[**Meta Database Engineer Professional Certificate**](https://www.coursera.org/professional-certificates/meta-database-engineer)

The purpose of creating stored procedures and functions is to wrap or encapsulate code in the body of a function or procedure.

**Functions –** Code that performs an operation and returns a result.

**–** Accept only input parameters.

**-** used to return specific values.

**Stored procedures -** Accept both input and output parameters.

**-** processing, manipulating and modifying data.

**Variable –** used to pass values between SQL statements, or between a procedure and a SQL statement.

**–** variables are created inside or outside of a stored procedure and select statement.

**Set command –** Assign a value to a variable within a stored procedure.

By using this variable to delete, update, or query the record.

→ You can create and reference a variable inside or outside a stored procedure using the SET command.

→ select \* from orders where orderid = @order\_id;

**create**  a variable inside a stored procedure using the declare command without an @ sign.

→ DECLARE variable\_name DATATYPE DEFAULT VALUE;

→ DECLARE minimum\_order\_cost DECIMAL (5, 2) DEFAULT 0;

**Variable inside SELECT commands**

→ SELECT @variable\_name := value;

→ SELECT @max\_order := MAX(cost) from orders;

to excute use → @max\_order;

→ SELECT function( ) from variable\_name from table\_name;

→ SELECT AVG(cost) INTO @average\_cost from orders;

**IN parameter**

→ Value -> procedure

**Syntax**

Create PROCEDURE procedure\_name(IN logic(Val1, val2,)) SELECT logic;

Create PROCEDURE CalculateTax(IN salary DECIMAL(10,2)) SELECT

salary \* 0.2 AS TAX;

to excute → CALL CalculateTax(1000);

to drop → DROP PROCEDURE procedure\_name

**OUT parameter**

Value → Procedure → Variable

**SYNTAX**

→ CREATE PROCEDURE GetLowestCost(OUT LowestCost DECIMAL(6,2))

SELECT MIN(cost) INTO LowetsCost FROM Orders;

to excute → CALL GetLowestCost(@order\_lowest\_cost)

**INOUT parameter**

→ Value → Procedure → New value

**SYNTAX**

**→** CREATE PROCEDURE SquareAnumber(INOUT aNUMBER INT)

BEGIN

set aNumber = aNumber \* aNumber;

END

SET @x\_number = 5;

to excute → CALL SquareAnumber (@x\_number)

→ SELECT @x\_number;

**Developing user-defined functions**

→ Creating to perform operations that can’t be completed with built-in functions.

User-defined code **→** Task → Result

**user-defined function syntax**

**→** CREATE FUNCTION function\_name()

RETURNS datatype DETERMINISTIC

RETURN

→ CREATE FUNCTION FindTotalCost(Cost DECIMAL(5,2))

RETURNS DECIMAL(5,2) DETERMINISTIC

RETURN (Cost – (Cost \* 0.1));

→ SELECT FindTotalCost(100)

→ **DELIMITER //** → command to compile the whole function as a single compound statement using begin end keywords.

→ **DELIMITER ;** → change the delimiter to the default semicolon so you can use MySQL as usual.

**//to dicount for our customers**

**Eg**: DELIMITER //

create function GetTotalCost(Cost DECIMAL(5,2)) returns DECIMAL(5,2) DETERMINISTIC BEGIN IF (Cost >= 100 AND Cost < 500) THEN SET Cost = Cost - (Cost \* 0.1); ELSEIF (Cost >= 500) THEN SET Cost = Cost - (Cost \* 0.2); END IF; RETURN (Cost); END//

DELIMITER ;

→ select GetTotalCost(500); NB: the output is 400

To delete the function use

→ DROP FUNCTION GetTotalCost;

**Create complex stored procedure**

When creating complex stored procedures, first you must change the delimiter from a semi-colon to another delimiter sign so that MySQL can compile your code in a BEGIN-END block as one compound statement.

DELIMITER //

CREATE PROCEDURE GetProductSummary(OUT NumberOfLowPriceProducts INT, OUT NumberOfHighPriceProducts INT) BEGIN SELECT COUNT(ProductID) INTO NumberOfLowPriceProducts FROM Products WHERE Price < 50; SELECT COUNT(ProductID) INTO NumberOfHighPriceProducts FROM Products WHERE Price >= 50; END //

Delimiter ;

CALL GetProductSummary(@TotalNumberOfLowPriceProducts, @TotalNumberOfHighPriceProducts);

# **Differences between functions and stored procedures**

## **Overview of functions and procedures in MySQL**

In many programming languages, functions are called procedures. In some other programming languages such as SQL there are several key differences between these two terms.

