DBMS

Introduction

A Database Management System (DBMS) is a software system that is designed to manage and organize **data** in a structured manner within a **database.** It allows users to **create, modify, and query a database**, as well as manage the security and access controls for that database.

Key Features of DBMS

1. **Data modeling:** A DBMS provides tools for creating and modifying data models, which define the structure and relationships of the data.
2. **Data storage and retrieval:** A DBMS is responsible for storing and retrieving data from the database and can provide various methods for searching and querying the data.
3. **Concurrency control:** A DBMS provides mechanisms for controlling concurrent access to the database, to ensure that multiple users can access the data without conflicting with each other.
4. **Data integrity and security:** A DBMS provides tools for enforcing data integrity and security constraints, such as constraints on the values of data and access controls that restrict who can access the data.
5. **Backup and recovery:** A DBMS provides mechanisms for backing up and recovering the data in the event of a system failure.

Classification of DBMS

Relational Database Management System (RDBMS)

- Organizes data in tables with rows and columns.
- Establishes relationships between tables using primary and foreign keys.

Non-Relational Database Management System (NoSQL)

- Organizes data in diverse structures like key-value pairs, documents, graphs, or column-based formats.
- Tailored to handle large-scale, high-performance scenarios efficiently.

History of DBMS

- In the **1950s and early 1960s**, data storage and processing were primarily handled using **magnetic tapes and punched card decks.** Data processing tasks, such as payroll, involved sequentially reading data from tapes or card decks, making the process time-consuming and rigid. Accessing data directly was not possible due to the sequential nature of tapes.
- During the early 1960s, **Charles Bachman** developed the **Integrated Data Store (IDS),** one of **the first database management systems** based on the network data model. This significant advancement earned him the **Turing Award.** IDS allowed more complex data relationships and laid the groundwork for future DBMS developments.
- In the **late 1960s, IBM** introduced the **Integrated Management System**, based on the **hierarchical data model**. This period also saw the introduction of hard disks, allowing direct data access and enabling more flexible and efficient data management, marking a technological shift.
- **The 1970s** marked a revolutionary change with **Edgar Codd's** introduction of the **relational database model.** This model simplified data management and querying by organizing data into tables with rows and columns. Its simplicity and efficiency quickly made it the standard for database systems.
- During **the 1980s, IBM developed the Structured Query Language (SQL)** as part of the System R project, which became the standard **language for managing and querying relational databases**. This decade saw the emergence of commercial relational databases like IBM DB2, Oracle, and Ingres, which eventually replaced older network and hierarchical models.
- In the early **1990s**, the focus shifted to **decision support and querying applications.** Tools for analyzing large datasets became increasingly important, and parallel database products were introduced to handle the growing data volumes. Databases also began incorporating object-relational support to enhance functionality.

- The **1990s** witnessed the explosive growth of the World Wide Web, significantly **impacting database deployment.** Databases needed to support high transaction-processing rates, reliability, and 24/7 availability. Web interfaces to databases became essential to meet the demands of web applications.
- In the **2000s**, **XML** and **XQuery emerged** for complex data types and data exchange, while relational databases remained core for large-scale applications. Open-source databases like **PostgreSQL and MySQL** saw significant growth. Specialized databases such as column-stores for data analysis and highly parallel systems were developed. Distributed data-storage systems for large web platforms like Amazon and Google also emerged, along with advancements in streaming data management and data-mining techniques.

Different Types of DBMS

There are **various types of database management systems** based on database structures. We can arrange data in various formats for a variety of use cases. Let's see these types of DBMS one by one:

Centralized DBMS

In a centralized database, a single central database is used to serve data to multiple devices. Each user can access the database after authentication and be able to work with it.

Decentralized DBMS

In the decentralized database, all the data is collectively stored in multiple databases. All these databases are connected with the help of networking. To the end-user, this entire system appears like a single coherent system.

Relational DBMS

The relational database management system is also known as RDBMS. It is one of the types of DBMS which is widely used for commercial applications. It contains tables in which data is stored in the form of rows and columns like an Excel sheet. Some of the tables possess the relationship among them and the data is retrieved with the help of join operation. This join operation helps us to get data from 2 or more tables with the help of logical queries.

NoSQL DBMS

NoSQL or non-relational databases are the most popular databases due to their high scalability and availability. In this type of database, the data is stored in collections, and it doesn't contain tables like relational databases.

Collection is simply the group of documents in which we have data with similar meanings and similar purposes.

In NoSQL, we can store data in key-value pairs as well as with the help of graphs. It increases productivity by a significant amount and comparatively, it is easy to work with them.

Hierarchical DBMS

In hierarchical databases, data is arranged in a tree-like format where we have a parent-child relationship between nodes. The parent can have many children, but children contain only one parent.

Network DBMS

The network database model has various nodes, and these nodes are connected with each other. These models are complex in nature. This model allows multiple parents for a single child node so we can create more complicated structures with it.

Object-Oriented DBMS

The Object-oriented database management system is one of the types of DBMS, in which data is stored in the objects. These objects are created from the classes. Classes are nothing but the description of an object. It is like object-oriented programming languages.

Some Popular Database Management Systems

There are several popular **Database Management Systems (DBMS)** that cater to different needs and preferences. Some widely used **DBMS** are:

1. MySQL:

- An **open-source relational database management system (RDBMS).**
- Known for its reliability, ease of use, and strong community support.
- Frequently used for web applications and small to medium-sized databases.

2. PostgreSQL:

- An **open-source object-relational database system.**
- Emphasizes extensibility and standards compliance.
- Suitable for complex applications and large-scale databases.

3. Microsoft SQL Server:

- A **relational database management system** developed by **Microsoft**.
- Offers a comprehensive suite of features and tools for enterprise-level applications.
- Commonly used in conjunction with Microsoft's .NET framework.

4. Oracle Database:

- A powerful and widely used **relational database management system.**
- Known for its scalability, security features, and support for complex transactions.
- Popular in large enterprises and critical business applications.

5. MongoDB:

- A leading **NoSQL database management system.**
- Stores data in flexible, JSON-like documents in a schema-less fashion.
- Ideal for handling large amounts of unstructured or semi-structured data.

6. SQLite:

- A self-contained, serverless, and zero-configuration **relational database engine.**
- Lightweight and suitable for embedded systems, mobile applications, and small-scale deployments.

7. Redis:

- An **in-memory data structure store** that is often used as a cache or message broker.
- Provides high-performance data storage and retrieval for key-value pairs. Commonly used in real-time applications as a caching mechanism.

Advantages of DBMS

1. **Security and Reliability:** DBMS ensures secure data storage through authentication and user authorization.
2. **Data Redundancy Reduction:** Normalization techniques help minimize and remove data redundancy.
3. **Multiple Data Views:** Provides different data views tailored for different users' needs.
4. **Backup and Recovery:** Facilitates data backup and recovery to prevent data loss.
5. **Integration with Programming Languages:** Can be integrated with languages like Python and Java to enhance database functionalities.

Disadvantages of DBMS

1. **Complexity:** DBMS systems can be complex to work with and manage.
2. **Cost of Hardware:** Involves significant cost for purchasing necessary hardware for data storage.
3. **Setup Time:** Setting up a DBMS can be time-consuming.
4. **Licensing Costs:** Many commercial DBMS products require paid licenses.
5. **Need for Skilled Staff:** Requires skilled technical staff, adding to the operational costs.

