Section A: Theory Questions

Subquery vs Joins

Subqueries execute independently within a query's WHERE or SELECT clause, processing row-by-row and often less efficient for large datasets, while joins combine tables using matching columns for set-based operations that perform better with proper indexing.

EXISTS Operator

EXISTS checks if a subquery returns any rows, returning true immediately upon finding one match without fetching all data; example: SELECT * FROM customers c WHERE EXISTS (SELECT 1 FROM orders o WHERE o.customerNumber = c.customerNumber) verifies customers with orders.

Primary Key vs Candidate Key

A primary key uniquely identifies table rows and cannot accept NULLs or duplicates, whereas candidate keys are potential primary keys that uniquely identify rows but may include NULLs until selected.

TRUNCATE vs ROUND

TRUNCATE removes all rows from a table instantly without logging individual deletions, while ROUND formats numeric values to specified decimals, e.g., ROUND(123.456, 2) = 123.46.

Aggregate Functions in WHERE

Aggregate functions like SUM cannot be used directly in WHERE clauses as they operate on groups post-filtering; use HAVING instead for grouped conditions.

Normalization

Normalization eliminates data redundancy through normal forms: 1NF (atomic values), 2NF (no partial dependencies), 3NF (no transitive dependencies), BCNF (determinants as superkeys).

PERCENT_RANK

PERCENT_RANK window function computes relative rank as (rank - 1) / (rows preceding - 1) within partitions, e.g., PERCENT_RANK() OVER (ORDER BY price) gives percentile position.

Check Constraint

CHECK constrains column values, e.g., ALTER TABLE bangalorehouses ADD CONSTRAINT chk_price CHECK (price > 0) ensures positive prices.

ACID Properties

ACID ensures Atomicity (all-or-nothing), Consistency (valid states), Isolation (concurrent safety), Durability (committed persistence).

Database Locks

Shared locks allow concurrent reads, exclusive locks block others for writes; types include row-level, table-level, and page-level.

Section B: House Database Queries (bangalorehouses table)

3a. Average price by location >100 lakhs

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

3b. Houses count per location (>2 baths, >1500 sqft, location >5 houses)

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

3c. Cheapest house per location (subquery)

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

3d. Houses with maximum price

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

3e. Running total of prices per location

```
SELECT location, price,
       SUM(price) OVER (PARTITION BY location ORDER BY price) AS
running_total
FROM bangalorehouses;
```

3f. Top 3 expensive houses per location

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

3g. Whitefield houses (>3 baths, top 10 prices)

```
SELECT *
FROM (
    SELECT *, ROW_NUMBER() OVER (PARTITION BY location ORDER BY
price DESC) AS rn
    FROM bangalorehouses
    WHERE location = 'Whitefield' AND bath > 3
) ranked
WHERE rn <= 10;
```

3h. Houses above Whitefield average price

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

Section C: Sales Database Queries

4a. Paris office employees (join)

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

4b. USA Classic Cars total value

```
SELECT SUM(od.quantityOrdered * od.priceEach) AS total_value
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
JOIN productlines pl ON p.productLine = pl.productLine
WHERE c.country = 'USA' AND pl.productLine = 'Classic Cars';
```

4c. Total spent per customer with contact

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

4d. Customers without orders

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

4e. TRAINORDERS view

```
CREATE VIEW TRAINORDERS AS
SELECT 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';
```

4f. Top 5 customers by payments (CTE + window)

```
WITH customer_totals AS (
    SELECT customerNumber, SUM(amount) AS total_payment,
           ROW_NUMBER() OVER (ORDER BY SUM(amount) DESC) AS rn
    FROM payments
    GROUP BY customerNumber
)
SELECT c.customerName, ct.total_payment
FROM customer_totals ct
JOIN customers c ON ct.customerNumber = c.customerNumber
WHERE ct.rn <= 5;
```

4g. Customers with >5 orders

```
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;
```