Section A – Short Answers (20 Marks)

Q1 a) How to add foreign keys in MySQL?

Answer: In MySQL, a foreign key is added either at table creation time using the FOREIGN KEY clause in CREATE TABLE, or later using ALTER TABLE with ADD CONSTRAINT FOREIGN KEY that references the parent table's primary (or unique) key. The child column(s) and the referenced parent column(s) must have compatible data types and indexing for the constraint to be valid.

Example (creating with foreign key):

```sql
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    CONSTRAINT fk_orders_customer
        FOREIGN KEY (customer_id) REFERENCES
customers(customer_id)
);
```

Example (adding later):

```sql
ALTER TABLE orders
ADD CONSTRAINT fk_orders_customer
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id);
```

Explanation: A foreign key enforces referential integrity by ensuring that a value in the child table exists in the parent table's referenced column, preventing creation of orphan records. MySQL can also be configured with ON DELETE and ON UPDATE actions (CASCADE, RESTRICT, SET NULL, etc.) to control what happens when the parent key is modified or deleted.

Q1 b) What are the levels at which check constraints can be created? Justify your answer on what all data types the constraints can be applied.

Answer: In MySQL 8.0 and later, CHECK constraints can be defined at the column level (directly on a single column definition) and at the table level (as a separate constraint that can involve one or more columns). CHECK conditions are Boolean expressions and can be applied to any data type that participates in logical comparisons, including numeric, character, date/time, and Boolean-like columns, as long as the expression evaluates to TRUE or UNKNOWN for valid rows.

Example – column level:

```sql
CREATE TABLE employees (
    salary DECIMAL(10,2)
        CHECK (salary > 0)
);
```

Example – table level:

```sql
CREATE TABLE orders (
    qty INT,
    price DECIMAL(10,2),
    CHECK (qty >= 0 AND price >= 0)
);
```

Explanation: A column-level CHECK typically restricts values for that single column, while a table-level CHECK can reference multiple columns to enforce more complex rules (for example, qty * price > 0). Data types such as INT, DECIMAL, VARCHAR, DATE, and DATETIME can all be validated using comparisons, ranges, IN-lists, and functions, provided the expression is deterministic and supported by MySQL's CHECK implementation.

Q1 c) What is the difference between isnull operator() and ifnull() function?

Answer: In MySQL, IS NULL is a logical operator used in WHERE, JOIN, or conditional expressions to test whether a column value is NULL and returns TRUE or FALSE, while IFNULL(expr1, expr2) is a function that returns expr2 when expr1 is NULL, otherwise returns expr1. IS NULL does not change the value, it only checks for NULL, whereas IFNULL is used for value substitution to avoid NULL in result sets or calculations.

Example – IS NULL:

```sql
SELECT * FROM employees
WHERE manager_id IS NULL;
```

Example – IFNULL:

```sql
SELECT IFNULL(commission, 0) AS commission_value
FROM employees;
```

Explanation: Use IS NULL in predicates when filtering rows based on the presence or absence of NULL values, such as in WHERE or ON clauses. Use IFNULL when you need to replace NULL with a default value (for example, turning NULL commission into 0) so that aggregates, arithmetic expressions, or display columns do not propagate NULL.

Q1 d) Is it possible to have two primary keys in a table? Explain what is meant by alternate keys in RDBMS with an example.

Answer: It is not possible to have two separate primary keys in a single MySQL table; a table can have only one PRIMARY KEY constraint, which may consist of one or more columns (a composite primary key). However, a table can have multiple candidate keys defined via UNIQUE constraints, and all candidate keys that are not chosen as the primary key are called alternate keys.

Example:

