**DATABASE MANAGEMENT SYSTEM**

**Notes by MOHAMMAD EMRAN AHMED EMON**

**BSc in SOFTWARE ENGINEERING,**

**SHAHJALAL UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**Starting: 29/10/2024**

**Ending: 31/10/24**

**Question Solving**

1. What is Database?
2. What is Level of Abstraction?
3. What is Data Model?
4. What is Schema and Instance?
5. Difference between DML vs DDL.
6. Explain Concurrent Access.
7. Big data vs large data.
8. E-R model and Relational model.
9. What is primary key and foreign key? Differentiate them.

10.What is Generalization?

A close-up of a document

Description automatically generated

**SOLUTIONS OF Q1**

1. **What are the appropriate primary keys?**

For each table:

* **Employee**: The primary key is EMPLOYEE\_ID because it uniquely identifies each employee.
* **Bonus**: The primary key is (EMPLOYEE\_REF\_ID, BONUS\_DATE) as it uniquely identifies each bonus record for an employee.

[// A **Super Key** in a database is a set of one or more attributes (columns) that can uniquely identify a row in a table]

* **Title**: The primary key is (EMPLOYEE\_REF\_ID, AFFECTED\_FROM) as it uniquely identifies each title assigned to an employee on a specific date

**B) Given your choice of primary keys, identify appropriate foreign keys.**

* In the **Bonus** table, EMPLOYEE\_REF\_ID is a foreign key that references EMPLOYEE\_ID in the **Employee** table.
* In the **Title** table, EMPLOYEE\_REF\_ID is also a foreign key that references EMPLOYEE\_ID in the **Employee** table.

**c) Write the following SQL queries using the Employee\_Info schema.**

**i. Write SQL script to create Employee table.**

CREATE TABLE Employee (

EMPLOYEE\_ID INT PRIMARY KEY,

FIRST\_NAME VARCHAR(50),

LAST\_NAME VARCHAR(50),

SALARY DECIMAL(10, 2),

JOINING\_DATE DATE,

DEPARTMENT VARCHAR(50)

);

**ii. Write an SQL query to fetch all values of DEPARTMENT from the Employee table.**

SELECT DEPARTMENT FROM Employee;

**iii. Add an Employee whose Employee ID, First Name, Last Name, Salary, Joining Date, and Department are 1001, Abdul, Quayum, 5000, 1/1/1990, HRM respectively.**

INSERT INTO Employee (EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, SALARY, JOINING\_DATE, DEPARTMENT)

VALUES (1001, 'Abdul', 'Quayum', 5000, '1990-01-01', 'HRM');

**iv. Delete all Employees from the marketing department.**

DELETE FROM Employee WHERE DEPARTMENT = 'Marketing';

**v. Update the Employee title "Jr. Officer" with "Junior Officer."**

UPDATE Title

SET EMPLOYEE\_TITLE = 'Junior Officer'

WHERE EMPLOYEE\_TITLE = 'Jr. Officer';

**SQL**

SQL stands for Standard Query Language

SQL? Lets you access and manipulate databases

SQL-

1. can execute queries against a database
2. can retrieve data from a database
3. can insert records in a database
4. can update records in a database
5. can delete records from a database
6. can create new databases
7. can create new tables in a database
8. can create stored procedures in a database
9. can create views in a database
10. can set permissions on tables, procedures, and views

**Major Commands- SELECT, UPDATE, DELETE, INSERT, WHERE**

**1.SELECT- Retrieving Data**

The SELECT command is used to retrieve data from one or more tables. It allows you to specify which columns you want to see and apply filtering or sorting

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

SELECT name, age

FROM students

WHERE age > 18;

**2.UPDATE-** **Modifying Existing Data**

The UPDATE command allows you to modify data in existing records. You specify the table, the new value(s), and a condition to determine which records to update

UPDATE table\_name

SET column1 = value1, column2 = value2, ...

WHERE condition;

UPADATE Employee

SET salary = 60000

WHERE employee\_id = 2;

**3.DELETE-** **Removing Data**

The DELETE command removes data from a table. Like UPDATE, it requires a condition to specify which rows to delete.

Caution: If no condition is specified, DELETE will remove all rows in the table

DELETE FROM table\_name

WHERE condition;

DELETE FROM order

WHERE order\_id < 55;

**4.INSERT-** **Adding New Data**

The INSERT INTO command is used to add new records to a table. You can specify the columns and values you want to insert.

