igotanoffer.com

Databases: system design interview concepts (2 of 9)

21-27 minutes

If you want to succeed in system design interviews for a tech role, then you'll probably need to understand databases and how to use them within the context of a larger system.

Databases are an important part of the world's biggest technology systems (i.e. Facebook, Amazon, etc.). And the way databases are used have a huge impact on the speed, scalability, and consistency of those systems.

This is a huge topic with a lot of moving parts. So, to make your life easier, we've put together the below overview of the most important database topics that you'll want to know during a system design interview. Here's what we'll cover:

1. Database basics
2. Relational databases
3. Non-relational databases
4. Summary of different databases
5. Example database interview questions
6. System design interview preparation

1. Database Basics

At its most fundamental, a **database** is responsible for the storage and retrieval of data for an application. A database can also be called a Database Management System (DBMS) because it’s an application that manages access to a physical data store. The core functions of a database are to:

- store data
- update and delete data
- return data according to a query
- administer the database

And when providing these data services, a database needs to be reliable, efficient, and correct. It turns out that doing all these things at once is fairly complex. We’ll start by looking at the CAP theorem to understand the tradeoff at the very heart of database design.

1.1 CAP Theorem

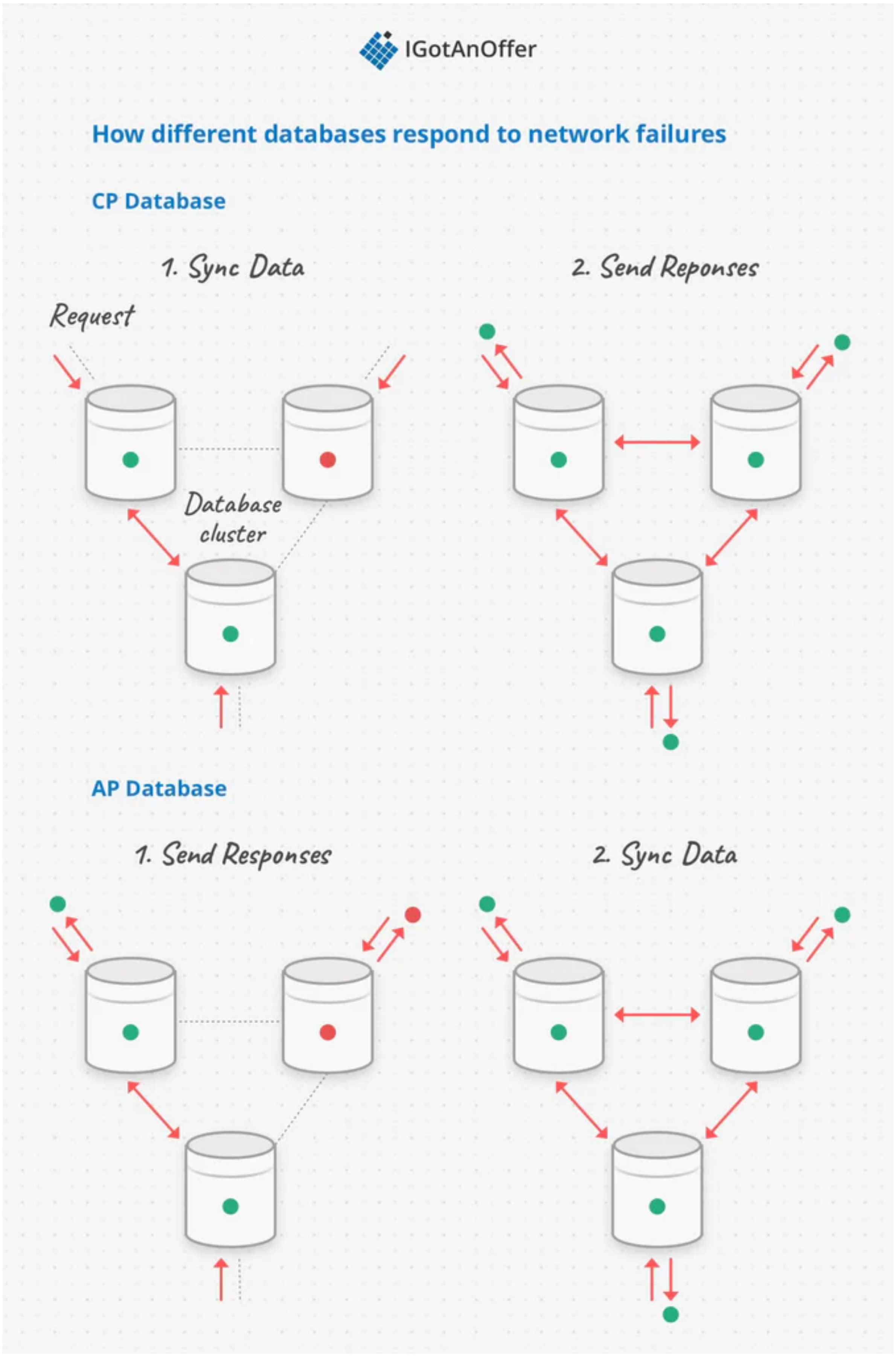
The **CAP Theorem** says that any distributed database can only satisfy two of the three features:

- **Consistency**: every node responds with the most recent version of the data
- **Availability**: any node can send a response
- **Partition Tolerance**: the system continues working even if communication between any of the nodes is broken

We live in a physical world and can't guarantee the stability of a network, so distributed databases must choose Partition Tolerance. This implies a tradeoff between Consistency and Availability. In other words if part of the cluster goes down, the system can either satisfy Consistency, roll back any unfinished operations and wait to respond until all nodes are stable again. Or it can satisfy Availability, continue responding but risk inconsistencies.

A system is called a **CP database** if it provides Consistency and Partition Tolerance, and an **AP database** if it provides Availability and Partition Tolerance.

Now that we’ve talked about the theoretical constraints of the CAP Theorem, let’s look at the fundamental unit of database operations: the transaction.



1.2 Transactions

A transaction is a series of database operations that are considered to be a "single unit of work". The operations in a transaction either all succeed, or they all fail. In this way, the notion of a transaction supports data integrity when part of a system fails. This is formalized in the "ACID" properties:

- **Atomicity** - all operations succeed or fail together, i.e. the transaction is an "atomic" unit
- **Consistency** - a successful transaction places the database in a valid state, that is, no schema violations
- **Isolation** - transactions can be executed concurrently
- **Durability** - a "committed" transaction is persisted to memory

Not all databases choose to support ACID transactions, usually because they are prioritizing other optimizations that are hard or theoretically impossible to implement together. Usually relational databases do support ACID transactions, and non-relational databases don’t.

Regardless of ACID support, database transactions make changes that need to be consistent with the organization of data within the database. Next we’ll look at the different ways a system designer can specify this data structure organization, called a schema, and the impacts these design choices have on performance.

1.3 Schemas

The role of a **schema** is to define the shape of a data structure, and specify what kinds of data can go where. In databases specifically, a schema can specify database-level structures like tables and indexes, and also data-level constraints like field types

(string, boolean, reference, etc.)

Schemas can be strictly enforced across the entire database, loosely enforced on part of the database, or they might not exist at all. There can be one schema for the entire database, or different entries can have different schemas. As we'll see below, all of these variations can be valuable in different use cases.

The advantage of a schema when strictly enforced, is that it is safe to assume that any queries to the system will return data that conforms to the schema assumptions. As nice as these guarantees can be, they have some major drawbacks:

- **Computationally expensive**: to enforce a schema, the schema properties have to be confirmed on every, write, update, and delete.
- **Difficult to scale**: especially if the schema specifies how data entries can reference each other, maintaining these constraints becomes more difficult as the reference span clusters and schema rules need to be verified across the network.

The last important feature of databases we'll talk about before jumping into specific models is scaling - how to respond to increasing database demands in a system.

1.4 Scaling

It is more important than ever to be able to implement databases in distributed clusters as dataset sizes continue to grow. Different databases are better and worse at scaling because of the features they provide. There are two kinds of scaling:

- **Vertical Scaling**: adding compute (CPU) and memory (RAM, Disk, SSD) resources to a single computer.