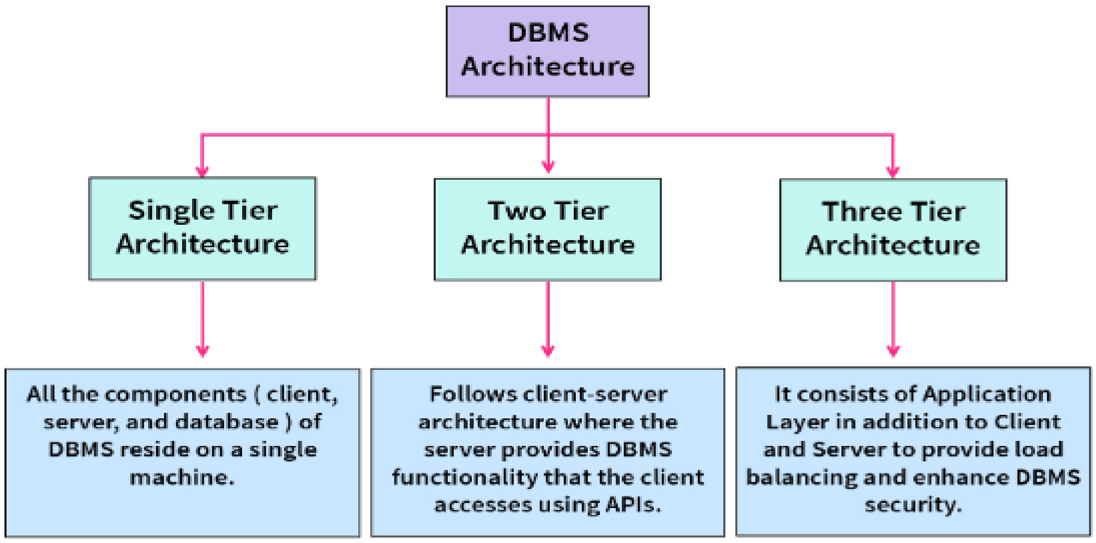
DBMS Architecture Types

Database management systems (DBMS) are organized into **multiple levels of abstraction** to ensure proper functioning. These layers describe both the design and the operations of the DBMS, facilitating a structured approach to database management. A **DBMS** is **not always directly accessible** by users or applications; instead, **various architectures** are employed based on how users connect to the database. These architectures are classified into **tiers**, defining the number of layers in the DBMS structure.

An **n-tier DBMS architecture** divides the entire DBMS into related but independent **layers**. For example, a **one-tier architecture** has a **single layer**, a **two-tier architecture** has **two layers**, and a **three-tier architecture** has **three layers.** As the **number of layers increases**, so does the **level of abstraction**, enhancing both the security and complexity of the DBMS. Importantly, these layers are independent of each other, meaning changes in one layer do not impact others, allowing for **modular** and **flexible** system management.

Now, let's look at the most common DBMS architectures:

- *Single Tier Architecture (One-Tier Architecture)*
- *Two-Tier Architecture*
- *Three-Tier Architecture*


DBMS Architecture
Single Tier Architecture
Two Tier Architecture
Three Tier Architecture
All the components (client, server, and database) of DBMS reside on a single machine.
Follows client-server architecture where the server provides DBMS functionality that the client accesses using APIs.
It consists of Application Layer in addition to Client and Server to provide load balancing and enhance DBMS security.

1-Tier Architecture:

Definition: In a 1-tier architecture, the database is directly accessible by the user or application without any intermediary layers.

Explanation: The user interacts directly with the database. All the data, business logic, and presentation logic are handled within a single layer, often on a single machine.

Example: When learning SQL, you set up an SQL server and a database on your local system. This setup allows you to interact directly with the relational database and execute operations without a network connection. This direct interaction on a single machine is an example of 1-Tier DBMS architecture.

2-Tier Architecture:

Definition: A 2-tier architecture consists of two layers: the client (user interface) and the server (database).

Explanation: In this architecture, the client communicates directly with the database server. The business logic and database management are handled on the server side, while the user interface runs on the client side.

Example: When you visit a bank to withdraw cash, the banker enters your withdrawal amount and account details into a system. The client-side application then communicates with the server-side database to check your account balance. This interaction between the client application and the server database is an example of 2-Tier DBMS architecture.

3-Tier Architecture

Definition: A 3-tier architecture includes three layers: the presentation layer (user interface), the application layer (business logic), and the database layer.

Explanation: The user interface interacts with the application server, which in turn communicates with the database server. This separation allows for more scalable and manageable systems, where the business logic is handled separately from data storage and user interaction.

Example: A web application, such as an online shopping site. The user's browser (presentation layer) interacts with a web server (application layer) that processes request and communicates with a database server (database layer) to fetch and store data.

Database

A **database** is an **organized collection of data** that is stored and **accessed electronically.** Databases are designed to efficiently **store, retrieve, and manage data.** They can be structured in various ways, such as in **tables (relational databases) or as documents (NoSQL databases).** A database typically supports operations like querying, updating, and managing data to ensure its integrity, security, and availability.

When we say a database is **"electronically accessed,"** it means that the data within the database can be **retrieved, updated, managed, and manipulated** using **electronic devices, typically computers.** This access is facilitated through various software tools and interfaces, allowing users and applications to interact with the data without needing to handle the physical storage media directly.

Why Use a Database?

Efficient Data Storage: Databases can store vast amounts of records effectively, ensuring data is organized and easily retrievable.

Quick Data Retrieval: Locating data in a database is fast and straightforward, enhancing productivity and decision-making.

Ease of Data Modification: Adding, updating, or deleting data in a database is simple, allowing for flexible data management.

Advanced Search Capabilities: Techniques like indexing and binary searching make it easy to search for specific data within a database.

Data Sorting and Importing: Databases enable quick and easy sorting of data and seamless import into other applications.

Multi-Access: Multiple users can access and use the same database simultaneously, promoting collaboration and efficiency.

Enhanced Security: Databases offer robust security measures, providing better protection for data compared to physical paper files.

Transaction Management: Databases ensure consistency and accuracy during transactions, maintaining data integrity.

Evolution of Databases

The history of databases spans over 50 years, evolving through several key stages:

Navigational Databases: Early systems like hierarchical databases (tree-like structure) and network databases (flexible relationship model) were the first to manage data.

Relational Databases: Gained popularity in the 1980s, offering more flexibility and efficiency in data management.

Object-Oriented Databases: Emerged in the 1990s, integrating object-oriented programming concepts with database systems.

NoSQL Databases: Developed in response to the need for faster processing of unstructured data due to the expansion of the internet.

Modern Databases: Cloud databases and self-driving databases are now used for faster processing and cloud-based data storage, reflecting the ongoing evolution of database technology.

Components of a Database

Databases are comprised of five key components, each playing a critical role in the DBMS environment:

Hardware: Physical devices like I/O devices, computers, and storage disks that interface between computers and real-world systems. They include data servers used to store database data.

Software: Programs that control and manage the database, including DBMS software, operating systems, network software, and applications for accessing data. These programs integrate with hardware to manage all data transactions.

Data: The raw information stored in the database, which can be texts, numbers, or binary data. It is the primary content that databases manage and process.

Procedures: Rules and guidelines for using the database, including creating and running databases and managing data. Procedures act as manuals for users.

Database Access Language (DAL): Programming languages like SQL used to read, update, and delete data from a database. DAL enables users to create databases, tables, and manipulate data efficiently.

Data Models in DBMS

Data models in DBMS help in understanding the design at conceptual, physical, and logical levels.

They describe how data is stored, accessed, and updated using symbols and text for clarity.

These models provide conceptual tools to represent the description of data, aiding developers in creating a physical database.

Types of Data Models

Hierarchical Model

- **Description:** Organizes data in a tree-like structure with records having a parent-child relationship.
- **History:** Developed by IBM in the 1950s.
- **Example:** Vehicle database classifying vehicles into two-wheelers and four-wheelers.
- **Drawback:** Supports only one-to-many relationships, limiting its modern application.

