SECTION A (20 MARKS)

1a. Compare sub-query and joins. (2 marks)

Question: Compare sub-query and joins.

```sql
-- Subqueries execute independently first, then outer query uses
results
-- Joins combine tables horizontally in single pass, more
efficient for large datasets

-- Subquery example (correlated - executes per outer row)
SELECT location, price
FROM bangalorehouses b1
WHERE price > (SELECT AVG(price) FROM bangalorehouses b2 WHERE
b2.location = b1.location);

-- Join example (more efficient)
SELECT b1.location, b1.price, b2.avg_price
FROM bangalorehouses b1
JOIN (SELECT location, AVG(price) as avg_price FROM
bangalorehouses GROUP BY location) b2
ON b1.location = b2.location
WHERE b1.price > b2.avg_price;
```

Explanation: Subqueries are easier to read for simple comparisons but can be slower as they execute repeatedly (correlated) or independently (non-correlated). Joins are more performant for complex relationships and allow indexing optimization. Use subqueries for readability in WHERE clauses, joins for SELECT/FROM clauses.

1b. Explain EXISTS operator with an example. (2 marks)

Question: Explain EXISTS operator with an example.

```sql
-- EXISTS returns TRUE if subquery returns any rows (doesn't
fetch data)
-- Faster than IN for large datasets as it stops at first match

SELECT DISTINCT location
FROM bangalorehouses h1
WHERE EXISTS (
    SELECT 1
    FROM bangalorehouses h2
    WHERE h2.location = h1.location
    AND h2.bath > 3
);
```

Explanation: EXISTS checks for existence only (uses semi-join internally), returning TRUE/FALSE immediately upon finding a match. Unlike IN (which transfers all

subquery results), EXISTS is more efficient for large tables. Correlated subquery executes once per outer row.

1c. State the difference between Primary key and candidate key. (2 marks)

Question: State the difference between Primary key and candidate key.

Answer:

- Primary Key: One chosen candidate key that uniquely identifies each row, cannot be NULL, no duplicates
- Candidate Key: Any column(s) that can uniquely identify rows (may have NULLs, multiple possible)

MySQL Example:

```sql
-- Candidate keys: customerNumber, (customerName+phone)
-- Primary key: customerNumber (chosen one)
CREATE TABLE customers (
    customerNumber INT PRIMARY KEY,
    customerName VARCHAR(50),
    phone VARCHAR(50)
);
```

Explanation: All primary keys are candidate keys, but not vice versa. Primary key enforces NOT NULL + UNIQUE + referenced in foreign keys. Candidate keys qualify but aren't designated as primary.

1d. Mention the difference between TRUNCATE and ROUND functions. (2 marks)

Question: Mention the difference between TRUNCATE and ROUND functions.

```sql
SELECT
    price,
    ROUND(price, 0) as rounded,      -- 39.07 -> 39
    TRUNCATE(price, 0) as truncated; -- 39.07 -> 39 (no rounding)
FROM bangalorehouses LIMIT 1;
```

Explanation:

- ROUND(x, d): Rounds to d decimal places (39.5 -> 40)
- TRUNCATE(x, d): Truncates toward zero (39.9 -> 39, -39.9 -> -39) Both remove decimals but ROUND considers mathematical rounding rules.

1e. Can we use aggregate functions in where clause? Justify your answer. (2 marks)

Question: Can we use aggregate functions in where clause? Justify your answer.

Answer: No, aggregate functions cannot be used directly in WHERE clause.

Correct approach - Use HAVING:

```sql
-- WRONG - Fails
SELECT location, AVG(price)
FROM bangalorehouses
WHERE AVG(price) > 100  -- ERROR!

-- CORRECT - Use HAVING
SELECT location, AVG(price)
FROM bangalorehouses
GROUP BY location
HAVING AVG(price) > 100;
```

Explanation: WHERE filters rows before GROUP BY/aggregation. HAVING filters groups after aggregation. Use subquery for WHERE aggregate filtering.

2a. What is normalization? What are the various forms of Normalization? (2 marks)

Question: What is normalization? What are the various forms of Normalization?

Answer: Normalization: Process of organizing data to eliminate redundancy and dependency.

