STRUCTURED QUERY LANGUAGE (SQL)

Assume the Relations given below. Student(Enrno, name, courseId, emailId, cellno) Course(courseId, course_nm, duration) Write SQL statements for following:

1. **Find out list of students who have enrolled in “computer” course.**
2. **List name of all courses with their duration.**
3. **List name of all students start with “a”.**
4. **List email Id and cell no of all mechanical engineering students.**

Certainly! Here are the SQL statements for each of the given tasks:

1. Find out list of students who have enrolled in the "computer" course.

```sql
SELECT s.name
FROM Student s
JOIN Course c ON s.courseId = c.courseId
WHERE c.course_nm = 'computer';
```

2. List name of all courses with their duration.

```sql
SELECT course_nm, duration
FROM Course;
```

3. List name of all students start with "a".

```sql
SELECT name
FROM Student
WHERE name LIKE 'a%';
```

4. List email Id and cell no of all mechanical engineering students.

```sql
SELECT s.emailId, s.cellno
FROM Student s
JOIN Course c ON s.courseId = c.courseId
WHERE c.course_nm = 'mechanical engineering';
```

Write the basic structure of SQL Queries. Explain working of each keyword in the structure.

The basic structure of SQL queries consists of three main clauses: SELECT, FROM, and WHERE. Each keyword in this structure plays a crucial role in defining and executing SQL queries effectively.

1. **SELECT Clause**:
    - The SELECT clause specifies the attributes desired in the query result, corresponding to the projection operation in relational algebra.
    - It allows you to list the specific columns you want to retrieve from the database.
    - For example, SELECT name, salary FROM instructor; retrieves the names and salaries of all instructors.
2. **FROM Clause**:
    - The FROM clause lists the relations involved in the query, specifying the tables from which data will be retrieved.
    - It is used to specify the source tables for the query.
    - For example, SELECT * FROM instructor; retrieves all data from the instructor table.
3. **WHERE Clause**:
    - The WHERE clause specifies conditions that the result must satisfy, allowing you to filter data based on specific criteria.
    - It includes comparison operators like <=, >, >=, =, and <> to filter data.
    - For example, SELECT name FROM student WHERE dept_name = 'cse'; retrieves the names of all students in the 'cse' department.

How are the following integrity constraints implemented in SQL:

a. Domain constraint

b. Referential integrity.

Explain the above with appropriate syntax and example.

a. **Domain Constraint:**

Domain constraints ensure that values entered into a column adhere to specified rules or conditions.

- **Types:**

  Not Null Constraint:

  - Ensures a column cannot contain null values.
  - Syntax:

```
CREATE TABLE table_name (
    column_name data_type NOT NULL,
    ...
);
```

  - Example:

```
CREATE TABLE employee (
    employee_id varchar(30),
    employee_name varchar(30) NOT NULL,
    salary NUMBER
);
```

In the above example, the employee_name column must always have a non-null value.

  Check Constraint:

  - Enforces a condition on values entered into a column.
  - Syntax:

```
CREATE TABLE table_name (
    column_name data_type CONSTRAINT constraint_name CHECK (condition),
    ...
);
```

- Example:

```
CREATE TABLE employee (
    employee_id varchar(30) not null check(employee_id > 0),
    employee_name varchar(30),
    salary NUMBER
);
```

The above example ensures that the employee_id column only includes integers greater than 0.

b. **Referential Integrity:**

Referential integrity constraints ensure that values in one table's foreign key column(s) correspond to values in another table's primary key column(s).

- Implemented using the FOREIGN KEY clause.
- Syntax:

```
CREATE TABLE table_name (
    ...
    foreign_key_column data_type,
    FOREIGN KEY (foreign_key_column) REFERENCES referenced_table(primary_key_column)
);
```

- Example:

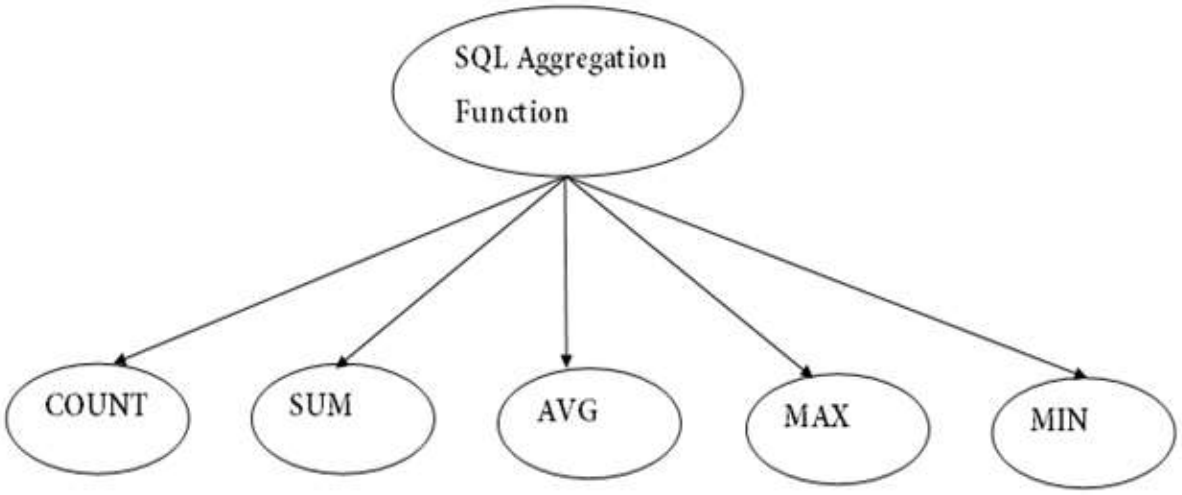
```
CREATE TABLE department (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50) NOT NULL
);

CREATE TABLE employee (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50) NOT NULL,
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES department(dept_id)
);
```

In the above example, the FOREIGN KEY constraint ensures that the dept_id column in the employee table references the dept_id primary key column in the department table, maintaining referential integrity.

List and explain aggregate functions of SQL with appropriate examples.

- SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
- It is also used to summarize the data.

Types of SQL Aggregation Function



1. COUNT FUNCTION

- COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
- COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.

Syntax

COUNT(*) or COUNT([ALL|DISTINCT] expression)

Sample table:

PRODUCT_MAST

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75

Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40
Item6	Cpm1	3	25	75
Item7	Com1	5	30	150
Item8	Com1	3	10	30
Item9	Com2	2	25	50
Item10	Com3	4	30	120

Example: COUNT()

```
1. SELECT COUNT(*)
2. FROM PRODUCT_MAST;
```

Output:

10

Example: COUNT with WHERE

```
1. SELECT COUNT(*)
2. FROM PRODUCT_MAST;
3. WHERE RATE>=20;
```

Output:

7

Example: COUNT() with DISTINCT

```
1. SELECT COUNT(DISTINCT COMPANY)
2. FROM PRODUCT_MAST;
```

Output:

3

Example: COUNT() with GROUP BY

```
1. SELECT COMPANY, COUNT(*)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY;
```

Output:

Com1 5

Com2 3

Com3 2

Example: COUNT() with HAVING

```
1. SELECT COMPANY, COUNT(*)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY
4. HAVING COUNT(*)>2;
```

Output:

Com1 5

Com2 3

2. SUM Function

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

Syntax

```
1. SUM()
2. or
3. SUM( [ALL|DISTINCT] expression )
```

Example: SUM()

```
1. SELECT SUM(COST)
2. FROM PRODUCT_MAST;
```

Output:

670

Example: SUM() with WHERE

```
1. SELECT SUM(COST)
2. FROM PRODUCT_MAST
3. WHERE QTY>3;
```

Output:

320

Example: SUM() with GROUP BY

```
1. SELECT SUM(COST)
2. FROM PRODUCT_MAST
3. WHERE QTY>3
4. GROUP BY COMPANY;
```

Output:

Com1 150

Com2 170

Example: SUM() with HAVING

```
1. SELECT COMPANY, SUM(COST)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY
4. HAVING SUM(COST)>=170;
```

Output:

Com1 335

Com3 170

3. AVG function

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

Syntax

1. AVG()
2. or
3. AVG([ALL|DISTINCT] expression)

Example:

1. SELECT AVG(COST)
2. FROM PRODUCT_MAST;

Output:

67.00

4. MAX Function

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

Syntax

1. MAX()
2. or
3. MAX([ALL|DISTINCT] expression)

Example:

1. SELECT MAX(RATE)
2. FROM PRODUCT_MAST;

30

5. MIN Function

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

Syntax

1. MIN()
2. or

3. MIN([ALL|DISTINCT] expression)

Example:

```
1. SELECT MIN(RATE)
2. FROM PRODUCT_MAST;
```

Output:

10

List and explain the different DML statements in SQL

In SQL, Data Manipulation Language (DML) statements are used to interact with the data stored in the database. Here are the different DML statements in SQL along with their explanations:

1. **SELECT**:
    - The SELECT statement is used to retrieve data from one or more tables in the database.
    - It allows you to specify the columns you want to retrieve and apply conditions to filter the data.
    - Example:

```
SELECT column1, column2 FROM table WHERE condition;
```

2. **INSERT**:
    - The INSERT statement is used to add new rows of data into a table.
    - It specifies the table name and the values to be inserted into the specified columns.
    - Example:

```
INSERT INTO table_name (column1, column2) VALUES (value1, value2);
```

3. **UPDATE**:
    - The UPDATE statement is used to modify existing data in a table.
    - It allows you to change the values of specific columns based on specified conditions.
    - Example:

```
UPDATE table SET column1 = value1 WHERE condition;
```

4. **DELETE**:
    - The DELETE statement is used to remove rows from a table based on specified conditions.

- It permanently deletes data from the table.
- Example:

```
DELETE FROM table WHERE condition;
```

5. **MERGE**:
    - The MERGE statement allows you to perform insert, update, or delete operations in a single statement.
    - It is useful for synchronizing two tables based on a specified condition.
    - Example:

```
MERGE INTO target_table USING source_table ON condition
WHEN MATCHED THEN UPDATE SET column1 = value1
WHEN NOT MATCHED THEN INSERT (column1, column2) VALUES (value1, value2);
```

These DML statements are essential for manipulating data in SQL databases, enabling users to retrieve, insert, update, and delete data efficiently based on their requirements.

Explain the following SQL constructs with examples:

(1) order by, (2) group by, (3) having, (4) as, (5) in

1. **ORDER BY**:
    - The ORDER BY clause is used to sort the result set of a SQL query in ascending or descending order.
    - Example:

```
SELECT name, salary FROM instructor ORDER BY salary DESC;
```

This query will retrieve the names and salaries of all instructors, sorted in descending order by salary.

2. **GROUP BY**:
    - The GROUP BY clause is used to group rows that have the same values into summary rows, like when you want to calculate the total sales per salesperson or the average order amount for each customer.
    - Example:

```
SELECT dept_name, AVG(salary) AS avg_salary
FROM instructor
GROUP BY dept_name;
```

This query will group the instructors by their department and calculate the average salary for each department.

3. **HAVING**:
    - The HAVING clause is used in conjunction with the GROUP BY clause to filter group rows that do not satisfy the specified condition.
    - Example:

```
SELECT dept_name, AVG(salary) AS avg_salary
FROM instructor
GROUP BY dept_name
HAVING AVG(salary) > 50000;
```

This query will group the instructors by their department, calculate the average salary for each department, and then filter the results to only include departments with an average salary greater than $50,000.

4. **AS**:
    - The AS clause is used to assign an alias to a table or column in a SQL query.
    - Example:

```
SELECT i.name, i.salary / 12 AS monthly_salary
FROM instructor i;
```

In this example, the column salary / 12 is aliased as monthly_salary.

5. **IN**:
    - The IN operator is used to check if a value matches any value in a list of values.
    - Example:

```
SELECT name, dept_name
FROM instructor
WHERE dept_name IN ('Physics', 'Computer Science', 'Electrical Engineering');
```

This query will retrieve the names and department names of all instructors who work in the Physics, Computer Science, or Electrical Engineering departments.

