Scalable Web Architecture and Distributed Systems

The Architecture of Open Source Applications
Volume II: Structure, Scale, and a Few More Fearless Hacks

Kate Matsudaira

Open source software has become a fundamental building block for some of the biggest websites. And as those websites have grown, best practices and guiding principles around their architectures have emerged. This chapter seeks to cover some of the key issues to consider when designing large websites, as well as some of the building blocks used to achieve these goals.

This chapter is largely focused on web systems, although some of the material is applicable to other distributed systems as well.

1.1. Principles of Web Distributed Systems Design

What exactly does it mean to build and operate a scalable web site or application? At a primitive level it's just connecting users with remote resources via the Internet—the part that makes it scalable is that the resources, or access to those resources, are distributed across multiple servers.

Like most things in life, taking the time to plan ahead when building a web service can help in the long run; understanding some of the considerations and tradeoffs behind big websites can result in smarter decisions at the creation of smaller web sites. Below are some of the key principles that influence the design of large-scale web systems:

- **Availability:** The uptime of a website is absolutely critical to the reputation and functionality of many companies. For some of the larger online retail sites, being unavailable for even minutes can result in thousands or millions of dollars in lost revenue, so designing their systems to be constantly available and resilient to failure is both a fundamental business and a technology requirement. High availability in distributed systems requires the careful consideration of redundancy for key components, rapid recovery in the event of partial system failures, and graceful degradation when problems occur.
- **Performance:** Website performance has become an important consideration for most sites. The speed of a website affects usage and user satisfaction, as well as search engine rankings, a factor that directly correlates to revenue and retention. As a result, creating a system that is optimized for fast responses and low latency is key.
- **Reliability:** A system needs to be reliable, such that a request for data will consistently return the same data. In the event the data changes or is updated, then that same request should return the new data. Users need to know that if something is written to the system, or stored, it will persist and can be relied on to be in place for future retrieval.
- **Scalability:** When it comes to any large distributed system, size is just one aspect of scale that needs to be considered. Just as important is the effort required to increase capacity to handle greater amounts of load, commonly referred to as the scalability of the system. Scalability can refer to many different parameters of the system: how much additional traffic can it handle, how easy is it to add more storage capacity, or even how many more transactions can be processed.
- **Manageability:** Designing a system that is easy to operate is another important consideration. The manageability of the system equates to the scalability of operations: maintenance and updates. Things to consider for manageability are the ease of diagnosing and understanding problems when they occur, ease of making updates or modifications, and how simple the system is to operate. (I.e., does it routinely operate without failure or exceptions?)
- **Cost:** Cost is an important factor. This obviously can include hardware and software costs, but it is also important to consider other facets needed to deploy and maintain the system. The amount of developer time the system takes to build, the amount of operational effort required to run the system, and even the amount of training required should all be considered. Cost is the total cost of ownership.

Each of these principles provides the basis for decisions in designing a distributed web architecture. However, they also can be at odds with one another, such that achieving one objective comes at the cost of another. A basic example: choosing to address capacity by simply adding more servers (scalability) can come at the price of manageability (you have to operate an additional server) and cost (the price of the servers).

When designing any sort of web application it is important to consider these key principles, even if it is to acknowledge that a design may sacrifice one or more of them.

1.2. The Basics

When it comes to system architecture there are a few things to consider: what are the right pieces, how these pieces fit together, and what are the right tradeoffs. Investing in scaling before it is needed is generally not a smart business proposition; however, some forethought into the design can save substantial time and resources in the future.

This section is focused on some of the core factors that are central to almost all large web applications: *services*, *redundancy*, *partitions*, and *handling failure*. Each of these factors involves choices and compromises, particularly in the context of the principles described in the previous section. In order to explain these in detail it is best to start with an example.

Example: Image Hosting Application

At some point you have probably posted an image online. For big sites that host and deliver lots of images, there are challenges in building an architecture that is cost-effective, highly available, and has low latency (fast retrieval).

Imagine a system where users are able to upload their images to a central server, and the images can be requested via a web link or API, just like Flickr or Picasa. For the sake of simplicity, let's assume that this application has two key parts: the ability to upload (write) an image to the server, and the ability to query for an image. While we certainly want the upload to be efficient, we care most about having very fast delivery when someone requests an image (for example, images could be requested for a web page or other application). This is very similar functionality to what a web server or Content Delivery Network (CDN) edge server (a server CDN uses to store content in many locations so content is geographically/physically closer to users, resulting in faster performance) might provide.

Other important aspects of the system are:

- There is no limit to the number of images that will be stored, so storage scalability, in terms of image count needs to be considered.
- There needs to be low latency for image downloads/requests.
- If a user uploads an image, the image should always be there (data reliability for images).
- The system should be easy to maintain (manageability).
- Since image hosting doesn't have high profit margins, the system needs to be cost-effective

Figure 1.1 is a simplified diagram of the functionality.

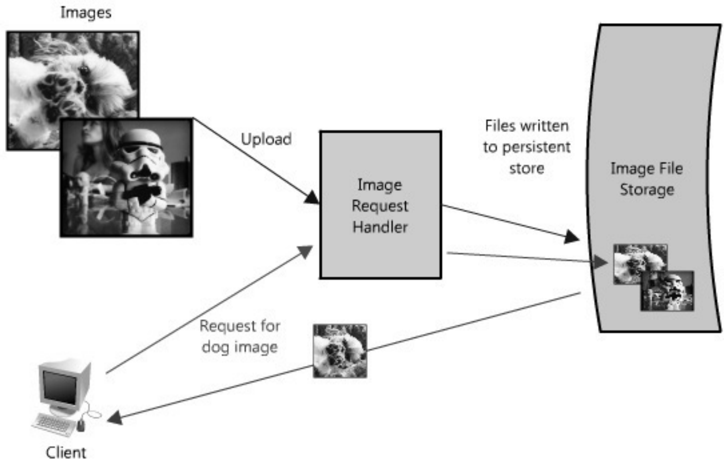


Figure 1.1: Simplified architecture diagram for image hosting application

In this image hosting example, the system must be perceivably fast, its data stored reliably and all of these attributes highly scalable. Building a small version of this application would be trivial and easily hosted on a single server; however, that would not be interesting for this chapter. Let's assume that we want to build something that could grow as big as Flickr.

Services

When considering scalable system design, it helps to decouple functionality and think about each part of the system as its own service with a clearly defined interface. In practice, systems designed in this way are said to have a Service-Oriented Architecture (SOA). For these types of systems, each service has its own distinct functional context, and interaction with anything outside of that context takes place through an abstract interface, typically the public-facing API of another service.

Deconstructing a system into a set of complementary services decouples the operation of those pieces from one another. This abstraction helps establish clear relationships between the service, its underlying environment, and the consumers of that service. Creating these clear delineations can help isolate problems, but also allows each piece to scale independently of one another. This sort of service-oriented design for systems is very similar to object-oriented design for programming.

In our example, all requests to upload and retrieve images are processed by the same server; however, as the system needs to scale it makes sense to break out these two functions into their own services.

Fast-forward and assume that the service is in heavy use; such a scenario makes it easy to see how longer writes will impact the time it takes to read the images (since they two functions will be competing for shared resources). Depending on the architecture this effect can be substantial. Even if the upload and download speeds are the same (which is not true of most IP networks, since most are designed for at least a 3:1 download-speed:upload-speed ratio), read files will typically be read from cache, and writes will have to go to disk eventually (and perhaps be written several times in eventually consistent situations). Even if everything is in memory or read from disks (like SSDs), database writes will almost always be slower than reads. (Pole Position, an open source tool for DB benchmarking, http://polepos.org/ and results http://polepos.sourceforge.net/results/PolePositionClientServer.pdf.).