However, it is important to realize that both terms represent the same concept of wrapping or encapsulating code together in the body of the function or the procedure. This function or procedure is then called to perform a specific operation by invoking the identifier name.

The main purpose of creating stored procedures and functions is to create reusable code that can be invoked and executed in an efficient way. So, instead of typing the same code repeatedly, you can save your blocks of code in a stored procedure or a function. You can then call these blocks when you need to use your code.

This makes your code more consistent, better organized, reusable and easier to maintain.

For example, the following query can be used to return the names of all clients in the Clients table from the Lucky Shrub database:

SELECT \* FROM Clients;

You can wrap this statement in a stored procedure as follows:

DELIMITER //

CREATE PROCEDURE GetAllClients()

BEGIN

SELECT \* FROM Clients;

END //

DELIMITER;

You can invoke this procedure simply by calling the identifier name. This is the output result of calling the CALL GetAllClients(); procedure.

The following query is an example of a function that returns the average cost of all orders in the Orders table from the Lucky Shrub database.

SELECT AVG(Cost) FROM Orders;

You can wrap this statement in a stored function as follows:

DELIMITER //

CREATE FUNCTION GetCostAverage() RETURNS DECIMAL(5,2) DETERMINISTIC

BEGIN

RETURN (SELECT AVG(Cost) FROM Orders);

END //

DELIMITER;

CALL GetCostAverage();

## **Key differences between functions and procedures**

A function in MySQL is also called a stored function. A procedure is called a stored procedure. Basic stored procedures and functions typically represent operations that contain empty parameters, or simple input parameters, and a single SQL statement.

More complex procedures and functions require the use of additional features like complex parameters, variables, changing delimiters and the use of BEGIN-END keywords. They also often require the use of multiple SQL statements in the body of the procedure.

In this context, key differences between functions and procedures are as follows:

* A function returns a single value, whereas a procedure may return a single value, multiple values or no value.
* Typically, functions encapsulate common formulas or generic business rules that are reusable among SQL statements and stored procedures. Procedures, on the other hand, are used mainly to process, manipulate and modify data in the database.
* Functions only accept input parameters, while stored procedures can accept IN, OUT and INOUT parameters.
* Functions can be invoked from anywhere, including SELECT statements and stored procedures. Stored procedures are invoked using the CALL statement only.
* A stored function is created using the CREATE FUNCTION statement. A stored procedure is created using the CREATE PROCEDURE statement.
* To build a function, you should specify if it is a DETERMINISTIC function or not. This means that you need to decide if the function always returns the same result for the same input parameters. If you don't use DETERMINISTIC, then MySQL uses the NOT DETERMINISTIC option by default.
* To build functions you must specify the data type of the return value in the RETURNS statement. This can be any valid MySQL data type. However, there’s no need to do this with stored procedures.

The following table provides a summary of the key differences between stored procedures and stored functions.

|  | Functions | Procedures |
| --- | --- | --- |
| 1 | Created using CREATE FUNCTION command | Created using the CREATE PROCEDURE command |
| 2 | Invoked using the SELECT statement | Invoked using the CALL statement |
| 3 | Must return a single value | Outputs values via the OUT parameter |
| 4 | Takes IN parameters only | Takes IN, OUT and INOUT parameters |
| 5 | Typically encapsulates common formulas or generic business rules | Typically used to process, manipulate and modify data in the database |
| 6 | Must specify the data type of the return value | User must specify the OUT parameter type |

## **Conclusion**

Functions and procedures are used to encapsulate code that can be executed to implement repetitive tasks such as equations, formulas or business rules.

In addition, functions and stored procedures make your code more consistent, reusable and easier to use and maintain.

However, you should be aware of the key differences between functions and procedures in MySQL so that you know when to use one over the other.

**What are MySQL triggers?**

A set of actions available in the form of a stored program invoked when an event occurs.

Add the IF EXISTS clause to a drop statement to prevent errors.

Trigger statements must be enclosed within a BEGIN END block.

Each trigger's name must be unique within the database.

Change the delimiter from ; to // as appropriate.

Use a NEW modifier to target a column value after an operation.

CREATE TRIGGER trigger\_name

TRIGGER TYPE

ON table\_name FOR EACH ROW

BEGIN

statement\_one

statement\_two

END;

DELIMITER //

CREATE TRIGGER OrderQtyCheck BEFORE INSERT ON Orders FOR EACH ROW BEGIN IF

NEW.Quantity < 0 THEN SET NEW.Quantity = 0; END IF; END //

DELIMITER ;

**DROP MySQL trigger**

**DROP TRIGGER schema\_name.trigger\_name**

**DROP TRIGGER IF EXISTS luckyShrub.**OrderQtyCheck;

**Working with MySQL Scheduled Events**

**Scheduled events**

A task that takes place at a specific time according to a schedule.

