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 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;
```

Alternatively, if you are trying to create a view by specifying the column list, you can execute the following command.

```
create view stview_2(c1, c2, c3) 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_cred* and these changes are reflected the view *stview*.

```
update university2.student
set student.tot_cred = 100
where student.ID = 11126;
```

Now try following query and check the effect.

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

Q. Create a view of the instructor table, and try updating the instructor table where the salary is null. Check if the results are reflected in the view created.

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

Q. 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 can retrieve information directly from this view. Try following query.

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

Notice that 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.

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 drops the st_view.

2 Window Functions

Window functions are SQL functions that operate on a set of records called the “window” or the “window frame”. The “window” is a set of rows that are somehow related to the row currently being processed by the query. They are used for tasks such as calculating cumulative sums, moving averages, or accessing the value from the previous row of the same result set.

2.1 Aggregate Window Functions

Aggregate functions are those that you use with `GROUP BY`. This includes:

- `COUNT()` counts the number of rows within the window.
- `AVG()` average value for a given column for all the records in the window.
- `MAX()` obtains the maximum value of a column for all the records in the window.
- `SUM()` returns the sum of all values in a given column within the window.

The main difference between window functions and aggregate functions with group by is that aggregate functions group multiple rows into a single result row; all the individual rows in the group are collapsed and their individual data is not shown. On the other hand, window functions produce a result for each individual row. This result is usually shown as a new column value in every row within the window.

```
SELECT dept_name, name, AVG(salary) OVER (PARTITION BY dept_name) AS
total_salary FROM instructor;
```

The clause `OVER (PARTITION BY dept_name)` creates a window of rows for each value in the department column. All the rows with the same value in the department column will belong to the same window. The `AVG()` function is applied to the window: the query computes the average salary in the given department.

Q List the total number of instructors in each department:

```
SELECT dept_name, COUNT(ID) OVER (PARTITION BY dept_name) AS
total_instructors FROM instructor;
```

Q Rank departments by their budget in descending order:

```
SELECT dept_name, budget, RANK() OVER (ORDER BY budget DESC) AS
budget_rank  FROM department;
```

Q List instructors and the number of courses they're teaching each semester, along with the total number of courses taught in that semester:

```
SELECT i.ID, i.name, s.semester, s.year,
       COUNT(t.course_id) OVER (PARTITION BY i.ID, s.semester, s.year)
AS courses_taught,
       COUNT(t.course_id) OVER (PARTITION BY s.semester, s.year) AS
total_courses
FROM instructor i
JOIN teaches t ON i.ID = t.ID
JOIN section s ON t.course_id = s.course_id AND t.sec_id = s.sec_id
AND t.semester = s.semester AND t.year = s.year;
```

2.2 Analytical Window Functions

Function	Description

LEAD(column, offset,default)	Returns a column value from the row following the current row, based on the specified offset. The default value is optional and is returned if the offset goes beyond the scope of the window.
LAG(column, offset, default)	Returns a column value from the row preceding the current row, based on the specified offset. The default value is optional and is returned if the offset goes beyond the scope of the window.
FIRST_VALUE(column)	Returns the first value in an ordered set of values within the window frame.
LAST_VALUE(column)	Returns the last value in an ordered set of values within the window frame. Adjustments with 'ROWS BETWEEN' or 'RANGE BETWEEN' may be necessary to get the expected result.
NTH_VALUE(column, n)	Returns the value of the specified nth row in a window frame (counting from 1). Requires specifying the column and the nth position.
NTILE()	Divides the rows in a partition into n groups and assigns a group number to each row. Requires the number of groups n as an argument.
PERCENT_RANK()	Computes the percentile rank of a row within a partition, with 0 as the lowest rank and 1 as the highest. No arguments are required.
CUME_DIST()	Calculates the cumulative distribution of a value within a partition, as the fraction of rows with smaller or equal values. No arguments are required.
ROW_NUMBER()	Assigns a unique number to each row starting from 1 within a partition, ordered by the specified column(s).
RANK()	Assigns a rank to each row within a partition, with ties receiving the same rank and leaving gaps in the ranking sequence for ties.
DENSE_RANK()	Similar to RANK(), but without gaps in the ranking sequence for ties, meaning if two rows tie for a rank, the next rank in the sequence is not incremented.

