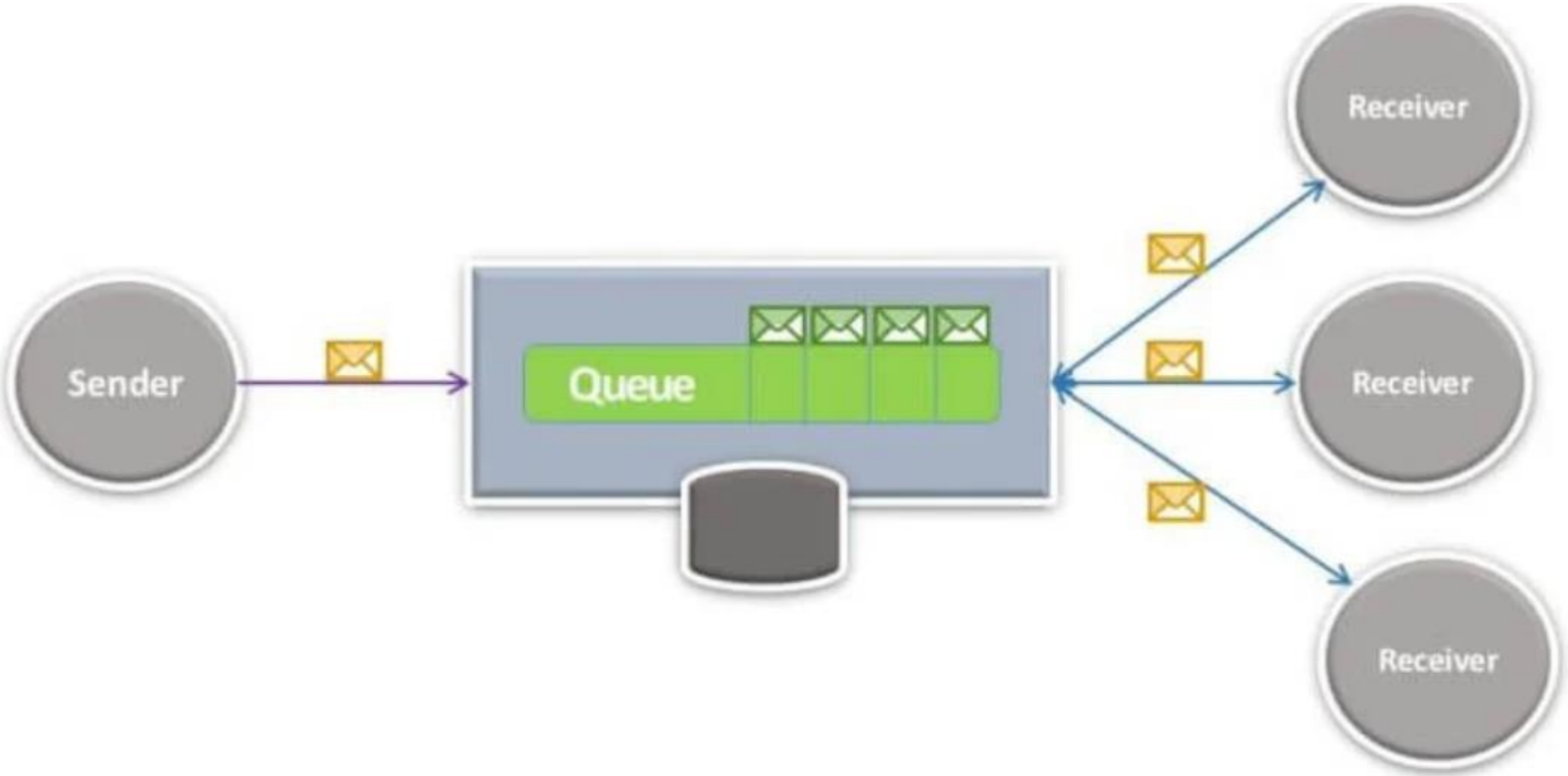
Performance Optimization Techniques

- Message Queue
- Distributed Caching
- Load balancing

Message Queue

- Imagine that you have a web service that receives many requests every second, where no request can get lost, and all requests need to be processed by a function that has a high throughput.
- In other words, the web service always has to be highly available and ready to receive a new request instead of being locked by the processing of previously received requests.
- In this case, placing a queue between the web service and the processing service is ideal.
- The web service can put the "start processing" message on a queue and the other process can take and handle messages in order.
- The two processes are decoupled from each other and do not need to wait.


