Window functions, query optimisation and stored procedures

Agenda

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

Stored procedures

Stored procedures are a way to store and reuse SQL statements. They are stored in the database and can be called from any application that can connect to the database. Stored procedures are compiled and stored in the database. They are compiled only once and then executed as many times as needed. This makes them faster than regular SQL statements.

Let us say we want to update the address of our students in the database. We would have to write the following SQL statement:

```
UPDATE students SET address = 'Bengaluru' WHERE address = 'London';
```

Now we would want to get the results of this query. We would have to write the following SQL statement:

```
SELECT * FROM students;
```

Storing these two SQL statements in a stored procedure would make it easier to update the address of our students. We would only have to call the stored procedure and the address of all our students would be updated and we would get the results of the query.

We can also define parameters for our stored procedures. Let us say we want to update the address of our students in the database based on the city they live in.

Syntax

We can use the `CREATE PROCEDURE` statement to create a stored procedure. A stored procedure can also have arguments. The arguments are passed to the stored procedure when it is called.

```
CREATE PROCEDURE procedure_name
(
    [parameter_name data_type]
)
BEGIN
    [SQL statement]
END
```

Creating a stored procedure for updating the address of our students would look like this:

```
CREATE PROCEDURE UpdateAddress(
    IN origin VARCHAR(255),
    IN target VARCHAR(255)
)
BEGIN
    UPDATE students SET address = target WHERE address = origin;
SELECT
    *
FROM
```

```
    `students`;
END
```

To call a stored procedure, we use the `CALL` statement.

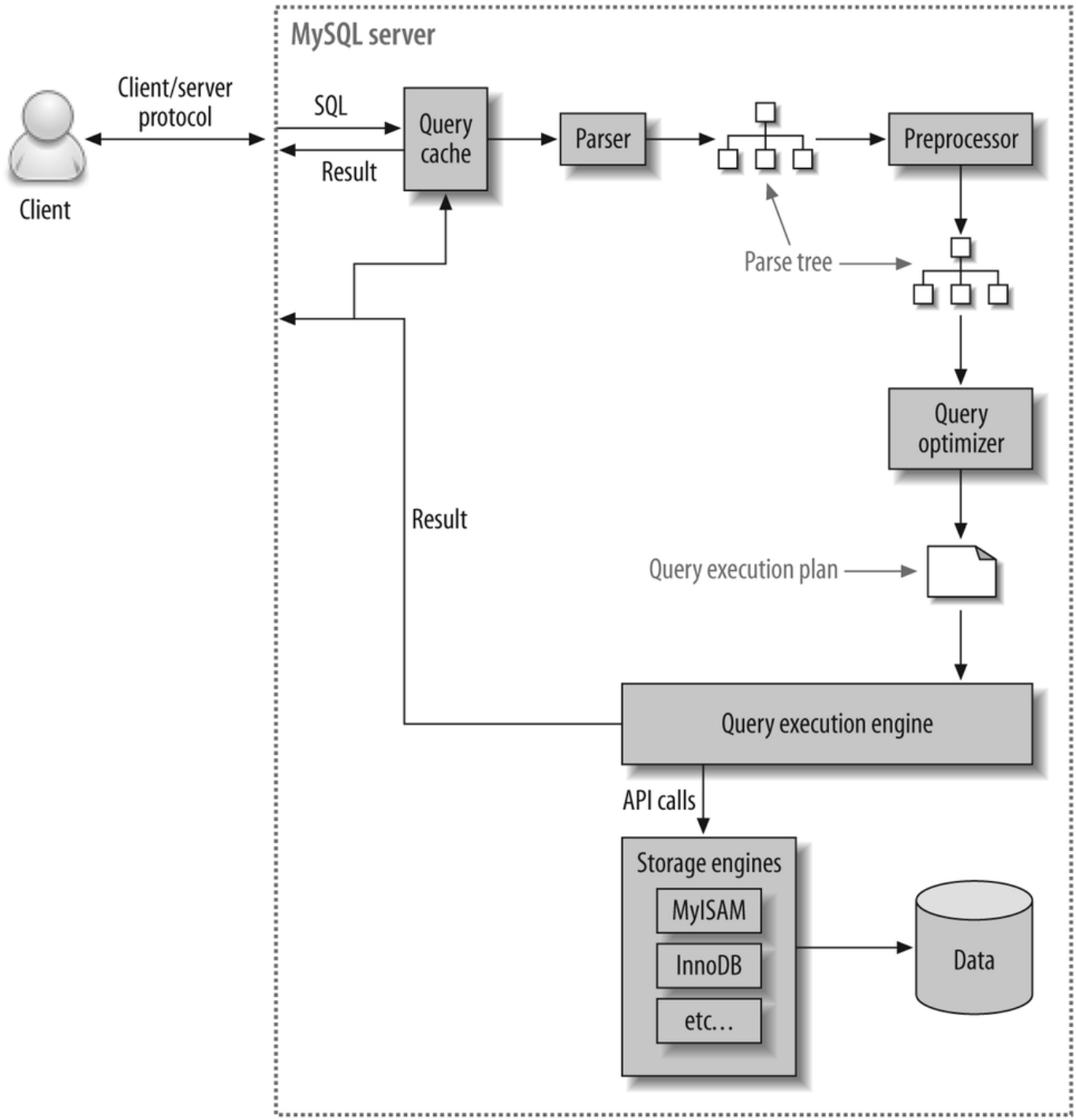
```
CALL UpdateAddress('London', 'Bengaluru');
```

