10
Performance Considerations

Users have high expectations when it comes to the performance of the applications they use. Performance needs are requirements that must be met and their importance should be reflected in the fact that the entire team must take ownership of performance. Performance is a quality attribute and should be considered throughout the development of an application.

After taking a look at the importance of performance and some of the common terminology related to it, this chapter will describe a systematic approach to improving performance. We will also cover server-side caching, web application performance, and techniques to improve database performance.

In this chapter, we will cover the following topics:

- The importance of performance
- Performance terminology
- Taking a systematic approach to performance improvement
- Server-side caching, including different caching strategies and usage patterns
- Improving web application performance, including HTTP caching, compression, minifying resources, bundling resources, using HTTP/2, using content delivery networks, and optimizing web fonts
- Database performance, including designing an efficient database schema, using database indexes, scaling up/out, and concurrency

The importance of performance

The performance of a software application indicates the responsiveness of the operations that it can perform. Users have greater expectations today in terms of the responsiveness of the applications that they use. They demand fast response times regardless of their location or the device that they are using.

Most importantly, software must serve its functional purpose, be reliable in its operation, and provides its functionality in a usable way. If it does not, then the speed of the application will not matter. However, once those needs are met, performance is of high importance.

Performance affects user experience

The speed of your application plays a major role in the overall **user experience** (**UX**). A user's satisfaction with the application is influenced by the speed of the application. The performance of your application affects the organization's bottom line, whether it is because customers are being gained/lost for a customer-facing site or because productivity is being gained/lost for an enterprise application.

For web and mobile applications, the loading time for a page is a major factor in page abandonment. If a page takes too long to load, many users will simply leave. This is evident when we look at things such as a site's bounce and conversion rates.

Bounce rate

A *bounce* occurs when a user has just a single-page session on a site and leaves without visiting any of the other pages. The **bounce rate**, which is sometimes referred to as the exit rate, is the percentage of users who bounce:

$$Bounce\ rate = \frac{Total\ number\ of\ bounces}{Total\ entries\ to\ a\ page}$$

As you would expect, as page load times increase, so too does the bounce rate. Examples of actions that result in a bounce include the user closing the browser window/tab, clicking on a link to visit a different site, clicking the back button to leave the site, navigating to a different site by typing in a new URL or using a voice command, or having a session timeout occur.

Conversion rate

The **conversion rate** is the percentage of site visitors who ultimately take the desired conversion action. The desired conversion action depends on the purpose of the site, but a few examples of common ones include placing an order, registering for membership, downloading a software product, or subscribing to a newsletter.

The conversion rate is represented by the following formula:

$$Conversion\ rate = \frac{Number\ of\ Goal\ Achievements}{Visitors}$$

Websites that have poor performance will have a lower conversion rate. If a site is slow, users will simply leave the site and go somewhere else.

Performance is a requirement

Speed is a feature of your application and if it is not fast enough, then it is not good enough. Performance is a quality attribute of software systems and cannot be considered as just an afterthought. It should play an integral part throughout the life cycle of a software application. It is a requirement of the system and, like other requirements, it must be unambiguous, measurable, and testable.

When we discussed requirements in `Chapter 3`, *Understanding the Domain*, it was stated that requirements must be specified clearly, measurable with specific values/limits when appropriate, and testable so that it can be determined whether the requirement has been satisfied. For example, it is not sufficient to simply state that the *web page must load in a timely manner*. In order to make it unambiguous, measurable, and testable, it would have to be written to state that the *web page must load within two seconds*.

Treating performance as a requirement also means that we should have tests for it. We can measure how long it takes to execute tests and assert that they can be completed within a time limit. While performance tests are not executed as often as unit tests, it should be easy to execute performance tests regularly.

Page speed affects search rankings

Page speed is a consideration in a site's mobile search ranking in Google search results. Currently, this criterion only affects pages with the slowest performance, but it shows the importance Google places on web page performance. For customer-facing websites, you do not want performance to negatively affect your site's search ranking.

Defining performance terminology

Before we explore the topic of performance further, let's define some of the common terms related to performance.

Latency

Latency is the amount of time (or delay) it takes to send information from a source to a destination. A phrase you may hear regarding latency is that it is the time spent *on the wire*, since it represents the amount of time a message spends traveling on a network. Something is *latent* if it is dormant and we must wait to perform any further processing while a message is traveling across a network.

Latency is usually measured in milliseconds. Factors such as the type of network hardware being utilized, the connection type, the distance that must be traveled, and the amount of congestion on the network all affect latency.

In many instances, a significant portion of the total latency takes place between your office or home and the **internet service provider** (**ISP**). This is known as **last-mile latency** because even if data travels across the country or even the world, it can be the first or last few hops that contribute most to the total latency.

Throughput

Throughput is a measure of a number of work items per a particular time unit. In the context of a network, it is the amount of data that can be transferred from one location to another in a given amount of time. It is typically measured in **bits per second** (**bps**), **megabits per second** (**Mbps**), or **gigabits per second** (**Gbps**).

In the context of application logic, throughput is how much processing can be done in a given amount of time. An example of throughput in this context would be the number of transactions that can be processed per second.

Bandwidth

Bandwidth is the maximum possible throughput for a particular logical or physical communication path. Like throughput, it is typically measured in terms of a bit rate, or the maximum number of bits that could be transferred in a given unit of time.

Processing time

Processing time is the length of time that it takes for a software system to process a particular request, without including any time where messages are traveling across the network (latency). Sometimes a distinction is made between server processing time and client processing time.

A variety of things can affect processing time, such as how the application code is written, the external software that works in conjunction with the application, and the characteristics of the hardware that is performing the processing.

Response time

Response time is the total amount of time between the user making a particular request and the user receiving a response to that request. Although some people use the terms latency and response time interchangeably, they are not synonymous. For a given request, response time is a combination of both the network latency and the processing time.

Workload

Workload represents the amount of computational processing a machine has been given to do at a particular time. A workload uses up processor capacity, leaving less of it available for other tasks. Some common types of workload that may be evaluated are CPU, memory, I/O, and database workloads.

Taking regular measurements of workload levels will allow you to predict when peak loads for your application take place and also allow you to compare the performance of your application at different load levels.

Utilization

Utilization is the percentage of time that a resource is used when compared with the total time that the resource is available for use. For example, if a CPU is busy processing transactions for 45 seconds out of a one-minute timespan, the utilization for that interval is 75%. Resources such as CPU, memory, and disk should be measured for utilization in order to obtain a complete picture of an application's performance. As utilization approaches the maximum throughput, response times will rise.

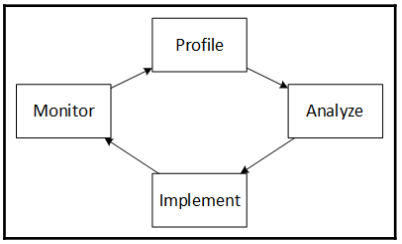
Taking a systematic approach to performance improvement

When looking to improve the performance of an application, the entire development team should be involved. Teams will have greater success at optimizing performance when the entire team, and not just certain individuals, take ownership of the performance of the software application.

