Chapter 20

Caching

1. Introduction to Caching

- **Context:** If a significant fraction of requests sent to a data store consists of a *small pool of frequently accessed entries*, introducing a cache can improve application performance and reduce load on the data store.
- **Definition:** A cache is a *high-speed storage layer* that temporarily buffers responses from an origin (like a data store), allowing future requests for the same data to be served directly from the cache.
- **Guarantees:** Caching provides *best-effort guarantees*; its state is disposable and can be rebuilt from the origin if necessary.
- **Prior Examples:** Caching principles have been seen in DNS protocol and Content Delivery Networks (CDNs).
- **Cost-Effectiveness:** For a cache to be cost-effective, the **hit ratio** (proportion of requests served directly from the cache) should be high.
- **Factors Affecting Hit Ratio:**
  - The *universe of cachable objects* (the fewer, the better).
  - The *likelihood of accessing the same objects repeatedly* (the higher, the better).
  - The *size of the cache* (the larger, the better).
- **Placement Principle:** As a general rule, the *higher up in the call stack* caching is implemented, the more resources can be saved downstream. This is why client-side HTTP caching was discussed early on.
- **Important Caveat:** Caching is an **optimization**. A system does not have

a scalable architecture if the origin (e.g., the data store) *cannot withstand the load without the cache* fronting it. If access patterns change leading to cache misses, or if the cache becomes unavailable, the application should *not fall over* (though it's acceptable for it to become slower).

2. Caching Policies

A. Cache Miss Handling (How missing objects are fetched)

When a cache miss occurs, the missing object must be requested from the origin. This can happen in two main ways:

- **Side Cache (or Cache-Aside):**
  - The application first requests the object from the cache.
  - If the cache returns an "object-not-found" error (cache miss), the application then requests the object directly from the *origin* (e.g., database).
  - After retrieving the object from the origin, the application *updates the cache* with this object.
  - In this pattern, the cache is typically treated as a *key-value store* by the application.
- **Inline Cache (e.g., Read-Through/Write-Through):**
  - The cache itself communicates *directly with the origin.*
  - When a miss occurs, the cache requests the missing object from the origin *on behalf of the application.*
  - The application *only ever interacts with the cache*, not the origin directly for cached data.
  - HTTP caching is an example of an inline cache mechanism.

B. Eviction Policy

- **Necessity:** Because a cache has limited capacity, one or more existing entries may need to be *evicted* to make room for new entries when the cache is full.
- **Determinants:** The choice of which entry to remove depends on the *eviction policy* used by the cache and the *access patterns* of the objects.
- **Example Policy:** A commonly used policy is **Least Recently Used (LRU)**, which evicts the entry that hasn't been accessed for the longest time.

C. Expiration Policy

- **Purpose:** Dictates when an object should be considered stale and potentially evicted from the cache.
- **Mechanism:** Often implemented using a **Time-To-Live (TTL)** for each cached object.
- **Behavior:** When an object has been in the cache for longer than its TTL, it *expires* and can be safely evicted.
- **Trade-off:**
  - *Longer TTLs* generally lead to a *higher hit ratio.*
  - However, they also increase the likelihood of serving *stale and inconsistent data* (data that has changed in the origin but not yet in the cache).
- **Deferred Expiration:** Expiration doesn't need to occur immediately when the TTL is reached. It can be *deferred to the next time the entry is requested.*
  - **Resilience Benefit:** If the origin (e.g., a data store) is temporarily unavailable, it might be more resilient to return an object with an expired TTL to the application rather than an error.

D. Cache Invalidation

- **Complexity:** True cache invalidation (actively removing or marking entries as invalid when the underlying data changes in the origin) is *very hard to implement correctly* in practice.
- **TTL as a Workaround:** An expiry policy based on TTL is often used as a simpler workaround for the complexities of direct cache invalidation.
- **Example Challenge:** If you cache the result of a database query, *every time any data touched by that query changes* (which could span thousands of records or more), the cached result would need to be invalidated.

3. Local Cache (In-Process Cache)

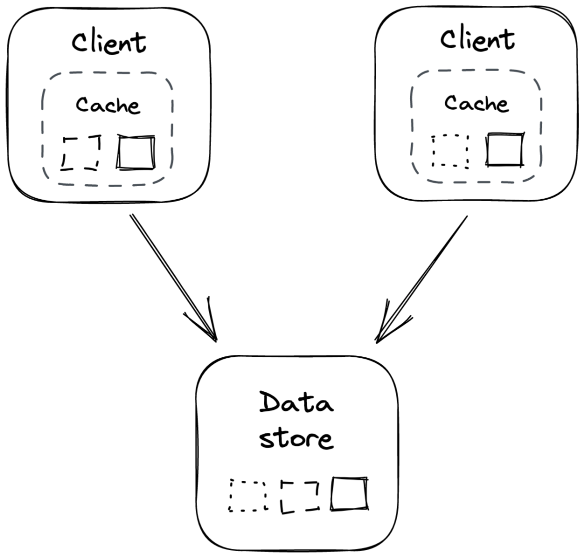

Figure 1: In-process cache

- **Implementation:** Simplest way to implement a cache is to *co-locate it with the client* (i.e., within the application process).
- **Examples:**
  - A simple *in-memory hash table* within the application.
  - An *embeddable key-value store*, like RocksDB.

