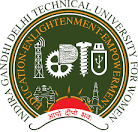
DATABASE MANAGEMENT SYSTEM LAB

(BIT-201)



Submitted to: -

Ms. Vibha Ma’am

Submitted by: -

Aditi Gulati

02101012021(Cse-1)

Indira Gandhi Delhi Technical University for Women

# CONTENT

**CHAPTERNO. PAGENO.**

1. **INTRODUCTION 01**
   1. **OBJECTIVES**
   2. **LIMITATIONS**

## STUDY OF EXISTING SYSTEM 02

* 1. **A CASE STUDY**
  2. **PROPOSED SYSTEM**

## DATABASE DESIGN 03-20

* 1. **SOFTWARE REQUIREMENT SPECIFICATION**
     1. **COLLECTION OF REQUIREMENTS**
     2. **SOFTWARE AND HARDWARE REQUIREMENTS**
  2. **CONCEPTUAL DESIGN**
     1. **ER DIAGRAM**

**3.2.3SCHEMA DIAGRAM**

* 1. **IMPLEMENTATION**
     1. **FRONTEND**
     2. **BACKEND**
     3. **TRIGGER**
     4. **STORED PROCEDURE**

## USER INTERFACES 21-31

**4.1 SCREENSHOTS**

## CONCLUSIONS FUTURE ENHANCEMENTS AND REFERENCES

CHAPTER-1

# INTRODUCTION

## OBJECTIVES:

* + - The main objective of the project is to design and develop a user friendly-system
    - Easy to use and an efficient computerized system.
    - To develop an accurate and flexible system, it will eliminate data redundancy.
    - To study the functioning of Students management System.
    - To make a software fast in processing, with good user interface.
    - To make software with good user interface so that user can change it and it should be used for a long time without error and maintenance.
    - To provide synchronized and centralized farmer and seller database.
    - Computerization can be helpful as a means of saving time and money.
    - To provide better Graphical User Interface (GUI).
    - Less chances of information leakage.
    - Provides Security to the data by using login and password method.
    - To provide immediate storage and retrieval of data and information.
    - Improving arrangements for students coordination.
    - Reducing paperwork.

## LIMITATIONS:

* + - Time consumption in data entry as the records are to be manually maintained faculties a lot of time.
    - Lot of paper work is involved as the records are maintained in the files and registers.
    - Storage Requires as files and registers are used the storage space requirement is increased.
    - Less Reliable use of papers for storing valuable data information is not at all reliable.
    - Aadhar linkage with the official aadhar database has not been done.

# CHAPTER-2

* 1. **CASE STUDY**

# STUDY OF EXISTING SYSTEM

The success of any organization such as School of Public Health, University of Ghana hinges on its ability to acquire accurate and timely data about its operations, to manage this data effectively, and to use it to analyze and guide its activities. Integrated student database system offer users (Student, Registrar, HOD) with a unified view of data from multiple sources. To provide a single consistent result for every object represented in these data sources, data fusion is concerned with resolving data inconsistency present in the heterogeneous sources of data. The main objective of this project is to build a rigid and robust integrated student database system that will track and store records of students. This easy-to-use, integrated database application is geared towards reducing time spent on administrative tasks. The system is intended to accept process and generate report accurately and any user can access the system at any point in time provided internet facility is available. The system is also intended to provide better services to users, provide meaningful, consistent, and timely data and information and finally promotes efficiency by converting paper processes to electronic form. The system was developed using technologies such as, HTML, CSS ,JS and

MySQL. PYTHON- FLASK, HTML and CSS are used to build the user interface and database was built using MySQL. The system is free of errors and very efficient and less time consuming due to the care taken to develop it. All the phases of software development cycle are employed and it is worthwhile to state that the system is very robust. Provision is made for future development in the system.

## PROPOSED SYSTEM

While there has been no consensus on the definition of Students Management in the literature, they have proposed that researchers adopt the below definition to allow for the coherent development of theory in the colleges. In order to have a successful students management, we need to make many decisions related to the flow of marks, attendance, and data. Each records should be added in a way to increase the scalability. Student management is more complex in colleges and other universities because of the impact on people’s number requiring adequate and accurate information of students need.

# CHAPTER 3

**3. DATABASE DESIGN**

## SOFTWARE REQUIREMENTS SPECIFICATION

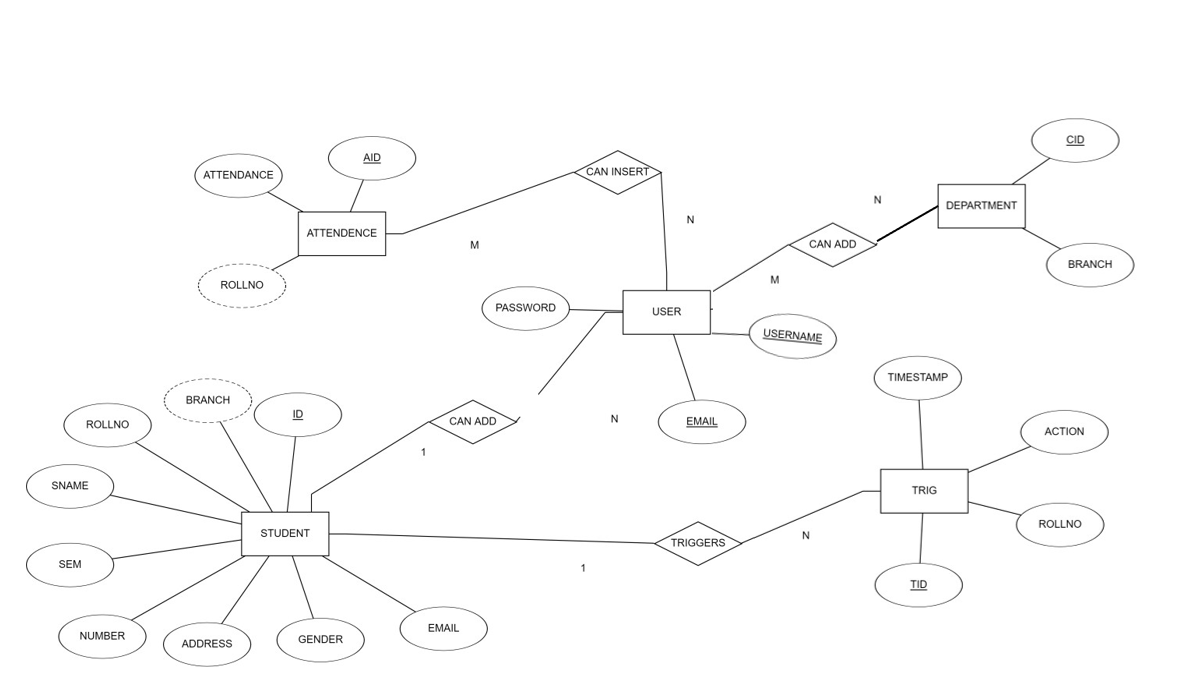
* + 1. **SOFTWARE REQUIREMENTS:** Frontend- HTML, CSS, Java Script, Bootstrap Backend-Python flask (Python 3.7), SQLAlchemy,
       - Operating System: Windows 10
       - Google Chrome/Internet Explorer
       - XAMPP (Version-3.7)
       - Python main editor (user interface): PyCharm Community
       - workspace editor: Sublime text 3

## HARDWARE REQUIREMENTS:

* + - * Computer with a 1.1 GHz or faster processor
      * Minimum 2GB of RAM or more
      * 2.5 GB of available hard-disk space
      * 5400 RPM hard drive
      * 1366 × 768 or higher-resolution display
      * DVD-ROM drive

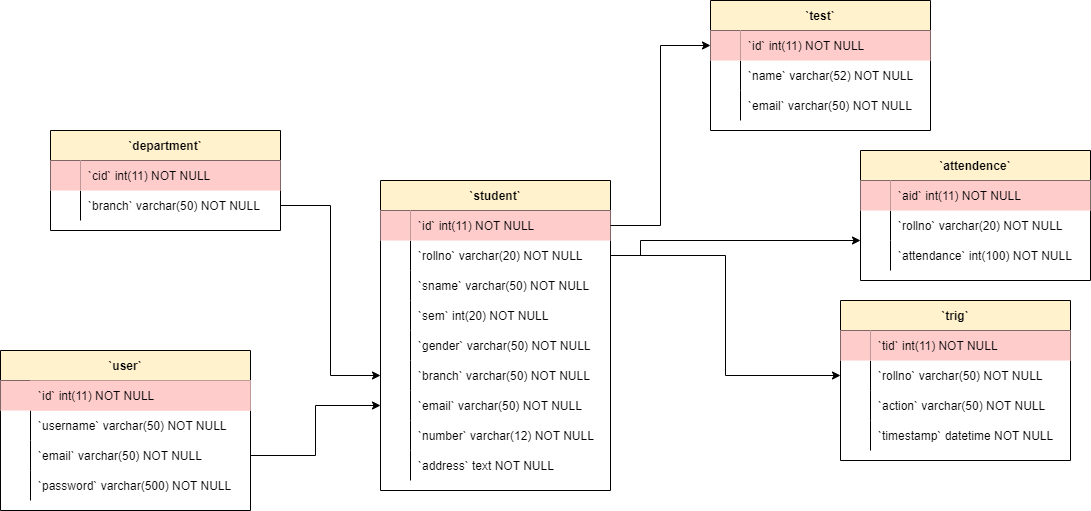
# CONCEPTUAL DESIGN:

## E-R DIAGRAM:

****

STUDENT MANAGEMENT SYSTEM

* + 1. **SCHEMA DIAGRAM:**

****

# IMPLEMENTATION:

An "implementation" of Python should be taken to mean a program or environment which provides support for the execution of programs written in the Python language, as represented by the [CPython](https://wiki.python.org/moin/CPython) reference implementation.

There have been and are several distinct software packages providing of what we all recognize as Python, although some of those are more like distributions or variants of some existing implementation than a completely new implementation of the language.

## Back End (MySQL) Database:

A Database Management System (DBMS) is computer software designed for the purpose of managing databases, a large set of structured data, and run operations on the data requested by numerous users. Typical examples of DBMSs include Oracle, DB2, Microsoft Access, Microsoft SQL Server, Firebird, PostgreSQL, MySQL, SQLite, FileMaker and Sybase Adaptive Server Enterprise. DBMSs are typically used by Database administrators in the creation of Database systems. Typical examples of DBMS use include accounting, human resources and customer support systems. Originally found only in large companies with the computer hardware needed to support large data sets, DBMSs have more recently emerged as a fairly standard part of any company back office.

A DBMS is a complex set of software programs that controls the organization, storage, management, and retrieval of data in a database. A DBMS includes:

* A modeling language to define the schema of each database hosted in the DBMS, according to the DBMS data model.
* The dominant model in use today is the ad hoc one embedded in SQL, despite the objections of purists who believe this model is a corruption of the relational model, since it violates several of its fundamental principles for the sake of practicality and performance. Many DBMSs also support the Open Database Connectivity API that supports a standard way for programmers to access the DBMS.
* Data structures (fields, records, files and objects) optimized to deal with very large amounts of data stored on a permanent data storage device (which implies relatively slow access compared to volatile main memory). A database query language and report

writer to allow users to interactively interrogate the database, analyze its data and update it according to the users privileges on data.

