

Advanced Database Labs

ADB Labs

Amira Nasreldeen, Aseel, Salma

© 2025 Amira – Aseel – Salma

Table of contents

1. Advanced Database Labs	4
1.1 What You Will Learn	4
1.2 Tutorial Support video	4
1.3 Labs Overview	4
1.4 What You Should Do	4
2. General Instructions	5
2.1 Please follow these instructions carefully:	5
3. Setup environment	6
3.1 Prerequisites: Install Microsoft Visual C++ 2019 Redistributable	7
3.2 MySQL	7
3.3 Python programming language	13
3.4 Alternative way to practice:	15
3.5 Assignment: Install MySQL, Create a Database, and Query It	15
4. Transaction Management and SQL Implementation	16
4.1 Understanding SQL Transactions	16
4.2 Assignment: Simulate The Unrepeatable Read Problem	26
5. Concurrency Control Techniques	28
5.1 InnoDB Storage Engine	28
5.2 Isolation Level on MySQL	28
5.3 InnoDB-Supported Concurrency Control Algorithms	29
5.4 Advanced work	34
5.5 Assignment (Simulate Deadlock Scenario)	34
6. Database Recovery Techniques	35
6.1 Basic Concepts Used in the Lab	35
6.2 Demonstrating InnoDB Crash Recovery Through a Practical Example	35
6.3 Assignment (Simulate Crash Recovery)	39
7. Database Security Measures	41
7.1 Objective:	41
7.2 Lab Content:	41
7.3 Database Security Principles	41
7.4 Practical Implementation of Database Security Concepts	42
7.5 1. Create Database and Tables	42
7.6 2. Create User Roles and Assign Permissions	43
7.7 3. Create Roles and Assign to User	43
7.8 4. Column-Level Encryption	45

7.9 5. Simulate SQL Injection & Prevention	46
8. Assignment 4	48
9. What to assign:	48
9.1 What to Submit	49

1. Advanced Database Labs

Welcome to the Advanced Database Lab course. This site provides hands-on labs to help you learn and simulate key concepts in **concurrency control** and **database recovery techniques**.

1.1 What You Will Learn

- How to install and use **MySQL**, **Python**, and **Visual Studio Code**
- An understanding of **transaction management** and **SQL implementation**
- Understanding concurrency control mechanisms (**2PL**, **Timestamp ordering**)
- Understanding Database recovery techniques (**ARIES**)
- An introduction to **database security** practices:
 - **Role-Based Access Control (RBAC)**
 - **Data encryption**

1.2 Tutorial Support video

Many labs include **video walkthroughs** that demonstrate how to solve key theoretical examples step by step.

- Transaction schedules and conflicts
- Concurrency Control mechanisms
- Database recovery Techniques

Be sure to watch them alongside the written material for deeper understanding.

1.3 Labs Overview

Each lab builds on the previous one. Please complete them in sequence:

1. **Lab 0:** Environment Setup
2. **Lab 1:** Basics of Transaction Management and Its Properties
3. **Lab 2:** Concurrency Control Mechanisms
4. **Lab 3:** Database Recovery Techniques (ARIES)
5. **Lab 4:** Database Security Measures



Note

You can download a single PDF that contains all labs. It will be updated each time a new lab is uploaded.

[Download"](#)

1.4 What You Should Do



Each lab includes an assignment that you are expected to complete as part of the learning process. **Please submit the assignments on time.**

2. General Instructions

2.1 Please follow these instructions carefully:

2.1.1 File Submission

- At the top of your document, please include the following information:
- Full Name
- Department
- Index Number
- Submit your assignment as a single PDF file.
- The PDF must include all required screenshots (code and output).

2.1.2 Lab Title

Write the lab title as "Lab 0, Lab 1, ..." clearly at the top of your PDF.

2.1.3 Screenshot Annotations

On every screenshot containing SQL script, add a comment at the top with the following information:

1. Student Name
2. Department
3. Student Index Number

```
-- Name: student name  
-- Department: Computer Science, ...  
-- Index: 01x-xxx
```

2.1.4 Deadline

1. Submissions must be uploaded before the deadline.
2. Late submissions may be penalised according to the situations.

2.1.5 Plagiarism

- You may discuss the assignment with other students, but all work must be completed individually.
- Plagiarism will be identified and penalized appropriately based on the circumstances.
- The use of AI tools to complete your tasks is not allowed. If detected, it will result in penalties.

3. Setup environment

Objective:

- To set up **MySQL Database Management System**
- To connect to **MySQL** server
- To set up **Python Programming Language**

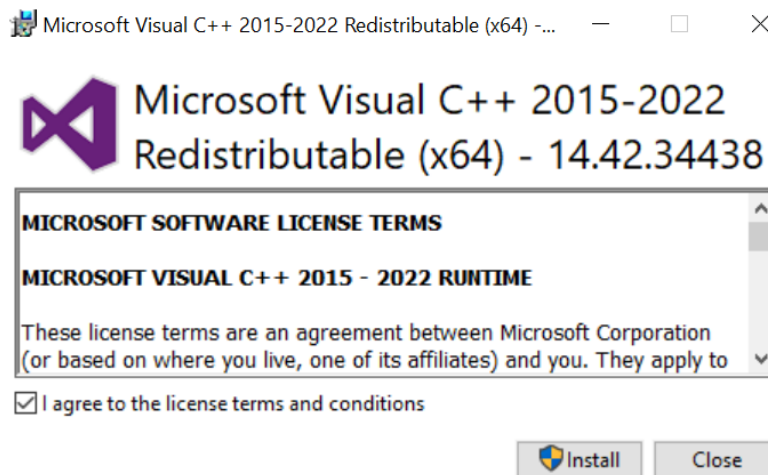
Software:

- MySQL installer (offline version)
- Visual Studio code (VS code)

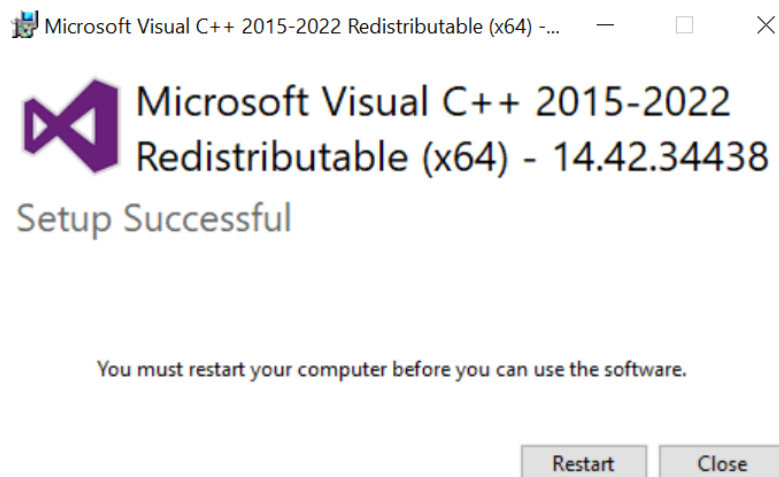
3.1 Prerequisites: Install Microsoft Visual C++ 2019 Redistributable

If you already have Microsoft Visual C++ 2019 Redistributable installed, you can skip this step and continue with the lab.

1. Go to the official Microsoft page. [Download VC++ Redistributables](#)
2. Download the file based on your system:
 - for 64-bit [VC_redist.x64.exe](#)
 - for 32-bit [VC_redist.x86.exe](#)
3. Open the downloaded file and check on **I agree to the license terms and conditions**.



4. After installation, **restart your PC**.



3.2 MySQL

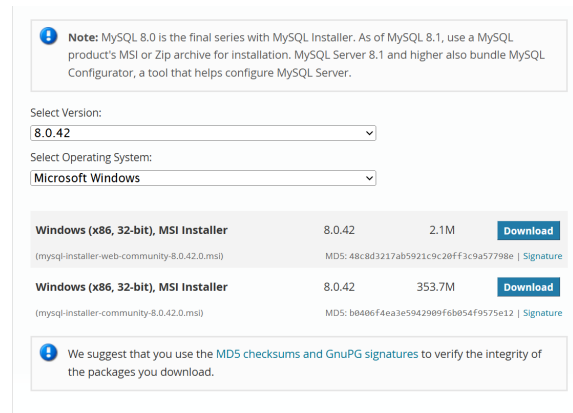
MySQL is an open-source relational database management system (RDBMS) that uses SQL to create, manage, and query data.

3.2.1 Why we will use MySQL?

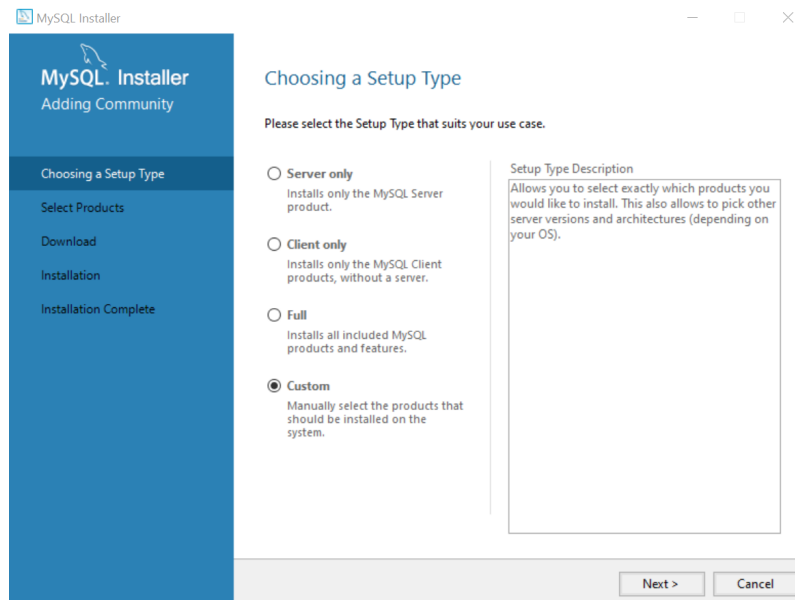
1. Support **ACID transaction properites**, essential for understanding transaction management
2. Uses the **InnoDB** transactional storage engine, which supports **concurrency control** and **recovery mechanisms**
3. Flexible and easy-to-use

3.2.2 Installing MySQL (Offline)

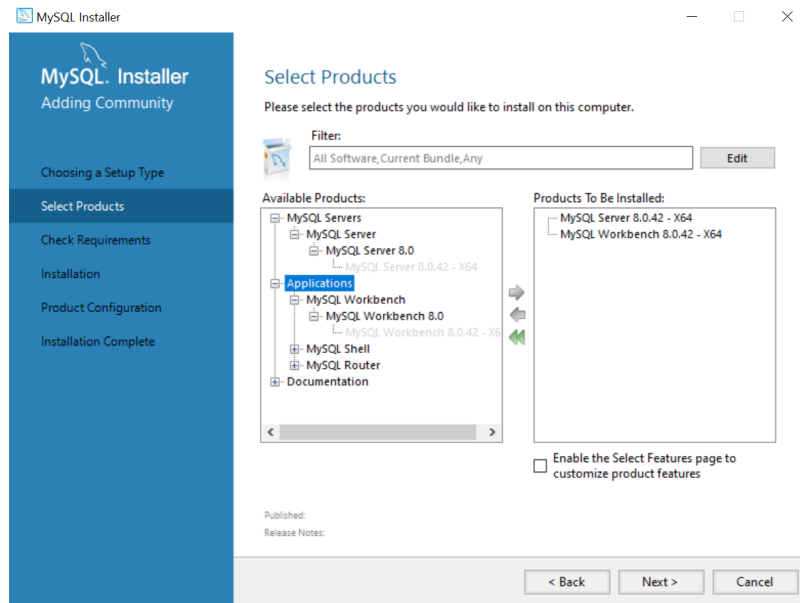
1. Go to the official [MySQL Downloads Page](#)
2. Download the **Windows (x86, 32-bit), MSI Installer** (Offline version ~353.7M)



