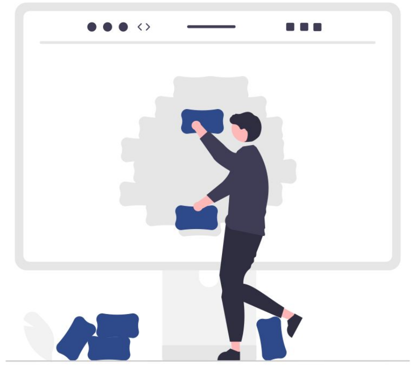
Episode 3

System Design for Noobs

August 9, 2022



Core topics covered in this slide:

1. SQL
    a. Scaling
    b. Federation
    c. Sharding
    d. Denormalization
    e. SQL Tuning
2. NoSQL
3. SQL v NoSQL
4. Designing a URL shortening service

FAQ Recap

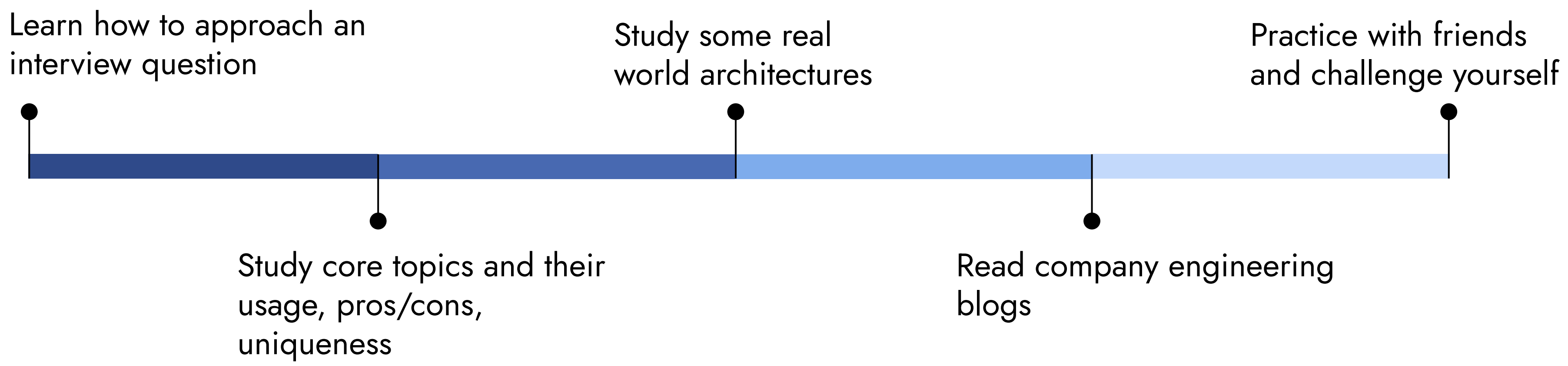
Do I need to know everything?

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

SQL

ACID

- Atomicity - Each transaction is all or nothing.
- Consistency - Any transaction will bring the database from one valid state to another.
- Isolation - Executing transactions concurrently has the same results as if the transactions were executed serially.
- Durability - Once a transaction has been committed, it will remain so.

Scaling a RDBMS

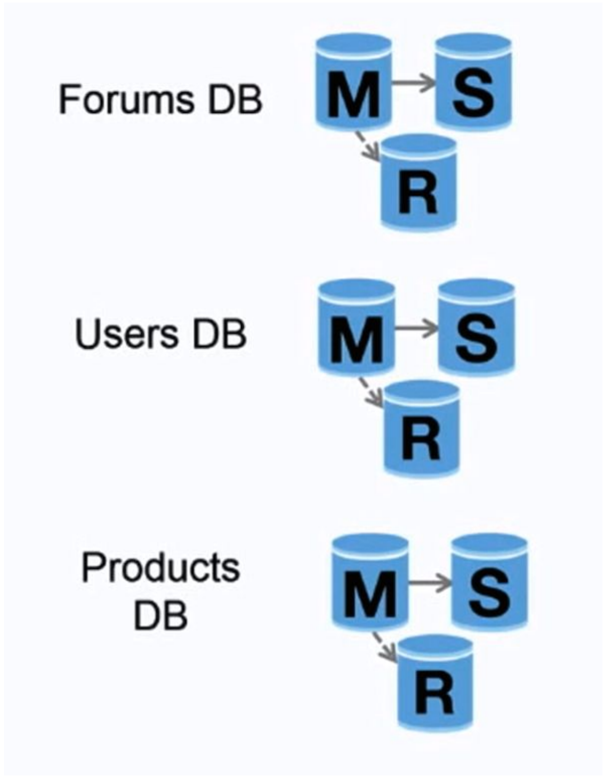
There are many techniques to scale a relational database:

master-slave replication, master-master replication, federation, sharding, denormalization, and SQL tuning.