* + Data security prevents unauthorized users from viewing or updating the database. Using passwords, users are allowed access to the entire database or subsets of it called sub schemas. For example, an employee database can contain all the data about an individual employee, but one group of users may be authorized to view only payroll data, while others are allowed access to only work history and student data.
  + If the DBMS provides a way to interactively enter and update the database, as well as interrogate it, this capability allows for managing personal databases. However, it may not leave an audit trail of actions or provide the kinds of controls necessary in a multi-user organization. These controls are only available when a set of application programs are customized for each data entry and updating function.
* A transaction mechanism, that ideally would guarantee the ACID properties, in order to ensure data integrity, despite concurrent user accesses (concurrency control), and faults (fault tolerance).
  + It also maintains the integrity of the data in the database.
  + The DBMS can maintain the integrity of the database by not allowing more than one user to update the same record at the same time. The DBMS can help prevent duplicate records via unique index constraints; for example, no two customers with the same customer numbers (key fields) can be entered into the database. See ACID properties for more information (Redundancy avoidance).

When a DBMS is used, information systems can be changed much more easily as the organization's information requirements change. to the Organizations may use one kind of DBMS for daily transaction processing and then move the detail onto another computer that uses another DBMS better suited for random inquiries and analysis. Overall systems design decisions are performed by data administrators and systems analysts. Detailed database design is performed by database administrators.

## SQL:

Structured Query Language (SQL) is the language used to manipulate relational databases. SQL is tied very closely with the relational model.

* In the relational model, data is stored in structures called relations or tables*.*

SQL statements are issued for the purpose of:

* Data definition: Defining tables and structures in the database (DDL used to create, alter and drop schema objects such as tables and indexes)

.

### : Stored Procedure

Routine name: proc Type: procedure

Definition: Select \* from register;

### : Triggers

It is the special kind of stored procedure that automatically executes when an event occurs in the database.

Triggers used :

1: Trigger name: on insert Table: register

Time: after Event: insert

INSERT INTO trig VALUES(null,NEW.rid,'Farmer Inserted',NOW())

2: Trigger name: on delete Table: register

Time: after Event: delete

Definition: INSERT INTO trig VALUES(null,OLD.rid,'FARMER DELETED',NOW())

3: Trigger name: on update Table: register

Time: after Event: update

Definition: INSERT INTO trig VALUES(null,NEW.rid,'FARMER UPDATED',NOW())

--

-- Database: `students`

--

-- --------------------------------------------------------

--

-- Table structure for table `attendence`

--

CREATE TABLE `attendence` (

  `aid` int(11) NOT NULL,

  `rollno` varchar(20) NOT NULL,

  `attendance` int(100) NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

--

-- Dumping data for table `attendence`

--

INSERT INTO `attendence` (`aid`, `rollno`, `attendance`) VALUES

(6, '1ve17cs012', 98);

-- --------------------------------------------------------

--

-- Table structure for table `department`

--

CREATE TABLE `department` (

  `cid` int(11) NOT NULL,

  `branch` varchar(50) NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

--

-- Dumping data for table `department`

--

INSERT INTO `department` (`cid`, `branch`) VALUES

(2, 'Information Science'),

(3, 'Electronic and Communication'),

(4, 'Electrical & Electronic'),

(5, 'Civil '),

(7, 'computer science'),

(8, 'IOT');

-- --------------------------------------------------------

--

-- Table structure for table `student`

--

CREATE TABLE `student` (

  `id` int(11) NOT NULL,

  `rollno` varchar(20) NOT NULL,

  `sname` varchar(50) NOT NULL,

  `sem` int(20) NOT NULL,

  `gender` varchar(50) NOT NULL,

  `branch` varchar(50) NOT NULL,

  `email` varchar(50) NOT NULL,

  `number` varchar(12) NOT NULL,

  `address` text NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

--

-- Triggers `student`

--

DELIMITER $$

CREATE TRIGGER `DELETE` BEFORE DELETE ON `student` FOR EACH ROW INSERT INTO trig VALUES(null,OLD.rollno,'STUDENT DELETED',NOW())

$$

DELIMITER ;

DELIMITER $$

CREATE TRIGGER `Insert` AFTER INSERT ON `student` FOR EACH ROW INSERT INTO trig VALUES(null,NEW.rollno,'STUDENT INSERTED',NOW())

$$

DELIMITER ;

DELIMITER $$

CREATE TRIGGER `UPDATE` AFTER UPDATE ON `student` FOR EACH ROW INSERT INTO trig VALUES(null,NEW.rollno,'STUDENT UPDATED',NOW())

$$

DELIMITER ;

-- --------------------------------------------------------

--

-- Table structure for table `test`

--

CREATE TABLE `test` (

  `id` int(11) NOT NULL,

  `name` varchar(52) NOT NULL,

  `email` varchar(50) NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

--

-- Dumping data for table `test`

--

INSERT INTO `test` (`id`, `name`, `email`) VALUES

(1, 'aaa', 'aaa@gmail.com');

-- --------------------------------------------------------

--

-- Table structure for table `trig`

--

CREATE TABLE `trig` (

  `tid` int(11) NOT NULL,

  `rollno` varchar(50) NOT NULL,

  `action` varchar(50) NOT NULL,

  `timestamp` datetime NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

--

-- Dumping data for table `trig`

--

INSERT INTO `trig` (`tid`, `rollno`, `action`, `timestamp`) VALUES

(7, '1ve17cs012', 'STUDENT INSERTED', '2021-01-10 19:19:56'),

(8, '1ve17cs012', 'STUDENT UPDATED', '2021-01-10 19:20:31'),

(9, '1ve17cs012', 'STUDENT DELETED', '2021-01-10 19:21:23');

-- --------------------------------------------------------

--

-- Table structure for table `user`

--

CREATE TABLE `user` (

  `id` int(11) NOT NULL,

  `username` varchar(50) NOT NULL,

  `email` varchar(50) NOT NULL,

  `password` varchar(500) NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

--

-- Dumping data for table `user`

--

INSERT INTO `user` (`id`, `username`, `email`, `password`) VALUES

(4, 'anees', 'anees@gmail.com', 'pbkdf2:sha256:150000$1CSLss89$ef995dfc48121768b2070bfbe7a568871cd56fac85ac7c95a1e645c8806146e9');

--

-- Indexes for dumped tables

--

--

-- Indexes for table `attendence`

--

ALTER TABLE `attendence`

  ADD PRIMARY KEY (`aid`);

--

-- Indexes for table `department`

--

ALTER TABLE `department`

  ADD PRIMARY KEY (`cid`);

--

-- Indexes for table `student`

--

ALTER TABLE `student`

  ADD PRIMARY KEY (`id`);

--

-- Indexes for table `test`

--

ALTER TABLE `test`

  ADD PRIMARY KEY (`id`);

--

-- Indexes for table `trig`

--

ALTER TABLE `trig`

  ADD PRIMARY KEY (`tid`);

--

-- Indexes for table `user`

--

ALTER TABLE `user`

  ADD PRIMARY KEY (`id`);

--

-- AUTO\_INCREMENT for dumped tables

--

--

-- AUTO\_INCREMENT for table `attendence`

--

ALTER TABLE `attendence`

  MODIFY `aid` int(11) NOT NULL AUTO\_INCREMENT, AUTO\_INCREMENT=7;

--

-- AUTO\_INCREMENT for table `department`

--

ALTER TABLE `department`

  MODIFY `cid` int(11) NOT NULL AUTO\_INCREMENT, AUTO\_INCREMENT=9;

--

-- AUTO\_INCREMENT for table `student`

--

ALTER TABLE `student`

  MODIFY `id` int(11) NOT NULL AUTO\_INCREMENT, AUTO\_INCREMENT=7;

--

-- AUTO\_INCREMENT for table `test`

--

ALTER TABLE `test`

  MODIFY `id` int(11) NOT NULL AUTO\_INCREMENT, AUTO\_INCREMENT=2;

--

-- AUTO\_INCREMENT for table `trig`

--

ALTER TABLE `trig`

  MODIFY `tid` int(11) NOT NULL AUTO\_INCREMENT, AUTO\_INCREMENT=10;

--

-- AUTO\_INCREMENT for table `user`

--

ALTER TABLE `user`

  MODIFY `id` int(11) NOT NULL AUTO\_INCREMENT, AUTO\_INCREMENT=5;

COMMIT;

**SQL COMMANDS**

**DDL:**

DDL is short name of **Data Definition Language,** which deals with database schemas and descriptions, of how the data should reside in the database.

● CREATE - to create a database and its objects like (table, index, views, store procedure, function, and triggers)

● ALTER - alters the structure of the existing database

● DROP - delete objects from the database

● TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed

● RENAME - rename an object

**DML:**

DML is short name of **Data Manipulation Language** which deals with data manipulation and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE, etc., and it is used to store, modify, retrieve, delete and update data in a database.

● SELECT - retrieve data from a database

● INSERT - insert data into a table

● UPDATE - updates existing data within a table

● DELETE - Delete all records from a database table

● MERGE - UPSERT operation (insert or update)

**DCL:**

DCL is short name of **Data Control Language** which includes commands such as GRANT and mostly concerned with rights, permissions and other controls of the database system.

● GRANT - allow users access privileges to the database

● REVOKE - withdraw users access privileges given by using the GRANT command

**TCL:**

TCL is short name of Transaction Control Language which deals with a transaction within a database.

● COMMIT - commits a Transaction

● ROLLBACK - rollback a transaction in case of any error occurs

● SAVEPOINT - to roll back the transaction making points within groups

**SQL:** SQL is a standard language for storing, manipulating and retrieving data in databases.

**SELECT:**

The SELECT statement is used to select data from a database.

**Syntax -**

● SELECT *column1*, *column2, ...*

FROM *table\_name*;

● Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax: ● SELECT \* FROM *table\_name*;

**Ex –**

● SELECT CustomerName, City FROM Customers;

**SELECT DISTINCT:**

The SELECT DISTINCT statement is used to return only distinct (different) values. **Syntax –**

● SELECT DISTINCT *column1*, *column2, ...*

FROM *table\_name*;

**Ex –**

● SELECT DISTINCT Country FROM Customers;

**WHERE:**

The WHERE clause is used to filter records.

**Syntax –**

● SELECT *column1*, *column2, ...*

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT \* FROM Customers

WHERE Country='Mexico';

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| <> | Not equal. **Note:** In some versions of SQL this operator may be written as != |

**AND, OR and NOT:**

The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

● The AND operator displays a record if all the conditions separated by AND are TRUE. ● The OR operator displays a record if any of the conditions separated by OR is TRUE.

The NOT operator displays a record if the condition(s) is NOT TRUE.

**Syntax –**

● SELECT *column1*, *column2, ...*

FROM *table\_name*

WHERE *condition1* AND *condition2* AND *condition3 ...*;

● SELECT *column1*, *column2, ...*

FROM *table\_name*

WHERE *condition1* OR *condition2* OR *condition3 ...*;

● SELECT *column1*, *column2, ...*

FROM *table\_name*

WHERE NOT *condition*;

**Ex –**

● SELECT \* FROM Customers

WHERE Country='Germany' AND City='Berlin';

● SELECT \* FROM Customers

WHERE Country='Germany' AND (City='Berlin' OR City='München');

**ORDER BY:**

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

**Syntax –**

● SELECT *column1*, *column2, ...*

FROM *table\_name*

ORDER BY *column1, column2, ...* ASC|DESC;

**Ex –**

● SELECT \* FROM Customers

ORDER BY Country;

● SELECT \* FROM Customers

ORDER BY Country ASC, CustomerName DESC;

**INSERT INTO:**

The INSERT INTO statement is used to insert new records in a table.