Q Assigns a row number to each payment done by each customer ordered by payment date.

```
SELECT customer_id, payment_id, payment_date,
ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY
payment_date) AS num
FROM payment;
```

Notice the difference between the ROW_NUMBER(), RANK(), and DENSE_RANK() analytic functions with this query :-

```
SELECT
    course_id,
    title,
```

```
    dept_name,
    credits,
    ROW_NUMBER() OVER (ORDER BY credits DESC) AS row_no,
    RANK() OVER (ORDER BY credits DESC) AS rnk,
    DENSE_RANK() OVER (ORDER BY credits DESC) AS dense_rnk
FROM
    course
ORDER BY
    credits DESC;
```

Q Fetch the first 3 students who scored an A grade in every course ordered by course ID.

```
SELECT *
FROM (
    SELECT
        c.course_id,
        c.title AS course_name,
        st.name AS student_name,
        tk.grade,

      ROW_NUMBER() OVER (PARTITION BY tk.course_id ORDER BY
tk.grade) as rank_in_course

    FROM student st
    JOIN takes tk ON st.ID = tk.ID
    JOIN course c ON tk.course_id = c.course_id

) AS ranked_students

WHERE ranked_students.rank_in_course <= 3
ORDER BY course_id, rank_in_course;
```

Q Ranks films by rental rate within their rating category.

```
SELECT rating, title, rental_rate,
RANK() OVER (PARTITION BY rating ORDER BY rental_rate DESC) AS
rank FROM film;
```

Q Ranks customers by the amount spent on rentals.

```
SELECT customer_id, SUM(amount) AS total_spent,
RANK() OVER (ORDER BY SUM(amount) DESC) AS spend_rank
FROM payment
GROUP BY customer_id;
```

Q Densely ranks films by length within their rating category.

```
SELECT rating, title, length,
```

```
DENSE_RANK() OVER (PARTITION BY rating ORDER BY length DESC) AS
dense_rank FROM film;
```

Q Densely ranks customers by the amount spent on rentals.

```
SELECT customer_id, SUM(amount) AS total_spent,
DENSE_RANK() OVER (ORDER BY SUM(amount) DESC) AS spend_dense_rank
FROM payment GROUP BY customer_id;
```

2.2.1 NTILE()

This function distributes the rows in an ordered partition into a specified number of approximately equal groups.

Examples:

Q Divides the films into 4 groups based on their length.

```
SELECT title, length,
NTILE(4) OVER (ORDER BY length) AS quartile
FROM film;
```

2.2.2 LEAD() and LAG()

These functions access data from a subsequent row (LEAD) or previous row (LAG) in the result set, without the need for a self-join.

Examples:

Q Gets the title of the next film in the inventory based on film_id.

```
SELECT film_id, title,
LEAD(title) OVER (ORDER BY film_id) AS next_film_title
FROM film;
```

Q List all rentals, showing the number of days elapsed since each customer's last rental, with a default value for the first rental, ordered by the longest time since the last rental.

```
SELECT
    customer_id,
    rental_id,
    rental_date,
LAG(rental_date, 1, '2000-01-01') OVER (PARTITION BY customer_id ORDER
BY rental_date) AS previous_rental_date,
DATEDIFF(rental_date, LAG(rental_date, 1, '2000-01-01') OVER
(PARTITION BY customer_id ORDER BY rental_date)) AS
days_since_last_rental
```

```
FROM rental
ORDER BY days_since_last_rental desc,customer_id, rental_date;
```

3 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 literals. 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.

3.1 Numeric

SQL can evaluate simple arithmetic expressions containing numeric columns and literals. 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 percentage of credit obtained from table student. Assume that 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 given a NULL parameter value return NULL.

3.2 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.

3.2.1 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';
```

3.2.2 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;
```

3.2.3 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;
```

3.2.4 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;
```

3.2.5 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().

3.2.6 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

3.3 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');
```

3.3.1 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.

3.3.2 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');
```

3.3.3 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');
```

3.3.4 Other Functions for Working with Dates

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

4 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.
2. Write the following queries in SQL.
    a. Get the dense rank of courses in each department based on the number of sections offered.

b. Calculate the percent rank of each instructor's salary within their department:
c. Calculates a running total of rental payment amounts.
d. Calculates the running average of rental payment amounts.
e. Rank the departments based on the total number of courses offered.

3. Answer the following Questions:
    a. What is a view, and why use it?
    b. Can we create a view based on another view?
    c. Can we still use a view if the original table is deleted?
    d. Describe the Difference Between Window Functions and Aggregate Functions .
    e. What is the difference between the DELETE and TRUNCATE statements?

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