How is caching done here in the code?

No matter whether the cache is read or written, it's done one block at a time. Each block also has a tag that includes the location where the data was stored in the cache. When data is requested from the cache, a search occurs through the tags to find the specific content that's needed in level one (L1) of the memory. If the correct data isn't found, more searches are conducted in L2.

Cache hit and cache miss

A cache hit describes the situation where content is successfully served from the cache., when a cache is found and then served here

Cold, warm and hot cache

**Cold, Warm, and Hot Caches**

A cache hit can also be described as cold, warm, or hot. In each of these, the speed at which the data is read is described.

A hot cache is an instance where data was read from the memory at the *fastest* possible rate. This happens when the data is retrieved from L1.

A cold cache is the *slowest* possible rate for data to be read, though, it's still successful so it's still considered a cache hit. The data is just found lower in the memory hierarchy such as in L3, or lower.

A warm cache is used to describe data that's found in L2 or L3. It's not as fast as a hot cache, but it's still faster than a cold cache. Generally, calling a cache warm is used to express that it's slower and closer to a cold cache than a hot one.

**What’s cache miss here?**

A cache miss refers to the instance when the memory is searched, and the data isn't found. When this happens, the content is transferred and written into the cache.

**What are 3 cache writing policies here**

**write through policy here**

Write-Through: The cache and the database are updated at the same time. This will ensure the cache and the database have the same data all the time but will slow down the write operation since it has to wait for both writes to succeed.

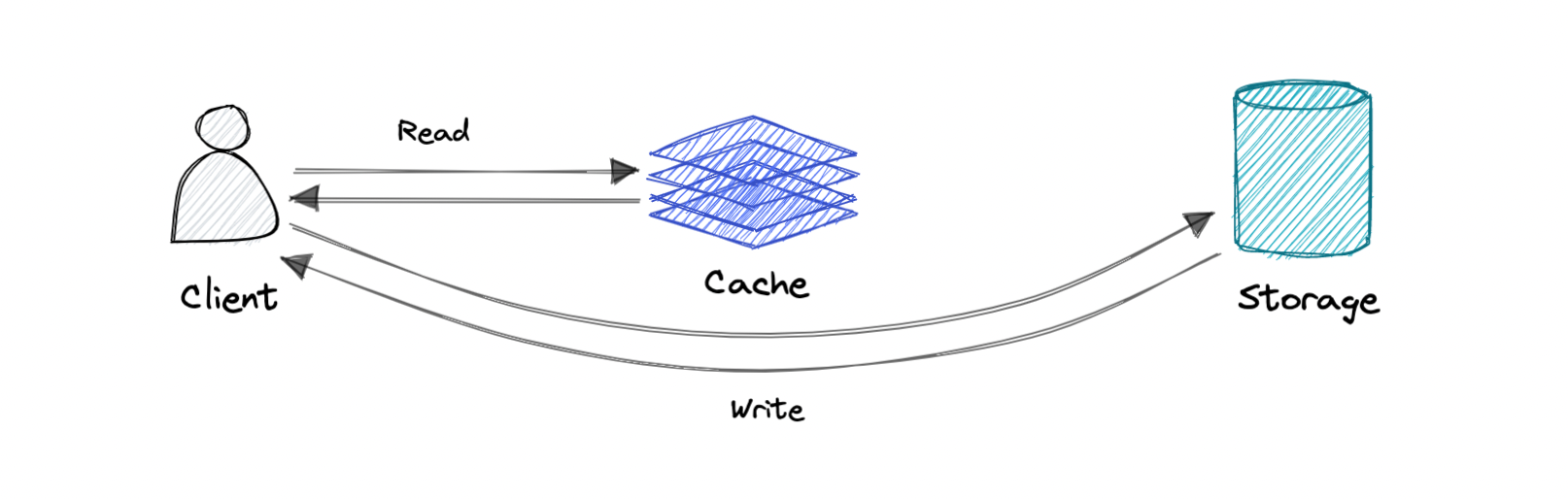
**Pros**

* **Ensures Data Consistency: The Write-Through Cache strategy ensures that the cache and the primary storage are always in sync, providing data consistency.**
* **Works Well with Read-Through Cache: The Write-Through Cache strategy can be used in conjunction with the Read-Through Cache strategy to ensure data consistency.**

