Unit 4: SQL Concepts – Detailed Answers

1. SQL Key Constraints

Primary Key

- Uniquely identifies each record in a table
- Cannot contain NULL values
- Only one primary key per table

Foreign Key

- Establishes relationship between two tables
- References primary key of another table
- Ensures referential integrity

Unique Key

- Ensures all values in column are distinct
- Allows NULL values (unlike Primary Key)
- Multiple unique keys per table

Not Null

- Ensures column cannot have NULL values
- Applied at column level

Commit

- Saves all transactions to database permanently
- Ends current transaction

Candidate Key

- Column(s) that could be chosen as primary key
- All candidate keys are unique and not null

Rollback

- Undoes all transactions since last commit
- Restores database to previous state

Example:

```
CREATE TABLE Students (
    stud_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(50) UNIQUE,
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Department(dept_id)
);
```

Practice Question: Create a table 'Employees' with emp_id as primary key, emp_name not null, and dept_id as foreign key referencing Departments table.

Solution:

```sql
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50) NOT NULL,
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Departments(dept_id)
);
```

2. Database Concepts

Weak Entity

- Depends on another entity for existence
- Has partial key
- Shown with double rectangle in ER diagram

Data Dictionary

- Stores metadata about database
- Contains information about tables, columns, constraints

Substring()

- Extracts part of a string
- Syntax: SUBSTRING(string, start, length)

Alter

- Modifies table structure
- Can add, modify, drop columns

Truncate

- Removes all records from table
- Cannot be rolled back
- Faster than DELETE

Drop

- Removes entire table from database
- Cannot be recovered

Example:

```
-- Alter example
ALTER TABLE Students ADD COLUMN phone VARCHAR(15);

-- Truncate example
TRUNCATE TABLE Temp_Data;

-- Drop example
DROP TABLE Backup_Data;
```

3. ON DELETE CASCADE

Definition

- Referential integrity action
- Automatically deletes child records when parent record is deleted

Syntax:

```
FOREIGN KEY (column) REFERENCES parent_table(parent_colun
ON DELETE CASCADE
```

Example:

```
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES Customers(custome
    ON DELETE CASCADE
);
```

When a customer is deleted, all their orders are automatically deleted.

Practice Question: Create two tables: Departments and Employees with ON DELETE CASCADE constraint.

Solution:

```sql
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50)
);

CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Departments(dept_id)
    ON DELETE CASCADE
);
```

4. SQL Joins

Definition

- Combines rows from two or more tables based on related column

Types of Joins:

1. **INNER JOIN**: Returns matching records from both tables
2. **LEFT JOIN**: All records from left table + matching from right
3. **RIGHT JOIN**: All records from right table + matching from left
4. **FULL JOIN**: All records when there's match in either table
5. **CROSS JOIN**: Cartesian product of both tables

Examples:

```sql
-- INNER JOIN
SELECT e.emp_name, d.dept_name
FROM Employees e
INNER JOIN Departments d ON e.dept_id = d.dept_id;

-- LEFT JOIN
SELECT e.emp_name, d.dept_name
FROM Employees e
LEFT JOIN Departments d ON e.dept_id = d.dept_id;
```

5. Natural Join

Definition

- Automatically joins tables based on columns with same name
- Eliminates duplicate columns

Example:

```sql
SELECT *
FROM Employees
NATURAL JOIN Departments;
```

This joins tables where dept_id exists in both tables.

Important Points:

- Column names must be identical in both tables
- Data types must be compatible
- Removes duplicate join columns

6. SQL Language Types

DDL (Data Definition Language)

- Defines database structure
- Commands: CREATE, ALTER, DROP, TRUNCATE

Example:

```
CREATE TABLE Students (
    id INT PRIMARY KEY,
    name VARCHAR(50)
);
```

DML (Data Manipulation Language)

- Manipulates data within tables
- Commands: SELECT, INSERT, UPDATE, DELETE

Example:

```
INSERT INTO Students VALUES (1, 'John');
UPDATE Students SET name = 'Mike' WHERE id = 1;
```

DCL (Data Control Language)

- Controls access to database

- Commands: GRANT, REVOKE

Example:

```
GRANT SELECT ON Students TO user1;
REVOKE DELETE ON Students FROM user2;
```

7. String Functions

Common String Functions:

1. **LENGTH()**: Returns string length
2. **UPPER()**: Converts to uppercase
3. **LOWER()**: Converts to lowercase
4. **TRIM()**: Removes leading/trailing spaces
5. **CONCAT()**: Combines strings
6. **SUBSTRING()**: Extracts part of string

Examples:

```
SELECT
    LENGTH('Hello') as str_length,
    UPPER('hello') as upper_case,
    LOWER('HELLO') as lower_case,
    TRIM('  hello  ') as trimmed,
    CONCAT('Hello', ' World') as concatenated,
    SUBSTRING('Hello World', 1, 5) as substring;
```

8. Two String Functions (Detailed)

SUBSTRING()

Syntax: `SUBSTRING(string, start, length)` **Example:**

```
SELECT SUBSTRING('Database', 1, 4) as result;
-- Output: 'Data'
```

CONCAT()

Syntax: `CONCAT(string1, string2, ...)` **Example:**

```
SELECT CONCAT('Hello', ' ', 'World') as greeting;
-- Output: 'Hello World'
```

Practice Question: Write query to display employee names in format "FirstName_LastName" using string functions.

