DBS Assignment 1 Access control activities

🟥 **fd598ead-dc52-483f-9e08-88fd9fea97ba.pdf**
PDF

exlain this pdf, and answer me step by step until i say next answer

Great 👍 We'll go **step by step** as you asked.

✅ Step 1 – Understanding the Problem (Activity 1 Overview)

From the PDF fd598ead-dc52-483f-9e08-88fd9fe… , the system is a **sales order database** with:

📌 Three Tables:

1. **CUSTOMER**
    - Primary Key: *(name, street_number)*
2. **ORDER**
    - Primary Key: *(date_of_receipt, foreign key from CUSTOMER)*
3. **ORDER_ITEM**
    - Primary Key: *(item_name, foreign key from ORDER)*

And the relationships are:

```
CUSTOMER  →  ORDER  →  ORDER_ITEM
(1 to many)   (1 to many)
```

That means:

- One customer can have many orders
- One order can have many items

🎯 Security Requirements

We also have **role-based access requirements**:

👩 Tracey (Supervisor)

- Can **see everything**
- Can **change everything**

👨 Bill, 👩 Sheila, 👨 Govind

- Do routine work
- Cannot create new customers
- Only **Tracey and Govind** can handle orders > £1000

👤 Temporary Staff

- Can process orders
- ❌ Cannot see customer details

🔍 What We Must Do First

Before writing security controls, we must:

1. Design the schema correctly
2. Ensure:
    - Primary Keys
    - Foreign Keys
    - Correct relationships

✅ Step 1A – Designing the Tables (Conceptual Schema)

🧾 CUSTOMER

Field	Type
name	VARCHAR
street_number	VARCHAR
city	VARCHAR
phone	VARCHAR

Primary Key: (name, street_number)

🧾 ORDER

Field	Type
order_id	INT
date_of_receipt	DATE
customer_name	VARCHAR
customer_street_number	VARCHAR
total_amount	DECIMAL

Foreign Key:
(customer_name, customer_street_number) → CUSTOMER

🧾 ORDER_ITEM

Field	Type
item_name	VARCHAR
order_id	INT
quantity	INT
price	DECIMAL

Foreign Key:
order_id → ORDER

🧠 Important Design Observation

Even though the problem says:

> ORDER is keyed on date-of-receipt and the foreign key from CUSTOMER

In real databases, this is dangerous (dates are not unique).

👉 So we introduce `order_id` as a **safe primary key**, which is best practice.

✅ Now Step 1B – SQL Schema Script

Here is the clean SQL:

```sql
CREATE TABLE CUSTOMER (
    name VARCHAR(50),
    street_number VARCHAR(10),
    city VARCHAR(50),
    phone VARCHAR(15),
    PRIMARY KEY (name, street_number)
);

CREATE TABLE ORDERS (
    order_id INT PRIMARY KEY,
    date_of_receipt DATE,
    customer_name VARCHAR(50),
    customer_street_number VARCHAR(10),
    total_amount DECIMAL(10,2),
    FOREIGN KEY (customer_name, customer_street_number)
        REFERENCES CUSTOMER(name, street_number)
        ON DELETE CASCADE
);

CREATE TABLE ORDER_ITEM (
    item_name VARCHAR(50),
    order_id INT,
    quantity INT,
    price DECIMAL(10,2),
    PRIMARY KEY (item_name, order_id),
    FOREIGN KEY (order_id)
        REFERENCES ORDERS(order_id)
        ON DELETE CASCADE
);
```

🔎 Why `ON DELETE CASCADE`?

If:

- A customer is deleted → Their orders delete
- An order is deleted → Order items delete

This directly relates to **Activity 7 question about cascade behavior** later 👀

✅ Summary of Step 1

✔️ Three tables created
✔️ Primary keys defined
✔️ Foreign keys defined
✔️ 1-to-many relationships implemented
✔️ Delete rules added

Say **“next answer”** and we will proceed to:

👉 **Step 2 – Populating the Database (Activity 2)**

Sources

Activity 2

✅ Activity 2 – Populating the Database

(As per the PDF instructions fd598ead-dc52-483f-9e08-88fd9fe…)

This is a **security exercise**, so the data:

- Does NOT need to be realistic
- Must be **clear and meaningful**
- Must respect **foreign key relationships**

🎯 Goal of Activity 2

We now:

1. Insert sample customers
2. Insert sample orders
3. Insert sample order items
4. Include:
    - Some normal orders (< £1000)
    - Some large orders (> £1000) ⚠️ for testing later

This will help us test:

- High-value order restriction
- Delete cascading
- Role-based access

✅ Step 2A – Insert Customers

```
INSERT INTO CUSTOMER VALUES
('Alice', '101', 'London', '9876543210'),
('Bob', '202', 'Manchester', '9123456780'),
('Charlie', '303', 'Birmingham', '9988776655');
```

✔️ Clear names
✔️ Different cities
✔️ Simple recognizable data

✅ Step 2B – Insert Orders

We now insert:

- Some small orders

- Some large orders (>1000)

```
INSERT INTO ORDERS VALUES
(1, '2026-02-20', 'Alice', '101', 450.00),    -- Normal order
(2, '2026-02-21', 'Alice', '101', 1500.00),   -- Large order
(3, '2026-02-22', 'Bob', '202', 750.00),      -- Normal order
(4, '2026-02-23', 'Charlie', '303', 2500.00); -- Large order
```

Now we have:

Order	Amount	Type
1	450	Normal
2	1500	Large ⚠️
3	750	Normal
4	2500	Large ⚠️

These will be important for **privilege testing later**.

✅ Step 2C – Insert Order Items

```
INSERT INTO ORDER_ITEM VALUES
('Keyboard', 1, 2, 100.00),
('Mouse', 1, 1, 50.00),
('Laptop', 2, 1, 1500.00),
('Printer', 3, 1, 750.00),
('Coffee Machine', 4, 5, 500.00);
```

✔️ Order 2 and 4 relate to large orders
✔️ Coffee Machine entry will help in Activity 7 scenario 👀

🧠 Why This Data Is Good for Security Testing

It allows us to test:

- Normal staff processing small orders
- Blocking large orders for some users
- Deletion cascade
- Temporary staff restrictions
- Insert attempts into high-value orders

✅ Result After Activity 2

Database now contains:

- 3 Customers
- 4 Orders
- 5 Order Items

- 2 High-value orders (>£1000)

The database is now ready for:

👉 **Activity 3 – Security Analysis (User-role matrix + GRANT statements)**

Say **“next answer”** to proceed to Activity 3.

Sources

next

✅ Activity 3 – Analysing the Problem (Security Design)

(As required in the PDF fd598ead-dc52-483f-9e08-88fd9fe…)

Now we convert the textual requirements into a **clear role–resource access matrix**, and then into SQL **`GRANT`** statements.

🎯 Step 3A – Identify Roles

Treat names as **database roles**:

- **`TRACEY`**
- **`BILL`**
- **`SHEILA`**
- **`GOVIND`**
- **`TEMP_STAFF`**

🎯 Step 3B – Identify Database Resources

Resources:

1. **`CUSTOMER`** table
2. **`ORDERS`** table
3. **`ORDER_ITEM`** table
4. (Later) Views for restricted access

🎯 Step 3C – Build Access Matrix

🔐 Access Requirements from Problem

Role	CUSTOMER	ORDERS	ORDER_ITEM	Special Rules
Tracey	SELECT, INSERT, UPDATE, DELETE	ALL	ALL	Full control
Bill	SELECT, UPDATE	SELECT, INSERT, UPDATE	SELECT, INSERT, UPDATE	❌ Cannot create customers
Sheila	Same as Bill	Same	Same	Same
Govind	Same as Bill + can handle large orders	Same	Same	Can process >£1000
Temp Staff	❌ No SELECT	INSERT, UPDATE	INSERT, UPDATE	Cannot see customer details

---

🎯 Important Special Condition

🔴 Orders > £1000

Only:

- Tracey
- Govind

Can handle these.

This **cannot be enforced using simple GRANT**.

👉 It requires:

- A VIEW
- Or a TRIGGER
- Or CHECK constraint with procedure logic

We will use a **VIEW-based solution** (clean and exam-friendly).

