Birla Institute of Technology and Science, Pilani
CS F212 Database Systems
Lab No # 1

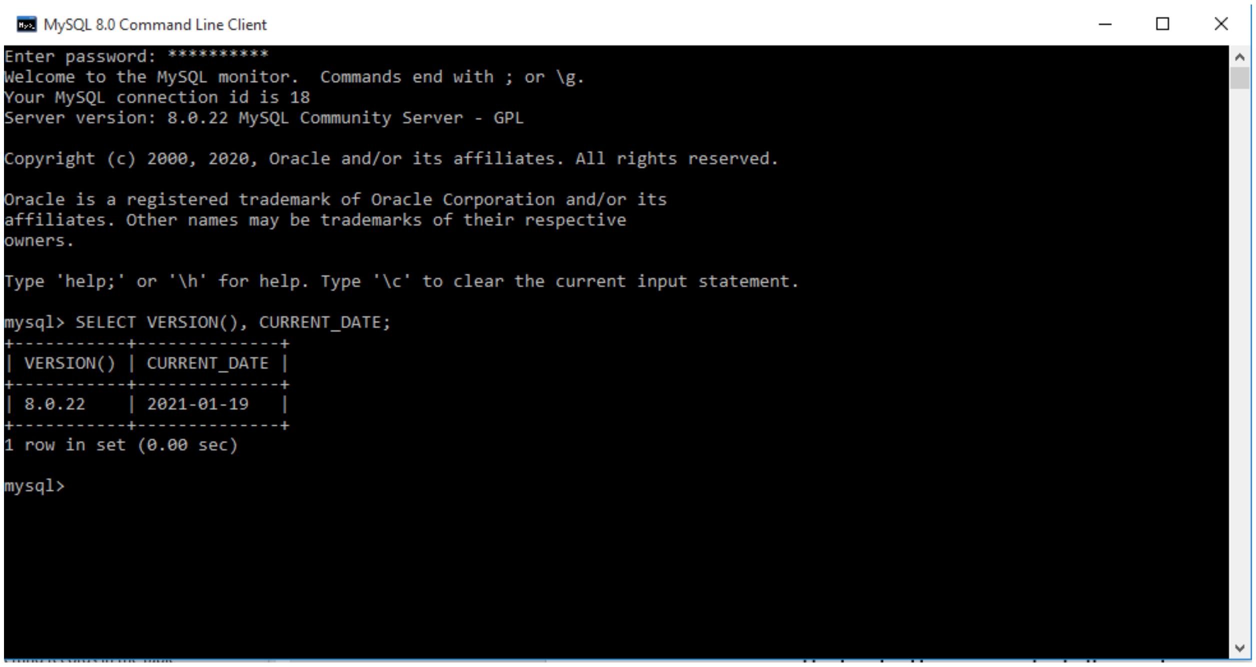
****

1. Introduction

A database is a separate application that stores a collection of data. Each database has one or more specific APIs for creating, accessing, managing, searching, and replicating the data it holds. Other kinds of data stores can also be used, such as files on the file system or large hash tables in memory, but data fetching and writing would not be so fast and easy with those type of systems. MySQL is a fast, easy-to-use RDBMS being used for many small and big businesses.

1.1 Entering Queries

You can execute queries either in MySQL Workbench or in MySQL Command Line Client. Here is a simple query that asks the server to tell you its version number and the current date.

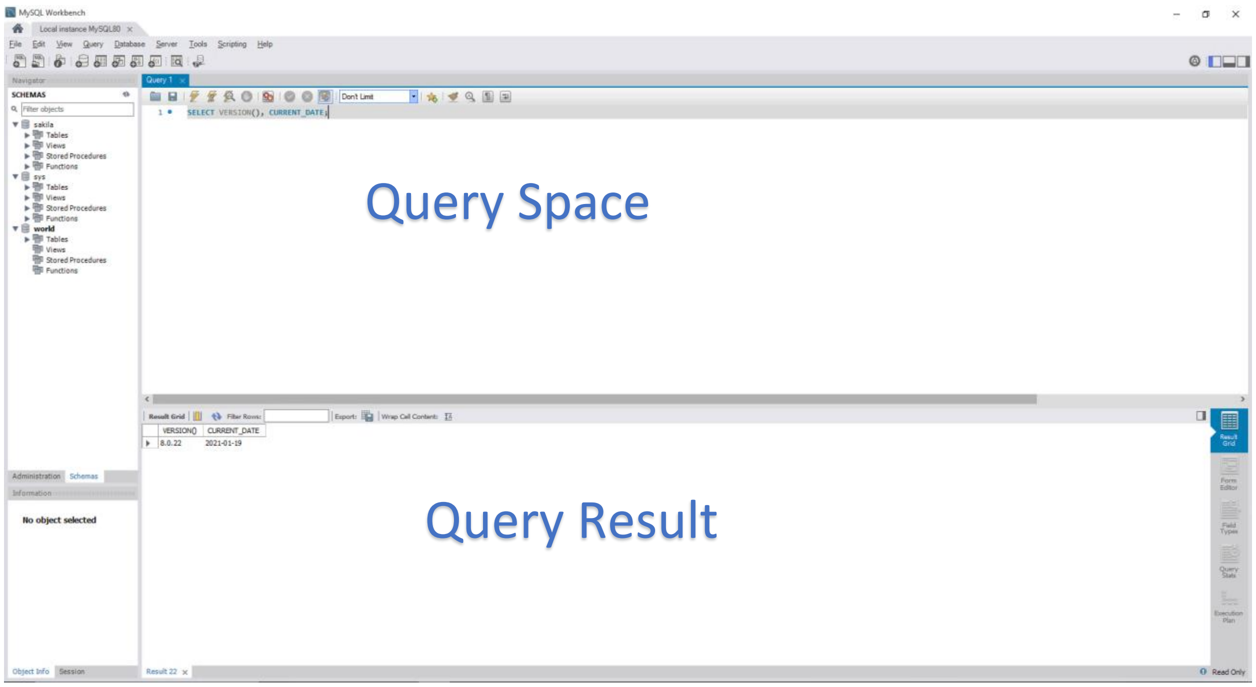


```
Enter password: **********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18
Server version: 8.0.22 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SELECT VERSION(), CURRENT_DATE;
+-----------+--------------+
| VERSION() | CURRENT_DATE |
+-----------+--------------+
| 8.0.22    | 2021-01-19   |
+-----------+--------------+
1 row in set (0.00 sec)

mysql>
```

Figure: 1 MySQL Command Line Client

Keywords may be entered in any letter-case. The following queries are equivalent:

```
mysql> SELECT VERSION(), CURRENT_DATE;
mysql> select version(), current_date;
mysql> SeLeCt vErSiOn(), current_DATE;
```



```
SELECT VERSION(), CURRENT_DATE;
```

```
VERSION()   CURRENT_DATE
8.0.22      2021-01-19
```

Figure: 2 MySQL Workbench

Here is another query. It demonstrates that you can use MySQL as a simple calculator:

```
mysql> SELECT SIN(PI()/4), (4+1)*5;
+------------------+---------+
| SIN(PI()/4)      | (4+1)*5 |
+------------------+---------+
| 0.70710678118655 |      25 |
+------------------+---------+
1 row in set (0.02 sec)
```

The queries shown thus far have been relatively short, single-line statements. You can even enter multiple statements on a single line. Just end each one with a semicolon:

```
mysql> SELECT VERSION(); SELECT NOW();
+-----------+
| VERSION() |
+-----------+
| 8.0.13    |
+-----------+
1 row in set (0.00 sec)

+---------------------+
| NOW()               |
+---------------------+
| 2018-08-24 00:56:40 |
+---------------------+
1 row in set (0.00 sec)
```

A query need not be given all on a single line, so lengthy queries that require several lines are not a problem. MySQL determines where your statement ends by looking for the terminating semicolon, not by looking for the end of the input line. In other words, MySQL accepts free-format input: it collects input lines but does not execute them until it sees the semicolon. Here is a simple multiple-line statement:

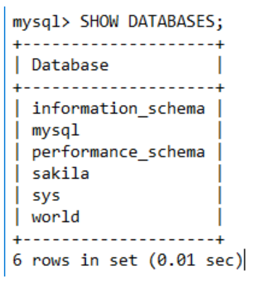
```
mysql> SELECT
    -> USER()
    -> ,
    -> CURRENT_DATE;
+---------------+--------------+
| USER()        | CURRENT_DATE |
+---------------+--------------+
| jon@localhost | 2018-08-24   |
+---------------+--------------+
```

If you decide you do not want to execute a query that you are in the process of entering, cancel it by typing \c:

```
mysql> SELECT
    -> USER()
    -> \c
mysql>
```

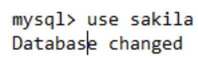
1.2 Creating and Using a Database

Use the SHOW statement to find out what databases currently exist on the server:



```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
| world              |
+--------------------+
6 rows in set (0.01 sec)
```

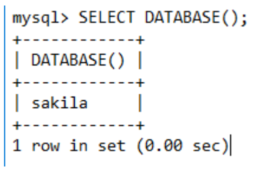
If the “sakila” database exists, try to access it:



```
mysql> use sakila
Database changed
```

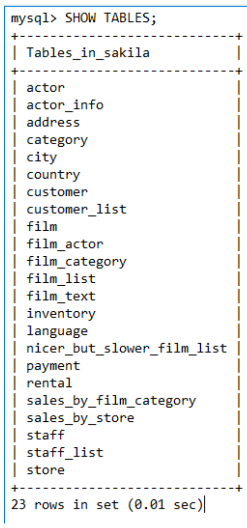
1.3 Getting Information About Databases and Tables

You have previously seen SHOW DATABASES, which lists the databases managed by the server. To find out which database is currently selected, use the DATABASE() function:



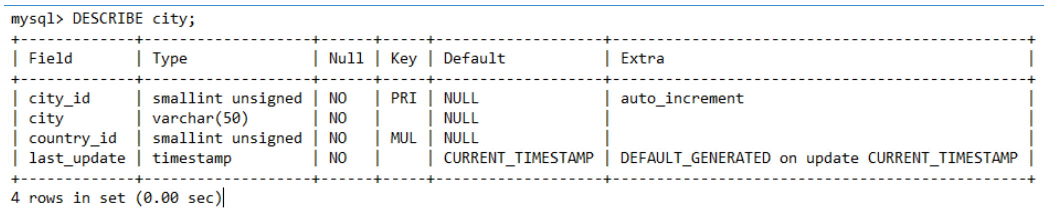
```
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| sakila     |
+------------+
1 row in set (0.00 sec)
```

To find out what tables the default database contains (for example, when you are not sure about the name of a table), use this statement:



```
mysql> SHOW TABLES;
+----------------------------+
| Tables_in_sakila           |
+----------------------------+
| actor                      |
| actor_info                 |
| address                    |
| category                   |
| city                       |
| country                    |
| customer                   |
| customer_list              |
| film                       |
| film_actor                 |
| film_category              |
| film_list                  |
| film_text                  |
| inventory                  |
| language                   |
| nicer_but_slower_film_list |
| payment                    |
| rental                     |
| sales_by_film_category     |
| sales_by_store             |
| staff                      |
| staff_list                 |
| store                      |
+----------------------------+
23 rows in set (0.01 sec)
```

If you want to find out about the structure of a table, the DESCRIBE statement is useful; it displays information about each of a table's columns:



```
mysql> DESCRIBE city;
+-------------+-------------------+------+-----+-------------------+-----------------------------------------------+
| Field       | Type              | Null | Key | Default           | Extra                                         |
+-------------+-------------------+------+-----+-------------------+-----------------------------------------------+
| city_id     | smallint unsigned | NO   | PRI | NULL              | auto_increment                                |
| city        | varchar(50)       | NO   |     | NULL              |                                               |
| country_id  | smallint unsigned | NO   | MUL | NULL              |                                               |
| last_update | timestamp         | NO   |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
+-------------+-------------------+------+-----+-------------------+-----------------------------------------------+
4 rows in set (0.00 sec)
```

Field indicates the column name, Type is the data type for the column, NULL indicates whether the column can contain NULL values, Key indicates whether the column is indexed, and Default specifies the column's default value. Extra displays special information about columns: If a column was created with the AUTO_INCREMENT option, the value is auto_increment rather than empty. DESC is a short form of DESCRIBE.

1.4 Basic Data Types in MySQL

Although there are lots of data types available but for our purpose, we shall be covering only the following:

Data Type	Format	Explanation
Number	INT	A normal-sized integer that can be signed or unsigned. If signed, the allowable range is from -2147483648 to 2147483647. If unsigned, the allowable range is from 0 to 4294967295. You can specify a width of up to 11 digits.
	FLOAT(M,D)	A floating-point number that cannot be unsigned. You can define the display length (M) and the number of decimals (D). This is not required and will default to 10,2, where 2 is the number of decimals and 10 is the total number of digits (including decimals). Decimal precision can go to 24 places for a FLOAT.
	DECIMAL(M,D)	An unpacked floating-point number that cannot be unsigned. In the unpacked decimals, each decimal corresponds to one byte. Defining the display length

		(M) and the number of decimals (D) is required. NUMERIC is a synonym for DECIMAL.
Character	CHAR(M)	A fixed-length string between 1 and 255 characters in length (for example CHAR(5)), right-padded with spaces to the specified length when stored. Defining a length is not required, but the default is 1.
	VARCHAR(M)	A variable-length string between 1 and 255 characters in length. For example, VARCHAR(25). You must define a length when creating a VARCHAR field.
Date and Time	DATE	A date in YYYY-MM-DD format, between 1000-01-01 and 9999-12-31. For example, December 30th, 1973 would be stored as 1973-12-30.
	DATETIME	A date and time combination in YYYY-MM-DD HH:MM:SS format, between 1000-01-01 00:00:00 and 9999-12-31 23:59:59. For example, 3:30 in the afternoon on December 30th, 1973 would be stored as 1973-12-30 15:30:00.
	TIMESTAMP	A timestamp between midnight, January 1st, 1970 and sometime in 2037. This looks like the previous DATETIME format, only without the hyphens between numbers; 3:30 in the afternoon on December 30th, 1973 would be stored as 19731230153000 (YYYYMMDDHHMMSS).

2. Creating Tables

CREATE TABLE creates a table with the given name. You must have the CREATE privilege for the table. By default, tables are created in the default database, using the InnoDB storage engine. An error occurs if the table exists, if there is no default database, or if the database does not exist. MySQL has no limit on the number of tables. The underlying file system may have a limit on the number of files that represent tables. Individual storage engines may impose engine-specific constraints. InnoDB permits up to 4 billion tables.

There are several aspects to the CREATE TABLE statement, described as following:

1. Table Name
    - tbl_name - The table name can be specified as db_name.tbl_name to create the table in a specific database. This works regardless of whether there is a default database, assuming that the database exists. If you use quoted identifiers, quote the database and table names separately. For example, write \`mydb\`.\`mytbl\`, not \`mydb.mytbl\`.
    - IF NOT EXISTS - Prevents an error from occurring if the table exists. However, there is no verification that the existing table has a structure identical to that indicated by the CREATE TABLE statement.
2. Temporary Tables - You can use the TEMPORARY keyword when creating a table. A TEMPORARY table is visible only within the current session and is dropped automatically when the session is closed.
3. Table Cloning and Copying
    - LIKE - Use CREATE TABLE ... LIKE to create an empty table based on the definition of another table, including any column attributes and indexes defined in the original table:

```
CREATE TABLE new_tbl LIKE orig_tbl;
```

- [AS] query_expression - To create one table from another, add a SELECT statement at the end of the CREATE TABLE statement:

```
CREATE TABLE new_tbl AS SELECT * FROM orig_tbl;
```

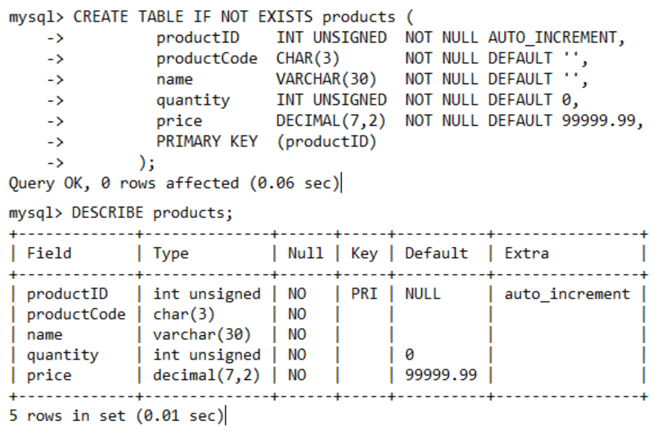
- IGNORE | REPLACE - The IGNORE and REPLACE options indicate how to handle rows that duplicate unique key values when copying a table using a SELECT statement.

4. Column Data Types and Attributes
    - data_type represents the data type in a column definition. For a full description of the syntax available for specifying column data types, as well as information about the properties of each type.
    - `CREATE TABLE t (c CHAR(20) CHARACTER SET utf8 COLLATE utf8_bin);`
    - If neither NULL nor NOT NULL is specified, the column is treated as though NULL had been specified.
    - Specifies a default value for a column. For more information about default value handling, including the case that a column definition includes no explicit DEFAULT value.
    - VISIBLE, INVISIBLE - Specify column visibility. The default is VISIBLE if neither keyword is present. A table must have at least one visible column. Attempting to make all columns invisible produces an error.
    - AUTO_INCREMENT - An integer or floating-point column can have the additional attribute AUTO_INCREMENT. When you insert a value of NULL (recommended) or 0 into an indexed AUTO_INCREMENT column, the column is set to the next sequence value. Typically this is value+1, where value is the largest value for the column currently in the table. AUTO_INCREMENT sequences begin with 1.
    - COMMENT - A comment for a column can be specified with the COMMENT option, up to 1024 characters long. The comment is displayed by the SHOW CREATE TABLE and SHOW FULL COLUMNS statements.
    - GENERATED ALWAYS - Used to specify a generated column expression.
5. Indexes, Foreign Keys, and CHECK Constraints – Described in Section 5.
6. Table Options - Table options are used to optimize the behavior of the table. In most cases, you do not have to specify any of them.
7. Table Partitioning - partition_options can be used to control partitioning of the table created with CREATE TABLE.

Create Table syntax:

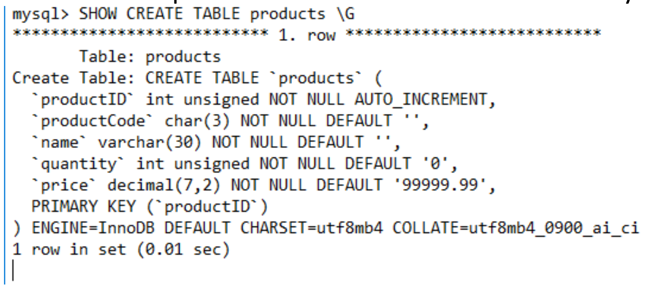
```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_options]
    [partition_options]

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(create_definition,...)]
    [table_options]
    [partition_options]
    [IGNORE | REPLACE]
    [AS] query_expression

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    { LIKE old_tbl_name | (LIKE old_tbl_name) }
```

Create Table Example:



```
mysql> CREATE TABLE IF NOT EXISTS products (
    ->          productID    INT UNSIGNED  NOT NULL AUTO_INCREMENT,
    ->          productCode  CHAR(3)       NOT NULL DEFAULT '',
    ->          name         VARCHAR(30)   NOT NULL DEFAULT '',
    ->          quantity     INT UNSIGNED  NOT NULL DEFAULT 0,
    ->          price        DECIMAL(7,2)  NOT NULL DEFAULT 99999.99,
    ->          PRIMARY KEY  (productID)
    ->        );
Query OK, 0 rows affected (0.06 sec)

mysql> DESCRIBE products;
+-------------+--------------+------+-----+----------+----------------+
| Field       | Type         | Null | Key | Default  | Extra          |
+-------------+--------------+------+-----+----------+----------------+
| productID   | int unsigned | NO   | PRI | NULL     | auto_increment |
| productCode | char(3)      | NO   |     |          |                |
| name        | varchar(30)  | NO   |     |          |                |
| quantity    | int unsigned | NO   |     | 0        |                |
| price       | decimal(7,2) | NO   |     | 99999.99 |                |
+-------------+--------------+------+-----+----------+----------------+
5 rows in set (0.01 sec)
```

Show the complete CREATE TABLE statement used by MySQL to create this table use following statement:

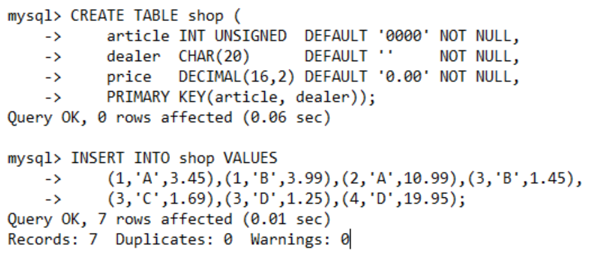


```
mysql> SHOW CREATE TABLE products \G
*************************** 1. row ***************************
       Table: products
Create Table: CREATE TABLE `products` (
  `productID` int unsigned NOT NULL AUTO_INCREMENT,
  `productCode` char(3) NOT NULL DEFAULT '',
  `name` varchar(30) NOT NULL DEFAULT '',
  `quantity` int unsigned NOT NULL DEFAULT '0',
  `price` decimal(7,2) NOT NULL DEFAULT '99999.99',
  PRIMARY KEY (`productID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.01 sec)
```

3. Examples Common Queries

Here are examples of how to solve some common problems with MySQL. To create and populate the example table, use these statements:

```
CREATE TABLE shop (
    article INT UNSIGNED  DEFAULT '0000' NOT NULL,
    dealer  CHAR(20)      DEFAULT ''     NOT NULL,
    price   DECIMAL(16,2) DEFAULT '0.00' NOT NULL,
    PRIMARY KEY(article, dealer));
INSERT INTO shop VALUES
    (1,'A',3.45),(1,'B',3.99),(2,'A',10.99),(3,'B',1.45),
    (3,'C',1.69),(3,'D',1.25),(4,'D',19.95);
```



```
mysql> CREATE TABLE shop (
    ->     article INT UNSIGNED  DEFAULT '0000' NOT NULL,
    ->     dealer  CHAR(20)      DEFAULT ''     NOT NULL,
    ->     price   DECIMAL(16,2) DEFAULT '0.00' NOT NULL,
    ->     PRIMARY KEY(article, dealer));
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO shop VALUES
    ->     (1,'A',3.45),(1,'B',3.99),(2,'A',10.99),(3,'B',1.45),
    ->     (3,'C',1.69),(3,'D',1.25),(4,'D',19.95);
Query OK, 7 rows affected (0.01 sec)
Records: 7  Duplicates: 0  Warnings: 0
```

After issuing the statements, the table should have the following contents:

```
SELECT * FROM shop ORDER BY article;
```

```
+---------+--------+-------+
| article | dealer | price |
+---------+--------+-------+
|       1 | A      |  3.45 |
|       1 | B      |  3.99 |
|       2 | A      | 10.99 |
|       3 | B      |  1.45 |
|       3 | C      |  1.69 |
|       3 | D      |  1.25 |
|       4 | D      | 19.95 |
+---------+--------+-------+
```

3.1 The Maximum Value for a Column

"What is the highest item number?"

```
SELECT MAX(article) AS article FROM shop;
```

```
+---------+
| article |
+---------+
|       4 |
+---------+
```

3.2 The Row Holding the Maximum of a Certain Column

Task: Find the number, dealer, and price of the most expensive article.

```
SELECT article, dealer, price
FROM   shop
WHERE  price=(SELECT MAX(price) FROM shop);
```

```
+---------+--------+-------+
| article | dealer | price |
+---------+--------+-------+
|    0004 | D      | 19.95 |
+---------+--------+-------+
```

3.3 Maximum of Column per Group

Task: Find the highest price per article.

```
SELECT article, MAX(price) AS price
FROM   shop
GROUP BY article
ORDER BY article;
```

```
+---------+-------+
| article | price |
+---------+-------+
|    0001 |  3.99 |
|    0002 | 10.99 |
|    0003 |  1.69 |
|    0004 | 19.95 |
+---------+-------+
```

3.4 The Rows Holding the Group-wise Maximum of a Certain Column

Task: For each article, find the dealer or dealers with the most expensive price.

