Jon's Performance Musings: On Testing

Jon E. Schmidt
Transaction Design, Inc.
San Rafael, CA, USA

Jon is the founder of Transaction Design, Inc. (TDI), a consulting firm located in the San Francisco Area which specializes in capacity/ performance studies with clients worldwide. He is the creator of the Ban Bottlenecks® service and has an extensive background in the implementation, testing, and tuning of high-availability systems

Stressed?

I had a conversation recently with a class I was teaching on performance, and the subject of testing came up. I got several surprises in how they were conducting their testing, and I thought I would share my thoughts about performance testing here. By the way, I won't be discussing functional testing. Performance only.

The Goal

Why do we do performance testing? All applications have work to accomplish. The basic question is: "Can the application accomplish the work in the time allowed with the resources allotted?"

So, what is "the work?" Is it a file or table to be processed? Is it a message-based transaction stream? Is it an interactive application with screens and browser interactivity? And since the testing involves work per unit of time, what is the volume of work that must be accomplished? How many rows, how many transactions per second? What is the required throughput?

And of course, you don't want to test for today's production volume. You want to test for the volumes which are anticipated for the projected life of the hardware involved. You want to test for the peak anticipated volume. So if the application is going onto a platform that is expected to have a 3-year life, the test designer needs to understand the historical peaks of demand (traffic), and then to project those peaks out three years. Those projections are the target of the tests. Never test for the average.

Challenges and Traps

It's not easy to do a volume test realistically. The hardware works against you, particularly in the case of disks. And the disk environment is usually where the bottlenecks are. So, your tests must be designed to:

1. Process enough data to get past the stage where you are working within disk cache without physically driving the disks, and
2. Have a wide enough distribution of account numbers, etc. such that you're not hitting the same section of indexes all the time, with the result that a small but busy section of index sits in cache.

Another temptation is to not call outside services such as an SOA service or an external authorizer. It's much easier to just test your own code and have your code call a 'stub' which doesn't really represent the full transaction path. This is a dangerous omission. We frequently find that a partner service is not prepared to handle the required volume.

A mistake which applies to OLTP transaction testing is to not have a realistic number of simultaneous transactions. As this column has frequently discussed, the ability to handle lots of threads efficiently is mandatory. Lack of threads can kill response, even if there is lots of hardware capacity.

Unit Testing

I define unit testing as processing a single transaction stream. That is, no parallelism or concurrency. Unit testing will give you:

1. Ideal response or service time. No contention with other streams will result in the best case numbers for response. If the transaction is slow or throughput insufficient during unit testing, you have 'slowness', not a bottleneck.
2. Rationality check. Processing N transactions should result in x*N calls to servers, services, and other entities, as well as x*N reads, inserts, and updates to files and tables. Count this activity, and see if it makes sense.

Volume Testing

I define volume testing as a simulation of the anticipated production environment. Volume testing differs from unit testing in that it attempts to emulate exactly how the system under test will behave in live production, and beyond. Some thoughts:

1. Volume testing should involve real volumes of transactions, and extended time for testing. It should not be unusual for tests to run for hours, or even days.
2. OLTP testing must simulate the volumes required, and the concurrency required. It's a very different thing to test for 100 transactions per second, and to test for 500 transactions 'in flight' or users on line at the same time. Testing interactive applications must have the scripts include "think time", simulating the user typing, browsing, or otherwise not immediately responding to the application. Including think time in a script can help find data locking problems.
3. I recommend that volume tests for OLTP be staged: Test at 50% anticipated peak rate, then at 100%, then at 150%, and perhaps beyond. At each rate, look at the response statistics, looking for slowdowns. If things slow down as volume increases, you have a bottleneck. The question

is whether the slower response is acceptable, or whether the bottleneck can be resolved or removed.

4. Over the longer term tests, again look for slowdowns. As databases build up over time, indexes get deeper and take longer to traverse, and other search techniques tend to slow down. Disk cache hit ratios drop and physical disk I/O goes up, slowing things down.
5. Once you've successfully hit your target throughput, concurrency, and response, rerun the tests while running expected normal background activity. This should include running online applications versus batch jobs running against shared data. It should also include things like backups, SAN snapshots, etc.
6. One last thing: Your testing should always include a detailed analysis of the test results from the application. This should include transaction success analysis, and outlier analysis for response times. A test where all transactions are denied is not a valid test. Your analysis should not only focus on average response, but should count the number of transactions that fall into different buckets: Good response, acceptable response, nearly broken response, and unacceptable response. An understanding of why some transactions are getting unacceptable response when the test is generally successful is mandatory. If you have a wide variation in transaction response, you probably have data locking issues that need to be addressed. I can guarantee you it will be worse when you go into live production.

Stress Testing

Testing at volumes above and beyond N times the anticipated workload might be interesting. If you continue ramping up the workload until the application breaks or response becomes unacceptable, you could learn where the limits are. If the application is breaking, you might discover a weakness which should be fixed before going live. Ideally, applications should never break. They should simply slow down and reject any extreme workload, but then recover as the workload comes back within capacity.

Breakage Testing

Since we're in the high availability, critical-system world, your volume testing needs to have two additional scenarios: Impact of an equipment failure during high volume, and impact of recovering from that failure during high volume. I can guarantee that these kinds of tests will be interesting. Fail CPUs, disks, nodes, IP ports, etc. and look for the impact on the transaction stream. There will be one. This kind of testing is excellent at setting expectations for the impact of failures and recovery in real-life production.

Summary

Performance testing is a mandatory, non-trivial exercise. It is not easy to do properly, yet the consequence of not doing it correctly can lead to very expensive surprises when the application or new release goes into production. Done correctly, it can lower the stress of the entire organization.