Exploratory Performance Testing

Have you ever found yourself in a situation where you are are organizing functional testing of a product, when suddenly, somebody comes up and asks about performance? And before you even had a chance to explain that you haven't even started thinking about non-functional requirements, they have launched into a long discussion about how important this product is, how detailed the tests need to be, and are talking about hundred thousand users simulation with correct distribution of use cases... Before you know it, you are in a meeting, requirements are being thrown out like snow-balls, and every person who is even remotely associated with the product has strong opinions on what you should do, how it should be done, which tools should be used, how should the results be reported and so on. And, if you have ever done any performance work, you realize that things these people are talking about will take months, if not longer, while they have made it very clear they expect you to provide them with the results in a few weeks.

Welcome to my world. I have been doing performance testing for Google since January 2005, and have dealt with more Google services than any other person at the company. (I do not really have a proof for this, and it is certainly possible that there is a person out there who has secretly tested more than 50 products, but I doubt it.) So, here is my story - what have I learned, what works, what does not work, and why I am now comfortable doing things that would have probably caused me to break in cold sweat ten years ago.

What Does Not Work

This was probably the most difficult lesson for me to learn, and I still have extremely hard time accepting it. I wish it were not true. I wish I were imagining things. But, rule number one, if you find yourself in the meeting I described above is: Do not try to use logic and reasoning in your discussion. Actually, do not discuss anything at all. Realize that nothing said is coming from experience, instead, people are talking about their hopes, their ideas, their emotions - and you are not required to do anything other than acknowledge this. Wait for the conversation to die down, and once it is done, acknowledge everything that was said as if it made sense, tell the team that you will definitely immediately start working on what they have suggested (using their scenarios, their tools, their ideas, etc.) however, you suspect that work will take a very long time. So, in parallel, you will quickly throw together a couple of performance tests to get a feel for the system, to help better design detailed scripts and guide the overall performance testing process.

By this point in time, you probably know who is the development lead and who are the developers working on most critical operations. You probably understand what are the critical functions, and even though you may not know everything as accurately as you would like, you can do an amazing amount of useful performance work, in a short time, as long as you do not get involved in long discussions about what and how should be done.

And, that would be the lesson that was hard to learn: Do not waste time trying to explain to people why they are wrong. They are not thinking and analyzing the situation, the data, time-lines and what can and cannot be done. They are reacting emotionally to the wrongly perceived risks and importance of their service. Same as you will never meet a parent who will describe their child as below average, not very smart or completely spoiled, you will never find a development team that will feel the same way about their baby/project. Coming in, and pointing out that the baby is ugly, isn't very important and could be left in a corner unattended, will certainly not win you any friends. (Trust me, I have done this many times and it never led to a satisfactory resolution.)

What Works

What works, invariably, is agreeing with the development team in principle. Yes, their product is very important (to them.) Yes, their product is very complex and difficult (is there a software product that isn't?) Yes, the time-lines are very tight and will require all of your time and concentration, so, you will need the team's full cooperation and help to make progress. Get the list of names of people who should be informed on a daily basis of what is going on. Get the contact people in case you need a question answered quickly (never discuss finding problems, because, clearly, their product will have none. Instead, make the request from the perspective of your inadequacy - in case you get stuck not being able to understand what just happened, who is the person that will be available to help?)

By this point in time, you will have a list of operations/transactions the team considers important. You may have to add one or two that you feel are missing. There is no hard and fast rule here, but if there is a transaction that somebody considers important, and you can include it in your tests, do so. Try to limit your test to 10-15 transactions at most, because more than that could make for more complex tests. Remember, this does not mean you will not be able to change this later, it just sets your starting point.

You are now ready to start exploratory performance testing, and gain an understanding for how your system behaves. By its nature, exploratory testing doesn't follow a prescribed algorithm - you tend to modify your future actions based on the results you have just obtained. However, there is an overall goal that you are trying to accomplish, and your actions will be chosen with that goal in mind.

End Goal

When I finish performance tests, I would like to give my team the two graphs that show overall behavior of the system as a function of increased load. Additionally, I will frequently provide a list of performance data for every individual transaction in a tabular format - I find that people quickly catch on to the concerns I have if the data is logically graphed, and also presented in a table. So, what I am hoping to accomplish is something like the graphs shown in Figures 1 and 2:

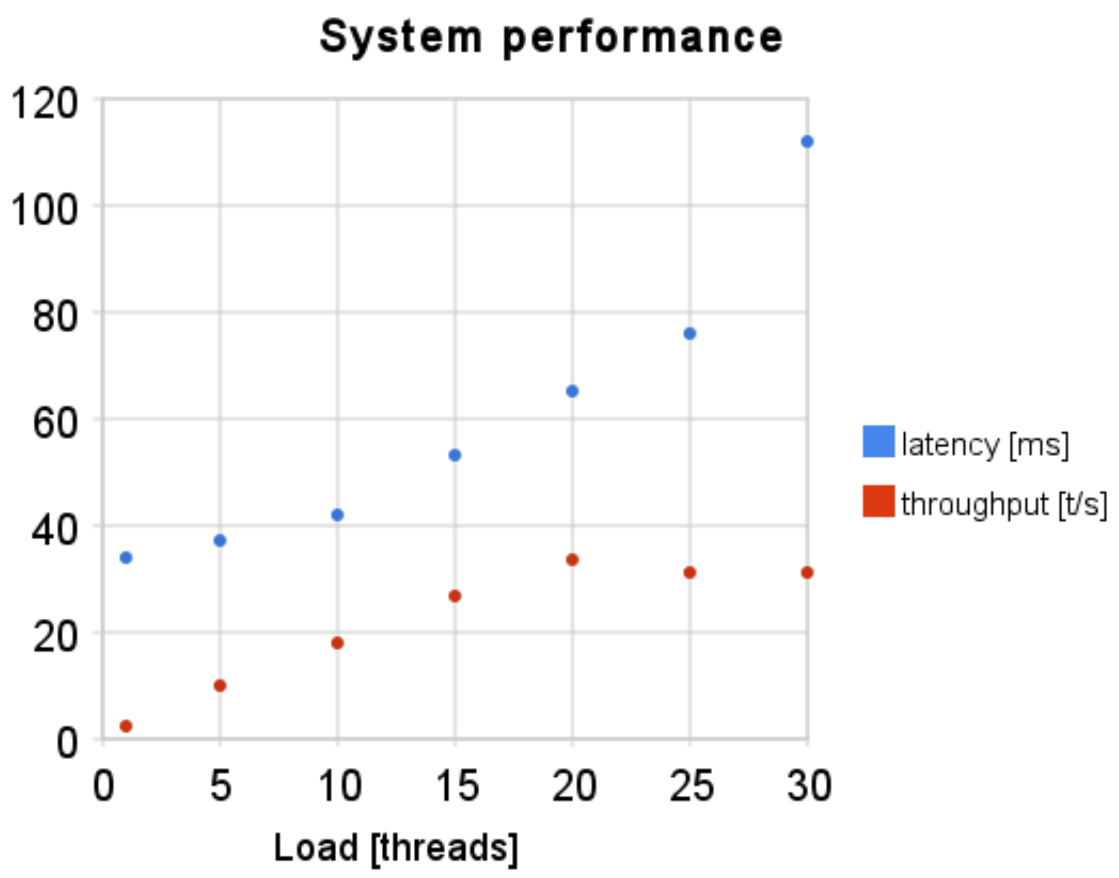


Figure 1. System performance graph

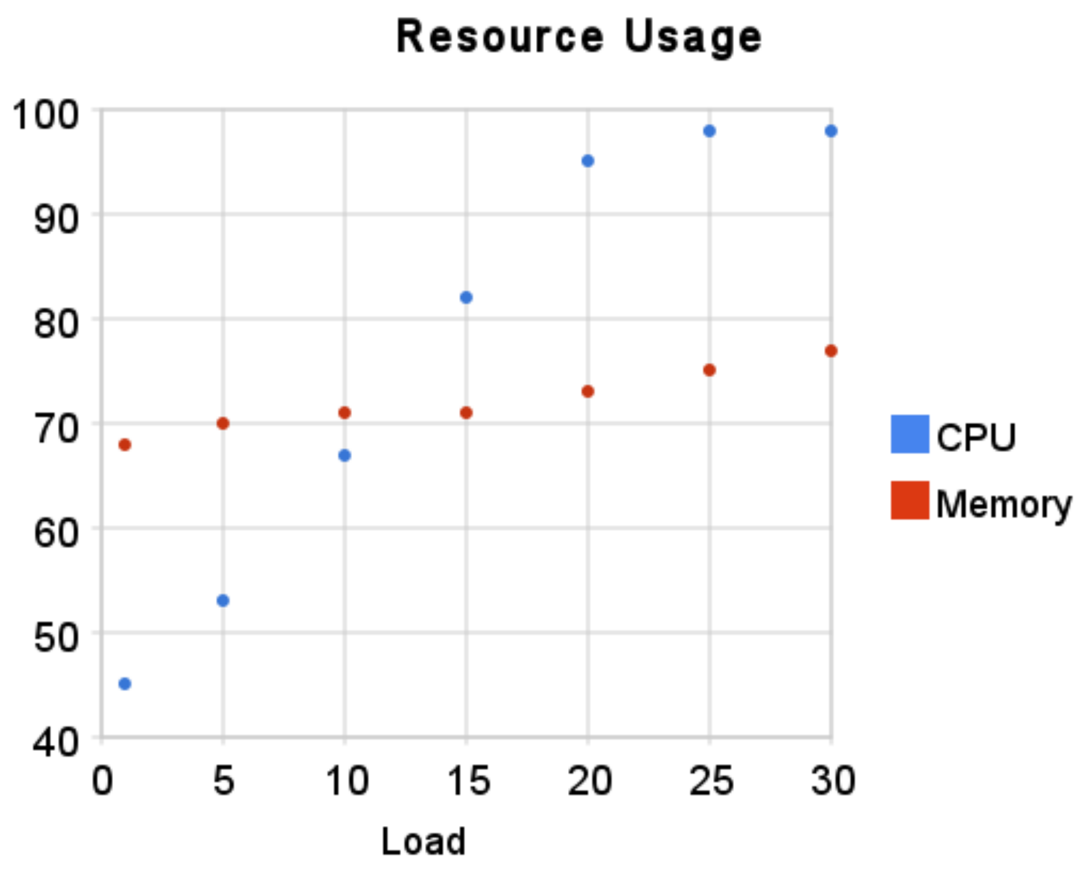


Figure 2. System resource consumption graph

Note: the graphs shown above are not from any particular product or service, instead, they are created by me to show a reasonable system behavior which you may encounter.

