10N

10n Software, LLC

Example Company, Inc.

Basic Load Testing Report

Prepared for:

Kevin Schroeder

Delivered On: July 3, 2017

Table of Contents

Introduction

This Load Test report was created for Example Company, Inc. (Customer) by 10n Software, LLC. (Consultant) This document presents the findings of the load test executed on June 8, 2017.

Engagement Type	Basic Load Test
Client	Example Company, Inc.
SOW ID	Example Company – LT062017SOW
Magento Version	Magento 2.1.2

This document was prepared by

Kevin Schroeder
Consultant
kschroeder@magiumlib.com

Executive Summary

On June 8, 2017, Consultant executed a load test on the `example.com` website. The purpose of the test was to determine if the site could process 700 orders per minute, with the intent to push the site to 800 orders per minute. CPU resource contention on the frontend servers kept us from reaching that limit. However, our findings also indicate that even if the front end CPU resource contention issue is resolved, there is still insufficient capacity in the overall system to handle 800 orders per minute due to resource contention on the database. To be sure, the database performed well during the test compared to the frontend servers. If the database performance envelope were to follow a linear trajectory (which we would expect until contention occurs) then the system, as a whole, could handle about 440 checkouts per minute. Based off of the information from New Relic, which reports that there are two machines responding to requests, it seems reasonable to conclude that about 14 front end servers could push the database server to capacity.

There are three primary issues that we discovered during testing.

The first issue is user CPU contention on the frontend servers. User CPU time is measured using the user time metrics provided by the server kernel. User CPU time is time spent executing actual PHP code. The system became saturated around 30 concurrent test threads and throughput (number of requests per second) maxed out around 20 concurrent test threads.

The second issue is database capacity. The test executed about 66 checkouts per minute. The bottleneck was the frontend servers. However, the database maxed out at 14%. From there we extrapolated that the database could handle around 470 concurrent checkouts.

The third significant issue was that, under load, the system CPU time was unusually high. Often, one expects a ratio of 8 or 9:1 between user CPU time and system CPU time. In this case, the ratio was 3:1. If the cause of that disparity can be determined and resolved, the front end capacity may increase 15% or so and reduce the final checkout page load time by 15%.

All that said, the conclusion of the load test is that there is very little by way of code or software implementation that is holding the system back, in terms of "out of the ordinary" execution. The throughput issues seem to largely be related to raw CPU capacity on the front end and database machines.

Testing Methodology

The load test consisted of 3 test runs of the full checkout process using Apache JMeter after an initial spike test, which provides a baseline of expected capacity. The first test ran for about 11 minutes and increased concurrency from 1 thread to 50 threads and then held that thread level for several minutes. The second test consisted of a 5-minute spike test to ensure that the peak throughput numbers from the first test were not skewed by average calculation. The third test was similar to the second test in that it was a peak test, but its peak was set at 75 concurrent threads. The purpose of that spike test is to validate that the detected peak held even at higher levels of concurrency.

Tests were all executed with `vmstat` running on the frontend machines and using the JMeter Response Times Over Time, Response Codes per Second, and Summary reporting tools.

A single thread correlates to a single user on the site. However, the test does not include user indecision or lag time as part of the test. Given that the purpose of this test was to determine how many checkouts the system could handle, throughput was the defining metric.

Findings

Result Summary

The intent of the site is to handle 800 successful checkouts per minute. However, as noted in the Executive Summary, the Consultant was not able to push near this level due to CPU resource contention on the frontend servers.

First Concurrency Test Run

The first test run ran for about 12 minutes using a gradual build up from 1 test thread to 50 test threads. During this test run, 654 checkouts were executed with a minimum checkout time of 13.4 seconds, a maximum checkout time of 42.6 seconds, with an average of 33.3 seconds. Throughput maxed out at 1.1 checkouts per second or about 66 per minute.