- **Horizontal Scaling**: adding more computers to a cluster

Vertical scaling is fairly straightforward, but has much lower overall memory capacity. Horizontal scaling on the other hand has much higher overall compute and storage capacity, and can be sized dynamically without downtime. The big drawback is that relational databases, the most popular database model, struggle to scale horizontally.

2. Relational Databases

2.1 General overview

All databases need a data model to specify logical organization and rules of the data. A **relational database** is simply a database that uses a **relational data model**, which organizes data in tables with rows of data entries and columns of predetermined data types. Relationships between tables are represented with **foreign key** columns that reference the **primary key** columns of other tables.

Most importantly, relational data models strictly enforce constraints to ensure that data values and relationships are always valid against the schema. ACID transactions are almost always implemented to ensure schema conformance.

Relational databases are also called SQL databases because **SQL (Structured Query Language)** is the standard query language for relational models. SQL is declarative, meaning the requesting entity tells the database what it wants, but the database, specifically the **query planner**, decides how to get it.

Tuning the query planner is one of the primary optimization techniques for relational databases.

When data is frequently accessed by the same column, we can add an **index** on that column to speed up the search. An index is essentially a table that has a copy of the column of interest and a foreign key reference to the original table.

Databases support special ordering data structures on indices to make access faster than just scanning row by row. However this performance boost on reads is balanced by slower writes, because each index needs to get updated in addition to the primary table.

Relational databases are almost always CP databases because guaranteeing consistency is important to upholding the relational model, and making sure that no matter what transactions occur, the database is always in a valid state.