When engaging in an effort to improve performance, it can be helpful to follow a systematic approach. An iterative process that consists of the following steps can be used for performance improvement:

- Profiling the application
- Analyzing the results
- Implementing changes
- Monitoring changes

The following diagram illustrates this process:



Let's now take a closer look at each of the steps in the process.

Profiling an application

The first step is to *profile* the application. Profiling is an analysis of a software system that results in measurements of the system's execution. Development teams should not be guessing where performance issues exist, as they will not always be where you expect. Rather than guessing, development teams should be acquiring precise measurements. These measurements can then be used to make decisions. Examples include how much time a particular method takes to execute, how often a method is called, the time spent in communication, the amount of I/O that is taking place, and how much of the CPU and memory is being used.

There are profiling tools, called *profilers*, available that can be leveraged to profile an application. Two broad categories of how profilers collect information are through instrumentation and by sampling. An effective profiling strategy might use both types of profilers to understand where performance issues may exist in a software system.

Instrumentation

When **instrumentation** is used, code is added to the software system being profiled in order to collect information. For example, to collect data on the time spent in a method and to get a count of how many times the method is used, instrumentation code is executed at the beginning and end of the method.

Instrumentation code can be manually added by the development team. However, profilers can add instrumentation automatically. Some profilers that use instrumentation modify source code, while others work at runtime. Either way, instrumentation can provide a great level of detail. However, a disadvantage of instrumentation is that the instrumentation code can affect the measurements. The degree of the effect really depends on what data is being collected and the extent of the instrumentation.

For example, the instrumentation code itself takes some time to execute. Profilers can take this into consideration by calculating the overhead they incur and subtracting that amount from their measurements. However, adding code to a method can change CPU optimizations and change the way that it executes the method. Consequently, very short methods can sometimes yield inaccurate results.

Statistical profilers

Profilers that work by **sampling**, which are sometimes known as **statistical profilers**, let applications execute without any runtime modifications. This type of profiling is conducted outside of the application's process and overcomes some of the disadvantages of instrumentation.

An **operating system** (**OS**) interrupts the CPU at regular intervals, giving it an opportunity for process switching. Sampling works by collecting information during these interruptions. Sampling is less intrusive than instrumentation, as it allows the software system to execute close to its normal speed. The downside is the fact that the data collected are is often an approximation and is not as numerically accurate as data that can be collected through instrumentation.

Analyzing the results

Once performance data is collected through profiling, it can be used to identify performance issues and areas of the application that are bottlenecks. **Bottlenecks** are parts of the software system that limit performance. These parts in the software are unable to keep pace given their capacity and a particular amount of work, which in turn slows down the overall performance of the application.

The focus of the performance improvement effort should be on optimizing the bottlenecks of the application. Software architects should not focus their attention on optimizing non-bottlenecks unless all of the identified bottlenecks have been addressed and there is time to optimize non-bottlenecks.

Some common bottlenecks include CPU, memory, network, database, and disk utilization. Different problems will lead you to different solutions. For example, if the network is too slow, you can look into ways to send less data across, such as compressing or caching data. If the database is too slow, you can work with the **database administrator** (**DBA**) to add indexes, optimize queries, make use of stored procedures, and possibly denormalize some of the data. If the CPU is the bottleneck, you can look into getting a faster processor, adding processors, storing/caching data so that it doesn't need to be calculated, or making improvements to the algorithms being used.

Some bottlenecks will lead you to conclude that you need to either scale horizontally or scale vertically. Vertical scaling involves increasing the capacity of existing servers by adding resources, such as adding memory and processors, replacing existing processors with faster ones, and increasing the size of available disk space. Horizontal scaling involves adding servers to your pool of resources in order to scale wider and handle more traffic.

Implementing changes

Profiling the application and analyzing the results are necessary steps prior to implementing any changes to improve performance because we do not want to make changes unless we know that it will be worth it. Once those steps are complete, though, we are ready for the development team to actually implement the changes based on the results of the previous steps.

The analysis may identify multiple areas that need improvement. Software architects should consider implementing one set of changes at a time, so as not to mix results and make it more difficult to recognize new performance issues that may have been introduced. When selecting which set of changes to implement for a particular iteration, the most important bottleneck and the one that is expected to provide the biggest payoff should be prioritized.

Monitoring results

Even once changes have been implemented to improve performance, the process is not complete. We must monitor the results to determine whether the changes that were implemented resolved the performance issues that were identified.

When changes are implemented to fix a bottleneck, either the performance issue will remain unresolved, it will be fixed, or the bottleneck will be transferred to another part of the system. If the issue is not fixed, one should consider whether it is appropriate to undo the changes that were made. Software architects need to be aware that eliminating one bottleneck may reveal another one. We must monitor the results because we may need to conduct additional iterations of the performance improvement process.

Even if our application is now performing in a satisfactory way, it must be monitored because things can change over time. As the source code changes with the introduction of new features and bug fixes, new performance issues and bottlenecks may be created. Other changes may also occur over time, such as a change in how many users the application has and how much traffic the application is generating.

Server-side caching

Software architects should take advantage of caching in order to improve performance and scalability. Caching involves copying data that may be needed again to fast storage so that it can be accessed quicker in subsequent uses. We will discuss HTTP caching and the use of content delivery networks in the *Improving web application performance* section later in this chapter. In this section, we will focus on **server-side caching** strategies.

Server-side caches can be used to avoid making expensive data retrievals from the original data store (for example, a relational database) repeatedly. The server-side cache should be placed as close to the application as possible to minimize latency and improve response times.

The type of storage used for a server-side cache is designed to be fast, such as an in-memory database. The more data and users that an application has to handle, the greater the benefits of caching.

Caching data in distributed applications

In distributed applications, there are two main types of data caching strategies that you can use. One is the use of a private cache and the other is a shared cache. Keep in mind that you can use both strategies in a single application. Some data can be stored in a private cache, while other data can be stored in a shared cache.

Using a private caching strategy

A *private cache* is held on the machine that is running the application that is using it. If multiple instances of an application are running on the same machine, then each application instance can have its own cache.

One of the ways that data is stored in a private cache is in-memory, which makes it extremely fast. If there is a need to cache more data than can fit in the amount of memory available on the machine, then cached data can be stored on the local file system.

In a distributed system using the private caching strategy, each application instance will have its own cache. This means that it is possible for the same query to yield different results depending on the application instance.

Using a shared caching strategy

A *shared cache* is located in a separate location, possibly accessible through a cache service, and all application instances use the shared cache. This resolves the issue of different application instances potentially having different views of cached data. It also improves scalability because a cluster of servers can be used for the cache. Application instances simply interact with the cache service, which is responsible for locating the cached data in the cluster.

A shared cache is slower than a private cache because rather than being available on the same machine as the application instance, it is located somewhere else; there will be some latency involved in interacting with the cache. However, if a greater level of consistency with data is important, the extra latency may be worth it.

Priming the cache

Software architects should consider *priming the cache*. This means that an application pre-populates the cache at application startup with data that will either be needed at startup, or is widely used enough that it makes sense to make the data available in the cache right from the start. This can help to improve performance as soon as initial requests are received by the server.

