What Big Data is, and How to Deal with It

By Jeff Heaton

Big data is a frequent participant in headlines today. The amount of electronic data available is growing at an exponential rate. Every few years new terms such as megabyte, gigabyte, terabyte, petabyte, exabyte, and even zettabyte enter everyday vocabulary as a new measure for "really large data." Some estimates state that the total amount of electronic data by 2020 will exceed 35 zetabytes. But what exactly is "big data," and how much of those 35 zetabytes will a typical company actually need to process?

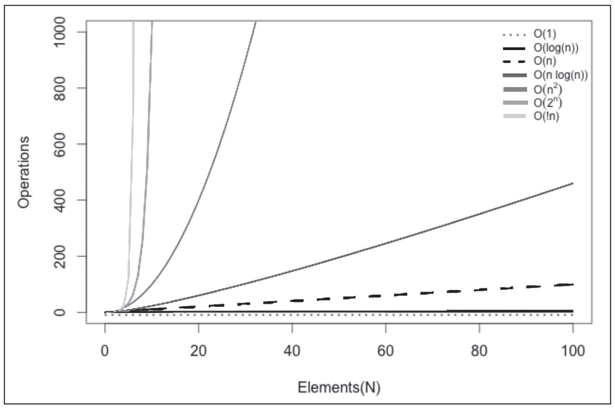
Much hype surrounds the term "big data," and several definitions exist for the term. One of the most useful definitions of big data comes from Wikipedia, "big data is a broad term for data sets so large or complex that traditional data processing applications are inadequate." There are several thresholds effectively established by this definition. Will the data fit into a server's RAM? Will the data fit onto a single hard disk drive? As the size of the data grows more traditional tools begin to fail. There are a multitude of companies ready to sell you new tools to handle big data. Often these tools cost big dollars.

Going by these definitions, big data is nothing new. If your computer has 16K of RAM, then 17K is "big data." Back in the 1990s I had to make many modifications to a C++ application to allow it to make use of its full 2 MB of RAM. The Intel architecture of the time could only access 1MB of RAM at a time. My program had to share the lower 640K with DOS and map sections of the EMS memory into the upper 384 MB of the address space. Was this "big data?" In a sense it was "big data," the problem had become large enough that it no longer fit into RAM.

WHY IS BIG DATA HARD

Big data is hard because computer programs do not always scale well. In computer science, the scalability of a computer program is measured in something called big O notation. You may have heard of algorithms referred to as running in O(log N), O(N2) squared or even O(N!) exponential time. These refer to how well the program scales to its data set.

The most efficient computer program would be O(1) time. Such a program will always run in the same amount of time, regardless of how large the data set is. Consider a program that finds the first name in a list. Such a program will always take the same amount of time because it does not matter if the list has 10 items or 10 million items. Very few things run in O(1) time, however, O(n) is reasonably good as well. Consider if I asked you to find the longest name in a list. For this you must visit each item in the list, so it is O(n). Assuming *n* is the number of items in your list. The processing time should scale linearly. If it takes 10 minutes to process 10 items, it should take 100 minutes to process 100 if you are dealing with an O(N) algorithm.



Not every algorithm behaves linearly. Knowing the O-magnitude of an algorithm can help you decide which to use. The seven most common magnitudes are shown on the following chart.

As can be seen from the chart algorithms, the most favorable magnitude algorithms are O(1) , O(log(n)) and O(n). The least favorable are O(n2), O(2n) and o(!n).

If n is relatively small, it does not matter what the magnitude of your algorithm is. However, as n grows, so does the processing time of the algorithm. Some algorithms simply do not work with big data because of their magnitude. When dealing with a high-magnitude algorithm, and big data, it is often necessary to accept an approximation, rather than process the entire data set. Some algorithms that initially seem high-magnitude can be rewritten to be more efficient.

