Welcome to Lecture 40!

Intro to MySQL (contd.)

Today: practical queries in MySQL Workbench

Agenda

1. Warmup
2. Live demo: INSERT, SELECT, WHERE, UPDATE, DELETE
3. Data types
4. Constraints
5. Better SELECT
6. Aggregate functions
7. Joins preview

Warmup

Live Demo

The 5 core verbs in Workbench

Setup: Use the database and table

We left off after creating the database and table.

Run these first:

```
USE students;

SELECT * FROM learners;
```

Tip: if SELECT shows no rows, that is fine

INSERT: add rows

INSERT adds new rows to a table.

If id is AUTO_INCREMENT, do not provide id.

```
INSERT INTO learners (name, course, marks)
VALUES ('Asha','Java',78);

INSERT INTO learners (name, course,marks)
VALUES ('Rohan','Java',92);
```

After INSERT, run SELECT to confirm

SELECT: view data

SELECT shows rows from a table.

Tip: Start with all columns, then narrow it down later.

```
SELECT * FROM learners;
```

WHERE: filter rows

WHERE filters which rows are returned.

This does not change the table.

```
SELECT * FROM learners
WHERE marks > 80;
```

UPDATE: change rows

UPDATE changes existing rows.

Tip: Always use WHERE, or you may update all rows.

```
UPDATE learners
SET marks = 80
WHERE name = 'Rahul';

SELECT * FROM learners;
```

Safe habit: SELECT first using the same WHERE

DELETE: remove rows

DELETE removes rows.

Tip: Always use WHERE, or you may delete all rows.

```
DELETE FROM learners
WHERE name = 'Rahul';

SELECT * FROM learners;
```

Safe habit: test the WHERE using SELECT first

Read it like English

What does this return?

Type one sentence in chat.

```
SELECT name, marks
FROM learners
WHERE marks >= 60;
```

Focus on meaning, not syntax perfection

Data Types

Pick the right kind of data for each column

Why data types matter

A data type decides what values a column can store.

Benefits:

1. Correctness (no text inside *marks* column)
2. Better comparisons and sorting
3. Efficient storage

Integers: pick a size that fits

MySQL supports several integer types:

1. TINYINT (very small counts or boolean flags)
   - Range: -2^7 to 2^7-1 (-127 to 127)
2. SMALLINT (small counts like age)
   - Range: -2^15 to 2^15-1 (approx. -32 thousand to +32 thousand)
3. MEDIUMINT (medium range counts)
   - Range: -2^23 to 2^23-1 (approx. -8 million to 8 million)
4. INT (common default)
   - Range: -2^31 to 2^31-1 (approx. -2 billion to 2 billion)
5. BIGINT (very large ids and counts)
   - Range: -2^63 to 2^63-1

General rule: choose the smallest that fits your real data range.

DECIMAL: exact values with decimals

Use DECIMAL when exact precision matters (example: money).

Range:

- Up to 65 digits before the decimal point
- Up to 30 digits after the decimal point

Example:

- price DECIMAL(10,2)
- values like 1999.50, 3.14

Strings: VARCHAR vs TEXT

VARCHAR(n): variable length, with a max size (common for names, emails).

- Range: Up to ~65,000 characters

TEXT: long free-form text (notes, descriptions). Size cannot be set.

- Range: Up to ~65,000 characters

Rule of thumb:

- If you know a reasonable max length, use VARCHAR.

Dates and times

DATE

- Only date (YYYY-MM-DD)
- Range: '1000-01-01' to '9999-12-31'
- Example: 2026-01-02

TIME

- Only time (HH:MM:SS)
- Example: 08:46:50

TIMESTAMP

- date + time
- Range: '1970-01-01 00:00:01.000000' UTC to '2038-01-19 03:14:07.499999'
- Example: 2026-01-02 08:46:50

BOOLEAN and JSON

BOOLEAN: true or false style values (active or inactive).
JSON: nested structured data (profile, settings).

Use JSON when the data is naturally nested.