Invalidating cached data

Phil Karlton, while working at Netscape, once said:

> *"There are only two hard things in Computer Science: cache invalidation and naming things."*

The joke is funny because there is truth to it. Data in a cache may become stale if it is changed after it was placed in the cache. *Cache invalidation* is the process of replacing or removing cached items. We must ensure that we are handling cached data properly so that stale data is replaced or removed. It may also be necessary to remove cached items if the cache becomes full.

Expiring data

When data is cached, we can configure the data to expire from the cache after a specified amount of time. Some caching systems allow you to configure a system-wide expiration policy in addition to an expiration policy for an individual cached item. The expiration is typically specified as an absolute value (for example, 1 day).

Evicting data

A cache may become full, in which case the caching system must know which items it can discard in order to make room for new data. The following are some of the policies that can be used to evict data:

- **Least recently used** (**LRU**): Based on the assumption that cached items that have recently been used are the most likely to be used again soon, this discards items that were least recently used first.
- **Most recently used** (**MRU**): Based on the assumption that cached items that have been recently used will not be needed again, this discards items that were most recently used first.

- **First-in, first-out** (**FIFO**): Like a FIFO queue, this discards the item that was placed in the cache first (oldest data). It does not take into consideration when the cached data was last used.
- **Last-in, first-out** (**LIFO**): This approach is the opposite of FIFO in that it discards the item that was placed in the cache most recently (newest data). It does not take into consideration when the cached data was last used.
- **Explicitly evicting data**: There are times when we want to explicitly evict data from a cache, such as after existing data is deleted or updated.

Cache usage patterns

There are two main ways that an application works with a cache. The application can either maintain the cache data itself (known as a cache-aside pattern), including reading/writing to the database, or it can treat the cache as the system of record and the cache system can handle reading/writing to the database (including read-through, write-through, and write-behind patterns).

Cache-aside pattern

In the cache-aside pattern, the application is responsible for maintaining the data in the cache. The cache is kept *aside* and it doesn't interact with the database directly. When values from the cache are requested by the application, the cache is checked first. If it exists in the cache, it is returned from there and the system-of-record is bypassed. If it does not exist in the cache, the data is retrieved from the system-of-record, stored in the cache, and returned.

When data is written to the database, the application must handle potentially invalidated cached data and ensure that the cache is consistent with the system-of-record.

Read-through pattern

With the read-through pattern, the cache is treated as the system-of-record and has a component that is able to load data from the actual system-of-record (the database). When an application requests data, the cache system attempts to get it from the cache. If it does not exist in the cache, it retrieves the data from the system-of-record, stores it in the cache, and returns it.

Write-through pattern

A caching system that uses the write-through pattern has a component that has the ability to write data to the system-of-record. The application treats the cache as the system-of-record and when it asks the caching system to write data, it writes the data to the system-of-record (the database) and updates the cache.

Write-behind pattern

The write-behind pattern is sometimes used instead of the write-through pattern. They both treat the cache as the system-of-record but the timing of the write to the system-of-record is slightly different. Unlike the write-through pattern, in which the thread waits for the write to the database to complete, the write-behind pattern queues the writing of the data to the system-of-record. The advantage of this approach is that the thread can move on quicker, but it does mean that there is a short time when the data between the cache and the system-of-record will be inconsistent.

Improving web application performance

In this section, we will look at techniques that can be used to improve the performance of web applications. These techniques include, but are not limited, to:

- HTTP caching
- Compression
- Minification
- Bundling
- HTML optimization
- HTTP/2
- Content delivery networks (CDNs)
- Web font optimization

Leveraging HTTP caching

Many roundtrips between a client and a server may be necessary to load a page, and retrieving resources for that page from the server can take up significant amounts of time. The ability to cache resources that might be needed again so that they do not need to be transferred over the network on subsequent trips is an important part of improving web application performance.

Browsers are capable of caching data so that it doesn't need to be fetched from a server again. Resources such as CSS or JavaScript files might be shared across multiple pages of your web application. As a user navigates to various pages, they will need these resources multiple times. In addition, users may return to your web application at some point in the future. In both cases, taking advantage of **HTTP caching** will improve performance and benefit the user.

To take advantage of HTTP caching, each response from your web server must include the appropriate HTTP header directives. The cache policy you decide to implement is ultimately dependent on the application's requirements and the type of data being served. Each resource may have different requirements related to caching that should be considered. Using the various header directives available to you will provide you with the flexibility to meet your requirements. Let's look at some of the header directives you can use to control HTTP caching.

Using a validation token

A common scenario with HTTP caching occurs when a response has expired from the cache but has not changed in any way. The client would be required to download the response again, which is wasteful since the resource has not changed.

A validation token in the `ETag` header of a response can be used to check whether an expired resource has changed. Clients can send the validation token along with a request. If a response has expired from the cache but the resource has not changed, there is no reason to download it again. The server will return a 304 Not Modified response and the browser will then know that it can renew the response in the cache and use it.

Specifying cache-control directives

Cache-control directives in a response can control whether the response should be cached, under what conditions it can be cached, and for how long it should be cached. If a response contains sensitive information that you do not want cached, a *no-store* cache-control directive can be used, which will prevent browsers as well as any intermediate caches (for example, a content delivery network), from caching the response. Alternatively, a response can be marked as *private*, which will allow caching in a user's browser but not in any intermediate caches.

A cache-control directive that is slightly different than *no-store* is the *no-cache* directive. It is used to specify that a response should not be used from the cache until a check is performed with the server first to see whether the response has changed. The validation token must be used to make this determination, and only if the resource has not changed can the cache be used.

The *max-age* directive is used to specify the maximum amount of time, in seconds, that a response can be reused from the cache. The value is relative to the time of the request.

You may find yourself in a situation where you want to invalidate a cached response even though it has not expired yet. Once a response is cached, it will continue to be used unless it expires or the browser's cache has been cleared in some way. However, there may be times when you want to change a response before it has expired. This can be accomplished by changing the URL of the resource, which will force it to be downloaded. A version number, or some other identifier such as a fingerprint of the file, can be included as part of the filename. Using this technique provides differentiation between different versions of the same resource.

Taking advantage of compression

Compression is an important technique for improving performance. It is the use of an algorithm to remove redundancy in a file in order to make it smaller. This improves transfer speed and bandwidth utilization.

Software developers do not need to programmatically compress data that is to be transmitted. Servers and browsers have compression implemented already. As long as both the server and the browser understand the compression algorithm, it can be used. It is just a matter of ensuring that the server is configured properly.

The two main types of compression that are used to improve web performance are file compression and content-encoding (end-to-end) compression.

File compression

Files that you transmit, such as images, video, or audio, can have high rates of redundancy. When a web page is downloaded, images might account for the majority of the bytes being downloaded. These types of files should be compressed to save storage space and increase transfer speed.

There are various tools and algorithms that can be used to compress different file formats. Among the choices you can make, depending on your needs, is whether to use a lossless or a lossy compression algorithm.

Lossless compression

With lossless compression, all of the bytes from the original file can be recovered when the file is decompressed. This type of compression algorithm may be necessary if the file you are compressing cannot afford to lose any information.