**Names**

All name events have unique names

**Statements**

Each event contains one or more SQL statements.

**Occurrence**

Events can occur once or multiple times.

**One-time events**

scheduled events that occur just once

**for example**,

inserting data into a table one hour from now.

**Recurring events**

Scheduled events that occur on a regular basis, like generating a weekly report from a database.

**Syntax of Scheduled events →**

CREATEEVENT **[**IF NOT EXISTS] event\_name

ON SCHEDULE schedule\_logic

DO

Event\_body

**Syntax of** **One-time events →**

CREATEEVENT **[**IF NOT EXISTS] event\_name

ON SCHEDULE AT CURRENT\_TIMESTAMP [+ INTERVAL]

DO

Event\_body

CREATE EVENT GenerateRevenueReport

ON SCHEDULE AT CURRENT\_TIMESTAMP + INTERVAL 12 HOUR

DO

BEGIN

INSERT INTO ReportData(OrderID, ClientID, ProductID, Quantity, Cost)

SELECT \* FROM Orders WHERE Cost BETWEEN 500 AND 1000;

END

**Syntax of Recurring events →**

CREATEEVENT [IF NOT EXISTS] event\_name

ON SCHEDULE

EVERY [+ INTERVAL]

DO

Event\_body

**Syntax of Recurring Scheduled events →**

CREATEEVENT [IF NOT EXISTS] event\_name

ON SCHEDULE

EVERY interval STARTS timestamp [+ INTERVAL] ENDS timestamp [+ INTERVAL]

DO

Event\_body

DELIMITER //

CREATE EVENT GenerateRevenueReport ON SCHEDULE AT CURRENT\_TIMESTAMP + INTERVAL 12 HOUR DO BEGIN INSERT INTO ReportData(OrderID, ClientID, Quantity,Cost) SELECT \* FROM Orders WHERE Cost BETWEEN 500 AND 1000; END //

CREATEEVENT DailyRestock

ON SCHEDULE

EVERY 1 DAY

DO

BEGIN

IF Products.NumberOfItems < 50 THEN

UPDATE Products SET NumberOfItems = 50;

END IF;

END//

**DROP MySQL trigger**

DROP EVENT [IF EXISTS] event\_name;

DROP EVENT IF EXISTS DailyRestock;

# Overview of MySQL triggers

MySQL triggers and their uses

A MySQL trigger is a stored program that is associated with a table. A trigger is activated before or after a defined SQL operation is performed on a table.

Triggers can be activated on SQL operations like INSERT, UPDATE and DELETE. MySQL supports Row-Level triggers that are invoked before or after a row is inserted, updated or deleted in a table.

You can specify a set of actions to be performed on a table when the specified operation occurs. This is done within the body of a trigger.

Triggers can be used in MySQL for the following purposes:

Enforcing business rules

Ensure data integrity (as a replacement to constraints)

Insert records to other tables for audit trail purposes

Query data for reporting purposes

Replicate data to different tables to achieve data consistency

Creating and removing MYSQL triggers

The syntax to create a trigger in MySQL is as follows:

CREATE TRIGGER trigger\_name

{BEFORE | AFTER} {INSERT | UPDATE| DELETE}

ON table\_name FOR EACH ROW

trigger\_body;

This syntax has 5 main parts:

**1. Trigger name**: All triggers must have a unique name within a schema or database name attached (optional). Triggers in different schemas can have the same name.

**2. Trigger time**: This specifies when to action the trigger. It can be BEFORE or AFTER the specified action to indicate that the trigger activates before or after each row that is inserted, modified or deleted.

3. **Trigger event**: This is the type of operation that activates the trigger. For example, INSERT, UPDATE and DELETE.

4. **Table name**: There must be a table to which the trigger is associated. It’s not possible to associate a trigger to a temporary table or a view.

5. **Trigger body**: Consists of the statements that execute when the trigger activates. The BEGIN END block should be used to execute multiple statements.

The syntax to remove a MySQL trigger is as follows:

DROP TRIGGER [IF EXISTS] [schema\_name.]trigger\_name

\The schema name must be specified if the trigger is not in the default (current) schema. When a table is dropped, any triggers associated with that table will also be dropped.

**Accessing table columns from within a trigger**

You often must access table column values when writing the code in the trigger body. Accessing a table column within the body of the trigger can be done using the OLD and NEW modifiers. OLD is used to access the value of the column before the operation (For example BEFORE INSERT, UPDATE and DELETE). NEW is used to access the value of the column after the operation.