Another potential problem with this design is that a web server like Apache or lighttpd typically has an upper limit on the number of simultaneous connections it can maintain (defaults are around 500, but can go much higher) and in high traffic, writes can quickly consume all of those. Since reads can be asynchronous, or take advantage of other performance optimizations like gzip compression or chunked transfer encoding, the web server can switch serve reads faster and switch between clients quickly serving many more requests per second than the max number of connections (with Apache and max connections set to 500, it is not uncommon to serve several thousand read requests per second). Writes, on the other hand, tend to maintain an open connection for the duration for the upload, so uploading a 1MB file could take more than 1 second on most home networks, so that web server could only handle 500 such simultaneous writes.

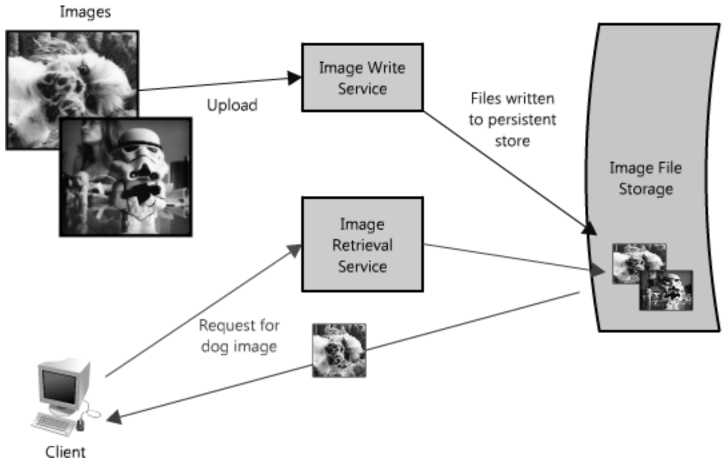


Figure 1.2: Splitting out reads and writes

Planning for this sort of bottleneck makes a good case to split out reads and writes of images into their own services, shown in Figure 1.2. This allows us to scale each of them independently (since it is likely we will always do more reading than writing), but also helps clarify what is going on at each point. Finally, this separates future concerns, which would make it easier to troubleshoot and scale a problem like slow reads.

The advantage of this approach is that we are able to solve problems independently of one another—we don't have to worry about writing and retrieving new images in the same context. Both of these services still leverage the global corpus of images, but they are free to optimize their own performance with service-appropriate methods (for example, queuing up requests, or caching popular images—more on this below). And from a maintenance and cost perspective each service can scale independently as needed, which is great because if they were combined and intermingled, one could inadvertently impact the performance of the other as in the scenario discussed above.

Of course, the above example can work well when you have two different endpoints (in fact this is very similar to several cloud storage providers' implementations and Content Delivery Networks). There are lots of ways to address these types of bottlenecks though, and each has different tradeoffs.

For example, Flickr solves this read/write issue by distributing users across different shards such that each shard can only handle a set number of users, and as users increase more shards are added to the cluster (see the presentation on Flickr's scaling, http://mysqldba.blogspot.com/2008/04/mysql-uc-2007-presentation-file.html). In the first example it is easier to scale hardware based on actual usage (the number of reads and writes across the whole system), whereas Flickr scales with their user base (but forces the assumption of equal usage across users so there can be extra capacity). In the former an outage or issue with one of the services brings down functionality across the whole system (no-one can write files, for example), whereas an outage with one of Flickr's shards will only affect those users. In the first example it is easier to perform operations across the whole dataset—for example, updating the write service to include new metadata or searching across all image metadata—whereas with the Flickr architecture each shard would need to be updated or searched (or a search service would need to be created to collate that metadata—which is in fact what they do).

When it comes to these systems there is no right answer, but it helps to go back to the principles at the start of this chapter, determine the system needs (heavy reads or writes or both, level of concurrency, queries across the data set, ranges, sorts, etc.), benchmark different alternatives, understand how the system will fail, and have a solid plan for when failure happens.

Redundancy

In order to handle failure gracefully a web architecture must have redundancy of its services and data. For example, if there is only one copy of a file stored on a single server, then losing that server means losing that file. Losing data is seldom a good thing, and a common way of handling it is to create multiple, or redundant, copies.

This same principle also applies to services. If there is a core piece of functionality for an application, ensuring that multiple copies or versions are running simultaneously can secure against the failure of a single node.

Creating redundancy in a system can remove single points of failure and provide a backup or spare functionality if needed in a crisis. For example, if there are two instances of the same service running in production, and one fails or degrades, the system can *failover* to the healthy copy. Failover can happen automatically or require manual intervention.

Another key part of service redundancy is creating a *shared-nothing architecture*. With this architecture, each node is able to operate independently of one another and there is no central "brain" managing state or coordinating activities for the other nodes. This helps a lot with scalability since new nodes can be added without special conditions or knowledge. However, and most importantly, there is no single point of failure in these systems, so they are much more resilient to failure.

For example, in our image server application, all images would have redundant copies on another piece of hardware somewhere (ideally in a different geographic location in the event of a catastrophe like an earthquake or fire in the data center), and the services to access the images would be redundant, all potentially servicing requests. (See Figure 1.3.) (Load balancers are a great way to make this possible, but there is more on that below).

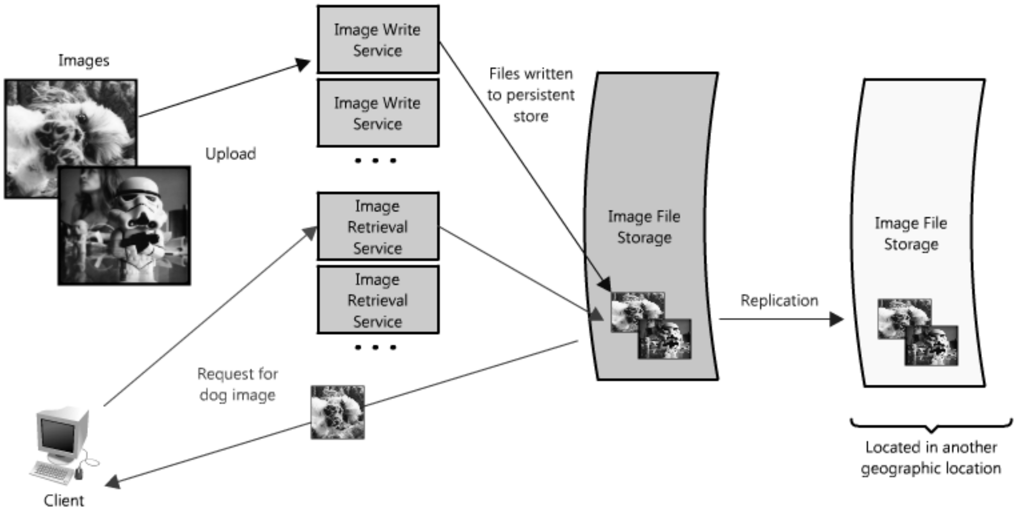


Figure 1.3: Image hosting application with redundancy

Partitions

There may be very large data sets that are unable to fit on a single server. It may also be the case that an operation requires too many computing resources, diminishing performance and making it necessary to add capacity. In either case you have two choices: scale vertically or horizontally.

Scaling vertically means adding more resources to an individual server. So for a very large data set, this might mean adding more (or bigger) hard drives so a single server can contain the entire data set. In the case of the compute operation, this could mean moving the computation to a bigger server with a faster CPU or more memory. In each case, vertical scaling is accomplished by making the individual resource capable of handling more on its own.

To scale horizontally, on the other hand, is to add more nodes. In the case of the large data set, this might be a second server to store parts of the data set, and for the computing resource it would mean splitting the operation or load across some additional nodes. To take full advantage of horizontal scaling, it should be included as an intrinsic design principle of the system architecture, otherwise it can be quite cumbersome to modify and separate out the context to make this possible.

When it comes to horizontal scaling, one of the more common techniques is to break up your services into partitions, or shards. The partitions can be distributed such that each logical set of functionality is separate; this could be done by geographic boundaries, or by another criteria like non-paying versus paying users. The advantage of these schemes is that they provide a service or data store with added capacity.

