Sub-queries, views and query optimization

Sub-queries

> A subquery is a SQL query nested inside a larger query. A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

You can use a sub-query in a SELECT, INSERT, DELETE, or UPDATE statement to perform the following tasks:

- Compare an expression to the result of the query.
- Determine if an expression is included in the results of the query.
- Check whether the query selects any rows.

A sub-query may occur in :

- A SELECT clause
- A FROM clause
- A WHERE clause

There are a few rules that sub-queries must follow –

- Sub-queries must be enclosed within parentheses.
- A sub-query can have only one column in the SELECT clause, unless multiple columns are in the main query for the sub-query to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY.
- Sub-queries that return more than one row can only be used with multiple value operators such as the IN operator.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Let us consider the `students` table below

id	first_name	last_name	batch_id	iq
1	John	Watson	1	120
2	Mycroft	Holmes	1	160
3	Moriarty	Patel	2	160

Now, we want all the students who are greater than the average IQ. We can compute the average IQ using the `AVG` function but the aggregated value can not be used within the WHERE clause. So, let us try to use a sub-query.

```
SELECT
    first_name, last_name, iq
FROM
    students
WHERE
    iq > (SELECT
            AVG(iq)
        FROM
            students);
```

This would first compute the average IQ of all the students and then compare the average IQ to the IQ of the current student. Since the sub-query returns a single value, it can be used in the WHERE clause without any special handling.

Let us modify the original query to see an example where the sub-query returns multiple values. We want all the students who have IQs greater than the students of batch_id 2;

One way to handle this query is to use the MAX function to get the highest IQ of the batch_id 2 students.

```
SELECT
    *
FROM
    students
WHERE
    iq > (SELECT
            MAX(iq)
        FROM
            students
        WHERE
            batch_id = 2);
```

Another way would be to use the `ALL` keyword. ALL means that the condition will be true only if the operation is true for all values in the range.

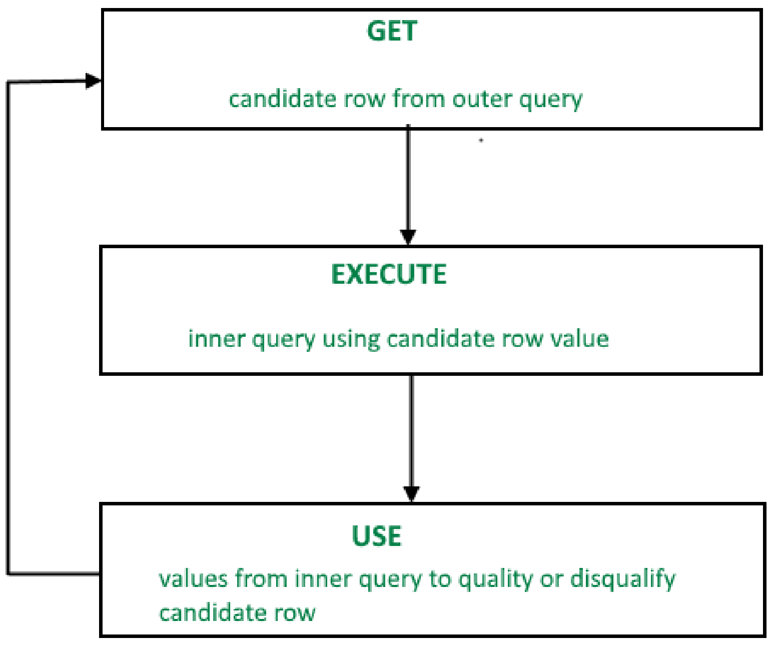
```
SELECT
    *
```

```
FROM
    students
WHERE
    iq > ALL (SELECT
            iq
        FROM
            students
        WHERE
            batch_id = 2);
```

Correlated Sub-queries

> A correlated subquery (also known as a synchronized subquery) is a subquery (a query nested inside another query) that uses values from the outer query. Because the subquery may be evaluated once for each row processed by the outer query, it can be slow.

> Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.


GET
candidate row from outer query
EXECUTE
inner query using candidate row value
USE
values from inner query to quality or disqualify candidate row

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query.

In other words, you can use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

EXISTS operator

The EXISTS operator tests for existence of rows in the results set of the subquery. If a subquery row value is found the condition is flagged TRUE and the search does not continue in the inner query, and if it is not found then the condition is flagged FALSE and the search continues in the inner query.