The OLD and NEW modifiers are available for INSERT, UPDATE and DELETE operations as follows:

| Trigger event | OLD | NEW |
| --- | --- | --- |
| INSERT | No | Yes |
| UPDATE | Yes | Yes |
| DELETE | Yes | No |

Ensure that you use the OLD and NEW modifiers within your trigger correctly. Use the INSERT, UPDATE and DELETE triggers as specified in the table above.

Below are two examples of using the OLD and NEW modifiers within the trigger body.

Using the NEW modifier:

CREATE TRIGGER OrderQtyCheck

BEFORE INSERT ON Orders

FOR EACH ROW

BEGIN

IF NEW.Quantity < 0 THEN

SET NEW.Quantity = 0;

END IF;

END;

In this example an order is inserted. The NEW modifier is used to access the Quantity column’s value after the insert operation.

Using the OLD modifier:

CREATE TRIGGER AfterDeleteOrder

AFTER DELETE

ON Orders FOR EACH ROW

INSERT INTO Audits VALUES(4,'AFTER',CONCAT('Order ',OLD.OrderID, ' was deleted at ', CURRENT\_TIME(), ' on ', CURRENT\_DATE()),'DELETE');

In this example an order is deleted. The OLD modifier is used to access the OrderID column's value before the DELETE operation.

## **Listing all triggers in the MySQL database**

It is also helpful to view the triggers in your MySQL database.

You can use the **SHOW TRIGGERS** statement to view this.

SHOW TRIGGERS

[{FROM | IN} database\_name]

[LIKE 'pattern' | WHERE search\_condition];

This syntax returns all triggers in the specified database.

The database name should be specified after the FROM or IN keyword.

It is possible to find triggers based on a pattern and find triggers that match a condition by adding the LIKE and the WHERE sections to this syntax.

## **Advantages and Disadvantages of triggers**

Advantages of triggers:

* Enables validation of data even before it is inserted or updated.
* Provides an alternative way to run scheduled tasks or to perform tasks automatically.
* Increases the performance of SQL queries because it does not need to compile each time that the query is executed.
* Reduces client-side code for validations. This saves time and effort.
* Easy to maintain.

Disadvantages of triggers:

* It's not possible to perform all kinds of constraint validations using a trigger.
* Not easy to troubleshoot because they work in the database layer and not in client/application layer. Therefore, the errors in triggers may be hidden to the developer.
* May increase the overhead of the database server.

## **Conclusion**

Triggers are a good way of enforcing business rules. Triggers are also a good alternative to constraints. They can be invoked automatically on specified actions on a database table. However, triggers should not be overused as it may increase the overhead of the MySQL database.

>>>>>>>>>>>>>> **week 2>>>>>>>>>>>>>>>>**

**Optimizing database queries**

→ Improving database performance to reduce the execution time of a query.

→ Database optimization means that your SQL statements are more efficient and target the required data more accurately.

**Data retrieval statements**

SQL statements that return data from the database

Data retrievalstatements are SELECT statements

Standard Query

SELECT => Database

STATEMENT <=

Result

**Data change statements**

Statements that alter data within the database.

Index → Table column's

Syntax's should avoid using with predicates in their SQL SELECT statements

**Functions**

Avoid using functions in predicates, particularly on a column that’s indexed.

**Leading wildcards**

Avoid using leading wildcards in predicates. MySQL can’t make use of an index if there’s a leading wildcard in the pattern.

**Method for optimizing databases**

method for optimizing databases involves using the INNER JOIN instead of the OUTER JOIN where possible.

**OUTER JOIN**

An OUTER JOIN retrieves all records from both tables, including rows that don't contain matching values. This takes longer for MySQL to process.

**INNER JOIN**

The INNER JOIN is more efficient because it retrieves only the necessary data or matching records from both tables. This helps to optimize your queries.

Often when creating SQL queries you'll use that distinct clause to eliminate duplicate values or the union clause to combine multiple query results. This can slow down the query because it must perform a sorting operation and eliminate duplicate records. However, if you use UNION ALL instead, then this eliminates the need for a sorting operation and speeds up the execution process. SELECT query slow down the database.

**Advantage of using database optimization**

Improved performance

More efficient SQL statements

Faster data retrieval

→ An optimal method for sorting through data in a MySQL database table is to create a custom column.

→ MySQL can scan a custom column in a table for matching values without relying on functions.

**REVERSE\_STRING QUERY**

**For all users**

ALTER TABLE clients ADD COLUMN ReverseFullName varchar(100);

UPDATE clients SET ReverseFullName = CONCAT(SUBSTRING\_INDEX(fullName, '

', -1), ' ', SUBSTRING\_INDEX(fullName, ' ', 1));

For single user

CREATE INDEX IdxReverseFullName ON clients (ReverseFullName);

SELECT \* FROM clients WHERE ReverseFullName LIKE 'sam%';

# Database optimization in practice with SELECT statements

## **Importance of SELECT query optimization**

Optimizing SELECT queries involves reducing the amount of time it takes for a query to retrieve  data from a database. The time taken to execute a SELECT query depends mostly on the amount of data (number of rows) that it must work with.