**Syntax –**

● INSERT INTO *table\_name* (*column1*, *column2*, *column3*, ...)

VALUES (*value1*, *value2*, *value3*, ...);

● INSERT INTO *table\_name*

VALUES (*value1*, *value2*, *value3*, ...);

\*In the second syntax, make sure the order of the values is in the same order as the columns in the table.

**Ex –**

● INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)

VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

**NULL Value:**

It is not possible to test for NULL values with comparison operators, such as =, <, or <>. We will have to use the IS NULL and IS NOT NULL operators instead.

**Syntax –**

● SELECT *column\_names*

FROM *table\_name*

WHERE *column\_name* IS NULL;

● SELECT *column\_names*

FROM *table\_name*

WHERE *column\_name* IS NOT NULL;

**Ex –**

● SELECT CustomerName, ContactName, Address

FROM Customers

WHERE Address IS NULL;

**UPDATE:**

The UPDATE statement is used to modify the existing records in a table. **Syntax –**

● UPDATE *table\_name*

SET *column1* = *value1*, *column2* = *value2*, ...

WHERE *condition*;

**Ex –**

● UPDATE Customers

SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'

WHERE CustomerID = 1;

**DELETE:**

The DELETE statement is used to delete existing records in a table.

**Syntax –**

● DELETE FROM *table\_name* WHERE *condition*;

● DELETE FROM *table\_name*;

In 2ndsyntax, all rows are deleted. The table structure, attributes, and indexes will be intact **Ex –**

● DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

**SELECT TOP:**

The SELECT TOP clause is used to specify the number of records to return. **Syntax –**

● SELECT TOP *number*|*percent column\_name(s)*

FROM *table\_name*

WHERE *condition*;

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE *condition*

LIMIT *number*;

● SELECT *column\_name(s)*

FROM *table\_name*

ORDER BY *column\_name(s)*

FETCH FIRST *number* ROWS ONLY;

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE ROWNUM <= *number*;

\*In case the interviewer asks other than the TOP, rest are also correct. (Diff. DB Systems) **Ex –**

● SELECT TOP 3 \* FROM Customers;

● SELECT \* FROM Customers

LIMIT 3;

● SELECT \* FROM Customers

FETCH FIRST 3 ROWS ONLY;

**Aggregate Functions:**

**MIN():**

The MIN() function returns the smallest value of the selected column. **Syntax –**

● SELECT MIN(*column\_name*)

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT MIN(Price) AS SmallestPrice

FROM Products;

**MAX():**

The MAX() function returns the largest value of the selected column. **Syntax –**

● SELECT MAX(*column\_name*)

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT MAX(Price) AS LargestPrice

FROM Products;

**COUNT():**

The COUNT() function returns the number of rows that matches a specified criterion. **Syntax –**

● SELECT COUNT(*column\_name*)

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT COUNT(ProductID)

FROM Products;

**AVG():**

The AVG() function returns the average value of a numeric column.

**Syntax –**

● SELECT AVG(*column\_name*)

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT AVG(Price)

FROM Products;

**SUM():**

The SUM() function returns the total sum of a numeric column.

**Syntax –**

● SELECT SUM(*column\_name*)

FROM *table\_name*

WHERE *condition*;

**Ex –**

● SELECT SUM(Quantity)

FROM OrderDetails;

**LIKE Operator:**

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. There are two wildcards often used in conjunction with the LIKE operator:

● The percent sign (%) represents zero, one, or multiple characters

● The underscore sign (\_) represents one, single character

**Syntax –**

● SELECT *column1, column2, ...*

FROM *table\_name*

WHERE *columnN* LIKE *pattern*;

|  |  |
| --- | --- |
| **LIKE Operator** | **Description** |
| WHERE CustomerName LIKE 'a%' | Finds any values that start with "a" |
| WHERE CustomerName LIKE '%a' | Finds any values that end with "a" |
| WHERE CustomerName LIKE '%or%' | Finds any values that have "or" in any position |

|  |  |
| --- | --- |
| WHERE CustomerName LIKE '\_r%' | Finds any values that have "r" in the second position |
| WHERE CustomerName LIKE 'a\_%' | Finds any values that start with "a" and are at least 2 characters in length |
| WHERE CustomerName LIKE 'a\_\_%' | Finds any values that start with "a" and are at least 3 characters in length |
| WHERE ContactName LIKE 'a%o' | Finds any values that start with "a" and ends with "o" |

**IN:**

The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.

**Syntax –**

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE *column\_name* IN (*value1*, *value2*, ...);

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE *column\_name* IN (*SELECT STATEMENT*);

**Ex –**

● SELECT \* FROM Customers

WHERE Country IN ('Germany', 'France', 'UK');

● SELECT \* FROM Customers

WHERE Country IN (SELECT Country FROM Suppliers);

**BETWEEN:**

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

**Syntax –**

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE *column\_name* BETWEEN *value1* AND *value2;*

**Ex –**

● SELECT \* FROM Products

WHERE Price BETWEEN 10 AND 20;

**Joins:**

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

**INNER JOIN:**

The INNER JOIN keyword selects records that have matching values in both tables. **Syntax –**

● SELECT *column\_name(s)*

FROM *table1*

INNER JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*;

**Ex –**

● SELECT Orders.OrderID, Customers.CustomerName

FROM Orders

INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

**LEFT (OUTER) JOIN:**

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

**Syntax –**

● SELECT *column\_name(s)*

FROM *table1*

LEFT JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*;

**Ex –**

● SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID

ORDER BY Customers.CustomerName;

**RIGHT (OUTER) JOIN:**

The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

**Syntax –**

● SELECT *column\_name(s)*

FROM *table1*

RIGHT JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*;

**Ex –**

● SELECT Orders.OrderID, Employees.LastName, Employees.FirstName FROM Orders

RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID ORDER BY Orders.OrderID;

**FULL (OUTER) JOIN:**

The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.

**Syntax:**

● SELECT *column\_name(s)*

FROM *table1*

FULL OUTER JOIN *table2*

ON *table1.column\_name* = *table2.column\_name*

WHERE *condition*;

**Ex –**

● SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID ORDER BY Customers.CustomerName;

**UNION:**

The UNION operator is used to combine the result-set of two or more SELECT statements.

● Every SELECT statement within UNION must have the same number of columns ● The columns must also have similar data types

● The columns in every SELECT statement must also be in the same order

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL

**Syntax –**

● SELECT *column\_name(s)* FROM *table1*

UNION

SELECT *column\_name(s)* FROM *table2*;

● SELECT *column\_name(s)* FROM *table1*

UNION ALL

SELECT *column\_name(s)* FROM *table2*;

**Ex –**

● SELECT City FROM Customers

UNION

SELECT City FROM Suppliers

ORDER BY City;

**GROUP BY:**

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions

(COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns. **Syntax –**

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE *condition*

GROUP BY *column\_name(s)*

ORDER BY *column\_name(s);*

**Ex –**

● SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country

ORDER BY COUNT(CustomerID) DESC;

**HAVING:**

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

\*WHERE is given priority over HAVING.

**Syntax –**

● SELECT *column\_name(s)*

FROM *table\_name*

WHERE *condition*

GROUP BY *column\_name(s)*

HAVING *condition*

ORDER BY *column\_name(s);*

**Ex –**

● SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country

HAVING COUNT(CustomerID) > 5;

**CREATE DATABASE:**

The CREATE DATABASE statement is used to create a new SQL database. **Syntax –**

● CREATE DATABASE *databasename*;

**DROP DATABASE:**

The DROP DATABASE statement is used to drop an existing SQL database. **Syntax –**

● DROP DATABASE *databasename*;

**CREATE TABLE:**

The CREATE TABLE statement is used to create a new table in a database.

**Syntax –**

● CREATE TABLE *table\_name* (

*column1 datatype*,

*column2 datatype*,

*column3 datatype*,

 ....

);

**DROP TABLE:**

The DROP TABLE statement is used to drop an existing table in a database. **Syntax –**

● DROP TABLE *table\_name*;

**TRUNCATE TABLE:**

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself. **Syntax –**

● TRUNCATE TABLE *table\_name*;

**ALTER TABLE:**

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

**Syntax –**

● ALTER TABLE *table\_name*

ADD *column\_name datatype*;

● ALTER TABLE *table\_name*

DROP COLUMN *column\_name*;

● ALTER TABLE *table\_name*

MODIFY COLUMN *column\_name datatype*;

**Ex –**

● ALTER TABLE Customers

ADD Email varchar(255);

● ALTER TABLE Customers

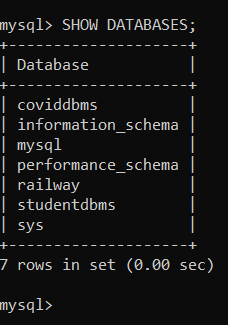
DROP COLUMN Email;

● ALTER TABLE Persons

ALTER COLUMN DateOfBirth year;

A. View all databases

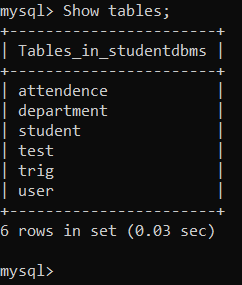
SHOW DATABASES;



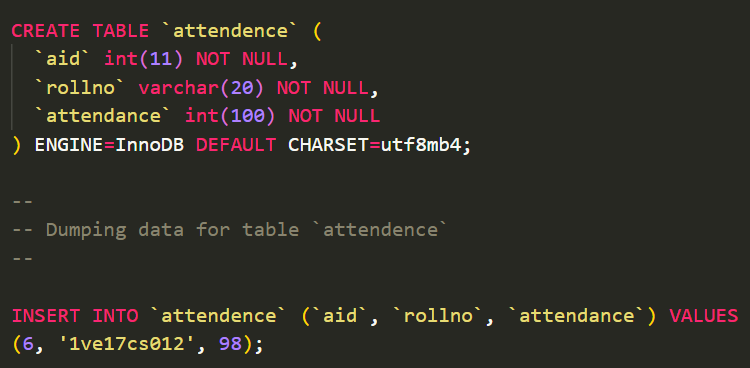
B. Create a Database

CREATE DATABASE studentdbms;

C. View all Tables in a Database



D. Create Tables (with constraints)/Inserting Records in a Table



E. Updating/Deleting Records in a Table

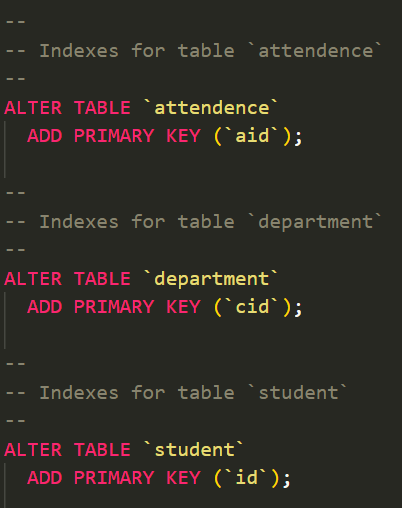
UPDATE student

SET sem = ‘4’

WHERE id = 2;

DELETE FROM department WHERE cid = ‘4’;

F. Altering a Table

**

G. Dropping

DROP TABLE *attendance;*

H. Truncating

TRUNCATE TABLE *department;*

I. Renaming Tables

SELECT sname AS StudentName

FROM student;

