Perfect — you've provided the schema and data for **warehouse_employees**, and the ESA paper has three sections (A: theory, B: SQL queries on warehouse_employees, C: SQL queries on sales schema). Let's go through **all questions neatly, with clear answers and code**.

📘 SECTION A – Theory (20 Marks)

1a. Difference between Primary Key and Unique Key

- **Primary Key**: Uniquely identifies each row in a table. Cannot contain NULL values. Only one per table.
- **Unique Key**: Ensures all values in a column are unique. Can contain a single NULL. Multiple unique keys allowed.

1b. Difference between DELETE and TRUNCATE

- **DELETE TABLE**: Removes rows one by one, can have `WHERE` clause, logs each deletion, slower.
- **TRUNCATE TABLE**: Removes all rows instantly, cannot use `WHERE`, resets auto-increment, faster, minimal logging.

1c. Transaction & ACID Properties

- **Transaction**: A sequence of SQL operations treated as a single logical unit of work.
- **ACID**:
  - **Atomicity**: All or nothing.
  - **Consistency**: Preserves database integrity.
  - **Isolation**: Transactions don't interfere.
  - **Durability**: Changes persist after commit.

1d. Concatenate Function in MySQL

- `CONCAT(str1, str2, …)` → joins strings.

1e. Rename Table

```
RENAME TABLE old_table TO new_table;
```

2a. View in MySQL

- **View**: Virtual table based on query result.

```
CREATE VIEW view_name AS SELECT ...;
DROP VIEW view_name;
```

2b. Commit & Rollback

- **COMMIT**: Saves transaction permanently.
- **ROLLBACK**: Undoes transaction to last commit.

2c. Error in Query

```
SELECT department_id, count(*)
FROM department
WHERE count(*) > 5
GROUP BY department_id;
```

❌ Wrong: `WHERE` cannot use aggregate functions.
✅ Correct:

```
SELECT department_id, COUNT(*)
FROM department
GROUP BY department_id
HAVING COUNT(*) > 5;
```

2d. Cross Join

- Produces Cartesian product of two tables (all combinations of rows).

2e. Types of Normalization

- **1NF**: Atomic values, no repeating groups.
- **2NF**: 1NF + no partial dependency.
- **3NF**: 2NF + no transitive dependency.
- **BCNF**: Stronger form of 3NF, every determinant is a candidate key.

📊 SECTION B – SQL on `warehouse_employees` (40 Marks)

3a. Box code with city

```
SELECT b.code, w.location
FROM boxes b
JOIN warehouses w ON b.warehouse = w.code;
```

3b. Warehouse codes with number of boxes

```
SELECT w.code, COUNT(b.code) AS num_boxes
FROM warehouses w
LEFT JOIN boxes b ON w.code = b.warehouse
GROUP BY w.code;
```

3c. Saturated warehouses

```
SELECT w.code
FROM warehouses w
JOIN boxes b ON w.code = b.warehouse
GROUP BY w.code, w.capacity
HAVING COUNT(b.code) > w.capacity;
```

3d. Boxes in Chicago (subquery)

```
SELECT code
FROM boxes
WHERE warehouse IN (
    SELECT code FROM warehouses WHERE location = 'Chicago'
);
```

3e. Employees with department data

```
SELECT e.*, d.*
FROM employees e
JOIN departments d ON e.department = d.code;
```

3f. Employee name + department name & budget

```
SELECT e.name, e.lastname, d.name AS dept_name, d.budget
FROM employees e
JOIN departments d ON e.department = d.code;
```

3g. Employees in departments with budget > 60000

```
SELECT e.name, e.lastname
FROM employees e
JOIN departments d ON e.department = d.code
WHERE d.budget > 60000;
```

3h. Departments with >2 employees (subquery)

```
SELECT name
FROM departments
WHERE code IN (
    SELECT department
    FROM employees
```

```
    GROUP BY department
    HAVING COUNT(*) > 2
);
```

3i. Employees in department with 2nd lowest budget

```
SELECT e.name, e.lastname
FROM employees e
JOIN departments d ON e.department = d.code
WHERE d.budget = (
    SELECT DISTINCT budget
    FROM departments
    ORDER BY budget ASC
    LIMIT 1 OFFSET 1
);
```

3j. Number of employees per department

```
SELECT department, COUNT(*) AS num_employees
FROM employees
GROUP BY department;
```

🎵 SECTION C – SQL on `sales` schema (40 Marks)

(Schema not fully given, but based on standard Chinook/Media DB)

4a. Customers with similar first name

```
SELECT c1.customerid, c1.firstname, c2.customerid, c2.firstname
FROM customer c1
JOIN customer c2 ON c1.firstname = c2.firstname AND c1.customerid
<> c2.customerid;
```

4b. Customer name, invoice date & previous invoice dates

```
SELECT c.customerid, c.firstname, i.invoicedate,
       LAG(i.invoicedate) OVER (PARTITION BY c.customerid ORDER
BY i.invoicedate) AS prev_invoice
FROM customer c
JOIN invoice i ON c.customerid = i.customerid;
```

4c. Total tracks per genre (unit price >0.9, genre starts with 's')

```
SELECT g.name, COUNT(t.trackid) AS total_tracks
FROM track t
```

```sql
JOIN genre g ON t.genreid = g.genreid
WHERE t.unitprice > 0.9 AND g.name LIKE 's%'
GROUP BY g.name;
```

4d. Rank customers by unique albums purchased

```sql
SELECT c.customerid, c.firstname, COUNT(DISTINCT t.albumid) AS
unique_albums,
       RANK() OVER (ORDER BY COUNT(DISTINCT t.albumid) DESC) AS
rank_customers
FROM customer c
JOIN invoice i ON c.customerid = i.customerid
JOIN invoiceline il ON i.invoiceid = il.invoiceid
JOIN track t ON il.trackid = t.trackid
GROUP BY c.customerid, c.firstname;
```

4e. Total tracks billed per composer (descending)

```sql
SELECT t.composer, COUNT(il.trackid) AS total_tracks
FROM track t
JOIN invoiceline il ON t.trackid = il.trackid
GROUP BY t.composer
ORDER BY total_tracks DESC;
```

4f. Virtual table: customerid, billing city, customer city, albums billed per order

```sql
SELECT c.customerid, i.billingcity, c.city AS customer_city,
       COUNT(DISTINCT t.albumid) AS albums_billed
FROM customer c
JOIN invoice i ON c.customerid = i.customerid
JOIN invoiceline il ON i.invoiceid = il.invoiceid
JOIN track t ON il.trackid = t.trackid
GROUP BY c.customerid, i.invoiceid, i.billingcity, c.city;
```

✅ That covers **all questions neatly** with explanations and SQL code.

Would you like me to also **run these queries on your provided warehouse_employees dataset** and show the actual outputs (tables of results)? That way you'll see the exact answers for Section B.