DATABASE ENGINEERING

Unit 3

Relational Algebra, Relational Calculus and SQL

Topic-wise study notes with importance ratings

36 topics — every relational algebra operator, both calculi, and the full SQL command set

Diagrams and worked tables are given wherever a concept needs one

Topic-wise study notes. Every topic carries an importance rating from one to three stars: three stars means the topic is asked almost every year and must be known perfectly, two stars means it is regularly asked and should be prepared, one star means it is supporting material to read once. The notes explain the concept only; worked questions and answers are kept in a separate file.

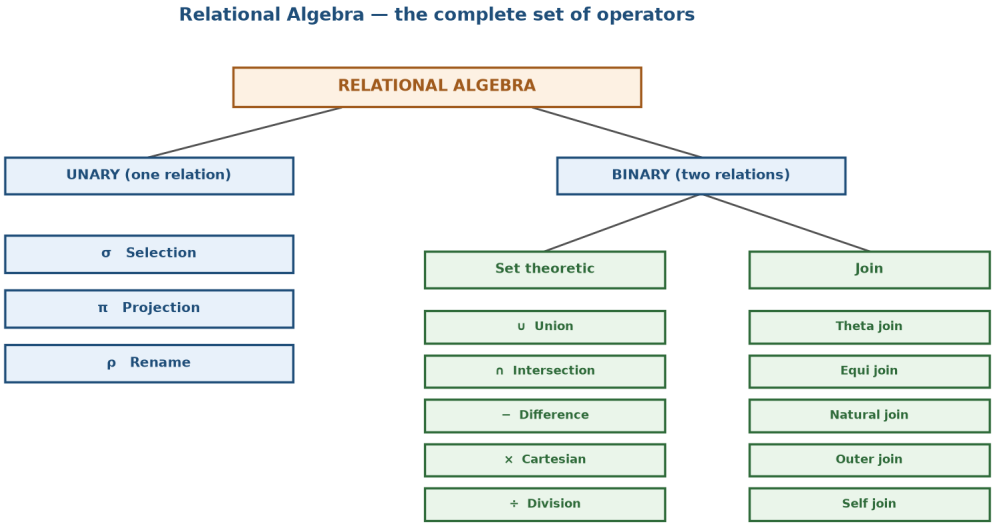
Index

Topic, importance rating and page number. ★★★ must know, ★★ should know, ★ read once.

Topic	Importance	Page
3.1 Relational Algebra — Introduction	★★★	4
3.2 Selection (σ)	★★★	4
3.3 Projection (π)	★★★	5
3.4 Rename (ρ)	★★	5
3.5 Union (∪) and Union Compatibility	★★★	6
3.6 Intersection (∩)	★★	6
3.7 Set Difference (−)	★★	6
3.8 Cartesian Product (×)	★★	7
3.9 Division (÷)	★★★	7
3.10 Theta Join (⋈θ)	★★★	8
3.11 Equi Join	★★★	8
3.12 Natural Join (⋈)	★★★	8
3.13 Outer Joins	★★★	9
3.14 Self Join	★★★	9
3.15 Multi-attribute Join	★★	10
3.16 Writing a Query — the Method	★★★	10
3.17 Relational Calculus — Introduction	★★	10
3.18 Tuple Relational Calculus (TRC)	★★★	11
3.19 Domain Relational Calculus (DRC)	★★	11
3.20 Quantifiers and Query Patterns	★★	12
3.21 Safe and Unsafe Expressions	★★	12
3.22 Relational Algebra vs Relational Calculus	★★	12
3.23 SQL — Introduction and Features	★★★	13
3.24 SQL Data Types and Constraints	★★	13
3.25 DDL — CREATE, ALTER, DROP, TRUNCATE	★★★	14
3.26 DML — INSERT, UPDATE, DELETE	★★★	14
3.27 DQL — SELECT with WHERE, LIKE, BETWEEN	★★★	14
3.28 ORDER BY, GROUP BY and HAVING	★★★	15
3.29 Aggregate Functions	★★	15
3.30 Joins in SQL	★★★	16
3.31 Subqueries	★★	16
3.32 Views	★★	16
3.33 DCL — GRANT and REVOKE	★★	17
3.34 TCL — COMMIT, ROLLBACK, SAVEPOINT	★★	17

Topic	Importance	Page
3.35 DROP vs TRUNCATE vs DELETE	★★★	17
3.36 QBE — Query By Example	★★	18

3.1 Relational Algebra — Introduction ★★★



Relational algebra is a **procedural** query language: each operation takes one or two relations as input and produces a new relation as output. Because every operator is closed over relations, operations can be nested to build complex queries.

