FUNDAMENTALS OF DATABASE

CHAPTER – ONE

INTRODUCTION TO DATABASE

DBMS stands for Database Management System , we can break it like this DBMS = Database + Management System.

Database is a collection of data and Management System is a set of programs to store and retrieve those data. Based on this we can define DBMS like this :- " DBMS is a collection of inter-related data and set of programs to store and access those data in an easy and effective manner"

A database management system (DBMS) is system software for creating and managing databases. The DBMS provides users and programmers with a systematic way to create , retrieve , update and manage data. A DBMS essentially serves as an interface between the database and end users or application programs , ensuring that data is consistently organized and remains easily accessible.

The DBMS manages three important things : the data , the database engine that allows data to be accessed , locked and modified and the database schema , which defines the database's logical structure. These three foundational elements help provide concurrency , security , data integrity and uniform administration procedures.

In the early days , database applications were built directly on top of file systems.

Draw backs of using file systems to store data

Data redundancy and inconsistency :- In file based systems , data redundancy and inconsistency are significant issues. Data redundancy refers to the problem of storing the same data in multiple files or records. This problem occurs when different users or applications create their own file to store the same data. As a result , there are multiple copies of the same data , which leads to storage space wastage and makes data maintenance and updates a daunting task.

Data inconsistency , on the other hand , refers to the problem of having different version of the same data. This problem arises when the same data appears in different files and one of the files get updated while others do not. This results in different versions of the same data , which can create confusion and lead to incorrect data analysis and decision making. Inconsistency can also occur when a file is updated by one user or application while other users are still using the old versions of the same file. For example , if a customer changes their address , the updated address needs to be reflected in all the files that contain the customer's information. If one file is updated , and others are not , this creates inconsistency in the data and can lead to problems such as delivering products to the wrong address.

Difficulty in accessing data

In file-based systems, accessing data can be difficult and time-consuming, especially when the amount of data grows. Since the data is stored in separate files, the system needs to search through all the files to find the relevant data, which can be a slow and inefficient process. Moreover, if the file names are not standardized or organized in a logical manner, finding the correct file can become a daunting task.

Integrity Problems

The issue of integrity problem comes in concurrent process , where multiple users may access the same file simultaneously. If two users try to modify the same data at the same time , it can lead to data inconsistency , where one user's changes overwrite the other user's changes. This can result in lost data , incorrect data or even corrupt files.

Atomicity of updates

Atomic updates refers to the ability to modify multiple related pieces of data as a single , indivisible operation , ensuring that the update is either fully completed or note done at all. The lack of atomic updates in file based systems can lead to integrity problems , as incomplete or inconsistent data may be left behind if an update is interrupted or fails halfway through.

Atomicity of updates refers to the concept of treating multiple related updates as a single, complete operation. Imagine you have a banking application that needs to transfer money between two accounts. Atomic updates ensure that either the entire transfer is completed successfully, or no changes are made at all.

Let's say you transfer money from Account A to Account B. Without atomicity, if the update to Account A succeeds but the update to Account B fails halfway through, you could end up with an inconsistent state. Account A would show the deducted amount, but Account B would still show the old balance. This can cause confusion and problems in managing finances.

To avoid such issues, atomic updates ensure that all related updates happen together or not at all. In a multi-user environment, where multiple people may be accessing and modifying the same data simultaneously, atomic updates are crucial. They prevent conflicts and data corruption that can occur when different users update the same data at the same time.

Relational database systems offers support for atomic updates through the user of transactions , which allows multiple related updates to be grouped together and treated as single , indivisible operation. If a transaction fails , all the changes made during the transaction can be rolled back , ensuring that the data remains consistent.

Concurrent access by multiple users

Concurrent access by multiple users is another problem with file-based approaches. In a file based-system , multiple users can access the same file simultaneously , which can lead to issues such as inconsistency , lost updates and even data corruption. For example , consider a scenario where two users try to update the same file at the same time. User 1 opens the file , makes some changes , and saves the file. However , before user 1 can close the file , user2 also opens the file and makes some changes , then saves the file. User 2's changes overwrite user 1's changes , and user 1's changes are lost. This is known as a lost update problem.

File-based systems do not provide any mechanisms for managing concurrent access to files. Therefore it is up to the individual users to coordinate their access to files to avoid these kinds of issues. This can be difficult and time consuming , especially in a large organization with many users and many files. Additionally , file based systems do not provide any way to enforce access control , so there is no way to prevent unauthorized access to files.

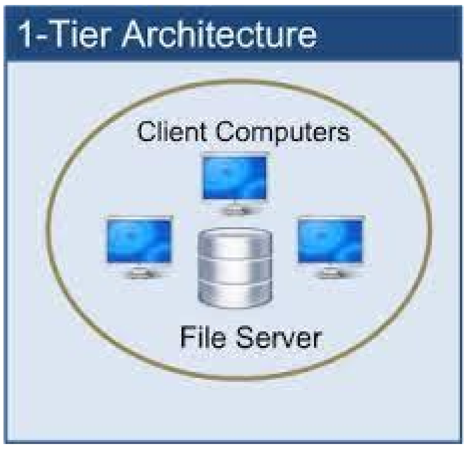
DBMS Architecture

The architecture of DBMS depends on the computer system on which it runs. For example , in a client server DBMS architecture , the database systems at a server machine can run several requests made by client machine. We will understand this communication with the help of diagrams.

There are three types of DBMS architecture :-

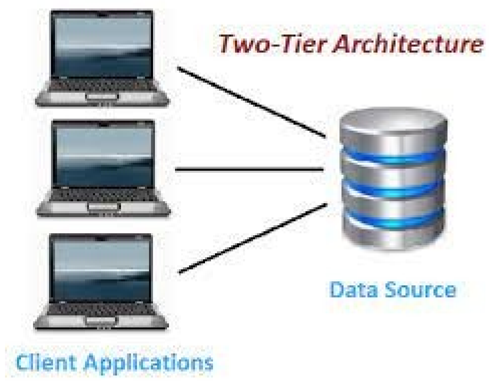
1) SINGLE TIER ARCHITECTURE

In this type of architecture the database is readily on the client machine , any request made by client does not require a network connection to perform the action on the database. For example , let's say we want to fetch the records of employee from the database and the database is available on your computer system , so the request to fetch employee details will be done by your computer and the records will be fetched from the database by your computer as well. This type of system is generally referred as local database system.

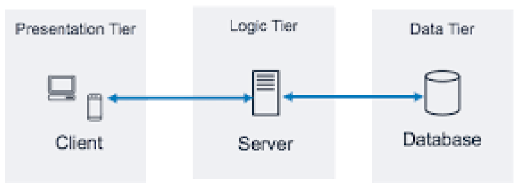


2) **TWO TIER ARCHITECTURE** :-

In two tier architecture , the Database system is present at the server machine and the DBMS application is present at the client machine , these two machines are connected with each other through a reliable network as shown in the above diagram. Whenever client machine makes a request to access the database present at server using a query language like sql , the server perform the request on the database and returns the result back to the client. The application connection interface such as JDBC , ODBC are used for the interaction between server and client.



3) THREE TIER ARCHITECTURE :- In three tier architecture , another layer is present between the client machine and the server machine. In this architecture , the client application doesn't communicate directly with the database systems present at the server machine , rather the client application communicates with server application and the server application internally communicates with the database system present at the server.



INSTANCES AND SCHEMAS

In the context of a Database Management System (DBMS), instances and schemas refer to different aspects of organizing and managing data.

1. **Instance**: An instance refers to the actual data stored in a database at a particular point in time. It represents a snapshot of the data that exists in the database at any given moment. It consists of all the records, tables, and relationships that have been created and populated with data.

Think of the instance as the concrete representation of the data. It includes the specific values stored in the database, such as the names, addresses, and other attributes of individuals in a customer database. The instance can change over time as new data is added, modified, or removed from the database.

2. **Schema**: A schema, on the other hand, defines the structure and organization of the database. It represents the logical blueprint or plan for how the data is stored, organized, and accessed. The schema includes the definitions of tables, columns, relationships, constraints, and other database objects.

In simpler terms, think of the schema as the framework or the rules that govern how the data is stored and organized. It defines the structure of the database without specifying the actual data values. The schema provides a blueprint for creating and maintaining the database and ensures consistency and integrity in the data.

The schema defines the types of data that can be stored in the database, the relationships between different tables, and the rules that govern data validation and integrity. It serves as a guide for developers, administrators, and users to understand how the database is structured and how to interact with it.

To summarize, an instance represents the actual data stored in a database at a specific point in time, while a schema defines the structure and organization of the database without including the specific data values. Instances change over time as data is added or modified, while schemas provide the overall framework for how the data is stored and accessed.

DBMS LANGUAGES

DBMS languages refer to the programming languages that are used to interact with databases managed by database management systems (DBMS). These languages provide users with the ability to create , manipulate and query databases. There are four types of DBMS languages :-

1) Data Definition Language (DDL) :- It is used to define the database schema. The schema describes the structure of the database and the relationships between the tables. DDL commands are used to create , alter and delete database objects such as tables.

CREATE :- To create the database instance
ALTER :- To alter the structure of database
DROP :- To drop database instances
TRUNCATE :- To delete tables in a database instance
RENAME :- To rename database instances
DROP :- To drop objects from database such as tables

2) Data Manipulation Languages (DML) :-

It is used to accessing and manipulate the data in the database. DML commands are used to insert , update and delete the data from the tables. These commands allow the users to interact with the data in the database.

SELECT :- To read records from tables
INSERT :- To insert records in to the tables
UPDATE :- Update the data in table(s)
DELETE :- Delete all the records from the table

3) Data Control Language (DCL) :- It is used to control the access to the database. DCL commands are used to grant and revoke permissions to the users and roles. These commands ensures the security of the data by controlling the access to the database objects.

GRANT :- To grant access to user
REVOKE :- To revoke access from user

4) Transaction Control Language :- It is used to manage the transactions in the database. Transactions are a set of operations that are executed as a single unit of work. TCL commands are used to control the transactions by committing or rolling back the changes made to the database.

COMMIT :- To persist the changes made by DML commands in the database

ROLLBACK :- To roll back the changes made to the database

DATA MODELS

In DBMS , a data model is a conceptual representation of the data structure that will be used to store , organize , and manage the data. It defines how data will be organized , what relationships will exist between different data entities and what constraints will be applied to the data.

A data model provides a high level view of the database system and allows database designers and developers to create a blue print for the database that can be used to guide the implementation process. The most common data models used in DBMS are **hierarchical** , **network model** , **relational** , **object oriented** , **ER-Model.**

1) Hierarchical Model :- The hierarchical data model organizes data in a tree like structure. In this model , each child record has only one parent but a parent record can have multiple child records.

2) Network Model :- The network model is a data model used in DBMS that is a modification of the hierarchical model. In this model , data is represented in a **graph-like structure** with nodes and edges. The nodes represent the data entities , and the edges represent the relationship between those entities.

In the network model , each node have multiple parent and child nodes. This allows for more complex relationships between data entities to be represented than in the hierarchical model. Nodes can also be connected to multiple edges , allowing for more flexibility in representing relationships.

One of the key benefits of the network model is its ability to represent complex data relationships , however this complexity can also make it difficult to design and maintain.

3) Relational Model :- The relational data model Is the most widely used data model in modern database management system. In this model , data is organized in tables consisting of rows and columns.

4) Object oriented Model :- The object oriented data model is designed to store complex data types such as images , sound and video. In this model , data is organized in objects , which consist of **attributes** and **methods**. It provides a high level of flexibility and supports complex relationships between objects. It is mainly used in multimedia applications.

5) ER-Model :- The entity relationship model is conceptual data model that represents **data as a set of entities and their relationships.** It is mainly used to design database systems before they are implemented. The model is composed of three components , **entities** , **attributes** and their **relationships.** It provides a graphical representation of the data model , which makes it easy to understand and communicate.

ENTITY RELATIONSHIP DIAGRAM

An entity – relationship diagram (ERD) is a graphical representation of entities and their relationships to each other. ER diagrams are typically used in software engineering and database design to help visually represent the relationships between entities.

An **entity** is a person , a place , a thing or event that is relevant to the system being modeled. For example , in a university database , entities might include students , courses , instructors , and departments.

A relationship is an association between entities. For example in the university database , there might be a relationship between students and courses , where a student can take many courses and a course can have many students.

ER diagrams typically include the following components :-

Entities :- Represented as rectangles , entities are the objects that exist with in the system being modeled. Each entity has attributes that describe its properties.

Attributes :- Represented as ovals or circles attached to the entity , attributes describe the properties of an entity. For example , a student entity might have attributes such as name , student ID , and date of birth.

Relationship :- Represented as lines between entities , relationships describe how entities are related to each other. **Relationships can be one to one , one to many , or many to many.**

SYMBOLS FOR ER-DIAGRAM

In an Entity-Relationship (ER) diagram, different symbols are used to represent the components of the database model. Here is a detailed description of each symbol:

1. **Rectangle**: A rectangle is used to represent an entity set, which represents a group or collection of similar entities. An entity set can represent a real-world object, such as a customer or an employee, or a concept, such as an order or a product.

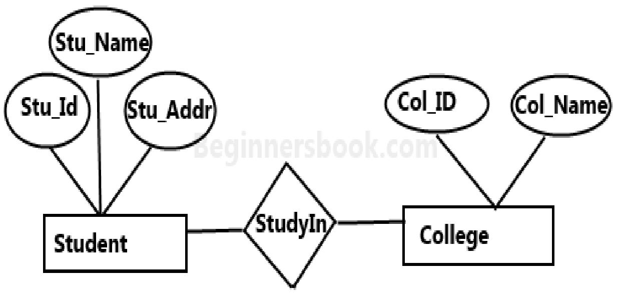
2. **Ellipses**: Ellipses are used to represent attributes, which are characteristics or properties of an entity. Attributes describe the specific details or information associated with an entity. For example, a customer entity may have attributes like name, address, and phone number.