I find these graphs incredibly useful, and, in my experience, every development team I have ever worked with has felt the same. The path to obtaining the graphs, provided everything works well, is relatively simple and straight forward. The overall benefits of preforming this work, even with imperfect load, is multi-fold:

1. it is easy to visually inspect if the behavior of the system and see if it makes sense
2. it is easy to see why/where the system is limited, which allows development team to make better choices in future changes
3. it is easy to pick a good load for a benchmark
4. it is easy to describe the behavior of the system to non-technical people
5. almost any load is likely to find big functional problems: deadlocks, bottle-necks, priority inversion, etc.
6. you have a better understanding of relative costs (in terms of system resources) of different transactions

Starting Exploratory Performance Testing

I would like to illustrate my approach on a simple service which I use for demonstration purposes frequently: Google Search.

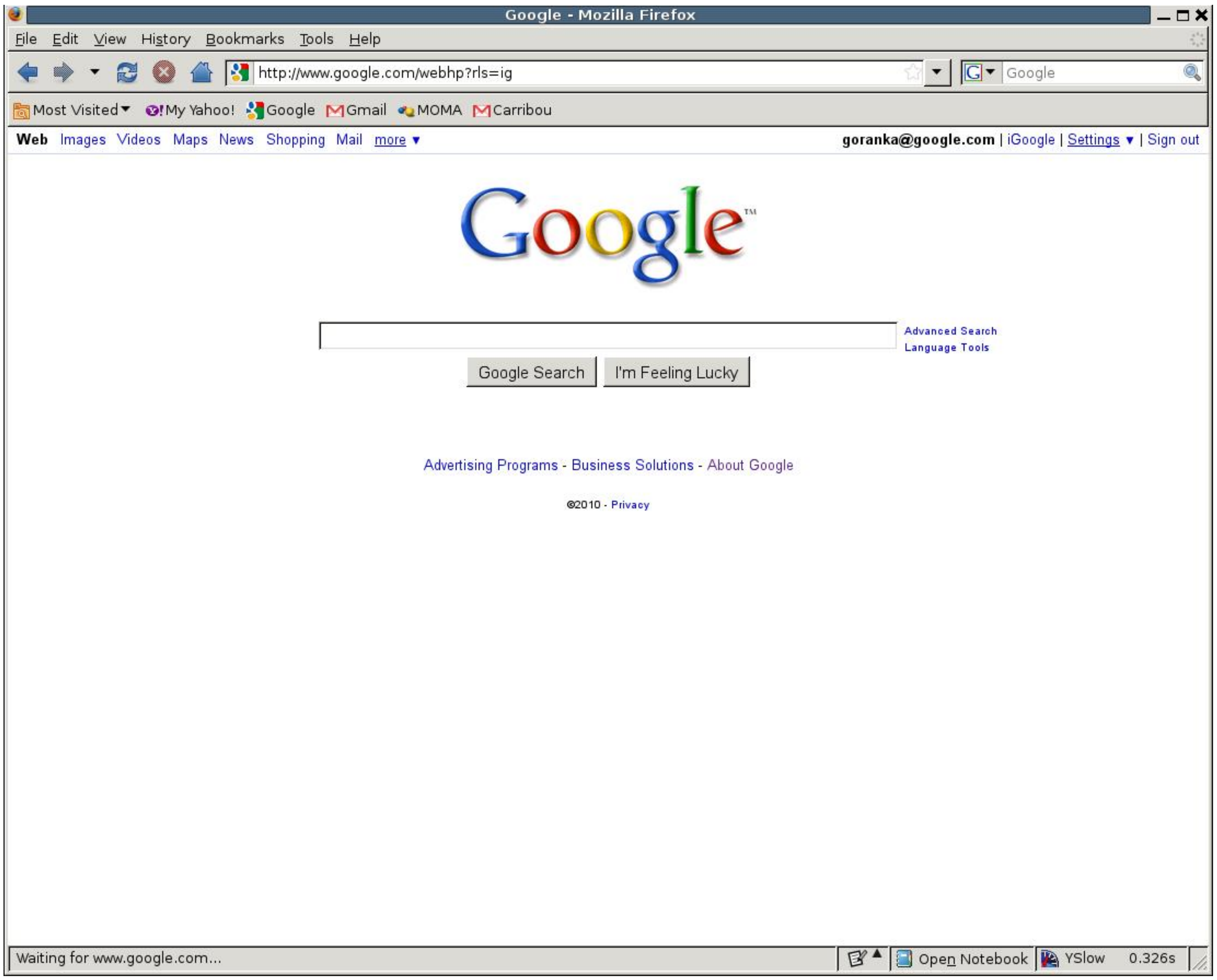

Google - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.google.com/webhp?rls=ig
Google
Most Visited My Yahoo! Google Gmail MOMA Carribou
Web Images Videos Maps News Shopping Mail more
goranka@google.com | iGoogle | Settings | Sign out
Google
Advanced Search
Language Tools
Google Search
I'm Feeling Lucky
Advertising Programs - Business Solutions - About Google
©2010 - Privacy
Waiting for www.google.com...
Open Notebook
YSlow 0.326s

Figure 3. Default Google Search page for user goranka@google.com

Let us assume that I do not know anything about the search patterns, have never worked on it, and have no logs from production to guide my work (not an unusual situation to be in.) Where to start, what to do? This is a very important crossroad which can confound the performance tester and stop any progress. Clearly, what you want to do is pick the **right** transactions, add the **right** time delays to those transactions, have the **right** distribution of users simulating exactly what will be happening in production... or do you?

