Database notes

SQL vs NoSQL

- **SQL databases** are relational, and **NoSQL databases** are non-relational.
- **NoSQL** stands for Not only SQL.
- **SQL databases** use structured query language (SQL) and have a *predefined* schema. **NoSQL databases** have *dynamic* schemas for unstructured data.
- **SQL databases** are *vertically* scalable, while **NoSQL databases** are *horizontally* scalable.
- *Horizontal* scaling refers to adding additional nodes, *vertical* scaling describes adding more power to your current machines.
- **SQL databases** are table-based, while **NoSQL databases** are document, key-value, graph, or wide-column stores.
- **SQL databases** are better for multi-row transactions, while **NoSQL** is better for unstructured data like documents or JSON.
- **NoSQL systems** are designed to be more flexible than traditional relational databases and can scale up or down easily to accommodate changes in usage or load.
- The decision of which type of database to use SQL or NoSQL - will depend on the particular needs and requirements of the project. For example, if you need a fast, scalable, and reliable database for web applications then a NoSQL system may be preferable.
- On the other hand, if your application requires complex data queries and transactional support then an SQL system may be the better choice.

- **SQL** follow *ACID* properties but **NoSQL** does not necessarily follow them.
- **SQL** uses normalized data structure but **NoSQL** don't.
- **SQL examples**: PostgreSQL, MySQL.
- **NoSQL examples**: MongoDB, Amazon DynamoDB, Cassandra.

Database Normalization and Denormalization

- **Database normalization** is a process used to organize a database into tables and columns. The idea is that a table should be about a specific topic and that only those columns which support that topic are included. This limits the number of duplicate data contained within your database. This makes the database more flexible by eliminating issues stemming from database modifications.
- **Denormalization** is the opposite of normalization. It is the process of adding redundant data to a database to improve read performance. This is done by adding duplicate data into multiple tables to avoid expensive joins. This is done at the expense of increased storage and decreased write performance.

Benefits of Normalization:

- Avoiding Data Duplication (Redundancy)
- Eliminate Insert, Delete, and Update Anomalies.
- Avoid Frequent Null Values.

Anomalies:

- *Insert Anomaly*: Insertion where insertion is restricted without the presence of other data.

- *Delete Anomaly*: Deletion where the deletion of any data leads to deletion of other useful data.
- *Update Anomaly*: Every copies is updated so as to update a single piece of data.

Functional Dependency:

- Full functional dependency occurs when a non-key attribute (an attribute that is not part of the primary key) is fully dependent on the entire primary key.
- Partial functional dependency occurs when a non-key attribute depends on only part of the primary key.
- Transitive functional dependency occurs when a non-key attribute depends on another non-key attribute that, in turn, depends on the primary key.

Normalization Levels:

- **1NF**: No Multivalued Attributes, No Composite Attributes, No Repeating Groups.
- **2NF**: 1NF, No Partial Dependencies.
- **3NF**: 2NF, No Transitive Dependencies.

Designing ERD (Entity-Relationship Diagram)

- Define Entities
- Define Attributes: simple, multivalued, composite and derived.
- Define Unique Identifiers
- Define Relations.

Candidates Key: attributes that can serve as a unique identifier for the entity

Weak entities are completely dependent on another entity, and their existence is tied to the presence of that other entity.

The degree of relationship refers to how many entities are involved in the relationship. Relationships can be classified into different degrees:

- Unary or Recursive
- Binary
- Ternary

Cardinality determines the maximum number of instances of each entity that can be related.

- one to one
- one to many
- many to many

Participation represents the minimum number of relationships an entity instance can have

- Must
- May

Identifying relationship, which is differentiated by a double line, links a weak entity to its owner entity. This relationship signifies the dependency of the weak entity on the owner entity.

Mapping ERD to Tables

- It is the process of converting *conceptual* design to *logical* design
- Simple attributes, like ID or Name, are directly represented as columns in the table.

