An Oracle White Paper
February 2010

Rapid Bottleneck Identification - A Better Way to do Load Testing

ORACLE

Introduction

You're ready to launch a critical Web application. Ensuring good application performance is crucial, but time is short. How can you optimally test the application and still meet your deadlines?

Rapid bottleneck identification (RBI) is a new testing methodology that allows quality assurance (QA) professionals to very quickly uncover Web application performance limitations and determine the impact of those limitations on the end user experience. Developed through years of testing engagements across all types of platforms, the RBI methodology dramatically reduces load testing cycles while allowing more—and more thorough—testing. Using this approach, organizations can improve application quality, enhance the customer experience, and lower the cost of deploying new systems.

Performance Testing Defined

Performance testing can be roughly defined as "testing conducted to evaluate the compliance of a system or component with specified performance requirements." However, every application has at least one bottleneck, and few, if any, systems ever meet initial performance requirements. To reflect this reality, let's redefine performance testing as "testing conducted to isolate and identify the system and application issues (bottlenecks) that will keep the application from scaling to meet its performance requirements."

This philosophical shift in perspective—from testing as an evaluation to testing as an active investigation to isolate and resolve problems—is what drove the creation of the RBI methodology. RBI combines a comprehensive understanding of bottlenecks with a refined testing methodology that enables organizations to create highly scalable Web applications.

Understanding Bottlenecks, Throughput, and Concurrency

Before delving into the specifics of the RBI methodology, we must first establish a common understanding of bottlenecks—and where they are found—as well as draw a distinction between throughput and concurrency testing.

Bottlenecks—Key Performance Inhibitors

Any system resource—such as hardware, software, or bandwidth—that places defining limits on data flow or processing speed creates a bottleneck. In Web applications, bottlenecks directly affect performance and scalability by limiting the amount of data throughput or restricting the number of application connections. These problems occur at all levels of the system architecture, including the network layer, the Web server, the application server, and the database server. Historically, based on our experience testing actual customer applications, bottlenecks have been distributed across these components as shown in Figure 1.

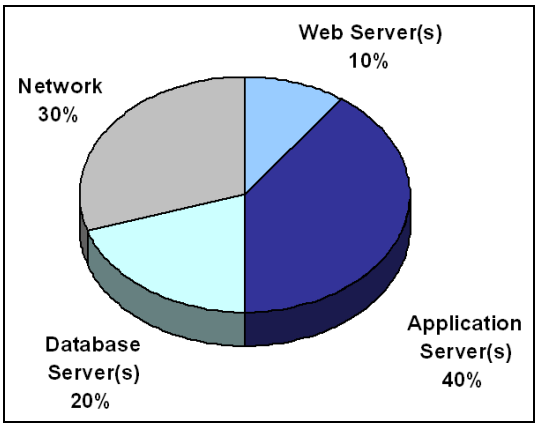

Figure 1. Estimated distribution of bottlenecks across the system infrastructure.

The Compounding Impact of Testing Complexity

The testing approach you choose directly impacts the difficulty of isolating and resolving bottlenecks. Unfortunately, too many testing procedures begin with complex usage scenarios where testers try to simulate exactly how the application will be utilized in production. This may involve running several different transactions to simulate different types of users who interact with the application in different ways. Unfortunately, this creates a significant testing roadblock because scenarios that are higher in complexity and involve multiple different transactions introduce more bottlenecks into the test, which makes it difficult to identify root causes.

For example, the graph in Figure 2 illustrates the test results of a standard e-commerce application that bottlenecked at approximately 2,000 concurrent users. In this sample test, the usage scenarios involved browsing, searching, and adding items to a shopping cart to complete a purchase. Although there were only three transactions being tested, each transaction interacted with all levels of the application architecture—and any one of them could have caused the bottleneck. To further complicate matters, the bottleneck could also have been caused by a system issue. Ultimately, the more variables involved in a test, the more difficult it is to determine the cause of the problem.