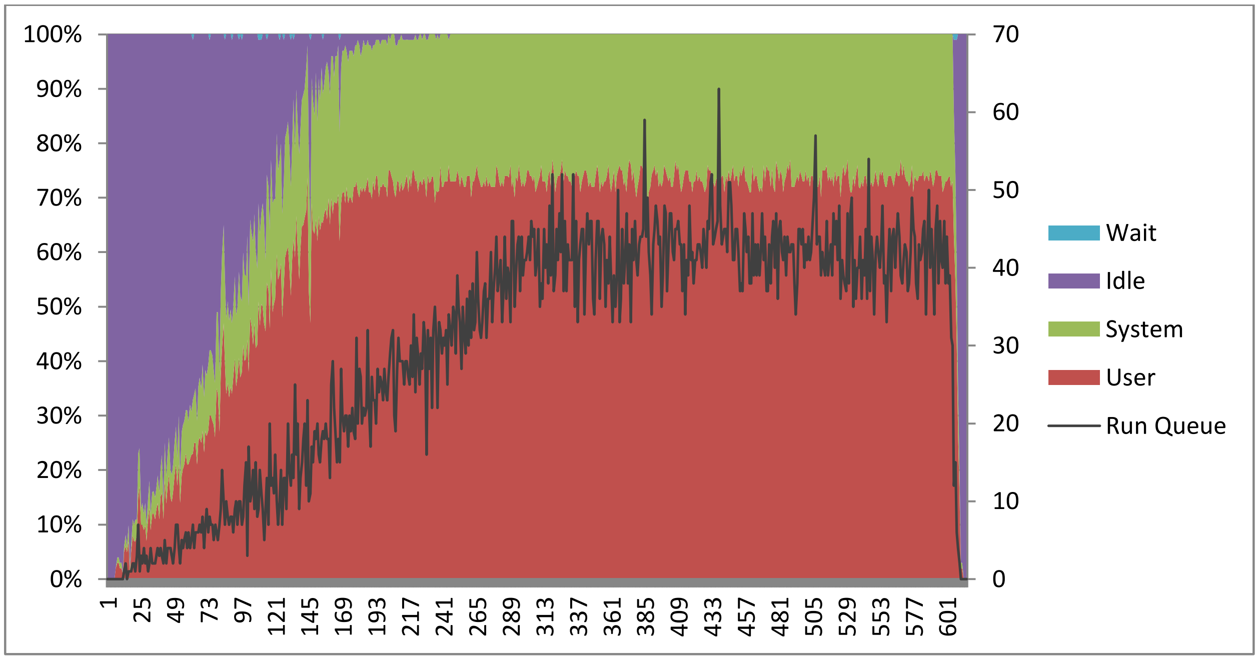


Figure 1 – CPU Resource Usage on Frontend Server

Figure 1 shows that frontend CPU usage increased fairly early in the load test. Concurrency was around 20 threads then.

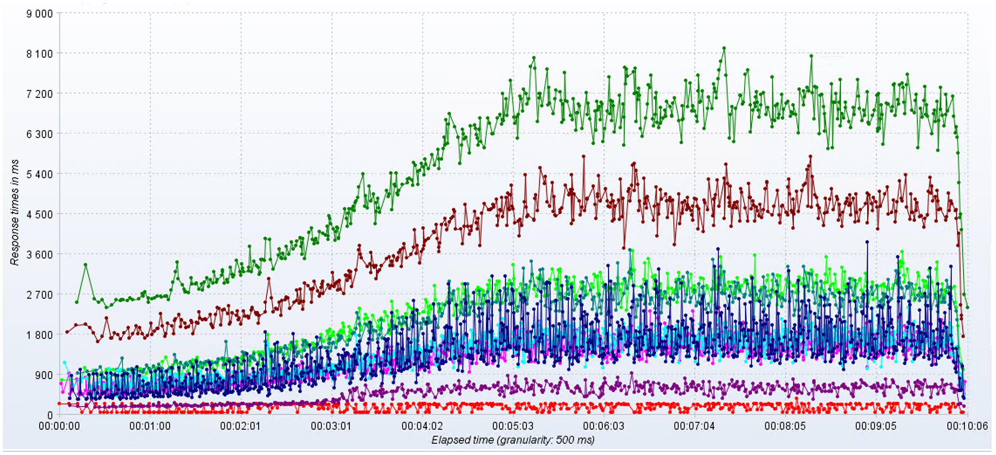


Figure 2 – Response Times over Time

Figure 2 shows that the response times stayed relatively normal for the first minute or two but then increase consistently until the test hits the concurrency plateau in the middle of the test. While the response times increase, the requests are healthy. A system under excessive load will start to show significant variability in response times as the server struggles to respond to requests.