INSERT INTO table\_name (column1, column2, ...)

VALUES (value1, value2, ...);

INSERT INTO students(id,name,age)

VALUE (7, ‘mehedi’, 23)

**5.WHERE- Filtering Records**

The WHERE clause is used in ***SELECT***, ***UPDATE***, and ***DELETE*** statements to filter records based on a specified condition. It allows you to specify which rows should be affected by the command. (***INSERT INTO*** doesn’t need this condition operator)

SELECT \* FROM table\_name WHERE condition;

UPDATE table\_name SET column = value WHERE condition;

DELETE FROM table\_name WHERE condition;

* SELECT - extracts data from a database
* UPDATE: Modify data in existing records.
* DELETE: Remove records.
* INSERT: Add new records.
* WHERE: Apply conditions to filter data
* INSERT INTO - inserts new data into a database
* CREATE DATABASE - creates a new database
* ALTER DATABASE - modifies a database
* CREATE TABLE - creates a new table
* ALTER TABLE - modifies a table
* DROP TABLE - deletes a table
* CREATE INDEX - creates an index (search key)
* DROP INDEX - deletes an index

\*SQL keywords are NOT case sensitive: select is the same as SELECT

\*If you want to return all columns, without specifying every column name, you can use the SELECT \*

LET’S LEARN THE TOPICS->>>

**1.SELECT Statement**

The SELECT statement is used to select data from a database.

SELECT column1, column2, ...

FROM table\_name;

SELECT CustomerName, City FROM Customers;

**2.SELECT DISTINCT Statement**

The SELECT DISTINCT statement is used to return only distinct (different) values.

SELECT DISTINCT *column1*,*column2, ...*  
FROM *table\_name*;

\*\*\*COUNT() return value, count the total number of desired elements

SELECT COUNT(DISTINCT Country) FROM Customers;

not supported in MS access database

SELECT COUNT(\*) FROM(SELECT DISTINCT Country FROM Customers); red portion select the distinct element, and count return those elements.

**3. WHERE Clause**

The WHERE clause is used to filter records.

It is used to extract only those records that fulfill a specified condition

SELECT *column1*,*column2, ...*  
FROM *table\_name*  
WHERE *condition*;

SELECT \* FROM Customers  
WHERE Country='Mexico';

WHERE operators-

=,>,<,>=,<=,<>,BETWEEN,LIKE,IN

* <>stands for not equal !=

**WHERE Price <> 18;**

* **BETWEEN**

**WHERE Price BETWEEN 50 AND 60;**

* **LIKE**- Search for a pattern

**WHERE City LIKE 's%'**

* **IN**- To specify multiple possible values for a column

**WHERE City IN ('Paris','London');**

**4.ORDER BY**

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

SELECT *column1*,*column2, ...*  
FROM *table\_name*  
ORDER BY *column1, column2, ...*ASC|DESC;

SELECT \* FROM Products  
ORDER BY Price;

Sort the products from highest to lowest price:

SELECT \* FROM Products  
ORDER BY Price DESC;

**ORDER BY Several Columns**

SELECT \* FROM Customers  
ORDER BY Country, CustomerName;

**Using Both ASC and DESC**

SELECT \* FROM Customers  
ORDER BY Country ASC, CustomerName DESC;

**5.AND OPERATOR**

The WHERE clause can contain one or many AND operators.

The AND operator is used to filter records based on more than one condition

Example

Select all customers from Spain that starts with the letter 'G':

SELECT \*  
FROM Customers  
WHERE Country = 'Spain' AND CustomerName LIKE 'G%';

Syntax

SELECT *column1*,*column2, ...*  
FROM *table\_name*  
WHERE *condition1* AND *condition2* AND *condition3 ...*;

\*\*\*The AND operator displays a record if *all* the conditions are TRUE.

The OR operator displays a record if *any* of the conditions are TRUE.

SELECT \* FROM Customers  
WHERE Country = 'Germany'  
AND City = 'Berlin'  
AND PostalCode > 12000;

**Combining AND and OR**

Select all Spanish customers that starts with either "G" or "R":

SELECT \* FROM Customers  
WHERE Country = 'Spain' AND (CustomerName LIKE 'G%' OR CustomerName LIKE 'R%');

Without parenthesis, the select statement will return all customers from Spain that starts with a "G", *plus* all customers that starts with an "R", regardless of the country value

SELECT \* FROM Customers  
WHERE *Country = 'Spain'****AND****CustomerName LIKE 'G%'* **OR** *CustomerName LIKE 'R%';*

**6.OR Operator**

The WHERE clause can contain one or more OR operators.