For example, if you are compressing a text file of data, source code for a program, or an executable file, you cannot afford to lose any of the contents and would want to use a lossless compression algorithm. For image, video, and audio files, you may or may not require lossless compression, depending on your needs for file size and quality.

Graphics Interchange File (**GIF**) and **Portable Network Graphics** (**PNG**) are examples of image file formats that provide lossless compression. If animation is required, you will want to use the GIF format. If you want to preserve high-quality images and not lose any fine detail, the PNG image format should be used.

Lossy compression

If a lossy compression algorithm is used, some of the bytes from the original file will be lost when the file is decompressed. For a file in which you can afford to lose some bytes, you can achieve a smaller file size as compared to using a lossless compression algorithm.

This type of compression works well with images, video, and audio because the loss of redundant information may be acceptable. There are different degrees of lossy compression, and how aggressive you are with the optimization depends on the trade-off you are willing to make between file size and quality. In some cases, you can use lossy compression and there will be no perceptible difference to the user.

Joint Photographic Experts Group (**JPEG**) is an example of an image file format that provides lossy compression. If you do not need the highest quality image and can afford to lose some fine detail in the image, JPEG can be used.

Content-encoding (end-to-end) compression

Significant performance improvements can be made when using content-encoding. The server compresses the body of an HTTP message prior to sending it to the client. It will remain compressed (end-to-end compression) until it reaches the client. Any intermediate nodes that it may pass through while traveling to the client do not decompress the message. Once the message reaches the client, the client decompresses the body of the HTTP message.

In order to use content-encoding, the browser and server must agree on the compression algorithm to use via *content negotiation*. Content negotiation is the process of selecting the best representation of particular content.

There are a number of different types of compression, but **gzip** is the most common. It is a lossless type of compression. Although it can be used for any stream of bytes, it works particularly well on text. Brotli (content-encoding type of **br**) is an open source, lossless data compression library. It is newer than gzip, but it is gaining support and popularity.

You should take advantage of content-encoding as much as possible, except when transferring files that have already been compressed with the aforementioned file compression, such as image, video, and audio files. This is because you will typically not gain anything by compressing something twice, and it could even lead to a file size that is slightly larger than if it is just compressed once.

Minifying resources

Minification is the process of removing all unnecessary or redundant data from a resource. It can be used to remove characters from source code that are not needed without changing any of the functionality.

Files such as JavaScript, HTML, and CSS are great candidates for minification. Although the minified files that result from the process are not as human-readable as their original counterparts, the file size will be smaller, resulting in faster load times.

For example, let's take the following JavaScript code:

```
// Class representing a rectangle
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
```

```
    // Method to calculate area
    calculateArea() {
        return this.width * this.height;
    }
}
```

After minifying it, it appears as follows:

```
class
Rectangle{constructor(t,h){this.height=t;this.width=h}calculateArea(){retur
n this.width*this.height}}
```

You can see that unnecessary characters for formatting and code comments have been removed and the names of the constructor parameters have been shortened. There are a number of tools available to you that can minify files. Some of the tools focus on a particular type of file (for example, JavaScript, HTML, or CSS). It is best to minify files prior to using the compression technique discussed in the *Taking advantage of compression* section.

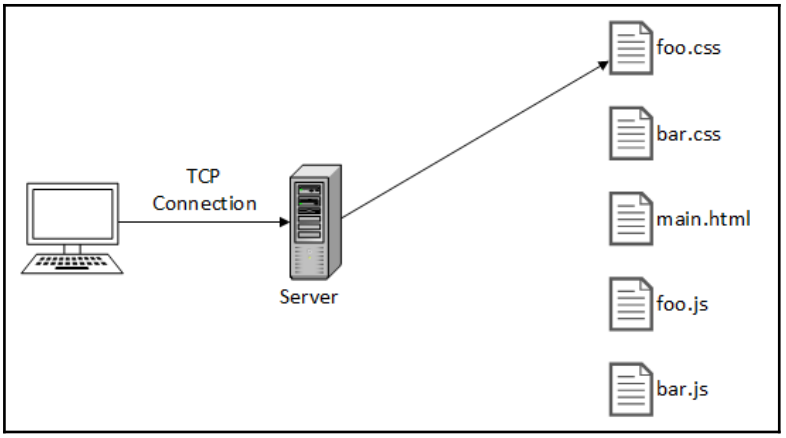
It is a good practice to keep two versions of code files that are minified: a version that has not been minified for debugging purposes and a minified version for deployment. They can be given different filenames so that it is clear which one is the minified version. For example, `invoice.min.js` could be used as the name for the minified version of `invoice.js`.

Bundling resources

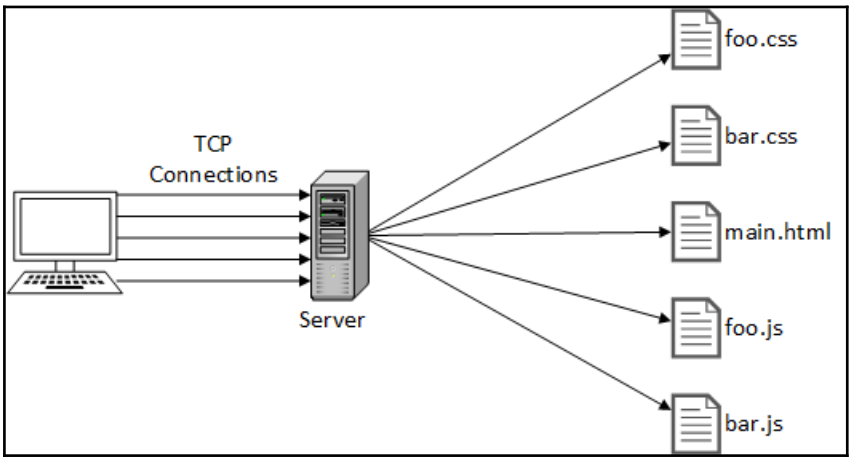
The first step in reducing the number of HTTP requests that need to be made in order to load a page is to remove all unnecessary resources. Once that is done, most web pages will still require multiple files of the same type, such as JavaScript or CSS files, in order to load. During development, it makes sense to separate this type of code into multiple files. However, the use of more individual files translates into more HTTP requests.

Bundling is the process of combining multiple files of the same type into a single file, which can then be transferred in a single request. The technique of bundling files is sometimes referred to as concatenating files. Fewer HTTP requests lead to faster page load performance. Bundling is a technique that is complementary with minifying files and the two are often used in conjunction with each other.