# BACKEND PYHTON WITH MYSQL CODE

from flask import Flask,render\_template,request,session,redirect,url\_for,flash

from flask\_sqlalchemy import SQLAlchemy

from flask\_login import UserMixin

from werkzeug.security import generate\_password\_hash,check\_password\_hash

from flask\_login import login\_user,logout\_user,login\_manager,LoginManager

from flask\_login import login\_required,current\_user

import json

# MY db connection

local\_server= True

app = Flask(\_\_name\_\_)

app.secret\_key='aditigulati'

# this is for getting unique user access

login\_manager=LoginManager(app)

login\_manager.login\_view='login'

@login\_manager.user\_loader

*def* load\_user(*user\_id*):

    return User.query.get(int(*user\_id*))

# app.config['SQLALCHEMY\_DATABASE\_URL']='mysql://username:password@localhost/databas\_table\_name'

app.config['SQLALCHEMY\_DATABASE\_URI']='mysql://root:Qwertyuiop12#@localhost/studentdbms'

db=SQLAlchemy(app)

# here we will create db models that is tables

*class* Test(db*.*Model):

    id=db.Column(db.Integer,*primary\_key*=True)

    name=db.Column(db.String(100))

    email=db.Column(db.String(100))

*class* Department(db*.*Model):

    cid=db.Column(db.Integer,*primary\_key*=True)

    branch=db.Column(db.String(100))

*class* Attendence(db*.*Model):

    aid=db.Column(db.Integer,*primary\_key*=True)

    rollno=db.Column(db.String(100))

    attendance=db.Column(db.Integer())

*class* Trig(db*.*Model):

    tid=db.Column(db.Integer,*primary\_key*=True)

    rollno=db.Column(db.String(100))

    action=db.Column(db.String(100))

    timestamp=db.Column(db.String(100))

*class* User(UserMixin,db*.*Model):

    id=db.Column(db.Integer,*primary\_key*=True)

    username=db.Column(db.String(50))

    email=db.Column(db.String(50),*unique*=True)

    password=db.Column(db.String(1000))

*class* Student(db*.*Model):

    id=db.Column(db.Integer,*primary\_key*=True)

    rollno=db.Column(db.String(50))

    sname=db.Column(db.String(50))

    sem=db.Column(db.Integer)

    gender=db.Column(db.String(50))

    branch=db.Column(db.String(50))

    email=db.Column(db.String(50))

    number=db.Column(db.String(12))

    address=db.Column(db.String(100))

@app.route('/')

*def* index():

    return render\_template('index.html')

@app.route('/studentdetails')

*def* studentdetails():

    query=db.engine.execute(f"SELECT \* FROM `student`")

    return render\_template('studentdetails.html',*query*=query)

@app.route('/triggers')

*def* triggers():

    query=db.engine.execute(f"SELECT \* FROM `trig`")

    return render\_template('triggers.html',*query*=query)

@app.route('/department',*methods*=['POST','GET'])

*def* department():

    if request.method=="POST":

        dept=request.form.get('dept')

        query=Department.query.filter\_by(*branch*=dept).first()

        if query:

            flash("Department Already Exist","warning")

            return redirect('/department')

        dep=Department(*branch*=dept)

        db.session.add(dep)

        db.session.commit()

        flash("Department Added","success")

    return render\_template('department.html')

@app.route('/addattendance',*methods*=['POST','GET'])

*def* addattendance():

    query=db.engine.execute(f"SELECT \* FROM `student`")

    if request.method=="POST":

        rollno=request.form.get('rollno')

        attend=request.form.get('attend')

        print(attend,rollno)

        atte=Attendence(*rollno*=rollno,*attendance*=attend)

        db.session.add(atte)

        db.session.commit()

        flash("Attendance Added","warning")

    return render\_template('attendance.html',*query*=query)

@app.route('/search',*methods*=['POST','GET'])

*def* search():

    if request.method=="POST":

        rollno=request.form.get('roll')

        bio=Student.query.filter\_by(*rollno*=rollno).first()

        attend=Attendence.query.filter\_by(*rollno*=rollno).first()

        return render\_template('search.html',*bio*=bio,*attend*=attend)

    return render\_template('search.html')

@app.route("/delete/<string:id>",*methods*=['POST','GET'])

@login\_required

*def* delete(*id*):

    db.engine.execute(f"DELETE FROM `student` WHERE `student`.`id`={*id*}")

    flash("Slot Deleted Successfully","danger")

    return redirect('/studentdetails')

@app.route("/edit/<string:id>",*methods*=['POST','GET'])

@login\_required

*def* edit(*id*):

    dept=db.engine.execute("SELECT \* FROM `department`")

    posts=Student.query.filter\_by(*id*=*id*).first()

    if request.method=="POST":

        rollno=request.form.get('rollno')

        sname=request.form.get('sname')

        sem=request.form.get('sem')

        gender=request.form.get('gender')

        branch=request.form.get('branch')

        email=request.form.get('email')

        num=request.form.get('num')

        address=request.form.get('address')

        query=db.engine.execute(f"UPDATE `student` SET `rollno`='{rollno}',`sname`='{sname}',`sem`='{sem}',`gender`='{gender}',`branch`='{branch}',`email`='{email}',`number`='{num}',`address`='{address}'")

        flash("Slot Updated Successfully","success")

        return redirect('/studentdetails')

    return render\_template('edit.html',*posts*=posts,*dept*=dept)

@app.route('/signup',*methods*=['POST','GET'])

*def* signup():

    if request.method == "POST":

        username=request.form.get('username')

        email=request.form.get('email')

        password=request.form.get('password')

        user=User.query.filter\_by(*email*=email).first()

        if user:

            flash("Email Already Exist","warning")

            return render\_template('/signup.html')

        encpassword=generate\_password\_hash(password)

        new\_user=db.engine.execute(f"INSERT INTO `user` (`username`,`email`,`password`) VALUES ('{username}','{email}','{encpassword}')")

        # this is method 2 to save data in db

        # newuser=User(username=username,email=email,password=encpassword)

        # db.session.add(newuser)

        # db.session.commit()

        flash("Signup Successful Please Login","success")

        return render\_template('login.html')

    return render\_template('signup.html')

@app.route('/login',*methods*=['POST','GET'])

*def* login():

    if request.method == "POST":

        email=request.form.get('email')

        password=request.form.get('password')

        user=User.query.filter\_by(*email*=email).first()

        if user and check\_password\_hash(user.password,password):

            login\_user(user)

            flash("Login Success","primary")

            return redirect(url\_for('index'))

        else:

            flash("Invalid Credentials","danger")

            return render\_template('login.html')

    return render\_template('login.html')

@app.route('/logout')

@login\_required

*def* logout():

    logout\_user()

    flash("Logout SuccessFully","warning")

    return redirect(url\_for('login'))

@app.route('/addstudent',*methods*=['POST','GET'])

@login\_required

*def* addstudent():

    dept=db.engine.execute("SELECT \* FROM `department`")

    if request.method=="POST":

        rollno=request.form.get('rollno')

        sname=request.form.get('sname')

        sem=request.form.get('sem')

        gender=request.form.get('gender')

        branch=request.form.get('branch')

        email=request.form.get('email')

        num=request.form.get('num')

        address=request.form.get('address')

        query=db.engine.execute(f"INSERT INTO `student` (`rollno`,`sname`,`sem`,`gender`,`branch`,`email`,`number`,`address`) VALUES ('{rollno}','{sname}','{sem}','{gender}','{branch}','{email}','{num}','{address}')")

        flash("Studented Added Successfully","info")

    return render\_template('student.html',*dept*=dept)

@app.route('/test')

*def* test():

    try:

        Test.query.all()

        return 'My Database is Connected'

    except:

        return 'My db is not Connected'

app.run(*debug*=True)

# FRONT END CODE

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta content="width=device-width, initial-scale=1.0" name="viewport">

<title>{% block title %}

{% endblock title %}</title>

<meta content="" name="description">

<meta content="" name="keywords">

{% block style %}

{% endblock style %}

<link href="https://fonts.googleapis.com/css?family=Open+Sans:300,300i,400,400i,700,700i|Raleway:3 00,400,500,700,800" rel="stylesheet">

<!-- Vendor CSS Files -->

<link href="static/assets/vendor/bootstrap/css/bootstrap.min.css" rel="stylesheet">

<link href="static/assets/vendor/venobox/venobox.css" rel="stylesheet">

<link href="static/assets/vendor/font-awesome/css/font-awesome.min.css" rel="stylesheet">

<link href="static/assets/vendor/owl.carousel/assets/owl.carousel.min.css" rel="stylesheet">

<link href="static/assets/vendor/aos/aos.css" rel="stylesheet">

<!-- Template Main CSS File -->

<link href="static/assets/css/style.css" rel="stylesheet">

</head>

<body>

<!-- ======= Header ======= -->

<header id="header">

<div class="container">

<div id="logo" class="pull-left">

<a href="/" class="scrollto">S.M.S</a>

</div>

<nav id="nav-menu-container">

<ul class="nav-menu">

<li class="{% block home %}

{% endblock home %}"><a href="/">Home</a></li>

<li><a href="/addstudent">Students</a></li>

<li><a href="/addattendance">Attendance</a></li>

<li><a href="/department">Department</a></li>

<li><a href="/triggers">Records</a></li>

<li><a href="/studentdetails">Student Details</a></li>

<li><a href="/search">Search</a></li>

<li><a href="/about">About</a></li>

{% if current\_user.is\_authenticated %}

<li class="buy-tickets"><a href="">Welcome</a></li>

<li class="buy-tickets"><a href="/logout">Logout</a></li>

{% else %}

<li class="buy-tickets"><a href="/signup">Signin</a></li>

{% endif %}

</ul>

</nav><!-- #nav-menu-container -->

</div>

</header><!-- End Header -->

<!-- ======= Intro Section ======= -->

<section id="intro">

<div class="intro-container" data-aos="zoom-in" data-aos-delay="100">

<h1 class="mb-4 pb-0">STUDENT MANAGEMENT SYSTEM </span> </h1>

<p class="mb-4 pb-0">DBMS Mini Project Using Flask & MYSQL</p>

<a href="" class="about-btn scrollto">View More</a>

</div>

</section><!-- End Intro Section -->

<main id="main">

{% block body %}

{% with messages=get\_flashed\_messages(with\_categories=true) %}

{% if messages %}

{% for category, message in messages %}

<div class="alert alert-{{category}} alert-dismissible fade show" role="alert">

{{message}}

</div>

{% endfor %}

{% endif %}

{% endwith %}

{% endblock body %}

<a href="#" class="back-to-top"><i class="fa fa-angle-up"></i></a>

<!-- Vendor JS Files -->

<script src="static/assets/vendor/jquery/jquery.min.js"></script>

<script src="static/assets/vendor/bootstrap/js/bootstrap.bundle.min.js"></script>