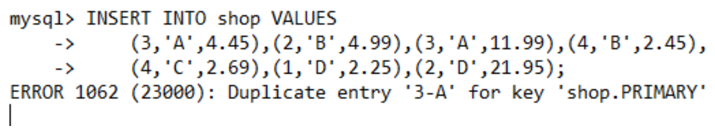
This problem can be solved with a subquery like this one:

```
SELECT article, dealer, price
FROM   shop s1
WHERE  price=(SELECT MAX(s2.price)
              FROM shop s2
              WHERE s1.article = s2.article)
ORDER BY article;
```

```
+---------+--------+-------+
| article | dealer | price |
+---------+--------+-------+
|    0001 | B      |  3.99 |
|    0002 | A      | 10.99 |
|    0003 | C      |  1.69 |
|    0004 | D      | 19.95 |
+---------+--------+-------+
```

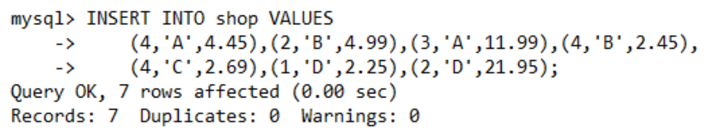
4. Inserting records in the table

In the previously created table, we define primary key on combination of column "article, dealer". So if we try to gives duplicate values to the combination of both columns. It gives error.



```
mysql> INSERT INTO shop VALUES
    ->     (3,'A',4.45),(2,'B',4.99),(3,'A',11.99),(4,'B',2.45),
    ->     (4,'C',2.69),(1,'D',2.25),(2,'D',21.95);
ERROR 1062 (23000): Duplicate entry '3-A' for key 'shop.PRIMARY'
```

After removing these duplicate values.



```
mysql> INSERT INTO shop VALUES
    ->     (4,'A',4.45),(2,'B',4.99),(3,'A',11.99),(4,'B',2.45),
    ->     (4,'C',2.69),(1,'D',2.25),(2,'D',21.95);
Query OK, 7 rows affected (0.00 sec)
Records: 7  Duplicates: 0  Warnings: 0
```

5. Constraints

- ✓ Your DBMS can do much more than just store and access data. It can also enforce rules (called constraints) on what data are allowed in the database. Such constraints are important because they help maintain data integrity. For example, you may want to ensure that each cgpa is not less than 2.0. When you specify the data type of a column, you constrain the possible values that column may hold. This is called a domain constraint. For example, a column of type INTEGER may only hold whole numbers within a certain range. Any attempt to insert an invalid value will be rejected by SQL. SQL allows the specification of many more constraint types. SQL enforces constraints by prohibiting any data in the database that violate any constraint. Any insert, update, or delete that would result in a constraint violation is rejected without changing the database.
- ✓ There are two forms of constraint specification: o Column level Constraints:- Apply to individual columns only (are specified along with the column definition only)   o Table Level constraints:- Apply to one or more columns (are specified at the end)
- ✓ Constraints can be added to the table at the time of creation or after the creation of the table using 'alter table' command.

5.1 NOT NULL

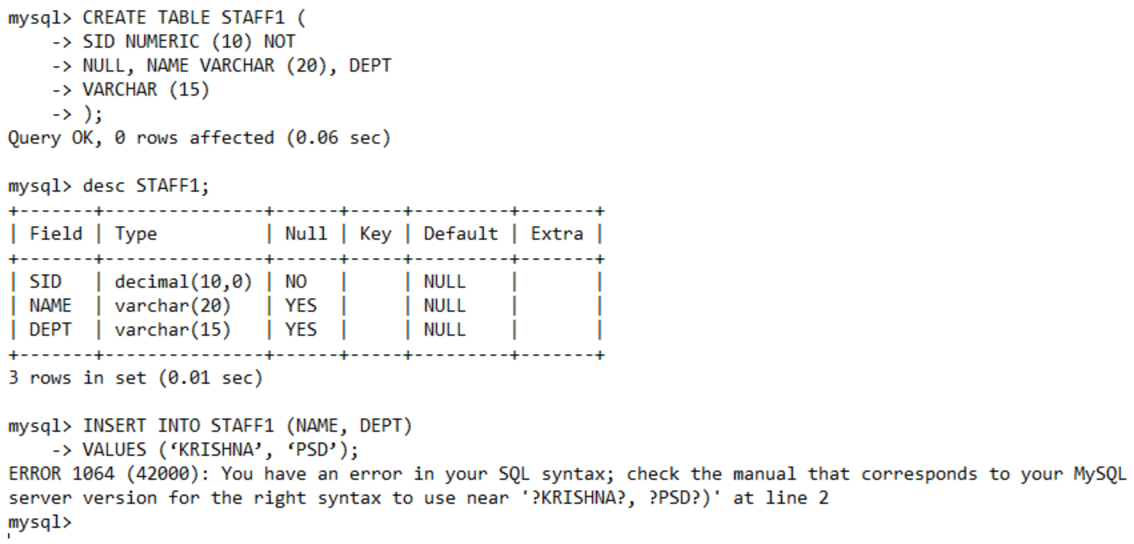
- ✓ By default, most DBMSs allow NULL as a value for any column of any data type. You may not be so keen on allowing NULL values for some columns. You can require the database to prohibit NULL values for particular columns by using the NOT NULL column constraint. Many DBMSs also include a NULL column constraint, which specifies that NULL values are allowed; however, because this is the default behavior, this constraint usually is unnecessary. Note that the NULL column constraint is not part of the SQL specification.

```
CREATE TABLE STAFF1 (
SID NUMERIC (10) NOT
NULL, NAME VARCHAR (20), DEPT
VARCHAR (15)
);
```

✓ Now if you try inserting,

```
INSERT INTO STAFF1 (NAME, DEPT)
VALUES ('KRISHNA', 'PSD');
```

You will get error "Cannot insert the value NULL into column …"



```
mysql> CREATE TABLE STAFF1 (
    -> SID NUMERIC (10) NOT
    -> NULL, NAME VARCHAR (20), DEPT
    -> VARCHAR (15)
    -> );
Query OK, 0 rows affected (0.06 sec)

mysql> desc STAFF1;
+-------+---------------+------+-----+---------+-------+
| Field | Type          | Null | Key | Default | Extra |
+-------+---------------+------+-----+---------+-------+
| SID   | decimal(10,0) | NO   |     | NULL    |       |
| NAME  | varchar(20)   | YES  |     | NULL    |       |
| DEPT  | varchar(15)   | YES  |     | NULL    |       |
+-------+---------------+------+-----+---------+-------+
3 rows in set (0.01 sec)

mysql> INSERT INTO STAFF1 (NAME, DEPT)
    -> VALUES ('KRISHNA', 'PSD');
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL
server version for the right syntax to use near '?KRISHNA?, ?PSD?)' at line 2
mysql>
```

5.2 UNIQUE

✓ The UNIQUE constraint forces distinct column values. Suppose you want to avoid duplicate course numbers. Just specify the UNIQUE column constraint on the courseno column as follows:

```
CREATE TABLE COURSE (
COMPCODE NUMERIC (4) UNIQUE,
COURSENO VARCHAR (9) NOT NULL UNIQUE,
COURSE_NAME VARCHAR (20),
UNITS NUMERIC (2) NOT NULL
);
```

Note that UNIQUE only applies to non-NULL values. A UNIQUE column may have many rows containing a NULL value. Of course, we can exclude all NULL values for the column using the NOT NULL constraint with the UNIQUE constraint.

✓ UNIQUE also has a table constraint form that applies to the entire table instead of just a single column. Table constraints are specified as another item in the comma-delimited list of table elements.

Such table constraints apply to groups of one or more columns. Consider the following CREATE TABLE statement:

```
DROP TABLE COURSE; --TO DELETE THE TABLE
```

```
CREATE TABLE COURSE (
COMPCODE NUMERIC (4),
COURSENO VARCHAR (9) NOT NULL,
COURSE_NAME VARCHAR (20),
UNITS NUMERIC (2) NOT NULL,
UNIQUE (COMPCODE, COURSENO)   -- TABLE LEVEL CONSTRAINT
);
```

- ✓ The combination of compcode, courseno is always unique. Note that table level unique constraint can also be for single column. Unique is nothing but the candidate key constraint but a unique column can take null values.

5.3 PRIMARY KEY

- ✓ It is similar to unique but with implicit NOT NULL constraint. The primary key of a table is a column or set of columns that uniquely identifies a row in the table. For example, idno is the primary key from the students table. We can declare a primary key using the PRIMARY KEY constraint. Here we show PRIMARY KEY used as a column constraint.

```
CREATE TABLE EMPLOYEE (
EMPID NUMERIC (4) PRIMARY KEY,
NAME VARCHAR (30) NOT NULL
);
```

- ✓ Creating primary key as a table constraint is shown below.

```
CREATE TABLE REGISTERED (
COURSENO VARCHAR (9),
IDNO CHAR (11),
GRADE VARCHAR (10),
PRIMARY KEY (COURSENO, IDNO)
);
```

- ✓ You can name your constraints by using an optional prefix.

```
CONSTRAINT <NAME> <CONSTRAINT
CREATE TABLE REGISTERED (
COURSENO VARCHAR(9),
IDNO CHAR(11),
GRADE VARCHAR(10),
CONSTRAINT PK_REGISTERED PRIMARY KEY(COURSENO, IDNO)   );
```

- ✓ This naming can be applied to unique constraints also. Naming constraint helps in altering or dropping that constraint at a later time.

5.4 FOREIGN KEY

- ✓ A foreign key restricts the values of a column (or a set of columns) to the values appearing in another column (or set of columns) or to NULL. In table registered (child table), courseno is a foreign key that refers to courseno in table course (parent table). We want all of the values of courseno in the registered table either to reference a courseno from course or to be NULL. Any other courseno in the registered table would create problems because you couldn't look up information about the courseno which is not offered.

- ✓ In SQL, we specify a foreign key with the REFERENCES column constraint.

```
REFERENCES <REFERENCED TABLE> [(<REFERENCED COLUMN>)]
```

- ✓ A column with a REFERENCES constraint may only have a value of either NULL or a value found in column <referenced column> of table <referenced table>. If the <referenced column> is omitted, the primary key of table <referenced table> is used.
- ✓ Before creating a foreign keys in registered table, let us re-create course, students tables with proper constraints.

```
DROP TABLE STUDENTS;
CREATE TABLE STUDENTS(
IDNO CHAR(11),
NAME VARCHAR(30),
DOB DATE,
CGPA NUMERIC(4,2),
AGE NUMERIC(2),
CONSTRAINT PK_STUDENTS PRIMARY KEY(IDNO)
);


DROP TABLE COURSE;
CREATE TABLE COURSE (
COMPCODE NUMERIC(4),
COURSENO VARCHAR(9) NOT NULL,
COURSE_NAME VARCHAR(20),
UNITS NUMERIC(2) NOT NULL,
CONSTRAINT UN_COURSE UNIQUE (COMPCODE, COURSENO),   -- TABLE
LEVEL CONSTRAINT
CONSTRAINT PK_COURSE PRIMARY KEY (COURSENO)
);


CREATE TABLE REGISTERED1 (
COURSENO VARCHAR (9) REFERENCES COURSE, --COLUMN LEVEL FOREIGN
KEY
IDNO CHAR(11) REFERENCES STUDENTS,
GRADE VARCHAR(10),
PRIMARY KEY(COURSENO, IDNO)
);
```

- ✓ The same can be declared as table level constraint with proper naming.

```
CREATE TABLE REGISTERED2(
COURSENO VARCHAR(9),
IDNO CHAR(11),
GRADE VARCHAR(10),
CONSTRAINT PK_REGISTERED2 PRIMARY KEY(COURSENO, IDNO),
CONSTRAINT FK_CNO FOREIGN KEY(COURSENO) REFERENCES COURSE,
CONSTRAINT FK_IDNO  FOREIGN KEY (IDNO) REFERENCES STUDENTS
);
```

- ✓ To create a foreign key reference, SQL requires that the referenced table/column already exist.
- ✓ Maintaining foreign key constraints can be painful. To update or delete a referenced value in the parent table, we must make sure that we first handle all foreign keys referencing that value in the child table. For example, to update or delete 2007A7PS001 from the students table, we must first update or delete all registered. idno. SQL allows us to specify the default actions for maintaining foreign key constraints for UPDATE and DELETE on the parent table by adding a referential action clause to the end of a column or table foreign key constraint:

```
ON UPDATE <ACTION>
ON DELETE <ACTION>
```

- ✓ Any UPDATE or DELETE on the parent table triggers the specified <action> on the referencing rows in the child table.

Action	Definition
SET NULL	Sets any referencing foreign key values to NULL.
SET DEFAULT	Sets any referencing foreign key values to the default value (which may be NULL).
CASCADE	On delete, this deletes any rows with referencing foreign key values. On update, this updates any row with referencing foreign key values to the new value of the referenced column.
NO ACTION	Rejects any update or delete that violates the foreign key constraint. This is the default action.
RESTRICT	Same as NO ACTION with the additional restriction that the action cannot be deferred

- ✓ Try the following.

```
DROP TABLE REGISTERED2; CREATE
TABLE REGISTERED2( COURSENO  VARCHAR(9) ,  IDNO CHAR(11),
GRADE VARCHAR(10) ,  CONSTRAINT
PK_REGISTERED2
PRIMARY KEY(COURSENO, IDNO),
CONSTRAINT FK_CNO FOREIGN KEY (COURSENO) REFERENCES COURSE ON
DELETE CASCADE,
CONSTRAINT FK_IDNO FOREIGN KEY (IDNO) REFERENCES STUDENTS ON
DELETE
CASCADE
);
```

- ✓ Modify the above query with other actions ON DELETE SET NULL, ON DELETE NO ACTION, ON UPDATE NO ACTION.

5.5 CHECK

- ✓ We can specify a much more general type of constraint using the CHECK constraint. A CHECK constraint specifies a boolean value expression to be evaluated for each row before allowing any data change. Any INSERT, UPDATE, or DELETE that would cause the condition for any row to evaluate to false is rejected by the DBMS.

```
CHECK (<condition>)
```

- ✓ A CHECK constraint may be specified as either a column or table constraint. In the following example, we specify CHECK constraints on the students table:

```
CREATE TABLE STUDENT1(
IDNO CHAR(11) PRIMARY KEY,
NAME VARCHAR(20) NOT NULL,
CGPA NUMERIC(4,2) CHECK(CGPA >= 2 AND CGPA <= 10),  -- CGPA
CONSTRAINT
ROOMNO NUMERIC(3) CHECK(ROOMNO >99),
HOSTEL_CODE VARCHAR(2) CHECK(HOSTEL_CODE IN ("VK","RP","MB"))
);
```

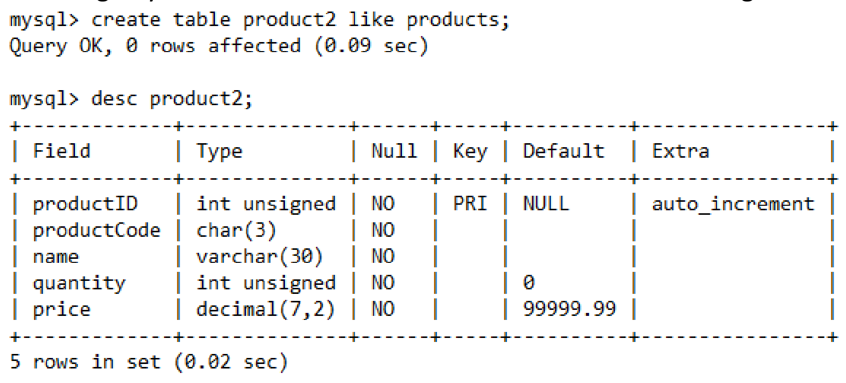
- ✓ Check constraints can also be named.
- ✓ Does a roomno with a NULL value violate the CHECK constraint? No. In this case, the CHECK condition evaluates to unknown. The CHECK constraint only rejects a change when the condition evaluates to false. In the SQL standard, a CHECK constraint condition may even include subqueries referencing other tables; however, many DBMSs do not implement this feature.

6. More on Creating, Deleting, and Altering Tables: (DDL)

In the previous section, creation tables with defaults values, NOT NULL constraint, UNIQUE constraint, Primary key, foreign key and check constraints have been discussed.

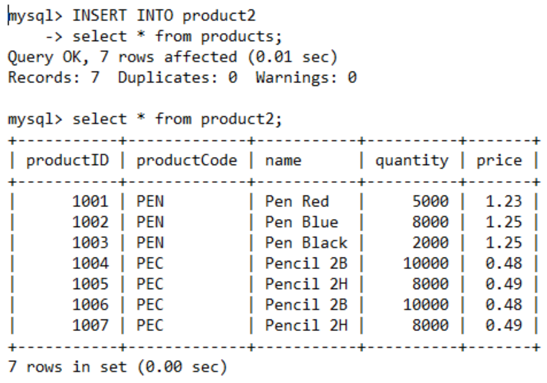
6.1 Creating a table from another table:

LIKE - Use CREATE TABLE … LIKE to create an empty table based on the definition of another table, including any column attributes and indexes defined in the original table:



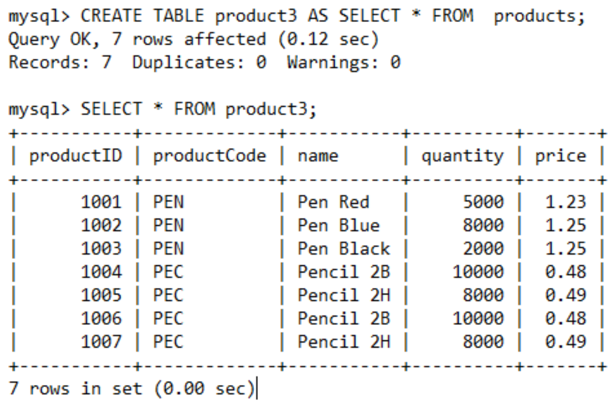
```
mysql> create table product2 like products;
Query OK, 0 rows affected (0.09 sec)

mysql> desc product2;
+-------------+--------------+------+-----+----------+----------------+
| Field       | Type         | Null | Key | Default  | Extra          |
+-------------+--------------+------+-----+----------+----------------+
| productID   | int unsigned | NO   | PRI | NULL     | auto_increment |
| productCode | char(3)      | NO   |     |          |                |
| name        | varchar(30)  | NO   |     |          |                |
| quantity    | int unsigned | NO   |     | 0        |                |
| price       | decimal(7,2) | NO   |     | 99999.99 |                |
+-------------+--------------+------+-----+----------+----------------+
5 rows in set (0.02 sec)
```

Now, can copy values from table "products" into "product2".



```
mysql> INSERT INTO product2
    -> select * from products;
Query OK, 7 rows affected (0.01 sec)
Records: 7  Duplicates: 0  Warnings: 0

mysql> select * from product2;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1001 | PEN         | Pen Red   |     5000 |  1.23 |
|      1002 | PEN         | Pen Blue  |     8000 |  1.25 |
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
|      1004 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1005 | PEC         | Pencil 2H |     8000 |  0.49 |
|      1006 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1007 | PEC         | Pencil 2H |     8000 |  0.49 |
+-----------+-------------+-----------+----------+-------+
7 rows in set (0.00 sec)
```

[AS] query_expression - To create one table from another, add a SELECT statement at the end of the CREATE TABLE statement:



```
mysql> CREATE TABLE product3 AS SELECT * FROM  products;
Query OK, 7 rows affected (0.12 sec)
Records: 7  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM product3;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1001 | PEN         | Pen Red   |     5000 |  1.23 |
|      1002 | PEN         | Pen Blue  |     8000 |  1.25 |
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
|      1004 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1005 | PEC         | Pencil 2H |     8000 |  0.49 |
|      1006 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1007 | PEC         | Pencil 2H |     8000 |  0.49 |
+-----------+-------------+-----------+----------+-------+
7 rows in set (0.00 sec)
```

6.2 Alter Table

Begin with a table t1 created as shown here:

```
CREATE TABLE t1 (a INTEGER, b CHAR(10));
```

To rename the table from t1 to t2:

```
ALTER TABLE t1 RENAME t2;
```

To change column a from INTEGER to TINYINT NOT NULL (leaving the name the same), and to change column b from CHAR(10) to CHAR(20) as well as renaming it from b to c:

```
ALTER TABLE t2 MODIFY a TINYINT NOT NULL, CHANGE b c CHAR(20);
```

6.3 ADD [COLUMN] <column definition>

To add a new TIMESTAMP column named d:

```
ALTER TABLE t2 ADD d TIMESTAMP;
```

6.4 MODIFY <column name>

To add a new AUTO_INCREMENT integer column named c:

```
ALTER TABLE t2 ADD c INT UNSIGNED NOT NULL AUTO_INCREMENT,
  ADD PRIMARY KEY (c);
```

6.5 DROP <column name>

To remove column c:

```
ALTER TABLE t2 DROP COLUMN c;
```

6.6 ADD <table constraint>

To add an index on column d and a UNIQUE index on column a:

```
ALTER TABLE t2 ADD INDEX (d), ADD UNIQUE (a);
```

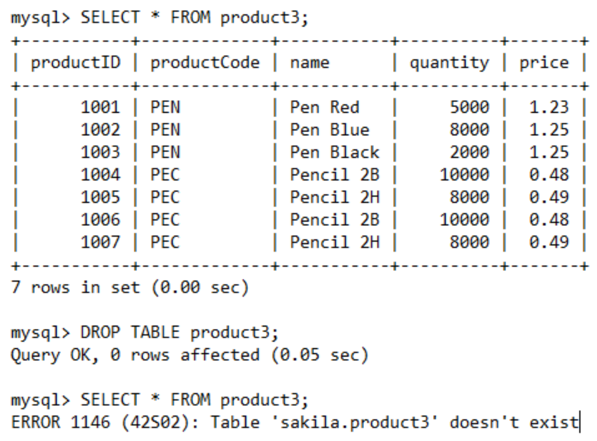
7. Dropping a Table

DROP TABLE removes one or more tables. You must have the DROP privilege for each table. Be careful with this statement! For each table, it removes the table definition and all table data. If the table is partitioned, the statement removes the table definition, all its partitions, all data stored in those partitions, and all partition definitions associated with the dropped table.

