Performance Testing

Load testing tools have been around for more than a decade but performance testing practices adopted by the IT industry are still far from maturity. There are still severe gaps between what is seen in tests and what one sees in production. The cost of running performance tests is also quite high given the significant investment in infrastructure, time, tools, and people. This document highlights several challenges that faced by the industry today and provides simple and cost effective solutions for the same.

Copyright (C) 2011 Rajesh Mansharamani
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is included in the section entitled "GNU Free Documentation License".

http://www.gnu.org/licenses/fdl.html
Performance Testing

Rajesh.Mansharamani@swperfengg.com

November 2011

Once all components of a software system are developed, they are integrated together and functionally tested in what is called integration testing. After integration testing is successfully completed, the non functional characteristics of the software system are evaluated. One may wonder why performance should not be tested in parallel with integration testing. It is definitely a good practice to test for performance at each component level during development and during unit testing. However, when all components are put together it is common to find a number of bugs during integration testing. Fixing these bugs will cause changes to the software which could impact the performance. Moreover, if a transaction fails during integration testing, it is likely that the transaction will also fail during performance testing. Therefore it is a preferred practice to wait for integration testing to be completed before embarking on performance testing. In case of large projects, with tight schedules, it is common to have a number of software releases. Thus while release 1 undergoes performance testing, release 2 can undergo integration testing, release 3 can undergo development and so on and so forth.

This document first covers the various maturity levels in performance testing, especially for mission critical and enterprise critical systems where performance is a mandatory requirement. We then get into the basics of the performance testing process. Thereafter, we highlight common mistakes that are to be avoided during performance testing. We finally delve in to use of performance modelling to show how performance testing can be accelerated when one works with a large number of concurrent users.

This document is not about performance testing tools. Performance testing is one area of performance engineering where tools abound in the market, both in the commercial space and in the open source. The tools have very good documentation available with them and it is redundant to replicate the same in this chapter. Instead, our focus is on the process aspects and based on our experience, on how to avoid mistakes during performance testing.

1. Performance Testing Maturity

Performance testing is a loose term that we use to cover a large number of tests that are required to prove that a given software system meets its performance requirements. There are several objectives that need to be addressed during the performance testing cycle, and these are:

- Does the software system meet the overall business requirements of throughput and response time, on the hardware architecture proposed for production?
- Does the software system scale on the underlying hardware architecture, such that workload growth will be handled gracefully?
• What is the performance of the software system as a function of database volumes?

• What is the stability of the software system in its ability to run without failure for hours and days?

• How does the software system perform under saturated conditions?

• How does the software system perform when there is interference from background load? (For example, online transactions and reports or batch.)

• How does the software system respond to sudden spikes in workload, either increase in number of users or sudden increase in throughput?

Performance of a system determined by three factors:

• Performance of application software architecture, design, and implementation

• Performance of underlying hardware and software platforms

• Business workload

When performance testing is conducted post integration testing, the software architecture, design, and implementation is frozen and assumed to be that which will roll out into production if the performance is acceptable. However, for performance testing to succeed in meeting the objectives stated above, it is equally important the underlying hardware and software platforms be sized and configured as they would in production, and business workload to be identical to that in production. These factors are not always possible to meet due to cost, time, and practical constraints, and this calls for certain levels of maturity in the performance test cycle.

We characterise the maturity of performance testing on the basis of the following ten dimensions, which are presented in no particular order of importance.

1. **Test Infrastructure**: To be able to accurately estimate the performance of an application in production, the hardware infrastructure during performance testing should be identical to that of production. While mission critical systems such as stock exchanges invest in such a mirror of production, many enterprises choose to invest in lower capacity hardware for testing. In such cases, the next natural choice is to avail of the infrastructure at the disaster recovery (DR) site. Usually, a DR site is not fully operational and has a significant amount of hardware capacity lying idle. With features available in hardware to partition servers, it is quite common to create an adequate test partition for the purposes of performance testing. It is a good practice to have at least 40% of the production server capacity available for performance testing (excluding the servers meant for high availability), so that issues under load are found out during performance testing and an estimation model can be made from the test system for production performance.
2. **Network Infrastructure**: While it is easy to procure production-like hardware for a test environment, it is practically impossible to reproduce a production wide area network (WAN) into a test environment. A production WAN has thousands of network lines from all over the world whose cost as well as characteristics would be impossible to reproduce in a test environment. In some enterprises, response time on a WAN is modelled using simple mathematical equations based on a transaction’s network communication in bytes and bandwidth and delay of a network link. In several other cases, for a few important geographies the production WAN is used to connect to the test servers. In more mature enterprises, a WAN emulator is used to for accuracy and early performance assurance in test.

3. **Performance Testing Tool**: In the past, it was common to have 10 to 25 concurrent users for applications. These days, many enterprises launch applications that need to be accessed by thousands of concurrent users. Without an automated performance testing tool it is impossible to conduct a realistic performance test. For systems with a few components or for those that deal with proprietary protocols it is common to find an in-house workload injection or performance testing tool in an enterprise. This approach also works well when there are good programmers and limited tool budgets. A number of mature enterprises choose to invest in industry standard performance testing tools from companies such as HP (HP LoadRunner) and IBM (Rational Performance Tester), while small sized enterprises choose to opt for open source performance testing tools (OpenSTA, Webload, Grinder). The right choice of tool depends on the level of standardization of performance testing in the enterprise, the tools budget available, and the types of technologies used to build and host applications.