Bundling is an effective technique when we are using HTTP/1.1. In order to understand the reasons behind that, let's examine how assets are sent. In order to load a web page, the browser has to load each of the files that it needs, one at a time, over a connection, as shown in the following diagram:

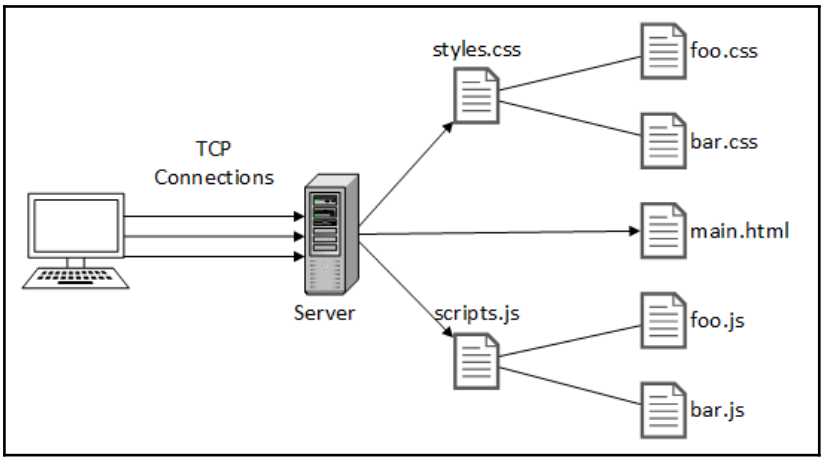


This process is too slow, so in order to get around this issue, browsers open up multiple connections per host, shown as follows:



The maximum number of simultaneous connections that can be made per host varies by browser but a common number is six. The browser will handle these connections so application developers do not need to make any modifications to their application to take advantage of that feature. There is some overhead involved with setting up each connection but it is worth it in order to have multiple connections available for communication.

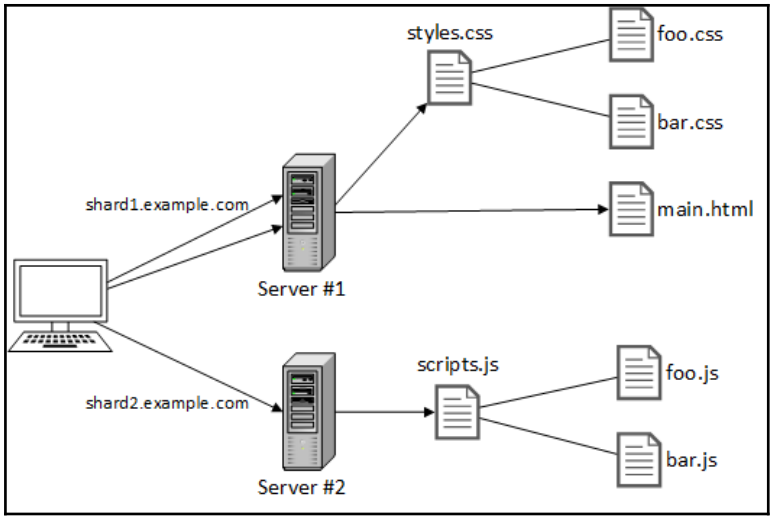
We can improve this approach by bundling the files so that fewer HTTP requests, and possibly fewer connections, will be necessary to get all of the required assets. In the following diagram, all of the CSS files are bundled together in **styles.css**, and all of the JavaScript files are bundled together in **scripts.js**:



A downside of caching is that it can cause the cache to be invalidated more frequently. Without bundling, we can control the caching of each individual file. Once we start bundling, if any of the files within the bundle have changed, the entire bundle will need to be downloaded to clients again. If a file that is changed is contained within more than one bundle, it could cause multiple bundles to be downloaded again.

Even with the use of bundling, the number of assets that a web page needs may be higher than the maximum number of connections. This means that additional requests for assets from the same host are queued by the browser and will have to wait until a connection becomes available. To work around this limitation, the technique of **domain sharding** was introduced.

If there is a limit to the number of connections per domain, a workaround is to introduce additional domains. Domain sharding is a technique in which resources are split among multiple domains, allowing more to be downloaded in parallel. Instead of using the same domain (for example, `www.example.com`), we can use multiple subdomains (**shard1.example.com**, **shard2.example.com**, and so on). Each shard is allowed the maximum number of connections, increasing overall parallelism and allowing more assets to be transferred at the same time. In the following diagram, we are only returning three assets, which doesn't exceed the maximum number of connections per host. However, if we did need more assets, we could retrieve more before queuing by the browser would become necessary:



There is overhead involved with adding shards, though, such as additional DNS lookups, the additional resources required on both ends, and the fact that application developers will need to manage how to split up their resources.

Let's take a look at HTTP/2 now and learn how it can improve performance. Differences between HTTP/1.x and HTTP/2 affect how we want to approach techniques such as bundling and domain sharding.

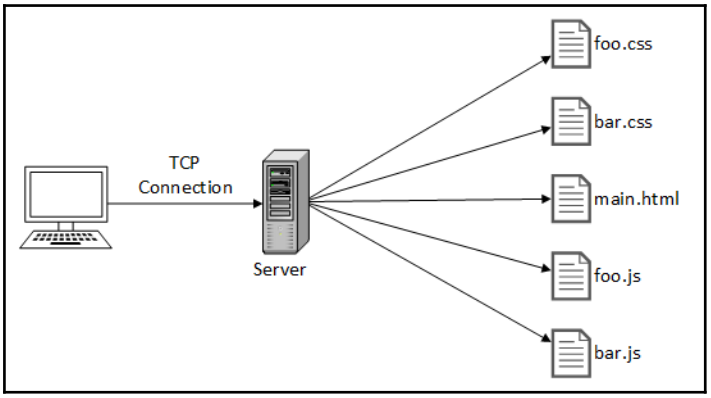
Using HTTP/2

HTTP/2 is the latest version of the application layer protocol for data communication. It is not a complete rewrite of the protocol. The HTTP status codes, verbs, methods, and most of the headers that you are already familiar with from using HTTP/1.1 will continue to be the same.

One difference between HTTP/2 and HTTP/1.1 is the fact that HTTP/2 is binary, whereas HTTP/1.1 is textual. HTTP/2 communication consists of binary-encoded messages and frames, making it more compact, efficient to parse, and less error-prone. The fact that HTTP/2 is binary is what enables some of the other HTTP/2 features that improve performance.

Multiplexing

One of the most important features of HTTP/2 is *multiplexing*. Multiplexing is the ability to send multiple HTTP requests and receive multiple HTTP responses asynchronously through a single TCP connection, as shown in the following diagram:



HTTP/1.1 does not support multiplexing, which is what led to the various workarounds to improve performance. With HTTP/2, we should no longer be concatenating files into a small number of large bundles. The expensive cache invalidation of a bundle that is necessary when any of the files in the bundle are changed can now be avoided or minimized. We can now transfer more granular assets, either by not bundling at all or having an increased number of bundles, where each one contains a small number of related files rather than just a few bundles containing many. A more granular approach allows us to provide an optimal cache policy for each individual file or bundle, maximizing the amount of content that is coming from our cache.

Another technique that is no longer necessary with HTTP/2 is domain sharding. Through the use of multiplexing, we can download multiple assets at the same time using a single connection. The overhead for each shard, which may have been worth it with HTTP/1.1, is no longer necessary.

Server push

HTTP/2 provides a feature in which the server can *push* responses that it thinks a client will need. When a resource is requested from a client, it may contain references to other resources that are needed. Rather than wait for the client to send additional requests for these required resources, the server already knows what resources will be needed and can proactively send them.