3. **Diamonds**: Diamonds are used to represent relationship sets, which define the associations or connections between entity sets. Relationship sets describe how entities from different entity sets are related or linked to each other. For example, a relationship set can represent the "works for" relationship between employees and departments.

4. **Lines**: Lines are used to represent the connections between different components in an ER diagram. They are used to connect attributes to entity sets and entity sets to relationship sets. For example, a line can connect an attribute like "age" to

an entity set like "student", indicating that "age" is an attribute of the "student" entity.

5. **Double Ellipses:** Double ellipses are used to represent multi-valued attributes. A multi-valued attribute is an attribute that can have multiple values for a single entity. For example, an entity representing a student may have a multi-valued attribute like "hobbies" that can have multiple hobbies associated with it.
6. **Dashed Ellipses**: Dashed ellipses are used to represent derived attributes. A derived attribute is an attribute whose value can be derived or calculated based on other attributes. It is not directly stored in the database but can be derived from other attributes. For example, the age of a person can be derived from their birth date.
7. **Double Rectangles:** Double rectangles are used to represent weak entity sets. A weak entity set is an entity set that cannot be uniquely identified by its own attributes. It depends on the existence of a related entity set called the owner entity set. For example, a "dependent" entity set may be a weak entity set that depends on the "employee" entity set as its owner.



Sample E-R Diagram

Components of a ER diagram

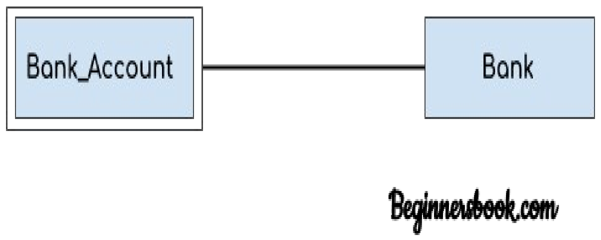
1) In entity – in relationship diagrams a strong entity is an entity that can exist independently , where as a weak entity is an entity that can not exist being associated with a specific strong entity.

A strong entity is also referred to as a regular entity because it is independent and has its own unique identifier. For example , in a database for a university , a student is a strong entity because they have their own unique identifier such as a student ID number.

On the other hand a **weak entity** is dependent on a strong entity and can not exist without it. A weak entity does not have its own unique identifier and instead relies on the identifier of the related strong entity. For example , in the same university database , a course offering may be a weak entity because it can not exist without being associated with a specific department , which is a strong entity.

Weak entities are typically represented in ER diagrams with a double rectangle , while strong entities with a single rectangle. The relationship between strong entity and a weak entity is represented with a solid diamond shape , with the weak entity being connected to the strong entity with a line.

For example , a bank account can not be uniquely identified with out knowing the bank to which the account belongs , so bank account is a weak entity.



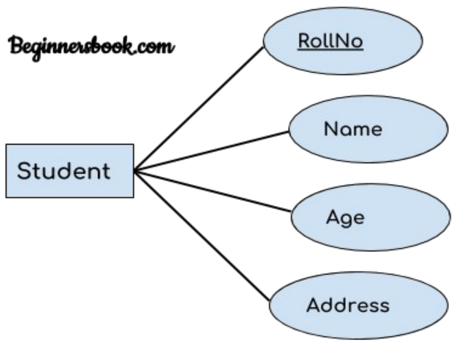
2) Attribute

An attribute describes the property of an entity. An attribute is represented as **Oval** in an ER diagram. There are four types of attributes.

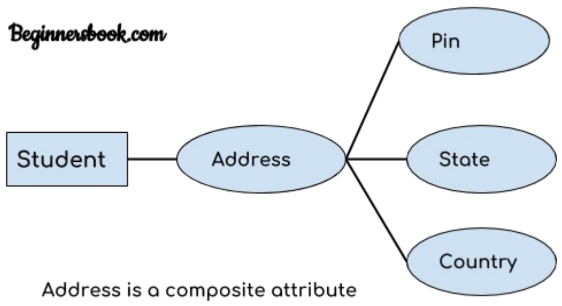
1) **Key attribute** :- A key attribute is an attribute that uniquely identifies an entity instance with in

an entity set. In other words , a key attribute is used to distinguish one entity from another , For example , in a customer entity set , a customer ID might be the key attribute.

A key attribute is represented by oval same as other attributes however the text of the key attribute is underlined.



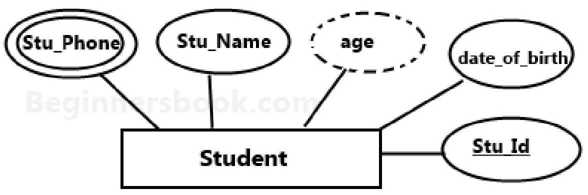
2) **Composite Attribute** :- A composite attribute is an attribute that is made up of multiple smaller attributes. For example , a person's address might be composed of multiple smaller attributes , such as street , city , state and zip code.



3) **Multi-valued attribute** :- A multi valued attribute is an attribute that can have more than one value for a single entity instance. For example , a person might have multiple phone numbers , such as home phone and a work phone. It is represented by double ovals in an ER diagram.

4) **Derived Attribute** :- A derived attribute is an attribute that is derived or calculated from other attributes. For example , a person's age can be derived from their date of birth.

It is represented by **dashed oval** in an ER diagram.



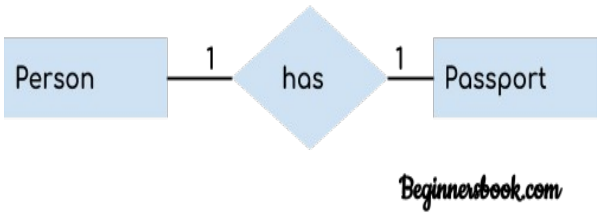
3) RELATIONSHIP

A relationship is represented by a **diamond shape** in ER diagram. It shows the relationship among entities. There are four types of relationships :-

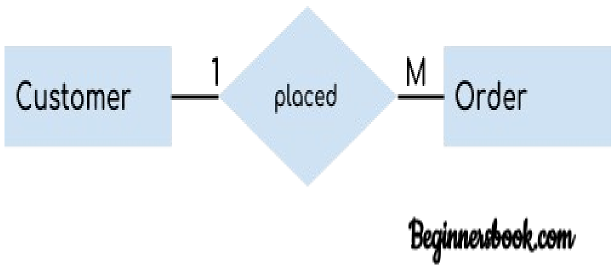
1) One to one
2) One to many
3) Many to one
4) Many to many

1) One to One Relationship (1:1)

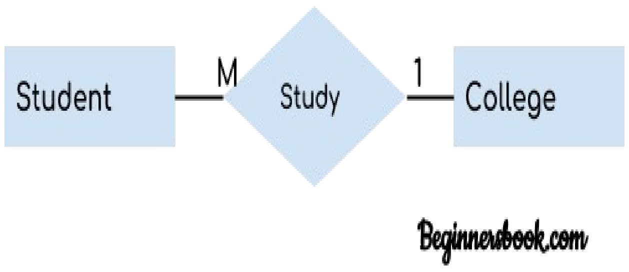
This type of relationship occurs when one instance of an entity is associated with only one instance of another entity. For example , one student is assigned to only one locker , and each locker is assigned to only one student.



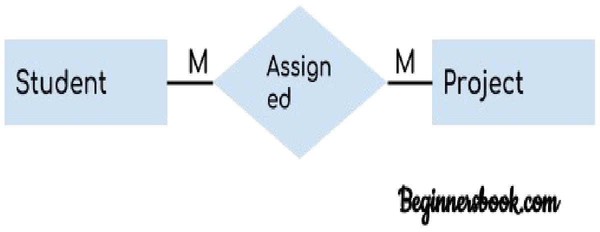
2) One to Many (1:N) :- This type of relationship occurs when one instance of an entity are associated with many instances of another entity. For example , one department may have many employees , but each employee is associated with only one department.



3) Many to one (N:1) :- This type of relationship occurs when many instances of an entity are associated with only one instance of another entity. For example , many employees work in one department , but each department is associated with only one manager.



4) Many to Many (N:N) :- This type of relationship occurs when many instances of an entity are associated with many instances of another entity. For example , many students can enroll in many courses , and each course can have students enrolled in it.



RELATIONAL MODEL CONCEPT

The relational model is a data model used to organize and represent data in a structured and efficient way. It is based on the concept of relations or tables , where **each row represents an entity or record** and **each column represents a specific attribute or characteristic** of that entity.

The relationships between tables are established through common attributes called **keys** , which allow for the creation of complex queries and reports that can retrieve and analyze data from multiple tables at once.

The relational model also includes a set of constraints such as **primary keys and foreign keys** , that ensure the accuracy and consistency of the data stored in the tables.

WHAT DOES IT MEAN BY CONSTRAINT IN DATABASE ?

In a database, a constraint is like a rule or restriction that is applied to make sure the data stored in the database is accurate and consistent. Constraints help to keep the data in good quality over time. They are enforced by the database system, which makes sure that the rules are followed and prevents any mistakes or inconsistencies.

There are different types of constraints that can be used in a database:

1. **Primary Key Constraint:** This makes sure that each row in a table is uniquely identified by a specific column or set of columns. It means that there can't be any duplicate rows with the same identification.
2. **Foreign Key Constraint:** This is used to link two tables together by connecting a column in one table to the primary key column in another table. It makes sure that the values in the linked column are valid and correspond to the values in the primary key column.
3. **Check Constraint**: This defines a rule that **limits the values that can be entered into a column.** It can be used to validate the data and ensure that it falls within a specific range or follows certain rules.
4. **Not Null Constraint**: This simply says that a column must always have a value and can't be left empty or null. It ensures that important information is always provided and not missing.
5. **Unique Constraint:** This guarantees that no two rows in a table have the same values in a specific column or set of columns. It makes sure that each value is unique and there are no duplicates.

These constraints help to maintain the quality and integrity of the data in the database, preventing mistakes and making sure that the data is reliable and accurate.

THE DIFFERENCE BETWEEN PRIMARY KEY AND UNIQUE KEY

A primary key and a unique key are both types of constraints used in a database to ensure the uniqueness of values in a column or set of columns. However, there are some differences between them:

Purpose:

- **Primary Key:** The primary key is a column or set of columns that uniquely identifies each row in a table. It is used to establish the identity of a record and ensure its uniqueness within the table.
- **Unique Key**: The unique key is also used to ensure the uniqueness of values in a column or set of columns. It prevents duplicate values from being entered but does not necessarily serve as the primary identifier for the record.

PRIMARY KEY AND FOREIGN KEY

In a relational database , a primary key and foreign key are two types of constraints that are used to establish and **enforce relationships between tables.**

A primary key is a unique identifier for each row in the table , It is a column or a combination of columns that uniquely identifies each record in the table. The primary key is used to enforce data integrity. It also serves as a reference point for foreign keys in other tables.

A foreign key is a column or a combination of columns in a table that refers to the primary key of another table. It establishes a link or relationship between two tables. The foreign key ensures that the data in the table is consistent and accurate by enforcing **referential integrity**. This means that a foreign key value must exist in the referenced table's primary key column.

For example , consider two tables : Customers and Orders. The customers table has a primary key column called "CustomerID" , the orders table has a foreign key column called "CustomerID" that references the "Customer ID" column in the customers table. This establishes a one to many relationship between the two tables , where each customer can have multiple orders.

Table :- Orders

Order Id	Customer Id	Order Date	Total
1	101	2022-03-15	500.00
2	102	2022-03-16	750.0
3	101	2022-03-17	1500.00
4	103	2022-03-18	200.00

In this example , the orders table has a foreign key column called "Customer ID" which refers to the primary key column "CustomerID" in the customers table. The "CustomerID" column in the orders table is used to establish a link or relationship between the orders and customers tables.

For each order in the Orders table , the "customerID" column specifies the customer who placed the order. In this case , orders 1 , 3 , and 2 were placed by customers 101 , 101 , and 102 respectively.

The foreign key constraint ensures that the values in the "CustomerID" column in the orders table correspond to valid values In the "CustomerID" column in the customers table. This helps to maintain data consistency and accuracy across two tables.

Foreign keys are important component of a relational database that help to establish and maintain relationships between tables. They ensure that data is consistent and accurate , and allow for efficient retrieval and manipulation of data.

PROPERTIES OF RELATIONS

- Name of the relation is distinct from all other relations
- Each relation cell constraints exactly one atomic (single) value
- Each attribute contains a distinct name
- Tuples has no duplicate value

- Order of tuple can have a different sequence

Uniqueness :- Each tuple in a relation is unique , meaning that no two tuples in the relation are exactly the same. This is enforced by the primary key constraint , which ensures that each tuple is uniquely by a specific attribute or combination of attributes.

Atomicity – each attribute in a relation represents a single , indivisible value. This means that each attribute can not be further divided in to smaller parts. For example, a "Name" attribute would represent a single person's name and not a combination of multiple names.

Order :- The tuples in a relation have no inherent order , meaning that they can be accessed in any order without affecting the meaning of the relation.

RELATIONAL ALGEBRA

Relational algebra is a mathematical notation or formal language used to represent and manipulate data in a relational database. It consists of a set of operators that can be applied to relations (tables) to produce new relations. These operators are similar to the operators in algebra , such as addition and multiplication.

The operators in relational algebra can be classified in to two groups :-

1) Unary Operators :- These operators take a single relation as input and produce a new relation as output. Examples of unary operators include selection and projection.

2) Binary Operators :- These operators take two relations as input and produce a new relation as output. Examples of binary operators include join and set operations such as union , intersection and difference.

