**Cache in System Design**

**🛠️ What is Caching?**

Caching is a technique used to store frequently accessed data in a fast storage layer to reduce latency and improve system performance. Instead of fetching data from a slow database or making an expensive computation every time, the system **retrieves it from a cache**, which is much faster.

**🛠️ Example: YouTube Video Streaming**

1️⃣ A user watches a video → YouTube fetches the video from its storage.  
2️⃣ Instead of retrieving the video **every time** from the main database, YouTube **caches popular videos** in a **Content Delivery Network (CDN)** near the user’s location.  
3️⃣ The next time another user requests the same video, it **loads instantly from the cache** instead of streaming from YouTube’s main servers.

✅ **Faster video load times**  
✅ **Reduced load on backend servers**

**🔴 Why is Caching Needed?**

🚫 **Slow Database Queries** → Fetching data from a relational database takes time.  
🚫 **Expensive Computation** → Calculating trending hashtags on Twitter **for every request** is inefficient.  
🚫 **High Traffic** → If millions of users request the same data, caching prevents the backend from overloading.

**🟢 Where is Caching Used?**

**1️⃣ Application-Level Cache (In-Memory)**

✅ Stores frequently used data inside the application memory.  
✅ Example: **Django & Express.js applications** use in-memory cache for user sessions.

**2️⃣ Database Query Cache**

✅ Stores results of **frequent database queries**.  
✅ Example: **Amazon** caches product details so repeated searches don’t hit the database.

**3️⃣ Content Delivery Network (CDN)**

✅ **Stores static files like images, videos, and scripts** close to users.  
✅ Example: **Netflix, YouTube** use CDNs to stream videos quickly.

**4️⃣ Web Browser Cache**

✅ Stores **CSS, JavaScript, and images** in the user’s browser.  
✅ Example: When you visit **Facebook**, your browser caches profile pictures to load faster next time.

**5️⃣ Distributed Caching (Global Scale)**

✅ Data is cached across multiple servers for **high availability**.  
✅ Example: **Twitter** caches tweets across **multiple data centers** worldwide.

**🛠️ Cache Strategies**

**1️⃣ Write-Through Cache**

✅ **Writes data to both the cache and database** at the same time.  
✅ Ensures **data consistency** but has slightly higher write latency.  
📌 **Use Case:** Banking transactions where data must always be consistent.

**2️⃣ Write-Back Cache**

✅ Writes data **only to the cache first**, and updates the database later.  
✅ Faster writes but risks **data loss** if the cache fails before syncing with the database.  
📌 **Use Case:** Caching product recommendations on **e-commerce websites**.

**3️⃣ Least Recently Used (LRU) Eviction**

✅ Removes **least accessed** data when cache is full.  
📌 **Use Case:** Web browsers evict older cached pages when memory is full.

**4️⃣ Time-to-Live (TTL) Expiry**

✅ Data **expires after a set time** to avoid stale information.  
📌 **Use Case:** **Stock prices** expire in **1 second** to keep financial data fresh.

**📌 Cache vs. Database**

| **Feature** | **Cache (Redis/Memcached)** | **Database (MySQL/PostgreSQL)** |
| --- | --- | --- |
| **Speed** | Fast (microseconds) | Slow (milliseconds) |
| **Storage** | Small (RAM-based) | Large (Disk-based) |
| **Persistence** | Temporary | Permanent |
| **Use Case** | High-speed retrieval | Long-term storage |

**🛠️ When to Use Caching?**

✅ When **data is frequently accessed** (e.g., user profiles, trending topics).  
✅ When **database queries are slow** and need optimization.  
✅ When handling **millions of concurrent users** and reducing server load.

🚫 **When NOT to Use Caching?**  
❌ When **data needs to be 100% consistent** (e.g., bank account balances).  
❌ When **data changes frequently** (e.g., real-time bidding in stock markets).

