HSLU Lucerne University of Applied Sciences and Arts

Database Management for Data Scientists

Exercise 5: SQL Performance

1. Learning objectives

You will learn:

1. Understand **bottlenecks** (I/O vs. memory, amount of data, joins)
2. Apply **indexes** (types, layout and access pattern) to optimize query performance.
3. Show **execution plans** and recognize **good/bad operators** for optimizing queries (explain, scan vs. seek)
4. **Syntactical** SQL performance optimization (good vs. bad SQL, **sargable** queries)

2. Learning video/slides (optional)

Watch the learning video "Presentation SQL Performance". You can find the link on ILIAS.

Review the slides in "Presentation SQL Performance.pdf"

3. Prerequisites

The following exercises are based on the database "RecommenDB" that you have created and loaded in the "MySQL Workbook".

4. Exercise: Hardware (Memory)

You are reviewing a query written by your team member that involves multiple **joins** between the **"bookings"** table and auxiliary tables. The **"bookings"** table is the largest in your database, containing **15 GB of data**, and it is already loaded into the server's **40 GB of dedicated memory**.

The query execution plan suggests that a **full table scan** on the **"bookings"** table might be used to retrieve the data. Your team member argues that the table scan is not a concern since all the data is already in memory, making the operation faster.

Do you agree or disagree with your team member's assessment? Explain the factors that could impact performance in this scenario, and why a table scan might still be problematic.

5. Exercise: Clustered vs. Nonclustered/Secondary Indexes

Task

1. **Explain** at least two important differences between a **Clustered** and a **Nonclustered (Secondary)** index.

2. **Use your favorite AI tool** to assist you in gathering and comparing information on these index types. Provide the prompt that you used to answer the question.

3. **Do you agree** with the answers provided by the AI tool? Provide reasons for your agreement or disagreement.

6. Exercise: Execution Plan and Indexes

You find several (SQL) tools in Appendix A – Meta data queries to gather meta data and performance stats (besides the ones described in the PowerPoint slides).

Preparation

First, make sure there are no indexes **with prefix "exercise"** in the database. If there are any, drop them first (DROP INDEX <IndexName> ON TABLE <TableName>). Use the meta data query in section 11.2 Meta data – All indexes.

Task

- You will create a **first baseline** for gathering the **total number of rows**.
- Then, you will run the **sample query** as a **second baseline** and **measure** the **performance** (duration, rows examined) as well as providing **Execution Plan details**.
- Next, you will **iterate** by **creating** various **indexes**, **re-run** the queries and measure the performance.
- Fill the following table with the gathered data.
- Finally, evaluate and explain the results.

Table

- **Actual rows examined:** Get the value from the Query Stats in MySQL Workbench (or use the query in 11.3 Performance Stats for the last queries).
- **Rows returned:** same as above.
- **Operators / Access Types:** Use either the operators of the graphical Execution Plan or the column "type" in the tabular plan
- **Execution Plan:** Provide the graphical or tabular plan as screenshot.
- **Query Stats:** If using MySQL Workbench, provide a screenshot of the Query Stats.

	Baseline – Sample Query	With single-column index	With composite index
Result			
Duration[1]			
Actual Rows examined			
Rows returned			
Operators / Access Types			
Execution Plan (Screenshot)			
Query Stats (Screenshot)			

Table 1: Sample query - Performance information

[1] Depends on your computer, you will see different numbers than in the solution. Important is the difference from column to column as we progress with indexing.

6.1. Baseline – Total number of rows

Run the following query and gather basic information for the first baseline (either using the Query Stats in MySQL Workbench or use the query in section 11.3).

Fill the information in the following Table 2.

```
SELECT COUNT(*)
FROM Ratings;
```

	Baseline – Total number of rows
Duration[2]	
Actual Rows examined	
Rows returned	

Table 2: Total number of rows – Performance information

6.2. Baseline – Sample Query

Now, start with the sample query, gather the required information and fill them in Table 1, column "Baseline – Sample Query".

```
SELECT COUNT(*)
FROM Ratings
WHERE Rating = 5
    AND YEAR(Timestamp) = 1998;
```

6.3. Single-column index – Sample Query

You might have noticed that the query was quite slow and took a lot of time (depending on your hardware).

- Create an **index** named "exercise_Rating" in the table "Ratings" with the single column "Rating".
- Re-run the sample query.
- Gather the required information and fill them in the table column "With single column index".

