series where each “Loan ID” and “Monthly Reporting Period” go together.

performance\_data %>% head() %>% kable()

| **loan\_id** | **monthly\_reporting\_period** | **servicer\_name** | **current\_interest\_rate** | **current\_upb** | **loan\_age** | **remaining\_months\_to\_legal\_maturity** | **adj\_remaining\_months\_to\_maturity** | **maturity\_date** | **msa** | **current\_loan\_delinquency\_status** | **modification\_flag** | **zero\_balance\_code** | **zero\_balance\_effective\_date** | **last\_paid\_installment\_date** | **foreclosed\_after** | **disposition\_date** | **foreclosure\_costs** | **prop\_preservation\_and\_repair\_costs** | **asset\_recovery\_costs** | **misc\_holding\_expenses** | **holding\_taxes** | **net\_sale\_proceeds** | **credit\_enhancement\_proceeds** | **repurchase\_make\_whole\_proceeds** | **other\_foreclosure\_proceeds** | **non\_interest\_bearing\_upb** | **principal\_forgiveness\_upb** | **repurchase\_make\_whole\_proceeds\_flag** | **foreclosure\_principal\_write\_off\_amount** | **servicing\_activity\_indicator** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 100001040173 | 2018-02-01 | QUICKEN LOANS INC. | 4.25 | NA | 0 | 360 | 360 | 2048-02-01 | 18140 | 0 | N |  | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |  | NA | N |
| 100001040173 | 2018-03-01 |  | 4.25 | NA | 1 | 359 | 359 | 2048-02-01 | 18140 | 0 | N |  | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |  | NA | N |
| 100001040173 | 2018-04-01 |  | 4.25 | NA | 2 | 358 | 358 | 2048-02-01 | 18140 | 0 | N |  | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |  | NA | N |
| 100001040173 | 2018-05-01 |  | 4.25 | NA | 3 | 357 | 357 | 2048-02-01 | 18140 | 0 | N |  | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |  | NA | N |
| 100001040173 | 2018-06-01 |  | 4.25 | NA | 4 | 356 | 356 | 2048-02-01 | 18140 | 0 | N |  | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |  | NA | N |
| 100001040173 | 2018-07-01 |  | 4.25 | NA | 5 | 355 | 355 | 2048-02-01 | 18140 | 0 | N |  | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |  | NA | N |

Let’s check out the data size. We can see it’s **4.6M rows by 31 columns!** Just a typical financial time series (seriously).

dim(performance\_data)

## [1] 4645448 31

**4.5 Convert to Tibbles to dtplyr Steps**

Next, we’ll use the lazy\_dt() function to convert the tibbles to dtplyr steps.

acquisition\_data\_dtplyr <- lazy\_dt(acquisition\_data)

performance\_data\_dtplyr <- lazy\_dt(performance\_data)

We can check the class() to see what we are working with.

class(acquisition\_data\_dtplyr)

## [1] "dtplyr\_step\_first" "dtplyr\_step"

The returned object is the first step in a dtplyr sequence.

**Key Point:**

* We are going to set up operations using a sequence of steps.
* The operations will not be fully evaluated until we convert to a data.table or tibble depending on our desired output.

**4.6 Join the Data Sets**

Our first data manipulation operation is a join. We are going to use the left\_join() function from dplyr. Let’s see what happens.

combined\_dtplyr <- performance\_data\_dtplyr %>%

left\_join(acquisition\_data\_dtplyr)

The output of the joining operation is a new step sequence, this time a dtplyr\_step\_subset.

class(combined\_dtplyr)

## [1] "dtplyr\_step\_subset" "dtplyr\_step"

Next, let’s examine what happens when we print combined\_dt to the console.