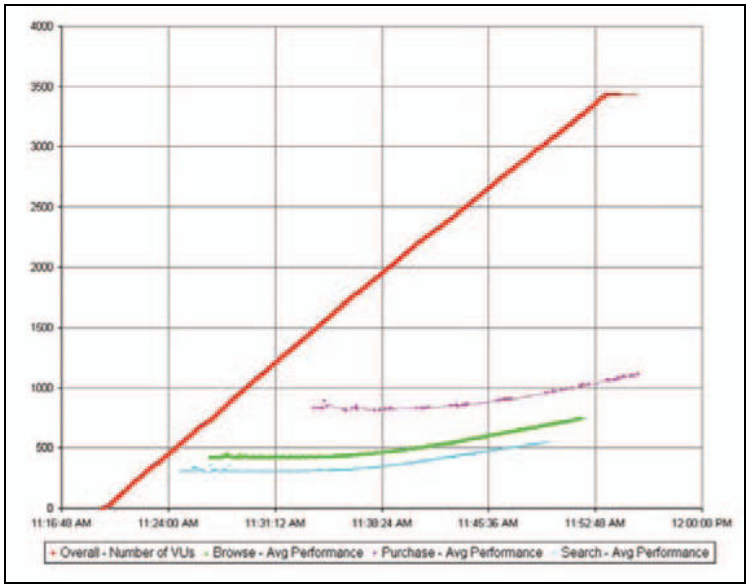

Figure 2. Graphical results of an e-commerce application test.

If the problem can be in any tier of the architecture, and the likelihood of it being in any one tier is not substantially greater than in any other, where else can you look for guidance?

Two Primary Issues: Throughput and Concurrency

Throughput is the amount of data flow a system can support, measured in hits per second, pages per second, and megabits of data per second. Concurrency is the number of independent users simultaneously connected and using an application. In our experience, a majority of all system and

application performance issues result from limitations in throughput. However, concurrency issues are also critical to application performance and can be even more difficult to isolate.

Testing for Throughput

Testing for throughput involves minimizing the number of user connections to a system and maximizing the amount of work being done by those users. This pushes the system and application to capacity so that all issues will be revealed.

For throughput testing at the system level, basic files can be added to the Web and application servers for testing purposes. The load test can then be set up to request these test files to assess maximum system throughput at each tier. Typically, testers use a large image file for bandwidth tests, a small text file or image for hit rate tests, and a very simple application page—a "Hello World" page, for instance—for page rate testing. If the system does not meet basic application performance requirements—just requesting these simple test pages—testing should cease until the system itself has been improved, either through tuning the settings, increasing the hardware capacity, or increasing the allocated bandwidth.

Throughput testing of the actual application then involves hitting key pages and user transactions in the application itself with limited delay between requests to find the page-per-second capacity limit of the various functional components. Obviously, the pages or transactions with the poorest page throughput need the most tuning.

Testing for Concurrency

On the system and application levels, concurrency is limited by sessions and socket connections. Code flaws and incorrect server configuration settings can also limit concurrency. Concurrency tests involve ramping up a number of users on the system and using realistic page-delay times at a ramp-up speed slow enough to gather useful data throughout the testing at each level of load. As with throughput testing, it is important to test the key pages and user transactions in the application under test.

The Difference Between Throughput and Concurrency Tests

The load generated from a 100 virtual user load test with 1-second think times is not equivalent to a 1,000 virtual user load test with 10-second think times. As Figure 3 illustrates, the two tests are identical in terms of throughput; however, in terms of concurrency they are vastly different.

Scenario	Throughput	Concurrency	Bottleneck Point
100 Users 1 Second/Page	Pages/Second = (100VU x 1 Page/VU) ÷ 1 Second = 100	100 Connections	50 Pages/Second
1,000 Users 10 Seconds/Page	Pages/Second = (1000VU x 1 Page/VU) ÷ 10 Seconds = 100	1,000 Connections	25 Pages/Second

Figure 3. 100 virtual user load test versus 1,000 virtual user load test.

In the first scenario, the throughput test, the application bottlenecked at 50 pages per second. In the second scenario, however, a concurrency test of the same transactions, the application bottlenecked at 25 pages per second. The only differences between these two tests were the number of users on the system and the length of time those users stayed on the pages. In the throughput test with fewer users and shorter page view delays, the application had more throughput capacity; the second test shows the application was limited in its concurrency. If the testers had checked only for throughput, the concurrency issue would not have been discovered until the application was in production.

Figure 4 and Figure 5 on the following page show the results of each test and highlight the importance of testing for both throughput and concurrency.

