Lecture 31: Database Objects

Subquery Recap, Views, and Indexes

Z2004: Database Management Systems



IIT Madras Zanzibar

Monday, 1 June 2026

Agenda for Today

Mastering the Complex, Simplifying the Output

1. **Recap: Standard Subqueries:** The "Calculate First" approach.
2. **Recap: Correlated Subqueries:** The "For-Each" Loop.
3. **Views (Virtual Tables):** Saving complex queries for later.
4. **Indexes (Performance):** Putting L17 into practical SQL code.

Lecture Objectives

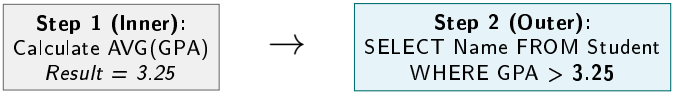
- Explain **standard vs. correlated subqueries** and predict how often the inner query runs.
- Explain what a **view** is and why it's useful (abstraction, reuse, security).
- Write SQL to **create, query, and maintain views** (and recognize when a view is updatable).
- Distinguish between **base tables, views, and temporary tables** (storage, lifespan, and typical use-cases).
- Explain what an **index** is and how it improves performance, including trade-offs for `INSERT`/`UPDATE`/`DELETE`.
- Choose **what to index** based on query patterns (`WHERE`, `JOIN`, `ORDER BY`) and interpret performance at a high level.

1. Standard Subqueries (The "Solve First" Method)

If you struggled with subqueries last week, remember this rule: **A standard subquery runs exactly ONE time.** The database solves the math inside the parentheses first, and hands the answer to the outer query.

Goal: Find students whose GPA is higher than the school average.

```
SELECT Name, GPA
FROM Student
WHERE GPA > (SELECT AVG(GPA) FROM Student);
```



2. Correlated Subqueries (The "For-Each" Loop)

A **Correlated** subquery is entirely different. It contains a reference to the outer table. Because of this, it cannot be solved just once. **It must run once for EVERY SINGLE ROW in the outer table.**

Goal: Find students whose GPA is higher than the average of *their specific department*.

```
SELECT s1.Name, s1.Dept, s1.GPA
FROM Student s1
WHERE s1.GPA > (
    SELECT AVG(s2.GPA)
    FROM Student s2
    WHERE s2.Dept = s1.Dept -- The correlation!
);
```

How it runs: 1. SQL looks at **Asha** (CSE). It runs the inner query to find the CSE average (3.4). Asha (3.8) $>$ 3.4. Keep her!
2. SQL looks at **Zahra** (EEE). It runs the inner query to find the EEE average (3.6). Zahra (3.1) $<$ 3.6. Discard!

3. Introducing VIEWS

Queries with multiple JOINs and subqueries get very long and messy. What if we want to run that exact same query tomorrow? We don't want to type it all again!

What is a View?

- A View is a **saved SQL query** that acts exactly like a virtual table.
- It does **not** store data on the hard drive! It just stores the SQL code.
- When you query the View, it runs the underlying code on the fly.

Creating a View:

```
CREATE VIEW HonorRoll AS
SELECT Name, Dept, GPA
FROM Student
WHERE GPA >= 3.5;
```

Using the View:

```
-- Treat it just like a table!
SELECT * FROM HonorRoll;
```

Why do we use Views?

Views are heavily used in the software industry for two main reasons:

1. Simplicity

Instead of forcing junior developers or business analysts to write complex 5-table JOINs, the Database Admin creates a View called `StudentFullDetails`. The analyst just writes `SELECT * FROM StudentFullDetails`.

2. Security

Suppose your `Employee` table has `Name`, `Department`, and `Salary`. You want the managers to see the names, but **hide** the salaries.

```
CREATE VIEW PublicDirectory AS
SELECT Name, Dept
FROM Employee;
```

You grant the managers access to the View, but block access to the real table!

Maintaining Views (Modify / Replace / Drop)

- A view stores *SQL code*. To change it, you typically replace or drop + recreate it.

Replace a view definition:

```
CREATE OR REPLACE VIEW HonorRoll AS
SELECT Name, Dept, GPA
FROM Student
WHERE GPA >= 3.6;
```

Remove a view:

```
DROP VIEW HonorRoll;
```

Are Views Updatable? (When can you INSERT/UPDATE?)

Usually updatable if the view is "simple":

- selects from **one** base table
- no `GROUP BY`, aggregates, `DISTINCT`, `UNION`
- no computed columns that lose information

Rule of thumb: if a view row maps to exactly *one* base-table row, updates are more likely to be allowed.

Works (simple view):

```
CREATE VIEW CSEStudents AS
SELECT ID, Name, Dept
FROM Student
WHERE Dept = 'CSE';

UPDATE CSEStudents
SET Name = 'Asha Khan'
WHERE ID = 17;
```

Doesn't work (aggregate view):

```
CREATE VIEW DeptAvg AS
SELECT Dept, AVG(GPA) AS AvgGPA
FROM Student
GROUP BY Dept;

-- UPDATE DeptAvg ... (typically rejected)
```

Views vs. Materialized Views

Normal View (virtual)
A named `SELECT` that the DB runs when you query the view; it typically stores *no rows* itself.

