Practical-2 — JOINS

---

1) Create database and use it

```
CREATE DATABASE practical2;

USE practical2;
```

Line-by-line:

- CREATE DATABASE practical2;
  - CREATE DATABASE — DDL command that creates a new database container.
  - practical2 — name of the new database. After this runs MySQL creates a folder/namespace to hold tables, views, etc.
- USE practical2;
  - USE — switches the current session to the specified database.
  - practical2 — after this, all subsequent CREATE/INSERT/SELECT statements run inside practical2.

---

2) Create table Student2

```
CREATE TABLE Student2(
  id INT PRIMARY KEY AUTO_INCREMENT,
  admission_no VARCHAR(45) NOT NULL,
  first_name VARCHAR(45) NOT NULL,
  last_name VARCHAR(45) NOT NULL,
  age INT,
  city VARCHAR(25) NOT NULL
);
```

Line-by-line:

- CREATE TABLE Student2(— starts creation of a new table named Student2.
- id INT — defines a column id of integer type.

- PRIMARY KEY — marks id as the primary key (uniquely identifies each row and cannot be NULL).
- AUTO_INCREMENT — automatic sequential value generation for id on each insert (1,2,3,...). This means you don't supply id when inserting — MySQL fills it.
- admission_no VARCHAR(45) NOT NULL — column admission_no stores text up to 45 characters; NOT NULL enforces that admission number must be provided (cannot be empty).
- first_name VARCHAR(45) NOT NULL — student's first name, required.
- last_name VARCHAR(45) NOT NULL — student's last name, required.
- age INT — age as integer (may be NULL since NOT NULL not specified).
- city VARCHAR(25) NOT NULL — city name up to 25 chars, required.
-); — ends the CREATE TABLE statement.

Effect: creates the Student2 table schema; no data yet.

---

3) Create table Fee

```
CREATE TABLE Fee(
  admission_no VARCHAR(45) NOT NULL,
  course VARCHAR(45) NOT NULL,
  amount_paid INT
);
```

Line-by-line:

- CREATE TABLE Fee(— starts a new table named Fee.
- admission_no VARCHAR(45) NOT NULL — admission number column (used to match with Student2.admission_no); required field.
- course VARCHAR(45) NOT NULL — course name; required.
- amount_paid INT — fee amount as integer (currency value); may be NULL.
-); — ends statement.

Note: Fee.admission_no is not declared as a foreign key here, but logically it corresponds to Student2.admission_no. That's fine for practicals — joins will match values.

---

4) Insert student rows into Student2

INSERT INTO Student2 (admission_no,first_name,last_name,age,city) VALUES

(1001,'Shital','Gayke',18,'Sinnar'),

(1002,'Sakshi','More',20,'Nashik'),

(1003,'Ajay','Mendade',22,'Satpur'),

(1004,'Nidhi','Jadhav',24,'Nashik'),

(1005,'Prashant','More',26,'Wani'),

(1006,'Alok','Pandit',28,'Nashik'),

(1007,'Sanju','Banka',30,'Bhagur');

Line-by-line:

- INSERT INTO Student2 (admission_no,first_name,last_name,age,city) — begin inserting rows into Student2, explicitly listing the target columns. Listing columns ensures you insert values in the correct columns and makes the statement tolerant to column order changes.
- VALUES — starts the list of row values to insert.
- Each parenthesized tuple e.g. (1001,'Shital','Gayke',18,'Sinnar') represents one row:
  - 1001 → value for admission_no
  - 'Shital' → first_name
  - 'Gayke' → last_name
  - 18 → age
  - 'Sinnar' → city
- Commas separate multiple row tuples so we insert 7 rows in one statement.
- Because id is AUTO_INCREMENT, MySQL fills id automatically for each row (1..7).

Effect: table now has 7 student records.