3. After open the Installer Choose **Custom**

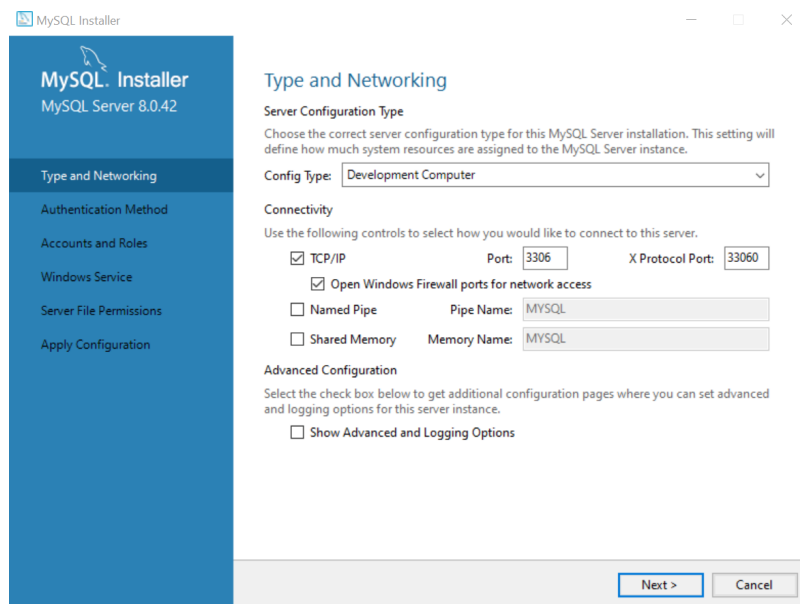


4. Select **MySQL Servers** and **MySQL Workbench**

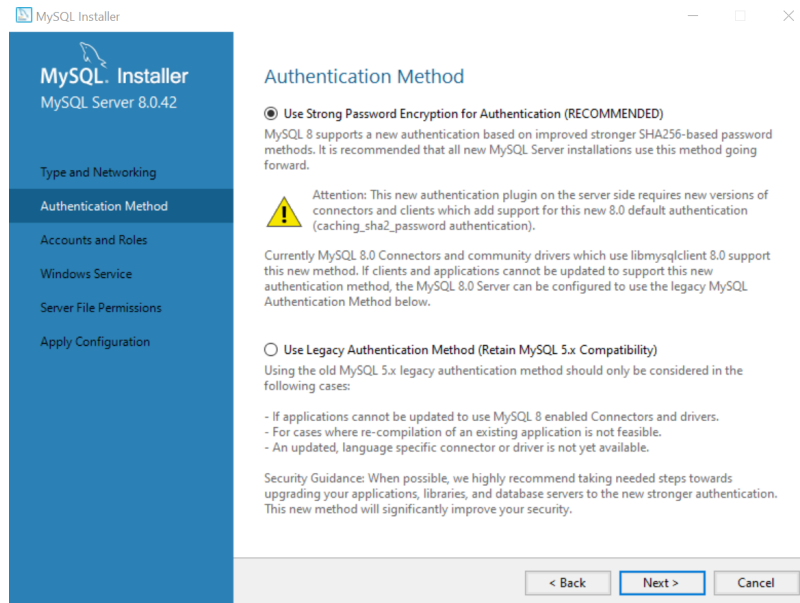


5. Proceed through the installation wizard

6. Use the default port (3306)

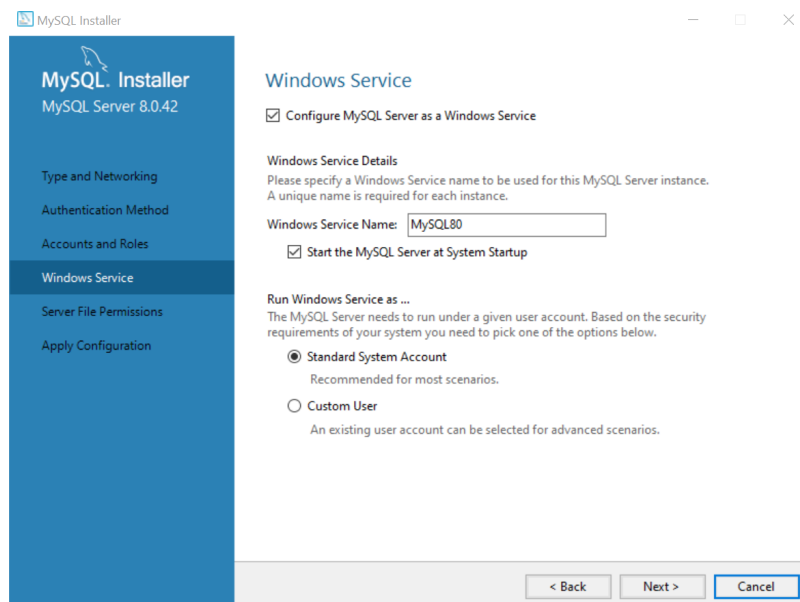


7. Choose the Authentication Method as below



8. Create a root password

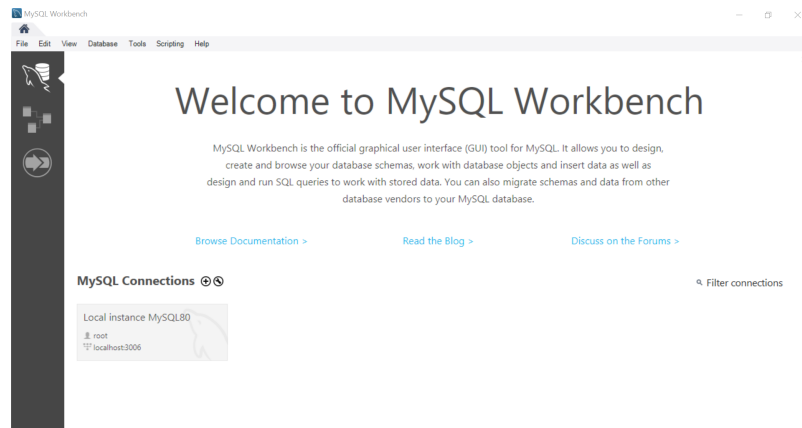
9. Choose Windows Service as below



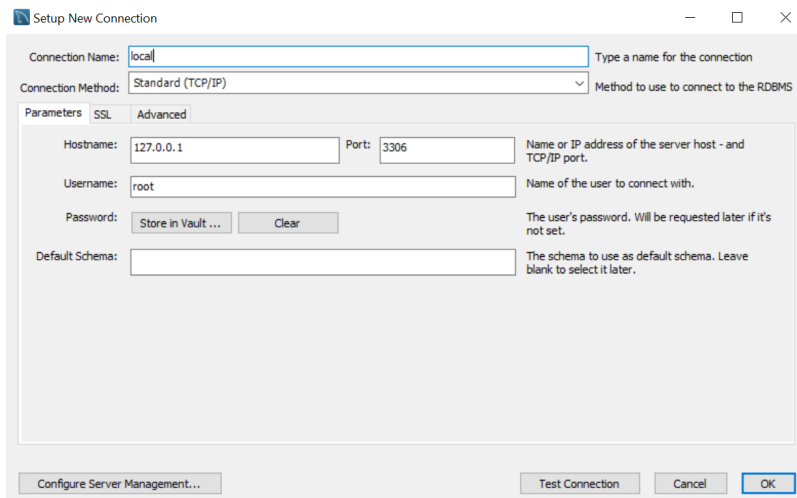
10. Proceed through the installation wizard until finish

3.2.3 What after install MySQL?

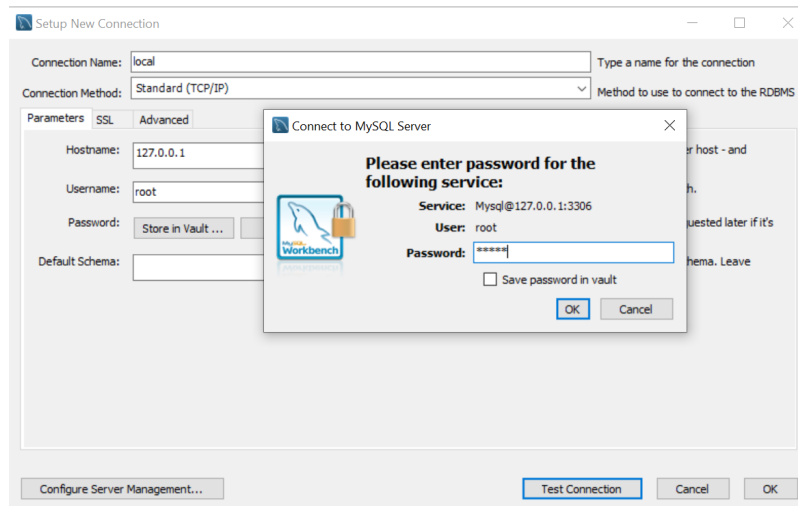
1. After installing MySQL, connect to the **MySQL Server**. by clicking the **+** symbol next to MySQL Connections




2. In the dialog box, enter a **Connection name**, e.g., `Local`

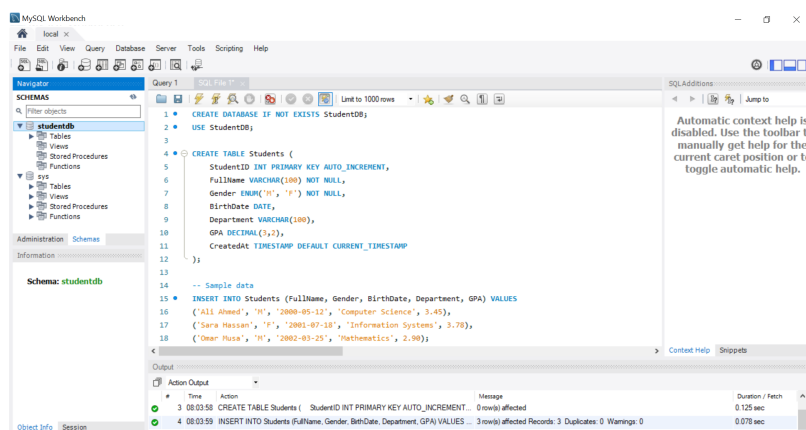


3. Click **Test Connection** and enter your MySQL root password to verify the connection



4. After successfully connecting to the MySQL server, create a new database called `StudentDB` :

- Open `local` connection
- Click **Create new SQL** from the top-left corner
- Write SQL commands to create the `StudentDB` database, a `Students` table, and insert sample data.
- Execute the Script using Execute button 



```

/*
Name: Amira Naser Aldeein
Index: 01x-xxx
Department: e.g., CS
*/

CREATE DATABASE IF NOT EXISTS StudentDB;
USE StudentDB;

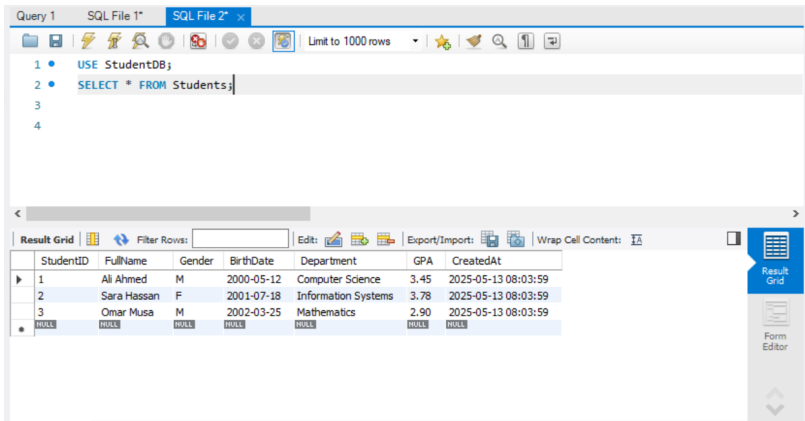
CREATE TABLE Students (
  StudentID INT PRIMARY KEY AUTO_INCREMENT,
  FullName VARCHAR(100) NOT NULL,
  Gender ENUM('M', 'F') NOT NULL,
  BirthDate DATE,
  Department VARCHAR(100),
  GPA DECIMAL(3,2),
  CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Sample data
INSERT INTO Students (FullName, Gender, BirthDate, Department, GPA) VALUES
('Ali Ahmed', 'M', '2000-05-12', 'Computer Science', 3.45),
('Sara Hassan', 'F', '2001-07-18', 'Information Systems', 3.78),
('Omar Musa', 'M', '2002-03-25', 'Mathematics', 2.90);

```

 Once you create the database, you will see it listed under the Schemas section on the left panel

5. Now you can run SQL queries on your new database



```
USE StudentDB;
SELECT * FROM Students;
```

What do we mean by Connection and Port?

A database connection is a communication link between a client application (e.g., MySQL Workbench) and the MySQL server. When an application (the client) needs to perform operations such as creating databases, running queries, or retrieving data, it establishes a connection to communicate with the server. The port is a communication endpoint on your machine. By default, MySQL uses port 3306 to listen for incoming client requests. In short, a connection is essential to allow your client tools or applications to interact with the MySQL server, run queries, manage data, and exchange information.

3.3 Python programming language

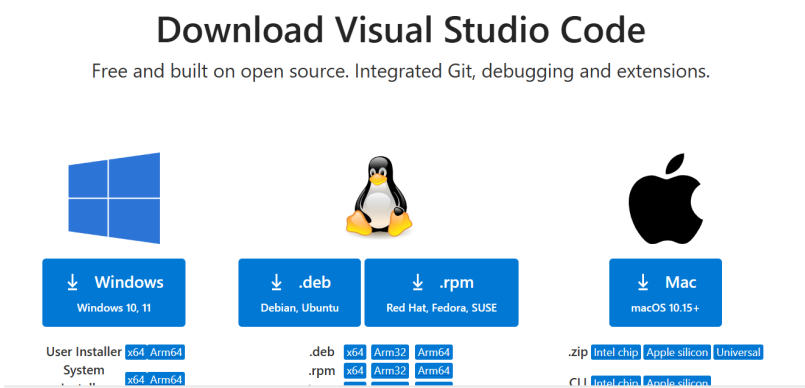
We will use the **Python programming language** to simulate the following: * Concurrency control algorithms * Database recovery techniques

3.3.1 Download and Install Python

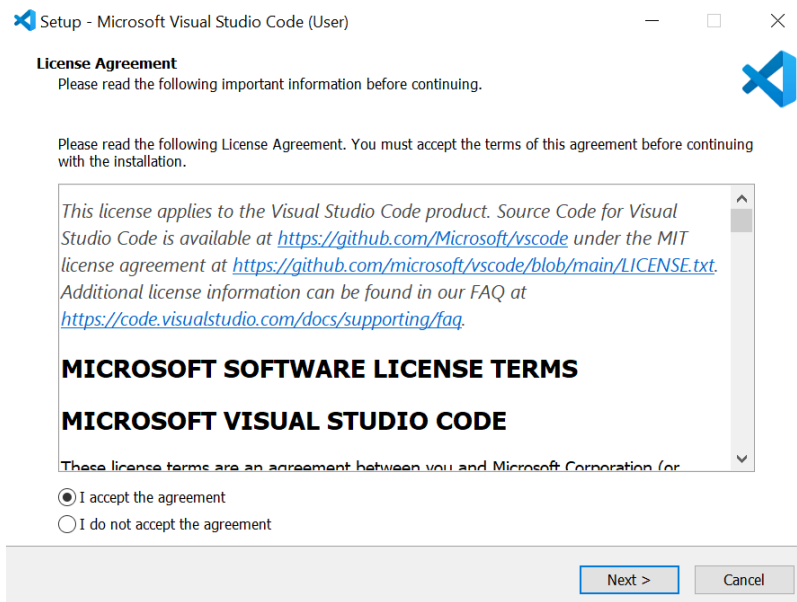
- 1. Go to the **Python** official website
- 2. Download the latest stable version
- 3. Open the installer and check
 - Add Python to PATH