This feature is similar to *inlining* a resource, which is a technique that is sometimes used to improve performance by reducing the number of requests necessary. Inlining is accomplished by embedding a resource, such as JavaScript, CSS, or images, into an HTML page. With server push, there is no longer a need to inline resources. We get the same advantages of inlining but with the added benefit of keeping the assets in separate files, each with their own caching policies.

There are a few caveats with the server push feature. You should be careful not to push too many assets at once; you do not want to delay rendering of the page and negatively affect perceived performance. You should put thought into what assets you are pushing and be selective about it.

If server push is not used properly, resources that a client already has could be transferred to it unnecessarily, which would actually hurt performance. As of today, use of the server push feature may require some experimentation in order for it to be used in an optimal way. Some web servers have the functionality to mitigate the problem of pushing assets that the client does not need, and some browsers may introduce a *cache digest* so that a client can let the server know what assets it already has in its local cache.

Header compression

HTTP/2 performs header compression to improve performance. With HTTP/1.x, headers are always sent as plain text, but HTTP/2 uses the HPACK compression format to reduce the size. HPACK is used for compression with HTTP/2 because it is resilient to some of the security attacks that target compression, such as Compression Ratio Info-leak Made Easy (CRIME).

HPACK uses Huffman coding, a lossless data compression algorithm that can significantly reduce the size of the headers, reducing latency and improving performance.

Implementing HTTP/2

In order for a client to use HTTP/2, all that is needed is that the browser must support it. The latest versions of most browsers support HTTP/2. On the server side, the steps required to configure HTTP/2 support varies. A number of web servers provide support for HTTP/2 and the list continues to increase. In most cases, servers will need to support both HTTP/1.1 and HTTP/2, so typically a server needs to be configured to fall back to HTTP/1.1 if a client does not support HTTP/2.

Using content delivery networks (CDNs)

Users visiting a website may potentially be located anywhere in the world. Round trips between clients and servers will take longer if the distance between them is greater. The additional latency may be just milliseconds but it all contributes to the overall time it takes to receive a response.

Content delivery networks (**CDNs**) are a geographically distributed group of servers that can deliver content to users quickly. The nodes of a CDN are deployed in multiple locations so that they are distributed spatially. This provides us with the ability to reduce network latency and serve up content closer to the end users, which improves load times.

CDNs are great for transferring website content, such as JavaScript, HTML, CSS, image, video, and audio files. In addition to reducing the physical distance between users and content, CDNs improve load times through efficient load balancing, caching, minification, and file compression.

The reliability and redundancy of web applications are increased when a CDN is used because when traffic increases, it can be load balanced among multiple servers. If a server, or even an entire data center, is having technical issues, traffic can be routed to servers that are operational. CDNs can also help to improve security by mitigating **distributed denial-of-service** (**DDoS**) attacks and maintaining up-to-date TLS/SSL certificates.

Optimizing web fonts

Good typography is an important part of designing a good user interface, readability, accessibility, and branding. There was a time when web designers were limited in the fonts they could use because there were only so many fonts that were guaranteed to be available on all systems. These were known as *web safe fonts*.

It is possible to use fonts other than just the *web safe fonts*. For instance, in CSS, you can use the `font-family` property to specify a list of fonts that can be used for an element, as follows:

```
p {
  font-family: Helvetica, Arial, Verdana, sans-serif;
}
```

With this approach, the browser will use the first one that it finds available on the system. A disadvantage of using this approach is that during testing you have to ensure that all of the fonts will work properly with your application.

A CSS feature called **web fonts** was introduced to overcome some of the challenges. It provides you with the ability to download font files so that any browser that supports web fonts can make the fonts that you want to use for your page available. Text using web fonts is selectable, searchable, zoomable, and will look good in a variety of screen sizes and resolutions.

However, the use of web fonts means that additional resources must be loaded. If a website or web application is using web fonts, it is important to consider them as part of your overall web performance strategy. Optimization of web fonts can reduce the overall size of a page and decrease rendering times. One thing that you should do is minimize the number of fonts (and their variants) that you use on your pages to minimize the number of resources that are needed.

To use web fonts, you must first select the one or more fonts that you want to use and consider the character sets that you need to support based on any localization requirements. The size of a font file is dependent on the complexity of the shapes in the various characters that make up the font.

Unfortunately, there is no standard on font formats, which means that different browsers support different font formats. This lack of a standard means that as of right now, you will need to support four different font formats for each font, and they are as follows:

- **Web Open Font Format version 2** (**WOFF 2.0**)
- **Web Open Font Format version 1** (**WOFF**)
- **TrueType font** (**TTF**)
- **Embedded Open Type** (**EOT**)

Once the web fonts have been selected, the `@font-face` CSS rule allows you to use a web font by allowing you to specify the font and the URL location where the font data can be found. Regardless of which web font you select and which of the four font formats is being used by a particular user, compression is effective in reducing the font size and should be used to improve performance. WOFF 2.0 and WOFF have built-in compression, but the TTF and EOT formats are not compressed by default, so servers should use compression when delivering these formats.

Web fonts can be large Unicode fonts with support for a variety of characters, not all of which will be needed at a given time. The `unicode-range` property in `@font-face` can be used to split up a font into multiple subsets so that only the characters that are actually needed will be downloaded.

One more note about optimizing fonts is to keep in mind the fact that font resources are not updated frequently. You should ensure that this type of resource is cached with a caching policy that will allow them to live in the cache for a long period of time, with a validation token so that even once it expires, it can be renewed in the cache and not downloaded again as long as it has not changed.

Optimizing the critical rendering path

An important step in reducing the amount of time that it takes for a page to render is to optimize the **critical rendering path** (**CRP**). The critical rendering path is the set of steps in between a browser receiving bytes from a server (for example, HTML, CSS, and JavaScript files) and the processing involved to render pixels on the device's screen.

Before a page can be rendered by a browser, it must construct both the **Document Object Model** (**DOM**) and the **CSS Object Model** (**CSSOM**). The HTML and CSS markup for the page are needed for this process. The DOM and the CSSOM are then combined to form a render tree, which has both the content as well as the style information for what will be visible on the screen.

Once the render tree has been constructed, the browser moves to the layout stage where it calculates the size and position of the various visible elements. Finally, the paint stage is reached, where the browser uses the results of the layout to paint pixels to the screen.

Optimizing the critical rendering path is the process of minimizing the time it takes to perform these various steps. We are mostly concerned with the portion of the page that is *above the fold*, which refers to the part of the page that is visible without scrolling. The term is a reference to the upper half, or visible portion, of a folded newspaper. Until the user scrolls down a page, which may not even occur, they will not see what is *below the fold*.

We want to prevent render blocking by reducing, as much as possible, the resources that will prevent the content above the fold from rendering. The initial step in this process is to determine what resources are truly necessary for the initial rendering of a page. We want to get these critical resources to the client as quickly as possible to speed up the initial rendering. For example, the HTML and CSS that are necessary to create the DOM and the CSSOM are render-blocking resources, so we want to get them to the client quickly. Using some of the techniques described previously for web application performance, such as compression and caching, can help to load critical resources faster.

