DBMS Module 2

1) Explain union, Intersection and Minus Operations of Relational algebra with examples

Consider these Tables for the examples:

STUDENT	
First	**Last**
Aisha	Arora
Bikash	Dutta
Makku	Singh
Raju	Chopra

FACULTY	
FirstN	**LastN**
Raj	Kumar
Honey	Chand
Makku	Singh
Karan	Rao

Union Operation

- It performs binary union between two given relations and is defined as:-
  - $R \cup S = \{ t \mid t \in R \text{ or } t \in S\}$, Where **R** and **S** are database relations.
- For a union operation, the following conditions must hold:-
  - R and S must have the same number of attributes.
  - Attribute domains must be compatible.
  - Duplicate tuples are eliminated automatically.

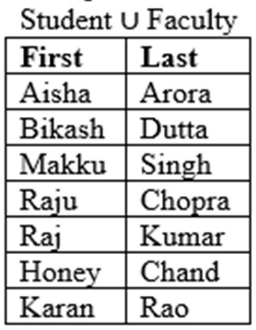
Intersection Operation

- It performs a binary intersection between two given relations and is defined as:-
  - $R \cap S = \{ t \mid t \in R \text{ and } t \in S\}$, Where **R** and **S** are database relations.
- The result of Intersection operation, which is denoted by R ∩ S, is a relation that basically includes all the tuples that are present in both R and S.

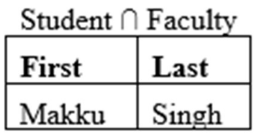
Minus Operation

- The result of set difference operation is tuples, which are present in one relation but are not in the second relation.
  - Notation: $R - S$
- Finds all the tuples that are present in **R** but not in **S**.

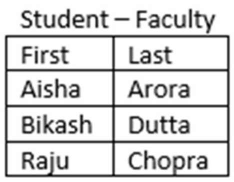
Examples:


Student ∪ Faculty

First	Last
Aisha	Arora
Bikash	Dutta
Makku	Singh
Raju	Chopra
Raj	Kumar
Honey	Chand
Karan	Rao


Student ∩ Faculty

First	Last
Makku	Singh


Student – Faculty

First	Last
Aisha	Arora
Bikash	Dutta
Raju	Chopra

2) **Describe Selection and Projector Operator in Relational Algebra and mention the difference between them with examples.**

Select Operation (σ)

- It selects tuples that satisfy the given predicate from a relation.
- Notation − $\sigma_p(r)$
- Where σ stands for selection predicate and r stands for relation. p is prepositional logic formula which may use connectors like and, or, and not. These terms may use relational operators like − =, ≠, ≥, < , >, ≤.

Example:

$\sigma_{subject}$ = "database"(Books)

Output − Selects tuples from books where subject is 'database'.

$\sigma_{subject}$ = "database" and price = "450"(Books)

Output − Selects tuples from books where subject is 'database' and 'price' is 450.

Project Operation (∏)

- It projects column(s) that satisfy a given predicate.
- Notation − $\prod_{A1, A2, An}$ (r)
- Where A1, A2 , An are attribute names of relation r.
- Duplicate rows are automatically eliminated, as relation is a set.

Example:

$\prod_{subject, author}$ (Books)

Selects and projects columns named as subject and author from the relation Books.

Differences between Select and Project:

Feature	**Select (σ)**	**Project (π)**
Function	Filters rows	Selects columns
Symbol	σ	π
Impact on Columns	No direct impact	Reduces columns
Indirect impact on Columns	Yes, if selection condition involves specific columns	No
Subset selection type	Horizontal	Vertical

3) **Develop the following queries in Using Relational Algebra.**

i) **Find the names of all the employees whose salary is greater than 30000.**

- Command: $\pi_{(name)} (\sigma_{(salary > 30000)}(Employee))$
- σ(salary > 30000)(Employee): This part selects rows from the Employee relation where the "salary" is greater than 30000.
- π(name) (...): This projects only the "name" column from the result of the selection operation.

ii) **Retrieve the name and emp id of all employees.**

- Command: $\pi_{(name,\ emp\_id)}(Employee)$
- π(name, emp_id)(Employee): This directly projects both "name" and "emp_id" columns from the Employee relation.

iii) **Find the fname and lname of employees in department 4 that earn > 50000**

- $\pi_{(fname,\ lname)} (\sigma_{(department = 4\ AND\ salary > 50000)}(Employee))$
- σ(department = 4 AND salary > 50000)(Employee): This selects rows where "department" is equal to 4 and "salary" is greater than 50000 from the Employee relation.
- π(fname, lname) (...): This projects only the "fname" and "lname" columns from the result of the selection operation.

