Logging Reports and Dashboards

DAVID LANG

David Lang is a Site Reliability Engineer at Google. He spent more than a decade at Intuit working in the Security Department for the Banking Division. He was introduced to Linux in 1993 and has been making his living with Linux since 1996. He is an Amateur Extra Class Radio Operator and served on the communications staff of the Civil Air Patrol California Wing, where his duties included managing the statewide digital wireless network. He was awarded the 2012 Chuck Yerkes award for his participation on various open source mailing lists.
david@lang.hm

Once you have set up a system to gather your logs, are able to filter and route the logs, and are alerted to the contents of the log messages [1], the next step is to figure out how to mine the logs for useful information to help you understand what your systems are doing and be proactive in dealing with problems. In this article, I will present strategies you can use to generate reports and dashboards from your logs as efficiently as possible.

Problems to Overcome

Before going into details on how to best generate reports, let's first examine the problems that you are going to be facing in a large environment.

High Log Volume Results in Reports that Take a Long Time to Generate

In an active network, generating anywhere from hundreds of GB to several TB worth of logs per day is easy. Doing anything with this much data is expensive, both in CPU time and, most noticeably, in the disk I/O required to read the data from disk in order to generate a report. The volume of data that you are dealing with is large enough that you are not going to have a machine with enough memory to cache all the logs for a day, let alone for reports covering a longer time frame.

Ending up with a situation in which it takes longer to produce your report than the period the report is supposed to cover is also easy. A system that requires 25 hours to produce a daily report leaves only the weekend to catch up—that is, if your weekend traffic is light enough. The stock response is that this is a "Big Data" problem; throw the data into a noSQL datastore and then query that datastore. This doesn't actually solve the problem, however, it just pushes out the wall that you will be running in to a bit. There are easier and simpler ways to deal with the volume issue.

Dashboards

A dashboard is a screen (usually in a browser) that is intended to give you an at-a-glance summary of your system, usually with graphs, dials, and other graphical elements to present the data. Dashboards frequently, but not always, have drill-down capabilities, allowing you to get more information about a particular element being displayed. This is the type of thing that managers love and put on large screens for everyone to see. Properly used, they are a wonderful tool for providing an overview of the health of your system, but improperly implemented, they can be a huge performance headache. And if the performance is bad enough, dashboards can end up misleading people working on the systems, reporting the health of your system sometime in the past.

Dashboards take a hard problem, resource issues, and make it even worse. Dashboards are best thought of as predefined reports that are run repeatedly, by several people at once.

The most common problem is that these different people are not asking for the exact same report. If an element of a dashboard is reporting how many hits your Web server has had over the past five minutes and you have 20 people viewing the dashboard, you will produce 20 different sets of results because no two people have started the report generation at exactly

the same time (one person is looking at the data from 9:20:00–9:25:00, the next is looking at 9:20:10–9:25:10, etc.). This can become a catastrophic performance problem if the data that needs to be retrieved to produce these reports (the working set) is larger than the RAM that your reporting system has available to cache the data, as each report will need to retrieve the raw data from disk separately.

The next biggest problem with dashboards is that, because they display their data graphically, putting a lot of information on a page is easy, but each item is generated independently of every other item. This means that if you have one dial that shows the total number of hits to your Web servers, and another that shows the number of dynamic pages being accessed, they will each go through all your Web server logs separately for the reporting period to get their results.

And, finally, dashboards frequently refresh faster than the length of the time on which they are reporting. So a dashboard reporting how many hits your Web servers have had over the past five minutes, but refreshing once per minute, will count each minute five times (once in each of five different refreshes until the data has aged enough not to be relevant). Because different elements may show data covering different time frames, this cannot be addressed by just changing the refresh time.

I have seen dashboards created that refresh every five minutes, have 10 dials, graphs, or tables on them, with each item covering logs for a 24-hour period and summarizing hundreds of millions of log events. Each item alone is a terrible resource hog, and when combined into a single screen and refreshed together, they can crush even large farms of servers. This is why the noSQL datastore is not the full solution; it will let you throw more hardware at the reporting problem, but inefficient algorithms can outrun Moore's Law no matter what your budget.

Ad Hoc vs. Pre-Planned Log Reporting

Ad hoc reports look for things that you did not think of ahead of time, and pre-planned reports cover what you know you are going to need, and can therefore plan for ahead of time. Most of the strategies in this article can only be applied to pre-planned reports.

