Kapasa Makasa University

Agriculture and Aquatic Sciences Department

ICT DATABASE SYSTEM PRACTICAL STUDENT MANUAL PART ONE (1)

TAKE NOTE:

Students are expected to work through this manual in conjunction with the course outline provided.

Students must attempt all the practical exercises in order to complete the course successfully.

COURSE OUTLINE

Background and Rationale:

This course introduces students to the fundamentals of database development and implementation. Database Systems involves designing, creating, and managing databases to efficiently store, organize, and retrieve data for various applications and business needs. Students will learn to design efficient databases, implement data models, and utilise database management systems effectively.

Learning Outcomes:

By the end of this course, students should be able to:

1. Understand fundamental database concepts and principles.
2. Design a normalized database schema based on given requirements.
3. Implement databases using a selected DBMS (e.g., MySQL, PostgreSQL).
4. Query databases using SQL and optimize queries for performance.
5. Ensure data integrity, security, and privacy within a database.
6. Develop skills for database optimization and performance tuning.
7. Implement backup, recovery, and maintenance strategies for databases.
8. Work on a database development project from inception to completion.

Course Content:

1. **Introduction to Databases**
    - Overview of databases and their importance - Types of databases: relational, NoSQL, etc.
    - Evolution of database management systems (DBMS)
2. **Database Design and Modeling**
    - Entity-Relationship Diagrams (ERD)
    - Data modeling and normalization
    - Functional dependencies and normalization forms (1NF to 3NF)
3. **Relational Database Management Systems (RDBMS)**
    - Introduction to SQL and its variations
    - Data types, operators, and expressions in SQL
    - Creating and managing databases and tables
4. **SQL Querying and Data Manipulation**
    - SELECT, INSERT, UPDATE, DELETE statements
    - JOINS, subqueries, and unions
    - Aggregation and grouping
5. **Database Implementation with a DBMS**
    - Choosing a DBMS and setting up the environment
    - Creating database schema and objects
    - Indexing and optimization techniques
6. **Database Security**
    - Access control and permissions
    - Encryption and data masking
    - Security best practices
7. **Database Performance Optimization**
    - Query optimization strategies
    - Indexing and query execution plans
    - Performance monitoring and profiling
8. **Backup, Recovery, and Maintenance**

- Backup types and strategies
- Recovery mechanisms and procedures
- Database maintenance and monitoring

9. **Database Development Project**
    - Designing and implementing a database project
    - Applying best practices and principles
    - Presenting and documenting the project **Assessment**

- Continuous Assessment 40%
- Final Examinations 60%

Prescribed Textbooks

Connolly, T. M., Begg, C. E. (2014). Database Systems: A Practical Approach to Design, Implementation, and Management. Netherlands: Pearson.

Garcia-Molina, H., Ullman, J. D., Widom, J. (2013). Database Systems: The Complete Book. United Kingdom: Pearson.

Elmasri, R., Navathe, S. (2017). Fundamentals of Database Systems. India: Pearson/Addison Wesley.

Hoffer, J. A., Ramesh, V., Topi, H. (2016). Modern Database Management. United Kingdom: Pearson.

Silberschatz. A, Korth, H.F. Korth, and Sudarshan. S (2020). Database System Concepts. 7th edition.

Recommended Text books

Coronel, C., Morris, S. (2014). Database Systems: Design, Implementation, & Management. United States: Cengage Learning.

Foster, E. C., Godbole, S. (2014). Database Systems: A Pragmatic Approach. United States: A press.

Ramakrishnan R and Gehrke, J. (2002). Database Management Systems. McGraw Hill

Sciore, E. (2020). Database Design and Implementation: Second Edition. Germany: Springer International Publishing

Web References:

https://www.javatpoint.com/mysql-tutorial https://dev.mysql.com/doc/mysql-tutorial-excerpt/8.0/en/tutorial.html

---

TOPIC: FUNCTIONAL DEPENDENCIES AND NORMALISATION

FUNCTIONAL DEPENDENCIES EXERCISE

A *functional dependency* (FD) is a relationship between two attributes, typically between the PK and other non-key attributes within a table. For any relation R, attribute Y is functionally dependent on attribute X (usually the PK), if for every valid instance of X, that value of X uniquely determines the value of Y. This relationship is indicated by the representation below :

X ———–> Y

The left side of the above FD diagram is called the *determinant*, and the right side is the *dependent*. Here are a few examples.

In the first example, below, SIN determines Name, Address and Birthdate. Given SIN, we can determine any of the other attributes within the table.

SIN ———-> Name, Address, Birthdate

For the second example, SIN and Course determine the date completed (Date Completed). This must also work for a composite PK.

SIN, Course ———> Date Completed

Source: https://opentextbc.ca/dbdesign01/chapter/chapter-11-functional-dependencies/

Example 1

Consider a scenario with a simple relational schema representing students and their courses. Here's the schema:

```

Student (Student ID, Name, Major)

Course (Course ID, Title, Instructor)

Enrollment (Student ID, Course ID, Grade)

```

Identify some functional dependencies based on the attributes in these relations:

1. Student ID → Name, Major: Each student ID uniquely determines the name and major of a student.
2. Course ID → Title, Instructor: Each course ID uniquely determines the title and instructor of a course.
3. (Student ID, Course ID) → Grade: The combination of student ID and course ID uniquely determines the grade a student receives in a course.

You can further explore more complex scenarios or constraints based on specific business rules or requirements. For instance, if there's a rule that a student can only be enrolled in a course once, you could express that as a functional dependency between Student ID and Course ID in the Enrollment relation.

Normalization

Some of the most difficult decisions that you face as a database developer are what tables to create and what columns to place in each table, as well as how to relate the tables that you create.

Normalization is the process of applying a series of rules to ensure that your database achieves optimal structure. Normal forms are a progression of these rules. Each successive normal form achieves a better database design than the previous form did. Although we discussed several levels of normal forms in class this lab focuses on 1st Normal Form (1NF), and Boyce-Codd Normal Form (BCNF).

If you do not understand functional dependencies, then review the discussion on functional dependencies.

Example 2

Consider a relation `Student` with the following attributes: `Student ID`, `Name`, `Age`, `Grade`, and `School`.

1. Identify the functional dependencies present in the `Student` relation.
2. Determine if the relation is in a normalized form, and if not, normalize it to eliminate redundancy and dependency.