4) Explain different types of Joins in SQL

The SQL Join clause is used to combine data from two or more tables in a database. When the related data is stored across multiple tables, joins help you to retrieve records combining the fields from these tables using their foreign keys.

Syntax:

```
SELECT column_name(s)
FROM table1
JOIN table2;
```

Types of joins in SQL

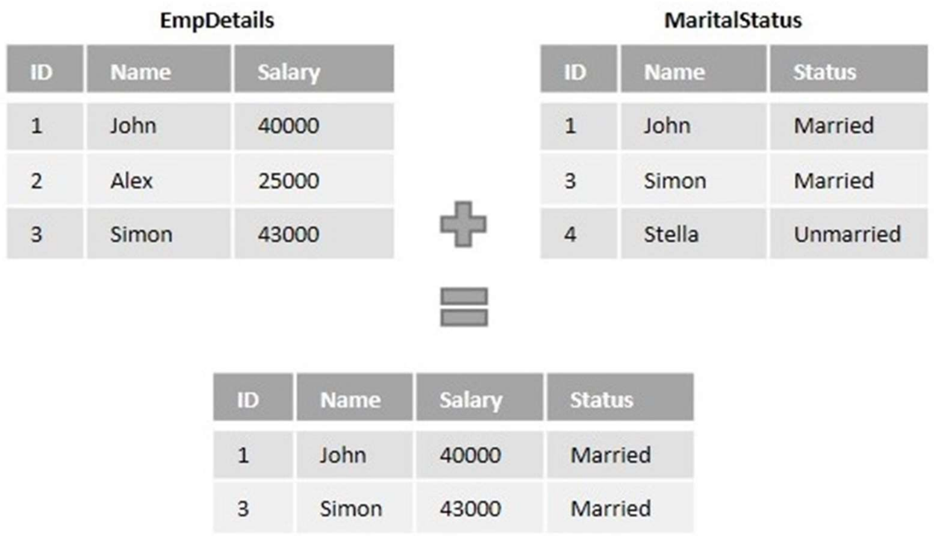
SQL provides various types of Joins that are categorized based on the way data across multiple tables are joined together. They are listed as follows −

Inner Join

An INNER JOIN is the default join which retrieves the intersection of two tables. It compares each row of the first table with each row of the second table. If the pairs of these rows satisfy the join-predicate, they are joined together.

Syntax:

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```



EmpDetails

ID	Name	Salary
1	John	40000
2	Alex	25000
3	Simon	43000

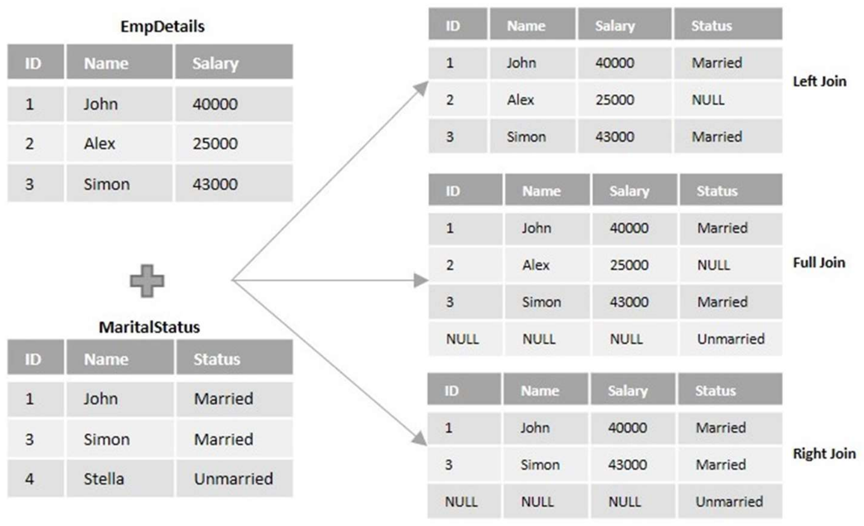
MaritalStatus

ID	Name	Status
1	John	Married
3	Simon	Married
4	Stella	Unmarried

ID	Name	Salary	Status
1	John	40000	Married
3	Simon	43000	Married

Outer Join

An Outer Join retrieves all the records in two tables even if there is no counterpart row of one table in another table, unlike Inner Join. Outer join is further divided into three subtypes - Left Join, Right Join and Full Join.