Network Model

- **Description:** Generalization of the hierarchical model allowing many-to-many relationships.
- **Structure:** Represents data as a graph with object types as nodes and relationships as edges.
- **Example:** College database linking departments to a director.
- **Advantages:** Efficient data access with multiple paths to a node.
- **Drawbacks:** Complex insertion and deletion processes.

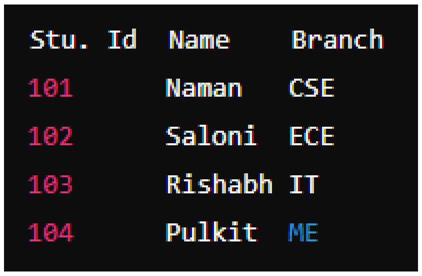
Entity-Relationship (ER) Model

- **Description:** Uses ER diagrams to describe the database structure pictorially.
- **Components:**
- **Entity:** Anything with an independent existence (e.g., Car, Employee).
- **Entity Set:** Collection of similar entities (e.g., Set of students).
- **Attributes:** Properties defining entities (e.g., Employee Name).
- **Relationships:** Associations between entities (e.g., Employee working in a Company).
- **Example:** ER diagram showing the relationship between Employee and Company, with attributes for each entity.

Relational Model

Description: Represents the database as a collection of relations (tables) in rows and columns.

Example:



```
Stu. Id  Name     Branch
101      Naman    CSE
102      Saloni   ECE
103      Rishabh  IT
104      Pulkit   ME
```

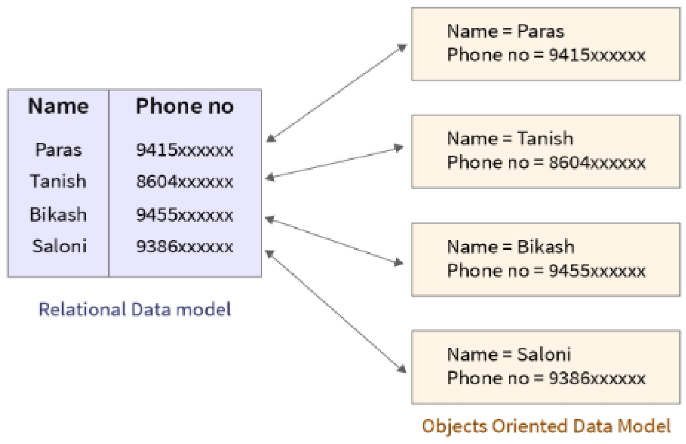
Attributes: Stu. Id, Name, and Branch.

Advantages: Simplifies data organization and access.

Object-Oriented Data Model

- **Description:** Combines object-oriented programming with relational data models.
- **Structure:** Data and their relationships are represented as objects.
- **Example:** Employee and Department objects linked by Department_Id.
- **Advantages:** Easily stores multimedia data (audio, video, images).

Object-Relational Data Model


Name
Phone no
Paras 9415xxxxxx
Tanish 8604xxxxxx
Bikash 9455xxxxxx
Saloni 9386xxxxxx
Relational Data model
Name = Paras Phone no = 9415xxxxxx
Name = Tanish Phone no = 8604xxxxxx
Name = Bikash Phone no = 9455xxxxxx
Name = Saloni Phone no = 9386xxxxxx
Objects Oriented Data Model

Description: Integrates object-oriented and relational models.

Structure: Supports objects, classes, and tabular structures.

Advantages: Combines features of both models.

Drawbacks: Complex and difficult to handle.

Relational Model

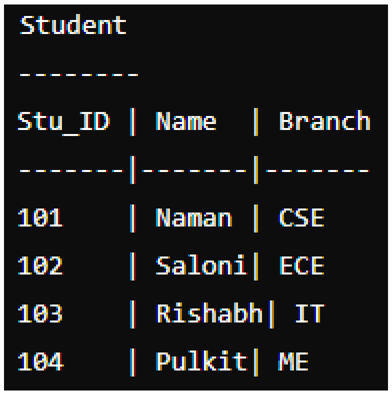
The **relational model** in DBMS is an abstract model used to organize and manage the data stored in a database. It stores data in **two-dimensional inter-related tables,** also known as **relations** in which each row represents an entity, and each column represents the properties of the entity.

The key concepts of the relational model include:

Relation

Definition: A two-dimensional table used to store a collection of data elements.

Example:



```
Student
--------
Stu_ID | Name   | Branch
-------|-------|-------
101    | Naman | CSE
102    | Saloni| ECE
103    | Rishabh| IT
104    | Pulkit| ME
```

Tuple

Definition: A row in the relation, representing a single data item or entity.

Example: The tuple (101, Naman, CSE) in the Student relation represents one student.

Attribute/Field

Definition: A column in the relation, representing a property that describes the relation.

Example: In the Student relation, Stu_ID, Name, and Branch are attributes.

Attribute Domain

- **Definition:** A set of predefined atomic values that an attribute can take.
- **Example:** For the Branch attribute in the Student relation, the domain might be {CSE, ECE, IT, ME}.

Degree

- **Definition:** The total number of attributes present in a relation.
- **Example:** The Student relation has a degree of 3 (attributes: Stu_ID, Name, Branch).

Cardinality

- **Definition:** The total number of tuples (rows) present in a relation.
- **Example:** The Student relation has a cardinality of 4 (four rows).

Relational Schema

- **Definition:** The logical blueprint of a relation, describing the design and structure, including table name, attributes, and their types.
- **Example:** STUDENT (Stu_ID INT, Name VARCHAR, Branch VARCHAR)

Relational Instance

- **Definition:** The collection of records present in the relation at a given time.
- **Example:** The current data in the Student table as shown above.

Relation Key

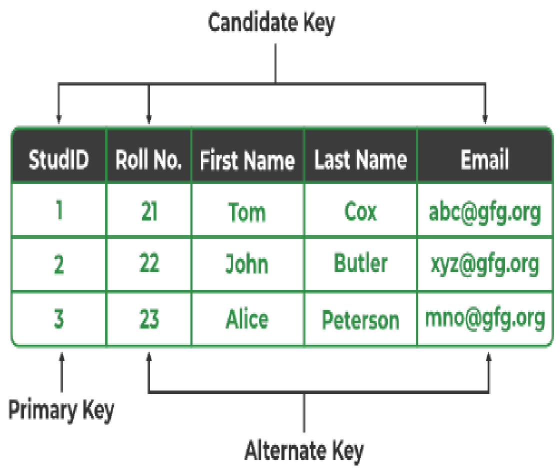
- **Definition:** An attribute or a group of attributes that can be used to uniquely identify an entity in a table or to determine the relationship between two tables.
- Relation keys are of 6 different types:

1. Candidate Key
2. Super Key
3. Composite Key
4. Primary Key
5. Alternate Key
6. Foreign Key

Keys in Relational Model

Keys are widely used to identify the **tuples(rows) uniquely** in the table. We also use keys to **set up relations** amongst various columns and tables of a relational database.

Candidate Key



StudID	Roll No.	First Name	Last Name	Email
1	21	Tom	Cox	abc@gfg.org
2	22	John	Butler	xyz@gfg.org
3	23	Alice	Peterson	mno@gfg.org

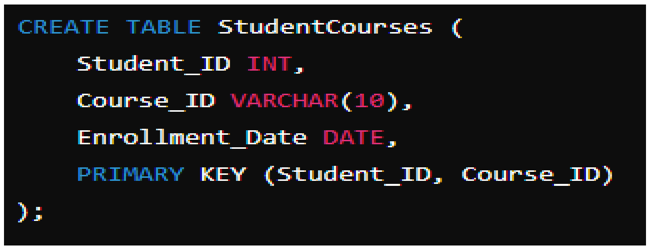
The minimal set of attributes that **can uniquely identify a tuple** is known as a **candidate key.**

Candidate key is a **super key** with **no repeated data** and can contain **NULL values.**

Every **table** must have **at least a single or more candidate keys** but can have only **one primary key**.

