Database Management Systems

Subqueries, sets and GROUP BY

The notes for the lesson “Subqueries, sets and GROUP BY”: chapters 3.3 to 3.4, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 3.3 to 3.4

About these notes

These are the notes for the University free n easy video lesson **Subqueries, sets and GROUP BY** on Database Management Systems: chapters 3.3 to 3.4, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

3.3 Set operations and nested subqueries

Hard questions are answered by building a query out of smaller queries. There are two ways. A **set operation** (UNION, INTERSECT, EXCEPT) joins the results of two whole queries, as with sets in school maths. A **nested subquery** is a complete SELECT written inside another query, usually in its WHERE clause, to test whether a value is in a set, to compare with a computed value, or to check whether any matching row exists.

You need: 3.2 Basic SQL queries and its data, and An earlier lesson, Set operations in algebra.

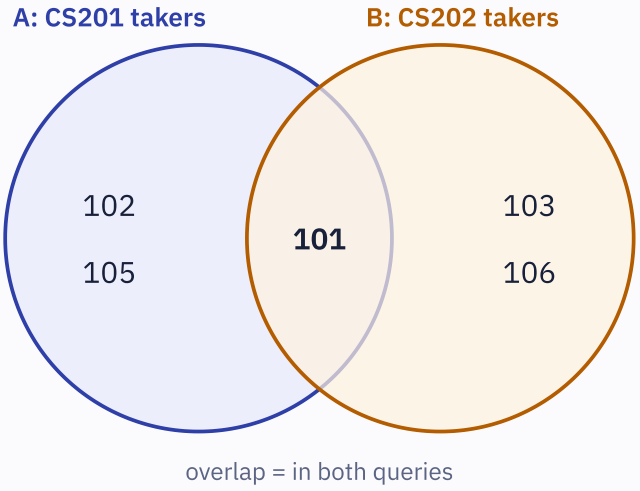
UNION, INTERSECT and EXCEPT

Let A be the roll numbers taking CS201 and B those taking CS202:

```
SELECT roll_no FROM enrols WHERE course_id = 'CS201'   -- A = {101, 102, 105}
UNION                                                  -- or INTERSECT, EXCEPT
SELECT roll_no FROM enrols WHERE course_id = 'CS202';  -- B = {101, 103, 106}
```

The two queries must be **union-compatible**: the same number of columns, with matching types in the same order. The result takes its column names from the first query, and one ORDER BY may follow at the very end. Oracle spells EXCEPT as `MINUS`. All three remove duplicate rows;

`UNION ALL` keeps them and is faster because it skips the duplicate check. Figure 3.3.1 shows the results.



UNION	101, 102, 103, 105, 106 (5 rows)
UNION ALL	6 rows, 101 twice
INTERSECT	101
A EXCEPT B	102, 105
B EXCEPT A	103, 106

Oracle writes EXCEPT as MINUS

Figure 3.3.1 The three set operations on two queries over enrols. Notice that EXCEPT is not symmetric (A − B ≠ B − A) and that only UNION ALL keeps the duplicate 101.

Subqueries with IN, ANY, ALL and EXISTS

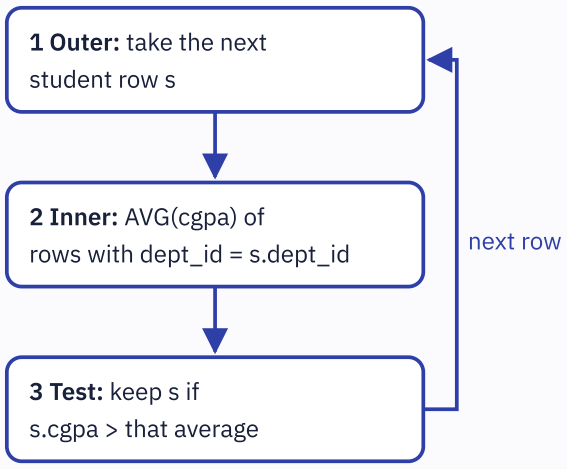
- **IN**: `WHERE roll_no IN (SELECT roll_no FROM enrols WHERE course_id = 'CS202')` finds Anu, Chitra, Farhan.
- **Scalar subquery**, one value: `WHERE cgpa > (SELECT AVG(cgpa) FROM student)`. The seven CGPAs add to 56.0, so the average is 56.0 / 7 = 8.0, and the answer is Anu, Chitra, Farhan (Esha's 8.0 is not greater). A scalar subquery that returns two rows is a run-time error.
- **ALL** and **ANY** (also spelt SOME) compare with every value of a set. ECE CGPAs are {8.4, 6.9}. `cgpa > ALL (ECE)` means above 8.4: Anu, Farhan. `cgpa > ANY (ECE)` means above 6.9: everyone except Dev, 6 rows. `= ANY` is the same as IN, and `<> ALL` the same as NOT IN.
- **EXISTS (subquery)** is true when the subquery returns at least one row; NOT EXISTS when it returns none. The SELECT list inside does not matter, so people write `SELECT *` or `SELECT 1`.