Normal Forms:

- 1NF: Atomic values, no repeating groups
- 2NF: 1NF + no partial dependencies
- 3NF: 2NF + no transitive dependencies
- BCNF: Stronger 3NF (every determinant is candidate key)

Explanation: Reduces anomalies (insert/update/delete) by breaking tables into smaller, related tables with proper keys. Higher normal forms = less redundancy but may hurt performance.

2b. What is percent rank? Explain with an example. (2 marks)

Question: What is percent rank? Explain with an example.

Answer:

```sql
-- PERCENT_RANK() = (rank - 1) / (total_rows - 1)
SELECT
    location,
    price,
    PERCENT_RANK() OVER (ORDER BY price) as percent_rank
```

```
FROM bangalorehouses
ORDER BY price;
```

Explanation: PERCENT_RANK() gives relative rank as percentile (0-1). 3rd out of 10 houses = (3-1)/(10-1) = 0.222 (22.2nd percentile). Useful for distribution analysis within partitions. Requires window frame.

2c. Explain Check Constraint in-detail with an example. (2 marks)

Question: Explain Check Constraint in-detail with an example.

Answer:

```
CREATE TABLE bangalorehouses (
    price DECIMAL(10,2),
    bath INT,
    CHECK (bath BETWEEN 1 AND 10),
    CHECK (price > 0)
);

-- INSERT fails: violates CHECK
INSERT INTO bangalorehouses (bath, price) VALUES (0, 100); --
ERROR!
```

Explanation: CHECK constraints validate data at INSERT/UPDATE. MySQL 8.0+ supports CHECK (disabled in <8.0). Multiple CHECKs per table. Improves data integrity at database level vs application logic

2d. What are ACID properties? (2 marks)

Question: What are ACID properties?

Answer:

- Atomicity: All or nothing (transaction fully completes or rolls back)
- Consistency: Database moves from valid state to valid state
- Isolation: Concurrent transactions appear serial
- Durability: Committed changes survive failures

MySQL Example:

```
START TRANSACTION;
UPDATE bangalorehouses SET price = price * 1.1 WHERE location =
'Whitefield';
COMMIT; -- ACID ensures all rows updated or none
```

Explanation: ACID guarantees reliable transactions. MySQL InnoDB provides full ACID support via MVCC, logging, crash recovery.

2e. State and explain in brief types of Locks in database. (2 marks)

Question: State and explain in brief types of Locks in database.

Answer:

- Shared Lock (S): Multiple readers (SELECT), blocks writers
- Exclusive Lock (X): Single writer (UPDATE/DELETE), blocks all
- Intent Shared (IS): Table-level intent to acquire row S locks
- Intent Exclusive (IX): Table-level intent to acquire row X locks

MySQL Example:

```
SELECT * FROM bangalorehouses WHERE location = 'Whitefield' FOR
SHARE; -- Shared
SELECT * FROM bangalorehouses WHERE location = 'Whitefield' FOR
UPDATE; -- Exclusive
```

Explanation: InnoDB uses row-level locking with MVCC. Locks prevent conflicts during concurrent access. FOR SHARE/FOR UPDATE hints explicit locking.

SECTION B (40 MARKS) - House Database

3a. Find the average price of houses by location where the average price is greater than 100 lakhs. (4 marks)

Question: Find the average price of houses by location where the average price is greater than 100 lakhs.

```
USE house;
SELECT location, ROUND(AVG(price), 2) as avg_price_lakhs
FROM bangalorehouses
GROUP BY location
HAVING AVG(price) > 100;
```

Explanation: Groups houses by location, calculates average price in lakhs, filters locations with avg > 100 lakhs using HAVING (post-aggregation). ROUND formats output. Price stored as lakhs in dataset

3b. Find the number of houses in each location with more than 2 bathrooms and a total area greater than 1500 sqft. Only include locations with more than 5 such houses. (4 marks)

Question: Find the number of houses in each location with more than 2 bathrooms and a total area greater than 1500 sqft. Only include locations with more than 5 such

houses.

Answer:

```sql
SELECT location, COUNT(*) as house_count
FROM bangalorehouses
WHERE bath > 2
  AND totalsqft > 1500
GROUP BY location
HAVING COUNT(*) > 5;
```

Explanation: Filters houses (>2 bath, >1500 sqft), groups by location, counts qualifying houses, shows only locations with >5 matches. totalsqft is numeric column.

