CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 11: Views, Functions & Security

Amit Kumar Dhar

IIT Bhilai

Beyond writing queries

- We can query and modify data. Now: **package** and **protect** it.
- **Views** — reusable, named queries (and how updates through them work).
- **Materialized views** — precomputed results for speed.
- **Stored functions/procedures** — logic that lives in the database.
- **Access control** — users, roles, `GRANT`/`REVOKE`, least privilege.

Agenda

- Views recap; why views
- Updatable views & `WITH CHECK OPTION`
- Materialized views & refresh
- Stored functions vs procedures
- Users, roles, privileges
- `GRANT` / `REVOKE`
- Views + privileges = security
- Row-level security (glimpse)

Views recap

```sql
CREATE VIEW cs_students AS
SELECT id, name, tot_cred
FROM   student
WHERE  dept_name = 'Comp. Sci.';
```

- A view is a **stored query** used like a virtual table.
- No data is stored; the query runs each time the view is used.
- `DROP VIEW cs_students;` removes the definition (not the base data).

Quick Check: Views Recap

Question: If you drop a view using `DROP VIEW`, what happens to the underlying base table data?

Quick Check: Views Recap

Question: If you drop a view using `DROP VIEW`, what happens to the underlying base table data?

Answer: Nothing. The base table data remains intact; only the view's definition (the stored query) is removed.

Why views (four reasons)

- **Abstraction** — hide join/aggregation complexity behind a name.
- **Reuse** — many queries/apps share one definition.
- **Security** — expose only chosen columns/rows.
- **Stability / logical independence** — change base tables, keep the view's interface (recall Lecture 1's logical data independence).

What a view is *not*

- Not a copy of data (that's a materialized view or a table).
- Not automatically faster — a view is expanded into its query each use.
- Not a place to "store results" — it re-runs every time.
- Think of a view as a **saved query with a name**, nothing more (until you materialize it).

Naming and organizing views

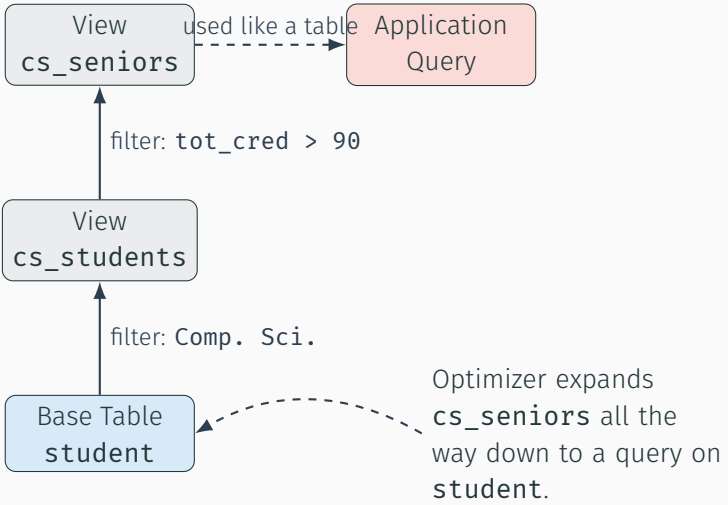
- Prefix by purpose: `rpt_` (reporting), `sec_` (security), `v_` generic.
- Keep view logic **thin** — deep stacks of views-on-views get hard to debug.
- Document the intent; downstream code depends on the view's **columns** as a contract.

Views compose

```sql
CREATE VIEW cs_seniors AS
SELECT * FROM cs_students WHERE tot_cred > 90;
```

- A view can be built on another view.
- The optimizer **expands** views into the underlying query at run time — no performance penalty for the abstraction itself.

How Views Compose (Concept)



Updating through a view

- Can you `INSERT`/`UPDATE`/`DELETE` through a view? **Sometimes.**
- A view is (in general) updatable when it maps **cleanly to one base table**:
  - selects from a single table,
  - no `DISTINCT`, `GROUP BY`, aggregates, or set operations,
  - includes the columns needed to identify/insert a row.

