Cache

Caching – System Design

Caching means storing frequently used data in a place where it can be accessed very fast.

The main goal of caching is to improve performance and reduce response time by avoiding repeated database calls.

A cache works like a temporary local store.
Getting data from cache is much faster than fetching it again and again from the database.

In a normal web application:

- **We keep a cache near the application server**
- **Often we use in-memory stores like Redis or Memcached**

1.How Cache Works

Twitter example:

- **When a tweet goes viral, millions of users request the same tweet**
- **If every request hits the database → database becomes slow**
- **So the tweet is stored in cache**
- **Users get the tweet directly from cache, very fast**
- **Database load is reduced**

2.How Does Cache Work?

In a web application, data is stored in a database.
Reading data from the database takes time because it involves network calls and I/O operations.

Cache helps by reducing database calls and making the system faster.

Step-by-step Flow

1. **First request**
   - **User requests some data**
   - **Data is not in cache**
   - **Application fetches data from the database**
   - **This is called a cache miss**
   - **The result is saved in cache**
   - **Data is sent to the user**
2. **Second request (same data)**
   - **Application checks cache first**
   - **Data is found in cache**

- This is called a cache hit
- Data is returned directly from cache

Result

- First request → slow
- Second request → much faster
- Database load → reduced

Cache stores data after the first request so future requests are served faster without hitting the database.

3.Why Don't We Store All Data in Cache?

Cache has many benefits, but we cannot store all data in cache. There are several reasons for this:

1. **Cache hardware is expensive**
   Cache (RAM / in-memory storage) costs much more than normal database storage.
2. **Search time can increase**
   If too much data is stored in cache, finding the required data can become slower.
3. **Cache is volatile**
   Cache data is lost when the system crashes or restarts.
   Important and long-term data must not be stored only in cache.
4. **Limited size**
   Cache has limited memory, so it cannot hold everything.

Types of Cache (System Design)

1. Client-side Cache

- Stored on the user's device or browser
- Examples:
  - Browser cache
  - Cookies
  - LocalStorage
- Reduces repeated requests to the server

2. CDN Cache (Content Delivery Network)

- Cache stored at servers near the user
- Used for:
  - Images
  - Videos
  - CSS / JS files
- Makes websites load faster globally

3. Application-level Cache

- Cache stored inside or near the application server
- Used for:
  - Frequently accessed API responses
- Example:
  - In-memory cache in the app

4. Distributed Cache

- Cache shared across multiple servers
- Very fast and scalable
- Examples:
  - Redis
  - Memcached
- Most common in system design interviews

5. Database Cache

- Cache built into the database itself
- Stores frequently accessed query results
- Example:
  - MySQL query cache (conceptually)

These are classic caching strategies in system design

1. Cache-Aside (Lazy Loading)

Most commonly used strategy

How it works (Read flow):

1.App checks cache first

2.If data not found → cache miss

3.App fetches data from database

4.App stores data in cache

5.Returns data to user

Write flow:

- App writes data directly to database
- Cache is updated or invalidated

Pros

- Simple to implement
- Cache contains only frequently used data

- Database remains source of truth

Cons

- First request is slow (cache miss)
- Risk of stale data if cache not updated properly
- App handles cache logic (more responsibility)
- for every new data there will always be cache miss

Use when:

- Read-heavy systems (Twitter, Instagram feeds)
- Data doesn't change very frequently

2. Read-Through Cache

How it works:

1.App requests data from cache

2.If cache miss → cache itself fetches from DB

3.Cache stores data

4.Cache returns data to app

App never talks to DB directly for reads

Pros

- Cleaner application code
- Cache logic centralized
- Easy for developers

Cons

- Cache becomes complex
- Slower cache miss (extra hop)
- Less control for app

Use when:

- Want simpler app logic
- Managed cache services

3. Write-Through Cache

How it works (Write flow):

1.App writes data to cache

2.Cache writes data to database

3.Write is successful only after DB write

Read flow:

- Reads come directly from cache

Pros

- Cache always consistent with DB
- No stale data
- Simple read logic

Cons

- Write latency is higher
- Unnecessary writes for rarely-read data
- Cache can get polluted
- slow
- 2 phase commit

Use when:

- Data consistency is very important
- Read-after-write scenarios

4. Write-Back (Write-Behind) Cache

How it works:

1.App writes data to cache only

2.Cache responds immediately

3.Cache writes data to DB asynchronously (later)

Pros

- Very fast writes
- High performance
- Reduced DB load

Cons

- Risk of data loss if cache crashes
- Data inconsistency possible

- More complex implementation

Use when:

- High write throughput needed
- Slight data loss is acceptable
- Analytics, counters, logs

5. Write-Around Cache

How it works:

1.Writes go directly to database

2.Cache is not updated on write

3.Cache is filled only on read

Pros

- Cache stays clean (only read-heavy data)
- No unnecessary cache writes
- Lower cache memory usage

Cons

- Cache miss after every write
- First read after write is slow
- Not good for read-after-write use cases

Use when:

- Write-heavy systems
- Data rarely read after being written

Cache Eviction Policies

1.LRU (Least Recently Used)

Messages:

A → B → C

New message D comes
A is removed

Final cache:

D,B, C,

Pros

- Keeps recently used messages
- Good hit rate
- Very popular in real systems

Cons

- Needs extra memory to track order
- Slightly complex to implement

2.MRU (Most Recently Used)

Messages:

A → B → C

New message D comes
C is removed

Final cache:

A, B, D

Pros

- Useful when recent data is rarely reused
- Simple logic

Cons

- Poor performance in general cases
- Rarely used in practice

3.LFU (Least Frequently Used)

Usage frquency:

A(5), B(2), C(1)

New message D comes
C is removed

Final cache:

A, B, D

Pros

- Keeps most popular messages

- Good for trending data

Cons

- Old popular data may stay forever
- Complex to implement

4.FIFO (First In First Out)

Insert order:

A → B → C

New message D comes
A is removed

Final cache:

B, C, D

Pros

- Very easy to implement
- Low overhead

Cons

- Ignores usage pattern
- Frequently used data may be removed

5.Random Eviction

Cache:

A, B, C

New message D comes
Randomly one message removed (say B)

Final cache:

A, C, D

Pros

- Very simple
- No tracking required

Cons

- **Unpredictable**
- **Low cache hit ratio**