Data

Data is a collection of raw, unorganized facts, figures, and symbols. It can be a collection of numbers, characters, images, or other observations. On its own, data may not have specific meaning or context. It is the basic unit of information that is processed or stored by a computer.

Types of Data

Data can be broadly classified into the following types based on its structure:

1. Structured Data:
    - This is data that adheres to a pre-defined data model and is therefore straightforward to analyze.
    - It is highly organized and formatted in a way that is easily searchable in relational databases.
    - Examples include data stored in relational database tables (like SQL databases), Excel spreadsheets, and data from sensors.
2. Unstructured Data:
    - This is data that does not have a pre-defined data model or is not organized in a pre-defined manner.
    - It is often text-heavy but may contain data such as dates, numbers, and facts as well.
    - Examples include text files, emails, social media posts, videos, audio files, and images.
3. Semi-Structured Data:
    - This is a form of structured data that does not conform with the formal structure of data models associated with relational databases or other forms of data tables, but nonetheless contains tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data.
    - It has some organizational properties that make it easier to analyze than unstructured data.
    - Examples include XML (eXtensible Markup Language) files, JSON (JavaScript Object Notation) files, and NoSQL databases.

Database

A database is an organized, systematic collection of data, stored and accessed electronically from a computer system. The primary purpose of a database is to store, retrieve, manage, and update a large amount of information efficiently and securely. The data is typically organized into tables, rows, columns, and indexes to make it easier to find relevant information.

DBMS (Database Management System)

A Database Management System (DBMS) is a software application that is used to create, manage, maintain, and interact with databases. It acts as an interface between the user/application and the database. The DBMS manages the data, the database engine, and the database schema, allowing for data to be manipulated, retrieved, and managed by users.

Key functions of a DBMS include:

- **Data Definition:** Creating, modifying, and removing definitions that define the organization of the data.
- **Data Manipulation:** Inserting, updating, deleting, and retrieving data from the database.
- **Data Security:** Providing mechanisms for controlling access to data.
- **Data Integrity:** Enforcing rules to ensure data is accurate and consistent.
- **Concurrency Control:** Managing simultaneous access to the database by multiple users.
- **Backup and Recovery:** Providing mechanisms for backing up data and recovering from failures.

Database Languages

Database languages are used to write commands to access, update, and manage data in a database. These commands are categorized based on their functionality.

1. DDL (Data Definition Language)

DDL commands are used to define or modify the database schema and structure. These commands deal with the objects in the database like tables, indexes, and users. Once a DDL command is executed, the changes are saved permanently.

- **`CREATE`**: Used to create new database objects like tables, views, indexes, and users.
    - Example: `CREATE TABLE Employees (ID INT, Name VARCHAR(255));`
- **`ALTER`**: Used to modify the structure of an existing database object. You can add, delete, or modify columns in an existing table.
    - Example: `ALTER TABLE Employees ADD Email VARCHAR(255);`
- **`DROP`**: Used to permanently delete existing database objects.
    - Example: `DROP TABLE Employees;`
- **`TRUNCATE`**: Used to remove all records from a table, including all spaces allocated for the records. The table structure remains.
    - Example: `TRUNCATE TABLE Employees;`
- **`RENAME`**: Used to rename a database object.
    - Example: `RENAME TABLE Employees TO Staff;`

2. DQL (Data Query Language)

DQL is used to retrieve data from the database. Its primary and often only command is `SELECT`. It is used to query the database and get data that matches criteria that you specify. In many contexts, DQL is considered a part of DML.

- **`SELECT`**: Used to retrieve data from one or more tables.
    - Example: `SELECT Name, Email FROM Employees WHERE ID = 101;`

3. DML (Data Manipulation Language)

DML commands are used for managing and manipulating data within the database objects. Unlike DDL, these commands are not auto-committed, meaning they can be rolled back.

- **`INSERT`**: Used to add new rows of data into a table.
    - Example: `INSERT INTO Employees (ID, Name) VALUES (102, 'Jane Doe');`
- **`UPDATE`**: Used to modify existing records in a table.
    - Example: `UPDATE Employees SET Email = 'jane.doe@example.com' WHERE ID = 102;`
- **`DELETE`**: Used to remove existing records from a table.
    - Example: `DELETE FROM Employees WHERE ID = 102;`

4. DCL (Data Control Language)

DCL commands are used to manage user access to the database. They deal with permissions and rights.