In our image server example, it is possible that the single file server used to store images could be replaced by multiple file servers, each containing its own unique set of images. (See Figure 1.4.) Such an architecture would allow the system to fill each file server with images, adding additional servers as the disks become full. The design would require a naming scheme that tied an image's filename to the server containing it. An image's name could be formed from a consistent hashing scheme mapped across the servers. Or alternatively, each image could be assigned an incremental ID, so that when a client makes a request for an image, the image retrieval service only needs to maintain the range of IDs that are mapped to each of the servers (like an index).

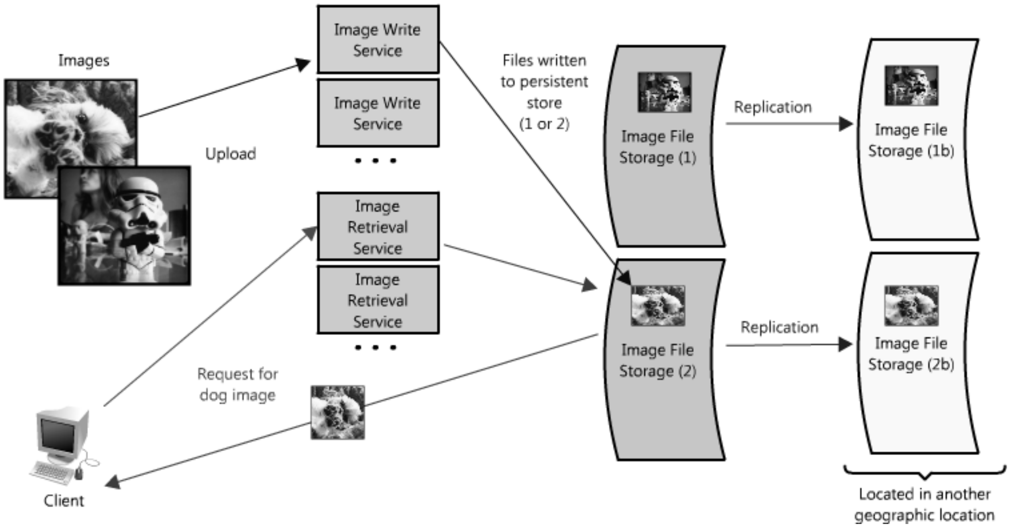

Images
Upload
Image Write Service
Image Write Service
Image Retrieval Service
Image Retrieval Service
Files written to persistent store (1 or 2)
Image File Storage (1)
Image File Storage (2)
Replication
Replication
Image File Storage (1b)
Image File Storage (2b)
Located in another geographic location
Request for dog image
Client

Figure 1.4: Image hosting application with redundancy and partitioning

Of course there are challenges distributing data or functionality across multiple servers. One of the key issues is *data locality*; in distributed systems the closer the data to the operation or point of computation, the better the performance of the system. Therefore it is potentially problematic to have data spread across multiple servers, as any time it is needed it may not be local, forcing the servers to perform a costly fetch of the required information across the network.

Another potential issue comes in the form of *inconsistency*. When there are different services reading and writing from a shared resource, potentially another service or data store, there is the chance for race conditions—where some data is supposed to be updated, but the read happens prior to the update—and in those cases the data is inconsistent. For example, in the image hosting scenario, a race condition could occur if one client sent a request to update the dog image with a new title, changing it from "Dog" to "Gizmo", but at the same time another client was reading the image. In that circumstance it is unclear which title, "Dog" or "Gizmo", would be the one received by the second client.

There are certainly some obstacles associated with partitioning data, but partitioning allows each problem to be split—by data, load, usage patterns, etc.—into manageable chunks. This can help with scalability and manageability, but is not without risk. There are lots of ways to mitigate risk and handle failures; however, in the interest of brevity they are not covered in this chapter. If you are interested in reading more, you can check out my blog post on fault tolerance and monitoring.

1.3. The Building Blocks of Fast and Scalable Data Access

Having covered some of the core considerations in designing distributed systems, let's now talk about the hard part: scaling access to the data.

Most simple web applications, for example, LAMP stack applications, look something like Figure 1.5.

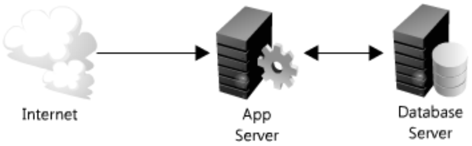

Internet
App Server
Database Server

Figure 1.5: Simple web applications

As they grow, there are two main challenges: scaling access to the app server and to the database. In a highly scalable application design, the app (or web) server is typically minimized and often embodies a shared-nothing architecture. This makes the app server layer of the system horizontally scalable. As a result of this design, the heavy lifting is pushed down the stack to the database server and supporting services; it's at this layer where the real scaling and performance challenges come into play.

The rest of this chapter is devoted to some of the more common strategies and methods for making these types of services fast and scalable by providing fast access to data.

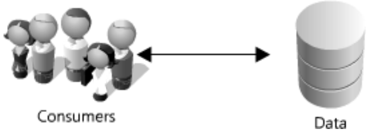

Consumers
Data

Figure 1.6: Oversimplified web application

Most systems can be oversimplified to Figure 1.6. This is a great place to start. If you have a lot of data, you want fast and easy access, like keeping a stash of candy in the top drawer of your desk. Though overly simplified, the previous statement hints at two hard problems: scalability of storage and fast access of data.

For the sake of this section, let's assume you have many terabytes (TB) of data and you want to allow users to access small portions of that data at random. (See Figure 1.7.) This is similar to locating an image file somewhere on the file server in the image application example.

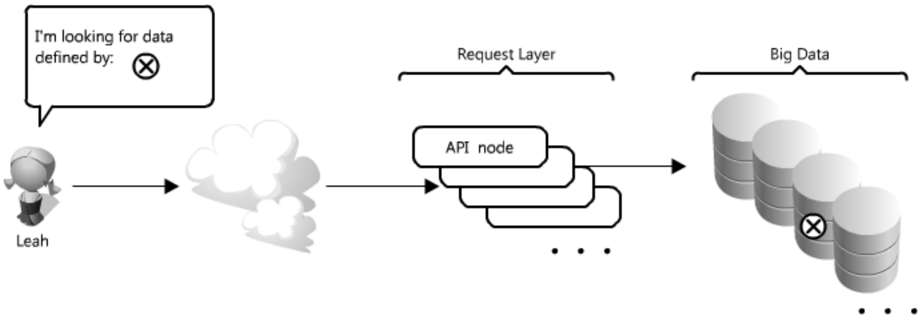

I'm looking for data defined by: ⊗
Leah
Request Layer
API node
Big Data
⊗

Figure 1.7: Accessing specific data

This is particularly challenging because it can be very costly to load TBs of data into memory; this directly translates to disk IO. Reading from disk is many times slower than from memory—memory access is as fast as Chuck Norris, whereas disk access is slower than the line at the DMV. This speed difference really adds up for large data sets; in real numbers memory access is as little as 6 times faster for sequential reads, or 100,000 times faster for random reads, than reading from disk (see "The Pathologies of Big Data", http://queue.acm.org/detail.cfm?id=1563874). Moreover, even with unique IDs, solving the problem of knowing where to find that little bit of data can be an arduous task. It's like trying to get that last Jolly Rancher from your candy stash without looking.

Thankfully there are many options that you can employ to make this easier; four of the more important ones are caches, proxies, indexes and load balancers. The rest of this section discusses how each of these concepts can be used to make data access a lot faster.

Caches

Caches take advantage of the locality of reference principle: recently requested data is likely to be requested again. They are used in almost every layer of computing: hardware, operating systems, web browsers, web applications and more. A cache is like short-term memory: it has a limited amount of space, but is typically faster than the original data source and contains the most recently accessed items. Caches can exist at all levels in architecture, but are often found at the level nearest to the front end, where they are implemented to return data quickly without taxing downstream levels.