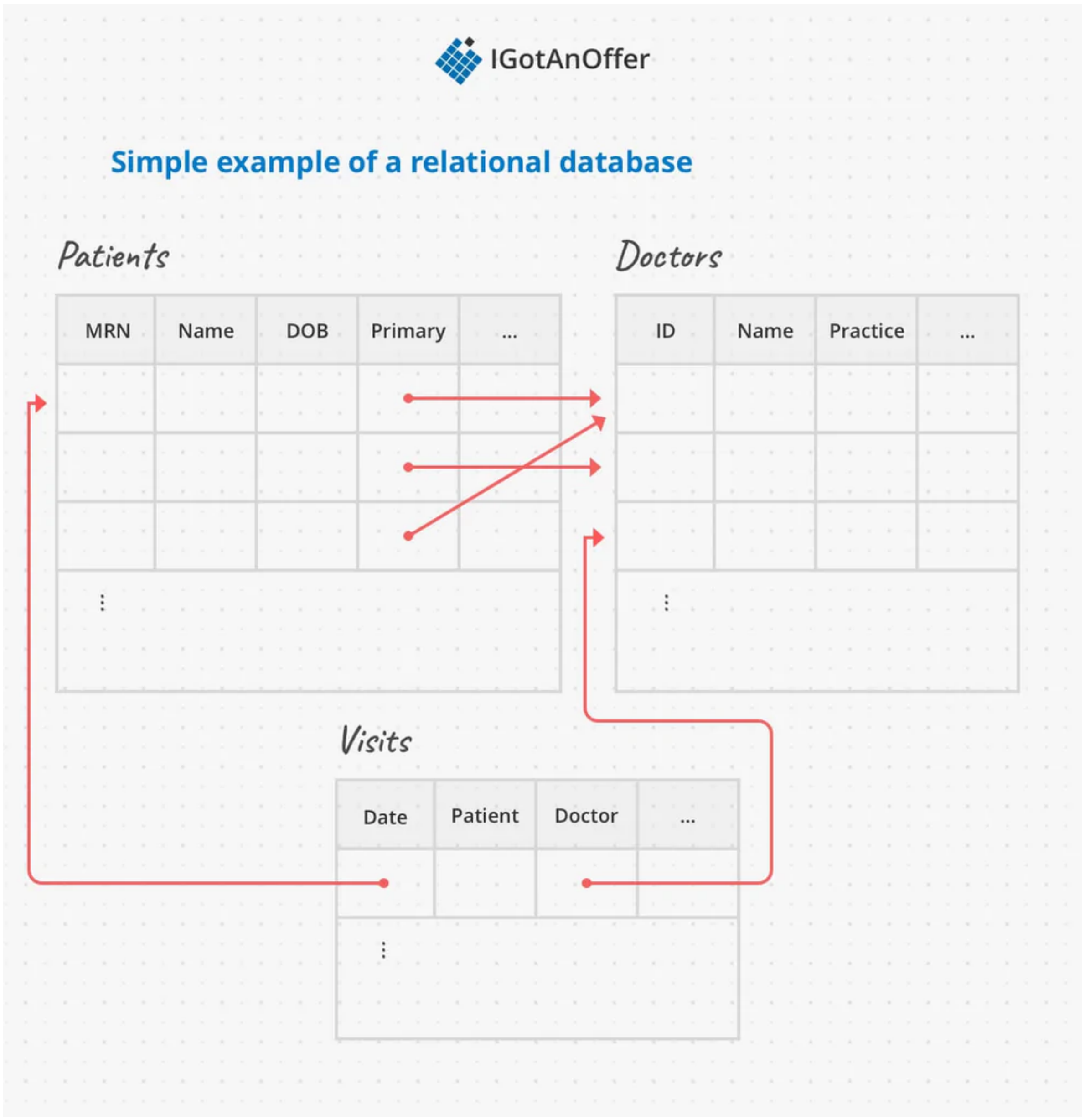
2.2 When to use

Let's look at an example to build an intuitive understanding of when all the work relational databases do to enforce schema constraints is worth it. In this simple example, we want to build a hospital a data system to support safe and accurate medical care. We can store the data in three tables: patient information, doctor information, and patient visits.

We’d like to be able to look up patients quickly, and in a clinical context patient data can come up in a lot of different ways. In one case we might need to look up a patient by name, like when they check in. Another situation might need to look up by the MRN (Medical Reference Number), like when a doctor prescribes a medication. To make sure the lookup is fast in both cases we’ll put two indices on the patient table: one on the Name column, and one on the MRN column.

To make sure the hospital can track a patient's medical history, the visits table needs to reference both the patient table and doctor table. We also want the patient table to reference the doctor table to note who a patient’s primary care physician is. We can already see in this very simplistic view of medical data how many relationships we need between tables. This is a sign that the relational model is a good fit.

Even if this medical data is distributed over a cluster, it is very important that the response is correct. The personal impacts could be severe if the database didn’t provide the most up to date information on a patient's medical care. This need for correctness is a sign that the relational model is a good fit. As we'll see, not all use cases have such strong requirements, and so other non-relational models can be considered.



In summary, relational databases are best when:

- there are many-to-many relationships between entries
- data needs to follow the predetermined schema
- relationships between data always need to be accurate

The top industry technologies for relational databases are:

- Oracle
- MySQL
- PostgresQL

2.3 Disadvantages

The biggest disadvantage of relational databases is that they are hard to scale over distributed clusters (horizontal scaling). Relational databases are most useful when there are many relationships in the data, so no matter what way you split up the data there will be relationships between data entries on different nodes.

When these cross-node relationships get updated, the nodes have to communicate with each other to “normalize” (keep in sync) the data. As a result, the database operations get slower because network communication is slower. You can learn more about these design tradeoffs in our article about data replication and partitioning (coming soon).

They also aren't particularly advantageous if the data doesn't have a lot of references, doesn't easily conform to a single schema, or changes shape frequently. The equipment around enforcing schemas becomes unnecessary optimization and slows down operations.

Increasingly, distributed systems are moving away from exclusively using relational models particularly because of difficulties around maintaining relational guarantees on a cluster and a need for more flexibility around schemas.

It's important now to be able to choose what kind of database is relevant to the data and use case at hand. Some databases implement "multi-model" support, supporting both relational and non-relational models. Some databases are only non-relational, though these tend to be experimental or less mature systems.

In the next section, we'll give you an overview of the multitude of non-relational database options available, so you can choose the best database approaches for any system design.