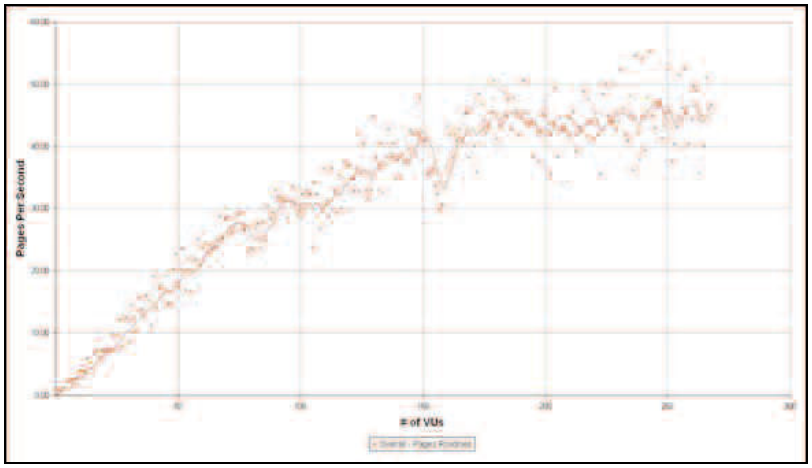

Figure 4. Throughput tests for 100 users with 1-second page views show a bottleneck at 50 pages per second.

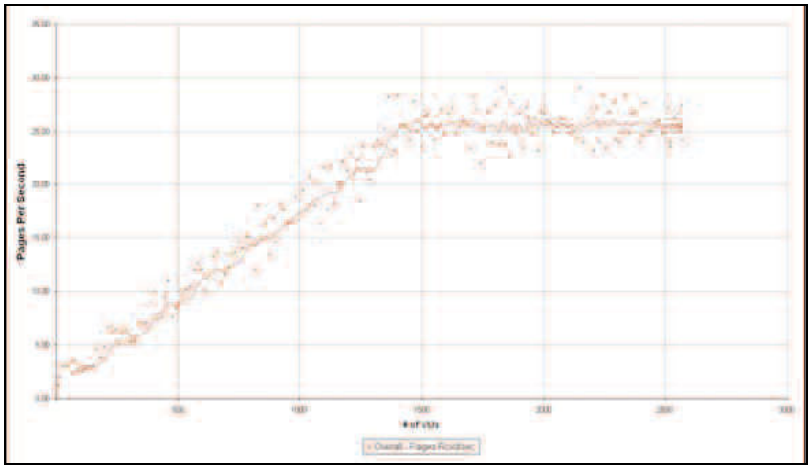

Figure 5. Concurrency tests for 1,000 users with 10-second page views show a bottleneck at 25 pages per second.

The RBI Testing Approach

Traditionally, performance testers focused on concurrent users as the key metric for application scalability. However, if a majority of application and system-level issues are found in throughput tests, a new approach is needed.

These three principles form the foundation of the RBI methodology.

- All Web applications have bottlenecks.
- These bottlenecks can only be uncovered one at a time.
- Focus should be placed where the bottlenecks are most likely to occur.

Although recognizing the importance of concurrency testing, the RBI methodology first focuses on throughput testing to root out the most common bottlenecks, followed by concurrency testing to assess performance under load conditions that reflect the actual number of users expected on the application. RBI testing also starts with the simple tests and then builds in complexity so that when an issue appears, all the other possible causes have been ruled out. Focusing on throughput testing, followed by concurrency testing, and using a structured approach to the test process ensures that bottlenecks are quickly isolated, which improves efficiency and reduces cost.

Benefits of RBI Testing

The RBI methodology enables rapid yet thorough testing that systematically uncovers all system and application issues—both simple and complex.

Reduce Testing Time

How much time can you save by focusing initially on throughput testing? Take an example of a system expected to handle 5,000 concurrent users, with users spending an average of 45 seconds on each page. If the application has a bottleneck that will limit its scalability to 25 pages per second, a concurrency test will find this bottleneck at approximately 1,125 users (25 pages per second at 45 seconds per page).

In the interest of not biasing the data, a typical concurrency test ramp up should proceed slowly. For example, you may consider ramping one user every five seconds. In this example, the bottleneck would have occurred 5,625 seconds or 94 minutes into the test (1,125 users at 5 seconds per user). However, to validate the bottleneck, the test would have to continue beyond that point to prove that the throughput was not climbing as users were added. A throughput test could have found this problem in less than 60 seconds.

Eliminate Initial Testing Complexity

Very often performance testing begins with overly complex scenarios exercising too many components at the same time, making it easy for bottlenecks to hide. The RBI methodology begins with system-level testing that can be carried out before the application is even deployed.

Improve QA Efficiency

The RBI methodology tests the simplest test cases first and then moves on to those with increased complexity. If the simplest test case works and the next level of complexity fails, the bottleneck lies in the newly added complexity. By uncovering bottlenecks using a tiered approach, you can quickly identify issues as well as isolate issues in components of which you have limited knowledge.