6.4. Composite index – Sample Query

Did the query duration improve a lot? Let's give it another try with a different index.

- Create a composite **index** named "exercise_Rating_Timestamp" in the table "Ratings" with the two columns "Rating" and "Timestamp".

[2] Depends on your computer, you will see different numbers than in the solution. Important is the difference from column to column as we progress with indexing.

- Re-run the sample query.
- Gather the required information and fill them in the table column "With composite index".

6.5. Results and evaluation

- What is your overall conclusion of these tests?
- How do you explain the differences between the baseline and each iteration?
- Do you have an explanation why the single-column index might be slower than the baseline sample query?

7. Exercise: Good vs. bad SQL

This exercise will continue with the sample query from the previous section. The composite index might have shown a big performance improvement compared to the baseline query.

Do you see further ways to improve the overall performance?

Task:

- Rewrite the sample query to improve the performance (Hint: see chapter Good vs. bad SQL in the PowerPoint slides).
- The goal is that the rows examined in the Execution Plan decreases to the number of rows found.

Write the performance information in Table 3 column "SQL query improvement":

	SQL query improvement
Result	
Duration[3]	
Actual Rows examined	
Rows returned	
Query Stats	

Table 3: SQL query improvement – Performance information

Results and Evaluation

- What is the performance issue with the **sample query**?
- What performance **improvements** can be **expected** after rewriting the query?

[3] Depends on your computer, you will see different numbers than in the solution. Important is the difference from column to column as we progress with indexing.

- Even with **indexes in place**, why might a **poorly written query** still perform badly?

8. Exercise: Materialized views

Let's evaluate the usage of Materialized views.

Goal

The goal is to get the following information:

- Return Genre Name, Rating Year, Rating Month, Rating, Number of ratings
- Filtered by the year 2015
- Grouped by Genre Name, Rating Year, Rating Month, Rating

Without Materialized view

Start querying without Materialized view but with an index on the Ratings table:

```
-- First, create an index to support the following query
-- It might take some time...
CREATE INDEX exercise_Timestamp ON Ratings (Timestamp, Rating, MovieID);


-- Take a note how long the query takes
SELECT
    g.Name AS GenreName,
    YEAR(mr.Timestamp) AS RatingYear,
    MONTH(mr.Timestamp) AS RatingMonth,
    mr.Rating,
    COUNT(*) AS NumberOfRatings
FROM Ratings AS mr
    INNER JOIN Movie_has_Genre AS h
    ON mr.MovieID = h.MovieID
    INNER JOIN Genres AS g
    ON h.GenreID = g.GenreID
WHERE mr.Timestamp >= '2015-01-01'
    AND mr.Timestamp < '2016-01-01'
GROUP BY
    g.Name,
    YEAR(mr.Timestamp),
    MONTH(mr.Timestamp),
    mr.Rating
ORDER BY
    GenreName,
    RatingYear,
    RatingMonth,
    Rating;
```

The query may take one minute or more depending on your machine.

Materialized view

We could convert the entire query into a Materialized view, but doing so would make it highly specific to that query. A better approach is to use a more general query for the Materialized view.

Let's create a materialized view with month-based granularity, grouped by MovieID and Rating.

```
CREATE TABLE mvRatingsByMonthAndRating
AS
    SELECT
        YEAR(r.Timestamp) AS RatingYear,
        MONTH(r.Timestamp) AS RatingMonth,
        r.Rating,
        r.MovieID,
        COUNT(*) AS NumberOfRatings
    FROM Ratings AS r
    GROUP BY
        YEAR(r.Timestamp),
        MONTH(r.Timestamp),
        r.Rating,
        r.MovieID;

ALTER TABLE mvRatingsByMonthAndRating
ADD CONSTRAINT PRIMARY KEY CLUSTERED
(
  RatingYear,
  RatingMonth,
  Rating,
  MovieID
);
```

If the creation of the Materialized view takes too long, go to Appendix B – Materialized view in batches, to see how you might execute the script in a faster (but also more complex) way.