Updatable view example

```
CREATE VIEW cs_students AS
SELECT id, name, tot_cred FROM student WHERE dept_name = 'Comp. Sci.';

UPDATE cs_students SET tot_cred = tot_cred + 4 WHERE id = '12345';  -- OK
```

- The update rewrites to the base `student` table.
- But note the next slide's subtlety…

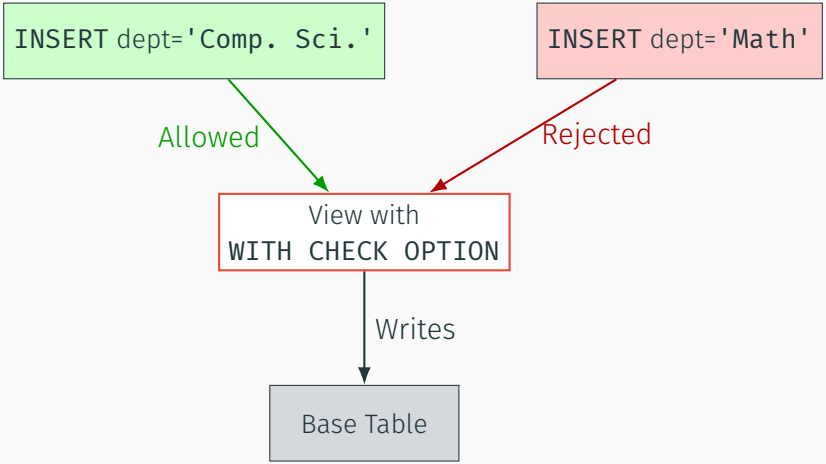
The ”disappearing row” problem

```
INSERT INTO cs_students (id, name, tot_cred) VALUES ('99999', 'New', 0);
```

- `dept_name` isn’t in the view → the new row gets `NULL` dept → it **isn’t** Comp. Sci. → it **vanishes** from the view!
- Fix with **`WITH CHECK OPTION`**:

```
CREATE VIEW cs_students AS SELECT ... WHERE dept_name = 'Comp. Sci.'
WITH CHECK OPTION;   -- rejects inserts/updates that would leave the view
```

Visualizing WITH CHECK OPTION



Quick Check: Updatable Views

Question: Why would an `INSERT` statement into a view succeed, but immediately after, you cannot see the new row when querying that same view?

Quick Check: Updatable Views

Question: Why would an `INSERT` statement into a view succeed, but immediately after, you cannot see the new row when querying that same view?

Answer: The inserted row might not satisfy the `WHERE` condition of the view's query (e.g., leaving a column `NULL`). Thus, the row is inserted into the base table, but filtered out by the view. Using `WITH CHECK OPTION` prevents this.

Non-updatable views

- Views with aggregation, `GROUP BY`, `DISTINCT`, joins, or set ops generally **can't** be updated directly — the mapping back to base rows is ambiguous.
- Workaround: **`INSTEAD OF` triggers** define custom update behaviour.
- SQLite: views are **read-only**; use `INSTEAD OF` triggers to allow writes.

Materialized views

- A normal view **recomputes** every time — costly for heavy aggregations.
- A **materialized view** stores the **result** on disk:

```
-- PostgreSQL
CREATE MATERIALIZED VIEW category_revenue AS
SELECT c.cat_name, SUM(oi.quantity*oi.unit_price) AS revenue
FROM   categories c JOIN products p ON p.cat_id=c.cat_id
       JOIN order_items oi ON oi.prod_id=p.prod_id
GROUP  BY c.cat_name;
```

- Reads are now fast — you query stored rows.