4. **Entry Criteria for Performance Testing**: It is quite common for performance testing teams to embark on testing with production-like workloads, which require a significant amount of setup and also significant expertise in determining performance bottlenecks. It is also useful to establish entry level criteria for performance before the performance testing phase. This can be a certification of performance for individual application and system components. Alternatively, it can be single user performance targets with and without emulators. It is not common for enterprises to adopt entry criteria since the feeling is that this would increase the project timelines. In the absence of such criteria it takes longer for the performance testing phase to complete and that too with lesser accountability as far as developers are concerned.

5. **Types of Performance Tests**: The most common type of performance testing, called **load testing**, is where a number of concurrent users are simulated. In load testing, a tool is used to record the behaviour of a representative end user. Thereafter, the tool can replay the user’s activities and also create multiple such users to inject load into the system. Another popular test that needs to be done in large enterprises is called a **volume test**. In volume testing, enterprise databases need to be populated with growing volumes to determine the performance and scalability as a function of database size. For mission critical systems it is useful to conduct a few
rounds of stress testing. In this type of testing, the stability of the system under test is assessed as a function of load and duration. More specifically, a stress test is run with peak load conditions such that CPUs are close to 100% utilization and the test runs for hours and sometimes days. If the system does not crash and if the throughput remains steady all throughout, then the test is deemed a success. More mature enterprises also invest in availability and reliability tests, wherein the sensitivity of the system is assessed against various points of failure and against various invalid input conditions. These tests are executed during conditions of peak load.

6. Utility of Performance Tests: The basic need for performance testing is for the purpose of assurance of performance in production and performance tests are usually run to find out the performance defects in the system under test. As an enterprise matures, performance tests are run to proactively determine bottlenecks and optimize resource consumption in order to lower costs. At a higher level of maturity, performance test results are used for planning purposes. In such cases, results of performance tests are made available in a repository. Whenever a decision needs to be made on procuring hardware for a new technology, the repository is queried to determine the right sizing for a given workload on that technology.

7. Workload Simulation: The most common type of workload is a synthetic workload, wherein a percentage mix of transactions, a constant number of concurrent users, and a specific load injection rate are chosen for the test. This allows various controls to be set and simulated. However, in real life, there is significant variation in these parameters which makes it difficult to reproduce these conditions through a synthetic workload simulator. Instead, what works better is to have a load injector that can replay the real life workload using log files from a real life production system. The log files should contain sufficient information about the transaction time, transaction type, and input parameters for the transaction. The ideal form of workload simulation is to have a hybrid approach where a real life workload can be replayed and can be adapted synthetically to simulate worst case scenarios along with growth in the workload.

8. Monitoring and Analysis: For large scale performance tests it is common to have a team of application, system, and performance engineers available at the time of the test. The test progress and results are monitored and analysed manually and this is a time consuming activity. However, the pressures of running a large number of tests with application specific output formats make it difficult to create automated analysis tools at the time of the test. It is recommended that such tools be studied and procured at the start of the performance testing project. There should be an allocation of effort and budget for the same. If the enterprise has the required budget, industry-standard monitoring and analysis tools should be procured. Some monitoring and analysis tools also come bundled with load testing tools.

9. Performance Test Frequency: Before a system is launched in a production it is common for a mature enterprise to execute a performance test program for the system. Thereafter, the system keeps undergoing changes due to application maintenance, or new business requirements, or
software and hardware upgrades. At the very least, a performance test should be run periodically, say once a quarter or twice a year, to ascertain that acceptable levels of performance will be maintained over time. It is better to plan for growth and schedule performance tests whenever higher growth is anticipated. Large enterprises with mission critical applications usually plan for performance tests after every software release to ensure a high degree of customer satisfaction.

10. **Performance Test Duration**: Estimation of time and effort for performance testing is an art and often enterprises get these numbers wrong. Less mature enterprises often feel that performance testing of any new release can be done in a matter of one or two days. In reality, for a new release of a very complex application, performance testing can take around 8 months. This has been corroborated in a number of cases where large systems have been delivered with millions of lines of code and 20000+ function points. These systems usually cater to thousands of concurrent users and millions of transactions per day. For small systems that cater to around 100 concurrent users and have around 10 to 20 types of transactions it is quite common to have performance testing last for at least 4 weeks of duration. For incremental releases, two weeks of testing and tuning usually works right. A script for a test case usually takes half to one person day to create, (for example, record replay and customization of a script for a banking deposit transaction). A performance testing team for a given application would usually have two to three core members responsible for scripting, execution, and analysis. The application monitoring, tuning, and bug fixing would need to be done by complementary teams.