By combining these operators , complex queries can be constructed to retrieve and manipulate data from a relational database. Relational algebra provides a formal and precise way of expressing these operations and is the basis for many relational database management systems.

The operations in relational algebra include :-

1) Selection :-

In relational algebra, the selection operator is a unary operator that is used to select a subset of tuples from a relation that satisfy a specific condition. The condition is specified using predicate which is a boolean expression that evaluates to True or False.

The selection operator is represented by the sigma symbol (σ) and is used in the following way :- σ<condition>® , Where σ represents the selection operator , <condition> represents the predicate used to select tuples and R represents the input relation.

The predicate P is used as a propositional logic formula which may use connectors like : AND , OR , and NOT. These relational can use as relational operators lime =, ≠, ≥, <, >, ≤

For example , consider the following relation named "Employees"

ID	Name	Age	Salary
1	John Doe	30	50000
2	Jane Smith	25	40000
3	Bob Jones	40	60000
4	Alice Lee	35	55000

To select all employees with a salary greater than 50000 , we can use the selection operator as follows :-

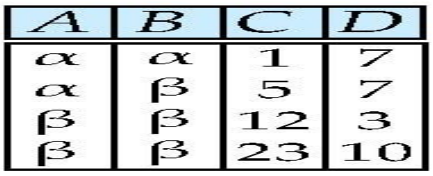
σ (Salary > 50000) (Emplyees)

The result of this operation would be a new relation that contains the following tuples.

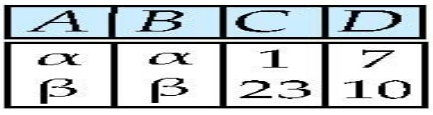
ID	Name	Age	Salary
3	Bob Jones	40	60000
4	Alice Lee	35	55000

In this example , the selection operator has been used to select a subset of tuples from the "Employees" relation that meet the specified condition. The resulting relation contains only those tuples where the "salary" attribute is greater than 50000.

Example :- $\sigma_{(A=B \wedge D > 5)}$ (R)



A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10



A	B	C	D
α	α	1	7
β	β	23	10

2) Projection

In relational algebra , **the projection operator is a unary operator** that is used to select a subset of attributes from a relation and create a new relation that contains only those attributes. The projection operator is represented by the Greek letter pi (π).

The projection operator is used in the following way :-

π<attribute list>®

Where π represent the projection operator , <attribute list> represents the list of attributes to be projected , and R represents the input relation.

For example , consider the following named "Customers"

ID	**Name**	**Age**	**Gender**	**Email**
1	John Smith	30	Male	john.smith@example.com
2	Jane Doe	25	Female	john.smith@example.com
3	Bob Jones	40	Male	john.smith@example.com
4	Alice Lee	35	Female	john.smith@example.co

To project only the "Name" and "Email" attributes from the "Customers" relation , we can use the projection operator as follows :-

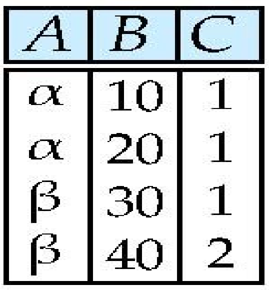
π(Name, Email)(Customers)

The result of this operation would be a new relation that contains the following attribute

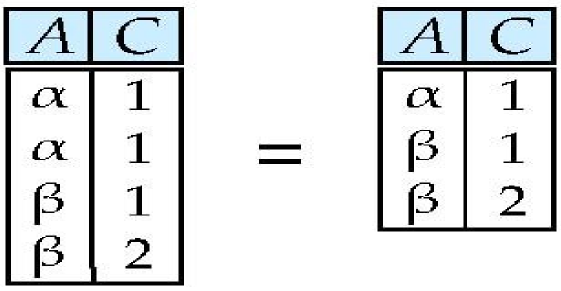
Name	Email
John Smith	john.smith@example.com
Jane Doe	jane.doe@example.com
Bob Jones	bob.jones@example.com
Alice Lee	alice.lee@example.com

In this example , the projection operator has been used to select a subset of attributes from the "customers" relation and create a new relation that contains only those attributes. The resulting relation contains only the "Name" and "Email" attributes and the other attributes have been removed.

Example :- π (A , C) (R)



A	B	C
α	10	1
α	20	1
β	30	1
β	40	2



A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to act on the data.

In the context of relational algebra, **union**, **intersection**, and **set difference** are fundamental operations that deal with two relations (tables in database terminology). These operations are based on set theory, and to be able to apply these operations, two relations must be union-compatible or set-compatible.

Two relations R(A1, A2, ..., An) and S(B1, B2, ..., Bn) are said to be union-compatible or set-compatible if they have:

1. The same number of attributes (n).
2. The corresponding attributes have the same or compatible data types.

This ensures that these operations can be carried out without confusion and that the resulting relation will still have a meaningful structure.

Relation R :

ID	**Name**	**Age**
1	Alice	22
2	Bob	25
3	Carol	30

Relation S :-

ID	**Name**	**Age**
2	Bob	25
4	David	26
5	Eric	35

Here, both tables have the same number of fields (3), and the corresponding fields have the same type (ID: integer, Name: string, Age: integer), so they are union-compatible.

Union (R ∪ S): This operation returns all the tuples that are either in R or S or in both. Duplicate tuples are eliminated.

ID	**Name**	**Age**
1	Alice	22
2	Bob	25
3	Carol	30
4	David	28
5	Eric	35

Intersection (R ∩ S): This operation returns all the tuples that are both in R and S.

ID	Name	Age
2	Bob	25

Set Difference (R - S): This operation returns all the tuples that are in R but not in S.

ID	**Name**	**Age**
1	Alice	22
3	Carol	30

Note: the result of these operations is a new table. For the intersection and set difference operations, if there are no matching records, the resulting table would simply be empty.

In relational algebra, the operations of union, intersection, and set difference are performed on two relations (tables) to combine or compare their tuples (rows). To perform these operations, certain conditions must be met to ensure compatibility and consistency.

1. **Union**: The union operation combines the tuples from two relations into a single relation, eliminating any duplicates. For the union to be performed, the following conditions must be satisfied:

- The two relations must have the same number of fields (columns).
- The corresponding fields (columns) in both relations must have the same data type.

Example: Consider two relations, "Employees" and "Managers." The "Employees" relation has fields such as EmployeeID, Name, and Department, while the "Managers" relation has fields like ManagerID, Name, and Department. To find the combined list of employees and managers without duplicates, we can perform the union operation. However, both relations must have the same number of fields and corresponding fields with the same data type, which in this case they do (Name and Department).

2. **Intersection**: The intersection operation compares the tuples from two relations and returns the common tuples found in both relations. For the intersection to be performed, the following conditions must be satisfied:

- The two relations must have the same number of fields (columns).
- The corresponding fields (columns) in both relations must have the same data type.

Example: Let's say we have two relations, "Students" and "Scholars." The "Students" relation has fields like StudentID, Name, and GPA, while the "Scholars" relation has fields such as ScholarID, Name, and ResearchArea. To find the common students who are also scholars, we can perform the intersection operation. However, both relations must have the same number of fields and corresponding fields with the same data type (Name), allowing us to identify the shared tuples.

3. **Set Difference:** The set difference operation compares the tuples from two relations and returns the tuples that are present in the first relation but not in the second relation. For the set difference to be performed, the following conditions must be satisfied:

- The two relations must have the same number of fields (columns).
- The corresponding fields (columns) in both relations must have the same data type.

Example: Consider two relations, "Employees" and "Managers." The "Employees" relation has fields like EmployeeID, Name, and Department, while the "Managers" relation has fields such as ManagerID, Name, and Department. To find the employees who are not managers, we can perform the set difference operation. However, both relations must have the same number of fields and corresponding fields with the same data type (Name and Department).

By ensuring compatibility, such as having the same number of fields and corresponding fields with the same data type, the union, intersection, and set difference operations can be performed accurately and reliably in relational algebra.

3) Union

In relational algebra , the union operator is a binary operator that combines two relations with the same schema and creates a new relation that contains all tuples from the input relations. The union operator is represented by the symbol u.

The union operator is used in the following way

R U S , where R and S are the input relations. For example , consider the following two relations named “Employees” and “Managers”.

Employees :-

ID	**Name**	**Age**	**Salary**
1	John Doe	30	50000
2	Jane Smith	25	40000
3	Bob Jones	40	60000
4	Alice Lee	35	55000

Managers

ID	**Name**	**Age**	**Department**
5	Sarah Johnson	45	Sales
6	Tom Jackson	50	Marketing

To combine the “Employees” and “Managers” relations in to a single relation , we can use the union operator as follows :-

Employees U Managers

The result of this operation would be a new relation that contains all tuples from both input relations.

ID	**Name**	**Age**	**Salary**	**Department**
1	John Doe	30	50000	
2	Jane Smith	25	40000	
3	Bob Jones	40	60000	
4	Alice Lee	35	55000	
5	Sarah Johnson	45		sales
6	Tom Jackson	50		Marketing

In this example , the union operator has been used to combine the "Employees" and "Managers" relations in to a single relation that contains **all tuples from both relations.** The resulting relation contains all attributes from both input relations and tuples are not duplicated.

In the union operator , both of the input relations must have the same number of attributes and the attributes must be of compatible data types.
For example , if one relation has an attribute of type integer then the other relation must also have an attribute of type integer at the same position , if the data types are not compatible , the union operation may result In an error or undefined behavior.

In addition , the attributes must be in the same order in both relations , for example if the first attribute in one relation is "ID" , then the first attribute in the other relation must also be "ID". The attribute names can be different , but the position and data type must be the same.

It is also worth nothing that the union operator removes duplicates from the resulting relation. If both input relations have the same tuple , it appears only once in the output relation.

Example :-

Deposit Relation

Customer Name	**Account Number**
Johnson	A-101
Smith	A-121
Mayes	A-321
Turner	A-176
Johnson	A-273
Jones	A-472
Lindsay	A-284

Borrow Relation

Customer Name	**Loan No**
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Williams	L-17

Input :- Deposit Relation U Borrow Relation

Customer Name	Account No	Loan No
Johnson	A-101	
Smith	A-121	L-23
Mayes	A-321	
Turner	A-176	
Johnson	A-273	
Jones	A-472	L-17
Lindsay	A-284	
Hayes		L-15
Jackson		L-14
Curry		L-93
Williams		L-17

Input :- Π CUSTOMER_NAME (BORROW) ∪ Π CUSTOMER_NAME (DEPOSITOR)

Customer Name
Johnson

Smith
Mayes
Turner
Jones
Lindsay
Hayes
Jackson
Curry
Williams

4 Set Intersection

In relational algebra , the set intersection operator is a binary operator that combines two relations and returns a new relation containing only the tuples that appear in both input relations , the set intersection operator is represented by the symbol n. The set intersection operator is used in the following way :-

R ∩ S

Where R and S are the input relations.

For example , consider the following two relations named "A" and "B"

A

ID	Name	Age
1	John Doe	30
2	Jane Smith	25
3	Bob Jones	40
4	Alice Lee	35

B

ID	Name	Age
2	Jane Smith	25
4	Alice Lee	35
6	Tom Jackson	50

To find the intersection of the "A" and "B" relation , we can use the set intersection as follows :

A ∩ B

The result of this operation would be a new relation that contains only the tuples that appear in both input relations.

ID	Name	Age
2	Jane Smith	25
4	Alice Lee	35

In this example , the set intersection operator has been used to find the common tuples in both input relations. The resulting relation contains only the tuples that appear in both relations.

5) Set Difference

In relational algebra , the set difference operator is a binary operator that combines two relations and returns a new relation containing only the tuples that appear in the first relation but not in the second relation. The set difference operator is represented by the symbol :- R – S

Where R and S are the input relations. For example , consider the following two relations named "A" and "B".

A :

ID	Name	Age
1	John Doe	30
2	Jane Smith	25
3	Bob Jones	40
4	Alice Lee	35

B.

ID	Name	Age
2	Jane Smith	25
4	Alice Lee	35
6	Tom Jackson	50

To find the set difference of the "A" and "B" relations we can use the set difference operator as follows :-
A – B

The result of this operation would be a new relation that contains only tuples that appear in relation “A” but not in relation “B”.

ID	Name	Age
1	John Doe	30
3	Bob Jones	40

In this example , the set difference operator has been used to find the tuples that appear in relation “A” but not in relation “B”. The resulting relation contains only the tuples that are unique to relation “A”.

The set difference operator is a useful tool for finding the elements that are unique to a particular set of data. It is widely used in SQL and other query languages to construct complex queries that can extract meaningful insights from large datasets.

6) Cartesian Product

In relational algebra , the Cartesian product is a binary operator that combines two relations and returns a new relation **that contains all possible combinations of tuples from both input relations.** The Cartesian product operator is represented by the symbol x.

The Cartesian product operator is used in the following way :-

R X S

Where R and S are the input relations. For example , consider the following two relations named “A” and “B”.

A

ID	Name	Age
1	John Doe	30
2	Jane Smith	25

B

ID	City	Country
1	London	UK
2	New York	USA
3	Paris	France

To find the Cartesian product of the “A” and “B” relations , we can use the Cartesian product operator as follows :-

A x B

The result of this operation would be a new relation that contains all possible combinations of tuples from both input relations

ID	Name	Age	City	Country
1	John Doe	30	London	UK
1	John Doe	30	NewYork	USA
1	John Doe	30	Paris	France
2	Jane Smith	25	London	UK
2	Jane Smith	25	NewYork	USA
2	Jane Smith	25	Paris	France

In this example , the Cartesian product operator has ben used to find all possible combinations of tuples from both input relations. The resulting relation contains all attributes from both input relations and all possible combinations of tuples.

