Chapter 1

Basic Performance Lingo

Let's begin our discussion of performance with a quick review of key performance terminology. We say "key" because we use this terminology throughout the book. If you're new to performance, you should read this chapter thoroughly. If you're an experienced performance analyst, you should at least glance through it for new terms or to be sure you understand our use of the more ambiguous performance terms. This chapter acquaints you with the concepts and the specific performance "lingo" used to define the concepts. We begin by defining the most basic measurement terminology including *load*, *throughput*, and *response time*. Next, we explore the terminology for performance optimization, including *path length*, *bottlenecks*, and *scalability*.

The terminology used in performance analysis may be new to you, but you already know the underlying concepts from everyday life. In this chapter, we apply performance terminology to familiar experiences. We use analogies based on "brick and mortar" stores in our neighborhoods to describe performance concepts found in the virtual world of web site software. These analogies also demonstrate how performance terminology really describes everyday reality.

Measurement Terminology

Load: Customers Using Your Web Site

Traditional Store

Let's consider a traditional brick and mortar bookstore. A traditional bookstore serves customers and contains a certain number of customers at any point in time. If we use the store's security camera to take a snapshot of the sales floor at some point during the day, we get a picture similar to Figure 1.1.

In this picture, we see some of the customers browsing the shelves, while others interact with the store clerks. The customers ready to make purchases go to the

clerks operating the cash registers. Other customers needing assistance with a book selection go to the clerk at the information desk.

The bookstore frequently contains more customers than clerks. Specifically, the bookstore contains more customers than cash registers. Intuitively, we know it usually takes some browsing time on the customer's part to pick out a selection for purchase, so we don't need a clerk for every customer.

On-Line Store

An on-line store also serves customers, though these customers are represented by requests to the web site instead of a physical presence in a store. The on-line store uses computing resources to handle customer requests (the electronic equivalent of our bookstore clerks).

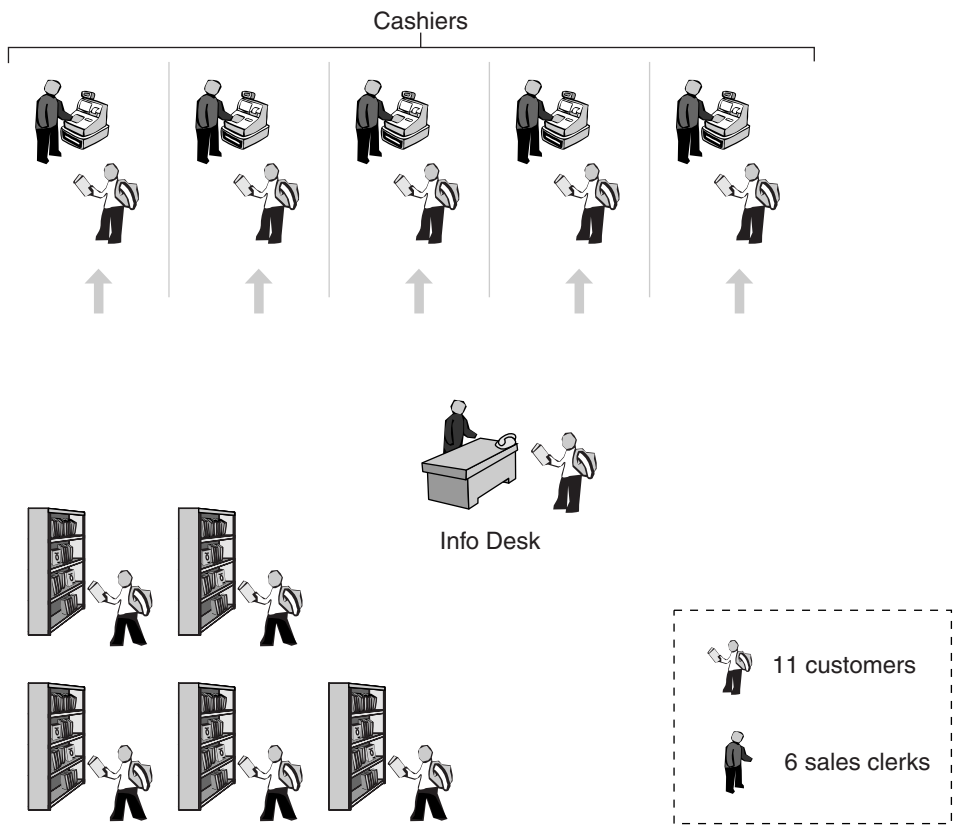

Figure 1.1 Customers in a traditional bookstore

Surprisingly, the customers using our on-line bookstore behave much like the customers visiting a brick and mortar bookstore. The on-line customers request pages from our web site (like asking the clerk at the info desk for help), and then spend some amount of time looking at a given page in their web browsers before making their next request. As they do in our brick and mortar store, the on-line customers may browse for a while before they make a purchase and often they make no purchase at all.

Figure 1.2 shows an on-line bookstore with a total of 11 customers: 2 with requests being handled by computing resources and 9 browsing recently created web pages. As this figure shows, at any point in time, an on-line store typically has more customers than the number of requests being handled. In fact, for some web sites, the customers reading pages far exceed those actively making requests to the web site.

So how does this all relate to load? *Load* is all of the customers using your web site at a point in time. Load includes customers making requests to your web site as well as those reading pages from previous requests. For example, the customer load in our bookstore snapshot in Figure 1.1 was 11 customers. (A performance expert might tell you the store is under an "11-customer load" at this point). In performance testing, we often "put the system under load." This means we plan to generate customer traffic against the system, often by using special test software to generate customers.

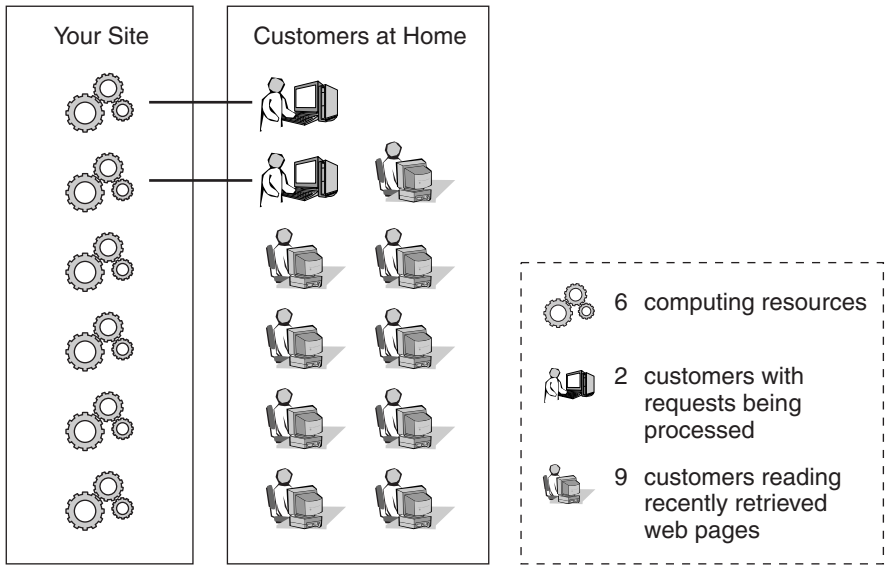

Figure 1.2 Customers using an on-line bookstore

Obviously, a term such as a *lightly loaded* system means the web site has few customers. Likewise, a *heavily loaded* system means the web site has many customers. Also, we often drop the term *customer* in the web space, preferring the terms *user*, *client*, or *visitor* instead. (Not all web sites actually sell things, so "customer" is not always appropriate.) However, we do need some way of differentiating between all clients and those clients actually making requests to our web site. We use the terms *concurrent load* and *active load* to make this distinction. Let's discuss these terms in more detail.[1]

Concurrent Load: Users Currently Using the Web Site

Traditional Store

As we noticed earlier, inside our traditional bookstore some customers look for books, while others interact with the clerks. For example, look at a security camera snapshot in Figure 1.3. At this point in time, the store contains 11 customers: 5 customers browsing, 1 requesting help from the info desk clerk, and 5 interacting with clerks at the cash registers to make purchases. *Concurrent load* refers to all of the customers in the store at a point in time, regardless of their activity.

On-Line Store

Web site visitors reading a previously requested web page resemble customers browsing in our traditional bookstore—they're using the web site, but not actively engaging it to satisfy a request. The concurrent load for your web site includes the customers browsing previously requested pages in addition to the active clients. Figure 1.4 shows an on-line bookstore with 6 active client requests, and 5 users browsing previously created pages for a total of 11 concurrent clients.

You may wonder why we care about the total users rather than just those making requests. Look at the traditional store example. All of the customers in the store use resources, even if they're not interacting with a clerk. For instance, the store owner provides floor space where browsing customers can stand as well as parking places out front for their cars. Likewise, web site visitors often require web site resources even when they're just reading a web page on their browsers. The web site often uses memory and other resources to keep information about users during their visit. (We expand this topic in Chapter 2's discussion of HTTP sessions.) Since any user potentially consumes resources, regardless of her current activity, we sometimes need to consider all of the users visiting our web site. This is concurrent load.

1. We know of no authoritative standard for defining concurrent vs. active clients. We prefer these terms. However, don't be surprised if you encounter others in your organization either using these terms differently, or using different terms altogether. In any case, make sure you *all* agree on definitions when planning your tests and your production web site.