There is nothing wrong with having all these things as close to the production as you can get them. But, if you have no information, you run the risk of searching for something that does not exist and wasting the time you do not have. You do not need all these things. What you do need, instead, is some load - any load will do! - and you need to **report your results correctly**.

Instead of focusing on getting the perfect performance testing load, I will look over the application and try and sort though the different categories of things a person could do on the site:

1. a user could search for a single word term
2. a user could search for a specified phrase
3. a user could search for multiple words
4. a user could perform any of the built-in calculations or conversions

5. a user could look at the different page of results, once those are presented
6. a user could leave the site at any point in time

Assuming I have absolutely no information, I would make some guesses: maybe 50% of the users will type a single word search. Another 30% will search using multiple words, while only 10% will type an exact phrase. The remaining 10% of the users will use the built-in functions. Why do I pick these numbers? No particular reasons, I need to pick something, this makes sense to me and I am perfectly happy to proceed. Somebody else would pick a different set of percentages, and that would have been perfectly fine. Think of this as making a cup of coffee on a new coffee-maker and with unknown coffee beans. You will make a guess for how much sugar/milk/cream/etc. you want. Your guess is not better or worse than the guess of a different person, using the machine right before or after you. You will end up with a cup of coffee, and if the cup is terrible, you will repeat the process an d change some of the parameters.

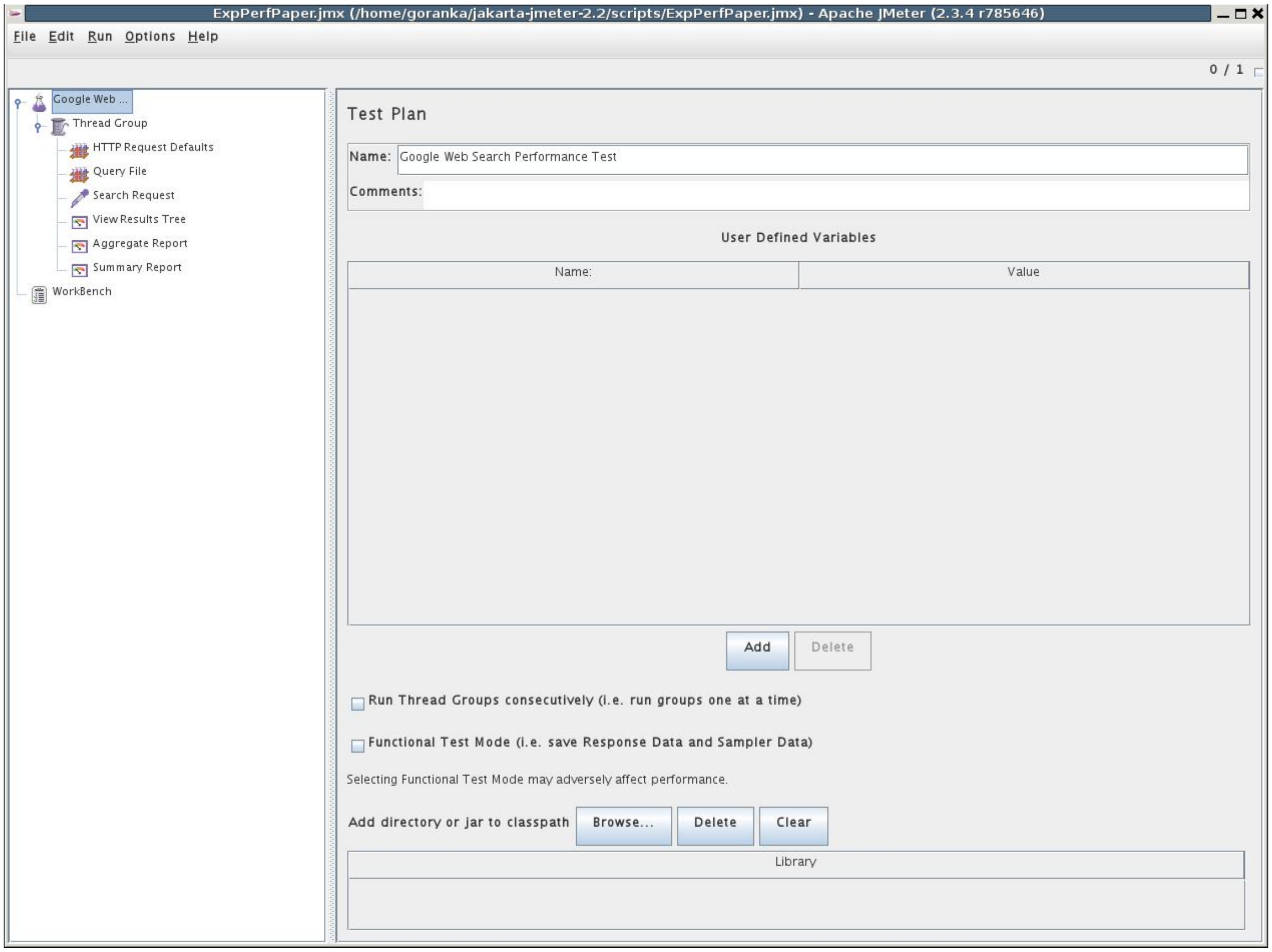
I will run by these preliminary choices by a person more familiar with the service - just to see if I am completely off. If necessary, I will adjust my guesses and proceed.