Property	Meaning
Procedural	The order of operations is fixed by the expression
Closed	The output of every operator is itself a relation
Foundation of SQL	Every SQL query is translated into an algebra tree
Used by the optimiser	The tree is rewritten to find a cheaper plan

The two families of operators

Unary operators act on one relation — selection, projection and rename. **Binary** operators act on two — union, intersection, difference, Cartesian product, division and the joins. All binary operators except the product require union compatibility.

Why it matters in practice

Relational algebra is not a language a user types; it is the internal form the DBMS works with. Understanding it explains why two SQL queries that look different can produce the same plan, and why one of them is faster.

3.2 Selection (σ) ★★★

Selection returns the tuples of a relation that satisfy a condition. It works on rows and does not change the degree.

Syntax

$\sigma_{condition}(R)$ — for example $\sigma_{Salary > 50000}(EMPLOYEE)$, or $\sigma_{Dept='CSE' \wedge CGPA>8}(STUDENT)$.

The condition

A condition may use the comparison operators $=, \neq, <, >, \leq, \geq$ combined with AND ($\wedge$), OR ($\vee$) and NOT ($\neg$). Conditions can compare an attribute with a constant or with another attribute of the same tuple.

Properties

- Commutative with itself: $\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$.
- Distributes over union, intersection and difference.
- Pushing a selection down to the leaves of a query tree is the most common optimisation, because it shrinks the inputs before any join.

3.3 Projection (π) ★★★

Projection returns only the named attributes, discarding the rest. It works on columns and may reduce the degree.

Syntax

$\pi_{A1, A2, ..., An}(R)$ — for example $\pi_{Name, Salary}(EMPLOYEE)$.

Duplicates are removed

Because a relation is a set, duplicate tuples created by a projection are eliminated automatically. In SQL the same result needs the keyword DISTINCT, which is why π and SELECT are not exactly the same operation.

Combining with selection

$\pi_{Name}(\sigma_{Dept='CSE'}(STUDENT))$ first filters the rows and then keeps one column. The two operators are independent: selection reduces the number of tuples, projection reduces the number of attributes.

3.4 Rename (ρ) ★★

Rename changes the name of a relation, of its attributes, or both. It produces no new data; it only makes an expression unambiguous.

Syntax

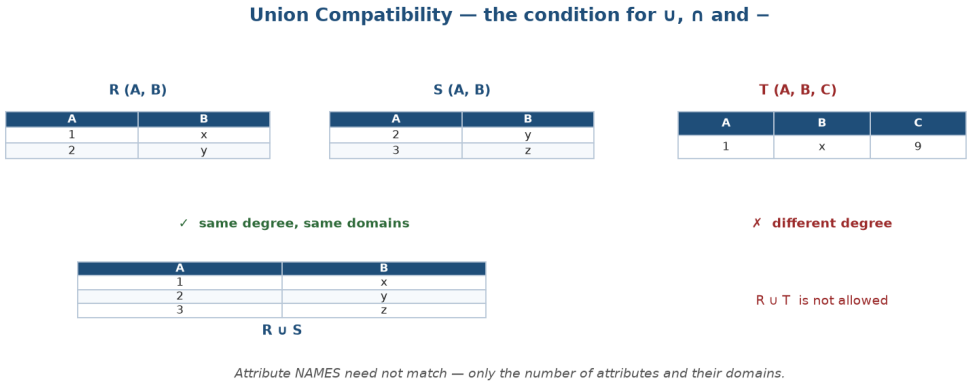
$\rho_S(R)$ renames the relation R to S.

$\rho_{S(B1,B2)}(R)$ renames the relation and its attributes as well.

Why it is needed

The same relation cannot appear twice in one expression under the same name. A self join therefore needs two renamed copies: $\rho_E(EMPLOYEE)$ and $\rho_M(EMPLOYEE)$. Rename is also used to give a meaningful name to an intermediate result, or to make two relations union compatible when their attribute names differ.

3.5 Union (∪) and Union Compatibility ★★★



Union returns the tuples that appear in R or in S or in both. Duplicates are removed because the result is a set.

Union compatibility

Two relations are union compatible when they have the **same degree** and the corresponding attributes are drawn from the **same domains**. Union, intersection and difference are defined only for such relations. The attribute names need not match — only the number and the domains.

Example

CSE_STUDENTS(Name, CGPA) ∪ AIDS_STUDENTS(Name, CGPA) gives all students of both departments. If the second relation were AIDS_STUDENTS(Name, CGPA, Hostel), the union would be invalid.

3.6 Intersection (∩) ★★

Intersection returns the tuples that appear in **both** R and S. Like union, it requires union compatibility.