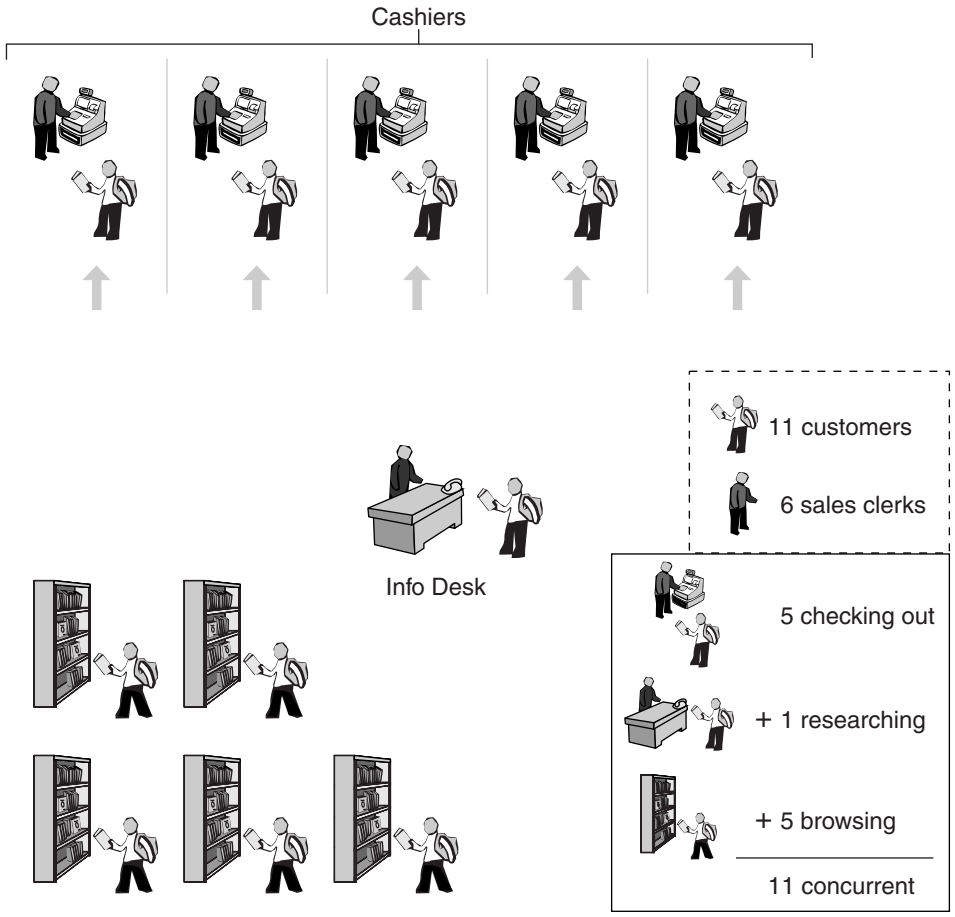

Figure 1.3 Concurrent users in a traditional bookstore

Active Load: Customers Making Requests of the Web Site

Traditional Store

Active load refers to customers currently making requests. In a brick and mortar store, the active customers are interacting with the sales staff. The bookstore may contain many customers, but only a subset of these actually is at the cash registers or at the information desk interacting with the sales clerks. We call these customers "active" because they actively want some service from the store to buy or find a book. Figure 1.5 shows a traditional bookstore with six active customers.

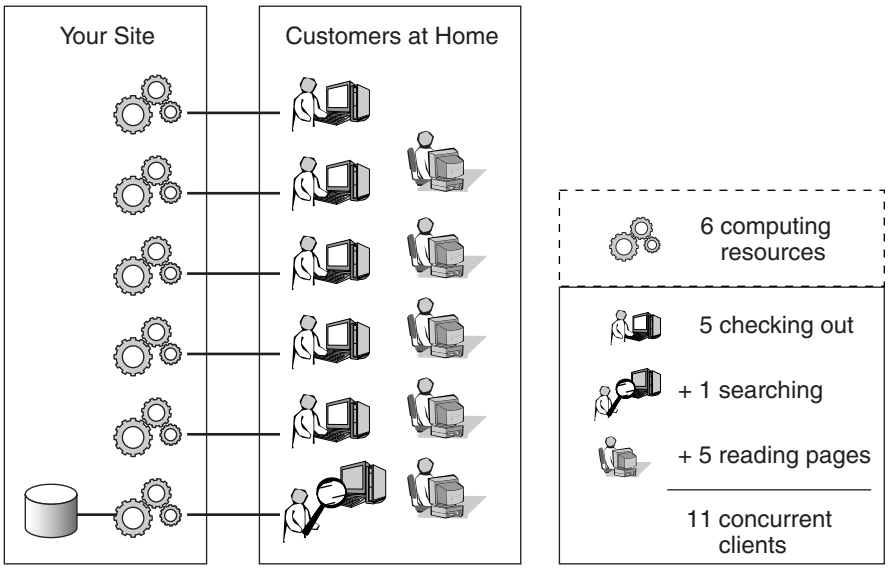

Figure 1.4 Concurrent web site load

On-Line Store

Much as they do in the traditional bookstore, on-line customers also make requests of the web site. These requests require web resources, particularly processing resources, to complete. For example, the on-line customer might request a search for books by a particular author, or request to purchase the items in an on-line shopping cart. See Figure 1.6 for an example. By definition, the term *active* applies to the user from the moment the request arrives at our web site until the requested information returns to the user.

Peak Load: Maximum Concurrent Web Site Customers

Traditional Store

After the bookstore opens, customer traffic rarely remains constant throughout the day. The store usually receives more customers during lunchtime or after school than in the morning or late at night. Over the course of a week, the store probably receives most of its traffic on Saturday. Over the course of a year, the Christmas holiday represents the store's overall busiest period, with the day after Thanksgiving being the busiest day of the year.

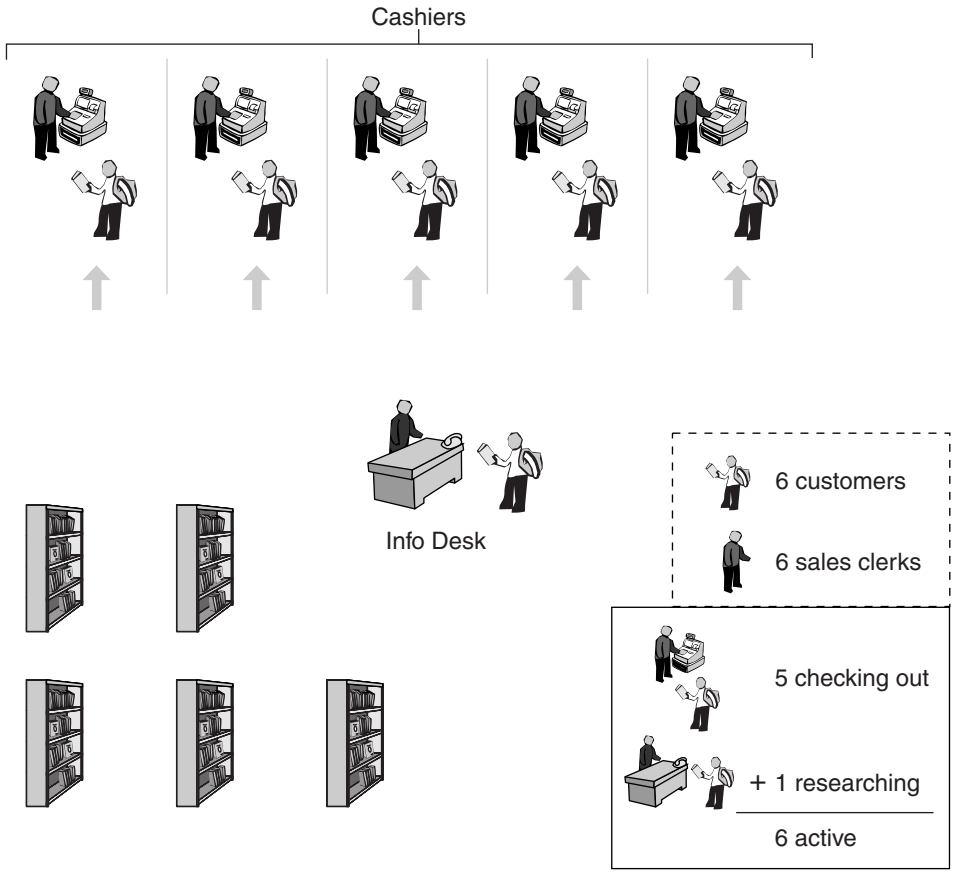

Figure 1.5 Active customers in a traditional bookstore

Peak load refers to the maximum concurrent customers in the store within some time period. For example, Figure 1.7 shows a graph of hourly customer visits during the course of a day. Notice that the store experiences two activity "spikes" during lunchtime and between 4 PM and 5 PM each day. The peak period, however, is the hour between 4 PM and 5 PM each day. The store receives its peak load (50 customers) during this period.

Peak load is not the average load. The store shown in Figure 1.7 averages 25 customers an hour during the day. If we only consider averages, the store might not schedule enough sales clerks to handle the 50 customers arriving between 4 PM and 5 PM. Note

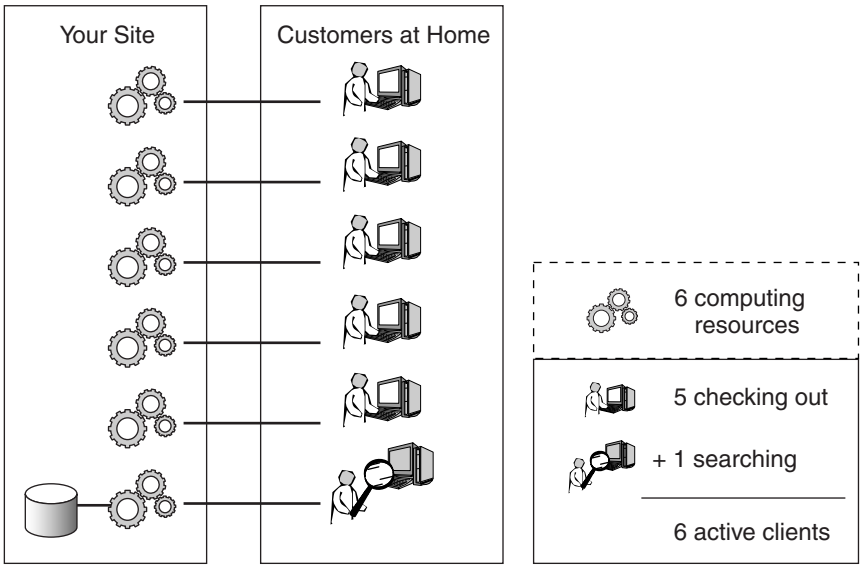

Figure 1.6 Active web site clients

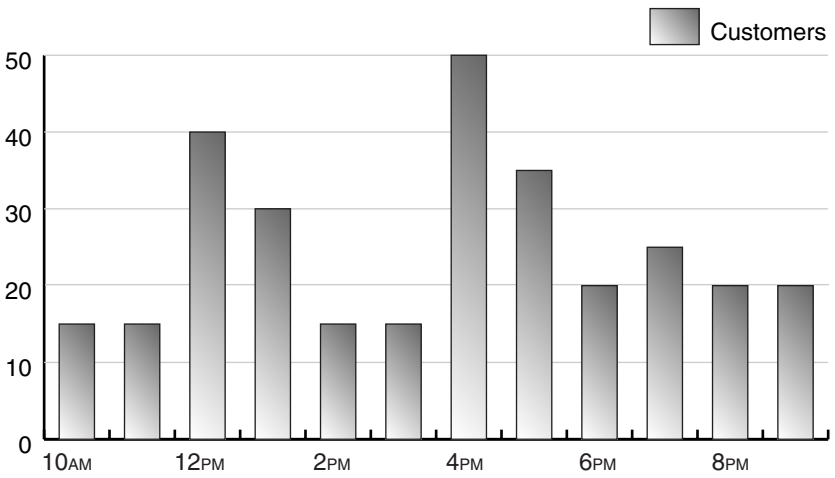

Figure 1.7 Traditional store customers per hour

that our concerns about averaging apply to the time scale as well. Perhaps Figure 1.7 represents an average day at the bookstore. While this information proves useful for scheduling most of the year, it doesn't tell us how many people arrive on our peak day (the day after Thanksgiving) or their arrival distribution during that day.

