Perfect — let's tackle this systematically. You've uploaded the **exam paper schema** for *Databases & SQL (MTech)*. I'll walk through **Section A, B, and C** one by one, giving clear answers with reasoning, SQL examples, and business interpretation where needed.

✅ SECTION A (Conceptual, 20 Marks)

1(a) Primary Key vs Candidate Key

- **Primary Key**: A unique identifier chosen from candidate keys. Only one per table, cannot be NULL.
- **Candidate Key**: Any column(s) that can uniquely identify a row. A table can have multiple candidate keys. 👉 Example: In `Employee(Employee_id, Email)`, both `Employee_id` and `Email` are candidate keys, but only one becomes the primary key.

1(b) WHERE vs GROUP BY

- **WHERE**: Filters rows *before* grouping/aggregation.
- **GROUP BY**: Groups rows into sets for aggregation functions (SUM, COUNT, AVG).
  👉 Example:

```
SELECT department, COUNT(*)
FROM employees
WHERE salary > 5000
GROUP BY department;
```
Here, `WHERE` filters salaries first, then `GROUP BY` counts per department.

1(c) DROP VIEW & Updating Views

- `DROP VIEW view_name;` → Deletes the view definition, not the base tables.
- **Updating Views**: Possible only if the view is based on a single table without aggregates, DISTINCT, GROUP BY, or joins. Otherwise, not updatable.

1(d) EXISTS Operator

- Checks if a subquery returns rows. Returns TRUE if at least one row exists.
  👉 Example:

```
SELECT employee_name
FROM employees e
WHERE EXISTS (
  SELECT 1 FROM projects p WHERE p.employee_id = e.employee_id
);
```
This lists employees assigned to projects.

1(e) RANK() vs DENSE_RANK()

- **RANK()**: Leaves gaps when ties occur.
- **DENSE_RANK()**: No gaps; next rank is consecutive.
  👉 Example:

```
SELECT employee_name, salary,
       RANK() OVER (ORDER BY salary DESC) AS rnk,
       DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rnk
FROM employees;
```
If two employees share 2nd place, RANK jumps to 4, DENSE_RANK goes to 3.

2(a) Joins vs Subqueries

Use **joins** when:

- You need data from multiple tables simultaneously.
- Performance matters (joins are optimized).
- You want to combine related rows.
  Subqueries are better for filtering or existence checks.

2(b) Types of Locks

- **Shared Lock (S)**: Multiple reads allowed, no writes.
- **Exclusive Lock (X)**: Only one transaction can modify.
- **Update Lock (U)**: Prevents deadlock during updates.
- **Intent Locks**: Signals intention to acquire locks at lower granularity.

2(c) Check Constraints

- Can be applied at **column level** or **table level**.
- Works on numeric, string, and date types.
  👉 Example:

```
salary INT CHECK (salary > 0);
```

2(d) Dropping a Table

- Drops **columns, constraints, indexes, defaults**.
- Does **NOT** drop views or stored procedures (they reference the table but remain unless explicitly dropped).

2(e) Multi-row Operators

- **IN, ANY, ALL** are multi-row operators.
  👉 Example:

```
SELECT employee_name
FROM employees
WHERE salary IN (SELECT salary FROM managers);
```

✅ SECTION B (Applied SQL, 30 Marks)

3(a) Reporting Manager, Client, Project

```
SELECT e.employee_name,
       m.employee_name AS manager_name,
       p.project_name,
       c.client_name
FROM employees e
JOIN employees m ON e.manager_id = m.employee_id
JOIN projects p ON e.employee_id = p.employee_id
JOIN clients c ON p.client_id = c.client_id;
```

3(b) Inactive Clients

Inactive = clients with **no projects**.

```
SELECT c.client_id, c.client_name, c.client_country
FROM clients c
WHERE c.client_id NOT IN (SELECT DISTINCT client_id FROM
projects);
```

3(c) Conversion Rate (CVR)

Formula:

[CVR = \frac{\text{Unique Orders}}{\text{Unique Sessions}} \times 100]

```
SELECT COUNT(DISTINCT s.session_id) AS sessions,
       COUNT(DISTINCT o.order_id) AS orders,
       (COUNT(DISTINCT o.order_id)*100.0 / COUNT(DISTINCT
s.session_id)) AS conversion_rate
FROM sessions_website s
LEFT JOIN orders o ON s.session_id = o.session_id
WHERE s.visited_at <= '2020-08-24'
  AND s.source_UTM IN ('gsearch','nonbrand');
```

