<https://github.com/andreis/interview>

System Design

1. Brainstorming

- What is the system for

- Who is using the system

- What are the functions of the system

- How many users do we expect this system to handle? Normally millions and billions

- How heavy are some of the components are. For example user connections

Tips:

- do not take the example as a whole think big

2. Different approaches: requirements and performance

- Give solutions and estimate the memory and time complexity

- Give the trade-offs and go beyond your solution

3. Implementation: apis, data structures

Twitter problem

1. Brainstorming: Scalibility and graph size

- How many users do we expect this system to handle? 10 million or 100 million requests per day

- For the following, how many users on average can be followed? 200 other users on average, some users with tens of thousands of followers

- How many requests of posting tweets and like being happened? maximum of 10 million tweets per day and each tweets will be favorited twice on average

- What availability is expected and what response times are tolerable? Load pretty quickly, online system

- Calculation:

We will have around 10 million users. Average number of followed other users is 200. This means that the network of users will have about 200 \* 10 million edges.

This makes 2 billion edges. If the average number of tweets per day is 10 million the number of favorites will then be 20 million.

- Summarize:

10 million users

10 million tweets per day

20 million tweet favorites per day

100 million HTTP requests to the site

2 billion “follow” relations

Some users and tweets could generate an extraordinary amount of traffic

2. Approaches:

- High-level design

1) Logic part

Posting new tweets - writing database POST

Updating tweets - updating database PUT

Following a user - writing database POST

Unfollowing a user - writing database DELETE

Favoriting a tweet - writing database POST

Unfavoriting a tweet - writing databse DELETE

Displaying data about users and tweets - reading database GET

> Describe the UI for implementing the logic functions with a small graph

> Back-end handling user requests

The app will receive around a few thousand requests per second at times.

The complexity of the requests that the app receives, one simple query, a few heavier queries, some CPU-intensive computations performed.

The technologies used to implement the application using less memory. Should talk about what kind of load can be handled by a single machine for sure and what is load that definitely needs more computing power. Different web frameworks, hosting services and so on.

Consider scaling up or out such as getting a better and more expensive server to handle the expected load. Scaling out would involve designing your architecture in a way that spreads the computations over a number of machines and distributes the load across them.

Consider the crash on the machine and performance. Use a load balancer, which handles initial traffic and sends requests to a set of servers running one or more instances of the application. Add new server based on needs and capacity.

\* Load balancing

<https://en.wikipedia.org/wiki/Load_balancing_(computing)>

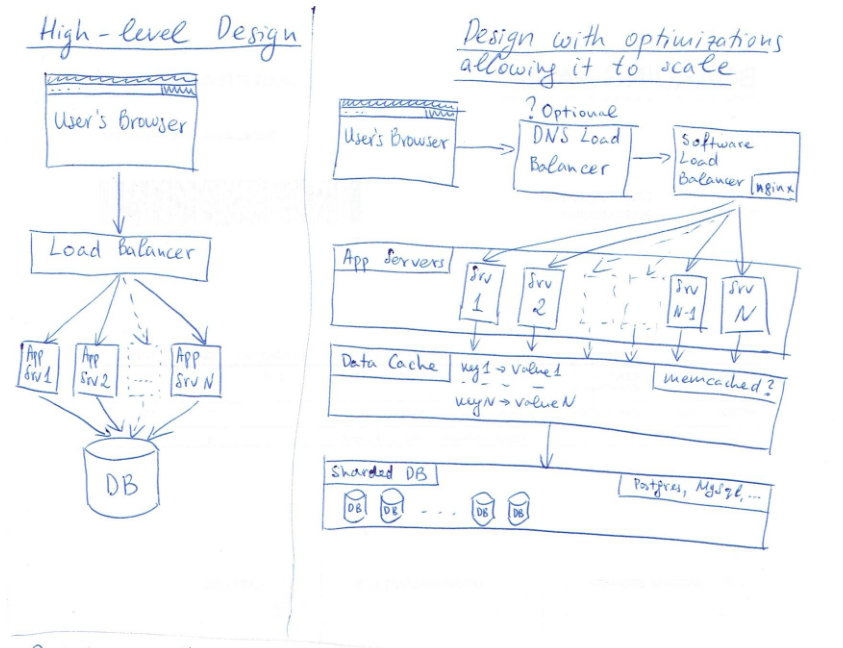
<https://aws.amazon.com/elasticloadbalancing/>

<http://nginx.org/en/docs/http/load_balancing.html>

<http://www.haproxy.org/>

The load balancing routes requests to the servers using some predefined logic and the application servers are able to understand the requests and return the proper data back to the browser.

Could mention about the frameworks and their benefits and drawbacks. NodeJS, Ruby on Rails, AngularJS, EmberJS, ReachJS, etc.



2) Data storage part

Discussion what should be stored. Objects and relationships.

Discussion the size of data to be stored.