combined\_dtplyr

## Source: local data table [?? x 55]

## Call: `\_DT2`[`\_DT1`, on = .(loan\_id)]

##

## loan\_id monthly\_reporti… servicer\_name current\_interes… current\_upb

##

## 1 100001… 2018-02-01 QUICKEN LOAN… 4.25 NA

## 2 100001… 2018-03-01 "" 4.25 NA

## 3 100001… 2018-04-01 "" 4.25 NA

## 4 100001… 2018-05-01 "" 4.25 NA

## 5 100001… 2018-06-01 "" 4.25 NA

## 6 100001… 2018-07-01 "" 4.25 NA

## # … with 50 more variables: loan\_age ,

## # remaining\_months\_to\_legal\_maturity ,

## # adj\_remaining\_months\_to\_maturity , maturity\_date ,

## # msa , current\_loan\_delinquency\_status ,

## # modification\_flag , zero\_balance\_code ,

## # zero\_balance\_effective\_date ,

## # last\_paid\_installment\_date , foreclosed\_after ,

## # disposition\_date , foreclosure\_costs ,

## # prop\_preservation\_and\_repair\_costs ,

## # asset\_recovery\_costs , misc\_holding\_expenses ,

## # holding\_taxes , net\_sale\_proceeds ,

## # credit\_enhancement\_proceeds ,

## # repurchase\_make\_whole\_proceeds ,

## # other\_foreclosure\_proceeds , non\_interest\_bearing\_upb ,

## # principal\_forgiveness\_upb ,

## # repurchase\_make\_whole\_proceeds\_flag ,

## # foreclosure\_principal\_write\_off\_amount ,

## # servicing\_activity\_indicator , original\_channel ,

## # seller\_name , original\_interest\_rate ,

## # original\_upb , original\_loan\_term ,

## # original\_date , first\_pay\_date , original\_ltv ,

## # original\_cltv , number\_of\_borrowers ,

## # original\_dti , original\_borrower\_credit\_score ,

## # first\_time\_home\_buyer , loan\_purpose ,

## # property\_type , number\_of\_units ,

## # occupancy\_status , property\_state , zip ,

## # primary\_mortgage\_insurance\_percent , product\_type ,

## # original\_coborrower\_credit\_score ,

## # mortgage\_insurance\_type ,

## # relocation\_mortgage\_indicator

##

## # Use as.data.table()/as.data.frame()/as\_tibble() to access results

**Key Points:**

* The important piece is the ***data.table translation code***, which we can see in the ouput: Call: \_DT2[\_DT1, on = .(loan\_id)]
* Note that we haven’t excecuted the data manipulation operation. dtplyr smartly gives us a glimpse of what the operation will look like though, which is really cool.

**4.7 Wrangle the Data**

We’ll do a sequence of data wrangling operations:

* Select specific columns we want to keep
* Arrange by loan\_id and monthly\_reporting\_period. This is needed to keep groups together and in the right time-stamp order.
* Group by loan\_id and mutate to calculate whether or not loans become delinquent in the next 3 months.
* Filter rows with NA values from the newly created column (these aren’t needed)
* Reorder the columns to put the new calculated column first.

final\_output\_dtplyr <- combined\_dtplyr %>%

select(loan\_id, monthly\_reporting\_period,

current\_loan\_delinquency\_status) %>%

arrange(loan\_id, monthly\_reporting\_period) %>%

group\_by(loan\_id) %>%

mutate(gt\_1mo\_behind\_in\_3mo = lead(current\_loan\_delinquency\_status, n = 3) >= 1) %>%

ungroup() %>%

filter(![is.na](http://is.na)(gt\_1mo\_behind\_in\_3mo)) %>%

select(gt\_1mo\_behind\_in\_3mo, everything())

The final output is a dtplyr\_step\_group, which is just a sequence of steps.

class(final\_output\_dtplyr)

## [1] "dtplyr\_step\_group" "dtplyr\_step"

If we print the final\_output\_dt object, we can see the *data.table translation* is pretty intense.

final\_output\_dtplyr

## Source: local data table [?? x 4]

## Call: `\_DT2`[`\_DT1`, .(loan\_id, monthly\_reporting\_period, current\_loan\_delinquency\_status),

## on = .(loan\_id)][order(loan\_id, monthly\_reporting\_period)][,

## `:=`(gt\_1mo\_behind\_in\_3mo = lead(current\_loan\_delinquency\_status,

## n = 3) >= 1), keyby = .(loan\_id)][![is.na](http://is.na)(gt\_1mo\_behind\_in\_3mo),

## .(gt\_1mo\_behind\_in\_3mo, loan\_id, monthly\_reporting\_period,

## current\_loan\_delinquency\_status)]

##

## gt\_1mo\_behind\_in\_… loan\_id monthly\_reporting… current\_loan\_delinq…

##

## 1 FALSE 10000104… 2018-02-01 0

## 2 FALSE 10000104… 2018-03-01 0

## 3 FALSE 10000104… 2018-04-01 0

## 4 FALSE 10000104… 2018-05-01 0

## 5 FALSE 10000104… 2018-06-01 0

## 6 FALSE 10000104… 2018-07-01 0

##

## # Use as.data.table()/as.data.frame()/as\_tibble() to access results

**Key Point**:

* The most important piece is that dtplyr correctly converted the grouped mutation to an **inplace calculation**, which is data.table speak for a super-fast calculation that makes no copies of the data. Here’s inplace calculation code from the dtplyr translation: [, :=(gt\_1mo\_behind\_in\_3mo = lead(current\_loan\_delinquency\_status, n = 3) >= 1), keyby = .(loan\_id)]

**4.8 Collecting The Data**

Note that up until now, ***nothing has been done to process the data*** – we’ve just created a *recipe* for data wrangling. We still need tell dtplyr to execute the data wrangling operations.

To implement all of the steps and convert the dtplyr sequence to a tibble, we just call as\_tibble().

final\_output\_dtplyr %>% as\_tibble()

## # A tibble: 3,352,231 x 4

## gt\_1mo\_behind\_in\_… loan\_id monthly\_reporting… current\_loan\_delinq…

##

## 1 FALSE 1000010… 2018-02-01 0

## 2 FALSE 1000010… 2018-03-01 0

## 3 FALSE 1000010… 2018-04-01 0

## 4 FALSE 1000010… 2018-05-01 0

## 5 FALSE 1000010… 2018-06-01 0

## 6 FALSE 1000010… 2018-07-01 0

## 7 FALSE 1000010… 2018-08-01 0

## 8 FALSE 1000010… 2018-09-01 0

## 9 FALSE 1000023… 2018-03-01 0

## 10 FALSE 1000023… 2018-04-01 0

## # … with 3,352,221 more rows

**Key Point**:

* Calling the as\_tibble() function tells dtplyr to execute the data.table wrangling operations.

