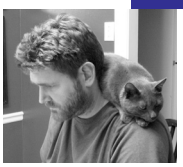
DAVE JOSEPHSEN

iVoyeur: hold the pixels



David Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and Senior Systems Engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his coauthored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.

dave-usenix@skeptech.org

"CHECK THIS OUT," HE SAYS WITH THE merest hint of pride in his voice. As admins go, he's one of the best I've had the pleasure of working with—clever, thorough, and tenacious, so when he has something to show off, it's usually pretty impressive. This particular hack happens to be a billing program based on customer bandwidth usage. As you can probably imagine, many things are involved: SNMP, data stores, and lots of arithmetic. His data structures are beautiful, and his design is modular and elegant, but there is a rather large problem, and it has nothing to do with his code. While I page through the file my subconscious nags me, "Let's see, you're getting data from routers, you're storing it in a database, you're performing math on it . . . yep, complete reinvention of the wheel." So after nodding appreciatively, and knowing the answer before I ask the question, I ask, "Why didn't you use RRDtool?"

"I didn't need to draw graphs," he replies. I sigh inwardly. I've seen this one before; heck, I've even been guilty of it myself. One of the traps good admins often fall into is under-scoping our tools. Given tools that do one thing well, we sometimes miss what that thing actually is, and "graphing stuff" certainly isn't RRDtool's one thing (that'd be gnuplot). What RRDtool [1] excels at is dealing with time-series data, whether you need pretty pictures or not.

In fact, just about all the graph-related options to RRDtool's graph command are optional, and this is by design. In practice RRDtool is a great tool from the command line, and in fact my friend's entire billing program sans the data collection portions could have been done using nothing but RRDtool in command-line mode. I know all of the gripes: Reverse polish notation is . . . well, weird, RRDtool's command-line syntax can't be committed to memory, the data storage aspects aren't very transparent, and on and on. I submit that RRDtool is still the best tool available for dealing with any sort of time-series data in general.

Let me show you what I mean. Since my friend put his data in a mysql database, he has to pro-

grammatically extract it, calculate usage information for the given time interval, and compute the cost of that bandwidth. If he had stored it in an RRDtool database (using a gauge RRA, or a compute RRA to compute and store raw byte counts) all of this is (arguably) four lines of code:

```
rrdtool graph foo.png --start='now -1month' \
DEF:a=routerA_customerinterface4.rrd:traffic_counter:MIN \
DEF:b=routerA_customerinterface4.rrd:traffic_counter:MAX \
CDEF:price=a,b,-,1024,/,.05,* \
PRINT:price:%8.2lf | tail -n1
```

It's not my aim in this article to teach you RRDtool syntax, so I'm mostly going to let these examples speak for themselves. Suffice to say that in the short example above, we extracted the data we needed, and used it to calculate the price of a customer's bandwidth usage over the last month, assuming a price of five cents a meg. I will point out the use of the print statement, which simply prints the answer to stdout, suitable for piping to other applications such as tail. PRINT makes RRDtool into a command-line app. Although we specified the name of an output png (this, for reasons unknown to me, is required), we specified no graph-related options, so RRDtool didn't draw one; its only output was the cost of customer4's bandwidth for the last month, in text, to STDOUT.

This implementation has obvious advantages over rolling your own solution in Perl, not the least of which is speed. RRDtool internally performs math really quickly. Using "time" to measure the execution time of this problem in RRDtool yields "real 0m0.004s" versus my friend's Perl solution at "real 0m0.103s." Your results will surely vary, but I've yet to find a faster way to process heaps of time-series data. RRDtool also scales really well. An aversion to RPN is no excuse; you can spend 30 minutes learning the syntax or an hour writing your own code to perform the math, and your code will almost certainly be slower.

RRDtool takes care of data compression and rotation, temporal interpolation, data glitches, conversions of counters to rate information, and I/O access, without you having to do anything. It does this remarkably efficiently and on tons of data. It is as efficient when processing a month of data starting five years ago as it is processing a month of data ending five minutes ago. To write an application that handled your particular time-series data more effectively you'd probably need to start by forking RRDtool.

Good admins might suspect that a tool this efficient was over-optimized for the solution set and couldn't possibly be very flexible. I might agree, but I'd also wager they hadn't played with some of RRDtools more esoteric features such as CDEF conditionals. For example, my friend's billing program doesn't actually bill the customer unless the customer uses more than 100 Mb of bandwidth. We can use CDEF conditionals to solve this problem:

```
CDEF:bandwidth=a,b,-,1024,/ \
CDEF:price=a,b,-,1024,/,100,-,.05,* \
CDEF:finalprice=bandwith,100,GT,0,price,IF \
```

In pseudo-code this comes out to "if (bandwidth > 100) then finalprice=price else finalprice=0." This doesn't actually require three separate CDEFs; it could have been accomplished with a single long, convoluted RPN expression if you, for example, hated those who will come after you and wanted to give them something painful to decrypt. The point is that RRDtool can do a surprising amount of the work you need done in-

ternally, and without involving any pixels. So if you ever find yourself writing code to process time-series data or find yourself using RRDtool fetch to export data so that you can process it in a general-purpose language, you're probably doing it wrong. I'm just saying.

Aberrant Behavior Detection

And while I'm at it, I should add that if you're exporting data so you can run statistical analysis on it or send alarms based on thresholds, you're probably doing it wrong here as well. A few years ago, Jake Brutlag added some really cool statistical forecasting and problem-detection software to RRDtool. He published his work at LISA 2000 [2]. His work, collectively referred to as "Aberrant Behavior Detection," is the coolest code I'm aware of that nobody *ever* uses. Nobody seems to use it mostly because everyone thinks "it's hard"; yes, what little documentation there is, is not fun to read. So allow me to take a crack at putting this down where the goats can reach it (to quote an old boss of mine).