Ad Hoc Reports

Ad hoc reports are the sort of thing that members of your security department are going to want to do frequently. They get a report of a problem with a given account, and then want to look at all the activity that happened on that account in the suspected time frame. They will then want to do further investigation to see what other activity happened from the IP addresses used to access that account (frequently over a larger time frame), and then are likely to want to look at activity on other accounts that those IP addresses accessed. Like tugging on a piece of yarn in a sweater, this activity can widen and unravel lots of interesting things.

Ad hoc reports also are commonly used during troubleshooting. You start off looking for all logs relevant to the place you see a problem, look for logs related to that place on other systems, and run similar reports for a time frame when you didn't have a problem to see what looks different.

Unfortunately, the only way to optimize ad hoc reports is to try to segment the logs into categories that match the likely ad hoc reports you will need to generate, partition them by time so that you don't have to look at logs outside of the required time frame, and try to make searching through the logs as efficient as possible.

The simple approach to this is to split the logs by category (so that your firewall logs are separate from your Web server logs, for example), and then rotate the log files every minute. This gives you a reasonable base to start from to grep through the logs and find things you didn't plan. Make sure you keep a copy of the logs that isn't split by category; although log events can and will get reordered a bit as they are delivered, the order they arrive in is the best approximation that you will have of the order in which they are generated, and sometimes you need to see what happened across wildly different systems.

Your archive analysis farm is a good place to do this. Log everything to one file and then have a series of filters in rsyslog match a particular type of log event, usually by program name [2]. You may want to have more sophisticated filters, especially ones that use metadata that you've added, so that your production, DR, QA, and development logs are separated from each other.

Ad hoc reports are where the Big Data approach to log storage can be a wonderful win. If you can have your logs in some sort of structured storage with full-text indexing (such as Splunk, Elasticsearch, Hadoop, etc.), you can run queries against the logs much more rapidly than you can with grep against flat files; however, these Big Data approaches tend to be very resource hungry (and, therefore, expensive), and although they are absolutely wonderful for ad hoc reports, using them for reports that you know about ahead of time is far more expensive (and can end up being significantly slower) than taking other approaches.

Pre-Planned Reports

The solution to the problems of generating pre-planned reports efficiently is easy to articulate, but much harder to implement.

The Golden Rule of Reporting: Never process a log event more than once.

This is an ideal to strive for, but you need to recognize that you will never achieve this in practice. As a result, you need to look carefully at the costs involved and work to minimize the overall expense of generating the report.

Because you only want to examine a given log message once, the Big Data approach to logs is not appropriate. If you use Splunk,

Elasticsearch, or Hadoop to produce your reports, you end up sending multiple queries for the same logs or types of logs, retrieving them, and generating one report item per query. In addition to the fact that your multiple queries all have to retrieve the same data, you also have the problem that the logs that you need to query to get one answer are going to be intermingled with other logs that have nothing to do with the report you are interested in, and the systems will need to read those logs to get the logs that they need to respond to the query.

So instead of throwing all the data in one place and then querying it, the idea is to split the data as early (and cheaply) as possible. This benefits you in a couple of ways.

1. The volume of logs is going to be large enough that no single process can keep up, so you want to be able to split the work across multiple processes to take advantage of the multiple CPU cores in modern systems and, if needed, multiple systems.
2. The analysis that you will need to do on each type of log is going to be very different, so it makes your report definitions much simpler if a given report only needs to deal with one type of log.

After you have split the logs by category, you can have a process go through each category. This process should not generate the reports themselves, but should instead summarize the logs to generate the data that the reports are based on. These summaries can be fed back into the logging system so that all of your analysis engines can benefit from one system summarizing the data.

If you do not have dashboards to support, running reports hourly or daily is practical; however, dashboards are valuable enough that it is worth complicating your hourly/daily reports to be able to support your dashboards efficiently, too. To do this, frequently create summaries of the logs you know you are going to be reporting on. For example, if you have a set of Web servers that are generating hundreds of millions of lines of logs per day, but you produce per-minute summaries of these logs, your reports only have to query and parse the summary data, not the raw data. Because the data is per-minute, dashboard reports also stop being different for different people; everyone who gets a report in a given minute will see the same results. The summary data is also much smaller, easily fitting in RAM, so you are not going to have to do much disk I/O when generating the reports.

There are two fundamental approaches to producing summary data: (1) storing the data, then summarizing it or (2) processing the data in real time and summarizing it.

STORE THE DATA, THEN SUMMARIZE IT

With this approach, you write the data someplace (as per-minute flat files or in a Big Data system), then run the summary routines against this storage.