Creating the Performance Test Script

For a project like this, I would use the tool called JMeter (http://jakarta.apache.org/jmeter/). However, even in this category, it would be difficult to go wrong - there are many excellent performance testing tools provided by both vendors and open source community. Use whichever tool you are most familiar with. In addition to JMeter, I will do most of my scripting using FireFox browser and I always make sure that LiveHTTPHeaders extension is installed and enabled. Note - these are my **personal** choices and they work for me - since I do most of my work from a Linux workstation, I find them essential. However, you can find an equivalent set of tools for every other major operating system, and I would definitely suggest doing some reading on the topic and exploring if you have never done any work in the area before.

In many cases, you can use JMeter's recording ability to create your first performance testing script. While I personally tend to almost never use the recorder, when teaching JMeter, I really like the ability. Basically, you can insert JMeter as a proxy between your browser and the outside world, and it will automatically record every request the browser makes and allow you to obtain your first script. My personal preference is to use LiveHTTPHeaders and look at the traffic between the browser and the Web, and script the requests directly.

I tend to do most of my performance script development iteratively: I will script one HTTP request at the time, play it back, correct any errors and continue until all of my requests have been scripted. Both the recorder and the LiveHTTPHEaders will catch all of the AJAX requests as well, and will allow me to adjust my script accordingly.



Name: Google Web Search Performance Test

Comments:

Run Thread Groups consecutively (i.e. run groups one at a time)

Functional Test Mode (i.e. save Response Data and Sampler Data)

Selecting Functional Test Mode may adversely affect performance.

Figure 4. A very simple JMeter script

A couple of things to note about the script:

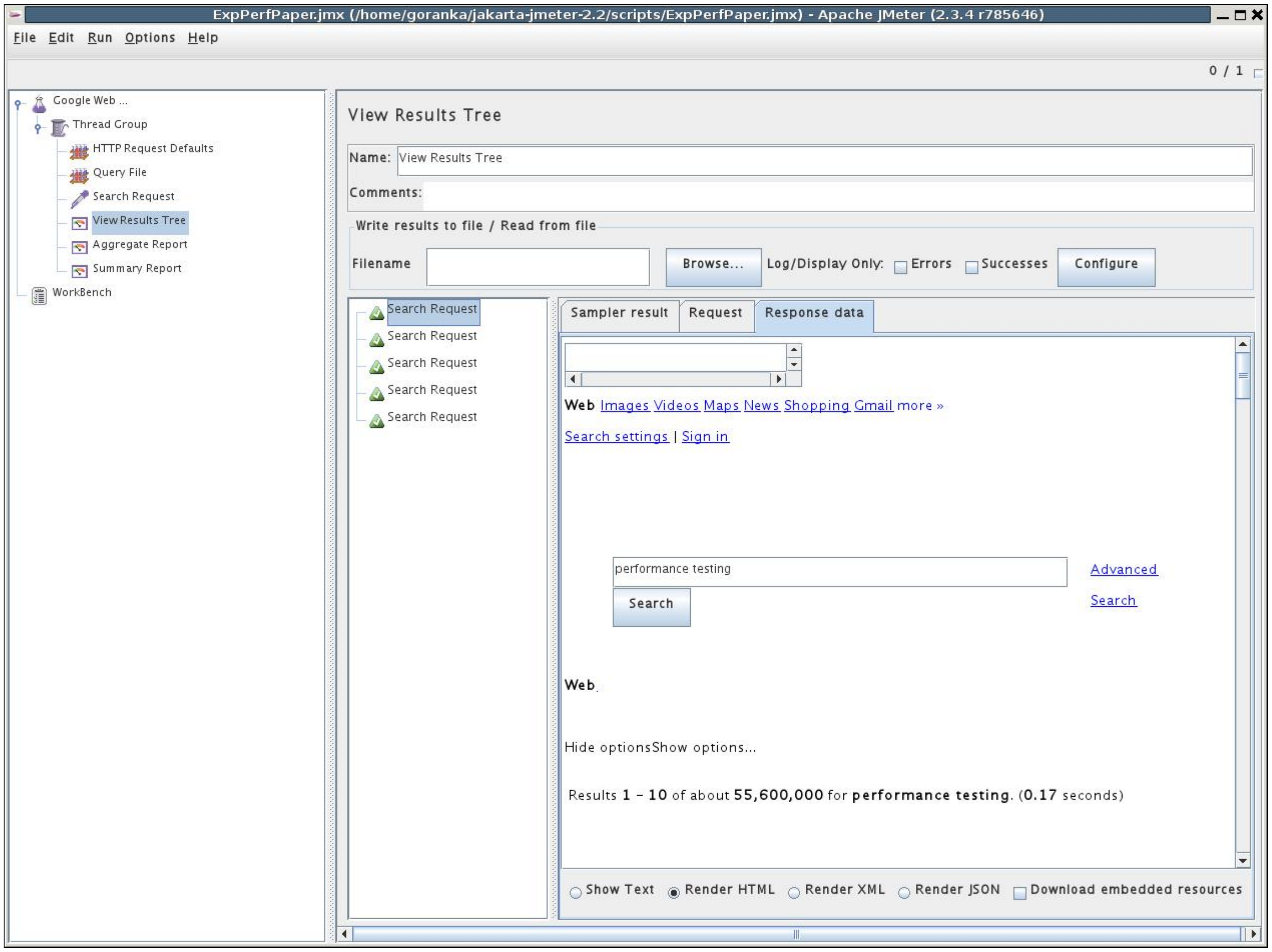
1. At this time, it has only one HTTP sampler. You can control the mix of requests by controlling the queries in the file, you could add multiple different samplers and control the throughput of each or you could have different thread groups with different number of threads. what you choose to do depends solely on your preference. In this case, to make the script very simple and fit on one page, I am controlling my query mix through the query file.
2. All of the defaults are specified in the HTTP Request Defaults, and can actually be read from the command line when starting the script. This allows for easy execution in different environments, and provides reasonable defaults.
3. There are three different listeners - View Results Tree, Aggregate Report and Summary Report. During the regular performance run, I would disable View Results Tree listener, as I tend to think of it as a perfect debugging tool. It allows me to see every bit of information exchanged between the server and the client, and it also has an option of presenting the data in HTML format. (Note: if your application is JavaScript heavy, you may have to use a lot of imagination to recognize it, however, I find this ability quite useful for quick inspection during the script development time even in that case.)

