User Experience, Not Metrics

by:
R. Scott Barber

Part 8: Choosing Tests and Reporting Results to Meet Stakeholder Needs

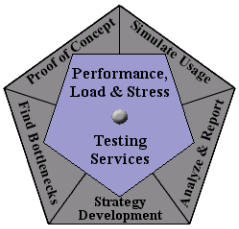
If you've been following this series of articles about how to do performance testing to determine how customers experience your Web site application, you've learned in detail how to model real users and capture meaningful response times. Now comes the moment of truth: running the right tests and presenting your results to stakeholders in a way that meets their needs. I keep trying to convince my clients that all they need from me at the end of a performance testing engagement is a Post-It note with either the words "Go Live" or "Don't" written on it, but they don't seem to think that provides enough value. If your clients are anything like mine, they'll want you to choose your tests carefully and report the results to them in a form they can understand and act on. This article outlines the types of performance-related tests that are commonly used to add value to a performance testing effort, and ways to present the results of these tests.

This is the eighth article in the "User Experience, Not Metrics" series. This article is the first installment in the third trio of articles, which will focus on reports to stakeholders. These articles will be less technical than the previous ones and will discuss how to present the results obtained from applying the concepts explained in the previous articles. After reading these articles, you should be able to present data from your entire battery of tests in concise tables and charts that highlight the significant aspects of the tests. This set of articles is intended for both Rational TestStudio users and managers with some Microsoft Excel experience. I'll give all the information you'll need in order to replicate the tables, charts, and graphs, but without the extensive Excel walkthroughs that I provided in Parts 6 and 7.

Choosing Which Tests to Run

According to "The Science of Website Load Testing," by Keynote Systems, Inc. 2000 "When load testing is not done properly, the results are, at best, useless and, in the worst case, misleading, causing a company to either underestimate or overestimate a site's capacity. A wrong result could cause unnecessary expenses, delays or potentially disastrous business decisions." From this, it's clear that the tester's responsibility to stakeholders is to conduct performance tests that will yield both accurate and useful results. Furthermore, conducting the right test for each stage of the development effort will limit the total amount of work needed to complete the effort.

It's important to remember that the ultimate goal of performance testing is to determine and/or maximize speed and scalability from the end user's



perspective, and the tests that are run should help achieve that goal. That's the key point to keep in mind as you choose which tests to run. The tests I'll outline below will provide 90+% of the results that stakeholders will ever want or need. While this article series has focused thus far on user experience measurements, other types of tests and measurements that are useful for tuning should be considered and will be mentioned here for the sake of completeness.

Almost every performance tester you ask will categorize types of performance tests differently. The categories I use here are based on how heavy a user load the tests put on the system and represent a good cross-section of types named by a wide variety of sources ranging from the Rational Unified Process to the software testing standards of the British Computer Society Specialist Interest Group in Software Testing (http://www.testingstandards.co.uk/). The names of certain tests may not be what you're used to, but all of the concepts in performance-related testing are captured by the tests described below.

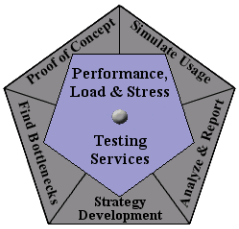
Here are the categories of tests I'll discuss:

- **Low-load** — Executed at not more than 15% of the expected production user load. Used to identify gross performance issues that would negate the value of testing at higher loads and/or provide a basis of comparison for future tests. Examples of low-load tests are baseline, bechmark, and component-based tests.
- **Load** — Executed at expected production user loads. Used to validate the actual performance of the system as experienced by users in production. Examples of load tests are response-time, scalability, and component-based tests.
- **Heavy-load** — Executed at greater-than-expected user loads. Most often used to test application and system stability and recovery, and to collect data for capacity planning. Examples of heavy-load tests are stress, spike, and hammer tests.
- **Specialty** — Specialty tests are generally created at the request of a developer or architect to help resolve a particular performance bottleneck.

Any of the tests mentioned here can be conducted at various connection rates. These tests will always provide best-case results for connection rates slower than the network you're connecting over but will give you a good idea of what the system will "feel" like to a user connected over, say, a 100 millibytes/second LAN or with a dial-up 56.6 kilobytes/second modem. Typically, only user experience tests are executed at various connection rates. You can get valuable results by conducting the same test under the same load several times, each with a different connection rate, and then comparing the results.

Low-Load Tests

A low-load test is any test executed at not more than 15% of the expected production user load. Most often, conducting low-load tests results in a collection of baselines and benchmarks that serve as a basis of comparison for multiuser tests and for verifying that the developed scripts are working properly. All performance-testing efforts should begin with some low-load testing to ensure that the scripts work properly and the system is stable enough to test at heavier loads. If any low-load test fails to meet the stated performance acceptance criteria, performance tuning should be done before moving on to heavier load tests. You can think of low-load tests as being similar to functional "smoke" tests.



Baseline Tests

Baseline tests are single-user tests that are most often employed to validate script functionality and get a feel for the overall performance of the system. "Low-hanging fruit" may be uncovered, but generally nothing will be found that wouldn't have been detected by manual screening of the deployment. Baseline response times are collected by executing each script (that is, a single user) individually over multiple test iterations. Baseline results can be used as a basis of comparison to analyze system response degradation as user load is increased.

