```
SELECT column1, column2 FROM table_name
WHERE condition
GROUP BY column
HAVING aggregate_condition
ORDER BY column ASC|DESC
LIMIT n;
```

Clause	Purpose	Example
WHERE	Filter rows before grouping	WHERE salary > 50000
GROUP BY	Aggregate rows by column value	GROUP BY department
HAVING	Filter groups after aggregation	HAVING COUNT(*) > 3
ORDER BY	Sort results	ORDER BY name ASC
LIMIT	Restrict number of rows returned	LIMIT 10

```
UPDATE employees
SET salary = salary * 1.10, dept = 'Senior IT'
WHERE dept = 'IT' AND years_exp >= 5;
```

```
INSERT INTO employees (name, dept, salary)
VALUES ('Alice', 'HR', 60000);
```

```
INSERT ALL
   INTO t (col1, col2, col3) VALUES ('val1_1', 'val1_2', 'val1_3')
   INTO t (col1, col2, col3) VALUES ('val2_1', 'val2_2', 'val2_3')
   INTO t (col1, col2, col3) VALUES ('val3_1', 'val3_2', 'val3_3')
SELECT 1 FROM DUAL;
```

```
DELETE FROM orders
WHERE customer_id IN (
 SELECT customer_id FROM customers WHERE region = 'Inactive'
);
```

```
SELECT e.name, d.dept_name
FROM employees e
INNER JOIN departments d ON e.dept_id = d.dept_id;

SELECT c.name, o.order_date
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

JOIN Type	Returns	Use Case
INNER JOIN	Only matching rows from both tables	Orders with known customers
LEFT JOIN	All rows from left + matching right	All customers, with or without orders
RIGHT JOIN	All rows from right + matching left	All orders, even with unknown customers
FULL OUTER JOIN	All rows from both tables	Complete combined dataset
CROSS JOIN	Cartesian product of both tables	All possible combinations
SELF JOIN	A table joined to itself	Employee-Manager hierarchy

Function	Description	Example
COUNT(*)	Count number of rows	SELECT COUNT(*) FROM employees
SUM(col)	Sum of values	SELECT SUM(salary) FROM employees
AVG(col)	Average value	SELECT AVG(salary) FROM dept
MAX(col)	Maximum value	SELECT MAX(hire_date) FROM employees
MIN(col)	Minimum value	SELECT MIN(price) FROM products

```
-- Subquery in WHERE
SELECT name FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);

-- Subquery in FROM (derived table)
SELECT dept, avg_sal FROM
 (SELECT dept, AVG(salary) AS avg_sal FROM employees GROUP BY dept) AS dept_avgs
WHERE avg_sal > 60000;
```

A. Create a delta table to log the editing history of posts. The table records post id, username, old text, group id and time stamp.

```
CREATE TABLE post_history (
    post_id INT,
    username VARCHAR(50),
    old_text VARCHAR(140),
    group_id INT,
    edit_time TIMESTAMP,
    PRIMARY KEY (post_id, edit_time)
);
```

B. Write a trigger that records any changes to the text of a post by populating all the fields in the delta table created above.

```
CREATE OR REPLACE TRIGGER log_post_edits
AFTER UPDATE OF text ON posts
FOR EACH ROW
BEGIN
    INSERT INTO post_history (post_id, username, old_text, group_id, edit_time)
    VALUES (:OLD.post_id, :OLD.username, :OLD.text, :OLD.group_id, SYSTIMESTAMP);
END;
/
```

C. Write an event to invoke that trigger.

