

INFS1200 — Week 12 Notes

Table of contents

Database Security	1
Today's outline	1
Why database security matters	2
Database control measures	3
Access control mechanisms	3
Discretionary Access Control (DAC)	4
Question 1 — Discretionary access control	6
Question 2 — Discretionary access control (propagation)	6
Mandatory Access Control (MAC)	7
Role-Based Access Control (RBAC)	9
Other control measures	9
SQL injection	10
Summary	12
Tutorial 5.1: Database Security	12
Section A — Discretionary Access Control	12
Section B — Mandatory Access Control	14
Bonus — Completing a database audit	18

Database Security

Module 5 — the final module. Covers threats to database security, the control measures used to defend against them, and SQL injection.

Today's outline

- Threats to database security

- Access control mechanisms — discretionary, mandatory, role-based
- Other control measures — inference control, flow control, encryption
- SQL injection

Why database security matters

Some of the largest data breaches on record (Yahoo, First American, Facebook, Marriott, MongoDB, Equifax, and others) collectively exposed billions of records — via hacking, poor security practice, or simple misconfiguration. Database security matters for:

- **Legal/policy reasons** — e.g. government policy protecting access to individuals' data (medical records, financial ratings).
- **Ethical reasons** — e.g. controlling who can see employee salary packages or student grades.
- **Technical reasons** — e.g. protection from malware, performance overloading.

Threats — the CIA principle

- **Loss of confidentiality** — unauthorized disclosure of confidential information.
- **Loss of integrity** — improper modification of information.
- **Loss of availability** — a legitimate user cannot access data objects.

Privacy vs. security

- **Security** concerns *how access to data is controlled* — availability for use (with integrity) and permitted access.
- **Privacy** concerns *how data can be used* — an individual's ability to control the terms under which their sensitive data is acquired and used (preventing storage of sensitive data; ensuring appropriate/authorized use of it).
- Security is a **required building block** for privacy, but they're distinct: privacy concerns both directly accessible information *and* information that can be **inferred** from accessible data.

Sensitive data is often double-edged — patient data is private, but also valuable for biomedical/public-health research; audio from CCTV can capture private conversations, but also evidence of criminal activity. This creates a **utility/privacy tradeoff**: protect all sensitive data, while making as much nonsensitive data available as possible.

Database control measures

Four broad categories:

1. **Access control** — provide access only to users with the right access authority.
2. **Inference control** — ensure information about individuals cannot be *derived* even indirectly (applies to statistical databases).
3. **Flow control** — prevent information from flowing to unauthorized users.
4. **Data encryption** — protect sensitive data in transit.

Access control mechanisms

Database access control has two components:

- **Authentication** — verifying the identity of whoever is accessing the database.
- **Authorization** — determining whether an authenticated user should be allowed to execute the transaction they're attempting.

Basic authentication security functions

- **User accounts** — users log in with an assigned username/password.
- **Login session** — a sequence of database operations by one user, recorded in the **system log** (also used for recovery, not just security).
- **Database audit** — reviewing the log to examine all accesses and operations applied during a period, to identify potential privacy breaches or unauthorised modifications. A log typically records who ran what operation, when, and (for modifications) the before/after values.

Three authorization mechanisms

1. **Discretionary Access Control (DAC)** — grant/revoke privileges to users.
2. **Mandatory Access Control (MAC)** — classify data and users into security classes, and implement a security policy across them.
3. **Role-Based Access Control (RBAC)** — assign users to roles, then apply DAC or MAC at the role level.

Discretionary Access Control (DAC)

Based on **granting** and **revoking** privileges.

```
CREATE USER <username> IDENTIFIED BY "<password>";

GRANT privilegeName
ON      objectName
TO      {userName | PUBLIC | roleName}
[WITH GRANT OPTION];

REVOKE privilegeName
ON      objectName
FROM    {userName | PUBLIC | roleName};
```