Solution:

```
SELECT CONCAT(first_name, '_', last_name) as full_name
FROM Employees;
```

9. Aggregate Functions

COUNT()

- Counts number of rows **Example:**

```
SELECT COUNT(*) as total_students FROM Students;
```

AVG()

- Calculates average value **Example:**

```
SELECT AVG(marks) as average_marks FROM Results;
```

Other Aggregate Functions:

- SUM(): Calculates sum
- MAX(): Finds maximum value
- MIN(): Finds minimum value

10. Views in SQL

Definition

- Virtual table based on result of SQL query
- Doesn't store data physically

Advantages:

1. **Security**: Restrict access to specific columns
2. **Simplicity**: Hide complex queries
3. **Consistency**: Provide consistent interface
4. **Logical Data Independence**: Can change underlying tables without affecting applications

Example:

```
CREATE VIEW HighScorers AS
SELECT name, marks
FROM Students
WHERE marks > 80;

SELECT * FROM HighScorers;
```

11. NULL in SQL

Definition

- Represents missing or unknown data
- Not same as zero or empty string

Important Properties:

1. **Comparison**: NULL = NULL returns NULL (not TRUE)
2. **Arithmetic**: Any operation with NULL returns NULL
3. **Logical**: NULL in boolean expressions is treated as UNKNOWN

Handling NULL:

```sql
-- Check for NULL
SELECT * FROM Students WHERE phone IS NULL;

-- Check for NOT NULL
SELECT * FROM Students WHERE phone IS NOT NULL;

-- Handle NULL with COALESCE
SELECT name, COALESCE(phone, 'No Phone') as contact
FROM Students;
```

Practice Question: Write query to find employees without email addresses.

Solution:

```sql
SELECT emp_name
FROM Employees
WHERE email IS NULL;
```

Exam Tips:

1. Always mention syntax with examples
2. Use proper SQL case conventions (keywords in uppercase)
3. Draw ER diagrams for relationship questions
4. Explain real-world applications of concepts
5. Practice writing error-free SQL queries
6. Understand difference between TRUNCATE, DELETE and DROP
7. Remember NULL handling in conditions (use IS NULL, not = NULL)

Common Mistakes to Avoid:

- Using = NULL instead of IS NULL
- Forgetting semicolons in SQL statements
- Confusing different join types
- Not specifying all required columns in INSERT
- Missing FOREIGN KEY constraints in relationship questions

Question 12: SQL Queries on Supplier-Parts Database

Database Schema Analysis

Tables Structure:

- **Supplier (S#, sname, status, city)**
  - S#: Supplier ID (Primary Key)
  - sname: Supplier Name
  - status: Supplier Status
  - city: Supplier City
- **Parts (P#, pname, color, weight, city)**
  - P#: Part ID (Primary Key)
  - pname: Part Name
  - color: Part Color
  - weight: Part Weight
  - city: Part City
- **SP (S#, P#, quantity)**
  - S#: Supplier ID (Foreign Key)

- P#: Part ID (Foreign Key)
- quantity: Supply Quantity
- Composite Primary Key (S#, P#)

Detailed SQL Queries with Explanations

A. Find name of supplier for city = 'Delhi'

```sql
SELECT sname
FROM Supplier
WHERE city = 'Delhi';
```

Explanation: Simple SELECT with WHERE clause to filter suppliers from Delhi.

B. Find suppliers whose name start with 'AB'

```sql
SELECT sname
FROM Supplier
WHERE sname LIKE 'AB%';
```

Explanation: Uses LIKE operator with wildcard '%' to find names starting with 'AB'.

C. Find all suppliers whose status is 10, 20 or 30

```sql
SELECT sname
FROM Supplier
WHERE status IN (10, 20, 30);
```

Alternative using OR:

```sql
SELECT sname
FROM Supplier
WHERE status = 10 OR status = 20 OR status = 30;
```

Explanation: IN operator simplifies multiple OR conditions.

D. Find total number of city of all suppliers

```sql
SELECT COUNT(DISTINCT city) as total_cities
FROM Supplier;
```

Explanation: COUNT with DISTINCT to count unique cities, avoiding duplicates.

E. Find S# of supplier who supplies 'red' part

```sql
SELECT DISTINCT SP.S#
FROM SP
JOIN Parts ON SP.P# = Parts.P#
WHERE Parts.color = 'red';
```

Explanation: JOIN between SP and Parts tables to find suppliers of red parts.

F. Count number of supplier who supplies 'red' part

```sql
SELECT COUNT(DISTINCT SP.S#) as red_part_suppliers
FROM SP
JOIN Parts ON SP.P# = Parts.P#
WHERE Parts.color = 'red';
```

Explanation: Counts distinct suppliers to avoid duplicate counting.

G. Sort the supplier table by sname

```sql
SELECT *
FROM Supplier
ORDER BY sname;
```

Explanation: ORDER BY clause for ascending sort (default).

H. Delete records in supplier table whose status is 40

```
DELETE FROM Supplier
WHERE status = 40;
```

Important Note: In exam, mention that DELETE should be used cautiously as it removes data permanently.

I. Add one field in supplier table

```
ALTER TABLE Supplier
ADD phone VARCHAR(15);
```

Explanation: ALTER TABLE with ADD to include new column.

J. Find name of parts whose color is 'red'

```
SELECT pname
FROM Parts
WHERE color = 'red';
```

Explanation: Simple selection from Parts table.

K. Find parts whose weight is less than 10 kg

```sql
SELECT pname
FROM Parts
WHERE weight < 10;
```

Explanation: Numerical comparison in WHERE clause.