Table 1 summarises our discussion in this section with recommendations for good, better, and best levels of maturity in performance testing.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>Better</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test Infrastructure</strong></td>
<td>At least 40% of production</td>
<td>Use of DR site for performance testing</td>
<td>Mirror of production</td>
</tr>
<tr>
<td><strong>Network Infrastructure</strong></td>
<td>Extrapolation of results for a WAN</td>
<td>Real WAN for select geographies</td>
<td>WAN emulator and real WAN for select geographies</td>
</tr>
<tr>
<td><strong>Testing Tool</strong></td>
<td>Home grown</td>
<td>Open source</td>
<td>Industry standard</td>
</tr>
<tr>
<td><strong>Entry Criteria</strong></td>
<td>Test plan</td>
<td>Test plan and acceptable single user performance</td>
<td>Test plan, acceptable single user performance on emulators, acceptable performance and scalability per component</td>
</tr>
</tbody>
</table>
# Types of Tests

<table>
<thead>
<tr>
<th>Good</th>
<th>Better</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Tests</td>
<td>Load and Volume Test</td>
<td>Load, Volume, Stress, Availability, Reliability Tests</td>
</tr>
</tbody>
</table>

## Test Utility

<table>
<thead>
<tr>
<th>Good</th>
<th>Better</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottleneck detection and fixing</td>
<td>Cost optimization</td>
<td>Capacity Planning</td>
</tr>
</tbody>
</table>

## Workload Simulation

<table>
<thead>
<tr>
<th>Good</th>
<th>Better</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthetic</td>
<td>Replay from production</td>
<td>Hybrid</td>
</tr>
</tbody>
</table>

## Monitoring & Analysis

<table>
<thead>
<tr>
<th>Good</th>
<th>Better</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>Custom built scripts</td>
<td>Industry standard tools</td>
</tr>
</tbody>
</table>

## Test Frequency

<table>
<thead>
<tr>
<th>Good</th>
<th>Better</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodic</td>
<td>Planning for growth</td>
<td>Every release</td>
</tr>
</tbody>
</table>

### 2. Performance Test Cycle

This section discusses the phases of performance testing. It is assumed that before one embarks upon performance testing, there is a *Performance Test Plan* in place. The performance test plan usually contains the following sections:

1. Objectives of performance testing based on customer requirements
2. Hardware and software system configuration for performance testing, including setup of load injectors and load testing tools
3. Assumptions and terms of reference
4. Workload conditions to be tested, both data population and user transactions
5. Performance metrics to be captured during testing
6. Process to be followed during performance testing
7. Table of test cases, configuration required for them, and estimated schedule
8. Format of test report

Once the test plan is signed off and the environment and application are available for testing, we are ready to start with performance testing. The performance test cycle comprises of three phases that are repeated over time until all test cases are complete or the project is terminated for lack of time or inability to get desired results. The three phases are Setup, Execution, and Analysis, and these will be elaborated next.

**Setup Phase:** In this phase, the following activities are executed for a given test case.

- Load testing scripts are written or modified
• Data input values are prepared per script, such that different transactions and different users have their own sets of data inputs. This activity is called *parameterization*. Essentially, it involves creating a set of input values for various parameters. For example, a script for a deposit transaction in a bank would take a customer ID, account number, amount deposited, date, and several other fields as input parameters. These parameters need to have different input values for each transaction instance and for different users. Parameterization is usually done by providing a list of input values in a file or by using random number generation to generate a desired input value that fits in to a specified range (for example amount between Rs. 1,000 and Rs. 1,00,000).

• Scripts are modified to ensure that *correlation* across steps is accounted for. For example, in a web transaction a session ID is generated for a given user after login. The same session ID needs to be available in subsequent requests by the user and it thus serves as a correlation key across interactions.

• Scripts are augmented to include time delays for user interaction such as data entry, viewing of output or idle time. The sum of these delays is often referred to as *think time*.

• Database is populated for appropriate volumes for the test. This can be time consuming, so it is always useful either to have a data set available upfront or to write tools to automate data population.

• Database is set to a valid state. Quite often when multiple rounds of load testing are done, the data nears an inconsistent state. For example, if we wish to simulate withdrawals transactions from a bank. Say you have Rs. 20,000 as your account balance and you withdraw Rs. 8,000. During the first and second test, your withdrawal will pass and your balance will become Rs. 4,000. When the third test is run the withdrawal transaction will fail since you have insufficient balance. Hence it is a good practice to keep a consistent image of the database handy before a test and refresh the database with this image before the test starts.

• Hardware and software configurations are set properly for a given test. This includes server partitioning, operating system tuning, web, application, database server configuration and tuning, network configuration and tuning, storage configuration and tuning, and setup of any other package that the test case depends upon. At times external interfaces need to be made available for a given test, and at times background jobs may need to be run before a test can be executed. The setup phase can be time consuming and needs to be planned carefully depending on resources available for testing.

• Monitoring scripts, if required, and created and kept ready for the execution phase.

**Execution Phase:** In this phase tests are executed and results are generated for the analysis phase. It is assumed that the setup phase is completed before a test is ready for execution. The execution phase involves the following activities.
Specifying the number of users for the test to be run

Specifying the think time per transaction and the inter-transaction iteration time (either as static values or randomly generated)

Specifying the run duration or the number of iterations each transaction needs to be run

Specifying the ramp up rate and the ramp up duration. Ramping up the number of users is as per a time profile. If all simulated or virtual users are started up simultaneously it could cause significant overload on the system's transient state; it pays to ramp up 10 to 50 users at a time over a few seconds.