Two levels of privileges

- **Account level** — privileges specified per account, independent of any particular relation: CREATE SCHEMA/TABLE/VIEW, ALTER/DROP TABLE, modification (INSERT/DELETE/UPDATE) and SELECT privileges.
- **Relation level** — privileges for a specific relation or view (can also be specified at the *attribute* level): **select** (read), **modification** (insert/delete/update), and **references** privileges. The REFERENCES privilege grants permission to create a *foreign key* reference to the specified table.

```
CREATE SCHEMA AUTHORIZATION oe
CREATE TABLE Product (color VARCHAR2(10) PRIMARY KEY, quantity NUMBER)
CREATE VIEW RedProduct AS
    SELECT color, quantity FROM Product WHERE color = 'RED'
GRANT select ON RedProduct TO hr;
```

This single statement creates a schema **oe**, creates **Product**, creates the view **RedProduct**, and grants **hr** the **SELECT** privilege on **RedProduct**.

Access matrix model

Each relation **R** is assigned an **owner** account. Owners can grant select/modification/references privileges to other users on any owned relation. An access matrix $M(i, j)$ captures the privilege(s) user **i** has on object **j**:

	Jake	Aiden	Nelly	Sham
Customer	READ	READ	MODIFY	—
CustOrder	—	—	READ	—
CustOrder.amt	—	READ	READ	—
Supp.status	—	READ	MODIFY	REFERENCE
Supp.address	—	READ	MODIFY	MODIFY

Revoking and propagating privileges

REVOKE cancels a privilege — useful for granting something temporarily. If account A grants a privilege to B **WITH GRANT OPTION**, B can then grant it onward to other accounts, without A’s direct knowledge — the DBMS must track this *chain of dependency* to support revocation correctly.

```
REVOKE privilegeName ON objectName FROM userName [RESTRICT | CASCADE];
```

- **CASCADE** — revokes the privilege *and* any dependent privileges that were granted as a result of it (following the chain onward).
- **RESTRICT** — refuses the revoke (returns an error) if the privilege has already been passed on to someone else.

A user’s authorization is valid **iff there is a path from the root of the authorization graph down to that user’s node**. So if u1 grants to u2, who grants to u3: revoking $u1 \rightarrow u2$ also cuts off u3 (no path remains); but revoking a *different* branch, or u2’s authorization being revoked by someone else entirely, doesn’t necessarily touch u1 (the grantor’s own authorization is independent of what they’ve granted downstream).

Specifying privileges through views

If owner A wants to give account B a *limited* capability to **SELECT** from a relation — e.g. only some columns, or only rows meeting a condition — A can create a view and grant access to the view instead of the base table:

```
-- Employee [ssn, name, dob, address, sex, salary, mgrSSN, dNum]
CREATE VIEW A3EMPLOYEE AS
  SELECT name, dob, address
  FROM   EMPLOYEE
  WHERE  dNum = 5;

GRANT SELECT ON A3EMPLOYEE TO A3;
```

A3 can now only ever see `name/dob/address` for department 5 employees — never salary, ssn, or other departments' staff.

Question 1 — Discretionary access control

Which statement removes a privilege from a user?

- A. Remove update on department from Amir
- B. Revoke update on employee from Amir
- C. Delete select on department from Raj
- D. Grant update on employee from Amir

i Solution

B. REVOKE is the actual SQL keyword for removing a privilege. REMOVE and DELETE aren't valid DCL keywords for this purpose, and GRANT *adds* a privilege rather than removing one.

Question 2 — Discretionary access control (propagation)

u1 grants authorization to u2, who in turn grants it to u3. Which is correct?

- A. If u1 revokes from u3, u2's authorization is revoked.
- B. If u1 revokes from u2, u3's authorization is also revoked.
- C. If u2's authorization is revoked, u1's authorization is also automatically revoked.

i Solution

B. A user has an authorization iff there's a path from the root of the authorization graph down to their node. $u1 \rightarrow u2 \rightarrow u3$ — revoking u1's grant to u2 removes the only path down to u3 as well, so u3 loses authorization too. Revoking from u3 directly (A) has no effect on u2's own (separately-granted) authorization. u2 losing authorization (C) doesn't propagate *upward* to u1 — u1 is the root/grantor, not a recipient.