How can a cache be used to make your data access faster in our API example? In this case, there are a couple of places you can insert a cache. One option is to insert a cache on your request layer node, as in Figure 1.8.

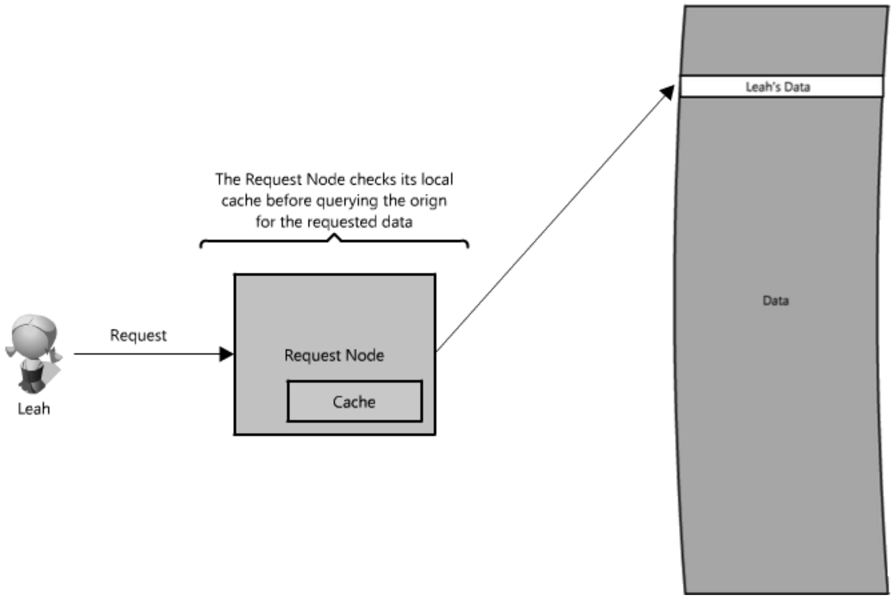

Leah's Data
The Request Node checks its local cache before querying the orign for the requested data
Request
Request Node
Cache
Leah
Data

Figure 1.8: Inserting a cache on your request layer node

Placing a cache directly on a request layer node enables the local storage of response data. Each time a request is made to the service, the node will quickly return local, cached data if it exists. If it is not in the cache, the request node will query the data from disk. The cache on one request layer node could also be located both in memory (which is very fast) and on the node's local disk (faster than going to network storage).

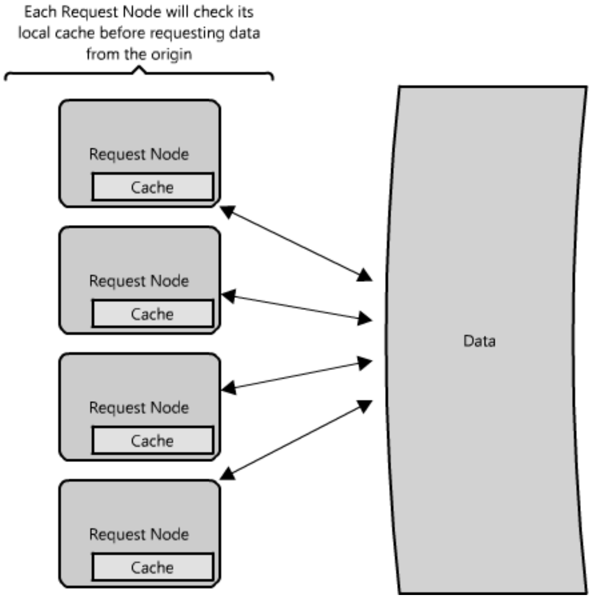

Each Request Node will check its local cache before requesting data from the origin
Request Node
Cache
Request Node
Cache
Request Node
Cache
Request Node
Cache
Data

Figure 1.9: Multiple caches

What happens when you expand this to many nodes? As you can see in Figure 1.9, if the request layer is expanded to multiple nodes, it's still quite possible to have each node host its own cache. However, if your load balancer randomly distributes requests across the nodes, the same request will go to different nodes, thus increasing cache misses. Two choices for overcoming this hurdle are global caches and distributed caches.

Global Cache

A global cache is just as it sounds: all the nodes use the same single cache space. This involves adding a server, or file store of some sort, faster than your original store and accessible by all the request layer nodes. Each of the request nodes queries the cache in the same way it would a local one. This kind of caching scheme can get a bit complicated because it is very easy to overwhelm a single cache as the number of clients and requests increase, but is very effective in some architectures (particularly ones with specialized hardware that make this global cache very fast, or that have a fixed dataset that needs to be cached).

There are two common forms of global caches depicted in the diagrams. In Figure 1.10, when a cached response is not found in the cache, the cache itself becomes responsible for retrieving the missing piece of data from the underlying store. In Figure 1.11 it is the responsibility of request nodes to retrieve any data that is not found in the cache.

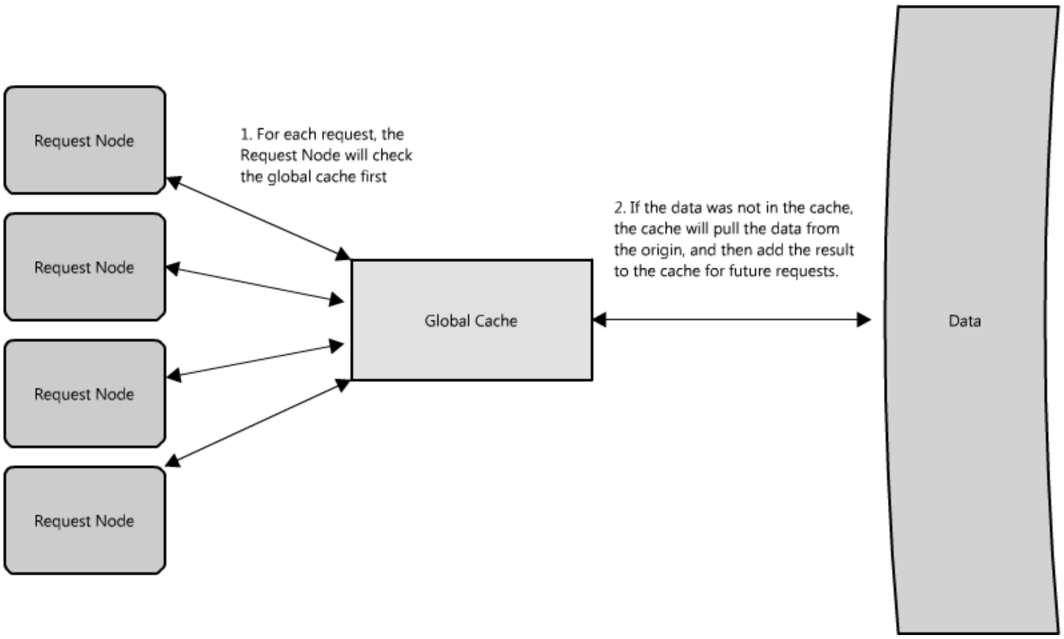

Request Node
Request Node
Request Node
Request Node
1. For each request, the Request Node will check the global cache first
2. If the data was not in the cache, the cache will pull the data from the origin, and then add the result to the cache for future requests.
Global Cache
Data