The problem with predicting the future is that a lot of noise gets in the way. One of the oldest, easiest methods we have to predict future data points within a set of time-series data is the simple moving average. Every time we get a new data point we re-average the data set, and the next point should land somewhere near that. Maybe we drop values that are more than *x* days old. This obviously isn't very accurate, and it suffers from a number of other problems as well. The two big ones are that it's very compute-intensive and we have to keep a lot of data points around (especially if we're polling every few minutes).

In the late 1950s, C.C. Holt proposed a formula that was more accurate and only needed two data points to compute. It's pretty clever and even came with a constant that you can tweak to give more recent values greater weight. In fact, giving more recent values greater weight in general was sort of a design goal of the equation, and it's where it derives its greater accuracy. This first equation is called "exponential smoothing." Within the Holt-Winters Method it's also referred to as the "baseline" equation. As your data set changes, the equations' predictions do so as well. If your data rises rapidly, it will outpace the predictions and your data may therefore be considered statistically improbable. The constant controls how quickly the data can change before it is considered abnormal. I'm not going to get into the specific math here because of the aforementioned goats. If you want to get a look at the equations, take a look at Brutlag's paper [2] and/or his implementation notes [3].

The only problem with exponential smoothing was that it didn't account for longer-term trends in the data set. This is of course what you would expect when you design a formula that gives greater weight to more recent values, so a few years later, Holt proposed another formula to account for longer term trends in the data. This formula, called "Double Exponential Smoothing," can basically be thought of as exponential smoothing of exponential smoothing. This formula also uses a tweakable constant to allow fine-tuning for the extent to which data trends affect the final prediction. This equation is sometimes referred to as the "slope" equation in the Holt-Winters Method.

The final piece of noise that Holt's equations didn't account for was periodic fluctuations (for example, people shopping more at Christmas or load from machines being backed up every night at 2 a.m.). In 1965 Winters proposed an equation to absorb the periodic (or "seasonal") noise.

This final equation uses yet another constant to control the weight of periodic data in the final prediction. The three equations combine to form the Holt-Winters Method, which can make pretty accurate predictions in sets of time-series data, or at least predictions that are good enough for our purposes.

Jake Brutlag took Holt-Winters and bolted it onto RRDtool, giving RRDtool the power to make real-time predictions for any given data set. He also added an easy mechanism for graphing error bars around the predicted data points, and for determining when a real data point or range of data points falls outside the error bars. All of this is implemented in a series of special-purpose RRAs, but in practice you don't need to know much about the inner workings of Holt-Winters or any of the special RRAs save one to get this working for you. All you need to do is create the HWPREDICT RRA and the others are implicitly created for you. The syntax looks like this:

```
RRA:HWPREDICT:<array length>:<alpha>:<beta>:<period>
```

The array length is the number of predictions you want to keep. This number needs to be as many data points as you want to graph error bars for, or at least as many data points as the seasonal period you want to account for. The seasonal period is usually 24 hours, so for an RRD with a 5-minute polling interval, the array length should be at least 288, but again, if you plan on drawing graphs with error bars, you probably want this to be a good deal bigger (10,080 gives you a week of error bars).

Alpha is the exponential smoothing constant I mentioned above. It is a number between 0 and 1. You can calculate this given a small judgment call on your part. The formula (taken from Brutlag's paper and translated to Perl syntax by me) for calculating the value is:

```
$alpha = 1-(exp(log(1-$pct)/$points))
```

The question you have to ask yourself is how quickly you want the baseline forecasts to adapt to changes in the data. If for example, you wanted observations in the last 30 minutes to account for 90% of the baseline weight, you would plug in .90 for $pct, and 6 for $points (assuming a 5-minute polling interval (30/5)). This yields an alpha of about .32, which, in my experience is a pretty reasonable starting point.

The same formula can be used to calculate beta, which is the slope (or trending) equation constant. Since the idea of the slope equation is to account for long-term trends, you should use a value for $points that is at least as long as the seasonal period. The seasonal period is most often 24 hours, so for a polling interval of 5 minutes $points should be at least 288, if not higher. A starting value of .0024 for beta will ensure that observations in the last day will account for no more than 50% of the slope equation smoothing weight. This is the same value Brutlag mentions in his paper and it's the one I usually start out with.

Finally, "period" is simply the periodic or seasonal period you want to account for. As I've mentioned this is usually 24 hours (288 points for a 5-minute polling interval).

If you want to get fancy, there's a whole lot more tweaking you can do, but for most admins wanting a fairly good and scalable method for detecting abnormal patterns in their time-series data, that's pretty much all there is to it.

Once the RRD with HWPREDICT in it is created, you can ask RRDtool to give you the instances of aberrant behavior for a given time interval from

the command line via the implicitly created FAILURES RRA. This is what most admins are looking for when they create Nagios plug-ins to dump and search through RRD data. For example, if I wanted to know the number of times customer4's interface displayed abnormal traffic patterns in the last 5 minutes, I could do this:

```
rrdtool graph foo.png --start='now -5minutes' \
DEF:failures=routerA_customerinterface4.rrd:traffic_counter:FAILURES \
PRINT:failures:%8lf | tail -n1
```

Anyway, I hope I've made a good case for the wholesale abandonment of the use of <insert your goto language here> for processing time-series data. RRDtool really is a heck of a piece of software, so if you thought it was all about graphing I urge you to take another look.

Take it easy.

REFERENCES

[1] http://oss.oetiker.ch/rrdtool/.

[2] http://www.usenix.org/events/lisa00/full_papers/brutlag/brutlag_html/index.html.

[3] http://cricket.sourceforge.net/aberrant/rrd_hw.htm.