The Cartesian product operator is a powerful tool for finding all possible combinations of data from different relations. It is widely used in SQL and other query languages to construct complex queries that can extract meaningful insights from large datasets.

7) Rename Operation

In relational algebra , the rename operations is a unary operator that is used to change the name of a relation or its attributes. The rename operation is represented by the Greek Letter rho(ρ). The rename operation is used in the following way :-

ρ<new relation or attribute names>®

Where **ρ** represents the rename operator <new relation or attribute name> represents the new names for the relation or its attributes , and R represents the input relation.

For example :- consider the following relation named "Employees"

ID	Name	Age	Salary
1	John Doe	30	50000
2	Jane Smith	25	40000
3	Bob Jones	40	60000
4	Alice Lee	35	55000

To change the name of the "Employees" relation to "Staff" , we can use the rename operator as follows

ρ (Staff)(Employees)

The result of this operation would be a new relation named "Staff" that has the same attributes an tuples as the original relation.

To change the name of the "salary" attribute to "Wage" in the "Employees" relation , we can use the rename operator as follows :-

ρ(ID, Name, Age, Wage)(Employees)

The result of this operation would be a new relation that has the same tuples as the original relation , but the "salary" attribute has been renamed to "Wage".

In this example , the rename operator has been used to change the name of the relation and its attributes. The resulting relation contains the same data as the original relation , but with different names for the relation or its attributes.

8) Join Operation

In relational algebra , the join operation is a binary operator that combines two relations based on a common attribute and creates a new relation that contains all possible combinations of tuples from both input relations that match the condition. The join operator is a fundamental operation in relational databases , and it is widely used in SQL and other query languages to extract meaningful information from large datasets.

The join operator is used in the following way :-

R ⋈<condition> S

Where R and S are the input relations , and <condition> is the condition that the tuples must satisfy in order to be included in the resulting relation. The condition is typically based on a common attribute that appears in both input relations.

For example , consider the following two relations named "Employees" and "Departments"

Employees :-

ID	Name	Age	Salary	Department Id
1	John Doe	30	50000	1
2	Jane Smith	25	40000	2
3	Bob Jones	40	60000	1
4	Alice Lee	35	55000	3

Departments

ID	Name
1	Sales
2	Marketing
3	Operations

To join the "Employees" and "Departments" relations based on the "DepartmentID" attribute , we can use the join operator as follow :-

Employees ⋈ Department ID = ID Departments

The result of this operation would be a new relation that contains all possible combinations of tuples from both input relations where the "Department ID" in the employees relation matches the "ID" in the "Departments " relation.

ID	Name	Age	Salary	Department Id	Name
1	John	30	50000	1	Sales

	Doe				
3	Bob Jones	40	60000	1	Sales
2	Jane Smith	25	40000	2	Marketing
4	Alice Lee	35	55000	3	Operations

In this example , the join operator has been used to combine the “Employees” and “Departments” relations based on the common “Department ID” attribute. The resulting relation contains all possible combinations of tuples where the “DepartmentID” in the “Employees” relation matches the “ID” in the “Departments ” relation.

There are several types of join operations , including inner join , left join , right join , and full outer join. These operations differ in how they handle tuples that do not have a matching value in the other input relation.

Types Of Join Operation

An inner join is a type of operation in a relational database that combines two tables based on a common attribute and returns only the matching records from both tables. It is one of the most commonly used join operations in SQL and helps to retrieve related information from multiple tables.

Imagine you have two sets of data, let's call them Set A and Set B. Each set contains information about different things, but there may be some common information between them. Inner join helps you find the common information between Set A and Set B.

To perform an inner join, you compare a specific attribute (a characteristic or property) in each element of Set A with the corresponding attribute in Set B. For example, you might compare the "ID" attribute in Set A with the "ID" attribute in Set B.

When comparing these attributes, you look for matching values. If the values in the attribute match between an element in Set A and an element in Set B, those two elements are considered a match. Only the matching elements are included in the result.

Any elements that don't have a matching value in the other set are excluded from the result. So, if an element in Set A doesn't have a corresponding match in Set B, it won't be included, and vice versa.

The purpose of an inner join is to find the common records or relationships between the two sets. It allows you to extract only the elements that have matching values in the specified attribute.

To represent an inner join, you use the symbol "⋈" (which represents the join operation) followed by a condition that specifies which attributes to compare. For example, you might write Set A ⋈ ID = ID Set B to perform an inner join based on the "ID" attribute.

EXAMPLE :- The **inner join** is another fundamental operation in relational algebra. This operation, also known as the "natural join", combines rows from two or more tables based on a related column between them. Here’s an example , let’s consider two relations (tables) customers and orders.

Customers Table

Customer ID	Name	Contact
1	Alice	1234
2	Bob	5678
3	Carol	9012

Orders Table :-

Order Id	Product	Customer ID
101	Bread	1
102	Butter	1
103	Milk	2

Now, suppose we want to get a table that contains the name of the customer along with the product they ordered. We can achieve this by applying an inner join operation on the Customers and Orders tables based on the CustomerID.

Result of Inner Join on Customers and Orders with respect to CustomerID:

Customer Id	Name	Contact	Order ID	Product
1	Alice	1234	101	Bread
1	Alice	1234	102	Butter
2	Bob	5678	103	Milk

This table tells us that Alice ordered Bread and Butter, and Bob ordered Milk. Note that Carol does not appear in this table because there is no matching CustomerID for Carol in the Orders table. An inner join only returns rows that have matching values in both tables.

It's important to note that the columns for the join condition (in this case, CustomerID) do not necessarily need to have the same name, but they do need to contain the same kind of data.

A left join is a type of join operation in a relational database that combines two tables based on a common attribute and returns all the records from the left (or first) table and the matching records from the right (or second) table. If there is no match in the right table, NULL values are used for the attributes of the right table in the result set.

Imagine you have two groups of friends, Group A and Group B. You want to find out which friends from Group A also belong to Group B. So, you decide to perform a left join.

In a left join:

- You take all your friends from Group A and write down their names.
- Then, you look at Group B and see if any of your friends from Group A are also there.
- If a friend from Group A is in Group B, you write down their name next to their friend from Group A.
- But if a friend from Group A is not in Group B, you leave that space empty, like a blank space.

The left join ensures that you include all your friends from Group A, even if they don't have any friends in Group B. So, the final list will have the names of all your friends from Group A, and if they have friends in Group B, their names will be written next to them. If they don't have any friends in Group B, the space next to their name will be empty.

That's how a left join works! It helps you see which friends from one group also belong to another group, and it ensures that you don't miss any friends from the first group, even if they don't have any connections in the second group.

EXAMPLE :- The **left join**, also known as the "left outer join", returns all records from the left table (table1), and the matched records from the right table (table2). If there is no match, the result is NULL on the right side.

Here's an example. Let's consider the same two relations (tables) **Customers** and **Orders**:

Customers Table:

Customer ID	Name	Contact
1	Alice	1234
2	Bob	5678
3	Carol	9012

Orders Table :-

Order ID	Product	Customer ID
101	Bread	1
102	Butter	1
103	Milk	2

Now, suppose we want to get a table that contains all the customers along with the products they ordered. In case a customer hasn't ordered anything, we still want them to appear in the result with NULL in the product column. We can achieve this by applying a left join operation on the Customers and Orders tables based on the CustomerID.

Result of Left Join on Customers and Orders with respect to CustomerID:

Customer Id	**Name**	**Contact**	**Order ID**	**Product**
1	Alice	1234	101	Bread
1	Alice	1234	102	Butter
2	Bob	5678	103	Milk
3	Carol	9012	NULL	NULL

This table tells us that Alice ordered Bread and Butter, Bob ordered Milk, and Carol hasn't ordered anything. The NULL values for OrderID and Product in Carol's row indicate that there was no matching record in the Orders table for her.

A right join is a type of join operation in SQL that combines records from two tables based on a specified condition and includes all tuples from the right (second) table, along with matching tuples from the left (first) table. If there is no match in the left table for a tuple in the right table, NULL values are used for the attributes of the left table in the result.

Imagine you have two groups of friends, Group A and Group B. This time, you want to find out which friends from Group B also belong to Group A. So, you decide to perform a right join.

In a right join:

- You take all your friends from Group B and write down their names.
- Then, you look at Group A and see if any of your friends from Group B are also there.
- If a friend from Group B is in Group A, you write down their name next to their friend from Group B.
- But if a friend from Group B is not in Group A, you leave that space empty, like a blank space.

The right join ensures that you include all your friends from Group B, even if they don't have any friends in Group A. So, the final list will have the names of all your friends from Group B, and if they have friends in Group A, their names will be written next to them. If they don't have any friends in Group A, the space next to their name will be empty.

In essence, the right join is similar to the left join, but it focuses on the right (second) group instead of the left (first) group. It helps you see which friends from one group also belong to another group, and it ensures that you don't miss any friends from the second group, even if they don't have any connections in the first group.

That's how a right join works! It allows you to find the connections between friends in different groups, considering the right group as the primary focus.

EXAMPLE :- The **right join**, also known as the "right outer join", is the exact opposite of the left join. It returns all records from the right table (table2), and the matched records from the left table (table1). If there is no match, the result is NULL on the left side.

Let's consider the two relations (tables) **Customers** and **Orders**:

Customers Table:

Customer ID	**Name**	**Contact**
1	Alice	1234
2	Bob	5678
3	Carol	9012

Orders Table :-

Order ID	**Product**	**Customer ID**
101	Bread	1
102	Butter	1
103	Milk	2
104	Cheese	4

Here, we have added an order with CustomerID 4, but we don't have a corresponding customer with CustomerID 4 in the Customers table.

Now, if we want to get a table that contains all the orders along with the customer who ordered them (if known), we can apply a right join operation on the Customers and Orders tables based on the CustomerID.

Result of Right Join on Customers and Orders with respect to CustomerID:

Customer ID	Name	Contact	Order ID	Product
1	Alice	1234	101	Bread
1	Alice	1234	102	Butter
2	Bob	5678	103	Milk
NULL	NULL	NULL	104	Cheese

In this table, Alice is listed as ordering Bread and Butter, Bob as ordering Milk, and the Cheese order is there, but with NULL for CustomerID, Name, and Contact because there's no matching customer for CustomerID 4 in the Customers table.

A full outer join is a type of join operation in SQL that combines records from two tables based on a specified condition and includes all tuples from both tables, whether they have a match or not. It ensures that no data is left out and includes both the matching tuples and the non-matching tuples from both input tables.

To understand a full outer join, let's imagine two tables: Table A and Table B, each containing different sets of data. A full outer join between these tables works as follows:

1. Take all the tuples from Table A and write down their attributes.
2. Look at Table B and check if any tuples have matching values based on the specified condition.
3. If a tuple from Table A matches a tuple from Table B, write down the attributes of both tuples together in the result set.
4. If a tuple from Table A doesn't find a match in Table B, still write down its attributes in the result set, using NULL values for the attributes from Table B.
5. Take all the tuples from Table B that didn't find a match in Table A and write down their attributes in the result set, using NULL values for the attributes from Table A.

In simple terms, a full outer join combines the data from both tables, ensuring that all tuples from both tables are included in the result. It helps you find relationships between data in the two tables and provides a complete view of the data, regardless of matching or non-matching values.

EXAMPLE :- The **full outer join,** also simply known as the "outer join", combines the results of both left and right outer joins. The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

Let's consider the two relations (tables) **Customers** and **Orders**:

Customers Table:

Customer ID	Name	Contact
1	Alice	1234
2	Bob	5678
3	Carol	9012

Orders Table :-

Order ID	Product	Customer ID
101	Bread	1
102	Butter	1
103	Milk	2
104	Cheese	3

In this example, we have a Customer (Carol) who has no Orders, and an Order (Cheese) with no corresponding Customer.

If we perform a full outer join operation on the Customers and Orders tables based on the CustomerID, we will get all the Customers with their corresponding Orders, and all the Orders with their corresponding Customers.

Result of Full Outer Join on Customers and Orders with respect to CustomerID:

Customer ID	**Name**	**Contact**	**Order ID**	**Product**
1	Alice	1234	101	Bread
Bread1	Alice	1234	102	Butter
2	Bob	5678	104	Milk
3	Carol	9012	NULL	NULL
NULL	NULL	NULL	104	Cheese

In this table, Alice is listed as ordering Bread and Butter, Bob as ordering Milk, Carol is listed with no orders, and the Cheese order is listed with no associated Customer.

The NULL values indicate there was no match for that record in the other table. Full outer join ensures that we don't lose any data from either table even if there's no match.

TYPES OF FUNCTIONAL DEPENDENCY

Functional dependency is a concept in database normalization that describes the relationship between attributes (columns) in a table. It defines how the values of one or more attributes determine the values of other attributes. In other words, it specifies the dependencies between attributes based on their values.

Full Functional Dependency: A full functional dependency occurs when an attribute is functionally dependent on the entire combination of values in the primary key, and not on any subset of the key. This means that the attribute depends on all the attributes of the primary key, and removing any attribute from the key would break the dependency. In simpler terms, the value of the attribute is determined by the entire primary key. If any proper subset of the primary key can determine the attribute, it is not a full functional dependency.

Example:- Consider a table with attributes (columns) Student ID, Course ID, and Course Instructor. If Course Instructor is fully functionally dependent on the combination of Student ID and Course ID, it means that for a given Student ID and Course ID, there is a unique Course Instructor associated with it.

Partial Functional Dependency: A partial functional dependency occurs when an attribute is functionally dependent on only a part of the primary key, and removing any attribute from the key would still maintain the dependency. This means that the attribute depends on a subset of the primary key rather than the entire key. Partial functional dependencies can lead to data anomalies and are undesirable in database design.

Example :- Continuing with the previous example, if Course Instructor depends only on the Student ID and not on the Course ID, it would represent a partial functional dependency. In this case, the Course Instructor is determined by the Student ID alone, irrespective of the Course ID.