- Composite attributes, such as Address, are divided into subparts (e.g., Street and Zone) and stored in separate columns.
- Multi-valued attributes, like Phone numbers, are stored in separate tables. Each value is linked to the corresponding primary key of the entity it belongs to.
- Derived attributes, like Age, are often not stored directly in the database to improve performance.
- Weak entities are mapped by using the primary key of their owner entity as a foreign key in a separate table.
- **One-to-Many Relationship** (Binary or Unary): The primary key of the "One" side becomes a foreign key in the "Many" side.
- **Many-to-Many Relationship**:
  - Create a new table to represent the relationship.
  - Include foreign keys for both participating entities in the new table.
  - Any additional attributes related to the relationship can be included in this table.
- **One-to-One Relationship**:
  - Must-May: The primary key of the "May" side becomes a foreign key in the "Must" side. For example, an employee (Must) manages a department (May).
  - Must-Must: Either merge the two tables or The primary key of one side becomes foreign key in the other side
  - May-May: Take the primary key of one side and make it a foreign key in the other.
- **Ternary Relationship**:

    - Create a new table to represent the relationship.
    - Include foreign keys for all participating entities.

SQL Commands

DDL – Data Definition Language:

- CREATE
- DROP
- ALTER
- TRUNCATE

DQL – Data Query Language:

- SELECT

DML – Data Manipulation Language:

- INSERT
- UPDATE
- DELETE

DCL – Data Control Language:

- GRANT
- REVOKE

TCL – Transaction Control Language:

- COMMIT
- ROLLBACK

Delete and Truncate:

- *TRUNCATE* and *DELETE* both remove data from a table.

- TRUNCATE is a DDL (Data Definition Language) command, which automatically commits and cannot be rolled back.
- *DELETE* is a DML (Data Manipulation Language) command that can be rolled back because it doesn't use automatic commit.
- TRUNCATE has better performance for all rows deletion as DELETE act on a row level but TRUNCATE act on the whole table at once.
- A Database Transaction is a sequence of multiple operations performed on a database, and all served as a single logical unit of work – taking place wholly or not at all.

ACID and BASE Models

**Relational databases follow ACID

properties. On the other side NoSQL usually follow BASE Model.**

ACID:

- Atomicity: Atomicity ensures that all steps in a single database transaction are either fully-completed or reverted to their original state.
- Consistency: guarantees that data meets predefined integrity constraints and business rules. Even if multiple users perform similar operations simultaneously, data remains consistent for all.
- Isolation: ensures that a new transaction, accessing a particular record, waits until the previous transaction finishes before it commences operation. It ensures that concurrent transactions do not interfere with each other, maintaining the illusion that they are executing serially.

- Durability: ensures that the database maintains all committed records, even if the system experiences failure.

BASE:

- Basic Availability: the database’s concurrent accessibility by users at all times. One user doesn’t need to wait for others to finish the transaction before updating the record.
- Soft-State: data can have transient or temporary states that may change over time, even without external triggers or inputs. It describes the record’s transitional state when several applications update it simultaneously. The record’s value is eventually finalized only after all transactions complete.
- Eventual Consistency: the record will achieve consistency when all the concurrent updates have been completed.

In a lot of ways, BASE is the exact opposite of ACID, and prioritizes availability over perfect consistency; but that’s kind of the point of NoSQL in the first place.

Indexes are used in databases to address two main problems: unsorted data and scattered data. They help speed up data retrieval by organizing data and providing a way to find records more efficiently. However, indexes can slow down data modification operations (Insert, Update, Delete) because changes must be reflected in both the table and the index. They also consume additional storage space.

Views are database objects that act as logical tables based on one or more tables or other views. They don't contain their own data but provide a window to access data from other tables or views.

==Stored Procedures vs. Functions ==

Stored Procedures

- **Purpose**: Perform a series of operations or tasks (e.g., modify data, handle transactions).
- **Return Value**: No mandatory return value; can return multiple values via output parameters.
- **Side Effects**: Can modify database tables (INSERT, UPDATE, DELETE).
- **Execution**: Using `EXEC` or `CALL`.
- **Usage**: For tasks with complex logic and multiple SQL statements.
- **Transaction Control**: Can manage transactions (commit, rollback).

Functions

- **Purpose**: Return a single value or a table, used for computations and transformations.
- **Return Value**: Always returns a value (scalar or table).
- **Side Effects**: Generally do not modify database tables.
- **Execution**: As part of a SQL statement.
- **Usage**: In SELECT statements, computed columns, CHECK constraints, WHERE clauses.
- **Transaction Control**: Do not manage transactions.