Receiver
Sender
Queue
Receiver
Receiver

Benefits

- Traffic spike: You don’t always know exactly how much traffic your application is going to have. We have no way to know what our clients are going to send us. By queuing the data we can be assured the data will be persisted and then be processed eventually, even if that means it takes a little longer than usual due to a high traffic spike.
- Batching is a great reason to use message queues.
- Message queues enable you to decouple different parts of your application and then scale them independently.

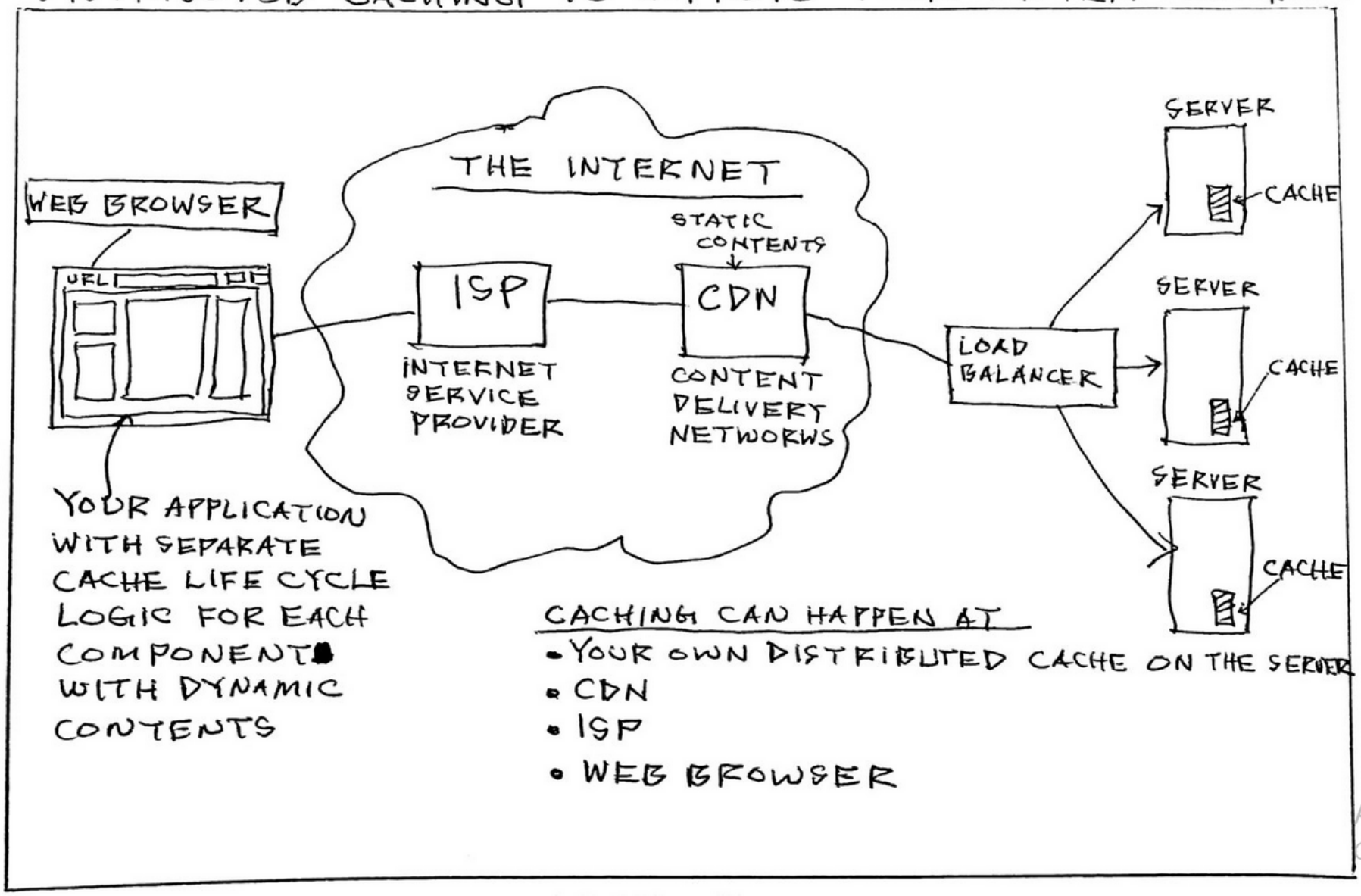
Benefits

- Queues can be useful in web applications to do complex logic in a background thread so the request can finish for the user quickly.
- If someone places an order on your website, that could involve a lot of different things that have to happen. You can do the minimum and return success to your user and kick off the rest of them in a background thread to finish up.
- Queues can be great in scenarios where your application needs something done but doesn't need it done now, or doesn't even care about the result.
- Instead of calling a web service and waiting for it to complete, you can write the message to a queue and let the same business logic happen later.
- Queues help with redundancy by making the process that reads the message confirm that it completed the transaction and it is safe to remove it.