Transitive Dependency: A transitive dependency occurs when an attribute is functionally dependent on another non-key attribute rather than directly on the primary key. In simpler terms, the value of an attribute depends on another non-key attribute, which, in turn, depends on the primary key. Transitive dependencies can also lead to data anomalies and are undesirable in a well-normalized database.

Example: Suppose we have a table with attributes (columns) Student ID, Course ID, and Course Instructor's Department. If the Course Instructor's Department depends on the Course ID, and the Course ID depends on the Student ID, then there is a transitive dependency between the Student ID and the Course Instructor's Department. The Course Instructor's Department indirectly depends on the Student ID through the Course ID.

Identifying and eliminating partial functional dependencies and transitive dependencies are important steps in achieving higher normal forms, such as the Boyce-Codd Normal Form (BCNF) or the Third Normal Form (3NF), which ensure a well-structured and non-redundant database design.

STRUCTURED QUERY LANGUAGE (SQL)

SQL is like a special language that computers use to talk to databases. Databases are like big organized storage places for information. With SQL, we can create databases, add or remove information, and ask the database questions to get the data we need.

Why do we use SQL? Well, it helps us access and manage data in databases. We can describe the data we want, like asking for specific information about a person or a product. We can also define how the data should be organized and change it when needed.

RDBMS (Relational Database Management System)

is a type of database management system that is based on the relational model. It is the foundation for modern database systems like MS SQL Server, Oracle, and MySQL. RDBMS helps manage and organize large amounts of data efficiently.

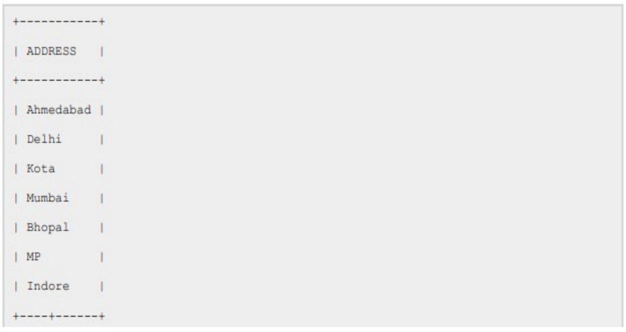
In an RDBMS, data is stored in database objects called **tables.** A table is like a collection of related data entries. It consists of columns (also known as fields) and rows (also known as records). Each row represents a single entry or record in the table, and each column represents a specific piece of information about the records.

For example, let's consider a table called "CUSTOMERS" with columns like ID, NAME, AGE, ADDRESS, and SALARY. Each row in this table represents a customer's information, such as their ID, name, age, address, and salary.

A field is a column in a table that holds specific information about each record. In our example, the ID, NAME, AGE, ADDRESS, and SALARY are the fields of the CUSTOMERS table.

A record, also known as a row, is an individual entry or a set of data in a table. Each record contains values for each field. In our example, each row in the CUSTOMERS table represents a customer's information, such as their ID, name, age, address, and salary.

A column is a vertical entity in a table that contains information for a specific field. It represents all the values for that particular field across all records in the table. For example, the ADDRESS column in the CUSTOMERS table holds the location descriptions, such as Ahmedabad, Delhi, Kota, etc.



```
+-----------+
| ADDRESS   |
+-----------+
| Ahmedabad |
| Delhi     |
| Kota      |
| Mumbai    |
| Bhopal    |
| MP        |
| Indore    |
+----+------+
```

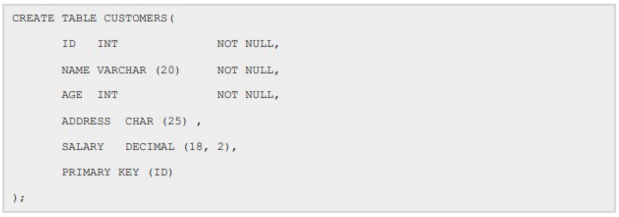
A NULL value in a table refers to a field that appears to be blank or has no value. It is different from a zero value or a field with spaces. A field with a NULL value means that it has not been filled or does not contain any specific value.

Understanding these concepts helps us organize and manipulate data effectively in relational databases using SQL.

SQL CONSTRAINTS

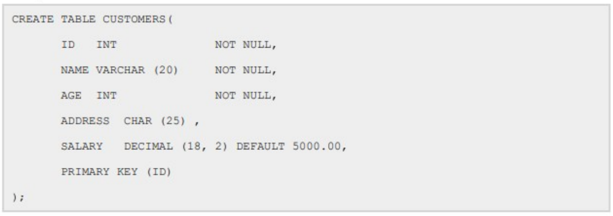
Constraints are the rules enforced on data columns on table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database. Constraints could be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.

- **NOT NULL Constraint:** The NOT NULL constraint ensures that a column cannot have a NULL value. It enforces that the column must always contain a value. This constraint is applied at the column level and prevents inserting or updating a row with a NULL value in that column.



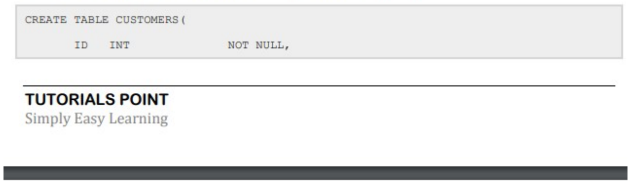
```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

- **DEFAULT Constraint:** The DEFAULT constraint provides a default value for a column when no value is specified during an INSERT statement. If a column with a DEFAULT constraint is not explicitly given a value, it will be assigned the default value defined for that column. This constraint is useful when you want to set a standard value for a column if no specific value is provided.



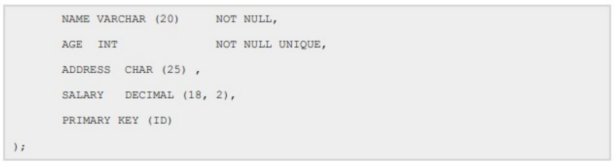
```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2) DEFAULT 5000.00,
       PRIMARY KEY (ID)
);
```

- **UNIQUE Constraint:** The UNIQUE constraint ensures that all values in a column are unique, meaning no duplicate values are allowed. Each value in the column must be distinct. The UNIQUE constraint can be applied to one or more columns, forming a unique combination of values across those columns. This constraint helps maintain data integrity by preventing duplicate entries.
- The UNIQUE Constraint prevents two records from having identical values in a particular column. In the CUSTOMERS table, for example, you might want to prevent two or more people from having identical age.



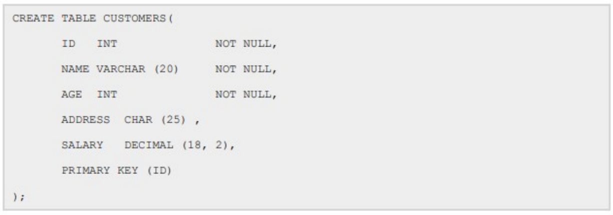
```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
```

TUTORIALS POINT
Simply Easy Learning



```
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL UNIQUE,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

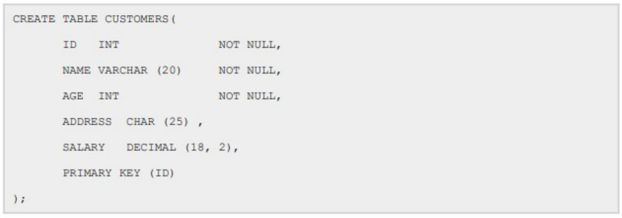
- **PRIMARY KEY Constraint:** The PRIMARY KEY constraint uniquely identifies each row/record in a database table. It ensures that the column or combination of columns specified as the primary key has a unique value for every record in the table. A primary key cannot contain NULL values or duplicate values. Typically, a primary key is a single column, but it can also be a combination of columns.



```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

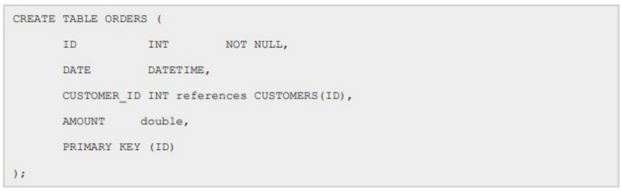
- **FOREIGN KEY CONSTRAINT**: The FOREIGN KEY constraint establishes a relationship between two tables by linking the primary key of one table to a column (known as a foreign key) in another table. It ensures referential integrity by enforcing that the values in the foreign key column must match the values in the referenced primary key column. This constraint maintains the consistency and relationships between related tables.

CUSTOMERS table:



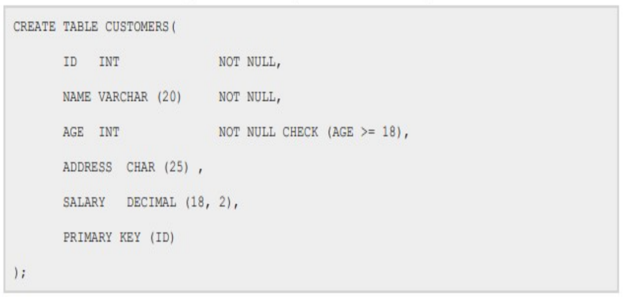
```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

ORDERS table:



```
CREATE TABLE ORDERS (
       ID          INT        NOT NULL,
       DATE        DATETIME,
       CUSTOMER_ID INT references CUSTOMERS(ID),
       AMOUNT     double,
       PRIMARY KEY (ID)
);
```

- **CHECK Constraint**: The CHECK constraint ensures that the values in a column satisfy certain conditions or expressions. It allows you to define specific rules or conditions that the data in the column must meet. If a row violates the condition specified by the CHECK constraint, it will not be inserted or updated. This constraint helps enforce data integrity by restricting the range or values allowed in a column.



```
CREATE TABLE CUSTOMERS(

       ID   INT               NOT NULL,

       NAME VARCHAR (20)      NOT NULL,

       AGE  INT               NOT NULL CHECK (AGE >= 18),

       ADDRESS  CHAR (25) ,

       SALARY   DECIMAL (18, 2),

       PRIMARY KEY (ID)

);
```

DATABASE NORMALIZATION

Normalization is a process in database design that helps organize and structure data efficiently. The goal of normalization is to eliminate data redundancy, improve data integrity, and ensure logical data dependencies. It involves breaking down a large table into smaller, more manageable tables while establishing relationships between them.

Normalization is based on a set of rules called normal forms, which define specific conditions that a database table should meet. The most commonly used normal forms are the First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF). Each normal form has its own criteria that must be satisfied for the table to be considered normalized.

The process of normalization involves analyzing the data and identifying relationships and dependencies between different attributes. Redundant data is removed by splitting tables and establishing relationships using primary keys and foreign keys. This ensures that data is stored efficiently, avoids data inconsistencies, and reduces the chances of anomalies during data manipulation.

By normalizing a database, you can achieve benefits such as reduced data duplication, improved data integrity, simplified data maintenance, and enhanced query performance. Normalization helps create a well-structured and organized database that can adapt to changes and support efficient data retrieval and manipulation operations.

How do we determine whether a table isn't normalized enough – in other words, how do we determine if there's a danger that redundant data could creep into the table? Well, it turns out that there are sets of criteria we can use to assess the level of danger. These sets of criteria have names like "first normal form", "second normal form", "third normal form", and so on.

Think of these normal forms by analogy to safety assessments. We might imagine an engineer doing a very basic safety assessment on a bridge. Let's say the bridge passes the basic assessment, which means it achieves **"Safety Level 1: Safe for Pedestrian Traffic"** That gives us some comfort, but suppose we want to know if cars can safely drive across the bridge? To answer that question, we need the engineer to perform an even stricter assessment of the bridge. Let's imagine that the engineer goes ahead and does this stricter assessment, and again the bridge passes, achieving **"Safety Level 2: Safe for Cars".** If even this doesn't satisfy us, we might ask the engineer to assess the bridge for **"Safety Level 3: Safe for Trucks."** And so on. The normal forms of database theory work the same way. If we discover that a table meets the requirements of first normal form, that's a bare minimum safety guarantee. If we further discover that the table meets the requirements of second normal form, that's an even greater safety guarantee. And so on.

First Normal Form (1NF)

First Normal Form (1NF) is the initial stage of database normalization. It sets the basic requirements for a table to be considered well-structured and eliminates duplicate data within a single row.

1) To understand 1NF, let's consider a scenario where we want to store information about the members of a band called "The Beatles."

In the given example, we have a table with a single column storing the names of the band members. The order of the names in each row may vary, but they represent the same set of members. However, relying on the order of the rows to convey meaning **violates 1NF** because **row order is not meaningful in a relational database.**

In the example, we have a table with a single column for storing the names of the band members. Here's what the initial table might look like :-

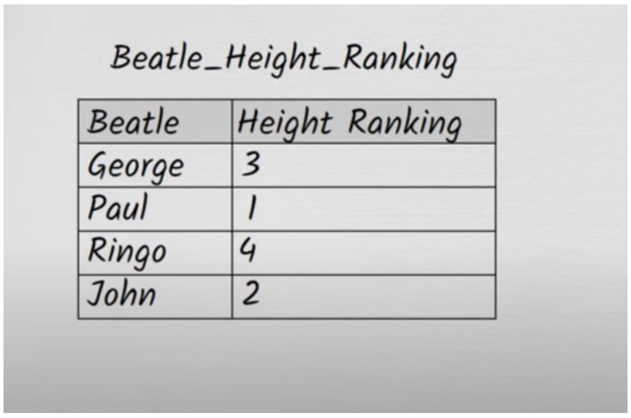
Members
John
Paul
George
Ringo

While the names are correctly listed, the order of the rows may vary. One possible order could be **John, Paul, George, and Ringo**, while another order could be **Paul, John, Ringo, and George.** The variation in row order doesn't affect the fact that these are the members of the band, and both orderings are equivalent.