- **`GRANT`**: Used to give a user access privileges to the database.
    - Example: `GRANT SELECT, UPDATE ON Employees TO user_name;`
- **`REVOKE`**: Used to take back permissions from a user.
    - Example: `REVOKE UPDATE ON Employees FROM user_name;`

5. TCL (Transaction Control Language)

TCL commands are used to manage transactions in the database. A transaction is a sequence of operations performed as a single logical unit of work.

- **`COMMIT`**: Used to permanently save any transaction to the database.
    - Example: `COMMIT;`
- **`ROLLBACK`**: Used to undo transactions that have not already been saved to the database.
    - Example: `ROLLBACK;`

- **`SAVEPOINT`**: Used to temporarily save a transaction so that you can roll back to that point whenever necessary.
    - Example: `SAVEPOINT my_savepoint;`

ER-Model (Entity-Relationship Model)

The ER-Model is a high-level, conceptual data model used to represent the structure of a database. It describes the data in terms of entities, attributes, and the relationships between those entities. The primary purpose of the ER-Model is to design a database from a conceptual perspective before it is physically implemented.

ER-diagram (Entity-Relationship Diagram)

An ER-diagram is the graphical representation of an ER-Model. It visually illustrates the logical structure of a database by showing the entities, their attributes, and the relationships among them. Standard symbols are used:

- **Rectangles** represent entity sets.
- **Ellipses/Ovals** represent attributes.
- **Diamonds** represent relationship sets.
- **Lines** link attributes to entity sets and entity sets to relationship sets.

Entity

An entity is a real-world object, concept, or event with an independent existence that can be uniquely identified. It is a fundamental component of the ER-Model. For example, in a university database, `Student`, `Professor`, and `Course` are all entities. An **Entity Set** is a collection of similar types of entities.

Types of Entity

1. **Strong Entity:**
    - A strong entity is an entity that can exist on its own and is not dependent on any other entity for its existence.
    - It always has its own primary key, which uniquely identifies each instance within the entity set.
    - In an ER-diagram, it is represented by a single rectangle.
2. **Weak Entity:**
    - A weak entity is an entity that cannot be uniquely identified by its attributes alone. Its existence depends on another entity, known as the "owner" or "identifying" entity (which is a strong entity).
    - It does not have a primary key but has a **partial key** (or discriminator) that uniquely identifies it in relation to its owner entity.

- In an ER-diagram, a weak entity is represented by a double rectangle, and the identifying relationship is shown with a double diamond.

Attributes

Attributes are the properties or characteristics that describe an entity. For example, for a `Student` entity, attributes could be `StudentID`, `Name`, `Address`, and `DateOfBirth`.

Types of Attributes

1. **Simple Attribute:** An attribute that cannot be broken down into smaller components. For example, `Age` is a simple attribute.
2. **Composite Attribute:** An attribute that can be subdivided into smaller, basic attributes. For example, `Address` can be a composite attribute composed of `Street`, `City`, and `ZipCode`.
3. **Single-valued Attribute:** An attribute that can hold only one value for each entity instance. For example, `StudentID` is single-valued because a student has only one ID.
4. **Multi-valued Attribute:** An attribute that can hold multiple values for a single entity instance. For example, `PhoneNumber` could be a multi-valued attribute if a student can have multiple phone numbers. It is represented by a double ellipse in an ER-diagram.
5. **Derived Attribute:** An attribute whose value is calculated or derived from another attribute. For example, `Age` can be derived from the `DateOfBirth` attribute. It is represented by a dashed or dotted ellipse.
6. **Key Attribute:** An attribute (or a set of attributes) whose values are unique for each entity instance in an entity set. It is used to identify an entity uniquely. The key attribute is often underlined in an ER-diagram.

Relationship

A relationship represents an association between two or more entities. For example, a `Student` entity is associated with a `Course` entity through an "enrolls in" relationship. A **Relationship Set** is a collection of similar relationships.

Cardinality

Cardinality, also known as cardinality ratio, defines the number of instances of one entity that can be associated with instances of another entity through a specific relationship. It specifies the minimum and maximum number of relationship instances an entity can participate in. The main types of cardinality ratios are:

1. **One-to-One (1:1):** One instance of entity A can be associated with at most one instance of entity B, and one instance of entity B can be associated with at most one instance of entity A.