Figure 1.10: Global cache where cache is responsible for retrieval

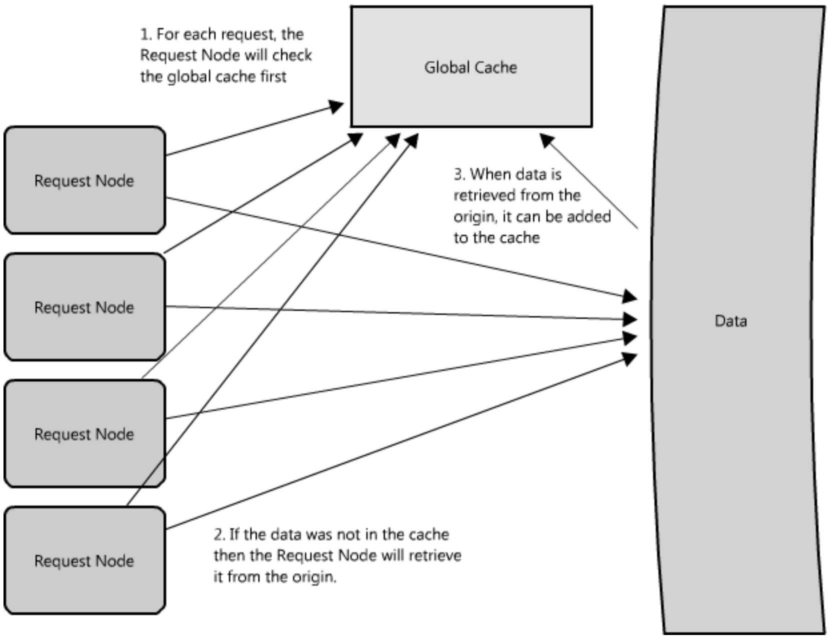

1. For each request, the Request Node will check the global cache first
Global Cache
3. When data is retrieved from the origin, it can be added to the cache
Request Node
Request Node
Request Node
Request Node
Data
2. If the data was not in the cache then the Request Node will retrieve it from the origin.

Figure 1.11: Global cache where request nodes are responsible for retrieval

The majority of applications leveraging global caches tend to use the first type, where the cache itself manages eviction and fetching data to prevent a flood of requests for the same data from the clients. However, there are some cases where the second implementation makes more sense. For example, if the cache is being used for very large files, a low cache hit percentage would cause the cache buffer to become overwhelmed with cache misses; in this situation it helps to have a large percentage of the total data set (or hot data set) in the cache. Another example is an architecture where the files stored in the cache are static and shouldn't be evicted. (This could be because of application requirements around that data latency—certain pieces of data might need to be very fast for large data sets—where the application logic understands the eviction strategy or hot spots better than the cache.)

Distributed Cache

In a distributed cache (Figure 1.12), each of its nodes own part of the cached data, so if a refrigerator acts as a cache to the grocery store, a distributed cache is like putting your food in several locations—your fridge, cupboards, *and* lunch box—convenient locations for retrieving snacks from, without a trip to the store. Typically the cache is divided up using a consistent hashing function, such that if a request node is looking for a certain piece of data it can quickly know where to look within the distributed cache to determine if that data is available. In this case, each node has a small piece of the cache, and will then send a request to another node for the data before going to the origin. Therefore, one of the advantages of a distributed cache is the increased cache space that can be had just by adding nodes to the request pool.

A disadvantage of distributed caching is remedying a missing node. Some distributed caches get around this by storing multiple copies of the data on different nodes; however, you can imagine how this logic can get complicated quickly, especially when you add or remove nodes from the request layer. Although even if a node disappears and part of the cache is lost, the requests will just pull from the origin—so it isn't necessarily catastrophic!

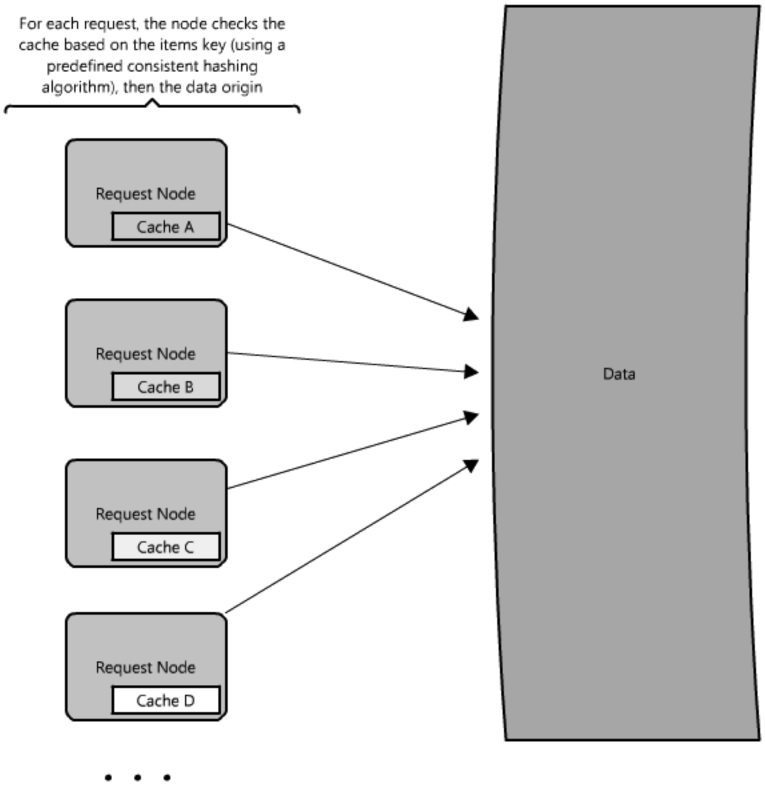

For each request, the node checks the cache based on the items key (using a predefined consistent hashing algorithm), then the data origin
Request Node
Cache A
Request Node
Cache B
Request Node
Cache C
Request Node
Cache D
Data

Figure 1.12: Distributed cache

The great thing about caches is that they usually make things much faster (implemented correctly, of course!) The methodology you choose just allows you to make it faster for even more requests. However, all this caching comes at the cost of having to maintain additional storage space, typically in the form of expensive memory; nothing is free. Caches are wonderful for making things generally faster, and moreover provide system functionality under high load conditions when otherwise there would be complete service degradation.

One example of a popular open source cache is Memcached (http://memcached.org/) (which can work both as a local cache and distributed cache); however, there are many other options (including many language- or framework-specific options).

Memcached is used in many large web sites, and even though it can be very powerful, it is simply an in-memory key value store, optimized for arbitrary data storage and fast lookups (*O(1)*).

Facebook uses several different types of caching to obtain their site performance (see "Facebook caching and performance"). They use `$GLOBALS` and APC caching at the language level (provided in PHP at the cost of a function call) which helps make intermediate function calls and results much faster. (Most languages have these types of libraries to improve web page performance and they should almost always be used.) Facebook then use a global cache that is distributed across many servers (see "Scaling memcached at Facebook"), such that one function call accessing the cache could make many requests in parallel for data stored on different Memcached servers. This allows them to get much higher performance and throughput for their user profile data, and have one central place to update data (which is important, since cache invalidation and maintaining consistency can be challenging when you are running thousands of servers).

Now let's talk about what to do when the data isn't in the cache…

Proxies

At a basic level, a proxy server is an intermediate piece of hardware/software that receives requests from clients and relays them to the backend origin servers. Typically, proxies are used to filter requests, log requests, or sometimes transform requests (by adding/removing headers, encrypting/decrypting, or compression).

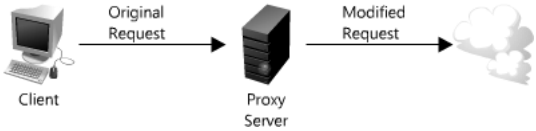


Figure 1.13: Proxy server

Proxies are also immensely helpful when coordinating requests from multiple servers, providing opportunities to optimize request traffic from a system-wide perspective. One way to use a proxy to speed up data access is to collapse the same (or similar) requests together into one request, and then return the single result to the requesting clients. This is known as collapsed forwarding.

Imagine there is a request for the same data (let's call it littleB) across several nodes, and that piece of data is not in the cache. If that request is routed thought the proxy, then all of those requests can be collapsed into one, which means we only have to read littleB off disk once. (See Figure 1.14.) There is some cost associated with this design, since each request can have slightly higher latency, and some requests may be slightly delayed to be grouped with similar ones. But it will improve performance in high load situations, particularly when that same data is requested over and over. This is similar to a cache, but instead of storing the data/document like a cache, it is optimizing the requests or calls for those documents and acting as a proxy for those clients.