Primary Key

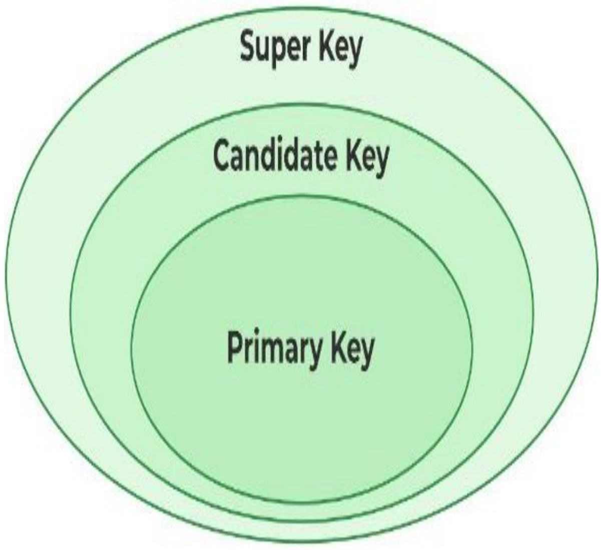
- There can be **more than one candidate key** in relation out of which one can be chosen as the **primary key.**
- Primary key is a **candidate key** with **no repeated data** and cannot contain **NULL values.**
- A **primary key** can be composed of **multiple columns**, and this is known as a **composite primary key.** It is used to uniquely identify records when a single column is insufficient for uniqueness. This helps maintain data integrity and ensures that each record can be uniquely identified by the combination of values from multiple columns.



```
CREATE TABLE StudentCourses (
    Student_ID INT,
    Course_ID VARCHAR(10),
    Enrollment_Date DATE,
    PRIMARY KEY (Student_ID, Course_ID)
);
```

A **StudentCourses** table with **primary keys (Student_ID, Course_ID)**

Super Key



The **set of attributes** that can **uniquely identify a tuple** is known as **Super Key.**

Adding zero or more attributes to the **candidate key** generates the **super key.**

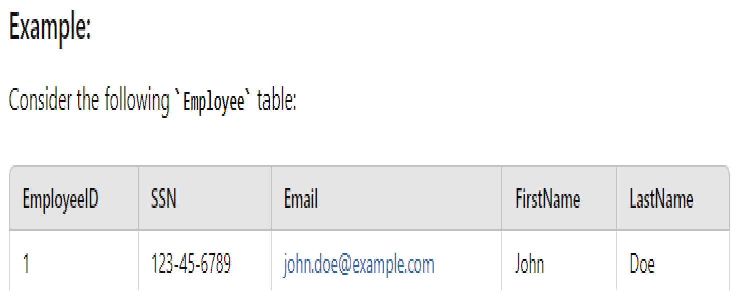
A **candidate key is a super key** but vice versa is not true.

Super Key must **contain Unique Values** but can contain **NULL Values.**

Example: STUDENTS(STUD_NO, STUD_MAIL)

Alternate Key

An **alternate key** in a relational database is **any candidate key** that is **not chosen** to be the **primary key**. Since a table can have multiple candidate keys, the ones that are not selected as the primary key are referred to as alternate keys.


Example:

Consider the following `Employee` table:

EmployeeID	SSN	Email	FirstName	LastName
1	123-45-6789	john.doe@example.com	John	Doe

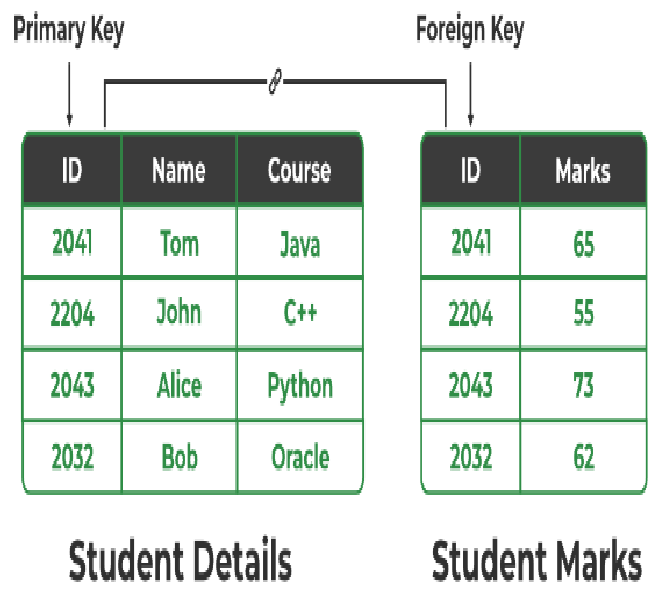
Candidate Keys: EmployeeID, SSN, and Email

Primary Key: EmployeeID

Alternate Keys: The remaining candidate keys, SSN & Email.

Foreign Key

- It is a key that acts as a **primary key** in one table, and it acts as **secondary key** in another table.
- It **combines two or more relations** (tables) at a time.
- They act as a **cross-reference** between the tables.
- **Primary Key: Student Details (ID) = Foreign Key: Student Marks(ID)**



ID	Name	Course
2041	Tom	Java
2204	John	C++
2043	Alice	Python
2032	Bob	Oracle

ID	Marks
2041	65
2204	55
2043	73
2032	62

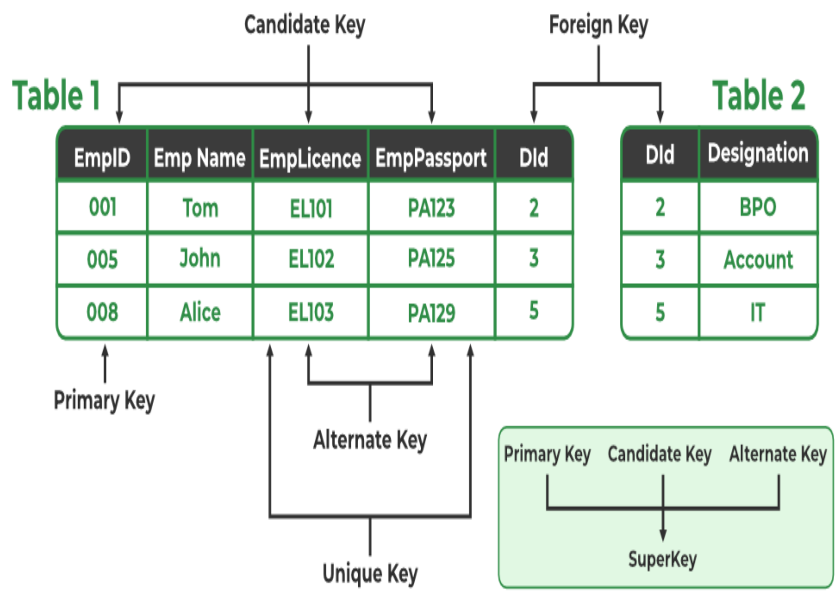
Foreign key can **contain null values** because a **null value** represents a **missing or unknown value**, which means that the specific row does not need to relate to a row in the referenced table.

Foreign key can **contain duplicate values** because the same foreign key value can be used in multiple rows to establish a relationship to the same row in the referenced table.

Composite Key

A **composite key is a primary key** that consists of **two or more columns** used together to uniquely identify a record in a table. This is useful when **a single column is not sufficient to uniquely identify rows.**

Summary



EmpID	Emp Name	EmpLicence	EmpPassport	DId
001	Tom	EL101	PA123	2
005	John	EL102	PA125	3
008	Alice	EL103	PA129	5

DId	Designation
2	BPO
3	Account
5	IT

Primary Key (EmpID): Uniquely identifies each row in **Table 1**.