3.3.2 Download and Install VS Code

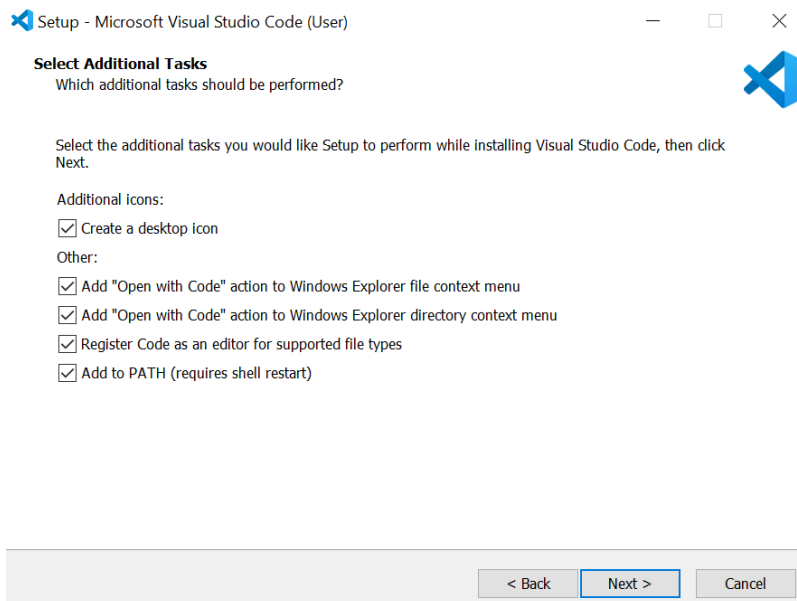
- 1. Visit the website **VS code** Click on **Windows** to download VS code for Windows, as shown below



- 2. After the download finished open the `VSCoDeUserSetup` executable file, when it open select `I accept the agreement` then click on `Next`



3. Select all the option as seen below



4. Finally we are ready to install the VS code, click on `install` and wait until the setup finish

5. On the Vs code, on the left-hand side, click on `extension` then in the search bar, write `Python` select `Python microsoft` and click on `Install` as shown below



3.4 Alternative way to practice:

- Online Postgres server [supabase](#)
- Online Python Editor [programiz](#)

3.5 Assignment: Install MySQL, Create a Database, and Query It

Due Date on 24/5/2025

See the requirement about the structures of the lab [here](#)

1. Download and Install MySQL
2. Connect to the server
3. Create a New Database
4. Create a Table
5. Insert Data
6. Query the Data

```
USE DatabaseName;
SELECT * FROM TableName;
```

3.5.1 What to assign:

Take Screenshots:

1. SQL code (on MySQL)
2. The output of your query
3. Put it all on one document

4. Transaction Management and SQL Implementation

Objective:

- Understand the ACID properties.
- Implement SQL transactions.
- Understand transaction control.

4.1 Understanding SQL Transactions

A **Transaction** is a sequence of read and write operations performed as a single logical unit of work.

SQL Server operates in the following transaction modes:

1. Autocommit transactions: Each individual statement is a transaction.
2. Explicit transactions: Each transaction is explicitly started with the `BEGIN TRANSACTION` statement and explicitly ended with a `COMMIT` or `ROLLBACK` statement (**which is concerned in this course**).
3. Implicit transactions: A new transaction is implicitly started when the prior transaction completes, but each transaction is explicitly completed with a `COMMIT` or `ROLLBACK` statement.

4.1.1 Transaction Control Commands – `SAVEPOINT`, `ROLLBACK`, and `COMMIT`

Objective: Understand how to manage multi-step database transactions using SQL transaction control commands.

BEGIN

Start a new transaction:

```
BEGIN;  
-- or  
START TRANSACTION;
```

SAVEPOINT

Set a point you can roll back to later:

```
BEGIN;  
  
INSERT INTO employees (id, name, department) VALUES (1, 'Alice', 'HR');  
SAVEPOINT sp1;  
  
INSERT INTO employees (id, name, department) VALUES (2, 'Bob', 'Finance');  
SAVEPOINT sp2;  
  
INSERT INTO employees (id, name, department) VALUES (3, 'Charlie', 'IT');
```

At this point, we have added 3 employees and created two savepoints: `sp1` and `sp2`.

ROLLBACK

Undo part or all of a transaction.

ROLLBACK TO A SAVEPOINT

```
ROLLBACK TO sp2;
```

This undoes only the insertion of **Charlie**, keeping **Alice** and **Bob**.

ROLLBACK ENTIRE TRANSACTION

```
ROLLBACK;
```

This undoes **all** changes made in the transaction.

COMMIT

Make all changes permanent:

```
COMMIT;
```

After committing, the changes cannot be undone.

Summary

- Use `SAVEPOINT` to mark logical checkpoints.
- Use `ROLLBACK` to undo mistakes.
- Use `COMMIT` to make sure the database saves your work.

This gives you control and safety when working with important or complex data operations.

4.1.2 Transaction properties and how SQL supports them

Transaction properties (ACID):

1. **A - Atomicity:** All actions in transaction happen, or none happen. “*All or nothing...*”
2. **C - Consistency:** If each transaction is consistent and the database starts in a consistent state, it ends in a consistent state. “*It looks correct to me...*”
3. **I - Isolation:** Execution of one transaction is isolated from other transactions. “*All by myself...*”
4. **D - Durability:** Once a transaction commits, its effects are permanent. “*I will survive...*”

How SQL Supports Each Property?

properties	SQL Feature
Atomicity	Use <code>START TRANSACTION</code> , followed by <code>COMMIT</code> or <code>ROLLBACK</code>
Consistency	Enforced through constraints (e.g., <code>PRIMARY KEY</code> , <code>FOREIGN KEY</code> , <code>CHECK</code>)
Isolation	Set using <code>SET TRANSACTION ISOLATION LEVEL</code>
Durability	Ensured by <code>COMMIT</code> and Logging mechanisms (e.g., write-ahead logs, redo logs)

4.1.3 Why Concurrency Control is Needed

Several problems can occur when concurrent transactions execute without proper control, especially due to conflicting operations such as **Read-Write**, **Write-Read**, and **Write-Write** conflicts.

Here is an illustrated problem with SQL transactions to simulate the problem:

1. Lost update problem

This problem occurs when two or more transactions read and update the same data item concurrently without being aware of each other's changes. As a result, the last update overwrites the previous ones, leading to the loss of some updates and causing data inconsistency.

Real-World Scenario

Two users (sessions) try to withdraw money from the same account at the same time:

- **User A** reads the balance = 100, decides to withdraw 30 → wants to update it to 70.
- **User B** also reads the balance = 100, decides to withdraw 50 → wants to update it to 50.
- If both updates happen without proper control, one will overwrite the other, leads to lost updates and incorrect final balance.

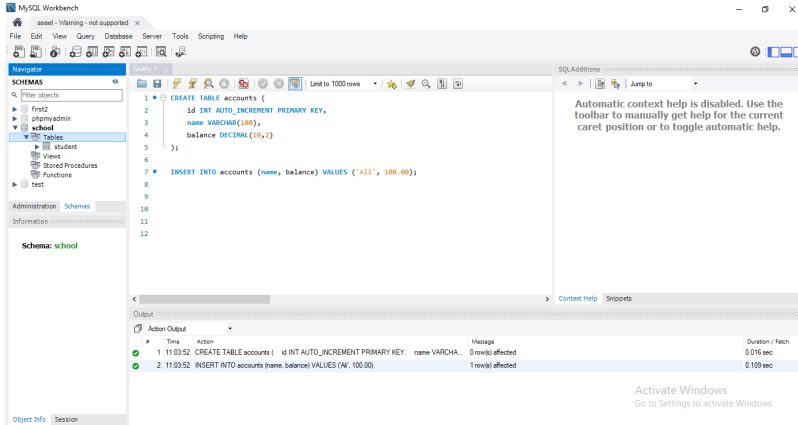
Example of the Scenario in SQL:

To simulate a Lost Update in MySQL using two concurrent sessions:

1. Create a Table: Create any Database and create a table named `accounts` and insert sample data into it:

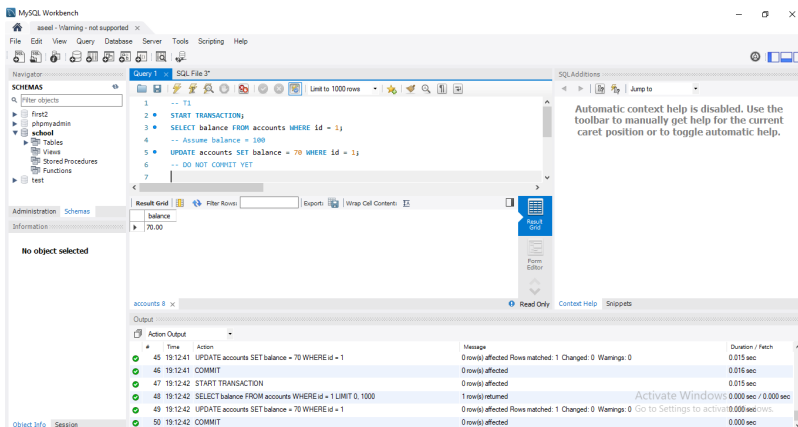
```
CREATE TABLE accounts (
  id SERIAL PRIMARY KEY,
  name TEXT,
  balance NUMERIC(10,2)
);

INSERT INTO accounts (name, balance) VALUES ('Ali', 100.00);
```



2. Session A (user A):

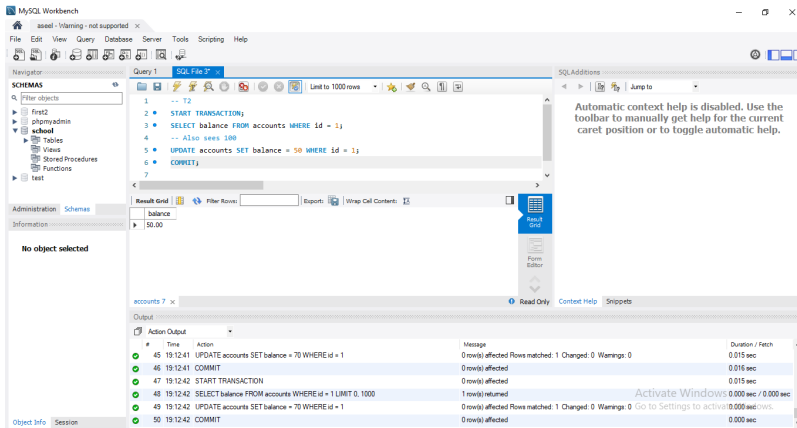
```
START TRANSACTION;
SELECT balance FROM accounts WHERE id = 1;
UPDATE accounts SET balance = 70 WHERE id = 1;
```



Here, user A reads the balance (100) and calculates the new balance: $100 - 30 = 70$, then updates the record without commit.