SELECT statements, also known as data retrieval statements, play an important role in a database application. They perform all the database search operations. Database search operations are expensive, and they require a fast response time. So, efficient SQL SELECT statements are important for the effective performance of a database application.

To optimize SELECT queries, create indexes on table columns that are frequently queried. Indexes reduce the number of rows that a query needs to read or scan. The MySQL Query Optimizer steps in when SELECT queries are executed and devises an optimal query execution plan.

As a database engineer, you can assist the MySQL query optimizer by applying simple techniques such as:

* Avoiding the use of unnecessary columns in the SELECT clause.
* Avoiding the use of functions in predicates (WHERE clause conditions).
* Avoiding the use of leading wildcards in predicates, particularly with the LIKE operator.
* Using INNER JOIN instead of OUTER JOIN.
* Using DISTINCT and UNION sparingly in SELECT queries.

## **Optimizing SELECT queries with the EXPLAIN statement**

When a SELECT query is issued against a database, the MySQL Query Optimizer devises an optimal plan for query execution. You can see what this plan looks like by prefixing the query with EXPLAIN. The EXPLAIN statement provides information about how the query is executed.

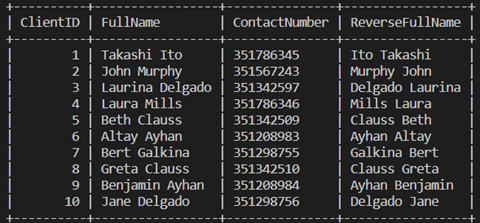
It is important that you understand each piece of information clearly, so that you can utilize it in practice to optimize SELECT queries that have bottlenecks or are designated as suboptimal.

An EXPLAIN statement is prepended to a SELECT query to retrieve information on the plan that MySQL uses to execute the given query, as the following syntax example shows:

EXPLAIN SELECT column\_name FROM table\_name WHERE VALUE

To understand this concept, let’s explore an example from the Lucky Shrub database.

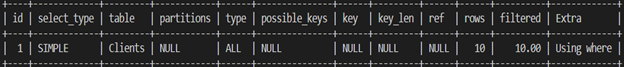
Lucky Shrub’s database has a Clients table that contains 10 rows. Lucky Shrub need to retrieve the contact number of the client Jane Delgado.



They can use the EXPLAIN statement and a SELECT query to generate a plan for the query execution:

EXPLAIN SELECT ContactNumber FROM Clients WHERE FullName='Jane Delgado';

This query shows a plan of how the query will be executed. Out of the 12 columns, this result has 10 important columns that provide useful information about how the query is executed. By analyzing the results, you can take action to improve the query’s performance.



## **Overview of EXPLAIN statement results**

Let’s examine these columns one by one.

Column 01: ID

The ID column is a sequential identifier for each SELECT statement within the query. This piece of information is more meaningful when there are nested queries, or subqueries.

The Lucky Shrub example has a value of 1 as it is a simple example of a single query.

Column 02: SELECT\_TYPE

This column displays the type of SELECT query to be executed.

There are several values that can be assigned to the SELECT query:

| Value | Description |
| --- | --- |
| SIMPLE | Simple SELECT query without any subqueries or UNIONs |
| PRIMARY | The SELECT is in the outermost query in a JOIN |
| DERIVED | The SELECT is part of a subquery within a FROM clause |
| SUBQUERY | The first SELECT in a subquery |
| DEPENDENT SUBQUERY | The SELECT statements is a subquery dependent on an outer query |
| UNCACHEABLE SUBQUERY | Subquery which is not cacheable (there are certain conditions for a query to be cacheable) |
| UNION | The SELECT query is the second or later statement of a UNION |
| DEPENDENT UNION | The second or later SELECT of a UNION is dependent on an outer query |
| UNION RESULT | The SELECT is a result of a UNION |

Lucky Shrub's query has returned a value of SIMPLE because it is a basic query that contains no subqueries or unions.

Column 03: Table

This column displays the name of the table referred to in the SELECT query. Lucky Shrub's SELECT query refers to the Clients table.

Column 04: Partitions

This column displays the partition in which the data resides (the area of physical storage that's scanned). Partitioning allows for the distribution of portions of table data across the file system. If the queries access only a fraction of table data, then there's less records to scan and queries can execute faster. However, partitioning is more meaningful when dealing with large data sets.

Lucky Shrub's Clients table is not partitioned. So, it shows a NULL value in that column.