Use the Materialized view

Instead of the table "Ratings", use now the Materialized view "mvRatingsByMonthAndRating".

```
SELECT
    g.Name AS GenreName,
    mr.RatingYear,
    mr.RatingMonth,
    mr.Rating,
    SUM(mr.NumberOfRatings) AS NumberOfRatings
FROM mvRatingsByMonthAndRating AS mr
    INNER JOIN Movie_has_Genre AS h
    ON mr.MovieID = h.MovieID
    INNER JOIN Genres AS g
```

```
    ON h.GenreID = g.GenreID
WHERE mr.RatingYear = 2015
GROUP BY
    g.Name,
    mr.RatingYear,
    mr.RatingMonth,
    mr.Rating
ORDER BY
    GenreName,
    RatingYear,
    RatingMonth,
    Rating;
```

Results and evaluation

Compare the duration of the query with the Ratings table and the duration with the Materialized view:

- How long did both queries take?
- Use your **favorite AI tool** to get the advantages and disadvantages of Materialized Views. Provide the prompt that you used to answer the task.
- Also ask the tool, how MySQL supports Materialized Views. Based on this information, would you recommend Materialized Views in general and in MySQL?

9. Semester project

- Show the status of your project.
- Show first performance evaluations/improvements that you have built for the queries (if already done).
  - Hint: Use the query that is filtered with a WHERE clause, otherwise an index might not help.

10. Discussion about exercise and presentation

- One group (or more) will present their findings from this exercise to the class.
- Please show the exercise, the status of your project as well as a glimpse of your project report.

11. Appendix A – Meta data queries

Database products offer a variety of tools to get meta data and performance information. Some of these tools (or SQL statements) are provided in the following sections.

11.1. Meta data – All table columns

Provides all table columns including the columns (here in Database "RecommenDB"):

```
SELECT
    TABLE_NAME,
    COLUMN_NAME,
    ORDINAL_POSITION,
    IS_NULLABLE,
    DATA_TYPE,
    CHARACTER_MAXIMUM_LENGTH,
    NUMERIC_PRECISION,
    NUMERIC_SCALE
FROM information_schema.COLUMNS AS isc
WHERE isc.TABLE_SCHEMA = 'RecommenDB'
ORDER BY isc.TABLE_NAME, isc.ORDINAL_POSITION;
```

11.2. Meta data – All indexes

Provides all indexes including the columns (here in Database "RecommenDB"):

```
SELECT
    TABLE_NAME,
    INDEX_NAME,
    GROUP_CONCAT(COLUMN_NAME ORDER BY SEQ_IN_INDEX) AS INDEX_COLUMNS,
    NON_UNIQUE
FROM information_schema.STATISTICS
WHERE TABLE_SCHEMA = 'RecommenDB'
GROUP BY TABLE_NAME, INDEX_NAME, NON_UNIQUE
ORDER BY TABLE_NAME, INDEX_NAME;
```

11.3. Performance Stats for the last queries

Get the performance statistics for the last executed queries. You might need to filter for your query. Be aware that queries are not always available in that list.

This query returns the duration, the rows examined and the SQL statement.

```
SET @TableName := 'Ratings';
SELECT
    (TIMER_END - TIMER_START) / 1000000000000 AS DurationInSec,
    ROWS_EXAMINED,
    ROWS_SENT,
    NO_INDEX_USED,
    NO_GOOD_INDEX_USED,
```

```
    SQL_TEXT
FROM performance_schema.events_statements_history
WHERE 1 = 1
     -- We are not interested in stats queries
    AND DIGEST_TEXT NOT LIKE '%performance_schema%'
    AND DIGEST_TEXT LIKE CONCAT('%', @TableName, '%')
ORDER BY TIMER_START DESC;
```

11.4. Tabular Execution Plan – Access Types

If you use EXPLAIN to return a tabular Execution Plan, you get the column "type". You need to understand the most important values called Access Types[4].

Disclaimer:

The following list was created using ChatGPT and serves as a general reference for mapping between tabular and graphical execution plan types/operators in MySQL. Please note that this is not an official mapping, and MySQL may represent or interpret these operators differently in the tabular and graphical execution plans.

Tabular Access Type (EXPLAIN)	Graphical Execution Plan Term	Description
ALL	Full Table Scan	Reads the entire table; worst-case performance
Index	Index Scan	Scans all rows in the index (not the full table but still all rows)
Range	Range Scan	Scans a range of index entries (e.g. BETWEEN, <, >)
ref	Index Lookup / Non-Unique Key Lookup	Uses a non-unique index to find matching rows.
eq_ref	Unique Key Lookup	Uses a unique index to find exactly one row per lookup
const / system	Constant / Table Constant	Optimized away; table treated like a constant (1-row table or indexed PK match)
index_subquery	Index Subquery	Uses an index to resolve a subquery
unique_subquery	Unique Index Subquery	Like eq_ref, but in a subquery

[4] (see https://dev.mysql.com/doc/refman/8.4/en/explain-output.html#explain-join-types)

12. Appendix B – Materialized view in batches

If the creation of the Materialized view takes too long, you can create it in batches.