Distributed caching

- Caching is a commonly used technique to improve system performance by storing frequently used data or files in memory or local file system to avoid roundtrip over the network.
- And if we have multiple servers behind a load balancer, we use a distributed caching technology such as Redis.
- We can further expand the context of distributed caching outside the system boundary.
  - For static contents, we can use CDN (Content Delivery Networks).
  - ISPs (Internet Service Provider) also cache contents for its users.
  - On the user-side, web browsers also have the caching capability.

DISTRIBUTED CACHING TO IMPROVE YOUR SYSTEM PERFORMANCE



Distributed Caching

- Saves:
  - Network communication
  - Computation
  - Database access
- When you do an update, you also update the cache entry, which the caching solution propagates in order to make the cache coherent.
- Cache Access Patterns:
  - Write Through
  - Write Around
  - Write Back

Questions regarding caching

- How much do you want your cache entries to live?
- How big do you need your cache to be?
- How should elements expire (cache eviction strategy) – least recently used, least frequently used, first-in-first-out?
- You have to constantly keep an eye on the cache statistics, do performances tests, measure and tweak.
- Data that are updated too often and read not that often, constantly updating the cache with them is an overhead that doesn't pay off.

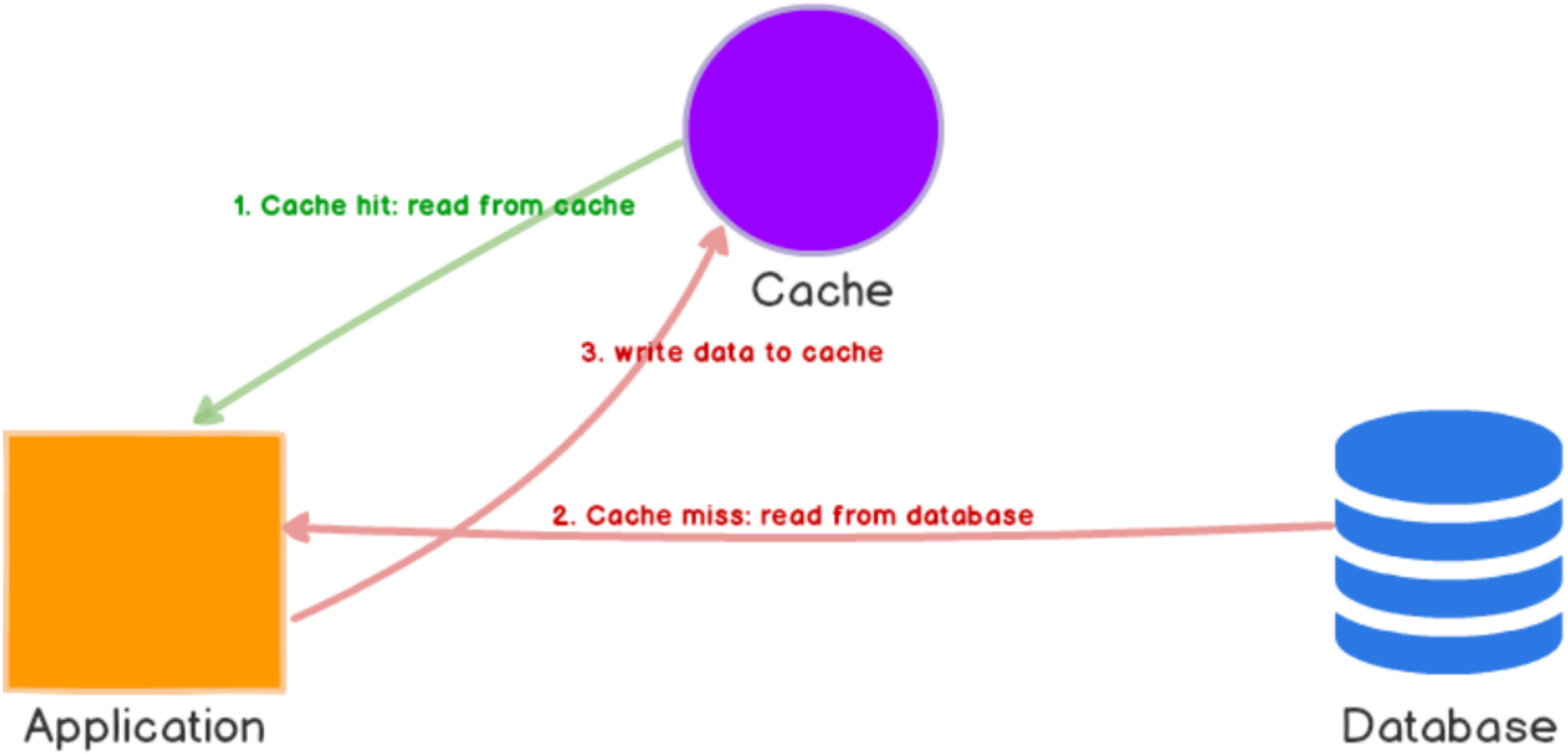
Choice of right caching strategy

