Database Access Control

Table of contents

What security is protecting against . . . 1
The four control measures . . . 2
Authentication vs authorisation . . . 2
The three authorisation mechanisms . . . 3
Discretionary Access Control (DAC) . . . 3
Mandatory Access Control (MAC) . . . 7
Role-Based Access Control (RBAC) . . . 9
Beyond access control . . . 9
See also . . . 10

Access control is the DBMS's answer to "who is allowed to do what to which data". It's one of four categories of database control measure, and by far the largest — three distinct authorisation mechanisms (DAC, MAC, RBAC), each with its own model and syntax. Introduced in 2026-05-11-database-security[1].

What security is protecting against

Threats to a database are usually framed with the **CIA principle**:

Threat	Meaning
Loss of **confidentiality**	Unauthorised disclosure of confidential information
Loss of **integrity**	Improper modification of information
Loss of **availability**	A legitimate user cannot access data objects

[1] courses/infs1200/2026-05-11-database-security.pdf

The motivations for defending against them are legal or policy-driven (government rules protecting medical records or financial ratings), ethical (who may see salary packages or student grades), and technical (malware, performance overloading).

Privacy is not the same as security

- **Security** concerns *how access to data is controlled* — availability for use, with integrity, and permitted access only.
- **Privacy** concerns *how data may be used* — an individual's ability to control the terms under which their sensitive data is acquired and used.

Security is a **required building block** for privacy, but the two are distinct: privacy also covers information that can be **inferred** from accessible data, not only what is directly readable. Sensitive data is often double-edged — patient records are private but valuable for public-health research; CCTV audio captures private conversations but also evidence of crime — giving a **utility/privacy trade-off**: protect all sensitive data while making as much non-sensitive data available as possible.

The four control measures

Measure	Purpose
Access control	Provide access only to users with the right access authority
Inference control	Ensure information about individuals can't be *derived* indirectly (statistical databases)
Flow control	Prevent information flowing to unauthorised users
Data encryption	Protect sensitive data in transit

The rest of this note is access control; the other three are summarised at the end.

Authentication vs authorisation

Database access control has two components, and they're often confused:

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

Supporting them are three basic security functions:

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

Auditing is judgement work, not a formula: in week12-tutorial-5-1-database-security[2] the entries worth flagging are a customer-service account running `SELECT fName, lName, max(wage) FROM staff` and an HR manager running `UPDATE stock SET price = ...` — neither is illegal SQL, both are outside the account's role.

The three authorisation mechanisms

Mechanism	Basis
Discretionary Access Control (DAC)	Granting and revoking privileges to individual users, at the owner's discretion
Mandatory Access Control (MAC)	Classifying data *and* users into security classes and enforcing a policy across them
Role-Based Access Control (RBAC)	Assigning users to organisational roles, then applying DAC or MAC at the role level

Discretionary Access Control (DAC)

DAC is built on **granting** and **revoking** privileges — `GRANT` and `REVOKE` being SQL's DCL family, alongside the DDL and DML of sql-ddl[3]:

```
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};
```

[2] courses/infs1200/week12-tutorial-5-1-database-security.pdf
[3] courses/infs1200/sql-ddl.pdf

Two levels of privilege

- **Account level** — privileges held by an account independent of any particular relation: `CREATE SCHEMA` / `TABLE` / `VIEW`, `ALTER` / `DROP TABLE`, and modification (`INSERT` / `DELETE` / `UPDATE`) and `SELECT` privileges.
- **Relation level** — privileges on a specific relation or view, and optionally on individual *attributes*: **select** (read), **modification** (insert/delete/update), and **references**. The `REFERENCES` privilege grants permission to create a *foreign key* pointing at the named table (see integrity-constraints[4]) — worth noting because it's a privilege about schema structure, not data access.