Worked exercise — GRANT/REVOKE chains

Four users A1, A2, A3, A4. DBA: GRANT CREATETAB TO A1. A1 creates Employee and Department, then runs:

```
GRANT INSERT, DELETE ON Employee, Department TO A2;  
GRANT SELECT ON Employee, Department TO A3 WITH GRANT OPTION;
```

Q1: *Can A3 execute GRANT SELECT ON Employee TO A4?*

i Solution

Yes — A3 was granted `SELECT WITH GRANT OPTION`, so A3 can pass that privilege on to A4.

Q2: *If A1 then runs `REVOKE SELECT ON Employee FROM A3`, does A4 still have `SELECT` on `Employee`?*

i Solution

No — A4's privilege is automatically revoked too, by propagation (there's no longer a path from the root down to A4).

Q3: *What type of access control is this?*

i Solution

Discretionary access control — privileges are granted/revoked at the *discretion* of individual account owners, rather than being fixed by a system-wide security classification (MAC) or role hierarchy (RBAC).

Mandatory Access Control (MAC)

An additional policy that classifies **both data and users** into security classes, for **multilevel security**. Typical classes (low to high): **Unclassified (U)** < **Confidential (C)** < **Secret (S)** < **Top Secret (TS)**. Users are called **subjects**; data (table, tuple, or attribute) are **objects**.

The Bell-LaPadula model

- **Simple security property** (“no read up”, NRU) — a subject can't *read* an object with a higher sensitivity label than the subject's own. E.g. a user with clearance **U** cannot view a salary value classified **C**.
- **Star property** (“no write down”, NWD) — a subject can't *write* to an object with a *lower* sensitivity label than the subject's own — this prevents information flowing from a higher to a lower classification. E.g. a user with clearance **S** cannot insert a new tuple containing only **C**-classified information (that would let a lower-clearance user infer something about a higher-clearance operation via the write).

Each **tuple** gets a **Tuple Classification (TC)** — the highest classification of any of its attribute values.

(a) EMPLOYEE (original)

Name	Salary	JobPerformance	TC
Smith (U)	40000 (C)	Fair (S)	S
Brown (C)	80000 (S)	Good (C)	S

(b) EMPLOYEE, as seen by a Confidential-clearance user

Name	Salary	JobPerformance	TC
Smith (U)	40000 (C)	NULL (C)	C
Brown (C)	NULL (C)	Good (C)	C

(c) EMPLOYEE, as seen by an Unclassified-clearance user

Name	Salary	JobPerformance	TC
Smith (U)	NULL(U)	NULL (U)	U

A C-clearance user sees Smith’s JobPerformance as NULL (it’s actually S-classified — above their clearance) and Brown’s Salary as NULL (S-classified).

Inference attacks and polyinstantiation

Suppose a C-clearance user runs `UPDATE EMPLOYEE SET JobPerformance = 'Excellent' WHERE Name = 'Smith'` (working from view (b), where they only ever saw NULL for that field). The system **must not reject** this — rejecting it would let the user *infer* that a real, higher-classified value already exists (a **covert channel** leak). The solution is **polyinstantiation** — maintaining multiple tuples with the same key but different classifications:

(d) EMPLOYEE, polyinstantiated

Name	Salary	JobPerformance	TC
Smith (U)	40000 (C)	Fair (S)	S
Smith (U)	40000 (C)	Excellent (C)	C
Brown (C)	80000 (S)	Good (C)	S

Both the original S-classified fact and the new C-classified “fact” now coexist — a C-clearance user sees “Excellent”; an S-or-higher user still sees the original “Fair” (plus, depending on implementation, awareness that a lower-classified duplicate also exists).

DAC vs. MAC

	Discretionary	Mandatory
Flexibility	High — owners choose who gets what	Low — rigid, centrally imposed
Propagation control	None — DAC doesn't restrict how information, once accessible, is further shared	Strong — enforces classification rules on every read/write
Protection	Weaker	Stronger — prevents illegal information flow

Role-Based Access Control (RBAC)

Permissions are associated with **organisational roles**, and users are assigned to the appropriate role(s) — roles can then have DAC or MAC methods applied to them as a unit.

```
CREATE ROLE manager;
DROP ROLE manager;

GRANT ROLE full-time TO emp_typ1;
GRANT ROLE intern TO emp_typ2;

GRANT privilegeName ON objectName TO {userName | PUBLIC | roleName} [WITH GRANT
↪ OPTION];
REVOKE privilegeName ON objectName FROM {userName | PUBLIC | roleName};
```

A typical hierarchy: **users** → **groups** → **roles** → **privileges** (e.g. *alice* is in group *admin*, which maps to *admin_role*, which grants *ALL ON SERVER server1*). RBAC's flexibility and easier administration make it a popular choice for web-based applications.