**5.0 The 3X Speedup – Time Comparisons**

Finally, let’s check the performance of the dplyr vs dtplyr vs data.table. We can seed a nice **3X speed boost!**

**5.1 Time using dplyr**

tic()

performance\_data %>%

left\_join(acquisition\_data) %>%

select(loan\_id, monthly\_reporting\_period,

current\_loan\_delinquency\_status) %>%

arrange(loan\_id, monthly\_reporting\_period) %>%

group\_by(loan\_id) %>%

mutate(gt\_1mo\_behind\_in\_3mo = lead(current\_loan\_delinquency\_status, n = 3) >= 1) %>%

ungroup() %>%

filter(![is.na](http://is.na)(gt\_1mo\_behind\_in\_3mo)) %>%

select(gt\_1mo\_behind\_in\_3mo, everything())

toc()

## 15.905 sec elapsed

**5.2 Time using dtplyr**

tic()

performance\_data\_dtplyr %>%

left\_join(acquisition\_data\_dtplyr) %>%

select(loan\_id, monthly\_reporting\_period,

current\_loan\_delinquency\_status) %>%

arrange(loan\_id, monthly\_reporting\_period) %>%

group\_by(loan\_id) %>%

mutate(gt\_1mo\_behind\_in\_3mo = lead(current\_loan\_delinquency\_status, n = 3) >= 1) %>%

ungroup() %>%

filter(![is.na](http://is.na)(gt\_1mo\_behind\_in\_3mo)) %>%

select(gt\_1mo\_behind\_in\_3mo, everything()) %>%

as\_tibble()

toc()

## 4.821 sec elapsed

**5.3 Time using data.table**

DT1 <- as.data.table(performance\_data)

DT2 <- as.data.table(acquisition\_data)

tic()

DT2[DT1, .(loan\_id, monthly\_reporting\_period, current\_loan\_delinquency\_status), on = .(loan\_id)] %>%

.[order(loan\_id, monthly\_reporting\_period)] %>%

.[, gt\_1mo\_behind\_in\_3mo := lead(current\_loan\_delinquency\_status, n = 3) >= 1, keyby = .(loan\_id)] %>%

.[![is.na](http://is.na)(gt\_1mo\_behind\_in\_3mo), .(gt\_1mo\_behind\_in\_3mo, loan\_id, monthly\_reporting\_period, current\_loan\_delinquency\_status)]

toc()

## 4.627 sec elapsed

**6.0 Conclusions and Learning Recommendations**

**For Big Data wrangling**, the dtplyr package represents a ***huge opportunity*** for data scientists to leverage the *speed* of data.table with the *readability* of dplyr. We saw an impressive ***3X Speedup*** going from dplyr to using dtplyr for wrangling a 4.6M row data set. This just scratches the surface of the potential, and I’m looking forward to seeing dtplyr mature, which will help bridge the gap between the two groups of data scientists using dplyr and data.table.

For new data scientists coming from other tools like Excel, my hope is that you see the awesome potential of learning R for data analysis and data science. The Big Data capabilities represent a ***massive opportunity for you to bring your organization data science at scale.***

**You just need to learn how to go from normal data to Big Data.**

***My recommendation is to start by learning dplyr*** – The popular data manipulation library that makes reading and writing R code very ***easy to understand***.

Once you get to an intermediate level, learn data.table. This is where you gain the benefits of scaling data science to Big Data. The data.table package has a steeper learning curve, but learning it will help you leverage its full performance and scalability.

**7.0 Additional Big Data Guidelines**

I find that students have an easier time picking a tool based on ***dataset row size*** (e.g. I have 10M rows, what should I use?). With that said, there are **2 factors** that will influence whhich tools you need to use:

1. **Are you performing Grouped and Iterative Operations?** Performance even on normal data sets can become an issue if you have a lot of groups or if the calculation is iterative. A particular source of pain in the financial realm are ***rolling (window) calculations***, which are both grouped and iterative within groups. In these situation, use high-performance C++ functions (e.g. Rolling functions from the [roll package](https://github.com/jjf234/roll) or [RcppRoll package](https://github.com/kevinushey/RcppRoll)).
2. **Do you have sufficient RAM?** Once you begin working with gig’s of data, then you start to run out of memory (RAM). In these situations, you will need to work in chunks and parellelizing operations. You can do this with distributed [sparklyr](https://spark.rstudio.com/), which will perform some operations in parallel and distribute across nodes.

**8.0 Recognizing the Developers**

I’d like to take a quick moment to thank the developers of data.table and dplyr. Without these two packages, *Business Science* probably would not exist.