Centralized Database Environment:

- **Client-Server Environment (Two-Tier):**
  - In the client-server environment, there were two tiers: a database server and client machines.
  - The client machines, known as "thick clients," had the application installed locally.
  - This distributed the point of failure to some extent. Application issues did not affect all users.
- **Internet Computing Environment (Three-Tier):**
  - In the three-tier environment, there were three tiers: a database server, application server, and thin clients.
  - Thin clients are smaller applications or applets accessed through browsers, reducing maintenance and support costs.
  - The application processing occurred on a separate server, making updates easier.
- **N-Tier Architecture:**
  - N-Tier architecture supports having more than one application server in parallel.
  - It's suitable for handling high loads or when multiple applications share the same database.
  - If one application server goes down, users can be routed to another operational server.
  - The centralized database still poses a single point of failure in this architecture.

Distributed databases offer high availability, making them suitable for critical systems that cannot tolerate downtime. They are valuable for systems used in long-distance separated locations where internet connectivity is challenging. Distributed databases, whether through replication or fragmentation, offer high availability and fault tolerance.

Replication involves making copies of the database, essentially copy-pasting it to different servers. There are two methods of replication: partial replication and full replication.

- Partial replication involves copying only a specific portion of the database that serves a particular branch or location.
- Full replication replicates the entire database, creating two identical servers that work in parallel.
- A signal or heartbeat is used to maintain synchronization between the servers, and if one server goes down, requests are rerouted to the other.

Fragmentation:

- Fragmentation is like cutting and pasting the database into different pieces, distributing the data across multiple servers.
- Fragmentation can be horizontal (grouping records), vertical (grouping columns), or a hybrid of both, maintaining the database's structure, rules, and constraints.
- The fragmented data is distributed across servers and connected through a specific network, creating a distributed database environment.
- This approach eliminates the single point of failure; if one fragment or server fails, the system continues to work without issues.
- Sharding is a specific type of horizontal fragmentation primarily used for scaling out databases by distributing data across multiple database instances or servers.

Locks are a mechanism to ensure data integrity, especially during concurrent operations. There are two major types of locks: **exclusive locks** and **shared locks**.

- **Shared locks** (read locks) allow multiple transactions to read from the same resource but prevent any transaction from updating it.
- **Exclusive locks** (write locks) lock the resource entirely, allowing only one transaction to update it while blocking other transactions from reading or writing to it.

It's important to note that multiple shared locks can be acquired on a resource at one time, but if the resource already has a shared lock, another process cannot acquire an exclusive lock on it. Similarly, a process cannot acquire a shared lock on a resource that is locked by an exclusive lock.

Optimistic locking allows processes to update the same records in parallel. Conflicts are resolved when one process successfully commits its changes, while the other process is informed of the conflict and must attempt the transaction again.

Pessimistic locking locks the row as soon as one process attempts to modify it, requiring other processes to wait before making any modifications.

Problems can arise when transaction isolation is not enforced:

- **Dirty Reads**: Occur when one transaction reads data modified by another transaction that has not yet committed. If the modifying transaction is rolled back, the read data becomes invalid.

- **Non-Repeatable Reads**: Occur when a transaction reads the same data multiple times within the same transaction but gets different results each time due to modifications by other transactions.
- **Phantom Reads**: Occur when a transaction reads a set of rows that satisfy a condition, but another transaction inserts or deletes rows that also satisfy the condition, causing the first transaction to see unexpected "phantom" rows.

Different levels of isolation address these problems:

- **Read Uncommitted**: Allows reading data not yet committed by other transactions, leading to potential issues.
- **Read Committed**: Only allows reading data committed by other transactions, reducing but not eliminating issues like non-repeatable and phantom reads.
- **Repeatable Reads**: Locks the resource throughout the transaction, preventing other transactions from updating the *same* rows, but it may still allow phantom reads.
- **Serializable**: Ensures that all concurrent transactions appear to be executed serially, eliminating issues like phantom reads.

Serializable is the highest level of isolation, providing the most strict guarantees against data anomalies.

The N + 1 Problem

The N + 1 problem, often encountered in database access and ORM (Object-Relational Mapping) frameworks, refers to a performance issue that arises when executing a query to fetch a collection of entities and then

executing an additional query for each entity in the collection. This results in N+1 queries being executed instead of a single, more efficient query.