Unique Key (EmpPassport): Uniquely identifies rows in **Table 1** but can differ from the primary key.

Alternate Key (EmpLicence): A candidate key not chosen as the primary key.

Candidate Key (EmpID, EmpLicence): A set of fields that can uniquely identify row in **Table 1;** includes the primary key.

Foreign Key (DId): Ensures referential integrity by linking **DId** in **Table 2 to DId in Table 1.**

Super Key: A set of one or more columns that can uniquely identify rows in a table. All keys in the image are super keys.

Relational Calculus

Relational Calculus is a **declarative query language,** which means it **tells** the system **what data to retrieve,** not how to retrieve it. It is **based on predicate logic** and has **two types:**

1. Tuple Relational Calculus (TRC)

- **TRC** in DBMS **uses a tuple variable (t)** that goes to each row of the table and checks if the **predicate is true or false** for the given row. Depending on the given predicate condition, it **returns the row or part of the row.**
- **Syntax:** {T | condition(T)}
- **Example:** {T | T ∈ Employee AND T.age > 25}
  ➔ Finds all tuples T in Employee where age > 25.

2. Domain Relational Calculus (DRC)

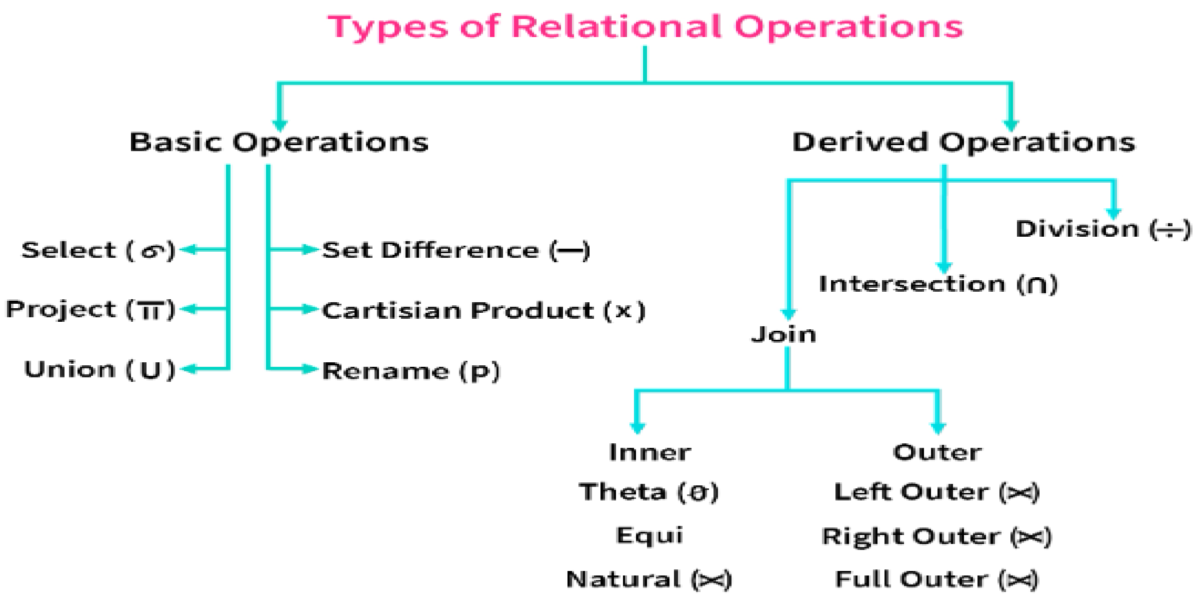
- **DRC** uses **domain Variables to get the column values** required from the database based on the predicate expression or condition.
- **Notation:** {<d1, d2, ..., dn> | condition (d1, d2, ..., dn)}
- **Example:** {<n, a> | ∃e (e ∈ Employee AND e.name = n AND e.age = a AND a > 25)}
  ➔ Finds names (n) and ages (a) of **employees(name, age)** older than 25.

Note: ***In TRC, the entire tuple is considered and returned if it satisfies the given predicate whereas in DRC, only the specified attribute values that satisfy the predicate are returned.***

Relational Algebra

Relational Algebra is a **procedural query language,** which means it tells the system **how to perform operations** to retrieve the desired data.

Relational Algebra came in **1970** and was given by **Edgar F. Codd (Father of DBMS).** It is also known as **Procedural Query Language (PQL)** as in **PQL,** a programmer/user must mention two things, **"What to Do" and "How to Do".**



Basic Operations

1. Select (σ)

- **Select operation** is used to **retrieve tuples(rows)** from the table where the given condition is satisfied. It is a **unary operator** means it requires only one operand.
- **Notation:** σ p(R)
  - ➔ Where **σ** is used to represent **SELECTION**
  - ➔ **R** is used to represent **RELATION**
  - ➔ **p** is the **logic formula**
- **Example:** σ AGE=20 (STUDENT)
  - ➔ **selects the row(s) from STUDENT Relation where "AGE" is 20**

2. Project (π)

- **Project operation** is used to **retrieve certain attributes(columns)** from the table. It is **also known as vertical partitioning** as it separates the table vertically. It is also **a unary operator.**
- **Notation:** ∏ a(r)
  - ➔ Where ∏ is used to represent **PROJECTION**
  - ➔ **r** is used to represent **RELATION**
  - ➔ **a** is the **attribute list**
- **Example:** ∏ NAME(STUDENT)
  - ➔ **Returns only the unique names from each tuple in the 'STUDENT' relation**

3. Union (∪)

- **Union operation** is used to **select all tuples from both relations** but with the exception that for the union of two relations/tables **both relations must have the same set of Attributes.** It is a **binary operator** as it requires two operands.
- If relations don't have the **same set of attributes,** then the **union** of such relations will result in **NULL.**
- **Notation:** R ∪ S
  - ➔ Where **R** is the **first relation**
  - ➔ **S** is the **second relation**
- **Example:** ∏ NAME(STUDENT) ∪ ∏ NAME(EMPLOYEE)
  - ➔ **Returns all the names from 'STUDENT' & 'EMPLOYEE' relations.**

4. Set Difference (—)

- **Set Difference** as its name indicates is the **difference between two relations (R-S).** It **returns** all **the tuples(rows)** which are **in relation R but not in relation S.** It is also a **binary operator.**
- It also **requires same set of attributes** just like **Union (U).**
- **Notation:** R – S
  - ➔ Where **R** is the **first relation**
  - ➔ **S** is the **second relation**

- **Example:** ∏ NAME(STUDENT) - ∏ NAME(EMPLOYEE)
  - ➔ **Returns the names that are available in 'STUDENT' relation but missing in 'EMPLOYEE' relation**

5. Cartesian Product (×)

- **Cartesian Product** is used to **combine all rows from two relations.** It is also a **binary operator.**
- **Notation:** R X S
  - ➔ Where **R** is the **first relation**
  - ➔ **S** is the **second relation**
- **Example:** STUDENT X EMPLOYEE
  - ➔ **Return a single table by combining all the tuples from 'STUDENT' & 'EMPLOYEE'**

6. Rename (ρ)

- **Rename** is used to **rename the output relation.** It is also a **binary operator.**
- It is like **alias** in **MySQL** which is used give a **temporary name** to the output table or attributes.
- **Notation:** ρ (R, S)
  - ➔ Where **R** is the **new relation's name**
  - ➔ **S** is the **old relation's name**
- **Example:** ρ (STUDENT_NAME, ∏ NAME(STUDENT))
  - ➔ **It fetches the names of students from STUDENT relation and renames this relation as STUDENT_NAME.**