L. Find all parts whose weight is from 10 to 20 kg

```sql
SELECT pname
FROM Parts
WHERE weight BETWEEN 10 AND 20;
```

Alternative:

```sql
SELECT pname
FROM Parts
WHERE weight >= 10 AND weight <= 20;
```

Explanation: BETWEEN operator for range queries.

M. Find average weight of all parts

```sql
SELECT AVG(weight) as average_weight
FROM Parts;
```

Explanation: AVG aggregate function for average calculation.

N. Find S# of supplier who supply part 'p2'

```sql
SELECT S#
FROM SP
WHERE P# = 'p2';
```

Explanation: Direct query on SP table for part 'p2'.

O. Find the name of the supplier who supplies maximum parts

```sql
SELECT s.sname
FROM Supplier s
JOIN (
    SELECT S#, COUNT(P#) as part_count
    FROM SP
    GROUP BY S#
    ORDER BY part_count DESC
    LIMIT 1
) max_supplier ON s.S# = max_supplier.S#;
```

Explanation: Subquery to find supplier with maximum part count, then

join to get name.

P. Sort the parts table by pname

```
SELECT *
FROM Parts
ORDER BY pname;
```

Explanation: Alphabetical sorting of parts.

Q. Delete records in the parts table whose color is 'blue'

```
DELETE FROM Parts
WHERE color = 'blue';
```

Exam Tip: Mention referential integrity - ensure no foreign key constraints violated.

R. Drop one field in the parts table

```
ALTER TABLE Parts
DROP COLUMN city;
```

Explanation: ALTER TABLE with DROP COLUMN to remove field.

Practice Question

Scenario: A university database has tables:

- Students (student_id, name, department, city)
- Courses (course_id, course_name, credits)
- Enrollments (student_id, course_id, grade)

Write queries for:

1. Find students from 'Computer Science' department
2. Count number of courses with more than 3 credits
3. Find students enrolled in 'Database Management' course
4. Add a new column 'email' to Students table

Practice Solutions

```sql
-- 1. Find students from 'Computer Science' department
SELECT name
FROM Students
WHERE department = 'Computer Science';

-- 2. Count number of courses with more than 3 credits
SELECT COUNT(*)
FROM Courses
WHERE credits > 3;

-- 3. Find students enrolled in 'Database Management' course
SELECT s.name
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
JOIN Courses c ON e.course_id = c.course_id
WHERE c.course_name = 'Database Management';

-- 4. Add a new column 'email' to Students table
ALTER TABLE Students
ADD email VARCHAR(100);
```

Exam Tips for SQL Questions

Common Mistakes to Avoid:

1. **Forgetting JOIN conditions** - leads to Cartesian products
2. **Missing DISTINCT** when counting unique entities
3. **Incorrect use of single vs double quotes** - Use single quotes for string literals
4. **Case sensitivity** in string comparisons
5. **Forgetting GROUP BY** with aggregate functions

Writing Style in Exams:

- Write clean, formatted SQL code
- Use meaningful aliases for tables and columns
- Comment complex queries to explain logic
- Show both main method and alternatives if possible
- Mention assumptions about data types and constraints

Important Concepts to Highlight:

- **JOIN types** and when to use each
- **Aggregate functions** with GROUP BY
- **Subqueries vs JOINs** performance considerations
- **Data modification** operations and their implications
- **Constraint violations** and error handling

Expected Marks Distribution:

- Basic SELECT queries: 1-2 marks each
- JOIN operations: 3-4 marks each
- Complex queries with subqueries/aggregates: 4-5 marks each
- DDL operations: 2-3 marks each
- Overall understanding and explanation: 3-4 marks

Pro Tip: Always test your queries mentally with sample data to ensure they work correctly!

Here's a detailed exam-focused answer for Questions 13 and 14 from Unit 4:

Questions 13 & 14: Advanced SQL Queries

Question 13: Student Exam Database

Database Schema Analysis

Tables Structure:

- **Student (rollno, name, branch)**
    - rollno: Student Roll Number (Primary Key)
    - name: Student Name
    - branch: Student Branch
- **Exam (rollno, subject_code, obtained_marks, paper_code)**
    - rollno: Student Roll Number (Foreign Key)
    - subject_code: Subject Code
    - obtained_marks: Marks Obtained
    - paper_code: Paper Code (Foreign Key)
- **Papers (paper_code, paper_setter_name, university)**
    - paper_code: Paper Code (Primary Key)
    - paper_setter_name: Name of Paper Setter
    - university: University Name

Detailed SQL Queries with Explanations

A. Display name of student who got first class in subject '130703'

```
SELECT s.name
FROM Student s
JOIN Exam e ON s.rollno = e.rollno
WHERE e.subject_code = '130703'
AND e.obtained_marks >= 60;
```

Explanation: Assuming first class means 60+ marks. JOIN between Student and Exam tables with marks condition.

A(ii). Display name of all students with their total marks

```
SELECT s.name, SUM(e.obtained_marks) as total_marks
FROM Student s
JOIN Exam e ON s.rollno = e.rollno
GROUP BY s.rollno, s.name;
```

Explanation: Uses SUM aggregate with GROUP BY to calculate total marks per student.

B. Display list number of student in each university

```sql
SELECT p.university, COUNT(DISTINCT s.rollno) as student_count
FROM Student s
JOIN Exam e ON s.rollno = e.rollno
JOIN Papers p ON e.paper_code = p.paper_code
GROUP BY p.university;
```

Explanation: Triple JOIN to connect Student-Exam-Papers and count students per university.