10 M tweets \* 365 = 3.65 B tweets – 10 B tweets in database \* 140 char/tweet = 1.4 Trillion char

2 B user and favorite connections \* 4-byte integer ID = 16 GB

20 M growing in favorite/day \* 365 = 7.3 B/year – 20 B connections between user and favorite in database \* (4-byte ID + 8-byte favorite tweet ID) = 240 GB

The expecting size of the data to store is around 2.6-2.7 terabytes

Need give some solutions about how to handle large data in relational database like MySQL or Postgres

<http://highscalability.com/blog/2011/12/19/how-twitter-stores-250-million-tweets-a-day-using-mysql.html>

<https://gigaom.com/2011/12/06/facebook-shares-some-secrets-on-making-mysql-scale/>

<http://webscalesql.org/>

<https://github.com/twitter-forks/mysql>

<https://github.com/facebook/mysql-5.6>

Caching solution stands in front of the database server. Some popular tool like Memcached should be discussed. Benefits of caching solution should be described. A database stores data on disk and it is much slower to read from disk that from memory. Memcached stores data in memory, which provides faster access. Although database has its own caching mechanisms but with Memcached we have better control over what gets cached and how. It is better to compare the speeds to read from hard drive and from RAM to CPU cache especially SSD.

Add proper indexes in the database for executing quick queries joining tables. Also think about partitioning the data in some way, which can improve the read/write speeds.

Scaling the database: vertical and horizonal.

- low-level issues

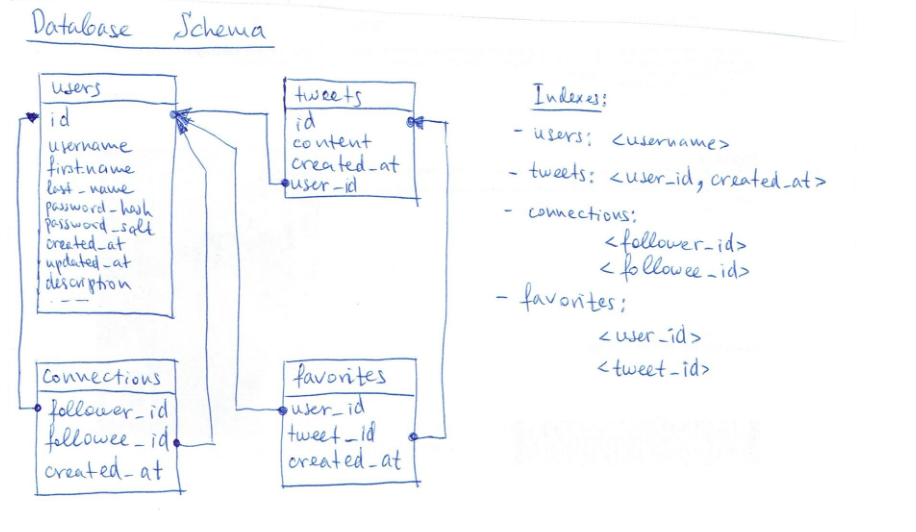
1) Database Schema

Two main entities: users and tweets

Relationships: connections/follows (many-to-many) and favorites (many-to-many)

Indexes: for all the entities tables and relation tables describe the reasons and what are the common queries going to be performed.

The drawback of indexes is it makes the write slower but make the read very fast.



2) RESTful APIs: sending and returning JSON objects

Will need the polices between the http call and the backend checking the authentication and validation

> fetch the profile details: GET /api/users/user\_id

> get the tweets for a user with pagenation: GET /api/users/user\_id/tweets?query

> get the users following a given user and followed by that user: GET /api/users/user\_id/followers and GET /api/users/user\_id/followees

> posting new tweets: POST /api/users/user\_id/tweets

> updating tweets: PUT /api/tweets/tweet\_id

> deleting tweets: DELETE /api/tweets/tweet\_id

> following a given user: POST /api/users/user\_id/followers

> seeing a list of all users that favorited a tweet: GET /api/users/user\_id/tweets/tweet\_id/favorites

> favoriting a tweet: POST /api/users/user\_id/tweets/tweet\_id/favorites

3. Bottlenecks and trade-offs

- Increase number of read requests

Database will be the first bottleneck and become overwhelmed.

Solution: replication with multiple database instances; Shard database and spread the data across different machines. Scaling a relational database.

Web application is the second bottleneck with slow response time.

Solution: load balancer can add more machines running the application and send requests to these machines.

The load balancer can also be a bottleneck with too many requests and hard to send to different machine.

Solution: additional load balancing using DNS and direct requests for our domain to different machine.

- Scaling the database

Partitioning this data and storing it on separate servers to increase availability and spread the load by sharing our data.

<https://instagram-engineering.com/sharding-ids-at-instagram-1cf5a71e5a5c>

<http://www.databasesoup.com/2012/04/sharding-postgres-with-instagram.html>

<http://code.flickr.net/2010/02/08/ticket-servers-distributed-unique-primary-keys-on-the-cheap/>

- Unexpecting traffic using cache and auto-scaling of the available computing nodes

The Summarization Problem

Design a system that uses our existing library and allows users to submit text articles