The OR operator is used to filter records based on more than one condition

if you want to return all customers from Germany but also those from Spain:

SELECT \*  
FROM Customers  
WHERE Country = 'Germany' OR Country = 'Spain';

Syntax

SELECT *column1*,*column2, ...*  
FROM *table\_name*  
WHERE *condition1* OR *condition2* OR *condition3 ...*;

**7.NOT Operator**

The NOT operator is used in combination with other operators to give the opposite result, also called the negative result.

Syntax

SELECT *column1*,*column2, ...*  
FROM *table\_name*  
WHERE NOT *condition*;

* Select only the customers that are NOT from Spain:

SELECT \* FROM Customers  
WHERE NOT Country = 'Spain';

* Select customers that does not start with the letter 'A':

SELECT \* FROM Customers  
WHERE CustomerName NOT LIKE 'A%';

* Select customers with a customerID not between 10 and 60:

SELECT \* FROM Customers  
WHERE CustomerID NOT BETWEEN 10 AND 60;

* Select customers that are not from Paris or London:

SELECT \* FROM Customers  
WHERE City NOT IN ('Paris', 'London');

* Select customers with a CustomerId not greater than 50:

SELECT \* FROM Customers  
WHERE NOT CustomerID > 50;

* Select customers with a CustomerID not less than 50:

SELECT \* FROM Customers  
WHERE NOT CustomerId < 50;

**8.INSERT INTO Statement**

The INSERT INTO statement is used to insert new records in a table

INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways:

1. Specify both the column names and the values to be inserted:

INSERT INTO *table\_name* (*column1*,*column2*,*column3*, ...)  
VALUES (*value1*,*value2*,*value3*, ...);

2. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. Here, the INSERT INTO syntax would be as follows:

INSERT INTO *table\_name*  
VALUES (*value1*,*value2*,*value3*, ...);

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)  
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

Insert Multiple Rows

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)  
VALUES  
('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway'),  
('Greasy Burger', 'Per Olsen', 'Gateveien 15', 'Sandnes', '4306', 'Norway'),  
('Tasty Tee', 'Finn Egan', 'Streetroad 19B', 'Liverpool', 'L1 0AA', 'UK');

**9. NULL Values**

A field with a NULL value is a field with no value.

We will have to use the IS NULL and IS NOT NULL operators instead

*IS NULL Syntax*

SELECT *column\_names*FROM *table\_name*  
WHERE *column\_name* IS NULL;

*IS NOT NULL Syntax*

SELECT *column\_names*FROM *table\_name*  
WHERE *column\_name* IS NOT NULL;

The following SQL lists all customers with a NULL value in the "Address" field:

SELECT CustomerName, ContactName, Address  
FROM Customers  
WHERE Address IS NULL;

The following SQL lists all customers with a value in the "Address" field:

SELECT CustomerName, ContactName, Address  
FROM Customers  
WHERE Address IS NOT NULL;

**10.UPDATE Statement**

The UPDATE statement is used to modify the existing records in a table.

UPDATE Syntax

UPDATE *table\_name*  
SET *column1*=*value1*,*column2*=*value2*, ...  
WHERE *condition*;

UPDATE Customers  
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'  
WHERE CustomerID = 1;

UPDATE Customers  
SET ContactName='Juan'; // will convert all name to Juan

**11.DELETE Statement**

The DELETE statement is used to remove existing records in a table.

DELETE Syntax

DELETE FROM *table\_name*

WHERE *condition*;

DELETE FROM Customers

WHERE CustomerName='Alfreds Futterkiste';

**Delete All Records**

DELETE FROM *Customer*;

**Delete a Table**

DROP TABLE Customers;

**12.Aggregate Functions**

An aggregate function is a function that performs a calculation on a set of values, and returns a single value.

Aggregate functions are often used with the GROUP BY clause of the SELECT statement. The GROUP BY clause splits the result-set into groups of values and the aggregate function can be used to return a single value for each group.