Example

STUDENT(StudentID) ∩ CRICKET_TEAM(StudentID) gives the students who are in both the cricket team and the general student list. Intersection can be expressed with difference: R ∩ S = R − (R − S).

3.7 Set Difference (−) ★★

Set difference returns the tuples that appear in R but **not** in S. It requires union compatibility, and unlike union and intersection it is **not commutative** — R − S and S − R are different.

Example

STUDENT(StudentID) − APPEARED(StudentID) gives the students who registered but did not appear for the examination.

Caution

Difference is the operator used to express "not" and "without". Most mistakes in relational algebra questions come from reversing the two operands, so read the question carefully — "students who did not appear" means STUDENT − APPEARED, not the other way round.

3.8 Cartesian Product (×) ★★

Cartesian product pairs every tuple of R with every tuple of S. The degree of the result is the sum of the two degrees, and the cardinality is the product of the two cardinalities.

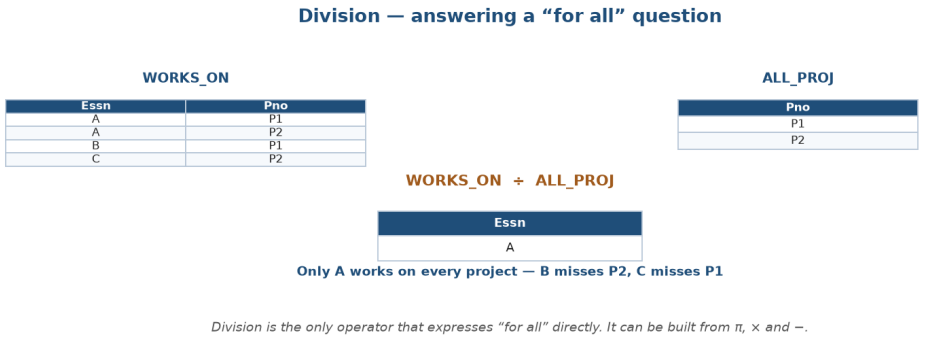
Example

If STUDENT has 4 tuples of degree 2 and COURSE has 3 tuples of degree 3, then STUDENT × COURSE has 12 tuples of degree 5.

When it is used

Almost never on its own, because most pairs are meaningless. It is a building block: a join is a product followed by a selection, and division can be expressed using the product. Writing a query as a product plus a selection is the standard way to answer a question when no simpler operator fits.

3.9 Division (÷) ★★★



Division is the only operator that expresses “for all” directly. It can be built from π, × and −.

Division answers "for all" questions. If R(X, Y) ÷ S(Y) is performed, the result contains every value of X in R that is paired with **every** value of Y in S.

Example

WORKS_ON(Essn, Pno) ÷ ALL_PROJ(Pno) gives the employees who work on every project in ALL_PROJ. An employee who works on three of four projects is excluded.

Building it from other operators

$R \div S = \pi_X(R) - \pi_X((\pi_X(R) \times S) - R)$. The idea is: take all candidate X values, pair them with every Y, subtract the pairs that actually exist, and whatever X remains is the one that was missing some pair.

Why it is examinable

Division is the only operator that expresses the universal quantifier directly, so any question containing "all", "every" or "each" in a relational algebra context is a division question.

3.10 Theta Join (⋈θ) ★★★

Natural, Equi and Theta Join on the same two relations

EMPLOYEE (EmpID, Name, DeptID)

EmpID	Name	DeptID
E1	Amit	D1
E2	Priya	D2
E3	Rahul	D1

DEPARTMENT (DeptID, DeptName)

DeptID	DeptName
D1	CSE
D2	AIDS

NATURAL JOIN — common column once

EmpID	Name	DeptID	DeptName
E1	Amit	D1	CSE
E2	Priya	D2	AIDS
E3	Rahul	D1	CSE

EQUI JOIN — join column twice

EmpID	Name	E.DeptID	D.DeptID	DeptName
E1	Amit	D1	D1	CSE
E2	Priya	D2	D2	AIDS

THETA JOIN — any comparison, here on a salary band

Name	Salary	Grade
Amit	32000	G1
Priya	48000	G2

A natural join is an equi join on all same-named attributes, with the duplicate column removed.

A **theta join** combines tuples from two relations that satisfy any comparison condition θ. The condition may use =, ≠, <, >, ≤ or ≥.

Syntax

$R \bowtie_{\theta} S$ — for example EMPLOYEE $\bowtie_{\text{Salary} \geq \text{MinSalary} \wedge \text{Salary} \leq \text{MaxSalary}}$ SALARY_GRADE, which assigns each employee the salary grade band his salary falls into.