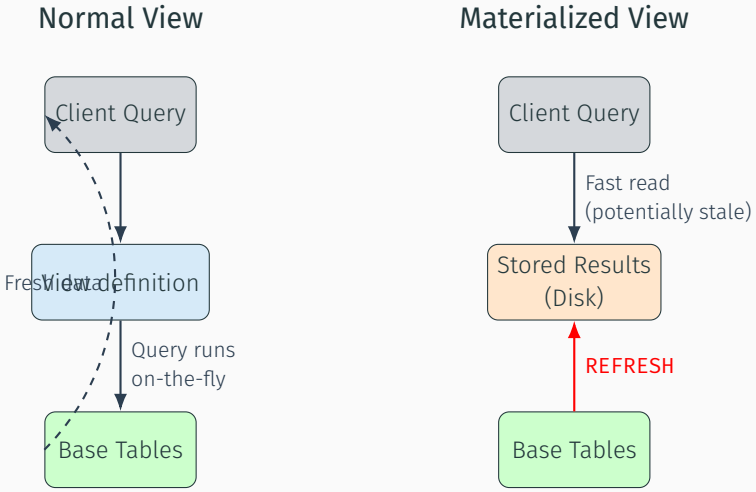
The trade-off: staleness

- Materialized data can become **out of date** as base tables change.
- Refresh it explicitly:

```
REFRESH MATERIALIZED VIEW category_revenue;                  -- recompute
REFRESH MATERIALIZED VIEW CONCURRENTLY category_revenue;     -- no read lock
```

- **View:** always fresh, recomputed each read.
- **Materialized view:** fast reads, but you manage refresh.
- SQLite has no materialized views — emulate with a table + triggers/cron.

Normal View vs Materialized View



Quick Check: Materialized Views

Question: Why not make all views materialized views to speed up queries?

Quick Check: Materialized Views

Question: Why not make all views materialized views to speed up queries?

Answer: Materialized views take up storage space and they become stale when base data changes. You must periodically refresh them, which takes computation and locking (unless concurrent). It's a trade-off between read speed and data freshness.

Stored functions & procedures

- Move logic **into** the database — run close to the data, reuse across apps.
- **Function** — returns a value, usable in queries.
- **Procedure** — performs actions, invoked with `CALL`; may manage transactions.

Stored function example (PostgreSQL)

```
CREATE FUNCTION order_total(oid INT) RETURNS NUMERIC AS $$
    SELECT COALESCE(SUM(quantity*unit_price), 0)
    FROM   order_items WHERE order_id = oid;
$$ LANGUAGE sql;

SELECT order_id, order_total(order_id) FROM orders;
```

- Encapsulates a computation; callable anywhere an expression is allowed.
- PL/pgSQL adds variables, loops, conditionals for richer procedures.

Functions: pros & cons

- **Pros:** less data shipped to the app; shared logic; can be `SECURITY DEFINER` to run with elevated rights for controlled tasks.
- **Cons:** logic split between app and DB; harder to version/test; dialect-specific (PL/pgSQL vs T-SQL vs PL/SQL).
- Use for data-intensive logic; keep business rules where your team can maintain them.

Quick Check: Functions

Question: Why would you write a stored function instead of just putting the logic in your Python or Java application?

Quick Check: Functions

Question: Why would you write a stored function instead of just putting the logic in your Python or Java application?

Answer: A stored function runs close to the data, which avoids shipping large amounts of data over the network. It also allows multiple different applications (e.g., a web app and a reporting script) to reuse the exact same logic natively within SQL queries.

Database security model

- Real databases are **multi-user**. Access must be controlled.
- Building blocks:
  - **Users / roles** — identities and groups of privileges.
  - **Privileges** — permissions on objects (SELECT, INSERT, …).
  - **GRANT / REVOKE** — hand out and take back privileges.

Users and roles

```
-- PostgreSQL
CREATE ROLE analyst;                          -- a group of privileges
CREATE ROLE ravi LOGIN PASSWORD 'secret';     -- a user
GRANT analyst TO ravi;                        -- ravi inherits analyst's rights
```

- A **role** bundles privileges; grant the role to many users.
- Manage permissions by role, not per-user — far more maintainable.

Privileges & GRANT

```
GRANT SELECT                 ON products   TO analyst;
GRANT SELECT, INSERT, UPDATE ON orders     TO app_writer;
GRANT ALL PRIVILEGES         ON categories TO admin;
```

- Object privileges: SELECT, INSERT, UPDATE, DELETE, REFERENCES, …
- Can even be column-level: `GRANT SELECT (name, city) ON customers TO analyst;`

REVOKE and the principle of least privilege

```
REVOKE INSERT ON orders FROM app_writer;
REVOKE ALL ON customers FROM analyst;
```

- **Least privilege:** grant only what each role needs, nothing more.
- Limits the blast radius of a bug or a compromised account.
- Start closed; open up deliberately.