However, in a relational database, the order of the rows is not significant. The database management system does not guarantee any specific order when retrieving data from the table unless explicitly specified using sorting operations.

By relying on row order to convey meaning, we violate the principles of 1NF. In database design, data should be stored in a way that eliminates dependencies on row order. The focus should be on the data itself, not the position of the data within the table.

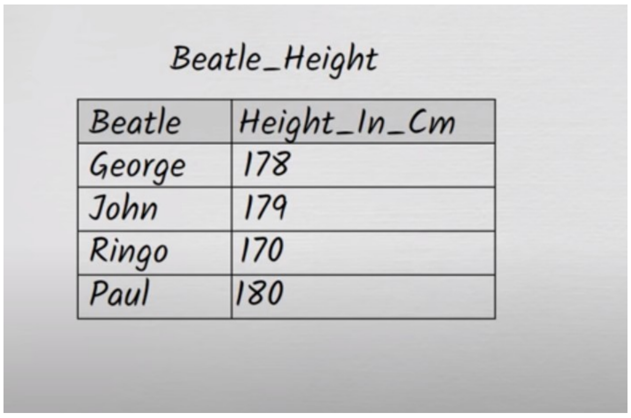
The solution is very simple. Be explicit – if we want to capture height information, we should devote a separate column to it – like this.



Beatle_Height_Ranking

Beatle	Height Ranking
George	3
Paul	1
Ringo	4
John	2

Or even better, like this. So far, we've seen one way in which a design can fail to achieve First Normal Form.

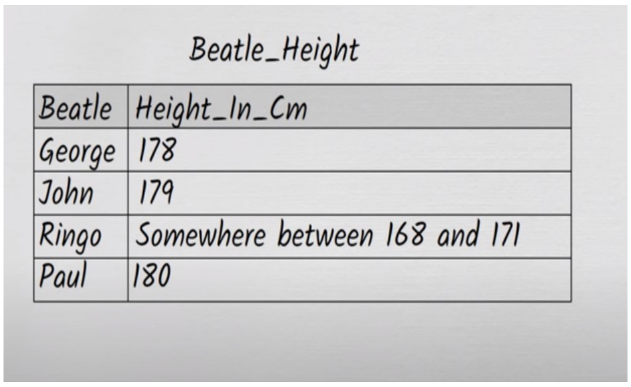


Beatle_Height

Beatle	Height_In_Cm
George	178
John	179
Ringo	170
Paul	180

2) First Normal Form (1NF) emphasizes having defined data types for columns and avoiding the mixing of different data types within the same column.

In a well-designed relational database, each column should have a specific data type that defines the kind of data it can store. For example, a column named "Height" might be intended to store numerical values representing the height of individuals.

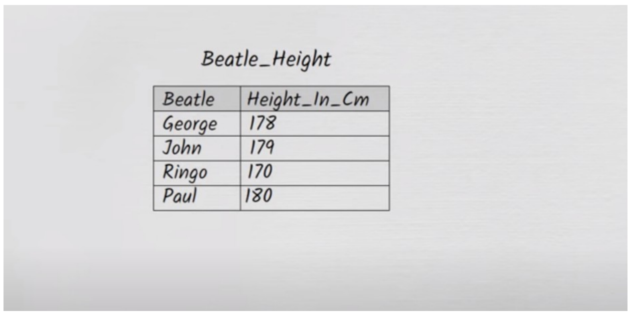


Beatle_Height

Beatle	Height_In_Cm
George	178
John	179
Ringo	Somewhere between 168 and 171
Paul	180

The importance of maintaining consistent data types within a column becomes apparent when we consider the need for data integrity and accurate querying. If we allow a mixture of data types within a column, such as storing both numbers and strings, it can lead to several problems like integrity , query analysis , and performance.

To adhere to 1NF and ensure consistent data types within columns, it's crucial to assign a specific data type to each column that accurately represents the type of data it will store. In our example, the "Height" column should be assigned a numerical data type, such as INTEGER or DECIMAL, to enforce the storage of valid numerical values only.



Beatle_Height

Beatle	Height_In_Cm
George	178
John	179
Ringo	170
Paul	180

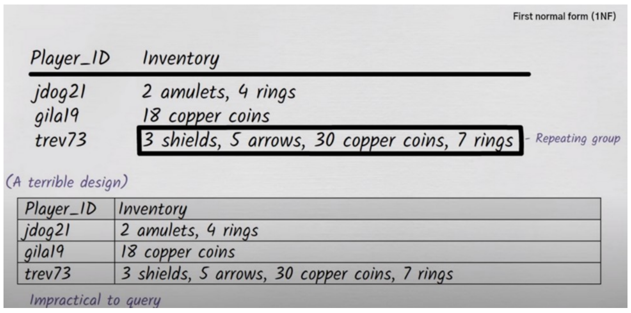
By maintaining defined data types within columns, we promote data integrity, facilitate accurate querying and analysis, and optimize database performance. It enables the database management system to enforce constraints and perform operations efficiently, ensuring the reliability and usability of the data stored in the database.

3) Furthermore, every table in 1NF should have a **primary key.** The primary key uniquely identifies each row in the table. In the case of the Beatles example, we can designate the "Beatle" column as the primary key since each row represents a specific Beatle. This helps maintain data integrity by preventing duplicate entries and ensuring each row is unique.

A primary key is a column or a combination of columns that uniquely identifies each row in a table. It serves as a unique identifier for the records within the table. In 1NF, it is crucial to have a primary key for several reasons:

Uniqueness: The primary key ensures that each row in the table is uniquely identified. It guarantees that no two rows can have the same combination of values in the primary key column(s). In our example of the Beatles, we can designate the "Beatle" column as the primary key. This means that each row represents a specific Beatle, and the primary key enforces the uniqueness of each Beatle's entry.

4) First Normal Form (1NF) that highlights the avoidance of repeating groups or multiple values in a single cell. In the context of a database for an online multiplayer game, let's consider the scenario of storing players' inventories. The goal is to accurately represent the items each player possesses. To adhere to 1NF, we need to avoid storing multiple values within a single cell or lumping all inventory data into a single row.



First normal form (1NF)

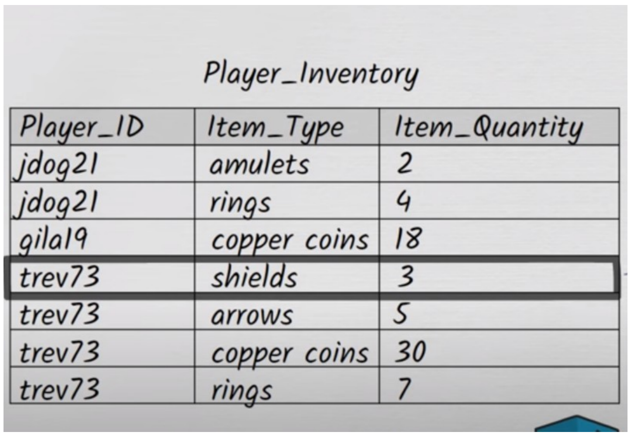
Player_ID	Inventory
jdog21	2 amulets, 4 rings
gila19	18 copper coins
trev73	3 shields, 5 arrows, 30 copper coins, 7 rings

Repeating group
(A terrible design)

Player_ID	Inventory
jdog21	2 amulets, 4 rings
gila19	18 copper coins
trev73	3 shields, 5 arrows, 30 copper coins, 7 rings

Impractical to query

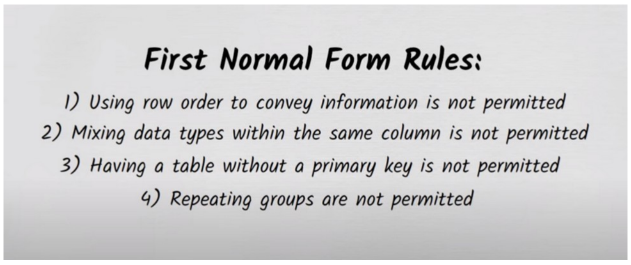
Instead of storing the entire inventory as a single string or using multiple columns for each possible item, the recommended approach is to create separate rows for each item a player possesses. Each row represents a unique combination of player, item type, and quantity.



Player_Inventory

Player_ID	Item_Type	Item_Quantity
jdog21	amulets	2
jdog21	rings	4
gila19	copper coins	18
trev73	shields	3
trev73	arrows	5
trev73	copper coins	30
trev73	rings	7

Generally



First Normal Form Rules:

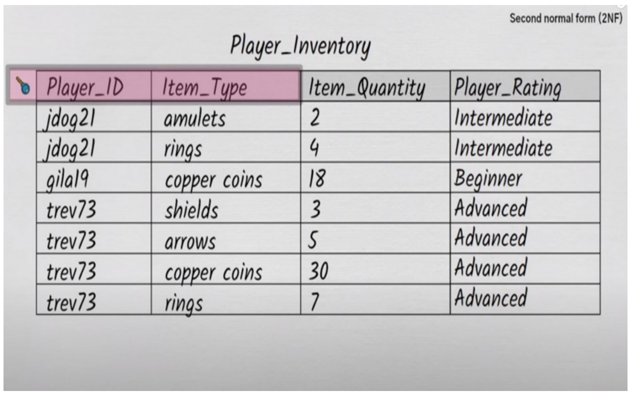
1) Using row order to convey information is not permitted
2) Mixing data types within the same column is not permitted
3) Having a table without a primary key is not permitted
4) Repeating groups are not permitted

SECOND NORMAL FORM (2NF)

In a database, we organize data into different tables. Each table has columns and rows to store information. The second normal form helps us ensure that our tables are well-organized and free from certain problems.

The second normal form focuses on how the data in a table relates to its primary key. The primary key is a special column or a combination of columns that uniquely identifies each row in the table.

To understand the second normal form, let's imagine we have two tables : a Player table and a Player_Inventory table. In the Player table, we store information about each player, like their name and rating (beginner, intermediate, advanced). In the Player_Inventory table, we store information about the items each player has in their inventory, like the item type and quantity.

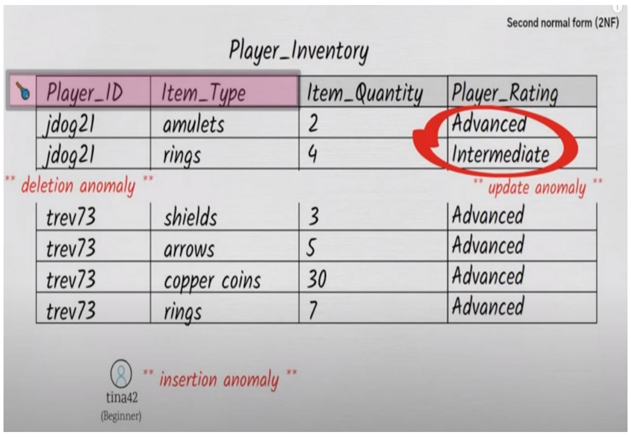


Second normal form (2NF)

Player_Inventory

Player_ID	Item_Type	Item_Quantity	Player_Rating
jdog21	amulets	2	Intermediate
jdog21	rings	4	Intermediate
gila19	copper coins	18	Beginner
trev73	shields	3	Advanced
trev73	arrows	5	Advanced
trev73	copper coins	30	Advanced
trev73	rings	7	Advanced

Here are some problems posed , that violates the second normal form :-



Second normal form (2NF)

Player_Inventory

Player_ID	Item_Type	Item_Quantity	Player_Rating
jdog21	amulets	2	Advanced
jdog21	rings	4	Intermediate
trev73	shields	3	Advanced
trev73	arrows	5	Advanced
trev73	copper coins	30	Advanced
trev73	rings	7	Advanced

" deletion anomaly "
" update anomaly "
" insertion anomaly "
tina42
(Beginner)

Insertion Anomaly: An insertion anomaly refers to a situation where the database design makes it difficult or impossible to insert certain data into a table without also inserting redundant or incomplete information. In the context of the Player Inventory table example, an insertion anomaly occurs when a new player, such as "tina42," has not yet acquired any items for their inventory. Since the table combines both player information and inventory items, it becomes impossible to insert a row for "tina42" without specifying an item, resulting in the player's rating going unrecorded. This limitation makes it challenging to represent certain data correctly during insertion

Deletion Anomaly: A deletion anomaly occurs when removing data from a table unintentionally removes other related data that is still required. In the given example,Now, let's talk about the deletion anomaly. Sometimes, when you want to remove a player from the list, you might accidentally remove important information associated with that player. For example, if a player named "Gila" loses all their items and you delete their entry from the list, you might also unintentionally lose the information about Gila's rating. So, if you later try to find out Gila's rating, you won't be able to because it got deleted along with the player's entry. This unintended loss of related data is called a deletion anomaly.

Update Anomaly: An update anomaly arises when inconsistencies occur as a result of incomplete or incorrect updates to a table. In the Player Inventory table, consider the scenario where a player, "jdog21" improves their rating from "Intermediate" to "Advanced." If an update is executed to modify the rating, but due to an error, only one of the two rows for "jdog21" gets updated, while the other remains unchanged. As a result, the data becomes inconsistent, indicating that "jdog21" has both an "Intermediate" and an "Advanced" rating simultaneously. This inconsistency caused by an incomplete update represents an update anomaly.

These anomalies demonstrate the shortcomings of a table design that does not adhere to Second Normal Form (2NF). By ensuring that each non-key attribute depends on the entire primary key, we can avoid these anomalies and maintain the integrity and consistency of the data in the database.

In database design, we organize data into tables, and each table consists of columns (attributes) and rows (records). The second normal form helps ensure that our table design minimizes data redundancy and maintains a clear relationship between non-key attributes and the primary key.