Read more about stored procedures here.

Query Execution

The following steps happen when you execute a query:

1. The client sends the SQL statement to the server.
2. The server checks the query cache. If there's a hit, it returns the stored result from the cache; otherwise, it passes the SQL statement to the next step.
3. The server parses, preprocesses, and optimizes the SQL into a query execution plan.
4. The query execution engine executes the plan by making calls to the storage engine API.
5. The server sends the result to the client.



To begin, MySQL's parser breaks the query into tokens and builds a "parse tree" from them. The parser uses MySQL's SQL grammar to interpret and validate the query. For instance, it ensures that the tokens in the query are valid and in the proper order, and it checks for mistakes such as quoted strings that aren't terminated.

The preprocessor then checks the resulting parse tree for additional semantics that the parser can't resolve. For example, it checks that tables and columns exist, and it resolves names and aliases to ensure that column references aren't ambiguous.

Next, the preprocessor checks privileges. This is normally very fast unless your server has large numbers of privileges.

MySQL uses a cost-based optimizer, which means it tries to predict the cost of various execution plans and choose the least expensive. The unit of cost is a single random four-kilobyte data page read.

Types of scans

Full table scans

A full table scan (also known as a sequential scan) is a scan made on a database where each row of the table is read in a sequential (serial) order and the columns encountered are checked for the validity of a condition. Full table scans are usually the slowest method of scanning a table due to the heavy amount of I/O reads required from the disk which consists of multiple seeks as well as costly disk to memory transfers.

Full Index scan

If your table has a clustered index and you are firing a query that needs all or most of the rows i.e. query without WHERE or HAVING clause, then it uses an index scan. It works similar to the table scan, during the query optimization process, the query optimizer takes a look at the available index and chooses the best one, based on information provided in your joins and where clause, along with the statistical information database keeps.

The main difference between a full table scan and an index scan is that because data is sorted in the index tree, the query engine knows when it has reached the end of the current it is looking for. It can then send the query, or move on to the next range of data as necessary

Index Range scan

Index range scan is a common operation for accessing selective data. It can be bounded (bounded on both sides) or unbounded (on one or both sides). Data is returned in the ascending order of index columns. Multiple rows with identical values are sorted.

Index seek

When your search criterion matches an index well enough that the index can navigate directly to a particular point in your data, that's called an index seek. It is the fastest way to retrieve data in a database. The index seeks are also a great sign that your indexes are being properly used.

This happens when you specify a condition in WHERE clause like searching an employee by id or name if you have a respective index.

Some guidelines for optimizing MySQL queries

- Avoid using functions in predicates

  ```
  SELECT * FROM students where upper(phone) = '123';
  ```

  Because of the UPPER() function, the database doesn't utilize the index on COL1. If there isn't any way to avoid that function in SQL, you will have to create a new function-based index or have to generate custom columns in the database to improve performance.
- Avoid using a wildcard (%) at the beginning of a predicate

```
SELECT * FROM students where phone like '%123';
```

The wildcard causes a full table scan.

- Avoid unnecessary columns in SELECT clause Instead of using 'SELECT *', always specify columns in the SELECT clause to improve MySQL performance. Because unnecessary columns cause additional load on the database, slowing down its performance as well whole systematic process.
- Pagination
- Avoid SELECT DISTINCT