On-Line Store

Like a traditional bookstore, web sites experience an uneven distribution of users throughout the day. The peak user arrival times vary from web site to web site. For example, brokerage web sites receive intense load every day at market opening, while an on-line bookstore receives most of its traffic at lunchtime or after the workday ends. Also, web sites, like their brick and mortar counterparts, experience unusually high traffic during peak times such as the Christmas holidays or a stock market rush. However, web site traffic patterns differ significantly from those of traditional stores. An on-line bookstore's lunchtime spike often lasts three hours as lunchtime rolls across time zones from the East Coast to the West Coast.

Again, the peak load is our planning focus. A web site unable to support its peak traffic is an unsuccessful web site. When planning your performance test, find the peak loading goals for your web site, and build the test to exercise this load. (See Chapter 6 for details on developing an accurate performance test.)

Throughput: Customers Served over Time

Traditional Store

A clerk sells a customer a book at the cash register. This transaction requires one minute. Let's assume it *always* takes one minute to complete a customer sale, regardless of how many books the customer buys. In this case, if the store contains five cashiers, the store serves five customers per minute. (Figure 1.8 illustrates this scenario.) In performance terms, five customers per minute is the store's *throughput.* Throughput measures the customers served relative to some unit of time. It is a unique metric because it has an upper bound. No matter how many customers come to the store, the maximum number handled during a specific time interval remains unchanged. (We know from experience the clerk cannot check out two customers simultaneously.)

So with five clerks operating five cash registers, our store serves a maximum of five customers a minute. This represents our maximum throughput, regardless of how

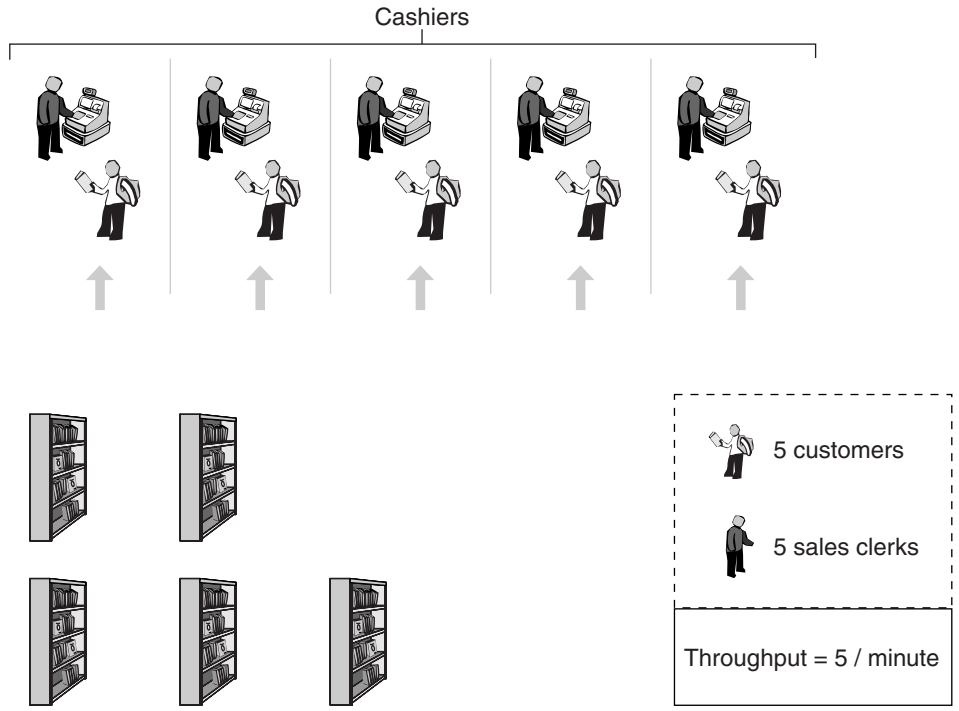

Figure 1.8 Throughput dynamics without waiting

many customers actually want to checkout during this time. For example, Figure 1.9 shows the same store, only now with ten customers; however, the clerks still sell at a rate of one customer per minute. So, with five clerks, we still only serve five customers per minute (our store's maximum throughput remains unchanged). Thus five of the customers must wait or queue. After reaching the throughput upper bound, adding more users to the store does not increase throughput. This is, as you might guess, an important concept to grasp.

On-Line Store

An on-line site exhibits the same throughput dynamics as the traditional store. An on-line store handles requests, typically initiated by a customer using a web browser. For example, a customer may search for particular books or purchase the contents of a shopping cart. A web site handles a specific number of requests in parallel; for example, a web site may handle 20 customer requests simultaneously. If each request takes one second to process, the on-line store throughput is 20 requests per second.

Just as in our traditional store, adding more requests does not increase throughput after we reach our throughput upper bound. If the web site receives 30 requests per

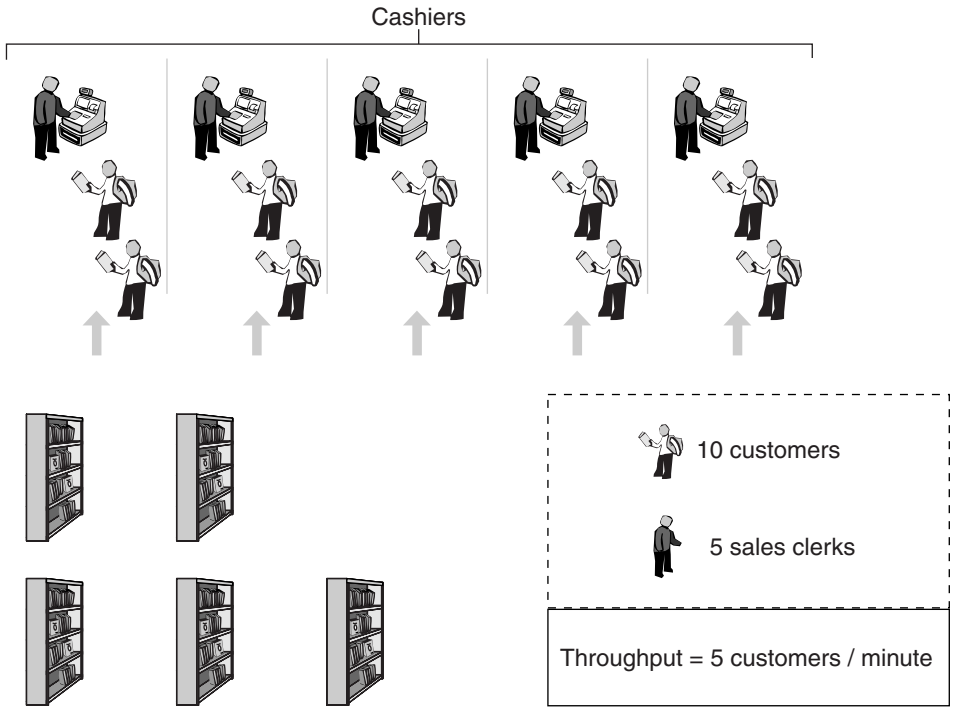

Figure 1.9 Throughput dynamics with waiting

second, but the maximum throughput is 20 requests per second, some of the requests must wait or queue.

Throughput Curve: Finding the Throughput Upper Bound

A performance test uses a series of test runs to understand the relationship between load and throughput. A graph of the data from these runs establishes the *throughput curve* for your system. As the load on your system increases, the throughput usually increases as well until it reaches the throughput upper bound (maximum throughput). On the graphs shown in this section, we plot the load on the *x*-axis and the throughput (requests per second or customers per second) on the *y*-axis.

Traditional Store

Figure 1.10 shows the throughput curve for the bookstore. With one customer, the throughput is one customer per minute; at two customers, it is two customers per minute. This pattern continues through five customers, with a throughput of five customers per minute. The graph shows that as the load increases between one and

five customers, the throughput increases to a maximum of five customers per minute. Once the load reaches five customers, all five clerks are busy at the cash registers. Throughput then remains a constant five customers per minute, even as the number of customers increases. After we reach maximum throughput, we have reached the throughput plateau. Beyond maximum throughput, adding load, or users, results in a consistent, flat throughput curve (a plateau). For the bookstore, the throughput plateau is five customers per minute.

On-Line Store

Web sites produce a similar throughput curve with a throughput plateau. Figure 1.11 shows a typical throughput curve for a web site. (Obviously, this graph contains more data points than the bookstore throughput graph.) The "Light load zone" in the figure shows that, as the number of user requests increases, the throughput increases almost linearly. At light loads, requests face very little congestion for resources. After some point, congestion starts to build up, and throughput increases at a much lower rate until it reaches a saturation point. This is the throughput upper-bound value.

The throughput maximum typically represents some bottleneck in the web site, usually a saturated resource (the on-line equivalent of all sales clerks being busy). The CPU often becomes the constraining resource on your web site. After your CPU(s) reach 100% utilization, the web site lacks processing capacity to handle additional requests.