3. Session B (user B, in another window):

```
START TRANSACTION;
SELECT balance FROM accounts WHERE id = 1;
UPDATE accounts SET balance = 50 WHERE id = 1;
COMMIT;
```

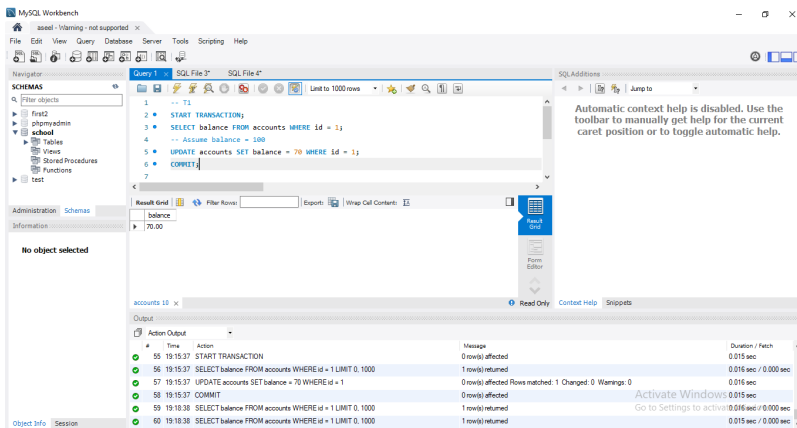


Here, user B reads the balance (100) and calculates the new balance:

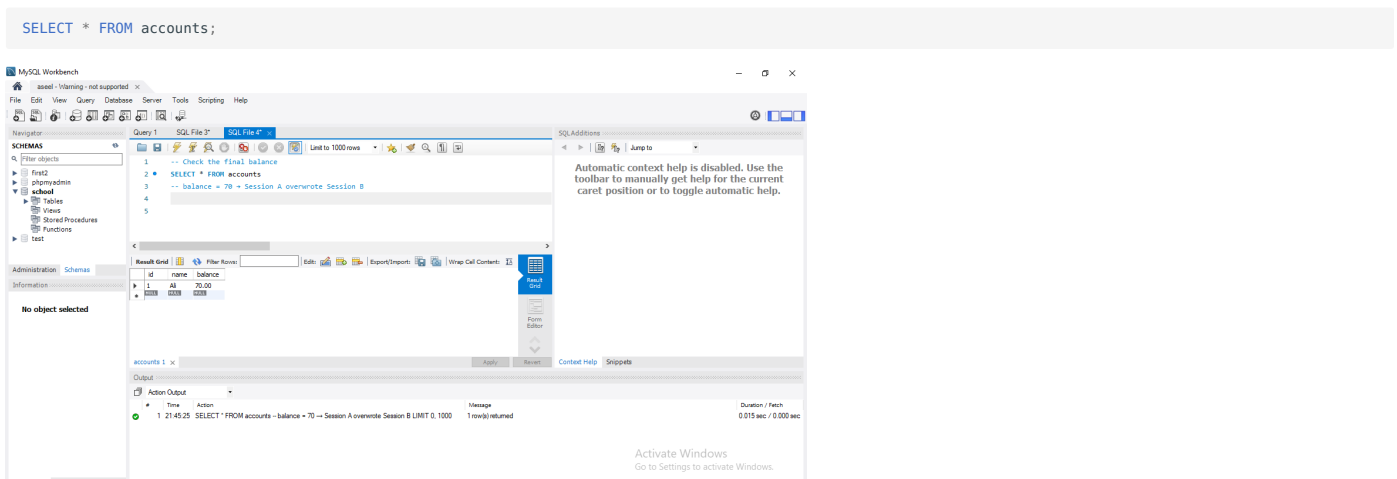
$100 - 50 = 50$, then updates the record and commit.

to open two sessions in MySQL Workbench; Go to File → New Query Tab to open Session A, Repeat to open another tab (Session B). Each tab is a separate session — you can run SQL commands independently in each

4. Back to session A: Now return back to session A again and commit the session:



5. Check the final balance:



Here we will see the balance becomes 70, and user B's update (which should make the balance 50) is lost.

6. Conclusion:

- Both transactions read the same initial value (100).
 - Each transaction updates the data without being aware of the other's changes.
 - The last transaction to commit determines the final value, possibly erasing the other's update.
 - Even though no errors appear, data integrity is lost. This demonstrates the importance of proper concurrency control in database systems.
-

2. The Temporary Update (or Dirty Read) Problem

This occurs when one transaction reads data that has been written by another transaction that hasn't committed yet. If the first transaction is later **rolled back**, the second transaction ends up working with invalid or temporary data — a dirty read.

Example Scenario (Simulated):

In an online electronics store:

1. Ahmed initiates a purchase of 200 laptops. A transaction begins and temporarily reduces the stock quantity to 0, but the transaction is not yet committed.
2. Meanwhile, Mohamed checks the available stock and sees 0 laptops — based on User A's uncommitted change.
3. Then, Ahmed cancels the purchase, and the transaction is rolled back. The stock returns to its original quantity (e.g., 200).
4. However, Mohamed made a decision based on incorrect (temporary) data — the system gave them a dirty read.

SQL Transaction for the scienario:

1. Create the `Products` table to simulate the scenario

```
-- Create ProductDB Database
CREATE DATABASE IF NOT EXISTS ProductDB;
USE ProductDB;

-- Create Products table
CREATE TABLE Products (
  Id INT PRIMARY KEY,
  ProductName VARCHAR(100),
  Quantity INT
);

-- Insert Sample data
INSERT INTO Products (Id, ProductName, Quantity) VALUES
(1, 'Mobile', 100),
(2, 'Tablet', 50),
(3, 'Laptop', 200);
```

2. In Session 1: Connect using the previous connection from **Lab0** (named local), and open a new SQL query tab

```
USE ProductDB;

-- Transaction 1: Simulate purchase
START TRANSACTION;
-- Pay 200 units from product with ID = 3 (assumed to be a laptop)
UPDATE Products
SET Quantity = Quantity - 200
WHERE Id = 3;
-- Simulate waiting for payment confirmation
DO SLEEP(15);
-- Payment failed – rollback the transaction
ROLLBACK;
```

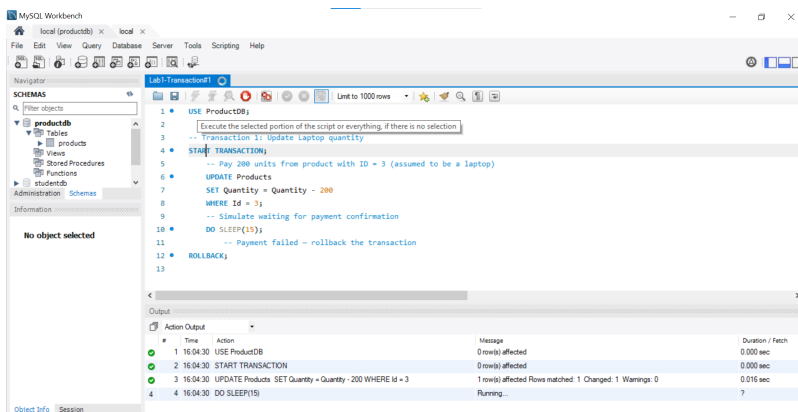
We use `SLEEP` to simulate concurrent transactions. Note that `DO SLEEP(15)` works only in MySQL 8.0.29 or later

3. Go to the Home tab and open the `local` connection again. (This step simulates a second session connected to the database.) Then, open a new SQL query tab to continue.

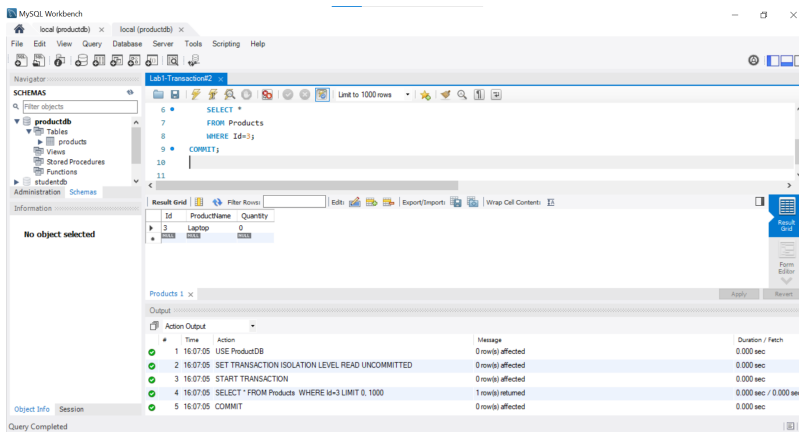
```
USE ProductDB;

-- Transaction 2: Retrieve Laptop record
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT *
FROM Products
WHERE Id=3;
COMMIT;
```

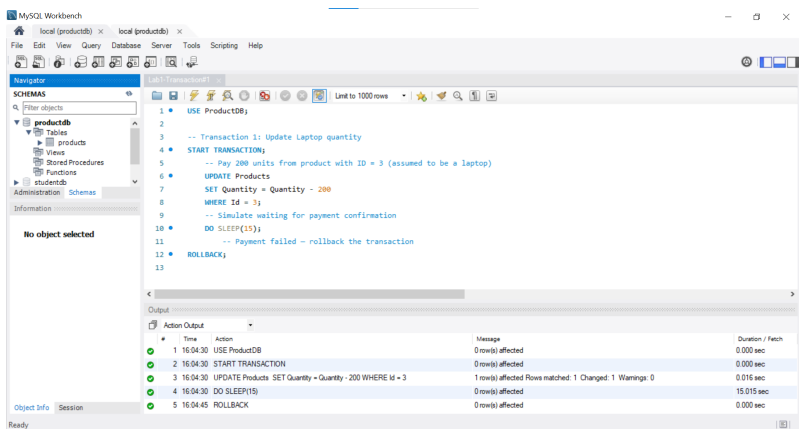
4. In Session 1: Run the transaction and wait



5. While Session 1 is still sleeping: Run Session 2



6. After Session 1 completes and rolls back



7. Conclusion: Session 2 was able to see the updated quantity (0 laptops) before Session 1 rolled back. This leads to inconsistent or misleading data, which is why isolation levels like `READ COMMITTED` or `REPEATABLE READ` are important in real-world applications.

3. The Incorrect Summary Problem

Problem Statement

When you use aggregate functions like `SUM`, `AVG`, and `COUNT` on a table while other transactions are modifying the data, the result may be **inconsistent**. This is because some rows might be updated **before** the function reads them, and others **after**.

1. Create the Table

```
CREATE TABLE accounts (
  id INT PRIMARY KEY,
  balance DECIMAL(10,2)
);
```

2. Insert Initial Data

```
INSERT INTO accounts (id, balance) VALUES
(1, 500.00),
(2, 750.00),
(3, 1200.00);
```

3. Simulate Concurrent Transactions

Transaction T1 (Session 1)

```
BEGIN;
UPDATE accounts SET balance = balance + 100 WHERE id = 1;
-- not committed
```

Transaction T2 (Session 2)

```
SELECT SUM(balance) FROM accounts;
```

At this point, `SUM()` still uses the old value for `id = 1` because the update in Session 1 (T1) hasn't been committed.

SUM(balance) = 2450.00

4. Add COMMIT to T1 and Re-run

Transaction T1 (Session 1)

```
BEGIN;
UPDATE accounts SET balance = balance + 100 WHERE id = 1;
COMMIT;
```

Transaction T2 (Session 2)

```
SELECT SUM(balance) FROM accounts;
```

Now, `SUM()` uses the new value for `id = 1` because the update in T1 has been committed.

SUM(balance) = 2550.00

5. Conclusion:

- If T1 is not committed, T2 sees old values.
- If T1 is committed, T2 sees updated values.
- Proper transaction management ensures consistent query results.

4.2 Assignment: Simulate The Unrepeatable Read Problem

 **Due Date on 14/6/2025**

- Read page 752 on the primary book (FUNDAMENTALS OF Database Systems)
- See the requirement about the structures of the lab [here](#)
- You can use any SQL editor (Online or Local), but be cautious — **transaction statements may vary slightly between different DBMSs**, and the use of `DO SLEEP()` depends on the specific database engine. For example, `DO SLEEP()` is supported in MySQL 8.0.29+, while in other systems or older versions, you may need to use `SELECT SLEEP()` or an alternative approach.

4.2.1 What to assign:

1. A paragraph describing the scenario you will use to simulate the Unrepeatable Read problem.
2. A screenshot of the table you created for the simulation.
3. A screenshot showing the result of the first transaction after execution.
4. A screenshot showing the result of the second transaction after execution.
5. A paragraph discussing what happened, and how it demonstrates the Unrepeatable Read issue.

5. Concurrency Control Techniques

Objective:

- To understand concurrency control techniques based on locking mechanisms.
- To explore concurrency control techniques based on the multiversion (MVCC) concept.
- To understand how MySQL (InnoDB) handles concurrent transactions effectively.

5.1 InnoDB Storage Engine

A storage engine is a software that is used by a database management system to create, read, and update data from a database. **InnoDB** is a general purpose **transactional storage engine** for MySQL (since version 5.5.5), designed for high reliability and performance.

5.1.1 Why Do We Use InnoDB in This Course?

- Its DML operations follow the **ACID** model, supporting transactions with **commit**, **rollback**, and **crash recovery**, which help protect user data.
- It allows us to observe how **concurrency control algorithms** work within a DBMS, including **strict Two-Phase Locking (2PL)** and **Multi-Version Concurrency Control (MVCC)**.
- It enables us to examine **recovery mechanisms**, such as **write-ahead logging (WAL)**, in action.

5.1.2 How to check if InnoDB enabled on MySQL?

To verify whether InnoDB is enabled and set as the default storage engine, execute the following query:

```
SHOW ENGINES;
```

Look for the InnoDB row in the result. It should show **DEFAULT** in the Support column, indicating that InnoDB is the default storage engine.

InnoDB	DEFAULT	*Supports transactions, row-level locking, and foreign k...	YES	YES	YES
--------	---------	-------------------------------------------------------------	-----	-----	-----

5.1.3 InnoDB Locking

- **Shared and Exclusive Locks:** InnoDB implements standard row-level locking, which includes two types of locks: shared (S) locks and exclusive (X) locks.
- **Intention Locks** InnoDB supports multiple granularity locking, allowing row-level locks and table-level locks to coexist. It introduces two types of intention locks: intention shared (IS) and intention exclusive (IX). For details, refer to the lock compatibility matrix on page 803 of the book.


5.2 Isolation Level on MySQL

Isolation level **instruct** the database engine on how to manage multiple transactions being performed concurrently, and what violations are possible.

1. **READ UNCOMMITTED** is the lowest isolation level. It allows transactions to read the most recent version of a row, even if the change has not been committed by other transactions. This leads to the dirty read anomaly, as explained in [Lab 1](#).
2. **READ COMMITTED** is one step above **READ UNCOMMITTED** and prevents dirty reads. In this mode, each **SELECT** operation retrieves the latest committed version of the row at the time of the query. However, this can result in non-repeatable reads.
3. **REPEATABLE READ** is the default isolation level in MySQL. It is one step above **READ COMMITTED** and prevents non-repeatable reads. In this mode, shared locks are placed on all rows read during the transaction and are held until the transaction ends. This prevents other transactions from modifying any of the data that was read.
4. **SERIALIZABLE** is the highest isolation level. It prevents all types of concurrency anomalies. However, it is often impractical due to the extensive locking it requires, which increases the risk of deadlocks and can significantly reduce performance.