**🛠️ Real-World Examples**

| **Company** | **Use Case** | **Caching Strategy** |
| --- | --- | --- |
| **Netflix** | Video Streaming | CDN caching |
| **Amazon** | Product search results | Query caching |
| **Twitter** | Trending topics | Distributed caching |
| **Facebook** | User profiles | In-memory caching |

**Cache Invalidation Techniques**

Cache invalidation is the process of **removing or updating cached data** when it becomes outdated or no longer valid. This ensures that users always get the **most recent and accurate** data while still benefiting from caching.

**🛠️ Why is Cache Invalidation Important?**

🚨 **Problem:** Cached data can become **stale (outdated)** if the source database is updated but the cache isn’t refreshed.  
✅ **Solution:** Proper invalidation techniques ensure consistency while keeping the system fast.

Example:

* If **Amazon updates product prices**, but the cache still has old prices, users might see incorrect prices!
* Cache invalidation ensures that outdated prices **are removed or updated** to match the database.

**🔴 Types of Cache Invalidation Strategies**

**1️⃣ Time-to-Live (TTL) Expiry ⏳**

🛠️ **How it Works:**

* Every cached item has an **expiration time** (TTL).
* After TTL expires, the cache **automatically removes** the item.
* The system fetches fresh data from the database when needed.

📌 **Example:**

* **Stock market data** → Prices change every second, so a TTL of **1 second** ensures users always get fresh data.
* **Weather apps** → TTL of **10 minutes** updates temperatures periodically.

✅ **Best for:** Frequently changing data (stock prices, leaderboard scores).  
❌ **Not ideal if:** You need real-time updates immediately.

**2️⃣ Least Recently Used (LRU) Eviction 📉**

🛠️ **How it Works:**

* When the cache **is full**, it **removes the least recently accessed** data.
* New data **replaces old, less-used** entries.

📌 **Example:**

* **Web browsers** → Your browser caches website assets but removes **older pages** if storage is full.
* **Redis & Memcached** use LRU to **automatically discard old entries**.

✅ **Best for:** Limited cache memory where some data is rarely accessed.  
❌ **Not ideal if:** You need to keep all important data cached.

**3️⃣ Write-Through Cache 📝**

🛠️ **How it Works:**

* **Every write operation** updates **both** the database **and** cache at the same time.
* Ensures **data consistency** between cache and database.

📌 **Example:**

* **Bank transactions** → A balance update must reflect **immediately** in both cache & database.
* **E-commerce inventory** → When an item is purchased, the stock count is updated **instantly**.

✅ **Best for:** Systems needing **strong consistency** (banking, order processing).  
❌ **Not ideal if:** Write operations are **frequent and slow down performance**.

**4️⃣ Write-Back (Lazy Write) Cache 📝**

🛠️ **How it Works:**

* **Writes go to the cache first**, and the database is **updated later** (asynchronously).
* Improves **write speed**, but risks **data loss** if the cache crashes before writing to the database.

📌 **Example:**

* **Product recommendations** → Updating recommendations **immediately** in cache but updating the database **later**.
* **Game leaderboards** → Storing score updates in the cache first, then updating the database **after some time**.

✅ **Best for:** **High-speed applications** where occasional data loss is acceptable.  
❌ **Not ideal if:** Data **must always be consistent** (e.g., bank transactions).

**5️⃣ Cache Aside (Lazy Loading) 🛠️**

🛠️ **How it Works:**

1. **Check the cache first**
   * If the data is **in cache** → Return it. ✅
   * If the data is **not in cache** → Fetch from database and store it in cache for next time. ✅

📌 **Example:**

* **Netflix movie metadata** → When a user requests a movie, check the cache. If it's not there, **fetch from the database and cache it** for the next user.

✅ **Best for:** **Read-heavy** systems with occasional updates.  
❌ **Not ideal if:** Cached data frequently becomes **outdated**.