- **Drawbacks:**
  - **Resource Waste (Duplication):** Since each client cache is independent, the *same objects are often duplicated* across many caches. For example, if every client has a local cache of 1GB, then no matter how many clients there are, the total *effective* size of unique cached data is at most 1GB.
  - **Consistency Issues:** Different clients might see *different versions* of the same object due to independent caching.
  - **Increased Origin Load with Scale:** As the number of clients grows, the number of requests to the origin (for cache misses or initial population) increases proportionally.
  - **Thundering Herd Effect:**
    - This issue is exacerbated when clients restart, or new ones come online, as their caches need to be *populated from scratch*, potentially overwhelming the origin.
    - Can also occur when a specific object that wasn't accessed before (and thus not cached) *becomes popular all of a sudden*, leading to many clients requesting it from the origin simultaneously.
  - **Mitigating Thundering Herd:** Clients can implement request *coalescing* for the same object. The idea is that, at any given time, there should be at most *one outstanding request per client* to fetch a specific missing object from the origin.

4. External Cache (Out-of-Process Cache)

- **Definition:** An external cache is a *service dedicated to caching objects*, typically in memory, separate from the application processes.

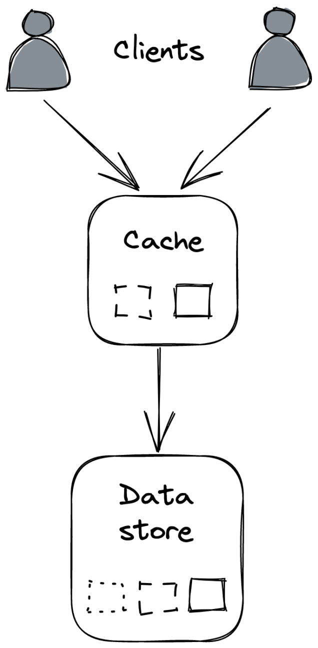

Figure 2: Out-of-process cache

- **Shared Nature:** It's *shared across multiple clients*, addressing some drawbacks of local caches, but introduces its own complexity and cost.
- **Examples:** Popular caching services include *Redis* or *Memcached.* These are also available as managed services on cloud platforms like AWS and Azure.
- **Scalability of External Cache:** Unlike a local cache, an external cache service can increase its *throughput and storage capacity* using techniques like **replication** and **partitioning**.
  - For example, Redis can automatically partition data across multiple nodes and replicate each partition using a leader-follower protocol.
- **Benefits over Local Cache:**
  - **Reduced Consistency Issues:** Since the cache is shared, there is typically only a *single version* of each object at any given time (assuming the external cache itself is not replicated in a way that introduces inconsistencies, or that it manages consistency across its replicas).
  - **Stable Origin Load:** The number of times an object is requested from

    the origin *doesn't grow directly* with the number of clients (as many clients will hit the shared cache).

- **Drawbacks & Considerations:**
  - **Load Shifts to Cache:** The load doesn't disappear; it merely shifts from the origin to the external cache. Therefore, the *cache itself will eventually need to be scaled out* if the load continues to increase.
  - **Rebalancing Data on Scale-Out:** When scaling out the cache (e.g., adding more cache nodes), as little data as possible should be moved around or dropped to avoid degrading cache performance or significantly dropping the hit ratio. Techniques like *consistent hashing* can help reduce the amount of data that needs to be shuffled during rebalancing.
  - **Maintenance Cost:** It's *another service* that needs to be operated, monitored, and maintained.
  - **Higher Latency (vs. Local):** Accessing an external cache requires a *network call*, which has higher latency than accessing an in-process local cache.
  - **Handling External Cache Unavailability:**
    - What should clients do if the external cache is down?
    - Simply bypassing the cache and hitting the origin directly might seem okay temporarily.
    - **Danger:** The origin might *not be prepared to withstand the sudden surge of traffic* that was previously absorbed by the cache.
    - Consequently, the external cache becoming unavailable could cause a *cascading failure*, resulting in the origin becoming unavailable as well.
  - **Defense Strategies:**

    * Clients could use an *in-process (local) cache as a defense* or fall-back when the external cache is unavailable (a multi-layer caching approach).
    * The origin system also needs to be prepared to handle these sudden "attacks" or surges by, for example, *shedding requests* (gracefully degrading performance rather than failing completely).
  - **Fundamental Reminder:** Caching is an optimization. The system needs to be able to *survive without the cache*, albeit potentially at the cost of being slower.