First, check if there is an open process creating the Materialized view (in one batch):

```
-- Run the first statement
-- If there is an CREATE TABLE mvRatingsByMonthAndRating AS,
-- get the Id from the first column.
SHOW PROCESSLIST;

-- Enter the Id from above and replace ProcessId to stop the process
-- Run SHOW PROCESSLIST again to verify the process has gone
KILL <ProcessId>;
```

Now, you start with the batch processing. In a nutshell, you will

- Drop the table if it already exists.
- Create the Materialized view as "empty" table.
- Add the primary key to the table.
- Fill the Materialized view with batches of several years, first 1990-2000, 2001-2006 etc.

The reason that this approach may be faster is that one batch is smaller and needs less space in the so-called transaction log.

```
-- Drop the Materialized view (table) if it already exists
DROP TABLE IF EXISTS mvRatingsByMonthAndRating;

-- We create the Materialized view as "empty" table
-- The trick is to use a WHERE clause with 1 = 0
-- The result set will be empty
CREATE TABLE mvRatingsByMonthAndRating
AS
    SELECT
        YEAR(r.Timestamp) AS RatingYear,
        MONTH(r.Timestamp) AS RatingMonth,
        r.Rating,
        r.MovieID,
        COUNT(*) AS NumberOfRatings
    FROM Ratings AS r
    WHERE 1 = 0
    GROUP BY
        YEAR(r.Timestamp),
        MONTH(r.Timestamp),
        r.Rating,
        r.MovieID;
```

```
-- Before filling the table, we create the primary key
-- It is late important that the insert statement will
-- have the same order as the primary key
ALTER TABLE mvRatingsByMonthAndRating
ADD CONSTRAINT PRIMARY KEY CLUSTERED
(
    RatingYear,
    RatingMonth,
    Rating,
    MovieID
);


-- Insert the first batch for years from 1900 up to 2000
INSERT INTO mvRatingsByMonthAndRating
(
    RatingYear,
    RatingMonth,
    Rating,
    MovieID,
    NumberOfRatings
)
SELECT
    YEAR(r.Timestamp) AS RatingYear,
    MONTH(r.Timestamp) AS RatingMonth,
    r.Rating,
    r.MovieID,
    COUNT(*) AS NumberOfRatings
FROM Ratings AS r
WHERE r.Timestamp >= '1990-01-01'
    AND r.Timestamp < '2000-01-01'
GROUP BY
    YEAR(r.Timestamp),
    MONTH(r.Timestamp),
    r.Rating,
    r.MovieID
ORDER BY
    RatingYear,
    RatingMonth,
    Rating,
    MovieID;


-- Insert the next batch for years from 2000 up to 2006
INSERT INTO mvRatingsByMonthAndRating
(
    RatingYear,
    RatingMonth,
    Rating,
    MovieID,
    NumberOfRatings
)
```

```
SELECT
    YEAR(r.Timestamp) AS RatingYear,
    MONTH(r.Timestamp) AS RatingMonth,
    r.Rating,
    r.MovieID,
    COUNT(*) AS NumberOfRatings
FROM Ratings AS r
WHERE r.Timestamp >= '2000-01-01'
    AND r.Timestamp < '2006-01-01'
GROUP BY
    YEAR(r.Timestamp),
    MONTH(r.Timestamp),
    r.Rating,
    r.MovieID
ORDER BY
    RatingYear,
    RatingMonth,
    Rating,
    MovieID;


-- Insert the next batch for years from 2006 up to 2012
INSERT INTO mvRatingsByMonthAndRating
(
    RatingYear,
    RatingMonth,
    Rating,
    MovieID,
    NumberOfRatings
)
SELECT
    YEAR(r.Timestamp) AS RatingYear,
    MONTH(r.Timestamp) AS RatingMonth,
    r.Rating,
    r.MovieID,
    COUNT(*) AS NumberOfRatings
FROM Ratings AS r
WHERE r.Timestamp >= '2006-01-01'
    AND r.Timestamp < '2012-01-01'
GROUP BY
    YEAR(r.Timestamp),
    MONTH(r.Timestamp),
    r.Rating,
    r.MovieID
ORDER BY
    RatingYear,
    RatingMonth,
    Rating,
    MovieID;
```

```
-- Insert the next batch for years from 2012 up to 2020
INSERT INTO mvRatingsByMonthAndRating
(
    RatingYear,
    RatingMonth,
    Rating,
    MovieID,
    NumberOfRatings
)
SELECT
    YEAR(r.Timestamp) AS RatingYear,
    MONTH(r.Timestamp) AS RatingMonth,
    r.Rating,
    r.MovieID,
    COUNT(*) AS NumberOfRatings
FROM Ratings AS r
WHERE r.Timestamp >= '2012-01-01'
    AND r.Timestamp < '2020-01-01'
GROUP BY
    YEAR(r.Timestamp),
    MONTH(r.Timestamp),
    r.Rating,
    r.MovieID
ORDER BY
    RatingYear,
    RatingMonth,
    Rating,
    MovieID;


-- It is good practice to verify the result
-- Let’s check if the number of rows in Ratings is equal to
-- the sum of “NumberOfRatings” in the Materialized view
SELECT
    t.RatingsCount,
    t.MVCount,
    t.RatingsCount - t.MVCount AS Diff
FROM
(
  SELECT
    (SELECT COUNT(*) FROM Ratings) AS RatingsCount,
    (SELECT SUM(NumberOfRatings) FROM mvRatingsByMonthAndRating) AS
MVCount
) AS t;
```