In a LAN proxy, for example, the clients do not need their own IPs to connect to the Internet, and the LAN will collapse calls from the clients for the same content. It is easy to get confused here though, since many proxies are also caches (as it is a very logical place to put a cache), but not all caches act as proxies.

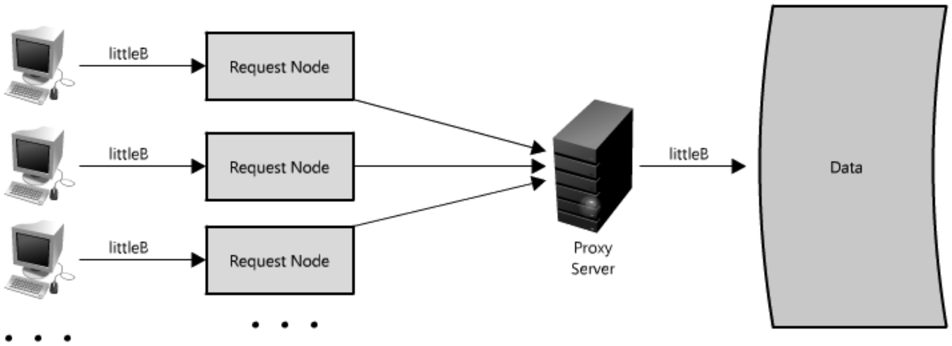


Figure 1.14: Using a proxy server to collapse requests

Another great way to use the proxy is to not just collapse requests for the same data, but also to collapse requests for data that is spatially close together in the origin store (consecutively on disk). Employing such a strategy maximizes data locality for the requests, which can result in decreased request latency. For example, let's say a bunch of nodes request parts of B: partB1, partB2, etc. We can set up our proxy to recognize the spatial locality of the individual requests, collapsing them into a single request and returning only bigB, greatly minimizing the reads from the data origin. (See Figure 1.15.) This can make a really big difference in request time when you are randomly accessing across TBs of data! Proxies are especially helpful under high load situations, or when you have limited caching, since they can essentially batch several requests into one.

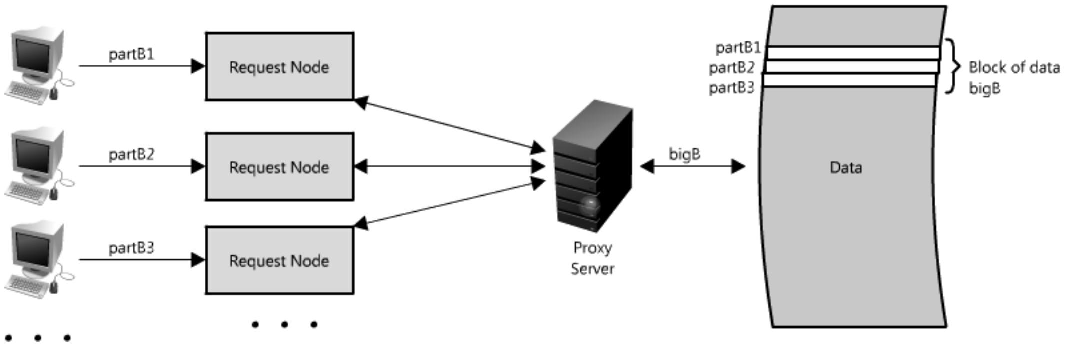


Figure 1.15: Using a proxy to collapse requests for data that is spatially close together

It is worth noting that you can use proxies and caches together, but generally it is best to put the cache in front of the proxy, for the same reason that it is best to let the faster runners start first in a crowded marathon race. This is because the cache is serving data from memory, it is very fast, and it doesn't mind multiple requests for the same result. But if the cache was located on the other side of the proxy server, then there would be additional latency with every request before the cache, and this could hinder performance.

If you are looking at adding a proxy to your systems, there are many options to consider; Squid and Varnish have both been road tested and are widely used in many production web sites. These proxy solutions offer many optimizations to make the most of client-server communication. Installing one of these as a reverse proxy (explained in the load balancer section below) at the web server layer can improve web server performance considerably, reducing the amount of work required to handle incoming client requests.

Indexes

Using an index to access your data quickly is a well-known strategy for optimizing data access performance; probably the most well known when it comes to databases. An index makes the trade-offs of increased storage overhead and slower writes (since you must both write the data and update the index) for the benefit of faster reads.

Just as to a traditional relational data store, you can also apply this concept to larger data sets. The trick with indexes is you must carefully consider how users will access your data. In the case of data sets that are many TBs in size, but with very small payloads (e.g., 1 KB), indexes are a necessity for optimizing data access. Finding a small payload in such a large data set can be a real challenge since you can't possibly iterate over that much data in any reasonable time. Furthermore, it is very likely that such a large data set is spread over several (or many!) physical devices—this means you need some way to find the correct physical location of the desired data. Indexes are the best way to do this.

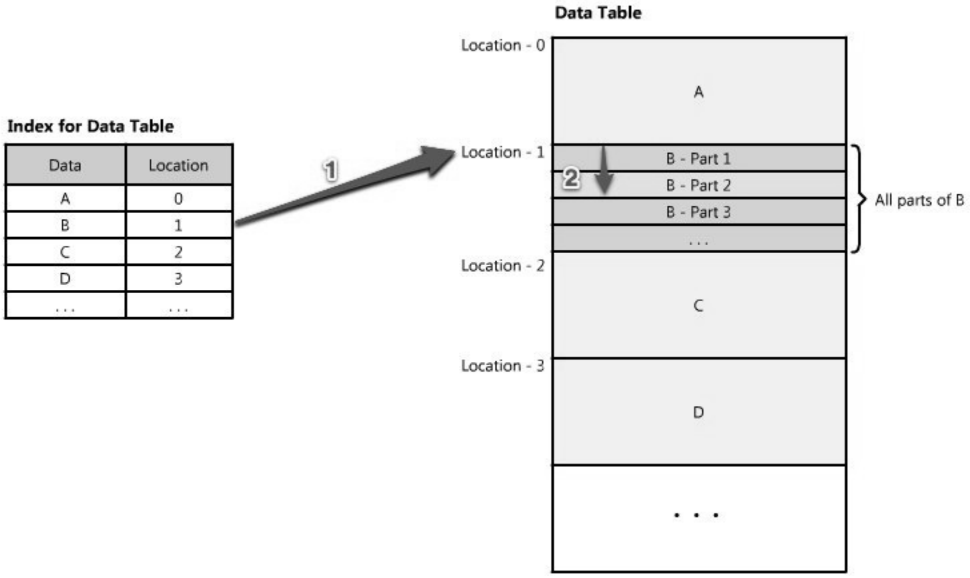


Figure 1.16: Indexes

An index can be used like a table of contents that directs you to the location where your data lives. For example, let's say you are looking for a piece of data, part 2 of B—how will you know where to find it? If you have an index that is sorted by data type—say data A, B, C—it would tell you the location of data B at the origin. Then you just have to seek to that location and read the part of B you want. (See Figure 1.16.)

These indexes are often stored in memory, or somewhere very local to the incoming client request. Berkeley DBs (BDBs) and tree-like data structures are commonly used to store data in ordered lists, ideal for access with an index.

Often there are many layers of indexes that serve as a map, moving you from one location to the next, and so forth, until you get the specific piece of data you want. (See Figure 1.17.)