Correlated subqueries

A subquery is **correlated** when it uses a column of the outer query's current row. It then cannot be worked out once; logically it runs again for every outer row. "Students whose CGPA is above their own department's average":

```
SELECT s.name FROM student s
WHERE s.cgpa > (SELECT AVG(t.cgpa) FROM student t
                WHERE t.dept_id = s.dept_id);
```

The table aliases s and t tell the two copies of student apart. Figure 3.3.2 runs it row by row. A subquery that does not mention the outer row, like the overall average above, is **non-correlated** and runs once.



s.name	dept	cgpa	inner AVG	kept?
Anu	CSE	9.1	8.50	yes
Bala	CSE	7.8	8.50	no
Chitra	ECE	8.4	7.65	yes
Dev	ECE	6.9	7.65	no
Esha	MECH	8.0	8.00	no
Farhan	CSE	8.6	8.50	yes
Gita	NULL	7.2	NULL	no

Gita: t.dept_id = NULL matches no row, so AVG is NULL

Figure 3.3.2 A correlated subquery runs once per outer row with that row's dept_id. CSE average = (9.1 + 7.8 + 8.6) / 3 = 8.50; ECE = (8.4 + 6.9) / 2 = 7.65. Result: Anu, Chitra, Farhan. Notice Esha: 8.0 > 8.00 is false.

A subquery may also sit in FROM, where its result acts as a temporary table (a **derived table**), or be named up front with `WITH name AS (SELECT …)` to keep a long query readable.

WORKED EXAMPLE

Question (10 marks). (a) Find the departments that have no students, first with NOT IN, then with NOT EXISTS, and explain why the answers differ. (b) Find the students who take every CSE course.

```
-- (a1)
SELECT dname FROM department
WHERE dept_id NOT IN (SELECT dept_id FROM student);
-- (a2)
SELECT d.dname FROM department d
WHERE NOT EXISTS (SELECT * FROM student s WHERE s.dept_id = d.dept_id);
```

1. The inner query of (a1) returns CSE, CSE, ECE, ECE, MECH, CSE, NULL (one value per student, Gita's NULL included).
2. For CIV, `NOT IN` means 'CIV' <> each value, all joined by AND. Six tests are true, but 'CIV' <> NULL is unknown, and true AND unknown is unknown. Unknown is not true, so CIV is dropped. CSE, ECE and MECH are in the list and are dropped too. **(a1) returns 0 rows.**
3. (a2) runs the inner query once per department. For CIV it looks for students with dept_id = 'CIV'. Gita's NULL = 'CIV' is unknown, so no row qualifies; the subquery is empty and NOT EXISTS is true. **(a2) returns Civil, 1 row**, the right answer.
4. Fix for (a1): add `WHERE dept_id IS NOT NULL` inside the subquery; it then also returns Civil.
5. (b) is "division": the CSE courses minus the student's courses must leave nothing.

```
SELECT s.name FROM student s
WHERE NOT EXISTS (
   SELECT course_id FROM course WHERE dept_id = 'CSE'
   EXCEPT                                        -- MINUS in Oracle
   SELECT course_id FROM enrols e WHERE e.roll_no = s.roll_no);
```

6. CSE courses = {CS201, CS202}. Anu takes {CS201, CS202}: difference empty, kept. Bala {CS201} and Esha {CS201} leave CS202; Chitra {CS202, EC201} and Farhan {CS202} leave CS201; Dev and Gita leave both. **Answer: Anu.**

EXAM TRAPS

- NOT IN over a subquery that can return NULL gives no rows at all. Prefer NOT EXISTS, or filter the NULLs out.
- `> ALL` of an empty set is true for every row; `> ANY` of an empty set is false.
- `<> ANY` is not NOT IN: it is true when the value differs from at least one member.
- UNION removes duplicates; write UNION ALL when the question counts every row.
- Calling every nested query "correlated": it is correlated only if the inner query uses a column of the outer row.