✅ Step 3D – Security Strategy

We will:

1. Create Roles
2. Revoke default privileges
3. Grant carefully
4. Create views:
    - `NORMAL_ORDERS` (≤ 1000)

- `CUSTOMER_HIDDEN_VIEW` (for temp staff)

✅ Step 3E – SQL Role Creation

```
CREATE ROLE TRACEY;
CREATE ROLE BILL;
CREATE ROLE SHEILA;
CREATE ROLE GOVIND;
CREATE ROLE TEMP_STAFF;
```

✅ Step 3F – Grant Privileges

👩 Tracey – Full Control

```
GRANT ALL PRIVILEGES ON CUSTOMER TO TRACEY;
GRANT ALL PRIVILEGES ON ORDERS TO TRACEY;
GRANT ALL PRIVILEGES ON ORDER_ITEM TO TRACEY;
```

👨 Bill & 👩 Sheila

Cannot create customers.

```
GRANT SELECT, UPDATE ON CUSTOMER TO BILL;
GRANT SELECT, UPDATE ON CUSTOMER TO SHEILA;

GRANT SELECT, INSERT, UPDATE ON ORDERS TO BILL, SHEILA;
GRANT SELECT, INSERT, UPDATE ON ORDER_ITEM TO BILL, SHEILA;
```

Notice:

- No INSERT on CUSTOMER

👨 Govind

Same as above, but will later receive access to large orders.

```
GRANT SELECT, UPDATE ON CUSTOMER TO GOVIND;
GRANT SELECT, INSERT, UPDATE ON ORDERS TO GOVIND;
GRANT SELECT, INSERT, UPDATE ON ORDER_ITEM TO GOVIND;
```

👤 Temporary Staff

Cannot see CUSTOMER table.

```
GRANT INSERT, UPDATE ON ORDERS TO TEMP_STAFF;
GRANT INSERT, UPDATE ON ORDER_ITEM TO TEMP_STAFF;
```

No SELECT on CUSTOMER.

🧠 But Wait...

Right now:

Bill, Sheila, Govind can still modify large orders.

We haven't enforced:

> Orders above £1000 only for Tracey and Govind

That is coming in the next step using **Views**.

✅ Summary of Activity 3 So Far

✔️ Roles identified
✔️ Access matrix developed
✔️ SQL GRANT drafted
✔️ High-value order issue pending enforcement

Next we implement:

👉 **High-Value Order Control using Views (critical security logic)**

Say **"next answer"** to continue.

Sources

next

✅ Activity 3 (Continued) – Enforcing the £1000 Restriction

From the problem statement fd598ead-dc52-483f-9e08-88fd9fe… :

> "Make sure that large orders (above £1000) cannot be handled by anyone except Tracey and Govind."

Simple GRANT statements cannot enforce value-based restrictions.

So we use **Views** to restrict access.

🎯 Strategy

We split the ORDERS table logically:

1. **NORMAL_ORDERS** → total_amount ≤ 1000
   → Accessible by Bill, Sheila

2. **ALL_ORDERS** → Full access
   → Accessible only by Tracey and Govind

We also REVOKE direct access to the base table from restricted users.

✅ Step 1 – Create Restricted View

```
CREATE VIEW NORMAL_ORDERS AS
SELECT *
FROM ORDERS
WHERE total_amount <= 1000
WITH CHECK OPTION;
```

🔎 Why `WITH CHECK OPTION`?

It ensures:

- Users cannot INSERT or UPDATE rows that violate the condition.
- So Bill/Sheila cannot insert an order > 1000 through this view.

This is the key control.

✅ Step 2 – Remove Direct Table Access

We now revoke direct ORDERS table access from Bill and Sheila.

```
REVOKE ALL ON ORDERS FROM BILL;
REVOKE ALL ON ORDERS FROM SHEILA;
```

Now they cannot bypass the view.

✅ Step 3 – Grant Access to View Instead

```
GRANT SELECT, INSERT, UPDATE ON NORMAL_ORDERS TO BILL;
GRANT SELECT, INSERT, UPDATE ON NORMAL_ORDERS TO SHEILA;
```

Now:

✔️ They can see small orders
✔️ They can insert small orders
❌ They cannot touch orders > 1000

✅ Step 4 – Give Full ORDERS Access to Tracey and Govind

```
GRANT SELECT, INSERT, UPDATE, DELETE ON ORDERS TO TRACEY;
GRANT SELECT, INSERT, UPDATE, DELETE ON ORDERS TO GOVIND;
```

They have full control.

🔥 Now the Rule is Properly Enforced

Role	Small Orders	Large Orders
Tracey	✅	✅
Govind	✅	✅
Bill	✅	❌
Sheila	✅	❌
Temp	Depends on view design	❌

🎯 Now Fix Temporary Staff Rule

> “Temporary member can process orders but cannot see customer details.”