Resources that are not required for the above-the-fold content or are otherwise not critical for the initial rendering can either be eliminated, their download could be deferred, or they can be loaded asynchronously. For example, if there are image files that are not needed for initial rendering, they can be deferred, and in order to prevent DOM construction any blocking JavaScript files can be loaded asynchronously.

Understanding what is involved in rendering a page, taking the time to think about what the critical resources are for the initial rendering, and optimizing the critical rendering path will allow a page to be constructed faster. Making a web page almost immediately visible and usable greatly improves a user's overall experience, which reduces the bounce rate and increases the conversion rate.

Database performance

A key part of a software system is the database. So, when looking to improve the performance of a system, improving database performance must be part of that effort. In this section, we will take a look at some of the things that can be done to improve database performance.

Designing an efficient database schema

The foundation of achieving peak database performance is an efficient and properly designed database schema. As a software architect, you may be working with a DBA who will be responsible for database design. However, it is good to be familiar with the different aspects of achieving a good database design.

Normalizing a database

Normalization is the process of designing tables (relations) and columns (attributes) so that they not only meet data requirements, but minimize data redundancy and increase data integrity.

In order to meet these goals, a database should contain the minimal number of attributes necessary to meet the requirements. Attributes with a close logical relationship should be placed together in the same relation. Redundancy of attributes should be kept to a minimum, which makes it easier to maintain data consistency and will minimize the size of the database.

Denormalizing a database

For performance and scalability reasons, there may be cases where it makes sense to denormalize part of the database. It is important to differentiate between a database that has not been normalized, and one that has been normalized and then is denormalized later. A database should be normalized first and then if there are cases where it makes sense to denormalize, it should be done after careful consideration.

Denormalization is a strategy used to improve performance, typically by shortening the execution time of certain queries. This can be accomplished by storing redundant copies of some data or grouping data together in order to minimize joins and improve query performance.

While denormalization might improve read performance, it will negatively affect write performance. Some mechanisms, such as a compensating action, will be required to keep redundant data consistent. Using database constraints can help to enforce rules that will keep the data consistent even when it is denormalized. Redundant data will also make the database larger and therefore take up more disk space.

Another reason to introduce denormalization is to keep historical data. For example, let's say that we have an address table and an orders table, and that each order is associated with an address. An order is created, and then in the future, that address is updated. Now when you look at the old order, you see the new address and not the address given at the time the order was created. By storing the address field values with each order record, you can maintain this historical data. It should be noted, however, that there are ways to accomplish this without denormalization. For instance, you could treat an address like a value object (immutable), and simply create a new address record when one is modified, leaving the one associated with the old address intact. Also, in an event-driven system, or one in which a data audit is being kept that stores the modifications that have been made to a record, you could *reconstruct* what the address was at the time the order was created.

Identifying primary and foreign keys

All of the primary and foreign keys for all of the tables in the database should be identified. The *primary key* of a table is the column, or combination of columns, that uniquely identify a row in the table.

Sometimes a row in a table must reference a row from another table. A *foreign key* is a column or combination of columns that hold the primary key value for a row in another table so that it can be referenced.

Database constraints based on the primary and foreign keys should be created to enforce data integrity. A p*rimary key constraint* for a table consists of the one or more columns that make up the primary key and a *foreign key constraint* consists of the one or more columns that make up the foreign key.

Selecting the most appropriate data types

When designing a database table, we should select the most appropriate data type for each column. In addition to the data type, the size and nullability of the column should also be considered. We want to choose a data type that will sufficiently hold all possible values but also be the smallest data type that is necessary. This will maximize efficiency not just for performance but also for storage size.

Using database indexes

Database indexes can be used to improve performance and provide more efficient data access and storage. They are stored on disk and associated with a database table or view to speed up data retrieval. The two main types of indexes are the primary/clustered index and the secondary/non-clustered index.

Primary/clustered indexes

When designing a table, one approach is to keep the rows unordered and create as many secondary indexes as necessary. Such an unordered structure is known as a *heap*. Another approach is to create a *primary index*, also known as a *clustered index*, to order the rows by the primary key. It is common to have a clustered index on a table which is known as a *clustered table*. The only time you might not want one is if the table is very small (and one that you know will remain small over time), such that the overhead of storing and maintaining the index is not worth it when compared to simply searching the table.

The one or more columns that make up the primary key of a table are the columns that make up the index definition. A clustered index sorts the rows in a table based on their key values and physically stores them on disk based on that order. Each table can only have one clustered index because the rows in the table can only be sorted and stored in one order.

Secondary/non-clustered indexes

In addition to specifying a primary index, many database systems provide the ability to create *secondary indexes*, also known as *non-clustered indexes*. The performance of a database can benefit from having secondary keys available for data access.

Non-clustered indexes are defined by one or more columns that are ordered logically and serve as pointers to find the rest of the data for a given record. The order of a non-clustered index does not match the physical order of how the records are stored on disk.

Non-clustered indexes provide a way to specify an alternate key other than the primary key for accessing records in a table. The key could be a foreign key or any column that will be frequently used in joins, where clauses, ordering, or grouping. The advantage of using non-clustered indexes is to improve performance for the common ways that data might be accessed for a particular table beyond the primary key. Sometimes the primary key is not the only way, and might not even be the most widely-used way, that records are retrieved from a table.

For example, let's say we have an `Order` table and a `Customer` table, with `OrderId` and `CustomerId` being the primary keys of those two tables, respectively. The `Order` table also has a `CustomerId` column as a foreign key to the `Customer` table in order to associate orders with customers. In addition to retrieving orders by `OrderId`, the system may need to retrieve orders by `CustomerId` on a frequent basis. Adding a non-clustered index on the `CustomerId` column of the `Order` table allows for more efficient data retrieval when it is accessed with that column.

Although non-clustered indexes are an important part of performance tuning a database, each table should be analyzed carefully when deciding which non-clustered indexes, if any, to add to a table and which column or columns should make up the index. As we will learn in a moment, there is overhead related to adding indexes to a table, so we do not want to add any unnecessary ones.

Having too many indexes

There is a cost associated with adding an index to a table, so when it comes to indexes, you can have too much of a good thing. Every time that a record is added or updated in a table, an index record also has to be added or updated, incurring some additional overhead to those transactions. In terms of storage, indexes take up additional disk space, increasing the overall size of your database.

The more indexes you have on a table, the more the query optimizer of the **database management system** (**DBMS**) will have to take into consideration for a particular query. The query optimizer is a component of the DBMS that analyzes queries to determine the most efficient execution plans for them.

For example, the query optimizer decides whether a particular join in a query should do a full table scan versus using an index. It must take into account, among other things, all of the indexes on that table. As a result, an increased number of indexes on a table could adversely affect performance.

For these reasons, one should be selective when considering what indexes to add to a table. Properly selected indexes will speed up data access performance, but you do not want to create unnecessary ones as they can slow down data access and operations, such as inserts and updates.