CHECK YOURSELF

– REMEMBER **Name the three SQL set operators, and Oracle's keyword for one of them.** 2 marks

UNION, INTERSECT, EXCEPT; Oracle writes EXCEPT as MINUS.

– UNDERSTAND **What makes a subquery correlated, and how often does it run?** 3 marks

It refers to a column of the outer query's current row, so logically it is evaluated once per outer row. A non-correlated subquery runs once and its result is reused.

– APPLY **How many rows do the CS201/CS202 roll-number queries give with UNION and with UNION ALL?** 2 marks

UNION: 5 (101, 102, 103, 105, 106). UNION ALL: 3 + 3 = 6, with 101 twice.

– APPLY **Give the output of `cgpa > ALL (SELECT cgpa FROM student WHERE dept_id = 'CSE')`, and of the same with ANY.** 4 marks

CSE CGPAs are {9.1, 7.8, 8.6}. ALL: above 9.1, **0 rows**. ANY: above 7.8, **Anu, Chitra, Esha, Farhan** (4 rows).

– APPLY **Rewrite "students enrolled in CS202" with EXISTS instead of IN.** 3 marks

`SELECT s.name FROM student s WHERE EXISTS (SELECT 1 FROM enrols e WHERE e.roll_no = s.roll_no AND e.course_id = 'CS202');` Output: Anu, Chitra, Farhan.

– ANALYZE **Why is `SELECT roll_no FROM enrols UNION SELECT name FROM student` refused, and how would you fix the intent?** 3 marks

The column types differ (number vs text), so the queries are not union-compatible. If the aim is "names of students who have enrolments", use a subquery or join instead of UNION.

BEYOND THE SYLLABUS

Not for the exam. Optimisers rarely run a correlated subquery row by row: they rewrite IN and EXISTS into semi-joins and NOT EXISTS into anti-joins, which is why both forms often cost the same (a later lesson covers query processing). The WITH clause has a recursive form (`WITH RECURSIVE` in PostgreSQL and MySQL) that walks hierarchies such as "all prerequisites of a course". Window functions (`AVG(cgpa) OVER (PARTITION BY dept_id)`) answer "above own department's average" without a subquery. Search next: "semi-join anti-join", "recursive CTE".

3.4 Aggregate functions and GROUP BY

An **aggregate function** turns a whole column of values into one value: how many, the total, the average, the smallest, the largest. **GROUP BY** first splits the rows into groups that share a value, so the aggregate gives one answer per group: average CGPA per department, students per course. **HAVING** then keeps only the groups that pass a test. Every report a college office prints, from pass percentages to fee collected per branch, is built this way.

You need: 3.2 Basic SQL queries and its data.

The five aggregate functions

On the 8 rows of enrols, whose marks add to 92 + 88 + 71 + 81 + 77 + 58 + 66 + 90 = 623:

FUNCTION	RETURNS	VALUE ON ENROLS
`COUNT(*)`	number of rows	8
`COUNT(DISTINCT course_id)`	number of different non-null values	3
`SUM(marks)`	total	623
`AVG(marks)`	SUM ÷ number of non-null values	623 / 8 = 77.875
`MIN(marks), MAX(marks)`	smallest, largest	58, 92