Let's start by identifying the functional dependencies:

1. `Student ID → Name`: Each `Student ID` uniquely determines the `Name` of the student.
2. `Student ID → Age`: Each `Student ID` uniquely determines the `Age` of the student.
3. `Student ID → Grade`: Each `Student ID` uniquely determines the `Grade` of the student.
4. `Student ID → School`: Each `Student ID` uniquely determines the `School` of the student.
5. `Name → Student ID`: Each `Name` uniquely determines the `Student ID`. (Assuming unique names)
6. `Name → Age`: Each `Name` uniquely determines the `Age`. (Assuming names are unique)
7. `Name → Grade`: Each `Name` uniquely determines the `Grade`. (Assuming names are unique)
8. `Name → School`: Each `Name` uniquely determines the `School`. (Assuming names are unique)

Now, let's analyze whether the relation is in a normalized form:

- The relation is in 1st Normal Form (1NF) because it has atomic values in each cell of the table.
- The relation is in 2nd Normal Form (2NF) because there are no partial dependencies.
  Each non-prime attribute is fully functionally dependent on the candidate keys.
- The relation is in 3rd Normal Form (3NF) because there are no transitive dependencies.
  All non-prime attributes are directly dependent on the candidate keys.

Therefore, the relation `Student` is already in 3rd Normal Form (3NF), and there is no further normalization required.

This exercise helps in understanding how to identify functional dependencies in a relation and determine the normalization level of a database schema.

Lab Exercise 1:

Consider the Students table, with the primary key underlined, and the following data:

Students:

Alpha	**Name**	**Email**	**Courses**	**Grade Points**
100111	John Doe	doe@usna.edu	NN204, SI204, IT221	2,3,3
092244	Matt Smith	smith@usna.edu	SM223, EE301	4,4
113221	Melinda Black	black@usna.edu	SI204	3
090112	Tom Johnson	Johnson@usna.edu	NN204, SI204, IT221	4,2,3

a) Is the Students table in 1NF? Why?

b) If the Students table is not in 1NF, redesign the tables such that all the information currently in the students table is found in the resulting tables, and the resulting tables are in 1NF. For each of the resulting tables, give the table name, column names, primary keys, and foreign keys.

Lab Exercise 2

For a table to be in Boyce-Codd normal form, the table must be in 1NF and the determinants of all the functional dependencies in that table must be candidate keys (either primary key or alternate key). Below is the Rentals table created for the DVD-by-mail division of Neatflix.

Rentals:

Rental ID	**Title**	**Customer ID**	**Mailed Out Date**	**Director**	**Movie Category**	**Price**
1	Die Hard	1001	3/3/2010	John McTiernan	Old	$4.25
1	The last man standing	1001	3/3/2010	Walter Hill	Old	$4.25

1	Wedding Crashers	1001	3/3/2010	David Dobkin	New	$5.50
2	Dodgeball	1002	3/4/2010	Rawson Marshall Thurber	New	$5.50
2	Die Hard	1002	3/4/2010	John McTiernan	Old	$4.25
3	As good as it gets	1003	1/7/2011	James Brooks	Old	$4.25
4	Forest Gump	1001	1/7/2011	Robert Zemeckis	Old	$4.25

The primary key of the Rentals table is the composite key (**Rental ID, Title)**.

a) Explain the conditions under which the following functional dependency is true: Rental ID -->Customer ID

b) Based on the sample data on the table, is the functional dependency Rental ID --> Customer ID true?

c) Explain the conditions under which the following functional dependency is true: Director --> Title

d) Based on the sample data on the table, is the functional dependency Director --> Title true?

e) Based on your general knowledge of movies and rentals, is the functional dependency Director --> Title true?

f) Write a functional dependency that expresses the fact that the cost of all movies in a given category is the same.

g) We discussed ***insertion anomalies, deletion anomalies and update anomalies*** as examples of problems that can appear in tables that are not normalized. The following is an example of an insertion anomaly in the Rentals table: if we want to create a new category of movies, “Must See”, there is no way to store the price of this type of movie

in the database, until someone rents a movie in this category, and the rental information is recorded into the Rentals table. Give one example of a deletion anomaly in the Rentals table.

h) State what you believe are reasonable functional dependencies for the Rentals table for a DVD-by-mail business (include the functional dependencies from points a) to f) that you believe are/should be true).

i) Given your answer above, decompose the Rentals table such that the resulting tables are in BCNF. For each of the resulting tables, give the table name, column names, primary keys, and foreign keys.

Lab Exercise 3:

Suppose we have the following Courses table with columns Course ID, Instructor, Book that stores the courses, the instructor teaching the course, and the recommended books for the course. The book(s) recommended for a course does not depend on the teacher teaching the course, just on the course. Here is an example of instantiation for this table:

Courses:

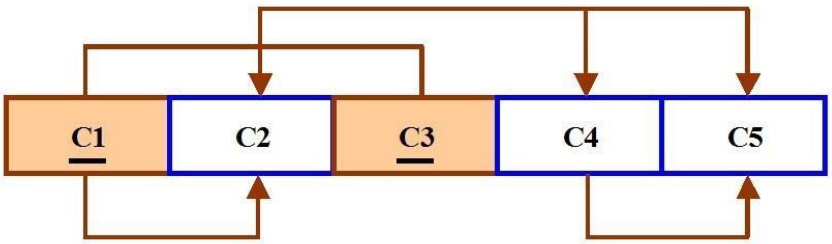
Course ID	Instructor	Book
IT360	Crainiceanu	Kroenke
IT360	Crainiceanu	Welling
IT360	DeLooze	Kroenke
IT360	DeLooze	Welling
SI440	Crainiceanu	Kroenke
SI440	Crainiceanu	Ramakrishnan
SI440	Crainiceanu	Stonebraker

a) Give an example of a multivalued dependency in the Courses table.

b) Is the Courses table in 4NF? If answer to yes, say why. If not, decompose the table such that the resulting tables are in 4^{th} normal form. For each of the resulting tables, give the table name, column names, primary keys, and foreign keys.

Attempt the following supplementary exercises:

1. What is normalization?
2. When is a table in 1NF?
3. When is a table in 2NF?
4. When is a table in 3NF?
5. Identify and discuss each of the indicated dependencies in the dependency diagram shown below:



6. To keep track of students and courses, a new college uses the table structure in the dependency diagram for this table.