When it is required

Whenever the match is a **range** rather than an exact value. A natural or equi join cannot express "salary lies between the minimum and the maximum of a grade".

3.11 Equi Join ★★★

An **equi join** is a theta join whose only condition is equality, written explicitly.

Syntax

$R \bowtie_{R.A = S.B} S$ — for example EMPLOYEE $\bowtie_{\text{EMPLOYEE.DeptID} = \text{DEPARTMENT.DeptID}}$ DEPARTMENT.

The duplicate column

Because the condition names both attributes, both appear in the result. The join column therefore appears **twice**, which is the visible difference from a natural join. A projection can remove the duplicate if it is not wanted.

3.12 Natural Join (⋈) ★★★

A **natural join** joins on all attributes that have the **same name** in both relations, and keeps only **one** copy of each common attribute.

Syntax

$R \bowtie S$ — the condition is implicit.

When it works

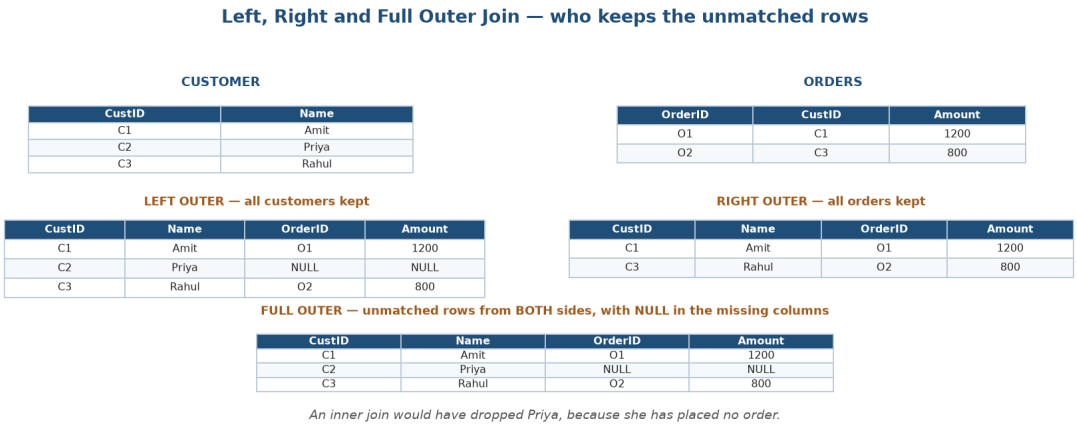
Only when the foreign key and the referenced primary key share a name. If they do not — for example EMPLOYEE.Dno and DEPARTMENT.Dnumber — a natural join produces a Cartesian product, which is the classic trap in exam questions.

The relationship between the three

Join	Condition	Duplicate column
Theta	Any comparison	Kept
Equi	Equality, written explicitly	Kept
Natural	Equality on same-named attributes	Removed

A natural join is therefore an equi join with the duplicate column dropped.

3.13 Outer Joins ★★★



An **outer join** keeps the tuples that have no match, filling the missing attributes with NULL.

Join	Preserves	Use when
Left outer (R ⟕ S)	Every tuple of R	You want all rows of the left table, with matches if they exist
Right outer (R ⟖ S)	Every tuple of S	You want all rows of the right table
Full outer (R ⟗ S)	Both sides	You want everything from both tables

Why they exist

An inner join silently drops unmatched rows. A report such as "all customers and their orders, including customers who have never ordered" would be wrong with an inner join, because those customers would vanish from the result. That is the whole reason for outer joins.

3.14 Self Join ★★★

A **self join** joins a relation with itself. Because the same relation appears twice, both copies must be renamed.

Syntax

$$\pi_{E.EmpName \to Employee,\ M.EmpName \to Manager}(\rho_E(EMPLOYEE) \bowtie_{E.ManagerID = M.EmpID} \rho_M(EMPLOYEE))$$

When it is needed

Whenever a recursive relationship has been stored in one table — an employee and his manager, a course and its prerequisite, a part and its sub-part.

The NULL caveat

The top-level manager has ManagerID = NULL, so an inner self join drops him. A **left outer** self join is needed to keep him in the result.

3.15 Multi-attribute Join ★★

When two relations share a **composite** key, the join condition must name every attribute of it.