Unfortunately, MySQL does not support loops in ad-hoc scripts. You need to create a stored procedure if you want to loop many times.

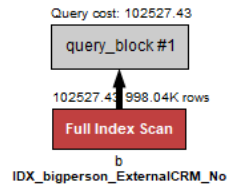
In our case, we only have four to five loops to fill the entire table. A stored procedure is not necessary. But let’s see how to build the Materialized view with the help of a stored procedure.

```
-- First, drop the stored procedures because there is no way
-- to CREATE OR REPLACE / CREATE OR ALTER possibility in MySQL
DROP PROCEDURE IF EXISTS FillRatingsMaterializedViewByBatches;

-- If you have several statements in the stored procedure, the standard
-- delimiter “;” must be changed to a different one because the standard
-- delimiter will be used inside the stored procedure. This is a bit
-- strange but that's how it works in MySQL.
-- Make sure to select and run the entire statement from DELIMITER to
-- DELIMITER to create the stored procedure
DELIMITER //
CREATE PROCEDURE FillRatingsMaterializedViewByBatches(IN startYear int,
IN numberOfYears int)
BEGIN
    SET @YearFrom = CAST(CONCAT(startYear, '-01-01') AS DATE);
    SET @YearUntil = CAST(CONCAT(startYear + numberOfYears, '-01-01') AS
DATE);

    INSERT INTO mvRatingsByMonthAndRating
    (
        RatingYear,
        RatingMonth,
        Rating,
        MovieID,
        NumberOfRatings
    )
    SELECT
        YEAR(r.Timestamp) AS RatingYear,
        MONTH(r.Timestamp) AS RatingMonth,
        r.Rating,
        r.MovieID,
        COUNT(*) AS NumberOfRatings
    FROM Ratings AS r
    WHERE r.Timestamp >= @YearFrom
        AND r.Timestamp < @YearUntil
    GROUP BY
        YEAR(r.Timestamp),
        MONTH(r.Timestamp),
        r.Rating,
        r.MovieID
    ORDER BY
        RatingYear,
        RatingMonth,
        Rating,
        MovieID;

END; //
DELIMITER ;

/*
CREATE INDEX exercise_Timestamp ON Ratings (Timestamp, Rating, MovieID);
*/
```

```
-- Drop the Materialized view (table) if it already exists
DROP TABLE IF EXISTS mvRatingsByMonthAndRating;

-- We create the Materialized view as “empty” table
-- The trick is to use a WHERE clause with 1 = 0
-- The result set will be empty
CREATE TABLE mvRatingsByMonthAndRating
AS
    SELECT
        YEAR(r.Timestamp) AS RatingYear,
        MONTH(r.Timestamp) AS RatingMonth,
        r.Rating,
        r.MovieID,
        COUNT(*) AS NumberOfRatings
    FROM Ratings AS r
    WHERE 1 = 0
    GROUP BY
        YEAR(r.Timestamp),
        MONTH(r.Timestamp),
        r.Rating,
        r.MovieID;

-- Before filling the table, we create the primary key
-- It is later important that the insert statement will
-- have the same order as the primary key
ALTER TABLE mvRatingsByMonthAndRating
ADD CONSTRAINT PRIMARY KEY CLUSTERED (RatingYear, RatingMonth, Rating,
MovieID);

-- Insert the data in batches with a starting year and number of years
CALL FillRatingsMaterializedViewByBatches(1990, 10);
CALL FillRatingsMaterializedViewByBatches(2000,  6);
CALL FillRatingsMaterializedViewByBatches(2006,  6);
CALL FillRatingsMaterializedViewByBatches(2012,  8);

-- It is good practice to verify the result
-- Let’s check if the number of rows in Ratings is equal to
-- the sum of “NumberOfRatings” in the Materialized view
SELECT
    t.RatingsCount,
    t.MVCount,
    t.RatingsCount - t.MVCount AS Diff
FROM
(
  SELECT
    (SELECT COUNT(*) FROM Ratings) AS RatingsCount,
    (SELECT SUM(NumberOfRatings) FROM mvRatingsByMonthAndRating) AS
MVCount
) AS t;
```

