Views and window functions

Views

> A view is a virtual table based on the result-set of an SQL statement. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

We often want to run queries that return a denormalized view of the data. For example, we may want to see the name of the student along with the name of the batch. We can do this by joining the `students` and `batches` tables.

But, this would require us to write the same query again and again. Creating such a table would be a waste of space and time. Also, we would have to update the table every time we update the `students` or `batches` table. This is where views come in. Views allow us to create a virtual table that is based on the result of a query.

Views are created using the `CREATE VIEW` statement. The syntax is as follows:

```sql
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition;
```

So in order to create a view that shows the name of the student along with the name of the batch, we can use the following query:

```sql
CREATE VIEW student_batch_view AS
SELECT
    s.first_name, s.last_name, b.name
FROM
    students s
        JOIN
    batches b ON s.batch_id = b.id;
```

We can now use this view as if it were a table. For example, we can use the `SELECT` statement to get all the students in the batches.

```
SELECT
    *
FROM
    student_batch_view;
```

Views provide a way to encapsulate complex queries. They can be used to hide the complexity of the query from the user. They can also be used to restrict access to the data. The following are some of the advantages of views –

- `Views can hide complexity` - If you have a query that requires joining several tables, or has complex logic or calculations, you can code all that logic into a view, then select from the view just like you would a table.

- `Views can be used as a security mechanism` - A view can select certain columns and/or rows from a table (or tables), and permissions set on the view instead of the underlying tables. This allows surfacing only the data that a user needs to see.

- `Views can simplify supporting legacy code` - If you need to refactor a table that would break a lot of code, you can replace the table with a view of the same name. The view provides the exact same schema as the original table, while the actual schema has changed. This keeps the legacy code that references the table from breaking, allowing you to change the legacy code at your leisure.

CRUD operations

Views are majorly used for reading data. However, they can also be used to create, update and delete data. The following are the CRUD operations that can be performed on views.

- `UPDATE` - Views can be used to update data. However, the view must have all the columns of the underlying table. The view must also have a `WHERE` clause that specifies which rows to update. The `UPDATE` statement can be used to update the data in the view.
- `CREATE` - New rows can be added to the view and the underlying table using the `INSERT` statement. However, the view must have all the columns of the underlying table and the view must have only one underlying table.

Window Functions

MySQL has supported window functions since version 8.0. The window functions allow you to solve query problems in new, easier ways and with better performance.

Window functions operate on a window frame, or a set of rows that are somehow related to the current row. They are similar to GROUP BY, because they compute aggregate values for a group of rows. However, unlike GROUP BY, they do not collapse rows; instead, they keep the details of individual rows.

Let's see an example of a window function with our `Jedi Academy` database:

```
CREATE TABLE `students` (
    `id` int NOT NULL AUTO_INCREMENT,
    `first_name` varchar(255) NOT NULL,
    `last_name` varchar(255) NOT NULL,
    `email` varchar(255) NOT NULL,
    `phone` varchar(255),
    `birth_date` date,
    `address` varchar(255),
    `iq` int,
    `batch_id` int,
    PRIMARY KEY (`id`)
);
```

How can I get the student names along with the number of students in their batch? We would have to use a subquery to get the number of students in each batch.

```
SELECT
    id,
    first_name,
    last_name,
    batch_id,
    (SELECT
            COUNT(*)
        FROM
            students s2
        WHERE
            s2.batch_id = s.batch_id)
FROM
    students s;
```

Another way to do the same thing is to use a window function which partitions the data into batches and computes the number of students in each batch.

```
SELECT
    id,
    first_name,
    last_name,
    batch_id,
    COUNT(*) OVER (PARTITION BY batch_id) batch_students
FROM
    students;
```

Like the aggregate functions with the GROUP BY clause, window functions also operate on a subset of rows but they do not reduce the number of rows returned by the query. In this example, the SUM() function works as a window function that operates on a set of rows defined by the contents of the OVER clause. A set of rows to which the SUM() function applies is referred to as a window.

Syntax

```
window_function_name(expression) OVER (
    [partition_definition]
    [order_definition]
    [frame_definition]
)
```

The partition_clause breaks up the rows into chunks or partitions. Two partitions are separated by a partition boundary. The window function is performed within partitions and re-initialized when crossing the partition boundary. You can specify one or more expressions in the PARTITION BY clause. Multiple expressions are separated by commas.

The partition_clause syntax looks like the following:

```
PARTITION BY <expression>[{,<expression>...}]
```

e.g.

```
PARTITION BY batch_id
```

The `order_by_clause` has the following syntax:

```
ORDER BY <expression> [ASC|DESC], [{,<expression>...}]
```

The ORDER BY clause specifies how the rows are ordered within a partition. It is possible to order data within a partition on multiple keys, each key is specified by an expression. Multiple expressions are also separated by commas.

Similar to the PARTITION BY clause, the ORDER BY clause is also supported by all the window functions. However, it only makes sense to use the ORDER BY clause for order-sensitive window functions.

Rank function

The RANK() function is used mainly to create reports. It computes the rank for each row in the result set in the order specified.

The ranks are sequential numbers starting from 1. When there are ties (i.e., multiple rows with the same value in the column used to order), these rows are assigned the same rank. In this case, the rank of the next row will have skipped some numbers according to the quantity of the tied rows. For this reason, the values returned by RANK() are not necessarily consecutive numbers.

How can we get the students ranked by their IQs?

```
SELECT
    id,
    first_name,
    last_name,
    batch_id,
    iq,
    RANK() OVER(ORDER BY iq DESC) AS rank_number
FROM
    students s;
```

After the RANK(), we have an OVER() clause with an ORDER BY. The ORDER BY is mandatory for ranking functions. Here, the rows are sorted in ascending order according to the column ranking_score. The order is ascending by default; you may use ASC at the end of the ORDER BY clause to clarify the ascending order, but it is not necessary.

How can we get the students ranked by their IQs in their batch?

```
SELECT
    id,
    first_name,
    last_name,
    batch_id,
    iq,
    RANK() OVER(PARTITION BY batch_id ORDER BY iq DESC) AS rank_number
FROM
    students s;
```

Row Number function

Another popular ranking function used in databases is ROW_NUMBER(). It simply assigns consecutive numbers to each row in a specified order.

How can we get the students ranked by their IQs using the row number function?

```
SELECT
    id,
    first_name,
    last_name,
    iq,
    ROW_NUMBER() OVER(ORDER BY iq DESC) as rank_score
FROM
    students s;
```

The query first orders the rows by iq in descending order. It then assigns row numbers consecutively starting with 1. The rows with ties in ranking_score are assigned different row numbers, effectively ignoring the ties.

Other window functions

- CUME_DIST
- DENSE_RANK
- FIRST_VALUE
- LAST_VALUE
- LEAD
- LAG

Practice problems

- Window functions - I
- Window functions - II
- Window functions - III
- Window functions - IV
- Window functions - V
- Window functions - VI
- Window functions - VII