```
UPDATE posts
SET text = 'This is a modified post text.'
WHERE post_id = 1111;
```

Student_ID	Student_Name	Major	Course_ID	Course_ Name	Instructor_ID	Instructor_Name	Department	Grades
S1	Alice	CS	C101, C102	DB, OS	I1, I2	Dr. Smith, Dr. Lee	CS, CS	A, B
S2	Bob	Math	C101	DB	I1	Dr. Smith	CS	B
S3	Charlie	CS	C103, C101	AI, DB	I3, I1	Dr. Adams, Dr. Smith	AI, CS	A, C

a) Convert to 1NF (First Normal Form)

To reach 1NF, you must remove **repeating groups** and ensure every cell contains only one value (atomicity).

- **Action:** Take S1 (Alice), who has two courses (C101, C102), and split her into two distinct rows.
- **Result:** The table will now have rows like:
  - `S1 | Alice | CS | C101 | DB | 11 | Dr. Smith | CS | A`
  - `S1 | Alice | CS | C102 | OS | 12 | Dr. Lee | CS | B`

b) Identify Primary Key

Once flattened, a single ID isn't enough because a student can take many courses, and a course has many students.

- **Primary Key: (Student_ID, Course_ID).**

c) List Partial Dependencies

A partial dependency occurs when a non-key attribute depends on only *part* of the composite primary key.

- **{Student_ID}** → `Student_Name`, `Major`.
- **{Course_ID}** → `Course_Name`, `Instructor_ID`, `Instructor_Name`, `Department`.

d) List Transitive Dependencies

A transitive dependency occurs when a non-key attribute depends on *another* non-key attribute (functional dependency through a third party).

- **{Instructor_ID}** → `Instructor_Name`, `Department`.
  (Note: Since Instructor_ID is not a key for the whole table, but determines Name/Dept, this is a transitive link.)

e) Convert to 2NF

To reach 2NF, we remove the **Partial Dependencies** identified in step (c) by creating separate tables.

1. **Student_Info:** (`Student_ID`, Student_Name, Major)
2. **Course_Instructor_Info:** (`Course_ID`, Course_Name, Instructor_ID, Instructor_Name, Department)
3. **Grades:** (`Student_ID`, `Course_ID`, Grade)

f & g) Convert to 3NF & Final Schema

To reach 3NF, we remove the **Transitive Dependencies** identified in step (d). We pull the Instructor details out of the Course table.

Final Tables:

- **Student:** (`Student_ID` [PK], Student_Name, Major)
- **Instructor:** (`Instructor_ID` [PK], Instructor_Name, Department)
- **Course:** (`Course_ID` [PK], Course_Name, `Instructor_ID` [FK])
- **Enrollment:** (`Student_ID` [FK], `Course_ID` [FK], Grade)

Primary Key Rule: A primary key must be unique and can **never** be NULL.

2NF Shortcut: If a table's primary key is a **single attribute** (like `Student_ID`), and it is in 1NF, it is **automatically in 2NF** because you cannot have a partial dependency on a single attribute.

Foreign Keys (FK): When you decompose a table into 2NF or 3NF, you *must* keep a common attribute (the Foreign Key) in the original table to link it back to the new one, or you lose the data relationship.

Closure Strategy: When finding keys, always start by checking attributes that never appear on the right side of an FD arrow—they *must* be part of your candidate key.

Suppose the social media platform wants to censor two words: "stupid" or "dump". Write a stored procedure with a cursor structure to make the posts what contain those censored words not visible (by updating the visibility attribute).

```
CREATE OR REPLACE PROCEDURE censor_posts AS
    CURSOR post_cursor IS
        SELECT post_id, text
        FROM posts
        WHERE text LIKE '%stupid%' OR text LIKE
'%dump%';

    v_post_id posts.post_id%TYPE;
    v_text posts.text%TYPE;
BEGIN
    OPEN post_cursor;
    LOOP
        FETCH post_cursor INTO v_post_id, v_text;
        EXIT WHEN post_cursor%NOTFOUND;

        UPDATE posts
        SET visibility = 'F'
        WHERE post_id = v_post_id;
    END LOOP;
    CLOSE post_cursor;
    COMMIT;
END;
/
```

Write a trigger to raise an application error if a new post in a group were posted by a user who does not meet group's age requirement. Here, the event is an insertion of a new tuple to the Post table.

```
CREATE OR REPLACE TRIGGER check_post_age
BEFORE INSERT ON posts
FOR EACH ROW
DECLARE
    v_user_age INT;
    v_min_age INT;
    v_max_age INT;
BEGIN
    -- Get user age
    SELECT age INTO v_user_age FROM users WHERE username = :NEW.username;
    -- Get group age requirements
    SELECT min_age, max_age INTO v_min_age, v_max_age FROM groups WHERE group_id = :NEW.group_id;