**📌 Summary Table**

| **Strategy** | **How It Works** | **Best Use Cases** | **Trade-offs** |
| --- | --- | --- | --- |
| **TTL Expiry** ⏳ | Data expires after a set time | Stock prices, weather data | May lead to stale data |
| **LRU Eviction** 📉 | Removes least used data when cache is full | Web browsers, Redis | Might remove important data |
| **Write-Through** 📝 | Writes to both cache & DB at the same time | Banking, order systems | Slower writes |
| **Write-Back** 📝 | Writes to cache first, DB later | Game leaderboards, recommendations | Risk of data loss |
| **Cache-Aside** 🚀 | Loads from cache if available, else fetches from DB | Netflix, news feeds | Cache miss causes delay |

**🔹 When to Use Which Cache Strategy?**

✅ **Use Write-Through** if **data consistency is crucial** (e.g., financial transactions).  
✅ **Use Write-Back** if you need **faster writes** and can tolerate **some data loss** (e.g., recommendation engines).  
✅ **Use TTL Expiry** for **frequently changing data** (e.g., weather updates, stock prices).  
✅ **Use LRU** when cache memory is **limited** and needs **automatic cleanup** (e.g., web browsers).  
✅ **Use Cache-Aside** for **read-heavy applications** where data doesn't change often (e.g., news articles).

**🔹 Real-World Examples**

| **Company** | **Cache Strategy Used** | **Use Case** |
| --- | --- | --- |
| **Netflix** | Cache-Aside + CDN | Streaming metadata & videos |
| **Amazon** | Write-Through | Product availability |
| **Facebook** | Write-Back | User feed recommendations |
| **Google Maps** | TTL Expiry | Traffic & location data |
| **Twitter** | LRU Eviction | Trending topics |

**📌 Key Takeaways**

✅ **Caching boosts performance** but needs **proper invalidation** to avoid stale data.  
✅ **Different strategies work best for different use cases** – **Choose based on your needs!**  
✅ **Mixing multiple strategies** (e.g., CDN + Cache-Aside + Write-Through) is common in **large-scale systems** like Netflix and Amazon.

**Redis vs. Memcached vs. CDN Caching: In-Depth Comparison**

Caching is essential for improving **performance, reducing latency, and scaling** applications. Three common caching solutions are **Redis, Memcached, and Content Delivery Networks (CDN).** Each has its **strengths and weaknesses** depending on the use case.

**🔹 1. Redis (Remote Dictionary Server)**

✅ **Best For:** Database caching, real-time analytics, message queues

**🛠️ How Redis Works**

* **In-memory data store** → Data is stored in **RAM** for ultra-fast access.
* Supports **key-value pairs, lists, sets, hashes, sorted sets**, etc.
* **Persistent storage** → Unlike Memcached, Redis can save data to disk.
* **Pub/Sub messaging** → Supports **real-time** messaging (e.g., chat applications).
* **Transactions & Scripting** → Allows atomic operations and scripting via Lua.

📌 **Example Use Cases**  
1️⃣ **Session Management** → Stores user sessions in web applications (e.g., login sessions in e-commerce).  
2️⃣ **Leaderboards & Ranking** → Uses sorted sets for real-time leaderboards (e.g., gaming apps).  
3️⃣ **Message Queues** → Acts as a broker for message queues (e.g., job processing in background tasks).  
4️⃣ **Rate Limiting** → Prevents API abuse by tracking request counts (e.g., GitHub API rate limiting).

✅ **Pros:**  
✔ **Supports persistence** (can recover after a crash).  
✔ **Rich data structures** (not just key-value storage).  
✔ **Fast read/write speed** (ideal for real-time apps).

❌ **Cons:**  
✖ **Consumes a lot of RAM** (costly at scale).  
✖ **More complex to set up than Memcached**.

**💡 When to Use Redis?**  
🔹 When you need **data persistence** (Memcached doesn’t support this).  
🔹 For **real-time applications** (e.g., chat apps, leaderboards).  
🔹 When you need **advanced data structures** (e.g., sorted sets, lists).