Attribute Name	Sample Value	Sample Value	Sample Value
StudentID	1	2	3
StudentName	John Smith	Sandy Law	Sue Rogers
CourseID	2	2	3
CourseName	Programming Level 1	Programming Level 1	Business
Grade	75%	61%	81%
CourseDate	Jan 5th, 2014	Jan 5th, 2014	Jan 7th, 2014

7. Using the dependency diagram you just drew, show the tables (in their third normal form) you would create to fix the problems you encountered. Draw the dependency diagram for the fixed table.
8. An agency called Instant Cover supplies part-time/temporary staff to hotels in Scotland. The table lists the time spent by agency staff working at various hotels. The national insurance number (NIN) is unique for every member of staff. Use the table to answer questions (a) and (b).

NIN	ContractNo	Hours	eName	hNo	hLoc
1135	C1024	16	Smith J.	H25	East Killbride
1057	C1024	24	Hocine D.	H25	East Killbride
1068	C1025	28	White T.	H4	Glasgow
1135	C1025	15	Smith J.	H4	Glasgow

a) This table is susceptible to update anomalies. Provide examples of insertion, deletion and update anomalies.

b) Normalize this table to third normal form. State any assumptions.

9. Fill in the blanks:

a) _____ produces a lower normal form.

b) Any attribute whose value determines other values within a row is called a(n) _____.

c) An attribute that cannot be further divided is said to display _____.

d) _____ refers to the level of detail represented by the values stored in a table's row.

e) A relational table must not contain _____ groups.

https://opentextbc.ca/dbdesign01/chapter/chapter-12-normalization/

Lab Exercise 4:

Grade Report

StudentID	StudentName	CampusAddress	Major	CourseID	CourseTitle	Instructor Name	Instructor Location	Grade
168300458	Williams	208 Brooks	IS	IS 350	Database Mgt	Codd	B 104	A
168300458	Williams	208 Brooks	IS	IS 465	Systems Analysis	Parsons	B 317	B
543291073	Baker	104 Phillips	Acctg	IS 350	Database Mgt	Codd	B 104	C
543291073	Baker	104 Phillips	Acctg	Acct 201	Fund Acctg	Miller	H 310	B
543291073	Baker	104 Phillips	Acctg	Mkgt 300	Intro Mktg	Bennett	B 212	A

The Table shows a relation called GRADE REPORT for a university.

a. Draw a relational schema and diagram the functional dependencies in the relation.

b. In what normal form is this relation?

c. Decompose GRADE REPORT into a set of 3NF relations.

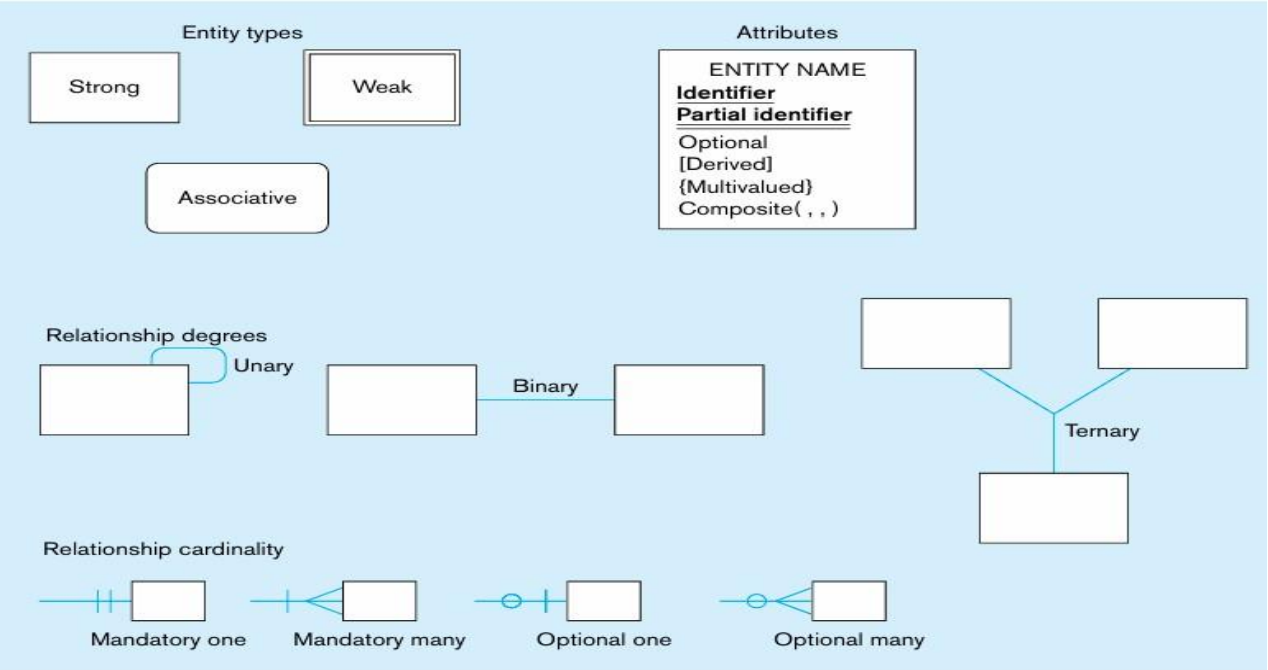
d. Draw a relational schema for your 3NF relations and show the referential integrity constraints.

e. Draw your answer to part d using Microsoft Visio (or any other appropriate tool)