Specifying the ramp down duration and the rate of ramping down users

Firing the test

Monitoring when the test reaches steady state (usually in a few minutes) wherein the system throughput tends to be steady

Monitoring for abnormal conditions that cause scripts or transactions to fail

Monitoring the system resource consumption and overall response time and throughput

Ensuring the test has completed and all test output is available in specified directories for analysis purposes

**Analysis Phase:** In this phase the test output is analysed for performance and for errors. The analysis phase involves the following activities.

- Collecting all test output files available at the end of the execution phase
- Analysing the output files through third party or custom built tools for determining the overall performance of the system, that is, response time (usually 95th percentile and average) and throughput
- Analysing output files through third party or custom built tools for determining resource consumption per server, usually CPU utilization, memory consumption, I/Os per second, and network bandwidth consumption as well as network bytes sent and received per transaction
- Analysing output files through third party or custom built tools for determining the throughput at individual component tiers
- Analysing output files through third party or custom built tools for determining the response times at individual component tiers
- Analysing output files for number of transactions that have failed and determining the reasons for failure. Coming up with recommendations for fixing the errors, if possible.
• If the overall performance objectives are not met, the determining which are the bottleneck components and further analysing the output for narrowing down the problems in the bottleneck components

• Prioritizing the bottlenecks—both system parameter tuning and application code/design/architecture optimization

• Creating an analysis report to be delivered to the performance test manager, the application owner, the application developers, and the system administrators

The cycle of setup, execute, analyse continues until overall performance objectives are met or until one realizes that the objectives will not be met in the given architecture or deployment of the system. If major architecture and design changes need to be done, then the performance test phase is suspended until the changes are complete.

3. Common Mistakes in Performance Testing

Even though performance testing has been predominant in the IT industry for years, a number of common mistakes are made by performance testing teams. This section presents the mistakes in no particular order of importance or occurrence. All the findings are based on real life experiences encountered by the author during project implementation and reviews. Many of the mistakes surface because people expect the performance testing tool to take care of the performance testing process.

1. Direct jump to multi-user tests: One of the most common mistakes is to directly jump to multi-user tests ignoring the fact that single user tests need to be performed. The optimistic belief by IT personnel that the software system developed will work right the first time, has been a long running tradition since the inception of IT systems. This belied optimism leads to significant delays and finger pointing during the course of performance testing. Imagine trying to drive from Mumbai to Pune in 2 hours time during peak hours. If you cannot reach in 2 hours during non-peak hours, how could you possibly reach on time during peak hours?

If single user tests have enough data to point out performance bottlenecks and if emulators are also used during the tests, more than 50% of the performance problems will be known and solved before one gets in to multi-user testing. Single user testing is easy with very little setup overheads and can often be done manually.

Here is a case study to highlight this point. A large clearing corporation had a distributed architecture spanning a dealer workstation at a bank branch, a clearing server at the branch, a wide area network connecting branches to the central data centre, and a host application server and database at the central data centre. All communication was done through messages using IBM MQ. The team had developed the software, completed functional and integration testing and were ready to start with performance testing. The average response time target was 3 seconds end-to-end for a dealer
transaction and the throughput target was several tens of thousands of transactions.

The team started with a full scale multi-user test on identical hardware as production, and the first round resulted in an average response time of more than 30 seconds and only 200 transactions per hour. The ‘smart’ developers in the team experimented with some tuning options and after two iterations reached an average response time of 10 seconds with close to 1000 transactions per hour. Thereafter, there was no progress despite all efforts at tuning and optimization. The team was left clueless on what to do and all options pointed to significant changes in design and architecture.

At that time, the team was advised to just run single user tests with timestamps inserted at appropriate sections of the transaction flow (dealer workstation, branch server, host application, and host database). Once the timestamps were collected, the bottlenecks became apparent. A lot of time was spent in encryption and decryption at the host end, and by just using compiler optimization a 50% boost was received. The next bottleneck was in some of the queues and it was found out that the team was redundantly using persistent queues at all places, whereas they really required them only in a few places. With some more queue optimizations and no change in code, the team was easily able to meet the performance targets in just about two weeks thanks to an investment of one week in single user testing and profiling.

2. **Test Results Not Validated**: When one does functional testing, the test case output is known and the tester verifies whether the functionality being tested provides the correct output. In the case of performance testing, what is known is the response time and throughput target, and the real performance of the system is never known before testing. In such a case how does one validate the output of a performance test? Often one may have a fault in the analysis scripts of the performance testing tool itself. There could be another background job in the environment that may go unnoticed and that may impact results.