If you use flat files, compressing them is a good idea. Using gzip, I find that it's faster to retrieve the compressed data off of disk and then uncompress it than it is to retrieve the uncompressed data off of disk. This is because (1) system RAM can hold a lot more data in its disk cache when it's compressed, so uncompressing something that the system already has in RAM is more likely than needing to retrieve the data from disk; and (2) CPU power is relatively cheap, so any system that has lots of RAM and a high performance disk subsystem usually has extra CPU power available.

Using flat files, you can have your summary routine make one pass through the data for a time frame and produce all the different stats that you are interested in for that time frame.

If you use a Big Data system, you can schedule queries to perform the various queries against the datastore to produce the results that you need. This is far more expensive because each query will be run independently from the others, requiring the data be accessed multiple times, but if you are doing this every minute against the last minute's data, the data you are querying should all be in RAM, so you at least avoid the expensive disk I/O. In any case, this is far more efficient than having each report issue independent queries against all the logs for the time frame the report is interested in.

Note that if you are using a Big Data system, you are paying (in license costs with Splunk, and in processing overhead and hardware for all systems) for the volume of all the log events, even if you end up only querying the summary data. In most cases you are probably better off summarizing external to your Big Data system and only putting the results into that system.

PROCESS DATA IN REAL-TIME, THEN SUMMARIZE IT

Instead of storing the data and then querying it, you can have rsyslog deliver the logs in real-time to your summary routines, have them parse and count the logs as they arrive, and then dump out the summary data periodically.

With this approach, the ability of rsyslog to normalize the logs with mmnormalize should be looked at carefully. This module lets you define log patterns and extract variables from those patterns. Having rsyslog dump out the data in a nicely structured and easily parsed format for your summary scripts to deal with, and might end up being far more efficient than parsing the raw formats in your summary scripts.

The best way to do this sort of summary is to have rsyslog run your program and deliver the logs directly through stdin. The rsyslog configuration for this looks like:

```
Module (load="omprog")
action(type="omprog" binary="/pathto/omprog.py
--parm1=\"value 1\" --parm2=value2" template="RSYSLOG_
TraditionalFileFormat")
```

Your program needs to exit when stdin gets closed, otherwise you will end up with a copy of it running after rsyslog restarts.

Note that if you use the old config format, you cannot have any spaces in the command line, so you will probably need to use an external script to start your program. Because you only need to do this on your analysis farms, you can be running a current version that supports the new syntax.

If you write your own summary script, you must have some method of having your script output its data on schedule. This can be as "simple" as having a cron job send it a signal and having a signal handler dump the data out and reset counters; however, you don't have to write this yourself. Simple Event Correlator (SEC) works well for this task and includes the ability to do things at specific times [3].

For example, the following SEC config file looks for Cisco ASA http log entries; creates a log entry containing total HTTP requests, number of servers accessed, and number of URLs accessed; then creates one file containing all the URLs accessed (and how many times they were accessed) and a second file for the servers accessed:

```
## On startup, zero the counters
type=Single
ptype=RegExp
pattern=(SEC_STARTUP|SEC_RESTART)
context=SEC_INTERNAL_EVENT
desc=Init counters with 0
action=eval %o %counters=()

type=Single
ptype=SubStr
pattern=%ASA-5-304001 \S Accessed URL ([^/])([^?^ ]+)
desc=gather most frequently accessed URLs
action=eval ( $counter{urls}{%1%2}++;
$counter{httpconnections}++; $counter{servers}{%1}++)

## output summary data and clear stats every minute
type=Calendar
time=* * * * *
desc=output summary data
context=!SEC_INTERNAL_EVENT
action=eval %a (scaler keys $counter{urls}); \
   eval %b (scaler keys $counter{servers}); \
  udgram /dev/log <30>summarydata: \
  CiscoLogCount=$counter{CiscoLogCount} \
  HttpConnectionCount=counter{httpconnections} \
  URLsAccessed=%a ServersAccessed=%b ;\
eval %o ( \
  open(output,">/var/log/urlcount"); \
  while (($key,$value) = each %counter{urls}) { \
    print "$key=$value\n"; \
  }; \
  close(output); \
  open(output,">/var/log/servercount"); \
  while (($key,$value) = each %counter{servers}) { \
    print "$key=$value\n"; \
  }; \
  close(output);\
  %counters=(); \
)
```

Using the Summary Data

Dashboards

If you use this summary data to drive the dials and graphs for your dashboards, you can cheaply create the dials and graphs, so when a lot of people want to look at the dashboard, it won't take your system down.

Reports

You should create reports for people using this data. Instead of creating reports structured around particular data sets, you should create reports structured around the needs of the user of that particular report, because aggregating the summary data, making calculations using that data, and inserting the data into a report is cheap.