3c. List the cheapest house in each location using a subquery. (5 marks)

Question: List the cheapest house in each location using a subquery.

Answer:

```sql
SELECT *
FROM bangalorehouses b1
WHERE b1.price = (
    SELECT MIN(b2.price)
    FROM bangalorehouses b2
    WHERE b2.location = b1.location
);
```

Explanation: Correlated subquery finds minimum price per location. Outer query selects full row where price matches that minimum. Returns cheapest house details per location. Handles ties by returning all minimums.

3d. Find the details of the houses with the maximum price in the dataset. (4 marks)

Question: Find the details of the houses with the maximum price in the dataset.

Answer:

```sql
SELECT *
FROM bangalorehouses
WHERE price = (SELECT MAX(price) FROM bangalorehouses);
```

Explanation: Non-correlated subquery finds global maximum price. Main query returns all houses matching that maximum. Simple scalar subquery executes once.

3e. Use a window function to rank locations based on the average house price. (4 marks)

Question: Use a window function to rank locations based on the average house price.

Answer:

```sql
SELECT DISTINCT
    location,
    RANK() OVER (ORDER BY AVG(price) DESC) as price_rank
FROM bangalorehouses
GROUP BY location;
```

Explanation: Window function RANK() over avg(price) DESC per location group. DISTINCT avoids duplicate ranks. RANK() assigns 1 to highest avg price location.

3f. Use window functions to find the top 3 most expensive houses in each location. (7 marks)

Question: Use window functions to find the top 3 most expensive houses in each location.

Answer:

```sql
SELECT *
FROM (
    SELECT *,
           ROW_NUMBER() OVER (PARTITION BY location ORDER BY
price DESC) as rn
    FROM bangalorehouses
) ranked
WHERE rn <= 3;
```

Explanation: PARTITION BY location creates window per location. ROW_NUMBER() DESC ranks houses by price. Outer query filters top 3 (rn <= 3). ROW_NUMBER() handles ties uniquely.

3g. Find the houses located in Whitefield that have more than 3 bathrooms and their prices are in the top 10 in the location. (7 marks)

Question: Find the houses located in Whitefield that have more than 3 bathrooms and their prices are in the top 10 in the location.

Answer:

```sql
SELECT *
FROM (
    SELECT *,
           DENSE_RANK() OVER (PARTITION BY location ORDER BY
price DESC) as price_rank
    FROM bangalorehouses
    WHERE location = 'Whitefield'
) whitefield_ranked
WHERE bath > 3 AND price_rank <= 10;
```

Explanation: Filters Whitefield houses first, then DENSE_RANK() DESC within location window. Final filter (>3 bath + top 10 rank). DENSE_RANK() skips no numbers on ties.

3h. Find the number of houses with a price above the average price of all houses in Whitefield. (5 marks)

Question: Find the number of houses with a price above the average price of all houses in Whitefield.

Answer:

```sql
SELECT COUNT(*)
FROM bangalorehouses b1
WHERE b1.price > (
    SELECT AVG(price)
    FROM bangalorehouses b2
    WHERE b2.location = 'Whitefield'
);
```

Explanation: Scalar subquery calculates Whitefield average once. Main query counts all houses (global) exceeding that threshold. Non-correlated subquery.

SECTION C (40 MARKS) - Sales Database

4a. Find all employees who work in Paris office use join. (3 marks)

Question: Find all employees who work in Paris office use join.

Answer:

```sql
USE sales;
SELECT e.*
FROM employees e
JOIN offices o ON e.officeCode = o.officeCode
WHERE o.city = 'Paris';
```

Explanation: INNER JOIN links employees to offices via officeCode. WHERE filters Paris office. Returns all Paris employee details.

4b. Find out the total value customers in the USA ordered from the productLine Classic Cars. (8 marks)

Question: Find out the total value customers in the USA ordered from the productLine Classic Cars. The total value is the sum quantityOrdered * priceEach.

Answer:

```sql
SELECT SUM(od.quantityOrdered * od.priceEach) as
total_classic_cars_value
```

```sql
FROM orderdetails od
JOIN orders o ON od.orderNumber = o.orderNumber
JOIN customers c ON o.customerNumber = c.customerNumber
JOIN products p ON od.productCode = p.productCode
WHERE c.country = 'USA'
  AND p.productLine = 'Classic Cars';
```

Explanation: Multi-table JOIN: orderdetails→orders→customers→products. Calculates total value (quantity×price) for USA customers, Classic Cars only

4c. Get the total amount spent by each customer, along with their contact information. Use customers, payments tables. (5 marks)

Question: Get the total amount spent by each customer, along with their contact information. Use customers, payments tables.