Syntax for Setting the Isolation Level in MySQL

```
SET TRANSACTION ISOLATION LEVEL [ISOLATION LEVEL];
```

 InnoDB use `REPEATABLE READ` as the default isolation level.

5.3 InnoDB-Supported Concurrency Control Algorithms

The InnoDB transaction model combines the strengths of Multi-Version Concurrency Control (MVCC) and strict Two-Phase Locking (2PL) to balance high concurrency with strong consistency guarantees. InnoDB uses MVCC to enable non-blocking reads, allowing multiple transactions to read data simultaneously without locking. At the same time, it applies strict 2PL for write operations to ensure serializability.

 When we refer to write operations, we mean SQL statements such as `INSERT`, `UPDATE` and `DELETE`

InnoDB uses two primary concurrency control mechanisms:

5.3.1 1. Two-Phase Locking (2PL)

InnoDB uses **strict two-phase locking** to ensure serializability. This means a transaction acquires locks during its execution phase but does not release any exclusive (write) locks until it either commits or rolls back

Example: Observing How InnoDB Uses Strict 2PL

Imagine Ahmed's family and Mohamed's family both trying to book seats for a summer vacation trip to Turkey. They access the reservation system at the same time. Ahmed wants to reserve five seats, while Mohamed wants to reserve four seats. Behind the scenes, InnoDB ensures data consistency by using strict two-phase locking (2PL) — meaning when Ahmed's transaction starts updating the available seat count, it locks the row exclusively. Mohamed's transaction, attempting to update the same row simultaneously, must wait until Ahmed's transaction commits or rolls back. This prevents overlapping writes and guarantees serializability.

Now we will simulate the example using MySQL and observe the InnoDB storage engine:

1. Create the Database and Table

```
CREATE DATABASE IF NOT EXISTS ReservationDB;
USE ReservationDB;

CREATE TABLE seats (
  Id INT PRIMARY KEY AUTO_INCREMENT,
  FlyName VARCHAR(100) NOT NULL,      -- Flight Name
  Available_seats INT                 -- Available seats on the flight
) ENGINE=InnoDB;

-- Insert some sample data
INSERT INTO seats (FlyName, Available_seats) VALUES
  ("Turkey", 10),
  ("Emaraty", 20);
```

2. In Session 1 (Ahmed's Transaction): Connect using the previous connection from **Lab0** (named local), and open a new SQL query tab

```
USE ReservationDB;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;

-- Ahmed tries to reserve 5 seats
UPDATE seats SET available_seats = available_seats - 5 WHERE id = 1;

-- Optional: Use SLEEP() to pause Ahmed's transaction.
-- This allows you to observe the lock behavior by giving Mohamed's session time to run concurrently.
-- If you prefer not to use SLEEP(), you can simply delay committing Ahmed's transaction
-- and manually run Mohamed's session before issuing the COMMIT.
DO SLEEP(20);

COMMIT;
```

3. In Session 2 (Mohamed's Transaction): Go to the Home tab and open the **local** connection again. (This step simulates a second session connected to the database.) Then, open a new SQL query tab to continue.

```
-- Mohamed's Transaction

USE ReservationDB;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;

-- Mohamed tries to reserve 4 seats
UPDATE seats SET available_seats = available_seats - 4 WHERE id = 1;

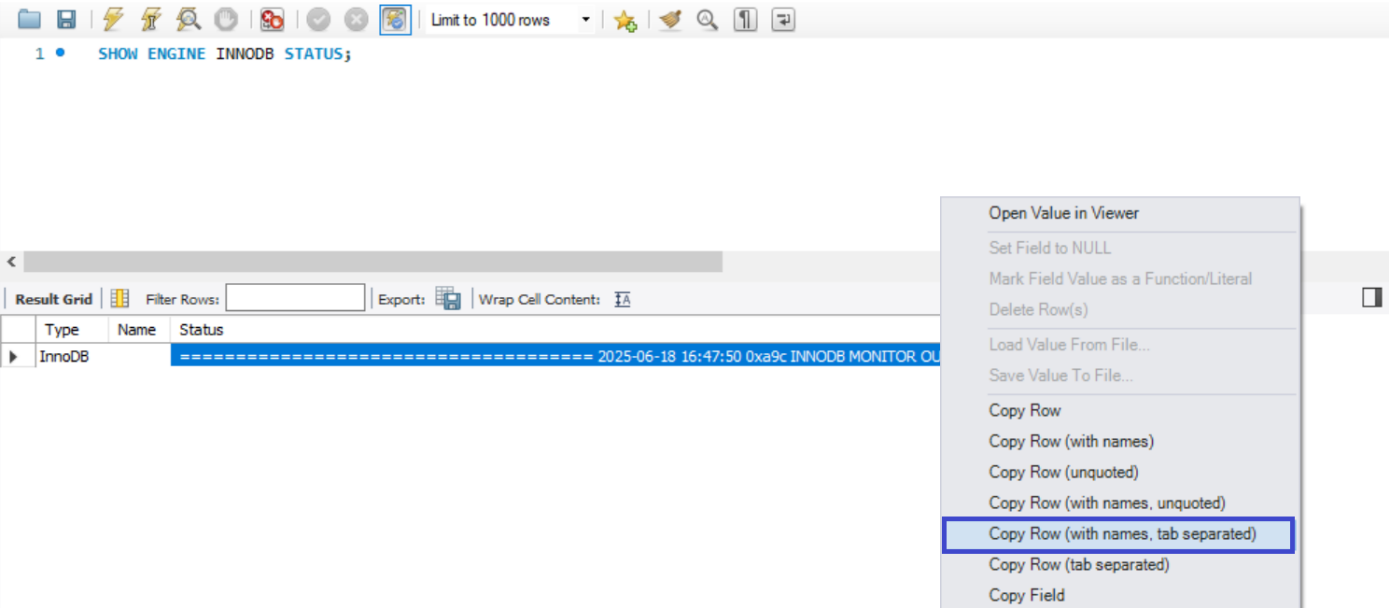
-- If Ahmed's transaction hasn't committed yet, this query will be BLOCKED.
-- It waits for the exclusive lock to be released.
COMMIT;
```

4. In Session 3

```
SHOW ENGINE INNODB STATUS;
```

`SHOW ENGINE INNODB STATUS` provides a detailed snapshot of the InnoDB storage engine's internal state. This output—often referred to as the InnoDB Monitor—includes several sections such as transaction and lock information, I/O activity, buffer pool usage, deadlock detection, and more.

- Run Session 1, Session 2, and Session 3 simultaneously. You will observe that the `UPDATE` operation in Session 2 keeps running (waiting) until Session 1 commits. Meanwhile, execute Session 3 during this wait period to view the InnoDB status and observe how Session 2 is blocked, waiting for Session 1 to release the lock.
- Right-click on the Status column in the InnoDB status output and select **Copy Row (with names, tab separated)**, then paste it into any text editor (e.g., Notepad) to examine the output in detail. In the snapshot below, you can see evidence that Transaction 2 is waiting for Transaction 1 to commit—this clearly demonstrates how InnoDB enforces strict two-phase locking (2PL).



Transaction 2

```
---TRANSACTION 4939, ACTIVE 3 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 1128, 1 row lock(s)
MySQL thread id 81, OS thread handle 14892, query id 3521 localhost 127.0.0.1 root updating
UPDATE seats SET available_seats = available_seats - 4 WHERE id = 1
----- TRX HAS BEEN WAITING 3 SEC FOR THIS LOCK TO BE GRANTED:
```

Transaction 1

```
---TRANSACTION 4938, ACTIVE 5 sec
2 lock struct(s), heap size 1128, 1 row lock(s), undo log entries 1
MySQL thread id 77, OS thread handle 14828, query id 3515 localhost 127.0.0.1 root User sleep
DO SLEEP(20)
```

Once the sleep duration completes, Transaction 1 commits and releases its exclusive lock, allowing Transaction 2 to acquire the lock and proceed. If you query the `available_seats`, you will find it equals 1, indicating that both transactions executed without any anomalies, as demonstrated in [Lab 1](#).

7. Conculation on this example we observe that how InnoDB follow the strict 2PL on updating operation. you can repeat this example by using `READ COMMITTED`, `SERIALIZABLE` Isolation level and observe what happen.

5.3.2 2. Multiversion Concurrency Control (MVCC)

Definition

MVCC is a concurrency control technique used by many modern databases (like PostgreSQL, MySQL InnoDB, and Oracle) to allow multiple transactions to access the same data simultaneously without locking.

How MVCC Works

- Every time a transaction reads data, it sees a snapshot of the database at the time the transaction started.
- When data is updated, the database creates a new version of the row instead of overwriting the old one.
- Other transactions can continue reading the old version until they commit or refresh.

Objective

Understand how MVCC in InnoDB allows multiple read operations to happen at the same time, without locking, and while updates may be happening in the background.

Step 1: Create the Table

```
CREATE TABLE employees (  
  id INT PRIMARY KEY,  
  name VARCHAR(100),  
  salary INT  
) ENGINE=InnoDB;
```

Step 2: Insert Initial Data

```
INSERT INTO employees (id, name, salary) VALUES  
(1, 'Alice', 5000),  
(2, 'Bob', 6000);
```

Step 3: Start Two Sessions

We will simulate **Session A (Reader)** and **Session B (Updater)**. You can do this using two different database connections (e.g., two tabs in MySQL Workbench).

Session A (Reader)

```
START TRANSACTION;  
SELECT * FROM employees WHERE id = 2;
```

Observation: All rows are returning back, WHERE id = 2 returns 6000.

Session B (Updater) (In another tab)

```
START TRANSACTION;  
UPDATE employees SET salary = 9000 WHERE id = 2;  
COMMIT;
```

Back to Session A (Reader Again)

```
SELECT * FROM employees WHERE id = 2;
```

Observation: Now it shows 6000 for id=2.

Transaction A does not see the update from Transaction B because MVCC provides a consistent snapshot of the data at the moment A starts.

MVCC keeps multiple versions of rows, so Transaction A reads the version that existed before B's update. Even if B commits, A continues to see the old data until it finishes, ensuring repeatable reads without blocking.

5.3.3 Key Discussion Points

Feature	Observation
MVCC	Keeps a snapshot per transaction (using undo logs).
Multiple Reads	Readers don't block writers, and writers don't block readers.
Isolation	REPEATABLE READ is default in InnoDB – keeps results stable within a transaction.
Consistency	Session A always sees the same data until it commits.
Performance	Improves concurrency: multiple users can safely read and write.

5.3.4 Summary

This simple example shows how MVCC enables multiple users to read data at the same time, even if another transaction is updating the same rows. Each reader gets a consistent snapshot — a powerful feature of InnoDB.

5.4 Advanced work

You can clone this [repository](#) to explore the internal details of concurrency control simulations.

5.5 Assignment (Simulate Deadlock Scenario)

 **Date on 28/6/2025**

- Read pages 789–792 from the primary textbook: Fundamentals of Database Systems.
- Please review the general lab structure requirements [here](#)
- You must use MySQL, as this lab focuses specifically on the InnoDB storage engine.

5.5.1 What to assign:

1. A brief paragraph describing the real-world situation you are simulating to create a deadlock.
2. A screenshot of the table structure (after creation) that you will use in the simulation.
3. A screenshot showing the result of the first session after execution.
4. A screenshot showing the result of the second session after execution.
5. While both Transaction 1 and Transaction 2 are still running and waiting on each other, open a third session and execute the following command: `SHOW ENGINE INNODB STATUS;`.
6. A snippet of the `SHOW ENGINE INNODB STATUS` output showing that Transaction 1 is waiting for a lock held by Transaction 2.
7. A snippet showing that Transaction 2 is waiting for a lock held by Transaction 1.
8. After the deadlock occurs and InnoDB resolves it automatically, re-run third session.
9. A snippet from the InnoDB Monitor output showing that a deadlock has been detected and resolved. to do this you should re-run step 5 again and observe the `LATEST DETECTED DEADLOCK`.
10. A brief paragraph discussing what happened and how InnoDB detects deadlock [Read this](#).

6. Database Recovery Techniques

Objective:

- To understand database recovery techniques based on logging mechanisms.
 - To learn how MySQL (specifically the InnoDB storage engine) performs crash recovery.
 - To gain a better understanding of how a DBMS handles non-catastrophic crashes (such as system failures or power outages, excluding physical hardware damage).
-

6.1 Basic Concepts Used in the Lab

Crash Recovery: Crash recovery involves restoring the database to its ^most recent consistent state^ before a system failure occurred. The recovery process ensures that the **atomicity** and **durability** properties of transactions are preserved.

Log Sequence Number (LSN) Is a unique incremental value which is assigned whenever changes occur in the InnoDB storage engine.

Write Ahead Log (WAL)/ Redo Log The Write-Ahead Log (WAL) called the redo log in **InnoDB**, is a sequential file that records all changes made to the database by ongoing transactions. These changes are written to the log before they are applied to the actual data files on disk.

Undo Log This log stores information required to roll back uncommitted transactions, It is also used to provide read consistency in multi-version concurrency control (MVCC) by preserving older versions of data for ongoing reads.

Binary Log (binlog) Is a set of files that record all changes made to the database. This includes both data modifications (such as **INSERT**, **UPDATE**, and **DELETE**) and schema changes (such as creating or altering tables).

Buffer Pool InnoDB uses a buffer pool in memory to cache frequently accessed data pages.

Dirty Pages When data is modified, the corresponding pages in the buffer pool become "dirty" (modified but not yet written to disk).