Benchmark Tests

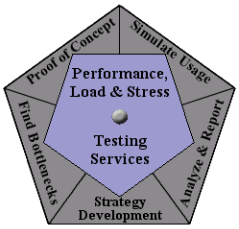
A benchmark test employs a scenario involving a small community of users compared to the target load. This community of users must be large enough to approximate a reasonable sample of the entire user community model while still being significantly smaller than the expected system capacity; 15% of total expected user load is generally a good benchmark volume. Executing benchmark tests ensures that the testing environment behaves as expected under a light load as well as validates that the scripts have been developed correctly. Additionally, the results of these tests serve as a benchmark against which to compare future test results. Each benchmark test should be executed several times to ensure statistical validity of the results.

Performance results obtained under the benchmark load should meet or exceed all indicated performance requirements; otherwise, tuning must begin with the benchmark load. Assuming no performance problems are noticed during this scenario, the results obtained can be used as "best case" results. These results indicate how the system performs when it's not under noticeable stress but is still performing all required functions, thus allowing conclusions to be drawn about the performance of the system during higher-load tests. Benchmark scenarios should be executed in a sterile environment and re-executed after each tuning effort to establish a new basis of comparison.

Component-Based Tests

Component-based tests place load on only one component or tier of the system and are most commonly used to verify tuning efforts. There are three main types of component measurements that are useful to collect at the low-load level: transaction rate, memory usage, and CPU usage.

- **Transaction-rate tests** are best used to determine the speed at which a component can process a transaction (such as a single database search or a file download) under various loads. These tests will uncover exactly how long the transaction takes to complete without having to subtract out the additional segments of time included in an end-to-end response-time measurement. A transaction-rate test will execute each test script (performing a single type of transaction) individually over multiple iterations. If performance becomes unacceptable from an end-to-end perspective, the results of this type of test can also be used as a basis of comparison to analyze system response degradation as the transaction load is increased.
- **Memory-usage tests** are used to monitor how memory is being used during a specific activity and are most useful in finding total memory requirements or memory leaks. Memory-usage monitoring is typically useful only if the activity being observed spawns an independent server-side process. For example, back-end scheduled processes that spawn memory-intensive processes are good



candidates for memory-usage tests because the amount of memory used by these types of processes needs to be known and accounted for.

To execute a memory-usage test, you simply execute a test that exercises an activity whose memory usage you want to observe and monitor the memory usage of that activity using a performance monitoring / diagnostic tool such as Task Manager, PerfMon, or PerfMeter. The diagnostic tool will show the total memory used by the activity, and whether total memory in use by the system returns to pretest levels between test iterations. If this doesn't occur, analysis should be done to determine the cause of this anomaly.

It's important to remember that diagnostic tools used for monitoring memory usage often use large amounts of memory themselves. Thus, the total memory used by the system won't be reflected accurately, but the difference in memory usage between different activities is generally accurate.

- **CPU-usage tests** are used to monitor how the CPU is being used during a specific activity. Monitoring CPU usage may be useful in finding total CPU requirements or errant processes. As with memory, CPU-usage monitoring is typically most useful if the activity being observed spawns an independent server-side process. Total CPU usage should return to pretest levels between test iterations. If this doesn't occur, analysis should be done to determine the cause of this anomaly using an appropriate diagnostic tool for the activity or application causing the CPU problem.

  CPU usage can generally be monitored with the same diagnostic tools used for monitoring memory. Because monitoring CPU usage is often CPU intensive, the total CPU usage won't be reflected truthfully, but the difference in CPU utilization between different activities is generally accurate.

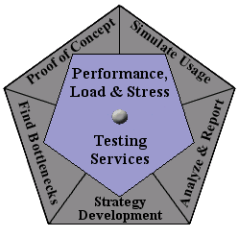
Load Tests

Load tests simulate expected real-world loads, from best-case to worst-case scenerios, and are used to determine what the actual performance of the system will be when the application is in production. If the tests are designed properly, end-to-end response times will represent what actual users will experience when accessing the system. It's also useful to monitor all of the system resources under this load to determine if they're adequate to support the expected user load. Load tests will also highlight any performance issues that users would experience and thus enable stakeholders to make informed decisions about the readiness of the application to be put into production.

As I've discussed in previous articles, end-to-end system response-time tests are the first and last tests to be executed during a performance-testing engagement. Scalability and component-based tests are generally conducted in the middle of the testing engagement to help pinpoint and tune specific bottlenecks uncovered by response-time tests. It's important to analyze early tests closely to ensure that the tests are executing properly and providing valid results; these tests often uncover "low-hanging fruit" and "quick fixes."

Response-Time (User-Experience) Tests

Response-time tests simulate actual users interacting with the system and measure the time between a



user action and the completed response to the user; in other words, they gather user-experience measurements. It's a good practice to start with low user-loads, possibly even partial scenarios, and scale up incrementally to observe patterns in performance. Eventually, the entire user community will be modeled at peak expected user-load levels to validate acceptable performance from the user's perspective. Early tests often uncover major performance issues, generally having to do with environment configuration, such as configuration settings that were left as defaults or ports that were left closed on a firewall. Response-time tests normally provide the results used to make "go-live" decisions about the application.

