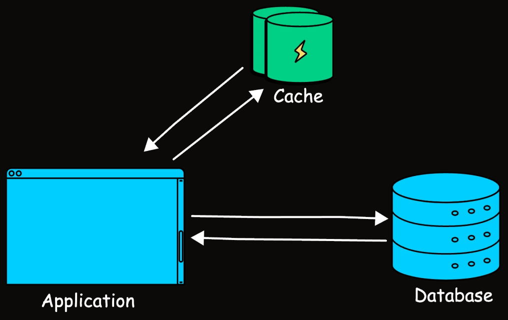
Caching

Ashish Pratap Singh

3 min read

aching is a technique used to temporarily store copies f data in **high-speed storage** layers (such as RAM) to educe the time taken to access data.



he primary goal of caching is to improve system performance by **reducing latency**, offloading the main data tore, and providing **faster data retrieval.**

. Why Use Caching?

aching is essential for the following reasons:

- **Improved Performance:** By storing frequently accessed data in a cache, the time required to retrieve that data is significantly reduced.
- **Reduced Load on Backend Systems:** Caching reduces the number of requests that need to be processed by the backend, freeing up resources for other operations.
- **Increased Scalability**: Caches help in handling a large number of read requests, making the system more scalable.
- **Cost Efficiency**: By reducing the load on backend systems, caching can help lower infrastructure costs.
- **Enhanced User Experience**: Faster response times lead to a better user experience, particularly for web and mobile applications.

2. Types of Caching

2.1 In-Memory Cache

In-memory caches store data in the **main memory (RAM)** for extremely fast access.

These caches are typically used for session management, storing frequently accessed objects, and as a front for databases.

> Examples: **Redis** and **Memcached**.

2.2 Distributed Cache

A distributed cache spans **multiple servers** and is designed to handle large-scale systems.

It ensures that cached data is available across different nodes in a distributed system.

> Examples: **Redis Cluster** and **Amazon ElastiCache**.

2.3 Client-Side Cache

Client-side caching involves storing data on the client device, typically in the form of **cookies**, **local storage**, or application-specific caches.

This is commonly used in web browsers to cache static assets like images, scripts, and stylesheets.

2.4 Database Cache

Database caching involves storing frequently queried database results in a cache.

This reduces the number of queries made to the database, improving performance and scalability.

2.5 Content Delivery Network (CDN)

CDN is used to store copies of content on servers distributed across different geographical locations.

This reduces latency by serving content from a server closer to the user.

3. Caching Strategies

- **Read-Through Cache**: The application first checks the cache for data. If it's not there (a cache miss), it retrieves the data from the database and updates the cache.
- **Write-Through Cache**: Data is written to both the cache and the database simultaneously, ensuring consistency but potentially impacting write performance.
- **Write-Back Cache**: Data is written to the cache first and later synchronized with the database, improving write performance but risking data loss.
- **Cache-Aside (Lazy Loading)**: The application is responsible for reading and writing from both the cache and the database.

4. Cache Eviction Policies

To manage the limited size of a cache, eviction policies are used to determine which data should be removed when the cache is full.

1. **Least Recently Used (LRU)**: LRU evicts the least recently accessed data when the cache is full. It assumes that recently used data will likely be used again soon.
2. **Least Frequently Used (LFU):** LFU evicts data that has been accessed the least number of times, under the assumption that rarely accessed data is less likely to be needed.
3. **First In, First Out (FIFO):** FIFO evicts the oldest data in the cache first, regardless of how often or recently it has been accessed.
4. **Time-to-Live (TTL):** TTL is a time-based eviction policy where data is removed from the cache after a specified duration, regardless of usage.

---

5. Challenges and Considerations

1. **Cache Coherence**: Ensuring that data in the cache remains consistent with the source of truth (e.g., the database).
2. **Cache Invalidation**: Determining when and how to update or remove stale data from the cache.
3. **Cold Start**: Handling scenarios when the cache is empty, such as after a system restart.
4. **Cache Eviction Policies**: Deciding which items to remove when the cache reaches capacity (e.g., Least Recently Used, Least Frequently Used).
5. **Cache Penetration**: Preventing malicious attempts to repeatedly query for non-existent data, potentially overwhelming the backend.
6. **Cache Stampede**: Managing situations where many concurrent requests attempt to rebuild the cache simultaneously.

6. Best Practices for Implementing Caching

- **Cache the Right Data:** Focus on caching data that is expensive to compute or retrieve and that is frequently accessed.
- **Set Appropriate TTLs:** Use TTLs to automatically invalidate cache entries and prevent stale data.

- **Consider Cache Warming**: Preload essential data into the cache to avoid cold starts.
- **Monitor Cache Performance:** Regularly monitor cache hit/miss ratios and adjust caching strategies based on usage patterns.
- **Use Layered Caching:** Implement caching at multiple layers (e.g., client-side, server-side, CDN) to maximize performance benefits.
- **Handle Cache Misses Gracefully:** Ensure that the system can handle cache misses efficiently without significant performance degradation.

7. Conclusion

Caching is a powerful technique in system design that, when implemented correctly, can drastically improve the performance, scalability, and cost-efficiency of a system.

However, it comes with its own set of challenges, particularly around consistency and invalidation.

By understanding the different types of caches, cache placement strategies, and best practices, you can design a robust caching strategy that meets the needs of your application.