In such cases, one needs to ensure:

- **Repeatability of test results**, that is, the test should be rerun to ensure that the results can be reproduced
- An external authentic source of verification. This is easiest done by having a human user run manual tests while load tests are in progress and verifying that response times are consistent. For throughput verification one may use log files available in the system and validate those against output shown by the load testing tool.
- Cross check whether average response time, throughput, and number of users are balancing as per Little’s Law. (For those not familiar with this aspect, please refer to the first reference in the section titled ‘Useful Reading’ below.)
3. **Workload Details Unknown**: It is assumed that having a performance test plan is mandatory before entering in to the performance testing phase, however, while proceeding to execute test cases one often realises that the workload specifications in the plan were not complete. A project management system was being tested for performance and when the tests were about to start, the testing team realised that the only workload specification was “1000 concurrent users.” There was no specification of the transaction mix and different assumptions on the mix would lead to different performance and different capacity requirements. The only way out was to sit down with the planning team and ensure that the detailed performance test plan was worked out, before proceeding to execute the tests.

4. **Run Duration Too Small**: How long should one run a performance test? One must understand that the outputs of the test are statistical quantities such as response time and throughput and these being random variables one needs enough samples to compute the statistics. Note that whenever a load test starts, the system undergoes a *transient* phase (for example, filling up of application server and database server cache, or internal auto tuning of parameters such as thread pools and connection pools). It is only after all users have been ramped up and have been running for sufficient time, that the system will reach a ‘steady state’. We usually need to take statistics during the steady state.

The best way to determine the transient and steady state durations is to plan the first performance test for a very long duration, say several hours. While the test is running, the test output parameters such as throughput and CPU utilization are monitored for reaching a steady pattern. The timings for reaching steady state are noted and based on the number of samples required to compute (statistical) confidence intervals the duration of steady state is calculated. Thereafter the test is aborted and the total computed duration is specified for subsequent tests. For most systems that cater to 100 to 1000 concurrent users, and at least 20+ web interactions per second, we have found that transient state is usually reached in less than 5 minutes and that a steady state duration of 15 to 20 minutes is sufficient.

Often enterprises go through the test plan for the sake of compliance and while in a hurry to complete the performance testing phase, they run tests for very small durations. In such cases the test output often bears no relation to what would really be seen in production. Take for example, Figure 1 and Figure 2. In Figure 1 we see that a single user test reaches steady state very quickly. However, when we look at Figure 2 we see that it has taken around 50 seconds to reach steady state. If the run duration is less than 50 seconds, the results obtained would be of no relevance to real life scenarios!
5. **Absence of Long Duration Sustainability Test**: We highlighted the need to run up to the steady state. This is to get an accurate estimate of performance output, given that we are dealing with statistical quantities. There are times when tests should be run for a longer duration. For example, when testing a 24x7 web application, one needs to ensure stable performance all throughout 24 hours. While performance may be acceptable in a 20 minute test, it is possible that the software can age and deteriorate in performance as time progresses. For example, if there is a memory leak in the software at a rate of 100MB per hour, and you have only 1GB available in your memory, then after 10 hours your software can crash.

Before launching an IT project management system in a large company, there were several rounds of performance testing. Test durations of 20 minutes were yielding acceptable performance on both Windows and HP-UX. A choice had to be made whether to go ahead with Windows or HP-UX. Two application servers were used in load balanced mode. When a 24 hour test was run on Windows, one of the application servers saturated to 100% in 12 hours and the second application server crashed in 3 hours. On the other hand, HP-UX was quite stable for 24 hours and it was finally chosen for the system launch.
Project teams tend to miss out these sustainability tests in the interest of quick completion of performance testing. While there is no denying that sustainability tests are of long duration, one usually plans one or two of such tests. If the software system does not sustain the load for 24 hours or more, then the bugs need to be fixed before launching in production. Let’s put it like this: it’s better to have a failure in a flight simulator than to have it happening on a real flight!

6. **Confusion on Definition of Concurrent Users**: It is quite common in many enterprises for IT managers to get confused about what is the meaning of concurrent users. An IT manager of large bank was rolling out a web application on the Internet. As per a business requirement there were to be 5000 concurrent users. The IT manager wanted the vendor to test the condition when all 5000 users simultaneously press the submit button since his understanding of concurrent was ‘simultaneous’. In several other cases, IT managers have viewed 5000 concurrent users to be 5000 hits/sec on the web server, or 5000 concurrently processing requests on the application server. The confusion stems from an improper delineation of the system under consideration. When we talk about users we mean the entire business processing system and by concurrent we mean that 5000 users are operating from their business terminals (or browsers in web systems). The user can submit a request and wait for a response, or view the output of a response, or wait for the next customer to call, or simply be drinking coffee and doing nothing on the terminal, but is still a concurrent user as far as the business processing system is concerned. The transaction rate on the system is determined by both the number of concurrent users and the think time per user.

7. **Data not Sufficiently Populated**: It is common to see performance tests being done on a small database and get results that are acceptable, and then to discover a disaster in production when the throughput just does not cope up on a large database. Sometimes the budget is insufficient for large storage in the test system and no formal way to extrapolate results for larger databases. At other times, such problems occur due to sheer lack of foresight. One of the billing applications in an IT department was being launched for the first time and no legacy data was kept in the system. Since it started out on a clean slate the database was empty when the billing application was launched. Over a period of two years, 70000 invoices were generated and the system started crawling. It used to take 10 minutes to generate an invoice instead of the usual time of 10 seconds. There was a big debate on increasing the number of CPUs, but people failed to realise that the application had never been tested for performance with a large database. When such a volume test was done the issues clearly stood out, and after proper tuning the performance was restored back to normal on the same hardware.