3(d) Monthly Session Volume (gsearch vs bsearch)

```
SELECT DATE_TRUNC('month', visited_at) AS month_start,
       SUM(CASE WHEN source_UTM='gsearch' AND
device_type='desktop' THEN 1 ELSE 0 END) AS g_dtop_sessions,
       SUM(CASE WHEN source_UTM='bsearch' AND
device_type='desktop' THEN 1 ELSE 0 END) AS b_dtop_sessions,
       (SUM(CASE WHEN source_UTM='bsearch' AND
device_type='desktop' THEN 1 ELSE 0 END) *1.0 /
        NULLIF(SUM(CASE WHEN source_UTM='gsearch' AND
device_type='desktop' THEN 1 ELSE 0 END),0)) AS b_pct_of_g_dtop,
       SUM(CASE WHEN source_UTM='gsearch' AND
device_type='mobile' THEN 1 ELSE 0 END) AS g_mob_sessions,
       SUM(CASE WHEN source_UTM='bsearch' AND
device_type='mobile' THEN 1 ELSE 0 END) AS b_mob_sessions,
       (SUM(CASE WHEN source_UTM='bsearch' AND
device_type='mobile' THEN 1 ELSE 0 END) *1.0 /
        NULLIF(SUM(CASE WHEN source_UTM='gsearch' AND
device_type='mobile' THEN 1 ELSE 0 END),0)) AS b_pct_of_g_mob
FROM sessions_website
WHERE visited_at BETWEEN '2012-08-01' AND '2012-12-22'
GROUP BY DATE_TRUNC('month', visited_at);
```

3(e)(i) Rank Dishes by Cook Time

```
SELECT name, ingredients,
       RANK() OVER (ORDER BY cook_time DESC) AS cook_rank
FROM indian_food;
```

3(e)(ii) Minimum Cook Time per Course

```
SELECT course, name, ingredients, cook_time
FROM indian_food i
WHERE cook_time = (
   SELECT MIN(cook_time)
   FROM indian_food
   WHERE course = i.course
);
```

✅ SECTION C (Clothing Database, 30 Marks)

4(a)(i) 5% Discount on Accessories

```
SELECT c.catalogue_entry_description,
       p.product_price AS old_price,
       p.product_price * 0.95 AS new_price
FROM products p
JOIN catalogue c ON p.catalogue_entry_id = c.catalogue_entry_id
WHERE c.catalogue_entry_description LIKE '%Accessories%';
```

4(a)(ii) Average Price > 1000

```sql
SELECT catalogue_entry_id,
       AVG(product_price) AS avg_price
FROM products
GROUP BY catalogue_entry_id
HAVING AVG(product_price) > 1000;
```

4(a)(iii) Count Products per Catalogue

```sql
SELECT catalogue_entry_id,
       COUNT(*) AS product_count
FROM products
GROUP BY catalogue_entry_id;
```

4(a)(iv) Parent & Grandparent Catalogue

```sql
SELECT c1.catalogue_entry_description AS prod_desc,
       c2.catalogue_entry_description AS main_prod_desc,
       c3.catalogue_entry_description AS parent_prod_desc
FROM catalogue c1
LEFT JOIN catalogue c2 ON c1.parent_catalogue_entry_id =
c2.catalogue_entry_id
LEFT JOIN catalogue c3 ON c2.parent_catalogue_entry_id =
c3.catalogue_entry_id;
```

4(a)(v) Adidas Shirts

```sql
SELECT c.catalogue_entry_description, s.size_description,
col.color_code_description,
       p.product_price, p.product_description
FROM products p
JOIN catalogue c ON p.catalogue_entry_id = c.catalogue_entry_id
JOIN sizes s ON p.size_code = s.size_code
JOIN colors col ON p.color_code = col.color_code
WHERE p.product_description LIKE '%Adidas%' AND
c.catalogue_entry_description LIKE '%Shirt%';
```

4(b) Undo Client Update (TCL)

```sql
SET autocommit = 0;
SAVEPOINT a;
UPDATE clients SET client_country='India';
ROLLBACK TO SAVEPOINT a;
COMMIT;
```

🎯 Wrap-up

I've answered **all exam questions** with:

- Clear conceptual differences (Section A)
- SQL queries with joins, subqueries, aggregates (Section B)
- ER-based queries with catalogue/products (Section C)

This mirrors how you’d solve in exams: concise theory + executable SQL.