EmpDetails

ID	Name	Salary
1	John	40000
2	Alex	25000
3	Simon	43000

MaritalStatus

ID	Name	Status
1	John	Married
3	Simon	Married
4	Stella	Unmarried

Left Join

ID	Name	Salary	Status
1	John	40000	Married
2	Alex	25000	NULL
3	Simon	43000	Married

Full Join

ID	Name	Salary	Status
1	John	40000	Married
2	Alex	25000	NULL
3	Simon	43000	Married
NULL	NULL	NULL	Unmarried

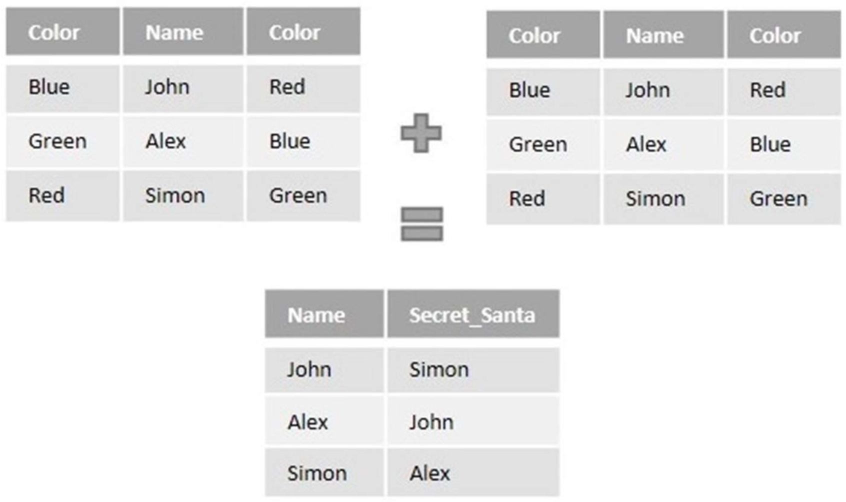
Right Join

ID	Name	Salary	Status
1	John	40000	Married
3	Simon	43000	Married
NULL	NULL	NULL	Unmarried

- LEFT JOIN – returns all rows from the left table, even if there are no matches in the right table.
  Syntax:
  ```
  SELECT column_name(s)
  FROM table1
  LEFT JOIN table2
  ON table1.column_name = table2.column_name;
  ```
- RIGHT JOIN – returns all rows from the right table, even if there are no matches in the left table.
  Syntax:
  ```
  SELECT table1.column1, table2.column2...
  FROM table1
  RIGHT JOIN table2
  ON table1.common_field = table2.common_field;
  ```
- FULL JOIN – returns rows when there is a match in one of the tables.
  Syntax:
  ```
  SELECT column_name(s)
  FROM table1
  FULL JOIN table2
  ON table1.column_name = table2.column_name;
  ```

Other Joins

- SELF JOIN – is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.



Color	Name	Color
Blue	John	Red
Green	Alex	Blue
Red	Simon	Green

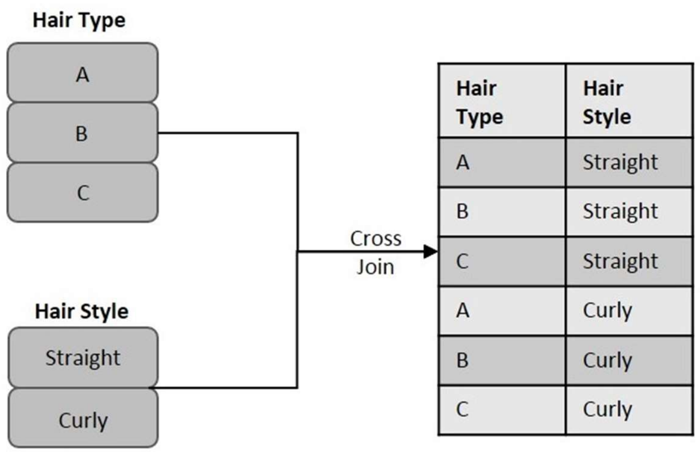
Color	Name	Color
Blue	John	Red
Green	Alex	Blue
Red	Simon	Green

Name	Secret_Santa
John	Simon
Alex	John
Simon	Alex

Syntax:

```
SELECT column_name(s)
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

- CROSS JOIN – returns the Cartesian product of the sets of records from the two or more joined tables.



Hair Type	Hair Style
A	Straight
B	Straight
C	Straight
A	Curly
B	Curly
C	Curly

Syntax:

```
SELECT column_name(s)
FROM table1
CROSS JOIN table2;
```

5) **Describe the different Join Operators with examples.**

Join is a combination of a Cartesian product followed by a selection process. A Join operation pairs two tuples from different relations, if and only if a given join condition is satisfied.

Theta (θ) Join

- Theta join combines tuples from different relations provided they satisfy the theta condition. The join condition is denoted by the symbol **θ**.
- Notation: $R1 \bowtie_{\theta} R2$
- R1 and R2 are relations having attributes (A1, A2, .., An) and (B1, B2,.. ,Bn) such that the attributes don't have anything in common, that is $R1 \cap R2 = \Phi$.
- Theta join can use all kinds of comparison operators.

Equijoin

- When Theta join uses only equality comparison operator, it is said to be equijoin.

Natural Join (⋈)

- Natural join does not use any comparison operator. It does not concatenate the way a Cartesian product does.
- We can perform a Natural Join only if there is at least one common attribute that exists between two relations.
- In addition, the attributes must have the same name and domain.
- Natural join acts on those matching attributes where the values of attributes in both the relations are same.

Left Outer Join(R Left Outer Join S)

- All the tuples from the Left relation, R, are included in the resulting relation.
- If there are tuples in R without any matching tuple in the Right relation S, then the S-attributes of the resulting relation are made NULL.

Right Outer Join: (R Right Outer Join S)

- All the tuples from the Right relation, S, are included in the resulting relation.
- If there are tuples in S without any matching tuple in R, then the R-attributes of resulting relation are made NULL.

Full Outer Join: (R Full Outer Join S)

- All the tuples from both participating relations are included in the resulting relation.
- If there are no matching tuples for both relations, their respective unmatched attributes are made NULL.

6) Explain different steps of ER to Relational Mapping algorithm.

Step 1: Mapping of Regular Entity Types.

- For each regular (strong) entity type E in the ER schema, choose one of the key attributes of E as the primary key for R.
- If the chosen key of E is composite, the set of simple attributes will t form the primary key of R.