Running the Script and Collecting the Data

My first run is always with the smallest possible load - one thread. Additionally, I will set up the service monitoring, and collect all of the basic resource consumption data I may be interested in: memory usage, CPU usage, disk I/O, and anything else I may need. Depending on the operating system and the performance testing tool you have chosen, you may have to do more or less work in this area. Most vendor tools come fully equipped with the monitoring capabilities, while most open source tools require you to solve this problem on your own. I have a very strong preference for using whichever monitoring solution will be in operation once the service is deployed and in production - so in my case, I tend to use Google specific performance monitoring solutions, which will not be shown in this paper.

How long should the performance test run for? This is another one of those experience variables that different people will have different solutions for. I still have a preference for 2 hour runs, although in a lot of cases I find I can get very good information in ran as short as 10 - 15 minutes. You could say the answer depends on the level of risk you are willing to tolerate, however it should always be longer than the time it takes for the system to reach its steady state. This means that you will have to warm up your caches, pre-populate your database, and do anything else necessary to bring the system to the state you expect it to be in during production.

The results from the verification test run with a single thread repeating five times are shown in Figures 5 and 6.



Name: View Results Tree

Comments:

Filename

Log/Display Only: ☐ Errors ☐ Successes

Search Request
Search Request
Search Request
Search Request
Search Request

Web Images Videos Maps News Shopping Gmail more »

