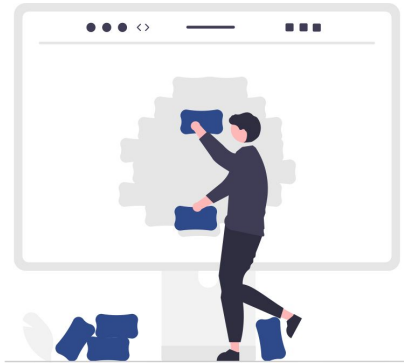
Episode 4

System Design for Noobs

August 20, 2022



Core topics covered in this slide:

1. Caching
2. Designing a URL shortening service

FAQ Recap

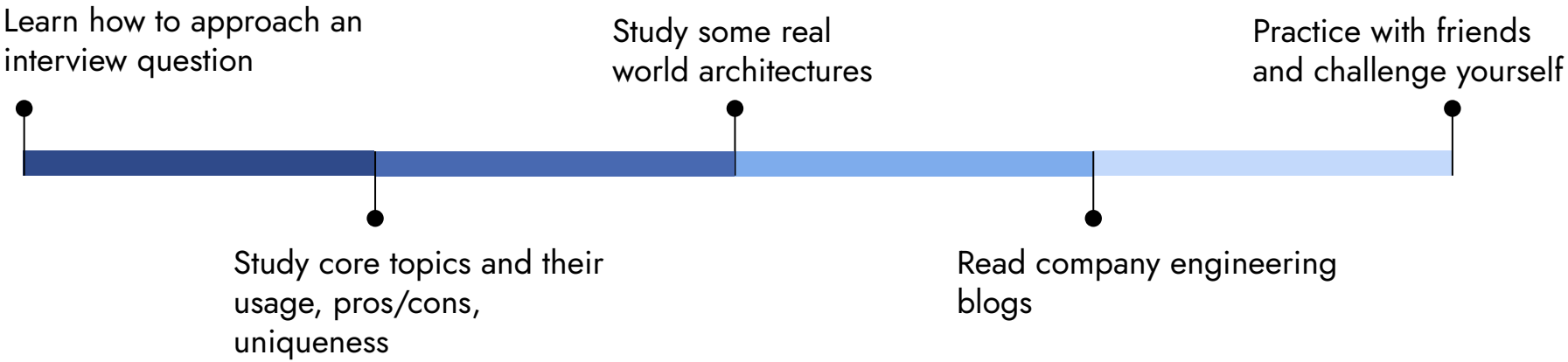
Do I need to know everything?

Absolutely f*cking **NOT!**

System Design questions mainly depends on some variables:

- Your experience
- Your technical background
- Applied company
- Applied position
- Luck!

The Plan


Learn how to approach an interview question
Study core topics and their usage, pros/cons, uniqueness
Study some real world architectures
Read company engineering blogs
Practice with friends and challenge yourself

The Approach

Step 1: Outline use cases, constraints, and assumptions

- Who is going to use it?
- How are they going to use it?
- How many users are there?
- What does the system do?
- What are the inputs and outputs of the system?
- How much data do we expect to handle?
- How many requests per second do we expect?
- What is the expected read to write ratio?

Step 2: Create a HLD (High Level Design)

- Draw the **core components** and their connections
- Question and justify your selections
- Specify the read/write scenarios

Step 3: Design the core components

No fixed formula.

Example:

- API design
- Database schemas
- Encryption

Step 4: Scale the system

- Load balancers
- Horizontal/Vertical scaling
- Caching
- Reverse proxy?
- DB replication
- Discuss trade-offs

A very good template:

https://leetcode.com/discuss/career/229177/My-System-Design-Template

Caching

Types of Caching

- Client
- CDN
- Web server
- Database
- Application
  - DB query level
  - Object level

Client Caching

Caches can be located on the client side (OS or browser), or in a distinct cache layer.

CDN Caching

Content delivery networks.

Whole purpose is to cache. (Discussed on Episode 1)

Web Server Caching

Reverse proxies can serve static and dynamic content directly.

Web servers can also cache requests, returning responses without having to contact application servers.

(Discussed in detail on Episode 2)

Database Caching

Database usually includes some level of caching in a default configuration, optimized for a generic use case.

Tweaking these settings for specific usage patterns can further boost performance.

Read this article from AWS:
https://docs.aws.amazon.com/whitepapers/latest/database-caching-strategies-using-redis/types-of-database-caching.html

In-memory Caching

In-memory caches such as Memcached and Redis are key-value stores between your application and your data storage.

Reading data from memory is faster than from the disk. In-memory caching avoids latency and improves online application performance.

Background jobs that took hours or minutes can now execute in minutes or seconds.

What to Cache?

