Importance ■■■■■ most asked ■■■□□ often ■□□□□ rarely

SELECT queries

Write the clauses in this order; leave out the ones you don't need:

```
SELECT DISTINCT col, expr AS alias
FROM table
WHERE condition
GROUP BY col
HAVING group_condition
ORDER BY col DESC;
```

IN WHERE	EXAMPLE
Compare	`Marks >= 80` · `Class <> '12A'`
AND, OR, NOT	`Class='12A' AND Marks>80`
BETWEEN	`Marks BETWEEN 70 AND 90` (both ends included)
IN	`City IN ('Delhi','Pune')`
LIKE	`%` any number of characters, `_` exactly one
NULL	`Marks IS NULL` · `IS NOT NULL`

LIKE PATTERN	MATCHES
`'A%'`	starts with A
`'%a'`	ends with a
`'%an%'`	has "an" anywhere
`'_a%'`	a is the second letter
`'____'`	exactly four characters

ORDER BY is ascending by default; `DESC` applies only to the column it follows. **DISTINCT** removes duplicate rows. An **alias** (`AS`) renames only the output heading.

! Never `= NULL`: write **IS NULL**. A NULL row fails every comparison, so it is in neither `Marks > 80` nor `Marks <= 80`. Text and dates need quotes; numbers don't.

Aggregates, GROUP BY, HAVING

Table S, used for every output below:

ROLL	NAME	CLASS	MARKS
1	Aman	12A	91
2	Riya	12B	78
3	Kabir	12A	NULL
4	Sara	12B	85
5	Dev	12A	62

FUNCTION	RESULT	WHY
`COUNT(*)`	5	counts rows, NULLs too
`COUNT(Marks)`	4	skips the NULL
`SUM(Marks)`	316	
`AVG(Marks)`	79.0000	316 ÷ 4, not ÷ 5
`MAX` · `MIN`	91 · 62	
`COUNT(DISTINCT Class)`	2	12A, 12B

```
SELECT Class, COUNT(*), AVG(Marks)
FROM S GROUP BY Class;          -- 12A 3 76.5000
                                -- 12B 2 81.5000
SELECT Class, COUNT(*) FROM S
GROUP BY Class HAVING COUNT(*) > 2;   -- 12A 3
```

WHERE filters rows **before** grouping; **HAVING** filters groups **after**. "Class-wise", "gender-wise", "category-wise" → GROUP BY that column, and put it in SELECT too.

! Aggregates can't go in WHERE: `WHERE COUNT(*) > 5` must become **HAVING COUNT(*) > 5**, written after GROUP BY. All aggregates except COUNT(*) ignore NULLs.

Relational terms and keys

TERM	MEANS
Relation	a table
Tuple	a row (record)
Attribute	a column (field)
Domain	the set of values a column may take
Degree	number of **columns**
Cardinality	number of **rows** (cardinality counts records)

KEY	MEANS
Candidate key	any column that can identify each row uniquely
Primary key	the candidate key chosen; unique, never NULL, one per table
Alternate key	candidate keys not chosen as primary
Foreign key	a column that refers to the primary key of another table; links tables

! 4 rows × 3 columns; add 2 rows, remove 1 column → degree **2**, cardinality **6**. "Keeps data integrity between tables" → **foreign key**.

DDL and DML commands

TYPE	COMMANDS	CHANGES
DDL (definition)	CREATE, ALTER, DROP	the structure
DML (manipulation)	INSERT, UPDATE, DELETE	the data
DQL (query)	SELECT	nothing; only reads

```
CREATE DATABASE School;  USE School;
CREATE TABLE S (Roll INT PRIMARY KEY,
  Name VARCHAR(20), Class CHAR(3), Marks INT);
ALTER TABLE S ADD City VARCHAR(15);
ALTER TABLE S DROP City;
ALTER TABLE S DROP PRIMARY KEY;
ALTER TABLE S ADD PRIMARY KEY (Roll);
INSERT INTO S VALUES (6,'Neha','12B',88);
UPDATE S SET Marks = Marks + 5 WHERE Roll = 2;
DELETE FROM S WHERE Marks IS NULL;
DROP TABLE S;       -- removes the whole table
```

Also: `SHOW DATABASES; SHOW TABLES; DESCRIBE S;` (or `DESC S;`) shows the structure.

	CHAR(N)	VARCHAR(N)
Length	fixed: always n characters	variable: up to n
'Hi' in (10)	uses 10, padded with spaces	uses 2

! ALTER changes the **degree**; INSERT and DELETE change the **cardinality**; UPDATE changes neither. Without WHERE, UPDATE and DELETE hit **every** row. DROP removes the table; DELETE only its rows.

Python and MySQL connectivity

Every program follows the same six steps. A full-marks answer shows all six.