**Cons**

* **Introduces Extra Write Latency: Since data is written to the cache and the primary storage at the same time, the Write-Through Cache strategy can introduce extra write latency.**

**Write-around cache**

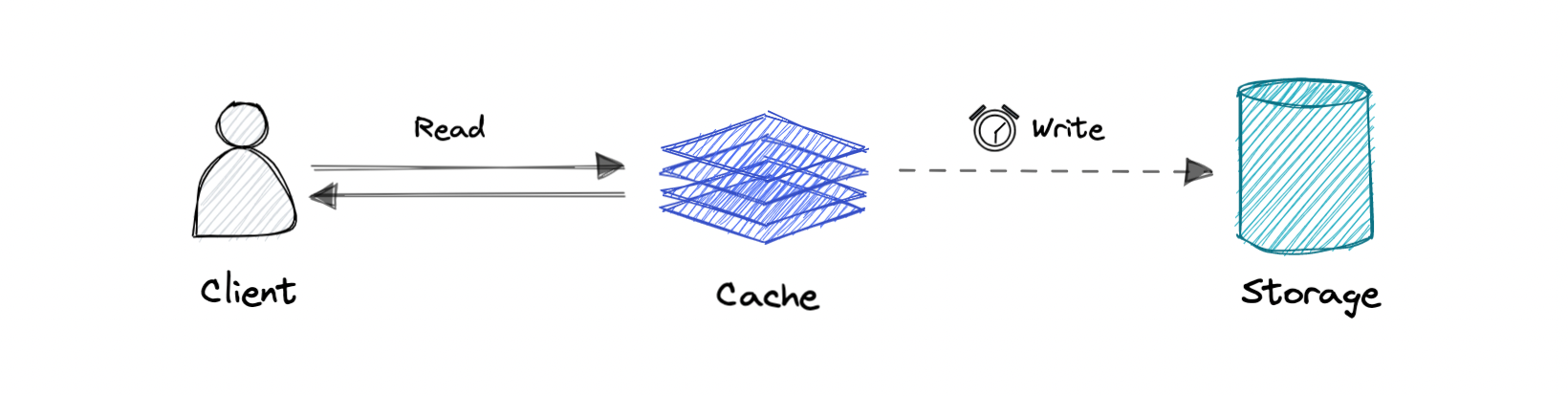


Where write directly goes to the database or permanent storage, bypassing the cache.

**Pro**: This may reduce latency.

**Con**: It increases cache misses because the cache system has to read the information from the database in case of a cache miss. As a result, this can lead to higher read latency in the case of applications that write and re-read the information quickly. Read happen from slower back-end storage and experiences higher latency.

**Write-back cache**



Where the write is only done to the caching layer and the write is confirmed as soon as the write to the cache completes. The cache then asynchronously syncs this write to the database.

In this strategy, the application promptly writes data to the cache, receiving an immediate acknowledgment.

Subsequently, the cache asynchronously updates the data back to the database.

**Difference between this and write back and write around here?**

While resembling Write-Through, there's a key distinction: Write-Through synchronously updates the main database when data is written to the cache, whereas Write-Back asynchronously updates the main database

**Pro**: This would lead to reduced latency and high throughput for write-intensive applications.

**Con**: There is a risk of data loss in case the caching layer crashes. We can improve this by having more than one replica acknowledging the write in the cache.

**What are the applications of each strategy as said?**

**When to Use Each Caching Strategy**[**Permalink**](https://thinhdanggroup.github.io/caching-stategies/#when-to-use-each-caching-strategy)

The choice of caching strategy depends on the specific requirements and access patterns of the system:

* **Cache-Aside and Read-Through Cache** are suitable for read-heavy workloads.
* **Write-Through Cache** is useful when data consistency is important.
* **Write-Around Cache** is appropriate for write-once, read-less-frequently scenarios.
* **Write-Back Cache** is beneficial for write-heavy workloads

**Eviction policies**

Following are some of the most common cache eviction policies:

* **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.
* **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.
* **Least Recently Used (LRU)**: Discards the least recently used items first.
* **Most Recently Used (MRU)**: Discards, in contrast to LRU, the most recently used items first.
* **Least Frequently Used (LFU)**: Counts how often an item is needed. Those that are used least often are discarded first.
* **Random Replacement (RR)**: Randomly selects a candidate item and discards it to make space when necessary.