Quick check (chat)

Pick the best type:

1. marks (0 to 100)   TINYINT (-127 to 127); INT
2. name (up to 50)   VARCHAR(50)
3. created_at   TIMESTAMP

Keep it simple: INT, VARCHAR(50), TIMESTAMP

Constraints

Rules that prevent bad data

What is a constraint?

Constraints are rules on columns or tables.
They help the database reject bad data.

Previously covered: PRIMARY KEY, AUTO_INCREMENT
Today we focus on FOREIGN KEY, NOT NULL, UNIQUE, DEFAULT

PRIMARY KEY and AUTO_INCREMENT

PRIMARY KEY: uniquely identifies each row.

AUTO_INCREMENT: MySQL generates the id for you.

Typical pattern:

 - id INT AUTO_INCREMENT PRIMARY KEY

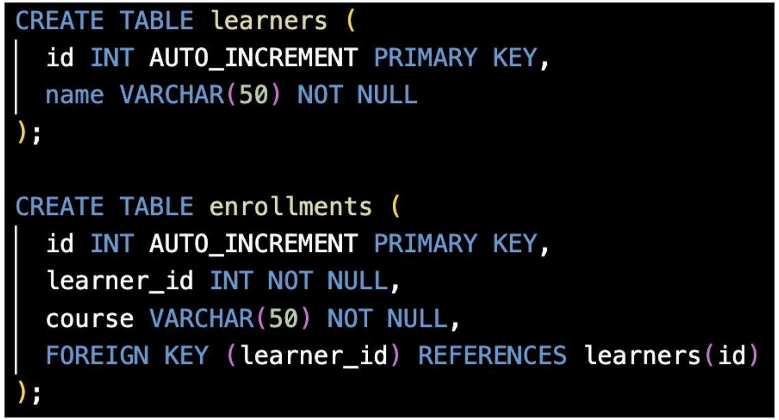
Read it in English:

 - define a column named id
 - whose data type is integer
 - where the id start at 1 for the first inserted row and
 - will increase by 1 for each new inserted row and
 - will act as the unique key to access any row

FOREIGN KEY

- A foreign key links a column in one table to the primary key of another table
- It helps keep relationships valid. Example: an enrollment must point to a real learner
- It prevents “orphan” rows (rows that reference an id that does not exist)

Example:



```sql
CREATE TABLE learners (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(50) NOT NULL
);

CREATE TABLE enrollments (
  id INT AUTO_INCREMENT PRIMARY KEY,
  learner_id INT NOT NULL,
  course VARCHAR(50) NOT NULL,
  FOREIGN KEY (learner_id) REFERENCES learners(id)
);
```

NOT NULL

NOT NULL means a value is required.

Example:

name VARCHAR(50) NOT NULL

Read it in English: name can have a maximum of 50 characters AND should not be empty

Note: If you try to insert NULL, the query fails.

UNIQUE

UNIQUE prevents duplicate values in a column.

Example:

```
email VARCHAR(100) UNIQUE
```

Read it in English: email can have a maximum of 100 characters AND should be unique in the table

Note: If you insert a duplicate, the query fails.

DEFAULT

DEFAULT sets a value when you do not provide one.

Example: is_active defaults to true, indicating that a product is active.

This helps keep inserts simple and consistent.

Quick check

If email is UNIQUE, what happens if we insert the same email twice?

1. Allowed
2. Fails
3. Updates old row

Answer is 2

Better SELECT

Make queries useful: sort, limit, filter patterns

Select columns (not always *)

Pick only the columns you need.

```
SELECT name, marks
FROM learners;
```

ORDER BY: sort results

Sort results by a column. Use DESC for high to low.

```
SELECT *
FROM learners
ORDER BY marks DESC;
```

LIMIT: top N rows

LIMIT shows only a fixed number of rows.

```
SELECT *
FROM learners
ORDER BY marks DESC
LIMIT 5;
```

DISTINCT: unique values

DISTINCT removes duplicates in the output.

```
SELECT DISTINCT marks
FROM learners
ORDER BY marks;
```

WHERE with AND / OR

Combine conditions using AND and OR.

```
SELECT *
FROM learners
WHERE marks >= 60 AND name LIKE 'A%';
– “Abc”, “AAbc”

SELECT *
FROM learners
WHERE marks >= 60 OR name LIKE 'A%';
```

WHERE with IN and BETWEEN

IN matches a list. BETWEEN matches a range.

```
SELECT *
FROM learners
WHERE marks IN (50, 60, 70);

SELECT *
FROM learners
WHERE marks BETWEEN 60 AND 80;
```

LIKE: simple pattern search

% means any characters.

```
SELECT *
FROM learners
WHERE name LIKE 'Me%';
```

Read it like English

What does this return?