Federation

Federation (or functional partitioning) splits up databases by function.

For example, instead of a single, monolithic database, you could have three databases: forums, users, and products, resulting in less read and write traffic to each database and therefore less replication lag. Smaller databases result in more data that can fit in memory, which in turn results in more cache hits due to improved cache locality. With no single central master serializing writes you can write in parallel, increasing throughput.



Good video:

- AWS re:Invent 2019: Scaling up to your first 10 million users (ARC211-R) https://www.youtube.com/watch?v=kKjm4ehYiMs

Federation Cons

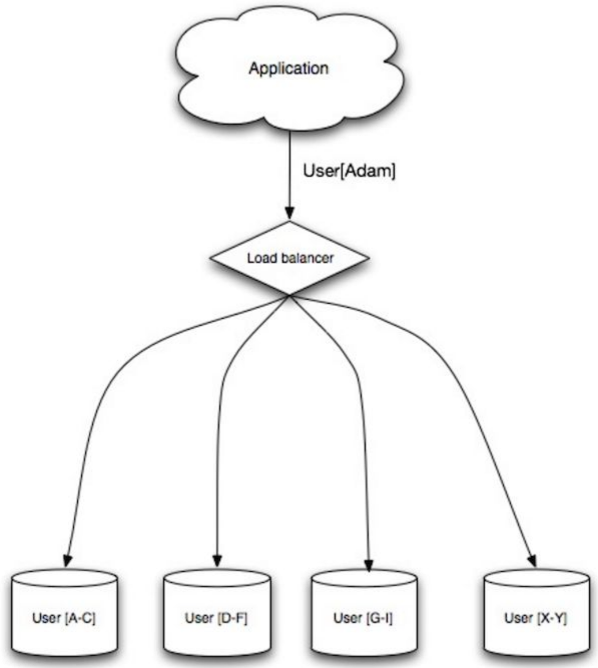
- Federation is not effective if your schema requires huge functions or tables.
- You'll need to update your application logic to determine which database to read and write.
- Joining data from two databases is more complex with a server link.
- Federation adds more hardware and additional complexity.

Sharding

Sharding distributes data across different databases such that each database can only manage a subset of the data.

Taking a users database as an example, as the number of users increases, more shards are added to the cluster.

http://highscalability.com/blog/2009/8/6/an-unorthodox-approach-to-database-design-the-coming-of-the.html



Sharding Pros

- Similar to the advantages of federation, sharding results in less read and write traffic, less replication, and more cache hits.
- Index size is also reduced, which generally improves performance with faster queries.
- If one shard goes down, the other shards are still operational, although you'll want to add some form of replication to avoid data loss.
- Like federation, there is no single central master serializing writes, allowing you to write in parallel with increased throughput.

Sharding Cons

- You'll need to update your application logic to work with shards, which could result in complex SQL queries.
- Data distribution can become lopsided in a shard. For example, a set of power users on a shard could result in increased load to that shard compared to others.
- Rebalancing adds additional complexity. A sharding function based on consistent hashing can reduce the amount of transferred data.
- Joining data from multiple shards is more complex.
- Sharding adds more hardware and additional complexity.

Denormalization

Denormalization attempts to improve read performance at the expense of some write performance. Redundant copies of the data are written in multiple tables to avoid expensive joins.

Some RDBMS such as PostgreSQL and Oracle support materialized views which handle the work of storing redundant information and keeping redundant copies consistent.

Denormalization Pros

- Once data becomes distributed with techniques such as federation and sharding, managing joins across data centers further increases complexity. Denormalization might circumvent the need for such complex joins.
- In most systems, reads can heavily outnumber writes 100:1 or even 1000:1. A read resulting in a complex database join can be very expensive, spending a significant amount of time on disk operations.

Denormalization Cons

- Data is duplicated which requires more storage.
- Constraints can help redundant copies of information stay in sync, which increases complexity of the database design.
- A denormalized database under heavy write load might perform worse than its normalized counterpart.
- Denormalization can make *update* and *insert* code harder to write.