Derived Operations

Also known as extended operations, these operations can be derived from basic operations and hence named Derived Operations. These include three operations: **Join Operations, Intersection operations, and Division operations.**

Intersection (∩)

- **Intersection operation** is used to **select all the tuples which are present in both relations.** It is a **binary operator** as it requires two operands. Also, it **eliminates duplicates.**
- **Notation:** R ∩ S
  - ➔ Where **R** is the **first relation**
  - ➔ **S** is the **second relation**
- **Example:** ∏ NAME(STUDENT) ∩ ∏ NAME(EMPLOYEE)
  - ➔ **Returns the names which are available in both 'STUDENT' & 'EMPLOYEE' relations.**

Division (÷)

- **Division operation** is used to find tuples in one relation (called the dividend) that match all tuples in another relation (called the divisor). It is a binary operator and is useful for queries involving "all" conditions.
- **Notation: R ÷ S**
  - ➔ Where **R** is the **dividend relation.**
  - ➔ **S** is the **divisor relation**

Join (⋈)

A **JOIN operation** is used to **combine rows from two or more tables** based on a related column between them.

Types of Joins

1. **Inner Join:** An INNER JOIN returns rows that have matching values in both tables.
2. **Left Join (Left Outer Join):** A LEFT JOIN returns all rows from the left table, and the matched rows from the right table. If no match is found, NULL values are returned for columns from the right table.

3. **Right Join (Right Outer Join):** A RIGHT JOIN returns all rows from the right table, and the matched rows from the left table. If no match is found, NULL values are returned for columns from the left table.
4. **Full Join (Full Outer Join):** A FULL JOIN returns all rows when there is a match in one of the tables. If there is no match, the result is NULL from the side that does not have a match.
5. **Cross Join:** A CROSS JOIN returns the Cartesian product of the two tables, i.e., it returns all possible combinations of rows.

Integrity Constraints in DBMS

Integrity constraints are **rules** that ensure the **accuracy and consistency** of data within a relational database. These **constraints enforce certain conditions** on the data in the database to maintain its integrity.

There are **four types of integrity constraints** in DBMS:

1. **Domain Constraint**
2. **Entity Constraint**
3. **Referential Integrity Constraint**
4. **Key Constraint**

1. Domain Constraint

- **Domain integrity constraint** contains a certain set of rules or conditions to **restrict the kind of attributes or values a column can hold** in the database table. The **data type** of a domain can be string, integer, character, DateTime, currency, etc.
- **Example**: If a column is defined to hold integers it cannot hold characters.

2. Entity Constraint

- **Entity Constraint** ensures that each table has **a primary key** and that the primary key is **unique and not null.**
- **Example:** Consider **Employees table having Id, Name, and salary** of employees where **id** is **primary key** and cannot hold null values.

3. Referential Integrity

- **Referential Integrity Constraint** ensures that there must always **exist a valid relationship** between two relational database tables. This valid relationship between the two tables confirms that **a foreign key in one table matches a primary key in another table,** maintaining consistency between related tables.
- **Example:** Consider an **Employee (Name, ID, Salary, Dept_ID)** and a **Department table** where **Dept_ID** acts as a foreign key between the two tables which is **primary key in Dept table.**

4. Key Constraint

- **Key integrity constraints** ensure that keys, such as primary keys and unique keys, **maintain their uniqueness and non-nullability within the database.** These constraints are fundamental for preserving the integrity of data and enabling efficient data retrieval.
- **Primary Key Constraint**
  - Ensures that a column or a set of columns uniquely identifies each row in a table. A primary key cannot have NULL values.
  - **Example:** Every employee must have a unique emp_id.
- **Unique Key Constraint**
  - Ensures that all values in a column or a set of columns are unique across the table. Unlike a primary key, a unique key can have one NULL value.
  - **Example:** Each employee must have a unique email address.

Functional Dependency

Functional Dependency is the **relationship between attributes** of a table related to each other.

A relation consisting of **functional dependencies** always follows a set of rules called **RAT rules.** They are proposed by **William Armstrong in 1974.**

A **functional dependency** is denoted by an **arrow** "**→**". The **functional dependency** of **A on B** is represented by **A → B.**

Example:

*Consider a relation with four attributes **A, B, C and D,***

R (ABCD): A → BCD, B → CD

*For the first functional dependency **A → BCD,** attributes **B, C and D** are **functionally dependent** on attribute **A.***

*Function dependency **B → CD** has two attributes **C and D functionally depending** upon attribute **B.***

*Everything on the **left side** of **functional dependency** is also referred to as **determinant set,** while everything on the **right side** is referred to as **depending set.***

Types Of Functional Dependency

1. Trivial Functional Dependency

- A **functional dependency** is called **trivial** if the attributes on the right side are the **subset** of the attributes on the left side of the functional dependency.
- **X** → **Y** is called a **trivial functional dependency** if **Y** is the **subset** of **X.**
- **Example: {Employee_Id, Name}** → **{Name}** is a **Trivial functional dependency,** since the dependent **Name** is the **subset** of determinant **{Employee_Id, Name}.**

2. Non-Trivial Functional Dependency

- It is the **opposite** of Trivial functional dependency. **In Non-Trivial functional dependency,** dependent is **not a subset** of the determinant.
- **X** → **Y** is called a **Non-trivial functional dependency** if **Y is not a subset of X.** So, a **functional dependency X** → **Y** where **X** is a **set of attributes**

and **Y** is also a **set of the attribute** but **not a subset of X,** then it is called **Non-trivial functional dependency.**

- **Example: {Employee_Id} → {Name}** is a **non-trivial functional dependency** because **Name**(dependent) is **not a subset of Employee_Id**(determinant).

3. Multivalued Functional Dependency

- In **Multivalued functional dependency,** attributes in the dependent set are **not dependent** on each other.
- X → {Y, Z}, if there exists is no functional dependency between Y and Z, then it is called as Multivalued functional dependency.
- **Example: {Employee_Id} → {Name, Age}** is a **Multivalued functional dependency,** since the dependent attributes **Name, Age are not functionally dependent** (i.e. Name → Age or Age → Name doesn't exist!).

4. Transitive Functional Dependency

- Consider two **functional dependencies A → B and B → C** then according to the **transitivity axiom A → C must also exist.** This is called a **transitive functional dependency.**
- In other words, **dependent is indirectly dependent on determinant** in Transitive functional dependency.
- Example: {**Employee_Id → Department} and {Department → Street Number} holds true.** Hence, according to the axiom of transitivity, **{Employee_Id → Street Number}** is a valid functional dependency.

Armstrong's Axioms/Properties of Functional Dependency

William Armstrong in 1974 suggested a few rules related to functional dependency. They are called **RAT rules.**

1. **Reflexivity(R):** If A is a set of attributes and B is a subset of A, then the functional dependency A → B holds true.
   - ➔ **Example:** {Employee_Id, Name} → Name is valid.

2. **Augmentation(A):** If a functional dependency A → B holds true, then appending any number of the attribute to both sides of dependency doesn't affect the dependency. It remains true.
   - ➔ **Example1:** X → Y holds true then, ZX → ZY also holds true.
   - ➔ **Example 2:** if {Employee_Id, Name} → {Name holds true then, {Employee_Id, Name, Age} → {Name, Age}

3. **Transitivity(T):** If two functional dependencies X → Y and Y → Z hold true, then X → Z also holds true by the rule of Transitivity.
   - ➔ **Example:** if {Employee_Id} → {Name} hold true and {Name} → {Department} hold true, then {Employee_Id} → {Department} also hold true.

