iVoyeur
Lies, Damned Lies, and Averages

DAVE JOSEPHSEN

Dave Josephsen is the sometime book-authoring developer evangelist at Librato.com. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

You know that movie where the guy takes hostages and duct-tapes them together and/or makes them wear gauche vests laden with random assortments of electronic components, and then demands all sorts of zany things like millions of dollars and a helicopter capable of flying him to Tahiti?

Sometimes I fantasize about being that guy. Not because I want to scare or harm anyone, and I certainly wouldn't wish those terrible vests on my worst enemy, but it *would* be fun to make zany commands over a bullhorn to a group of confused, yet eager to please, FBI agents.

Just think of the fun we could have. We could establish a holiday for things that are pickled. We could demand that every law-enforcement-related uniform and vehicle in the nation, regardless of jurisdiction, be painted pink (especially the drones, tanks, and mobile command-center RVs). We could bring back *Firefly*, banish Michael Bay AND George Lucas, force Starbucks to admit that granulated sugar really is sweeter than raw sugar...we could outlaw tactical vests.

You know, while we're on the subject, there is something that's been bothering me. Something for which I'd like to demand a fix. There's some talk going around lately about how we collect and persist metrics from systems and applications in the wild (a good thing) [1]. If I could strap ugly vests to people and demand something today, it might be a fix for one of my own metrics pet peeves that, for whatever reason, doesn't seem to have entered the discussion.

Metrics data is deceptively large because it's composed of such disarmingly innocuous little date/value tuples. It just seems unlikely that such harmless little measurements could possibly strain the storage device of even a respectable smartphone much less a grandiose server.

They add up, though. Every one of those little metrics, stored as a float, measured every five seconds, and persisted for a year, takes up around 400 MB of space. Two metrics from a single source, stored in their raw format, therefore, requires almost a gigabyte of storage. This modest storage conundrum is the primary hurdle to overcome in time-series data systems. We simply don't have the space to store thousands of measurements from hundreds of systems in their raw form, for periods of a year or more.

Enter Consolidation Functions

Most contemporary databases that are designed to store time-series data begin with a fundamental observation, namely, the older the data is, the less we care about it. If this is true, it means we don't actually need to store the raw measurements forever. Instead, we can keep the raw measurements for a short time and consolidate the older data points to form a smaller set of samples that adequately summarizes the data set as a whole.

This is usually accomplished automatically inside the datastore with a series of increasingly drastic data consolidations. You can think of the datastore itself as a series of time buckets. High-resolution, short-term buckets are very large. They can keep a bunch of data points in

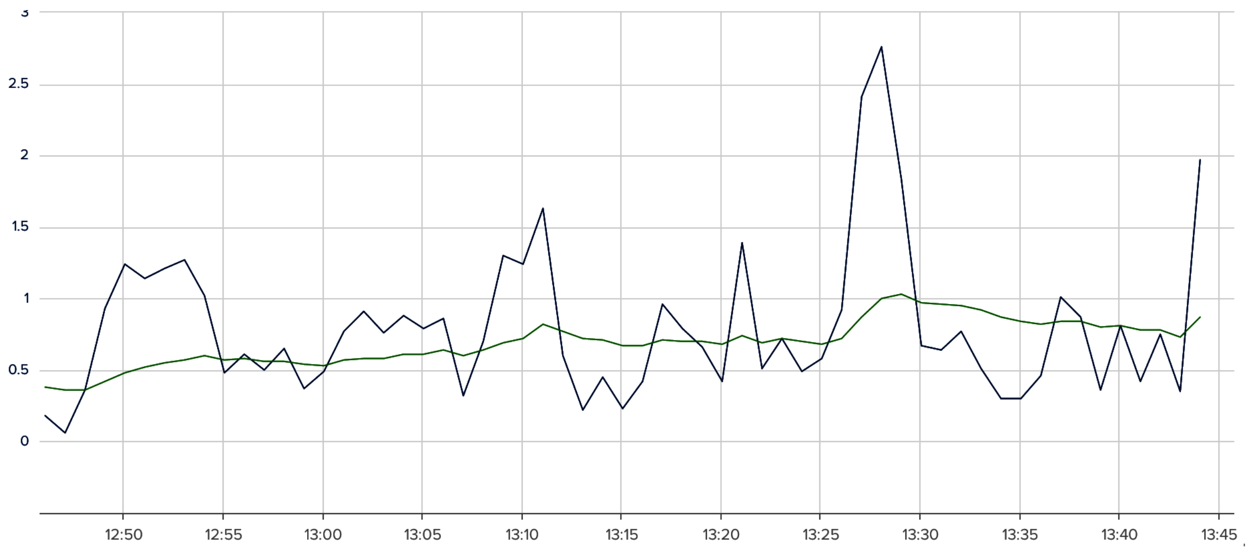


Figure 1: The effect of consolidating individual data points (the spiky line) using the arithmetic mean (the smooth line)