- stores only the **query**
- results computed **on demand**
- always reflects latest base-table data

Materialized View (stored result)
A view where the `SELECT` result is *precomputed and stored*; it must be refreshed to stay up-to-date.

- stores the **output rows** physically
- faster reads, but can become **stale**
- needs refresh (DB-specific syntax)

Examples: View vs. Materialized View

Normal View (virtual)

```sql
-- Stores only the query
CREATE VIEW CSEHonorRoll AS
SELECT ID, Name, GPA
FROM Student
WHERE Dept = 'CSE' AND GPA >= 3.5;

-- Always reflects latest Student data
SELECT * FROM CSEHonorRoll;
```

Materialized View (stored result)

```sql
-- DB-specific syntax (example: PostgreSQL)
CREATE MATERIALIZED VIEW DeptAvgGPA AS
SELECT Dept, AVG(GPA) AS AvgGPA
FROM Student
GROUP BY Dept;

-- Refresh to update stored rows
REFRESH MATERIALIZED VIEW DeptAvgGPA;
```

4. Practical Indexes in SQL

In Lecture 17, we learned that databases use **B+ Trees** (Indexes) to prevent massive Full Table Scans. How do we actually build one in SQL?

The Scenario: Our school app allows teachers to search for students by their Email address. Right now, this takes a long time because the DB scans the whole table.

```
-- Create an index to speed up Email searches
CREATE INDEX idx_student_email
ON Student (Email);
```

- Now, when we run `SELECT * FROM Student WHERE Email = 'x@uni.edu'`, the Optimizer will use the B+ Tree to find the row in 2 milliseconds!

Indexes for JOINs + Reading Performance

JOINs get faster when we index the join columns (especially foreign keys).

Example: We often join `Enrollment(StudentID)` to `Student(ID)`.

```
-- Helpful index for lookups during the join
CREATE INDEX idx_enrollment_student
ON Enrollment (StudentID);

-- Typical join
SELECT s.Name, e.CourseID
FROM Student s
JOIN Enrollment e ON e.StudentID = s.ID
WHERE s.Dept = 'CSE';
```

How do we tell it's using the index? Use a query plan tool (DB-specific):

```
EXPLAIN SELECT ...
```

Look for terms like `Index Scan/Seek` instead of `Full Table Scan`.

When should we use Indexes?

If Indexes make reading data so fast, why don't we put an Index on every single column in the database?

The Trade-off: Reads vs. Writes
Every time you run an `INSERT`, `UPDATE`, or `DELETE` on the table, the database must also update the B+ Tree Index! If you have 20 indexes on a table, inserting a single row will take 20 times longer!

Best Practices:

- **DO** Index columns you frequently use in `WHERE`, `JOIN`, or `ORDER BY` clauses.
- **DO NOT** Index columns that are rarely searched (e.g., `DateJoined`).
- **DO NOT** Index tables that are updated constantly (e.g., a live sensor logging table).

Base vs. View vs. Temporary Table

- **Base table (permanent)**: A real table that **physically stores** rows on disk. It stays in the database until you `DROP` it, and it supports full `INSERT/SELECT/UPDATE/DELETE`.
- **View (virtual table)**: A **saved** `SELECT` statement. It typically stores **no data** itself; when queried, the DB runs the underlying query on the current base-table data (some DBs also support *materialized* views that store results).
- **Temporary table**: A table that exists **only for a session/connection or transaction** (DB-specific). It's usually visible only to the creator and is automatically removed when it goes out of scope (used for intermediate results).

Comparison Table: Base vs. View vs. Temporary

Feature	Base Table	View (Virtual Table)	Temporary Table
Stores Actual Data?	Yes	No (stores a saved query)	Yes (temporarily)
Storage Location	Permanent disk	None (fetches from base tables)	Memory or temporary disk
Lifespan	Permanent	Permanent	Deleted when session ends
Performance Impact	Fast access	Dependent on underlying query	Fast, but uses system memory

Summary

Key takeaways (L31: Database Objects)

- **Standard subquery**: runs once, then the outer query uses the computed value(s).
- **Correlated subquery**: runs repeatedly (typically once per outer-row) because it references the outer query.
- **Base table**: permanent, physically stores rows (the real data).
- **View**: a saved `SELECT` (virtual table); usually stores no data and is computed on demand.
- **Temporary table**: short-lived table used for intermediate results (scope is session/transaction, DB-specific).
- **Index**: B+ tree structure that speeds up reads (`WHERE`/`JOIN`/`ORDER BY`) but adds overhead to writes.

Use the right object for correctness, simplicity, and performance.