Fuzzy Checkpointing Instead of flushing all dirty pages at once, InnoDB uses **fuzzy checkpointing**—a process that incrementally flushes dirty pages to disk. This approach reduces I/O overhead, maintains system performance, and ensures recoverability.

Purge— A concept related to garbage collection, referring to the process of permanently removing records that have been marked for deletion and are no longer visible to any active transactions.

6.2 Demonstrating InnoDB Crash Recovery Through a Practical Example

6.2.1 Configuring `my.ini` for Crash Recovery Monitoring

To enable detailed crash recovery tracking, you'll need to edit the MySQL `my.ini` configuration file. This file uses an initialization (INI) format, organized into sections like `[mysqld]` and `[client]`, each containing key-value pairs.

Steps: [See this video for more details.](#)


1. Open Nodebad editor as administrator "Run as administrator".
2. Open the Configuration File
 - Go to: C:\ProgramData\MySQL\MySQL Server 8.0\
 - Switch the file type filter to All Files to display `my.ini`
 - Open the `my.ini` file
3. Edit the `[mysqld]` Section Add the following lines:

```
log_error_verbosity = 3
log_bin = mysql-bin
server-id = 1
binlog_format = ROW
```

4. Save and Close the file.

Configuration Overview

Directive	Purpose
log_error_verbosity = 3	Enables detailed error logging: Errors, Warings, and Informational Notes
log_bin = mysql-bin	Activates binary logging to track data changes
server-id = 1	Required to enable binary logging
binlog_format = ROW	Logs changes at the row level, you can see exactly which rows were inserted, updated, or deleted

 The modification on the `my.ini` should be done before you open MySQL Wokbench

6.2.2 Crash Scenario Overview


We are working with a Bank database containing a simple accounts table with two columns: `account_id` and `balance`. To explore how InnoDB handles crash recovery, we simulate three transactions with distinct timing relative to a manually enforced checkpoint:

1. **Transaction T1:** Begins and commits before the checkpoint is enforced.
2. **Transaction T2:** Begins before the checkpoint and while T1 is running, but remains uncommitted when the crash occurs.
3. **Transaction T3:** Begins after the checkpoint is enforced and commits before the crash.

```
gantt
    title Crash Recovery Timeline
    dateFormat HH:mm
    axisFormat %H:%M

    section System Events
    Checkpoint      : milestone, cp, 10:03, 0m
    Crash           : milestone, crash, 10:07, 0m

    section Transactions
    T1 (Starts first, ends before Checkpoint) :done, t1, 10:00, 2m
    T2 (Starts during T1, uncommitted):active, t2, 10:01, 7m
    T3 (Starts after Checkpoint, commits) :done, t3, 10:04, 2m
```

 While InnoDB performs automatic checkpoints whenever the volume of dirty pages exceeds a certain threshold, in this experiment, we trigger a manual checkpoint at a specific point in time. It's important to understand that during the execution of these transactions, several background checkpoints may still occur, but our focus is on the manually enforced checkpoint for clarity and observation purposes.

6.2.3 Transaction Script for Simulation

1. Create Bank Database

```
CREATE DATABASE IF NOT EXISTS bank;
```

2. Create Accounts Table

```
-- Create the accounts table
USE bank;

CREATE TABLE accounts (
    account_id INT PRIMARY KEY,
    balance DECIMAL(10,2)
);
```

3. Transaction 1

```
-- Session T1: Credit accounts with low balance

USE bank;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;
SET SQL_SAFE_UPDATES = 0;
UPDATE accounts
    SET balance = balance + 500
WHERE balance <= 2000;

DO SLEEP(20); -- Pause the transaction to allow T2 to start before T1 commits

SHOW ENGINE INNODB STATUS; -- Display InnoDB status at time T2 for monitoring purposes
SHOW MASTER STATUS; -- Record current binary log file and position for monitoring purposes

COMMIT; -- T1 completes and is committed before the checkpoint
```

4. Transaction 2

```
-- Session T3: Update balance with account id after checkpoint
-- T3 starts after the checkpoint is triggered, while T2 is still in progress
-- T3 completes and is committed before the crash occurs

USE bank;

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;
SHOW ENGINE INNODB STATUS;
SET SQL_SAFE_UPDATES = 0;
UPDATE accounts
    SET balance = balance + 1000
WHERE account_id = 1;

SHOW ENGINE INNODB STATUS; -- Display InnoDB status at time T2 for monitoring purposes
SHOW MASTER STATUS; -- Record current binary log file and position for monitoring purposes

-- COMMIT will not occur; crash happens before this
```

5. Enforce checkpoint

```
-- Trigger checkpoint: flush dirty pages and force redo log to disk
FLUSH TABLES;
FLUSH LOGS;
```

6. Transaction 3

```
-- Session T2: Apply tax deduction based on the account id
-- T2 starts while T1 is still running, but T2 remains uncommitted until the crash
USE bank;
SHOW ENGINE INNODB STATUS;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;
SET SQL_SAFE_UPDATES = 0;
UPDATE accounts
    SET balance = CAST((
        CASE
            WHEN account_id BETWEEN 2 AND 1000 THEN balance - balance * 0.20
            WHEN account_id BETWEEN 1001 AND 2000 THEN balance - balance * 0.10
            ELSE balance
        END
    ) AS DECIMAL(10,2))
WHERE account_id BETWEEN 2 AND 2000;
```

```
SHOW ENGINE INNODB STATUS; -- Display InnoDB status at time T2 for monitoring purposes
SHOW MASTER STATUS; -- Record current binary log file and position for monitoring purposes
COMMIT;
```

6.2.4 Simulating the Crash Event

1. Create bank Database and accounts table. [See this video for Steps 1-2.](#)
2. Import the accounts data:
 - Download the CSV file: [accounts](#)
 - In MySQL Workbench, right-click on the Tables section under the bank database.
 - Select Table Data Import Wizard
 - Browse to the accounts file and follow the wizard steps.
3. Open four separate sessions. In each one, paste the transaction and checkpoint scripts provided earlier. [See this video](#)
4. Run Transaction 1. [See this video for Steps 4-11.](#)
5. While Transaction 1 is waiting (use `DO SLEEP(20)`), run Transaction 2.
6. Once Transaction 1 finishes, observe that Transaction 2 is still waiting. This is expected due to the use of REPEATABLE READ isolation and because T1 locks the entire table. Then, run the checkpoint session.
7. Open PowerShell as Administrator, then run the following command to find the mysqld process ID. Look for the instance with the largest memory usage: [See this video for Steps 7-8.](#)

```
Get-Process "mysqld"
```

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Windows\system32> Get-Process "mysqld"

Handles NPM(K) PM(K) WS(K) CPU(s) Id SI ProcessName
-----
128 12 33864 8896 0.02 14524 0 mysqld
511 386 647620 66848 15.13 18964 0 mysqld

PS C:\Windows\system32>
```

8. Prepare the stop command. Replace 18964 with your actual process ID. Do not run it yet:

```
Stop-Process -Id 18964 -Force
```

9. Run Transaction 3.
10. Execute the stop command (step 8) as quickly as possible, ideally within 30 seconds. This simulates the crash.
11. Restart MySQL80 to simulate recovery after the crash.

```
Start-Service MySQL80
```

You can also use Task Manager to stop the mysqld service and restart MySQL80.

6.2.5 Step-by-Step InnoDB Crash Recovery Process [See this video for more explanation.](#)

In this section, we'll demonstrate the steps InnoDB takes to detect a crash and perform recovery, using insights drawn from our example log file. You can open the error log from: `C:\ProgramData\MySQL\MySQL Server 8.0\Data`

Then scroll to the timestamp corresponding to the crash. It should resemble this entry: 2025-07-03T19:00:44.462348Z 0 [System] [MY-010116] [Server] C:\Program Files\MySQL\MySQL Server 8.0\bin\mysqld.exe (mysqld 8.0.42) starting as process 16816

1. **Tablespace Discovery** This is the process that InnoDB uses to identify tablespaces that require redo log application. In our example:

```
The latest found checkpoint is at lsn = 155289321 in redo log file .\#innodb_redo\#ib_redo47.
The log sequence number 152428710 in the system tablespace does not match the log sequence number 155289321 in the redo log files!
Database was not shutdown normally!
Starting crash recovery.
```

These messages indicate a mismatch between the last checkpoint in the system tablespace (LSN 152428710) and the redo log (LSN 155289321)—a difference of 2,860,611. This gap means not all changes had been flushed to disk at shutdown. InnoDB recognizes this as a crash and begins recovery.

2. **Redo Log Application** Now, InnoDB replays changes from the redo logs to bring the database back to a consistent state

```
Starting to parse redo log at lsn = 155289106, whereas checkpoint_lsn = 155289321 and start_lsn = 155289088
Doing recovery: scanned up to log sequence number 155289651
Log background threads are being started...
Applying a batch of 3 redo log records ...
100%
Apply batch completed!
```

InnoDB starts parsing slightly before the checkpoint—at LSN 155289106—to ensure no redo log entries are missed. It then scans through to LSN 155289651 and re-applies three redo records. In our example, it replays transaction 3 change.

3. **Rollback of Incomplete Transactions** Next, InnoDB undoes the effects of uncommitted transactions using the undo log


```
Using undo tablespace '.\undo_001'.
Using undo tablespace '.\undo_002'.
Opened 2 existing undo tablespaces.
GTID recovery trx_no: 447747
```

InnoDB accesses the undo tablespaces to roll back any incomplete transactions. In our example, it undoes Transaction 2's updates.

6.2.6 Conclusion

In this simulation, we used the error log, binary files, and multiple executions of `SHOW ENGINE INNODB STATUS`; to observe the transaction IDs, log sequence numbers, and the last checkpoint that occurred during a running transaction. This allowed us to detect the crash event and analyze how the DBMS—specifically InnoDB—handles crash recovery.

6.3 Assignment (Simulate Crash Recovery)

 **Due Date on 19/7/2025**

- Please review the [general lab structure requirements](#) here
- You must use **MySQL**, as this lab focuses specifically on the InnoDB storage engine.

6.3.1 What to assign:

1. Use the same database, table, and table data provided in the lab setup.
2. Simulate the following crash recovery scenario, take a screenshot of each transaction, and identify which one is Transaction 1, Transaction 2, and Transaction 3:
 - a. **Transaction T1**: Begins before T2 and before the checkpoint, but remains uncommitted when the crash occurs.
 - b. **Transaction T2**: Begins after T1 and commits before the checkpoint is enforced.
 - c. **Transaction T3**: Begins after the checkpoint and commits before the crash. This should be a heavy transaction, so that the redo phase includes applying changes to modified pages.

```
gantt
title Crash Recovery Timeline
dateFormat HH:mm
axisFormat %H:%M
```

```

section System Events
Checkpoint      : milestone, cp, 10:04, 0m
Crash           : milestone, crash, 10:07, 0m