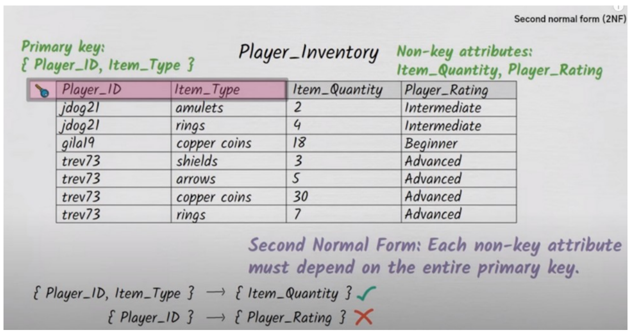
The second normal form focuses on the dependencies between non-key attributes and the primary key of a table. The primary key is a unique identifier for each record in the table, usually composed of one or more columns.

In the case of the Player Inventory table example, we have two non-key attributes: Item_Quantity and Player_Rating. The primary key is a combination of the Player and Item Type columns.

To determine if the table complies with 2NF, we analyze the dependencies of the non-key attributes on the entire primary key. This means that each non-key attribute should be functionally dependent on the entire set of columns that make up the primary key.

Looking at the dependency of **Item_Quantity**, we find that it satisfies the requirements of 2NF. The quantity of items (Item_Quantity) depends on both the Player and Item Type columns. For every unique combination of Player and Item Type, there is a specific value of Item_Quantity associated with it. This dependency reflects a proper relationship between the non-key attribute and the entire primary key.

However, when examining the dependency of Player_Rating, we discover that it only depends on the Player column and not the entire primary key. Player_Rating represents the rating (e.g., beginner, intermediate, advanced) of a player, which remains the same for all the rows associated with that player. This partial dependency violates 2NF because the attribute is not functionally dependent on the entire primary key.



Primary key:
{ Player_ID, Item_Type }

Player_Inventory

Non-key attributes:
Item_Quantity, Player_Rating

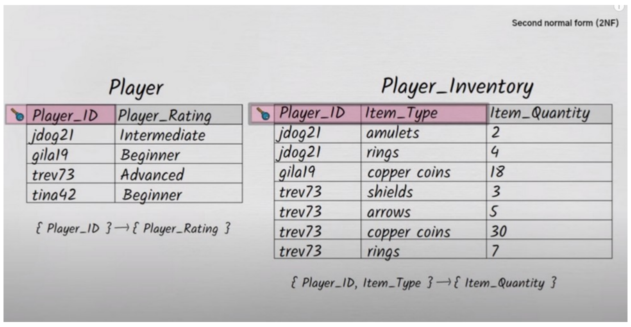
Player_ID	Item_Type	Item_Quantity	Player_Rating
jdog21	amulets	2	Intermediate
jdog21	rings	4	Intermediate
gila19	copper coins	18	Beginner
trev73	shields	3	Advanced
trev73	arrows	5	Advanced
trev73	copper coins	30	Advanced
trev73	rings	7	Advanced

Second Normal Form: Each non-key attribute must depend on the entire primary key.

{ Player_ID, Item_Type } ⟶ { Item_Quantity } ✓
{ Player_ID } ⟶ { Player_Rating } ✗

To address this issue and achieve 2NF, we can separate the Player attribute and its related attributes (such as Player_Rating) into their own table. This new table, called the Player table, would have a primary key that uniquely identifies each player, such as a player ID. The Player_Rating attribute can then be directly dependent on the primary key of the Player table.

By splitting the data into two tables, we eliminate the partial dependency in the Player Inventory table and ensure that each attribute in both tables is functionally dependent on the entire primary key of its respective table. This improves data integrity and avoids anomalies related to dependencies in the table design.



Second normal form (2NF)

Player

Player_ID	Player_Rating
jdog21	Intermediate
gila19	Beginner
trev73	Advanced
tina42	Beginner

{ Player_ID } → { Player_Rating }

Player_Inventory

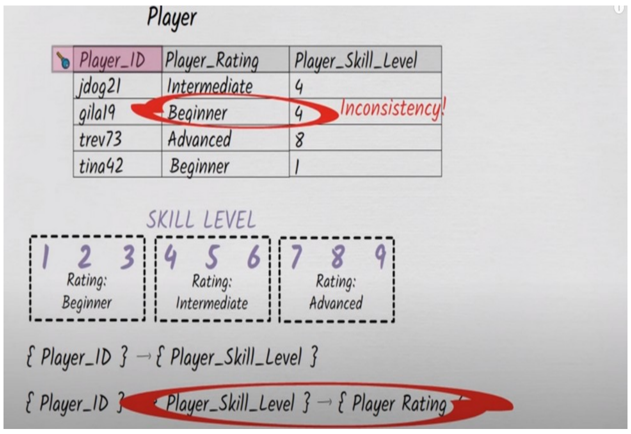
Player_ID	Item_Type	Item_Quantity
jdog21	amulets	2
jdog21	rings	4
gila19	copper coins	18
trev73	shields	3
trev73	arrows	5
trev73	copper coins	30
trev73	rings	7

{ Player_ID, Item_Type } → { Item_Quantity }

So generally the second normal form focuses on the dependencies between non-key attributes and the primary key. It ensures that non-key attributes depend on the entire primary key, thereby reducing redundancy and maintaining a logical relationship between data elements in a table.

THIRD NORMAL FORM

The third normal form (3NF) builds upon the concepts of the first and second normal forms and focuses on eliminating transitive dependencies between non-key attributes within a table. It ensures that the table is free from any undesired dependencies, which helps improve data integrity and maintain a well-structured database design.



Player

Player_ID	Player_Rating	Player_Skill_Level
jdog21	Intermediate	4
gila19	Beginner	4
trev73	Advanced	8
tina42	Beginner	1

Inconsistency!
SKILL LEVEL
1 2 3 Rating: Beginner
4 5 6 Rating: Intermediate
7 8 9 Rating: Advanced

{ Player_ID } → { Player_Skill_Level }

{ Player_ID } → { Player_Skill_Level } → { Player Rating }

In the given example, the Player table has been enhanced with a new column called Player_Skill_Level. This column represents the skill level of each player in a multiplayer game. The skill levels are on a scale from 1 to 9, where 1 represents a beginner level and 9 represents an advanced level. Additionally, each skill level corresponds to a specific player rating: "Beginner" for skill levels 1 to 3, "Intermediate" for skill levels 4 to 6, and "Advanced" for skill levels 7 to 9.

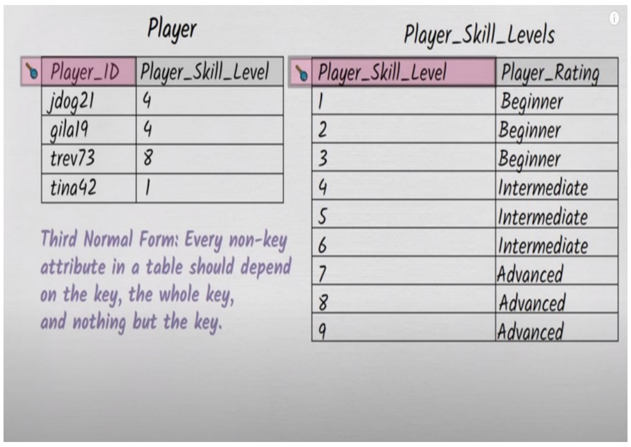
The introduction of the Player_Skill_Level column is intended to capture and represent the skill progression of each player. It provides a measure of their expertise in the game. This additional attribute helps in understanding the players' skill levels beyond just their ratings.

However, a problem arises when both Player_Rating and Player_Skill_Level exist in the Player table. The concern is that the player's rating is indirectly dependent on the player's skill level. If a player's skill level changes, the player's rating should also be updated accordingly to reflect their new skill level accurately.

For example, if a player's skill level increases from 3 to 4, according to the skill level to rating mapping, their rating should be updated from "Beginner" to "Intermediate." However, if an update is performed only on the skill level attribute and the player rating remains unchanged, it results in a data inconsistency. The player's rating would not match their actual skill level, causing confusion and incorrect representation of the player's abilities.

To address this issue and achieve the third normal form (3NF), **it is necessary to remove the transitive dependency** between Player_Rating and Player_Skill_Level. This can be accomplished by removing the Player_Rating attribute from the Player table. Instead, a separate table, called Player_Skill_Levels, can be introduced. This new table will contain the mappings between skill levels and ratings.

By separating the attributes into different tables, each attribute remains directly dependent on the primary key (Player_ID) without any indirect dependencies. The Player table contains attributes related to the player's identification and other player-specific information, while the Player_Skill_Levels table holds the mappings between skill levels and ratings.



Player

Player_ID	Player_Skill_Level
jdog21	4
gila19	4
trev73	8
tina42	1

Player_Skill_Levels

Player_Skill_Level	Player_Rating
1	Beginner
2	Beginner
3	Beginner
4	Intermediate
5	Intermediate
6	Intermediate
7	Advanced
8	Advanced
9	Advanced

Third Normal Form: Every non-key attribute in a table should depend on the key, the whole key, and nothing but the key.

This restructuring ensures that each attribute in a table depends solely on the key, the whole key, and nothing but the key, in adherence to the principles of the third normal form (3NF). It eliminates data inconsistencies and improves the overall integrity and structure of the database design.

The third normal form (3NF) is a database normalization technique that builds upon the concepts of the first and second normal forms. It aims to further eliminate redundancy and improve the integrity of data in a relational database.

In the context of 3NF, a table is considered to be in the third normal form if it satisfies the following conditions:

1. It is in second normal form (2NF).
2. All non-key attributes (columns) are functionally dependent on the table's primary key.
3. There are no transitive dependencies between non-key attributes.

Let's break down these conditions:

1. **Second Normal Form (2NF):** To meet the 3NF requirements, a table must first comply with 2NF. This means that it should be free from partial dependencies, where a non-key attribute depends on only a portion of the primary key. 2NF ensures that each non-key attribute is fully dependent on the entire primary key.
2. **Functional Dependency** on the Primary Key: In a 3NF table, every non-key attribute must depend solely on the primary key. It should be functionally related to the primary key and not dependent on other non-key attributes within the table.
3. **Absence of Transitive Dependencies:** A transitive dependency occurs when a non-key attribute depends on another non-key attribute rather than directly on the primary key. In 3NF, all non-key attributes should be directly dependent on the primary key and not indirectly through other non-key attributes. Transitive dependencies are eliminated by moving the **dependent attribute** to a separate table.

By eliminating transitive dependencies and ensuring that all non-key attributes directly depend on the primary key, the third normal form helps to reduce data redundancy, improve data integrity, and maintain a well-structured database design.

It's important to note that achieving 3NF may not always be necessary for every database design. The level of normalization required depends on the specific requirements and complexities of the data model. In some cases, achieving 3NF may be sufficient, while in others, higher levels of normalization might be necessary to meet the desired goals of data integrity and efficiency.

Fourth Normal Form

Fourth Normal Form (4NF) is a level of database normalization that builds upon the concepts of the first, second, and third normal forms. It addresses the issue of multi-valued dependencies, which occur when a non-key attribute is functionally dependent on part, but not all, of a candidate key.

To understand 4NF, it's essential to grasp the concept of multi-valued dependencies (MVDs). An MVD exists when there is a dependency between two non-key attributes such that for each value of one attribute, there can be multiple values of the other attribute. In other words, the two attributes are independent of each other.

This means that the value of one attribute determines multiple values for another attribute, leading to data redundancies and potential anomalies.

The goal of 4NF is to eliminate or minimize these multi-valued dependencies by splitting them into separate relations. This reduces data redundancy and improves data integrity and efficiency.

To achieve 4NF, the following conditions must be met:

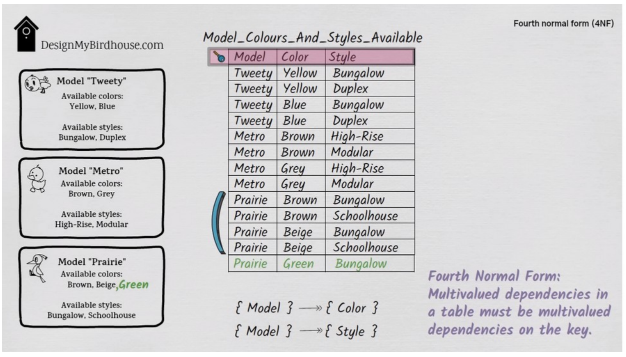
1. The table must be in 3NF.
2. There should be no non-key attribute that is functionally dependent on a subset of the candidate key.
3. Any multi-valued dependencies must be identified and separated into new relations.

By splitting the original table into separate tables, each containing one multi-valued dependency, the resulting relations are free from multi-valued dependencies and redundancies. This ensures that each attribute is functionally dependent on the entire primary key and contributes to a more robust and efficient database design.

4NF is especially relevant in complex database systems with many interrelated tables, as it helps maintain data consistency and avoids data anomalies associated with multi-valued dependencies.

It's important to note that achieving higher levels of normalization, such as 4NF, may not always be necessary or practical for every database design. The level of normalization depends on the specific requirements and complexity of the data model.

Example :- In the example provided, DesignMyBirdhouse.com is a website that allows customers to customize birdhouses by selecting a model, color, and style. Initially, all the combinations of models, colors, and styles were stored in a single table following the Third Normal Form (3NF) of database normalization. However, a problem arose when new colors were introduced for certain birdhouse models. This caused data inconsistencies, as the table incorrectly represented that certain colors were available only for specific styles. The issue stemmed from a type of dependency called multivalued dependency. To resolve this problem and achieve the Fourth Normal Form (4NF), the data was split into multiple tables, each representing the dependencies between models, colors, and styles. This ensured consistency and avoided anomalies in the data.