CONTINUED ON **PAGE 44**

WHAT BIG DATA IS … | FROM PAGE 43

THE LANGUAGE OF BIG DATA

Big data has its own terminology, just like any other field. Doug Laney, of the META Group defined datasets in terms of three V's. This has come to be known as the three V's of big data. The first V, volume, describes the size of the data set. This is the characteristic that frequently comes to mind when discussing big data. The second V, variety, describes the complexity of the data. When dealing with big data there will often be several large datasets of different variety. This can pose unique challenges for the algorithms that must process these datasets. The final V, velocity, describes the rate at which the data is changing. The underlying dataset will often change during the time that the big data algorithm is processing.

Velocity introduces streaming, which is another important big data concept. Streaming, or real-time processing, refers to a large amount of data that arrives continuously over time. The amount of data arriving in the stream may increase and decrease as the stream of information flows into your program. Examples of stream data include trading, fraud detection, system monitoring, and others.

Out-of-core, or external memory algorithms, is another important concept for big data. Such algorithms do not use computer RAM to process their datasets. It is very common practice to load an entire dataset into memory and then process it. However, this is not always necessary. Even if a low-magnitude O(N) algorithm is chosen, it will fail as soon as n grows to the point that the list can no longer fit into memory. Consider calculating the mean of numbers in a very large list. A computer program could read the list, number-by-number, and maintain two variables. The first variable keeps a sum of the numbers encountered, and the second variable keeps a count of the number elements processed so far. At the end, these two variables will hold the sum of the list, as well as the count of items in the list. Simply divide the sum by the count and you have the mean. It does not matter how large the list is, you will have sufficient memory to calculate this mean.

Vowpal Wabbit is a popular out-of-core machine learning framework. By using memory only as a cache, Vowpal Wabbit is capable of processing any size dataset. It might take Vowpal Wabbit a very long time to process a dataset; however, it would not run out of memory and crash, like many similar programs. This is very similar to how programs were written in the past when RAM was scarce. Modern computers, with their large memory systems, often encourage programmers to not pay attention to their memory usage. Programs that naively load entire datasets into RAM simply will not scale to large amounts of data.

TOOLS FOR BIG DATA

Two of the most commonly used tools for big data are Hadoop and Spark. The Apache Foundation manages both of these programs. Hadoop is the foundation upon which Spark is built. Hadoop provides distributed file storage and the communication infrastructure needed by Spark. Hadoop uses the map-reduce algorithm to perform distributed processing. Map-reduce requires considerable disk I/O, as large problem spaces are mapped into parts, and those parts combine and reduce into the ultimate solution. Spark uses Resilient Distributed Data (RDD) to break the problem into many pieces that can be processed in RAM on the nodes. Whereas Hadoop needs fast disk I/O, Spark needs considerable RAM. For the right tasks, this can mean processing time increases of 100 times compared to Hadoop alone.

One type of problem that excels under Spark is machine learning. The ability to break the problem into many units executed in RAM is very conducive to many machine learning algorithms. Spark has a model called MLlib, or Machine Learning Library that provides many machine learning models right out of the box. Hadoop, along with another Apache framework called Pig, is very good at performing traditional SQL queries over very large datasets.

TOWARD SCALABLE ALGORITHMS

Traditional programming wisdom says to first focus on getting a working program and optimize later. Donald Knuth is quoted as saying, "Premature optimization is the root of all evil (or at least most of it) in programming." While this is generally true, big data forces optimization to increase in priority. Analytics often forces many runs before the desired result is achieved. The shorter a runtime that you achieve, the more experimentation you can do.