The most commonly used SQL aggregate functions are:

* **MIN**() - returns the smallest value within the selected column
* **MAX**() - returns the largest value within the selected column
* **COUNT**() - returns the number of rows in a set
* **SUM**() - returns the total sum of a numerical column
* **AVG**() - returns the average value of a numerical column

Aggregate functions ignore null values (except for COUNT())

**13.MIN() and MAX() Functions**

The MIN() function returns the smallest value of the selected column.

The MAX() function returns the largest value of the selected column.

Find the lowest price in the Price column:

SELECT MIN(Price)  
FROM Products;

Find the highest price in the Price column:

SELECT MAX(Price)  
FROM Products;

**Syntax**

SELECT MIN(*column\_name*)  
FROM *table\_name*  
WHERE *condition*;

SELECT MAX(*column\_name*)  
FROM *table\_name*  
WHERE *condition*;

**Set Column Name (Alias)**

When you use MIN() or MAX(), the returned column will not have a descriptive name. To give the column a descriptive name, use the **AS** keyword:

Example

SELECT MIN(Price) AS SmallestPrice  
FROM Products;

**Use MIN() with GROUP BY**

Here we use the MIN() function and the **GROUP BY** clause, to return the smallest price for each category in the Products table:

***Example***

SELECT MIN(Price) AS SmallestPrice, CategoryID  
FROM Products  
GROUP BY CategoryID;

**14. COUNT() Function**

The COUNT() function returns the number of rows that matches a specified criterion

***Syntax***

SELECT COUNT(*column\_name*)  
FROM *table\_name*  
WHERE *condition*;

Find the total number of rows in the Products table:

SELECT COUNT(\*)  
FROM Products;

Find the number of products where the ProductName is not null:

SELECT COUNT(ProductName)  
FROM Products;

**Add a WHERE Clause**

You can add a WHERE clause to specify conditions:

Find the number of products where Price is higher than 20:

*SELECT COUNT(ProductID)  
FROM Products  
WHERE Price > 20;*

**Ignore Duplicates**

You can ignore duplicates by using the DISTINCT keyword in the COUNT() function.

If DISTINCT is specified, rows with the same value for the specified column will be counted as one.

How many *different* prices are there in the Products table:

SELECT COUNT(DISTINCT Price)  
FROM Products;

**Use an Alias**

Give the counted column a name by using the AS keyword.

Name the column "Number of records":

SELECT COUNT(\*) AS [Number of records]  
FROM Products;

**Use COUNT() with GROUP BY**

Here we use the COUNT() function and the GROUP BY clause, to return the number of records for each category in the Products table:

SELECT COUNT(\*) AS [Number of records], CategoryID  
FROM Products  
GROUP BY CategoryID

**15.SUM() Function**

The SUM() function returns the total sum of a numeric column.

**Syntax**

SELECT SUM(*column\_name*)  
FROM *table\_name*  
WHERE *condition*;

* SELECT SUM(Quantity)  
  FROM OrderDetails;
* SELECT SUM(Quantity)  
  FROM OrderDetails  
  WHERE ProductId = 11;

Here we use the SUM() function and the GROUP BY clause, to return the Quantity for each OrderID in the OrderDetails table:

* SELECT OrderID, SUM(Quantity) AS [Total Quantity]  
  FROM OrderDetails  
  GROUP BY OrderID;

If we assume that each product in the OrderDetails column costs 10 dollars, we can find the total earnings in dollars by multiply each quantity with 10

* SELECT SUM(Quantity \* 10)  
  FROM OrderDetails;

We can also join the OrderDetails table to the Products table to find the actual amount, instead of assuming it is 10 dollars

Join OrderDetails with Products, and use SUM() to find the total amount:

* SELECT SUM(Price \* Quantity)  
  FROM OrderDetails  
  LEFT JOIN Products ON OrderDetails.ProductID = Products.ProductID;

**16.AVG() Function**

The AVG() function returns the average value of a numeric column.