**🔹 2. Memcached**

✅ **Best For:** Simple key-value caching, high-speed reads

**🛠️ How Memcached Works**

* **In-memory key-value store** with **lightweight architecture**.
* **Does NOT support persistence** → Data is lost on restart.
* **Distributed caching** → Can be easily scaled across multiple nodes.
* **Ideal for simple caching** of database queries, API responses, or HTML fragments.

📌 **Example Use Cases**  
1️⃣ **Database Query Caching** → Caches expensive SQL queries (e.g., caching product details in an e-commerce store).  
2️⃣ **Web Page Caching** → Stores rendered HTML fragments to reduce load times.  
3️⃣ **API Response Caching** → Reduces backend requests for frequently accessed API data.

✅ **Pros:**  
✔ **Blazing fast** (optimized for simple key-value lookups).  
✔ **Scales easily** → Can distribute cache across multiple servers.  
✔ **Lower memory usage compared to Redis**.

❌ **Cons:**  
✖ **No data persistence** (data is lost on reboot).  
✖ **Limited data types** (only key-value storage).  
✖ **No built-in replication or clustering** (Redis supports these).

**💡 When to Use Memcached?**  
🔹 When **data persistence is not required** (e.g., caching database queries).  
🔹 When you need **simple, ultra-fast key-value storage**.  
🔹 When you want a **lightweight solution** that’s easier to scale.

**🔹 3. CDN (Content Delivery Network)**

✅ **Best For:** Caching static files, reducing latency for global users

**🛠️ How a CDN Works**

* A **network of globally distributed servers** caches and delivers **static content** (images, videos, CSS, JavaScript).
* **Reduces latency** by serving content from a **server closer to the user**.
* **Reduces bandwidth costs** by offloading traffic from the origin server.

📌 **Example Use Cases**  
1️⃣ **Website Optimization** → Delivers images, videos, CSS, and JavaScript faster (e.g., Amazon, YouTube, Facebook).  
2️⃣ **Streaming Services** → Caches video content close to users for smooth playback (e.g., Netflix, Twitch).  
3️⃣ **Gaming Servers** → Delivers game patches and updates with low latency (e.g., Fortnite, Call of Duty).

✅ **Pros:**  
✔ **Massively reduces latency** (faster content delivery).  
✔ **Lowers server load** (offloads traffic from the origin).  
✔ **Improves global availability** (better user experience worldwide).

❌ **Cons:**  
✖ **Doesn’t cache dynamic content** (only works for static files).  
✖ **More expensive than local caching** (e.g., Redis/Memcached).

**💡 When to Use a CDN?**  
🔹 When serving **global users** (reduces latency for international traffic).  
🔹 When caching **static content** (e.g., images, videos, CSS).  
🔹 When handling **high-traffic applications** (e.g., e-commerce, streaming).

**🔹 Key Differences: Redis vs. Memcached vs. CDN**

| **Feature** | **Redis** | **Memcached** | **CDN** |
| --- | --- | --- | --- |
| **Type** | In-memory key-value store | In-memory key-value store | Distributed caching for static content |
| **Persistence** | ✅ Yes | ❌ No | ❌ No |
| **Data Types** | ✅ Supports lists, sets, hashes, sorted sets | ❌ Key-value only | ❌ Static files only |
| **Scaling** | ✅ Clustering, replication | ✅ Horizontal scaling | ✅ Global distribution |
| **Use Case** | Database caching, messaging, rate limiting | Simple caching, session storage | Static content delivery |
| **Best For** | Real-time apps, complex caching | Lightweight, simple caching | Website performance, media streaming |

**📌 When to Use What?**

🔹 **Use Redis**  
✅ When you need **persistence** and **advanced data types**.  
✅ Best for **real-time applications**, **message queues**, and **rate limiting**.  
✅ Example: **Chat applications, leaderboards, user sessions**.