SQL Tuning

This is a broad topic. The main goal is to keep your SQL schemas/queries optimized

Tighten up the schema

- Use CHAR instead of VARCHAR for fixed-length fields.
- CHAR effectively allows for fast, random access, whereas with VARCHAR, you must find the end of a string before moving onto the next one.
- Use TEXT for large blocks of text such as blog posts. TEXT also allows for boolean searches. Using a TEXT field results in storing a pointer on disk that is used to locate the text block.
- Use INT for larger numbers up to 2^32 or 4 billion.
- Use DECIMAL for currency to avoid floating point representation errors.
- Avoid storing large BLOBS, store the location of where to get the object instead.
- VARCHAR(255) is the largest number of characters that can be counted in an 8 bit number, often maximizing the use of a byte in some RDBMS.
- Set the NOT NULL constraint where applicable to improve search performance.

Use good indices

- Columns that you are querying (SELECT, GROUP BY, ORDER BY, JOIN) could be faster with indices.
- Indices are usually represented as self-balancing B-tree that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
- Placing an index can keep the data in memory, requiring more space.
- Writes could also be slower since the index also needs to be updated.
- When loading large amounts of data, it might be faster to disable indices, load the data, then rebuild the indices.

Avoid expensive joins

- Denormalize where performance demands it.
-

Partition tables

- Break up a table by putting hot spots in a separate table to help keep it in memory.

Tune the query cache

- In some cases, the query cache could lead to performance issues.

NoSQL

NoSQL is a collection of data items represented in a **key-value store, document store, wide column store, or a graph database.**

Data is denormalized, and joins are generally done in the application code.

Most NoSQL stores lack true ACID transactions and favor eventual consistency.

BASE

- Basically available - the system guarantees availability.
- Soft state - the state of the system may change over time, even without input.
- Eventual consistency - the system will become consistent over a period of time, given that the system doesn't receive input during that period.

In comparison with the CAP Theorem, BASE chooses availability over consistency.

Key-value Store (hash table)

Key-value stores provide high performance and are often used for simple data models or for rapidly-changing data, such as an in-memory cache layer.

Since they offer only a limited set of operations, complexity is shifted to the application layer if additional operations are needed.

A key-value store is the basis for more complex systems such as a document store, and in some cases, a graph database.

Document Store (key-value store with documents stored as values)

A document store is centered around documents (*XML*, *JSON*, *binary*, etc), where a document stores all information for a given object.

Document stores provide APIs or a query language to query based on the internal structure of the document itself.

Based on the underlying implementation, documents are organized by collections, tags, metadata, or directories. Although documents can be organized or grouped together, documents may have fields that are completely different from each other.

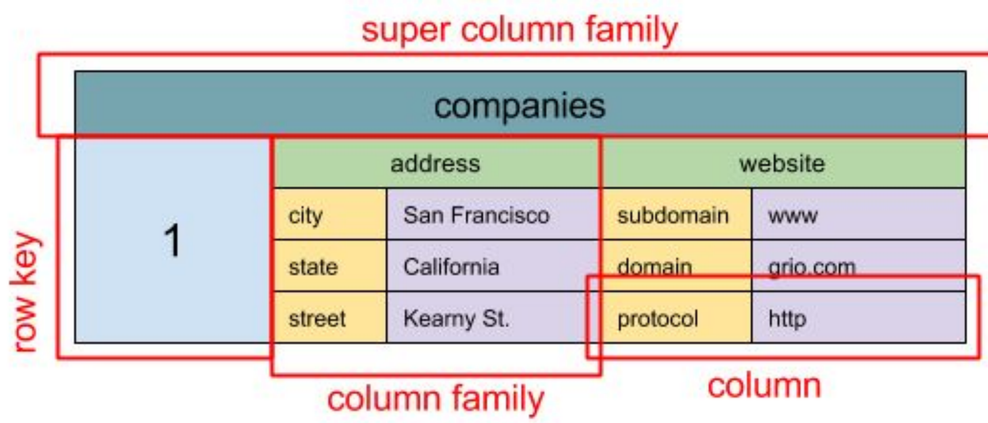
Important reads:

- https://www.mongodb.com/mongodb-architecture
- https://blog.couchdb.org/2016/08/01/couchdb-2-0-architecture/
- https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up

Wide Column Store (nested map ColumnFamily<RowKey, Columns<ColKey, Value, Timestamp>>)

A wide column store's basic unit of data is a column (name/value pair). A column can be grouped in column families (analogous to a SQL table).

