Short Answers

1. What is a relational database?

A relational database is a type of database that organizes and stores data in tables (or relations) structured in rows and columns. Each row represents an entry in the table, also known as a record, and each column represents a field or attribute that defines a specific piece of data about that record. There is usually an ID column that serves as the primary key, acting as a unique identifier for each record. A table may also contain a foreign key, which establishes a relationship with another table by referencing its primary key. Referential integrity ensures that all foreign keys point to valid entries in the related table, maintaining consistency across the database.

Relational Database Management Systems (RDBMS) enforce a strict and predefined structure, support normalization, and ensure transactions adhere to ACID properties. SQL (Structured Query Language) is the most widely used language for querying, reading, and writing data in an RDBMS.

2. What is data modeling? Can you give an example and explain data modeling in terms of entities and their relationships?

Data modeling is the process of organizing and structuring data to define how it is stored, accessed, and related within a database. For example, on a social media website, users can post content, leave comments, like content, and follow each other. In this case, the user, post (content), like, comment, and follow are the entities. Each entity has properties, including an ID property as its primary key, along with additional attributes.

For example, the user entity could have properties such as username, email, and password. The post entity could include content, timestamp, likes count, and a foreign key to the comment entity (via the post ID). The like entity would most likely have both the user ID and post ID as foreign key properties, along with a timestamp property. Similarly, the comment entity would have the post ID and user ID as foreign keys, as well as content and timestamp properties. Finally, the follow entity would include the ID of the follower and the ID of the user being followed, establishing the relationship between users.

3. What is a primary key and a foreign key?

The primary key is a column in a table that uniquely identifies each record in that table. The primary key ensures that no two rows in the table have the same value for that key, making each record distinct. The primary key must always have a unique value for each record and cannot be `null`, ensuring data integrity. Typically, a primary key is used to quickly identify and access records within the table. For example, in a Users table, the `userId` could be the primary key, ensuring that each user has a unique ID.

The foreign key is a column in a table that references the primary key of another table. The foreign key creates a relationship between the two tables, enabling you to link records from different tables. The foreign key ensures that the data in the referencing table (the one containing the foreign key) corresponds to a valid entry in the referenced table (the one containing the primary key). This helps maintain referential integrity, ensuring that relationships between tables are consistent. For example, in a social media Posts table, a `userId` foreign key can reference the `userId` primary key from the Users table, establishing a relationship between each post and the user who created it.

4. Explain normalization and why it is useful.

Normalization is the process of organizing datasets efficiently by splitting tables into separate tables to eliminate redundancy and avoid anomalies (inconsistencies) that can arise when data is not structured properly. It involves breaking down large tables into smaller, more manageable ones and establishing relationships between them using primary and foreign keys. By eliminating redundancy, it minimizes the repetition of data across a database, resulting in a more compact and efficient database. Anomalies that may arise include the update anomaly, the delete anomaly, and the insert anomaly. The update anomaly occurs when you don't properly update all the related entries. The delete anomaly happens when you delete one thing and unintentionally delete more data. The insert anomaly occurs when you can't add data because you don't have other data.

5. What is a transaction? Assuming part of it succeeded and another failed, what happens?

A transaction is the execution of one or more SQL statements as a set. If successful, all the changes are committed (applied to the table); if there is an error, all changes are erased or rolled back. In other words, there are no partial completions—it's all or nothing.

Transactions must be controlled to ensure data integrity and must adhere to the ACID properties—Atomicity, Consistency, Isolation, and Durability—to ensure the reliability of database transactions. Atomicity guarantees that each transaction is treated as a single, indivisible unit, meaning it either completes fully or does not happen at all, preventing partial updates that could lead to data corruption. Consistency ensures transactions bring the database from one valid state to another, maintaining predefined rules such as constraints and relationships. Isolation ensures transactions operate independently, preventing interference from other transactions and preserving data integrity during concurrent operations. Finally, Durability guarantees that once a transaction is committed, its results are permanently saved, even in the event of a system crash.

6. What is a non-relational database?

A non-relational database (NoSQL) is optimized for specific data needs and scalability, meaning it can distribute data across multiple systems easily. Unlike relational databases, NoSQL databases don't use tables or primary/foreign keys. They are typically cheaper to maintain, as they allow for horizontal scaling, and they store data in a de-normalized form, meaning some redundancy is allowed to improve performance. This results in better throughput (can better handle high volumes of data and requests more efficiently than relational databases), as NoSQL databases can handle more data and higher request volumes efficiently.

There are various types of NoSQL databases designed for different use cases. Document databases (like MongoDB and CouchDB) store data as field-value pairs in a "document", typically in binary JSON (BSON). Key-value databases (like Redis and DynamoDB) manage data as key-value pairs within a single object. Graph databases (like Neo4j and Amazon Neptune) are designed to handle highly interconnected entities, making them ideal for complex relationships. Finally, columnar databases (like Cassandra and HBase) store data in columns, similar to relational databases but optimized for large-scale data analytics. This flexibility allows NoSQL databases to efficiently handle different data models and requirements.

7. What does it mean to have eventual consistency?

Eventual consistency is part of the CAP theorem, which stands for Consistency, Availability, and Partition Tolerance. In CAP, "C" stands for consistency: all clients see the same, most up-to-date data at any time. "A" stands for availability: any client that sends a request will receive a response. "P" stands for partition tolerance: the system will continue to operate even when some connections are lost. Brewer's CAP Theorem states that during network failures, distributed systems can only guarantee two out of the three properties. For example, MongoDB and Redis can guarantee consistency and availability, while CouchDB and Cassandra can guarantee availability and partition tolerance. In this context, eventual consistency comes into play, meaning that after an update, the data in distributed databases may not be immediately consistent, but will eventually synchronize over time. For example, this could apply to scenarios like ordering the last few items in stock, where updates to inventory may take some time to propagate across all database replicas.