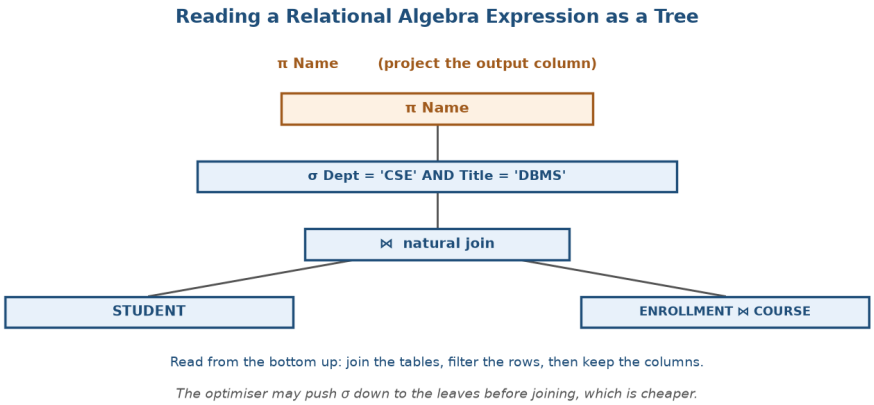
Example

$$\text{RESULT} \bowtie_{\text{RESULT.StudentID} = \text{ATTENDANCE.StudentID} \wedge \text{RESULT.CourseID} = \text{ATTENDANCE.CourseID}} \text{ATTENDANCE}$$

Why one attribute is not enough

Matching on StudentID alone would pair marks from one course with attendance from another, producing rows that never existed. This is the same mistake as a lossy join, and it is worth stating explicitly in an answer.

3.16 Writing a Query — the Method ★★★



The three steps

1. **Find the relations** that hold the required attributes. 2. **Join** them on their common keys, one join per relationship. 3. **Apply σ** for the conditions and **π** for the output columns.

Example

Names of CSE students enrolled in DBMS:

$$\pi_{\text{Name}}(\sigma_{\text{Dept='CSE'} \wedge \text{Title='DBMS'}}((\text{STUDENT} \bowtie \text{ENROLLMENT}) \bowtie \text{COURSE}))$$

The optimisation

The optimiser may push $\sigma_{\text{Dept='CSE'}}$ down to STUDENT and $\sigma_{\text{Title='DBMS'}}$ down to COURSE before joining. The result is identical but the joins operate on far fewer rows, which is why selections are always pushed as low in the tree as possible.

3.17 Relational Calculus — Introduction ★★

Relational calculus is a **declarative** query language based on predicate logic. A query describes the properties the required tuples or values must satisfy, rather than the sequence of operations to obtain them.

The two forms

Form	Variables stand for	Example shape
TRC (tuple relational calculus)	Whole tuples	{ t such that P(t) }
DRC (domain relational calculus)	Individual values	{ ⟨x1, x2⟩ such that P(...) }

Expressiveness

Relational algebra and relational calculus have **equivalent expressive power** for safe expressions. Algebra says *how*, calculus says *what*, and SQL sits on top of both as a practical language with extra features.

3.18 Tuple Relational Calculus (TRC) ★★★

In TRC a variable stands for a whole tuple, and the result is written as a set of tuples satisfying a predicate.

The form

{ t such that P(t) }, where t is a tuple variable and P(t) is the condition. Attributes are accessed as t.Name.

Worked queries

Requirement	TRC expression
All CSE students	{ s such that s ∈ STUDENT ∧ s.Dept = 'CSE' }
Names with CGPA above 8.5	{ t such that ∃s (s ∈ STUDENT ∧ s.CGPA > 8.5 ∧ t.Name = s.Name) }
Students enrolled in at least one course	{ s such that s ∈ STUDENT ∧ ∃e (e ∈ ENROLLMENT ∧ e.StudentID = s.StudentID) }
Students not enrolled in any course	{ s such that s ∈ STUDENT ∧ ¬∃e (e ∈ ENROLLMENT ∧ e.StudentID = s.StudentID) }

The membership condition

The clause s ∈ STUDENT is what restricts the variable to existing tuples. Forgetting it is the most common mistake in a TRC answer, because without it the expression is unbounded and unsafe.

3.19 Domain Relational Calculus (DRC) ★★

In DRC a variable stands for an individual attribute value, and the result is a set of value tuples.

The form

{ ⟨x1, x2, ..., xn⟩ such that P(...) }, where each xi is a domain variable. A relation is written as a predicate, such as STUDENT(sid, name, dept, cgpa).

Worked queries

Requirement	DRC expression
Names of CSE students	{ ⟨name⟩ such that ∃sid ∃cgpa (STUDENT(sid, name, 'CSE', cgpa)) }

Requirement	DRC expression
Names and course titles	{ ⟨name, title⟩ such that ∃sid ∃dept ∃cgpa ∃cid ∃grade ∃cdept (STUDENT(sid,name,dept,cgpa) ∧ ENROLLMENT(sid,cid,grade) ∧ COURSE(cid,title,cdept)) }

How the relations are connected

By using the **same variable** in several predicates. The sid that appears in STUDENT and in ENROLLMENT is what performs the join, without any join operator being written.