section Transactions
T1 (Starts first, uncommitted):active, t2, 10:00, 8m
T2 (Starts during T1, commits) :done, t1, 10:01, 2m
T3 (Starts after Checkpoint, commits) :done, t3, 10:05, 2m

```

3. Identify:

- The LSN of the last checkpoint created by InnoDB.
- The latest LSN on the data file.
- The latest LSN in the redo log.

4. Determine:

- When the redo phase begins.
- How many redo batches are executed.

5. Provide evidence that transaction T3 was replayed during the redo phase. Observe the checkpoint activity based on InnoDB status during T3's execution.

6. Identify which transactions are undone during recovery. Support your answer using the transaction IDs observed in the InnoDB status while the transactions are active.

7. Based on your study of ARIES, do you think InnoDB crash recovery is simpler or more complex? Explain your reasoning.

7. Database Security Measures

7.1 Objective:

- Understand security principles, including authentication, authorization, and access controls.
- Create different user roles and assign permissions to database objects.
- Set up role-based access (RBAC) to control who can access specific database objects.
- Apply basic encryption to protect sensitive data stored in the database.
- Simulate SQL injection attacks and explain prevention methods such as prepared statements.

7.2 Lab Content:

- Database Security Principles
 - a. Authentication
 - b. Authorization
 - Practical Implementation of Database Security Concepts
 - a. Create Database and Tables
 - b. Create User Roles and Assign Permissions
 - c. Create Roles and Assign to User
 - d. Column-Level Encryption
 - e. Simulate SQL Injection & Prevention
 - Assignment Instructions
-

7.3 Database Security Principles

7.3.1 1. Authentication

Definition:

Verifying a user's identity before granting database access.

Examples:

- Username/password login validation
- Multi-factor authentication (MFA)

Purpose:

Ensures only verified users can interact with the database system.

7.3.2 2. Authorization

Definition:

Determines what actions authenticated users are allowed to perform, such as read, write, or modify.

Examples:

- Granting `SELECT` privileges on specific tables - Restricting `DROP / ALTER` commands to DBAs

Purpose:

Prevents unauthorized data access or modifications.

Often, a user must log in to a system using some form of authentication. After authentication, access control mechanisms determine which operations the user can or cannot perform by comparing their identity to an access control list (ACL). While authorization policies define what an individual or group is allowed to access, access controls are the mechanisms used to enforce these policies."

7.4 Practical Implementation of Database Security Concepts

7.5 1. Create Database and Tables

- Create a DB (university_db)
- Create Tables: students , courses
- Insert data

The screenshot displays a SQL editor window titled "SQL File 4*" with the following SQL code:

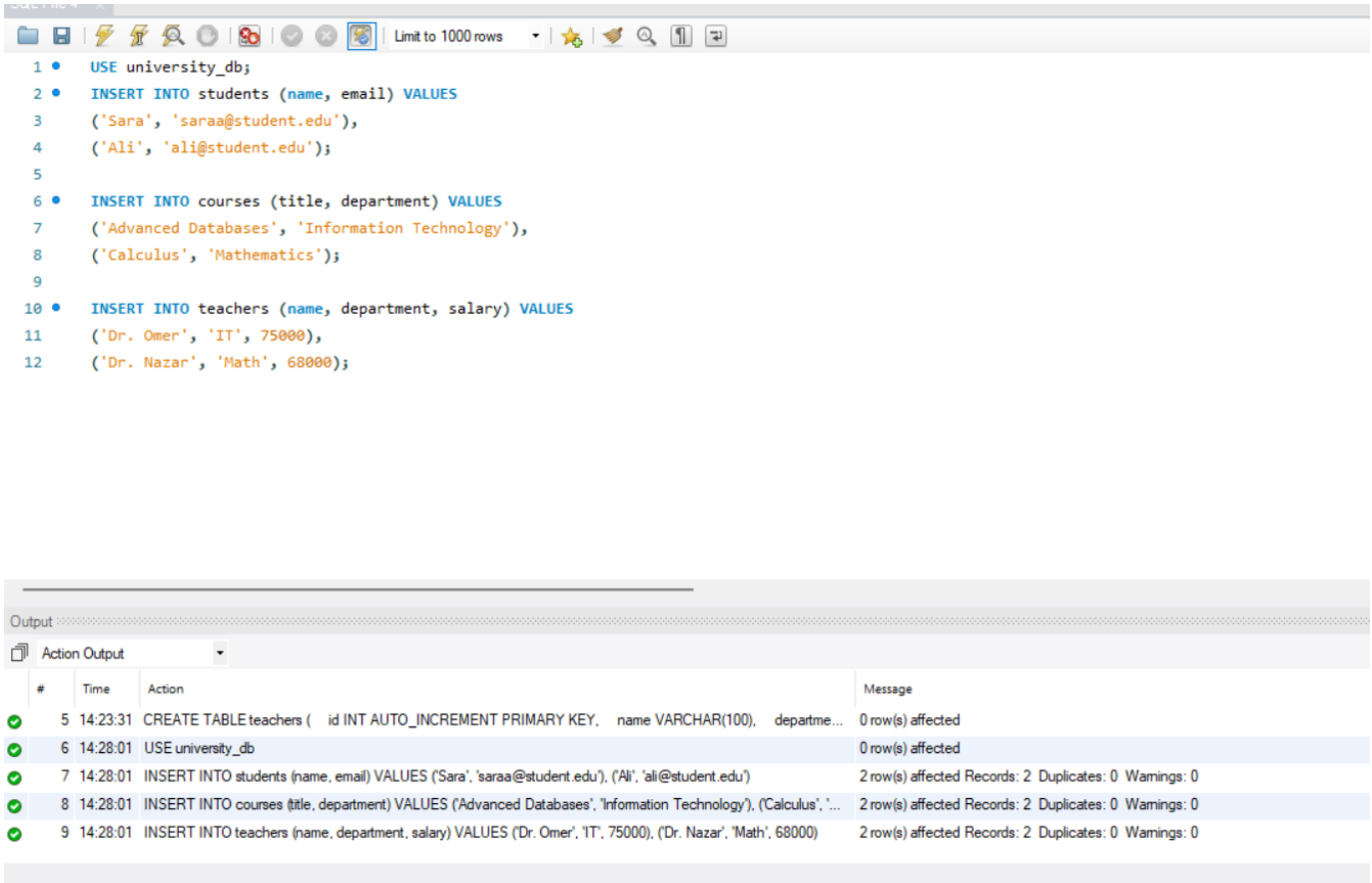
```

1  CREATE DATABASE university_db;
2  USE university_db;
3  CREATE TABLE students (
4      id INT AUTO_INCREMENT PRIMARY KEY,
5      name VARCHAR(100),
6      email VARCHAR(100),
7      ssn VARBINARY(255) -- For encryption
8  );
9  CREATE TABLE courses (
10     id INT AUTO_INCREMENT PRIMARY KEY,
11     title VARCHAR(100),
12     department VARCHAR(100)
13 );
14 CREATE TABLE teachers (
15     id INT AUTO_INCREMENT PRIMARY KEY,
16     name VARCHAR(100),
17     department VARCHAR(100),
18     salary DECIMAL(10,2)
19 );

```

Below the code editor is an "Output" window showing the execution results:

#	Time	Action	Message
1	14:23:30	CREATE DATABASE university_db	1 row(s) affected
2	14:23:30	USE university_db	0 row(s) affected
3	14:23:30	CREATE TABLE students (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(100), email VARC...	0 row(s) affected
4	14:23:30	CREATE TABLE courses (id INT AUTO_INCREMENT PRIMARY KEY, title VARCHAR(100), department VA...	0 row(s) affected
5	14:23:31	CREATE TABLE teachers (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(100), department ...	0 row(s) affected



```

1 • USE university_db;
2 • INSERT INTO students (name, email) VALUES
3 ('Sara', 'saraa@student.edu'),
4 ('Ali', 'ali@student.edu');
5
6 • INSERT INTO courses (title, department) VALUES
7 ('Advanced Databases', 'Information Technology'),
8 ('Calculus', 'Mathematics');
9
10 • INSERT INTO teachers (name, department, salary) VALUES
11 ('Dr. Omer', 'IT', 75000),
12 ('Dr. Nazar', 'Math', 68000);

```

Output

Action Output

#	Time	Action	Message
✓ 5	14:23:31	CREATE TABLE teachers (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(100), departme...	0 row(s) affected
✓ 6	14:28:01	USE university_db	0 row(s) affected
✓ 7	14:28:01	INSERT INTO students (name, email) VALUES ('Sara', 'saraa@student.edu'), ('Ali', 'ali@student.edu')	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✓ 8	14:28:01	INSERT INTO courses (title, department) VALUES ('Advanced Databases', 'Information Technology'), ('Calculus', '...	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✓ 9	14:28:01	INSERT INTO teachers (name, department, salary) VALUES ('Dr. Omer', 'IT', 75000), ('Dr. Nazar', 'Math', 68000)	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0

7.6 2. Create User Roles and Assign Permissions

Create user roles with specific permissions to control access based on each user's responsibilities. For example, allowing actions like data viewing `Select` while preventing modifications `Update` and `Insert`. This strengthens security, minimizes errors, and improves operational control.

Here we will create a read-only user named `data_viewer` and grant access to the tables `students` and `courses`.

```

-- Create a new user
CREATE USER 'data_viewer'@'localhost' IDENTIFIED BY 'viewerpass123';

-- Grant SELECT permission directly on specific tables
GRANT SELECT ON university_db.students TO 'analyst'@'localhost';
GRANT SELECT ON university_db.courses TO 'analyst'@'localhost';

```

7.7 3. Create Roles and Assign to User

Instead of assigning permissions to each user individually, we create a role (e.g., `readonly`) with specific privileges and assign it to multiple users. This is easier to manage, especially for large teams, because roles group privileges into reusable sets.

Here, we'll create a role named `readonly` and assign it to the `data_viewer` user we created earlier.

```

-- Create a read-only role
CREATE ROLE 'readonly';

-- Grant SELECT permissions to the role
GRANT SELECT ON university_db.* TO 'readonly';

-- Assign role to user
GRANT 'readonly' TO 'data_viewer'@'localhost';

```

Expected Output:

The user can only view data; they cannot insert, update, or delete.

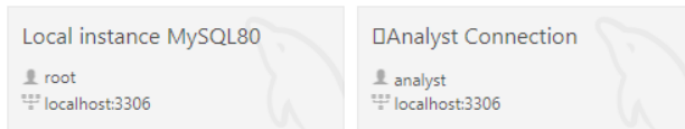
7.7.1 Test Instructions

1. Log in as `data_viewer`.
2. Click the (+) icon on the welcome screen to **Add a New Connection**.

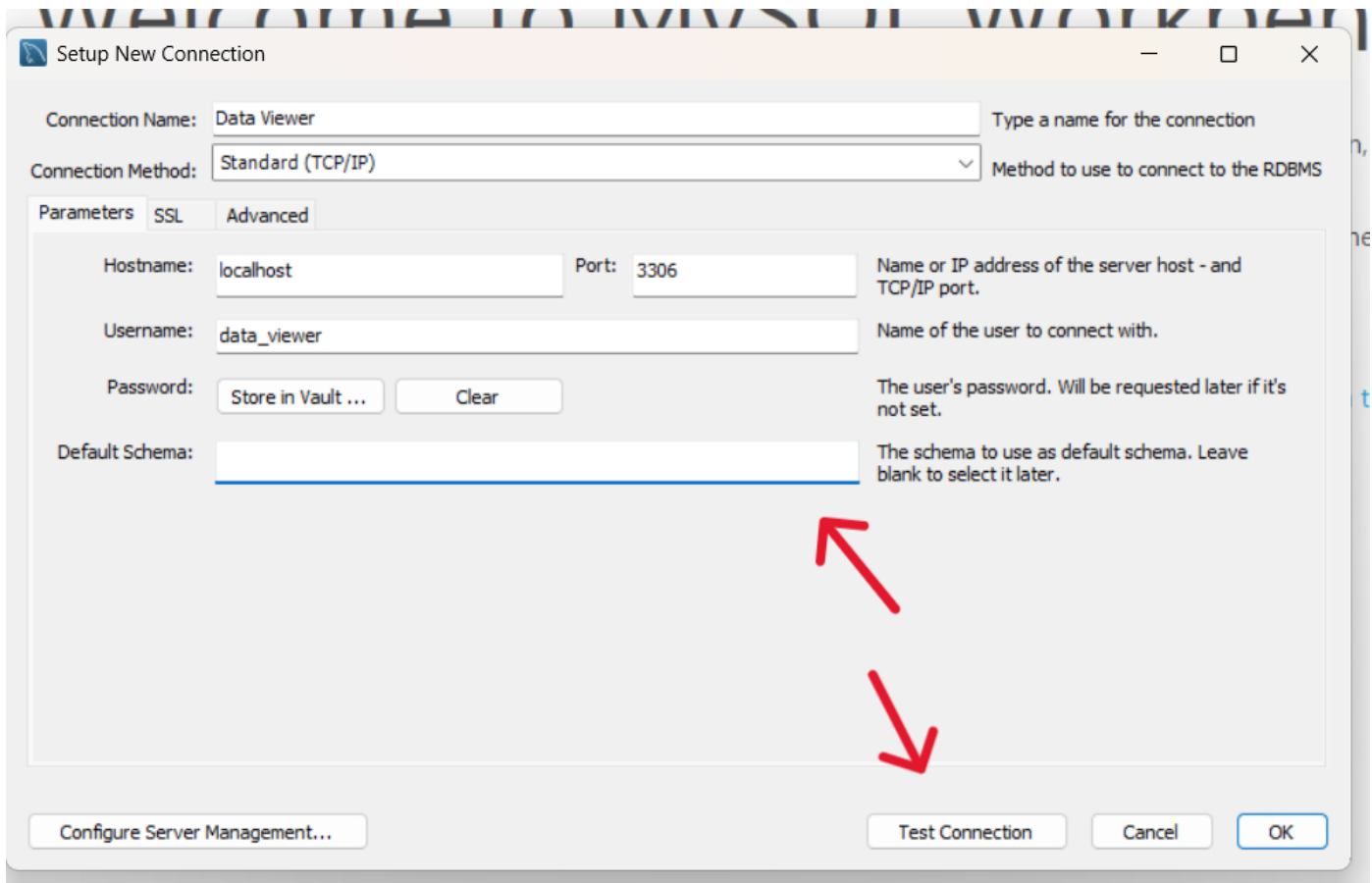
[Browse Documentation >](#)

[Read the Blog >](#)

MySQL Connections

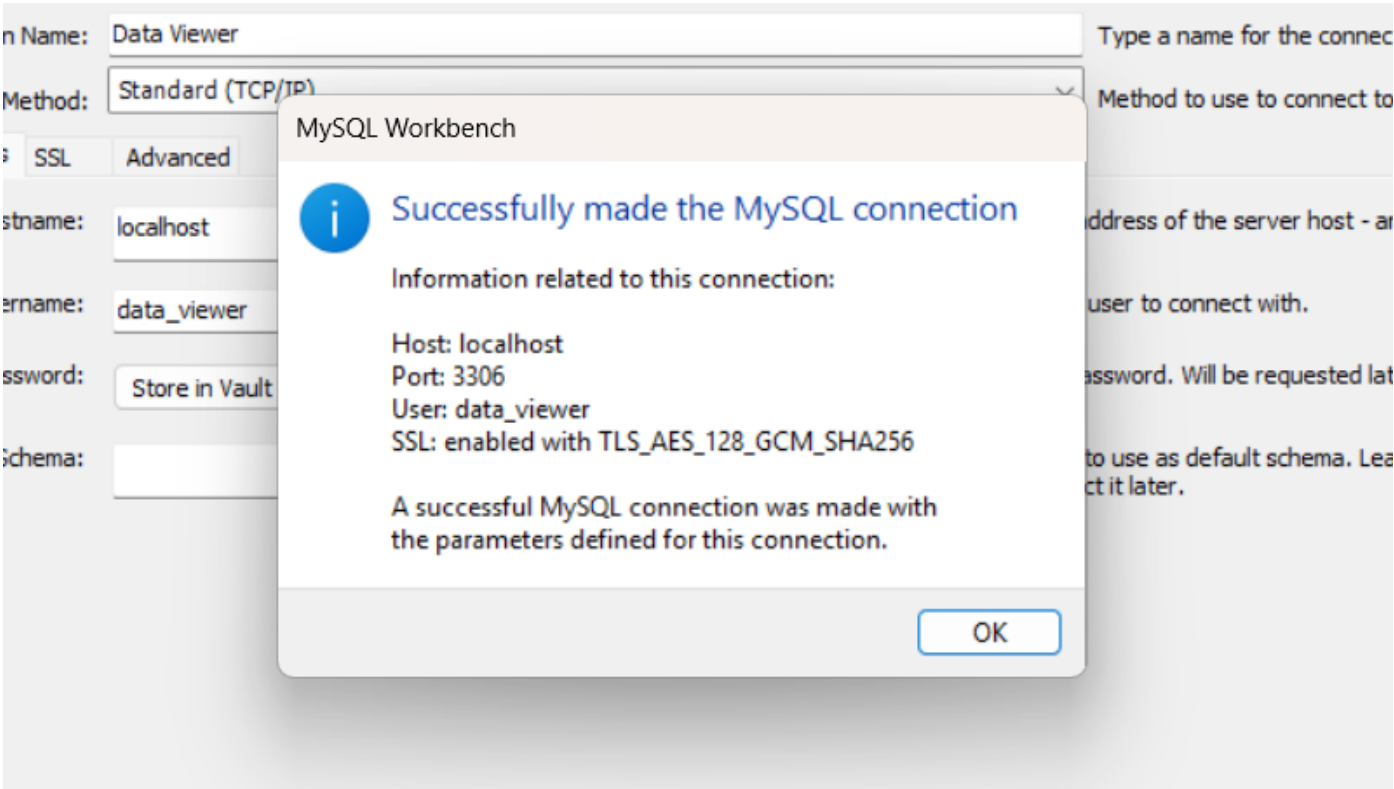


3. Fill out the connection details and click **Test Connection**.



The screenshot shows the 'Setup New Connection' dialog box. The 'Connection Name' is 'Data Viewer'. The 'Connection Method' is 'Standard (TCP/IP)'. The 'Parameters' tab is selected, showing 'Hostname' as 'localhost', 'Port' as '3306', 'Username' as 'data_viewer', and 'Password' as 'Store in Vault ...'. The 'Default Schema' is empty. The 'Test Connection' button is highlighted with a red arrow. Another red arrow points to the 'Test Connection' button from below.

4. Once the successful connection message appears, click **OK** to proceed.



5. Run the Following Code in the data viewer editor.