Other control measures

Inference control

Statistical databases provide aggregate statistics about a population (e.g. for government statisticians or market researchers) without exposing individual-level confidential data — only **statistical queries** using aggregates (COUNT, SUM, MIN, MAX, AVG, standard deviation) are permitted:

```
SELECT COUNT(*) FROM PERSON WHERE <condition>;
SELECT AVG(Income) FROM PERSON WHERE <condition>;
```

The risk: a narrow enough **WHERE** condition can isolate a group small enough (even a single individual) that an “aggregate” query effectively discloses that person’s data. Mitigations:

- **k-anonymity** — enforce a minimum threshold on the number of tuples any query’s result can be based on.
- **Prohibit sequences of queries** that all refer to the same (or overlapping) population of tuples, which together could be combined to isolate an individual.
- **Differential privacy** — introduce carefully calibrated noise/ inaccuracy into results.

Flow control

Regulates the distribution of information among accessible objects, verifying information doesn’t flow — explicitly or implicitly — into *less* protected objects (this is exactly the Bell-LaPadula “no write down” idea, generalised). A **flow policy** specifies which channels information may move along — e.g. in a simple confidential (C) / nonconfidential (N) scheme, flow from C to N is prohibited. Example: an income tax computing service might be allowed to retain a customer’s address, but not their income/deductions data.

Data encryption

Encryption converts data into ciphertext, using an encryption algorithm and a key; recovering the original data requires the corresponding decryption key.

- **DES (Data Encryption Standard)** — developed by the US government for general public use.
- **AES (Advanced Encryption Standard)** — a newer, more difficult to crack standard.

SQL injection

SQL injection is one of the most common threats to a database system: an attacker injects string input through an (often web-facing) application in a way that changes or manipulates the resulting SQL statement to their advantage.

Methods

- **SQL manipulation** — changes an existing SQL command, e.g. adding conditions to a **WHERE** clause. Classic target: the login form.
- **Code injection** — adds *entirely new* SQL statements/commands by exploiting improper handling of untrusted input.
- **Function call injection** — inserts a database or OS function call into a vulnerable SQL statement, to manipulate data or make a privileged system call.

SQL manipulation example

A naive login check builds its query by directly concatenating user input:

```
SELECT * FROM Users WHERE Username = '<input>' AND Password = '<input>';
```

Normal login (Jack / Pass123) produces `WHERE Username = 'Jack' AND Password = 'Pass123'` — matches the stored row, access granted. But an attacker who knows the username Jack can enter the password field as:

```
' OR 'x'='x
```

giving the executed query:

```
SELECT * FROM Users WHERE username = 'Jack' AND (password = '' OR 'x'='x');
```

`'x'='x'` is always true, so the AND collapses to just `username = 'Jack'` — the row is returned regardless of the actual password, and access is granted.

Code injection example

Going further, appending a second statement via a semicolon:

```
wrongpass' OR 'x'='x'; drop table Users;
```

If the application naively executes whatever SQL string results (and the driver/DBMS allows statement-stacking), this doesn't just bypass login — it can **destroy the entire Users table**.

Risks associated with SQL injection

- **Database fingerprinting** — determining the type of database in use (to target version-specific exploits).
- **Denial of service** — denying service to valid users.
- **Bypassing authentication** — gaining access without valid credentials (as shown above).
- **Identifying injectable parameters** — learning about the backend structure through error messages/behaviour.
- **Executing remote commands** — running harmful commands remotely (e.g. via function call injection).
- **Performing privilege escalation** — upgrading the attacker's own access level.