TOPIC: MODELLING DATA

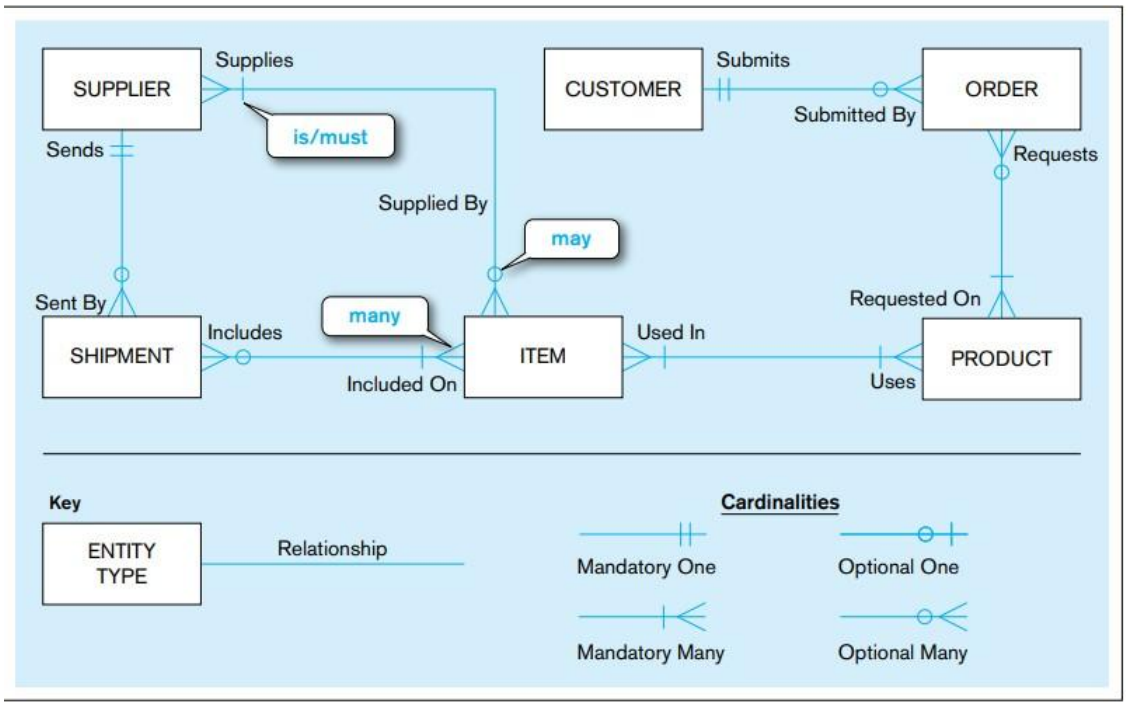
An entity-relationship model (e-r model) is a detailed, logical representation of the data for an organization or for a business area. The E-R model is expressed in terms of entities in the business environment, the relationships (or associations) among those entities, and the attributes (or properties) of both the entities and their relationships.

The notation we use for E-R diagrams is shown in Figure below and combines most of the desirable features of the different notations that are commonly used in E-R drawing tools today and also allows us to model accurately most situations that are encountered in practice.



Sample E-R diagram

The Figure below presents a simplified E-R diagram for a small furniture manufacturing company, Pine Valley Furniture Company.



A number of suppliers supply and ship different items to Pine Valley Furniture. The items are assembled into products that are sold to customers who order the products. Each customer

order may include one or more lines corresponding to the products appearing on that order. The diagram shows the entities and relationships for this company. (Attributes are omitted to simplify the diagram for now.) Entities (the objects of the organization) are represented by the rectangle symbol, whereas relationships between entities are represented by lines connecting the related entities. The entities are as follows:

CUSTOMER	A person or an organization that has ordered or might order products. *Example*: L. L. Fish Furniture.
PRODUCT	A type of furniture made by Pine Valley Furniture that may be ordered by customers. Note that a product is not a specific bookcase, because individual bookcases do not need to be tracked. *Example*: A 6-foot, 5-shelf, oak bookcase called O600.
ORDER	The transaction associated with the sale of one or more products to a customer and identified by a transaction number from sales or accounting. *Example*: The event of L. L. Fish buying one product O600 and four products O623 on September 10, 2015.
ITEM	A type of component that goes into making one or more products and can be supplied by one or more suppliers. *Example*: A 4-inch ball-bearing caster called I-27–4375.
SUPPLIER	Another company that may provide items to Pine Valley Furniture. *Example*: Sure Fasteners, Inc.
SHIPMENT	The transaction associated with items received in the same package by Pine Valley Furniture from a supplier. All items in a shipment appear on one bill-of-lading document. *Example*: The receipt of 300 I-27-4375 and 200 I-27-4380 items from Sure Fasteners, Inc., on September 9, 2015.

Note: it is important to clearly define, as metadata, each entity. For example, it is important to know that the CUSTOMER entity includes persons or organizations that have not yet purchased products from Pine Valley Furniture. It is common for different departments in an organization to have different meanings for the same term. For example, Accounting may designate as customers only those persons or organizations that have ever made a purchase, thus excluding potential customers, whereas Marketing designates as customers anyone they have contacted or who has purchased from Pine Valley Furniture or any known competitor. An accurate and thorough ERD without clear metadata may be interpreted in different ways by different people. The symbols at the end of each line on an ERD specify relationship **cardinalities**, which represent how many entities of one kind relate to how many entities of another kind. On

examining the diagram, we can see that these cardinality symbols express the following business rules:

1. A SUPPLIER may supply many ITEMs (by “may supply,” we mean the supplier may not supply any items). Each ITEM is supplied by any number of SUPPLIERs (by “is supplied,” we mean that the item must be supplied by at least one supplier).

2. Each ITEM must be used in the assembly of at least one PRODUCT and may be used in many products. Conversely, each PRODUCT must use one or more ITEMs.

3. A SUPPLIER may send many SHIPMENTs. However, each shipment must be sent by exactly one SUPPLIER. Notice that sends and supplies are separate concepts. A SUPPLIER may be able to supply an item but may not yet have sent any shipments of that item.

4. A SHIPMENT must include one (or more) ITEMs. An ITEM may be included on several SHIPMENTs.

5. A CUSTOMER may submit any number of ORDERs. However, each ORDER must be submitted by exactly one CUSTOMER. Given that a CUSTOMER may not have submitted any ORDERs, some CUSTOMERs must be potential, inactive, or some other customer possibly without any related ORDERs.

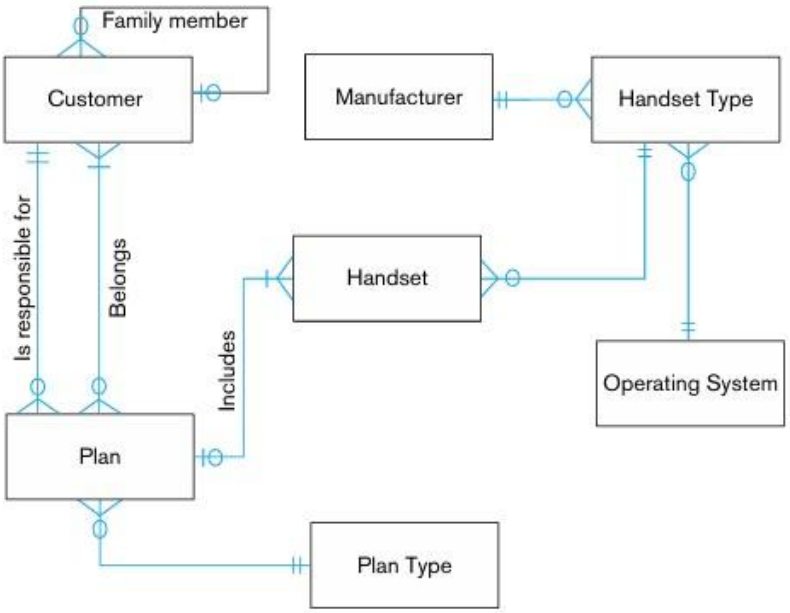
6. An ORDER must request one (or more) PRODUCTs. A given PRODUCT may not be requested on any ORDER or may be requested on one or more orders. There are actually two business rules for each relationship, one for each direction from one entity to the other.