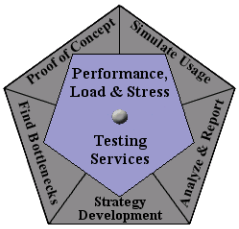
Scalability Tests

Scalability tests are used to determine how many real users can access the system before the system response time becomes unacceptable. Scalability tests are often the exact same tests as used to determine end-to-end system response time, executed with slowly increasing numbers of virtual testers. Ultimately, a system should scale to at least 125% of the expected peak user load before response time degrades "ungracefully." We'll discuss the relationship between response time and scalability in detail in Part 10 of this series. The results gathered from scalability tests are most useful for capacity planning.

Component-Based Tests

The three areas of component-based tests discussed in the low-load section also apply under load scenarios. Component-based tests executed at a low load should be executed again under load, and the results should be compared. A couple of additional component-based measurements that can add value under load are throughput tests and bandwidth tests.

- **Throughput tests** are generally executed against load balancers and Web servers for capacity-planning purposes. Many common load balancers and Web servers come with diagnostic tools to measure throughput. If so, these tools can be used to monitor throughput of these devices accurately without the adverse effects of using a separate monitoring tool while your response-time tests are executing. If these tools aren't included with the particular component being used, network bandwidth measuring tools can be used in combination with load-generation tools to measure throughput. The results of these tests are useful in determining if the system's limitations are based on the capabilities of the specific component being monitored. Once configured properly, Web servers and load balancers are rarely the cause of bottlenecks in multitier systems.
- **Bandwidth tests** are very similar to throughput tests, except that *bandwidth* generally refers to networks rather than systems. Network bandwidth is typically monitored either using network sniffers or through switches/routers/gateways. Once again, this monitoring should occur while response-time tests are being executed. The resulting measurements are useful in determining if the system's environment is adequate to support itself. Network bandwidth utilization should never be over 75%. These tests are most useful when executed in the production environment with all other systems in the environment operating normally.



Heavy-Load Tests

Heavy-load tests are executed at greater-than-expected user loads, generally far heavier than a system is ever expected to have to handle. They're used not to determine *if* a system will fail, but where it will fail first, how badly, and why. Answering the *why* question can help determine whether a system is as stable as it should be. The majority of significant deficiencies in the system will already have been identified during the execution of load tests, so this phase deals more with assessing the impact on performance and functionality under an unexpectedly heavy load. These tests are designed to find subtle performance issues, such as minor memory leaks, caching, and database locking. Heavy-load scenarios will also identify system bottlenecks not previously noticed, which may be found to be partially responsible for earlier identified problems.

Countless possible stability-focused tests can be executed, but the most common are spike, stress, and hammer tests.

Spike Tests

Spike tests use real-world distributions and user communities but with extremely fast ramp-up and ramp-down times. It's common to execute spike tests that ramp up to 100% or 150% of expected peak user-load in 10% of the normal ramp-up time. These tests are generally only executed after several rounds of tuning. If all components of the system continue to function normally, no matter how slowly, they pass a spike test.

Stress Tests

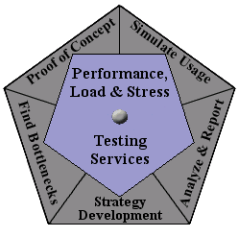
Stress tests use real-world distributions and user communities but under extreme conditions. It's common to execute stress tests that model 100% of expected peak user-load sustained over 8 to12 hours, and 150% (or more) of expected peak user-load with normal ramp-up and ramp-down times. These tests are generally only executed after several rounds of tuning. If all components of the system continue to function normally, with reasonable response times, and recover properly after the test execution ends, they pass a stress test.

Hammer Tests

Hammer tests bear little or no resemblance to real-world distributions and user communities. These tests take all existing load-generation scripts and methods, eliminate user think times, and increase load until failure occurs. These tests are designed to find the breakpoints in a system so that appropriate risk-mitigation strategies can be developed from those breakpoints. These tests are generally only executed after several rounds of tuning. There are no pass/fail criteria for hammer tests, as the intent is to make the system fail.

Specialty Tests

Specialty tests are most commonly "black-box" and/or "white-box" tests of specific system components used to identify and tune the *why* behind the performance issues identified during load and heavy-load testing. These tests are generally requested by the developers or architects tuning the



system and are designed to address a specific bottleneck or performance issue that requires the gathering of more information. Specialty tests generally bear no relationship to the user community model employed for other types of tests; they're used for iterative tuning of very specific components but don't generate reportable measurements.

Specialty tests may be necessary but should nonetheless be limited. It's important to develop these tests quickly and efficiently. Making specialty tests as simple as possible saves time and ensures repeatable results.

Black-Box Tests

Black-box testing has traditionally involved testing a system as a whole; however, with the advent of multitiered systems, black-box testing has come to refer also to testing tiers or components of a total system. Tests that focus on a specific tier can determine whether the detected bottleneck resides in that tier only. Black-box tests generally don't directly identify a specific bottleneck. Instead, diagnostics must be performed on the tier while generating a load to find the actual bottleneck.