2. **One-to-Many (1:N):** One instance of entity A can be associated with zero, one, or many instances of entity B, but one instance of entity B can be associated with at most one instance of entity A.
3. **Many-to-One (N:1):** Many instances of entity A can be associated with at most one instance of entity B, but one instance of entity B can be associated with zero, one, or many instances of entity A.
4. **Many-to-Many (M:N):** One instance of entity A can be associated with many instances of entity B, and one instance of entity B can also be associated with many instances of entity A.

Generalization

Generalization is a **bottom-up** approach in which two or more entities with common attributes are combined to form a more general, higher-level entity. The common attributes and relationships of the lower-level entities are abstracted into the higher-level entity (called a superclass or parent). The lower-level entities (called subclasses or child entities) inherit all the attributes and relationships of the superclass.

This process helps in reducing redundancy by identifying and grouping common features. For example, the entities `Car` and `Truck` can both be generalized into a higher-level entity called `Vehicle`. Both cars and trucks have attributes like `Model`, `Year`, and `Price`, which would belong to the `Vehicle` superclass.

Specialization

Specialization is a **top-down** approach, the reverse of generalization. In this process, a higher-level entity is divided into two or more lower-level, more specific entities. The superclass is broken down into subclasses based on some distinguishing characteristics.

While the subclasses inherit all the attributes and relationships from the superclass, each subclass may also have its own unique attributes or relationships that are not shared by other subclasses. For example, the entity `Account` can be specialized into `Savings_Account` and `Current_Account`. Both will inherit the `AccountNumber` and `Balance` attributes from `Account`, but `Savings_Account` might have a specific attribute like `InterestRate`, while `Current_Account` might have an `OverdraftLimit`.

Relational Model

The Relational Model represents a database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship. The table name and column names are helpful to interpret the meaning of values in each row.

The main components of the Relational Model are:

- **Relation:** A table with columns and rows.
- **Tuple:** A single row of a table, which contains a single record for that relation.
- **Attribute:** A column of a table, which represents a property of the entity.
- **Domain:** The set of permissible values for an attribute. For example, the domain for a `Month` attribute could be the set {Jan, Feb, ..., Dec}.
- **Degree:** The total number of attributes (columns) in a relation.
- **Cardinality:** The total number of tuples (rows) in a relation.

Relational Model Keys

Keys are a crucial part of the relational model. They are an attribute or a set of attributes used to identify, access, and establish relationships between tables.

- **Super Key:**

  A set of one or more attributes that, taken collectively, can uniquely identify a tuple (row) in a relation. There can be multiple super keys in a relation.

- **Candidate Key:**

  A minimal super key. It's a super key from which no attribute can be removed without losing its uniqueness property. A relation can have one or more candidate keys.

- **Primary Key:**

  A candidate key that is selected by the database designer to uniquely identify tuples in a table. A primary key value cannot be NULL and must be unique for each tuple. Each table can have only one primary key.

- **Alternate Key:**

  The candidate keys that are not chosen as the primary key are known as alternate keys.

- **Foreign Key:**

  An attribute or a set of attributes in a table whose values are required to match the primary key of another table. It is used to link two tables together and enforce referential integrity. A foreign key can have a NULL value.

- **Composite Key:**

  A key that consists of two or more attributes that together uniquely identify a record. This is used when a single attribute is not sufficient to uniquely identify a tuple.

Integrity Constraints

Integrity constraints are a set of rules applied to the database to ensure that the data is accurate, consistent, and reliable. They prevent accidental damage to the database from authorized users.

1. **Domain Constraint:**
    - This constraint specifies that the value taken by any attribute must be an atomic value from its defined domain.
    - It defines the data type, format, and range of values that an attribute can hold. For example, an `Age` attribute might be constrained to be an integer between 0 and 120.
2. **Entity Integrity Constraint:**
    - This constraint states that the primary key of a table cannot contain a `NULL` value.
    - This is because the primary key is used to uniquely identify individual tuples in a relation; if it were `NULL`, we wouldn't be able to identify some tuples.
3. **Referential Integrity Constraint:**
    - This constraint is specified between two tables and is maintained using a foreign key.
    - It states that for any foreign key value in the referencing table, that same value must exist in the primary key of the referenced table, or the foreign key value must be `NULL`.
    - This ensures that a relationship between two tables remains valid and prevents records that reference non-existent records.
4. **Key Constraint:**
    - This constraint dictates that all tuples in a relation must be unique. No two tuples can have the same value for all their attributes.
    - This is enforced by requiring every relation to have at least one candidate key, whose values must be unique for every tuple.