Dropping a table also drops any triggers for the table. DROP TABLE causes an implicit commit, except when used with the TEMPORARY keyword.

```
DROP [TEMPORARY] TABLE [IF EXISTS]
    tbl_name [, tbl_name] ...
    [RESTRICT | CASCADE]
```

The RESTRICT and CASCADE keywords do nothing. They are permitted to make porting easier from other database systems.



```
mysql> SELECT * FROM product3;
+-----------+-------------+------------+----------+-------+
| productID | productCode | name       | quantity | price |
+-----------+-------------+------------+----------+-------+
|      1001 | PEN         | Pen Red    |     5000 |  1.23 |
|      1002 | PEN         | Pen Blue   |     8000 |  1.25 |
|      1003 | PEN         | Pen Black  |     2000 |  1.25 |
|      1004 | PEC         | Pencil 2B  |    10000 |  0.48 |
|      1005 | PEC         | Pencil 2H  |     8000 |  0.49 |
|      1006 | PEC         | Pencil 2B  |    10000 |  0.48 |
|      1007 | PEC         | Pencil 2H  |     8000 |  0.49 |
+-----------+-------------+------------+----------+-------+
7 rows in set (0.00 sec)

mysql> DROP TABLE product3;
Query OK, 0 rows affected (0.05 sec)

mysql> SELECT * FROM product3;
ERROR 1146 (42S02): Table 'sakila.product3' doesn't exist
```

7 Exercise

1. Write the following queries in SQL, using the university schema.
    a. Find the titles of courses in the Comp. Sci. department that have 3 credits.
    b. Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.
    c. Find the highest salary of any instructor.
    d. Find all instructors earning the highest salary (there may be more than one with the same salary).
    e. Find the enrollment of each section that was offered in Fall 2017.
    f. Find the maximum enrollment, across all sections, in Fall 2017.

    g. Find the sections that had the maximum enrollment in Fall 2017.

2. Write the following inserts, deletes, or updates in SQL, using the university schema.
    a. Increase the salary of each instructor in the Comp. Sci. department by 10%.
    b. Delete all courses that have never been offered (i.e., do not occur in the *section* relation).
    c. Insert every student whose *tot cred* attribute is greater than 100 as an instructor in the same department, with a salary of $10,000.

***********************************END***************************************

Birla Institute of Technology and Science, Pilani
CS F212 Database Systems
Lab No # 2

****

1 Data Manipulation Language (DML)

In today's lab, DML part of SQL will be discussed. DML provide a set of operations related to data manipulation.

2 Modifying Data: Common Operations

SQL provides three statements for modifying data: INSERT, UPDATE, and DELETE.

2.1 INSERT

The INSERT statement adds new rows to a specified table. There are two variants of the INSERT statement. One inserts a single row of values into the database, whereas the other inserts multiple rows returned from a SELECT.

The most basic form of INSERT creates a single, new row with either user-specified or default values. This is covered in the previous lab.

The INSERT statement allows you to create new rows from existing data. It begins just like the INSERT statements we've already seen; however, instead of a VALUES list, the data are generated by a nested SELECT statement. The syntax is as follows:

```
INSERT INTO <table name>

[(<attribute>, : : :, <attribute>)]

<SELECT statement>;
```

SQL then attempts to insert a new row in *<table name>* for each row in the SELECT result table. The attribute list of the SELECT result table must match the attribute list of the INSERT statement.

```
mysql> INSERT INTO
`sakila`.`actor`(`first_name`,`last_name`)VALUES("BITS","PILANI");

Query OK, 1 row affected (0.01 sec)
```

2.2 UPDATE

You can change the values in existing rows using UPDATE.

```
UPDATE <table-name> [[AS] <alias>]

SET <column>=<expression>, : : :, <attribute>=<expression>

[WHERE <condition>];
```

UPDATE changes all rows in *<table name>* where *<condition>* evaluates to true. For each row, the SET clause dictates which attributes change and how to compute the new value. All other attribute values do not change. The WHERE is optional, but beware! If there is no WHERE clause, UPDATE changes *all* rows. UPDATE can only change rows from a single table.

```
mysql> update actor set actor.last_name="Vidyavihar" where actor.first_name =
"BITS";
Query OK, 2 rows affected (0.01 sec)
Rows matched: 2  Changed: 2  Warnings: 0
```

2.3 DELETE

You can remove rows from a table using DELETE.

```
DELETE FROM <table name> [[AS] <alias>]
[WHERE <condition>];
```

DELETE removes all rows from *<table name>* where *<condition>* evaluates to true. The WHERE is optional, but beware! If there is no WHERE clause, DELETE removes *all* rows. DELETE can only remove rows from a single table.

```
mysql> delete from actor where actor.first_name = "BITS";
Query OK, 2 rows affected (0.04 sec)
```

3 The Sakila Schema

One of the best example databases out there is the Sakila Database, which was originally created by MySQL and has been open sourced under the terms of the BSD License.

The Sakila database is a nicely normalized schema modelling a DVD rental store, featuring things like films, actors, film-actor relationships, and a central inventory table that connects films, stores, and rentals.

This Database has following table:

1 actor Table
2 address Table
3 category Table
4 city Table
5 country Table
6 customer Table
7 film Table
8 film_actor Table
9 film_category Table
10 film_text Table
11 inventory Table
12 language Table
13 payment Table
14 rental Table
15 staff Table
16 store Table

The following diagram provides an overview of Sakila sample database structure.

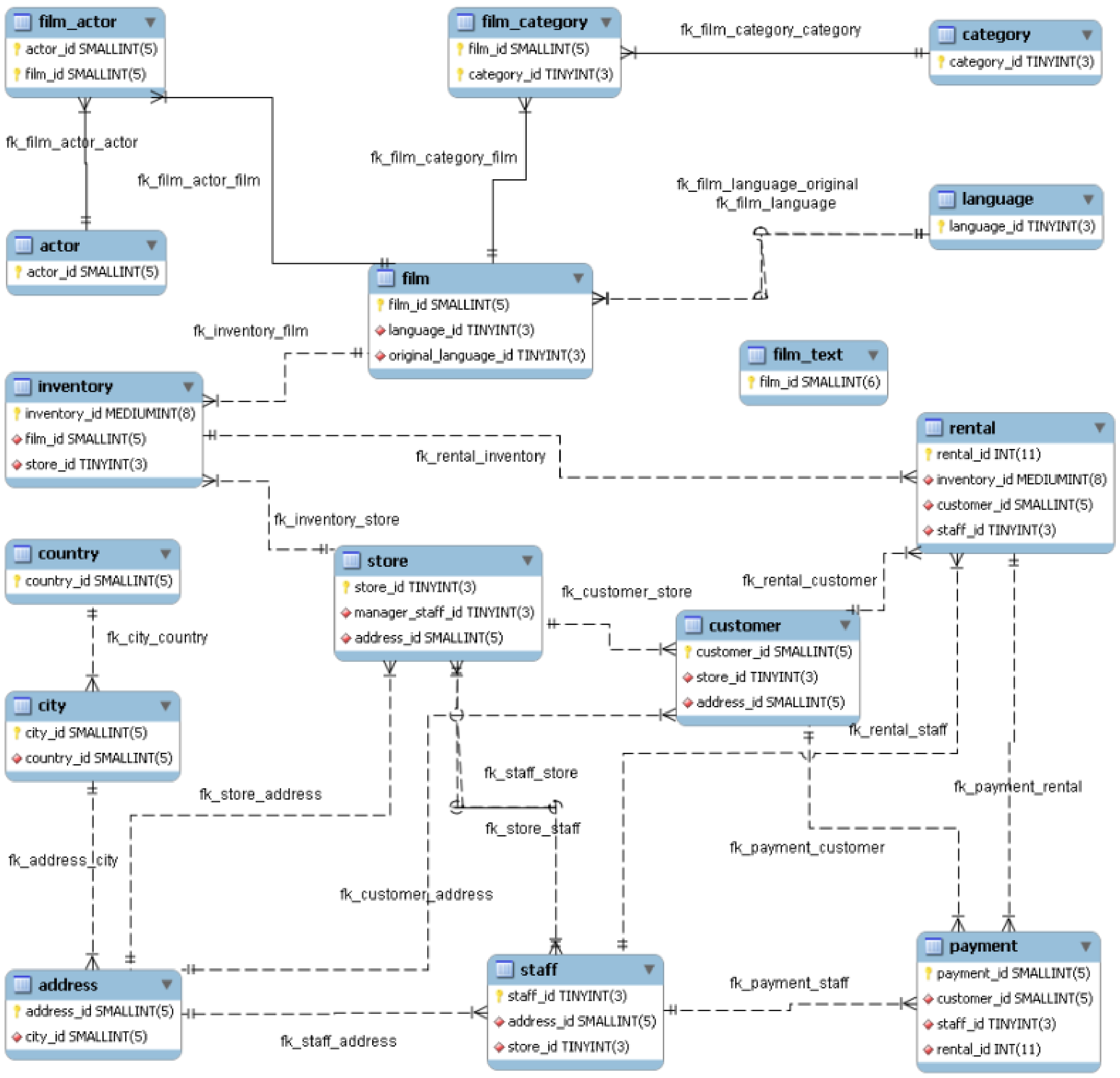


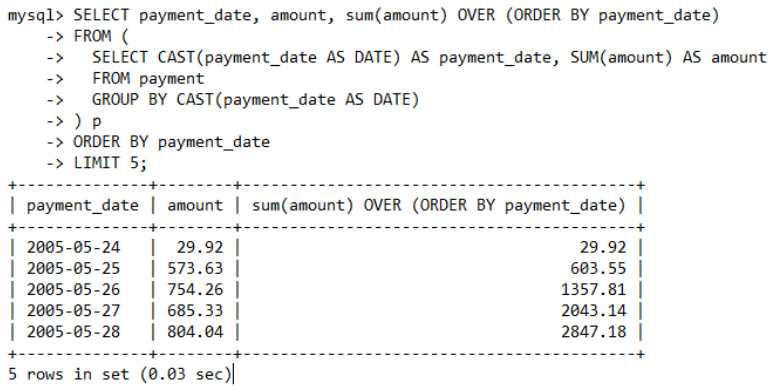
Figure 1: “Sakila” Database EER Diagram.

4 Data Retrieval

The database allows for nice example queries like the following one that finds the actor with most films.

```
mysql> SELECT first_name, last_name, count(*) films
    -> FROM actor AS a
    -> JOIN film_actor AS fa USING (actor_id)
    -> GROUP BY actor_id, first_name, last_name
    -> ORDER BY films DESC
    -> LIMIT 1;
+------------+-----------+-------+
| first_name | last_name | films |
+------------+-----------+-------+
| GINA       | DEGENERES |    42 |
+------------+-----------+-------+
1 row in set (0.06 sec)
```

Or, let's calculate the cumulative revenue of all stores.



```
mysql> SELECT payment_date, amount, sum(amount) OVER (ORDER BY payment_date)
    -> FROM (
    ->   SELECT CAST(payment_date AS DATE) AS payment_date, SUM(amount) AS amount
    ->   FROM payment
    ->   GROUP BY CAST(payment_date AS DATE)
    -> ) p
    -> ORDER BY payment_date
    -> LIMIT 5;
+--------------+--------+----------------------------------------+
| payment_date | amount | sum(amount) OVER (ORDER BY payment_date) |
+--------------+--------+----------------------------------------+
| 2005-05-24   |  29.92 |                                  29.92 |
| 2005-05-25   | 573.63 |                                 603.55 |
| 2005-05-26   | 754.26 |                                1357.81 |
| 2005-05-27   | 685.33 |                                2043.14 |
| 2005-05-28   | 804.04 |                                2847.18 |
+--------------+--------+----------------------------------------+
5 rows in set (0.03 sec)
```

5 Filtering the Query Result

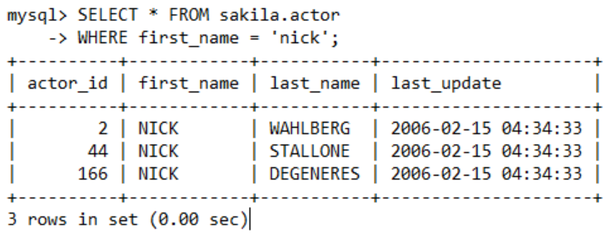
For retrieval, SELECT, WHERE, FROM, GROUP BY, HAVING clauses are used extensively. We will practice some SQL queries on a schema of “Sakila”. The schema has 16 tables, 7 views, 3 stored procedures, and 3 functions.

5.1 WHERE

The SELECT clause is used to select data from a database. The SELECT clause is a query expression that begins with the SELECT keyword and includes a number of elements that form the expression. WHERE clause is used to specify the Search Conditions. The operators commonly used for comparison are =, > , < , >=, <= , <>.

For Example:

1. Selects all records from the actor table in the sakila database where the value of the first_name column is equal to nick.



```
mysql> SELECT * FROM sakila.actor
    -> WHERE first_name = 'nick';
+----------+------------+-----------+---------------------+
| actor_id | first_name | last_name | last_update         |
+----------+------------+-----------+---------------------+
|        2 | NICK       | WAHLBERG  | 2006-02-15 04:34:33 |
|       44 | NICK       | STALLONE  | 2006-02-15 04:34:33 |
|      166 | NICK       | DEGENERES | 2006-02-15 04:34:33 |
+----------+------------+-----------+---------------------+
3 rows in set (0.00 sec)
```

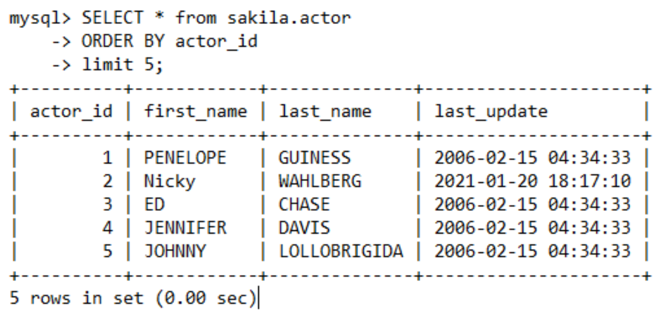
The WHERE clause is also used with the SQL UPDATE statement to specify the record that is to be updated:

```
mysql> UPDATE sakila.actor
    -> SET first_name = 'Nicky'
    -> WHERE actor_id = 2;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

5.2 ORDER BY

The ORDER BY clause can be used within an SQL statement to sort the result set by one or more fields.

Statement selects all records from the actor table in the sakila database, then orders them by the actor_id field in ascending order.



```
mysql> SELECT * from sakila.actor
    -> ORDER BY actor_id
    -> limit 5;
+----------+------------+--------------+---------------------+
| actor_id | first_name | last_name    | last_update         |
+----------+------------+--------------+---------------------+
|        1 | PENELOPE   | GUINESS      | 2006-02-15 04:34:33 |
|        2 | Nicky      | WAHLBERG     | 2021-01-20 18:17:10 |
|        3 | ED         | CHASE        | 2006-02-15 04:34:33 |
|        4 | JENNIFER   | DAVIS        | 2006-02-15 04:34:33 |
|        5 | JOHNNY     | LOLLOBRIGIDA | 2006-02-15 04:34:33 |
+----------+------------+--------------+---------------------+
5 rows in set (0.00 sec)
```

The ORDER BY clause orders the results in ascending order by default. You can also add ASC to the clause in order to be explicit about this. Like this:

```
SELECT * from sakila.actor
ORDER BY actor_id ASC;
```

You can use DESC so that the results are listed in descending order. Like this:

```
SELECT * from sakila.actor
ORDER BY actor_id DESC;
```

You can use more than one field in your ORDER BY clause. The results will be ordered by the first column specified, then the second, third, and so on.

```
SELECT * from sakila.actor
WHERE first_name LIKE 'An%'
ORDER BY first_name, last_name DESC;
```

5.3 GROUP BY

The GROUP BY clause groups the returned record set by one or more columns. You specify which columns the result set is grouped by.

Let Consider sakila.actor table. We can see that the last_name column contains a lot of duplicates — many actors share the same last name.

Now, if we add GROUP BY last_name in select statement.

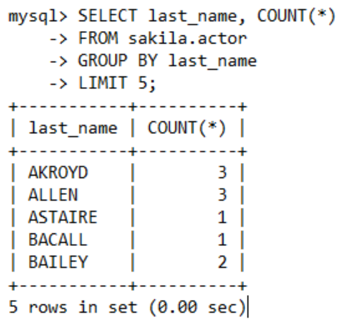
```
SELECT last_name
FROM sakila.actor
GROUP BY last_name;
```

We have selected all actors' last names from the table and grouped them by the last name. If two or more actors share the same last name, it is represented only once in the result set. For example, if two actors have a last name of "Bailey", that last name is listed once only.

Using COUNT() with GROUP BY :-

A benefit of using the GROUP BY clause is that you can combine it with aggregate functions and other clauses to provide a more meaningful result set.

For example, we could add the COUNT() function to our query to return the number of records that contain each last name.

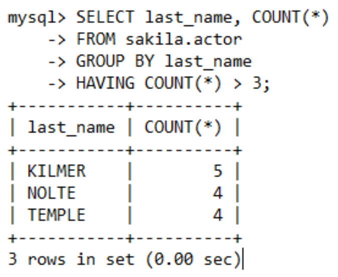


```
mysql> SELECT last_name, COUNT(*)
    -> FROM sakila.actor
    -> GROUP BY last_name
    -> LIMIT 5;
+-----------+----------+
| last_name | COUNT(*) |
+-----------+----------+
| AKROYD    |        3 |
| ALLEN     |        3 |
| ASTAIRE   |        1 |
| BACALL    |        1 |
| BAILEY    |        2 |
+-----------+----------+
5 rows in set (0.00 sec)
```

5.4 HAVING

The HAVING clause can be used as a filter on a GROUP BY clause. It is used to apply a filter to a group of rows or aggregates. This contrasts with the WHERE clause, which is applied before the GROUP BY clause.

Consider the following example.



```
mysql> SELECT last_name, COUNT(*)
    -> FROM sakila.actor
    -> GROUP BY last_name
    -> HAVING COUNT(*) > 3;
+-----------+----------+
| last_name | COUNT(*) |
+-----------+----------+
| KILMER    |        5 |
| NOLTE     |        4 |
| TEMPLE    |        4 |
+-----------+----------+
3 rows in set (0.00 sec)
```

In the above example, we use the HAVING clause to filter the result set to only those records that have a count of greater than three (i.e. HAVING COUNT(*) > 3).

If we didn't use the HAVING CLAUSE, it would have returned all records — regardless of their count.

5.5 DISTINCT

The DISTINCT keyword can be used within an SQL statement to remove duplicate rows from the result set of a query.

Consider the following example (which doesn't use the DISTINCT option):

```
SELECT first_name
FROM sakila.actor
WHERE first_name LIKE 'An%';
```

You can see that there are two records containing the value of Angela. Now let's add the DISTINCT keyword:

```
SELECT DISTINCT first_name
```

```
FROM sakila.actor
WHERE first_name LIKE 'An%';
```

There is now only one record that contains the value of Angela. This is because the DISTINCT keyword removed the duplicates. Therefore, we know that each row returned by our query will be distinct — it will contain a unique value.

Using DISTINCT with COUNT() :-

You can insert the DISTINCT keyword within the COUNT() aggregate function to provide a count of the number of matching rows.

```
SELECT COUNT(DISTINCT first_name)
FROM sakila.actor
WHERE first_name LIKE 'An%';
```

Result of above query is 3.

If we remove the DISTINCT option (but leave COUNT() in):

```
SELECT COUNT(DISTINCT first_name)
FROM sakila.actor
WHERE first_name LIKE 'An%';
```

We end up with 4 (instead of 3 as we did when using DISTINCT):

6 More with WHERE

The following operators are also used in WHERE clause.

AND, OR, NOT	Logical Operation	Logical Operation
BETWEEN	Between two values(inclusive)	BETWEEN 2 AND 10
IS NULL	Value is Null	Referred by IS NULL
LIKE	Equal Sting using wilds cards(e.g. '%','_')	Name LIKE 'pri%'
IN	Equal to any element in list	Name IN ('soda',' water')
NOT	Negates a condition	NOT item IN ('GDNSD','CHKSD')

6.1 Logical operators-AND, OR, NOT

AND operator return the result which satisfy both conditions given in where clause.

```
SELECT * from sakila.actor
where first_name = "DAN" AND actor_id = 116;
```

OR operator return the results either of condition satisfy in where clause.

```
SELECT * from sakila.actor
where first_name = "DAN" or actor_id = 115;
```

NOT operator return all rows other than which meet the condition in where clause.

```
SELECT * from sakila.actor
where NOT actor_id = 116;
```

NOTE: The precedence order is NOT, AND, OR from highest to lowest.
Example: Select rows from sakila.actor in which first_name start with either S or M and actor_id is >=100.