There are multiple levels you can cache that fall into two general categories: **database queries** and **objects**:

- Row level
- Query-level
- Fully-formed serializable objects
- Fully-rendered HTML

AVOID file-based caching. It makes cloning and scaling difficult so not a good system design.

How to Cache?

1. Caching at the database query level
2. Caching at the object level

Query Level Caching

Whenever you query the database, *hash the query as a key* and store the result to the cache.

Issues:

- If one piece of data changes such as a table cell, you need to delete all cached queries that might include the changed cell

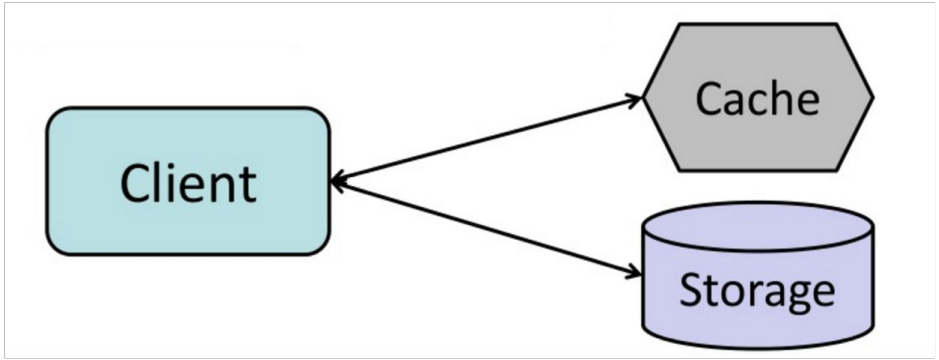
Object Level Caching

See your data as an object, similar to what you do with your application code. Have your application assemble the dataset from the database into a class instance or a data structure(s).

Object caching is a kind of server-side caching. That means that the cache is stored on the server, and cached queries are served from there.

When to update the cache?

Cache-aside



The cache does not interact with storage directly.

The application does the following:

- Look for entry in cache, resulting in a cache miss
- Load entry from the database
- Add entry to cache
- Return entry

Cache-aside

```
def get_user(self, user_id):
    user = cache.get("user.{0}", user_id)
    if user is None:
        user = db.query("SELECT * FROM users WHERE user_id = {0}", user_id)
        if user is not None:
            key = "user.{0}".format(user_id)
            cache.set(key, json.dumps(user))
    return user
```

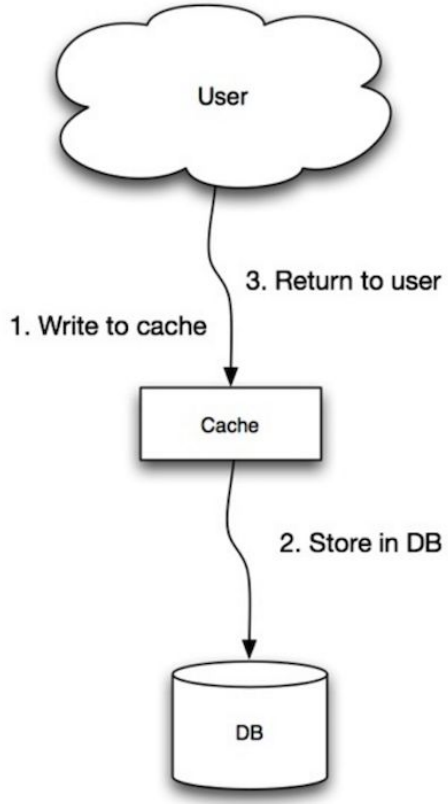
Cache-aside Cons

- Each cache miss results in three trips, which can cause a noticeable delay.
- Data can become stale if it is updated in the database. This issue is mitigated by setting a time-to-live (TTL) which forces an update of the cache entry, or by using write-through.
- When a node fails, it is replaced by a new, empty node, increasing latency.

Write-through

The application uses the cache as the main data store, reading and writing data to it, while the cache is responsible for reading and writing to the database:

- Application adds/updates entry in cache
- Cache synchronously writes entry to data store
- Return



Write-through

```
## Application code
set_user(12345, {"foo":"bar"})


## Cache code
def set_user(user_id, values):
    user = db.query("UPDATE Users WHERE id = {0}", user_id, values)
    cache.set(user_id, user)
```

