dynatrace.com

Differentiating Performance from Scalability | Dynatrace

9-11 minutes

Application performance is an area of increasing importance. We are building bigger and bigger applications. The functionality of today's applications is getting more and more powerful. At the same time we use highly distributed, large scale architectures which also integrate external services as central pieces of the application landscape. To optimize these applications we need profound knowledge of how to measure and optimize the performance of Enterprise applications. Before we delve into the technical details how to optimize performance we look at the important underlying concepts.

A solid understanding of the topic helps to manage the performance of an application more efficiently. It will help to make performance management appear less complex. As you understand the technical background, many things suddenly get much simpler and you will also see your efficiency in solving performance problems increase.

While we discuss fundamental principles of performance management, experienced readers will appreciate a detailed discussion of important performance concepts. Especially a solid understanding of different measurement techniques and their advantages and drawbacks as well as measurement overhead and measurement precision is key in interpreting measurement results.

A deep understanding of data collection techniques and data representation also helps us in selecting the proper tools for our analysis tasks.

Differentiating Performance from Scalability

Throughout this book, we'll be discussing performance and scalability: how to measure them, identify problems, and then optimize. And yet, many people use these terms as synonyms. People are likely to say that "the performance of the application is bad", but does this mean that response times are too high or that the application cannot scale up to more than some large number of concurrent users? The symptoms might be the same, but these are two quite different problems that describe different characteristics of an application.

When discussing performance we must carefully differentiate these two points. It's not always an easy task, because at the same time, they are interrelated and the one can, and often does, affect the other.

Defining and Measuring Performance

Application state determines the way in which requests are processed. The current load, the complexity of the request, and other application and system factors (like CPU or network usage) all impact application responsiveness. It is the characteristics of response that define application performance. More specifically, there are three basic measures of performance:

- Response time: This is the most widely used metric of performance and it is simply a direct measure of how long it takes to process a request.
- Throughput: A straightforward count of the number of requests that the application can process within a defined time interval. For Web

applications, a count of page impressions or requests per second is often used as a measure of throughput.

- System availability: Usually expressed as a percentage of application running time minus the time the application can't be accessed by users. This is an indispensable metric, because both response time nor throughput are zero when the system is unavailable.

  Performance can also be defined by resource requests and by measuring resource re-quests in relation to throughput. This becomes a key metric in the context of resource planning. For instance, you may need to know which resources are needed to achieve full application scaling, which is a matter of knowing the frequency of requests for each resource type.

  Resource consumption is becoming even more important for applications deployed in "elastic" environments, such as the cloud. With these larger scale and geographically-dispersed applications, scalability and performance become inextricably linked that can most easily be described from a resource-centric point of view.

  Remember, even the most amply-supplied and equipped applications have limited resources. At the same time, increasing system load will make unequal demands on whatever resources are required. This becomes the ultimate, real-world, measure of performance.

  The more requests users send to the application, the higher the load. The results can be plotted as simple as basic mechanics. So much as friction or gravity will ultimately bring a moving body to rest, as load increases, performance decreases.

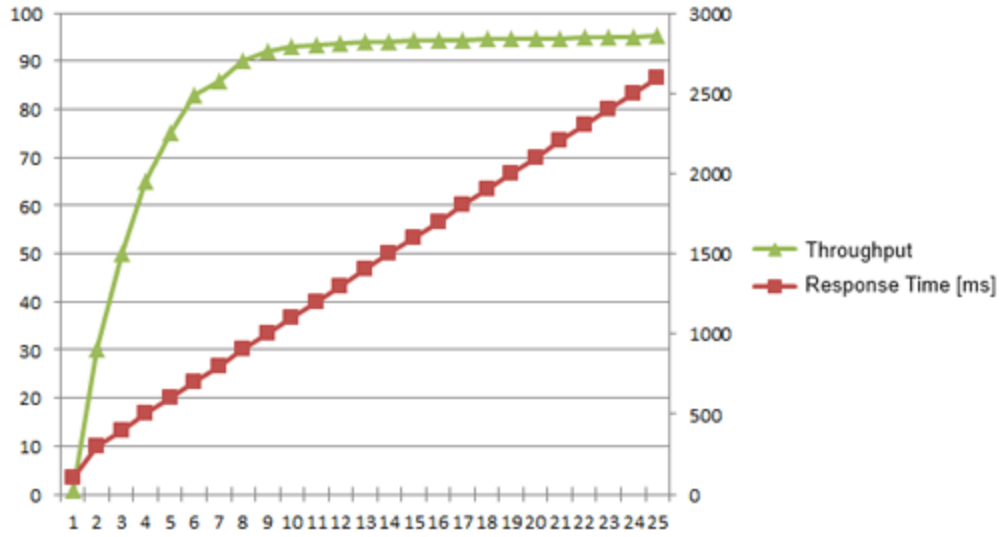

Figure 1.1: Increasing load affects the response time and throughput of an application

In the same way, application performance will always be affected by resource constraints. Your users may experience these constraints as application instability, but the underlying cause can be any of the metrics that define application performance.

It is therefore vitally important to describe performance as a ratio of some measurable quantity—response, throughput, availability, or requests—over time. One is easily tempted to use a kind of "black-box" shorthand, such as

Response time is 2 seconds.