We must prevent SELECT on CUSTOMER.

They already have no SELECT privilege.

To make it cleaner, we create a processing view.

✅ Step 5 – Create Safe Processing View

```
CREATE VIEW ORDER_PROCESSING_VIEW AS
SELECT order_id, date_of_receipt, total_amount
FROM ORDERS;
```

Notice:

- No customer_name
- No street number

Now grant:

```
GRANT INSERT, UPDATE, SELECT ON ORDER_PROCESSING_VIEW TO TEMP_STAFF;
```

This allows:

✔️ Processing
✔️ But no customer visibility

🧠 Security Status After This

✔️ Value-based restriction enforced
✔️ Customer confidentiality enforced
✔️ Direct table bypass removed
✔️ Access is strictly role-controlled

✅ Activity 3 Completed

We now have:

- Roles
- Grants
- Views
- Enforcement logic
- Proper restrictions

Next:

👉 **Activity 4 – Executing the Security Script & What Happens Internally**

Say **"next answer"** to continue.

Sources

next

✅ Activity 4 – Executing the Security Script

As stated in the PDF fd598ead-dc52-483f-9e08-88fd9fe… :

> "Execute the security script (if you have a DBMS environment). Otherwise, get a friend to critique."

Here we assume we are executing it in a real DBMS (e.g., MySQL / PostgreSQL / Oracle style system).

🎯 What Happens When We Execute the Script?

We execute in this order:

1. Create Tables
2. Insert Data
3. Create Roles
4. Revoke Unwanted Privileges
5. Create Views
6. Grant Privileges

🔎 Important: Default Behavior in SQL Systems

When tables are created:

- The creator (usually DBA / Owner) automatically gets ALL privileges.
- Other users get nothing unless granted.
- PUBLIC privileges may exist depending on DBMS.

So execution must ensure:

```
REVOKE ALL ON CUSTOMER FROM PUBLIC;
REVOKE ALL ON ORDERS FROM PUBLIC;
REVOKE ALL ON ORDER_ITEM FROM PUBLIC;
```

This prevents accidental leaks.

🧠 Internal DBMS Behavior After GRANT

When we run:

```
GRANT SELECT ON NORMAL_ORDERS TO BILL;
```

The DBMS:

- Records this in system catalog tables
- Associates the role BILL with that object
- Enforces privilege check during query parsing

So when Bill runs:

```
SELECT * FROM ORDERS;
```

The DBMS checks:

❌ No privilege → Access denied

When Bill runs:

```
SELECT * FROM NORMAL_ORDERS;
```

✔️ Allowed

🎯 If Execution Fails – Possible Reasons

1. Foreign key constraint errors
2. View created before base table
3. Attempting to REVOKE privilege never granted
4. Missing CHECK OPTION support (older DBMS)

🔥 What Should Be Verified After Execution?

You must confirm:

✔️ Bill cannot insert customers
✔️ Sheila cannot insert customers
✔️ Bill cannot access orders > 1000
✔️ Temp cannot view CUSTOMER table
✔️ Tracey can modify everything
✔️ Govind can handle large orders

⚠️ Common Mistakes in This Activity

Students often:

- Forget to revoke direct table access
- Forget WITH CHECK OPTION
- Forget to prevent PUBLIC access
- Grant too much privilege accidentally
- Give SYSADMIN level permissions

These errors will appear in Activity 7 👀

✅ Outcome of Activity 4

If correctly executed:

- Access control is active
- Privileges are enforced dynamically
- Views restrict sensitive operations
- Value-based access control is functional

Next:

👉 **Activity 5 – Testing the Access Control Mechanism Thoroughly**

This is where we simulate what Bill, Sheila, Govind, Temp and Tracey try to do.

Say **“next answer”** to continue.

Sources

next

✅ Activity 5 – Testing the Access Control Mechanism

As required in the PDF fd598ead-dc52-483f-9e08-88fd9fe… :

> “Formulate SELECT statements from the problem specification and issue SELECT statements to check that they have been correctly implemented.”

Now we simulate **real user behavior**.

We log in as each role and test.

🔍 Test 1 – Bill Tries to Create Customer

```
INSERT INTO CUSTOMER VALUES ('David', '404', 'Leeds', '9000000000');
```

✅ Expected Result:

❌ Access Denied

Reason:
Bill does not have INSERT on CUSTOMER.