C. Display list of student who has not given any exam

```sql
SELECT s.name
FROM Student s
LEFT JOIN Exam e ON s.rollno = e.rollno
WHERE e.rollno IS NULL;
```

Explanation: LEFT JOIN with NULL check to find students without exam records.

Question 14: Car Insurance Database

Database Schema Analysis

Tables Structure:

- **Person (ss#, name, address)**
    - ss#: Social Security Number (Primary Key)
    - name: Person Name

    - address: Person Address

- **Car (license, year, model)**
    - license: License Number (Primary Key)
    - year: Manufacturing Year
    - model: Car Model

- **Accident (date, driver, damage_amount)**
    - date: Accident Date
    - driver: Driver involved (references Person)
    - damage_amount: Damage Cost

- **Owns (ss#, license)**
    - ss#: Social Security Number (Foreign Key)
    - license: License Number (Foreign Key)
    - Composite Primary Key (ss#, license)

- **Log (license, date, driver)**
    - license: License Number (Foreign Key)
    - date: Log Date
    - driver: Driver Name

Detailed SQL Queries with Explanations

A. Find the total number of people whose cars were involved in accidents in 2009

```sql
SELECT COUNT(DISTINCT o.ss#) as total_people
FROM Owns o
JOIN Accident a ON o.license = a.driver
WHERE EXTRACT(YEAR FROM a.date) = 2009;
```

Alternative using LIKE:

```sql
SELECT COUNT(DISTINCT o.ss#) as total_people
FROM Owns o
JOIN Accident a ON o.license = a.driver
WHERE a.date LIKE '2009%';
```

Explanation: JOIN between Owns and Accident, filtering by year 2009.

B. Find the number of accidents in which the cars belonging to "S.Sudarshan"

```sql
SELECT COUNT(*) as accident_count
FROM Accident a
JOIN Owns o ON a.driver = o.license
JOIN Person p ON o.ss# = p.ss#
WHERE p.name = 'S.Sudarshan';
```

Explanation: Triple JOIN to connect Accident-Owns-Person and count accidents for specific person.

C. Add a new customer to the database

```
INSERT INTO Person (ss#, name, address)
VALUES ('123-45-6789', 'John Doe', '123 Main St, City');
```

Explanation: Basic INSERT statement to add new person record.

D. Add a new accident recorded for the Santro belonging to "KORTH"

```
INSERT INTO Accident (date, driver, damage_amount)
SELECT CURRENT_DATE, c.license, 5000
FROM Car c
JOIN Owns o ON c.license = o.license
JOIN Person p ON o.ss# = p.ss#
WHERE c.model = 'Santro' AND p.name = 'KORTH';
```

Explanation: Uses subquery with JOIN to find the correct license for KORTH's Santro.

Advanced Query Techniques Demonstrated

Complex JOIN Operations

```sql
-- Multiple table JOIN with aggregation
SELECT p.university, AVG(e.obtained_marks) as avg_marks
FROM Papers p
JOIN Exam e ON p.paper_code = e.paper_code
GROUP BY p.university
HAVING AVG(e.obtained_marks) > 50;
```

Subquery Applications

```sql
-- Find students with above average marks
SELECT s.name, e.obtained_marks
FROM Student s
JOIN Exam e ON s.rollno = e.rollno
WHERE e.obtained_marks > (
    SELECT AVG(obtained_marks)
    FROM Exam
);
```

Date Operations

```sql
-- Find accidents in last 30 days
SELECT *
FROM Accident
WHERE date >= CURRENT_DATE - INTERVAL 30 DAY;
```

Practice Questions

Practice Question 13A:

Scenario: Find the university that has the highest average marks across all exams.

Solution:

```
SELECT p.university, AVG(e.obtained_marks) as avg_marks
FROM Papers p
JOIN Exam e ON p.paper_code = e.paper_code
GROUP BY p.university
ORDER BY avg_marks DESC
LIMIT 1;
```

Practice Question 14A:

Scenario: Find the person who owns the most expensive car (based on accident damage amounts).

Solution:

```
SELECT p.name, MAX(a.damage_amount) as max_damage
FROM Person p
JOIN Owns o ON p.ss# = o.ss#
JOIN Accident a ON o.license = a.driver
GROUP BY p.ss#, p.name
ORDER BY max_damage DESC
LIMIT 1;
```

Exam Writing Strategy

For Complex Queries:

1. **Break down the problem** - Identify required tables and relationships
2. **Start with FROM/JOIN** - Build the table connections first
3. **Add WHERE conditions** - Apply filters
4. **Include SELECT columns** - Choose what to display
5. **Add GROUP BY/HAVING** - For aggregate operations
6. **Finalize with ORDER BY** - For sorting

Sample Exam Answer Structure:

```
1. Problem Analysis:
   - Tables required: Student, Exam, Papers
   - Relationships: Student <img src="/assets/twemoji/2194.svg" wi
   - Conditions: university-based counting

2. SQL Query:
   [Write clean, formatted SQL]

3. Explanation:
   - JOIN logic used
   - Aggregate functions applied
   - Filter conditions
   - Expected output format
```

Common Pitfalls & Solutions:

Pitfall 1: Incorrect JOIN conditions

```sql
-- WRONG: Missing JOIN condition
SELECT * FROM Student, Exam, Papers;

-- CORRECT: Proper JOIN with conditions
SELECT *
FROM Student s
JOIN Exam e ON s.rollno = e.rollno
JOIN Papers p ON e.paper_code = p.paper_code;
```