The three statement forms that repairing a real set of accounts needs — all three used in week12-tutorial-5-1-database-security[5]:

```
-- everything the account holds on that object, in one statement
REVOKE ALL PRIVILEGES ON `schema`.`Table` FROM 'user';

-- the ability to delegate, revoked separately from the privilege itself
REVOKE GRANT OPTION ON `schema`.`Table` FROM 'user';

-- attribute level: the column list attaches to the privilege, not the object
GRANT SELECT (`fName`, `lName`) ON `schema`.`Table` TO 'user';
```

There is no statement that *narrows* a privilege. Downgrading an account from `ALL PRIVILEGES` to `SELECT` is always a `REVOKE ALL PRIVILEGES` followed by a `GRANT SELECT`, and taking away someone's grant option is a second statement on top of taking away the privilege they were delegating.

Privileges can be issued as part of schema creation:

```
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;
```

The access matrix model

Every relation is assigned an **owner** account, and owners may grant select/modification/references privileges on anything they own. The resulting state is an **access matrix** `M(i, j)` — the privileges user `i` holds on object `j`. Attribute-level entries appear as their own rows:

[4] courses/infs1200/integrity-constraints.pdf
[5] courses/infs1200/week12-tutorial-5-1-database-security.pdf

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

Building this matrix from a Universe of Discourse is what Section A of week12-tutorial-5-1-database-security[6] asks for: read each person's stated responsibilities, and grant the *narrowest* privilege that satisfies them — read-only where they only need to look, nothing at all on tables outside their remit, attribute-level `SELECT` where they need two columns rather than a whole table.

Propagation and revocation

`REVOKE` cancels a privilege, which makes temporary grants possible. The complication is delegation: if `A` grants a privilege to `B` **`WITH GRANT OPTION`**, `B` can grant it onward without `A`'s knowledge, so the DBMS must track the resulting *chain of dependency*:

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

- **`CASCADE`** — revoke the privilege *and* every dependent privilege granted as a result of it.
- **`RESTRICT`** — refuse the revoke with an error if the privilege has already been passed on.

The governing rule is graph-shaped:

> A user's authorisation is valid **iff there is a path from the root of the authorisation graph down to that user's node.**

So for `u1 → u2 → u3`: revoking `u1`'s grant to `u2` also cuts off `u3`, because no path to `u3` remains. Revoking directly from `u3` leaves `u2` untouched, and `u2` losing authorisation never propagates *upward* to `u1` — the grantor's own authorisation is independent of what they've handed downstream.