Many common programming tasks have both naive and optimized implementations. Consider some of the following operations on a big list of numbers:

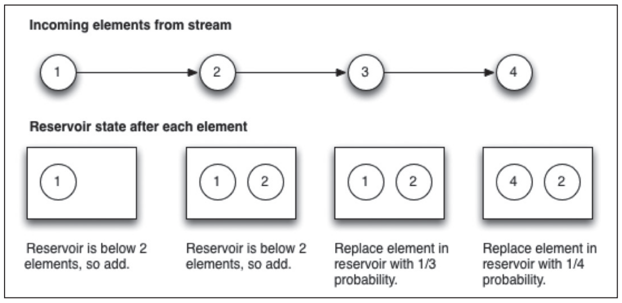
- Percentile and Quintile Estimation,
- Randomly sampling a subset,
- Sorting,
- Taking the mean,
- Taking the standard deviation,
- and more.

Each of the above algorithms has naïve and optimized approaches. Searching and sorting are among the most researched algorithms in computer science for efficiency. Consider the standard deviation, which normally requires two passes over the data. First you calculate the mean, and then you calculate the mean deviation of each data point from that mean. A naïve standard deviation calculation requires two passes over the data. There are algorithms that can do it with one pass. These same algorithms are also good for calculating the mean and standard deviation over an endless stream of numbers.

Reservoir sampling is a very common big data technique that can be used to randomly sample a set of numbers from a very large pool. Consider if you wanted to randomly choose two people from the world population. The naïve approach would be to visit each person in the world once to obtain an accurate count and place him or her into a consistent ordering. You would then select two random numbers up to the world population count. Using this number, you would now visit everyone in the world again, and stop at the index numbers that you randomly chose in the previous step.

This approach has several issues. First, between the two passes, people would have been born and died. Barring a large-scale natural disaster, the world population count would be higher for your second pass than the first. The sample would no longer be uniform, and would bias against those that were born since the first pass. This is the velocity problem of big data. However, the biggest problem is that it is potentially necessary to visit everyone twice. The two-pass method also becomes nearly impossible to use when dealing with an endless stream of data.

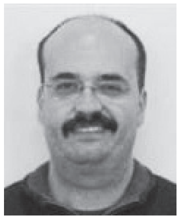
The following figure illustrates how to use reservoir sampling with a stream of numbers.


Incoming elements from stream
1
2
3
4
Reservoir state after each element
1
1 2
1 2
4 2
Reservoir is below 2 elements, so add.
Reservoir is below 2 elements, so add.
Replace element in reservoir with 1/3 probability.
Replace element in reservoir with 1/4 probability.

To sample two elements from a large stream of numbers you simply add the first two to the reservoir. When selecting the third element you now replace an element in the reservoir with #3 with a 1/3 probability. Likewise, for the fourth element you replace an element in the reservoir with a 1/4 probability. This continues for as much data as you have.

CONCLUSIONS

Big data presents many challenges for analytics systems. It is very important to choose tools and underlying algorithms that will scale to the size of your data. Data have a tendency to grow as systems mature. The sooner in the development cycle that you make scalability decisions the better. Tools designed to work with big data can help to facilitate this growth, even if you are not dealing with big data today. ▼


Jeff Heaton

Jeff Heaton is data scientist, Global R&D at RGA Reinsurance Company in Chesterfield, Mo. He can be reached at jheaton@rgare.com.

A **'Hot Date'** with *Julia*: Parallel Computations of Stochastic Valuations

By Charles Tsai

Meet "Julia," a free programming language licensed by MIT that may help you with parallel computing. It may be an alternative tool for those who are interested in nested stochastic processes for actuarial research (if not for regulatory compliance).

Nested stochastic processes may become more relevant and prevalent as stakeholders consider a broader spectrum of possible outcomes. Such "stochastic-in-stochastic" analyses often add color to actuaries' palette of tail risks and conditional tail dependencies (if any). However, they also introduce issues of runtime and memory allocation. The article "Nested Stochastic Pricing"[1] provides a comprehensive summary of nested stochastic applications in response to recent regulatory reforms. IFRS seems to require a comprehensive range of scenarios that reflects the full range of possible outcomes for calculating fulfillment cash flows. Economic capital calculations may likewise require stochastic-in-stochastic simulations. A practice that may have been previously deemed as a costly bonus may evolve into a minimum expectation for actuaries in the near future.