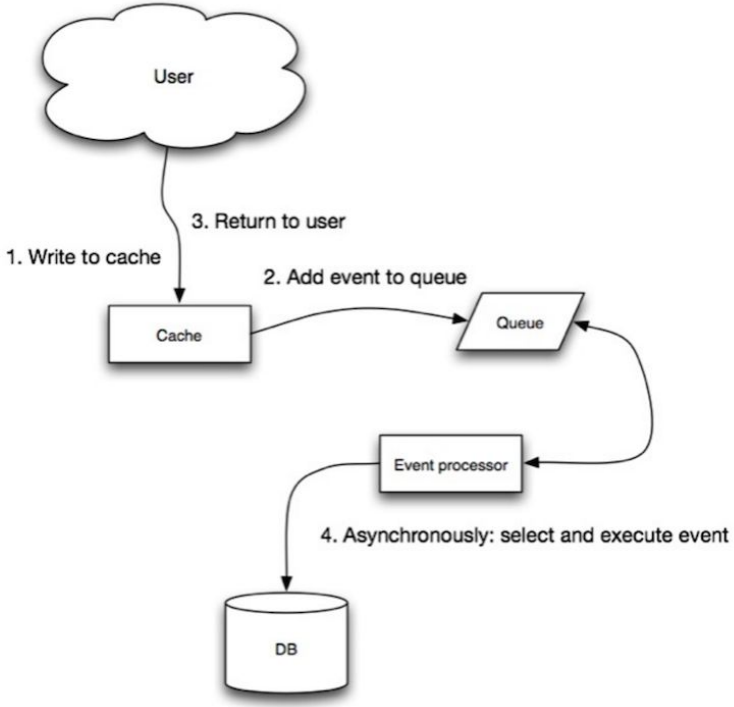
Write-through Cons

- When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database. Cache-aside in conjunction with write through can mitigate this issue.
- Most data written might never be read, which can be minimized with a TTL.

Write-behind

The application does the following:

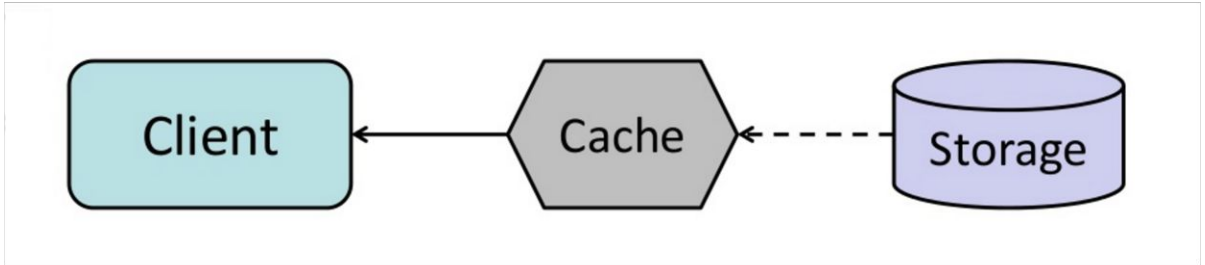
- Add/update entry in cache
- Asynchronously write entry to the data store, improving write performance



Write-behind Cons

- There could be data loss if the cache goes down prior to its contents hitting the data store.
- It is more complex to implement write-behind than it is to implement cache-aside or write-through.

Refresh-ahead



Implementations of Refresh-ahead are cache provider-dependent.

But the goal is to configure the cache to automatically refresh any recently accessed cache entry prior to its expiration.

Refresh-ahead can result in reduced latency vs read-through if the cache can accurately predict which items are likely to be needed in the future.

Refresh-ahead Cons

- Not accurately predicting which items are likely to be needed in the future can result in reduced performance than without refresh-ahead.

Important reads:

- https://hazelcast.com/blog/a-hitchhikers-guide-to-caching-patterns/
- https://www.slideshare.net/tmatyashovsky/from-cache-to-in-memory-data-grid-introduction-to-hazelcast

Designing a URL shortening service

Step 1: Outline use cases, constraints, and assumptions

- Validity of URL - Tanmoy
- User modifies URL or not (Cache invalidation) - Tanmoy
- User base assumption - Tanmoy
- Web client - Tanmoy
- URL generation cap for users? - Saad
- Service deleting expired pastes - Saad
- User not auth needed? - Adel
- Custom URL generation and input constraints - Tanmoy
- Pre-generate URLs? - Saad

Step 1: Outline use cases, constraints, and assumptions

- 10 million users
- 100 million writes/month
- 1 billion reads/month
- 10:1 read to write ratio
- 1.27 kB per paste (shortlink - 7 bytes, expiration_length_in_minutes - 4 bytes, created_at - 5 bytes, paste_path - 255 bytes, total = ~1.27 KB)
- Total data per month: 100million * 1.27kB

Step 2: Create a HLD (High Level Design)

- Draw the **core components** and their connections
- Question and justify your selections
- Specify the read/write scenarios

Step 3: Design the core components

No fixed formula.

Step 4: Scale the system

- Load balancers
- Horizontal/Vertical scaling
- Caching
- Reverse proxy?
- DB replication
- Discuss trade-offs

Important design solutions

https://tianpan.co/notes/2016-02-13-crack-the-system-design-interview

Thank you.

Do you have any questions?

azmainadel47@gmail.com
Direct/WhatsApp: +880 1684 723252