🔹 **Use Memcached**  
✅ When you need **simple, ultra-fast caching** with **high throughput**.  
✅ Best for **database query caching, web page fragments, and API responses**.  
✅ Example: **Reducing database load in e-commerce apps**.

🔹 **Use a CDN**  
✅ When serving **global users with static content** (images, videos, CSS).  
✅ Best for **reducing latency and bandwidth costs**.  
✅ Example: **Netflix streaming, Amazon product images, Facebook profile pictures**.

**🔹 Real-World Use Cases**

| **Company** | **Caching Solution Used** | **Use Case** |
| --- | --- | --- |
| **Netflix** | Redis + CDN | Streaming metadata + video delivery |
| **Facebook** | Memcached | Caching user posts & comments |
| **Amazon** | Redis | Product recommendations, cart management |
| **Twitch** | CDN | Live video streaming |
| **Twitter** | Redis | Caching timeline & notifications |

**📌 Final Thoughts**

✅ **Redis is best for real-time applications** and **persistent caching**.  
✅ **Memcached is best for simple, high-speed caching** (without persistence).  
✅ **CDN is best for delivering static content** efficiently to global users.

**Redis Replication, Clustering & Pub/Sub Explained in Detail**

Redis is widely used for **high-performance caching, real-time applications, and distributed systems.** Three key features—**Replication, Clustering, and Pub/Sub**—make Redis **scalable, resilient, and suitable for distributed architectures.** Let’s break them down with real-world examples.

**🔹 1. Redis Replication**

✅ **Best For:** High availability, read scalability, disaster recovery

**🛠️ How Redis Replication Works?**

* Redis **Primary (Master) replicates data** to one or more **Replica (Slaves).**
* **Read operations** can be performed on replicas to reduce the load on the primary.
* If the primary fails, a replica can be **promoted** as the new primary.

📌 **Example Use Cases**  
1️⃣ **Read-Heavy Applications** → Offload reads to replicas (e.g., caching user profiles in Twitter).  
2️⃣ **Disaster Recovery** → If the primary crashes, a replica can take over (e.g., banking systems).  
3️⃣ **Multi-Region Deployment** → Replicas in different regions reduce latency (e.g., Netflix, global CDNs).

✅ **Pros:**  
✔ Improves **fault tolerance** (replica can take over if the primary fails).  
✔ **Scales reads** by distributing queries across replicas.  
✔ **Automatic resynchronization** if a replica disconnects.

❌ **Cons:**  
✖ **Writes still go to one primary**, which can become a bottleneck.  
✖ Replication lag can cause **eventual consistency issues**.

**💡 When to Use Redis Replication?**  
🔹 When you have **more reads than writes** and want to distribute the load.  
🔹 When you need **high availability** and a **backup** in case the primary fails.  
🔹 When serving **global users** (deploy replicas in different regions).

**🔹 2. Redis Clustering**

✅ **Best For:** **Horizontal scaling, high availability, sharding**

**🛠️ How Redis Clustering Works?**

* **Data is partitioned (sharded) across multiple Redis nodes.**
* Each **node holds a subset of the data** and handles requests for that subset.
* **Automatic failover** if a node fails (another node takes over).

📌 **Example Use Cases**  
1️⃣ **Scalable Caching** → Large-scale apps (e.g., Amazon, Google) store massive amounts of data.  
2️⃣ **Real-Time Analytics** → Fast processing of data from multiple sources (e.g., Uber's trip tracking).  
3️⃣ **Gaming Leaderboards** → Distributed across multiple nodes for low latency (e.g., Fortnite).

✅ **Pros:**  
✔ **Supports horizontal scaling** (no single primary bottleneck).  
✔ **Fault tolerance** (if one node fails, others continue).  
✔ **Handles millions of requests per second**.