These SQL constructs are essential for manipulating and filtering data in a database, allowing you to sort, group, filter, and alias the results of your queries.

Consider the following Database design :

Customer (cid, custname, coststreet, custcity)
Account (accno, branchname, balance)
Loan (loanno, branchname, amount)
Borrower (cid, loanno)
Branch (branchname, branchcity, asset)
Depositor (cid, accno)
Solve the following queries in SQL

1. **Display the name of customers who have both account and loan at the bank.**
2. **Update amount of loan to 10000 where loan number is “L-101”.**
3. **Change the column name custcity to ccity.**
4. **Find all customers who an account but no loan at bank.**

Here are the SQL queries to solve the given tasks:

a. Display the name of customers who have both account and loan at the bank.

```sql
SELECT c.custname
FROM Customer c
INNER JOIN Borrower b ON c.cid = b.cid
INNER JOIN Account a ON c.cid = a.cid;
```

b. Update the amount of the loan to 10000 where the loan number is "L-101".

```sql
UPDATE Loan
SET amount = 10000
WHERE loanno = 'L-101';
```

c. Change the column name custcity to ccity.

```sql
ALTER TABLE Customer
RENAME COLUMN custcity TO ccity;
```

d. Find all customers who have an account but no loan at the bank.

```sql
SELECT c.custname
FROM Customer c
LEFT JOIN Borrower b ON c.cid = b.cid
LEFT JOIN Loan l ON b.loanno = l.loanno
WHERE b.cid IS NULL;
```

The following relations keep track of Library Management system.

Book_info(bookid, bname, bauthor, price, edition, publication, pur_date,)
Student(lib_car_num, stud_name, class, branch, roll_no)
Issue_table(issue_date, sub_date, bookid, lib_car_num, due)

Write the following SQL queries:

1. **Find the details of the books issued to the library card number 1.**
2. **Give all the information about student and the book issued with ascending order of library card number**
3. **Find the author, edition, price of book.**
4. **Find the names of the students with dues on the book issue.**

Here are the SQL queries to solve the given tasks:

1. Find the details of the books issued to the library card number 1.

```sql
SELECT bi.*
FROM Book_info bi
JOIN Issue_table it ON bi.bookid = it.bookid
WHERE it.lib_car_num = 1;
```

2. Give all the information about students and the books issued with ascending order of library card number.

```sql
SELECT s.*, bi.*
FROM Student s
JOIN Issue_table it ON s.lib_car_num = it.lib_car_num
JOIN Book_info bi ON it.bookid = bi.bookid
ORDER BY s.lib_car_num ASC;
```

3. Find the author, edition, and price of the book.

```sql
SELECT bauthor, edition, price
FROM Book_info;
```

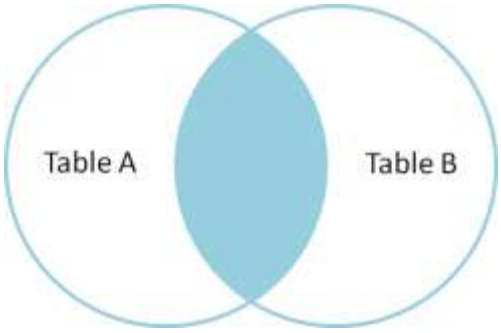
4. Find the names of the students with dues on the book issue.

```sql
SELECT s.stud_name
FROM Student s
JOIN Issue_table it ON s.lib_car_num = it.lib_car_num
WHERE it.due > 0;
```

List and explain the types of Join in SQL.

In SQL, there are different types of joins that allow you to combine rows from two or more tables based on a related column between them. The common types of joins in SQL are:

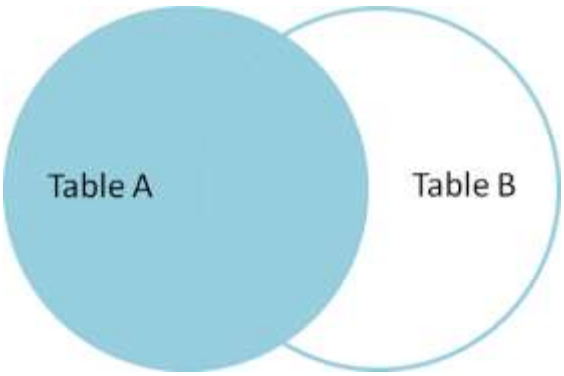
1. **Inner Join**:
    - An inner join returns rows when there is at least one match between the columns in the two tables.
    - It only includes rows that have matching values in both tables.