Step 2: Mapping of Weak Entity Types

- For each weak entity type W in the ER schema with owner entity type E, create a relation R & include all simple attributes of W as attributes of R.
- Also, include primary key relations as the foreign key of R.
- The primary key of R is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any.

Step 3: Mapping of Binary 1:1 Relation Types

- For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R.
- Choose one of the relation S and include a foreign key in S, the primary key of T. Choose an entity type with total participation in R in the role of S.

Step 4: Mapping of Binary 1:N Relationship Types.

- For each regular binary 1:N relationship type R, identify the relation S that represent the participating entity type at the N-side of the relationship type.
- Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R and include any simple attributes of the 1:N relation type as attributes of S

Step 5: Mapping of Binary M:N Relationship Types.

- For each regular binary M:N relationship type R, create a new relation S to represent R.
- Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S. Also include any simple attributes of the M:N relationship as attributes of S.

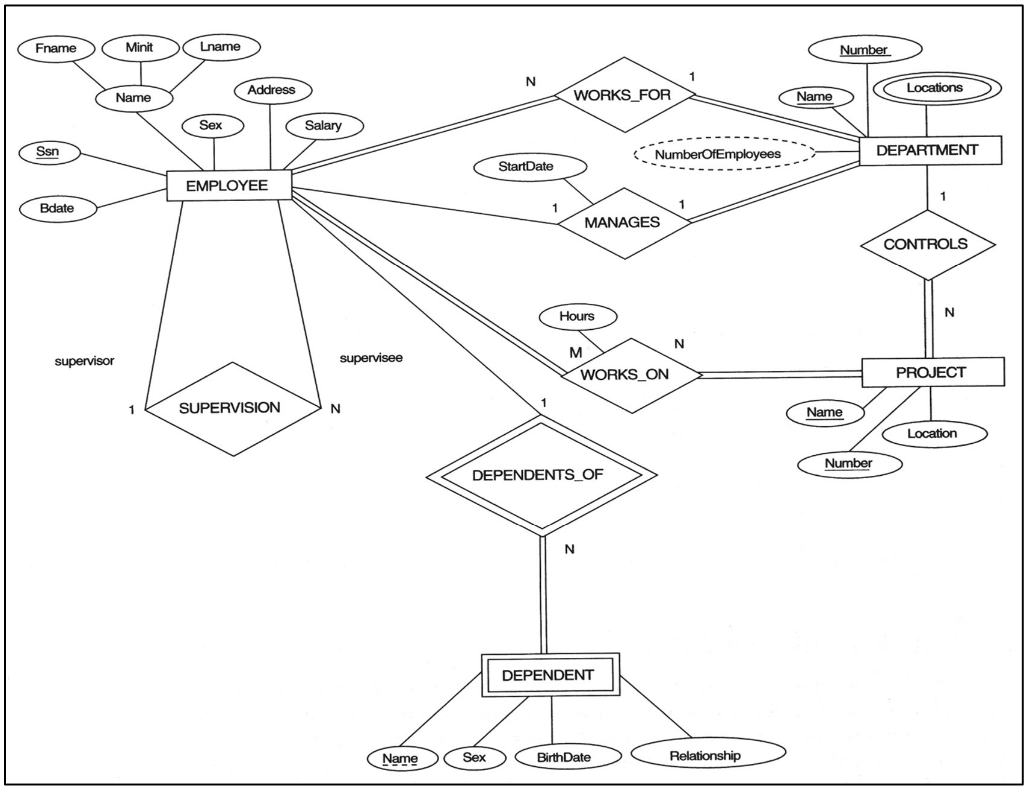
Step 6: Mapping of Multivalued attributes.

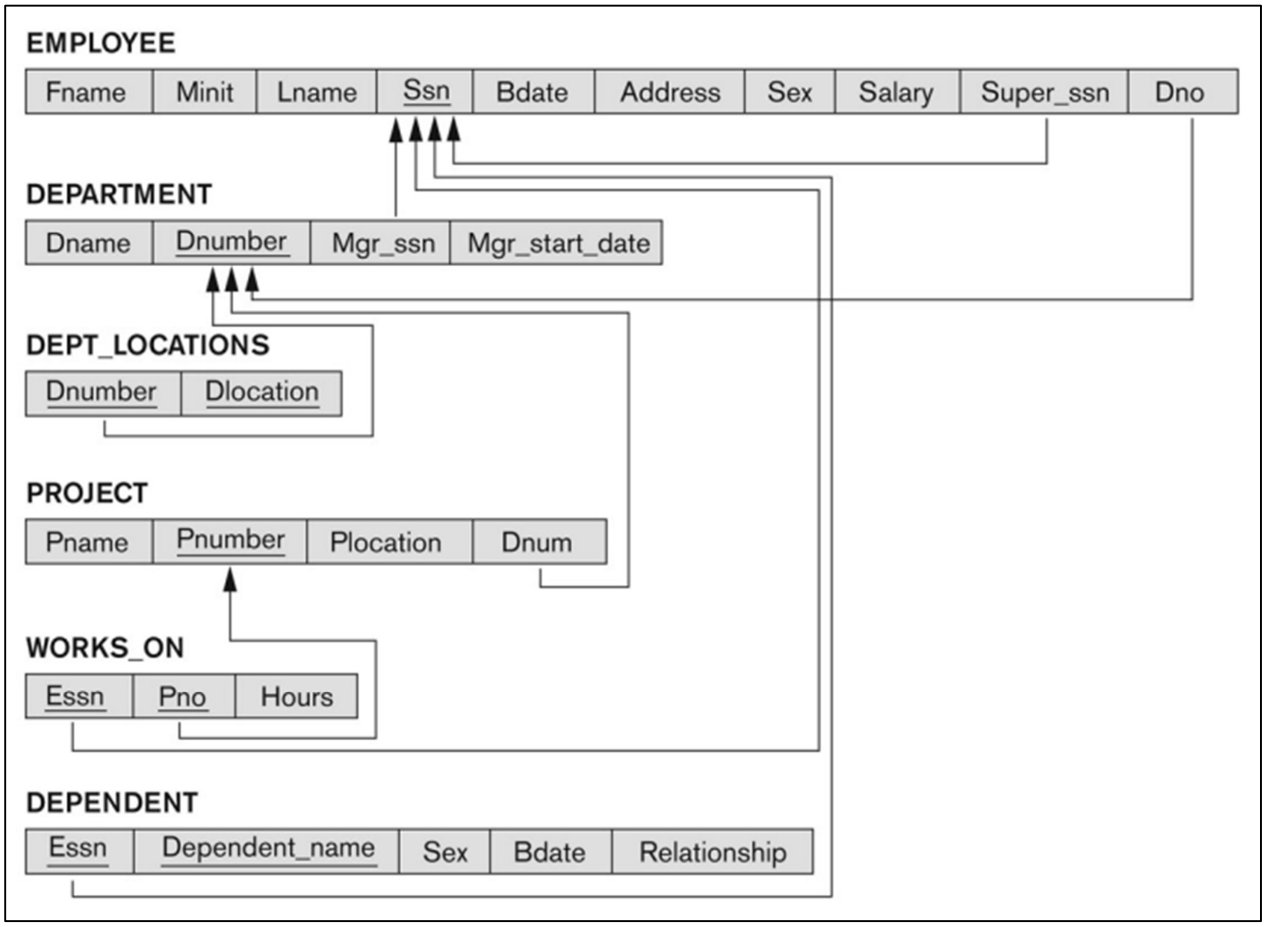
- For each multivalued attribute A, create a new relation R.
- This relation R will include an attribute corresponding to A, plus the primary key attribute K-as a foreign key in R-of the relation that represents the entity type of relationship type that has A as an attribute.

Step 7: Mapping of N-ary Relationship Types.

- For each n-ary relationship type R, where n>2, create a new relationship S to represent R.
- Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the n-ary relationship type as attributes of S

7) Convert the following ER Diagram to Relational Model





EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

DEPT_LOCATIONS

Dnumber	Dlocation

PROJECT

Pname	Pnumber	Plocation	Dnum

WORKS_ON

Essn	Pno	Hours

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

Module 3

1) Explain the data types that are allowed for SQL attributes.

- **Integer (INT):**
  - **Description:** Stores whole numbers (positive, negative, or zero).
  - **Syntax:** `column_name INT`
  - **Example:**