ERD Lab Exercise

Attempt to draw the Entity Relationship Diagrams using a software tool that you are familiar with.

The entity type TEACHER has the following attributes: Teacher Name, Email, Phone, Age, Trainer, and Training Count. Trainer represents some workshop organized by the teacher, and Training Count represents the number of times teacher has organized such workshops. This implies a teacher may organize more than one workshop.

1. Draw an ERD for this situation. What attribute or attributes did you designate as the identifier for the TEACHER entity? Why?
2. A cellular operator needs a database to keep track of its customers, their subscription plans, and the handsets (mobile phones) that they are using.



The E-R diagram in the Figure above illustrates the key entities of interest to the operator and the relationships between them. Based on the figure, answer the following questions and explain the rationale for your response.

For each question, identify the element(s) in the E-R diagram that you used to determine your answer.

a. Can a customer have an unlimited number of plans?

b. Can a customer exist without a plan?

c. Is it possible to create a plan without knowing who the customer is?

d. Does the operator want to limit the types of hand sets that can be linked to a specific plan type?

e. Is it possible to maintain data regarding a hand set without connecting it to a plan?

f. Can a handset be associated with multiple plans?

g. Assume a handset type exists that can utilize multiple operating systems. Could this situation be accommodated within the model included in Figure shown above? h. Is the company able to track a manufacturer without maintaining information about its handsets?

i. Can the same operating system be used on multiple handset types?

j. There are two relationships between Customer and Plan. Explain how they differ.

k. Characterize the degree and the cardinalities of the relationship that connects Customer to itself. Explain its meaning.

l. Is it possible to link a handset to a specific customer in a plan with multiple customers?

m. Can the company track a handset without identifying its operating system?

MySQL LAB WORK

Start the MySQL command line client by typing:

- mysql -u root –p
- This command runs the MySQL client with the root user.
- The -p flag indicates that you want to provide a password. Press Enter after typing the command
- When prompted, enter the MySQL root password. By default, the root password is blank (no password), so you can just press Enter.
- If you have set a root password during the installation of XAMPP or at any later point, enter that password.

1. PART ONE

MySQL - INSERT STATEMENT

The INSERT statement is used to add data to a table. The INSERT INTO command inserts one or multiple rows into a MySQL table. Depending on the number of rows you want to add, the syntax slightly differs.

MySQL INSERT INTO – General Syntax

When inserting a **single** row into the MySQL table, the syntax is as follows:

INSERT INTO table_name (column_1, column_2, column_3)
VALUES (value_1, value_2, value_3);
In the INSERT INTO query, you should specify the following information:

- table_name: A MySQL table to which you want to add a new row.
- (column_1, column_2, column_3): A list of columns the new row will contain. The columns are separated by a comma in round brackets.
- (value_1, value_2, value_3): A list of values separated by a comma in round brackets that will be used for each corresponding column.

The number of columns and values should match; otherwise, it fails.

When inserting **multiple** rows into a MySQL table, the syntax is as follows:

INSERT INTO table name (column_1, column_2, column_3, ...)
VALUES
(value_list_1),
(value_list_2),
...
(value_list_n);
Here you should enter the following information:

- Table name: A table into which you want to insert data.
- column_1, column_2, column_3: A list of columns to store data. The columns are separated by a comma in round brackets.
- value_list_1/2/.../n: A list of values to be inserted in one row. The values are separated by a comma in round brackets. The number of values in each list should match the number of columns specified in (column_1, column_2, column_3, ...).

In addition, rows are separated by commas in the VALUES clause.

MySQL INSERT Examples

Let's explore some examples of using a MySQL INSERT statement when performing data-related tasks.
To start with, create three tables: *Customers*, *Products01*, and *Smartphones* by executing the following script:

```
CREATE TABLE Customers
(
  ID int,
  Age int,
  FirstName varchar (20),
  LastName varchar (20)
);

-- Creating the Products table

CREATE TABLE Products01
(
  ID int,
  ProductName varchar (30) NOT NULL,
  Manufacturer varchar (20) NOT NULL,
  Price decimal NOT NULL,
  ProductCount int DEFAULT 0
);

CREATE TABLE Smartphones
(
  ID int,
  ProductName varchar (30) NOT NULL,
  Manufacturer varchar (20) NOT NULL,
  Price decimal NOT NULL,
  ProductCount int DEFAULT 0
);
```

MySQL INSERT command used to insert specified values

This command can be used when it is necessary to add values and columns for the specified rows. For example, add data only for the **FirstName** and **LastName** columns of the *Customers* table.

```
INSERT INTO Customers (ID, FirstName, LastName)
VALUES ('1', 'User', 'Test');
```

The command inserts values in the **FirstName** and **LastName** columns of the C*ustomers* MySQL table. Since the **Age** column uses Null as a default value, MySQL inserts Null into the **Age** column if you do not specify explicitly its value in the INSERT statement.

MySQL INSERT command used to insert values to all columns

This command can be used in order to add values to all columns. For example, add values to the *Product01* table. Keep in mind that the order of values should match the order of columns in the table.

```
INSERT INTO Products01
VALUES ('01', 'iPhoneX', 'Apple', 35000, 3);
```

The statement inserts the specified values into all columns of the *Product01* table.

MySQL INSERT command used to insert the default value

This command can be used when the default value is specified for the column. For example, the default value for the **Product Count** column of the *Products01* table equals 0. To insert the default value, execute either of the following queries:

```
INSERT INTO Products01(ProductName, Manufacturer, Price, Product Count)
VALUES ('Nokia 9', 'HD Global', '41000', DEFAULT);
INSERT INTO Products01 (ProductName, Manufacturer, Price, Product Count)
VALUES ('Nokia 9', 'HD Global', '41000', NULL);
```

The statement inserts the default value for the **Product Count** column.

MySQL INSERT used to insert a specific value

To insert values into the columns, we can use the SET clause instead of the VALUES clause.

INSERT INTO Customers SET ID=2, FirstName='User2';

In the output, the statement inserts values into the columns based on the explicitly specified values in the SET clause.