As client load increases in the "Heavy load zone" in Figure 1.11, throughput remains relatively constant; however, the response time increases proportionally to the user

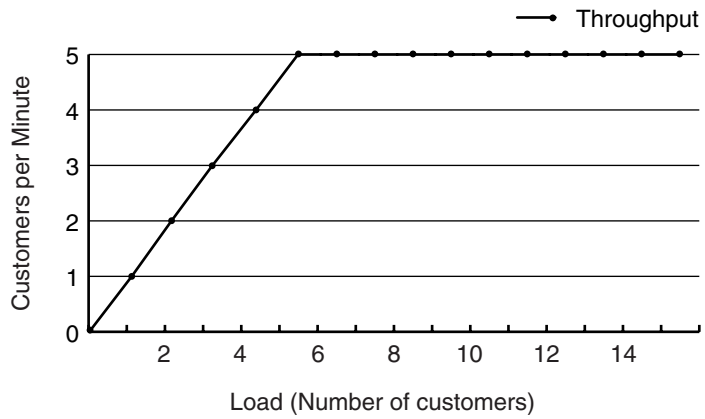

Figure 1.10 Throughput curve: brick and mortar store

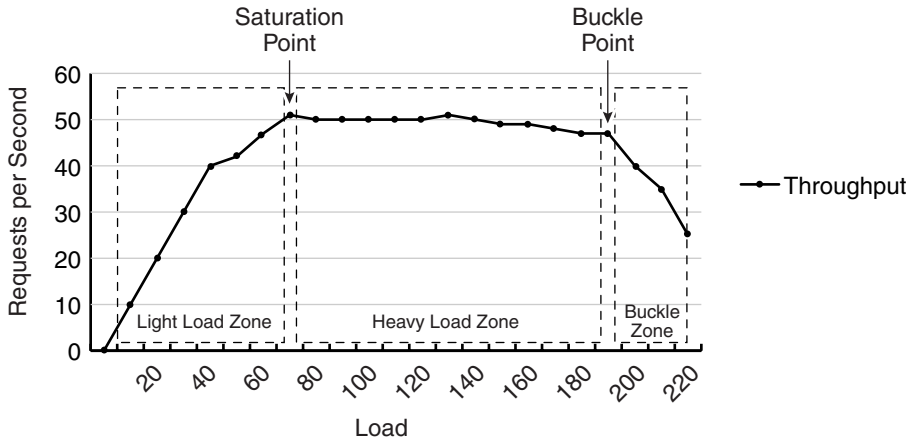

Figure 1.11 Typical web site throughput curve. From Gennaro Cuomo, "A Methodology for Production Performance Tuning," an IBM WebSphere Standard/Advanced White Paper. Copyright 2000 by IBM Corp. Reprinted by permission of IBM Corp.

load (see the next section on response time for more details). At some point, represented by the "buckle zone," one of the system components becomes exhausted, and throughput starts to degrade. For example, the system might enter the buckle zone if the network connections at your web server exhaust the limits of the network adapter, or if you exceed the operating system limits for file handles.

So far, we've discussed throughput generically as "requests per second." We often hear throughput discussed in terms of hits, transactions, pages, or users in some unit of time (usually a second, but sometimes in terms of a day or week). Not surprisingly, how you measure your throughput makes a big difference in how you set up your tests, and also affects your hardware plan. We briefly discuss the differences between hits, transactions, pages, and users in the next sections. A more thorough discussion, with an example, appears in Chapter 6.

Hit

A *hit* may mean one of several different things. For an HTTP server specialist, a hit means a request to the HTTP server. Because HTML (Hyper Text Markup Language) pages usually contain embedded elements, such as gifs or jpegs, one HTML page might require multiple HTTP "hits" as the browser retrieves all of the elements from the server to build a page.

Regrettably, the rest of the world uses the term *hit* in very ambiguous ways. Sometimes *hit* refers to an entire page, including embedded elements. Also, many companies routinely use *hit* to mean an entire site visit by a given user. (A site visit usually encompasses many pages, not to mention the embedded elements and frames

included in those pages.) Therefore, you must first make sure everyone discussing a "hit" is discussing the same thing. Any misunderstanding on this point drastically impacts the success of the performance test and the production web site.

Transaction Rate

Transaction rate is the most common measurement of throughput. (Web sites often measure their throughput in transactions per second.) Usually, web sites define a *transaction* as a single HTTP request and response pair. However, the definition for *transaction*, like that for *hit*, often means different things to different people. Sometimes transactions involve more complex behavior than dealing with a simple request/response pair. Within your team, establish your definition of a transaction, and use it consistently.

Page Rate

Page rate measures the pages returned by a web site during a specified period of time. For a web site, a web page is the fully formatted content of one page as displayed in a browser. As noted above, web pages almost always contain embedded elements such as gifs, jpegs, JavaScript, and, in the case of frames, other pages.

In order to complete a web page, the browser makes multiple HTTP requests to retrieve imbedded elements. For example, the first request returns the basic page content, and the browser issues additional requests for each embedded gif file or JavaScript element. Again, almost all web pages contain embedded elements. (See Chapter 5 for an expanded discussion of web page construction based on web site function.)

Since it is important to look at your web site as a whole, understanding throughput in terms of pages per second makes sense. Your web site may handle a high request volume in terms of HTTP requests during the day. However, if each of your web pages contains many embedded elements, this request volume may not translate into a high page volume. (It takes many requests, in this case, to build a single page.) From a user's perspective, the throughput of your web site could be much lower than your transaction volume.

User Rate

The *user rate* measures the users visiting the web site during a period of time. The web site receives many visitors over the course of the day. They interact with the web site to perform one or more tasks, which may involve navigating through several pages.

The most common definition of *user* refers to one of potentially many visits a user might make to the web site over the course of day. The user visit includes the set of

web pages navigated while using the web site. Because a user visit encompasses multiple pages, the user rate is usually lower than the page rate or transaction rate for a web site.

In practice, the definition of *user* varies from team to team. For example, web masters frequently interchange *user* with *hit* (a single request/response pair, not a multipage web site visit). See Chapter 6 for an expanded discussion. Again, you and your team need to pick a consistent definition and stick with it.

Response Time: Time to Serve the Customer

Traditional Store

In the traditional store, each customer sale takes a certain amount of time. For example, it may take a sales clerk one minute to check out a customer and complete a sale. Prior to actually purchasing a book, you may wait in line for an available clerk. This wait time adds to the length of time required to complete a purchase. Figure 1.12 shows the last customer in line waiting four minutes to reach the sales clerk. After the customer reaches the clerk, the sale requires the standard one minute of processing time to complete. Therefore, this customer's total checkout time includes the four minutes of waiting, plus the one minute of actually checking out, resulting in a total checkout time of five minutes.

On-Line Store

A customer who initiates a request from a browser waits a certain amount of time before the web page resulting from the request appears in the browser. For example, if the on-line customer issues a request to purchase a book, the browser submits the request to the web site, and the customer waits until the web site returns an order confirmation page to the browser. Note that this works much like our traditional bookstore: Our purchase takes some quantity of time, regardless of whether we wait in line or not prior to making the purchase.

Web site *response time* refers to the time from when the customer initiates a request from his browser until the resulting HTML page returns to the browser. (Technically, response time refers to the time between the request and the display of all the page's content. However, the user's perceived response time spans the request to the first appearance of returning page data in the browser.)[2]

Remember that the customer's request shares the web site with potentially many other simultaneous requests. If the request finds the processing capacity of your web

2. See Patrick Killelea, *Web Performance Tuning*.

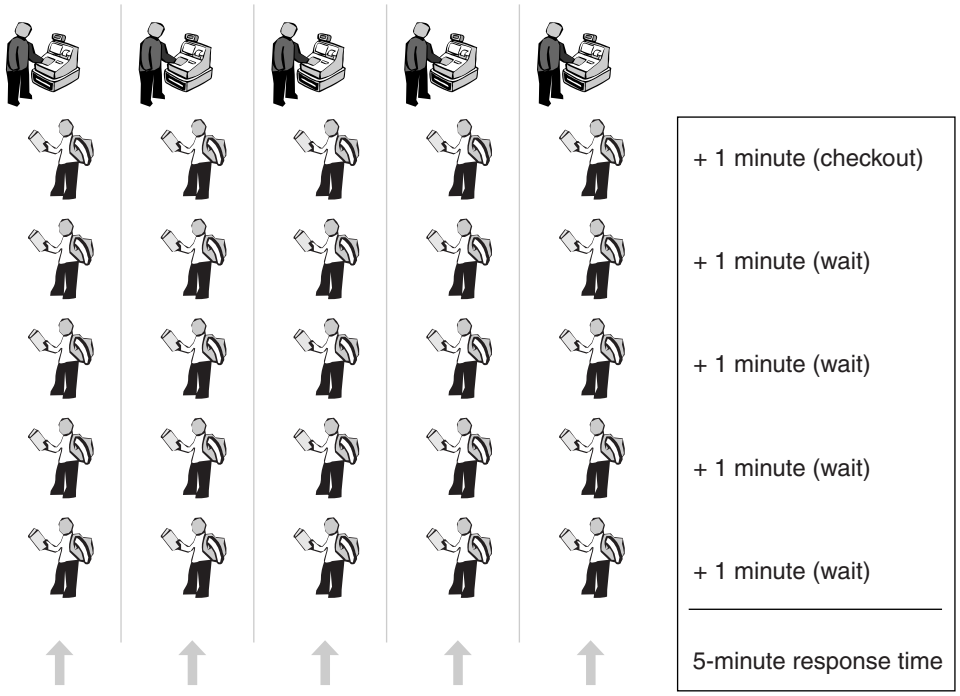

Figure 1.12 Total checkout time in a traditional store

site fully engaged, the request "waits." (Actually, the web site usually queues the request until processing capacity becomes available to satisfy it.) Just as in our traditional bookstore, the time spent waiting adds to the actual service time of the request. The total response time in this case is the wait time plus the actual service time.

Again, response time is the time it takes to serve the customer. In both the brick and mortar example and the on-line store, the response time consists of the total time it takes to purchase the book. For example, if a customer submits a request to purchase a book, we measure the response time from when the user submits the request via a browser until the browser displays the confirmation page. This time includes any "wait time" for busy resources. Figure 1.13 shows the response time for a typical web request.

Also, in either a traditional or on-line environment, customers come with limited patience. If response time grows too long, the customer stops waiting and leaves, maybe never to return to your store or web site. Therefore, response time is the critical measurement for most web sites. Performance testing strives to minimize response time and ensure it does not exceed your web site objectives, even during peak loading.