🔍 Test 2 – Bill Views Small Orders

```
SELECT * FROM NORMAL_ORDERS;
```

✅ Expected Result:

✔️ Only orders ≤ 1000 displayed

From our data:

- Order 1 (450)
- Order 3 (750)

Correct behavior.

🔍 Test 3 – Bill Tries to View Large Orders Directly

```
SELECT * FROM ORDERS;
```

✅ Expected Result:

❌ Access Denied

Because:

- Direct ORDERS access revoked.

🔍 Test 4 – Bill Attempts to Insert Large Order

```
INSERT INTO NORMAL_ORDERS
VALUES (10, '2026-02-25', 'Alice', '101', 2000);
```

✅ Expected Result:

❌ Rejected

Reason:
`WITH CHECK OPTION` prevents inserting rows > 1000.

🔍 Test 5 – Govind Inserts Large Order

```
INSERT INTO ORDERS
VALUES (11, '2026-02-25', 'Bob', '202', 3000);
```

✅ Expected Result:

✔️ Success

Govind has full ORDERS privileges.

🔍 Test 6 – Temporary Staff Tries to View Customer Data

```
SELECT * FROM CUSTOMER;
```

✅ Expected Result:

❌ Access Denied

Correct enforcement of privacy rule.

🔍 Test 7 – Temp Staff Uses Processing View

```
SELECT * FROM ORDER_PROCESSING_VIEW;
```

✅ Expected Result:

✔️ Allowed
✔️ No customer name shown

Confidentiality preserved.

🔍 Test 8 – Tracey Deletes a Customer

```
DELETE FROM CUSTOMER
WHERE name = 'Alice' AND street_number = '101';
```

✅ Expected Result:

✔️ Success

And because of:

`ON DELETE CASCADE`

✔️ All Alice's orders deleted
✔️ All related order items deleted

This answers part of Activity 7 later 👀

🧠 How We Detect Violations

When unauthorized access is attempted:

- DBMS throws permission error
- Can be logged in:
  - Audit logs
  - Error logs
  - Security monitoring system

✅ If All These Behave Correctly

Then:

✔️ Privilege enforcement works
✔️ View-based restriction works
✔️ Value restriction works
✔️ Confidentiality rule works
✔️ Cascade behavior works

🔥 Activity 5 Completed

Now comes:

👉 **Activity 6 – Conclusion (Additional Security Measures + Strengths & Weaknesses Analysis)**

This is usually a theory-heavy evaluation section.

Say **"next answer"** to continue.

Sources

next

✅ Activity 6 – Conclusion

(As required in the PDF fd598ead-dc52-483f-9e08-88fd9fe…)

Now we must:

1. Suggest **additional SQL security measures**
2. Comment on **strengths**
3. Comment on **weaknesses**

🔐 Part 1 – Additional SQL Security Measures

Even though our design works, more security enhancements can be added:

✅ 1. Use Stored Procedures Instead of Direct Table Access

Instead of:

```
GRANT INSERT ON ORDERS TO BILL;
```

We could:

- Remove direct insert privilege
- Create a stored procedure `process_order()`
- Grant EXECUTE on that procedure

✔️ Better control
✔️ Business rules enforced centrally

✅ 2. Row-Level Security (If DBMS Supports It)

Modern DBMSs allow:

```
CREATE POLICY ...
```

This enforces:

- Value-based conditions automatically
- Stronger enforcement than views

Better than `WITH CHECK OPTION`.

✅ 3. Audit Logging

Enable:

- Login audit

- Failed access logging
- Data modification logging

This would answer Activity 7 detection question.

✅ 4. Password Policies

Enforce:

- Strong passwords
- Password expiry
- Account lockout after failed attempts

This limits misuse by staff.

✅ 5. Remove ON DELETE CASCADE (Optional Security Choice)

Instead of cascade:

- Use **`ON DELETE RESTRICT`**

Prevents accidental mass deletion.

💪 Strengths of Our Security Design

✔️ Role-based access control (RBAC)
✔️ Value-based restriction via view
✔️ Proper privilege separation
✔️ No unnecessary SYSADMIN rights
✔️ Privacy preserved for temporary staff
✔️ Cascade relationships clearly defined

⚠️ Weaknesses of Our Approach

1. View-based restriction can sometimes be bypassed by direct table access if mistakenly granted.
2. Value restriction depends on CHECK OPTION — not very flexible.
3. ON DELETE CASCADE could allow large accidental data loss.
4. Tracey has full privileges — risk of internal abuse.
5. No multi-level approval system for high-value orders.