---

5) Insert fee rows into Fee

INSERT INTO Fee VALUES

(1001,'Android',10000),

(1002,'Data Science',15000),

(1003,'SQL',18000),

(1004,'Python',20000),

(1005,'Java',8000),

(1010,'ML',25000),

(1011,'Cyber Security',22000);

Line-by-line:

- INSERT INTO Fee VALUES — inserts rows into Fee. NOTE: columns not listed, so values must match table column order (admission_no, course, amount_paid).
- Each tuple (1001,'Android',10000) gives:
  - 1001 → admission_no (student who paid)
  - 'Android' → course
  - 10000 → amount_paid
- Rows with admission_no 1010 and 1011 correspond to fee records with no matching student in Student2 (used to demonstrate RIGHT/FULL joins).
- Effects: Fee table now has 7 rows.

Now the data is ready. Below are the **10 queries** with complete line-by-line explanations and expected reasoning about output.

Query 1 — INNER JOIN (exact match rows only)

SELECT s.admission_no, s.first_name, s.last_name, f.course, f.amount_paid

FROM Student2 s

INNER JOIN Fee f ON s.admission_no = f.admission_no;

Line-by-line:

- SELECT s.admission_no, s.first_name, s.last_name, f.course, f.amount_paid
  - SELECT chooses columns to display.

    - s.admission_no etc. — prefix s. means take column from table alias s (Student2). f. from Fee.
    - Choosing specific columns avoids showing id or unwanted columns and keeps output neat.
- FROM Student2 s
    - Use Student2 as the left table and assign it alias s (short name) for concise references.
- INNER JOIN Fee f
    - INNER JOIN instructs MySQL to return only rows where a matching pair exists in both tables.
    - Fee f assigns alias f to Fee.
- ON s.admission_no = f.admission_no;
    - ON specifies the join condition — rows are matched when Student2.admission_no equals Fee.admission_no.
    - Only rows satisfying this equality are included.

Why it returns 5 rows: students with admission_no 1001–1005 have fee records. Students 1006 & 1007 have no fees → excluded. Fee rows 1010 & 1011 have no matching students → excluded.

Query 2 — LEFT JOIN (all students, fees if present)

SELECT s.admission_no, s.first_name, s.last_name, f.course, f.amount_paid

FROM Student2 s

LEFT JOIN Fee f ON s.admission_no = f.admission_no;

Line-by-line:

- SELECT ... — same selected columns.
- FROM Student2 s — left table is Student2 (all rows from this table will appear).
- LEFT JOIN Fee f — left join returns all rows from Student2 and matching rows from Fee.
- ON s.admission_no = f.admission_no; — same join condition.

Effect: returns 7 rows (all students). For students without fee rows (1006,1007), f.course and f.amount_paid will be NULL. Useful when you must list all students regardless of payment.

Query 3 — RIGHT JOIN (all fee records, students if present)

SELECT s.admission_no, s.first_name, s.last_name, f.course, f.amount_paid

FROM Student2 s

RIGHT JOIN Fee f ON s.admission_no = f.admission_no;

Line-by-line:

- SELECT ... — same columns.
- FROM Student2 s — left table is Student2.
- RIGHT JOIN Fee f — returns all rows from Fee (right table) and matching Student2 rows. If a fee record has no student, student columns are NULL.
- ON s.admission_no = f.admission_no; — join condition.

Effect: returns 7 rows (all Fee rows). For admission_no 1010 and 1011 there's no student — s.first_name etc. are NULL.

Query 4 — FULL JOIN simulation via UNION (all students and all fee rows)

SELECT s.admission_no, s.first_name, s.last_name, f.course, f.amount_paid

FROM Student2 s

LEFT JOIN Fee f ON s.admission_no = f.admission_no

UNION

SELECT s.admission_no, s.first_name, s.last_name, f.course, f.amount_paid

FROM Student2 s

RIGHT JOIN Fee f ON s.admission_no = f.admission_no;

Line-by-line:

- First SELECT ... FROM Student2 s LEFT JOIN Fee f ON ... — returns all Student2 rows plus matching Fee rows (left side).
- UNION — combines results of the two SELECTs and removes duplicate rows (UNION does distinct by default). Using UNION ALL would keep duplicates. We

use UNION to avoid duplicate rows that appear in both left and right join results for matching admission_no.