Pitfall 2: GROUP BY errors

```sql
-- WRONG: Non-aggregated column not in GROUP BY
SELECT s.name, AVG(e.obtained_marks)
FROM Student s JOIN Exam e ON s.rollno = e.rollno;

-- CORRECT: All non-aggregated columns in GROUP BY
SELECT s.name, AVG(e.obtained_marks)
FROM Student s JOIN Exam e ON s.rollno = e.rollno
GROUP BY s.rollno, s.name;
```

Pitfall 3: Subquery in wrong context

```sql
-- WRONG: Multiple rows from subquery
SELECT name FROM Student WHERE rollno = (
    SELECT rollno FROM Exam WHERE obtained_marks > 80
);

-- CORRECT: Use IN for multiple values
SELECT name FROM Student WHERE rollno IN (
    SELECT rollno FROM Exam WHERE obtained_marks > 80
);
```

Marking Scheme Insights

Typical Distribution:

- **Correct SQL Syntax**: 30%
- **Proper JOIN Logic**: 25%
- **Aggregate Functions**: 20%
- **Query Efficiency**: 15%
- **Explanation Quality**: 10%

Pro Tips for Maximum Marks:

1. **Always use table aliases** for better readability
2. **Include comments** for complex logic
3. **Show multiple approaches** if time permits
4. **Mention assumptions** about data and relationships
5. **Test edge cases** mentally (NULL values, empty results)

Final Exam Advice:

"Practice writing SQL queries by hand regularly. Focus on understanding the logic behind JOIN operations and aggregate functions. In the exam,

read each question carefully to identify all required conditions and table relationships before writing your solution."

Questions 15-20: Advanced SQL Implementation

Question 15: Employee Database Queries

Database Schema

- **Employee (employee-name, street, city)**
- **Works (employee-name, company-name, salary)**
- **Company (company-name, city)**
- **Manages (employee-name, manager-name)**

SQL Queries:

A. Find name of all employees who work for State Bank

```
SELECT employee-name
FROM Works
WHERE company-name = 'State Bank';
```

B. Find names and cities of residence of all employees who work for State Bank

```sql
SELECT e.employee-name, e.city
FROM Employee e
JOIN Works w ON e.employee-name = w.employee-name
WHERE w.company-name = 'State Bank';
```

B(ii). Find all employees who do not work for State Bank

```sql
SELECT employee-name
FROM Works
WHERE company-name != 'State Bank';
```

C. Find employees who earn more than every employee of UCO Bank

```sql
SELECT employee-name
FROM Works
WHERE salary > ALL (
   SELECT salary
   FROM Works
   WHERE company-name = 'UCO Bank'
);
```

Alternative using MAX:

```sql
SELECT employee-name
FROM Works
WHERE salary > (
    SELECT MAX(salary)
    FROM Works
    WHERE company-name = 'UCO Bank'
);
```

Question 16: Student Marks Management

Database Schema

- **Student (Rollno, Name, Age, Sex, City)**
- **Student_marks (Rollno, Sub1, Sub2, Sub3, Total, Average)**

SQL Queries:

A. Calculate and store total and average marks

```sql
-- Update existing records
UPDATE Student_marks
SET Total = Sub1 + Sub2 + Sub3,
    Average = (Sub1 + Sub2 + Sub3) / 3;

-- For new inserts, use computed columns or triggers
```

B. Display students with >60 marks in Sub1

```sql
SELECT s.Name
FROM Student s
JOIN Student_marks m ON s.Rollno = m.Rollno
WHERE m.Sub1 > 60;
```

C. Display students with total and average marks

```sql
SELECT s.Name, m.Total, m.Average
FROM Student s
JOIN Student_marks m ON s.Rollno = m.Rollno;
```

D. Display students with equal marks in Sub2

```sql
SELECT s1.Name, s2.Name, s1.Sub2 as Common_Marks
FROM Student_marks s1
JOIN Student_marks s2 ON s1.Sub2 = s2.Sub2 AND s1.Rollno != s
JOIN Student st1 ON s1.Rollno = st1.Rollno
JOIN Student st2 ON s2.Rollno = st2.Rollno;
```

Question 17: Student Table Implementation

A. Create table and insert records

```sql
CREATE TABLE Student (
    stud_no INT PRIMARY KEY,
    stud_name VARCHAR(50),
    sub1 INT,
    sub2 INT,
    totalmark INT,
    percentage DECIMAL(5,2)
);

-- Insert 5 records
INSERT INTO Student VALUES
(101, 'Amit', 85, 90, 175, 87.5),
(102, 'Priya', 78, 82, 160, 80.0),
(103, 'Rahul', 92, 88, 180, 90.0),
(104, 'Neha', 65, 70, 135, 67.5),
(105, 'Karan', 88, 85, 173, 86.5);

-- Display data
SELECT * FROM Student;
```

B. Calculate totals and create view

```sql
-- Update calculations
UPDATE Student
SET totalmark = sub1 + sub2,
    percentage = (sub1 + sub2) / 2;

-- Create view with ascending order
CREATE VIEW Student_Results AS
SELECT * FROM Student
ORDER BY totalmark ASC;

-- Display view
SELECT * FROM Student_Results;
```

C. Update marks and recalculate

```sql
-- Update sub1 mark
UPDATE Student
SET sub1 = 50
WHERE stud_no = 111;

-- Recalculate totals
UPDATE Student
SET totalmark = sub1 + sub2,
    percentage = (sub1 + sub2) / 2
WHERE stud_no = 111;
```

Question 18: Employee Queries

Database Schema

Employee (emp_no, emp_name, department, city, salary)