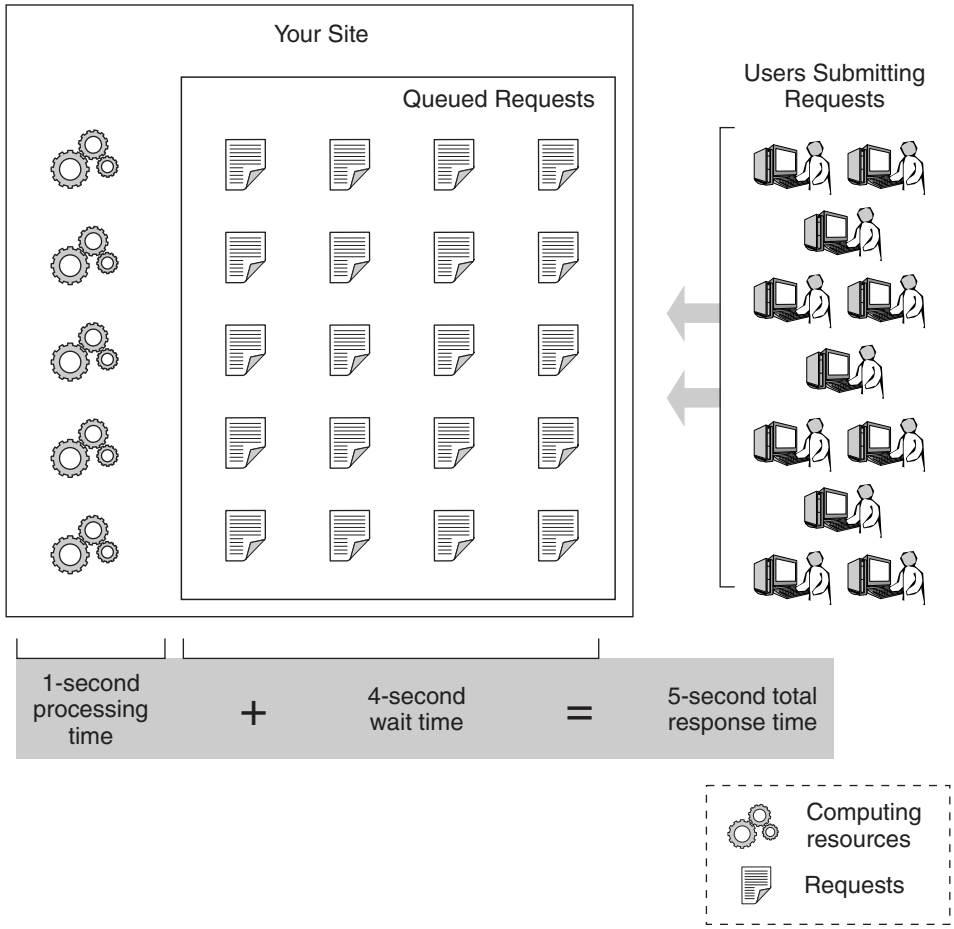

Figure 1.13 Response time in an on-line store

Think Time

Think time is the time a user takes between submitting requests. Think time and response time represent two different concepts. Response time measures the time the user waits for a response to return from the web site (including any wait time for server resources). Think time, however, measures the interval between a user's requests. During the think time, the server performs no work for the user because the user has not made a request.

For example, the user's first request to search for a book may require five seconds to process. After the server returns a list of books matching the search criteria, the user

reads through the list and decides what action to take next. Maybe the user chooses to submit another search, or maybe he requests more details about a particular book on the list.

During this think time, the user may read material previously obtained from the server, choose further server activity, or even go for a snack. Regardless, from your web site's perspective, the user is not active during this time. For example, Figure 1.14 shows a user making two separate requests, one minute apart. The response time for each request is only five seconds, but the total time the user spends shopping is one minute and ten seconds. Chapter 7 contains a more detailed discussion of think time and the important role it plays in performance testing and capacity planning.

While think time is beyond the scope of our control, we can (and must) carefully control and monitor our web site's response time. Make no mistake: Response time is the critical performance measurement for any web site because it represents how long users wait for a given request. Like their traditional store counterparts, on-line customers often refuse to wait patiently for an overworked server. If your site cannot deliver pages quickly, even under heavy load, you lose customers. Discouraged customers often never return to an underperforming web site. The primary objective of

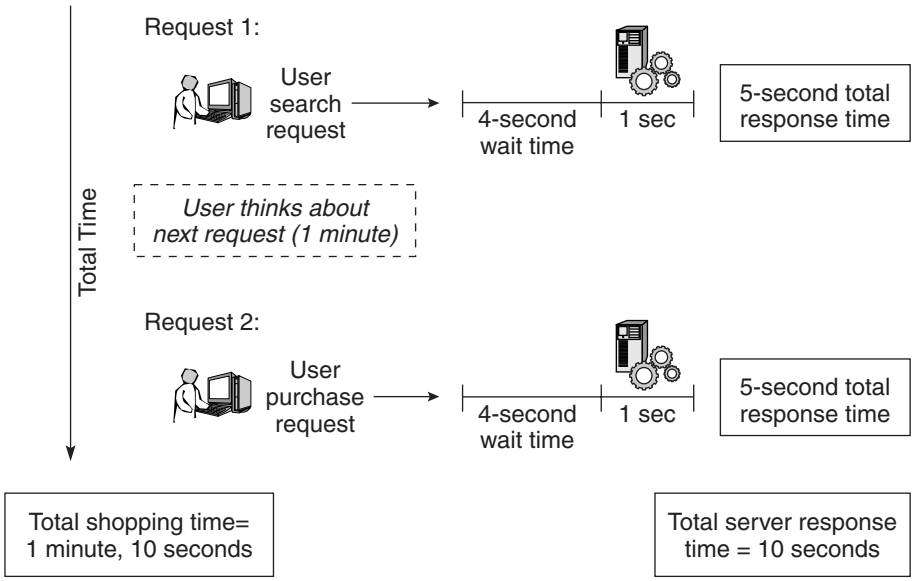

Figure 1.14 Think time example

your performance test should be to minimize and measure your web site's response time. Even at peak load, keep your web site's response time under its responsiveness objectives.

Response Time Graph

Understanding your web site's response time requires an understanding of the relationship between load, throughput, and response time. As discussed earlier, your web site does not possess infinite throughput potential, but achieves its maximum throughput at a certain user load. Beyond this point (known as the throughput saturation point), throughput remains constant. However, the response time begins to increase. Response time increases after throughput saturation because of resource constraints. Additional load waits for these limited resources before actually doing useful work.

Traditional Store

Let's look at the load, throughput, and response time dynamics in a brick and mortar store. If only one customer wants to check out, she receives immediate service from the cashier. If we require one minute to complete her sale, the store's throughput becomes one customer per minute. Let's extend our example by assuming that the store contains five cashiers. If five customers arrive at the same time for checkout, they all receive simultaneous service from the individual cashiers (see Figure 1.15). Since these five customers did not wait, they experienced one-minute response time (the time required to check out). Likewise, since our store served multiple customers in parallel, our throughput increased to five customers per minute.

This throughput (five customers per minute) represents our store's throughput upper bound. A cashier cannot check out multiple customers at once, so our store experiences a resource constraint (cashiers) that prevents it from exceeding this boundary. If a sixth person arrives while all five cashiers are busy with other customers, the sixth person waits, or queues, until a resource (in this case, a cashier) becomes available. Time spent in the queue waiting for a resource is called *wait time*. In our store, the maximum wait time for the sixth person in line is one minute (the time required to check out the person currently being served by the cashier). Figure 1.15 shows an example of customers experiencing wait time.

Wait time also influences the response time for the sixth customer. We must add the wait time to his overall service time. In the case of our sixth person, the response time increases to two minutes: one minute waiting for a cash register to clear plus one minute actually spent checking out (yes, 1 + 1 = 2). However, because of limited resources (again, the cashiers), the store's throughput does not increase as more customers arrive. This demonstrates our previous claim about throughput: Once you hit

the throughput upper bound, response time begins to increase as additional load enters the system. As Figure 1.15 shows, the response time doubles as the number of customers doubles past the throughput saturation point.

On-Line Store

The on-line store exhibits the same dynamics in the relationship between throughput, response time, and load. If our on-line store handles a maximum of 5 requests per second with a one-second response time, the next 10 simultaneous requests experience a maximum response time of two seconds, while 20 simultaneous requests experience a maximum response time of four seconds, and so forth. Figure 1.16 shows the linear relationship between response time and load after reaching the

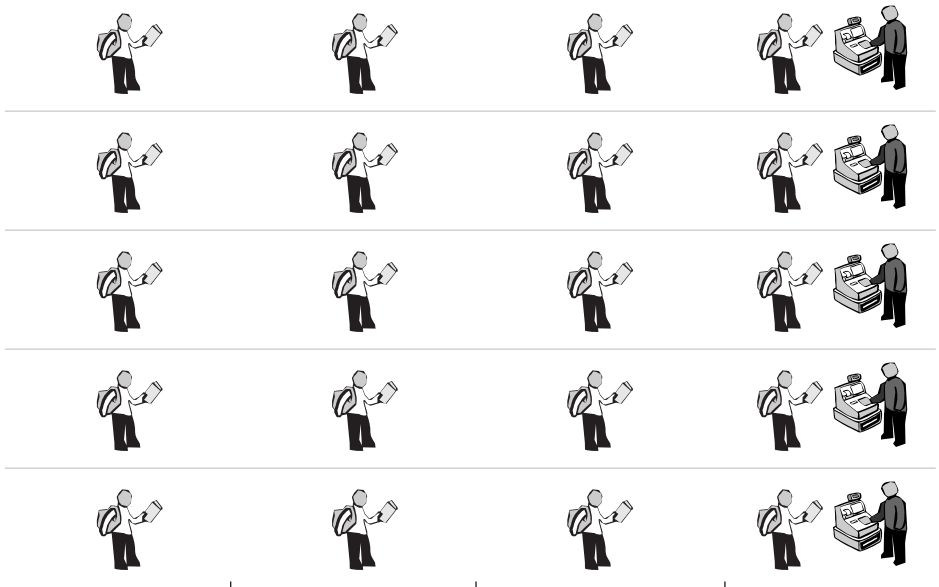

Figure 1.15 Response time queue for a brick and mortar store

throughput upper bound. (This graph does not show the throughput buckle zone, as displayed in Figure 1.11, but it still exists. Once the load goes beyond a certain point, average response time growth is more than linear.)

Understanding how many customers your web site may handle simultaneously and how long your waiting lines and response times become at your busiest times is the primary motivation behind performance load testing. This understanding, in our opinion, forms the basis of "due diligence" for any commercial web site. Clearly, we need more than a throughput curve like the one shown in Figure 1.11. We also need a response time graph to show the relationship between load, response times, and the throughput upper bound, like that shown in Figure 1.16.

This graph resembles the throughput graph in Figure 1.11 in several ways. The left *y*-axis plots our throughput, while the *x*-axis plots our load. However, we add another *y*-axis on the far right to plot our response times. As you see, with one to five customers, the response time averages one second. However, the web site reaches throughput saturation at this point. Beyond this point, the graph shows unchanging throughput despite increased load (a sure sign of throughput saturation). The response times for the increased load, however, grow linearly beyond the response time at the throughput saturation point. (Again, watch out for the buckle zone, where response times often grow exponentially.)