Answer:

```sql
SELECT
    c.customerNumber,
    c.contactLastName,
    c.contactFirstName,
    SUM(p.amount) as total_spent
FROM customers c
LEFT JOIN payments p ON c.customerNumber = p.customerNumber
GROUP BY c.customerNumber, c.contactLastName, c.contactFirstName;
```

Explanation: LEFT JOIN includes customers with no payments (total=0). GROUP BY customer aggregates payments. Shows contact info + total spent.

4d. Find customer who has not ordered any products. (3 marks)

Question: Find customer who has not ordered any products.

Answer:

```sql
SELECT c.customerName
FROM customers c
LEFT JOIN orders o ON c.customerNumber = o.customerNumber
WHERE o.orderNumber IS NULL;
```

Explanation: LEFT JOIN keeps all customers. WHERE NULL identifies customers without matching orders (no orders placed). Anti-join pattern.

4e. Create a view called TRAINORDERS that shows all the orders that were placed in 2003 and included models of trains from Trains product line. Include orderNumber, orderDate, shippedDate, and customerNumber fields into the view. (6 marks)

Question: Create a view called TRAINORDERS that shows all the orders that were placed in 2003 and included models of trains from Trains product line. Include orderNumber, orderDate, shippedDate, and customerNumber fields into the view.

Answer:

```sql
CREATE VIEW TRAINORDERS AS
SELECT DISTINCT
    o.orderNumber,
    o.orderDate,
    o.shippedDate,
    o.customerNumber
FROM orders o
JOIN orderdetails od ON o.orderNumber = od.orderNumber
JOIN products p ON od.productCode = p.productCode
WHERE YEAR(o.orderDate) = 2003
  AND p.productLine = 'Trains';
```

Explanation: JOINs orders→orderdetails→products. YEAR() extracts 2003 orders with Trains products. DISTINCT avoids duplicate orderNumbers. VIEW materializes query.

4f. Find the Top 5 Customers and Their Corresponding Total Payments use CTE and window function. (7 marks)

Question: Find the Top 5 Customers and Their Corresponding Total Payments use CTE and window function.

Answer:

```sql
WITH customer_totals AS (
    SELECT
        c.customerNumber,
        c.customerName,
        SUM(p.amount) as total_payments,
        ROW_NUMBER() OVER (ORDER BY SUM(p.amount) DESC) as
payment_rank
    FROM customers c
    JOIN payments p ON c.customerNumber = p.customerNumber
    GROUP BY c.customerNumber, c.customerName
)
SELECT customerNumber, customerName, total_payments
FROM customer_totals
WHERE payment_rank <= 5;
```

Explanation: CTE calculates total payments per customer. Window ROW_NUMBER() ranks by total DESC. Final SELECT gets top 5. CTE improves readability.

4g. Customers who placed more than 5 orders. (3 marks)

Question: Customers who placed more than 5 orders.

Answer:

```sql
SELECT c.customerNumber, c.customerName, COUNT(o.orderNumber) as
order_count
FROM customers c
JOIN orders o ON c.customerNumber = o.customerNumber
GROUP BY c.customerNumber, c.customerName
HAVING COUNT(o.orderNumber) > 5;
```

Explanation: JOIN customers→orders, GROUP BY customer, COUNT orders, HAVING filters >5. Shows active customers.

4h. Find all Motorcycles that do not have the scale numbers 1:18. Include Bike name, scale, and description into the result. Sort ascending by Scale and then Bike Name. (5 marks)

Question: Find all Motorcycles that do not have the scale numbers 1:18. Include Bike name, scale, and description into the result. Sort ascending by Scale and then Bike Name.

Answer:

```sql
SELECT
    productName as bike_name,
    productScale as scale,
    productDescription as description
FROM products
WHERE productLine = 'Motorcycles'
  AND productScale != '1:18'
ORDER BY productScale ASC, productName ASC;
```

Explanation: Filters Motorcycles productLine, excludes 1:18 scale. SELECT renames columns. ORDER BY sorts scale then name ascending.