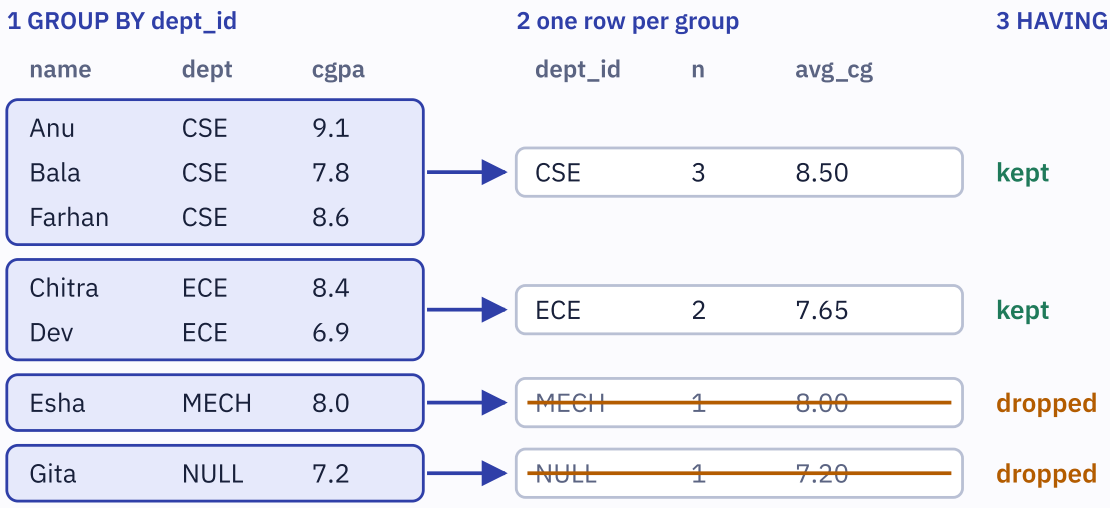
Two rules about NULL decide many exam answers. First, every aggregate except COUNT(*) **skips NULLs**: on student, `COUNT(*)` = 7 but `COUNT(dept_id)` = 6, because Gita's dept_id is NULL. Second, on an **empty** set of rows COUNT gives 0 and the others give NULL, not 0. MIN and MAX also work on text and dates (`MIN(name)` = 'Anu'); SUM and AVG need numbers.

GROUP BY and HAVING

```
SELECT   dept_id, COUNT(*) AS n, AVG(cgpa) AS avg_cg
FROM     student
GROUP BY dept_id
HAVING   COUNT(*) >= 2
ORDER BY avg_cg DESC;
```

The full logical order of a query is now **FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY**. WHERE throws away rows before any group exists. GROUP BY puts rows with equal dept_id together; all NULLs form one group of their own. The aggregates are then computed once per group, and HAVING throws away whole groups. Figure 3.4.1 follows the query above.

Because the result has one row per group, SELECT may show only the grouping columns, aggregates, and constants. `SELECT dept_id, name, AVG(cgpa) … GROUP BY dept_id` is refused: the CSE group has three names, and SQL will not guess which one to print.



Result after ORDER BY avg_cg DESC: CSE 3 8.50, then ECE 2 7.65

Figure 3.4.1 GROUP BY collapses each group into one row; HAVING removes whole groups. CSE: (9.1 + 7.8 + 8.6) / 3 = 8.50. ECE: (8.4 + 6.9) / 2 = 7.65. Notice that Gita's NULL forms its own group.

WHERE or HAVING?

Ask: does the test look at one row, or at a group? "Students with CGPA of at least 7" is a row test, so WHERE. "Departments whose average is above 8" is a group test, so HAVING. An aggregate inside WHERE, such as `WHERE AVG(cgpa) > 8`, is an error, because no groups exist yet when WHERE runs. A row test written in HAVING works on the grouping column but wastes effort, so put it in WHERE.

WORKED EXAMPLE

Question (8 marks). (a) For each course give the number of students and the average, highest and lowest marks. (b) Show only courses whose average is above 70. (c) What changes if `WHERE marks >= 60` is added? (d) For each department, count the students with CGPA of at least 7, showing only departments with at least 2 such students.

```
SELECT course_id, COUNT(*) AS n, ROUND(AVG(marks),2) AS avg_m,
       MAX(marks) AS hi, MIN(marks) AS lo
FROM enrols
GROUP BY course_id
HAVING AVG(marks) > 70;          -- leave HAVING out for part (a)
```