🧠 Overall Evaluation

The system implements:

- Discretionary Access Control (DAC)
- Basic Role-Based Access Control (RBAC)
- Partial Mandatory Access (through value condition)

Security level: Moderate
Not enterprise-grade.

✅ Activity 6 Completed

Now comes the final and most interesting part:

👉 **Activity 7 – The Tracey Scenario (Privilege Abuse Testing)**

This is where many students realize they gave too much power 😄

Say **"next answer"** to continue.

Sources

next

✅ Activity 7 – The Tracey Scenario

(From the PDF fd598ead-dc52-483f-9e08-88fd9fe…)

Now we analyse what happens after everything is implemented.

This section checks whether your security design is **correct or dangerously over-permissive**.

🔍 Question 1

❓ Tracey tried to delete the CUSTOMER table. Did she succeed?

🔎 Analysis

We gave Tracey:

```
GRANT ALL PRIVILEGES ON CUSTOMER TO TRACEY;
```

⚠️ Important:

`ALL PRIVILEGES` usually means:

- SELECT

- INSERT
- UPDATE
- DELETE
- REFERENCES

It does **NOT automatically include DROP TABLE** unless:

- She is table owner
- Or has DBA/SYSADMIN role

✅ Correct Answer:

No, she should NOT succeed (if implemented correctly).

Unless:
You accidentally made her schema owner or DBA.

🔍 Question 2

❓ If she succeeded, why?

Because you might have:

- Created tables under Tracey's login
- Given her SYSADMIN role
- Granted **`DROP`** privilege explicitly
- Granted **`WITH GRANT OPTION`** carelessly

That would mean:

⚠️ You unintentionally gave her administrative privileges.

🔍 Question 3

❓ She tried to delete some customers. Did she succeed? Did deletes cascade?

We granted:

```
GRANT DELETE ON CUSTOMER TO TRACEY;
```

And table has:

```
ON DELETE CASCADE
```

✅ Result:

✔️ She can delete customers
✔️ Related ORDERS deleted
✔️ Related ORDER_ITEMS deleted

So YES — deletion succeeds and cascades.

This is expected.

But it's also risky.

🔍 Question 4

❓ She tried to insert a line in all orders over £1000 for 500 coffee machines. Did she succeed?

She has:

✔️ Full ORDERS access
✔️ Full ORDER_ITEM access

So yes, she can insert:

```
INSERT INTO ORDER_ITEM VALUES ('Coffee Machine', 2, 500, 500);
```

No restriction blocks her.

✅ So YES, she succeeds.

Unless you implemented:

- Business logic constraint
- Quantity restriction trigger

Which we did not.

🔍 Question 5

❓ How was the problem detected?

Through:

✔️ Audit logs
✔️ Transaction logs
✔️ Unexpected inventory change
✔️ Report anomalies
✔️ Sales reporting mismatch

Without logging?
❌ It may go unnoticed.

🔍 Question 6

❓ She tried to change her password. Did she succeed?

In most DBMS:

- Users CAN change their own password.

Unless restricted by DBA.

So:

✅ Yes, she likely succeeds.

But this does not increase privilege.

🔍 Question 7

❓ How much privilege can any one individual ever be given?

This is a theory answer.

🔐 Principle of Least Privilege:

No individual should be given:

- Structural privileges (DROP, ALTER, CREATE)
- System-level privileges
- Both business and admin control

Separation of Duties must apply.

Best practice:

Role	Capability
DBA	Structure management
Supervisor	Business data control
Staff	Limited operational rights

Never combine DBA + Supervisor in one person.

🎯 Final Evaluation of Tracey

Tracey has:

✔️ Full business access
❌ No structural control

❌ No SYSADMIN powers

If you implemented correctly:
Your system is secure.

If she could drop tables:
You made a serious design mistake.

🏁 Entire Exercise Summary

You have implemented:

✔️ Database schema
✔️ Test data
✔️ Role-based access
✔️ Value-based restriction
✔️ View-based isolation
✔️ Confidentiality control
✔️ Cascade behavior testing
✔️ Abuse analysis

If you'd like, I can now:

- Give you a **clean exam-ready answer summary**
- Or provide a **complete combined SQL script**
- Or explain weakest point in whole design deeply
- Or simulate viva questions

Tell me what you want next.

Sources