Search settings | Sign in

performance testing

Search

Advanced Search

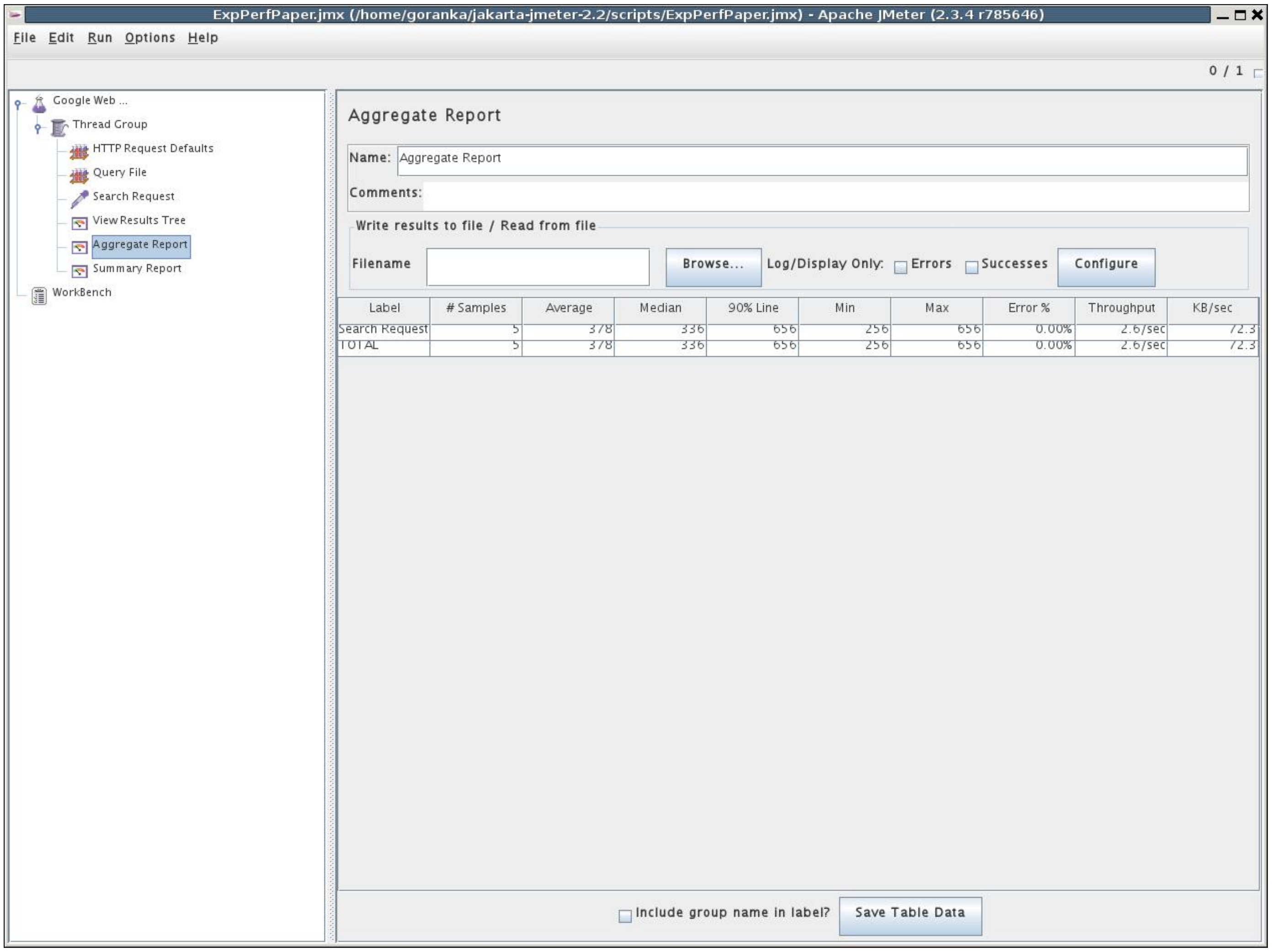
Web

Hide optionsShow options...

Results **1** - **10** of about **55,600,000** for **performance testing**. (**0.17** seconds)

○ Show Text [✓ ● Render HTML] ○ Render XML ○ Render JSON ☐ Download embedded resources

Figure 5. Verifying the script is working correctly with View Results Tree



Name: Aggregate Report

Comments:

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
Search Request	5	378	336	656	256	656	0.00%	2.6/sec	72.3
TOTAL	5	378	336	656	256	656	0.00%	2.6/sec	72.3

Figure 6. Performance test results for the verification run using Aggregate Report listener

Note a couple of things:

1. Figure 5 shows that there were a total of five requests made, all of them successful (marked by a little green triangle with a check-mark inside)
2. The tab of the View Results Tree is set to Response Data tab, and instead of displaying the results in simple text (default) I have selected Render HTML button on the bottom of the screen, allowing me to easily see if the results match my expectations.
3. The resulting image does not look like the results Google would return (JMeter will not render JavaScript correctly), however, I can easily see that the query is "performance testing" it was read from the file correctly, and the results returned match what I expect.
4. The Aggregate Report listener, at the same time, correctly post-processed all of my performance run data, and calculated both the throughput, and major latency statistics.

Post Processing your Performance Run Data

Most of the reasonable performance testing tools will automatically post-process your transaction data and calculate average transactions per second the system sustained during the run. Additionally, you will find several different latency values - minimum and maximum, average, median, 90%, etc... It seems very natural to take the average or median as your representative numbers and move on. Unfortunately, this is a very bad decision which will prevent you from understanding the real user experience. 90% latency is the typical value the system should be judged by, and in some cases, a system may require even higher standard (95% or 99% users received a response in the amount of time less than X seconds is not an unusual requirement). Obtaining these values is not particularly difficult even if the tool doesn't provide them for you - all that is needed is sorting the data set and reading off the results.

System resource consumption data is simply averaged out. These four values (90% latency, throughput, average CPU usage and average memory usage) are the beginning of the final performance graph. For the smallest interesting load (one thread/virtual user) going as fast as possible (not including any thinking time between requests) the system is represented by these four values. These values are the beginning of the graphs shown in Figures 1 and 2.

Completing the Preliminary Scalability Test

The scalability test consists of many individual performance test runs with different loads. After each run is post-processed, the machine consumption data (averages for CPU, memory and anything else you may find important) and the system performance data (90% latency and throughput) are graphed either individually or combined into two separate graphs. My personal preference is to create a machine consumption graph with all machine variables on a single graph and a separate graph showing throughput and latency as functions of load. See Figures 1 and 2 again.

In many situations, I receive requests to transform the system performance graph into a latency vs. throughput graph. Clearly, this can be done easily, but I find the resulting graph misleading and prefer to look at the graph as shown in Figure 1. While the reason for this may be a simple nitpick, I believe that the x-axis of any line graph should show independent variables, while the y-axis shows dependent variables. Since the only thing I actually control is the number of threads making requests and not the throughput itself, I prefer to leave this graph as shown, since it more accurately displays what actually goes on. Again, this is my preference - you are certainly free to do whatever you like.

The really interesting performance work starts here. As I mentioned, this is a set of graphs obtained on a well-behaved system, so we are not even looking for potential problems. What the project team is usually interested in are the following questions:

1. What is the maximum throughput of our system?
2. What is the maximum sustainable throughput of our system?
3. What load should be chosen as a maximum value for safe operation?
4. If the system needs to be modified in order to increase its performance characteristics, where should we start?
5. How can I tell this is a well-behaved system and I am not missing a huge underlying problem?
6. What would be a good load for a performance benchmark?

All of these questions can be answered by analyzing our graphs shown in Figures 1 and 2.

What is the maximum throughput of our system?