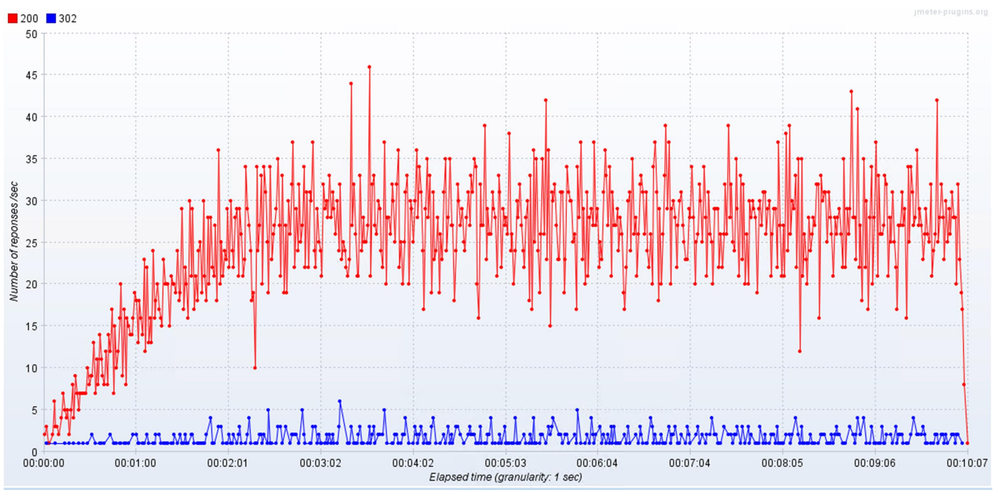


Figure 3 – Response Codes per Second

But while the system is not exhibiting failure it is not increasing throughput. Figure 3 shows us that while the concurrency continues to increase, saturation of the CPU effectively limits the throughput to around 30 requests per second.

Second Peak Test Run

No additional comments needed beyond comments on system time. The test confirms throughput numbers from the previous test.

