Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have, as well as making otherwise unattainable product requirements feasible. 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.

### 1. Application server cache

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 requesting 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).

What happens when you expand this to many nodes? 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.

### 2. Distributed cache

In a distributed cache, each of its nodes own part of the cached data. 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 ease by which we can increase the cache space, which can be achieved just by adding nodes to the request pool.

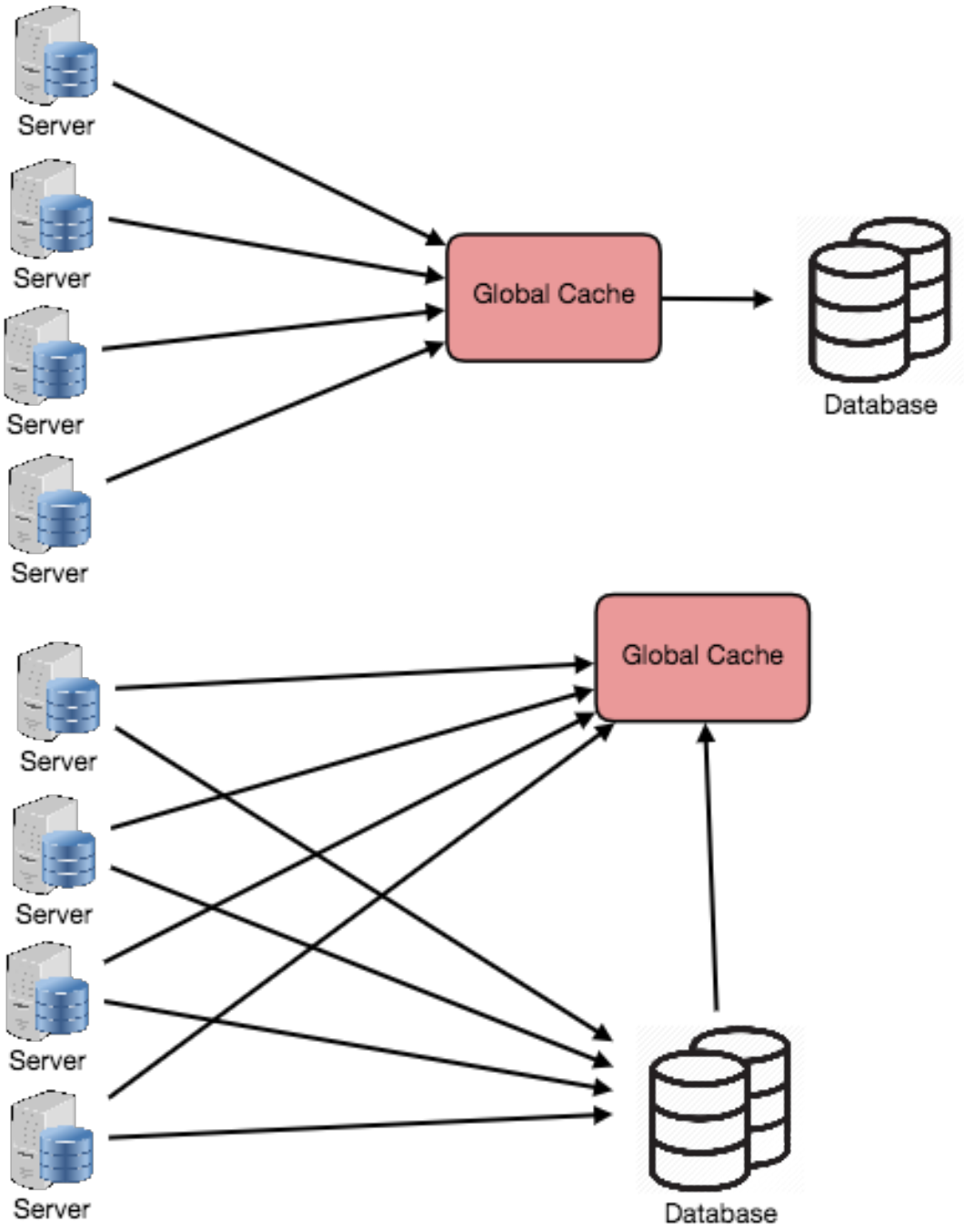
A disadvantage of distributed caching is resolving a missing node. Some distributed caches get around this by storing multiple copies of the data on different nodes; however, you can imagine how this logic can get complicated quickly, especially when you add or remove nodes from the

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!

### 3. 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 following diagram. First, 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. Second, it is the responsibility of request nodes to retrieve any data that is not found in the cache.




Most 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.)

# 4. Content Distribution Network (CDN)

CDNs are a kind of cache that comes into play for sites serving large amounts of static media. In a typical CDN setup, a request will first ask the CDN for a piece of static media; the CDN will serve that content if it has it locally available. If it isn't available, the CDN will query the back-end servers for the file and then cache it locally and serve it to the requesting user.

If the system we are building isn't yet large enough to have its own CDN, we can ease a future transition by serving the static media off a separate subdomain (e.g. static.yourservice.com) using a lightweight HTTP server like Nginx, and cutover the DNS from your servers to a CDN later.

# Cache Invalidation

While caching is fantastic, it does require some maintenance for keeping cache coherent with the source of truth (e.g., database). If the data is modified in the database, it should be invalidated in the cache, if not, this can cause inconsistent application behavior.

Solving this problem is known as cache invalidation, there are three main schemes that are used:

**Write-through cache:** Under this scheme data is written into the cache and the corresponding database at the same time. The cached data allows for fast retrieval, and since the same data gets written in the permanent storage, we will have complete data consistency between cache and storage. Also, this scheme ensures that nothing will get lost in case of a crash, power failure, or other system disruptions.

Although write through minimizes the risk of data loss, since every write operation must be done twice before returning success to the client, this scheme has the disadvantage of higher latency for write operations.

**Write-around cache:** This technique is similar to write through cache, but data is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write operations that will not subsequently be re-read, but has the disadvantage that a read request for recently written data will create a "cache miss" and must be read from slower back-end storage and experience higher latency.

**Write-back cache:** Under this scheme, data is written to cache alone, and completion is immediately confirmed to the client. The write to the permanent storage is done after specified intervals or under certain conditions. This results in low latency and high throughput for write-intensive applications, however, this speed comes with the risk of data loss in case of a crash or other adverse event because the only copy of the written data is in the cache.

# Cache eviction policies

Following are some of the most common cache eviction policies:

1. First In First Out (FIFO): The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.
2. Last In First Out (LIFO): The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
3. Least Recently Used (LRU): Discards the least recently used items first.
4. Most Recently Used (MRU): Discards, in contrast to LRU, the most recently used items first.
5. Least Frequently Used (LFU): Counts how often an item is needed. Those that are used least often are discarded first.
6. Random Replacement (RR): Randomly selects a candidate item and discards it to make space when necessary.