SQL

SQL, which stands for **Structured Query Language**, is a standard programming language used for managing and manipulating data held in a relational database management system (RDBMS). It is used to perform tasks such as retrieving, updating, inserting, and deleting data in a database. SQL is a declarative language, meaning you specify what you want to do, and the DBMS figures out how to do it.

SQL Command Categories

SQL commands are divided into several categories based on their functionality.

1. DDL (Data Definition Language) Used to define and manage the structure of database objects.

- **`CREATE`**: Creates a new table, database, or view.
- **`ALTER`**: Modifies the structure of an existing database object.
- **`DROP`**: Deletes a database, table, or view.
- **`TRUNCATE`**: Removes all records from a table quickly.

2. DQL (Data Query Language) Used to retrieve data from the database.

- **`SELECT`**: The primary command used to extract data from one or more tables.

3. DML (Data Manipulation Language) Used to manage the data within database objects.

- **`INSERT`**: Adds new rows of data into a table.
- **`UPDATE`**: Modifies existing data within a table.
- **`DELETE`**: Removes one or more rows from a table.

4. DCL (Data Control Language) Used to control access to data in the database.

- **`GRANT`**: Gives users permissions to the database.
- **`REVOKE`**: Removes user permissions.

5. TCL (Transaction Control Language) Used to manage transactions in the database.

- **`COMMIT`**: Saves all the work done in a transaction.
- **`ROLLBACK`**: Undoes changes made in the current transaction.
- **`SAVEPOINT`**: Sets a point within a transaction to which you can later roll back.

SQL Clauses

Clauses are keywords used with SQL statements to perform various operations like filtering, sorting, and grouping.

- **`FROM`**: Specifies the table(s) from which to retrieve the data.
- **`WHERE`**: Filters records based on a specified condition.
    - `SELECT * FROM Employees WHERE Department = 'HR';`
- **`ORDER BY`**: Sorts the result set in ascending (`ASC`) or descending (`DESC`) order.
    - `SELECT * FROM Employees ORDER BY Salary DESC;`
- **`GROUP BY`**: Groups rows that have the same values in specified columns into summary rows. It is often used with aggregate functions.
    - `SELECT Department, COUNT(*) FROM Employees GROUP BY Department;`
- **`HAVING`**: Filters the results of a `GROUP BY` based on a specified condition. The `WHERE` clause filters rows before grouping, while `HAVING` filters groups after they are created.

    - `SELECT Department, AVG(Salary) FROM Employees GROUP BY Department HAVING AVG(Salary) > 50000;`

SQL Joins

Joins are used to combine rows from two or more tables based on a related column between them.

- **`INNER JOIN`**: Returns records that have matching values in both tables.
- **`LEFT JOIN` (or `LEFT OUTER JOIN`)**: Returns all records from the left table and the matched records from the right table. If there is no match, the result is `NULL` from the right side.
- **`RIGHT JOIN` (or `RIGHT OUTER JOIN`)**: Returns all records from the right table and the matched records from the left table. If there is no match, the result is `NULL` from the left side.
- **`FULL OUTER JOIN`**: Returns all records when there is a match in either the left or the right table. It combines the results of both `LEFT JOIN` and `RIGHT JOIN`.
- **`SELF JOIN`**: A join where a table is joined with itself. This is useful for querying hierarchical data or comparing rows within the same table.
- **`CROSS JOIN`**: Produces the Cartesian product of the two tables involved in the join, returning all possible combinations of rows.

SQL Aggregate Functions

Aggregate functions perform a calculation on a set of rows and return a single, summary value.

- **`COUNT()`**: Returns the number of rows.
- **`SUM()`**: Returns the total sum of a numeric column.
- **`AVG()`**: Returns the average value of a numeric column.
- **`MIN()`**: Returns the minimum value in a set.
- **`MAX()`**: Returns the maximum value in a set.

Subquery (or Nested Query)

A subquery is an SQL query nested inside a larger query. The subquery is executed first, and its result is used by the outer query. Subqueries can be used in `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements, as well as inside other subqueries.

Example: `SELECT EmployeeName FROM Employees WHERE Salary > (SELECT AVG(Salary) FROM Employees);`

SQL Indexes

An index is a special lookup table that the database search engine can use to speed up data retrieval. An index is a pointer to data in a table. Indexes are created on columns that are frequently used in `WHERE` clauses or `JOIN` conditions. While they speed up `SELECT` queries, they can slow down data modification operations (`INSERT`, `UPDATE`, `DELETE`) because the indexes also need to be updated.