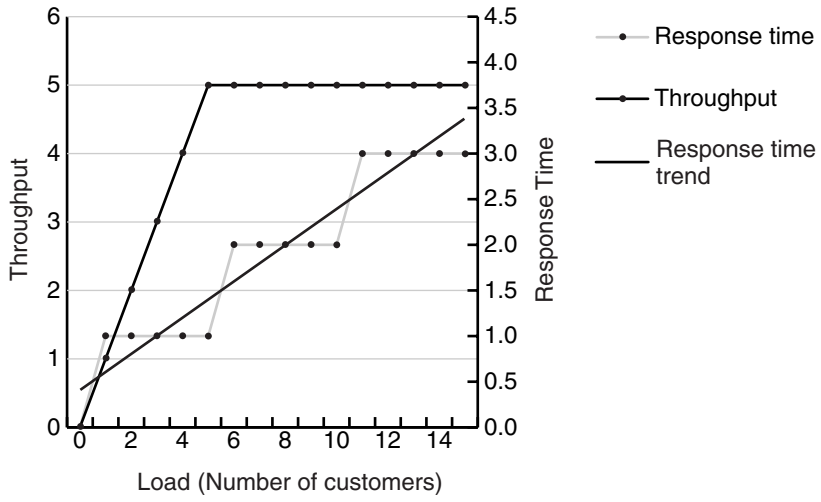

Figure 1.16 Throughput curve and response time graph

Throughout the rest of the book, we discuss what to do if your throughput and response time numbers do not meet your expectations. In fact, few web sites start with ideal throughput and response times. Usually, the first performance tests show throughput much lower and response times much higher than you actually require. The process of performance tuning focuses on improving these numbers and giving your web site the performance your customers demand. Even if your results fall well below expectations by a factor of 10, or even 100, don't despair. Keep reading. Sometimes very small performance adjustments make a tremendous difference in overall performance.

Steady State

Traditional Bookstore

Most bookstores open for limited periods during each day, for example, 10 AM to 9 PM. Although the store opens at 10 AM, the manager might panic if a large number of people arrived at 10:01 AM to buy books. Why? Because the store needs time to fully prepare. Before the bookstore works at optimum efficiency, the staff must complete some opening activities. The bookshelves need dusting and straightening. Returned books need reshelving. The cash tills in the registers might be low on change, so the manager makes a quick trip to the bank for coins. Maybe a few employees get stuck in traffic and arrive a few minutes late. In any case, the store works most efficiently after the store staff arrives, and finishes these start-up activities. Also, the traffic levels at off-peak times tend to be low. The store rarely experiences peak traffic at 10 AM or 9 PM simply because customers do not come in large numbers during those hours.

When we measure the response time for our store, we want to measure at a time when the store operates efficiently and when it experiences significant (even peak) loading. Otherwise, our data may not be valid. For example, if we measure response time at 10:01 AM, just after opening, we may get mixed results. A customer might experience terrific response time because she's the only customer in the store. However, on some days, a customer might wait an unusually long time if the store is out of change and the manager must go to the bank for coins. If we take measurements at 10:01 AM, our measurements are outside the store's *steady state*. The store needs more load, or more preparation, to demonstrate its typical behavior.

On-Line Store

Surprisingly, our performance tests must also consider steady-state issues to obtain valid data. The web site under test also experiences "opening the store" activities, albeit quite different than dusting shelves. "Opening activities" for web sites include loading servlets into memory, compiling JavaServer Pages (JSPs), priming caches, and other activities with a one-time cost. Since the vast majority of your web site

users never experience the cost associated with these activities, try factoring them out of your tests by selecting your measurements carefully.

Just as the web site requires preparation time, your test usually requires time to reach full loading. Most tests start with a few virtual users and increase the load over time until achieving the maximum load for the test. We call this the ramp-up period; the load ramps up to maximum users. Likewise, all virtual users in your load may not stop at the same time, but finish the test scenario over some time period. We call this the ramp-down period as you wait for the test to complete.

We want to capture data only after all of the users start. We also want to delay the measurement slightly after starting all of the users to give the system time to adjust to the load. Figure 1.17 demonstrates this concept. The ramp-up occurs as the user load increases up to 100 users. During this time, our web site goes through its "opening activities," such as bringing servlets into memory and priming caches. Again, we do not include this time in our measurements because the data tends to be ambiguous. Likewise, we do not want to take measurements while the tests finish during the ramp-down period. Instead, note the time period during full loading when we actually take measurements. This gives us an accurate picture of how the web site operates under this load. In this case, we take measurements from our full load period, giving ourselves two time periods of buffer after ramp-up and before ramp-down.

Remember, you want measurements best representing your site under load. Failing to account for the ramp-up and ramp-down periods of your performance test may invalidate the entire test.

Optimization Terminology

Now that we've covered some of the basic performance measurement terms, let's discuss some of the terminology associated with optimizing your web site. After all, we want not only to know about our web site's performance, but also to improve it. So how do you improve your web site's performance? For example, recall our earlier discussion of performance issues such as the throughput upper bound. What do you do after you find the upper bound, and it is lower than you hoped? At this point, you need to optimize your web site or apply scaling techniques to push the site beyond its current performance.

Optimization usually focuses on the web application. You may reduce the number of steps the application performs (known as "shortening the application's path length") to save time. You also may eliminate key resource bottlenecks by pooling resources, or making other adjustments to improve resource availability. Optimizing a web site requires a knowledge of how it behaves for any given user, as well as how it behaves

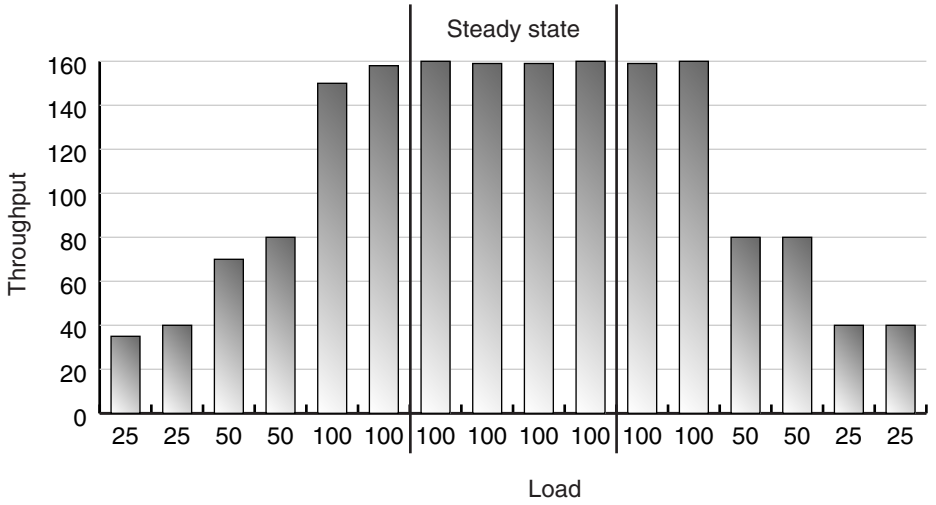

Figure 1.17 Steady-state measurement interval

under load from many users. For example, we might discover our web site's "checkout" function requires 20 seconds to complete for a single user. We must optimize this function to improve the response time. Likewise, under a light load of ten users, we find our search function response time increases from 3 seconds to 50 seconds. In this case, we need to remove a resource contention only present under load.

Scaling techniques, on the other hand, focus on adding resources to increase the web site's capacity. For example, we might double our throughput capacity by doubling the number of available servers. However, not all web sites scale well, so we must test the web site before assuming additional hardware provides any capacity benefit. This section covers the basic terminology of optimization. Later chapters give more detail on how to apply one or all of these techniques to your web site.

Path Length: The Steps to Service a Request

Traditional Store

In the previous section, we measured key activities in our brick and mortar store. In this section, we discuss some optimizations we might make now that we better understand the operation of the store. Let's look first at the checkout process, which takes one minute per customer to complete. So what happens during the checkout process? During this one minute, the clerk performs multiple actions, such as greeting the customer, entering her name and address, entering the price of the book into

the cash register, taking money from the customer, providing change, and putting the book into a bag. Figure 1.18 outlines this list of activities, or *path*. Keep in mind that if we reduce checkout time from 1 minute to 30 seconds, we not only reduce the store's response time but also increase its overall throughput. For example, assuming five cashiers, throughput increases from five to ten customers per minute. (One customer every 30 seconds results in two customers per minute. Multiply by five cashiers, and you get an overall throughput of ten customers per minute.)

We decide to reduce the checkout time by reducing the path length of the checkout activity. The *path length* is the number of steps required to complete an activity, as well as the time each step takes. We reduce the path length either by (1) speeding up a step or (2) removing a step entirely. For example, as shown in Figure 1.18, the clerk types in the customer's name and address as part of the checkout process. The store never uses this information, so we remove this step, which reduces the number of steps required and removes ten seconds from every checkout. While entering the customer's name proved optional, entering the price of the customer's purchases remains mandatory. Entering the price takes 30 seconds, a significant part of our processing time, because each clerk manually enters the price from a tag on the item. Purchasing a bar-code scanner and automating this step speeds up the checkout process by 20 seconds. These adjustments cut the checkout time in half and doubled throughput. However, these long-term performance gains required an investment in a detailed understanding of the checkout process.

On-Line Store

Much like the cashier, your web application code executes "steps" to complete each request. These steps form the path through your code for each request. Likewise, making the path through your code shorter or more efficient improves both response time and throughput. Of course, before improving web application code, you must first understand it. Code analysis requires both programming skill and time, particularly if you did not originally develop the software under analysis. While it does require an investment, *code path optimization* is usually the most effective technique for improving web site performance. Just as with the brick and mortar store, reducing a web application's code path involves two tactics: (1) Removing unnecessary code and (2) improving the performance of the remaining code. Again, the *path* is the execution path your code takes. A code path might include the following steps: initialization, reading data, comparing values, and writing updates.