them, but longer-term buckets are smaller. As the data comes in, it's passed from bucket to bucket on a set interval; first into the *now* bucket, then into the *five seconds ago* bucket, and so on. Eventually, when the data points reach the *24-hours ago* bucket, they'll find that it's too small to fit them all. So they need to elect a representative to continue on for them, and so a means of carrying out this election must be chosen (this is not at all how they actually work internally, but it's a useful mental model).

As a user of these databases, you'll commonly need to configure a storage layout like the one laid out above, which, for example, stores raw measurements for the first 24 hours, then keeps one consolidated data point for each hour for the next two weeks, and then keeps one data point for every five hours, for six months, and etc. This summarization is a critically important piece of every time-series database. In a practical sense, it's what makes storing time-series data possible.

The databases that do automatic data summarization also expect you to control the method they use to consolidate the individual data points into summarized data points. Usually called the "summarization function" or "consolidation function," this is the means by which the database will decide who keeps going when the buckets get too small. You commonly need to configure this when you first create the datastore, and once set, it cannot be changed. This is dangerous, because your choice of consolidation function has a dramatic impact on the quality of your stored measurements over time, and although computing the arithmetic mean (AM) of all the data points in a period is a terribly destructive way to accomplish this, it's also by far the most commonly used consolidation function.

Averages Produce Below-Average Results

Using AM in this context is bad for two reasons. First, averages are horribly lossy. In the graph in Figure 1, for example, I've plotted the same data twice. The spiky line is the raw plot, while the smooth line is a five-minute average of the same data.

Second, averages are not distributive, which is to say, you start to get mathematically incorrect answers when you take the average of already averaged data. Both of these effects are detrimental in the context of monitoring computery things, because they have a tendency to smooth the data, when the peaks and valleys are often what we're really interested in.

Every time you create an RRD [2] with an RRA set to AVERAGE, or fail to modify the default storage-schemas.conf in Whisper [3], you're employing AM to consolidate your data points over time. These effects corrupt your data whenever you scale a graph outside the raw window or call a function that includes already averaged data.

Yes, even if your raw-window is 24 hours and your graph is displaying 24.5 hours, the entire data set you're looking at is averaged. If your raw-window is 24 hours, and you're calling a function to compare last week's data to this week's data, your entire data set has been averaged.

Worst of all, if your raw-window is 24 hours, and you're doing something like pulling a week's worth of data and running a function on it to depict it as thingies per hour instead of its native resolution (like for the marketing team or whatever), then you're looking at the average of already averaged data (once averaged for the rollup consolidation, and then again in the function to re-summarize it at a different scale). What you're seeing in this case is almost certainly mathematically incorrect.

To be sure, sometimes using the arithmetic mean is the best all around option, but if we all took a moment to fully understand the storage layer, and think about what we're measuring on a per-metric basis before we committed to the consolidation function, I think we'd pretty commonly choose one of the alternatives.

When I check the weather at wunderground.com, I don't get the average temperature for the day because that would be meaningless and silly. Instead, I get the max and min temperature for the day, and usually, because I'm a Texan, the max is the only value I care about.

Likewise, if I'm measuring 95th percentile inter-service latency, I want the *max*, which is an alternative consolidation function to average that drops all values in the period except the largest. This way, I preserve an accurate representation of the maximum 95th percentile latency value for that hour, or day, or week. In fact, in this example (like many others), the older the data gets, the more irrelevant the average becomes (and the more relevant the max).

Many of my day-to-day metrics are incrementor counters. That is, they're just +1s, adding up to some value that I don't actually care about, because I'm turning around and computing the derivative of that number to make it into a rate metric. So I don't even need to know the *value* of these metrics (because their value is always "1"), I really only need to know how many of them there are. For these, a consolidation function that just counts the number of measurements in each interval equates to lossless data compression.

Amazon.com shows me the average customer review score on every item I look at, but they can also give me a histogram of that data. Unfortunately, there is no *sum-of-squares* consolidation function in RRDtool or Whisper, but if there were, I could compute a statistical distribution from that value at display time.

Spread Data to the Rescue

So if it were me in the movie strapping vests to frightened extras, here would be my unreasonable demand this week: Let's store spread data in lieu of date/value tuples.