Super column families further group column families. You can access each column independently with a row key, and columns with the same row key form a row. Each value contains a timestamp for versioning and for conflict resolution.



Wide Column Store

Google introduced Bigtable as the first wide column store, which influenced the open-source HBase often-used in the Hadoop ecosystem, and Cassandra from Facebook.

http://www.read.seas.harvard.edu/~kohler/class/cs239-w08/chang06bigtable.pdf

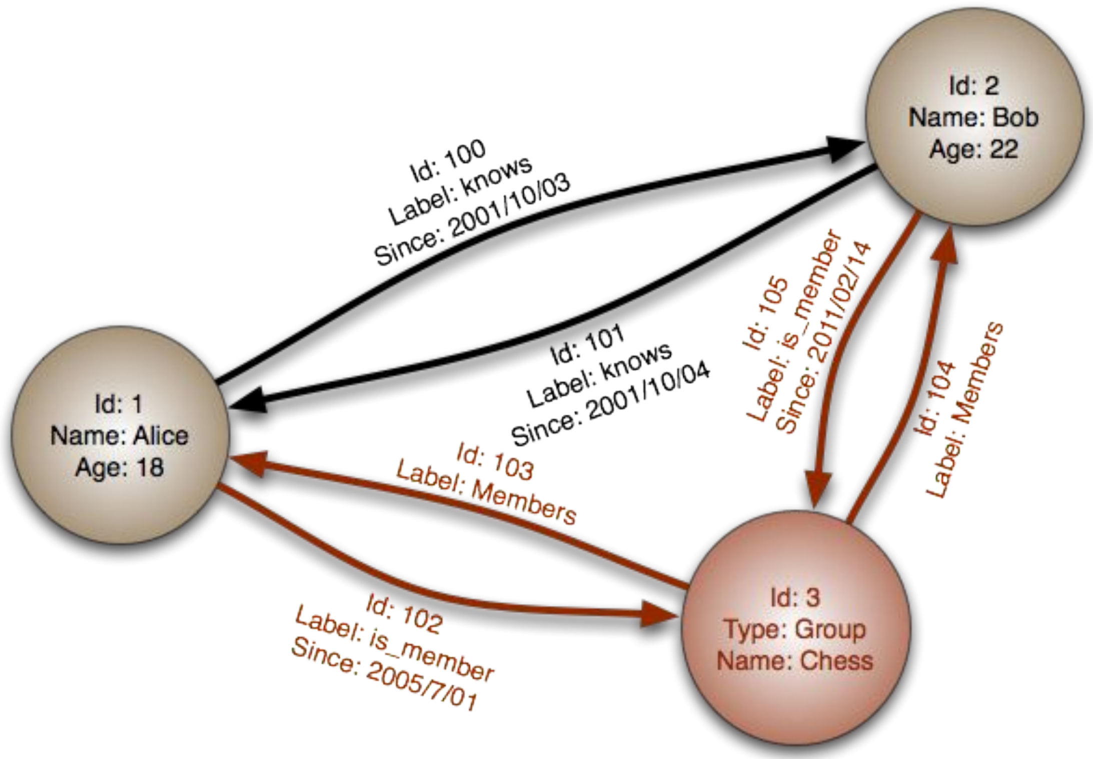
http://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archIntro.html

Graph DB (graph)

In a graph database, each node is a record and each arc is a relationship between two nodes. Graph databases are optimized to represent complex relationships with many foreign keys or many-to-many relationships.

Graphs databases offer high performance for data models with complex relationships, such as a social network. They are relatively new and are not yet widely-used; it might be more difficult to find development tools and resources. Many graphs can only be accessed with REST APIs.

Graph DB (graph)



Some graph DBs:

- https://neo4j.com/
- https://blog.twitter.com/2010/introducing-flockdb

SQL v NoSQL

SQL v NoSQL

- Structured data
- Strict schema
- Relational data
- Need for complex joins
- Transactions
- Clear patterns for scaling
- More established: developers, community, code, tools, etc
- Lookups by index are very fast

- Semi-structured data
- Dynamic or flexible schema
- Non-relational data
- No need for complex joins
- Store many TB (or PB) of data
- Very data intensive workload
- Very high throughput for IOPS (input/output operations per second)

https://www.sitepoint.com/sql-vs-nosql-differences/

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

Thank you.

Do you have any questions?

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