```
select * from sakila.actor
where first_name like "S%" or first_name like "M%"
and actor_id >=100;

select * from sakila.actor
where (first_name like "S%" or first_name like "M%")
and actor_id >=100;
```

Run above two queries and check the difference in result based on the precedence of operators.

6.2 BETWEEN

BETWEEN keywork provides a range of values which satisfy the condition in where clause. BETWEEN include the start and value in the provide range. For example statement given bellow prints the rows from sakila.actor table which has actor_id between 100 to 105.

```
SELECT * from sakila.actor
where actor_id between 100 and 105;
```

6.3 IN, NOT IN

IN and NOT IN keyword applied on a finite set of values. IN check for matches in the given set and restrict the result of select statement only for the matches in given set. And NOT IN perform reverse of IN.

```
select * from sakila.actor
where actor_id IN (100, 110, 120);

select * from sakila.actor
where actor_id NOT IN (100, 110, 120);
```

Run above queries and check result to identify the difference in both keywords.

6.4 IS NULL

SQL interprets NULL as unknown. So comparing any value with NULL returns unknown result.

Select the rows from address table which has null for address2 attribute.

```
select * from sakila.address
where address2 = null;
```

Check the output and try the following.

```
select * from sakila.address
where address2 IS NULL;
```

Do you see the difference? Comparing NULL with any attribute gives an unknown result.

6.5 LIKE

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 represents a single character

LIKE OPERATOR	DESCRIPTION
LIKE 'A%'	Finds any values that start with "a"
LIKE '%A'	Finds any values that end with "a"
LIKE '%OR%'	Finds any values that have "or" in any position
LIKE '_R%'	Finds any values that have "r" in the second position
LIKE 'A_%'	Finds any values that start with "a" and are at least 2 characters in length
LIKE 'A__%'	Finds any values that start with "a" and are at least 3 characters in length
LIKE 'A%O'	Finds any values that start with "a" and ends with "o"

LIKE keyword is illustrated in section 5.4 and 5.5.

7 Conditional Expressions

SQL also provides basic conditional constructs to determine the correct result. CASE provides a general mechanism for specifying conditional results. SQL also provides the COALESCE and NULLIF statements to deal with NULL values, but these have equivalent CASE statements. MySQL provides various conditional expressions, function and statement constructs. Here, we only discussing few of them for understanding purpose.

7.1 CASE

The CASE statement for stored programs implements a complex conditional construct.

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]

END CASE
```

Or

```
CASE
    WHEN search_condition THEN statement_list
    [WHEN search_condition THEN statement_list] ...
    [ELSE statement_list]

END CASE
```

Example:

```
SELECT `products`.`productID`,
    `products`.`productCode`,
    `products`.`name`,
    `products`.`quantity`,
    `products`.`price` ,
    CASE `products`.`quantity`
    WHEN 2000 THEN "NOT GOOD"
    WHEN 8000 THEN "AVERAGE"
    WHEN 10000 THEN "GOOD"
    ELSE "UNKNOWN"
    END AS STATUS
FROM `sakila`.`products`;


SELECT `products`.`productID`,
    `products`.`productCode`,
    `products`.`name`,
    `products`.`quantity`,
    `products`.`price` ,
    CASE
    WHEN QUANTITY >8000 THEN "GOOD"
    ELSE "NOT GOOD"
    END AS STATUS
FROM `sakila`.`products`;
```

7.2 IF

IF(expr1,expr2,expr3)

If expr1 is TRUE (expr1 <> 0 and expr1 <> NULL), IF() returns expr2. Otherwise, it returns expr3.

Note: There is also an IF statement, which differs from the IF() function.

```
mysql> SELECT IF(1>2,2,3);
        -> 3
mysql> SELECT IF(1<2,'yes','no');
        -> 'yes'
mysql> SELECT IF(STRCMP('test','test1'),'no','yes');

        -> 'no'
```

7.3 IFNULL

IFNULL(expr1,expr2)

If expr1 is not NULL, IFNULL() returns expr1; otherwise it returns expr2.

```
mysql> SELECT IFNULL(1,0);
        -> 1
mysql> SELECT IFNULL(NULL,10);
        -> 10
mysql> SELECT IFNULL(1/0,10);
        -> 10
mysql> SELECT IFNULL(1/0,'yes');

        -> 'yes'
```

The default return type of IFNULL(expr1,expr2) is the more "general" of the two expressions, in the order STRING, REAL, or INTEGER. Consider the case of a table based on expressions or where MySQL must internally store a value returned by IFNULL() in a temporary table:

```
mysql> CREATE TABLE tmp SELECT IFNULL(1,'test') AS test;
mysql> DESCRIBE tmp;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| test  | varbinary(4) | NO   |     |         |       |

+-------+--------------+------+-----+---------+-------+
```

7.4 NULLIF

NULLIF(expr1,expr2)

Returns NULL if expr1 = expr2 is true, otherwise returns expr1. This is the same as CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END.

The return value has the same type as the first argument.

```
mysql> SELECT NULLIF(1,1);
        -> NULL
mysql> SELECT NULLIF(1,2);
        -> 1
```

7.5 COALESCE

COALESCE(value,...)

Returns the first non-NULL value in the list, or NULL if there are no non-NULL values.

The return type of COALESCE() is the aggregated type of the argument types.

```
mysql> SELECT COALESCE(NULL,1);
        -> 1
mysql> SELECT COALESCE(NULL,NULL,NULL);
        -> NULL
```

8 EXERCISES

1. Suppose you are given a relation grade points(grade, points) that provides a conversion from letter grades in the takes relation to numeric scores; for example, an “A” grade could be specified to correspond to 4 points, an “A−” to 3.7 points, a “B+” to 3.3 points, a “B” to 3 points, and so on. The grade points earned by a student for a course offering (section) is defined as the number of credits for the course multiplied by the numeric points for the grade that the student received. Given the preceding relation, and our university schema, write each of the following queries in SQL. You may assume for simplicity that no takes tuple has the null value for grade.
    a. Find the total grade points earned by the student with ID '12345', across all courses taken by the student.
    b. Find the grade point average (GPA) for the above student, that is, the total grade points divided by the total credits for the associated courses.
    c. Find the ID and the grade-point average of each student.
    d. Now reconsider your answers to the earlier parts of this exercise under the assumption that some grades might be null. Explain whether your solutions still work and, if not, provide versions that handle nulls properly.
2. Write the following queries in SQL, using the university schema.
    a. Find the ID and name of each student who has taken at least one Comp. Sci. course; make sure there are no duplicate names in the result.
    b. Find the ID and name of each student who has not taken any course offered before 2017.
    c. For each department, find the maximum salary of instructors in that department. You may assume that every department has at least one instructor.
    d. Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query.
3. Write the SQL statements using the university schema to perform the following operations:
    a. Create a new course “CS-001”, titled “Weekly Seminar”, with 0 credits.
    b. Create a section of this course in Fall 2017, with sec id of 1, and with the location of this section not yet specified.
    c. Enroll every student in the Comp. Sci. department in the above section.
    d. Delete enrollments in the above section where the student’s ID is 12345.
    e. Delete the course CS-001. What will happen if you run this delete statement without first deleting offerings (sections) of this course?
    f. Delete all takes tuples corresponding to any section of any course with the word “advanced” as a part of the title; ignore case when matching the word with the title.

************************************END***

Birla Institute of Technology and Science, Pilani
CS F212 Database Systems
Lab No # 3

****

1 Introduction

In this lab, we will continue practicing SQL queries related to aggregate functions and joins on a schema of a DVD Store and university database. Schema for the sakila database is given below:

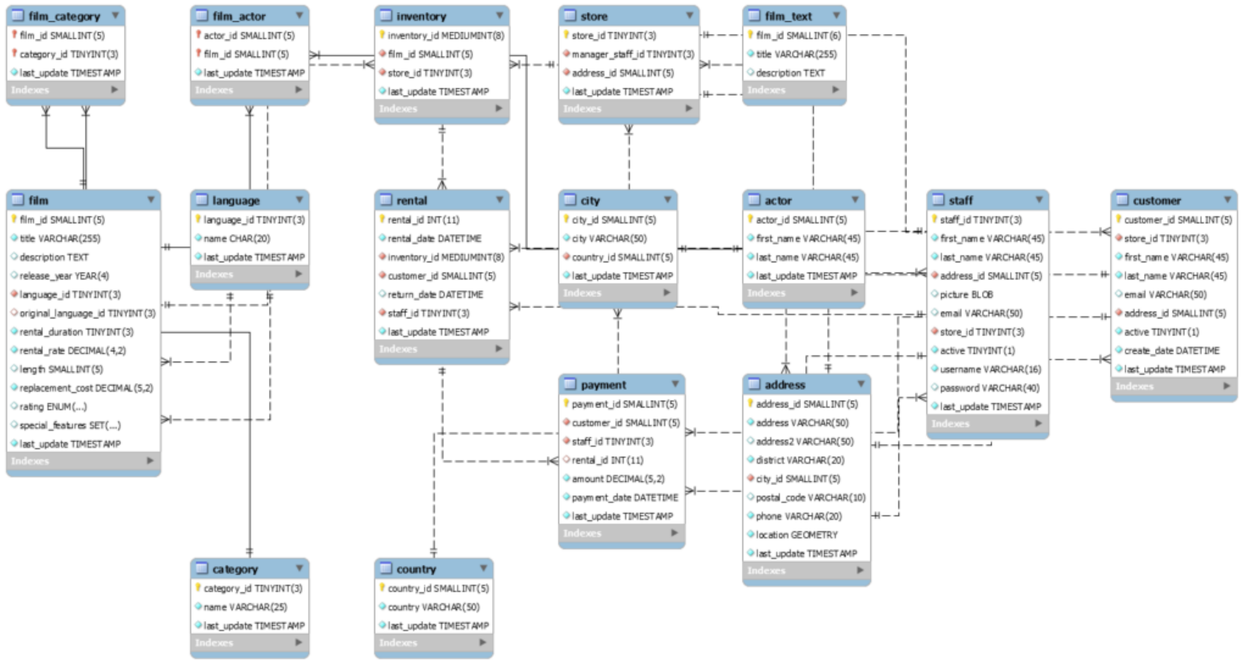


Figure:1 Sakila Schema

2 SQL Aggregate Functions

SQL aggregate functions return a single value, calculated from values in a column. The aggregate function when used in select clause, computation applies to set of all rows selected. They are commonly used in combination with GROUP BY and HAVING clauses.

MySQL's aggregate function is used to perform calculations on multiple values and return the result in a single value like the average of all values, the sum of all values, and maximum & minimum value among certain groups of values. We mostly use the aggregate functions with SELECT statements in the data query languages.

The following are the syntax to use aggregate functions in MySQL:

```
function_name (DISTINCT | ALL expression)
```

In the above syntax, we had used the following parameters:

1. First, we need to specify the name of the aggregate function.
2. Second, we use the DISTINCT modifier when we want to calculate the result based on distinct values or ALL modifiers when we calculate all values, including duplicates. The default is ALL.
3. Third, we need to specify the expression that involves columns and arithmetic operators.

There are various aggregate functions available in MySQL. Some of the most used aggregate functions are summarized in the below table:

Aggregate Function	Descriptions
count()	It returns the number of rows, including rows with NULL values in a group.
sum()	It returns the total summed values (Non-NULL) in a set.
average()	It returns the average value of an expression.
min()	It returns the minimum (lowest) value in a set.
max()	It returns the maximum (highest) value in a set.
groutp_concat()	It returns a concatenated string.

Examples:

Find out the average of credits from course table in university database.

```
select avg(course.credits) as 'Avg Credit'
from university.course;
```

Find out the maximum and minimum value for credits from course table in university database.

```
select max(course.credits) as 'Max Credit',
min(course.credits) as 'Min Credits'
from university.course;
```

Find out the number of courses in the course table in university database.

```
select count(course.credits) as '# Course'
from university.course;
```

2.1 Removing Repeating Data with DISTINCT before Aggregation

How do aggregate functions handle repeated values? By default, an aggregate function includes all rows, even repeats, with the noted exceptions of NULL. We can add the DISTINCT qualifier to remove duplicates prior to computing the aggregate function.

Count number of different departments and number of total courses in the course table.

```
select count(course.title) as 'NoCourse', count(distinct
course.dept_name) as 'NoDept'
from university.course;
```

2.2 Group Aggregation Using GROUP BY

Aggregate functions return a single value, so we usually cannot mix attributes and aggregate functions in the attribute list of a SELECT statement.

Find out the average course credits and number of courses for each department in course table.

```
select course.dept_name, avg(course.credits) as 'Avg Credit'
from university.course;
```

The above query results in error. Because there are many departments and only one average course credit, SQL doesn't know how to pair them.

The following query will mix literals and aggregates.

```
select course.dept_name, count(distinct course.course_id) as
'NoCourse',
avg(course.credits) as 'AvgCredit'
from university.course group by course.dept_name;
```

2.3 Removing Rows before Grouping with WHERE

We may want to eliminate some rows from the table before we form groups. We can eliminate rows from groups using the WHERE clause.

Example:

Find number of courses and average credits for all course in Math department from course table.

```
select course.dept_name, count(distinct course.course_id) as
'NoCourse',
avg(course.credits) as 'AvgCredit'
from university.course
where course.dept_name = 'Math'
group by course.dept_name;
```

2.4 Sorting Groups with ORDER BY

We can order our groups using ORDER BY. It works the same as ORDER BY without grouping except that we can now also sort by group aggregates. The aggregate we use in our sort criteria need not be an aggregate from the SELECT list.

Example:

Find out the average course credits and number of courses for each department in course table and sort them by name of department

```
select course.dept_name, count(distinct course.course_id) as
'NoCourse',
avg(course.credits) as 'AvgCredit'
from university.course
group by course.dept_name
order by course.dept_name;
```

2.5 Removing Groups with HAVING

Use the HAVING clause to specify a condition for groups in the final result. This is different from WHERE, which removes rows before grouping. Groups for which the HAVING condition does not evaluate to true are eliminated. Because we're working with groups of rows, it makes sense to allow aggregate functions in a HAVING predicate.

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

A. If a GROUP BY clause is specified, the HAVING clause is applied to the groups created by the GROUP BY clause.

B. If a WHERE clause is specified and no GROUP BY clause is specified, the HAVING clause is applied to the output of the WHERE clause and that output is treated as one group.
C. If no WHERE clause and no GROUP BY clause are specified, the HAVING clause is applied to the output of the FROM clause and that output is treated as one group.

Find out the average course credits and number of courses for each department having average course credits more than or equal to 3.5 in course table and sort them by name of department.

```
select course.dept_name, count(distinct course.course_id) as
'NoCourse',
avg(course.credits) as 'AvgCredit'
from university.course
group by course.dept_name
having avg(course.credits) >=3.5
order by course.dept_name;
```

2.6 Section Exercise

1 Find and print the average salary of each department in the University Database
2 Find the name and average total credits of departments where the minimum total credits of a student is 5.

3 Join – Retrieve data from multiple tables

3.1 SELF Join

The self-join is used to join a table to itself when using a join.

A self-join is useful when you want to combine records in a table with other records in the same table that match a certain join condition.

Consider the following example:

To retrieve all customers whose last name matches the first name of another customer. We achieve this by assigning aliases to the customer table while performing an inner join on the two aliases. The aliases allow us to join the table to itself because they give the table two unique names, which means that we can query the table as though it was two different tables.

```
SELECT
  a.customer_id,
  a.first_name,
  a.last_name,
  b.customer_id,
   b.first_name,
   b.last_name
FROM customer a
INNER JOIN customer b
ON a.last_name = b.first_name;
```

Self joins aren't limited to the INNER JOIN. You can also use a LEFT JOIN to provide all records from the left "table" regardless of whether there's a match on the right one.

Try the following query:

```
SELECT
  a.customer_id,
  a.first_name,
  a.last_name,
  b.customer_id,
  b.first_name,
  b.last_name
FROM customer a
LEFT JOIN customer b
ON a.last_name = b.first_name
ORDER BY a.customer_id;
```

And of course, you can also use a RIGHT JOIN to provide all records from the right "table" regardless of whether there's a match on the left one.

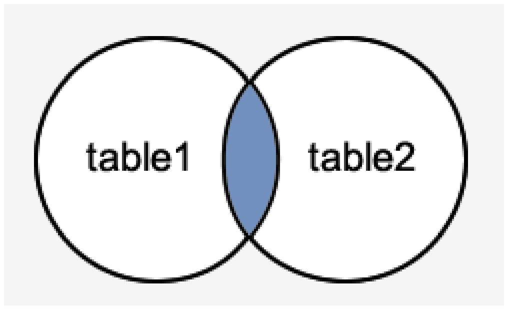
Try the following query:

```
SELECT
     a.customer_id,
     a.first_name,
     a.last_name,
     b.customer_id,
     b.first_name,
     b.last_name
FROM customer a
RIGHT JOIN customer b
ON a.last_name = b.first_name
ORDER BY b.first_name;
```

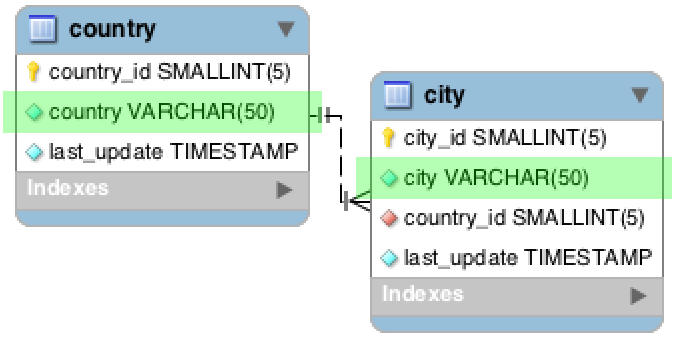
3.2 INNER Join

The INNER JOIN is used to return data from multiple tables. More specifically, the INNER JOIN is for when you're only interested in returning the records where there is at least one row in both tables that match the join condition.

We can understand it with the following visual representation where Inner Joins returns only the matching results from table1 and table2:



Consider the following tables in Sakila database:

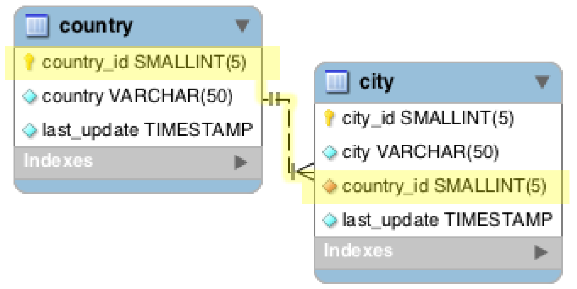


If we want to select data from the two highlighted fields (country and city), we could run the following query (which includes an inner join):

```
SELECT city, country
FROM city
INNER JOIN country ON
city.country_id = country.country_id;
```

In the above example, we use an inner join to display a list of cities alongside the country that it belongs to. The city info is in a different table to the country info. Therefore, we join the two tables using the country_id field — as that is a common field in both tables (it's a foreign key field).

Here's a diagram of those two tables (with the foreign key relationship highlighted).



Inner Joins with GROUP BY and Aggregate Functions:

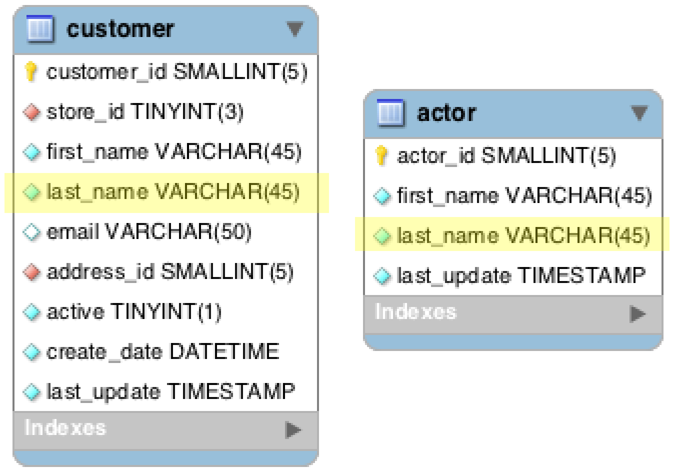
In the following example, we switch it around and provide a list of countries in one column, with the number of cities that each country contains in another column.

```
SELECT country, COUNT(city)
FROM country a
INNER JOIN city b
ON a.country_id = b.country_id
GROUP BY country;
```

3.3 Left Join

The LEFT JOIN is used to return data from multiple tables. In particular, the "LEFT" part means that all rows from the left table will be returned, even if there's no matching row in the right table. This could result in NULL values appearing in any columns returned from the right table.

Consider the following tables:



Let's return a list of all customers (from the customer table). And if the customer shares the same last name with an actor (from the actor table), let's display that actor's details too.

But the important thing is that we display all customers — regardless of whether they share their last name with an actor. Therefore, if a customer doesn't share the same last name as an actor, the customer is still listed.

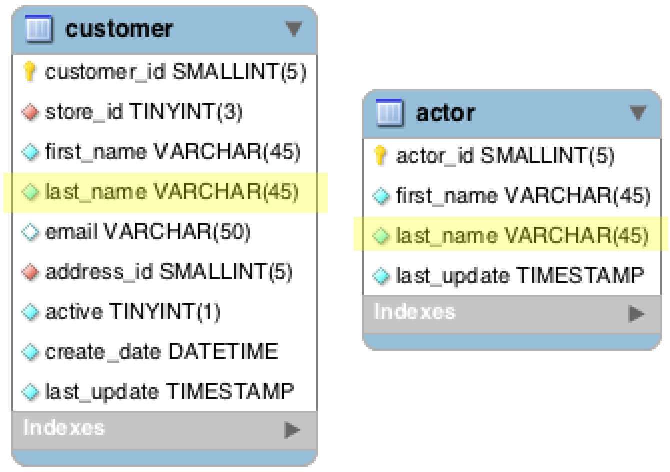
We could achieve that with the following query:

```
SELECT
     c.customer_id,
    c.first_name,
    c.last_name,
    a.actor_id,
    a.first_name,
    a.last_name
FROM customer c
LEFT JOIN actor a
ON c.last_name = a.last_name
ORDER BY c.last_name;
```

3.4 Right Join

The RIGHT JOIN is used to return data from multiple tables. In particular, the "RIGHT" part means that all rows from the right table will be returned, even if there's no matching row in the left table. This could result in NULL values appearing in any columns returned from the left table.

Consider the following tables:



Let's return a list of all actors (from the actor table). And if the actor shares the same last name with a customer (from the customer table), let's display that customer's details too.

But the important thing is that we display all actors — regardless of whether they share their last name with a customer. Therefore, if an actor doesn't share the same last name as a customer, the actor is still listed.