White-Box Tests

White-box testing treats the system as a collection of many parts. During white-box testing, diagnostics may be run on the server(s), the network, and even on clients. This allows the cause(s) of bottlenecks to be identified much more easily and addressed. White-box testing can go as deep as individual lines of code, database query optimization, or memory management of segments of code.

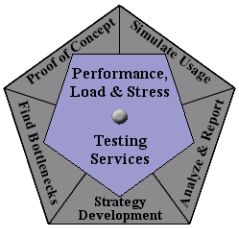
White-box testing requires much greater technical knowledge of the system than does black-box testing — it requires knowledge about the components that make up the system or tiers, how they interact, the algorithms involved with the components, and the configuration of the components. The tester must also know how to perform diagnostics against these components, and which diagnostics are called for. White-box tests are often difficult and time consuming to develop and execute. For this reason, white-box tests should be used only when there's a high expectation that they'll yield noticeable performance improvements.

Presenting Results to Stakeholders

There are three basic methods for reporting performance test results: text explanations, tables, and charts/graphs. In reporting the results of many performance tests, I've found several reporting methods that have been well received and intuitively understood by client stakeholders and that are good for many types of tests. I'll share these with you here, with the intention of giving you a starting point for thinking about how to present your own results. If you find that in your particular situation these methods don't sufficiently show or explain the point you want to make, please supplement these with text, tables, and/or graphs of your own design.

Text Explanations

All results should have at least a short verbal summary associated with them, and some results are best or most easily presented in writing alone. All of the tables and charts discussed in the next two sections



deserve accompanying text to accentuate the highlights of the graphic. However, you'll see that there's not a table or chart for every type of test discussed above. Specifically, some component-based tests and heavy-load tests, and all specialty tests should be explained with text exclusively rather than with tables or charts of data. These types of tests generally aren't executed so that the actual results can be included in formal reports but rather to find a single specific value, determine the cause of an anomaly, or evaluate a "what-if" scenario. If valuable data is obtained that's relevant to production and involves an issue that's not fixed on the spot, then the reason for the test, the way the test was executed, and the results and/or recommendations reached should all be described in a formal report.

Valuable Tables

In general, most people would rather view data and statistics in graphical form instead of in tables. But in some cases, tables are the most efficient way to show calculated results or *all* of the data. I recommend using tables sparingly in reports but including the tabular form of the data used to create charts and graphs as an appendix or attachment to a report, so that interested stakeholders can refer to it if they're interested.

Results from the following types of tests are well represented in a tabular format:

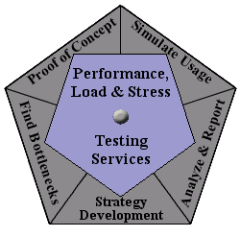
- baseline
- benchmark
- scalability
- any other user-experience–based test

The first type of table that can deliver value to stakeholders is the table of data used to create the performance report output chart, as shown in Figure 1. You've seen this type of table in Parts 5, 6, and 7 of this series. The **performance report output table** is well suited for displaying results from all four of the types of tests listed above.

CmdID	NUM	MEAN	STD DEV	MIN	50th	70th	80th	90th	95th	MAX
Home Page	99	4.53	1.47	3.33	4.08	4.57	4.87	5.77	7.99	11.05
Page1	100	2.94	0.85	2.26	2.59	2.91	3.48	4.08	4.65	6.44

Figure 1: Performance report output table

The other tables that complement the performance report output table are the **consolidated response time by test execution table** (Figure 2) and the **summary comparison table** (Figure 3). The latter is only useful for user-experience–based results. Part 9 of this series will discuss the creation and interpretation of these tables.



95th Percentile Time Comparison								
Page / Users	LAN 1	LAN 50	LAN 100	LAN 150	LAN 200	128 kbs 100	56.6 kbs 100	28.8 kbs 100
Home Page	5.72	0.60	1.93	2.94	46.48	7.81	9.02	16.92
Page 1	3.85	0.28	0.36	0.26	17.95	4.51	9.90	19.42
Page 2	0.21	0.20	2.37	0.16	8.53	2.89	6.24	12.12
Page 3	4.99	1.07	3.31	3.70	29.90	28.88	10.72	19.87

Figure 2: Response time by test execution table

Summary Comparison								
Statistic / Users	LAN 1	LAN 50	LAN 100	LAN 150	LAN 200	128 kbs 100	56.6 kbs 100	28.8 kbs 100
Times Recorded	177	175	175	176	176	176	169	169
Times Under Goal	146	160	132	133	6	78	47	30
% Times Under Goal	82.5%	91.4%	75.4%	75.6%	3.4%	44.3%	27.8%	17.8%

Figure 3: Response time summary comparison table

Valuable Charts/Graphs

1. Charts and graphs are normally the preferred method of data presentation. If you're not comfortable with creating charts and graphs, I recommend reading the work of Edward Tufte, PhD. Dr. Tufte has dedicated his career to accurate graphical presentation of quantitative information.

Of the tests we've discussed, the following are best represented with charts and/or graphs:

- baseline
- benchmark
- scalability
- response-time
- transaction-rate
- CPU-usage
- memory-usage
- throughput
- bandwidth

The two most universally valuable charts are the performance report output chart and the response-vs.-

time scatter chart, which we've discussed in several previous articles.