8. **Significant Difference between Test and Production Environments**: We have discussed this issue a number of times in this book. Less mature organizations invest less in test environments and there have been cases where customers have asked to extrapolate results from a 1 CPU test server
to a 64 CPU production server. Such extrapolations are not practical. In general, we advocate that the test system be at least 40% of the production capacity (refer to Table 1).

9. **Network Bandwidth not Simulated**: Performance testing is usually done on a LAN environment and the focus is on ensuring that the application performs and scales with load. One important aspect that is ignored is that the real life application is going to be accessed from a WAN. If the application is very chatty or has large pages or does not adapt well to network failures, there are bound to be significant performance problems on the WAN. To overcome these issues one should either use a WAN emulator, or test the application over real WAN links, or analytically model the behaviour of the application on a WAN.

10. **Performance Testing Schedules Underestimated**: We have dealt with this topic in Section *Error! Reference source not found.*

11. **Incorrect Extrapolation of Pilots**: It is often the case that when one embarks of a large performance test, one needs to use a benchmark laboratory at the premises of a hardware vendor. To estimate the size of the servers required, a pilot is run in a smaller test lab and results extrapolated. While analysing pilot test results, it is critical that the bottlenecks are understood well at the time of extrapolating the results.

In one case, an income tax assessment application was run on a 4 CPU server with 1, 5, 10, and 100 concurrent users. The throughput saturated at 2 transactions per second (tps) after 10 users. In real life, 80 tps was required and the team extrapolated the number of CPUs required to service 80 tps as \( \frac{80}{2} \times 4 \text{ CPUs} = 160 \text{ CPUs} \). There was no server with 160 CPUs available in the market and this forced the team to reassess the extrapolation. The team ran the tests again but this time with 1 to 100 concurrent users, in steps of 10. The throughput increased to 10 tps at 50 users and then started dropping until it reached 2 tps at 100 concurrent users. It was a pure coincidence that both 10 and 100 concurrent users yielded 2 tps. However, the missing data points made it appear as if the throughput saturated earlier. The reason for the anomaly was the application was very memory intensive and after 50 concurrent users the application started **thrashing**. With the new throughput peak of 10 tps, the extrapolation yielded 32 CPUs for the vendor lab configuration and that turned out to be an accurate estimate when the larger tests were executed.

12. **Inappropriate Baselining of Configurations**: When an IT system needs to scale, one examines the impact of adding more servers. When moving components to new servers, the resource consumption of the component needs to be proportionately considered. Also, putting in additional servers will lead to additional communication across servers and this aspect is often ignored when creating an appropriate baseline for scalability. Let us illustrate this with an example.

A very large bank had a single application server and a single database server on a single 32 CPU server. They wanted to analyse the impact of
horizontally scaling to multiple servers. Two tests were conducted. The first was on two 16 CPU servers, with the application running on one server and the database on the second. In the second test, the database was further split in to two servers of 8 CPUs each using database clustering technology. In each case there was a total of 32 CPUs to be used. The team had a baseline throughput of 500 tps on the single 32 CPU server. When they ran test 1 the throughput dropped significantly and there was a further drop in test 2. The team concluded that the application would not horizontally scale as a result of these tests.

In reality, the application to database resource consumption ratio was 5:3. This effectively meant that out of 32 CPUs, the application needed 20 and the database needed 12. Once the application and database were on separate servers, there was a 25% communication overhead which needed to be accounted. Hence an equivalent configuration (to the single 32 CPU server) was a 24 CPU application server and a 16 CPU database server. Test 1 was repeated with this configuration and the throughput was 500 tps which proved that horizontally splitting the system was not detrimental. The 16 CPU database was again split in to 8 and 8 CPU partitions, but again 25% was added for database clustering overheads. This resulted in a 24 CPU application server and two 10 CPU database servers for test 2, which also resulted in 500 tps. The three baselines were thus proven equivalent because now the baseline configurations were appropriate.

The issues highlighted in this section are commonplace issues that occur in a large number of projects. The reader may feel that these are quite obvious, but remember there’s no such thing as the ‘obvious’: when project pressures take their toll the ‘obvious’ is forgotten in a race against time.

4. Accelerated Performance Testing

These days we often encounter performance tests that need to be conducted for several thousand concurrent users. In some projects tests have been done in excess of 10000 concurrent users. Some nationwide sites even talk about the need to support 250000 concurrent users. Under realistic test conditions with thousands of users and tens of seconds of think time, a single test run can run in to hours, considering the time for ramp up of users, the time for collecting sufficient number of samples in the steady state, and then the time taken for the ramp down of users. It would be desirable to determine where the system will bottleneck as soon as possible and even with smaller number of users predict whether the system will scale for larger number of users.

Before we proceed with our discussion on this topic, it will be useful to cover some theoretical background that will assist us in accelerating performance tests. Section 4.1 covers Little’s Law and Bottleneck Law, and then Section 4.2 provides a methodology for accelerated performance testing.