SQL Queries:

A. Complex condition query

```
SELECT *
FROM Employee
WHERE emp_no < 100
  AND salary > 25000
  AND department = 'Account';
```

B. Count and sum operations

```
SELECT
    COUNT(*) as total_employees,
    SUM(salary) as total_salary
FROM Employee;
```

C. Delete employee with minimum salary

```
DELETE FROM Employee
WHERE salary = (SELECT MIN(salary) FROM Employee);
```

Safer approach (avoid multiple deletions):

```sql
DELETE FROM Employee
WHERE emp_no = (
   SELECT emp_no FROM Employee
   ORDER BY salary ASC
   LIMIT 1
);
```

Question 19: EMP-DEPT Table Queries

Database Schema

- **EMP(empno, ename, jobtitle, managerno, hiredate, sal, comm, deptno)**
- **DEPT(deptno, dname, loc)**

SQL Queries:

A. Salary > 3000 in department 20

```sql
SELECT ename, sal
FROM EMP
WHERE sal > 3000 AND deptno = 20;
```

B. Employees without commission

```
SELECT ename
FROM EMP
WHERE comm IS NULL OR comm = 0;
```

C. Count distinct job titles

```
SELECT COUNT(DISTINCT jobtitle) as job_count
FROM EMP;
```

D. Total salary per job category

```
SELECT jobtitle, SUM(sal) as total_salary
FROM EMP
GROUP BY jobtitle;
```

E. Employee count per department

```sql
SELECT d.dname, COUNT(e.empno) as employee_count
FROM DEPT d
LEFT JOIN EMP e ON d.deptno = e.deptno
GROUP BY d.deptno, d.dname;
```

F. Employees with no manager

```sql
SELECT ename
FROM EMP
WHERE managerno IS NULL;
```

G. Salary range query

```sql
SELECT ename, sal
FROM EMP
WHERE sal BETWEEN 1500 AND 3500;
```

Question 20: Advanced EMP-DEPT Queries

Part A:

A(i). Employees in departments 10,20,30

```
SELECT ename
FROM EMP
WHERE deptno IN (10, 20, 30);
```

A(ii). Names starting with 'A' (case-insensitive)

```
SELECT ename
FROM EMP
WHERE UPPER(ename) LIKE 'A%';
```

B. Employees with department names

```
SELECT e.ename, d.dname
FROM EMP e
JOIN DEPT d ON e.deptno = d.deptno;
```

C. Employees managed by KING

```sql
SELECT e.ename
FROM EMP e
JOIN EMP m ON e.managerno = m.empno
WHERE m.ename = 'KING';
```

D. Employees in Smith's department

```sql
SELECT e.ename
FROM EMP e
WHERE e.deptno = (
   SELECT deptno FROM EMP WHERE ename = 'SMITH'
);
```

E. Employees earning more than Allen

```sql
SELECT ename, sal
FROM EMP
WHERE sal > (SELECT sal FROM EMP WHERE ename = 'ALLEN');
```

F. Maximum salary earners per department

```sql
SELECT d.dname, e.ename, e.sal
FROM EMP e
JOIN DEPT d ON e.deptno = d.deptno
WHERE (e.deptno, e.sal) IN (
    SELECT deptno, MAX(sal)
    FROM EMP
    GROUP BY deptno
);
```

Part B: T1 and T2 Tables

C. Display rows with salary > 5000

```sql
SELECT * FROM T1 WHERE Salary > 5000;
```

D. Find deptno for ename='syham'

```sql
SELECT t2.Deptno
FROM T1 t1
JOIN T2 t2 ON t1.Empno = t2.Empno
WHERE t1.Ename = 'syham';
```

E. Add deptname column

```
ALTER TABLE T2 ADD deptname VARCHAR(30);
```

F. Update designation

```
UPDATE T1
SET Designation = 'senior clerk'
WHERE Ename = 'ram';
```

F(v). Total salary of all rows

```
SELECT SUM(Salary) as total_salary FROM T1;
```

G. Display joined data

```
SELECT t1.Empno, t1.Ename, t2.Deptno, t2.deptname
FROM T1 t1
LEFT JOIN T2 t2 ON t1.Empno = t2.Empno;
```

H. Drop table T1

```sql
DROP TABLE T1;
```

Exam Tips for Advanced SQL

Performance Optimization:

1. **Use EXISTS instead of IN** for large subqueries
2. **Create indexes** on frequently searched columns
3. **Avoid SELECT *** - specify only needed columns
4. **Use JOIN instead of subqueries** when possible

Common Mistakes:

```sql
-- WRONG: Using = with NULL
WHERE comm = NULL;

-- CORRECT: Using IS NULL
WHERE comm IS NULL;

-- WRONG: Missing GROUP BY columns
SELECT deptno, ename, AVG(sal) FROM EMP;

-- CORRECT: Include non-aggregated columns in GROUP BY
SELECT deptno, ename, AVG(sal)
FROM EMP
GROUP BY deptno, ename;
```

Transaction Control:

```
BEGIN TRANSACTION;
UPDATE Accounts SET balance = balance - 100 WHERE acc_no =
UPDATE Accounts SET balance = balance + 100 WHERE acc_no =
COMMIT;
-- Use ROLLBACK in case of errors
```

Advanced Functions for Exams:

- **Window Functions**: RANK(), DENSE_RANK(), ROW_NUMBER()
- **String Functions**: CONCAT(), SUBSTRING(), REPLACE()
- **Date Functions**: DATEADD(), DATEDIFF(), GETDATE()