- Second SELECT ... FROM Student2 s RIGHT JOIN Fee f ON ... — returns all Fee rows plus matching Student2 rows (right side).
- Combined result is logically a FULL OUTER JOIN: all students and all fee records; unmatched columns on either side appear as NULL.

Important:

- UNION removes duplicate rows; if you need to keep duplicates (rare here), use UNION ALL.
- The result includes rows where either student exists without fee (1006,1007) or fee without student (1010,1011) plus the matched ones.

Query 5 — Subquery: row(s) with minimum fee

SELECT * FROM Fee

WHERE amount_paid = (SELECT MIN(amount_paid) FROM Fee);

Line-by-line:

- SELECT * FROM Fee — select all columns from Fee table (admission_no, course, amount_paid).
- WHERE amount_paid = — filter rows where amount_paid equals...
- (SELECT MIN(amount_paid) FROM Fee) — subquery: calculates the minimum amount_paid across the Fee table.
    - MIN(amount_paid) returns the smallest numeric value.
    - The subquery returns that scalar value (here 8000).
- Outer query returns the fee row(s) whose amount_paid equals that minimum (admission_no 1005, course Java).

Why use subquery: dynamic — if data changes, MIN recomputes automatically.

Query 6 — Subquery: students living in the same city as 'Shital'

SELECT * FROM Student2

WHERE city = (SELECT city FROM Student2 WHERE first_name='Shital');

Line-by-line:

- SELECT * FROM Student2 — return all columns of students matching the WHERE.
- WHERE city = — filter rows whose city equals…
- (SELECT city FROM Student2 WHERE first_name='Shital') — subquery:
    - Finds the city value of the student whose first_name = 'Shital'. If there are multiple Shitals this subquery must return single value else it fails — in your dataset only one Shital so it returns 'Sinnar'.
- Outer query returns students with city = 'Sinnar'. In your data only Shital lives in Sinnar → returns that single row.

Note: If the subquery might return multiple rows, use IN instead of =.

Query 7 — Subquery: students with fee >= average fee

SELECT s.admission_no, s.first_name, s.last_name, f.amount_paid

FROM Student2 s

JOIN Fee f ON s.admission_no = f.admission_no

WHERE f.amount_paid >= (SELECT AVG(amount_paid) FROM Fee);

Line-by-line:

- SELECT s.admission_no, s.first_name, s.last_name, f.amount_paid — pick student identification and amount paid.
- FROM Student2 s JOIN Fee f ON s.admission_no = f.admission_no
    - JOIN (INNER JOIN) ensures we only consider students who have a fee record.
- WHERE f.amount_paid >= — filter rows where fee is greater than or equal to…
- (SELECT AVG(amount_paid) FROM Fee) — subquery computing average fee across Fee table.
    - AVG(amount_paid) returns the arithmetic mean (e.g., if fees are [10000,15000,18000,20000,8000,25000,22000] average ≈ 16571).
- Outer query returns students whose amount_paid is >= average (Ajay 18000, Nidhi 20000 in your data).

Why useful: find students paying above-average — common analysis.

Query 8 — (Repeated) Subquery minimum fee — same as Query 5

```
SELECT * FROM Fee

WHERE amount_paid = (SELECT MIN(amount_paid) FROM Fee);
```

Explanation: same as Query 5. (Sometimes repeated in labs to show both MIN and repeated usage.)

Query 9 — Subquery: student(s) with maximum fee

```
SELECT * FROM Fee

WHERE amount_paid = (SELECT MAX(amount_paid) FROM Fee);
```

Line-by-line:

- SELECT * FROM Fee — show the fee record(s).
- WHERE amount_paid = — filter equality to...
- (SELECT MAX(amount_paid) FROM Fee) — subquery that computes the maximum amount_paid.
  - MAX(amount_paid) returns the largest fee (here 25000).
- Outer query returns the fee row(s) with that maximum (admission_no 1010, course ML).