SELECT AVG(Price)  
FROM Products;

SELECT AVG(*column\_name*)  
FROM *table\_name*  
WHERE *condition*;

Return the average price of products in category 1:

SELECT AVG(Price)  
FROM Products  
WHERE CategoryID = 1;

Name the column "average price":

SELECT AVG(Price) AS [average price]  
FROM Products;

\*Return all products with a higher price than the average price:

SELECT \* FROM Products  
WHERE price > (SELECT AVG(price) FROM Products);

Here we use the AVG() function and the GROUP BY clause, to return the average price for each category in the Products table:

SELECT AVG(Price) AS AveragePrice, CategoryID  
FROM Products  
GROUP BY CategoryID;

**17.LIKE Operator**

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the LIKE operator:

* The percent sign % represents zero, one, or multiple characters
* The underscore sign \_ represents one, single character

Select all customers that starts with the letter "a":

SELECT \* FROM Customers

WHERE CustomerName LIKE 'a%';

SELECT *column1, column2, ...*  
FROM *table\_name*  
WHERE *columnN* LIKE *pattern*

**17.a.The \_ Wildcard**

The \_ wildcard represents a single character.

It can be any character or number, but each \_ represents one, and only one, character.

example

Return all customers from a city that starts with 'L' followed by one wildcard character, then 'nd' and then two wildcard characters:

SELECT \* FROM Customers  
WHERE city LIKE 'L\_nd\_\_';

**17.b.The % Wildcard**

The % wildcard represents any number of characters, even zero characters.

example

Return all customers from a city that *contains* the letter 'L':

SELECT \* FROM Customers  
WHERE city LIKE '%L%';

**i.Starts With-** To return records that starts with a specific letter or phrase, add the % at the end of the letter or phrase.

Return all customers that starts with 'a' or starts with 'b':

SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a%' OR CustomerName LIKE 'b%';

**ii.Ends With**- To return records that ends with a specific letter or phrase, add the % at the beginning of the letter or phrase.

Return all customers that starts with "b" and ends with "s":

SELECT \* FROM Customers  
WHERE CustomerName LIKE 'b%s';

**iii.Contains-** To return records that contains a specific letter or phrase, add the % both before and after the letter or phrase.

Return all customers that contains the phrase 'or'

SELECT \* FROM Customers  
WHERE CustomerName LIKE '%or%';

**iv.Combine Wildcards-** Any wildcard, like % and \_ , can be used in combination with other wildcards.

* Return all customers that starts with "a" and are at least 3 characters in length:

SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a\_\_%';

* Return all customers that have "r" in the second position:

SELECT \* FROM Customers

WHERE CustomerName LIKE ‘\_r%’

**v.Without Wildcard –** If no wildcard is specified, the phrase has to have an exact match to return a result.

* Return all customers from Spain:

SELECT \* FROM Customers  
WHERE Country LIKE 'Spain';

**18.WILDCARD**

Wildcard characters are used with the [LIKE](https://www.w3schools.com/sql/sql_like.asp) operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

**A screenshot of a computer

Description automatically generated**

**Using the % Wildcard**

The % wildcard represents any number of characters, even zero characters.

Example

Return all customers that ends with the pattern 'es':

SELECT \* FROM Customers  
WHERE CustomerName LIKE '%es';

**Using the \_ Wildcard**

The \_ wildcard represents a single character.

It can be any character or number, but each \_ represents one, and only one, character.

Example

Return all customers with a City starting with any character, followed by "ondon":

SELECT \* FROM Customers  
WHERE City LIKE '\_ondon';

**Using the [] Wildcard**

The [] wildcard returns a result if *any* of the characters inside gets a match.

Example

Return all customers starting with either "b", "s", or "p":

SELECT \* FROM Customers  
WHERE CustomerName LIKE '[bsp]%';

**Using the - Wildcard**

The - wildcard allows you to specify a range of characters inside the [] wildcard.

Example

Return all customers starting with "a", "b", "c", "d", "e" or "f":

SELECT \* FROM Customers  
WHERE CustomerName LIKE '[a-f]%';

**Combine Wildcards**

Any wildcard, like % and \_ , can be used in combination with other wildcards.

Example

Return all customers that starts with "a" and are at least 3 characters in length:

SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a\_\_%';

**Without Wildcard**

If no wildcard is specified, the phrase has to have an exact match to return a result.

Example

Return all customers from Spain:

SELECT \* FROM Customers  
WHERE Country LIKE 'Spain';

**19.IN Operator**

The IN operator allows you to specify multiple values in a WHERE clause

Return all customers from 'Germany', 'France', or 'UK'

SELECT \* FROM Customers  
WHERE Country IN ('Germany', 'France', 'UK');

Return all customers that have an order in the **Order** table:

SELECT \* FROM Customers  
WHERE CustomerID IN (SELECT CustomerID FROM Orders);

**20.BETWEEN Operator**

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

SELECT \* FROM Products  
WHERE Price BETWEEN 10 AND 20;

**21.Aliases**

SQL aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of that query.