Column 05: type

Scanning the table means performing a search operation or finding matches specified by the SELECT query.

The following table outlines a list of the most prominent possible values:

| Value | Description |
| --- | --- |
| system | The table has only one row or zero rows. This return value typically indicates that the search was performed on a system table. |
| const | Indicates that the value of the searched column can be treated as a constant (there is one row matching the query) |
| eq\_ref | Indicates that the index is clustered and is being used by the operation (either the index is a PRIMARY KEY or UNIQUE INDEX with all key columns defined as NOT NULL) |
| ref | Indicates that the indexed column was accessed using an equality operator. You can see an example of this in the optimized Lucky Shrub query EXPLAIN statement results, as the indexed column FullName is matched or accessed using the = operator in the WHERE clause. |
| full text | The scan uses the table’s FULLTEXT index. Full-text indexes are created on text-based columns (CHAR , VARCHAR , or TEXT columns) |
| index | The entire index is scanned to find a match for the query |
| all | The entire table is scanned to find matching rows. This is the worst scan type and usually indicates the lack of appropriate indexes on the table. |

Lucky Shrub’s query returns a value of ALL. This indicates that MySQL scans the entire table (each row) to find the matching rows. This suggests that there’s a problem with executing this query. In other words, this is a suboptimal query. This is because there’s no index defined on the table.

Column 06: possible\_keys

This column shows the keys that can be used by MySQL to find rows from the table. However, these keys may or may not be used in practice. If the column value is NULL, then it indicates that no relevant indexes are found.

The Lucky Shrub example has a NULL value, which indicates that there are no keys or indexes in the table which MySQL can use to find or filter rows. This indicates a problem that needs to be addressed.

Column 07: key

Indicates the actual index used by MySQL.

In the Lucky Shrub example, this column returns a NULL value. This means there’s no index in the table that can be used by the optimizer. This is a problem that needs to be addressed, because an index is required for optimization.

Column 08: key\_len

This column indicates the length of the index the Query Optimizer chooses to use. For example, a key\_len value of 4 means it requires memory to store four characters.

Lucky Shrub’s database returns a NULL value again because there is no index that can be used by this query.

Column 09: ref

This column shows which table columns have been compared to the index to perform the search. A value of const means a constant, while a value of func means that the value that was used was derived from a function.

In Lucky Shrub's database, no columns or constants are compared against an index, because the table doesn’t have one. Therefore, the value is NULL. This once again highlights the problem of not using an index in a table.

Column 10: rows

Lists the number of records that were examined to produce the output.

Lucky Shrub's results indicate that there were 10 records examined by the query. This means that every row in the table was examined. This is an inefficient use of the database's resources.

If there were hundreds, or even thousands of records in a table, then an inefficient query would have to examine each one. This would take much longer, and use more resources, than an efficient query which only needs to examine the required records.

Column 11: filtered

This column indicates an approximate percentage of the number of table rows that have been filtered by a specified condition. The higher the percentage is, the better the performance of the query.

In the Lucky Shrub query, 10% of the rows in the Clients table are filtered by the WHERE clause condition.

Column 12: Extra

Contains additional information regarding the query execution plan. Pay attention if there are values like  Using temporary or Using filesort in this column as they indicate a problematic query. Let's take a quick look at what these values mean.

* Using temporary: This value Indicates that MySQL needs to create a temporary table to hold the result of this query. This typically happens if the query contains GROUP BY and ORDER BY clauses that list columns differently (In other words, when the columns listed in the GROUP BY and ORDER BY clauses are different).
* Using filesort: This value indicates that MySQL must perform an extra pass to determine how to retrieve the rows in sorted order. The sort is performed by going through all rows according to the type (mentioned earlier) and storing the sort key and pointer to the row for all rows that match the WHERE clause. The keys then are sorted, and the rows are retrieved in sorted order.

Lucky Shrubs example has a value of Using where. This indicates that MySQL first reads each row within the table before it filters them. This is not an efficient way for a query to make use of the WHERE clause. If an index was defined on the Clients table, then the value here would be different.

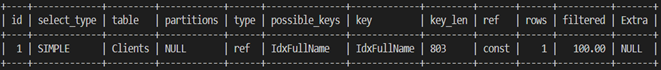
## **Planning a more efficient SELECT query**

The use of an index on the required column, or columns, results in more efficient SELECT queries that processes records faster using less resources.

For example, Lucky Shrub can add an index on the FullName column using the following code:

CREATE INDEX IdxFullName ON Clients(FullName);

Then, when the EXPLAIN statement is executed, it can return more detailed results:



Attention needs to be paid to the following columns:

* type
* possible\_keys,
* keys,
* ref,
* rows
* and Extra.