The **performance report output chart** (and accompanying table) is ideally suited to graphically display the results from baseline, benchmark, scalability, and any other user-experience–based tests. This chart (shown in Figure 4) can also be used with transaction-specific component-based tests but isn't useful for displaying any heavy-load or non-transaction-based component measurement, like CPU usage.

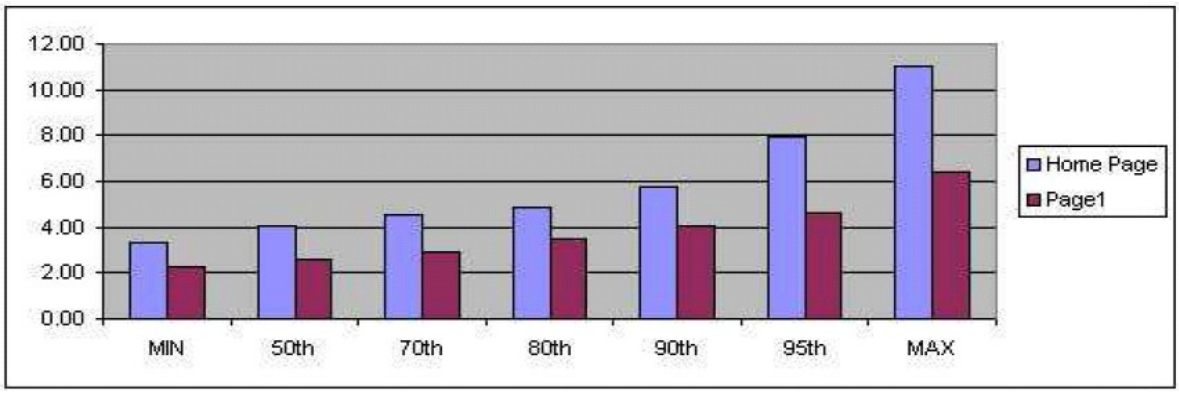


Figure 4: Performance report output chart

The **response-vs.-time scatter chart** is also well suited for displaying patterns in the results of baseline, benchmark, and scalability tests; response-time tests identifying outliers; transaction-specific component-based tests; and all of the heavy-load tests. Figure 5 is a customized version of the response-vs.-time scatter chart that represents each activity (in this case, page load time) with a different color or symbol. This is easily done in Excel by identifying the data about each page load time as a separate series while generating the chart. As you can see, this chart is all about identifying patterns. By looking at the chart, you can quickly and easily determine which activities take the most time, if activities take more or less time as the test execution progresses, if there are peaks and/or troughs in overall performance, and the like. This chart is the one I always turn to when I'm beginning my search for a bottleneck.

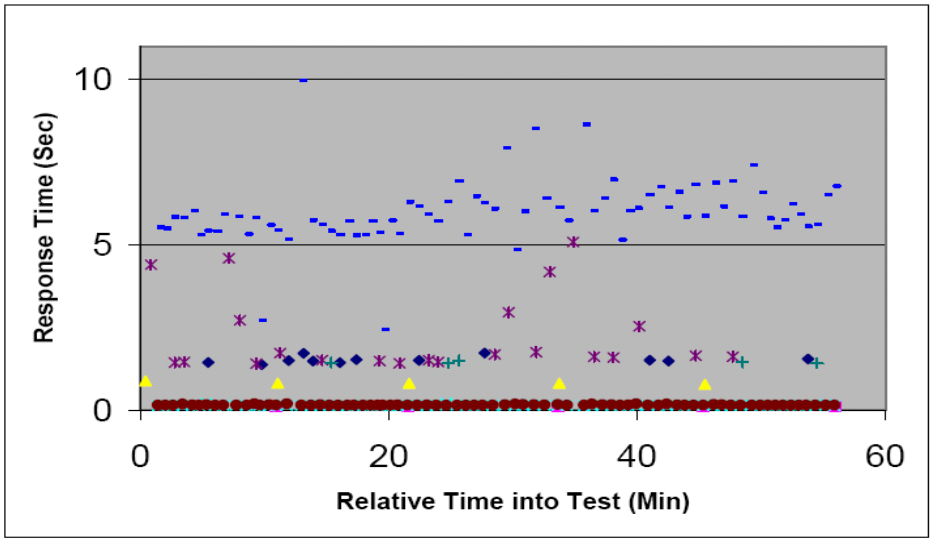


Figure 5: Customized response-vs.-time scatter chart

Another valuable chart is the **component performance chart**, which collects component-based measurements such as CPU usage. (Note that you must capture this type of performance data on the server, not the client machine or the master station.) Figure 6 is an example of this kind of chart. This particular chart comes from Perfmon, the performance monitoring utility that comes on the server versions of all Microsoft operating systems. It's very similar to the chart generated by PerfMeter, which comes on Solaris operating systems, and the charts that are generated by TestManager when "View resources" is selected. There are literally dozens of performance-monitoring tools on the market that produce such charts, which are all quite similar but differ in minor details.