3.20 Quantifiers and Query Patterns ★★

English phrase	Calculus form	Note
at least one, any	∃	Existential quantifier
every, all, each	∀	Usually written with ⇒
none, not any	¬∃	Negation applied to the existential
not all	¬∀	Rare, but appears in tricky questions

The universal pattern

Students enrolled in every required course: { s such that s ∈ STUDENT ∧ ∀r (r ∈ REQUIRED ⇒ ∃e (e ∈ ENROLLMENT ∧ e.StudentID = s.StudentID ∧ e.CourseID = r.CourseID)) }

Why implication is used

"Every required course" does not mean every tuple in the database; it means every tuple that is a required course. The implication restricts the quantifier to the relevant rows, and writing ∧ instead of ⇒ here is a classic error.

3.21 Safe and Unsafe Expressions ★★

A **safe** expression produces only values drawn from the **active domain** of the database, so the result is always finite. An **unsafe** expression can describe an infinite set and therefore cannot be evaluated.

Safe	Unsafe
{ s such that s ∈ STUDENT ∧ s.Dept = 'CSE' }	{ t such that ¬(t ∈ STUDENT) }
Result drawn from stored values	Would need every tuple in the universe
Finite, computable	Infinite, meaningless

The test to apply

Ask whether every output value can be traced back to a value stored in the database. If it can, the expression is range-restricted and safe.

3.22 Relational Algebra vs Relational Calculus ★★

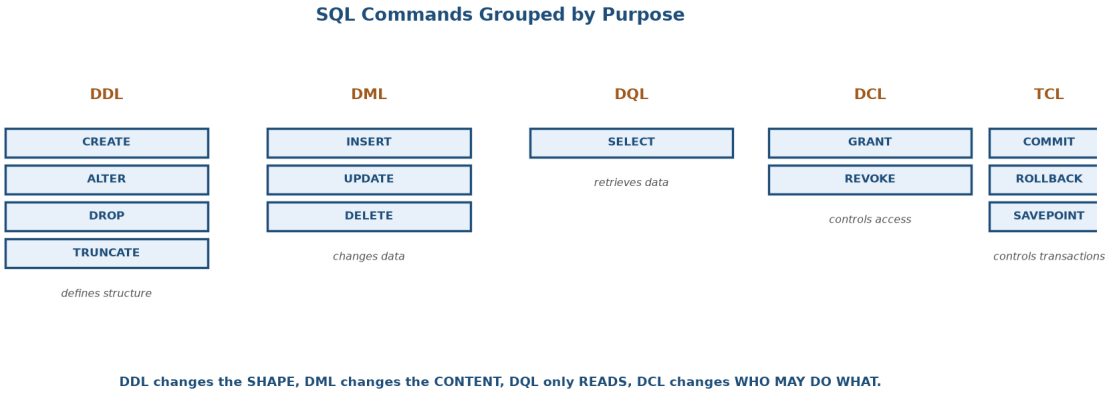
Relational Algebra	Relational Calculus
Procedural — says how to compute	Declarative — says what is required
Operators: σ, π, ⋈, ∪, −, ×	Predicates, ∧, ∨, ¬, ∃, ∀
Variables are not central	Tuple or domain variables are central

Relational Algebra	Relational Calculus
Order of operations matters	Order decided by the DBMS
Equivalent power for safe queries	Equivalent power for safe queries

The practical difference

Algebra is what the DBMS executes after optimisation; calculus is what a query language is specified in. SQL borrows its syntax from algebra and its semantics from calculus, which is why the same query can be written either way.

3.23 SQL — Introduction and Features ★★★



SQL (Structured Query Language) is the standard language for relational databases. It is declarative: the user states the result required, and the DBMS decides how to compute it.

Feature	Meaning
Declarative	Say what you want, not how to get it
Non-procedural	No loops or navigation in the basic language
Standardised	ANSI and ISO standards; most DBMSs add extensions
Set-oriented	Operates on sets of rows, not one row at a time
Embeddable	Can be used inside a host language program

The five groups of commands

DDL defines structure, DML changes data, DQL retrieves data, DCL controls access, and TCL controls transactions. Confusing DROP with DELETE — one removes the table, the other removes rows — is the most common error in a short answer.