Caching strategy depends on the data and **data access patterns**. In other words, how the data is written and read. For example:

- is the system write heavy and reads less frequently? (e.g. time based logs)
- is data written once and read multiple times? (e.g. User Profile)
- is data returned always unique? (e.g. search queries)

- Choosing the right caching strategy is the key to improving performance.

Cache-Aside

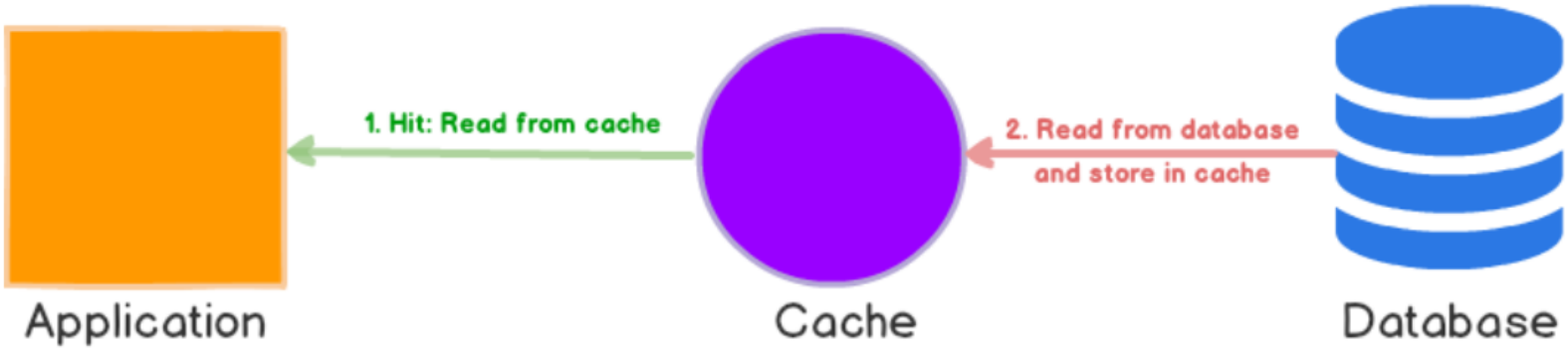

1. Cache hit: read from cache
Cache
3. write data to cache
2. Cache miss: read from database
Application
Database

Detail

1. The application first checks the cache.
2. If the data is found in cache, we've *cache hit*. The data is read and returned to the client.
3. If the data is **not found** in cache, we've *cache miss*. The application has to do some **extra work**. It queries the database to read the data, returns it to the client and stores the data in cache so the subsequent reads for the same data results in a cache hit.

- Cache-aside caches are usually general purpose and work best for **read-heavy workloads**. *Memcached* and *Redis* are widely used.
- Systems using cache-aside are **resilient to cache failures**. If the cache cluster goes down, the system can still operate by going directly to the database.
- When cache-aside is used, the most common write strategy is to write data to the database directly. When this happens, cache may become inconsistent with the database.
- To deal with this, developers generally use time to live (TTL) and continue serving stale data until TTL expires.