Nested stochastic processes may become more acceptable with parallel computations. One may boil down "parallel computing" to daily applications with an analogy. Imagine an investment banker who is planning a date with a lady. He barely has enough time to smoke, and he has completing the following four tasks in mind: 1) dress up, 2) buy flowers, 3) research a restaurant's menu, and 4) fold a thousand origami cranes. He has made these preparations in solo for all of his previous dates. Would it not be nice for him to have friends help him perform the latter three tasks *simultaneously*? Delegation may take some time, but it may be more efficient than performing all four tasks in sequence. Parallel computing is a form of dividing and conquering problems using multiple processes concurrently. It may help actuaries slam-dunk tasks like traversing a thousand scenarios, even if the tasks already take less time than folding a thousand origami cranes.

Julia allows users to distribute and execute processes (such as nested stochastic valuations) in parallel. In essence, a computer may have four Central Processing Units (CPUs) in resemblance to a soccer team with four members. Programmers can leverage Julia's multiprocessing environment to specify certain tasks to those CPUs on the bench. On the one hand, the art of scheduling may be a bulk process for infrequent and smaller tasks. On the other hand, the flexibility to pass messages to multiple processors may be one's niche in strategic scalability and performance. Actuaries may then manage disparate layers of stochastic simulations via a multiprocessing environment. Shorter runtimes may be a doomsday for a few students who use waiting time as an opportunity for studying. However, such efficiency opens doors to comprehensive iterations and widens windows of perspectives.

IS JULIA A DISRUPTIVE INNOVATION?

Julia has several features[2] that supplement its power in parallelism and distributed computation. Some features are for specialists like Sheldon Cooper (of *The Big Bang Theory*) while others may be easier for amateurs like me to appreciate.

- First, it is free and open sourced as licensed by MIT. Actuaries can share research results seamlessly at SOA/CAS events without worrying about whether the audiences have access to the same tools to review (and build upon) the findings.
- Second, users can define composite types that are equivalent to "objects" in other languages. These user-defined types can run "as fast and compact as built-ins".[3]
- Third, users can call C functions directly, and their programs' performances can approach those of languages like C. Such speed makes it a considerable alternative to proprietary computational software tools.[4]
- Fourth, one does not need to be a genius like Gaston Julia in order to learn the language. Justin Domke's blog post "Julia, Matlab, and C"[5] presents a crystal clear comparison of syntactic and runtime complexity tradeoffs. Learning Julia is like learning Matlab® and C++ for Towers Watson MoSes® simultaneously.

- Last but not least, Julia is a functional programming language like OCaml, which is adopted by niche firms like Jane Street. Functional programming frameworks can help actuaries adapt to and master recursions.

Julia also has several Achilles' heels that may significantly jeopardize its adoption among actuaries.

- One obstacle is communication. Due diligence may be lost in translation. A few know how to use and interpret proprietary actuarial software products due to limited availability. Fewer know how to read and review (or even find) its generated C++ codes. In a like manner, few have learned (or are willing to learn) the Julia language, and its graphical features are still under development. Some actuaries may still prefer parallel computations via multiple Microsoft Excel® sessions. Calibrations of Julia programs with validated Microsoft Excel® workbook models might just have exceeded paychecks.

- Another hindrance is the language's relative immaturity. Development commenced in 2009.[6] Its scale of recognition seems to be light years from the tipping point for a stabilized discussion ecosystem to exist. Online inquiries for relevant debugging notes make passing bills during gridlocks look easy. A tool may only be as valuable as its received appreciation.

- Lastly, the manipulation of processes in parallel computations requires an acute awareness of read-write conflicts. In light of the previous analogy, the banker may wish to match his suit with the flowers purchased, or the flowers purchased with the restaurant's cuisine. Tasks may not be completely independent from each other. Inexperienced users may inadvertently manipulate and designate processes in manners inconsistent with intentions.