    IF (v_user_age < v_min_age OR v_user_age > v_max_age) THEN
        RAISE_APPLICATION_ERROR(-20001, 'User does not meet the age requirement for this group.');
    END IF;
END;
/
```

Concept	Symbol / Shape	Description
Entity	Rectangle	A real-world object or concept (e.g., Student, Course)
Attribute	Oval / Ellipse	A property of an entity (e.g., Name, DOB)
Relationship	Diamond	Association between two or more entities
Weak Entity	Double Rectangle	Cannot be identified without a related strong entity
Key Attribute	Underlined oval	Uniquely identifies an entity instance
Derived Attribute	Dashed oval	Can be calculated from other attributes (e.g., Age from DOB)
Multivalued Attribute	Double oval	Can have multiple values (e.g., Phone numbers)
Composite Attribute	Oval with child ovals	Made up of sub-attributes (e.g., Address)

ER Element	Relational Schema Rule
Strong Entity	Create a table with all attributes; PK = key attribute
Weak Entity	Create a table; PK = partial key + PK of owner entity
1:1 Relationship	Add FK of one entity into the other (or merge into one table)
1:N Relationship	Add PK of the '1' side as FK on the 'N' side table
M:N Relationship	Create a new junction/associative table with PKs of both entities as composite PK
Multivalued Attribute	Create a separate table with FK to parent entity
Composite Attribute	Use the sub-attributes as individual columns; omit the composite
Derived Attribute	Typically omit from schema; compute when needed

Scenario to ER Diagram

1. Identify entities - look for nouns (people, places, things, concepts).
2. Identify attributes for each entity - properties that describe the entity.
3. Identify the primary key for each entity.
4. Identify relationships - look for verbs connecting entities.
5. Determine cardinality (1:1, 1:N, M:N) for each relationship.
6. Determine participation (total or partial) for each entity in each relationship.

A functional dependency X -> Y means: for every tuple in the relation, if two tuples agree on the value of X, they must also agree on Y. We say X determines Y.

Types of Functional Dependencies:
Trivial FD: Y is a subset of X. e.g., {A, B} -> A
Non-trivial FD: Y is NOT a subset of X. e.g., StudentID -> Name
Full FD: Y depends on the whole of X, not a proper subset.
Partial FD: Y depends on a proper subset of X (violates 2NF).
Transitive FD: X -> Z via X -> Y -> Z, where Y is not a key (violates 3NF).

Normal Form	Requirement	Removes	How to Fix (Decomposition)	Violation Example	Problem	The Fix (Normalized)
1NF	All attributes are atomic (no repeating groups, no arrays)	Repeating groups / non-atomic values	Expand multi-valued attributes into separate rows.	101, Alice, 'Math, Science'	Courses is not atomic.	Separate rows for Math and Science.
2NF	In 1NF + every non-key attribute is fully functionally dependent on the entire primary key	Partial dependencies	Move attributes that depend on only *part* of a composite key to a new table.	OrderID, ProductID → ProductName	ProductName only needs ProductID.	**Table A:** OrderID, ProductID, Qty **Table B:** ProductID, ProductName
3NF	In 2NF + no transitive dependencies (non-key -> non-key)	Transitive dependencies	Move attributes that depend on other non-prime attributes to a new table.	EmpID → DeptID → DeptName	DeptName depends on DeptID, not the Key.	**Table A:** EmpID, DeptID **Table B:** DeptID, DeptName
BCNF	In 3NF + for every FD X -> Y, X must be a superkey	Remaining anomalies where a non-superkey determines another attribute	Ensure every left-side attribute in a functional dependency can uniquely identify a row.	Teacher → Course	Teacher is not a superkey.	**Table A:** Teacher, Course **Table B:** Student, Teacher

Attribute Closure
The attribute closure of a set of attributes X (written X+) is the set of all attributes functionally determined by X, given a set of FDs F.