WITH GRANT OPTION

```
GRANT SELECT ON products TO manager WITH GRANT OPTION;
```

- Lets `manager` re-grant `SELECT` on `products` to others.
- Revoking from `manager` can **cascade** to those they granted.
- Use cautiously — it delegates authority.

Views as a security mechanism

- Grant access to a **view**, not the base table:

```
CREATE VIEW public_customers AS
SELECT cust_id, name, city, country FROM customers;   -- no sensitive columns

GRANT SELECT ON public_customers TO analyst;   -- analyst can't see base table
REVOKE ALL   ON customers        FROM analyst;
```

- The view hides columns (and can hide rows via `WHERE`) — **column/row-level security** built from views + privileges.

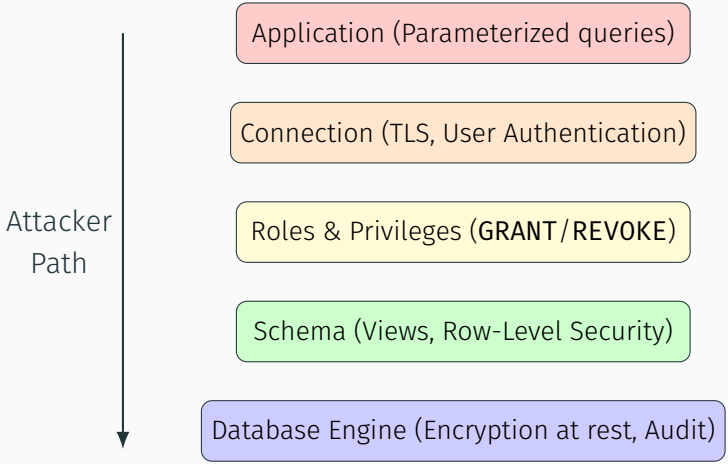
Row-level security (a glimpse)

- Sometimes users should see **only their own rows**.
- **PostgreSQL Row-Level Security (RLS):**

```
ALTER TABLE orders ENABLE ROW LEVEL SECURITY;
CREATE POLICY own_orders ON orders
    USING (cust_id = current_setting('app.current_customer')::int);
```

- The DBMS filters rows automatically per policy — no app-side filtering needed.
- More robust than "remember to add `WHERE cust_id = ...`" in every query.

Defense in Depth (Security Layers)



Defense in depth

- **Parameterized queries** (Lecture 10) — stop injection.
- **Least-privilege roles** — limit what each account can do.
- **Views / RLS** — restrict visible columns and rows.
- **Encryption & TLS** — protect data at rest and in transit.
- Security is **layers**, not a single switch.

Quick Check: Security

Question: If a user is granted SELECT on a view, do they also need SELECT on the base table to query the view?

Quick Check: Security

Question: If a user is granted `SELECT` on a view, do they also need `SELECT` on the base table to query the view?

Answer: No. The view acts as a security boundary. As long as the view's owner has access to the base table, the user only needs access to the view itself.

View dependencies

- Objects depend on views; dropping carelessly breaks them:

```
DROP VIEW cs_students;              -- fails if cs_seniors depends on it
DROP VIEW cs_students CASCADE;      -- also drops dependents (Postgres)
```

- The DBMS tracks dependencies in the **catalog**. Alter base columns a view uses and the view may break — plan schema changes with views in mind.

Common view patterns

- **Denormalized read view** — pre-join tables for reporting convenience.
- **Filtered view** — a stable subset (e.g., `active_customers`).
- **Aggregation view** — per-group summaries behind a name.
- **Security view** — projects away sensitive columns.
- Views are cheap to create and drop — use them to keep app SQL simple.

INSTEAD OF triggers (writable complex views)

```
-- SQLite: make a read-only view writable
CREATE TRIGGER ins_cs_students
INSTEAD OF INSERT ON cs_students
BEGIN
    INSERT INTO student (id, name, dept_name, tot_cred)
    VALUES (NEW.id, NEW.name, 'Comp. Sci.', NEW.tot_cred);
END;
```

- The trigger defines exactly how a write on the view maps to base tables.
- Lets you offer a friendly writable interface over a non-trivial view.