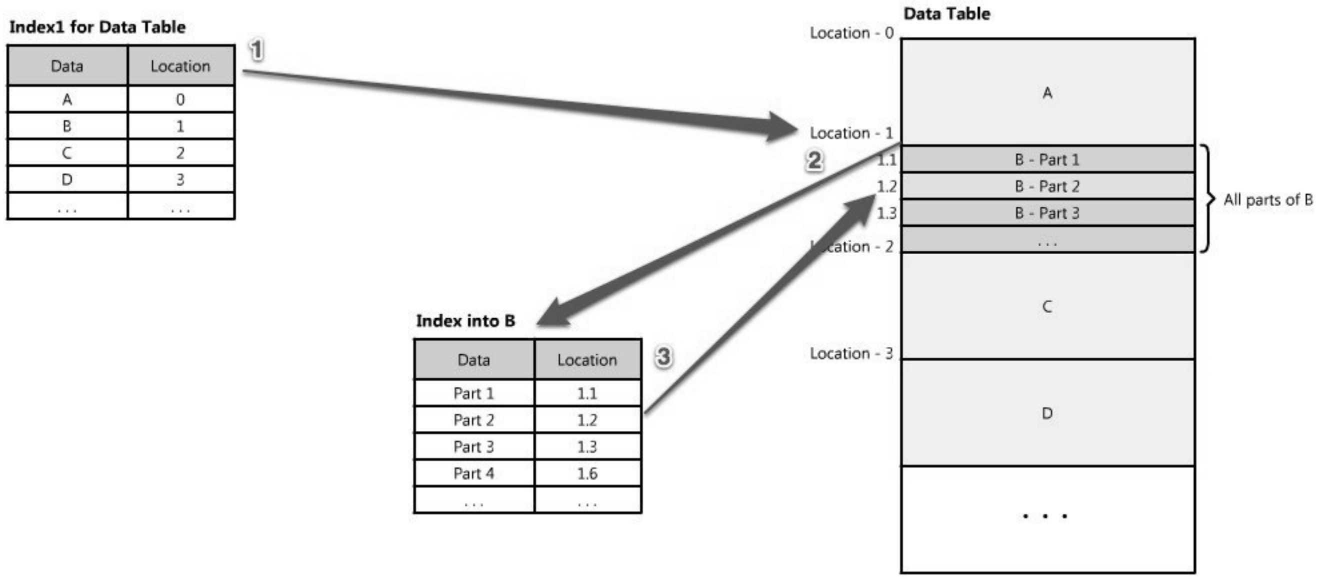


Figure 1.17: Many layers of indexes

Indexes can also be used to create several different views of the same data. For large data sets, this is a great way to define different filters and sorts without resorting to creating many additional copies of the data.

For example, imagine that the image hosting system from earlier is actually hosting images of book pages, and the service allows client queries across the text in those images, searching all the book content about a topic, in the same way search engines allow you to search HTML content. In this case, all those book images take many, many servers to store the files, and finding one page to render to the user can be a bit involved. First, inverse indexes to query for arbitrary words and word tuples need to be easily accessible; then there is the challenge of navigating to the exact page and location within that book, and retrieving the right image for the results. So in this case the inverted index would map to a location (such as book B), and then B may contain an index with all the words, locations and number of occurrences in each part.

An inverted index, which could represent Index1 in the diagram above, might look something like the following—each word or tuple of words provide an index of what books contain them.

Word(s)	Book(s)
being awesome	Book B, Book C, Book D
always	Book C, Book F
believe	Book B

The intermediate index would look similar but would contain just the words, location, and information for book B. This nested index architecture allows each of these indexes to take up less space than if all of that info had to be stored into one big inverted index.

And this is key in large-scale systems because even compressed, these indexes can get quite big and expensive to store. In this system if we assume we have a lot of the books in the world—100,000,000 (see Inside Google Books blog post)—and that each book is only 10 pages long (to make the math easier), with 250 words per page, that means there are 250 billion words. If we assume an average of 5 characters per word, and each character takes 8 bits (or 1 byte, even though some characters are 2 bytes), so 5 bytes per word, then an index containing only each word once is over a terabyte of storage. So you can see creating indexes that have a lot of other information like tuples of words, locations for the data, and counts of occurrences, can add up very quickly.

Creating these intermediate indexes and representing the data in smaller sections makes big data problems tractable. Data can be spread across many servers and still accessed quickly. Indexes are a cornerstone of information retrieval, and the basis for today's modern search engines. Of course, this section only scratched the surface, and there is a lot of research being done on how to make indexes smaller, faster, contain more information (like relevancy), and update seamlessly. (There are some manageability challenges with race conditions, and with the sheer number of updates required to add new data or change existing data, particularly in the event where relevancy or scoring is involved).

Being able to find your data quickly and easily is important; indexes are an effective and simple tool to achieve this.

Load Balancers

Finally, another critical piece of any distributed system is a load balancer. Load balancers are a principal part of any architecture, as their role is to distribute load across a set of nodes responsible for servicing requests. This allows multiple nodes to transparently service the same function in a system. (See Figure 1.18.) Their main purpose is to handle a lot of simultaneous connections and route those connections to one of the request nodes, allowing the system to scale to service more requests by just adding nodes.

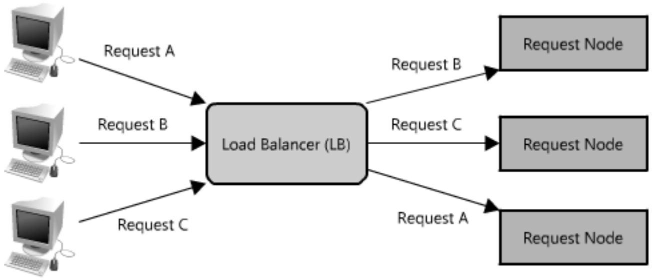


Figure 1.18: Load balancer

There are many different algorithms that can be used to service requests, including picking a random node, round robin, or even selecting the node based on certain criteria, such as memory or CPU utilization. Load balancers can be implemented as software or hardware appliances. One open source software load balancer that has received wide adoption is HAProxy).

In a distributed system, load balancers are often found at the very front of the system, such that all incoming requests are routed accordingly. In a complex distributed system, it is not uncommon for a request to be routed to multiple load balancers as shown in Figure 1.19.

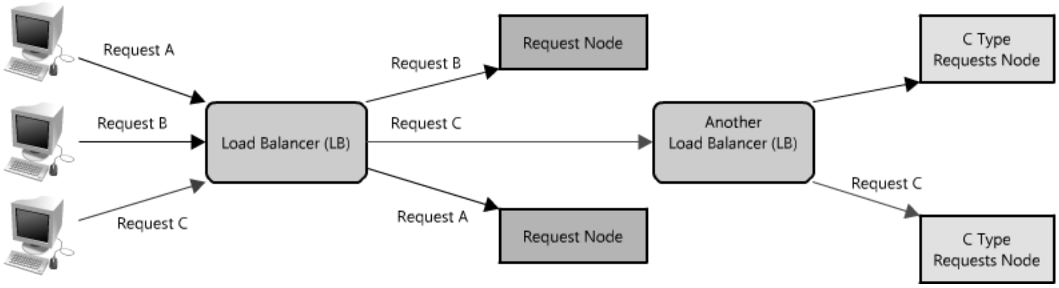


Figure 1.19: Multiple load balancers

Like proxies, some load balancers can also route a request differently depending on the type of request it is. (Technically these are also known as reverse proxies.)