Enhance Testing Effectiveness with Knowledge Aggregation

The modular and iterative nature of the methodology means that when a bottleneck appears, all the previously tested components have already been ruled out. For instance, if hitting the home page shows no bottlenecks but hitting the home page plus executing a search shows very poor performance, the cause of the bottleneck lies in the search functionality.

RBI Testing for Common System Bottlenecks

Any performance testing should begin with an assessment of the basic network infrastructure supporting the application. If this basic system cannot support the anticipated user load on a system, even infinitely scalable application code will bottleneck. Basic system-level tests should be run to validate bandwidth, hit rate, and connections. Additionally, simple test application pages should be exercised—simple "hello world" pages, for instance.

The Application

After validating that the system infrastructure meets the most basic needs of the end users, turn to the application itself. Once again, start with the simplest possible test case.

If testing has progressed this far without uncovering system-level issues (or those issues have been resolved), any remaining problems are caused by the application itself. For example, if a test application page achieved 100 pages per second and the home page bottlenecks at only 10 pages per second, the problem lies in the overhead required to display the home page.

At this point, the test application page test provides two valuable pieces of information. First, because we know that the system itself is not the bottleneck, the culprit can only be the code on the home page. Second, we can see how much tuning the home page could improve performance. The difference between the performance of the test application page (100 pages per second) and the home page (10 pages per second) determines the maximum performance improvement tuning could provide. Likewise, multipage transactions can be assessed by breaking down the performance of individual

pages in the transaction and evaluating how each contributes to the performance of the overall transaction.

Since any real-world application page likely requires more processing power than a "hello world" test page, it is reasonable to expect some drop-off in performance. However, the greater that drop-off, the greater the need for—and potential gain from—tuning. It is also important to note that if the drop-off between the test page and an actual application page is not substantial and the performance is still insufficient to meet the needs of the application user base, you need to add more hardware capacity.

Up to this point no mention has been made of page-response times. Although response times are a key metric of overall performance, response times will be the same for one user as they are for 1,000 or 100,000 users unless a bottleneck is encountered. So in this methodology, response times are only useful as an indicator that a bottleneck has been reached (if response times begin to spike) or as failure criteria (if response times exceed some predefined threshold), with poorly performing pages (those that experience errors or high response times) most in need of code optimization.

RBI Testing for Application Bottlenecks

As with the system-level testing, the RBI methodology begins application testing with the simplest possible test case and then builds in complexity. In a typical e-commerce application, you would test the home page first and then add pages and business functions until complete, real-world transactions are being tested, first individually and then in complex scenario usage patterns. As steps are added, any degradation in response times or page throughput will have been caused by the newly added step, making it easier to isolate what code needs to be investigated.

Once each of the business functions and transactions has been tested and optimized (as necessary), the transactions can be combined into complete scenario concurrency tests. These concurrency tests must focus on two key components. First, the concurrency test must accurately reflect what real users do on the site—browse, search, register, login, and purchase. Second, the steps in those transactions must be performed at the same pace as real-world visitors with appropriate "think times" between each step. This data can be gathered with a Web logging tool that exposes session length, pages viewed per session (to determine the user pacing), and percentages of pages hit (to determine the actual business functions used).

Once the test has been designed from real-world data—or educated assumptions for an application not yet deployed—the test must be executed in a way that gathers valuable information at various user load levels. If the site is expected to handle 1,000 concurrent users, then it's important *not* to start those users all at once. Instead ramp your test slowly, adding one or more users at defined time intervals, until you reach 1000. This will allow you to determine overall performance at each level of user load and also make it easier to identify performance problems when they begin to occur.

Conclusion

The RBI methodology for load testing improves testing efficiency by focusing first on where bottlenecks most often occur—in the throughput. Once throughput has been thoroughly tested, you can test the system and application for concurrency to assess performance under realistic user loads. By following a structured approach from system testing to application testing and slowly, systematically introducing complexity into the test cases, you can quickly isolate bottlenecks and their associated root cause.

Although this paper focuses on methodology, it is important to point out that much of this process can and should be automated using an automated testing tool. Oracle Application Testing Suite is the centerpiece of Oracle Enterprise Manager for functional and load testing for Web and service-oriented architecture applications.

Rapid Bottleneck Identification –
A Better Way to do Load Testing
February 2010

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Oracle is committed to developing practices and products that help protect the environment