4.1 Theoretical Background

Little’s Law is a well known theorem from operations research, which was published in 1961 by J.D. Little. We use a simplified version of it for closed systems
(having a fixed number of users). Consider a system, as shown in Figure 1, with $N$ (business) users, with an average response time of $R$ seconds per business transaction, and having a throughput of $X$ business transactions per second. (We could might as well as use business interaction or web page per request.) Let us assume that the average think time per user is $Z$ seconds, per business transaction. By think time we mean the time that a user spends in data entry, or in viewing results, or in doing anything other than waiting for a response.

![Figure 3: Closed System](image)

Now, as shown in Figure 3, the average cycle time for one user is $C = R + Z$ and therefore the throughput per user is $1/(R+Z)$. For the overall system we get:

$$R = X \cdot C = X \cdot (R + Z)$$

The only assumption behind this law is that the system is work conserving, in that no work is generated within the system and no work is destroyed by the system.

For example, if a system has 3,000 concurrent users and a throughput target of 100 transactions per second, the average cycle time per transaction is $3,000 / 100 = 30$ seconds. If the think time per transaction is 25 seconds, then the average system response time target is $30 - 25 = 5$ seconds.

**Bottleneck Law**

Before we get on to this law let us elaborate on some important terms, namely, service time, visit count, and demand.

Service time is the time spent by a resource in servicing a single request. For example, a single banking transaction makes 5 requests to a web server with an average of 5ms CPU time per request, 2 requests to an application server with an average of 10ms CPU time per request, 4 requests to a database server with an average of 20ms CPU time per request, and 10 requests to the disk subsystem with an average of 5ms disk service time per request. Note that the service times in this example are time spent in servicing the request, and they do not including queuing
time or wait time at the resource, which forms part of response time. In other words, service time at a resource can be thought of as response time at the resource under idle conditions.

We use the symbol $S$ to denote average service time.

In the previous example we saw that a single transaction makes multiple visits to sub-systems and resources. The average number of visits to a resource is called the visit count of that entity at the resource. Note that visit count by definition is an average. Also note that visit count is a relative number. In the example above, one banking transaction makes 4 requests to the database server and 10 requests to the disk subsystem. Thus the visit count is 4 at the database server and 10 at the disk subsystem. This is relative to the banking transaction. At the disk subsystem the visit count relative to the database is 2.5 (10/4). Visit count can also be a fraction which is less than one. In the example above, if we have 8 CPUs at the database server, then the visit count per CPU is 4/8 = 0.5.

We use the symbol $V$ to denote visit count.

Whether we make 4 requests to the database server with service time 20ms per request, or 1 request with service time 80ms, the total service demand at the database server remains the same, that is, 80ms. Thus the average demand at a resource is the product of average service time at that resource and the visit count at that resource.

The symbol $D$ is used to denote average demand. Thus at each resource $i$ in the system the average demand is:

$$D_i = V_i S_i$$

Now let us get down to the Bottleneck Law. More specifically, we are looking at an upper bound on throughput or the maximum throughput that is achievable. Using that we can also derive a lower bound on average response time thanks to Little’s Law.

Consider an end-user system as illustrated in Figure 4. An entity requesting services of the system visits several resources, with a certain visit count and average service time. The circles in the system denote resources, and the tuples shown next to the circles specify the visit count and average service time at the resources.
As defined in above, the average demand at a resource is the product of the visit count and the service time. For the purpose of our analysis of demands, we can equate Figure 4 to Figure 5, which shows the system as a pipeline of resources each having service time equal to demand. In other words, instead of specifying that a resource \(i\) is visited \(V_i\) times with an average service time of \(S_i\), we specify that the resource is visited once with average demand of \(D_i\). For the purpose of the bounds derived in this section, this translation works appropriately.

If we consider any pipelined system such as the one in Figure 5, the maximum throughput of the system cannot exceed the throughput at the slowest stage of the pipeline. In the example in Figure 5, the maximum throughput of the system is 1/5.

Let the maximum average demand in the system, across all resources \(i\), be denoted by \(D_{\text{max}}\):

\[
D_{\text{max}} = \max_i \{D_i\}
\]

We therefore have the upper bound for system throughput as:
\[ X \leq \frac{1}{D_{\text{max}}} \]

This is what we mean by the **Bottleneck Law**, i.e., the bottleneck resource determines what the maximum overall system throughput will be.

The upper bound holds, regardless of the system workload. When the system saturates this, the upper bound becomes an equality. By definition \( D_{\text{max}} \) depends on visit counts and service times. \( D_{\text{max}} \) can be reduced by optimizing the software design and implementation to reduce service times, or by using faster CPUs or disks to reduce service times, or by increasing the number of resources at a service centre to reduce the visit count per resource, or by changing the architecture of the system to reduce visit counts. For example, if database visit counts are high, one can either increase the number of CPUs or disks, or introduce caching at the application server in order to reduce the visit counts.

From Little’s Law we get:

\[ R = \frac{N}{X} - Z \]

We now apply the upper bound on throughput to get a lower bound on average response time.

\[ R \geq ND_{\text{max}} - Z \]

These bounds on throughput and average response time, become equalities upon system saturation (unless the system is not work conserving and thrashes after a certain load).