Read-Through



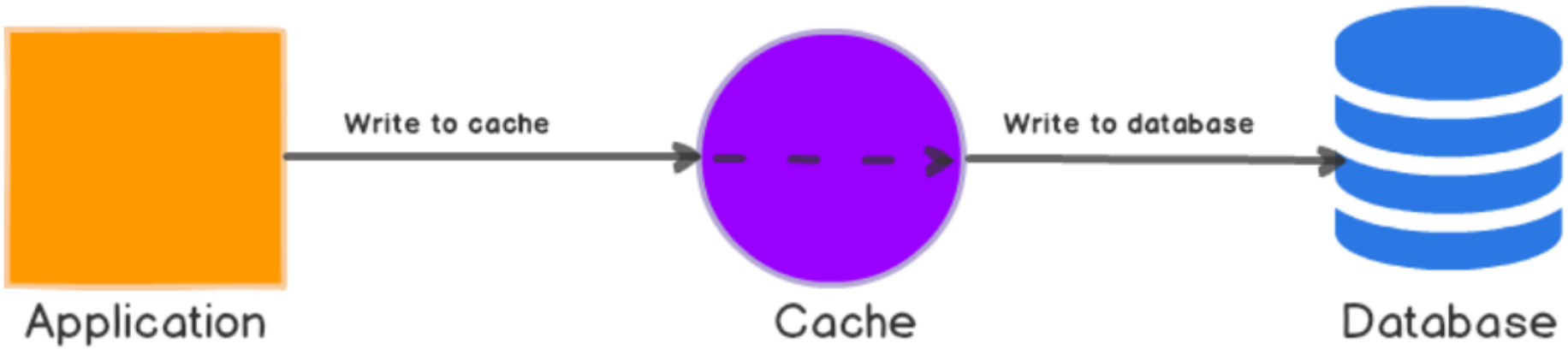
- Read-through cache sits in-line with the database. When there is a cache miss, it loads missing data from database, populates the cache and returns it to the application.
- While read-through and cache-aside are very similar, there are at least two key differences:

1. In cache-aside, the application is responsible for fetching data from the database and populating the cache. In read-through, this logic is usually supported by the library or stand-alone cache provider.
2. Unlike cache-aside, the data model in read-through cache cannot be different than that of the database.

Detail

- Both cache-aside and read-through strategies load data **lazily**, that is, only when it is first read.
- Read-through caches work best for **read-heavy** workloads when the same data is requested many times. For example, a news story.
- The disadvantage is that when the data is requested the first time, it always results in cache miss and incurs the extra penalty of loading data to the cache.
- Developers deal with this by '*warming*' or 'pre-heating' the cache by issuing queries manually.

Write-Through

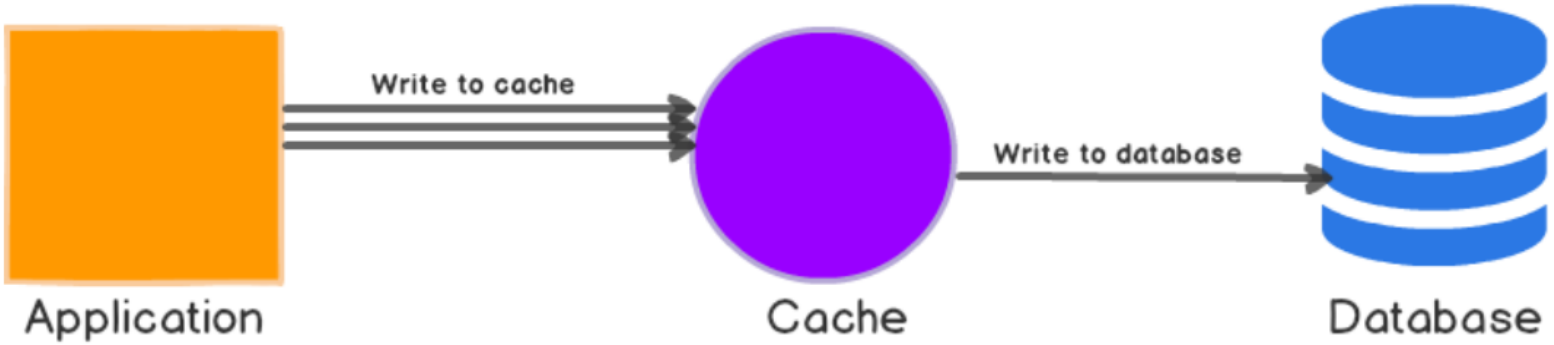


- In this write strategy, data is first written to the cache and then to the database. The cache sits in-line with the database and writes always go *through* the cache to the main database.
- They introduce extra write latency because data is written to the cache first and then to the main database. But when paired with read-through caches, we get all the benefits of read-through and we also get data consistency guarantee, freeing us from using cache invalidation techniques.
- DynamoDB Accelerator (DAX) is an example of read-through / write-through cache.