Functional Dependency (FD)

A functional dependency is a constraint between two sets of attributes in a relational database. It describes the relationship between attributes in a relation.

For a relation (table) `R`, an attribute `B` is said to be functionally dependent on an attribute `A` if each value of `A` is associated with exactly one value of `B`. This is denoted as A→B, which reads "A determines B".

- **Determinant:** The attribute on the left side of the arrow (e.g., `A`) is called the determinant.
- **Dependent:** The attribute on the right side of the arrow (e.g., `B`) is called the dependent.

For example, in a `Students` table with columns `StudentID`, `StudentName`, and `Email`, there is a functional dependency StudentID→StudentName. This is because for any given `StudentID`, there can only be one corresponding `StudentName`.

Normalization

Normalization is the process of organizing the columns (attributes) and tables (relations) of a database to minimize data redundancy and improve data integrity. The main goal is to decompose larger, problematic tables into smaller, well-structured tables and define relationships between them.

The primary objectives of normalization are:

- **Eliminate Data Redundancy:** Storing the same piece of data in multiple places is avoided.
- **Prevent Data Anomalies:** It helps resolve issues that can occur during data modification, specifically:
    - **Insertion Anomaly:** Difficulty in inserting a new record because some data is missing.
    - **Deletion Anomaly:** The unintentional loss of data when a record is deleted.
    - **Update Anomaly:** Inconsistencies that arise from having to update the same data in multiple locations.
- **Ensure Data Dependencies are Logical:** It ensures that data is stored in the correct table.

Normal Forms

Normal forms are a series of stages or guidelines used in the normalization process. A database is said to be in a certain normal form if it satisfies a specific set of constraints.

1. First Normal Form (1NF) A relation is in 1NF if it meets the following conditions:

- It contains only atomic (indivisible) values.
- Each attribute (column) must have a unique name.
- The order in which data is stored does not matter.

This means there should be no repeating groups or multi-valued columns. For instance, a `PhoneNumber` column should not contain multiple numbers in a single cell.

2. Second Normal Form (2NF) For a relation to be in 2NF, it must:

- Be in 1NF.
- Have no partial dependencies. A partial dependency exists when a non-prime attribute (an attribute that is not part of any candidate key) is functionally dependent on only a part of a composite primary key.

2NF is only relevant for tables with a composite primary key (a primary key made of two or more attributes).

3. Third Normal Form (3NF) For a relation to be in 3NF, it must:

- Be in 2NF.
- Have no transitive dependencies. A transitive dependency occurs when a non-prime attribute is functionally dependent on another non-prime attribute. In other words, if A→B and B→C are two FDs, then A→C is a transitive dependency, where `A` is the primary key.

To achieve 3NF, you must eliminate fields that do not depend on the primary key.

4. Boyce-Codd Normal Form (BCNF) BCNF is a stricter version of 3NF. For a relation to be in BCNF, it must:

- Be in 3NF.
- For every non-trivial functional dependency X→Y, X must be a superkey.

This means that the determinant of every functional dependency must be a candidate key or a superkey. BCNF resolves certain anomalies that 3NF does not handle, especially in cases with multiple overlapping candidate keys.

Transaction

A transaction is a single logical unit of work that consists of a sequence of one or more database operations, such as reads, writes, updates, or deletes. A transaction is treated as an indivisible and atomic unit. This means that either all of the operations within the transaction are completed successfully and their changes are permanently saved (committed) to the database, or none of the operations are applied and the database remains unchanged (rolled back).

ACID Properties

ACID is an acronym for a set of four properties that guarantee the reliability of database transactions.

1. Atomicity:
    - This property ensures that a transaction is treated as a single, indivisible unit. It either executes completely or not at all. If any operation within the transaction fails, the entire transaction is rolled back, and the database is left in the state it was in before the transaction started. This is often referred to as the "all or nothing" rule.
2. Consistency:
    - This property ensures that a transaction brings the database from one valid state to another. The data must remain consistent with all defined rules, constraints, triggers, and cascades. If a transaction would result in an invalid state, it is rolled back to maintain the integrity of the database.
3. Isolation:
    - This property ensures that the execution of concurrent transactions does not interfere with each other. The intermediate state of a transaction is not visible to other transactions. From any given transaction's perspective, it appears as if it is the only transaction executing on the system. This prevents issues like dirty reads, non-repeatable reads, and phantom reads.
4. Durability:
    - This property guarantees that once a transaction has been successfully committed, its effects are permanent and will persist even in the event of a system failure, such as a power outage or crash. The committed changes are stored in non-volatile memory.