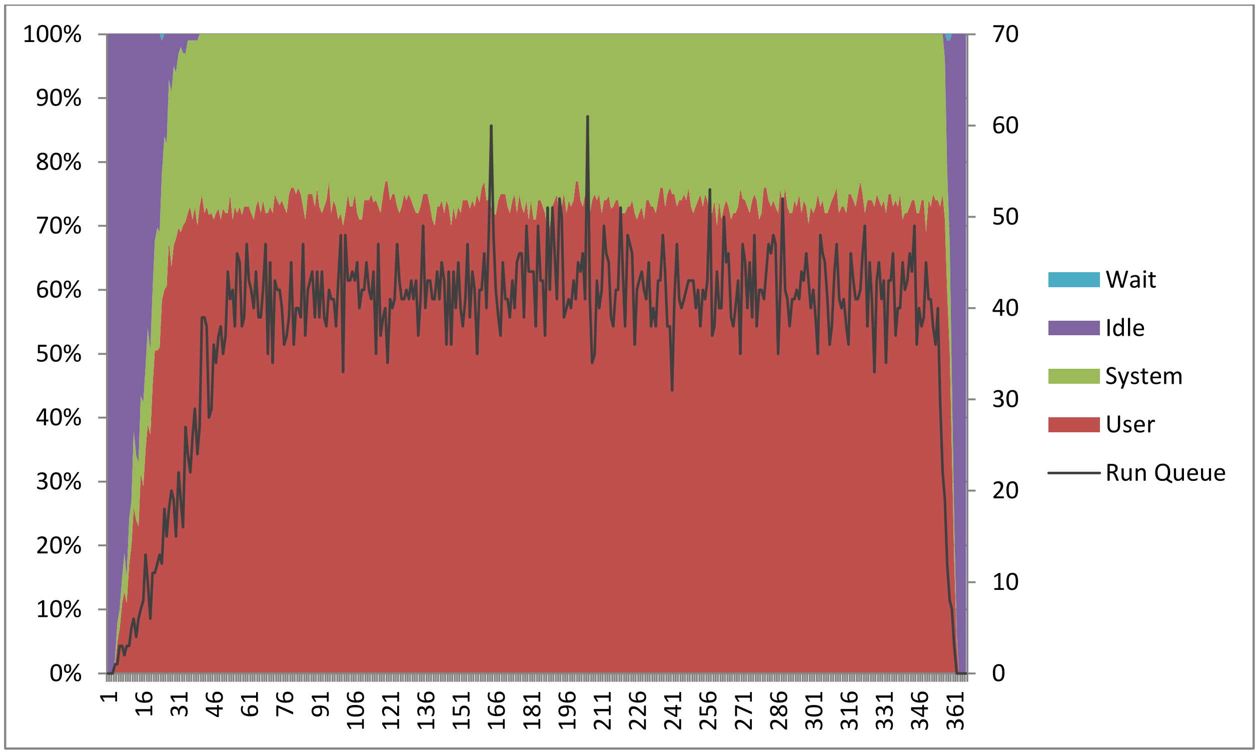


Figure 4 – Front End CPU Usage

Figure 4 shows that system CPU time takes up about 30% of all CPU time. System Time is the time that is spent doing things on behalf of the program. Some examples of this include file operations or memory management. On PHP this usually correlates to file system access. The cause for this is not directly clear and beyond the scope of the load test. But as we will show in the Code Findings section, there is a culprit.

Third Peak Test Run

The only comment necessary on the third test run is that it confirms the same data points as the first two. First, that increase concurrency did not increase throughput. Second, that System CPU time remains at about 30%. Third, as Figure 5 shows, the only thing additional concurrency accomplishes is an increased run queue.

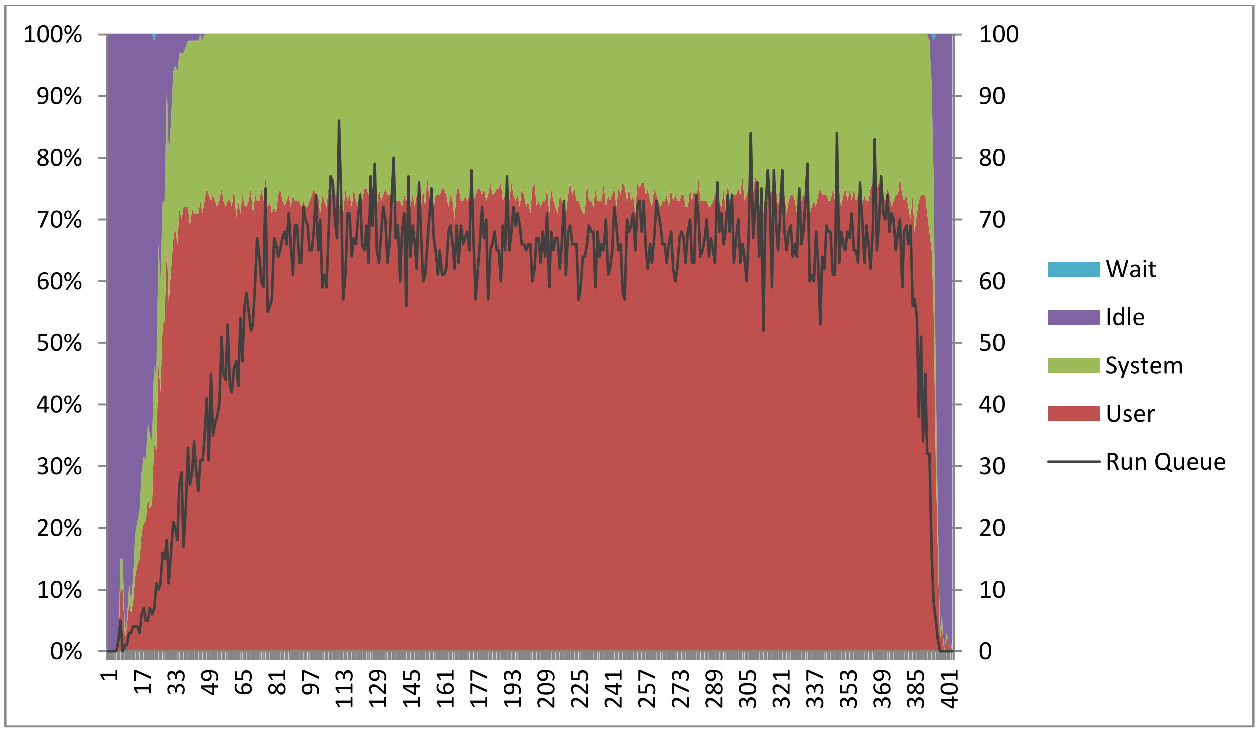
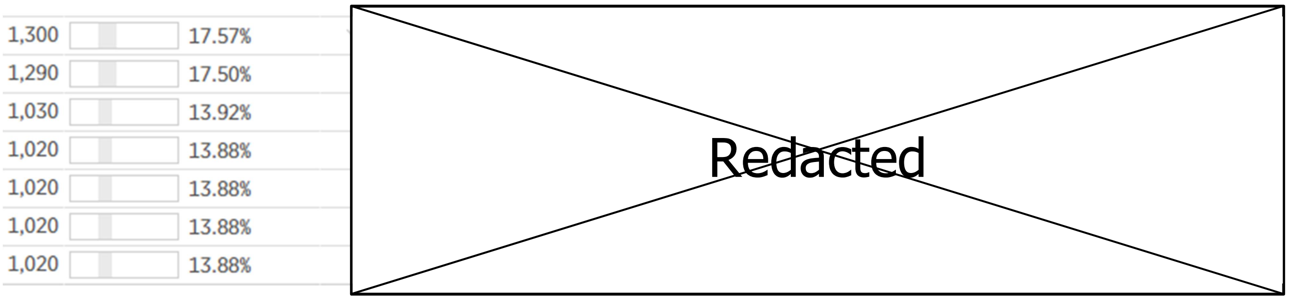