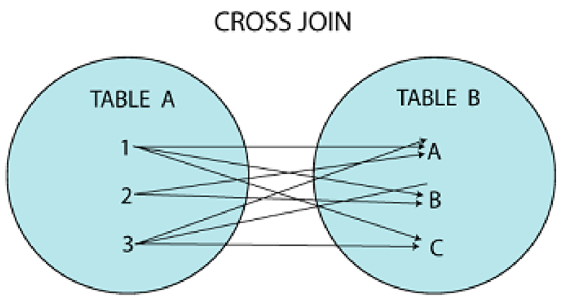
We could achieve that with the following query:

```
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    a.actor_id,
    a.first_name,
    a.last_name
FROM customer c
RIGHT JOIN actor a
ON c.last_name = a.last_name
ORDER BY a.last_name;
```

3.5 CROSS Join

MySQL CROSS JOIN is used to combine all possibilities of the two or more tables and returns the result that contains every row from all contributing tables. The CROSS JOIN is also known as CARTESIAN JOIN, which provides the Cartesian product of all associated tables. The Cartesian product can be explained as all rows present in the first table multiplied by all rows present in the second table. It is like the Inner Join, where the join condition is not available with this clause.

We can understand it with the following visual representation where CROSS JOIN returns all the records from table1 and table2, and each row is the combination of rows of both tables.



The CROSS JOIN keyword is always used with the SELECT statement and must be written after the FROM clause. The following syntax fetches all records from both joining tables:

```
SELECT column-lists
```

```
FROM table1
CROSS JOIN table2;
```

Example: Try the following query. It returns all rows from course table, repeating each row for number of rows in department table.

```
select * from university.course cross join
university.department;
```

3.6 NATURAL Join

When we combine rows of two or more tables based on a common column between them, this operation is called joining. A natural join is a type of join operation that creates an implicit join by combining tables based on columns with the same name and data type. It is like the INNER or LEFT JOIN, but we cannot use the ON or USING clause with natural join as we used in them.

Points to remember:

A. There is no need to specify the column names to join.
B. The resultant table always contains unique columns.
C. It is possible to perform a natural join on more than two tables.
D. Do not use the ON clause.

The following is a basic syntax to illustrate the natural join:

```
SELECT [column_names | *]
FROM table_name1
NATURAL JOIN table_name2;
```

Example:

Show the list of courses from table course along with department details from department table.

```
select * from university.course
natural join university.department;
```

3.7 Section Exercise

Use the University schema for:

1. Find all instructors earning the highest salary (there may be more than one with the same salary).
2. Find the enrollment of each section that was offered in Fall 2017.

3. Find the maximum enrollment, across all sections, in Fall 2017.

4 UNION and UNION ALL

MySQL Union clause allows us to combine two or more relations using multiple SELECT queries into a single result set. By default, it has a feature to remove the duplicate rows from the result set.

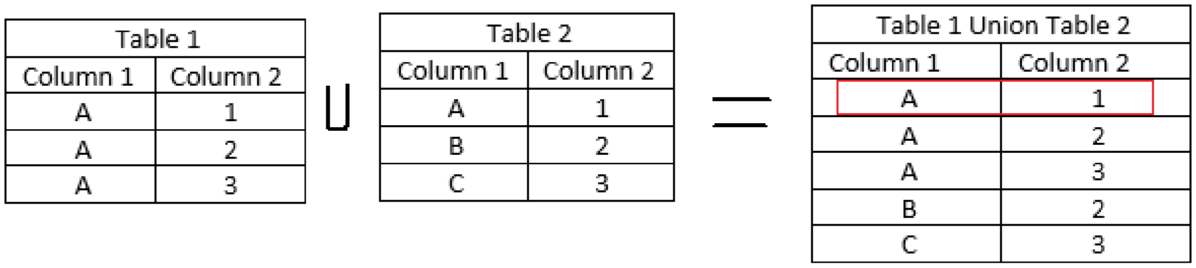


Table 1	
Column 1	Column 2
A	1
A	2
A	3

∪

Table 2	
Column 1	Column 2
A	1
B	2
C	3

=

Table 1 Union Table 2	
Column 1	Column 2
A	1
A	2
A	3
B	2
C	3

Union clause in MySQL must follow the rules given below:

A. The order and number of the columns must be the same in all tables.
B. The data type must be compatible with the corresponding positions of each select query.
C. The column name in the SELECT queries should be in the same order.

Syntax:

```
SELECT column_name(s) FROM table_name1
UNION
SELECT column_name(s) FROM table_name2;
```

Example:

```
select * from university.course
union
select * from university.department;
```

The above query gives an error because name and number of columns are different in both tables. The UNION clause combine data of two or tables of same type and remove duplicates.

```
select * from university.course
union
select * from university2.course;
```

The above query combine data of course table from two different university.

This operator returns all rows by combining two or more results from multiple SELECT queries into a single result set. It does not remove the duplicate rows from the result set.

We can understand it with the following pictorial representation:

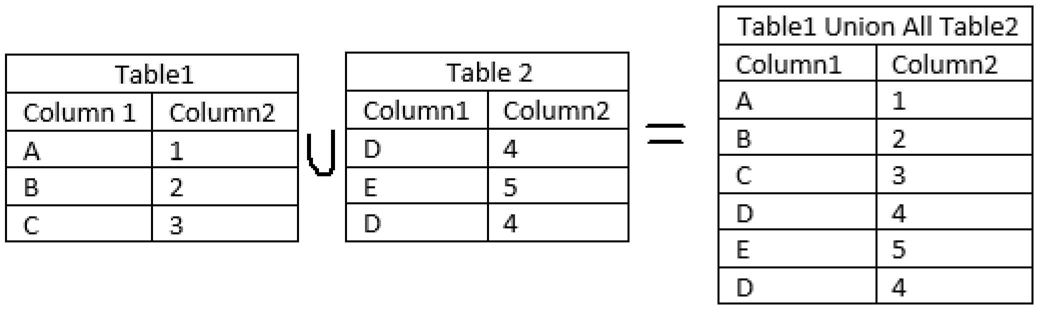


Table1	
Column 1	Column2
A	1
B	2
C	3

$\cup$

Table 2	
Column1	Column2
D	4
E	5
D	4

$=$

Table1 Union All Table2	
Column1	Column2
A	1
B	2
C	3
D	4
E	5
D	4

4.1 Section Exercise

Q1) You have 2 tables Customer1, Customer2, both with columns City and Country. Write a query that returns the Indian Cities (only distinct values) from both tables.

Q2) Assume that apart from union, intersection and difference (the same definition as that in set theory also exist in MySQL. Using them can you find the entries belonging to exactly 1 table of 2 given tables? If yes, write the query for the same with tables univ1.courses and univ2.courses.

5 Points to remember

- ✓ The difference between Union and Union All operators is that "Union" returns all distinct rows (eliminate duplicate rows) from two or more tables into a single output. In contrast, "Union All" returns all the rows, including duplicated rows.
- ✓ In MySQL, JOIN, CROSS JOIN, and INNER JOIN are syntactic equivalents (they can replace each other). In standard SQL, they are not equivalent. INNER JOIN is used with an ON clause, CROSS JOIN is used otherwise.
- ✓ A USING clause can be rewritten as an ON clause that compares corresponding columns. However, although USING and ON are similar, they are not quite the same. Consider the following two queries:

```
a LEFT JOIN b USING (c1, c2, c3)
```

```
a LEFT JOIN b ON a.c1 = b.c1 AND a.c2 = b.c2 AND a.c3 = b.c3
```

With respect to determining which rows satisfy the join condition, both joins are semantically identical.

With respect to determining which columns to display for SELECT * expansion, the two joins are not semantically identical. The USING join selects the coalesced value of corresponding columns, whereas the ON join selects all columns from all tables. For the USING join, SELECT * selects these values:

```
COALESCE(a.c1, b.c1), COALESCE(a.c2, b.c2), COALESCE(a.c3, b.c3)
```

For the ON join, SELECT * selects these values:

```
a.c1, a.c2, a.c3, b.c1, b.c2, b.c3
```

With an inner join, COALESCE(a.c1, b.c1) is the same as either a.c1 or b.c1 because both columns have the same value. With an outer join (such as LEFT JOIN), one of the two columns can be NULL. That column is omitted from the result.

6 Exercise

1. Write the following queries in SQL, using the university schema.
    a. Find the names of all students who have taken at least one Comp. Sci. course; make sure there are no duplicate names in the result.
    b. Find the IDs and names of all students who have not taken any course offering before Spring 2009.
    c. For each department, find the maximums alary of instructors in that department. You may assume that every department has at least one instructor.
    d. Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query.
2. Write the following queries in SQL, using the university schema.
    a. Create a new course "CS-001", titled "Weekly Seminar", with 0 credits.
    b. Create a section of this course in Autumn 2009, with section id of 1.
    c. Enroll every student in the Comp. Sci. department in the above section.
    d. Delete enrollments in the above section where the student's name is Chavez.
    e. Delete the course CS-001. What will happen if you run this delete statement without first deleting offerings (sections) of this course.
    f. Delete all takes tuples corresponding to any section of any course with the word "database" as a part of the title; ignore case when matching the word with the title.

***********************************END**

Birla Institute of Technology and Science, Pilani
CS F212 Database Systems
Lab No # 4

****

1 Views

1.1 Introduction

A view is a virtual table defined by a query. It provides a mechanism to create alternate ways of working with the data in a database. A view acts much like a table. We can query it with a SELECT, and some views even allow INSERT, UPDATE, and DELETE. There are several uses for a view.

- **Usability—**We can use a view as a wrapper around very complex SELECT statements to make our system more usable.
- **Security—**If we need to restrict the data a user can access, we can create a view containing only the permitted data. The user is given access to the view instead of the base table(s).
- **Reduced Dependency—**The database schema evolves over time as our enterprise changes. Such changes can break existing applications that expect a certain set of tables with certain columns. We can fix this by having our applications access views rather than base tables. When the base tables change, existing applications still work if the views are correct.

1.2 Creating Views

```
CREATE VIEW <view name> [(<column list>)] AS <SELECT statement>
```

This creates a view named <view name>. The column names/types and data for the view are determined by the result table derived by executing <SELECT statement>. Optionally, we can specify the column names of the view in <column list>. The number of columns in <column list> must match the number of columns in the <SELECT statement>.

Now let us see an example, try out the following query which creates a view called stview, which has ID, name, and tot_cred columns and tot_cred is greater than 50 from the student table of university2 database.

```
create view stview as
select student.ID, student.name, student.tot_cred
from university2.student
where student.tot_cred>50;
```

1.3 Select Query on Views

Selecting from views is just like selecting from a table. View is just a virtual table.

```
select * from stview
order by stview.tot_cred;
```

The above query selects all rows from the view stview. We can also use any clause or conditions as in tables.

Notice that the above query gives the same result when we use the original tables in the query without using the view.

```
select student.ID, student.name, student.tot_cred
from university2.student
where student.tot_cred>50
order by student.tot_cred;
```

Because the view is just a virtual table, any changes to the base tables are instantly reflected in the view data.

See the following query updates the tot_red and these changes are reflected the view stview.

```
update university2.student
set student.tot_cred = 100
where student.ID = 54321;
```

Now try following query and check the effect.

```
select * from stview
order by stview.tot_cred;
```

Now, lets see another example which shows how to create view on a complex query.

Create a view, stu_info with ID, name, tot_cred, from student table and corresponding grade from takes table of university schema. If any student does not have any grade, then fill null.

```
create view stu_info as
select student.ID, student.name, student.tot_cred, takes.grade
from university.student left join
university.takes
on student.ID = takes.ID
where student.tot_cred>30
order by student.tot_cred;
```

Now retrieve all rows from stu_info view.

```
select * from stu_info;
```

Now, we can use stu_info as a new table that can retrieve information directly from this view. Try the following query.

```
select name, grade from stu_info;
```

Notice that the above queries would have been more complex without creating views.

1.4 Restrictions on Views

Restrictions on DML operations for views use the following criteria in the order listed:

- If a view is defined by a query that contains SET or DISTINCT operators, a GROUP BY clause, or a group function, then rows cannot be inserted into, updated in, or deleted from the base tables using the view.
- If a view is defined with WITH CHECK OPTION, a row cannot be inserted into, or updated in, the base table (using the view), if the view cannot select the row from the base table.
- If a NOT NULL column that does not have a DEFAULT clause is omitted from the view, then a row cannot be inserted into the base table using the view.
- If the view was created by using an expression, such as DECODE(deptno, 10, "SALES", ...), then rows cannot be inserted into or updated in the base table using the view.

1.5 Updating a Join View

An updatable join view (also referred to as a modifiable join view) is a view that contains more than one table in the top-level FROM clause of the SELECT statement and is not restricted by the WITH READ ONLY clause.

The rules for updatable join views are shown in the following table. Views that meet these criteria are said to be inherently updatable.

Rule	**Description**
General Rule	Any INSERT, UPDATE, or DELETE operation on a join view can modify only one underlying base table at a time.
UPDATE Rule	All updatable columns of a join view must map to columns of a key-preserved table*. If the view is defined with the WITH CHECK OPTION clause, then all join columns and all columns of repeated tables are not updatable.

DELETE Rule	Rows from a join view can be deleted as long as there is exactly one key-preserved table in the join. The key preserved table can be repeated in the FROM clause. If the view is defined with the WITH CHECK OPTION clause and the key preserved table is repeated, then the rows cannot be deleted from the view.
INSERT Rule	An INSERT statement must not explicitly or implicitly refer to the columns of a non-key preserved table. If the join view is defined with the WITH CHECK OPTION clause, INSERT statements are not permitted.

- The concept of a key-preserved table is fundamental to understanding the restrictions on modifying join views. A table is key preserved if every key of the table can also be a key of the result of the join. So, a key-preserved table has its keys preserved through a join.
- Finally, even if a view is updatable, not all columns within the view may be updatable.
- It is important to note that updates through views can have unexpected consequences, depending on the behavior of a DBMS.
- In general, updates through views work best when the view is defined as a subset of a table and all attributes that determine if a row is in a view are updatable.

Note: the update operations supported on views are generally specific to the SQL vendor.

Example: We have a view st_view which has all columns from course table and another view stu_info, which has some columns from course and takes table. Now try the following query:

```
insert into stu_info (ID, name, tot_cred)
values (987456, 'upendra', 100);



insert into st_view (ID, name, dept_name, tot_cred)
values (98745, 'upendra', 'Comp. Sci.', 100);
```

And evaluates the differences in result.

To know more updatability and restriction on views go through the following links.

https://dev.mysql.com/doc/refman/8.0/en/view-restrictions.html

https://dev.mysql.com/doc/refman/8.0/en/view-updatability.html

1.6 Alter View

Altering view can also be done by dropping it and recreating it. but that would drop the associated granted permissions on that view. By using alter clause, the permissions are preserved.

```
ALTER VIEW [ schema_name . ] view_name [ ( column [ ,...n ] ) ]
[ WITH <view_attribute> [ ,...n ] ]
AS select_statement
[ WITH CHECK OPTION ] [ ; ]
<view_attribute> ::=
{
    [ ENCRYPTION ]
    [ SCHEMABINDING ]
    [ VIEW_METADATA ]       }
```

Try the following query to alter stu_info view and retrieve all rows from stu_info view.

```
alter view stu_info as
select  student.ID,  student.name,  student.tot_cred,  takes.grade,
takes.year
from university2.student left join
university2.takes
on student.ID = takes.ID
where student.tot_cred>30
order by student.tot_cred;



select * from stu_info;
```

1.7 Drop View

This removes the specified view; however, it does not change any of the data in the database. SQL does not allow a view to be dropped if view is contained in the SELECT statement of another view. This is default behavior and is equivalent to the effect of using RESTRICT Option. If we want to drop a view which is being used in a select command, we must use the CASCADE option.

```
DROP VIEW <view name> [CASCADE | RESTRICT];

drop view st_view;
```

The above query drop the st_view.

1.8 Section Exercise

Q) Use the sakila schema for

a) Create a view for finding the average running time of films by category.
b) Create a view to find out the current due date for “Academy Dinosaur”, see how the results of this view change after updating the database.
c) Create a view to display the total amount rung up by each staff member in August of 2005.

2 Derived Attributes

Data can be somewhat raw. SQL provides the ability to create attributes in our result table that are derived using operations and functions over existing attributes and literal. The default column name of a derived attribute is system-dependent; however, a name can be assigned using a column alias. We will discuss some basic functions available in MySQL.

2.1 Numeric

SQL can evaluate simple arithmetic expressions containing numeric columns and literals. The following table shows the SQL arithmetic operators in precedence order from highest to lowest. Unary +/- have the highest precedence. Multiplication and division have the next highest precedence. Addition and subtraction have the lowest precedence. Operators with the same precedence are executed left to right. We can control the order of evaluation using parentheses. SQL evaluates an expression for each row in the table specified in the FROM clause that satisfies the condition in the WHERE clause. Let’s look at an example.

Example: Show the student name, tot_cred along with the percentage of credit obtained from table student. Assume that the total credits are 150.

```
select name, tot_cred, tot_cred/150*100 as percent
from student
where tot_cred>=40;
```

SQL also includes many standard mathematic functions. Table below contains some of the more common functions. The exact set of available functions is DBMS dependent. In fact, your DBMS will likely have additional functions.

Function	**Returns**	**Example**
ABS(N)	Absolute value of N	ABS(inventory – 100)
CEIL[ING](N)	Ceiling of N	CEILING(inventory/10)
EXP(N)	eN	EXP(5)
FLOOR(N)	Floor of N	FLOOR(inventory/10)
LOG(N)	Natural log of N	LOG(5)
N%D	Remainder of N divided by D	11%3 (--returns 2)

POWER(B, E)	B to the power of E (BE)	POWER(2, 3)
SQRT(N)	√N	SQRT(4)

- What about the infamous NULL? An arithmetic expression evaluated with NULL for any value returns NULL. Arithmetic functions are given a NULL parameter value return NULL.

2.2 Character String

The typical database is full of character strings, such as names, addresses, and ingredients. The string you really want may be a combination of data strings, substrings, string literals, and so on. Perhaps you want to generate address labels or salutations (e.g., "Dear first name last name,"). SQL provides a wide range of mechanisms for combining and manipulating character strings.

2.2.1 Concatenating Strings With CONCAT(str1,str2,...)

Returns the string that results from concatenating the arguments. May have one or more arguments. If all arguments are nonbinary strings, the result is a nonbinary string. If the arguments include any binary strings, the result is a binary string. A numeric argument is converted to its equivalent nonbinary string form.

Note: CONCAT() returns NULL if any argument is NULL.

Example: Retrieve address from address table in proper format.

```
select   CONCAT(address.address,'  Dist.  ',address.district,'  PIN:
',address.postal_code) as mail
from sakila.address;
```

For quoted strings, concatenation can be performed by placing the strings next to each other:

```
SELECT 'My' 'S' 'QL';
```

2.2.2 SUBSTRING: Getting the String within the String

```
SUBSTRING(str,pos), SUBSTRING(str FROM pos), SUBSTRING(str,pos,len),
SUBSTRING(str FROM pos FOR len)
```

The forms without a len argument return a substring from string str starting at position pos. The forms with a len argument return a substring len characters long from string str, starting at position pos. The forms that use FROM are standard SQL syntax. It is also possible to use a negative value for pos. In this case, the beginning of the substring is pos characters from the end of the string, rather than the beginning. A negative value may be used for pos in any of the forms of this function. A value of 0 for pos returns an empty string.

Example: List all actors name by taking initial letter with '.' From first name and attach last name with a space between.

```
select concat(substring(actor.first_name, 1, 1),'. ', actor.last_name)
as Actor_Name from sakila.actor limit 10;
```

2.2.3 TRIM: Removing Unwanted Leading and Trailing Characters

```
TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str), TRIM([remstr
FROM] str)
```

Returns the string str with all remstr prefixes or suffixes removed. If none of the specifiers BOTH, LEADING, or TRAILING is given, BOTH is assumed. remstr is optional and, if not specified, spaces are removed.

LTRIM(str):

Returns the string str with leading space characters removed.

RTRIM(str):

Returns the string str with trailing space characters removed.

Try the following queries.

```
select '     Singh      ';
select ltrim('     Singh      ') as l_trim;
select rtrim('     Singh      ') as r_trim;
select trim('     Singh      ') as trim;
SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx') as trim_exam;
SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx') as trim_exam;
SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz') as trim_exam;
```

2.2.4 UPPER and LOWER: Controlling Character Case

UPPER(str)

Returns the string str with all characters changed to uppercase according to the current character set mapping. The default is utf8mb4.

```
select UPPER(CONCAT(address.address,' Dist. ',address.district,' PIN:
',address.postal_code)) as mail
from sakila.address;
```

LOWER(str)

Returns the string str with all characters changed to lowercase according to the current character set mapping. The default is utf8mb4.

```
select LOWER(CONCAT(address.address,' Dist. ',address.district,' PIN:
',address.postal_code)) as mail
from sakila.address;
```

2.2.5 CHAR_LENGTH: Counting Characters in a String

CHAR_LENGTH(str)

Returns the length of the string str, measured in characters. A multibyte character counts as a single character. This means that for a string containing five 2-byte characters, LENGTH() returns 10, whereas CHAR_LENGTH() returns 5.

```
select char_length(last_name) from sakila.actor limit 10;
```

Note: CHARACTER_LENGTH() is a synonym for CHAR_LENGTH().

2.2.6 The REPLACE Function

REPLACE(str,from_str,to_str)

Returns the string str with all occurrences of the string from_str replaced by the string to_str. REPLACE() performs a case-sensitive match when searching for from_str.

```
mysql> SELECT REPLACE('www.mysql.com', 'w', 'Ww');
```

```
        -> 'WwWwWw.mysql.com'
```

Note: for more on MySQL string functions go through the following link-

https://dev.mysql.com/doc/refman/8.0/en/string-functions.html

2.3 Temporal

Time is no simple concept. To deal with this complexity, SQL provides a wide range of techniques for operating on temporal data types. Unfortunately, temporal types and arithmetic are not supported by all DBMSs. Consult your documentation for the exact limitations and syntax of your system.

Let’s start working with dates with the following query:

```
SELECT CURDATE();
```

This gives the current date of the system. Note this query only returns the current date, to retrieve complete details you may run the following query:

```
SELECT DATE_FORMAT(CURDATE(), '%W %D %M %Y');
```

2.3.1 Finding the Current Date or Time

Let's start with the basic question: "What time is it?" SQL provides several functions to help you find out.