STEP	CODE
1 Import the connector	`import mysql.connector as m`
2 Connect to a database	`con = m.connect(host=…, user=…, passwd=…, database=…)`
3 Make a cursor	`cur = con.cursor()`
4 Run the SQL	`cur.execute(q, values)`
5 Read or save	`cur.fetchall()` for SELECT · `con.commit()` for changes
6 Close	`con.close()`

Display records (SELECT)

```
import mysql.connector as m
con = m.connect(host='localhost', user='root',
                passwd='tiger', database='School')
cur = con.cursor()
q = "SELECT Name, Marks FROM S WHERE Marks > %s"
cur.execute(q, (80,))
for row in cur.fetchall():
    print(row)
print("Rows:", cur.rowcount)
con.close()
```

```
('Aman', 91)
('Sara', 85)
Rows: 2
```

Insert values typed by the user (%s)

```
# steps 1-3: import, connect, cursor as above
r = int(input("Roll: "))
n = input("Name: ")
c = input("Class: ")
mk = int(input("Marks: "))
q = "INSERT INTO S VALUES (%s, %s, %s, %s)"
cur.execute(q, (r, n, c, mk))
con.commit()
print(cur.rowcount, "record inserted")
con.close()
```

```
Roll: 6
Name: Neha
Class: 12B
Marks: 88
1 record inserted
```

Update or delete (format)

```
# steps 1-3: import, connect, cursor as above
q = "UPDATE S SET Marks = {} WHERE Name = '{}'"
cur.execute(q.format(80, 'Riya'))
con.commit()
print(cur.rowcount, "record updated")
cur.execute("DELETE FROM S WHERE Marks IS NULL")
con.commit()
print(cur.rowcount, "record deleted")
con.close()
```

```
1 record updated
1 record deleted
```

	%S PLACEHOLDER	FORMAT()
Values go	in a tuple: `execute(q, (80,))`	into the string: `q.format(80)`
Quotes for text	the connector adds them	you write them: `'{}'`

METHOD	GIVES
`fetchone()`	next row as a tuple, or None
`fetchall()`	all remaining rows, a list of tuples
`fetchmany(n)`	next n rows
`cur.rowcount`	rows fetched or changed

! **commit()** is on the connection (`con.commit()`), not the cursor; without it INSERT, UPDATE and DELETE are lost. One %s value still needs a **tuple**: `(80,)`. Don't put quotes around %s.

Joins

- **Cartesian product**: `SELECT * FROM A, B;` pairs every row of A with every row of B. 3 rows × 2 rows = **6** rows; degree = columns of A + columns of B.
- **Equi-join**: join on equal values of a common column; name the table for that column.

```
SELECT Name, GName FROM PLAYER, GAMES
WHERE PLAYER.GCode = GAMES.GCode;
```

Natural join: like an equi-join, but the common column appears only once: `SELECT * FROM PLAYER NATURAL JOIN GAMES;`

Constraints

CONSTRAINT	RULE
NOT NULL	value must be given
UNIQUE	no duplicates; NULL allowed; many per table
PRIMARY KEY	unique and not NULL; one per table

Database concepts

A **database** is organised data; a **DBMS** is the software that manages it. Over a **file system** it removes **redundancy** (the same data stored many times) and **inconsistency** (copies that disagree). **MySQL** is an open-source **RDBMS**; **SQL** is the language it understands, and it is not case-sensitive for keywords.

What's the output?

On table S above. Cover the right column and answer; every result was run on MySQL.

QUERY	OUTPUT
`SELECT Name FROM S WHERE Name LIKE '%a';`	Riya, Sara
`SELECT COUNT(*) FROM S WHERE Marks > 80;`	2
`SELECT MAX(Marks) - MIN(Marks) FROM S;`	29
`SELECT Class, MAX(Marks) FROM S GROUP BY Class;`	12A 91 · 12B 85
`SELECT Name FROM S WHERE Marks BETWEEN 78 AND 85 ORDER BY Marks DESC;`	Sara, Riya
`SELECT COUNT(*) FROM S WHERE Marks = NULL;`	0

Find the error

WRONG	RIGHT
`WHERE COUNT(*) > 2`	`HAVING COUNT(*) > 2`
`WHERE Marks = NULL`	`WHERE Marks IS NULL`
`DELETE * FROM S;`	`DELETE FROM S;`
`WHERE Name = Riya`	`WHERE Name = 'Riya'`
`... ORDER BY Marks WHERE Class='12A'`	WHERE before ORDER BY
`... HAVING ... GROUP BY Class`	GROUP BY before HAVING
`WHERE Marks BETWEEN 70, 90`	`BETWEEN 70 AND 90`
`UPDATE S Marks = 90 ...`	`UPDATE S SET Marks = 90 ...`
`cur.commit()`	`con.commit()`
`cur.execute(q, 80)`	`cur.execute(q, (80,))`