[6] courses/infs1200/week12-tutorial-5-1-database-security.pdf

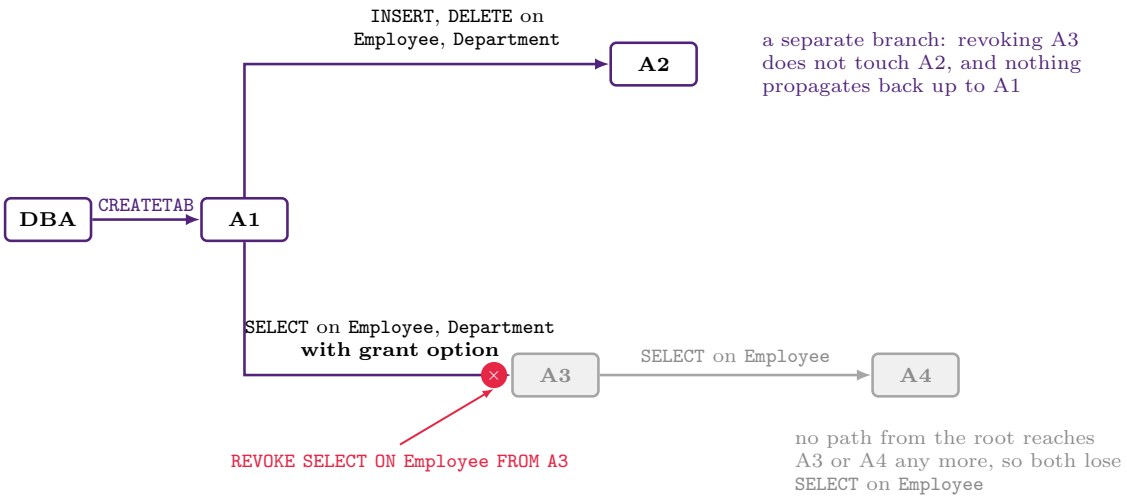


Figure 1: The A1–A4 grant chain from *Worked chain* below, drawn as the authorisation graph the path rule is stated over. Nothing here is a house convention. Revocation is about *reachability*, not about who was named in the REVOKE: A4 is never mentioned in the statement and loses the privilege anyway, purely because the edge it depended on is gone, while A2 on the untouched branch keeps everything.

Views as a security mechanism

If an owner wants to give an account a *limited* read capability — some columns only, or only rows meeting a condition — the idiomatic answer is to create a view and grant on the view rather than the base table (see sql-views[7]):

```
-- 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` and `address` for department 5 — never salary, never `ssn`, never another department's staff. The view is doing both column filtering and row filtering that a table-level `GRANT` could not express.

[7] courses/infs1200/sql-views.pdf

Worked chain

The DBA runs `GRANT CREATETAB TO A1;` `A1` creates `Employee` and `Department` and then runs:

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

- Can `A3` run `GRANT SELECT ON Employee TO A4`? **Yes** — `A3` holds the grant option.
- If `A1` then runs `REVOKE SELECT ON Employee FROM A3`, does `A4` keep `SELECT`? **No** — it's revoked by propagation; no path from the root reaches `A4` any more.
- What kind of access control is this? **Discretionary** — privileges are granted and revoked at the discretion of individual account owners, not fixed by a system-wide classification (MAC) or role hierarchy (RBAC).

Mandatory Access Control (MAC)

MAC classifies **both data and users** into security classes to provide **multilevel security**. The typical classes, low to high, are **Unclassified (U)** < **Confidential (C)** < **Secret (S)** < **Top Secret (TS)**. Users are **subjects**; data — a table, tuple, or individual attribute value — are **objects**.

The Bell-LaPadula model

Property	Rule	Effect
Simple security property ("no read up", NRU)	A subject can't *read* an object classified above its own clearance	A `U` user can't view a `C`-classified salary
Star property ("no write down", NWD)	A subject can't *write* to an object classified below its own clearance	An `S`-cleared user can't insert a tuple containing only `C`-classified data

"No write down" is the counter-intuitive one. It exists to stop information flowing *downward*: if a high-clearance subject could write into a low-classified object, the content of that write would leak higher-clearance context to lower-clearance readers.

Each tuple carries a **Tuple Classification (TC)** — the highest classification of any of its attribute values — and a *table's* classification is in turn the highest `TC` of any tuple in it, which is the level the star property is applied at when someone writes to the table as a whole. A user sees a *filtered* version of the relation:

```
(a) EMPLOYEE (as stored)
Name       Salary      JobPerformance  TC
Smith (U)  40000 (C)   Fair (S)        S
Brown (C)  80000 (S)   Good (C)        S

(b) as seen by a Confidential-clearance user
Smith (U)  40000 (C)   NULL (C)        C
Brown (C)  NULL  (C)   Good (C)        C