Temporal Function	Description	Return Type
CURRENT_TIME[(precision)]	Current time with time zone displacement	TIME WITH TIMEZONE
CURRENT_DATE	Current date	DATE
CURRENT_TIMESTAMP [(precision)]	Current date and time with timezone displacement	TIMESTAMP WITH TIMEZONE
LOCALTIME[(precision)]	Current time	TIME WITHOUT TIMEZONE
LOCALTIMESTAMP[(precision)]	Current date	TIMESTAMP WITHOUT TIMEZONE

The optional precision argument specifies the fractional seconds precision. LOCALTIME and LOCALTIMESTAMP are not widely implemented.

2.3.2 Using Arithmetic Operators with Temporal Types

SQL allows the use of basic arithmetic operators with temporal data types. For example, you may want to know the date 3 days from now. To get that, you can add the current date to an interval of 3 days. Of course, not all arithmetic operations make sense with temporal data. For example, dividing a date by an interval makes no sense so it is not allowed by SQL. The operation must make sense. Subtracting a DATE value from a TIME value has no meaning so it is not allowed by SQL. Let's look at examples.

```
SELECT ADDDATE(CURDATE(), 30);
SELECT SUBDATE(CURDATE(), 30);
SELECT ADDTIME(CURTIME(), '02:00:00');
SELECT SUBTIME(CURTIME(), '02:00:00');
```

2.3.3 EXTRACT(unit FROM date): Getting Fields from Temporal Data

The EXTRACT() function uses the same kinds of unit specifiers as DATE_ADD() or DATE_SUB(), but extracts parts from the date rather than performing date arithmetic.

```
SELECT EXTRACT(YEAR_MONTH FROM '2019-07-02 01:02:03');
SELECT EXTRACT(DAY_MINUTE FROM '2019-07-02 01:02:03');
SELECT EXTRACT(MICROSECOND FROM '2003-01-02 10:30:00.000123');
```

2.3.4 Other Functions for Working with Dates

To know more on temporal functions, go through the following linkhttps://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html

2.4 Section Exercise

Q) Write a query to find the primary key of any table from the university database and add a column to the table which is the concatenation of all the primary key columns and make this column the primary key. Think about the pros and cons of this approach of redefining the primary key for each table.

Q) Use the sakila database for

a) Update the "orders" table by adding a new field "time(India)" which is 5:30 hrs ahead of the entries in "the time" column.
b) Round off the prices in "price" column of the items table using only the functions mentioned above in the table.

3 Exercise

1. Show how to define a view tot credits (year, num credits), giving the total number of credits taken by students in each year.

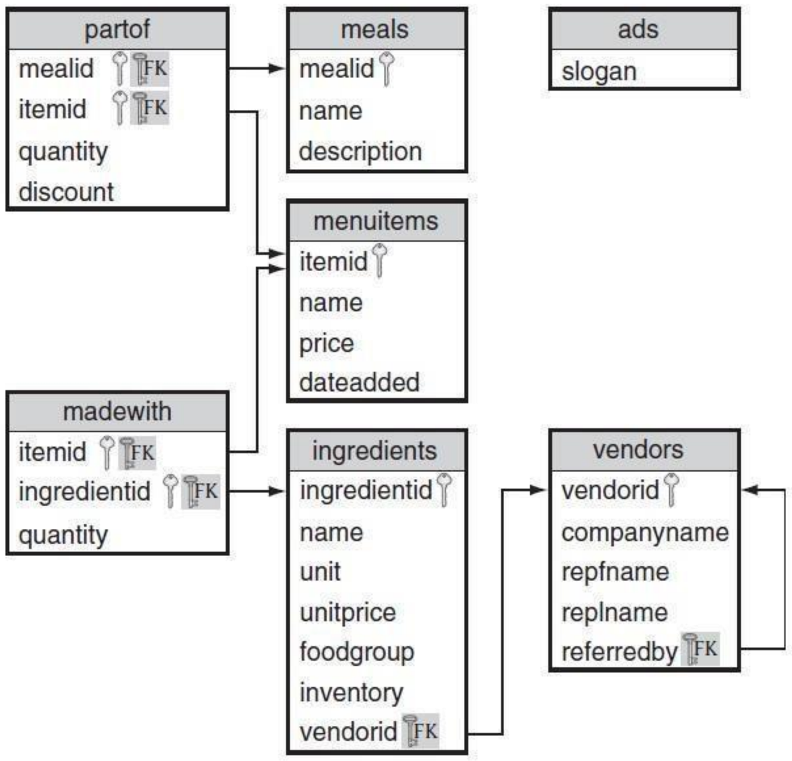
************************************END**

Birla Institute of Technology and Science, Pilani
CS F212 Database Systems
Lab No # 5

****

1 Sub-Queries

In this lab, we will continue practicing SQL queries related to subqueries on a schema of a DVD Store and a University created and populated in the previous labs. We also perform some examples on restaurant schema. The table schema for the restaurant is given below.



So far, we've seen joins and set operators for combining tables together. SQL provides another way to combine tables. You can nest queries within queries. Such an embedded query is called a subquery. A subquery computes results that are then used by an outer query. Basically, a subquery acts like any other expression we've seen so far. A subquery can be nested inside the SELECT, FROM, WHERE, and HAVING clauses. You can even nest subqueries inside another subquery.

Example: Find store manager's mailing address of store 2.

```
select concat(address.address,', Dist. - ',address.district,'
PIN - ', address.postal_code) as address
from sakila.address
where address.address_id =
(select store.address_id from sakila.store
where store.manager_staff_id=2);
```

Some points to remember when using subqueries:

- SQL marks the boundaries of a subquery with parentheses.
- Only the columns of the outermost query can appear in the result table. When creating a new query, the outermost query must contain all the attributes needed in the answer.
- There must be some way of connecting the outer query to the inner query. All SQL comparison operators work with subqueries.
- Subqueries are restricted in what they can return. First, the row and column count must match the comparison operator. Second, the data types must be compatible.

1.1 Subqueries using IN

In the above query, the = operator makes the connection between the queries. Because = expects a single value, the inner query may only return a result with a single attribute and row. If the subquery returns multiple rows, we must use a different operator. We use the IN operator, which expects a subquery result with zero or more rows.

Example 1: List all customer who done a payment with staff id =2.

```
select concat(customer.first_name,' ', customer.last_name) as
CustName
from sakila.customer
where customer.customer_id in
(select payment.customer_id from sakila.payment
where payment.staff_id=2);
```

Example 2: List name of the student from student table who takes a course in Fall semester.

```
select student.name from university2.student
where student.ID IN
(select takes.id from university2.takes
where takes.semester = 'Fall');
```

1.2 Subqueries Using NOT IN

IN over an empty table always returns false, therefore NOT IN always returns true.

Example: List name of the student from student table who does not take a course in Fall semester.

```
select student.name from university2.student
where student.ID NOT IN
(select takes.id from university2.takes
where takes.semester = 'Fall');
```

1.3 Subqueries using BETWEEN

Subqueries can even be combined with other predicates in the WHERE clause, including other Subqueries. Look at the following query.

Example: Select instructor's name and ID whose salary is 5000 less or more than the average salary of all instructors in the university.

```
select instructor.ID, instructor.name
from university.instructor
where instructor.salary between
(select avg(instructor.salary)-5000
from university.instructor)
and
(select avg(instructor.salary)+5000
from university.instructor);
```

1.4 Subqueries with Empty Results

What happens when a subquery result is empty? The answer depends on what SQL is expecting from the subquery. Let's look at a couple of examples. Remember that the standard comparison operators expect a scalar value; therefore, SQL expects that any subqueries used with these operators always return a single value. If the subquery result is empty, SQL returns NULL.

```
select student.name from university.student
where student.ID =
(select takes.id from university.takes
where takes.semester = 'No Semester');
```

Here the subquery results are empty, so the subquery returns NULL. Evaluating the outer query, student.ID = NULL returns unknown for each row in vendors. Consequently, the outer query returns an empty result table.

When using the IN operator, SQL expects the subquery to return a table. When SQL expects a table from a subquery and the subquery result is empty, the subquery returns an empty table. IN over an empty table always returns false, therefore NOT IN always returns true.

```
select student.name from university.student
where student.ID NOT IN
```

```
(select takes.id from university.takes
where takes.semester = 'No Semester');
```

The above subquery returns a table with zero rows (and one column). For every row in the student table, NOT IN returns true over the subquery, so every row is returned.

1.5 Section exercise

Q) Use the sakila schema for

a) Write a query to get the titles of movies starting with the letter ‘K’ or ‘A’ and the language is English.
b) Write a query to get all the actors’ full names who appears in the film “Alone Trip”.

2 More on Sub-Queries

2.1 Multilevel Subquery Nesting

Example: Find the list of courses from the course table, whose instructors get more salaries than the average salary of the university.

```
select course.course_id, course.title, course.credits
from university.course
where course.course_id in
(select teaches.course_id
from university.teaches, university.instructor
where teaches.ID in
(select avgsal.id from
(select instructor.id, instructor.salary
from university.instructor
having instructor.salary >
(select avg(instructor.salary) from university.instructor))
avgsal));
```

- GROUP BY, HAVING can be used in subqueries as well. Using ORDER BY in a subquery makes little sense because the order of the results is determined by the execution of the outer query.

2.2 Combining JOIN and Subqueries

- Nested queries are not restricted to a single table.

Example: Find the list of instructor's names, department, salary, section, semester, course title, and credits whose salary is more than the average salary of university.

```
select instructor.name, instructor.dept_name, instructor.salary,
teaches.sec_id, teaches.semester,
course.title, course.credits
from instructor natural join teaches
natural join course
having instructor.salary >
(select avg(instructor.salary) from university.instructor);
```

2.3 Standard Comparison Operators with Lists Using ANY, SOME, or ALL

We can modify the meaning of the SQL standard comparison operators with ANY, SOME, and ALL so that the operator applies to a list of values instead of a single value.

Using ANY or SOME:

- The ANY or SOME modifiers determine if the expression evaluates to true for at least one row in the subquery result.

Find all items that have a price that is greater than any salad item.

```
SELECT name
FROM items  WHERE price > ANY
(SELECT price
FROM items
WHERE name LIKE '%Salad');
```

Find all ingredients not supplied by Veggies_R_Us or Spring Water Supply

```
SELECT name
FROM ingredients
WHERE ingredientid NOT IN
(SELECT ingredientid
FROM ingredients  WHERE vendorid = ANY
(SELECT vendorid
FROM vendors
WHERE companyname = 'Veggies_R_Us' OR companyname
= 'Spring Water Supply'));
```

- Be aware of the difference between <> ANY and NOT IN. x <> ANY y returns true if any of the values in y are not equal to x. x NOT IN y returns true only if none of the values in y are equal to x or if the list y is empty.

Using ALL:

- The ALL modifier determines if the expression evaluates to true for all rows in the subquery result.

Find all ingredients that cost at least as much as every ingredient in a salad.

```
SELECT name
FROM ingredients
WHERE unitprice >= ALL
(SELECT unitprice
FROM ingredients ing JOIN madewith mw on
mw.ingredientid=ing.ingredientid JOIN
items i on mw.itemid=i.itemid WHERE i.name LIKE '%Salad');
```

Find the name of all ingredients supplied by someone other than Veggies_R_Us or Spring Water Supply.

```
SELECT name  FROM ingredients
WHERE vendorid <> ANY
(SELECT vendorid
FROM vendors
WHERE companyname = 'Veggies_R_Us' OR  companyname = 'Spring
Water Supply');
```

- Correct query:

```
SELECT name
FROM ingredients
WHERE ingredientid NOT IN (SELECT ingredientid FROM ingredients
WHERE vendorid = ANY
(SELECT vendorid FROM vendors  WHERE companyname =
'Veggies_R_Us' OR  companyname = 'Spring Water Supply'));
```

Find the most expensive items.

```
SELECT *
FROM items WHERE price >= ALL
(SELECT price FROM items);
```

What is wrong with the above query? Correct query:

```
SELECT *
FROM items WHERE price >= ALL
(SELECT price FROM items where price is not null);


SELECT *
```

```
FROM items WHERE price >= ALL
(SELECT max(price) FROM items);
```

- You might be tempted to believe that >= ALL is equivalent to >= (SELECT MAX()), but this is not correct. What's going on here? Recall that for ALL to return true, the condition must be true for all rows in the subquery. NULL prices evaluate to the unknown; therefore, >= ALL evaluates to unknown, so the result is empty. Of course, we can solve this problem by eliminating NULL prices. However, NULL isn't the only problem. What happens when the subquery result is empty?
- A common mistake is to assume that the = ALL operator returns true if all the rows in the outer query match all of the rows in the inner query. This is not the case. A row in the outer query will satisfy the = ALL operator only if it is equal to all the values in the subquery. If the inner query returns multiple rows, each outer query row will only satisfy the = ALL predicate if all rows in the inner query the same value and that value have equaled the outer query row value. Notice that the exact same result is achieved by using = alone and ensuring that only a distinct value is returned by the inner query.

2.4 Division Operation

One of the most challenging types of queries for SQL is one that compares two groups of rows to see if they are the same. These types of queries arise in many different applications. Some examples are as follows:

- Has a student taken all the courses required for graduation?
- List the departments and projects for which that department has all the tools required for the project.
- Find all the items that contain all the ingredients provided by a vendor.

There are two approaches.

1. Using set operations
2. Using set cardinality.

- The cardinality of a relation is the number of tuples in that set.

Suppose A is a set of customers and B is a set of payments. To check whether a customer has done or not done a payment on the DVD store, compare two sets A and B.

Example 1: List all the customers who have done any payment DVD store.

```
select * from customer
 where exists
(select * from payment
where payment.customer_id = customer.customer_id);
```

Example 2: List all the customers who have not done any payment DVD store.

```
select * from customer
 where not exists
(select * from payment
where payment.customer_id = customer.customer_id);
```

- If a subquery returns any rows at all, EXISTS subquery is TRUE, and the NOT EXISTS subquery is FALSE.

2.5 Comparing relational cardinality

Comparing relational cardinality: In this case, we compare whether for every combination of A and B whether the number of elements in A and B is equal or not.

Find all items and vendors such that all ingredients in the item are supplied by that vendor.

```
SELECT i.name, companyname
FROM items i, vendors v
WHERE
(SELECT COUNT(DISTINCT m.ingredientid) -- number of ingredients
in item
FROM madewith m
WHERE i.itemid = m.itemid)
= -- number of ingredients in item supplied by vendor
(SELECT COUNT(DISTINCT m.ingredientid)
FROM madewith m, ingredients n
WHERE i.itemid = m.itemid AND m.ingredientid = n.ingredientid
AND
n.vendorid = v.vendorid);
```

Find all items and vendors such that all ingredients supplied by that vendor are in the item.

```
SELECT name, companyname FROM items i, vendors v
WHERE -- number of ingredients in item supplied by vendor
(SELECT COUNT(DISTINCT m.ingredientid)  FROM madewith m,
ingredients ing
WHERE i.itemid = m.itemid AND m.ingredientid = ing.ingredientid
AND ing.vendorid = v.vendorid)
=
(SELECT COUNT(DISTINCT ing.ingredientid) -- number of
ingredients supplied by vendor
FROM ingredients ing
WHERE ing.vendorid = v.vendorid);
```

2.6 Correlated Subqueries

Correlated subqueries are not independent of the outer query. Correlated subqueries work by first executing the outer query and then executing the inner query for each row from the outer query.

Find the name of students who took less than 10 courses.

```
select student.name from university.student
where
(select count(*) from takes
where student.ID = takes.ID)<10;
```

Look closely at the inner query. It cannot be executed independently from the outer query because the WHERE clause references the student table from the outer query. Note that in the inner query we must use the table name from the outer query to qualify ID. How does this execute? Because it's a correlated subquery, the outer query fetches all the rows from the student table. For each row from the outer query, the inner query is executed to determine the number of courses for the particular ID.

EXISTS: EXISTS is a conditional that determines if any rows exist in the result of a subquery. EXISTS returns true if <subquery> returns at least one row and false otherwise.

Find the customer who do not done any payment at DVD store.

```
select * from sakila.customer
where not exists
(select * from sakila.payment
where payment.customer_id = customer.customer_id);
```

2.7 Subqueries in the SELECT Clause

We can include subqueries in the SELECT clause to compute a column in the result table. It works much like any other expression. You may use both simple and correlated subqueries in the SELECT clause.

Example: Show the list of instructors in the university along with their ID, Dept Name, Salary, University's average salary, and difference from university's average salary from the instructor table.

```
select instructor.ID, instructor.name, instructor.salary,
(select avg(instructor.salary) from university.instructor) as
'AvgSalary',
(select avg(instructor.salary) from university.instructor)-
instructor.salary as 'Diff Salary'
from university.instructor;
```

- Correlated subqueries in the SELECT clause work just like they do in the WHERE clause.

2.8 Derived Relations — Subqueries in the FROM Clause

There are two advantages of derived relations. First, it allows us to break down complex queries into easier-to-understand parts. The second advantage of derived relations is that we can improve the performance of some queries. If a derived relation is much smaller than the original relation, then the query may execute much faster.

Example: Show the list of instructors in the university along with their ID, Dept Name, Salary, Department's average salary from the instructor table.

```
select instructor.ID, instructor.name, instructor.dept_name,
instructor.salary, DeptAvgSal.AvgSal
from university.instructor,
(select instructor.dept_name, avg(instructor.salary) as 'AvgSal'
from university.instructor
group by instructor.dept_name) DeptAvgSal
where DeptAvgSal.dept_name = instructor.dept_name;
```

In the above query, the select query generates a table with two fields, department name, and department average salary. This table to use to show the department's average salary corresponds to the department of instructor.

2.9 Subqueries in the HAVING Clause

We can embed both simple and correlated subqueries in the HAVING clause. This works much like the WHERE clause subqueries, except we are defining predicates on groups rather than rows.

Example: Select the id, name, department name, salary, and course id who gets a salary more than the average salary of a university.

```
select instructor.id, instructor.name, instructor.dept_name,
instructor.salary, teaches.course_id
from university.instructor, university.teaches
where instructor.id = teaches.id
having instructor.salary >
(select avg(instructor.salary) from university.instructor);
```

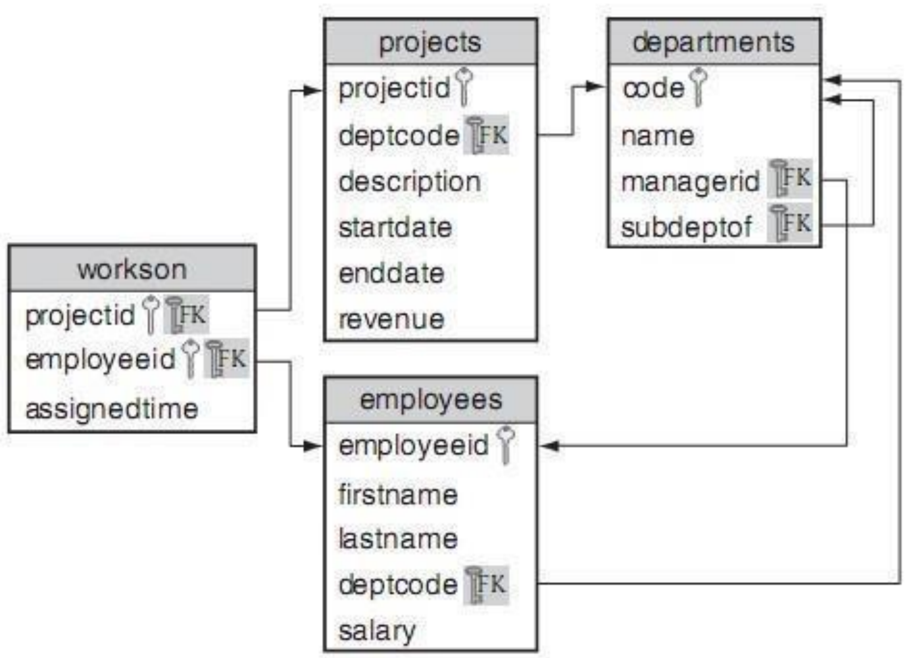
2.10 Section Exercise

Q) Use the sakila schema for

a) Write a query to find the average payment for each customer.
b) Write a query to find the name of each film, the category it belongs to, and the no. of films that fall in that particular category.
c) Write a query to get the most frequently rented movies in descending order.
d) Write a query to find the full name (concatenation of first and last name) and email of all the customers that have rented a movie from the "action" category.

3 Exercises

Write a single SQL query for each of the following based on the employee database created in the last lab. The scheme for the employee database is shown below. Not all these queries can be solved using subqueries. Some of them require correlated subqueries and division operations.



1. Find the names of all people who work in the Consulting department.
2. Find the names of all people who work in the Consulting department and who spend more than 20% of their time on the project with ID ADT4MFIA.
3. Find the total percentage of time assigned to employee Abe Advice.
4. Find the names of all departments not currently assigned a project.
5. Find the first and last names of all employees who make more than the average salary of the people in the Accounting department.

6. Find the descriptions of all projects that require more than 70% of an employee's time.
7. Find the first and last names of all employees who are paid more than someone in the Accounting department.
8. Find the minimum salary of the employees who are paid more than everyone in the Accounting department.
9. Find the first and last name of the highest-paid employee(s) in the Accounting department.
10. For each employee in the department with code ACCNT, find the employee ID and number of assigned hours that the employee is currently working on projects for other departments. Only report an employee if she has some current project to which she is assigned more than 50% of the time and the project is for another department. Report the results in ascending order by hours.
11. Find all departments where all their employees are assigned to all of their projects.
12. Use correlated subqueries in the SELECT and WHERE clauses, derived tables, and subqueries in the HAVING clause to answer these queries. If they cannot be answered using that technique, explain why.
    a. Find the names of all people who work in the Information Technology department.
    b. Find the names of all people who work in the Information Technology department and who spend more than 20% of their time on the health project.
    c. Find the names of all people who make more than the average salary of the people in the Accounting department.
    d. Find the names of all projects that require more than 50% of an employee's time. (e) Find the total percentage time assigned to employee Bob Smith.
    e. Find all departments that did not assign a project.
    f. Find all employees who are paid more than someone in the Information Technology department.
    g. Find all employees who are paid more than everyone in the Information Technology department.
    h. Find the highest-paid employee in the Information Technology department.