Protection techniques

- **Bind variables (parameterized statements)** — pass user input as *bound parameters*, never concatenated directly into the SQL string. This is the primary defence: the database driver treats parameter values purely as data, never as executable SQL syntax — and as a bonus, it also improves performance (query plans can be cached and reused).
- **Filtering input (input validation)** — strip/escape characters (like unescaped quotes) that could otherwise be used to break out of a string literal and inject manipulation.
- **Function security** — restrict which standard and custom database functions are callable, to limit the blast radius of function call injection.

Summary

You should now be able to explain the threats to database security and the CIA principle; describe DAC, MAC, and RBAC and apply them (given grant/revoke commands, or a Bell-LaPadula classification scenario) to determine resulting access; and explain SQL injection, its risks, and how to defend against it. This completes the course content. See [week12-tutorial-5-1-database-security¹](#) for practice.

Reading: Elmasri & Navathe Chapter 30.

Tutorial 5.1: Database Security

Practice for [2026-05-11-database-security²](#).

Section A — Discretionary Access Control

Universe of discourse: Rob's Convenience Store has a database system to manage business activities. System admin Paris has created accounts for herself and four staff members, with these requirements:

- **Paris** (sysadmin) — full access to the whole database, and can grant permissions to others.
- **Rachel** (HR manager) — full access to **Employee**; can view (but not modify) **Restock**.
- **Aaron** (supply chain lead) — full access to **Item**, **Restock**, and **Sale**.
- **Chris** (customer service) — full access to **Customer**; read access to **Sale**.
- **Emily** (security manager, investigates missing stock) — read access to **Sale** and **Restock**; can view only the **fName/lName** attributes of both **Employee** and **Customer**.

¹[courses/infos1200/week12-tutorial-5-1-database-security.pdf](#)

²[courses/infos1200/2026-05-11-database-security.pdf](#)

Schema: Employee [id, fName, lName, role, email], Restock [employee, item, time, quantity] (FKs to Employee.id, Item.id), Item [id, name, description, price], Sale [customer, item, time, quantity] (FKs to Customer.id, Item.id), Customer [id, fName, lName, dob].

Question A.1 — Identify and fix incorrect access

The permission table below (as currently configured) contains several errors. Fix it, using the UoD above (– = not directly granted).

	Paris	Rachel	Aaron	Chris	Emily
Employee	ALL	ALL	UPDATE	—	SELECT
Restock	ALL	ALL	ALL	—	SELECT
Item	ALL	—	ALL	—	—
Sale	—	—	ALL	UPDATE, GRANT	—
Customer	SELECT, UPDATE	—	SELECT	ALL	—
Customer.fName/lName	—	—	—	—	SELECT
Employee.fName/lName	—	—	—	—	—

Working

Corrected table:

	Paris	Rachel	Aaron	Chris	Emily
Employee	ALL	ALL	—	—	—
Restock	ALL	SELECT	ALL	—	SELECT
Item	ALL	—	ALL	—	—
Sale	ALL	—	ALL	SELECT	SELECT
Customer	ALL	—	—	ALL	—
Customer.fName/lName	—	—	—	—	SELECT
Employee.fName/lName	—	—	—	—	SELECT

Errors fixed (bold): Rachel had **ALL** on **Restock** (should only be **SELECT** — she’s not allowed to make changes there). Aaron had **UPDATE** on **Employee** and **SELECT** on **Customer** (should have **none** — he’s supply chain, not HR/customer-facing). Chris had **UPDATE**, **GRANT** on **Sale** (should be **SELECT** only, and definitely no grant option — he only needs *read* access, and never had a stated need to further delegate access). Emily had no access to **Sale** at all (should have **SELECT**, per her investigative role) and no access to **Employee.fName/lName** (should have **SELECT**, matching her **Customer.fName/lName** access) — and Paris, as sysadmin, needed **ALL** on **Sale** and **Customer** (both were missing/blank), consistent with her full-database access.

Question A.2 — SQL grant/revoke queries

Write the SQL queries to fix the incorrectly-configured accounts from A.1.