Scaling up and out

Scaling your database server vertically (up) or horizontally (out) to improve performance is not something that should be done without an understanding that it is necessary. Prior to scaling up or scaling out, software architects and DBAs should ensure that the database schema has been designed properly and that indexes have been applied properly. In addition, the application using the database should be optimized to improve performance and remove bottlenecks.

Once those measures have been taken, if the database server is experiencing high levels of resource use, it's time to consider scaling the server up or scaling it out. For database servers, it is best to scale up first by performing actions such as replacing the server with a better machine or by adding processors and/or memory.

Scaling up should be done first because there are additional complications with scaling a database server out. When you have multiple servers, you may need to horizontally partition some of the tables and consider data replication. Plans for disaster recovery and failover are also more complex when there are multiple database servers. However, if database performance is still not at the level you need it to be after scaling up, scaling out may be necessary.

Database concurrency

A relational database system can handle many simultaneous connections. Having a database that performs well is only useful if it can handle multiple processes accessing and changing data at the same time. This is what **database concurrency** is all about.

Concurrency control ensures that database transactions that are performed concurrently maintain data integrity. We'll now begin looking at concurrency by learning about database transactions.

Database transactions

A database transaction is a sequence of operations that are performed as a single unit of work. They play an important role in maintaining data integrity and consistency even when data is being accessed and changed at the same time. They provide units of work that either complete in their entirety or will not be committed at all.

Transactions can recover from failures and keep the database in a consistent state. Transactions also provide isolation so that a record that is in the process of being modified by one transaction is not affected by a concurrent transaction that must update the same record. Transactions, once they are complete, are written to durable storage.

Optimistic versus pessimistic concurrency control

Concurrency control ensures that databases transactions that are performed concurrently maintain data integrity. Many databases offer two main types of concurrency control: optimistic and pessimistic.

Optimistic concurrency control (or optimistic locking) works under the assumption that resource conflicts between multiple users, while possible, are not common. Therefore, it allows transactions to execute without locking resources. If data is being changed, resources are checked for conflicts. If there is a conflict, only one transaction is successful while the others fail.

In contrast, pessimistic concurrency (or pessimistic locking) assumes the worst, such as assuming that more than one user will want to update the same record at the same time. In order to prevent that, it locks the appropriate resources as they are required for the duration of the transaction. Unless a deadlock takes place, pessimistic concurrency ensures that a transaction will be completed successfully. It should be noted that most database systems have different types of locks. For example, one type of lock might specify that a record that is locked can still be read by another user, while another type of lock might prevent that type of read.

CAP theorem

The **Consistency, Availability, and Partition tolerance** (**CAP**) theorem, also known as Brewer's Theorem after Eric Brewer who published it, states that a distributed system can only achieve two of the following three guarantees, but not all three:

- **Consistency**: Every read either returns the latest data or an error. Every transaction either completes successfully and is committed or is rolled back due to a failure.
- **Availability**: A system always provides a response to every request.
- **Partition tolerance**: In a distributed system (data is partitioned to different servers), if one of the nodes fails, the system should still be able to function.

Databases will stress some of these guarantees over others. A traditional relational database management system will focus on consistency and availability. They will favor strong consistency, which is also known as immediate consistency, so that any read of the data will reflect any changes that have been made to that data. These types of databases will follow an ACID consistency model.

Some databases, such as some NoSQL databases, will value availability and partition tolerance over consistency. For such databases, eventual consistency, rather than strong consistency, is acceptable. Eventually, the data will reflect all of the changes made to it, but at any given point in time it is possible to read data that may not reflect the latest changes. These types of databases follow a BASE consistency model.

ACID model

Databases that want to ensure consistency and availability will follow the **ACID consistency model**. Traditional relational databases follow the ACID model. Database transactions adhere to the ACID properties in that they must be atomic, consistent, isolated, and durable. These properties guarantee the validity of the data even when errors or failures occur. Strong consistency will place limits on performance and scalability.

Atomicity

A transaction must be an atomic unit of work, meaning that either all of its data modifications are performed or none at all. This provides reliability because if there is failure in the middle of a transaction, none of the changes in that transaction will be committed. For example, in a financial transaction, you may insert one record to represent the credit part of the transaction and another to represent the debit part of the transaction. You don't want one of those inserts to take place without the other, so you place them both as part of one transaction. Either they will both be committed or neither of them will be committed.

Consistency

After a transaction takes place, all of the data must be in a consistent state. This property ensures that all transactions maintain data integrity constraints, leaving the data consistent. If a transaction leaves data in an invalid state, the transaction is aborted and an error is reported. For example, if you had a column with a check constraint that states a column value must be greater than or equal to zero (so as not to allow negative numbers), the transaction would fail if it attempted to insert or update a record with a value less than zero for that particular column.

Isolation

Changes made by concurrent transactions must be isolated from changes made by any other concurrent transactions. Many DBMSs have different isolation levels that control the degree to which locking occurs on data being accessed. For example, a DBMS may place a lock on a record being updated so that another transaction cannot update that same record at the same time.

Durability

Once a transaction completes and is committed, its changes are persisted permanently in the database. For example, a DBMS may implement durability by writing all transactions to a transaction log. The transaction log can be used to recreate the system state at any point, such as right before failure.

BASE model

Some databases, such as some distributed NoSQL databases, focus on availability and partition tolerance. In some situations, it may be an acceptable tradeoff to have eventual consistency, rather than strong consistency, in order to focus on partition tolerance, performance, and scalability. This approach enables a higher degree of scalability and can yield faster performance. These databases use the **BASE consistency model** instead of the ACID model.

Basic availability

Most of the time, conflicts do not take place. The database is available most of the time and a response will be sent for every request. However, conflicts can occur and the response may indicate that a failure occurred when trying to access or change data.

Soft state

Rather than following consistency requirements such as those in the ACID model, the concept here is that the state of the system could change over time. Even if no additional transactions are being created, changes could take place due to eventual consistency.

Eventual consistency

A data change will eventually propagate to everywhere that it needs to go. If there are no further changes to a piece of data, eventually the data will be in a consistent state. This means it is possible to read stale data if the latest updates to it have not been applied yet.

Unlike a strong consistency model in which all data changes are atomic and the transaction is not allowed to complete until either the change finishes successfully or is rolled back due to a failure, the system will not check the consistency of every transaction.

Summary

It is more important to ensure that your code is correct than fast. Fast performance is not of any use if the application does not yield the correct results. Having said that, performance is an important part of designing and developing a successful software application. It plays a large part in the overall user experience for people who use the application. Regardless of the device they are using or their location, users expect a high level of responsiveness from their applications.

Performance is a quality attribute of the software system and performance requirements should be documented. Like all requirements, they need to be measurable and testable. The entire team should take ownership of performance. A systematic, iterative approach to performance improvement can help a development team reach their performance goals. Some problems will only be discovered later, so development teams should be prepared to analyze and optimize in an iterative way. In this chapter, you learned how to use server-side caching, about different techniques to improve web application performance, and how to improve database performance.

In the next chapter, we will explore the various security considerations that a software architect must make. We will examine the goals of security and the design principles and practices that will help us to achieve them. The chapter will cover techniques such as threat modeling, and topics such as cryptography, identity, and access management, and how to handle common web application security risks.