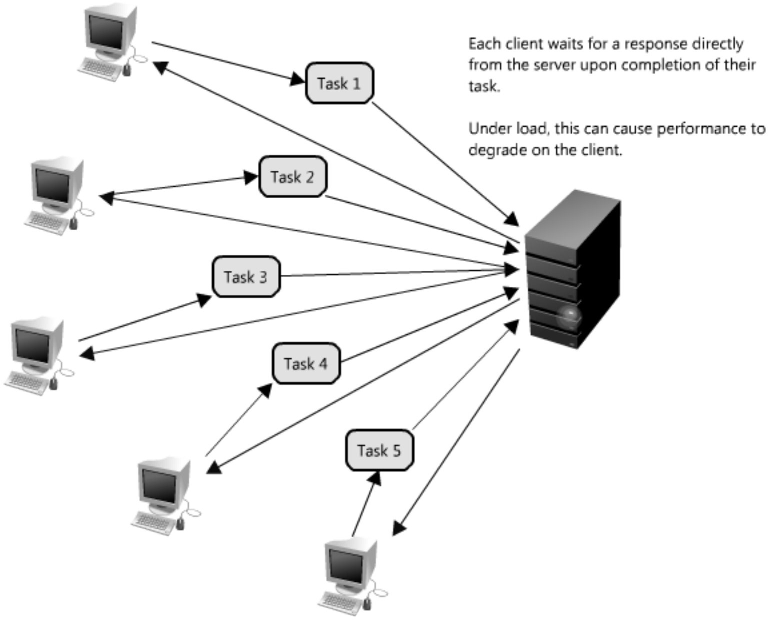
One of the challenges with load balancers is managing user-session-specific data. In an e-commerce site, when you only have one client it is very easy to allow users to put things in their shopping cart and persist those contents between visits (which is important, because it is much more likely you will sell the product if it is still in the user's cart when they return). However, if a user is routed to one node for a session, and then a different node on their next visit, there can be inconsistencies since the new node may be missing that user's cart contents. (Wouldn't you be upset if you put a 6 pack of Mountain Dew in your cart and then came back and it was empty?) One way around this can be to make sessions sticky so that the user is always routed to the same node, but then it is very hard to take advantage of some reliability features like automatic failover. In this case, the user's shopping cart would always have the contents, but if their sticky node became unavailable there would need to be a special case and the assumption of the contents being there would no longer be valid (although hopefully this assumption wouldn't be built into the application). Of course, this problem can be solved using other strategies and tools in this chapter, like services, and many not covered (like browser caches, cookies, and URL rewriting).

If a system only has a couple of a nodes, systems like round robin DNS may make more sense since load balancers can be expensive and add an unneeded layer of complexity. Of course in larger systems there are all sorts of different scheduling and load-balancing algorithms, including simple ones like random choice or round robin, and more sophisticated mechanisms that take things like utilization and capacity into consideration. All of these algorithms allow traffic and requests to be distributed, and can provide helpful reliability tools like automatic failover, or automatic removal of a bad node (such as when it becomes unresponsive). However, these advanced features can make problem diagnosis cumbersome. For example, when it comes to high load situations, load balancers will remove nodes that may be slow or timing out (because of too many requests), but that only exacerbates the situation for the other nodes. In these cases extensive monitoring is important, because overall system traffic and throughput may look like it is decreasing (since the nodes are serving less requests) but the individual nodes are becoming maxed out.

Load balancers are an easy way to allow you to expand system capacity, and like the other techniques in this article, play an essential role in distributed system architecture. Load balancers also provide the critical function of being able to test the health of a node, such that if a node is unresponsive or over-loaded, it can be removed from the pool handling requests, taking advantage of the redundancy of different nodes in your system.

Queues

So far we have covered a lot of ways to read data quickly, but another important part of scaling the data layer is effective management of writes. When systems are simple, with minimal processing loads and small databases, writes can be predictably fast; however, in more complex systems writes can take an almost non-deterministically long time. For example, data may have to be written several places on different servers or indexes, or the system could just be under high load. In the cases where writes, or any task for that matter, may take a long time, achieving performance and availability requires building asynchrony into the system; a common way to do that is with queues.



Each client waits for a response directly from the server upon completion of their task.

Under load, this can cause performance to degrade on the client.

Figure 1.20: Synchronous request

Imagine a system where each client is requesting a task to be remotely serviced. Each of these clients sends their request to the server, where the server completes the tasks as quickly as possible and returns the results to their respective clients. In small systems where one server (or logical service) can service incoming clients just as fast as they come, this sort of situation should work just fine. However, when the server receives more requests than it can handle, then each client is forced to wait for the other clients' requests to complete before a response can be generated. This is an example of a synchronous request, depicted in Figure 1.20.

This kind of synchronous behavior can severely degrade client performance; the client is forced to wait, effectively performing zero work, until its request can be answered. Adding additional servers to address system load does not solve the problem either; even with effective load balancing in place it is extremely difficult to ensure the even and fair distribution of work required to maximize client performance. Further, if the server handling requests is unavailable, or fails, then the clients upstream will also fail. Solving this problem effectively requires abstraction between the client's request and the actual work performed to service it.

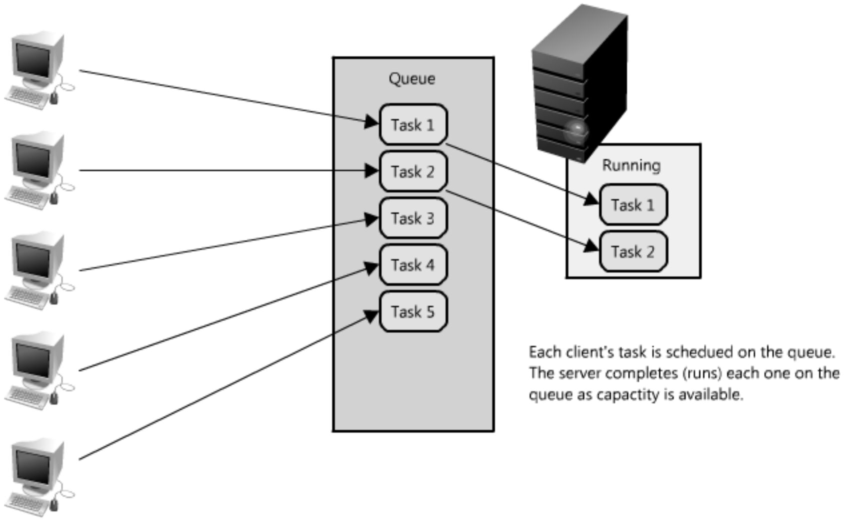

Queue
Task 1
Task 2
Task 3
Task 4
Task 5
Running
Task 1
Task 2
Each client's task is scheduled on the queue. The server completes (runs) each one on the queue as capactity is available.

Figure 1.21: Using queues to manage requests

Enter queues. A queue is as simple as it sounds: a task comes in, is added to the queue and then workers pick up the next task as they have the capacity to process it. (See Figure 1.21.) These tasks could represent simple writes to a database, or something as complex as generating a thumbnail preview image for a document. When a client submits task requests to a queue they are no longer forced to wait for the results; instead they need only acknowledgement that the request was properly received. This acknowledgement can later serve as a reference for the results of the work when the client requires it.

Queues enable clients to work in an asynchronous manner, providing a strategic abstraction of a client's request and its response. On the other hand, in a synchronous system, there is no differentiation between request and reply, and they therefore cannot be managed separately. In an asynchronous system the client requests a task, the service responds with a message acknowledging the task was received, and then the client can periodically check the status of the task, only requesting the result once it has completed. While the client is waiting for an asynchronous request to be completed it is free to perform other work, even making asynchronous requests of other services. The latter is an example of how queues and messages are leveraged in distributed systems.

Queues also provide some protection from service outages and failures. For instance, it is quite easy to create a highly robust queue that can retry service requests that have failed due to transient server failures. It is more preferable to use a queue to enforce quality-of-service guarantees than to expose clients directly to intermittent service outages, requiring complicated and often-inconsistent client-side error handling.

Queues are fundamental in managing distributed communication between different parts of any large-scale distributed system, and there are lots of ways to implement them. There are quite a few open source queues like RabbitMQ, ActiveMQ, BeanstalkD, but some also use services like Zookeeper, or even data stores like Redis.

1.4. Conclusion

Designing efficient systems with fast access to lots of data is exciting, and there are lots of great tools that enable all kinds of new applications. This chapter covered just a few examples, barely scratching the surface, but there are many more—and there will only continue to be more innovation in the space.

Back to top
Back to *The Architecture of Open Source Applications*.

This work is made available under the Creative Commons Attribution 3.0 Unported license. Please see the full description of the license for details.