An alias is created with the AS keyword.

SELECT CustomerID AS ID, CustomerName AS Customer  
FROM Customers;

Using Aliases With a Space Character

If you want your alias to contain one or more spaces, like "My Great Products", surround your alias with square brackets or double quotes.

SELECT ProductName AS [My Great Products]  
FROM Products;

Or// SELECT ProductName AS "My Great Products"  
FROM Products;

**22. Joins**

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

**23.INNER JOIN**

**24. LEFT JOIN Keyword**

**25. RIGHT JOIN Keyword**

**26. FULL OUTER JOIN Keyword**

**27. Self Join**

**28. GROUP BY Statement**

ER DIAGRAM IN DBMS

* ER model stands for an Entity-Relationship model. It is a high-level data model. This model is used to define the data elements and relationship for a specified system.
* It develops a conceptual design for the database. It also develops a very simple and easy to design view of data.
* In ER modeling, the database structure is portrayed as a diagram called an entity-relationship diagram

**For example,** Suppose we design a school database. In this database, the student will be an entity with attributes like address, name, id, age, etc. The address can be another entity with attributes like city, street name, pin code, etc and there will be a relationship between them.

A diagram of a student

Description automatically generated

**Component of ER Diagram**

**1.Entity**

An entity may be any object, class, person or place. In the ER diagram, an entity can be represented as rectangles.

Consider an organization as an example- manager, product, employee, department etc. can be taken as an entity

A diagram of er model

Description automatically generated

**a. Weak Entity**

An entity that depends on another entity called a weak entity. The weak entity doesn't contain any key attribute of its own. The weak entity is represented by a double rectangle

A long thin line of metal

Description automatically generated with medium confidence

**2.Attribute**

The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute.

**For example,** id, age, contact number, name, etc. can be attributes of a student.

A diagram of a student

Description automatically generated

**a. Key Attribute**

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined

A diagram of a student

Description automatically generated

**b. Composite Attribute**

An attribute that composed of many other attributes is known as a composite attribute. The composite attribute is represented by an ellipse, and those ellipses are connected with an ellipse.

A diagram of a name

Description automatically generated

**c. Multivalued Attribute**

An attribute can have more than one value. These attributes are known as a multivalued attribute. The double oval is used to represent multivalued attribute.

It is not possible to represent multiple values in a single column.

A black and white oval with text

Description automatically generated

**d. Derived Attribute**

An attribute that can be derived from another attribute is known as a derived attribute. It can be represented by a dashed ellipse.

**For example,** A person's age changes over time and can be derived from another attribute like Date of birth

A diagram of a student

Description automatically generated

**3. Relationship**

A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent the relationship.

A black and white diamond with black text

Description automatically generated

Types of relationship are as follows:

**a. One-to-One Relationship**

When only one instance of an entity is associated with the relationship, then it is known as one-to-one relationship.

**For example,** A female can marry to one male, and a male can marry to one female

A diamond shaped sign with black text

Description automatically generated

**b. One-to-many relationship**

When only one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then this is known as a one-to-many relationship.

**For example,** Scientist can invent many inventions, but the invention is done by the only specific scientist.

A diamond shaped sign with black text

Description automatically generated

**c. Many-to-one relationship**

When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.

**For example,** Student enrolls for only one course, but a course can have many students.

A black and white diamond with black text

Description automatically generated

**d. Many-to-many relationship**

When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.

**For example,** Employee can assign by many projects and project can have many employees

A black and white diamond with black text

Description automatically generated

**KEYS**

* Keys play an important role in the relational database.
* It is used to uniquely identify any record or row of data from the table. It is also used to establish and identify relationships between tables.

For example, ID is used as a key in the student table because it is unique for each student. In the PERSON table, passport\_number, license\_number, SSN are keys since they are unique for each person.

A blue rectangular sign with white text

Description automatically generated

**Types of keys:**

We just need the primary key and Foreign Key

**Primary Key**

It is the first key used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys, as we saw in the PERSON table. The key which is most suitable from those lists becomes a primary key.

Unique key to identify an entity

**Foreign Key**

Foreign keys are the column of the table used to point to the primary key of another table.