1. **(a)** Groups: CS201 {92, 71, 66}, CS202 {88, 81, 90}, EC201 {77, 58}. CS201: n = 3, sum 229, avg 229 / 3 = 76.33, hi 92, lo 66. CS202: n = 3, sum 259, avg 86.33, hi 90, lo 81. EC201: n = 2, sum 135, avg 67.50, hi 77, lo 58.
2. **(b)** HAVING AVG(marks) > 70 keeps **CS201 (76.33) and CS202 (86.33)**; EC201 (67.50) is dropped.
3. **(c)** WHERE runs first and removes the single row with 58. EC201 now holds only {77}: n = 1, avg 77.00, which passes HAVING. Result: **CS201 76.33, CS202 86.33, EC201 77.00**. One extra row filter changed which groups survive.
4. **(d)** `SELECT dept_id, COUNT(*) FROM student WHERE cgpa >= 7 GROUP BY dept_id HAVING COUNT(*) >= 2;` WHERE removes Dev (6.9). Groups: CSE 3, ECE 1 (Chitra), MECH 1, NULL 1. Answer: **CSE 3**, one row.

EXAM TRAPS

- COUNT(*) counts rows; COUNT(column) skips NULLs. On student they give 7 and 6.
- AVG skips NULLs: for marks 80, NULL, 60, AVG = 140 / 2 = 70, not 140 / 3 = 46.67.
- SUM over no rows is NULL, not 0. Write `COALESCE(SUM(x), 0)` when a zero is wanted.
- An aggregate in WHERE is an error; a group condition belongs in HAVING.
- Every non-aggregated column in SELECT must be in GROUP BY. (MySQL with its ONLY_FULL_GROUP_BY mode switched off accepts it and prints any one value; do not rely on that.)

CHECK YOURSELF

– REMEMBER **Name the five standard aggregate functions. Which form of COUNT includes NULL rows?** 2 marks

COUNT, SUM, AVG, MIN, MAX. COUNT(*) counts every row, NULLs included.

– UNDERSTAND **Explain the difference between WHERE and HAVING with one example each.** 4 marks

WHERE filters single rows before grouping and cannot use aggregates (`WHERE cgpa >= 7`). HAVING filters groups after grouping and usually uses aggregates (`HAVING AVG(cgpa) > 8`).

– APPLY **Give COUNT(*), COUNT(dept_id) and COUNT(DISTINCT dept_id) on student.** 3 marks

7, 6 and 3 (CSE, ECE, MECH; the NULL is not counted).

– APPLY **What does `SELECT block, SUM(budget) FROM department GROUP BY block` return?** 3 marks

A 900000; B 700000; C 500000 + 400000 = 900000. Three rows.

– APPLY **Write a query for the highest CGPA in each department, and give its output.** 3 marks

`SELECT dept_id, MAX(cgpa) FROM student GROUP BY dept_id;` CSE 9.1, ECE 8.4, MECH 8.0, NULL 7.2.

– ANALYZE **Find the department with the highest average CGPA without using LIMIT. Explain why `MAX(AVG(cgpa))` alone does not answer it.** 5 marks

`SELECT dept_id FROM student GROUP BY dept_id HAVING AVG(cgpa) >= ALL (SELECT AVG(cgpa) FROM student GROUP BY dept_id);` Averages are 8.50, 7.65, 8.00, 7.20, so the answer is **CSE**. Oracle accepts the nested `MAX(AVG(cgpa))` but returns only the number 8.50, not which department has it; most other systems refuse nested aggregates.

BEYOND THE SYLLABUS

Not for the exam. Reports often need subtotals and a grand total in one result; SQL's `GROUP BY ROLLUP (block, dept_id)`, `CUBE` and `GROUPING SETS` produce them, and data warehouses are built around such queries. Window functions keep every row while adding a group value beside it, for example each student's rank within the department. On very large data, engines offer approximate distinct counts (based on the HyperLogLog sketch) that are far cheaper than an exact COUNT(DISTINCT). Search next: "GROUP BY ROLLUP example", "SQL window functions RANK".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Logical order	FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY	3.2, 3.4
Set operations	UNION, INTERSECT, EXCEPT (MINUS); remove duplicates unless ALL	3.3
Subquery operators	= ANY ≡ IN; <> ALL ≡ NOT IN; NOT IN fails on a NULL; EXISTS = at least one row	3.3
Aggregates	COUNT(*) counts rows; others skip NULLs; AVG = SUM ÷ COUNT(column)	3.4
WHERE vs HAVING	rows before grouping vs groups after grouping	3.4