In order to optimize the code, we want to remove any unnecessary code from the path a given request takes. For example, the code path for our checkout function may contain a loop. At each pass through the loop, several statements execute. (If we loop ten times, we execute the statements ten times each.) Sometimes we find a statement, such as a constant declaration or the like, that does not need to be repeated inside the

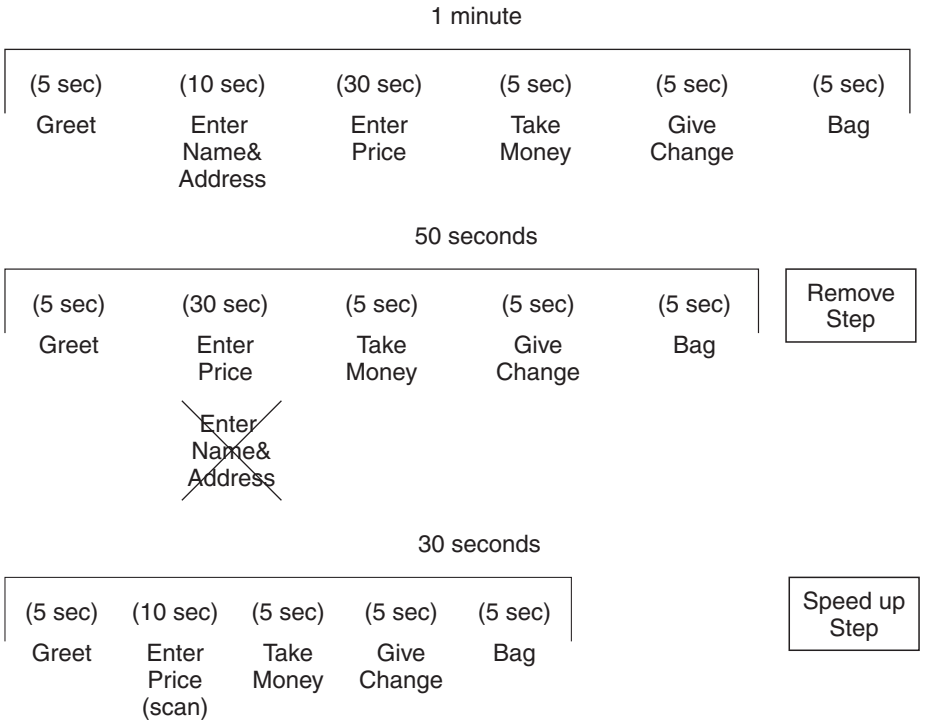

Figure 1.18 Reducing the checkout process path length

loop. Moving this statement outside the loop reduces the steps we execute on our code path (in ten iterations, we reduce the path length by nine statements.)

However, just as with our brick and mortar store, we cannot always change the number of steps we execute. In these cases, we consider making the steps themselves faster. For example, we might pool database connections to make obtaining a connection in our code faster. If we frequently manipulate `String` objects, we might build our own lightweight parser rather than using `StringTokenizer` throughout our code. (See Chapter 4 for specific Java tuning advice.)

Bottleneck: Resource Contention under Load

Bottleneck refers to a resource contention inside your web site. As the name implies, a bottleneck chokes the flow of traffic through your site as requests contend for one or more limited resources.

Traditional Store

In order to uncover bottlenecks, we need to look at our store as a whole rather than focusing on individual areas. Our store's response time and throughput rely on more than just the number of customers going through the checkout line.

For example, during Christmas, the manager receives many complaints from customers about long lines. However, the cashiers say they often wait for customers and never see more than one or two people waiting to check out, even during the busy part of the day. The manager decides to explore other areas of the store, such as the information desk, for long lines. As Figure 1.19 shows, he finds the line in the gift-wrap department. While customers make it through checkout quickly, they experience long delays if they want their purchases wrapped. The gift-wrap desk acts as a bottleneck on the throughput and response time of the entire store. Sufficient resources in one part of your store do not guarantee acceptable overall throughput and response time. The store operates as a system: A bottleneck in one resource often impacts the throughput and response time of the whole.

Removing bottlenecks is both an iterative and ongoing process. It is iterative because we may find that removing one bottleneck introduces a new one. For example, if the information desk became a bottleneck, improving that situation may put more pressure on the cashiers (people move more quickly to the cash registers rather than stalling at the information desk). After removing a bottleneck, be sure to reevaluate the system for new sources of contention. The process is ongoing because usage patterns change. Our gift-wrap department probably doesn't get a lot of traffic in August, but at Christmas it becomes a bottleneck because our customers' use of the store changes. Instead of purchasing books for themselves, they buy gifts for others. Constant monitoring allows you to identify usage shifts, and reallocate (or add) resources as needed to meet the new traffic patterns.

On-Line Store

On-line stores frequently experience bottlenecks. We discussed earlier tuning the code paths throughout your web application. However, code path tuning focuses on single-user performance; bottlenecks emerge only when we put the system under load. While code path tuning gives us more efficient code, we also must study the system under load to look for these resource sharing issues.

Technically, we find a bottleneck wherever the code processing a request must wait for a resource. For example, when your web site receives a request to purchase a book, the code path to handle this request probably includes a database update. If the database update locks rows needed by a request from a second user who wants to purchase a different book, the processing of the second user's request waits until the

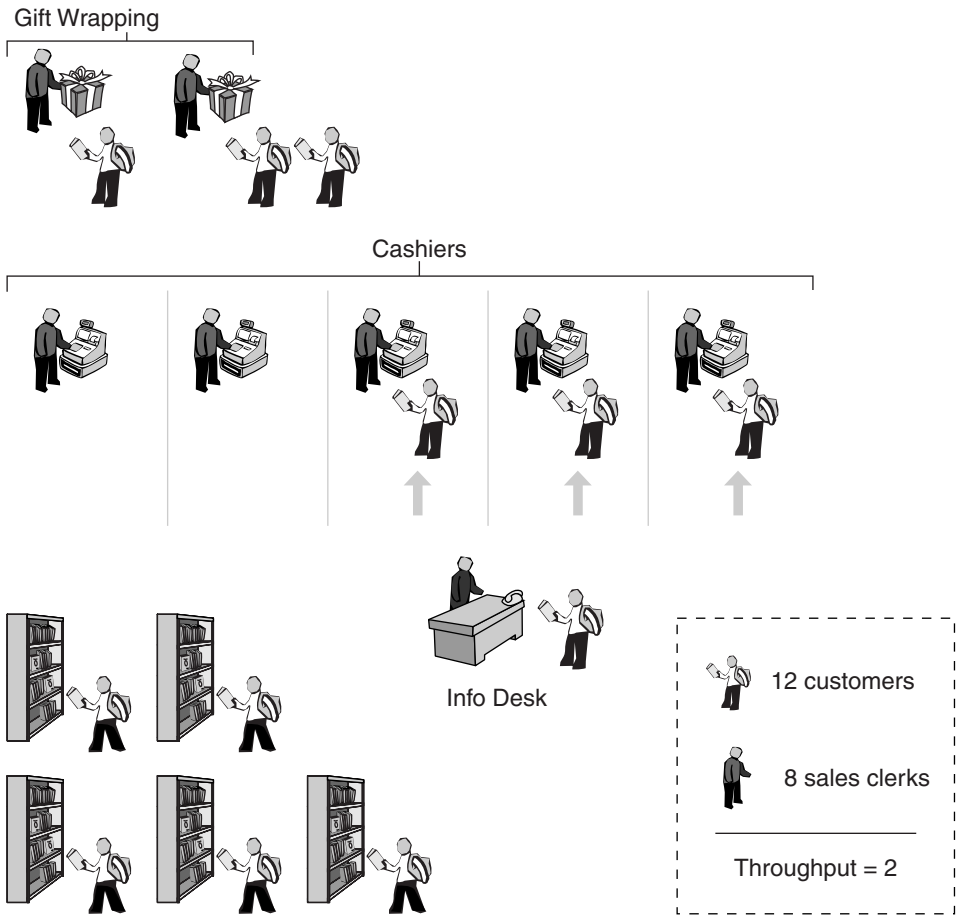

Figure 1.19 Example bottleneck: gift-wrap department

desired rows become available. This locking design works great in a single-user test, giving us problems only when we add load (additional users). This example again underscores the importance of concurrency and load testing prior to releasing your web site.

While poor database sharing strategies often develop into serious bottlenecks, you may also uncover bottlenecks associated with sharing hardware resources. Each request requires CPU and memory resources from your web site. If you handle many simultaneous requests, these hardware resources may quickly deplete, forcing request processing to wait. CPU and memory often become bottlenecks under load. For

more information on identifying and resolving bottlenecks, see Chapter 13. Bottlenecks often produce unusual symptoms, so check this chapter for more details on how to spot resource sharing problems.

Scaling: Adding Resources to Improve Performance

Scaling a web site simply means adding hardware to improve its performance. Of course, this only works if your hardware resources, such as CPU and memory, present the largest bottleneck in the web site's system. If your code spends most of its time waiting for CPU resource or memory, adding more hardware improves performance. Regrettably, web site teams often upgrade or add hardware only to find little or no performance benefits after they finish the upgrade. We recommend tuning the web site and performing at least fundamental bottleneck analysis before adding hardware. Dedicate some time to making your web applications more efficient. Most important, find and eliminate any major bottlenecks, such as database locking issues. If you perform these optimization steps well, your hardware resources eventually become your only remaining bottlenecks. At this point, scaling provides real performance benefits.

In a perfect world, scaling does not affect your application. You simply add new machines, or increase memory, or upgrade your CPUs. However, adding resources often changes the dynamics of your application. For example, if your application used to run on a single server, adding a new server may introduce remote communications between the application and a naming service or a database. Because of these unknowns, we recommend performing a scalability test prior to scaling your web site. This test tells you how well the web site performs after you add new hardware and whether the new configuration meets your performance expectations. Also, before purchasing new equipment, decide which form of scaling works best for your web application. Most scaling focuses on two basic approaches: vertical scaling and horizontal scaling.

Vertical Scaling

Traditional Store

In scaling, we increase throughput by handling more customers in parallel. Inside a bookstore, this means additional cash registers and cashiers. Figure 1.20 shows five additional cash registers, giving the store a total of ten registers. This raises the throughput of the store to a maximum of 20 customers per minute (two customers per minute per cashier multiplied by ten cashiers, assuming—here and in the rest of

this chapter's examples—our improved response time of 30 seconds per customer). We call adding resources to our existing store *vertical scaling*. We relieve resource contention within our physical store by increasing the constrained resource (in this case, cashiers and cash registers).