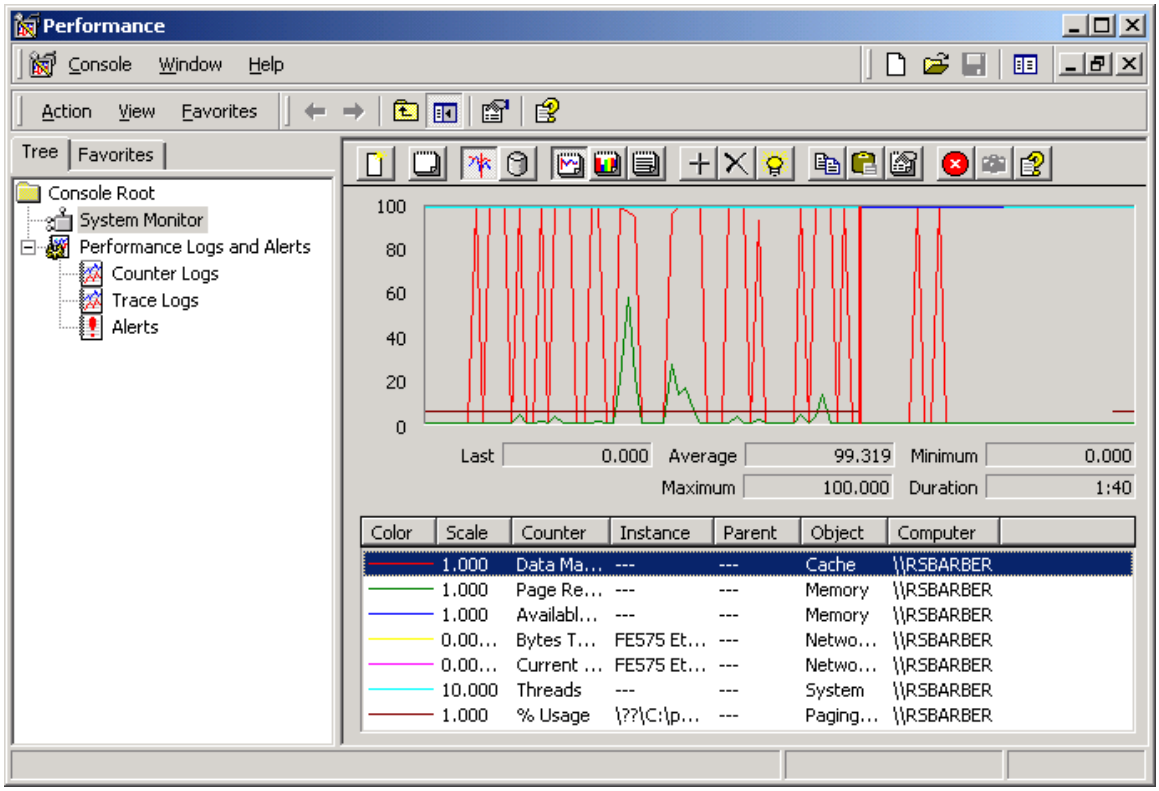


Figure 6: Component performance chart

A valuable chart that's related to the performance report output chart is the **response time summary comparison chart**, which I'll show you how to create in Part 9. This chart, as shown in Figure 7, summarizes graphically the number of measurements that met specific predetermined goals during performance tests with various parameters. It's valuable for presentation of all user-experience–based results.

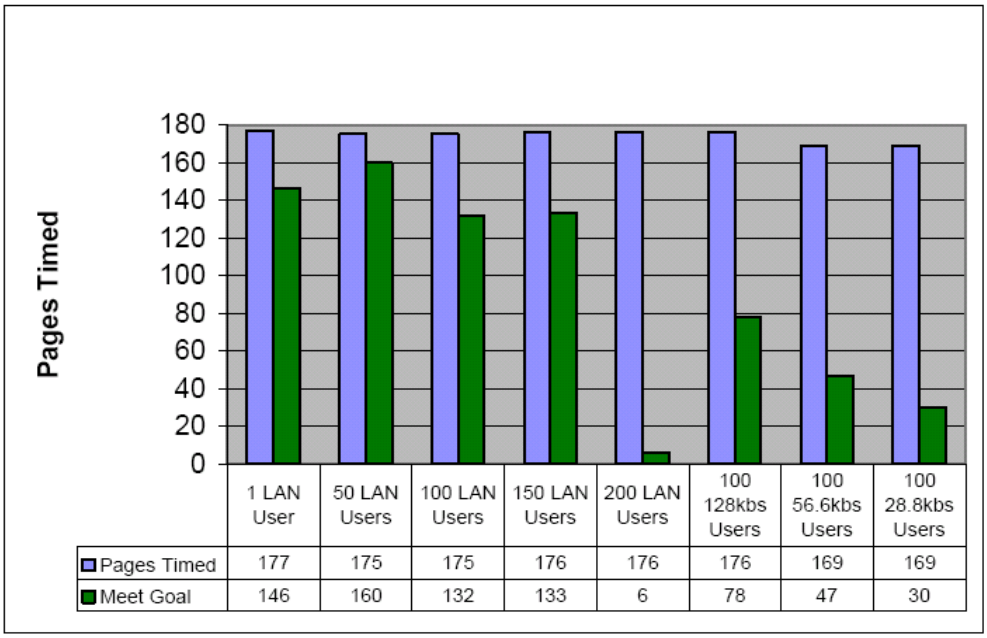


Figure 7: Response time summary comparison chart

Figure 8 shows another summary chart for collection of user-experience test measurements, called the **response time by test execution chart**. Again, I'll show you how to create this type of chart in Part 9.