Don't do it! Without a properly descriptive context, this sort of statement is meaningless and ultimately useless. Instead, be verbose:

System response time is 2 seconds at 500 concurrent requests, with a CPU load of 50%, and a memory utilization of 92%.

We'll discuss tools for measuring performance as well as methods for dealing with these performance issues in the following chapters.

How to Enable Application Scalability

The ability to overcome performance limits by adding resources is defined as scalability. No matter how much hardware we have at a certain point we will see decreasing performance. This means increasing response times or a limit in throughput. Will adding additional hardware solve the problem? If yes, then we can scale. If not, we have a scalability problem.

Depending on the changes to the application topology we differentiate the following two scalability approaches:

Vertical Scaling or scaling up a single node: Adding hardware resources to an existing node. This is often used in virtualized environments where it's possible to expand system resources dynamically. It has the clear advantage that it requires no changes to the application architecture. At the same time the level of scalability will always be hardware constrained.

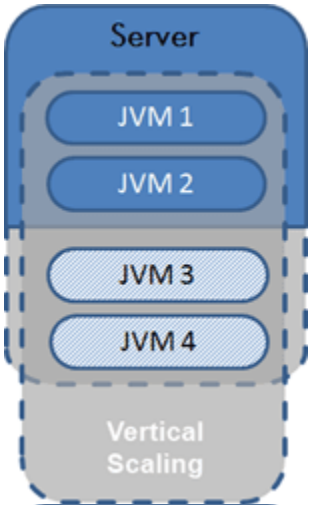

Figure 1.2: Vertical scaling by adding hardware resources on to existing servers

Horizontal Scaling or scaling out by adding nodes: Dispatching requests among additional nodes is the preferred scaling solution when it is easier to add additional nodes than increasing the resources of a single node. This is, especially true in cloud-based environments (see figure below). Eventually we have to scale horizontally anyways as a single machine always has limited

capacity. An additional advantage is that inexpensive nodes provide both increased failover capacity and improved application availability. The obvious disadvantage is that clusters must be designed so that nodes can be added and removed with ease.

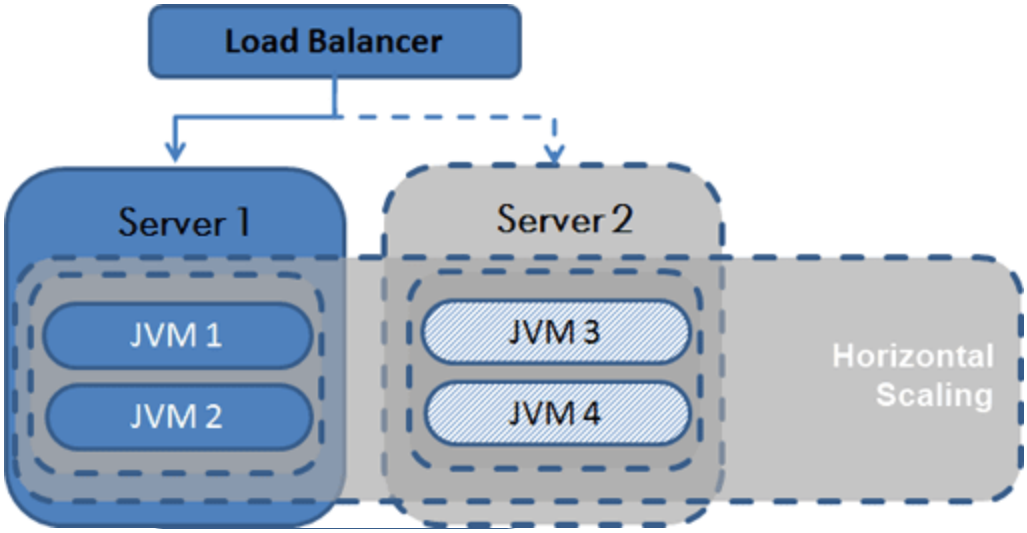

Figure 1.3: Horizontal Scaling by adding new servers, scaling out

Comparing these two scaling methods, Vertical Scaling is the simpler of the two and therefore more easily implemented. Horizontal Scaling requires the redistribution and synchronization of data across the new node configuration. Without proper planning, it can be expensive and disruptive to retrofit clusters for scaling out.

Will Scaling Solve Our Performance Problems?

Scaling an application by adding resources is the ideal case, but at some point, this will become too expensive. We must consider adding hardware, scaling out, or changing our architecture.

However frequently the scalability problem is not hardware related and adding hard-ware will not help. If we observe that resources are not overloaded then this usually indicates a synchronization-related problem, which is often related to serialized access to shared data rather than a performance problem. Adding resources will not help because this is not where the bottleneck is.

Let's take the example of updating an inventory. If we want to keep it consistent at all times, then only one process may update it at any given time. The duration of this processing indicates where our scalability limits are. If processing takes 200 milliseconds, the scalability threshold of our application is five requests per second– regardless of the available hardware.

If we want to achieve higher throughput we have to improve the overall performance rather than improving scalability. This could mean that we decide to give up consistency. We would place our inventory changes in a queue that are then updated at some later point in time. This is called "eventual consistency". With this approach it is however no longer possible to check whether a certain amount of an item is currently in stock.

As this example shows we really need good performance measures to guide us in making scalability decisions.