Write-Around

- Here, data is written directly to the database and only the data that is read makes it way into the cache.
- Write-around can be combined with read-through and provides good performance in situations where data is written once and read less frequently or never. For example, real-time logs or chatroom messages.
- Likewise, this pattern can be combined with cache-aside as well.

Write-Back



- Here, the application writes data to the cache which acknowledges immediately and after some *delay*, it writes the data *back* to the database.
- Write back caches improve the write performance and are good for **write-heavy** workloads.
- When combined with read-through, it works good for mixed workloads, where the most recently updated and accessed data is always available in cache.
- It's resilient to database failures and can tolerate some database downtime.
- Some developers use Redis for both cache-aside and write-back to better absorb spikes during peak load.

Consequence of wrong choice

- If you choose *write-through/read-through* when you actually should be using *write-around/read-through* (written data is accessed less frequently), you'll have useless junk in your cache.
- If the cache is big enough, it may be fine. But in many real-world, high-throughput systems, when memory is never big enough and server costs are a concern, the right strategy, matters.

Load Balancing

- Load balancing distributes incoming network traffic among multiple servers and resources. It helps ensure that no single server becomes overworked. It also prevents:
  - Slowdowns
  - Dropped requests
  - Server crashes
- When a server is unable to handle incoming requests, a load balancing server will direct incoming traffic to another available server.
- A load balancer receives incoming requests from endpoint devices (laptops, desktops, cell phones, IoT devices, etc.) and uses algorithms to route each request to one or more servers in its server group.

Types of Load Balancers

- Load balancing can be performed:
  - By physical servers: hardware load balancers
  - By virtualized servers: software load balancers
  - As a cloud service: Load Balancer as a Service (LBaaS), such as AWS Elastic
- Types based on OSI layer:
  - **Layer 4 (L4 OSI Transport layer) balancers do not inspect the contents of each packet.** They make routing decisions based on the port and IP addresses of the incoming packets and use Network Address Translation (NAT) to route requests and responses between the selected server and the client.
  - **Layer 7 (L7 OSI Application layer) balancers route traffic at the application level.** They inspect incoming content on a package-by-package basis. L7 balancers route client requests to selected servers using different factors than an L4 balancer, such as HTTP headers, SSL session IDs, and types of content (text, graphics, video, etc.).

Benefits

- **Efficiency.** Load balancers distribute requests across the WAN and the internet, preventing server overload. They also increase response time by using multiple servers to process many requests at the same time.
- **Flexibility.** Servers can be added and removed from server groups as needed. Individual servers can be brought down for maintenance or upgrade without affecting processing.
- **High availability.** Load balancers only send traffic to servers that are currently online. If one server fails, others are still available to handle requests. Large commercial Web sites such as Amazon, Google, and Facebook deploy thousands of load balancing and associated app servers worldwide. Smaller organizations may also employ load balancers to route traffic to redundant servers.
- **Redundancy.** Multiple servers ensure that processing will continue, even when a server failure occurs.
- **Scalability.** When traffic increases, new servers can be automatically added to a server group without bringing down services. When high-volume traffic events end, servers can be removed from the group without disrupting service.

GSLB

- Global server load balancing (GSLB)s can route traffic between geographically dispersed servers located in on premise data centers, in the public cloud, or in private clouds.
- GSLBs are generally configured to send client requests to the closest geographic server or to servers that have the shortest response time.
- Benefits
  - **Disaster recovery.** If a local data center outage occurs, other load balancers in different centers around the world can pick up the traffic
  - **Compliance.** Load balancer settings can be configured to conform to local regulatory requirements
  - **Performance.** Closest server routing can reduce network latency.

Load Balancing Algorithms

- **Least Connection Method.** Clients are routed to servers with the least number of active connections.
- **Least Bandwidth Method.** Clients are routed to servers based on which server is servicing the least amount of traffic, measured in bandwidth.
- **Least Response Time.** Server routing occurs based on the shortest response time generated for each server. Least response time is sometimes used with the least connection method to create a two-tiered method of load balancing.
- **Hashing methods.** Linking specific clients to specific servers based on information in client network packets, such as the user's IP address or another identification method.
- **Round Robin.** Clients are connected to servers in a server group through a rotation list. The first client goes to server 1, second to server 2, and so on, looping back to server 1 when reaching the end of the list.