MySQL INSERT used to insert data from another table

Using the SELECT clause in the MySQL INSERT INTO statement allows inserting the rows generated by the SELECT statement into the target MySQL table.

INSERT INTO Products01 SELECT * FROM Smartphones;

In this case, the INSERT INTO statement copies the rows you retrieved with the SELECT statement from the *Smartphones* table and inserts them into the *Products01* table.

If you want to retrieve specific rows based on the condition, apply filtering options in the WHERE clause. Keep in mind that the structure of both tables should match; otherwise, it won't work.

INSERT INTO Products01 SELECT * FROM Smartphones WHERE Price=34000;

This command adds to the *Products01* table values from the **Smartphones** table where the price equals 34000.

You can also copy and insert specified columns from another table. However, in this case, make sure that there is no record with the specified name in the target table; otherwise, it fails, and the error occurs.

MySQL INSERT – ignore errors when inserting data

In case you face errors or any unexpected behaviour while executing the insertion of rows into the MySQL table, the statement execution is interrupted. Rows with valid and invalid values are not added to the table. For example, execute the following query:

INSERT INTO Products01

VALUES ('01', 'iPhoneX', 'Apple', 35000, 3);

This statement returns the error because the record with this ID already exists. Now, modify the statement with IGNORE. It allows you to insert rows with valid values but overpass the rows with invalid values.

```
INSERT IGNORE INTO Products01
VALUES ('01', 'iPhoneX', 'Apple', 35000, 3);
```

As a result, the script was executed without errors. MySQL will notify you that rows were added to the table.

Class Exercise

1. Create 3 tables Customer, Products01 and Smartphones
2. Enter data into the tables using the examples given above
3. Practice inserting data from Smartphones table into Products01 table

---

2. PART TWO

MySQL - SELECT STATEMENT

The primary task in database management is writing and executing queries – commands for retrieving and manipulating data in databases. Users describe the tasks via these queries, and the database management system performs all the physical operations.

Among the most widely used MySQL queries, one would certainly name CREATE, UPDATE, and DELETE. However, when we need to apply any changes to the database, we first must define the "target". For that, we turn to the SELECT statement, which is one of the MySQL operation pillars.

MySQL SELECT statement queries the database according to the criteria set by the operator and returns the rows/columns that match those criteria. With the help of this MySQL query data method, database administrators can retrieve, group, summarize and analyze data. An absolute majority of all MySQL queries begin with the SELECT query.

MySQL SELECT Statement Syntax

The SELECT query follows the standard syntax:
SELECT column_to_select FROM table_to_select WHERE conditions;

Or, you may see the following format:
SELECT column_to_select
FROM table_to_select
WHERE conditions;

It does not matter if you write this query in one line or column, with each keyword starting a new line. It is a matter of personal preferences and convenience only. Many specialists prefer the “column” format because it makes the query easier to read and understand

When MySQL evaluates the SELECT command, it always starts from evaluating the FROM clause and then proceeds to the SELECT clause.

At the end of the statement, you have to put a semicolon (;) as the statement delimiter in MySQL. Often, queries consist of several commands. Then, you separate them by semicolons, thus telling MySQL to execute all those commands individually.

Note: SQL is not case-sensitive, so you may write the entire query in low case. The code will work correctly. Writing clauses keywords in upper case is a formatting method to improve the code readability for users.

Besides the FROM and WHERE clauses, the MySQL SELECT query can include other optional clauses:

- **GROUP BY** organizes the retrieved data by grouping them by columns.
- **HAVING** relates to GROUP BY. When this clause is applied, it selects and returns only those rows having the specified conditions TRUE.
- **ORDER BY** sorts the records in the returned result-set.

- **DISTINCT** removes duplicates from the result-set.
- **LIMIT** controls the number of rows retrieved by the query.
- **ALL** returns all matching rows.
- **HIGH_PRIORITY** determines that MySQL must execute the SELECT statement before all the UPDATE operators waiting for the same resource.

SELECT * – "asterisk" – is a replacement for the ALL clause and quite a popular method. However, it also has its catches.

MySQL SELECT * returns all data even from those columns that you don't use and don't need for this query. This may overload the network traffic and produce unnecessary I/O.

MySQL – SELECT Examples

The simplest case of using the MySQL SELECT command is **retrieving all data in a table according to a particular criterion**. Have a look at the following example:

```
SELECT *
FROM book_prices
WHERE quantity >= 15;
```

We use the "**asterisk**" option to get **ALL** fields from the *book_prices* table with the quantity greater or equal to 15.

If we want to **select specific fields from the table**, we must specify them in the SELECT statements:

```
SELECT book_title, quantity, author_name, book_price
FROM books
WHERE price < 15
ORDER BY price ASC;
```

This command returns the data on the book titles, quantities of items, and names of the authors for the books with a price less than $15. The results will be present in ascending order.

MySQL SELECT INTO – a specific case of usage

As mentioned at the beginning of this article, using the SELECT clause in MySQL does not affect the database like changing the stored data. However, there is a case when using this command has a more substantial impact.

The **SELECT … INTO** statement serves to create new tables and populate tables with data.

The standard syntax of this MySQL statement is as follows:

```
SELECT column1, column2, column3, ...
INTO table1
FROM table2
WHERE condition;
```

In this statement, note the following components:

- ***The list of columns*** after the SELECT clause in the MySQL query specifies which columns we want to take and insert into a new table.
- ***A table1*** should be the name of a table that we'll create and populate with the data from columns after the SELECT keyword. Note that we won't replace any existing table with this method – we create a new table with a unique name.
- ***A table2*** stands for the "source" table where we'll take the data to insert into the new table. Also, with the help of JOINs, data from multiple tables can be retrieved and inserted into the new table.
- ***WHERE*** is an optional condition for filtering retrieved records.

To **retrieve all columns from a table and copy them into a new one**, use the following syntax:

```
SELECT *
INTO table1
FROM table2
WHERE condition;
```

To **retrieve some columns from the table and copy them into a new table**, use the following syntax:

```
SELECT column1, column2, column3, ...
INTO table1
FROM table2
WHERE condition;
```

SELECT … INTO OUTFILE writes the data to a file in a specific output format. This method suggests using columns and line terminators to ensure the specified format. Note: You need the FILE privilege to use this command, as it creates a file on the server host.

SELECT … INTO DUMPFILE writes a single row to a file without using any formatting.