8. What is the difference between vertical and horizontal scaling?

When we talk about scalability, there are two types: vertical scalability and horizontal scalability. Vertical scaling involves upgrading a server with a better CPU, more RAM, or additional storage space. This allows you to maintain the application's architecture, but it can be expensive and comes with technological limitations, as you can only scale up to a certain point. Horizontal scaling, on the other hand, involves distributing the dataset and query load across multiple servers, each handling a subset of the overall workload. While this solution is typically cheaper, it requires changes in the application architecture, making it more complex and harder to maintain. Whether or not scaling is necessary depends on factors like the size of the data and the query rate.

Replication, partitioning, and sharding are all techniques related to horizontal scaling, where data is distributed across multiple servers to improve performance and reliability. Replication involves creating copies of a master database and hosting them on separate servers, resulting in replica (formerly "slave") databases. Data can be read from any replica, but it is only written to the master. This setup provides fault tolerance, allowing the system to continue operating even during partial failures.

Partitioning involves splitting datasets into smaller subsets, all within the same server. Horizontal partitioning divides data by rows, distributing them across different tables while maintaining the same shape and structure of the data. Vertical partitioning divides data by columns, which results in a different shape and structure of the data.

Finally, sharding involves spreading a dataset across multiple databases distributed across multiple servers, where none of the servers communicate with each other. Sharding strategies include key-based, range-based, dictionary-based, hierarchy-based, and entity-group-based partitioning, each suited to different use cases for distributing data effectively.

9. When would you choose a RDBMS vs NoSQL database?

When choosing between an RDBMS and a NoSQL database, the decision depends on several factors, such as data structure, scalability needs, consistency requirements, and the nature of the application. An RDBMS is best suited for applications with structured, static, pre-defined schemas and strong consistency. RDBMSs excel in handling complex transactions, ensuring ACID properties, and supporting SQL queries with joins between multiple entities. They typically scale vertically, requiring upgrades to a single server, and are ideal for applications with fixed data models and strong data integrity needs, such as banking systems.

On the other hand, NoSQL databases are more suitable for applications that require horizontal scalability and flexibility. They are designed to handle dynamic, unstructured, or semi-structured data with dynamic schemas. NoSQL databases prioritize availability and scalability over strict consistency, making them ideal for applications that can tolerate eventual consistency, such as social media platforms or real-time data processing systems. They use different models like document, key-value, graph, and columnar, and can scale across multiple servers using techniques like sharding and partitioning.

10. What is mongoose? Why would we use it instead of interacting with the native MongoDB shell?

First off, MongoDB is a document-based database that stores data in JSON-like format (in the form of objects), using BSON (binary JSON). Each document is an object of field-value pairs that represent an instance of an entity, similar to how a row represents a record or instance of an entity in a relational database. In MongoDB, a collection is a group of all documents for a specific entity, which is akin to a table in an RDBMS. Data that is accessed together should be stored together in a collection, ensuring efficient access and organization, just as related columns are stored together in relational database tables. To query and manipulate data in document-based databases, we use an Object Document Mapper (ODM). This differs from relational databases (RDBMS), which use Object Relational Mappers (ORMs) to interact with data. For example, SQL databases often use ORMs like Sequelize (for Node.js) or Flask-SQLAlchemy (for Python) to manage data.

For MongoDB, the ODM (Object Document Mapper) is Mongoose. Mongoose is a powerful MongoDB ODM that provides a more structured way to interact with MongoDB in an asynchronous environment. It supports both promises and callbacks for handling asynchronous operations. Mongoose also buffers (temporarily stores) queries until the server is connected to MongoDB, ensuring that no operations are missed if the connection has not yet been established. Additionally, Mongoose provides built-in data validation, allowing you to define validation rules within schemas, ensuring that only valid data is saved to the database. This structure helps to maintain consistency and integrity within the data while simplifying database interactions in Node.js applications.

11. Explain embedded vs reference relationships.

MongoDB uses models, which are objects created based on a schema definition. Instances of a model are documents. The syntax for creating a model is `mongoose.model(modelName, schema, collectionName)`. The schema is an abstract representation of a MongoDB collection that defines the shape and types of the documents within that collection.

Embedded and reference relationships are directly related to the model. Embedded relationships occur when documents of one model contain nested documents from another model. Reference relationships occur when documents of one model contain references (using document IDs) to documents from another model. This follows a similar logic to relationships in RDBMS databases and its use of foreign keys, and how it defines one-to-one (1:1), one-to-many (1:N), and many-to-many (N:N) relationships. Embedded relationships are typically used for one-to-one or one-to-many relationships, while reference relationships are more common for many-to-many relationships.

n MongoDB, denormalization is typically achieved through embedded relationships, where data is stored together within a single document. This approach allows for faster querying, as the entire document and its related data are retrieved in a single query. This is ideal for use cases where the data is mostly unique and is frequently retrieved together, such as when a blog post includes its comments and author information in one document. However, there are limitations, such as the document size limit, which can constrain the amount of data you can store in a single document.

On the other hand, reference relationships in MongoDB are more aligned with normalization, where data is split across multiple documents, and one document contains references (usually via document IDs) to another. This reduces redundancy and allows for easier updates to related data without needing to modify large documents. Reference relationships are often used when data is rarely retrieved with its parent, or when the data set is more static, such as when storing user information in one document and their posts in another. In these cases, querying typically involves additional queries or subqueries to retrieve the related data.

Use embedded relationships for one-to-one or one-to-many relationships where data is frequently retrieved together, while reference relationships are ideal for many-to-many relationships, data that is rarely retrieved with its parent, or when independent updates are needed.