    - Example:

```sql
SELECT column1, column2
FROM table1
INNER JOIN table2 ON table1.column = table2.column;
```

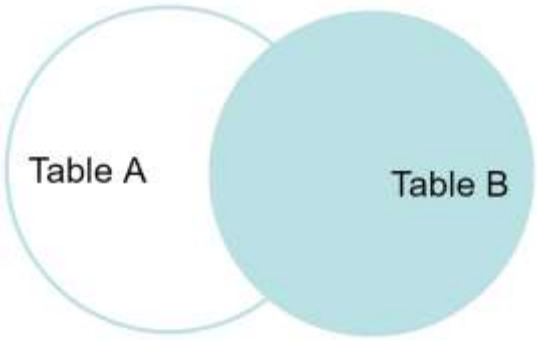
2. **Left Outer Join**:
    - A left outer join returns all rows from the left table and the matched rows from the right table.
    - If there is no match, NULL values are returned for the columns from the right table.



- Example:

```sql
SELECT column1, column2
FROM table1
LEFT OUTER JOIN table2 ON table1.column = table2.column;
```

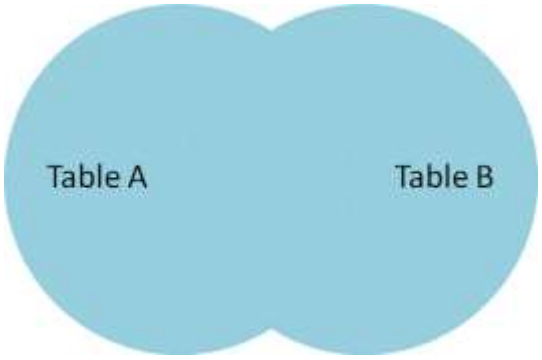
3. **Right Outer Join**:
    - A right outer join returns all rows from the right table and the matched rows from the left table.
    - If there is no match, NULL values are returned for the columns from the left table.



- Example:

```sql
SELECT column1, column2
FROM table1
RIGHT OUTER JOIN table2 ON table1.column = table2.column;
```

4. **Full Outer Join**:
    - A full outer join returns all rows when there is a match in either the left or right table.
    - It combines the results of both left and right outer joins.



- Example:

```sql
SELECT column1, column2
FROM table1
FULL OUTER JOIN table2 ON table1.column = table2.column;
```

These join types in SQL allow you to combine data from multiple tables based on specified conditions, providing flexibility in querying and retrieving data from relational databases.

Why is the Domain Constraint called as elementary Database constraint?

The term "elementary database constraint" typically refers to constraints that are applied at the level of individual attributes or fields within a database. Domain constraints are indeed considered as elementary database constraints because they are applied to the domain of values that a particular attribute can take.

Here's why domain constraints are called elementary:

1. **Scope:**
    - Domain constraints apply at the attribute level, defining the allowable values for a specific attribute or field in a table. They restrict the type and range of values that can be stored in that attribute.
2. **Granularity:**
    - These constraints operate at a granular level, focusing on individual attributes rather than the relationships between tables or rows.
3. **Simplicity:**
    - Domain constraints are relatively simple compared to other types of constraints like referential integrity or check constraints. They define basic rules such as data type, length, and format restrictions for attribute values.
4. **Foundational:**
    - Domain constraints establish the foundation for data integrity by ensuring that only valid and meaningful values are stored in the database.

The Domain Constraint is referred to as an elementary database constraint because it defines the allowable values for an attribute within a database. This constraint specifies the domain of values associated with each attribute, ensuring that only valid and permissible values can be stored in the database. By defining the domain of values for each attribute, the Domain Constraint plays a fundamental role in maintaining data integrity and consistency within

the database. It restricts the range of values that can be entered into a specific attribute, enforcing data accuracy and reliability.