Transaction States

During its execution, a transaction passes through several states:

- **Active:** The initial state where the transaction is executing. The database operations (read, write) are being performed.
- **Partially Committed:** The state after the final statement of the transaction has been executed. The transaction has finished its execution, but the changes are still stored in temporary buffers in main memory and have not yet been permanently written to the database on disk.

- **Committed:** The state after a transaction has completed successfully. All its changes have been permanently recorded to the database.
- **Failed:** The state a transaction enters if it cannot proceed due to some error or hardware failure.
- **Aborted (Rolled Back):** The state after a transaction has failed and all its changes have been undone. The database is restored to its state before the transaction began.
- **Terminated:** The final state of a transaction. It is reached after the transaction is either `Committed` or `Aborted`.

Indexing

Indexing is a database performance-tuning technique used to speed up data retrieval operations. An index is a special data structure that provides a quick lookup of data in a table based on the values in one or more columns.

It works similarly to an index in a book. Instead of scanning the entire table row by row (a full table scan) to find the desired data, the database can use the index to directly locate the corresponding rows. The index stores the column values and a pointer to the physical address of the record on the disk.

Key characteristics of indexing:

- **Faster Retrieval:** Drastically reduces the time it takes to execute `SELECT` queries with a `WHERE` clause on indexed columns.
- **Slower Modifications:** Slows down data modification operations (`INSERT`, `UPDATE`, `DELETE`) because every time data is modified, the index must also be updated.
- **Additional Storage:** Indexes require additional disk space to store the index structure.

Common types of indexes:

- **Clustered Index:** Determines the physical order of data in a table. A table can have only one clustered index.
- **Non-Clustered Index:** Has a logical order that is separate from the physical storage of the rows. A table can have multiple non-clustered indexes.

SQL vs. NoSQL Comparison

Feature	SQL (Relational Databases)	NoSQL (Non-Relational Databases)
Full Name	Structured Query Language	Not Only SQL
Data Model	Data is stored in a relational model, organized into tables with rows and columns. Relationships are established via foreign keys.	Data is stored in non-relational models. Common types are Document, Key-Value, Wide-Column, and Graph.

Schema	Requires a pre-defined, fixed schema (schema-on-write). The structure of the data must be known in advance.	Typically uses a dynamic or flexible schema (schema-on-read). The structure is not rigid and can evolve.
Scalability	Primarily scales **vertically**. To handle more load, you increase the resources (CPU, RAM, SSD) of a single server.	Primarily scales **horizontally**. To handle more load, you add more servers to a cluster (distributed architecture).
Query Language	Uses the standard SQL (Structured Query Language) for defining and manipulating the data.	Each database has its own query language or API. The query language is not standardized across different NoSQL databases.
Consistency Model	Follows the **ACID** properties (Atomicity, Consistency, Isolation, Durability), which guarantees high consistency and reliability.	Generally follows the **BASE** model (Basically Available, Soft State, Eventual Consistency), which prioritizes high availability and scalability over strict consistency.
Data Integrity	Enforces strong data integrity and consistency through constraints like Primary Keys, Foreign Keys, and transaction management.	Data consistency and integrity are often handled at the application layer. Offers eventual consistency rather than strong consistency.
Use Cases	Ideal for applications that require complex queries, multi-row transactions, and high data integrity. Examples: banking systems, e-commerce platforms, and traditional data warehousing.	Best suited for handling large volumes of unstructured or semi-structured data, big data applications, and real-time systems. Examples: social media feeds, IoT data, and content management systems.
Examples	MySQL, PostgreSQL, Oracle, Microsoft SQL Server, SQLite	MongoDB (Document), Redis (Key-Value), Apache Cassandra (Wide-Column), Neo4j (Graph)

Types of databases

1. Classification Based on Data Model

This is the most common way to classify databases. It describes how the data is logically structured, organized, and manipulated.

a. Relational Databases (SQL)

These databases are based on the relational model. Data is stored in a structured format using tables with rows (tuples) and columns (attributes). Relationships between different tables are established using foreign keys. They use Structured Query Language (SQL) for querying and managing data.