Algorithm to compute X+:
1. Start: closure = X
2. For each FD (A -> B) in F: if A is a subset of closure, add B to closure.
3. Repeat step 2 until closure no longer changes.

Example: R(A,B,C,D), FDs: A->B, B->C, A->D. Compute {A}+: Start: {A}. A->B: add B => {A,B}. B->C: add C => {A,B,C}. A->D: add D => {A,B,C,D}. So {A}+ = {A,B,C,D}.

a) Compute Closure (F^+) for $R = (A, B, C, D, E)$

Given: $A \to BC, CD \to E, B \to D, E \to A$

To find the candidate keys, we check the closures of individual attributes:

- A^+: $\{A\} \to \{A, B, C\} \xrightarrow{B \to D} \{A, B, C, D\} \xrightarrow{CD \to E} \{A, B, C, D, E\}$. (**Candidate Key**)
- E^+: $\{E\} \to \{E, A\} \to \ldots$ (same as A). (**Candidate Key**)
- BC^+: $\{B, C\} \xrightarrow{B \to D} \{B, C, D\} \xrightarrow{CD \to E} \{B, C, D, E\} \xrightarrow{E \to A} \{A, B, C, D, E\}$. (**Candidate Key**)

Relation $R(A, B, C, D)$

Functional Dependencies (FDs): $\{A \to B, C \to D\}$

A only determines B.

C only determines D.

Since neither A nor C can determine the entire row alone, we combine them.

The closure of $\{AC\}$, denoted as $(AC)^+$, is $\{A, C, B, D\}$.

Candidate Key: $\{A, C\}$.

ii. Decompose the relation into 2NF

A relation is in **2NF** if it is in 1NF and every non-prime attribute (attributes not part of a candidate key) is **fully** functionally dependent on the entire candidate key.

- **The Problem:** We have "Partial Dependencies."
  - B is dependent only on A (part of the key).
  - D is dependent only on C (part of the key).
- **The Solution:** Decompose the table into smaller tables to eliminate these partial dependencies.

Decomposed Tables:

1. $R_1(A, B)$: Where A is the Primary Key.
2. $R_2(C, D)$: Where C is the Primary Key.
3. $R_3(A, C)$: This table preserves the relationship between the key attributes, where (A, C) is the Primary Key.

Candidate Keys
A candidate key is a minimal set of attributes whose closure is the entire relation schema (determines all attributes).

Steps to find all candidate keys:
1. Compute attribute closure for each single attribute.
2. If a single attribute determines all, it is a candidate key.
3. Try combinations of 2, 3, ... attributes until you find minimal superkeys.
4. A candidate key must be minimal - removing any attribute from it makes it no longer a superkey.

A	B	C
2	1	5
8	2	4
3	3	4
8	4	5

$B \to C$: **Yes.** Each unique value of B (1, 2, 3, 4) maps to exactly one value of C.

$C \to B$: **No.** Attribute $C = 4$ maps to $B = 2$ and $B = 3$.

$A \to BC$: **No.** Attribute $A = 8$ maps to $BC = (2, 4)$ in row 2, and $BC = (4, 5)$ in row 4. One "A" value maps to two different "BC" results.

Construct an E-R diagram for a car insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars and has one or more premium payments associated with it. Each payment is for a particular period of time, and has an associated due date, and the date when the payment was received.

Entities & Attributes:
Customer: SSN (PK), Name, Address.
Car: VIN (PK), Model, Make, Year.
Accident: Accident_ID (PK), Date, Location.
Policy: Policy_Number (PK), Type, Expiry_Date.
Payment: Payment_ID (PK), Due_Date, Amount, Date_Received.
Relationships:
Owns: Customer (1) to Car (M).
Participated: Car (1) to Accident (M) — Note: Total participation on Accident side since an accident must involve a car.
Covers: Policy (1) to Car (M).
Contains: Policy (1) to Payment (M).

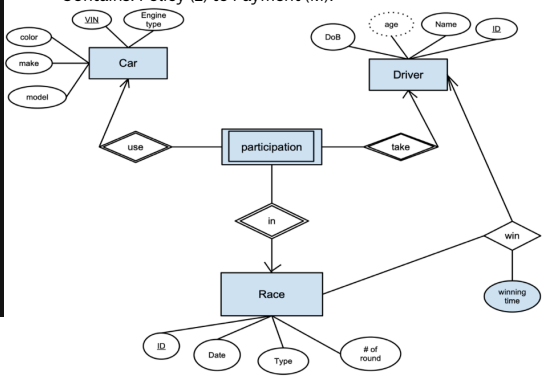

Table	Attributes	Primary Key (PK)	Foreign Key (FK)
Car	**VIN**, Engine_Type, Color, Make, Model	VIN	None
Driver	**ID**, Name, DoB, *Age (derived)*	ID	None
Race	**ID**, Date, Type, Rounds	ID	None
Participation	**VIN, Driver_ID, Race_ID**	(VIN, Driver_ID, Race_ID)	VIN, Driver_ID, Race_ID
Win	**Race_ID**, Driver_ID, Winning_Time	Race_ID	Race_ID, Driver_ID