Advantages of Functional Dependency

1. It is used to **maintain the quality** of data in the database.
2. It expresses the **facts about the database design**.
3. It helps in clearly defining **the meanings and constraints of databases.**
4. It helps to **identify bad designs.**
5. Functional Dependency **removes data redundancy** where the same values should not be repeated at multiple locations in the same database table.
6. The process of **Normalization** starts with identifying the candidate keys in the relation. Without functional dependency, it's impossible to find candidate keys and **normalize the database.**

Decomposition in DBMS

Decomposition in DBMS refers to **the process of breaking down a large, complex table into smaller, simpler tables** without losing information. The main goal of decomposition is to **eliminate redundancy, avoid anomalies, and preserve dependencies,** thus ensuring that the database remains consistent and efficient.

Whenever we decompose a relation, there are **certain properties that must be satisfied** to ensure no information is lost while decomposing the relations. These **properties** are:

1. **Lossless Join Decomposition.**
2. **Dependency Preserving.**

Lossless Join Decomposition

A lossless Join decomposition **ensures two things:**

1. No information is lost while decomposing from the original relation.
2. If we join back the sub decomposed relations, the same relation that was decomposed is obtained.

Dependency Preserving

The second property of lossless decomposition is dependency preservation which says that after decomposing a relation R into R1 and R2, all dependencies of the original relation R must be present either in R1 or R2 or they must be derivable using the combination of functional dependencies present in R1 and R2.

Normalization is the step-by-step process of organizing data to minimize redundancy and dependency by following a series of normal forms, such as 1NF, 2NF, and 3NF. Decomposition, on the other hand, is the practical process of breaking down larger tables into smaller ones to eliminate redundancy and ensure data integrity while preserving functional dependencies and achieving lossless joins. While normalization focuses on meeting theoretical guidelines, decomposition is a specific action within normalization to improve database structure and efficiency.

Normalization

Normalization is the process of organizing the data in a database to reduce redundancy and improve data integrity. The goal is to divide large tables into smaller, more manageable pieces while preserving the relationships among the data. This process involves several normal forms, each with specific rules and requirements.

Key Concepts

Anomalies: Problems in the database that can occur due to redundancy and poorly structured tables. Normalization provides a method to remove them.

Insertion Anomaly: Inability to add data to the database due to absence of other data.

Update Anomaly: Inconsistencies that occur when updating redundant data.

Deletion Anomaly: Unintended loss of data due to deletion of other data.

Normal Forms: Guidelines or rules to follow for structuring database tables to eliminate anomalies and redundancy. The most common normal forms are:

1. *1NF*
2. *2NF*
3. *3NF*
4. *BCNF*

First Normal Form (1NF)

A table is in **1NF** if:

1. All values in the table are atomic (indivisible).
2. Each column contains unique values.
3. Each record is unique.

Example: Consider a student database where each student's course registrations are listed in a single column as a comma-separated list. To convert this to 1NF, each course registration should be listed in a separate row.

Second Normal Form (2NF)

A table is in 2NF if:

1. It is already in1NF.
2. It has no partial dependency; non-key attributes are fully dependent on the primary key.

Example: In a student-course table where each row contains student ID, course ID, and course name, the course name depends only on the course ID, not on the student ID. To achieve 2NF, split this table into two: one table for student-course relationships and another for course details.

Third Normal Form (3NF)

A table is in 3NF if:

1. It is already in 2NF.
2. It has no transitive dependency; non-key attributes are not dependent on other non-key attributes.

Example: In a student table where each row contains student ID, course ID, and instructor name, the instructor's name depends on the course ID. To achieve 3NF, move the instructor information to a separate table linked by the course ID.

Boyce-Codd Normal Form (BCNF)

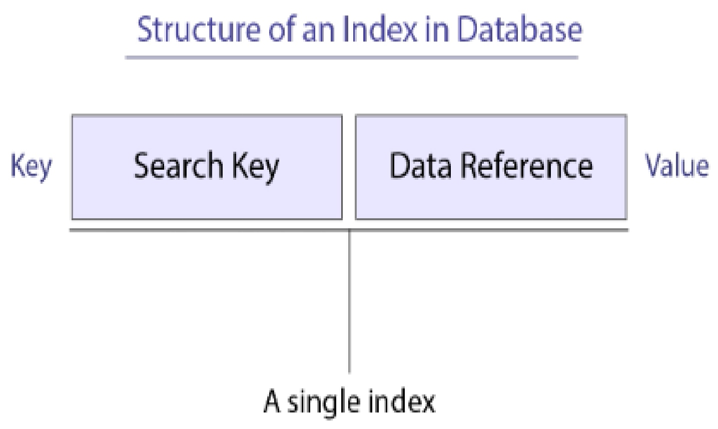
A table is in BCNF if:

1. It is already in 3NF.
2. For every functional dependency ($A \rightarrow B$), A is a super key.

Example: Consider a table where a combination of Professor ID and course ID uniquely identifies the classroom. If the classroom depends only on course ID, decompose the table so that each dependency is based on a super key.

Indexing

Indexing is a technique used in database management systems to **optimize the speed and performance of data retrieval operations.** An index creates an entry for each value that allows the database to quickly locate data without scanning every row in a table.



Indexing is achieved by creating an **Index-table or Index.**

Index usually consists of two columns which are a **key-value pair.** The two columns of the index table (i.e., the key-value pair) contain copies of selected columns of the tabular data of the database.

Here, the **Search Key** contains the copy of the **Primary Key or the Candidate Key** of the database table. Generally, we store the selected Primary or Candidate keys in a sorted manner so that we can reduce the overall query time or search time (from linear to binary).

Data Reference contains a **set of pointers that hold the address of the disk block.** The pointed disk block contains the actual data referred to by the Search Key. Data Reference is also called Block Pointer because it uses block-based addressing.

Types of Indexing Methods

B-Tree Indexing:

- Uses a tree structure with index keys and disk addresses.
- Provides balanced and efficient searching, insertion, and deletion.
- Commonly used for general-purpose indexing.

Bitmap Indexing:

- Uses bit arrays to store the address of data rows.
- More compact and performs faster retrieval for columns with low cardinality.
- Suitable for data warehouse environments.

Indexing Attributes

Ascending and Descending:

- Columns in an index can be sorted in ascending or descending order.
- Default sorting order for character data is by ASCII values, numeric data from smallest to largest, and dates from earliest to latest.

Column and Functional Indexing:

- Index can be created on specific column values or on the results of functions like UPPER () or LOWER ().

Single-Column and Concatenated Indexing:

- Index can be created on a single column or multiple columns.
- Multi-column indexes are useful for queries with WHERE clauses involving multiple columns.

Non-Partitioned and Partitioned Indexing:

- Index can be non-partitioned or partitioned according to the partitioning schema of the table.
- Partitioned indexing helps maintain query performance for partitioned tables.

Types of Indexes

Single Level Indexing:

1. ***Primary Indexing:*** Created using primary keys, ensuring unique, sorted keys with fast searching.
2. ***Secondary Indexing:*** Uses candidate keys, can point to sorted or unsorted data, faster than clustered but slower than primary indexing.
3. ***Cluster Indexing:*** Groups related records, index created on non-key values which may or may not be unique.

Ordered Indexing:

1. ***Dense Indexing:*** Contains records for every search key value, fast searching but space-consuming.
2. ***Sparse Indexing:*** Contains records for some search keys, points to blocks with grouped data, slower but uses less space.

Multi-Level Indexing:

- Uses multiple levels of indexing to handle large data sets, with the outer block small enough to be stored in main memory.
- B+ Tree data structure is often used, with leaf nodes containing actual data pointers.