Final Tip: Practice writing complex JOIN queries and understand the difference between IN, EXISTS, and JOIN operations for optimal performance in exams. Here's the comprehensive answer for Question 21 and the PL/SQL concepts:

Question 21: Student Management System Queries

Database Schema Analysis

Tables Structure:

- **Student (RollNo, Name, Age, Sex, City)**
    - RollNo: Student Roll Number (Primary Key)
    - Name: Student Name
    - Age: Student Age

- Sex: Student Gender (M/F)
- City: Student City

- **Student_marks (RollNo, Sub1, Sub2, Sub3, Total, Average)**
  - RollNo: Student Roll Number (Foreign Key)
  - Sub1: Subject 1 Marks
  - Sub2: Subject 2 Marks
  - Sub3: Subject 3 Marks
  - Total: Total Marks
  - Average: Average Marks

Detailed SQL Queries with Explanations

I. Display name and city of students with total marks > 225

```
SELECT s.Name, s.City
FROM Student s
JOIN Student_marks m ON s.RollNo = m.RollNo
WHERE m.Total > 225;
```

Explanation: Simple JOIN with WHERE condition on total marks.

J. Display students with >60 marks in each subject

```sql
SELECT s.Name
FROM Student s
JOIN Student_marks m ON s.RollNo = m.RollNo
WHERE m.Sub1 > 60 AND m.Sub2 > 60 AND m.Sub3 > 60;
```

Alternative using comparison:

```sql
SELECT s.Name
FROM Student s
JOIN Student_marks m ON s.RollNo = m.RollNo
WHERE LEAST(m.Sub1, m.Sub2, m.Sub3) > 60;
```

Explanation: Checks all three subjects exceed 60 marks.

K. Display cities with more than 10 students

```sql
SELECT City, COUNT(*) as student_count
FROM Student
GROUP BY City
HAVING COUNT(*) > 10;
```

Explanation: Uses GROUP BY with HAVING clause for aggregate condition.

L. Display unique pairs of male and female students

```sql
SELECT m.Name as Male_Student, f.Name as Female_Student
FROM Student m
CROSS JOIN Student f
WHERE m.Sex = 'M' AND f.Sex = 'F'
LIMIT 10; -- To avoid too many combinations
```

Alternative with row numbering:

```sql
SELECT
    m.Name as Male_Student,
    f.Name as Female_Student
FROM (
    SELECT Name, ROW_NUMBER() OVER () as rn
    FROM Student
    WHERE Sex = 'M'
) m
JOIN (
    SELECT Name, ROW_NUMBER() OVER () as rn
    FROM Student
    WHERE Sex = 'F'
) f ON m.rn = f.rn;
```

Explanation: Creates pairs of male and female students using CROSS JOIN or row numbering.

PL/SQL Concepts

1. Advantages of PL/SQL

Definition

PL/SQL (Procedural Language/Structured Query Language) is Oracle's procedural extension to SQL that adds programming capabilities to database operations.

Key Advantages:

1. Block Structure

- Organized code in logical blocks (DECLARE, BEGIN, EXCEPTION, END)
- Better code organization and maintenance

2. Procedural Capabilities

- Variables, constants, and data types
- Conditional statements (IF-THEN-ELSE)
- Looping constructs (FOR, WHILE)
- Exception handling

3. Better Performance

- Reduces network traffic by executing multiple SQL statements in single block
- Compilation and storage in database

4. Error Handling

- Comprehensive exception handling mechanism
- User-defined exceptions

5. Integration with SQL

- Seamless integration of SQL with procedural statements
- Support for cursors and transactions

6. Portability

- Runs on any platform where Oracle runs
- Platform-independent code

2. COMMIT and ROLLBACK Commands

COMMIT

- Makes all changes permanent in the database
- Ends the current transaction
- Releases transaction locks

Syntax:

```
COMMIT;
```

Example:

```
BEGIN
   UPDATE Accounts SET balance = balance - 100 WHERE acc_nc
   UPDATE Accounts SET balance = balance + 100 WHERE acc_n
   COMMIT; -- Makes both updates permanent
END;
```

ROLLBACK

- Undoes all changes made in the current transaction

- Restores database to state before transaction began
- Can rollback to savepoints

Syntax:

```
ROLLBACK; -- Rollback entire transaction
ROLLBACK TO savepoint_name; -- Rollback to specific savepoint
```

Example:

```
BEGIN
   SAVEPOINT sp1;
   UPDATE Employees SET salary = salary + 5000;

   -- If error occurs
   IF SQL%NOTFOUND THEN
      ROLLBACK TO sp1; -- Rollback to savepoint
   ELSE
      COMMIT;
   END IF;
END;
```

3. Cursors in PL/SQL

Definition

Cursors are database objects used to retrieve and manipulate multiple rows returned by SQL queries.