(c) as seen by an Unclassified-clearance user
Smith (U)  NULL (U)    NULL (U)        U
```

There are two levels of hiding, both of which the tutorial tests:

- An individual over-classified *cell* inside a tuple the user can otherwise see is **masked to `NULL`** — Smith's `JobPerformance` above, and every `NULL` in the tutorial's Section B answers.
- A tuple with **nothing readable left in it disappears entirely** — it is not returned as a row of `NULL`s. That is why Brown vanishes from view (c): at `U`, none of `Brown` (C), `80000` (S) or `Good` (C) is readable, so there is nothing left to show.

The exact test for the second case is worth being careful about. A high `TC` alone does not remove a tuple — in the tutorial, rows classified `S` are still returned to a `C` user with their `S` cells masked. Nor is an unreadable *key* enough on its own: `CodeName`'s key is the composite `{spyID, name}`, and the tutorial's answers still return the `BunnyBunny` tuple to a `C` user with its `S`-classified `name` masked. What removes a tuple is having no readable value in it at all. A join between two tables can therefore return the empty set even when matching rows plainly exist, if every candidate pairing has at least one over-classified side — the join condition simply cannot be established at the user's level.

Polyinstantiation

Suppose a `C`-clearance user, seeing view (b), runs `UPDATE EMPLOYEE SET JobPerformance = 'Excellent' WHERE Name = 'Smith'`. The system **must not reject** it — a rejection would reveal that a higher-classified value already exists, which is itself a leak (a **covert channel**). The resolution is **polyinstantiation**: keep multiple tuples with the same key at different classifications.

```
(d) EMPLOYEE, polyinstantiated
Smith (U)  40000 (C)   Fair (S)        S
Smith (U)  40000 (C)   Excellent (C)   C
Brown (C)  80000 (S)   Good (C)        S
```

The `C` user sees "Excellent"; an `S`-or-higher user still sees the original "Fair". The same mechanism is why an apparent key-constraint violation can succeed in a MAC database — two "identical" tuples at different classifications are allowed to coexist.

DAC vs MAC

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

Role-Based Access Control (RBAC)

Permissions attach to **organisational roles**; users are assigned to roles, and DAC or MAC methods are then applied to the role 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};
```

The typical hierarchy is **users → groups → roles → privileges** — e.g. `alice` is in group `admin`, which maps to `admin_role`, which holds `ALL ON SERVER server1`. RBAC's flexibility and much easier administration (change the role, not every user) make it the popular choice for web-based applications.

Beyond access control

Inference control

A **statistical database** publishes aggregate statistics about a population — for government statisticians or market researchers — without exposing individual records, so only aggregate

queries (`COUNT`, `SUM`, `MIN`, `MAX`, `AVG`, standard deviation) are permitted (see sql-aggregation-and-grouping[8]):

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

The risk is that a sufficiently narrow `WHERE` clause isolates a group small enough — possibly a single person — that the "aggregate" discloses an individual. Mitigations:

- **k-anonymity** — enforce a minimum number of tuples any result may be based on.
- **Prohibit query sequences** over the same or overlapping populations, which could be differenced to isolate an individual.
- **Differential privacy** — add carefully calibrated noise to results.

Flow control

Regulates the distribution of information among accessible objects, verifying that information doesn't flow — explicitly or implicitly — into *less* protected objects. This is Bell-LaPadula's "no write down" generalised beyond a single DBMS. A **flow policy** names the channels information may travel along; in a simple confidential (C) / nonconfidential (N) scheme, C → N flow is prohibited. Example: an income tax computing service may be allowed to retain a customer's address, but not their income and deductions.

Data encryption

Encryption converts data into ciphertext using an algorithm and a key; recovering the original requires the corresponding decryption key. **DES** (Data Encryption Standard) was developed by the US government for general public use; **AES** (Advanced Encryption Standard) is the newer, harder-to-crack replacement.

See also

- sql-injection[9] — the attack that bypasses all of this by making the application issue the attacker's SQL under its *own* privileges
- sql-views[10] · integrity-constraints[11] · sql-aggregation-and-grouping[12]

[8] courses/infs1200/sql-aggregation-and-grouping.pdf
[9] courses/infs1200/sql-injection.pdf
[10] courses/infs1200/sql-views.pdf
[11] courses/infs1200/integrity-constraints.pdf
[12] courses/infs1200/sql-aggregation-and-grouping.pdf

- 2026-05-11-database-security[13] — Module 5
- week12-tutorial-5-1-database-security[14] — DAC permission-matrix, MAC query and audit practice

[13] courses/infs1200/2026-05-11-database-security.pdf
[14] courses/infs1200/week12-tutorial-5-1-database-security.pdf