**What are the top Cache Reading Policies: These policies decide how to read data.**

**1. What’s cache aside and lazy loading?**

In the Cache-Aside strategy, the application looks for data in the cache first. If it doesn’t find the data (a cache miss), it loads data from the source, stores it in the cache, and then returns it. This strategy ensures that only requested data is cached, thus preventing the caching of unnecessary data.

This strategy is best when reads are much more frequent than writes, and data doesn’t get updated often.

**Example**:

Let’s take an e-commerce site like Amazon. When you search for a product, the cache intercepts the request. If it can’t find the product (cache miss), the cache itself queries the database, caches the result, and then returns the product information to you.

**2. Read-Through (Eager Loading)**

In the Read-Through strategy, cache is completely abstracted from the application. When a cache miss occurs, the cache itself is responsible for reading data from the data source, storing it, and then returning it. This approach maintains data consistency and saves the application from interacting with the data source.

This strategy works best when the cost of a cache miss is high and when you want to keep your application logic simple.

**Comparing the different cache evication polciities pros and cons**

1. **Cache Eviction Policies: These policies decide which item should be removed from the cache when the cache is full and a new item needs to be added. LRU and TTL fall into this category.**
   * LRU: Discards the least recently used items first.
   * TTL: Data is kept or removed based on a time limit.

**Use cases**

Caching can have many real-world use cases such as:

* Database Caching
* Content Delivery Network (CDN)
* Domain Name System (DNS) Caching
* API Caching

**When not to use caching?**

Let's also look at some scenarios where we should not use cache:

* Caching isn't helpful when it takes just as long to access the cache as it does to access the primary data store.
* Caching doesn't work as well when requests have low repetition (higher randomness), because caching performance comes from repeated memory access patterns.
* Caching isn't helpful when the data changes frequently, as the cached version gets out of sync, and the primary data store must be accessed every time.

*It's important to note that a cache should not be used as permanent data storage. They are almost always implemented in volatile memory because it is faster, and thus should be considered transient.*

**Advantages**

Below are some advantages of caching:

* Improves performance
* Reduce latency
* Reduce load on the database
* Reduce network cost
* Increase Read Throughput

**Examples**

Here are some commonly used technologies for caching:

**In-memory Caching**[**Permalink**](https://thinhdanggroup.github.io/caching-stategies/#in-memory-caching)

In-memory caching is a technique where frequently accessed data is stored in the computer’s main memory (RAM) for faster retrieval. Since RAM is much faster than disk-based storage, in-memory caching can significantly improve the performance of an application.

**Advantages of In-memory Caching**[**Permalink**](https://thinhdanggroup.github.io/caching-stategies/#advantages-of-in-memory-caching)

* **Faster Data Retrieval**: As data is stored in RAM, in-memory caching provides faster access and low latency.
* **Improved Application Performance**: By reducing the need to access slower data storage systems, in-memory caching can significantly improve application performance.
* **Reduced Load on Backend Systems**: In-memory caching can reduce the load on backend datastores, improving their performance and longevity.

**Disadvantages of In-memory Caching**[**Permalink**](https://thinhdanggroup.github.io/caching-stategies/#disadvantages-of-in-memory-caching)

* **Limited Storage Capacity**: The amount of data that can be stored in-memory is limited by the amount of RAM available.
* **Data Loss on System Failure or Restart**: In-memory caches are typically not persistent. This means that if the system crashes or restarts, any data stored in the cache will be lost.
* **Cost**: RAM is more expensive than disk-based storage. Therefore, in-memory caching can be more costly, especially for large datasets.