### 4.2 Methodology for Accelerated Load Testing

If we wish to get to the bottleneck quickly, we want the system to saturate at the maximum throughput with the minimum number of users. This can be achieved by setting think time to zero and increasing the number of users until the throughput saturates as shown in Figure 6. With normal values of think time the throughput saturates when the number of users \( N \) reaches \( N_s \) users, and with zero think time the throughput saturates when \( N \) reaches \( N_0 \) users. Typically, \( N_0 \) is an order of magnitude less than \( N_s \).
As discussed in Section 4.1, the maximum service $D_{\text{max}}$ limits maximum throughput. $D_{\text{max}}$ is a property of the system architecture, hardware, and implementation. It does not depend on the number of users. Hence $1/D_{\text{max}}$ will be the same regardless of whether the system throughput saturates at $N_0$ users or at $N_s$ users.

In an income tax benchmark, the customer specified a think time of 39 seconds, and a target average response time 6.1 seconds for a workload target of 4000 concurrent users. The system saturated at 48 assessments per second with $N=4000$ users with a response time average of 40.5 seconds. For a think time of 3 seconds$^1$, the limiting throughput of 48/sec was reached at $N=500$ users. In fact, since we know that $R \leq 6.1$ seconds, at $Z=3$ we would need:

$$N \leq 48 \times (6.1 + 3) = 437 \text{ users to saturate the system.}$$

Once we get the maximum throughput in this manner, we get the maximum demand as $D_{\text{max}} = 1/(\text{maximum throughput})$. We have seen from Section 4.1 that

$$R \geq ND_{\text{max}} - Z$$

We found $D_{\text{max}} = 20\text{ms}$, seeing the system saturate at $N_0 = 500$ users. If we wished to know whether response times will be within 6.1 seconds at $N = 4000$ users and $Z=40$ seconds. Rather than test and tune at 4000 users, we use the lower bound on response time to get:

$$R \geq 4000 \times 20\text{ms} - 39 = 41 \text{ seconds.}$$

This matched the test result of 40.5 seconds. Since response time is far away from its target value of 6.1 seconds, there is no need to test at $N=4000$ until the system is well tuned at a lower number of users. What really matters is the maximum demand $D_{\text{max}} = 20\text{ms}$. To achieve the target of 6.1 second response time, we need $D_{\text{max}}$ to be:

$$D_{\text{max}} \leq (6.1+40)/4000 = 11.5\text{ms}$$

$^1$ When $Z=0$ was tried out, the system crashed. So when we say zero think time, we mean the smallest value at which the system remains stable for the duration of the test.
Thus during the test cycle, one should either instrument the system sufficiently to get service demands at individual resources, and tune the system until maximum demand reaches 11.5ms, or test with a smaller number of users and zero think times, until throughput reaches 90 transactions/sec (1/11.5ms).

To summarize:

1. We first set think times to zero and keep increasing number of users until the system saturates. If the throughput at saturation meets the overall throughput targets we proceed to step 2, otherwise, we keep tuning or increasing capacity until either the throughput target is met or the target is deemed infeasible within the budget available.

2. If the throughput target is met, we use $D_{\text{max}}$ as $1/(\text{max throughput})$ obtained from step 1, and then check whether response time bound (1) is satisfied at the desired value of $N$. If it is, then we do a final round of testing at the desired value of $N$, otherwise we keep tuning the system or increasing capacity, and repeating step 1 until $D_{\text{max}}$ is sufficiently small to meet the response time bound.

**Useful Reading**

The material presented in this document is derived in part from some of the references below. Other references have been added as useful reading to supplement what is presented in this document.


**Exercises**

1. Two performance testing teams evaluated an Application A under identical workload conditions and identical server configurations. The first team used Performance Testing Tool B and the second team used Performance Testing Tool C, for testing with a workload of 5000 concurrent users and 20 seconds think time. The first team got an average response time of 5 seconds and a system throughput of 1000 tps, when using Tool B. The second team got an average response time of 30 seconds and a throughput of 100 tps, when using Tool C. Which tool has provided the correct output?

2. You need to create a performance test plan for a bank, for 10000 concurrent users and a target response time of 5 seconds per transaction. The bank has specified a throughput target of 10 million transactions per day, 9% of which occurs in a peak 30 minutes. What is the average think time per transaction that you will use?

3. A financial services firm needs to process a peak of 1000 transactions per second at 0.2 seconds average response time and 0.05 seconds average think time for its algorithmic trading system. How many concurrent users will you use for performance testing?

4. Performance testing for a customer needs to be done for 2000 concurrent users and 30 seconds think time. You use the accelerated testing algorithm discussed in this course and test with 100 concurrent users and zero think time, under which you see that the system saturates at 50 tps with a response time of 30 seconds.

   a. What is the maximum demand across all resources?
   b. What will be the average system response time at 2000 users?
Answers to exercises

1. Tool C since it conforms to Little’s Law
2. 15 seconds
3. 250
4. a) $D_{\text{max}} = 20\text{ms}$  b) $R = 10\text{ seconds}$