Working

```
-- Rachel
REVOKE ALL PRIVILEGES ON `Rob's Convenience Store`.`Restock` FROM 'Rachel';
GRANT SELECT ON `Rob's Convenience Store`.`Restock` TO 'Rachel';

-- Aaron
REVOKE ALL PRIVILEGES ON `Rob's Convenience Store`.`Employee` FROM 'Aaron';
REVOKE ALL PRIVILEGES ON `Rob's Convenience Store`.`Customer` FROM 'Aaron';

-- Chris
REVOKE ALL PRIVILEGES ON `Rob's Convenience Store`.`Sale` FROM 'Chris';
REVOKE GRANT OPTION ON `Rob's Convenience Store`.`Sale` FROM 'Chris';
GRANT SELECT ON `Rob's Convenience Store`.`Sale` TO 'Chris';

-- Emily
REVOKE ALL PRIVILEGES ON `Rob's Convenience Store`.`Employee` FROM 'Emily';
GRANT SELECT ON `Rob's Convenience Store`.`Sale` TO 'Emily';
GRANT SELECT (`fName`, `lName`) ON `Rob's Convenience Store`.`Employee` TO
↳ 'Emily';
```

Section B — Mandatory Access Control

Universe of discourse: ASIO tracks undercover agents’ identities in a MAC database, to keep classified information from riskier assignments protected.

Schema: Spy [ID, fName, lName, dob, email], CodeName [spyID, name, dateEffective, dateExpired, operation] (FK spyID → Spy.id, operation → Operation.name),

Operation [name, description, budget].

DBMS accounts (low to high): Cody = U, Jane = C, Tanya = S, Jack = TS.

Sample data, with each attribute's classification noted, and the overall Tuple Classification (TC = the highest classification of any attribute present):

```
Spy - id(U), fName(U), lName(TS), dob(C), email(S)
id  fName  lName  dob      email      TC
1   Daniel Teal   18/06/1995  bmw@hotmail.com    TS
2   May    Lee    03/02/1998  may@stevefam.com    TS
3   John   Smith  12/05/1990  john@smith.com      TS
4   April  Fuller 25/06/1980  ilikefood@korea.com TS
```

CodeName - spyID always (C); name/operation classification varies per row;
dateEffective/dateExpired always (U)

```
spyID  name      dateEffective  dateExpired  operation      TC
1      Handbrake (C)  01/02/2019    03/06/2020  Le-Ferrari (S) S
2      BunnyBunny (S) 12/03/2019    12/04/2019  Hotpot (C)     S
4      BubbleTea (C) 19/04/2019    24/04/2019  Hotpot (C)     C
2      BBQ (TS)      27/09/2019    04/10/2019  KimChi (TS)    TS
```

```
Operation
name      description      budget      TC
Hotpot (S) "Infiltrate McDonalds..." (S) 1000000 (S) S
Le-Ferrari (C) "Identify the Ferrari-stealing syndicate leader" (U) 10000000 (U) C
KimChi (S) "Someone stole ASIO's KimChi supply..." (S) 500000000 (S) S
```

Question B.1 — Identifying security denials

For each query, determine whether it succeeds (S) or fails (F), and why.

i Working

User	Query	Result	Reason
Cody	SELECT id FROM Spy;	S	Cody's clearance (U) the classification of every id value (U).

Cody	<pre>SELECT name FROM Operation WHERE budget > 10000000</pre>	F	The one matching tuple (KimChi) has TC = S, above Cody's U clearance — the query returns nothing even though the WHERE condition is logically true for it.
Cody	<pre>INSERT INTO CodeName (spyID, name, dateEffective, dateExpired, operation) VALUES (2, "BBQ", "16/05/2019", "26/05/2019", "Hotpot")</pre>	S	Looks like it should violate the key constraint (another spyID=2, name="BBQ" combination already exists at TS) — but since the two “identical” tuples have different security classifications, the DBMS applies polyinstantiation to let them coexist.
Jack	<pre>INSERT INTO Operation (name, description, budget) VALUES ("Ace", "Find an unidentified hacker...", 350000)</pre>	F	Operation's highest classification is S. Jack's clearance is TS — higher than S — so the star property (no write down) blocks this insert, to prevent Jack from writing lower-classified information into a table that could then leak higher-clearance context.