```
CREATE TABLE customers (customer_id INT PRIMARY KEY, age INT );
```

- **Float:**
  - **Description:** Stores single-precision floating-point numbers, which can represent real numbers with decimal places.
  - **Syntax:** column_name FLOAT
  - **Example:**

```
CREATE TABLE products ( price FLOAT NOT NULL );
```

- **Char(n):**
  - **Description:** Stores a fixed-length character string, where 'n' specifies the maximum number of characters allowed. Pads shorter strings with spaces to reach the defined length.
  - **Syntax:** column_name CHAR(n)
  - **Example:**

```
CREATE TABLE states ( state_code CHAR(2) );
```

- **Varchar(n):**
  - **Description:** Stores a variable-length character string, with a maximum capacity of 'n' characters. Only uses the space needed for the actual data.
  - **Syntax:** column_name VARCHAR(n)
  - **Example:**

```
CREATE TABLE users (username VARCHAR(30) UNIQUE, email VARCHAR(25));
```

- **Date:**
  - **Description:** Stores only the year, month, and day information.
  - **Syntax:** column_name DATE
  - **Example:**

```
CREATE TABLE orders( order_id INT PRIMARY KEY, order_date DATE NOT
NULL);
```

- **Time:**
  - **Description:** Stores only the time component, including hours, minutes, and seconds (sometimes with fractions of a second).
  - **Syntax:** column_name TIME
  - **Example:**

```
CREATE TABLE flights( flight_id INT PRIMARY KEY, departure_time TIME
NOT NULL );
```

- **Datetime:**
  - **Description:** Combines both date and time information in a single column.
  - **Syntax:** column_name DATETIME
  - **Example:**

```
CREATE TABLE transactions ( transaction_id INT PRIMARY KEY,
timestamp DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP);
```

2) **Outline the different constraints in SQL while creating table.**

The constraints while creating a table in SQL are:

Primary Key

- The PRIMARY KEY constraint uniquely identifies each record in a table.
- Primary keys must contain UNIQUE values and cannot contain NULL values.
- A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

Syntax:

```
CREATE TABLE table_name ( attribute_name datatype PRIMARY KEY);
```

Foreign Key

- The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
- A FOREIGN KEY is a field in one table, that refers to the PRIMARY KEY in another table.
- The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.
- The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

Syntax:

```
CREATE TABLE table_name (attribute_name datatype FOREIGN KEY
REFERENCES referenced_table(referenced_primary_key));
```

Not Null

- By default, a column can hold NULL values.
- The NOT NULL constraint enforces a column to NOT accept NULL values.
- This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

Syntax:

```
CREATE TABLE table_name (attribute_name datatype NOT NULL);
```

Unique

- The UNIQUE constraint ensures that all values in a column are different.
- Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint.
- However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

Syntax:

```
CREATE TABLE table_name (attribute_name datatype UNIQUE);
```

3) **Explain the schema change statements of SQL.**

In SQL, schema changes are modifications made to the structure of your database. This can involve altering existing tables, views, indexes, or even creating or dropping them entirely. There are two primary statements used for schema changes:

- ALTER TABLE: This statement allows you to modify the definition of a table.
  - Add a new column: Specify the column name, data type, and any.
    `ALTER TABLE customers ADD phone_number VARCHAR(15);`
  - Modify an existing column: Change the data type, rename the column, or alter constraints.
    `ALTER TABLE products MODIFY COLUMN price FLOAT;`
  - Drop a column: Remove a column from the table.
    `ALTER TABLE posts DROP COLUMN created_at;`
- DROP TABLE: This permanently removes the specified table along with all its data, columns etc.
  - Some database systems allow specifying CASCADE with DROP TABLE to automatically drop dependent objects as well.
    `DROP TABLE orders;`

4) **Write the SQL queries for the following:**

a) **Write the syntax to Create, Alter and Drop table.**

- `CREATE TABLE table_name ( column1 data_type[constraint1, constraint2,...], column2 data_type[constraint1, constraint2...]);`
- `ALTER TABLE table_name ADD COLUMN new_column_name data_type [constraint1, constraint2, ...], OR ALTER COLUMN existing_column_name data_type [constraint1, constraint2, ...], OR DROP COLUMN column_to_drop;`
- `DROP TABLE table_name;`

b) **Write SQL Query to Create Employee table with the following attributes: eid, ename and salary.**

- `CREATE TABLE employee ( eid INT PRIMARY KEY, ename VARCHAR(50) NOT NULL, salary DECIMAL(10,2) NOT NULL );`

c) **Alter table employee by adding one more attribute called address.**

- `ALTER TABLE employee ADD COLUMN address VARCHAR(25);`

d) **Give syntax to drop table employee and drop column salary.**

- `DROP TABLE employee;`
- `ALTER TABLE employee DROP COLUMN salary;`

5) **Write the SQL queries for the following:**

a) **Retrieve the birth date and address of employee whose employee id is 10.**

- `SELECT dob, address FROM employee WHERE eid = 10;`

b) **Retrieve the name and address of all employees who work for 'Research' department.**

- `SELECT ename, address FROM employee WHERE department = 'Research';`

c) **Retrieve all employees in department 5 whose salary is between 30000 and 40000**

- `SELECT * FROM employee WHERE department = 5 AND salary BETWEEN 30000 AND 40000;`

d) **Retrieve distinct salaries of employees.**

- `SELECT DISTINCT salary FROM employee;`

6) **Explain the ALTER TABLE command. Explain how the new constraint can be added and also an existing constraint can be removed using suitable examples.**

The ALTER TABLE command in SQL is a powerful tool for modifying the structure of existing tables. It allows you to add new features, change existing definitions, or remove unnecessary elements without completely recreating the table.

- You can use ALTER TABLE to add various constraints to existing columns in a table. These constraints enforce data integrity and ensure the validity of your data.
  - `ALTER TABLE customers ADD CONSTRAINT pk_customers PRIMARY KEY (customer_id);`
  - `ALTER TABLE orders ALTER COLUMN order_date SET NOT NULL;`
  - `ALTER TABLE orders ADD CONSTRAINT fk_orders_customer FOREIGN KEY (customer_id) REFERENCES customers(customer_id);`
- You can also use ALTER TABLE to remove constraints that are no longer needed.
  - `ALTER TABLE products DROP CONSTRAINT unique_product_name;`

7) Explain INSERT, DELETE, UPDATE statements in SQL taking suitable examples.

In SQL, modifying data within tables is achieved through three primary statements: INSERT, UPDATE, and DELETE.

INSERT

- **Purpose:** Used to add new rows (records) to an existing table.
- **Syntax:**
```
INSERT INTO table_name (column1, column2, ...) VALUES (value1,
value2, ...);
```
- **Example:**
```
INSERT INTO customers (customer_name, email, phone_number) VALUES
('John Doe','john.doe@example.com', '123-456-7890');
```

UPDATE

- **Purpose:** Used to modify existing data within a table based on a specific condition.
- **Syntax:**
```
UPDATE table_name SET column1 = new_value1, column2 = new_value2,
... WHERE condition;
```
- **Example:**
```
UPDATE products SET price = price * 1.1  -- Increase price by 10%
WHERE category = ‘Electronics’;
```

DELETE

- **Purpose:** Used to remove rows from a table based on a specific condition.
- **Syntax:**
```
DELETE FROM table_name WHERE condition;
```
- **Example:**
```
DELETE FROM orders WHERE order_status = 'Cancelled';
```

8) **Explain the aggregate functions in SQL? Explain with examples.**

Aggregate functions in SQL perform calculations on groups of data in a table and return a single summarized value. They are essential for condensing large datasets and generating meaningful insights.

COUNT(*)

- **Purpose:** Counts the number of rows in a table or the number of non-null values in a specific column.
- **Example:**

```
SELECT COUNT(*) FROM customers;  -- Counts all customer records
```

SUM(column_name)

- **Purpose:** Calculates the sum of values in a numeric column.
- **Example:**

```
SELECT SUM(salary) FROM employees;
```

AVG(column_name)

- **Purpose:** Calculates the average of values in a numeric column. Ignores null values by default.
- **Example:**