With the index in place on the FullName column, type indicates that MySQL now scans all the matching rows of the indexed column FullName. In this case there’s only 1 row that matches the indexed column’s value - Jane Delgado.

Possible\_keys indicates that there’s a possible index that can be used to find the required rows. This is the IdxFullName index. The index speeds up the query’s performance.

key indicates that the index named IdxFullName is used to locate the required value. By using the name of the index, the query can locate the required information faster.

Ref shows that a constant value (Jane Delgado) is compared to the index shown in the key column. In other words, the statement has a value to reference against the index.

The rows column indicates that only one row was examined to produce the output. This shows that MySQL was able to execute the statement quickly and efficiently without the need to examine all records in the table.

## **Conclusion**

Running SELECT queries with an EXPLAIN statement is a helpful method for optimizing SELECT queries in practice. Attention needs to be paid to the type, possible\_keys, keys, ref and rows columns in the EXPLAIN statement results that indicate specific issues in a SELECT query that require optimization.

**Indexes in MySQL**

Index is a data structure that provides pointer to a stored data.

**Two types of indexes**

Primary index → Primary or clustered index.

A primary index is an index that is stored within the table itself. It's generated automatically once you create a table that contains a primary or unique key, the index enforces the order of rows in the table within the table itself.

Secondary index → Secondary or non-clustered index.

A secondary index is created using the My SQL create index statement. The syntax begins with create index, then write the name of the index. A commonly used approach is to write the name of the column you want to create the index on, prefaced by idx for index.

Database optimization in practice with indexes

This reading provides an overview of how MySQL utilizes indexes on query operations and how indexes can be used to optimize database performance in a practical scenario.

Introduction to Indexes and MySQL Indexes

As you may know already, an index is a data structure that is used to help search for data in a table for a given SELECT query without scanning every row. Indexes help retrieve data faster and speed up search queries. This means that indexes play a key role in database optimization.

In MySQL, indexes are created on table columns. However, not all the columns in a table should be indexed. You must decide which columns to index in a table based on which ones are used to regularly filter SELECT queries.

An indexed column’s data is stored in a separate location known as the index. The data is ordered according to type (for example numeric, alphabetical or date order). Each piece of data in the index points to the original table row. By creating a relationship, or reference, between the index and table, it becomes easier and faster to search for data.

It is important to note that indexes may hinder the performance of data change queries (INSERT, UPDATE or DELETE queries) while improving the performance of SELECT queries. The reason for this is that when inserting, updating or deleting a row in a table, data also needs to be inserted, updated or deleted in the index as well.

Query operations and indexes in MySQL

It is important to be aware of what query operations MySQL uses indexes on. This can help you to identify which table columns you need to create indexes on when creating SQL queries.

MySQL uses indexes when performing the following operations in queries:

Filtering rows using the WHERE clause: If there are multiple indexes used on the same column, MySQL chooses the index that works with the minimum number of rows.

Performing JOIN operations: To get the best performance from an index, the columns used in a JOIN should both have the same data type.

MIN and MAX functions: An index can be used with these functions to find the minimum and maximum value of a specific column.

Sorting and grouping operations: An index can be used with these operations.

When you know which operations MySQL uses indexes on, you can then inspect your queries and create indexes on columns that are used to perform those operations. This helps the MySQL query optimizer optimize your queries to a greater extent.

Using indexes to optimize query performance

If a queried table contains very high data volumes (such as thousands or even millions of records), then performing search operations and obtaining results gives rise to slow response times. This leads to a suboptimal and inefficient database.

Let’s take a few moments to see how Lucky Shrub can use indexes to optimize query performance.

Lucky Shrub expands their business rapidly across many cities within the state of Arizona and gains tens of thousands of new clients. As part of this expansion, they have also introduced a loyalty scheme. So, the database’s clients table now stores data on each client’s city, state, date of birth (DOB) and loyalty points in addition to their id, full name and contact number.

Lucky Shrub need to query the Clients table to retrieve the following data:

The names of clients who are from Tucson, Arizona.

The names of clients who are from Arizona but outside of Tucson.

A list of clients with the maximum loyalty points.

A list of clients whose date of birth falls on a given date.

To retrieve this data, they can write the following queries:

1. All clients from Tucson, Arizona

SELECT ClientID, FullName FROM Clients WHERE City = Tucson;

2. The names of clients who are from Arizona but outside of Tucson.

SELECT ClientID, FullName FROM Clients WHERE State = Arizona AND City <> Tucson;

3. A list of clients with the maximum loyalty points.

SELECT ClientID, MAX(LoyaltyPoints) FROM Clients;

4. A list of clients whose date of birth falls on a given date.

SELECT ClientID, FullName FROM Clients WHERE DOB = 1990-01-01;

With tens of thousands of clients in the table, these queries will take a long time to return the results. This is an inefficient approach and doesn't adhere to database optimization guidelines.