On-Line Store

On-line stores also use vertical scaling to process more requests in parallel. Vertical scaling at a web site helps us get the most out of each available server machine; it occurs at both the hardware and software levels. From a hardware perspective, we might increase the processors (CPUs) for each server. For example, we might upgrade a two CPU machine (also known as a *two-way*) to a four CPU machine (a *four-way*). In theory, this gives the machine twice the processing capabilities. In reality, we rarely

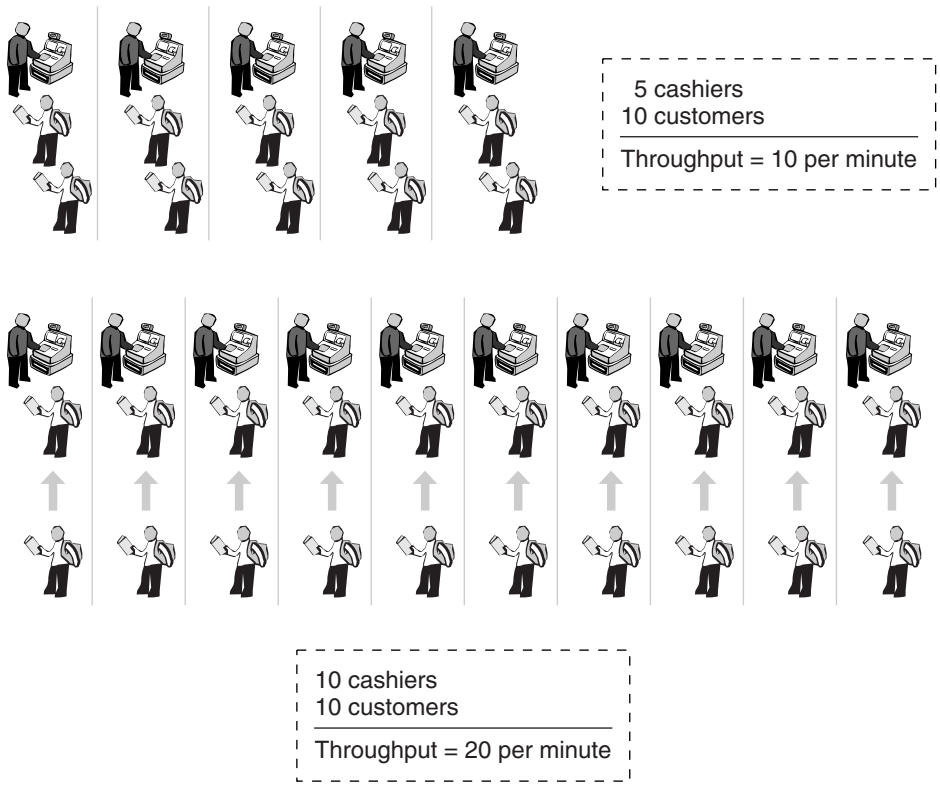

Figure 1.20 Traditional store handling more customers in parallel. Each cashier handles two customers per minute.

obtain a true performance doubling with this technique. Our ability to use extra CPUs inside an existing server depends on several factors:

- The existing servers must be upgradeable (they permit us to add more CPUs).
- The operating system and the Java Virtual Machine (JVM) support symmetric multiprocessing (SMP). Unless your OS and JVM support SMP, they cannot distribute work to multiple processors, thus making it difficult to take advantage of additional CPUs. (Happily, most newer operating systems and JVMs support SMP to a point.)

SMP is an important factor in vertical scaling. (In fact, you may hear the term *SMP scaling* as another name for vertical scaling.) As we mentioned, most modern operating systems and JVMs support SMP scaling, but only to a point. On extremely large servers (12 processors or more), a single JVM may not fully exercise all of the processors available. With such servers, we turn to *vertical software scaling*. Rather than one JVM, we execute two or more until all CPUs on the machine become fully engaged. Running multiple JVM processes allows us to better utilize all the resources of these large servers. (Chapters 2 and 11 discuss vertical scaling in more detail.)

Horizontal Scaling

Traditional Store

At some point, the bookstore may physically exhaust the available floor space and be unable to add more cashiers. When this happens, we cannot continue to increase the capacity of the store, even though the store does not meet the demands of our peak customer loads.

In the brick and mortar world, one of the following three alternatives typically transpires in this situation:

1. The store relocates to a larger facility.
2. A competitor opens a store nearby and takes customers from your store.
3. Your company opens a second store some distance away and customers shop at the most convenient location.

In any case, we expand our physical resources to meet customer demand. Either we provide more facilities, or our competition does. By opening the second store, we anticipate increasing our throughput, or at least reducing our response time, by distributing the customers between multiple stores.

For example, Figure 1.21 shows our throughput if a second store opens with ten checkout lines and a throughput of 20 customers per minute. If the first store continues to handle ten customers per minute, the overall throughput increases to 30 customers per minute. We call this *horizontal scaling*. We cannot continue to grow inside a single store (vertical scaling), so we add more stores.

On-Line Store

When it comes to horizontal scaling, on-line stores beat traditional stores hands down. Web sites grow by adding servers, not by building new bookstores. It's much easier and cheaper for a web site to grow to meet explosive customer demand (and fend off the competition) than for a traditional store.

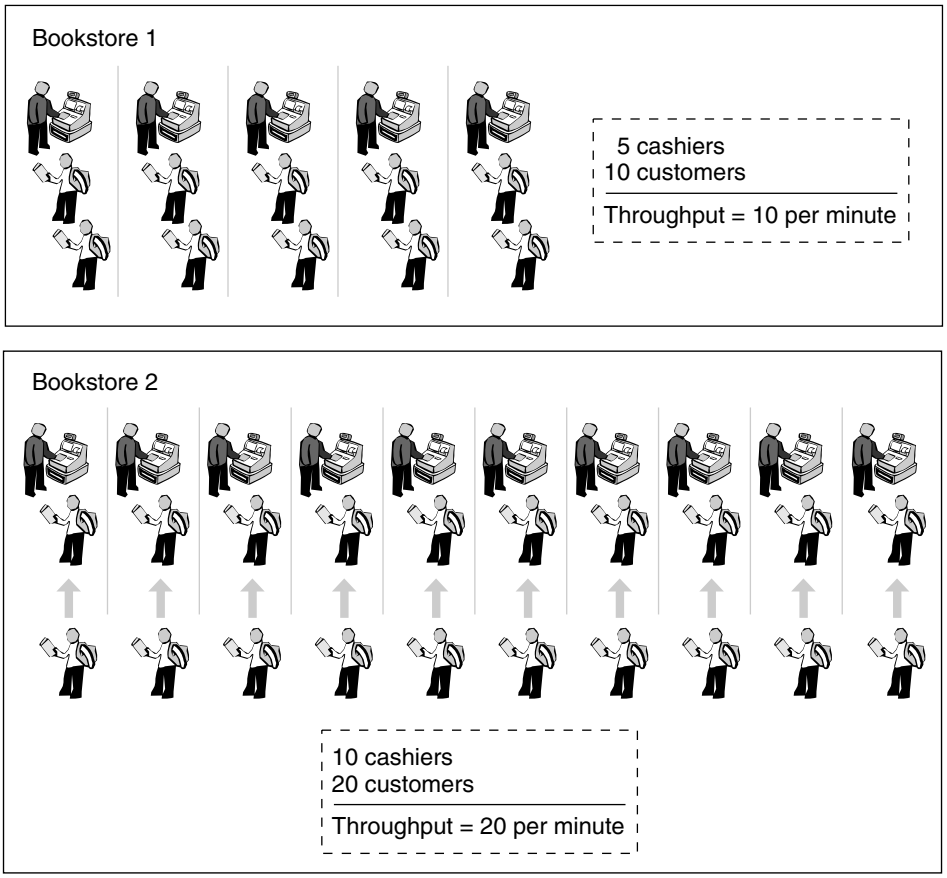

Figure 1.21 Scaling with multiple bookstores. Each cashier handles two customers per minute.

Horizontal scaling in the web world means adding duplicate resources (servers, networks, and the like) to handle increasing load. If you employ the clever technique of load balancing, your users never know how many servers make up your web site. Your URL points to the load balancer, which picks a server from your web site's server pool to respond to the user's request. Load balancing algorithms vary from the simplistic (round-robin) to the highly sophisticated (monitoring server activity, and picking the least busy server to handle the next request). Because the load balancer interacts with a pool of servers, known as a *cluster*, we sometimes call horizontal scaling *cluster scaling*. Figure 1.22 show a small cluster of two servers. The load balancer sends incoming requests to Server A or Server B for resolution. This cluster also allows the web site to support double the throughput of either server acting alone. (See Chapter 3 for more details on horizontal scaling.)

Linear Scaling

In an ideal world, we experience *linear scaling* when we increase our web site's capacity. Linear scaling means that as our resources double, so does our throughput. In the real world, perfect linear scaling seldom occurs. However, horizontal scaling techniques come closest to ideal linear scaling. Vertical scaling, while often improving overall throughput, rarely achieves true linear scaling.

Traditional Store

In Figure 1.20, doubling the number of cash registers and cashiers doubled the throughput in our store. This is an example of linear scaling. In reality, linear scaling often depends on complex interactions among various resources. For example, in order to increase our throughput, we need more customers in the store. This

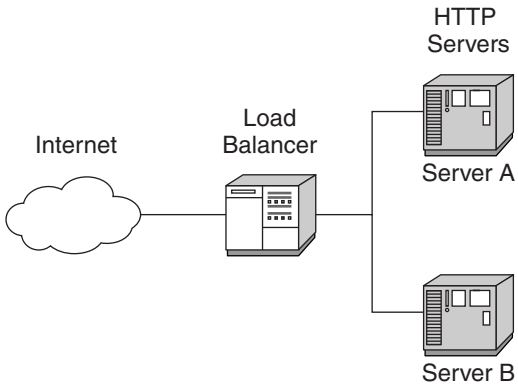

Figure 1.22 Horizontal scaling example

requires also increasing a range of other resources from the information desk personnel to parking spaces. If we fail to grow any of these resources, we probably won't experience true linear throughput growth.

On-Line Store

Scaling a web site also requires taking into consideration all required resources. You may want to double your throughput by upgrading your two-processor server to a four-processor machine. However, as you double your processors, you must also appropriately scale other resources such as memory and disk space.

Frequently, adding capacity also creates more complex concurrency issues within the server. More load often exposes new resource bottlenecks within your application. Again, consider testing your web site with its new capacity to determine the actual benefit of hardware upgrades.

Considerations for scaling your web site and setting appropriate expectations with respect to linear scaling are discussed in more detail in Chapter 11.

Summary

The same performance issues you encounter every day at brick and mortar stores also prevent your web site from achieving its full performance potential. In this chapter, we defined some basic performance terminology by showing how e-Commerce concepts also apply to the familiar world of traditional retail stores. By now you should understand the fundamental measurement concepts (load, throughput, and response time) of performance. You should also understand the relationships between these measurements, and how they impact each other. Also by now, you should be familiar with some basic optimization terminology (path length, bottlenecks, and scaling). The upcoming chapters expand and build on these basic performance concepts, so it is important to be comfortable with them before moving ahead.

While a traditional store and an on-line store share similar performance issues, resolving these issues on your web site requires a great deal more planning, testing, and analysis. The upcoming chapters cover in detail the knowledge and skills you need for measuring and tuning Java web sites.