```
SELECT AVG(order_total) FROM orders;
```

MIN(column_name)

- **Purpose:** Returns the minimum value in a column.
- **Example:**

```
SELECT MIN(price) FROM products;
```

MAX(column_name)

- **Purpose:** Returns the maximum value in a column.
- **Example:**

```
SELECT MAX(score) FROM exams;
```

9) **Explain the command used for ordering the query results? Explain with the syntax and an example**

ORDER BY sorts the retrieved data based on columns in ascending (default) or descending order.

Syntax:

```
SELECT column1, column2, ... FROM table_name [WHERE condition] ORDER
BY column_name ASC|DESC [, column_name2 ASC|DESC, ...];
```

Explanation:

column_name: The name of the column you want to sort by.

ASC: Sorts the data in ascending order This is the default order if not specified.

DESC: Sorts the data in descending order

Example:

```
SELECT name, age FROM customers ORDER BY age DESC;
```

10) Write SQL queries for following:

Student(Enrno, name, courseId, emailId, cellno)
Course(courseId, course_nm, duration)

a) Add a column city in student table.

- `ALTER TABLE student ADD COLUMN city VARCHAR(50);`

b) Find out list of students who have enrolled in "computer" course.

-
  ```
  SELECT name FROM student s INNER JOIN course c ON s.courseId
  c.courseId WHERE c.course_nm = 'computer';
  ```

c) List name of all courses with their duration.

- `SELECT course_nm, duration FROM course;`

d) List name of all students start with „a".

- `SELECT name FROM student WHERE name LIKE 'a%';`

e) List email Id and cell no of all mechanical engineering students

-
  ```
  SELECT emailId, cellno FROM student s INNER JOIN course c ON
  s.courseId = c.courseId WHERE c.course_nm = 'mechanical';
  ```

11) Explain Group by and having clause in SQL with an example.

<u>GROUP BY Clause</u>

- **Purpose:** Used in conjunction with aggregate functions to group rows based on a specific column before performing the aggregation.
- **Example:**
  ```
  SELECT department, COUNT(*) AS employee_count
  FROM employees
  GROUP BY department;
  ```

<u>HAVING Clause (Optional)</u>

- **Purpose:** Used with aggregate functions after the GROUP BY clause to filter groups based on a condition on the aggregate values.
- **Example:**
  ```
  SELECT department, AVG(salary) AS average_salary
  FROM employees
  GROUP BY department
  HAVING AVG(salary) > 50000;
  ```

12) **Explain Views in SQL. Give the syntax to create and drop views.**

In SQL, views act as virtual tables that provide a customized view of the underlying data stored in one or more base tables. They offer several advantages:

- Data Abstraction: Views can simplify complex queries by hiding the underlying table structure and join logic from the user. Users can interact with the view as if it were a regular table.
- Security: Views can restrict access to sensitive data by limiting the columns or rows visible through the view.
- Data Focus: Views can present a focused subset of data from multiple tables, making it easier for users to retrieve the information they need.

Syntax:

- `CREATE VIEW view_name AS SELECT column1, column2, ... FROM table_name1 [JOIN table_name2 ON join_condition ...] [WHERE condition];`
- `DROP VIEW view_name;`

13) **Consider the below table:**
Orders(ord_no , purch_amt, ord_date , customer_id, salesman_id)

a) **Write a SQL query to calculate total purchase amount of all orders.**
- `SELECT SUM(purch_amt) AS total_purchase_amount FROM Orders;`

b) **Write a SQL query to calculate the average purchase amount of all orders.**
- `SELECT AVG(purch_amt) AS average_purchase_amount FROM Orders;`

c) **Write a SQL query that counts the number of unique salespeople.**
- `SELECT COUNT(DISTINCT salesman_id) AS unique_salespeople FROM Orders;`

d) **Write a SQL query to find the maximum and minimum purchase amount.**
- `SELECT MAX(purch_amt) AS max_purchase, MIN(purch_amt) AS min_purchase FROM Orders;`

14) **Develop the SQL queries for the following:**

a) **Retrieve the birth date and address of employee whose employee id is 10.**
- `SELECT dob, address FROM employee WHERE eid = 10;`

b) **Retrieve the name and address of all employees who work for 'Research' department.**
- `SELECT ename, address FROM employee WHERE department = 'Research';`

c) **Retrieve all employees in department 5 whose salary is between 30000 and 40000.**
- `SELECT * FROM employee WHERE department = 5 AND salary BETWEEN 30000 AND 40000;`

d) **Retrieve distinct salaries of employees.**
- `SELECT DISTINCT salary FROM employee;`