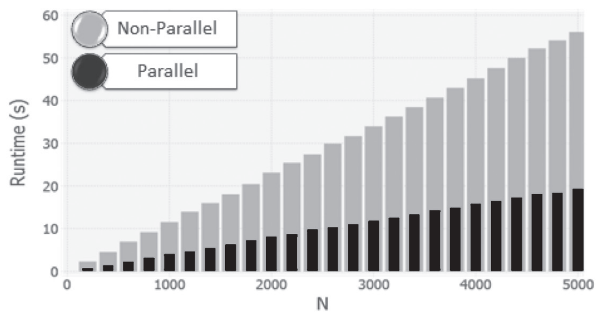
A SIMPLIFIED GMMB CASE STUDY

I have drafted an exemplary Julia application of an actuarial model. It is available at *https://github.com/Chuckles2013/GMMB_RSLN2_Julia*, and is an independent project for educational purposes only. All parameters and values have been arbitrarily chosen. The case study involves calculating the present values of liabilities for an extremely simplified Guaranteed Minimum Maturity Benefit (GMMB).

The scale of the project can be partitioned into two major layers. The first layer involves simulating parameters for N world scenarios. For simplicity, I have structured all key parameters to be the same across all N world scenarios. It is easy to see that one can simply modify the codes to utilize simulated parameter inputs for considering different world scenarios and economic environments. The second layer involves simulating fund returns for 1000 funds, from which one can derive a conditional tail expectation of liabilities. Both layers provide N figures of conditional tail expectations, from which one can extract a maximum level.

The superimposed bar graph below compares runtimes for non-parallel versus parallel computations under various numbers (*N*) of world scenarios. Four processors performed the parallel computations. The absolute values of the excess time elapsed are evident in the divergent gap.

Runtime Comparisons Across Different Numbers of Scenarios (When 4 CPUs Are Available)



CONTINUED ON PAGE 48

A '*HOT DATE*' WITH JULIA … | FROM PAGE 47

NEXT STEPS

One's vision for Julia in actuarial science can be the development of packages. A few companies were bold enough to have utilized R, and none has adopted (or even plan to leverage) Julia to my knowledge. Full adoption of Julia among actuaries within the next decade may be more of a fantasy than a reality, just as few actuaries have learned Python since its inception in 1991.[7] Nevertheless, open-source packages for broader usage are lower hanging fruit for intrigued actuaries to consider. To the best of my knowledge, there are no Julia packages similar to the lifecontingencies and actuar packages in R libraries. Templates of actuarial functions in Julia may capture more attention and appreciation for the beauty of parallel computations for nested stochastic valuations. ▼

Charles Tsai

Charles Tsai, ASA, is a Life Actuarial Analyst at AIG in Shanghai, China. He can be reached at *charles-cw.tsai@aig.com.*

ENDNOTES

1 "Nested Stochastic Pricing: The Time Has Come" by Milliman®'s Craig Reynolds and Sai Man is available at http://www.milliman.com/insight/insurance/pdfs/Nested-stochastic-pricing-The-time-has-come/

2 http://julialang.org/

3 http://nbviewer.ipython.org/github/bensadeghi/julia-datascience-talk/blob/master/datascience-talk.ipynb

4 Professor Fernández-Villaverde's "A Comparison of Programming Languages in Economics", which is available at www.econ.upenn.edu/~jesusfv/comparison_languages.pdf

5 http://justindomke.wordpress.com/2012/09/17/julia-matlab-and-c/

6 web.maths.unsw.edu.au/~mclean/talks/Julia_talk.pdf

5 This is a rather fun proof left for the reader. First, prove that each row of (I – S) sums to zero. What does this imply about the triangularized matrix?

7 http://svn.python.org/view/*checkout*/python/trunk/Misc/HISTORY