3.24 SQL Data Types and Constraints ★★

Category	Common types
Numeric	INT, SMALLINT, DECIMAL(p,s), NUMERIC, FLOAT
Character	CHAR(n), VARCHAR(n), TEXT
Date and time	DATE, TIME, TIMESTAMP
Large objects	BLOB, CLOB

Constraint	Purpose
PRIMARY KEY	Uniqueness plus NOT NULL
UNIQUE	Uniqueness, NULL usually allowed
FOREIGN KEY ... REFERENCES	Referential integrity
NOT NULL	Value must be present
CHECK (condition)	Domain or row-level rule
DEFAULT	Value used when none is supplied

3.25 DDL — CREATE, ALTER, DROP, TRUNCATE ★★★

DDL defines and modifies the **structure** of database objects. It updates the data dictionary and is normally auto-committed.

Statement	Syntax	Purpose
CREATE TABLE	CREATE TABLE Emp (EmpID INT PRIMARY KEY, Name VARCHAR(30) NOT NULL, Salary NUMBER(10,2), DeptID INT REFERENCES Dept(DeptID));	Creates a table with constraints
ALTER TABLE	ALTER TABLE Emp ADD (Phone VARCHAR(15));	Adds, modifies or drops a column
DROP TABLE	DROP TABLE Emp;	Removes the structure and the data
TRUNCATE	TRUNCATE TABLE Emp;	Removes all rows, keeps the structure
CREATE INDEX	CREATE INDEX idx_name ON Emp(Name);	Builds an index
CREATE VIEW	CREATE VIEW CSE AS SELECT * FROM Emp WHERE DeptID = 10;	Defines a virtual table

3.26 DML — INSERT, UPDATE, DELETE ★★★

DML changes the **data** inside the tables. It participates in transactions, so it can be rolled back.

Statement	Syntax	Purpose
INSERT one row	INSERT INTO Dept VALUES (10, 'HR');	Adds a row
INSERT many rows	INSERT INTO Dept VALUES (20,'IT'), (30,'Finance');	Adds several rows
UPDATE	UPDATE Emp SET Salary = Salary * 1.10;	Modifies existing rows
DELETE	DELETE FROM Emp WHERE Salary < 50000;	Removes selected rows

Caution

An UPDATE or DELETE without a WHERE clause affects **every** row. In an exam answer, always show the WHERE clause even when the question says "all rows", and say explicitly that the statement affects the whole table.

3.27 DQL — SELECT with WHERE, LIKE, BETWEEN ★★★

DQL retrieves data and has one statement, SELECT.

The general form

SELECT columns FROM tables WHERE row-condition GROUP BY columns HAVING group-condition ORDER BY columns;

The WHERE clause

Operator	Meaning	Example
=, <, >, <=, >=, <>	Comparison	WHERE Salary > 50000
BETWEEN ... AND ...	Inclusive range	WHERE Salary BETWEEN 70000 AND 90000
IN (list)	Matches any value in the list	WHERE Dept IN ('HR','IT')
LIKE	Pattern match, % for any string, _ for one character	WHERE Name LIKE 'S%'
IS NULL / IS NOT NULL	Tests for NULL	WHERE Phone IS NULL
AND, OR, NOT	Combine conditions	WHERE Dept='CSE' AND CGPA>8

The LIKE pattern

'%' matches any number of characters and '_' matches exactly one. So Name LIKE 'M_____' matches a seven-character name beginning with M, while Name LIKE 'M%' matches any name beginning with M.

3.28 ORDER BY, GROUP BY and HAVING ★★★

Clause	Purpose	Example
ORDER BY	Sorts the result, ASC or DESC	ORDER BY Salary DESC
GROUP BY	Groups rows that share a value, so aggregates apply per group	GROUP BY DeptID
HAVING	Filters **groups** after grouping	HAVING AVG(Salary) > 50000

The difference between WHERE and HAVING

WHERE filters **rows before** grouping and cannot contain an aggregate. HAVING filters **groups after** grouping and usually does. A query that needs both writes WHERE first, then GROUP BY, then HAVING.

Order of evaluation

FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY. This order explains why a column alias defined in SELECT cannot be used in WHERE, but can be used in ORDER BY.

3.29 Aggregate Functions ★★

Function	Returns
COUNT(*)	Number of rows, including those with NULL
COUNT(column)	Number of non-NULL values in that column
SUM(column)	Total of the non-NULL values
AVG(column)	Average of the non-NULL values
MIN(column)	Smallest non-NULL value
MAX(column)	Largest non-NULL value

The NULL rule

Every aggregate except COUNT(*) **ignores NULLs**. This is why AVG(Salary) divides by the number of employees whose salary is known, not by the total number of employees — a distinction that changes the answer whenever a column is nullable.