Imagine for a moment that you were building a system that needed to record and display at a one-second resolution of a metric that was being measured 400 times per second. In this example, there isn't a huge difference between just keeping the first metric that arrived in every one-second interval, or averaging all 400 together. No single measurement within the one-second is more important than any other. If the first measurement was extremely aberrant, I would probably choose to keep it over the average. The point is, even though we don't know what we're measuring, and even though we have 400 samples to average, the average of them still isn't as interesting as any single point in the set.

But it's a shame to throw away all of that wonderful data, even if you only strictly need 1/400th of it. I think most of us would like to have some idea of how it's distributed, some way of meaningfully combining those 400 measurements into something that is more significant than any single measurement alone. I think this is why it "feels" like taking the AM is the right thing to do. What if, instead of just storing a date/value tuple for this set, we stored something like this instead:

- date: What's the timestamp on this set?
- count: How many data points make up this set?
- sum: What's the sum of all data points in the set?
- min: What was the smallest value in the set?
- max: What was the largest value in the set?
- sos: What's the sum of squares for the set?

If we stored a struct like this instead of date/value, we wouldn't need to make the user choose a consolidation function when they created the datastore, because these data points self-summarize. When you need to consolidate them over a period of time, you compute the sum and sos, record the max, min, and count, and slap a new timestamp on it.

Even better, when the user wants a graph of this data, *then* you can ask them what they would like displayed. Do they want you to display the average value for the set? No problem, divide the sum by the count (this, by the way, ensures that you never average already averaged data). Do they want a min, max, sum, or count? No problem, display those things.

Notice that this struct doesn't even contain a variable to hold the original value of the measurement. That's because *value* is superseded for single measurements by sum, min, and max; all of those summarizations yield the correct value for an individual measurement (value/1 == value for averages, etc.), so you don't need to detect that case, it'll *just work* with the user-provided consolidation function at display time.

The drawback, of course, is that this struct is roughly 3x the size of a date/value tuple (assuming six floats instead of two), but I think fat data points are worth the stretch for a number of reasons. First, we could use fat data points as a better default consolidation function than arithmetic average. If the end-user wants to hard-code a consolidation function up front and gain a 3x reduction in storage requirements, that's a win for everyone, otherwise they get fat data points.

Second, some of the more modern data stores, like OpenTSDB, are eschewing consolidation entirely by making metrics collection a big data problem. I think fat data points fit very well between classic time/value stores like RRDtool and something like OpenTSDB that's going to require Hadoop infrastructure.

Finally, the future of metrics persistence is in purpose-specific data-handling layers built atop general-purpose databases like Cassandra, LMDB, and LevelDB. Graphite is moving in this direction with the Cyanite [4] project, and InfluxDB [5] was designed that way from the get-go. This trend is largely driven by the requirement to horizontally scale the persistence layer, and with that in place, the price of using fat data points is vastly reduced.

So let's all adopt fat data points before something happens to the imaginary hostages in my head. I think I speak for all of them when I say it's an easy fix that will simplify your time-series persistence layer while helping you preserve the integrity of your time series data.

Take it easy.

References

[1] Metrics 2.0: http://metrics20.org/.

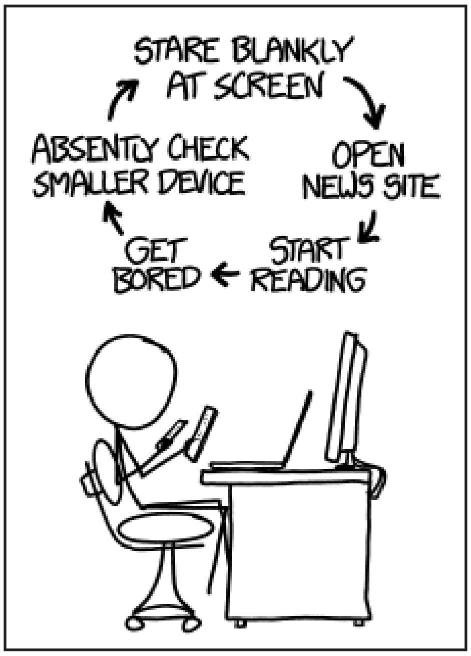
[2] RRDtool: http://oss.oetiker.ch/rrdtool/.

[3] Graphite, Whisper: graphite.wikidot.com/whisper.

[4] Cyanite: https://github.com/pyr/cyanite.

[5] InfluxDB: http://influxdb.com/docs/v0.8/advanced_topics/sharding_and_storage.html.

XKCD



xkcd.com