Type one sentence in chat.

```
SELECT name, marks
FROM learners
WHERE marks >= 60
ORDER BY marks DESC
LIMIT 3;

Selecting name & marks from the learners table
Where marks is greater than or equal to 60
Order the rows by the marks columns, in a descending fashion
And we finally return 3 rows
```

Aggregates

One answer from many rows

What is an aggregate?

Aggregate functions summarize many rows into a smaller answer.

Examples of questions:

1. How many learners are there?
2. What is the highest mark?
3. What is the average mark?

COUNT: how many rows?

COUNT returns the number of rows.

```
SELECT COUNT(*) AS total_learners
FROM learners;
```

MIN and MAX

MIN is smallest value. MAX is largest value.

```
SELECT MIN(marks) AS lowest, MAX(marks) AS highest
FROM learners;
```

AVG and SUM

AVG is average. SUM is total.

```
SELECT AVG(marks) AS avg_marks
FROM learners;

SELECT SUM(marks) AS total_marks
FROM learners;
```

GROUP BY (preview only)

GROUP BY is used when you want one aggregate per group.

Example: average marks per batch or per city.

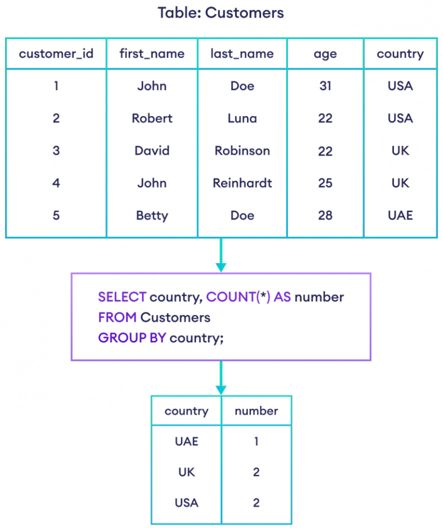


Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT country, COUNT(*) AS number
FROM Customers
GROUP BY country;
```

country	number
UAE	1
UK	2
USA	2

Quick check (chat)

Which aggregate fits?

1. How many learners scored above 80? (COUNT, MAX, AVG)
2. What is the highest mark? (COUNT, MAX, AVG)
3. What is the average mark? (COUNT, MAX, AVG)

Expected: 1 COUNT, 2 MAX, 3 AVG

Joins Preview

Why and how tables connect

Why multiple tables exist

A single big table causes repetition and messy updates.

Databases usually split data into related tables.

Example idea:

- learners table (who)
- enrollments table (which course)

Tables connect using ids

We connect tables using matching ids.

Example columns:

- learners.id
- enrollments.learner_id

Join idea: match rows where these values are equal.

Tiny example tables

learners

1	Asha
2	Rahul
3	Meera

enrollments

201	learner_id=1	course=SQL
202	learner_id=2	course=Java
203	learner_id=2	course=MySQL

Question: What is Rahul enrolled in?

What a join produces

A join combines related data in one result.

Output idea:

```
Rahul  Java
Rahul  MySQL
Asha   SQL
```

INNER JOIN (preview)

INNER JOIN returns rows only when a match exists in both tables.

```
SELECT l.name, e.course
FROM learners l
INNER JOIN enrollments e
ON l.id = e.learner_id;
```

We will go deeper later (join types are not for today)

Read it like English

Start with learners.

Match enrollments where id equals learner_id.

Show name and course.

Prompt: explain the result in one sentence.

What we are NOT doing today

Not today:

1. LEFT JOIN or RIGHT JOIN
2. Multi-table joins
3. Complex join conditions

Goal today is only the join idea.