3.30 Joins in SQL ★★★

Join	SQL form
Inner join	SELECT * FROM E JOIN D ON E.DeptID = D.DeptID;
Natural join	SELECT * FROM E NATURAL JOIN D;
Left outer	SELECT * FROM E LEFT JOIN D ON E.DeptID = D.DeptID;
Right outer	SELECT * FROM E RIGHT JOIN D ON E.DeptID = D.DeptID;
Full outer	SELECT * FROM E FULL OUTER JOIN D ON E.DeptID = D.DeptID;
Self join	SELECT a.Name, b.Name FROM Emp a JOIN Emp b ON a.ManagerID = b.EmpID;

Old and new syntax

The comma form SELECT * FROM E, D WHERE E.DeptID = D.DeptID is an inner join written with a Cartesian product and a selection. The explicit JOIN ... ON form is preferred because it separates the join condition from the filter and makes an outer join possible.

3.31 Subqueries ★★

A **subquery** is a SELECT nested inside another statement. It may return a single value, a single column, or a whole table.

Type	Operator	Example
Scalar subquery	comparison	WHERE Salary > (SELECT AVG(Salary) FROM Emp)
Multi-row subquery	IN, ANY, ALL	WHERE DeptID IN (SELECT DeptID FROM Dept WHERE Location='Delhi')
Correlated subquery	EXISTS, NOT EXISTS	WHERE EXISTS (SELECT 1 FROM Emp e WHERE e.DeptID = d.DeptID)

Correlated versus simple

A **simple** subquery is evaluated once and its result is reused. A **correlated** subquery references the outer query, so it is evaluated once per outer row — more expensive, but it is the natural way to express "for all" and "not exists".

3.32 Views ★★

A **view** is a stored query that behaves like a table. It holds no data of its own; the DBMS evaluates the query each time the view is used.

Advantage	Explanation
Security	A user can be restricted to a view that hides sensitive columns or rows
Simplicity	A complex join can be presented as a single simple table
Independence	A view can hide a change in the base tables
Reusability	The same definition is shared by many queries

Updatability

A simple view on one table can usually be updated, and the change is passed to the base table. A view involving joins, aggregates or DISTINCT is normally read-only, because there is no unambiguous way to translate an update back to the base tables.

3.33 DCL — GRANT and REVOKE ★★

DCL controls **who may do what**. It is how the DBA implements the security policy without changing any program.

Statement	Syntax	Effect
GRANT	GRANT SELECT, INSERT ON Emp TO user1;	Gives privileges
GRANT with option	GRANT SELECT ON Emp TO user1 WITH GRANT OPTION;	Allows user1 to pass the privilege on
REVOKE	REVOKE SELECT ON Emp FROM user1;	Takes the privilege back

Levels of privilege

Privileges may be granted on a table, on a view, or on individual columns. A view is the usual way to give a user access to part of a table, because column-level grants are supported unevenly across DBMSs.

3.34 TCL — COMMIT, ROLLBACK, SAVEPOINT ★★

TCL controls the boundaries of a transaction.

Statement	Effect
COMMIT	Makes all changes of the transaction permanent
ROLLBACK	Undoes all changes since the last commit
SAVEPOINT name	Marks a point inside the transaction to which you can return
ROLLBACK TO name	Undoes only the work after that savepoint

Why it matters

Without an explicit COMMIT, many DBMSs run in autocommit mode and each statement is committed as it executes, which makes a partial failure impossible to undo. Grouping the statements and committing once is what makes atomicity real.

3.35 DROP vs TRUNCATE vs DELETE ★★★

Aspect	DELETE	TRUNCATE	DROP
Removes	Selected rows (or all rows)	All rows	The table itself
Structure kept	Yes	Yes	No
WHERE clause	Allowed	Not allowed	Not allowed
Rollback	Yes	Usually not	No
Speed	Slow, row by row	Very fast	Fast
Triggers fired	Yes	No	No
Belongs to	DML	DDL	DDL

The one-line answer

DELETE removes **rows**, TRUNCATE empties the **table**, DROP removes the **table**.

3.36 QBE — Query By Example ★★

QBE is a visual, form-based query language. The user is shown a skeleton table for each relation and fills it in: a value means "match this value", a condition means "satisfy this condition", and a variable prefixed with P. means "print this column".

How a query is written

Step	What the user does
1	Opens the skeleton tables for the relations involved
2	Types a constant to filter on that value
3	Uses the same example variable in two tables to join them
4	Writes P.column to include a column in the output
5	Uses operators such as _x for "not equal" or a condition box for ranges

Where it is used

QBE was designed for non-programmers and survives as a graphical front end in some desktop database products. It is in the syllabus as a concept, and a short answer needs only the idea, the P. notation and one example of a join written with a shared variable.