A blue rectangular object with red line

Description automatically generated

* Every employee works in a specific department in a company, and employee and department are two different entities. So, we can't store the department's information in the employee table. That's why we link these two tables through the primary key of one table.
* We add the primary key of the DEPARTMENT table, Department\_Id, as a new attribute in the EMPLOYEE table.
* In the EMPLOYEE table, Department\_Id is the foreign key, and both the tables are related.

**Generalization**

**Generalization** is the process of extracting common attributes and creating a generalized superclass that encompasses the common features of several entities. It is the inverse of specialization. For example, Student and Teacher entities can generalize into a Person entity, with common attributes like name and address moved to Person.

**Specialization**

**Specialization** in DBMS is the process of defining sub-entities within a larger entity by adding specific attributes or roles. It’s the opposite of **generalization** and is used when we want to model sub-categories that have more specific characteristics than their parent entity

**Example of Specialization**

Suppose we have an entity called **Employee** in a company database. The **Employee** entity has attributes like EmployeeID, Name, and Address. However, employees can have specific roles such as **Engineer** and **Manager**.

Using specialization, we can create sub-entities **Engineer** and **Manager** with additional attributes specific to each:

* **Employee**: EmployeeID, Name, Address
* **Engineer**: Inherits from Employee; additional attribute: TechnicalSkill
* **Manager**: Inherits from Employee; additional attribute: TeamSize

**Aggregation**

**Aggregation** in DBMS is a concept used to express a relationship between a relationship set and an entity set. It’s useful when we need to represent a complex relationship involving multiple entities and relationships.

**Why Aggregation is Needed**

Sometimes, a relationship set may need to participate in another relationship. Aggregation allows us to treat this relationship set as a single entity, simplifying the structure and making it easier to represent in an ER (Entity-Relationship) diagram.

**Reduction of ER diagram to Table**

The database can be represented using the notations, and these notations can be reduced to a collection of tables.

In the database, every entity set or relationship set can be represented in tabular form

A diagram of a course

Description automatically generated

There are some points for converting the ER diagram to the table:

* **Entity type becomes a table.**

In the given ER diagram, LECTURE, STUDENT, SUBJECT and COURSE forms individual tables.

* **All single-valued attribute becomes a column for the table.**

In the STUDENT entity, STUDENT\_NAME and STUDENT\_ID form the column of STUDENT table. Similarly, COURSE\_NAME and COURSE\_ID form the column of COURSE table and so on.

* **A key attribute of the entity type represented by the primary key.**

In the given ER diagram, COURSE\_ID, STUDENT\_ID, SUBJECT\_ID, and LECTURE\_ID are the key attribute of the entity.

* **The multivalued attribute is represented by a separate table.**

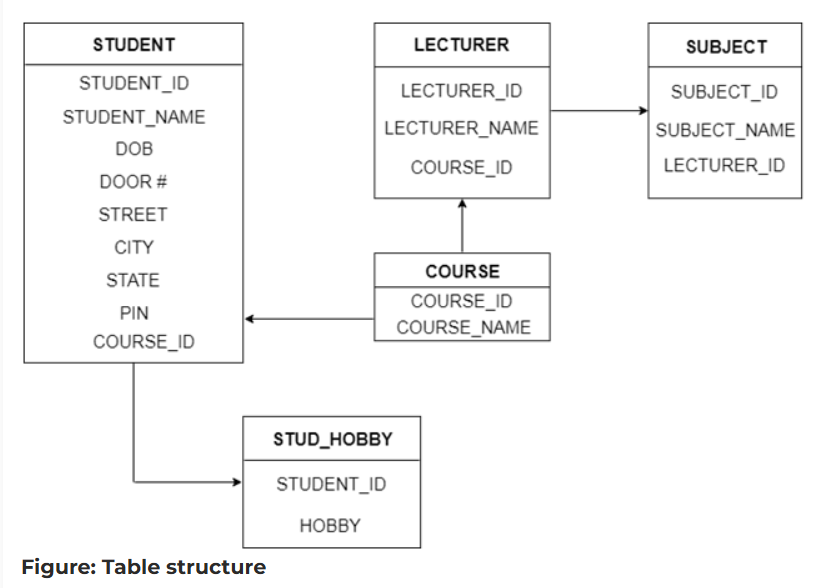
In the student table, a hobby is a multivalued attribute. So it is not possible to represent multiple values in a single column of STUDENT table. Hence we create a table STUD\_HOBBY with column name STUDENT\_ID and HOBBY. Using both the column, we create a composite key.