❌ **Cons:**  
✖ **Complex setup & management** compared to single-node Redis.  
✖ **Requires client-side logic** to interact with multiple shards.

**💡 When to Use Redis Clustering?**  
🔹 When you need **massive scalability** beyond a single machine.  
🔹 When **replication alone is not enough** (to handle both high reads and writes).  
🔹 When running **real-time, distributed applications**.

**🔹 3. Redis Pub/Sub (Publish/Subscribe Messaging)**

✅ **Best For:** **Event-driven systems, real-time notifications, chat apps**

**🛠️ How Redis Pub/Sub Works?**

* **Publishers send messages** to a Redis channel.
* **Subscribers listen** to the channel and receive messages in real-time.
* **No message persistence** → If a subscriber is offline, they miss messages.

📌 **Example Use Cases**  
1️⃣ **Chat Applications** → Users receive messages instantly (e.g., WhatsApp, Slack).  
2️⃣ **Real-Time Notifications** → Updates like stock price changes, sports scores.  
3️⃣ **Live Streaming Events** → Broadcasting updates to multiple users (e.g., Twitch chat).

✅ **Pros:**  
✔ **Low latency** (messages delivered instantly).  
✔ **Decouples services** (microservices communicate without direct dependencies).  
✔ **Efficient for real-time apps**.

❌ **Cons:**  
✖ **No message history** → Missed messages are not stored.  
✖ **Not ideal for guaranteed delivery** (use Kafka for persistent messaging).

**💡 When to Use Redis Pub/Sub?**  
🔹 When you need **instant, real-time communication**.  
🔹 When **message persistence isn’t critical**.  
🔹 When designing **event-driven systems** (e.g., stock market apps, push notifications).

**🔹 Key Differences: Replication vs. Clustering vs. Pub/Sub**

| **Feature** | **Replication** | **Clustering** | **Pub/Sub** |
| --- | --- | --- | --- |
| **Use Case** | High availability, read scalability | Horizontal scaling, large datasets | Real-time messaging, event streaming |
| **Scalability** | ✅ Read scaling only | ✅ Read + write scaling | ❌ Not for data storage |
| **Failure Handling** | ✅ Automatic failover | ✅ Auto-recovery in clusters | ❌ No failover |
| **Data Persistence** | ✅ Yes | ✅ Yes | ❌ No |
| **Best For** | Backup, read-heavy workloads | Large-scale apps, distributed caching | Chat apps, real-time notifications |

**📌 When to Use What?**

🔹 **Use Redis Replication**  
✅ If you need **fault tolerance and read scalability**.  
✅ Best for **reducing database load** and **ensuring high availability**.  
✅ Example: **Multi-region database caching, failover support in financial systems**.

🔹 **Use Redis Clustering**  
✅ If you need **massive scalability and high availability**.  
✅ Best for **large-scale applications with millions of requests**.  
✅ Example: **Netflix caching, Uber ride-tracking, gaming leaderboards**.

🔹 **Use Redis Pub/Sub**  
✅ If you need **real-time messaging with low latency**.  
✅ Best for **chat applications, notifications, event-driven systems**.  
✅ Example: **Slack, WhatsApp, stock market updates, IoT event streaming**.

**📌 Real-World Use Cases**

| **Company** | **Redis Feature Used** | **Use Case** |
| --- | --- | --- |
| **Twitter** | **Replication** | Caching tweets, high availability |
| **Netflix** | **Clustering** | Large-scale caching of video metadata |
| **WhatsApp** | **Pub/Sub** | Real-time chat messaging |
| **Uber** | **Clustering** | Tracking trips & ride requests |
| **Stock Exchange** | **Pub/Sub** | Real-time stock updates |

**📌 Final Thoughts**

✅ **Redis Replication** → Use when **scaling reads and ensuring high availability**.  
✅ **Redis Clustering** → Use when **handling massive scale and sharding data**.  
✅ **Redis Pub/Sub** → Use when **real-time messaging is required**.