DesignMyBirdhouse.com
Fourth normal form (4NF)
Model "Tweety" Available colors: Yellow, Blue Available styles: Bungalow, Duplex
Model "Metro" Available colors: Brown, Grey Available styles: High-Rise, Modular
Model "Prairie" Available colors: Brown, Beige, Green Available styles: Bungalow, Schoolhouse

Model_Colours_And_Styles_Available

Model	Color	Style
Tweety	Yellow	Bungalow
Tweety	Yellow	Duplex
Tweety	Blue	Bungalow
Tweety	Blue	Duplex
Metro	Brown	High-Rise
Metro	Brown	Modular
Metro	Grey	High-Rise
Metro	Grey	Modular
Prairie	Brown	Bungalow
Prairie	Brown	Schoolhouse
Prairie	Beige	Bungalow
Prairie	Beige	Schoolhouse
Prairie	Green	Bungalow

$\{ \text{Model} \} \twoheadrightarrow \{ \text{Color} \}$

$\{ \text{Model} \} \twoheadrightarrow \{ \text{Style} \}$

Fourth Normal Form: Multivalued dependencies in a table must be multivalued dependencies on the key.

DesignMyBirdhouse.com is a website where customers can customize birdhouses by choosing a model, color, and style. Initially, all the combinations of models, colors, and styles were stored in a single table, which followed the Third Normal Form (3NF) of database normalization.

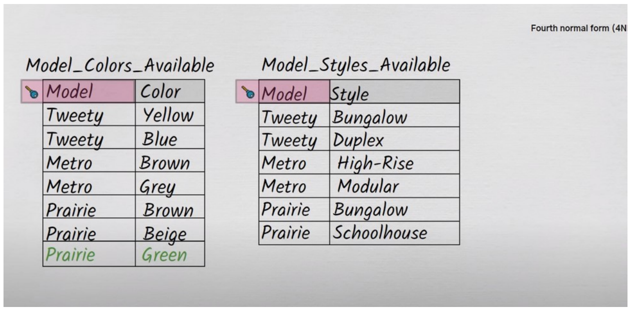
However, there was a problem with this approach. Let's focus on the example of the "Prairie" birdhouse model. Initially, it had two available colors: brown and beige. But later, the website decided to introduce a new color, green, for the "Prairie" model.

To accommodate the new color, additional rows needed to be added to the table. However, if by mistake, only one row for the green bungalow was added, without adding the row for the green schoolhouse, a data inconsistency occurred. This meant that the table was incorrectly indicating that customers could only choose green for the bungalow style but not for the schoolhouse style. This inconsistency didn't make sense because if the "Prairie" model has green as an available color, all its styles should have that option.

The problem arose due to a specific kind of dependency called a multivalued dependency. In this case, the color availability depends on the model, and each model can have a specific set of available colors. Similarly, each model can have a specific set of available styles. These multivalued dependencies are represented by double-headed arrows.

The Fourth Normal Form (4NF) addresses this issue. It states that the only allowable multivalued dependencies in a table should be dependencies on the key. In the given example, the key is not just the model but a combination of the model, color, and style.

To resolve the problem and achieve 4NF, the solution is to split the data into multiple tables. For example, one table would store the available colors for each model, and another table would store the available styles for each model. This way, if the range of colors for the "Prairie" model is expanded to include green, a new row can simply be added to the colors table, ensuring consistency and avoiding anomalies.


Fourth normal form (4N

Model_Colors_Available

Model	Color
Tweety	Yellow
Tweety	Blue
Metro	Brown
Metro	Grey
Prairie	Brown
Prairie	Beige
Prairie	Green

Model_Styles_Available

Model	Style
Tweety	Bungalow
Tweety	Duplex
Metro	High-Rise
Metro	Modular
Prairie	Bungalow
Prairie	Schoolhouse

In simpler terms, the issue was that the table was not properly organized to handle the introduction of new colors for specific birdhouse models. The solution was to split the data into separate tables based on the dependencies, ensuring that each table contained only the relevant and consistent information.

Fifth Normal Form

Fifth Normal Form (5NF), also known as Project-Join Normal Form (PJNF), is the highest level of database normalization. It builds upon the concepts of the previous normal forms (1NF, 2NF, 3NF, and 4NF) and addresses the issue of join dependencies.

The Fifth Normal Form (5NF) states that a database should eliminate all potential data redundancies by decomposing it into smaller, more specific tables based on functional dependencies and joining them together when needed.

In simpler terms, 5NF tells us to organize the database in such a way that we don't store the same information in multiple places. Instead, we break down the data into smaller tables and connect them together when necessary.

By following 5NF, we can ensure that the database is structured efficiently, without any unnecessary repetitions of information. This helps maintain data integrity, reduces storage space, and allows for more flexibility in querying and manipulating the data.

Imagine you have a classroom with students attending different subjects. To keep track of their attendance, you create two tables: one for student details and another for subject attendance.

Table: Student Details

Student ID	Student Name	Age
1	Alice	10
2	Bob	12
3	Claire	11

Table : Subject Attendance

Student ID	Subject	Attendance
1	Math	Present
1	Science	Absent
2	English	Present

3	Math	Present

In this example, the Student Details table stores information about students, including their unique ID, name, and age. The Subject Attendance table records the attendance for each student in various subjects.

Now, let's focus on Alice (Student ID 1). We can see that she was present for Math but absent for Science. Similarly, Bob (Student ID 2) was present for English, and Claire (Student ID 3) was present for Math.

The database seems fine so far, but what if Alice and Bob also attended the same subject, say Math, while Claire attended Science as well? Storing this information separately for each student results in redundancy and inefficiency.

To adhere to the Fifth Normal Form (5NF) and eliminate redundancy, we can modify the structure:

Revised Tables (In 5NF):

Table: Student Details

Student ID	**Student Name**	**Age**
1	Alice	10
2	Bob	12
3	Claire	11

Table : Subjects

Subject Id	**Subject**
1	Math
2	Science
3	English

Table : Attendance

Student ID	**Subject ID**	**Attendance**
1	1	Present
1	2	Absent
2	3	Present
3	1	Present

In the revised structure, we have created a separate Subjects table to store the unique subjects along with their corresponding IDs. The Attendance table now uses these IDs to represent the subjects attended by each student.

By splitting the data into separate tables and using IDs to establish relationships, we avoid repeating subject names for each student and make the database more organized and efficient.

In simpler terms, imagine having a list of students and their details, like names and ages, in one table. Then, you have another table that lists the subjects along with unique subject IDs. Finally, you have a table that connects the students with the subjects they attended, using the student IDs and subject IDs.

By organizing the data this way, we can avoid repeating subject names for each student and make the database more efficient. This helps maintain data integrity, reduce redundancies, and allows for easier management of attendance records, following the principles of the Fifth Normal Form (5NF).

SQL SYNTAX

SQL Statements: SQL is a language used to interact with databases. SQL statements are used to perform various operations on the data stored in a database. Common SQL statements include SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, and SHOW. These keywords indicate the purpose of the statement.

Semicolon: In SQL, statements are typically terminated with a semicolon (;). It acts as a delimiter to separate multiple SQL statements when executing them.

Case Sensitivity: SQL is generally case-insensitive, meaning that keywords and commands can be written in any case (uppercase, lowercase, or a combination). For example, SELECT and select are considered the same in SQL statements. However, database systems like MySQL may treat table names as case-sensitive. Therefore, when working with MySQL, it's important to use the correct case for table names as they exist in the database.

SQL SELECT Statement: The SELECT statement is used to retrieve data from a database table. It specifies the columns that you want to retrieve data from and the table name from which the data should be retrieved. The basic syntax is as follows:

SELECT column1, column2, ..., columnN FROM table_name;

column1, column2, ..., columnN are the names of the columns you want to select.

table_name is the name of the table from which you want to retrieve data. This statement selects specific columns from a table and retrieves all the rows from that table.

SQL DISTINCT Clause: The DISTINCT keyword is used to eliminate duplicate values in the result set. It is often used in conjunction with the SELECT statement. The syntax is as follows :-

SELECT DISTINCT column1, column2, ..., columnN FROM table_name;

`column1, column2, ..., columnN` are the names of the columns you want to select without duplicate values.

`table_name` is the name of the table from which you want to retrieve data.

SQL WHERE Clause: The WHERE clause is used to specify conditions for retrieving data from a table. It allows you to filter the rows based on specific conditions. The syntax is as follows:

SELECT column1, column2, ..., columnN FROM table_name WHERE condition;

column1, column2, ..., columnN are the names of the columns you want to select.

table_name is the name of the table from which you want to retrieve data. condition is the condition that the rows must satisfy to be included in the result set. It is typically a logical expression involving column names and comparison operators, such as equal to (=), not equal to (!=), greater than (>), less than (<), etc.

SQL AND/OR Clause: The AND and OR operators are used to combine multiple conditions in a WHERE clause. The AND operator is used to specify that all conditions must be true for a row to be included in the result set. The OR operator is used to specify that at least one of the conditions must be true for a row to be included. The syntax is as follows:

SELECT column1, column2, ..., columnN FROM table_name WHERE condition1 {AND|OR} condition2;

column1, column2, ..., columnN are the names of the columns you want to select.

table_name is the name of the table from which you want to retrieve data. condition1 and condition2 are logical expressions that specify the conditions for row selection. This statement retrieves rows that satisfy either both conditions (when using AND) or at least one of the conditions (when using OR).

SQL IN Clause: The IN operator is used to specify multiple values for a column in a WHERE clause. It allows you to match a column's value against a list of specified values. The syntax is as follows:

SELECT column1, column2, ..., columnN FROM table_name WHERE column_name IN (val1, val2, ..., valN);

column1, column2, ..., columnN are the names of the columns you want to select. table_name is the name of the table from which you want to retrieve data. column_name is the name of the column you want to compare against the specified values. (val1, val2, ..., valN) is the list of values to match against the column's value.

This statement retrieves rows where the column's value matches any of the specified values.

SQL BETWEEN Clause: The BETWEEN operator is used to select values within a range in a WHERE clause. It allows you to specify a range of values for a column. The syntax is as follows:

SELECT column1, column2, ..., columnN FROM table_name WHERE column_name BETWEEN value1 AND value2;

column1, column2, ..., columnN are the names of the columns you want to select.

table_name is the name of the table from which you want to retrieve data. column_name is the name of the column you want to compare against the range of values. value1 and value2 are the starting and ending values of the range.

This statement retrieves rows where the column's value falls within the specified range.

SQL DESC Statement: The DESC statement is used to retrieve information about the columns and their properties in a table. It provides a description of the table structure, including column names, data types, and constraints. The syntax is:

DESC table_name;

This statement is often used for inspecting the structure of a table without displaying the actual data.

SQL TRUNCATE TABLE Statement: The TRUNCATE TABLE statement is used to remove all rows from a table while keeping the table structure intact. It is faster than the DELETE statement for removing all data from a table. The syntax is:

TRUNCATE TABLE table_name;

This statement is useful when you want to delete all the records in a table but retain the table's structure.

SQL ALTER TABLE Statement: The ALTER TABLE statement is used to modify an existing table structure. It allows you to add, drop, or modify columns in a table. The syntax varies based on the operation:

- **To add a new column:** ALTER TABLE table_name ADD column_name data_type;
- **To drop a column:** ALTER TABLE table_name DROP column_name;
- **To modify a column:** ALTER TABLE table_name MODIFY column_name data_type;

This statement helps in modifying the structure of a table without deleting and recreating it.

SQL ALTER TABLE Statement (Rename): The ALTER TABLE statement with the RENAME option is used to rename an existing table to a new name. The syntax is:

ALTER TABLE table_name RENAME TO new_table_name;

This statement is useful when you want to change the name of a table.

SQL INSERT INTO Statement: The INSERT INTO statement is used to insert new records into a table. It allows you to specify the columns and their corresponding values for the new row. The syntax is

INSERT INTO table_name (column1, column2, ..., columnN) VALUES (value1, value2, ..., valueN);

This statement helps in adding new data rows to a table.

SQL UPDATE Statement: The UPDATE statement is used to modify existing records in a table. It allows you to update the values of one or more columns for selected rows based on specified conditions. The syntax is:

UPDATE table_name SET column1 = value1, column2 = value2, ..., columnN = valueN [WHERE condition];
This statement helps in updating the existing data in a table.

SQL DELETE Statement: The DELETE statement is used to delete one or more rows from a table based on specified conditions. The syntax is:

DELETE FROM table_name WHERE {CONDITION};

This statement helps in removing specific rows or all rows from a table, depending on the specified conditions.

SQL CREATE DATABASE Statement: The CREATE DATABASE statement is used to create a new database in a database management system (DBMS). The syntax is:

CREATE DATABASE database_name;

This statement is used to create a new database with the specified name.

SQL DROP DATABASE Statement: The DROP DATABASE statement is used to delete an existing database from a DBMS. The syntax is:

DROP DATABASE database_name;

This statement permanently deletes the specified database and all its associated tables and data.

SQL USE Statement: The USE statement is used to select and start using a specific database in a DBMS. The syntax is:

USE DATABASE database_name;

This statement sets the specified database as the current active database for executing SQL statements.

SQL COMMIT Statement: The COMMIT statement is used to permanently save the changes made within a transaction in a DBMS. The syntax is:

COMMIT;

This statement ensures that all changes made since the start of the transaction are permanently saved and made visible to other users.

SQL ROLLBACK Statement: The ROLLBACK statement is used to undo or revert the changes made within a transaction in a DBMS. The syntax is:

ROLLBACK;

This statement discards all changes made since the start of the transaction, bringing the database back to its previous state.

These SQL statements provide the ability to delete data from tables, create and drop databases, switch between databases, and manage transactions. They are essential for data manipulation, database management, and ensuring data consistency and integrity.