A standard example of the SELECT … INTO syntax is as follows:

```
SELECT order_id, unit_price, customer_id
FROM order_details
WHERE quantity < 200
ORDER BY customer_id
INTO OUTFILE 'orders.txt'
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n';
```

This query selects particular data from the table containing details of orders – it returns the data on the orders, prices, and customers' IDs where quantity is less than 200. The results sorted by the customers' IDs will be stored in a new file called orders.txt.

Source: https://blog.devart.com/mysql-select-statement-basics.html

Class Exercise

Sample table: employees

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	PHONE_NUMBER	EMAIL	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID

1. Enter relevant data into the table.
2. Write a query to display (employee_id, first_name, last_name and department_id)
3. Write a query to display the names (first_name, last_name) using **alias name** "First Name", "Last Name".
4. Write a query to get all employee details from the employee table order by first name, in descending order.
5. Write a query to get the total salaries payable to employees
6. Write a query to get the maximum and minimum salary from employees' table
7. Write a query to get the average salary and number of employees in the employees' table
8. Write a query to get the number of employees working with the company.
9. Write a query turn all first name from employees table into upper case.
10. Write a query to get the names (first_name, last_name), salary, NAPSA of all the employees (NAPSA is calculated as 10% of salary).

3. PART THREE

MySQL SELECT query with JOIN

In most cases, databases contain multiple tables with different data. In your work, you often need to retrieve the data from several tables simultaneously. The SELECT query in MySQL offers two options. The first one is to define which tables the command should refer to. You specify the column names after the FROM clause and separate them by commas. The second option is to use the JOIN clause.

JOIN combines several tables to retrieve their data with a single query. When you include JOIN into the MySQL SELECT statement, the syntax is the following:

```
SELECT table1.column1, table2.column2
FROM table1
JOIN table2
ON table1.related_column=table2.related_column;
```

Assume we want to know the names of certain customers and which orders they made. Have a look at the code:

```
SELECT order_details.order_id, customers.customer_name
FROM customers
INNER JOIN order_details
ON customers.customer_id = order_details.customer_id;
```

The query joins the tables containing the order details with the table containing the customers' names. The condition determining which data to give is the *customer_id* value. This SELECT command will return the rows from the two joined tables where the *customer_id* values match.

Use **JOIN to copy the data from several tables into a new table**:

```
SELECT Customers.CustomerName, Orders.OrderID
INTO CustomersOrderBackup2021
FROM Customers
```

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

A specific case of using the SELECT … INTO command is writing the selected data into files. It helps to save some data separately for further reference. Also, this statement works as a part of Events processes, creating error logs and application event logs this way.

Class Exercise

Sample table: locations

location_id	street_address	postal_code	city	state_province	country_id
1000	1297 Via Cola di Rie	989	Roma		IT
1100	93091 Calle della Te	10934	Venice		IT
2400	8204 Arthur St		London		UK
1200	2017 Shinjuku	1689	Tokyo	Tokyo	JP
1300	9450 Kamiya-cho	6823	Hiroshima		JP
1800	147 Spadina Ave	M5V 2L7	Toronto	Ontario	CA
3200	President Ave	10101	Lusaka	Lusaka	ZM

Sample table: countries

Country id	Country	Region id
AR	ARGENTINA	2
AU	AUSTRALIA	3
BE	BELGIUM	1
BR	BRAZIL	2
CA	CANADA	2
CH	SWITZERLAND	1
CN	CHINA	3
DE	GERMANY	1
DK	DENMARK	1
EG	EGYPT	4
FR	FRANCE	1
HK	HONGKONG	3
IL	ISRAEL	4
IT	ITALY	1
JP	JAPAN	3
KW	KUWAIT	4

MX	MEXICO	2
NG	NIGERIA	4
NL	NETHERLANDS	1
SG	SINGAPORE	3
UK	UNITED KINGDOM	1
US	UNITED STATES	2
ZM	ZAMBIA	4
ZW	ZIMBABWE	4

1. Write a MySQL query to find the addresses (location_id, street_address, city, state_province, country_name) of all the departments.

   Hint: Use NATURAL JOIN.

2. From the following tables write a SQL query to find the salesperson and customer who reside in the same city. Return Salesman, cust_name and city. *Sample table*: salesman

```
salesman_id|    name    |   city    | commission
-----------+------------+-----------+-------------
      5001 | James Hoog | New York  |       0.15
      5002 | Nail Knite | Paris     |       0.13
      5005 | Pit Alex   | London    |       0.11
      5006 | Mc Lyon    | Paris     |       0.14
      5007 | Paul Adam  | Rome      |       0.13
      5003 | Lauson Hen | San Jose  |       0.12
```

Sample table: customer

```
 customer_id |   cust_name     |    city     | grade | salesman_id
-------------+-----------------+-------------+-------+-------------
        3002 | Nick Rimando    | New York    |   100 |        5001
        3007 | Brad Davis      | New York    |   200 |        5001
        3005 | Graham Zusi     | California  |   200 |        5002
        3008 | Julian Green    | London      |   300 |        5002
        3004 | Fabian Johnson  | Paris       |   300 |        5006
        3009 | Geoff Cameron   | Berlin      |   100 |        5003
        3003 | Jozy Altidor    | Moscow      |   200 |        5007
        3001 | Brad Guzan      | London      |       |        5005
```

3. From the following tables write a SQL query to find those orders where the order amount exists between 500 and 2000. Return ord_no, purch_amt, cust_name, city.

Sample table: customer

customer_id	cust_name	city	grade	salesman_id
3002	Nick Rimando	New York	100	5001
3007	Brad Davis	New York	200	5001
3005	Graham Zusi	California	200	5002
3008	Julian Green	London	300	5002
3004	Fabian Johnson	Paris	300	5006
3009	Geoff Cameron	Berlin	100	5003
3003	Jozy Altidor	Moscow	200	5007
3001	Brad Guzan	London		5005

Sample table: orders

ord_no	purch_amt	ord_date	customer_id	salesman_id
70001	150.5	2012-10-05	3005	5002
70009	270.65	2012-09-10	3001	5005
70002	65.26	2012-10-05	3002	5001
70004	110.5	2012-08-17	3009	5003
70007	948.5	2012-09-10	3005	5002
70005	2400.6	2012-07-27	3007	5001
70008	5760	2012-09-10	3002	5001
70010	1983.43	2012-10-10	3004	5006
70003	2480.4	2012-10-10	3009	5003
70012	250.45	2012-06-27	3008	5002
70011	75.29	2012-08-17	3003	5007
70013	3045.6	2012-04-25	3002	5001