Figure 5 – CPU on Frontend Webserver

Code Findings

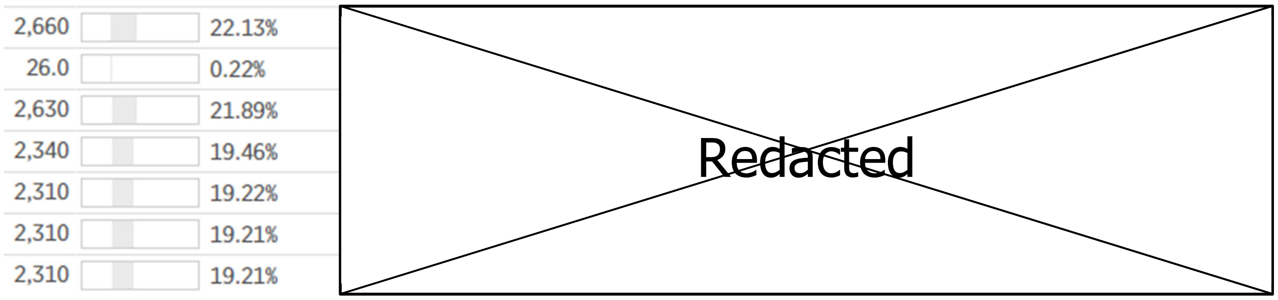
During the test, New Relic triggered a few transaction traces of the `payment` request. This request is responsible for finalizing the transaction before redirecting the customer to the success page. As we see in Figure 1, this is the request that takes the longest to execute and is also the one that increases its wall clock time the most. When we look at a transaction during the 50 concurrent thread run, we see that this API call takes about 1,300 ms (figure 6).



1,300	17.57%
1,290	17.50%
1,030	13.92%
1,020	13.88%
1,020	13.88%
1,020	13.88%
1,020	13.88%

Figure 6 – Example Module API Call during 50 Concurrent Thread Test

However, when we look at the same API call during the 75 concurrent thread run we can see that the API call takes significantly longer.



2,660	22.13%
26.0	0.22%
2,630	21.89%
2,340	19.46%
2,310	19.22%
2,310	19.21%
2,310	19.21%

Redacted

Figure 7 – Example Module API Call during 75 Concurrent Thread Test

Figure 7 shows that the response time doubles. The code in Figures 6 and 7 calls the Composer component which loads and validates several JSON files and instantiates several plugins. The offending code is in `Example\Module\Helper\Config::getClientName()`. This code calls the Magento metadata manager who depends on Composer to calculate its version. Caching the value calculated in `getClientName()` could reduce the response time on the payment page more than one second.

Given that Composer is reading many files during that operation, it seems reasonable to conclude that at least some of the System time is accounted for by this call.

Conclusion

The conclusion of the load test is that except the Example Module issue the primary problem, much of the performance issues are due simply to an insufficient level of CPU processing power available on the front end and in the database. It may be that the database will exhibit more significant issues as the test pushes it to capacity, but there are currently an insufficient number of frontend machines to test that condition. There are two steps in rectifying the throughput issue. The first is to increase the number of front-end machines to a level where the database can be pushed to near 100% utilization. The second step is to redo the load test with the higher concurrency. During that test, a database user with `PROCESS` privileges should be monitoring the database using `SHOW INNODB ENGINE STATUS` while watching New Relic to monitor how the database responds to PHP in a saturation condition.