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.

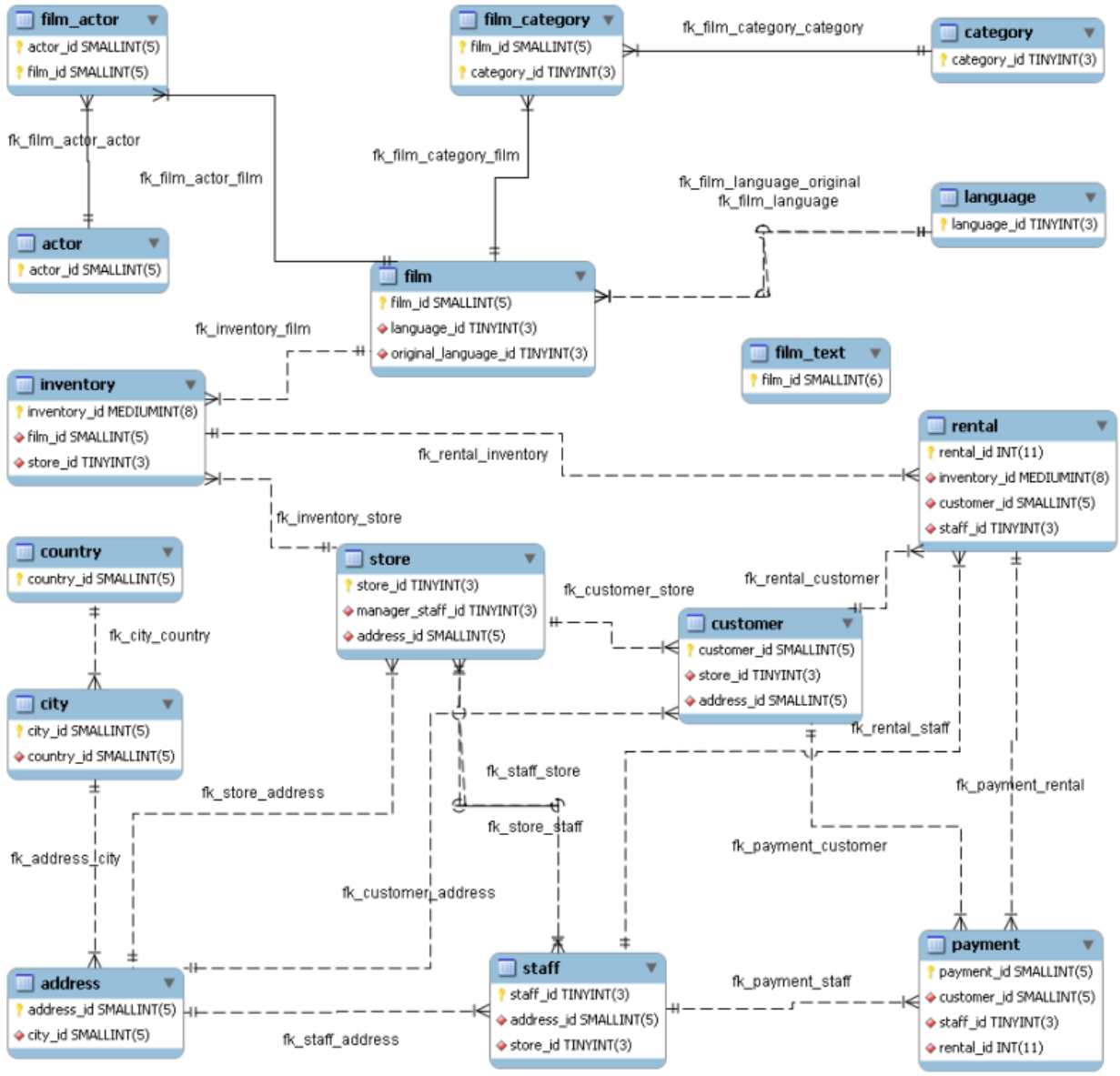

Figure 1: "Sakila" Database EER Diagram.

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**********