*********************************END**************************************

Birla Institute of Technology and Science, Pilani

CS F212 Database Systems

Lab No # 6

****

1 Transaction

Databases are all about sharing data, so it is common for multiple users to be accessing and even changing the same data at the same time. The simultaneous execution of operations is called concurrency. Sometimes concurrency can get us into trouble if our changes require multiple SQL statements. In general, if two or more users access the same data and one or more of the statements changes the data, we have a conflict. This is a classic problem in database systems; it is called the isolation or serializability problem. If the users perform multiple steps, conflicts can cause incorrect results to occur. To deal with this problem, databases allow the grouping of a sequence of SQL statements into an indivisible unit of work called a transaction. A transaction ends with either a commit or a rollback:

Commit—A commit permanently stores all the changes performed by the transaction.

Rollback—A rollback removes all the updates performed by the transaction, no matter how many rows have been changed. A rollback can be executed either by the DBMS to prevent incorrect actions or explicitly by the user.

The DBMS provides the following guarantees for a transaction, called the ACID properties: Atomicity, consistency, Isolation, and durability. These properties will be covered in the course in detail.

SQL starts a transaction automatically when a new statement is executed if there is no currently active transaction. This means that a new transaction begins automatically with the first statement after the end of the previous transaction or the beginning of the session.

MySQL Transaction:

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]

SET autocommit = {0 | 1}
```

Start Transaction

- MySQL provides a START TRANSACTION statement to begin the transaction. It also offers a "BEGIN" and "BEGIN WORK" as an alias of the START TRANSACTION.

Commit

- We will use a COMMIT statement to commit the current transaction. It allows the database to make changes permanently.

Rollback

- We will use a ROLLBACK statement to roll back the current transaction. It allows the database to cancel all changes and goes into their previous state.

Try following set of SQL query and observe the results.

```
START TRANSACTION;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *2
where items.itemid = 'CHKSD';
SELECT * FROM `restaurant`.`items`;
ROLLBACK;
SELECT * FROM `restaurant`.`items`;


START TRANSACTION;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *2
where items.itemid = 'CHKSD';
SELECT * FROM `restaurant`.`items`;
commit;
rollback;
SELECT * FROM `restaurant`.`items`;


START TRANSACTION;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *0.5
```

```
where items.itemid = 'CHKSD';
SELECT * FROM `restaurant`.`items`;
ROLLBACK;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *2
where items.itemid = 'CHKSD';
SELECT * FROM `restaurant`.`items`;
ROLLBACK;
SELECT * FROM `restaurant`.`items`;
```

Any transaction started by "START TRANSACTION" is ended with the "COMMIT" or a "ROLLBACK" statement.

Auto Commit: We will use a SET auto-commit statement to disable/enable the auto-commit mode for the current transaction. By default, the COMMIT statement executed automatically. So, if we do not want to commit changes automatically, use the below statement:

```
SET autocommit = 0;
OR,
SET autocommit = OFF:
```

Again, use the below statement to enable auto-commit mode:

```
SET autocommit = 1;
OR,
SET autocommit = ON:
```

2 Save Point

InnoDB supports the SQL statements SAVEPOINT, ROLLBACK TO SAVEPOINT, RELEASE SAVEPOINT and the optional WORK keyword for ROLLBACK.

```
SAVEPOINT identifier
ROLLBACK [WORK] TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

The SAVEPOINT statement sets a named transaction savepoint with a name of identifier. If the current transaction has a savepoint with the same name, the old savepoint is deleted and a new one is set.

2.1 Rollback to Save Point

The ROLLBACK TO SAVEPOINT statement rolls back a transaction to the named savepoint without terminating the transaction. Modifications that the current transaction made to rows after the savepoint was set are undone in the rollback, but InnoDB does not release the row locks that were stored in memory after the savepoint. (For a new inserted row, the lock information is carried by the transaction ID stored in the row; the lock is not separately stored in memory. In this case, the row lock is released in the undo.) Savepoint that were set at a later time than the named savepoint are deleted.

If the ROLLBACK TO SAVEPOINT statement returns the following error, it means that no savepoint with the specified name exists:

```
ERROR 1305 (42000): SAVEPOINT identifier does not exist
```

2.2 Release Save Point

- The RELEASE SAVEPOINT statement removes the named savepoint from the set of savepoints of the current transaction. No commit or rollback occurs. It is an error if the savepoint does not exist.
- All savepoints of the current transaction are deleted if you execute a COMMIT, or a ROLLBACK that does not name a savepoint.
- A new savepoint level is created when a stored function is invoked, or a trigger is activated. The savepoints on previous levels become unavailable and thus do not conflict with savepoints on the new level. When the function or trigger terminates, any savepoints it created are released and the previous savepoint level is restored.

Try following set of SQL queries and observe the results.

```
START TRANSACTION;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *0.5
where items.itemid = 'CHKSD';
savepoint point1;
SELECT * FROM `restaurant`.`items`;
update items
set price = price *2
where items.itemid = 'CHKSD';
SELECT * FROM `restaurant`.`items`;
savepoint point2;
SELECT * FROM `restaurant`.`items`;
rollback to savepoint point1;
SELECT * FROM `restaurant`.`items`;
commit;
SELECT * FROM `restaurant`.`items`;
rollback to savepoint point1;
SELECT * FROM `restaurant`.`items`;
```

- When you execute second last query; it gives error, because after commit all savepoints are released.

3 Statements That Cause an Implicit Commit

Some statements cannot be rolled back. In general, these include data definition language (DDL) statements, such as those that create or drop databases, those that create, drop, or alter tables or stored routines.

You should design your transactions not to include such statements. If you issue a statement early in a transaction that cannot be rolled back, and then another statement later fails, the full effect of the transaction cannot be rolled back in such cases by issuing a ROLLBACK statement.

3.1 Data definition language (DDL) statements that define or modify database objects

Statements use keywords like ALTER, CREATE, DROP, INSTALL, RENAME, TRUNCATE, and UNINSTALL use implicit commit.

CREATE TABLE and DROP TABLE statements do not commit a transaction if the TEMPORARY keyword is used. (This does not apply to other operations on temporary tables such as ALTER TABLE and CREATE INDEX, which do cause a commit.) However, although no implicit commit occurs, neither can the statement be rolled back, which means that the use of such statements causes transactional atomicity to be violated

3.2 Statements that implicitly use or modify tables in the MySQL database

Statements use keywords like ALTER USER, CREATE USER, DROP USER, GRANT, RENAME USER, REVOKE, SET PASSWORD use implicit commit.

3.3 Transaction-control and locking statements

- BEGIN, LOCK TABLES, SET autocommit = 1 (if the value is not already 1), START TRANSACTION, UNLOCK TABLES.
- UNLOCK TABLES commits a transaction only if any tables currently have been locked with LOCK TABLES to acquire non-transactional table locks. A commit does not occur for UNLOCK TABLES following FLUSH TABLES WITH READ LOCK because the latter statement does not acquire table-level locks.
- Transactions cannot be nested. This is a consequence of the implicit commit performed for any current transaction when you issue a START TRANSACTION statement or one of its synonyms.

3.4 Data loading statements

- LOAD DATA. LOAD DATA causes an implicit commit only for tables using the NDB storage engine.

3.5 Administrative statements

- ANALYZE TABLE, CACHE INDEX, CHECK TABLE, FLUSH, LOAD INDEX INTO CACHE, OPTIMIZE TABLE, REPAIR TABLE, RESET (but not RESET PERSIST).

3.6 Replication control statements

- START REPLICA | SLAVE, STOP REPLICA | SLAVE, RESET REPLICA | SLAVE, CHANGE REPLICATION SOURCE TO, CHANGE MASTER TO.

4 Exercise

Q1) Using the sakila database, write a transaction to update the name of an actor HARPO WILLIAMS to GROUCHO WILLIAMS, see what happens if you try to roll back the transaction. Is there a way to undo the changes of a rollback call (like the redo function in text editors)?

Q2) Is the nesting of the transaction(s) possible in MySQL? If yes, what will happen if you use ROLLBACK, i.e. which transaction will be undone first? Try it out.

Q3) What are the 4 characteristics of transactions in SQL. Describe them.

Q4) What will happen if you try to roll back to a savepoint which is defined in the latter part of your query code.

*************************************END**

Birla Institute of Technology and Science, Pilani
CS F212 Database Systems
Lab No # 7

****

1 Procedures

The following SELECT statement returns all rows in the table customer from the Sakila database.

```
select customer.customer_id, customer.first_name,
customer.last_name
from sakila.customer;
```

If you want to save this query on the database server for execution later, one way to do it is to use a stored procedure.

The following *CREATE PROCEDURE* statement creates a new stored procedure that wraps the query above:

```
DELIMITER $$
CREATE PROCEDURE GetCustomers()
BEGIN
select customer.customer_id, customer.first_name,
customer.last_name
from sakila.customer;
END$$
DELIMITER ;
```

By definition, a stored procedure is a segment of declarative SQL statements stored inside the MySQL Server. In this example, we have just created a stored procedure with the name GetCustomers().

Once you save the stored procedure, you can invoke it by using the CALL statement:

```
CALL GetCustomers();
```

And the statement returns the same result as the query.

☞ The first time you invoke a stored procedure, MySQL looks up for the name in the database catalog, compiles the stored procedure's code, place it in a memory area known as a cache, and execute the stored procedure.

- If you invoke the same stored procedure in the same session again, MySQL just executes the stored procedure from the cache without having to recompile it.
- A stored procedure can have parameters so you can pass values to it and get the result back. For example, you can have a stored procedure that returns customers by country and city. In this case, the country and city are parameters of the stored procedure.
- A stored procedure may contain control flow statements such as IF, CASE, and LOOP that allow you to implement the code in the procedural way.
- A stored procedure can call other stored procedures or stored functions, which allows you to modularize your code.

MySQL stored procedures advantages

- You can avoid having to store all your SQL code in files. Stored procedures are stored in the database itself, so you never have to search through files to find the code you want to use.
- You can execute a stored procedure as often as you like from any machine that can connect to the database server.
- If you have a report that needs to be run frequently, you can create a stored procedure that produces the report. Anyone who has access to the database and permission to execute the stored procedure will be able to produce the report at will. They don't have to understand the SQL statements in the stored procedure. All they have to know is how to execute the stored procedure.
- Reduce network traffic: Stored procedures help reduce the network traffic between applications and MySQL Server. Because instead of sending multiple lengthy SQL statements, applications have to send only the name and parameters of stored procedures.
- Centralize business logic in the database: You can use the stored procedures to implement business logic that is reusable by multiple applications. The stored procedures help reduce the efforts of duplicating the same logic in many applications and make your database more consistent.
- Make database more secure: The database administrator can grant appropriate privileges to applications that only access specific stored procedures without giving any privileges on the underlying tables.

MySQL stored procedures disadvantages

- Resource usages: If you use many stored procedures, the memory usage of every connection will increase substantially. Besides, overusing a large number of logical operations in the stored procedures will increase the CPU usage because the MySQL is not well-designed for logical operations.
- Troubleshooting: It’s difficult to debug stored procedures. Unfortunately, MySQL does not provide any facilities to debug stored procedures like other enterprise database products such as Oracle and SQL Server.

- Maintenances: Developing and maintaining stored procedures often requires a specialized skill set that not all application developers possess. This may lead to problems in both application development and maintenance.

Create procedure syntax:

```
CREATE
    [DEFINER = user]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:

    Valid SQL routine statement
```

- *CREATE* statements are used to create a stored procedure. That is, the specified routine becomes known to the server. By default, a stored routine is associated with the default database. To associate the routine explicitly with a given database, specify the name as *db_name.sp_name* when you create it.
- Each parameter is an *IN* parameter by default. To specify otherwise for a parameter, use the keyword *OUT* or *INOUT* before the parameter name.
- The *routine_body* consists of a valid SQL routine statement. This can be a simple statement such as *SELECT* or *INSERT*, or a compound statement written using *BEGIN* and *END*.
- The *COMMENT* characteristic is a MySQL extension, and may be used to describe the stored routine.
- The *LANGUAGE* characteristic indicates the language in which the routine is written. The server ignores this characteristic; only SQL routines are supported.
- A routine is considered “deterministic” if it always produces the same result for the same input parameters, and “not deterministic” otherwise. If neither *DETERMINISTIC* nor *NOT DETERMINISTIC* is given in the routine definition, the default is *NOT DETERMINISTIC*.
- *CONTAINS SQL* indicates that the routine does not contain statements that read or write data. This is the default if none of these characteristics is given explicitly.
- *NO SQL* indicates that the routine contains no SQL statements.
- *READS SQL DATA* indicates that the routine contains statements that read data (for example, SELECT), but not statements that write data.

- ☞ *MODIFIES SQL DATA* indicates that the routine contains statements that may write data (for example, INSERT or DELETE).
- ☞ *The SQL SECURITY* characteristic can be DEFINER or INVOKER to specify the security context; that is, whether the routine executes using the privileges of the account named in the routine DEFINER clause or the user who invokes it.
- ☞ The *DEFINER* clause specifies the MySQL account to be used when checking access privileges at routine execution time for routines that have the *SQL SECURITY DEFINER* characteristic.

Example: - Now redefine the GetCustomer() procedure.

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `GetCustomers`()
    READS SQL DATA
    DETERMINISTIC
    SQL SECURITY INVOKER
    COMMENT 'Stored procedure example'
BEGIN
     select customer.customer_id, customer.first_name,
     customer.last_name
     from sakila.customer;
END$$
DELIMITER ;
```

Example 1: - Write a MySQL procedure to show student list for semester. Name of semester is given as input.

```
     DELIMITER $$
     CREATE DEFINER=`root`@`localhost` PROCEDURE
     `list_semester`(IN sem_name VARCHAR(10))
         READS SQL DATA
         DETERMINISTIC
         SQL SECURITY INVOKER
         COMMENT 'Smester wise list of student, Input -
     Semester name and Output - Student count'
     BEGIN
     select student.name from university2.student
     where student.ID IN
     (select takes.id from university2.takes
     where takes.semester = sem_name);
     END$$
```

```
DELIMITER ;
```

Now you can call the above procedure as follows.

```
call list_semester('Spring');
```

Example 2: - Write a MySQL procedure to show student list for semester. Name of semester is given as input and count the number of students in that semester.

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE
`list_semester_and_count`(IN sem_name VARCHAR(10), out
student_count int)
    READS SQL DATA
    DETERMINISTIC
    SQL SECURITY INVOKER
    COMMENT 'Smester wise list of student, Input -
Semester name.'
BEGIN
select student.name from university2.student
where student.ID IN
(select takes.id from university2.takes
where takes.semester = sem_name);
    select count(student.name) into student_count from
university2.student
where student.ID IN
(select takes.id from university2.takes
where takes.semester = sem_name);
END$$
DELIMITER ;
```

Now you can call the above procedure as follows.

```
call list_semester_and_count('Fall', @st_count);
select @st_count;
```

The following example demonstrates how to use an INOUT parameter in the stored procedure.

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `SetCounter`(
INOUT counter INT,     IN inc INT )
    DETERMINISTIC
    SQL SECURITY INVOKER
    COMMENT 'INOUT example.'
BEGIN
SET counter = counter + inc;
END$$
DELIMITER ;
```

In this example, the stored procedure SetCounter() accepts one INOUT parameter (counter) and one IN parameter (inc). It increases the counter (counter) by the value of specified by the inc parameter.

These statements illustrate how to call the SetSounter stored procedure:

```
SET @counter = 1;
CALL SetCounter(@counter,1); -- 2
CALL SetCounter(@counter,1); -- 3
CALL SetCounter(@counter,5); -- 8
SELECT @counter; -- 8
```

- ☞ An IN parameter passes a value into a procedure. The procedure might modify the value, but the modification is not visible to the caller when the procedure returns.
- ☞ An OUT parameter passes a value from the procedure back to the caller. Its initial value is NULL within the procedure, and its value is visible to the caller when the procedure returns.
- ☞ An INOUT parameter is initialized by the caller, can be modified by the procedure, and any change made by the procedure is visible to the caller when the procedure returns.
- ☞ For each OUT or INOUT parameter, pass a user-defined variable in the CALL statement that invokes the procedure so that you can obtain its value when the procedure returns.

Note: - MySQL permits routines to contain DDL statements, such as CREATE and DROP. MySQL also permits stored procedures (but not stored functions) to contain SQL transaction statements such as COMMIT. Stored functions may not contain statements that perform explicit or implicit commit or rollback.

2 Functions

A stored function in MySQL is a set of SQL statements that perform some task/operation and return a single value. It is one of the types of stored programs in MySQL. When you will create a stored function, make sure that you have a CREATE ROUTINE database privilege. Generally, we used this function to encapsulate the common business rules or formulas reusable in stored programs or SQL statements.

The stored function is almost similar to the procedure in MySQL, but it has some differences that are as follows:

- The function parameter may contain only the IN parameter but can't allow specifying this parameter, while the procedure can allow IN, OUT, INOUT parameters.
- The stored function can return only a single value defined in the function header.
- The stored function may also be called within SQL statements.
- It may not produce a result set.

Thus, we will consider the stored function when our program's purpose is to compute and return a single value only or create a user-defined function.

The syntax of creating a stored function in MySQL is as follows:

```
CREATE
    [DEFINER = user]
    FUNCTION sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

func_parameter:
    param_name type

type:
    Any valid MySQL data type

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:

    Valid SQL routine statement
```

Example: - Consider Sakila database, find out that whether a DVD rental is overdue? If yes, then count overdue days.

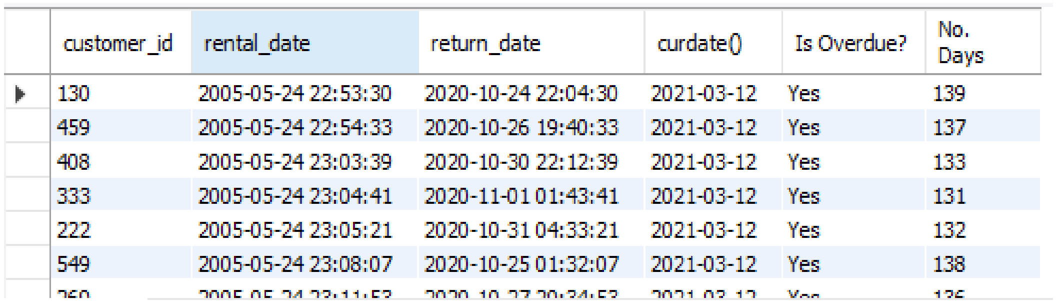
We create to stored function, 1) to check overdue, 2) to count days.

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` FUNCTION
`overdue`(return_date DATE) RETURNS varchar(3) CHARSET
utf8mb4
    DETERMINISTIC
BEGIN
     DECLARE sf_value VARCHAR(3);
```

```
        IF curdate() > return_date
            THEN SET sf_value = 'Yes';
        ELSEIF  curdate() <= return_date
            THEN SET sf_value = 'No';
        END IF;
     RETURN sf_value;
    END$$
DELIMITER ;

DELIMITER $$
CREATE DEFINER=`root`@`localhost` FUNCTION
`count_days`(return_date DATE) RETURNS int
    DETERMINISTIC
begin
     declare days INT;
        IF curdate() > return_date
            THEN SET days = DATEDIFF(curdate(),
return_date);
        ELSEIF  curdate() <= return_date
            THEN SET days = 0;
        END IF;
        return days;
end$$
DELIMITER ;
```

To call these functions as follows.

```
select rental.customer_id, rental.rental_date,
rental.return_date, curdate(),
overdue (rental.return_date) as 'Is Overdue?',
count_days(rental.return_date) as 'No. Days'
from sakila.rental;
```



customer_id	rental_date	return_date	curdate()	Is Overdue?	No. Days
130	2005-05-24 22:53:30	2020-10-24 22:04:30	2021-03-12	Yes	139
459	2005-05-24 22:54:33	2020-10-26 19:40:33	2021-03-12	Yes	137
408	2005-05-24 23:03:39	2020-10-30 22:12:39	2021-03-12	Yes	133
333	2005-05-24 23:04:41	2020-11-01 01:43:41	2021-03-12	Yes	131
222	2005-05-24 23:05:21	2020-10-31 04:33:21	2021-03-12	Yes	132
549	2005-05-24 23:08:07	2020-10-25 01:32:07	2021-03-12	Yes	138

3 Cursors

MySQL supports cursors inside stored programs. The syntax is as in embedded SQL. Cursors have these properties:

1. Asensitive: The server may or may not make a copy of its result table
2. Read only: Not updatable
3. Nonscrollable: Can be traversed only in one direction and cannot skip rows

In MySQL, Cursor can also be created. Following are the steps for creating a cursor.

1. Declare Cursor

   ```
   DECLARE cursor_name CURSOR FOR
   Select statement;
   ```

2. Open Cursor

   ```
   Open cursor_name;
   ```

3. Fetch Cursor

   ```
   FETCH cursor_name INTO variable_list;
   ```

4. Close Cursor

   ```
   Close cursor_name;
   ```

Note: - Cursor declarations must appear before handler declarations and after variable and condition declarations.

Example: - Suppose there is two tables (t1, and t2) containing two fields item and price from two different vendors. We wish to find out the lowest price for each item and store it in another table. We can do this by using stored procedure and cursor.

```
CREATE PROCEDURE low_price()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE a CHAR(16);
  DECLARE b, c INT;
  DECLARE cur1 CURSOR FOR SELECT item, price FROM t1;
  DECLARE cur2 CURSOR FOR SELECT price FROM t2;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
  OPEN cur1;
  OPEN cur2;
```

```
  read_loop: LOOP
    FETCH cur1 INTO a, b;
    FETCH cur2 INTO c;
    IF done THEN
      LEAVE read_loop;
    END IF;
    IF b < c THEN
      INSERT INTO t3 VALUES (a,b);
    ELSE
      INSERT INTO t3 VALUES (a,c);
    END IF;
  END LOOP;
  CLOSE cur1;
  CLOSE cur2;
END;
```

4 Nested call of stored routines

Function *inventory_in_stock* return that whether a film is in stock or not. And procedure *film_in_stock* return the count of stock.