<script src="static/assets/vendor/jquery.easing/jquery.easing.min.js"></script>

<script src="static/assets/vendor/php-email-form/validate.js"></script>

<script src="static/assets/vendor/venobox/venobox.min.js"></script>

<script src="static/assets/vendor/owl.carousel/owl.carousel.min.js"></script>

<script src="static/assets/vendor/superfish/superfish.min.js"></script>

<script src="static/assets/vendor/hoverIntent/hoverIntent.js"></script>

<script src="static/assets/vendor/aos/aos.js"></script>

<!-- Template Main JS File -->

<script src="static/assets/js/main.js"></script>

</body>

</html> 2.Students.html

{% extends 'base.html' %}

{% block title %} Add Students

{% endblock title %}

{% block body %}

<h3 class="text-center"><span>Add Student Details</span> </h3>

{% with messages=get\_flashed\_messages(with\_categories=true) %}

{% if messages %}

{% for category, message in messages %}

<div class="alert alert-{{category}} alert-dismissible fade show" role="alert">

{{message}}

</div>

{% endfor %}

{% endif %}

{% endwith %}

<br>

<div class="container">

<div class="row">

<div class="col-md-4"></div>

<div class="col-md-4">

<form action="/addstudent" method="post">

<div class="form-group">

<label for="rollno">Roll Number</label>

<input type="text" class="form-control" name="rollno" id="rollno">

</div>

<br>

<div class="form-group">

<label for="sname">Student Name</label>

<input type="text" class="form-control" name="sname" id="sname">

</div>

<br>

<div class="form-group">

<label for="sem">Sem</label>

<input type="number" class="form-control" name="sem" id="sem">

</div>

<br>

<div class="form-group">

<select class="form-control" id="gender" name="gender" required>

<option selected>Select Gender</option>

<option value="male">Male</option>

<option value="female">Female</option>

</select>

</div>

<br>

<div class="form-group">

<select class="form-control" id="branch" name="branch" required>

<option selected>Select Branch</option>

{% for d in dept %}

<option value="{{d.branch}}">{{d.branch}}</option>

{% endfor %}

</select>

</div>

<br>

<div class="form-group">

<label for="email">Email</label>

<input type="email" class="form-control" name="email" id="email">

</div>

<br>

<div class="form-group">

<label for="num">Phone Number</label>

<input type="number" class="form-control" name="num" id="num">

</div>

<br>

<div class="form-group">

<label for="address">Address</label>

<textarea class="form-control" name="address" id="address"></textarea>

</div>

<br>

<button type="submit" class="btn btn-danger btn-sm btn-block">Add Record</button>