- **Best for:** Structured data, applications requiring ACID compliance (e.g., transactional systems, financial applications).
- **Examples:** MySQL, PostgreSQL, Microsoft SQL Server, Oracle.

b. Non-Relational Databases (NoSQL)

These databases do not use the traditional table-based relational model. They are designed for unstructured or semi-structured data and are built for scalability, flexibility, and high performance. There are several types of NoSQL databases:

- **Document Databases:**
  - Store data in document-like structures, such as JSON or BSON. Each document contains key-value pairs. They are flexible and allow for a hierarchical data structure.
  - **Best for:** Content management systems, mobile app data, semi-structured data.
  - **Example:** MongoDB.
- **Key-Value Stores:**
  - The simplest type of NoSQL database. Data is stored as a collection of key-value pairs. This model is highly efficient for read/write operations.
  - **Best for:** Caching, session management, real-time applications.
  - **Example:** Redis, Amazon DynamoDB.
- **Wide-Column Stores:**
  - Store data in tables, but the columns are dynamic and not pre-defined for each row. They are optimized for queries over large datasets.
  - **Best for:** Big data applications, IoT data, real-time analytics.
  - **Example:** Apache Cassandra, HBase.
- **Graph Databases:**
  - Designed to store and navigate relationships between data entities. They use nodes (to store entities) and edges (to store relationships between entities).
  - **Best for:** Social networks, recommendation engines, fraud detection.
  - **Example:** Neo4j.

2. Classification Based on Architecture

a. Centralized Database

A centralized database is one in which all data is stored and maintained in a single, central location (e.g., a single server or mainframe computer). Users from different locations can access this data, but it presents a single point of failure.

b. Distributed Database

A distributed database consists of multiple, logically interrelated databases that are spread across various physical locations and connected via a network. The system synchronizes the data, making it appear to the user as a single database. This improves reliability and availability.

3. Other Common Types

- Cloud Databases:
    - A database that is built and accessed through a cloud platform (e.g., AWS, Google Cloud, Azure). They offer scalability, high availability, and are managed by the cloud provider (offered as a DBaaS - Database-as-a-Service). They can be either SQL or NoSQL.
    - **Examples:** Amazon RDS, Azure SQL, Google Cloud Spanner.
- Object-Oriented Databases:
    - Based on object-oriented programming (OOP) concepts, this type of database stores data in the form of objects. It is useful for applications that deal with complex data objects.
- Hierarchical Databases:
    - An early database model where data is organized in a tree-like structure. Each record has a single parent, and the relationship is one-to-many. It is less flexible than the relational model.

Database Scaling

Database scaling is the process of increasing the capacity of a database system to handle growth. This growth can be in the amount of data stored (volume) or the number of concurrent requests (load). There are two primary methods for scaling a database:

1. Vertical Scaling (Scaling Up)

- **What it is:** Vertical scaling involves increasing the resources of a single server. This means adding more powerful hardware, such as a faster CPU, more RAM, or faster storage (SSDs).
- **Analogy:** It's like upgrading a single car with a bigger engine.
- **Pros:** It is relatively simple to implement as the application and database logic remain the same.
- **Cons:**
    - **Limits:** There is a physical limit to how much you can upgrade a single server.
    - **Cost:** High-end hardware can be very expensive.
    - **Single Point of Failure:** If the single server fails, the entire database goes down.

2. Horizontal Scaling (Scaling Out)

- **What it is:** Horizontal scaling involves distributing the load across multiple servers (also known as nodes). Instead of making one server more powerful, you add more servers to the database cluster.
- **Analogy:** It's like adding more cars to a fleet to handle more passengers.
- **Pros:**
    - **High Scalability:** You can add more servers as needed, providing virtually limitless scalability.
    - **Fault Tolerance:** If one server fails, the others can continue to operate, providing higher availability.
    - **Cost-Effective:** It often uses commodity hardware, which can be cheaper than high-end single servers.
- **Cons:** It adds architectural complexity for distributing data and queries and maintaining consistency across nodes. Partitioning and sharding are techniques used to achieve horizontal scaling.

Partitioning

Partitioning is the process of splitting a very large database table into smaller, more manageable pieces called partitions. **Crucially, all these partitions are stored on a single database server.** The database system is aware of the partitions and can route queries to the correct partition, improving performance.