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Apply

– APPLY **Tables R(a) = {1, 2, 3} and S(a) = {2, 3, 4, NULL}. Give the result of R UNION S, R INTERSECT S, R EXCEPT S, and of `SELECT a FROM R WHERE a NOT IN (SELECT a FROM S)`.** 8 marks

UNION: {1, 2, 3, 4, NULL} (5 rows). INTERSECT: {2, 3}. EXCEPT: {1}. NOT IN: for 1, 1 <> NULL is unknown, so **no rows**; with NOT EXISTS the answer is {1}. Set operations treat NULLs as equal to each other when removing duplicates; comparisons do not.

Analyze and evaluate

– ANALYZE **Write "departments with no students" with NOT IN, NOT EXISTS and an outer join. Which are correct when student.dept_id can be NULL?** 12 marks

NOT IN returns no rows because of Gita's NULL; NOT EXISTS returns Civil; `department LEFT JOIN student … WHERE s.roll_no IS NULL` returns Civil. NOT EXISTS and the outer join are correct; NOT IN needs `WHERE dept_id IS NOT NULL` in the subquery. Mention three-valued logic, and that optimisers run NOT EXISTS as an anti-join. Spans Chapters 3.3 and 3.5.

– ANALYZE **Find the students who take every CSE course in two ways: NOT EXISTS with EXCEPT, and GROUP BY with HAVING COUNT. Compare them.** 10 marks

Method 1 as in Chapter 3.3: Anu. Method 2: join enrols with course WHERE dept_id = 'CSE', GROUP BY roll_no HAVING COUNT(DISTINCT course_id) = (SELECT COUNT(*) FROM course WHERE dept_id = 'CSE'): Anu (2 = 2). Method 1 follows the definition of division; method 2 is easier to read but relies on counting distinct courses correctly.

Glossary (terms used in this lesson)

Aggregate function	A function that returns one value for a set of rows: COUNT, SUM, AVG, MIN, MAX.
Correlated subquery	A subquery that uses a column of the outer query's current row, so it is evaluated once per outer row.
Derived table	A subquery in FROM whose result is used as a table.
GROUP BY	The clause that splits rows into groups of equal values for aggregation.
HAVING	The clause that keeps or drops whole groups after GROUP BY.
Scalar subquery	A subquery that returns exactly one value.

Union-compatible	Two queries with the same number of columns and matching types, as set operations require.

Sources and further reading

Textbooks

- Abraham Silberschatz, Henry F. Korth and S. Sudarshan, *Database System Concepts*, 7th edition, McGraw-Hill: the chapters Introduction to SQL, Intermediate SQL (joins, views, transactions, integrity constraints) and Advanced SQL (functions, procedures, triggers).
- Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems*, 7th edition, Pearson: the chapters on basic SQL and on complex queries, triggers and views.
- Raghu Ramakrishnan and Johannes Gehrke, *Database Management Systems*, 3rd edition, McGraw-Hill: the chapter on SQL queries, constraints and triggers.
- Ivan Bayross, *SQL, PL/SQL: The Programming Language of Oracle*, BPB Publications: the Oracle lab book many Indian colleges use.
- Steven Feuerstein with Bill Pribyl, *Oracle PL/SQL Programming*, O'Reilly.

Open resources

- Oracle Database documentation: SQL Language Reference and PL/SQL Language Reference, docs.oracle.com/en/database.
- PostgreSQL documentation, part "The SQL Language", postgresql.org/docs/current.
- MySQL Reference Manual, CHECK constraints (enforced from 8.0.16), dev.mysql.com (fact checked through a web-search summary; page not opened).
- SQLite, "SQL As Understood By SQLite", sqlite.org/lang.html: a free database for trying every query in this book.
- Companion site of *Database System Concepts*, db-book.com, with slides and sample university data.

Syllabus: the University free n easy universal syllabus, built from top universities' syllabi.

Built for the University free n easy YouTube channel from the universal syllabus of Database Management Systems (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.