Types of Cursors:

1. Implicit Cursors

- Automatically created by Oracle for DML statements
- Attributes: SQL%FOUND, SQL%NOTFOUND, SQL%ROWCOUNT

Example:

```
BEGIN
   UPDATE Employees SET salary = salary * 1.1 WHERE dept_id =

   IF SQL%FOUND THEN
      DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' employees up
   END IF;
END;
```

2. Explicit Cursors

- Programmer-defined for complex result sets
- More control over data retrieval

Example:

```
DECLARE
   CURSOR emp_cursor IS
      SELECT employee_id, name, salary FROM Employees WHER
   emp_rec emp_cursor%ROWTYPE;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor INTO emp_rec;
      EXIT WHEN emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(emp_rec.name || ' - ' || emp_rec.sala
   END LOOP;
   CLOSE emp_cursor;
END;
```

3. Parameterized Cursors

- Accept parameters for dynamic queries

Example:

```
DECLARE
   CURSOR dept_cursor(p_dept_id NUMBER) IS
      SELECT * FROM Employees WHERE dept_id = p_dept_id;
BEGIN
   FOR emp_rec IN dept_cursor(10) LOOP
      DBMS_OUTPUT.PUT_LINE(emp_rec.name);
   END LOOP;
END;
```

4. REF Cursors

- Dynamic cursors that can be associated with different queries

- Used for returning result sets from stored procedures

4. Database Triggers in PL/SQL

Definition

Triggers are stored programs that automatically execute when specified database events occur.

Trigger Components:

1. **Triggering Event**: INSERT, UPDATE, DELETE
2. **Trigger Time**: BEFORE or AFTER
3. **Trigger Level**: ROW level or STATEMENT level
4. **Trigger Condition**: WHEN clause

Types of Triggers:

1. DML Triggers

- Fire on INSERT, UPDATE, DELETE operations

Example: Audit trigger

```
CREATE OR REPLACE TRIGGER audit_employee_changes
   BEFORE UPDATE OR DELETE ON Employees
   FOR EACH ROW
BEGIN
   IF UPDATING THEN
      INSERT INTO audit_table
      VALUES ('UPDATE', :OLD.employee_id, SYSDATE, USER);
   ELSIF DELETING THEN
      INSERT INTO audit_table
      VALUES ('DELETE', :OLD.employee_id, SYSDATE, USER);
   END IF;
END;
```

2. DDL Triggers

- Fire on DDL statements (CREATE, ALTER, DROP)

Example:

```
CREATE OR REPLACE TRIGGER prevent_drop_table
   BEFORE DROP ON DATABASE
BEGIN
   RAISE_APPLICATION_ERROR(-20001, 'Table drops not allowed');
END;
```

3. Database Event Triggers

- Fire on database events (LOGON, LOGOFF, STARTUP, SHUTDOWN)

Example: Logon trigger

```
CREATE OR REPLACE TRIGGER logon_trigger
   AFTER LOGON ON DATABASE
BEGIN
   INSERT INTO login_audit VALUES (USER, SYSDATE);
END;
```

Trigger Example: Auto-calculate total marks

```
CREATE OR REPLACE TRIGGER calculate_student_marks
   BEFORE INSERT OR UPDATE ON Student_marks
   FOR EACH ROW
BEGIN
   :NEW.Total := :NEW.Sub1 + :NEW.Sub2 + :NEW.Sub3;
   :NEW.Average := (:NEW.Sub1 + :NEW.Sub2 + :NEW.Sub3) / 3;
END;
```

Practice Questions & Solutions

Practice Question 1: Write a PL/SQL block to find employee with highest salary

```
DECLARE
    v_emp_name Employees.name%TYPE;
    v_max_salary Employees.salary%TYPE;
BEGIN
    SELECT name, salary INTO v_emp_name, v_max_salary
    FROM Employees
    WHERE salary = (SELECT MAX(salary) FROM Employees);

    DBMS_OUTPUT.PUT_LINE('Highest paid employee: ' || v_emp_nar
    DBMS_OUTPUT.PUT_LINE('Salary: ' || v_max_salary);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No employees found');
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('Multiple employees with same salar
END;
```

Practice Question 2: Create a trigger to maintain department employee count

```
CREATE OR REPLACE TRIGGER maintain_dept_count
   AFTER INSERT OR DELETE OR UPDATE OF dept_id ON Emplo
   FOR EACH ROW
BEGIN
   -- For INSERT
   IF INSERTING THEN
      UPDATE Departments
      SET emp_count = emp_count + 1
      WHERE dept_id = :NEW.dept_id;

   -- For DELETE
   ELSIF DELETING THEN
      UPDATE Departments
      SET emp_count = emp_count - 1
      WHERE dept_id = :OLD.dept_id;

   -- For UPDATE (department change)
   ELSIF UPDATING AND :OLD.dept_id != :NEW.dept_id THEN
      UPDATE Departments SET emp_count = emp_count - 1
      WHERE dept_id = :OLD.dept_id;

      UPDATE Departments SET emp_count = emp_count + 1
      WHERE dept_id = :NEW.dept_id;
   END IF;
END;
```

Exam Tips for PL/SQL

Common Mistakes to Avoid:

1. **Forgetting exception handling** in production code
2. **Not closing cursors** explicitly
3. **Using SELECT without INTO** in PL/SQL
4. **Mutating table errors** in triggers
5. **Infinite loops** in procedural code

Writing Style in Exams:

- Always include exception handling section
- Use meaningful variable names
- Comment complex logic
- Show both successful and error scenarios
- Mention performance considerations

Important PL/SQL Features to Remember:

- **%TYPE and %ROWTYPE** for variable declaration
- **BULK COLLECT** for better performance
- **FORALL** for bulk DML operations
- **Autonomous transactions** for independent commits
- **Package specifications and bodies**

Marking Scheme for PL/SQL Questions:

- **Correct Syntax**: 25%
- **Logic Implementation**: 30%
- **Exception Handling**: 15%
- **Code Efficiency**: 15%
- **Explanation Quality**: 15%

Final Tip: Practice writing PL/SQL blocks with different scenarios - data validation, complex calculations, error handling, and database triggers. Understand when to use cursors versus bulk operations for optimal performance.