</form>

<br>

<br>

</div>

<div class="col-md-4"></div>

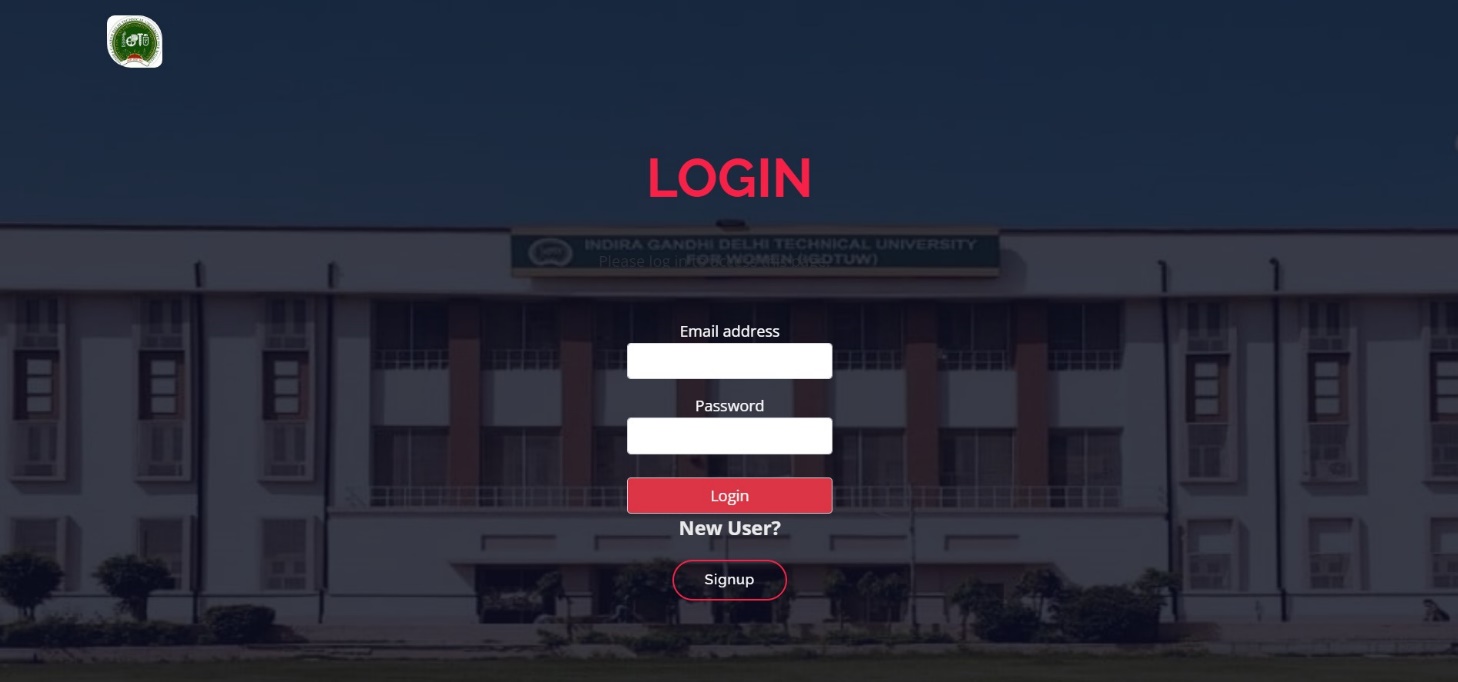
</div></div>

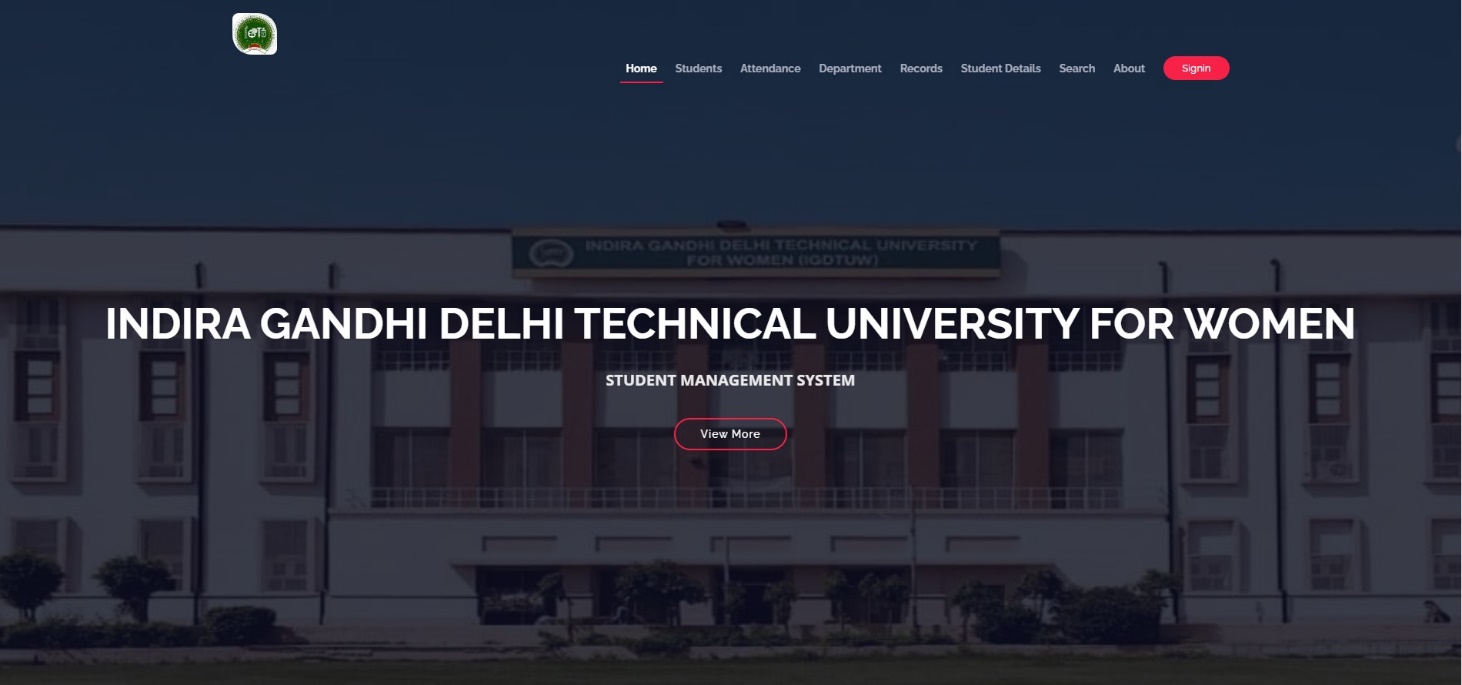
{% endblock body %}

## USER INTERFACE

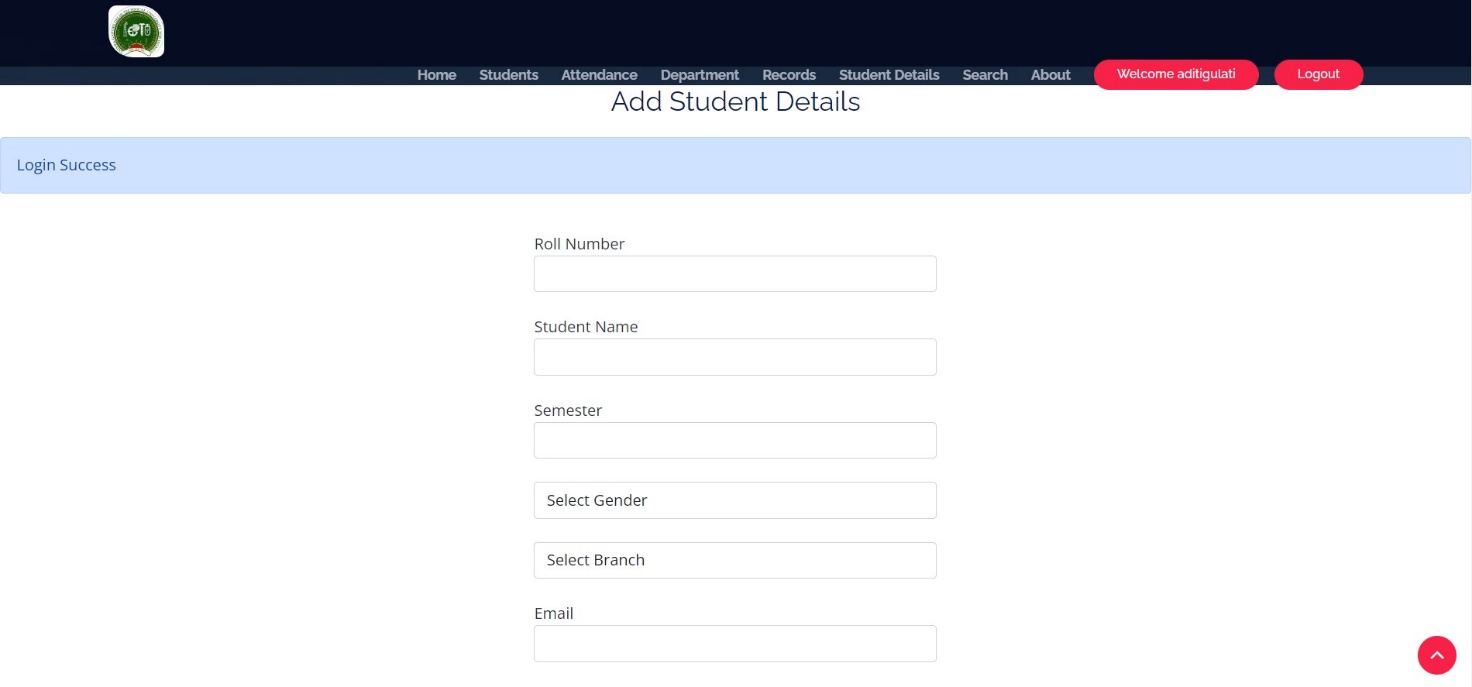
**4.1 SCREEN SHOTS**

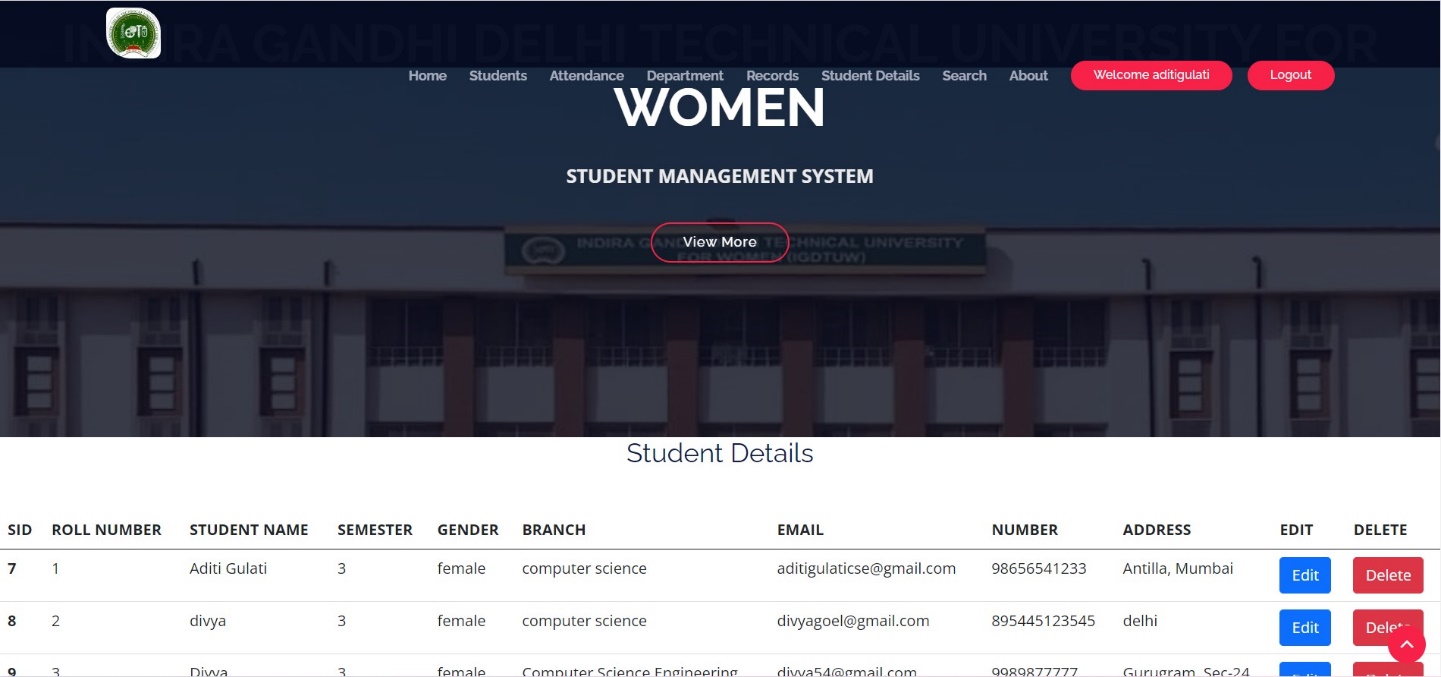
### LOGIN PAGE:



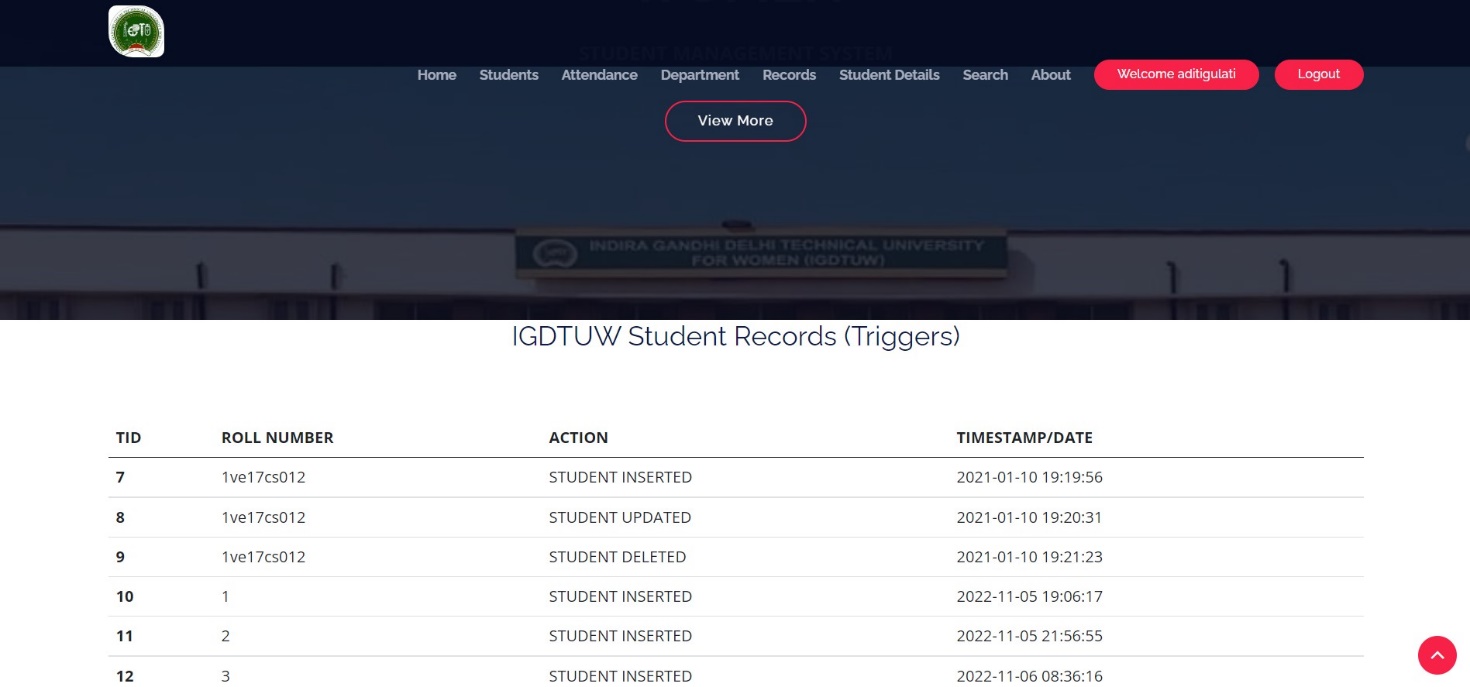


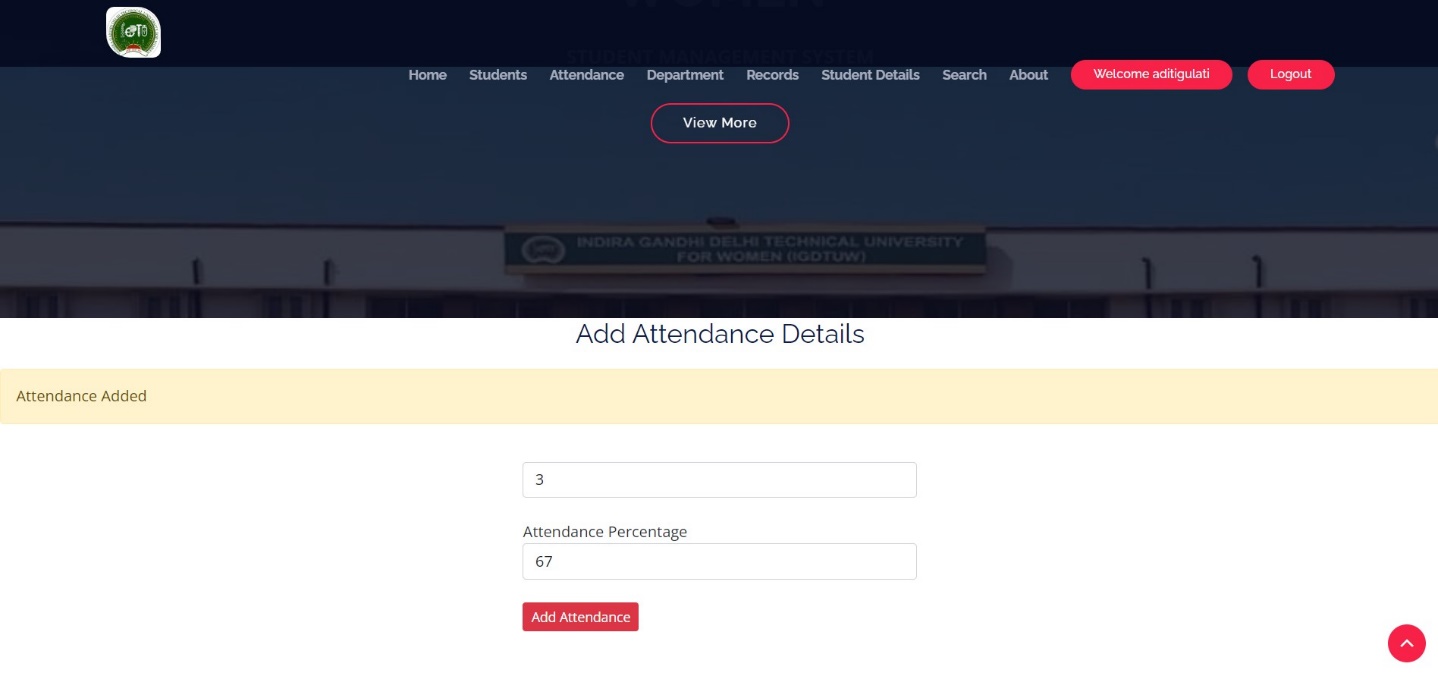
**ADD STUDENTS INFO**

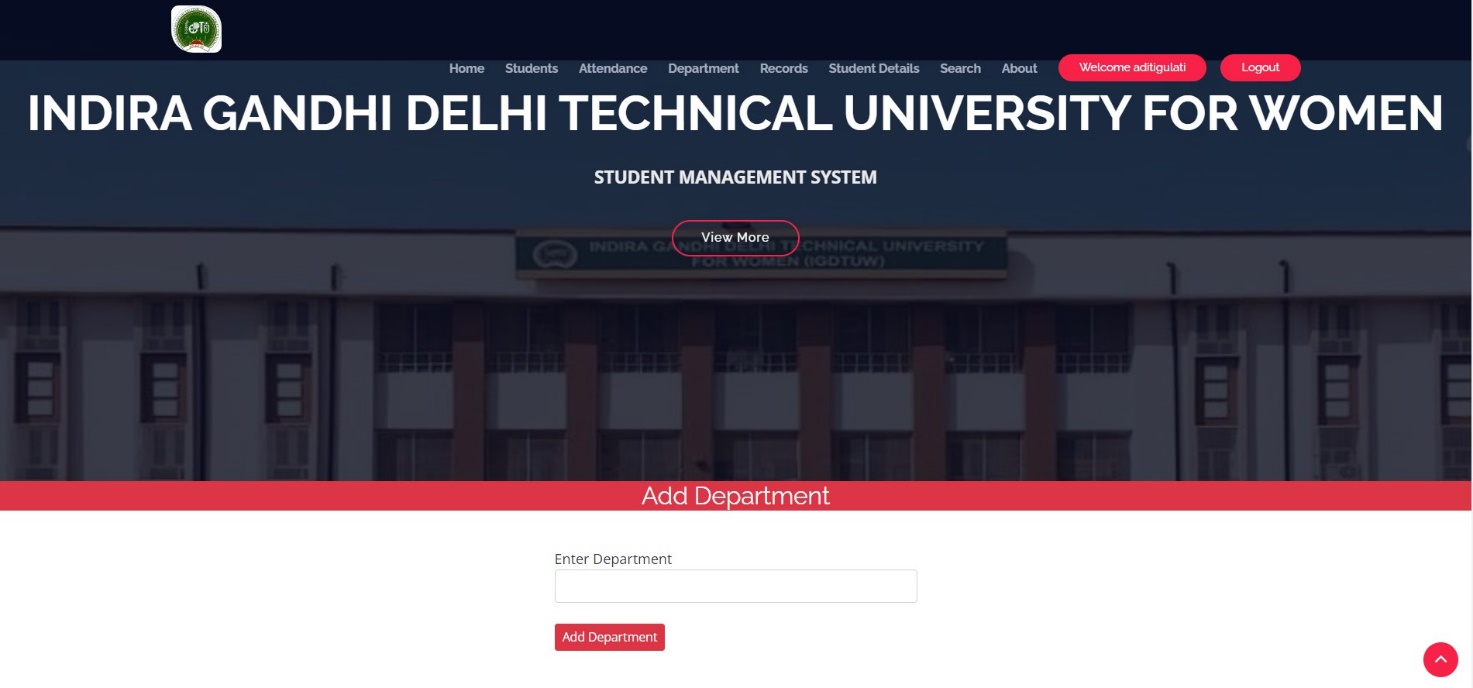


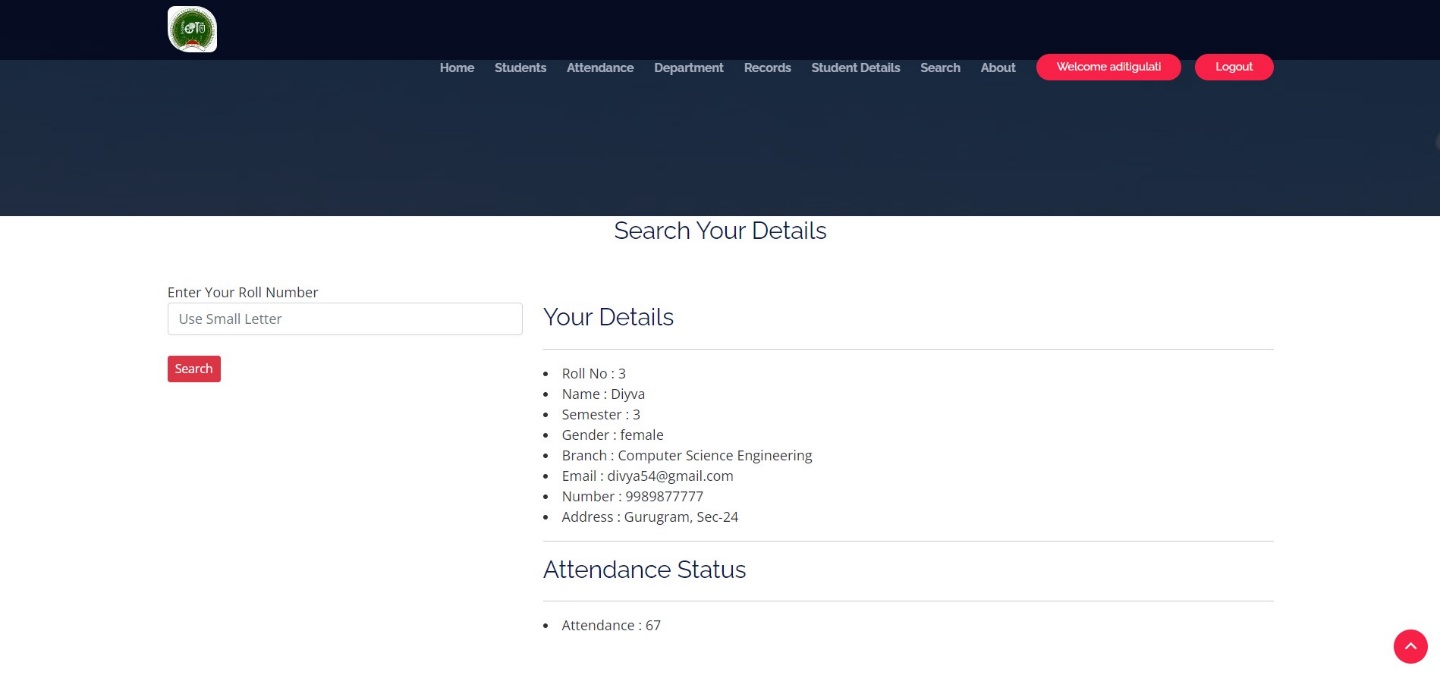


## TRIGGERS RECORDS

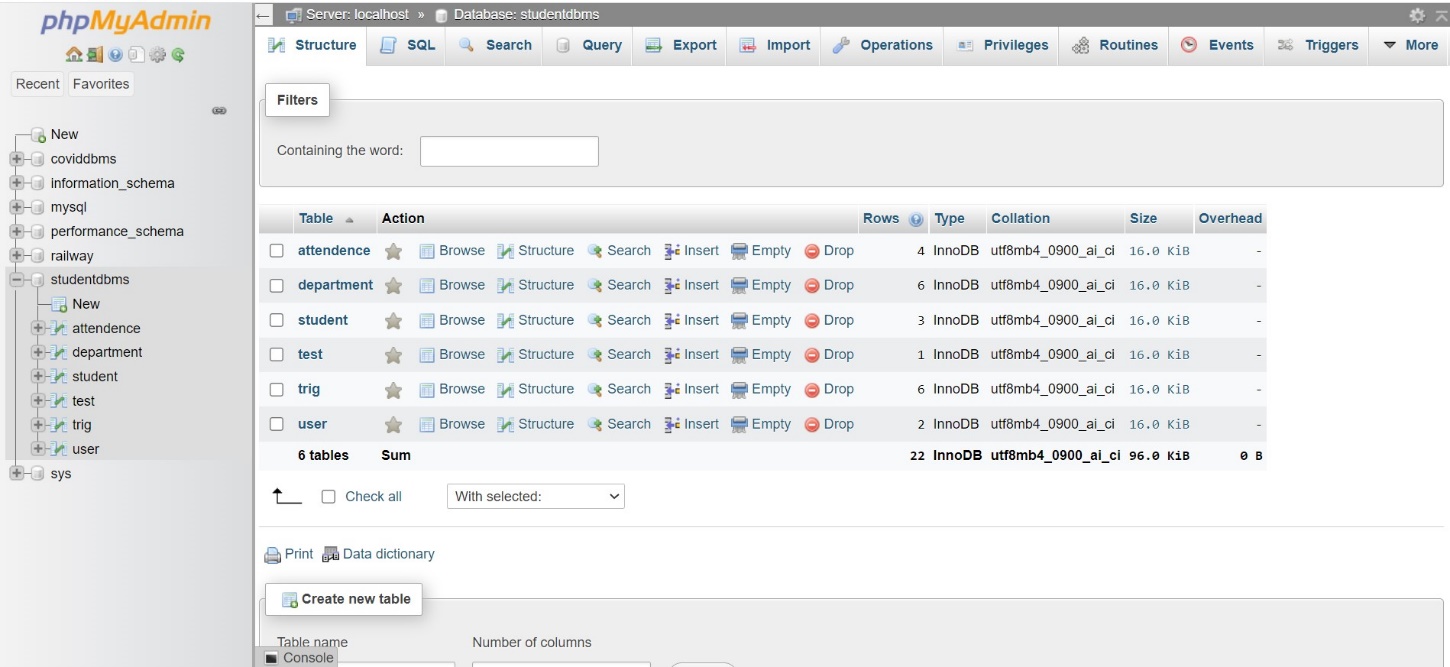


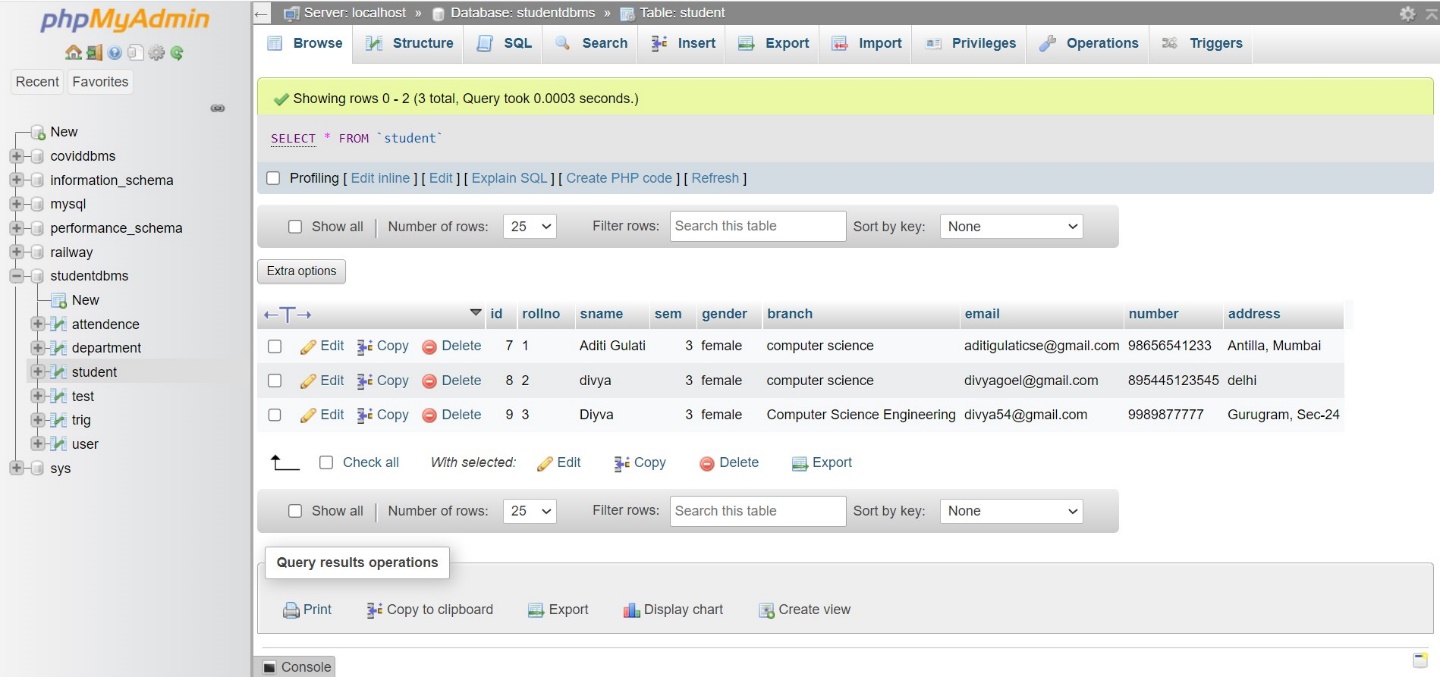


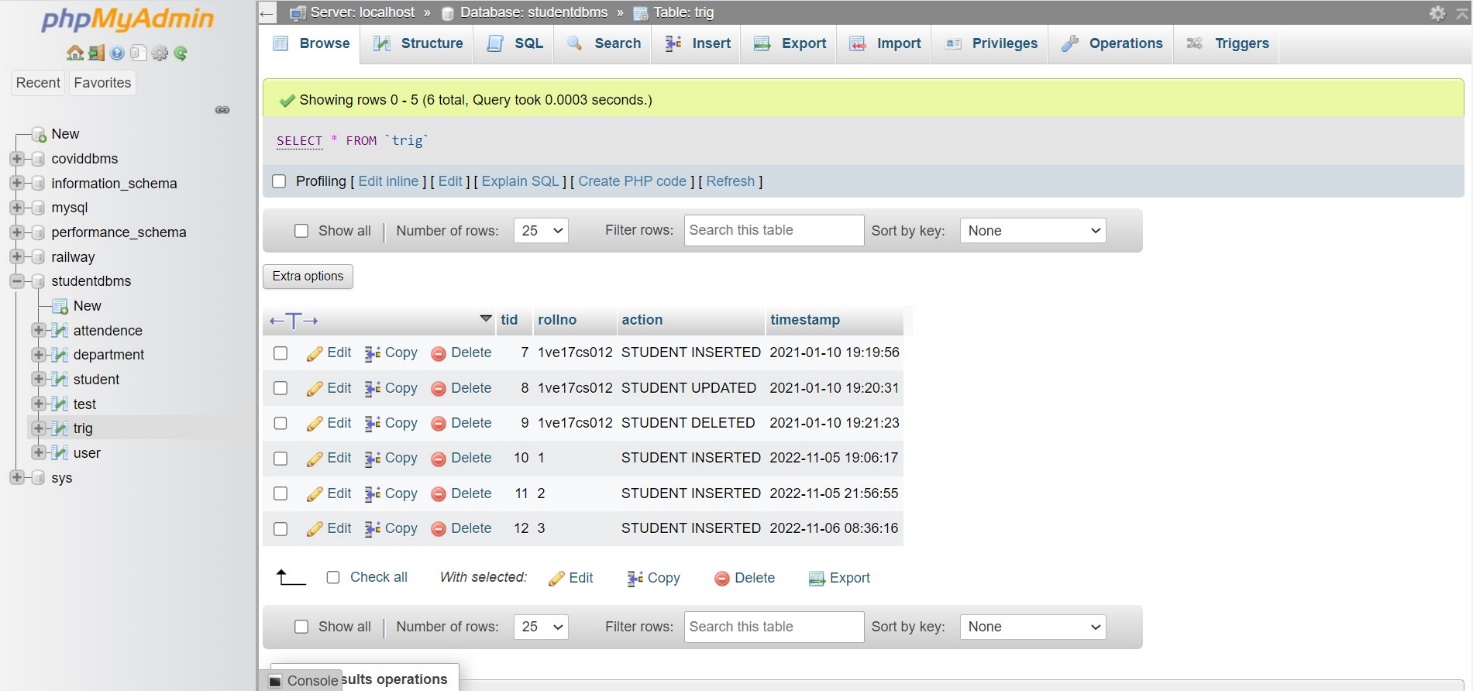


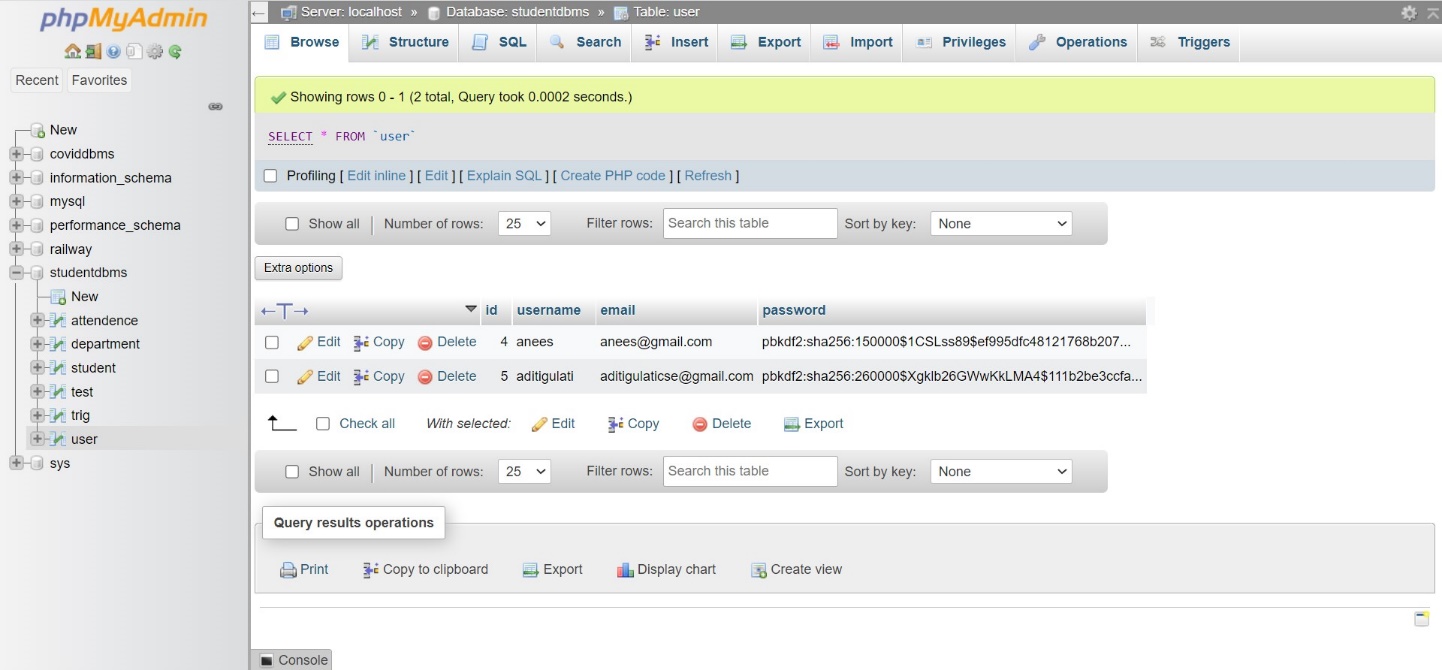


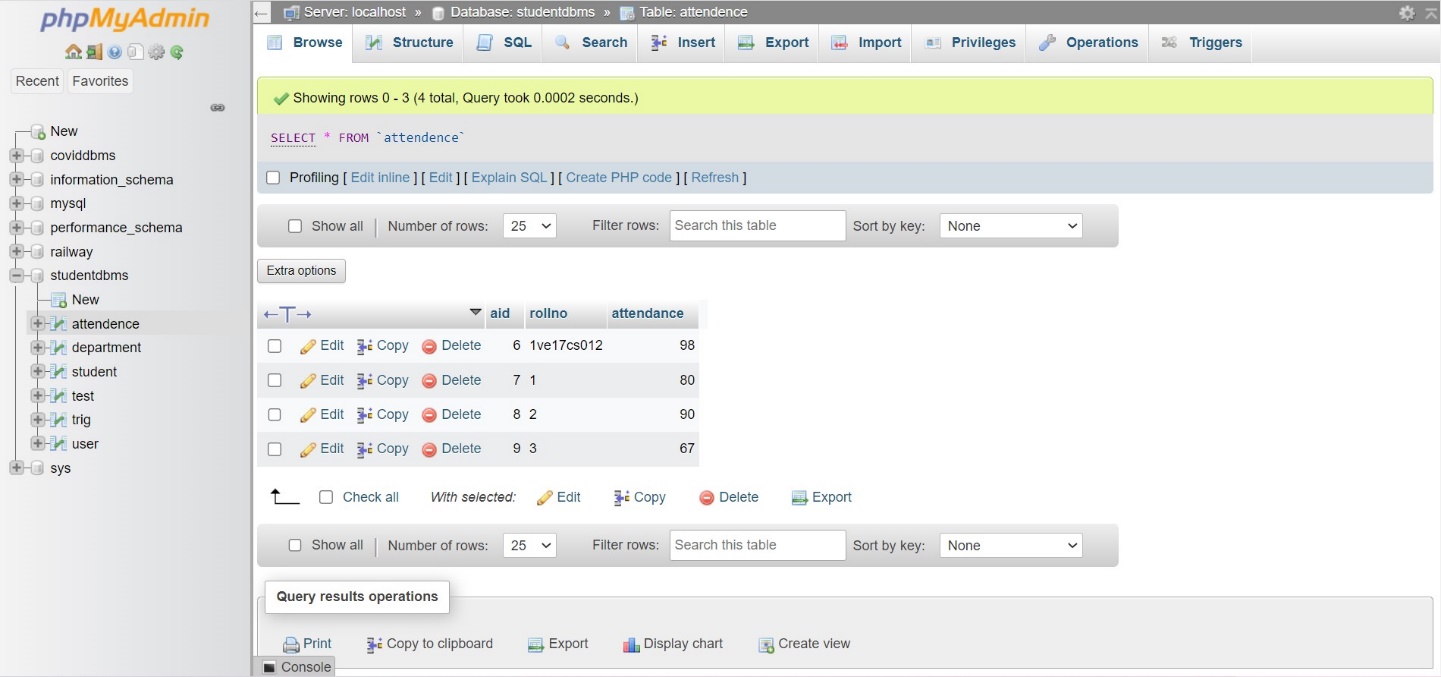
**DATABASE LOCALHOST**











# CONCLUSION

STUDENT MANAGEMENT SYSTEM successfully implemented based on online data filling which helps us in administrating the data user for managing the tasks performed in students. The project successfully used various functionalities of Xampp and python flask and also create the fully functional database management system for online portals.

Using MySQL as the database is highly beneficial as it is free to download, popular and can be easily customized. The data stored in the MySQL database can easily be retrieved and manipulated according to the requirements with basic knowledge of SQL.

With the theoretical inclination of our syllabus it becomes very essential to take the atmost advantage of any opportunity of gaining practical experience that comes along. The building blocks of this Major Project “Students Management System” was one of these opportunities. It gave us the requisite practical knowledge to supplement the already taught theoretical concepts thus making us more competent as a computer engineer. The project from a personal point of view also helped us in understanding the following aspects of project development:

* The planning that goes into implementing a project.
* The importance of proper planning and an organized methodology.
* The key element of team spirit and co-ordination in a successful project

# FUTURE ENHANCEMENT

* Enhanced database storage facility
* Enhanced user friendly GUI
* more advanced results systems
* online feedbacks forms

# REFERENCES

* https://[www.youtube.com](http://www.youtube.com/)
* <http://www.etbootstrap.com>
* <https://www.db-book.com/db6/slide-dir/>
* <https://www.w3schools.com/sql>
* https://[www.google.com](http://www.google.com/)