Jane

```
UPDATE CodeName S
SET name = "Iced
Coffee" WHERE
spyID = 4
```

Jane's clearance (C) the classification of **spyID** (C, so she can read/select the row) and the classification of **name** for that row (C, so she can write it) — both read and write permission hold, so the update succeeds.

Question B.2 — Returning data in a MAC database

What data is returned for each query?

i Working

Jane: SELECT spyID, name, operation FROM CodeName →

spyID	name	operation
1	Handbrake	NULL
2	NULL	Hotpot
4	BubbleTea	Hotpot

(Row `spyID=2`, `name="BBQ"` is entirely TS-classified — above Jane's C clearance — so it doesn't appear at all. Within the visible rows, any individual cell above Jane's clearance is masked to NULL rather than hiding the whole row — e.g. row 1's `Handbrake` is C, visible, but row 2's `BunnyBunny` is S, masked.)

Jane: `SELECT name, dob FROM CodeName, Spy WHERE id = spyID AND dateEffective < "01/06/2020" →`

name	dob
Handbrake	18/06/1995
NULL	03/02/1998
BubbleTea	25/06/1980

Cody: `SELECT name, budget FROM Operation WHERE description LIKE "%a%" →`

name	budget
NULL	10,000,000

(Only `Le-Ferrari` — `TC=C` — is within reach of Cody's U clearance enough to appear at all, and even then its `name` attribute specifically, being C-classified, is masked to NULL for a U user; `budget` for that row is U-classified so it's shown.)

Jane: `SELECT spyID, Operation.name, Operation.budget FROM Operation, CodeName WHERE Operation.name = CodeName.operation → empty set.` Every possible join row pairs a `CodeName.operation` value with an `Operation.name` value where at least one side is classified S or TS — above Jane's C clearance — so no valid (joinable, visible-to- Jane) pairing exists, and the join returns nothing.

Bonus — Completing a database audit

Universe of discourse: SummerStyles (a clothing retailer) audits its DBMS log monthly for unauthorised changes or potential privacy violations.

Using the log below, flag any potentially suspicious records (there's no exact formula — use judgement).

event_time	user_host	thread_id	argument
17-05-2020 8:27:18am	SystemAdmin@localhost	2097	SHOW COLUMNS FROM SummerStyles.cus
17-05-2020 8:28:00am	SystemAdmin@localhost	2097	SHOW INDEXES FROM SummerStyles.cus
17-05-2020 8:30:07am	HRManager@localhost	2103	UPDATE SummerStyles.staff SET Waga
17-05-2020 8:31:07am	CustomerService@localhost	2129	SELECT * FROM SummerStyles.custom
17-05-2020 8:33:30am	SystemAdmin@localhost	2129	SELECT TABLE_NAME FROM information

17-05-2020 8:35:05am	CustomerService@localhost	2129	SELECT fName, lName, max(wage) FROM
17-05-2020 8:35:43am	StockManager@localhost	2129	DELETE FROM SummerStyles.deliveri
17-05-2020 8:38:32am	StockManager@localhost	2128	UPDATE SummerStyles.currentStock
17-05-2020 8:41:14am	SystemAdmin@localhost	2129	ALTER TABLE SummerStyles.staff AD
17-05-2020 8:45:41am	HRManager@localhost	2129	INSERT INTO SummerStyles.staff (i
17-05-2020 8:57:37am	HRManager@localhost	2131	UPDATE SummerStyles.stock SET pri

i Working

Two entries stand out as worth flagging:

1. **8:35:05am - CustomerService:** SELECT fName, lName, max(wage) FROM SummerStyles.staff. A customer service rep querying *staff wage* data is well outside their normal remit — salary information is HR’s domain (per the DAC exercise above, this kind of cross-role access is exactly what discretionary permissions should normally prevent) — a potential unauthorised-access/privacy concern worth following up.
2. **8:57:37am - HRManager:** UPDATE SummerStyles.stock SET price = 11.00 WHERE name LIKE "%Calvin Klein%". An HR manager modifying *product pricing* is outside HR’s normal duties (that’s **StockManager**’s job, as seen elsewhere in the log) — an unauthorised-modification concern, and potentially indicative of privilege misuse.