13. Appendix C – Execution Plan in MySQL

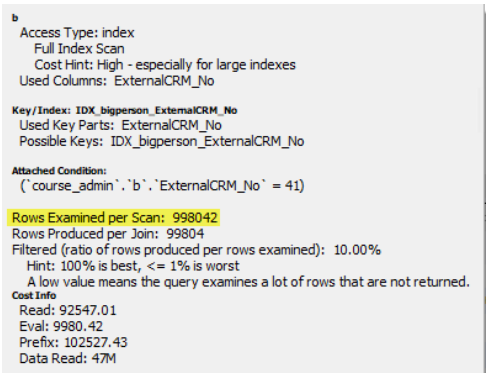
To analyze the execution plan of a query in MySQL, follow these steps:



- Press CTRL+ALT+X to open the Execution Plan window.
- Hover over an operator and you will see the details.
- You can also change to tabular plan (additional information EXPLAIN FORMAT=JSON)

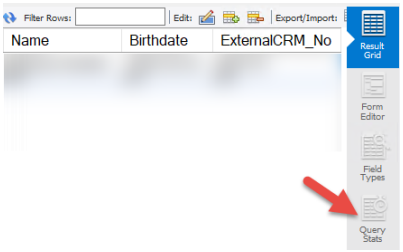
When you hover over any operator in the execution plan, detailed information about that operator will be displayed. This can include things like the type of operation (e.g., SCAN, JOIN), the table being accessed, and other relevant details.

Note: the "Rows Examined per Scan" are **estimates** based on the database's statistics, not actual row counts.



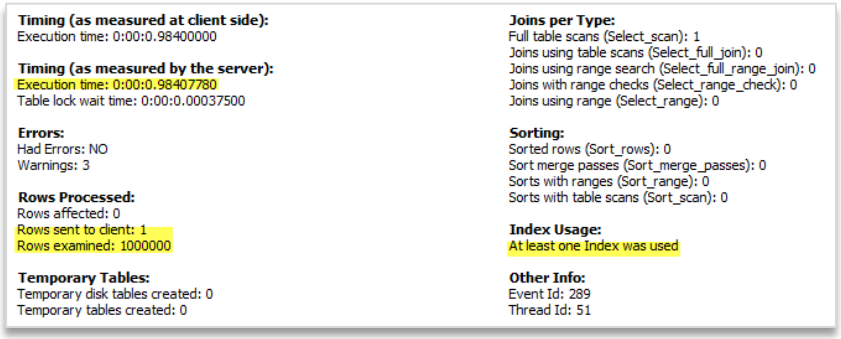
Query Stats

The Query Stats provides actual performance details about a query **after it has been executed**.



On the right side of the result grid, click "Query Stats" to open the panel which offers insights into the query execution.

The panel includes:



- Execution time (the total time taken to execute the query)
- Rows sent to client (the number of rows sent to the client in the result set.)
- Rows examined (the number of rows MySQL had to read from the table(s) to fulfill the query. This can be much larger than the number of rows returned, especially for non-indexed operations and/or inefficient queries)
- Index usage (details about whether indexes were used during query execution and how effectively they were utilized)