This question is relatively easy to answer - you can just read it off the throughput graph on Figure 1. Maximum throughput is the top of the hump shown in that graph (about 35 transactions per second in the graph shown). Considering that I am peacefully discussing "the hump" and that I have already stated this is a well-behaved system, you can infer that I am not surprised to see it there, and that I do not consider it a side-effect of a bad test execution or some other anomaly. I expect to see the hump (it may be more or less pronounced) in any system that is designed to queue requests which cannot be processed as soon as they arrive. Which tends to be most of the systems I work with.

"The hump" is the overhead you pay for creating, maintaining and processing the queue. Prior to this point (to the left of the hump on the graph) the system was fast enough and it processed requests as quickly as they came in. The top of the hump is the optimal operation of the system - the requests are coming in at maximum processing rate, thus not triggering the need to queue but using all the processing power to respond.

The only useful value of analyzing the hump, however is understanding how expensive queue keeping is. I expect to pay about 5% in maximum throughput, but if I see the cost being more than 10%, it is something I will discuss with the development team. Keep in mind, performance testing is interesting only when it comes to the analysis part. It is interesting precisely because there are no hard and fast answers and every data point, graph or result obtained needs to be understood in the context of the system analyzed.

The only other thing the hump tells you is where your queuing system kicks in. If you are familiar with queuing theory, however, this will also tell you that the latency of your requests to the right of the hump will be dominated by the length of the queue, and will tell you little about any other part of the system. Thus, choosing the benchmark load that is to the right of the hump makes sense only if what you are interesting in is the performance of that particular subsystem.

What is the maximum sustainable throughput of our system?

That question is also not too difficult - the maximum sustainable throughput is the tail end of the throughput graph on Figure 1 (about 32 transactions per second in this case). This is an important number - because it tells you how fast your system will be able to cope with requests coming in, even when overloaded. For capacity planning purposes, this should be the absolute maximum number you ever report, because your system will not be able to support any higher throughput.

What load should be chosen as a maximum value for safe operation?

This is a tricky question, and it very much depends on things that have not been addressed yet. The true answer is - it depends. What is the importance of the system? What are the risks associated with system not being available or responding very slowly? The answers to these questions will greatly influence the answer to what is the maximum value for safe operation.

However, for Unix-based systems, from experience, I will pick a point where no system resource (memory, CPU, disk I/O) is operating at higher than 80% of the maximum load, on average. The reason for this is fairly simple - if overloaded for an extended period of time, Unix OS itself can become flaky and not behave the way you expect. Once that happens, you will run into strange behaviors that are difficult to reproduce, analyze or debug. So, from experience, I will pick the 80% maximum usage as my rule-of-thumb cut-off point. For the system described by Figure 1 and 2, this would be somewhere around 15 thread load.

If the system needs to be modified in order to increase its performance characteristics, where should we start?

A secondary benefit of completing exploratory performance tests for a system is finding out additional information about the system that can be incredibly useful for future benchmarking efforts. For example, the system represented by Figures 1 and 2 is clearly CPU limited. This means that any optimizations in the code should be done to trade off memory benefits for CPU benefits. It also means that any increases in CPU utilization will impact the system throughput much faster than changes in memory utilization, and therefore, benchmark tests should be more sensitive about increases in CPU usage.

How can I tell this is a well-behaved system and I am not missing a huge underlying problem?

You cannot. If you are concerned, once you have all of these runs completed and all the graphs ready, by all means, consult a performance testing expert. However, in my experience, most common performance problems are easy to see and even diagnose during these exploratory performance runs, and they will show themselves the first time you start a run with multiple parallel threads. Most often, you will notice your error rates going up - a clear indication that the system is not behaving as it should.

What would be a good load for a performance benchmark?

Usually, to get to the point where you have Figures 1 and 2, and a well behaved system, you will spend some time debugging the problems, working with new patched up versions and iterating. The process is interesting, and typically provides a lot of valuable feedback to the team. However, what you can do once the exploratory performance testing is complete, where you work really shines and provides the most value is if it is turned into a well-selected benchmark. Basically, you are creating a "unit" performance test, that can run on a frequent (nightly?) basis and can detect performance degradation quickly. With this type of information, you never put your development team in a situation where they have to sort through months of code changes looking for one or two lines that have seriously degraded system performance.

So, what should a good benchmark do?

1. It will target the parts of the system we are interested in
2. It will try and eliminate any environment related noise
3. It will try and simulate all parts of the system the way we expect them to run in production

My rule of thumb is very simple - a good benchmark (unless you are interested in evaluating your queuing sub-system, of course) should be to the left of the maximum throughput hump in Figure 1. So, technically, anything below 20 threads (in this example) would be acceptable, anything above it not. I usually select a load that corresponds to whatever has made any average system characteristic close to 80%. Looking at Figure 2, that would indicate something around 15 threads. I was recently asked why I would not select something much lower than that - wouldn't 5 threads do as good of a job?

Truthfully, there is no reason why you could not run a benchmark with 5 threads only. Chances are, you would be able to detect performance degradations in that setup as well (there is no theoretical reason why the idea should be rejected outright.) The reason why I prefer to use a relatively high load is twofold:

1. It is more accurate representation of what the system is likely to experience once deployed
2. At higher loads, you are more likely to trigger additional problems which may remain hidden otherwise. While this is not strictly a performance issue, it certainly makes sense to test as much as possible.

Happy performance testing!