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` FUNCTION
`inventory_in_stock`(p_inventory_id INT) RETURNS
tinyint(1)
    READS SQL DATA
BEGIN
    DECLARE v_rentals INT;
    DECLARE v_out     INT;

    #AN ITEM IS IN-STOCK IF THERE ARE EITHER NO ROWS IN
THE rental TABLE
    #FOR THE ITEM OR ALL ROWS HAVE return_date POPULATED

    SELECT COUNT(*) INTO v_rentals
    FROM rental
    WHERE inventory_id = p_inventory_id;

    IF v_rentals = 0 THEN
      RETURN TRUE;
    END IF;
    SELECT COUNT(rental_id) INTO v_out
    FROM inventory LEFT JOIN rental USING(inventory_id)
    WHERE inventory.inventory_id = p_inventory_id
    AND rental.return_date IS NULL;

    IF v_out > 0 THEN
      RETURN FALSE;
    ELSE
      RETURN TRUE;
    END IF;
END$$
DELIMITER ;

DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE
`film_in_stock`(IN p_film_id INT, IN p_store_id INT, OUT
p_film_count INT)
    READS SQL DATA
BEGIN
     SELECT inventory_id
     FROM inventory
     WHERE film_id = p_film_id
     AND store_id = p_store_id
     AND inventory_in_stock(inventory_id);

     SELECT FOUND_ROWS() INTO p_film_count;
END$$
DELIMITER ;
```

To know more on –

- Restrictions on Stored Programs
  https://dev.mysql.com/doc/refman/8.0/en/stored-program-restrictions.html
- Stored Routines and MySQL Privileges
  https://dev.mysql.com/doc/refman/8.0/en/stored-routines-privileges.html
- DECLARE ... HANDLER Statement
  https://dev.mysql.com/doc/refman/8.0/en/declare-handler.html
- Server Error Message Reference
  https://dev.mysql.com/doc/mysql-errors/8.0/en/server-error-reference.html

5 Exercise

Write stored routines for following task

1. List name of student from student table who takes course in given semester.
2. List all customer who done a payment with a given staff id.
3. List instructor's name and ID whose salary is less or more than a given amount to the average salary of all instructors in the university.
4. Find the name of all ingredients supplied by a specific vendor.

***********************************END***************************************

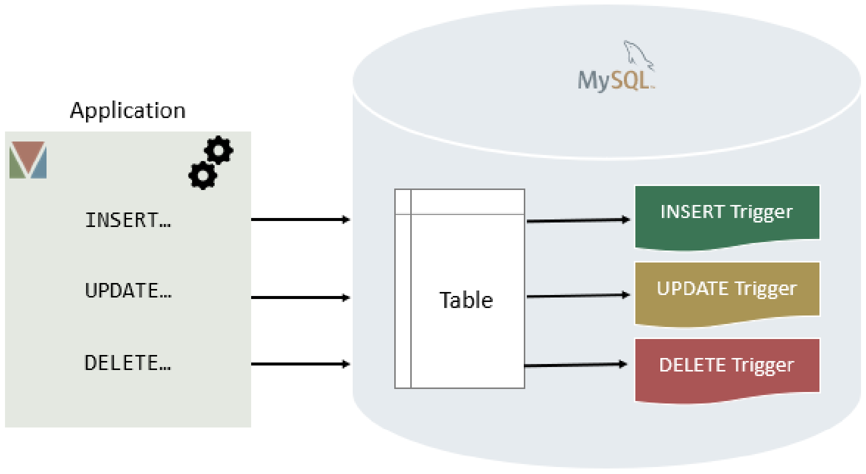
Birla Institute of Technology and Science, Pilani
CS F212 Database Systems
Lab No # 8

****

All examples in this lab sheet are not based on the database used in previous labs to preserve modification in the schema of existing sample databases. You can create a new temporary database and test the examples given in this lab sheet.

1 Triggers

A trigger in MySQL is a set of SQL statements that reside in a system catalog. It is a special type of stored procedure that is invoked automatically in response to an event. Each trigger is associated with a table, which is activated on any DML statement such as INSERT, UPDATE, or DELETE.



1.1 Features of Triggers

- Triggers help us to enforce business rules.
- Triggers help us to validate data even before they are inserted or updated.
- Triggers help us to keep a log of records like maintaining audit trails in tables.
- SQL triggers provide an alternative way to check the integrity of data.
- Triggers provide an alternative way to run the scheduled task.
- Triggers increases the performance of SQL queries because it does not need to compile each time the query is executed.
- Triggers reduce the client-side code that saves time and effort.
- Triggers help us to scale our application across different platforms.
- Triggers are easy to maintain.

1.2 Benefits of Triggers

- Audit data modifications
- Log events transparently
- Enforce complex business rules
- Derive column values automatically
- Implement complex security authorizations
- Maintain replicated tables

1.3 Limitations of Using Triggers in MySQL

- MySQL triggers do not allow to use of all validations; they only provide extended validations. For example, we can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints for simple validations.
- Triggers are invoked and executed invisibly from the client application. Therefore, it isn't easy to troubleshoot what happens in the database layer.
- Triggers may increase the overhead of the database server.

1.4 Types of Triggers in MySQL

We can define the maximum six types of actions or events in the form of triggers:

- Before Insert: It is activated before the insertion of data into the table.
- After Insert: It is activated after the insertion of data into the table.
- Before Update: It is activated before the update of data in the table.
- After Update: It is activated after the update of the data in the table.
- Before Delete: It is activated before the data is removed from the table.
- After Delete: It is activated after the deletion of data from the table.

1.5 Difference between Triggers and Subprograms

Triggers	**Stored Procedures**
A Trigger is implicitly invoked whenever any event such as INSERT, DELETE, UPDATE occurs in a TABLE.	A Procedure is explicitly called by user/application using statements or commands such as exec or CALL procedure_name
Only nesting of triggers can be achieved in a table. We cannot define/call a trigger inside another trigger.	We can define/call procedures inside another procedure.
Transaction statements such as COMMIT, ROLLBACK, SAVEPOINT are not allowed in triggers.	All transaction statements such as COMMIT, ROLLBACK is allowed in procedures.
Triggers are used to maintain referential integrity by keeping a record of activities performed on the table.	Procedures are used to perform tasks defined or specified by the users.

We cannot return values in a trigger. Also, as an input, we cannot pass values as a parameter.	We can return 0 to n values. However, we can pass values as parameters.

1.6 Using Triggers

1.6.1 Creating Database Trigger

The CREATE TRIGGER statement creates a new trigger. Here is the basic syntax of the CREATE TRIGGER statement:

```
CREATE
    [DEFINER = user]
    TRIGGER trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    [trigger_order]
    trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE| DELETE }
ON table_name FOR EACH ROW
trigger_body;
```

In this syntax:

- This statement creates a new trigger. A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table.
- The DEFINER clause determines the security context to be used when checking access privileges at trigger activation time.
- trigger_time is the trigger action time. It can be BEFORE or AFTER to indicate that the trigger activates before or after each row to be modified.
- trigger_event indicates the kind of operation that activates the trigger. These trigger_event values are permitted:
    - INSERT: The trigger activates whenever a new row is inserted into the table (for example, through INSERT, LOAD DATA, and REPLACE statements).
    - UPDATE: The trigger activates whenever a row is modified (for example, through UPDATE statements).
    - DELETE: The trigger activates whenever a row is deleted from the table (for example, through DELETE and REPLACE statements). DROP TABLE and

TRUNCATE TABLE statements on the table do not activate this trigger, because they do not use DELETE. Dropping a partition does not activate DELETE triggers, either.

- ☞ To distinguish between the value of the columns BEFORE and AFTER the DML has fired, you use the *NEW* and *OLD* modifiers.

Points to remember:

- ☞ The trigger becomes associated with the table named tbl_name, which must refer to a permanent table.
    - o You cannot associate a trigger with a TEMPORARY table or a view.
- ☞ Trigger names exist in the schema namespace, meaning that all triggers must have unique names within a schema.
- ☞ CREATE TRIGGER requires the TRIGGER privilege for the table associated with the trigger. If the DEFINER clause is present, the privileges required depend on the user value.
- ☞ The trigger_event does not represent a literal type of SQL statement that activates the trigger so much as it represents a type of table operation. For example, an INSERT trigger activates not only for INSERT statements but also LOAD DATA statements because both statements insert rows into a table.
- ☞ It is possible to define multiple triggers for a given table that have the same trigger event and action time.
    - o By default, triggers that have the same trigger event and action time activate in the order they were created.
- ☞ To affect trigger order, specify a trigger_order clause that indicates FOLLOWS or PRECEDES and the name of an existing trigger that also has the same trigger event and action time.
    - o With FOLLOWS, the new trigger activates after the existing trigger. With PRECEDES, the new trigger activates before the existing trigger.
- ☞ Triggers cannot use NEW.col_name or use OLD.col_name to refer to generated columns. For information about generated columns

Examples:

When inserting a row in orders table, we need add 18% GST before inserting the new row. The following trigger add 18% GST into the new price of order.

```
DELIMITER //
CREATE TRIGGER add_gst BEFORE INSERT ON orders
FOR EACH ROW
     BEGIN
          SET NEW.price = NEW.price*1.18;
     END //
DELIMITER ;
```

Example 2:

```
CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 INT NOT NULL AUTO_INCREMENT PRIMARY
KEY);
CREATE TABLE test4(
  a4 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  b4 INT DEFAULT 0
);
```

The following trigger is associated with table test1 and executed before insert statement. It insert same value in table test2, delete that value from test3, and update the value in table test4.

```
delimiter |
CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;
|
delimiter ;
```

Now insert into table test3 and test4.

```
INSERT INTO test3 (a3) VALUES
  (NULL), (NULL), (NULL), (NULL), (NULL),
  (NULL), (NULL), (NULL), (NULL), (NULL);

INSERT INTO test4 (a4) VALUES
  (0), (0), (0), (0), (0), (0), (0), (0), (0), (0);
```

Suppose that you insert the following values into table test1 as shown here:

```
INSERT INTO test1 VALUES
       (1), (3), (1), (7), (1), (8), (4), (4);
```

Now, after insert statement on table test1, retrieve data from all four tables and observe the result to identify the effect of above trigger.

Note: - The following table illustrates the availability of the OLD and NEW modifiers:

Trigger Event	OLD	NEW
INSERT	No	Yes
UPDATE	Yes	Yes
DELETE	Yes	No

1.6.2 Execution of Triggers

- It is executed implicitly when DML, DDL, or system event occurs.
- The act of executing a trigger is also known as firing the trigger.
- When multiple triggers are to be executed, they have to be executed in a sequence.
- Database triggers execute with the privileges of the owner, not the current user.

1.6.3 Managing Triggers

Some DBS vendors provide modification in existing triggers such as ENABLE or DISABLE a trigger and ALTER trigger. MySQL does not provide ALTER TRIGGER and ENABLE or DISABLE facility.

1.6.4 Removing Triggers

To destroy the trigger, use a DROP TRIGGER statement. You must specify the schema name if the trigger is not in the default schema:

```
DROP TRIGGER testref;
```

1.6.5 Create Multiple Triggers

Here is the syntax for defining a trigger that will activate before or after an existing trigger in response to the same event and action time:

```
DELIMITER $$

CREATE TRIGGER trigger_name
{BEFORE|AFTER}{INSERT|UPDATE|DELETE}
ON table_name FOR EACH ROW
{FOLLOWS|PRECEDES} existing_trigger_name
BEGIN
    -- statements
END$$

DELIMITER ;
```

- The FOLLOWS allow the new trigger to activate after an existing trigger.
- The PRECEDES allow the new trigger to activate before an existing trigger.

2 Indexing

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows.

Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees. Exceptions: Indexes on spatial data types use R-trees; MEMORY tables also support hash indexes; InnoDB uses inverted lists for FULLTEXT indexes.

MySQL uses indexes for these operations:

- To find the rows matching a WHERE clause quickly.
- To eliminate rows from consideration.
    - If there is a choice between multiple indexes, MySQL normally uses the index that finds the smallest number of rows.
- If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to look up rows.
- To retrieve rows from other tables when performing joins. MySQL can use indexes on columns more efficiently if they are declared as the same type and size.
- To sort or group a table if the sorting or grouping is done on a leftmost prefix of a usable index.

2.1 CREATE INDEX Statement

- Normally, you create all indexes on a table at the time the table itself is created with CREATE TABLE.
    - especially important for InnoDB tables, where the primary key determines the physical layout of rows in the data file.
- CREATE INDEX enables you to add indexes to existing tables.
- CREATE INDEX is mapped to an ALTER TABLE statement to create indexes.
- CREATE INDEX cannot be used to create a PRIMARY KEY; use ALTER TABLE instead.
- A key_part specification can end with ASC or DESC to specify whether index values are stored in ascending or descending order.
    - ASC and DESC are not permitted for HASH indexes.
    - ASC and DESC are also not supported for multi-valued indexes.
    - As of MySQL 8.0.12, ASC and DESC are not permitted for SPATIAL indexes.

CREATE INDEX Statement Syntax: -

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX index_name
    [index_type]
    ON tbl_name (key_part,...)
    [index_option]
    [algorithm_option | lock_option] ...

key_part: {col_name [(length)] | (expr)} [ASC | DESC]

index_option: {
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
  | {VISIBLE | INVISIBLE}
  | ENGINE_ATTRIBUTE [=] 'string'
  | SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
}

index_type:
    USING {BTREE | HASH}

algorithm_option:
    ALGORITHM [=] {DEFAULT | INPLACE | COPY}

lock_option:

    LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}
```

The following sections describe different aspects of the CREATE INDEX statement:

2.1.1 Column Prefix Key Parts

For string columns, indexes can be created that use only the leading part of column values, using col_name(length) syntax to specify an index prefix length:

- Prefixes can be specified for CHAR, VARCHAR, BINARY, and VARBINARY key parts.
- Prefixes must be specified for BLOB and TEXT key parts. Additionally, BLOB and TEXT columns can be indexed only for InnoDB, MyISAM, and BLACKHOLE tables.
- Prefix limits are measured in bytes. However, prefix lengths for index specifications in CREATE TABLE, ALTER TABLE, and CREATE INDEX statements are interpreted as number of characters for nonbinary string types (CHAR, VARCHAR, TEXT) and number of bytes for binary string types (BINARY, VARBINARY, BLOB).

If a specified index prefix exceeds the maximum column data type size, CREATE INDEX handles the index as follows:

- For a nonunique index, either an error occurs (if strict SQL mode is enabled), or the index length is reduced to lie within the maximum column data type size and a warning is produced (if strict SQL mode is not enabled).

- For a unique index, an error occurs regardless of SQL mode because reducing the index length might enable insertion of nonunique entries that do not meet the specified uniqueness requirement.

The statement shown here creates an index using the first 10 characters of the name column (assuming that name has a nonbinary string type):

```
CREATE INDEX part_of_name ON customer (name(10));
```

2.1.2 Functional Key Parts

A "normal" index indexes column values or prefixes of column values. For example, in the following table, the index entry for a given t1 row includes the full col1 value and a prefix of the col2 value consisting of its first 10 characters:

```
CREATE TABLE t1 (
  col1 VARCHAR(10),
  col2 VARCHAR(20),
  INDEX (col1, col2(10))

);
```

MySQL 8.0.13 and higher supports functional key parts that index expression values rather than column or column prefix values. Use of functional key parts enables indexing of values not stored directly in the table. Examples:

```
CREATE TABLE t1 (col1 INT, col2 INT, INDEX func_index ((ABS(col1))));
CREATE INDEX idx1 ON t1 ((col1 + col2));
CREATE INDEX idx2 ON t1 ((col1 + col2), (col1 - col2), col1);

ALTER TABLE t1 ADD INDEX ((col1 * 40) DESC);
```

- An index with multiple key parts can mix nonfunctional and functional key parts.
- ASC and DESC are supported for functional key parts.

Functional key parts must adhere to the following rules.

- In index definitions, enclose expressions within parentheses to distinguish them from columns or column prefixes.
  ```
  INDEX ((col1 + col2), (col3 - col4))
  ```
  This produces an error; the expressions are not enclosed within parentheses:
  ```
  INDEX (col1 + col2, col3 - col4)
  ```
- A functional key part cannot consist solely of a column name. For example, this is not permitted:
  ```
  INDEX ((col1), (col2))
  ```
  Instead, write the key parts as nonfunctional key parts, without parentheses:
  ```
  INDEX (col1, col2)
  ```
- A functional key part expression cannot refer to column prefixes.
- Functional key parts are not permitted in foreign key specifications.

- For CREATE TABLE ... LIKE, the destination table preserves functional key parts from the original table

Functional indexes are implemented as hidden virtual generated columns, which has these implications:

- Each functional key part counts against the limit on total number of table columns.
- Functional key parts inherit all restrictions that apply to generated columns. Examples:
- Only functions permitted for generated columns are permitted for functional key parts.
- Subqueries, parameters, variables, stored functions, and user-defined functions are not permitted.

2.1.3 Unique Indexes

A UNIQUE index creates a constraint such that all values in the index must be distinct. An error occurs if you try to add a new row with a key value that matches an existing row. If you specify a prefix value for a column in a UNIQUE index, the column values must be unique within the prefix length. A UNIQUE index permits multiple NULL values for columns that can contain NULL.

2.1.4 Multi-Valued Indexes

You can create a multi-valued index in a CREATE TABLE, ALTER TABLE, or CREATE INDEX statement. This requires using CAST(... AS ... ARRAY) in the index definition, which casts same-typed scalar values in a JSON array to an SQL data type array. A virtual column is then generated transparently with the values in the SQL data type array; finally, a functional index (also referred to as a virtual index) is created on the virtual column. It is the functional index defined on the virtual column of values from the SQL data type array that forms the multi-valued index.

The following example a multi-valued index zips can be created on an array $.zipcode on a JSON column custinfo in a table named customers. In each case, the JSON array is cast to an SQL data type array of UNSIGNED integer values.

```
CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    modified DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    custinfo JSON,
    INDEX zips( (CAST(custinfo->'$.zip' AS UNSIGNED
ARRAY)) )
    );
```

A multi-valued index can also be defined as part of a composite index. This example shows a composite index that includes two single-valued parts (for the id and modified columns), and one multi-valued part (for the custinfo column):

ALTER TABLE customers ADD INDEX comp(id, modified,

(CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)));

Now inset values in customer table:

```
INSERT INTO customers VALUES
    (NULL, NOW(),
'{"user":"Jack","user_id":37,"zipcode":[94582,94536]}'),
    (NULL, NOW(),
'{"user":"Jill","user_id":22,"zipcode":[94568,94507,94582]
}'),
    (NULL, NOW(),
'{"user":"Bob","user_id":31,"zipcode":[94477,94507]}'),
    (NULL, NOW(),
'{"user":"Mary","user_id":72,"zipcode":[94536]}'),
    (NULL, NOW(),
'{"user":"Ted","user_id":56,"zipcode":[94507,94582]}');
```

The optimizer uses a multi-valued index to fetch records when the following functions are specified in a WHERE clause:

Using MEMBER OF()

```
SELECT * FROM customers
    WHERE 94507 MEMBER OF(custinfo->'$.zipcode');
```

Using JSON_CONTAINS()

```
SELECT * FROM customers
    WHERE JSON_CONTAINS(custinfo->'$.zipcode',
CAST('[94507,94582]' AS JSON));
```

Using JSON_OVERLAPS()

```
SELECT * FROM customers
    WHERE JSON_CONTAINS(custinfo->'$.zipcode',
CAST('[94507,94582]' AS JSON));
```

2.1.5 Spatial Indexes

The MyISAM, InnoDB, NDB, and ARCHIVE storage engines support spatial columns such as POINT and GEOMETRY. However, support for spatial column indexing varies among engines. Spatial and nonspatial indexes on spatial columns are available according to the following rules.

Spatial indexes on spatial columns have these characteristics:

- ☞ Available only for InnoDB and MyISAM tables. Specifying SPATIAL INDEX for other storage engines results in an error.
- ☞ An index on a spatial column must be a SPATIAL index. The SPATIAL keyword is thus optional but implicit for creating an index on a spatial column.
- ☞ Available for single spatial columns only. A spatial index cannot be created over multiple spatial columns.
- ☞ Indexed columns must be NOT NULL.
- ☞ Column prefix lengths are prohibited. The full width of each column is indexed.
- ☞ Not permitted for a primary key or unique index.

Nonspatial indexes on spatial columns (created with INDEX, UNIQUE, or PRIMARY KEY) have these characteristics:

- ☞ Permitted for any storage engine that supports spatial columns except ARCHIVE.
- ☞ Columns can be NULL unless the index is a primary key.
- ☞ The index type for a non-SPATIAL index depends on the storage engine. Currently, B-tree is used.
- ☞ Permitted for a column that can have NULL values only for InnoDB, MyISAM, and MEMORY tables.

2.1.6 Index Options

Following the key part list, index options can be given. An index_option value can be any of the following:

❖ KEY_BLOCK_SIZE [=] value

For MyISAM tables, KEY_BLOCK_SIZE optionally specifies the size in bytes to use for index key blocks.

❖ index_type

Some storage engines permit you to specify an index type when creating an index.

Storage Engine	Permissible Index Types
InnoDB	BTREE

Storage Engine	Permissible Index Types
MyISAM	BTREE
MEMORY/HEAP	HASH, BTREE
NDB	HASH, BTREE

The index_type clause cannot be used for FULLTEXT INDEX or (prior to MySQL 8.0.12) SPATIAL INDEX specifications

❖ WITH PARSER parser_name

This option can be used only with FULLTEXT indexes. It associates a parser plugin with the index if full-text indexing and searching operations need special handling. InnoDB and MyISAM support full-text parser plugins.

❖ COMMENT 'string'

Index definitions can include an optional comment of up to 1024 characters.

❖ VISIBLE, INVISIBLE

Specify index visibility. Indexes are visible by default. An invisible index is not used by the optimizer. Specification of index visibility applies to indexes other than primary keys.

2.1.7 Table Copying and Locking Options

ALGORITHM and LOCK clauses may be given to influence the table copying method and level of concurrency for reading and writing the table while its indexes are being modified. They have the same meaning as for the ALTER TABLE statement.

2.2 Dropping an Index

DROP INDEX drops the index named index_name from the table tbl_name. This statement is mapped to an ALTER TABLE statement to drop the index.

```
DROP INDEX index_name ON tbl_name
    [algorithm_option | lock_option] ...

algorithm_option:
    ALGORITHM [=] {DEFAULT | INPLACE | COPY}

lock_option:

    LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}
```

To drop a primary key, the index name is always PRIMARY, which must be specified as a quoted identifier because PRIMARY is a reserved word:

```
DROP INDEX `PRIMARY` ON t;
```

3 Exercise

1. Create a trigger on the table employees, which after an update or insert, converts all the values of first and last names to upper case. (Hint: Use cursors to retrieve the values of each row and modify them.)
2. Create a trigger that restores the values before an update operation on the employees table if the salary exceeds 100000. (Hint: Use the inserted and deleted tables to look at the old and new values in the table respectively.)

**********************************END***************************************