Note: If multiple rows have same max value, all are returned.

Query 10 — Create a VIEW Student2FeeView and SELECT from it

```
CREATE VIEW Student2FeeView AS

SELECT s.admission_no, s.first_name, s.last_name, s.city, f.course, f.amount_paid

FROM Student2 s

LEFT JOIN Fee f ON s.admission_no = f.admission_no

ORDER BY s.first_name;


SELECT * FROM Student2FeeView;
```

Line-by-line (VIEW creation):

- CREATE VIEW Student2FeeView AS — defines a new view (virtual table) named Student2FeeView. The view stores the query definition, not the data.
- SELECT s.admission_no, s.first_name, s.last_name, s.city, f.course, f.amount_paid — the columns that the view will expose.
- FROM Student2 s LEFT JOIN Fee f ON s.admission_no = f.admission_no — view joins students with fees using LEFT JOIN so all students are present; fee columns will be NULL if no fee record.
- ORDER BY s.first_name; — orders the view result by student first name when you select from the view. (Note: some MySQL versions ignore ORDER BY in view definition; selecting with ORDER BY at query time is safest. But you used ORDER BY in creation and it worked in your session.)

Line-by-line (selecting from view):

- SELECT * FROM Student2FeeView; — fetches rows from the view as if it were a table. The DB executes the underlying SELECT and returns rows.

Effect: View provides a reusable, simplified interface to joined student+fee information sorted by first name.

Quick notes about NULLs and matching behavior

- When using LEFT JOIN, if there is no matching Fee row for a Student2 row, f.course and f.amount_paid appear as NULL. This signals "no data".
- When using RIGHT JOIN, if there is a Fee row with no matching Student2 row (admission_no 1010,1011), s.first_name etc. are NULL.
- The UNION in FULL JOIN eliminates duplicate matched rows; unmatched rows from both sides are included.

Example outputs — why they look like that (brief)

- **INNER JOIN** returns 5 rows for admission_no 1001–1005 (only those have both student and fee).
- **LEFT JOIN** returns 7 student rows; 1006 & 1007 show NULL in fee columns.
- **RIGHT JOIN** returns 7 fee rows; fee rows 1010 & 1011 show NULL in student columns.
- **FULL (UNION)** returns 9 rows: all matched + unmatched students (2) + unmatched fees (2) but one duplicate removed if present.

- **MIN/MAX** subqueries directly return Fee row(s) with min or max amount_paid.
- **AVG** subquery filters fee rows >= average amount; returns the students paying >= average.

How to explain this practical to the external (20 seconds)

"Sir, I created two tables: Student2 (student details) and Fee (course + amount). I inserted 7 students and 7 fee records (including two fee records that have no corresponding student to demonstrate unmatched rows). I used INNER, LEFT, RIGHT joins and simulated FULL join with UNION to show matching and non-matching rows. I used subqueries with MIN, MAX, and AVG to find students paying min, max, and above-average fees, and finally created a view Student2FeeView that combines student and fee information ordered by first name."

Viva (common Q&A) — short answers

1. **Q:** What is INNER JOIN?
   A: Returns rows where matching keys exist in both tables.
2. **Q:** What is LEFT JOIN?
   A: Returns all rows from the left table and matching rows from the right table; unmatched right columns are NULL.
3. **Q:** What is RIGHT JOIN?
   A: Returns all rows from the right table and matching rows from left; unmatched left columns are NULL.
4. **Q:** How to get FULL JOIN in MySQL?
   A: Use LEFT JOIN ... UNION RIGHT JOIN ....
5. **Q:** What does AUTO_INCREMENT do?
   A: Automatically generates sequential integer values for a column on inserts.
6. **Q:** Difference between UNION and UNION ALL?
   A: UNION removes duplicate rows; UNION ALL preserves duplicates.
7. **Q:** Why use subqueries?
   A: To compute values (min/max/avg) or filter based on results of other queries.
8. **Q:** Does a view store data?
   A: No — a view stores the query; the data is fetched from base tables when you query the view.