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.

```sql
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

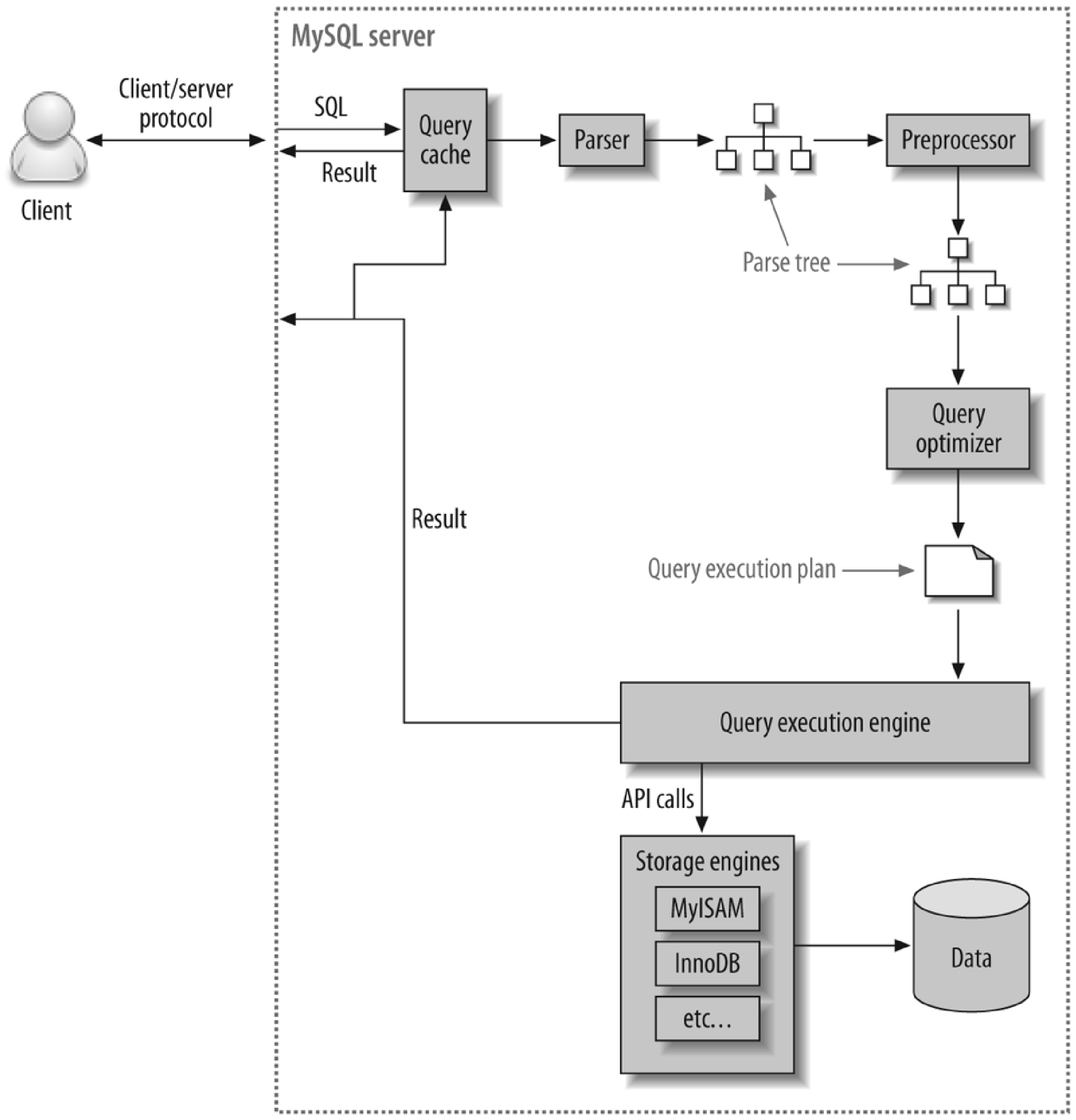
Views are majorly used for reading data. However, they can also be used to create, update and delete data. The following are the CRUD operations that can be performed on views.

- `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.

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

```sql
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

Practice questions

- Sub-queries - I
- Sub-queries - II
- Sub-queries - III
- Sub-queries - IV
- Sub-queries - V
- Sub-queries - VI