4. From the following tables write a SQL query to find the salesperson(s) and the customer(s) he represents. Return Customer Name, city, Salesman, commission. *Sample table*: customer

customer_id	cust_name	city	grade	salesman_id
3002	Nick Rimando	New York	100	5001
3007	Brad Davis	New York	200	5001
3005	Graham Zusi	California	200	5002
3008	Julian Green	London	300	5002
3004	Fabian Johnson	Paris	300	5006
3009	Geoff Cameron	Berlin	100	5003
3002	ozy Altidor	Moscow	200	5007
3001	Brad Guzan	London		5005

Sample table: salesman

salesman_id	name	city	commission
5001	James Hoog	New York	0.15
5002	Nail Knite	Paris	0.13

```
5005 | Pit Alex   | London   |       0.11
5006 | Mc Lyon    | Paris    |       0.14
5007 | Paul Adam  | Rome     |       0.13
5003 | Lauson Hen | San Jose |       0.12
```

4. PART FOUR

MySQL UPDATE statement

The **UPDATE** statement updates data in a table. It allows you to change the values in one or more columns of a single row or multiple rows.

The following illustrates the basic syntax of the UPDATE statement:

```
UPDATE [LOW_PRIORITY] [IGNORE] table_name.
SET
   column_name1 = expr1,
column_name2 = expr2,
   ...
[WHERE condition];
```

In this syntax:

- First, specify the name of the table that you want to update data after the `UPDATE` keyword.
- Second, specify which column you want to update and the new value in the `SET` clause. To update values in multiple columns, you use a list of comma-separated assignments by supplying a value in each column's assignment in the form of a literal value, an expression, or a subquery.
- Third, specify which rows to be updated using a condition in the **WHERE** clause. The `WHERE` clause is optional. If you omit it, the `UPDATE` statement will modify all rows in the table.

Notice that the WHERE clause is so important that you should not forget. Sometimes, you may want to update just one row; However, you may forget the WHERE clause and accidentally update all rows of the table.

MySQL supports two modifiers in the UPDATE statement.

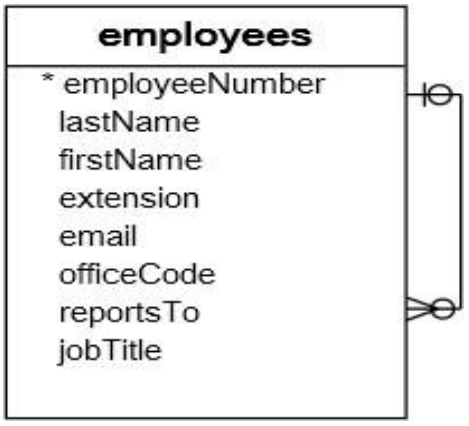
1. The `LOW_PRIORITY` modifier instructs the `UPDATE` statement to delay the update until there is no connection reading data from the table. The `LOW_PRIORITY` takes effect for the storage engines that use table-level locking only such as `MyISAM`, `MERGE`, and `MEMORY`.
2. The `IGNORE` modifier enables the `UPDATE` statement to continue updating rows even if errors occurred. The rows that cause errors such as duplicate-key conflicts are not updated.

MySQL UPDATE examples

Let's practice the UPDATE statement.

1) Using MySQL UPDATE to modify values in a single column example

See the following employees table from the sample database.



In this example, we will update the email of **Mary Patterson** to the new

email mary.patterso@classicmodelcars.com.

First, find Mary's email from the **employees** table using the following SELECT statement:

```
SELECT
    Firstname,
    lastname,
    email
FROM
    employees
WH ERE
       employeeNumber = 1056;
```

	firstname	lastname	email
▶	Mary	Patterson	mpatterso@classicmodelcars.com

Second, update the email address of `Mary` to the new email

`mary.patterson@classicmodelcars.com`:

```
UPDATE employees
SET
    email = 'mary.patterson@classicmodelcars.com' WHERE
    employee Number = 1056;
```

MySQL issued the number of rows affected:

```
1 row(s) affected
```

In this UPDATE statement:

- The **WHERE** clause specifies the row with employee number `1056` will be updated.
- The `SET` clause sets the value of the `email` column to the new email.
- Third, execute the `SELECT` statement again to verify the change:

```
SELECT
      Firstname,
      lastname,
      email
FROM
      employees
WHERE
    employeeNumber = 1056;
```

	firstname	lastname	email
▶	Mary	Patterson	mary.patterson@classicmodelcars.com

2) Using MySQL UPDATE to modify values in multiple columns

To update values in the multiple columns, you need to specify the assignments in the SET clause.

For example, the following statement updates both last name and email columns of employee number 1056:

```
UPDATE employees
SET
    lastname = 'Hill',
    email = 'mary.hill@classicmodelcars.com'
```

```
WHERE
    employeeNumber = 1056;
```

Let's verify the changes:

```
SELECT
   firstname,
   lastname,
   email
FROM
   employees
WHERE
      EmployeeNumber = 1056;
```

	firstname	lastname	email
▶	Mary	Hill	mary.hill@classicmodelcars.com

3) Using MySQL UPDATE to replace string example

The following example updates the domain parts of emails of all Sales Reps with office code 6:

```
UPDATE employees
SET email = REPLACE (email, '@classicmodelcars.com', '@mysqltutorial.org')
WHERE
   jobTitle = 'Sales Rep' AND
   officeCode = 6;
```

In this example, the `REPLACE ()` function replaces `@classicmodelcars.com` in the email column with `@mysqltutorial.org`.

Class Exercise

1. Write a SQL statement to change salary of employee to 8,000 whose ID is 105, if the existing salary is less than 5,000.
2. Write a SQL statement to change the email address of employee whose ID is 105, to lmusonda1@xyz.com
3. Write a SQL statement to change date of engagement from 01/08/24 of employee 103 to 01/08/23

Sample table: Employees

EMP_ID	FNAME	LNAME	EMAIL	PHONE No.	DATE ENG	SALARY
101	Mwansa	Chibwe	mchibwe@xyz.com	111 111	01/01/24	7,000
102	Lusubilo	Musonda	lmusonda@xyz.com	222 222	01/01/23	8,000
103	Linda	Mutale	lmutale@xyz.com	333 333	01/08/24	6,000
104	Mathew	Sakala	msakala@xyz.com	444 444	01/01/24	5,000
105	Luke	Musonda	lmusonda@xyz.com	555 555	01/02/24	3,500