* **Composite attribute represented by components.**

In the given ER diagram, student address is a composite attribute. It contains CITY, PIN, DOOR#, STREET, and STATE. In the STUDENT table, these attributes can merge as an individual column.

* **Derived attributes are not considered in the table.**

In the STUDENT table, Age is the derived attribute. It can be calculated at any point of time by calculating the difference between current date and Date of Birth.

Using these rules, you can convert the ER diagram to tables and columns and assign the mapping between the tables. Table structure for the given ER diagram is as below: 

**Normalization**

A large database defined as a single relation may result in data duplication. This repetition of data may result in:

* Making relations very large.
* It isn't easy to maintain and update data as it would involve searching many records in relation.
* Wastage and poor utilization of disk space and resources.
* The likelihood of errors and inconsistencies increases

**What is Normalization?**

* Normalization is the process of organizing the data in the database.
* Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
* Normalization divides the larger table into smaller and links them using relationships.
* The normal form is used to reduce redundancy from the database table.

A chart of different types of normal forms

Description automatically generated

A screenshot of a computer

Description automatically generated

Advantages of Normalization

* Normalization helps to minimize data redundancy.
* Greater overall database organization.
* Data consistency within the database.
* Much more flexible database design.
* Enforces the concept of relational integrity.

Disadvantages of Normalization

* The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF.
* It is very time-consuming and difficult to normalize relations of a higher degree.

**1.First Normal Form (1NF)**

* A relation will be 1NF if it contains an atomic value.
* It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
* First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

**Example:** Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP\_PHONE.

A screenshot of a phone number

Description automatically generated

The decomposition of the EMPLOYEE table into **1NF** has been shown below

A screenshot of a phone number

Description automatically generated

**2.Second Normal Form (2NF**)

* In the 2NF, relational must be in 1NF.
* In the second normal form, all non-key attributes are fully functional dependent on the primary key

**Example:** Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

A screenshot of a computer

Description automatically generated

In the given table, non-prime attribute TEACHER\_AGE is dependent on TEACHER\_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**3.Third Normal Form (3NF)**

* A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
* 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
* If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

**Example:** **A table with numbers and letters

Description automatically generated**

Super key in the table above: {EMP\_ID}, {EMP\_ID, EMP\_NAME}, {EMP\_ID, EMP\_NAME, EMP\_ZIP}....so on

Candidate key: {EMP\_ID}

**Non-prime attributes:** In the given table, all attributes except EMP\_ID are non-prime.

Here, EMP\_STATE & EMP\_CITY dependent on EMP\_ZIP and EMP\_ZIP dependent on EMP\_ID. The non-prime attributes (EMP\_STATE, EMP\_CITY) transitively dependent on super key(EMP\_ID). It violates the rule of third normal form.

That's why we need to move the EMP\_CITY and EMP\_STATE to the new <EMPLOYEE\_ZIP> table, with EMP\_ZIP as a Primary key.

A screenshot of a computer

Description automatically generated

A screenshot of a table

Description automatically generated

*Here, I don’t understand the super key, what is super key in dbms?*

A **Super Key** in a database is a set of one or more attributes (columns) that can uniquely identify a row in a table*.*

**Possible Super Keys for a employee Table**

1. **{Employee\_ID}**: Since Employee\_ID is unique for each employee, it can uniquely identify each row.
2. **{Email}**: Each employee has a unique email, so this column alone can uniquely identify each row.
3. **{Employee\_ID, Name}**: Although Employee\_ID alone is enough, adding Name to it still allows unique identification. Thus, {Employee\_ID, Name} is also a super key.
4. **{Employee\_ID, Phone}**: Combining Employee\_ID with Phone can also uniquely identify each row, even though Employee\_ID alone would suffice

**Note**: A super key can include extra columns that aren’t necessary for unique identification. For example, {Employee\_ID, Name} is a super key, but {Employee\_ID} alone is sufficient to identify a row. So, {Employee\_ID} would be a **candidate key** (minimal super key), while {Employee\_ID, Name} is still a valid super key

**Super Key vs. Candidate Key**

* **Super Key**: Any combination of columns that can uniquely identify rows in the table.
* **Candidate Key**: A minimal super key — no extra columns. It’s the simplest super key that still maintains uniqueness.