3. Non-Relational databases

The core question facing system designers today when choosing a database is given the structure of your data, what model do you want to choose to store it with? Non-relational databases are optimized for specific use cases that need scalability, schema flexibility, or specialized query support. As you'll see in this section, there are many kinds of non-relational databases. We'll go over a few of the most important to know when designing systems.

Non-Relational Databases are often called NoSQL databases because they can use other query languages to optimize for their use case. But often non-relational databases will implement SQL or SQL-like query support to make developer’s lives easier. The underlying execution of these queries, however, will be very different in relational and non-relational systems.

Non-relational databases are either AP databases or CP databases, because they’re targeting specific use cases that have varying priorities between availability and consistency. In the case of AP non-relational databases, the model of **eventual consistency** is used to make sure that consistency still happens over time, it’s just not guaranteed exactly after a transaction completes.

As you'll see in this section, there are many kinds of non-relational databases. We'll go over a few of the most important to know when designing systems. Lets jump in!

3.1 Graph Database

Graph databases model data with nodes and edges. They are the most similar of the non-relational databases to a relational data model, because they also are good at representing data with lots of relationships. The main benefit of a graph database is that queries don't need joins to follow relationship edges, because data isn't stored in tables. So they're quite well suited for queries that traverse many edges of a graph, such as social network analytics.

The top graph databases in the industry are Neo4J and CosmosDB.

3.2 Document Store

Document stores are usually simple JSON objects stored with a key identifier. Documents represent a set of information pertaining to a single topic, that in a relational database could be spread out over different tables. For example, a document store might archive medical records related to a single patient so that only one document needs to be accessed at a time, and all the relevant information is there.

Notably, documents can have a variety of different schemas, which makes it easy to update a single document without updating the entire database. For example, a news outlet might choose to start adding a location field to their articles, and in a document store the location field can be added to new documents without having to update old ones.

The top document store databases in the industry are MongoDB, Couchbase, Firebase, CouchDB, and DynamoDB.

3.3 Key-Value Store

A Key-Value store is similar to a document store, but the data stored in the value is opaque. The key value store has no notion of what is stored in the value, so it only provides simple read, overwrite, and delete operations.

There are no schemas, no joins or indices - you can think of it as a very large hash table. Because of this minimal overhead they are easy to scale. Key-value stores are particularly suitable for caching implementations.

When the values need to be large, this kind of database is referred to as an **Object Store**, or **Blob Store**. In this case, the data might be serialized and optimized for large file sizes. Use cases include videos, images, audio, disk images, and log binaries.

The top key-value stores in the industry are Redis, DynamoDB, CosmosDB, Memcached, and Hazelcast.

3.4 Column-Family Database

Column families are a set of columns that are typically retrieved together. A column-family database models data in tables like a relational databases, but stores column families together in files instead of rows, and doesn't enforce relational constraints.

For data with strong column-family access patterns, this model boosts performance by limiting how much data needs to be read. And since columns tend to hold a repeating type of information, they can be compressed to save space, which is especially helpful if the data is sparse.

For example, a column-family database might group name columns, so regardless of which components of a name someone has, (because names are complex!) all the possible fields will be retrieved together. This model also effectively partitions the data table for horizontal scaling.

The top column-family databases in the industry are Cassandra, HBase, and CosmosDB.

3.5 Search Engine Database

A search engine database provides the specialized feature of full text search over very large amounts of unstructured text data. The data can possibly come from multiple sources, and users of the database may also want "fuzzy search", meaning the results may not exactly match the search string.

A well-known example of this is searching websites (like Google), but it can also be valuable for example in being able to search and debug large volumes of system logs.

The top search engine databases in the industry are Elasticsearch, Splunk, and Solr.

3.6 Time Series Database

A time series database is optimized for data entries that need to be strictly ordered by time. The main use case is storing real time data streams from system monitors.

For example, a huge social network service might want to log every time a user runs into an error, and feed all the different error-handling services into a single searchable time-series database for engineers to easily track down and debug problems.

Time series databases are write heavy, and usually provide services for sorting streams as they come in to make sure they are appended in the correct order. These databases can be easily partitioned by time range.

The top time series databases in the industry are InfluxDB, Kdb+, and Prometheus.