Types of Partitioning:

- **Horizontal Partitioning:** Divides a table by rows. The table is split into multiple partitions, where each partition has the same columns but contains a different subset of rows based on a "partition key."
    - **Example:** An `Orders` table could be partitioned by year, so all 2024 orders are in one partition, and all 2025 orders are in another.
- **Vertical Partitioning:** Divides a table by columns. The table is split into partitions that have the same number of rows but fewer columns. This is often done to separate frequently accessed columns from less frequently accessed or large columns.
    - **Example:** A `UserProfile` table might be vertically partitioned to store frequently accessed data like `UserID` and `Username` in one partition, and large, less-frequently accessed data like `ProfilePicture_BLOB` in another.

Sharding

Sharding is a specific type of database partitioning that separates a large database into smaller, faster, more easily managed parts called **shards**. The key difference from partitioning is that

these shards are spread across multiple database servers. Each server operates as an independent database, holding its own shard of the data.

Sharding is the primary method for achieving horizontal scaling.

Why use Sharding?

- **Scalability:** It allows a database to scale beyond the resource limits of a single server.
- **High Availability:** If one shard (server) goes down, the other shards are unaffected, so the entire database is not offline.
- **Improved Performance:** Queries are distributed across multiple servers, reducing the load on any single machine and improving response times.

How it works: Like horizontal partitioning, sharding uses a "shard key" to determine which server a specific row of data should be stored on. The application or a routing layer is responsible for directing queries to the correct shard.

MySQL

MySQL is one of the world's most popular open-source **relational database management systems (RDBMS)**. It is a client-server based system.

- **Database Model**: **Relational**. Data is organized into tables consisting of rows and columns. It uses foreign keys to enforce relationships between tables.
- **Schema**: It has a **pre-defined, fixed schema** (schema-on-write). You must define the table structure (columns and their data types) before you can insert data.
- **Language**: It uses **SQL (Structured Query Language)** for all operations.
- **Key Features**:
    - **ACID Compliant**: Provides strong support for transactions with Atomicity, Consistency, Isolation, and Durability.
    - **Client-Server Architecture**: The database runs as a separate server process that clients connect to over a network.
    - **Mature and Reliable**: It is well-established, with a large community and extensive documentation.
    - **Platform Independent**: Runs on various operating systems like Windows, Linux, and macOS.
- **Typical Use Cases**: It is a cornerstone of the **LAMP stack** (Linux, Apache, MySQL, PHP/Python/Perl) and is widely used for **web applications**, **e-commerce platforms**, and **transactional systems**.

MongoDB

MongoDB is a popular open-source **NoSQL database** designed for scalability and handling large volumes of unstructured or semi-structured data.

- **Database Model**: **Document-oriented**. Data is stored in flexible, JSON-like documents called BSON (Binary JSON). This allows for nested data structures and arrays.
- **Schema**: It has a **dynamic schema** (schema-on-read). Documents in the same collection do not need to have the same set of fields, and the data structure can be changed at any time.
- **Language**: It uses the **MongoDB Query Language (MQL)**, which has a syntax similar to JavaScript/JSON.
- **Key Features**:
    - **Horizontal Scalability**: Designed to scale out by distributing data across multiple servers (**sharding**).
    - **High Availability**: Provides high availability through a feature called **replica sets**, which are multiple copies of the data.
    - **Flexible Data Model**: Ideal for applications with evolving requirements.
    - **High Performance**: Delivers high performance for read and write operations, especially for large datasets.
- **Typical Use Cases**: Best suited for **big data applications**, **content management systems**, **real-time analytics**, **IoT**, and **mobile applications**.

SQLite

SQLite is a C-language library that implements a self-contained, serverless, zero-configuration, transactional **SQL database engine**. It is the most widely deployed database engine in the world.

- **Database Model**: **Relational**. Like MySQL, it stores data in tables with rows and columns.
- **Schema**: It uses a **pre-defined schema**, though it is more flexible with data types than other RDBMSs (a concept called manifest typing).
- **Language**: It uses a standard dialect of **SQL**.
- **Key Features**:
    - **Serverless**: It does not have a separate server process. The database engine runs in the same process as the application.
    - **Self-Contained**: The entire database (definitions, tables, indexes, and data) is stored as a single cross-platform file on a host machine.
    - **Zero-Configuration**: No setup or administration is needed.
    - **Transactional**: It is fully ACID compliant, ensuring reliable transactions.
    - **Lightweight**: It has a very small memory footprint and library size.

- **Typical Use Cases**: Perfect for **mobile applications** (Android and iOS), **desktop software**, **embedded systems**, and as a local data cache for web browsers. It is not suitable for high-concurrency, client-server applications.