```
-- Activate the readonly role
SET ROLE 'readonly';

-- Select the database
USE university_db;

-- This should work (SELECT permission granted)
SELECT * FROM students;

-- This should fail (no INSERT permission for the students table)
INSERT INTO students (name, email) VALUES ('Test User', 'test@example.com');

-- This should fail (no UPDATE permission for the teacher table)
UPDATE teachers SET salary = 90000 WHERE id = 1;
```

7.7.2 Direct Grant vs Role-Based Access Control (RBAC)

Direct Grant refers to assigning permissions directly to individual users on a specific database object (`GRANT SELECT ON students TO 'data_viewer'`). While this approach works well for small systems, it becomes difficult to manage as the number of users increases, often leading to inconsistent or redundant privilege assignments. In contrast, Role-Based Access Control (RBAC) provides a more scalable and organized method by assigning permissions to roles rather than to users directly. Users are then granted roles based on their responsibilities. (e.g., `GRANT SELECT ON students TO 'readonly'; GRANT 'readonly' TO 'data_viewer';`) This makes it easier to maintain consistent access policies, simplifies permission updates, and aligns with the principle of least privilege.

7.8 4. Column-Level Encryption

Column-level encryption is a database security technique used to protect sensitive data stored in specific columns of a table allows you to encrypt sensitive data, such as (Social Security Numbers, passwords, or medical records) at the column level within a table. In MySQL, Functions like `AES_ENCRYPT()` and `AES_DECRYPT()`

are commonly used to perform encryption and decryption, ensuring that even if someone gains access to the database, the data remains unreadable without the decryption key

- `AES_ENCRYPT(data, key)` → encrypts data
- `AES_DECRYPT(data, key)` → decrypts encrypted data

Encrypted data is stored as binary (e.g., `VARBINARY`), and both encryption & decryption must use the same key.

7.8.1 Example: Encrypting Student SSNs in the previously created students table

1. Insert encrypted data into the students table

```
INSERT INTO students (name, email, ssn)
VALUES (
  'Mazen',
  'mazen@student.edu',
  AES_ENCRYPT('123-45-6789', 'secret_key')
);
```

The `secret_key` is the encryption key used for encrypting sensitive data. You must use the **same key** to decrypt the data later.

2. Select and Decrypt Data

The following SQL query selects the `name` and `email` columns and decrypts the `ssn` (social security number) column using the AES encryption key `secret_key`:

```
SELECT name, email, AES_DECRYPT(ssn, 'secret_key') AS decrypted_ssn
FROM students;
```

Note: If the wrong key is used, `decrypted_ssn` will return `NULL`.

7.9 5. Simulate SQL Injection & Prevention

SQL injection is a common attack where an attacker inserts malicious SQL code into input fields to manipulate queries. This section demonstrates an unsafe query and how to prevent it using the prepared statements technique.

7.9.1 Example (Using the admin “Root” editor)

7.9.2 Simulate SQL Injection (Unsafe Query)

This example will demonstrate how malicious input like `' OR '1'='1'` can trick a query into returning all rows, even if the condition should match only one row.

1. Simulate unsafe user input

```
SET @user_input = "' OR '1'='1'";
```

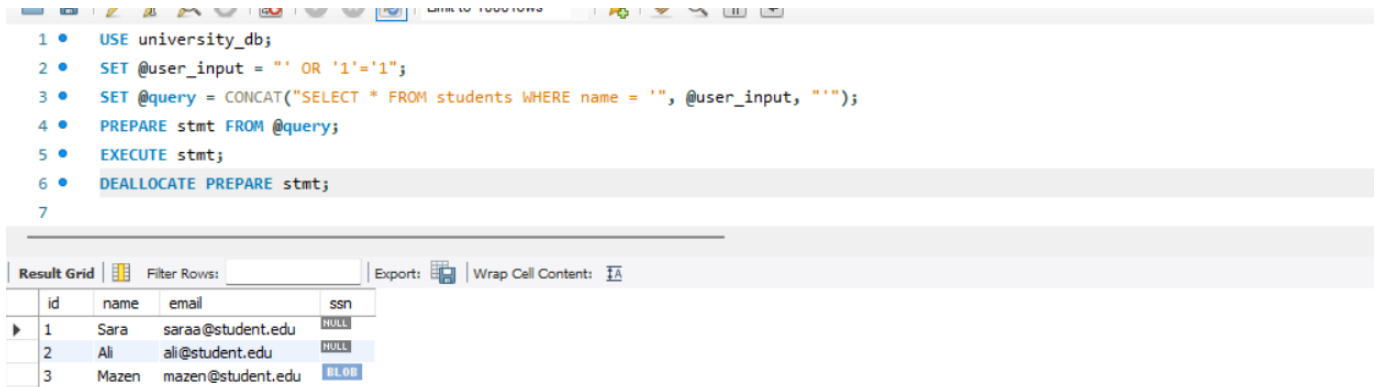
2. Dynamically build an unsafe SQL query

```
SET @query = CONCAT("SELECT * FROM students WHERE name = '", @user_input, "'");
```

3. Prepare and execute the query

```
PREPARE stmt FROM @query;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

Effect: Returns all users due to the injected



The screenshot shows a SQL IDE with a query editor and a result grid. The query editor contains the following SQL code:

```

1 • USE university_db;
2 • SET @user_input = '' OR '1'='1';
3 • SET @query = CONCAT("SELECT * FROM students WHERE name = '", @user_input, "'");
4 • PREPARE stmt FROM @query;
5 • EXECUTE stmt;
6 • DEALLOCATE PREPARE stmt;
7

```

The result grid shows the following data:

	id	name	email	ssn
1	Sara	saraa@student.edu	NULL	
2	Ali	ali@student.edu	NULL	
3	Mazen	mazen@student.edu	BLOB	

7.9.3 Prevent SQL Injection- Safe Approach (Prepared Statements)

Prepared statements separate SQL code from user input. Instead of embedding user input directly into a SQL query—which can be exploited by attackers—prepared statements use placeholders (e.g., `?`) for input values. These are known as parameters. An SQL statement template is created and sent to the database with these placeholders, while the actual data is supplied later. For example: `INSERT INTO MyGuests VALUES (?, ?, ?)`. In this query, the values are not directly inserted into the SQL string; instead, they are safely passed and bound at execution time. This ensures that user input is treated strictly as data, not executable code.

The process involves two key steps: preparing the statement and then executing it, and here are the steps to follow.

Step 1: Prepare a safe query with a placeholder (Tells MySQL to prepare an SQL query in advance with a placeholder `?` for user input, which guarantees the query structure is fixed and cannot be changed by user input.)

```
PREPARE stmt FROM 'SELECT * FROM students WHERE name = ?';
```

Step 2: Define the user input

```
SET @name_input = "Sara";
```

Step 3: Execute the statement safely using bound input (Runs the query and plugs in the value stored in `@name_input`, which blocks any attempt to inject SQL.)

```
EXECUTE stmt USING @name_input;
DEALLOCATE PREPARE stmt;
```

Expected Result: Returns only user 'Sara' without risk of injection.

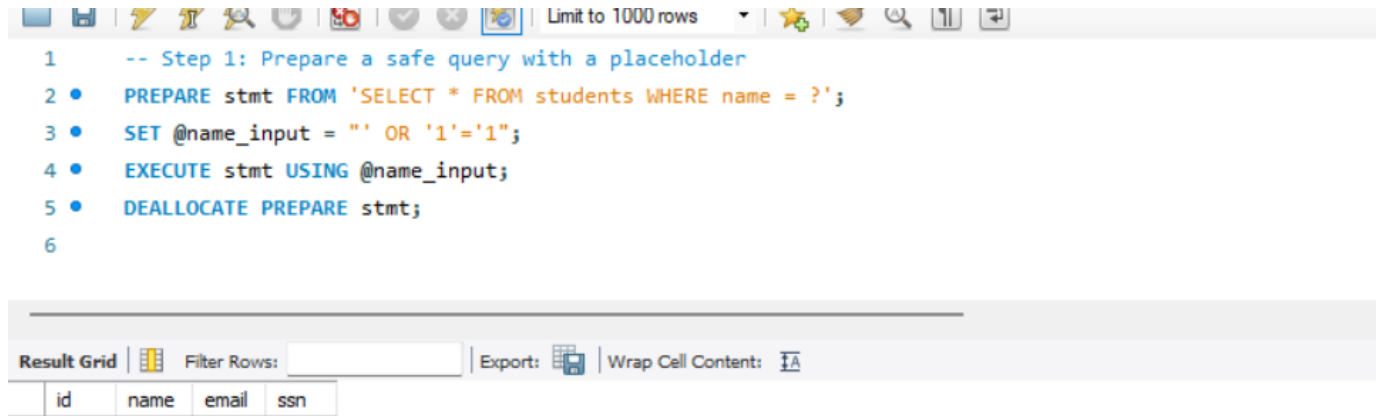
Try modifying the input to include something malicious, like `' OR '1'='1'`. You will notice that MySQL treats it as a `string`, not as part of the `SQL logic`.

```

PREPARE stmt FROM 'SELECT * FROM students WHERE name = ?';
SET @name_input = '' OR '1'='1';
EXECUTE stmt USING @name_input;
EXECUTE stmt USING @name_input;

```

Expected Result: No row will return.



The screenshot shows a SQL IDE interface. The top toolbar includes icons for file operations, execution, and a dropdown menu set to "Limit to 1000 rows". The query window contains the following SQL code:

```

1  -- Step 1: Prepare a safe query with a placeholder
2  • PREPARE stmt FROM 'SELECT * FROM students WHERE name = ?';
3  • SET @name_input = '' OR '1'='1';
4  • EXECUTE stmt USING @name_input;
5  • DEALLOCATE PREPARE stmt;
6

```

Below the query window is the "Result Grid" section. It includes a "Filter Rows:" input field, an "Export:" button, and a "Wrap Cell Content:" toggle. The result grid itself is empty, with column headers "id", "name", "email", and "ssn" visible.

Result 1 x

8. Assignment 4

9. What to assign:

9.0.1 1. Create users and privilege assignments:

- One user with **read-only access** to the entire database. [Direct Grant]
- One user with **read, insert and Update** permissions, but **no delete**. [Role-Based Access Control: A data-entry role]
- One user with access to **only specific columns** (e.g., can view names but **not emails or IDs**). [Direct Grant]
- **Screenshot Requirements:**
- User and Role creation commands and privilege assignment.
- Evidence of successful and failed attempts to access or modify tables or columns.

9.0.2 2. Encrypt a sensitive column (e.g., national ID):

Show how:

- Data looks when encrypted.
- Data can be safely decrypted.

Screenshot Requirements:

- Encrypted data insertion.
- Decryption query and its output.

- View comparing encrypted binary data vs readable decrypted results.

9.0.3 3. SQL Injection Methods:

Report:

- Brief explanation (1-2 paragraphs) of what SQL injection is and why preventing it is important.
 - Common prevention methods (Prepared statements, Input sanitization, etc.)
-

9.1 What to Submit

A PDF document including:

- Screenshots for each task, with a caption under each screenshot explaining what it shows.
- A (1-2 paragraphs) report on SQL injection (as described above).

 Due Date on 29/7/2025

- Please review the [general lab structure requirements](#) [here](#)