4. Summary of different databases

When choosing a database, it is important to consider data size, structure, relationships, and how important it is to enforce schemas and ensure consistency. Remember that especially in large systems, one database might not be able to do everything you need, so it's ok to choose more than one.

As scale becomes more important, relational databases become too expensive to maintain, so start looking at non-relational alternatives for parts of the system that don't need strong schemas and consistency guarantees. For reference, we've summarized the key points for each kind of database:

Relational database

- many-to-many relationships
- data and data relationships need to strictly follow schema
- consistent transactions are important
- hard to scale because relationships are hard to partition effectively

Graph database

- many-to-many relationships (graph structure)
- fast at following graph edges

- suited to complex network analytics
- less mature technology than Relational

Document store

- isolated documents
- retrieve by a key
- documents with different schemas that are easy to update
- easy to scale

Key-value store / object store

- opaque values
- no schema or relationships known to the database
- very simple operations
- easy to scale

Column-family database

- groups related columns for storage (easy to scale)
- memory effective for sparse data

Search engine database

- large amounts of unstructured data
- full text search or fuzzy search service

Time series database

- data is ordered by time
- many data streams
- real time entry ordering functionality

5.Example database questions

The questions asked in system design interviews tend to begin with a broad problem or goal, so it's unlikely that you'll get an interview question entirely about a database.

However, you may be asked to solve a problem where databases will be an important part of the solution. As a result, what you really need to know is WHEN you should bring it up and how you should approach it.

To help you with this, we've compiled the below list of sample system design questions. Databases are a relevant part of the answer for all of the below questions.

- Design Dropbox (Read the answer)
- Design Pastebin (Read the answer)
- Design a web crawler (Read the answer)
- Design Twitter (Read the answer)
- Design Flickr (Read about Flickr's actual architecture)

6. System design interview preparation

Databases are often an important part of a system. But to succeed on system design interviews, you'll also need to familiarize yourself with a few other concepts. And you'll need to practice how you communicate your answers.

It's best to take a systematic approach to make the most of your practice time, and we recommend the steps below. For extra tips, take a look at our article: 19 system design interview tips from FAANG ex-interviewers.

6.1 Learn the concepts

There is a base level of knowledge required to be able to speak intelligently about system design. To help you get this foundational knowledge (or to refresh your memory), we've published a full series of articles like this one, which cover the primary concepts that you'll need to know:

- Network protocols and proxies
- Databases
- Latency, throughput, and availability
- Load balancing
- Leader election
- Caching
- Sharding
- Polling, SSE, and WebSockets
- Queues and pub-sub

We'd encourage you to begin your preparation by reviewing the above concepts and by studying our system design interview prep guide, which covers a step-by-step method for answering system design questions. Once you're familiar with the basics, you should begin practicing with example questions.

6.2 Practice by yourself or with peers

Next, you'll want to get some practice with system design questions. You can start with the examples listed above, or with our list of 31 example questions.

We'd recommend that you start by interviewing yourself out loud. You should play both the role of the interviewer and the candidate, asking and answering questions. This will help you develop your communication skills and your process for breaking down questions.

We would also strongly recommend that you practice solving system design questions with a peer interviewing you. A great place to start is to practice with friends or family if you can. If you don't have anyone in your network who can interview you, then you might want to check out our system design mock interview peer group.

6.3 Practice with ex-interviewers

Practicing with peers can be a great help, and it's usually free. But, at some point you'll start noticing that the feedback you are getting from peers isn't helping you that much anymore. Once you reach that stage, we recommend practicing with ex-interviewers from top tech companies.

If you know someone who has experience running interviews at Facebook, Google, or another big tech company, then that's fantastic. But for most of us, it's tough to find the right connections to make this happen. And it might also be difficult to practice multiple hours with that person unless you know them really well.

Here's the good news. We've already made the connections for you. We've created a coaching service where you can practice system design interviews 1-on-1 with ex-interviewers from leading tech companies. Learn more and start scheduling sessions today.

Learn more about system design interviews

This is just one of 9 concept guides that we've published about system design interviews. Check out all of our system design articles on our Tech blog.