Advantages of Indexing

1. ***Speed:*** Significantly reduces the amount of time it takes to retrieve data.
2. ***Efficiency:*** Helps in efficiently executing queries and improving overall database performance.
3. ***Reduced I/O Operations:*** Minimizes the number of disk accesses required to fetch data.

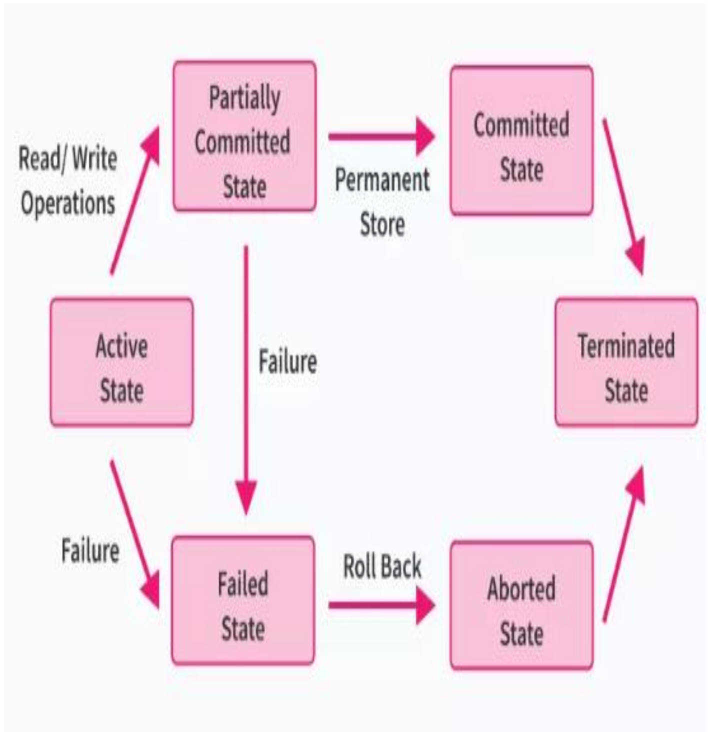
Disadvantages of Indexing

1. *Storage Overhead:* Indexes require additional storage space.
2. *Maintenance:* Indexes need to be updated whenever data is inserted, deleted, or updated, which can impact performance.
3. *Insert/Update Overhead:* Slows down insertions, deletions, and updates due to the extra work needed to maintain the index.

Transactions in DBMS

Transactions in DBMS are **sets of operations** performed to **modify data,** including insertion, updates, or deletions. These transactions **have various states** indicating their progress and required actions. They ensure data consistency even during system failures, demonstrating a key advantage of DBMS.

Transaction States



Active: When the operations of a transaction are running then the transaction is said to be in an **active state.**

Partially Committed: If all the read and write operations are performed without any error then transaction progresses to the **partially committed state.**

Failed State: If any operation during the transaction fails due to some software or hardware issues, then the transaction goes to the **failed state.**

Aborted State: If the transaction fails during its execution, it goes from failed state to **aborted state** and because in the previous states all the changes were only made in the main memory, these uncommitted changes are either deleted or rolled back. The transaction at this point can restart and start afresh from the active state.

Committed State: Committed state occurs, If the transaction completes all sets of operations successfully, all the changes made during the partially committed state are permanently stored and the transaction is stated to be completed, thus the transaction can progress to finally get terminated in the terminated state.

Terminated State: If the transaction gets aborted after roll-back or the transaction comes from the committed state, then the database comes to a consistent state and is ready for further new transactions since the previous transaction **is now terminated.**

ACID Properties of Transaction

Atomicity (A)

- Ensures that all operations within a transaction are completed successfully. If any operation fails, the entire transaction is rolled back, leaving the database in its original state.
- **Example:** If a transaction transfers money from one account to another, both the debit and credit operations must be completed successfully. If one fails, the other must be rolled back.

Consistency (C)

- Ensures that a transaction brings the database from one valid state to another, maintaining all predefined rules and constraints.
- **Example:** If a transaction updates the balance of a bank account, the total balance of all accounts should remain consistent according to predefined rules, such as sum of debits and credits.

Isolation (I)

- Ensures that transactions are executed independently of each other. Intermediate states of a transaction are invisible to other transactions, preventing data inconsistency.
- **Example:** If two transactions are updating the same account balance, the final balance should reflect either one transaction's updates or the others, but not a mix of both.

Durability (D)

- Ensures that once a transaction is committed, the changes are permanent and survive system failures.
- **Example:** Once a bank transfer is completed and the transaction is committed, the changes to the account balances must be saved permanently, even in the event of a system crash.

Transaction Control Commands

1. **BEGIN:** Starts a new transaction.
2. **COMMIT:** Saves all changes made by the transaction permanently to the database.
3. **ROLLBACK:** Undoes all changes made by the transaction, reverting the database to its state before the transaction began.
4. **SAVEPOINT:** Sets a point within a transaction to which a rollback can occur.
5. **RELEASE SAVEPOINT:** Removes a save point created within a transaction.
6. **SET TRANSACTION:** Sets the properties of the transaction, such as isolation level.

Schedules in DBMS

In DBMS, a **schedule** refers to the **order in which transactions are executed.** It determines how transactions, consisting of one or more database operations, are interleaved and executed concurrently.

Schedules are mainly classified as follows:

1. **Serial Schedule**
2. **Concurrent Schedule (Non – Serial Schedule)**

Serial Schedule

A schedule where transactions are executed sequentially, one after another, without interleaving.

Example:

Consider two transactions, T1 and T2, each performing operations on a bank account:

- ***T1: Transfer $100 from Account A to Account B***
- ***T2: Withdraw $50 from Account A***

T1 starts

T1 completes (Transfer $100 from A to B)

T2 starts

T2 completes (Withdraw $50 from A)

In a serial schedule, transactions are executed one after another. Here, T1 completes its transfer before T2 starts its withdrawal, ensuring consistency but potentially sacrificing performance.

Concurrent Schedule

A schedule where transactions overlap in their execution, allowing multiple transactions to run simultaneously.

Example:

Consider the same above T1 & T2 transactions:

Possible interleaving:

T1 starts

T2 starts

T1 completes (Transfer $100 from A to B)

T2 completes (Withdraw $50 from A)

In a concurrent schedule, transactions can overlap, allowing T1 and T2 to execute simultaneously. This improves performance but introduces challenges such as ensuring data consistency and managing conflicts.

Serializability

Serializability in DBMS is a concept that **helps to identify which non-serial schedules are correct** and will maintain the consistency of the database. A serializable schedule always leaves the database in a consistent state. A serial schedule is always a serializable schedule because, in a serial Schedule, a transaction only starts when the other transaction has finished execution. A non-serial schedule of n transactions is said to be a serializable schedule, if it is equivalent to the serial Schedule of those n transactions.

Types of Serializability

Conflict Serializable Schedule:

- A schedule where the order of conflicting operations (like read/write or write/write conflicts) between transactions can be rearranged to produce an equivalent serial schedule.
- **Example:** T1 reads data updated by T2, and T2 writes data read by T1.

View Serializable Schedule:

- A schedule where transactions produce the same set of data values as some serial execution.
- **Example:** T1 reads data, and T2 reads the same data without overlapping writes.

Disclaimer

- The following notes are general summaries and overviews of the topics discussed.
- These notes are not exhaustive and do not cover all aspects of the subject matter.
- The information provided herein is intended for educational purposes only and should not be used as a substitute for professional advice, detailed study, or official course materials.

References

For more detailed information, please refer to the following resources:

Reference 1: Complete DBMS Tutorial

Reference 2: Complete DBMS Tutorial

Reference 3: DBMS Cheat Sheet

Reference 4: DBMS Most Asked Questions

Reference 5: Top DBMS Interview Questions

Prepared By: ***Reddy Venkat Kalyan***