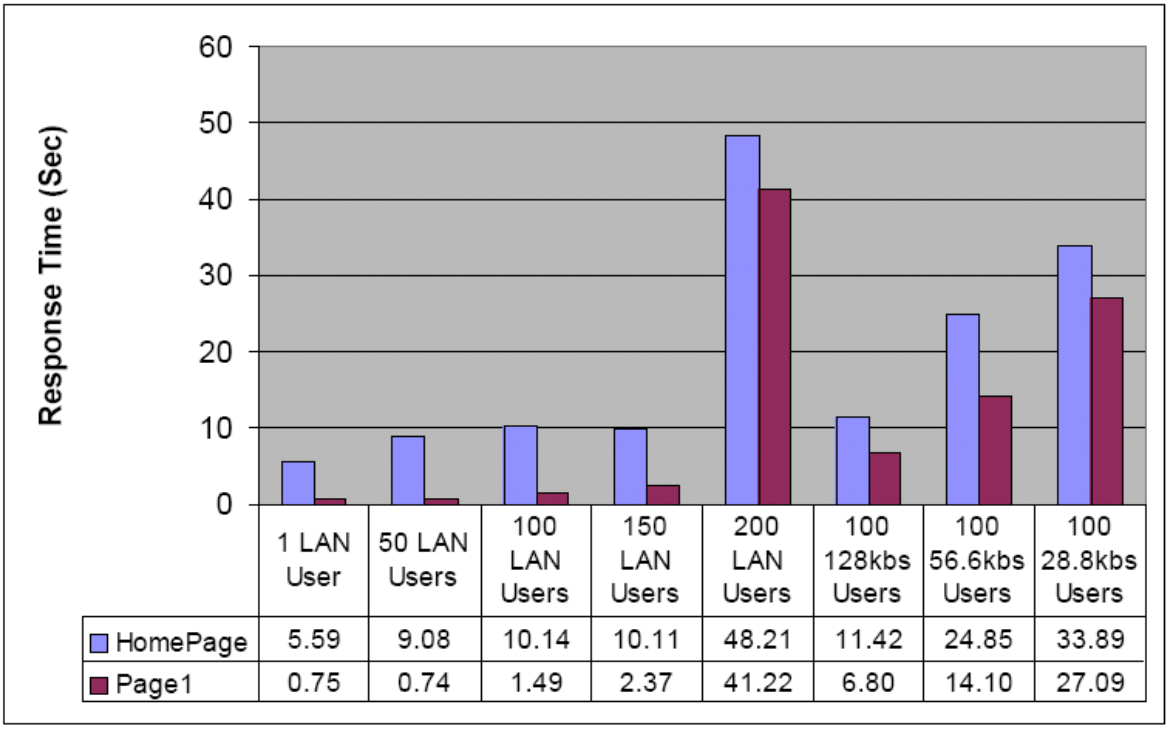
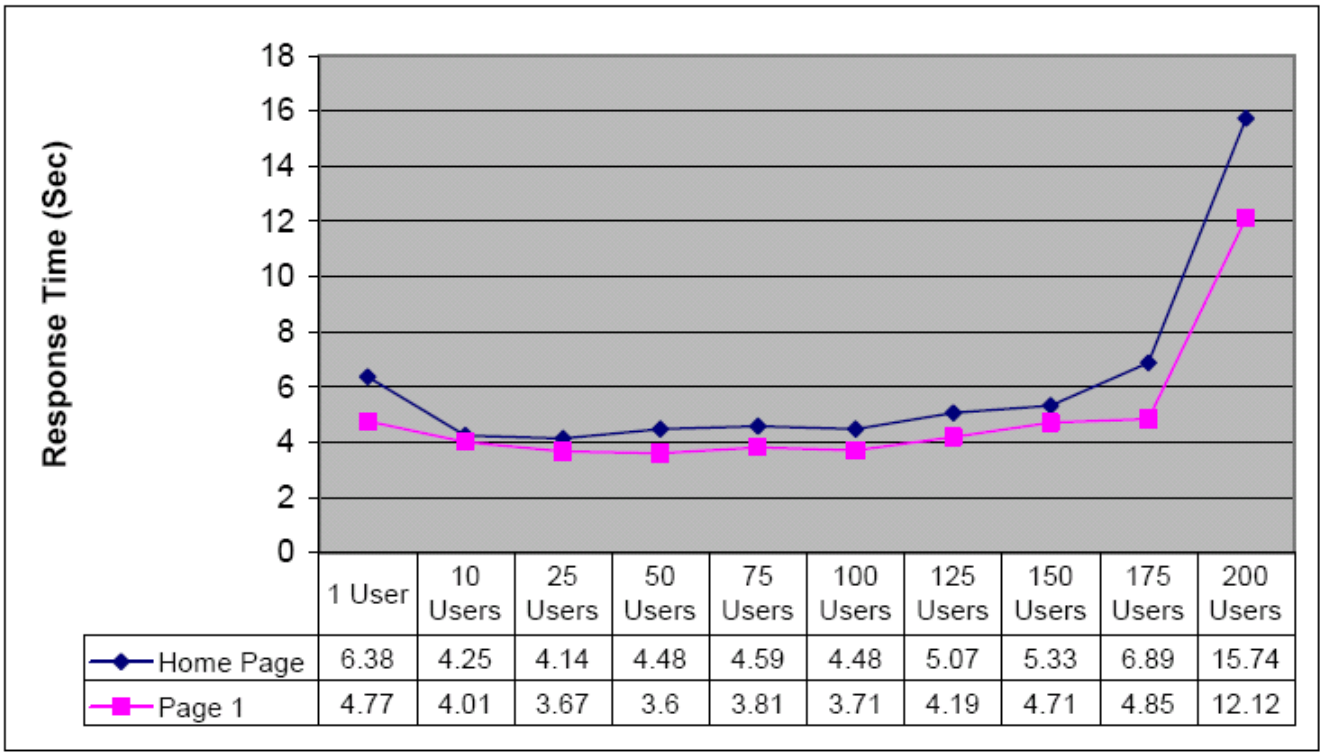