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(20) UNIQUE
);
```

Explanation: In the above example, student_id, email, and phone are all candidate keys because each can uniquely identify a row, but only student_id is chosen as the primary key. The remaining UNIQUE columns (email and phone) act as alternate keys and still enforce uniqueness for their respective attributes. Alternate keys improve data integrity by preventing duplicate values in important identification columns other than the primary key.

Q1 e) What is a view? Mention differences between Simple and Complex view.

Answer: A view in MySQL is a virtual table defined by a stored SELECT query whose result set can be queried like a regular table, but the data is not physically stored; instead, it reflects data from the underlying base tables each time it is accessed. Views can encapsulate complex joins, filters, and calculations, providing logical abstraction and security by exposing only selected columns and rows.

Differences between Simple and Complex view:

- Simple view:
  - Based on a single base table.
  - Typically does not contain GROUP BY, DISTINCT, aggregate functions, or subqueries.
  - Often updatable (subject to MySQL's updatability rules).
  - Example: CREATE VIEW active_customers AS SELECT customer_id, customer_name FROM customers WHERE status = 'ACTIVE';
- Complex view:
  - Based on multiple tables or involves joins, subqueries, GROUP BY, UNION, or aggregate functions.
  - Often not directly updatable because of derived or aggregated data.
  - Used primarily for reporting and analytics rather than direct DML.
  - Example: CREATE VIEW region_sales_summary AS SELECT region, SUM(sales) AS total_sales FROM orders GROUP BY region;

Explanation: Views help hide underlying schema complexity, enforce column-level security, and provide reusable query logic. Simple views closely mirror a single table's structure, whereas complex views are designed for summarized or combined data, often trading updatability for richer analytical power.

Q2 a) What are the uses of Window Function?

Answer: In MySQL, window functions compute values across sets of rows that are related to the current row, without collapsing them into a single summary row, using the OVER() clause. They are used for running totals, moving averages, rankings, percentiles, lag/lead comparisons, and partition-based analytics while still returning one row per input row.

Example:

```
SELECT
    customer_id,
    order_date,
    amount,
    SUM(amount) OVER (PARTITION BY customer_id ORDER BY
order_date) AS running_total
FROM orders;
```

Explanation: Unlike GROUP BY aggregates, window functions allow analytical calculations to be performed "over" a window (partition and order) while retaining full row-level detail. This enables powerful time-series and comparative analysis such as rank within a partition, row-numbering, and frame-based aggregates within the same SELECT.

Q2 b) Explain about EXISTS operator.

Answer: The EXISTS operator in MySQL is used in subqueries to test whether the subquery returns at least one row; it evaluates to TRUE if the subquery yields any row and FALSE otherwise. EXISTS is often used with correlated subqueries to check the existence of related data in another table without needing to retrieve actual column values from the subquery.

Example:

```
SELECT c.customer_id, c.customer_name
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id
);
```

Explanation: EXISTS is semantically a Boolean existence test and can be more efficient than IN for correlated checks, because the subquery can stop searching as soon as a matching row is found. It is commonly used to implement "where related rows exist / do not exist" logic, such as listing customers who have at least one order or those who have no orders (using NOT EXISTS).

Q2 c) What is percent rank? Explain with an example.

Answer: PERCENT_RANK is a window function that computes the relative rank of a row within a partition as a value between 0 and 1, defined as (rank - 1) / (total_rows_in_partition - 1), giving 0 for the first row and 1 for the last row (when there is more than one row). It shows the position of each row as a fraction of the total, useful for percentile-style analysis and identifying outliers.

Example:

```sql
SELECT
    employee_id,
    salary,
    PERCENT_RANK() OVER (ORDER BY salary) AS salary_percent_rank
FROM employees;
```

Explanation: In this example, employees are ordered by salary, and PERCENT_RANK shows each employee's relative standing in the salary distribution: values close to 0 are among the lowest paid, and values close to 1 are among the highest paid. When the partition has only one row, PERCENT_RANK returns 0 because both numerator and denominator reduce to 0 by definition.

Q2 d) State and explain in brief types of Locks in database.

Answer: In MySQL (especially with InnoDB), the main lock types are shared (read) locks and exclusive (write) locks at the row or table level. Shared locks allow concurrent transactions to read the same rows but prevent those rows from being modified, while exclusive locks block both reads (that require shared locks) and writes from other transactions on the locked rows until the lock is released.

Key lock types:

- Shared (S) lock:
  - Acquired for read operations.
  - Multiple transactions can hold shared locks on the same rows concurrently.
- Exclusive (X) lock:
  - Acquired for insert, update, or delete operations.
  - Only one transaction can hold an exclusive lock on a resource at a time.
- Intention locks (IS, IX):
  - Placed at the table level to signal a transaction's intent to acquire row-level locks, helping the engine coordinate table and row locking.
- Table locks:
  - Lock the entire table for read or write, used by some storage engines or explicitly via LOCK TABLES.

Explanation: The locking mechanism ensures isolation and consistency by coordinating concurrent access to rows or tables; shared locks protect data during reads, and exclusive locks protect data integrity during writes. InnoDB's row-level locking combined with intention locks improves concurrency, reducing contention compared with coarse table-level locking.

Q2 e) Explain the different ways you can restrict the data in a column. Demonstrate with examples.

Answer: In MySQL, data in a column can be restricted using constraints and data type definitions such as NOT NULL, DEFAULT, CHECK, UNIQUE, ENUM/SET, and FOREIGN KEY, as well as application of domain-specific data types (for example, appropriate length limits). These mechanisms ensure that only valid, consistent values are inserted or updated in the column.

Common ways with examples:

1. NOT NULL:

   - Ensures the column cannot store NULL.
   - Example:

   ```
   CREATE TABLE employees (
       emp_id INT PRIMARY KEY,
       name VARCHAR(100) NOT NULL
   );
   ```

2. CHECK:

   - Enforces a Boolean condition on the column.
   - Example:

   ```
   CREATE TABLE products (
       price DECIMAL(10,2),
       CHECK (price > 0)
   );
   ```

3. UNIQUE:

   - Enforces uniqueness of column values.
   - Example:

   ```
   CREATE TABLE users (
       user_id INT PRIMARY KEY,
       email VARCHAR(100) UNIQUE
   );
   ```

4. ENUM / SET:

   - Restricts values to a predefined list.
   - Example:

   ```
   CREATE TABLE orders (
       status ENUM('NEW', 'SHIPPED', 'CANCELLED') NOT NULL
   );
   ```

5. Data type and length:

   - Using appropriate type and size limits.
   - Example:

   ```
   CREATE TABLE customers (
       country_code CHAR(2)
   );
   ```

6. FOREIGN KEY:

   - Restricts values to those existing in a referenced table.
   - Example:

   ```
   CREATE TABLE orders (
       customer_id INT,
   ```

```
            CONSTRAINT fk_cust FOREIGN KEY (customer_id)
                REFERENCES customers(customer_id)
        );
```

Explanation: Combining these mechanisms allows definition of strong domain rules: NOT NULL and CHECK capture presence and range conditions, UNIQUE and FOREIGN KEY enforce identity and referential integrity, and ENUM/SET ensure membership in controlled categorical values. Properly designed column constraints reduce reliance on application-side validation and prevent invalid data from being stored in the first place.

Q3 a) Use superstore database to solve below. Write a query to find total sales for each region and ship mode combination for orders in year 2020.

Answer: Assuming the Superstore schema has an Orders-like table with columns: order_date, region, ship_mode, and sales, the MySQL query is:

```
SELECT
    region,
    ship_mode,
    SUM(sales) AS total_sales
FROM orders
WHERE YEAR(order_date) = 2020
GROUP BY
    region,
    ship_mode
ORDER BY
    region,
    ship_mode;
```

Explanation: The WHERE clause filters orders to those whose order_date falls in the year 2020, using YEAR(order_date) = 2020. The GROUP BY groups the remaining rows by region and ship_mode and SUM(sales) aggregates the sales within each group, producing one row per (region, ship_mode) combination with its total_sales.

Q3 b) Write a query to print first name and last name of a customer using orders table (everything after first space can be considered as last name) customer_name, first_name, last_name.

Answer: Assuming the Orders table has a customer_name column containing the full name, the MySQL query is:

```
SELECT
    customer_name,
    SUBSTRING_INDEX(customer_name, ' ', 1) AS first_name,
    SUBSTRING(customer_name,
LENGTH(SUBSTRING_INDEX(customer_name, ' ', 1)) + 2) AS last_name
FROM orders;
```

Explanation: SUBSTRING_INDEX(customer_name, ' ', 1) returns the substring before the first space, which is treated as the first name. The last name is obtained by taking a SUBSTRING starting after the first space: the length of the first name plus one space (hence +2 for space plus 1-based indexing) so that everything after the first space is returned as last_name, even if it contains multiple words.

Q3 c) Find the segments where the total sales exceed the average total sales across all segments.

Answer: Assuming Orders has columns segment and sales, the MySQL query is:

```
WITH segment_sales AS (
    SELECT
        segment,
        SUM(sales) AS total_sales
    FROM orders
    GROUP BY segment
),
avg_sales AS (
    SELECT AVG(total_sales) AS avg_total_sales
    FROM segment_sales
)
SELECT
    s.segment,
    s.total_sales
FROM segment_sales s
CROSS JOIN avg_sales a
WHERE s.total_sales > a.avg_total_sales;
```

Explanation: The first CTE segment_sales aggregates total_sales per segment. The second CTE avg_sales computes the average of these segment-level totals. The final SELECT joins each segment's total_sales with the overall average and filters to only those segments where total_sales is strictly greater than the average total_sales across all segments.

Q3 d) Determine the month with the highest average discount rate.

Answer: Assuming Orders has discount (as a numeric rate per row) and order_date, the MySQL query is:

```
SELECT
    DATE_FORMAT(order_date, '%Y-%m') AS year_month,
    AVG(discount) AS avg_discount
FROM orders
GROUP BY
    DATE_FORMAT(order_date, '%Y-%m')
ORDER BY
    avg_discount DESC
LIMIT 1;
```

Explanation: DATE_FORMAT(order_date, '%Y-%m') groups orders by calendar year and month. For each month group, AVG(discount) computes the mean discount rate. Ordering by avg_discount in descending order and using LIMIT 1 returns the single month with the highest average discount rate.

Q3 e) Calculate the moving average of sales for each product over a 3-month period.

Answer: Assuming Orders has product_id (or product_name), order_date, and sales, the MySQL query using window functions is:

```
SELECT
    product_id,
    DATE_FORMAT(order_date, '%Y-%m') AS year_month,
    SUM(sales) AS monthly_sales,
    AVG(SUM(sales)) OVER (
        PARTITION BY product_id
        ORDER BY DATE_FORMAT(order_date, '%Y-%m')
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS moving_avg_3_month
FROM orders
GROUP BY
    product_id,
    DATE_FORMAT(order_date, '%Y-%m')
ORDER BY
    product_id,
    year_month;
```

Explanation: First, the query aggregates sales per product per month using GROUP BY product_id and year_month. Then, a window function AVG(SUM(sales)) OVER with PARTITION BY product_id and a frame of ROWS BETWEEN 2 PRECEDING AND CURRENT ROW computes the 3-row (3-month) moving average across consecutive months for each product, giving smoothed sales trends over time.

Q3 f) Find the best-selling and most profitable category in the given dataset.

Answer: Assuming Orders has category, sales, and profit, the MySQL query is:

```
WITH category_stats AS (
    SELECT
        category,
        SUM(sales)  AS total_sales,
        SUM(profit) AS total_profit
    FROM orders
    GROUP BY category
),
best_selling AS (
    SELECT
        category AS best_selling_category,
        total_sales
    FROM category_stats
    ORDER BY total_sales DESC
    LIMIT 1
),
most_profitable AS (
    SELECT
        category AS most_profitable_category,
        total_profit
    FROM category_stats
    ORDER BY total_profit DESC
    LIMIT 1
)
SELECT
    b.best_selling_category,
    b.total_sales,
    m.most_profitable_category,
    m.total_profit
FROM best_selling b
CROSS JOIN most_profitable m;
```

Explanation: The category_stats CTE computes total_sales and total_profit for each category. The best_selling CTE selects the category with the highest total_sales, while most_profitable picks the one with the highest total_profit. The final SELECT combines both to report the best-selling category (highest revenue) and the most profitable category (highest profit), which may or may not be the same.

Q3 g) Identify the top 5 most profitable months in terms of total profit.

Answer: Assuming Orders has order_date and profit, the MySQL query is:

```
SELECT
    DATE_FORMAT(order_date, '%Y-%m') AS year_month,
    SUM(profit) AS total_profit
FROM orders
GROUP BY
    DATE_FORMAT(order_date, '%Y-%m')
ORDER BY
    total_profit DESC
LIMIT 5;
```

Explanation: The query aggregates total profit for each calendar year-month using GROUP BY on DATE_FORMAT(order_date, '%Y-%m'). Ordering by total_profit in descending order places the most profitable months first, and LIMIT 5 returns the top five months with the highest aggregated profit.

Q3 h) Which country sold the most products?

Answer: Assuming Orders (or a related table) has country and quantity_sold (or quantity) per order line, the MySQL query is:

```
SELECT
    country,
    SUM(quantity) AS total_quantity_sold
```

```
FROM orders
GROUP BY country
ORDER BY
    total_quantity_sold DESC
LIMIT 1;
```

Explanation: The query groups rows by country and sums the quantity column to find the total number of products sold in each country. Sorting by total_quantity_sold in descending order and limiting to one row yields the country with the highest total quantity of products sold, which answers which country sold the most products.

Q4 a) Use orders schema. Find all employees who work in 'Paris' office.

I. Write correlated subquery to get the results.
II. Write the same query using JOIN.

Answer: I. Correlated subquery:

```
SELECT
    e.*
FROM employees e
WHERE e.officeCode = (
    SELECT o.officeCode
    FROM offices o
    WHERE o.city = 'Paris'
);
```

II. JOIN:

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

Explanation: In the correlated subquery version, for each employee row, MySQL evaluates the subquery that returns the officeCode for the Paris office and selects employees whose officeCode matches it. In the JOIN version, employees are joined directly to offices on officeCode, and the WHERE clause filters to the office whose city is 'Paris', returning all employees located at that office.

Q4 b) Find out the total value customers in the USA ordered from the productLine 'Classic Cars'. The total value is the sum quantityOrdered * priceEach. Order by total value descending.

Answer: Using the classic orders schema (customers, orders, orderdetails, products, productlines), the MySQL query is:

```
SELECT
    c.customerNumber,
    c.customerName,
    SUM(od.quantityOrdered * od.priceEach) AS total_value
FROM customers c
JOIN orders o
    ON c.customerNumber = o.customerNumber
JOIN orderdetails od
    ON o.orderNumber = od.orderNumber
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'
GROUP BY
    c.customerNumber,
    c.customerName
ORDER BY
    total_value DESC;
```

Explanation: The query joins customers with orders, orderdetails, products, and productlines to combine customer, order line, and product category information. The WHERE clause restricts to customers in the USA and products from the 'Classic Cars' product line. For each customer, the total value is computed as the sum of quantityOrdered * priceEach across all matching order lines, then ordered in descending order of total_value.

Q4 c) 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: Assuming products has columns productName (bike name), productScale, productDescription, and productLine, the MySQL query is:

```
SELECT
    p.productName  AS bike_name,
    p.productScale AS scale,
    p.productDescription AS description
FROM products p
WHERE
    p.productLine = 'Motorcycles'
    AND p.productScale <> '1:18'
ORDER BY
    p.productScale ASC,
    p.productName ASC;
```

Explanation: The WHERE clause restricts products to those in the 'Motorcycles' product line and excludes any rows whose productScale is '1:18'. The SELECT list returns the requested columns (bike name, scale, description), and the ORDER BY sorts first by scale and then by bike_name in ascending order.

Q4 d) Find customer who has not ordered any products.

Answer: Using customers and orders in the orders schema, one way is to use NOT EXISTS:

```
SELECT
    c.*
FROM customers c
WHERE NOT EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customerNumber = c.customerNumber
);

Alternatively, using LEFT JOIN:

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

Explanation: Both queries identify customers whose customerNumber does not appear in the orders table. The NOT EXISTS version checks for absence of any matching order row, while the LEFT JOIN version joins customers to orders and then filters rows where orderNumber is NULL, which only happens when no orders are associated with that customer.

Q4 e) Create a view called TRAIN_ORDERS 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: Using orders, orderdetails, products, and productlines, the MySQL statement is:

```
CREATE VIEW TRAIN_ORDERS 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
JOIN productlines pl
    ON p.productLine = pl.productLine
WHERE
    YEAR(o.orderDate) = 2003
    AND pl.productLine = 'Trains';
```

Explanation: The view TRAIN_ORDERS is defined by joining orders with orderdetails, products, and productlines to identify orders that contain at least one product whose product line is 'Trains'. The WHERE clause restricts orders to those placed in the year 2003, and DISTINCT ensures that each qualifying order appears only once in the view even if multiple train products are in the same order. The view exposes only the requested columns: orderNumber, orderDate, shippedDate, and customerNumber.

```
In [ ]:
```