SECURITY DEFINER functions

```
CREATE FUNCTION redeem_points(cid INT, pts INT) RETURNS void AS $$ ... $$
LANGUAGE plpgsql SECURITY DEFINER;  -- runs with the FUNCTION OWNER's rights
```

- Normally a function runs with the **caller's** privileges (`SECURITY INVOKER`).
- `SECURITY DEFINER` runs with the **owner's** — lets low-privilege users perform a *controlled* privileged action without granting them broad table access.
- Powerful but sensitive — write them carefully (set a safe `search_path`).

Schema-level and default privileges

```
GRANT USAGE ON SCHEMA app TO analyst;          -- see objects in schema
GRANT SELECT ON ALL TABLES IN SCHEMA app TO analyst;
ALTER DEFAULT PRIVILEGES IN SCHEMA app
    GRANT SELECT ON TABLES TO analyst;         -- applies to FUTURE tables too
```

- Manage access at the **schema** level, and set **defaults** so new tables are automatically covered — less error-prone than per-table grants.

Auditing changes

- Combine triggers + a log table to record **who changed what, when**:

```
CREATE TRIGGER audit_price AFTER UPDATE OF price ON products
BEGIN
    INSERT INTO price_audit(prod_id, old_price, new_price, changed_at)
    VALUES (OLD.prod_id, OLD.price, NEW.price, datetime('now'));
END;
```

- Essential for compliance and debugging ”who dropped that price?”.

Table vs view vs materialized view

	Stores data?	Always fresh?	Fast reads?
Table	yes	(it *is* the data)	yes
View	no	yes (recomputed)	depends on query
Materialized view	yes (cached)	no (needs refresh)	yes

- Choose by workload: hot analytics → materialized; live truth → view.

Common mistakes

- Assuming any view is updatable (aggregated/joined ones aren't).
- Forgetting `WITH CHECK OPTION` → rows "escape" a filtered view.
- Never refreshing a materialized view → stale reports.
- Granting broad privileges (`ALL`, superuser) out of convenience.
- Filtering rows in application code instead of the database.

Summary — key takeaways

- **Views** abstract, reuse, and secure queries; updatable only when they map cleanly to one base table (`WITH CHECK OPTION` keeps rows in scope).
- **Materialized views** trade freshness for speed (you manage refresh).
- **Stored functions/procedures** put logic near the data.
- **Roles + `GRANT`/`REVOKE` + views/RLS** enforce **least-privilege** security.

Exercises

1. Create an updatable view of `Physics` students; test the disappearing-row problem, then fix it with `WITH CHECK OPTION`.
2. Create a materialized view of monthly revenue; refresh after inserting an order.
3. Write a stored function `customer_spend(cid)` returning total spend.
4. Design roles `analyst` (read-only, no PII) and `app_writer`; write the `GRANT`/`REVOKE` statements using a view to hide customer emails.

Quick reference — views & privileges

```
CREATE VIEW v AS SELECT ...;              CREATE VIEW v AS ... WITH CHECK OPTIO
CREATE MATERIALIZED VIEW mv AS ...;       REFRESH MATERIALIZED VIEW mv;
CREATE ROLE r;   GRANT r TO user;
GRANT SELECT, INSERT ON t TO r;           REVOKE INSERT ON t FROM r;
GRANT SELECT (col1, col2) ON t TO r;      -- column-level
```

Where security lives

- **In the app:** parameterized queries (stop injection).
- **In the connection:** TLS, least-privilege login role.
- **In the schema:** views hide columns; RLS hides rows; `GRANT`/`REVOKE`.
- **In the DB engine:** encryption at rest, audit logs.
- No single layer is enough — combine them.

Before next lecture

- Try the view/role exercises in PostgreSQL (SQLite lacks roles/materialized views).
- Optional reading: Silberschatz §4.2 (views), §4.6 (authorization).

Next time — Lecture 12: transactions from the user's view — ACID, `COMMIT`/`ROLLBACK`, isolation levels, and a bridge into the internals module.

Questions?