**Distributed Caching**[**Permalink**](https://thinhdanggroup.github.io/caching-stategies/#distributed-caching)

Distributed caching involves storing cached data across multiple nodes or servers in a network. This strategy improves the scalability and availability of the cache, as it can handle more data and requests than a single in-memory cache.

**Advantages of Distributed Caching**[**Permalink**](https://thinhdanggroup.github.io/caching-stategies/#advantages-of-distributed-caching)

* **Scalability**: Distributed caching allows for greater storage capacity and improved scalability, as data is distributed across multiple nodes.
* **High Availability and Fault Tolerance**: If one node fails, the data is still available on other nodes. This makes distributed caching highly available and fault-tolerant.
* **Consistency**: Distributed caching solutions often provide consistency mechanisms to ensure that all nodes have the same view of the cached data.

**Disadvantages of Distributed Caching**[**Permalink**](https://thinhdanggroup.github.io/caching-stategies/#disadvantages-of-distributed-caching)

* **Increased Complexity**: Managing a distributed cache can be more complex than managing an in-memory cache. This includes dealing with issues like data consistency, partitioning, and replication.
* **Network Overhead**: In a distributed cache, data must be transmitted over the network, which can introduce latency and increase the load on the network.
* **Data Consistency Challenges**: Ensuring data consistency across all nodes in a distributed cache can be challenging and may require additional mechanisms or protocols.

What are the different cache invalidation strategies here?

**2 — Time-to-Live (TTL) Invalidation**

This strategy allocates a predetermined time interval to each item stored in the cache. Once this interval elapses, the cache is automatically invalidated, and the data is retrieved again from the original source. This method is particularly useful for data that doesn’t require constant accuracy and can endure minor outdatedness.

Consider a news website where articles have a 15-minute TTL in the cache. After this period, the cache is invalidated automatically, prompting the retrieval of the latest article versions.

*A black screen with white text

Description automatically generated*

**This strategy involves two primary methods:**

1. Absolute Expiry: This method ensures that the cache expires at a predetermined time, irrespective of the frequency of access. For example, with a 15-minute expiration setting, the cache will be cleared after 15 minutes, regardless of whether it has been accessed 20 times or not within that period.
2. Sliding Expiry: Contrary to absolute expiry, sliding expiry extends the cache’s lifespan each time it is accessed. Using the same 15-minute timeframe as an example, if the cache is accessed at the 1st, 4th, 7th, 10th, and 12th minutes, its expiration will be postponed for an additional 12 minutes beyond the initial 15-minute mark. This means that frequently accessed caches could, in theory, remain indefinitel

Benefit: Good implementation

Drawback: f not managed properly, cached data could consume significant memory and processing resources, especially if the cached items are large or if the sliding expiry leads to data being retained for extended periods.

Usecases ?

* Known frequency for updating the data, Such as a dashboard displaying daily sales metrics updated every 24 hours.
* For static data that seldom or never changes, such as configuration, feature flags, details requiring software modifications or deployment for updates.

**3 — Event-Based Invalidation**

When a significant event occurs, such as a data update, a system can dispatch a message or signal to the cache, indicating that certain data has been modified. The cache can then proceed to invalidate the pertinent data. This approach requires a messaging/event infrastructure.

In a real-time chat application, when a user sends a message, the messaging server sends an event to all participants in the conversation, signaling an update. This triggers the cache to be invalidated for that specific conversation.

A screen shot of a computer program

Description automatically generated

How can we achieve this?

This process can be achieved through various methods:

1. **Event Generation with Service Brokers**: Utilizing service brokers that are subscribed to by multiple services, ensuring they invalidate their respective caches upon receiving an event.
2. **Incorporating Distributed Caches:** Employing a distributed cache system where the micro-service updating the data source also clears the related cache entries.

**Benefits**:

* Achieves faster consistency, with cache data eviction occurring almost immediately (ranging from milliseconds to seconds) after data changes.
* It scales well, handling high volumes of data changes efficiently.

**Drawbacks**:

* Implementation Complexity: Accurately setting up event-driven invalidation can be challenging, with a high risk of missing event publication or consumption.