Alerting

One obvious thing you can do is have a tool like SEC alert you if these numbers cross a given threshold. There are limits to how useful this is, however; numbers that might worry you at 2 a.m. on Sunday because they are so high that they indicate something is wrong or you are under attack, may be numbers that you would also want to be alerted to in prime-time on Monday morning because they are so low that they indicate that something is broken and you aren't serving your users. Such alerting is useful, but in practice is limited to notifying you when you are exceeding capacity.

Aberrant Behavior Detection

The round-robin database tool (RRDtool) [4] not only makes producing a wide range of time-based graphs to display the summary data you have generated easy, but it also has the interesting ability to take the historic data that you feed it, predict a range in which new data should fall, and flag when the new data is outside of this range. This uses the Holt-Winters Time Series Forecasting Algorithm to predict what the next value should

be, and allows you to calculate confidence limits from this so that you can do things like alert if the measured value is more than two standard deviations away from what is expected. This algorithm will detect repeated patterns in your data so that it not only matches your daily usage pattern variation but, once it has about 10 cycles worth of data, can detect the difference between weekdays and weekends, for example, while still accounting for a continuing increase in use over time. The details of how to do this are out of scope for this article, but there is a good writeup at http://cricket.sourceforge.net/aberrant/rrd_hw.htm.

Artificial Ignorance

In addition to counting how many times something happens, another useful report to have is an "unknown log report" of the type produced by the "artificial ignorance" approach described by Marcus Ranum [5]. This consists of deliberately filtering out log entries that you understand, then prioritizing the remaining log entries based on what shows up the most:

1. Filter any log entries that you want to report on to a process to generate the appropriate report for that type of log entry.
2. In what's left, filter out any log entries that you know are not important, but count them and report this count. If the number of times that an insignificant event happens changes drastically, this may be significant.
3. Take what's left and sort the logs based on their contents, and produce a report that shows the most common logs.

A person can then look at this report and quickly spot strange things that have happened.

Taken to the extreme, you can tune your artificial ignorance report to the point that you have no logs in it at all. At that point, anything that shows up in the report becomes significant.

Getting there is a lot of work, and you quickly reach the point of diminishing returns. Even on a large network, surprisingly few different log entries are produced. Large networks tend to have a lot of the same thing on them, so once you identify what should be done with a given log entry, you don't care whether you have two servers producing that log entry or 2000; either way, it's handled. A few days' worth of effort filtering the log messages probably can get you down to a report that shows you events that have happened fewer than a dozen times in the first couple of pages of the report. Also, the report probably will show you some errors that are happening on your network that you were not aware of and want to fix before going a lot further.

Running separate artificial ignorance reports against each category of log messages is best. Dump all the messages that don't match your reporting rules into a file for that category and then periodically run this data through a filter along the lines of:

```
cut -c 17- |sed -e s/"port [0-9]* "/"port PORT "/g \
  -e s/\[[0-9]*\]/"[PID]"/g -e s/"pid=[0-9]*"/pid=PID/g\
  |sort |uniq -c |sort -rn >other-logs.report
```

You may find that on your network, there are some other fields that are in frequent log messages that make otherwise identical messages look different, which is what the sed statement in this filter chain is addressing.

Then take a look at the results. If you have a log message that shows up a lot (or a lot of similar log messages that show up a lot), add a rule to match them. Repeat until you can scan the entire report in a short enough time that you no longer care; you don't need to drive it all the way to empty.

Summary

Producing dashboards and reports from a high volume of logs—and doing so efficiently—is possible, but if you are not careful, you easily could find yourself with a system that is orders of magnitude larger than you would need for efficient generation, and still running into performance problems.

Split up the work, and try to make it so that no log ever needs to be examined in detail more than once, and try to limit the number of times it must go through a filter.

At this point, I have covered the basics of a full enterprise logging system. In future articles I will go into the various topics in more detail, which includes covering performance tuning of different tools. If you have specific topics on which you would like me to focus, please email Rik Farrow (rik@usenix.org) or me and let us know.

References

[1] "Enterprise Logging," *;login:*, vol. 38, no. 4, August 2013: https://www.usenix.org/publications/login/august-2013-volume-38-number-4/enterprise-logging.

[2] "Log Filtering with Rsyslog," *;login:*, vol. 38, no. 5, October 2013: https://www.usenix.org/publications/login/october-2013-volume-38-number-5/log-filtering-rsyslog.

[3] "Using SEC," *;login:*, vol. 38, no. 6, December 2013: https://www.usenix.org/publications/login/december-2013-volume-38-number-6/using-sec.

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

[5] Artificial ignorance: http://www.ranum.com/security/computer_security/papers/ai/.