Figure 8: Response time by test execution chart

Finally, the chart that I've found to be absolutely the most useful in reporting to stakeholders is the **degradation curve chart**, shown in Figure 9. This will be the sole topic of Part 10. It probably won't ruin the suspense if I tell you that this chart is also used to summarize user-experience and scalability

measurements across multiple test runs and is particularly useful for identifying the user load at which performance degrades quickly, or ungracefully.



	1 User	10 Users	25 Users	50 Users	75 Users	100 Users	125 Users	150 Users	175 Users	200 Users
Home Page	6.38	4.25	4.14	4.48	4.59	4.48	5.07	5.33	6.89	15.74
Page 1	4.77	4.01	3.67	3.6	3.81	3.71	4.19	4.71	4.85	12.12

Figure 9: Response-time degradation curve chart

Now You Try It

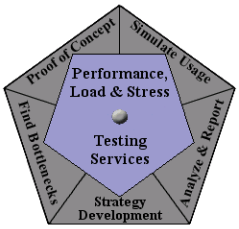
As you may have noticed, this article has presented many concepts but not many that lend themselves to scripted exercises. Those concepts that do are the topics to be covered in the next two articles. If you want an exercise, I'd recommend that you import some of your own data from previous tests into Excel and experiment with presenting it in different charts, graphs, and tables. When you find one that you think really highlights the point you want to make about the data, go find a manager in your company and ask him or her to tell you what he or she thinks the chart/graph or table means. If he or she immediately recognizes the point you were trying to make, you've created a valuable chart or table.

Summing It Up

This article has described the most common types of performance-related tests that deliver value to stakeholders, and some reliable ways to present the information obtained from those tests. The most important ideas to take away from this article are these:

- Always make certain that you're performing only the tests that will provide the most value.
- Always try to present the results of those value-providing tests in a format that highlights the key points of the test.

No matter which tests you execute or how you present the data, remember, if the test doesn't ultimately



lead to determining or improving the user's experience you should think twice before even developing it.

Acknowledgments

- The original version of this article was written on commission for IBM Rational and can be found on the IBM DeveloperWorks web site

About the Author

Scott Barber is the CTO of PerfTestPlus (www.PerfTestPlus.com) and Co-Founder of the Workshop on Performance and Reliability (WOPR – www.performance-workshop.org). Scott's particular specialties are testing and analyzing performance for complex systems, developing customized testing methodologies, testing embedded systems, testing biometric identification and security systems, group facilitation and authoring instructional or educational materials. In recognition of his standing as a thought leading performance tester, Scott was invited to be a monthly columnist for Software Test and Performance Magazine in addition to his regular contributions to this and other top software testing print and on-line publications, is regularly invited to participate in industry advancing professional workshops and to present at a wide variety of software development and testing venues. His presentations are well received by industry and academic conferences, college classes, local user groups and individual corporations. Scott is active in his personal mission of improving the state of performance testing across the industry by collaborating with other industry authors, thought leaders and expert practitioners as well as volunteering his time to establish and grow industry organizations. His tireless dedication to the advancement of software testing in general and specifically performance testing is often referred to as a hobby in addition to a job due to the enjoyment he gains from his efforts.

About PerfTestPlus

PerfTestPlus was founded on the concept of making software testing industry expertise and thought-leadership available to organizations, large and small, who want to push their testing beyond "state-of-the-practice" to "state-of-the-art." Our founders are dedicated to delivering expert level software-testing-related services in a manner that is both ethical and cost-effective. PerfTestPlus enables individual experts to deliver expert-level services to clients who value true expertise. Rather than trying to find individuals to fit some pre-determined expertise or service offering, PerfTestPlus builds its services around the expertise of its employees. What this means to you is that when you hire an analyst, trainer, mentor or consultant through PerfTestPlus, what you get is someone who is passionate about what you have hired them to do, someone who considers that task to be their specialty, someone who is willing to stake their personal reputation on the quality of their work - not just the reputation of a distant and "faceless" company.