Key Takeaways:

* Leveraging managed services and storage engines with built-in invalidation capabilities can enhance performance and simplify implementation.

Usecase:

* Cases where performance is a key consideration, and the risks associated with data inconsistency are manageable.

What’s version based invalidation?

**Version-based cache invalidation (or validation on access)**

Version-based cache invalidation, also known as validate-on-access, is a caching strategy where the validity of cached data is determined based on its version comparison with the source data. This approach checks if the cached data is up-to-date whenever it is accessed, ensuring consistency between the cache and the data source.

**Benefits:**

* Data Consistency: Ensures high levels of data consistency, as cached data is always compared with the latest version before being served.
* Efficiency for Infrequently Updated Data: Highly efficient for data that doesn’t change often, as it reduces unnecessary cache refreshes.
* On-Demand Cache Updates: Cache updates occur only when necessary, optimizing resource usage and reducing the load on the backend systems.

**Drawbacks:**

* Increased Latency: Each cache access requires a version check, which can introduce latency, particularly if the version information is stored remotely.
* Undermine the core reason why cache was used in the first place, as version check may be expensive.
* Complexity in Implementation: Implementing and maintaining versioning information can add complexity to the system.

**Appropriate Usage Scenarios**:

* When dealing with static data that doesn’t change often, the backend can use versioned URLs. These URLs, provided to the browser, direct it to download data from a CDN (Content Delivery Network). When there’s an update to the data, the backend generates a new versioned URL. This new URL signals the CDN to fetch the updated data from the backend.
* For large amounts of cached data, accessing different versions can be made very fast. This is done through methods like indexing or using quick search techniques. These methods help in efficiently retrieving the correct version of the data.
* In situations where even the slightest data inconsistency is unacceptable, using version checks on the data source is beneficial. Even with these checks, caching the content leads to a significant boost in performance. This approach ensures that users always get the most current and accurate data, while still enjoying the speed benefits of caching.

**Key Takeaways:**

* Ensure the overhead of verifying the version against the data source is minimal, and caching continues to offer performance advantages.
* Utilize versioning when maintaining consistency is of utmost importance for your scenario.
* Employing a versioning strategy does not significantly complicate the overall solution.

*Given the varied nature of applications and their specific requirements, a hybrid strategy that combines elements of these approaches can often be the most effective solution. This approach allows for the flexibility to balance between data freshness and system performance, adapting to the changing needs and patterns of usage.*

**How to measure how well your cache is doing?**

**How to Measure Caching Performance**

Caching performance can be measured using various metrics:

1. **Cache Hit Rate**: The percentage of cache accesses that result in a hit (i.e., the data is found in the cache). A high cache hit rate indicates efficient cache utilization.
2. **Cache Miss Rate**: The percentage of cache accesses that result in a miss (i.e., the data is not found in the cache and needs to be fetched from the primary storage). A high cache miss rate can indicate poor cache utilization.
3. **Cache Latency**: The time it takes to access data from the cache. Low cache latency indicates fast access to cache data.
4. **Cache Coherence**: The consistency of data across multiple caches in a multi-core system. High cache coherence is desirable for data consistency and synchronization.

How to Interpreting Caching Performance Metrics

Interpreting caching performance metrics requires understanding the specific metrics being measured and their impact on system performance:

1. **Cache Hit Rate**: A high cache hit rate indicates that the cache is effectively serving requests from the cache without needing to fetch data from the backend. This is desirable as it leads to faster response times and lower load on the backend systems.
2. **Cache Miss Rate**: A high cache miss rate may indicate that the cache is not effectively storing frequently accessed data. This can lead to increased load on the backend systems and slower response times. In this case, you may need to optimize your caching strategy or increase your cache size.
3. **Cache Latency**: Low cache latency indicates fast access to cache data, which can improve overall system performance. High cache latency, on the other hand, can lead to slower response times and may indicate a need for optimization.
4. **Cache Coherence**: High cache coherence indicates that the data in the cache is consistent across all cores in a multi-core system. Low cache coherence may indicate potential data inconsistency issues

U