The solution is for Lucky Shrub to create indexes on the columns within the Client’s table that MySQL uses to carry out these searches. These are the City, State, LoyaltyPoints, and DOB columns. By creating indexes on these columns, Lucky Shrub significantly reduce the time it takes to execute each query. This leads to a more efficient and more optimal database.

## **Conclusion**

Indexes help speed up searches on a database by allowing the MySQL query optimizer to execute queries according to the best possible plan. The real performance impact of indexes is best observed in a database that has a large amount of data as opposed to a database that has a small amount of data. It’s important that indexes are created on the correct table columns. Don’t create indexes unnecessarily as they could have a negative impact on data change statements like INSERT, UPDATE and DELETE.

# Additional resources

The following resources offer additional reading material on the topic of database optimization in MySQL:

* [Dev.mysql guide to optimization in MySQL](https://dev.mysql.com/doc/refman/8.0/en/optimization.html)
* [O’Reilly guide to query performance optimization](https://www.oreilly.com/library/view/high-performance-mysql/9780596101718/ch04.html)

**MySQL Transaction**

**START TRANSACTION**

A standard SQL statement to start the transaction process.

**Syntax**

**→** BEGIN

statement\_1

statement\_2

statement\_3

COMMIT

**COMMIT**

Commit transaction changes to the database.

START TRANSACTION

statement\_1

statement\_2

statement\_3

COMMIT

**ROLLBACK**

Rollback transaction and cancel changes.

START TRANSACTION

statement\_1

statement\_2

statement\_3

ROLLBACK

**MySQL Common table expression (CTE)**

A CTE is a method of optimizing complex database queries by compiling them into simple blocks of code. These blocks can then be used to rewrite the query by calling the CTE when required. This simplifies the query and makes it much easier to read and maintain. A common table expression can be created for one or multiple queries. It all depends on the requirements of your database.

**Syntax of CTE expression with multiple queries**

WITH common\_table\_name AS (query) SELECT \* FROM common\_table\_name;

WITH

cte1 AS (query1),

cte2 AS (query2),

cte3 AS (query3),

SELECT \* FROM cte1;

Or

WITH

cte1 AS (query1),

cte2 AS (query2),

cte3 AS (query3),

SELECT \* FROM cte1

UNION

SELECT \* FROM cte2

UNION

SELECT \* FROM cte3

examples:

SELECT CONCAT(AVG(Cost), "(2020)") AS "Average Sale" FROM Orders WHERE YEAR (Date) = 2020

UNION SELECT CONCAT(AVG(Cost), "(2021)") FROM Orders WHERE YEAR(Date) = 2021 UNION SELECT CONCAT(AVG(Cost), "(2022)") FROM Orders WHERE YEAR(Date) = 2022;

You can click Enter to execute the above statements and return the average sale for each year. Although they work as intended, these queries are quite complex and difficult to manage, but you can use a common table expression or CTE to improve their readability. Start by using the WITH clause.

WITH Average\_Sales\_2020 AS (SELECT CONCAT(AVG(Cost), " (2020)") AS "Average Sale" FROM Orders WHERE YEAR(Date) = 2020), Average\_Sales\_2021 AS (SELECT CONCAT(AVG(Cost), " (2021)") FROM Orders WHERE YEAR(Date) = 2021), Average\_Sales\_2022 AS (SELECT CONCAT(AVG(Cost), " (2022)") FROM Orders WHERE YEAR (Date) = 2022) SELECT \* FROM Average\_Sales\_2020 UNION SELECT \* FROM Average\_Sales\_2021 UNION SELECT \* FROM Average\_Sales\_2022;

**MySQL Prepared Statement**

A reusable MySQL statements that acts as a template and holds unspecified values as parameters.

A prepared statement is already compiled and parsed by MySQL before it's executed.

**Syntax**

PREPARE GetOrderStatement FROM 'SELECT ClientID, ProductID, Quantity, Cost FROM Orders WHERE OrderID = ?';

SET @order\_id = 10;

EXECUTE GetOrderStatement USING @order\_id;

# **MySQL JSON**

**Syntax**

CREATE TABLE Activity(ActivityID INT PRIMARY KEY, Properties JSON);

INSERT INTO Activity(ActivityId, Properties) VALUES (1, '{"ClientID": "Cl1","ProductID": "P1", "Order": "True"}'), (2, '{"ClientID": "Cl2", "ProductID": "P4", "Order": "False"}'), (3, '{"ClientID": "Cl5", "ProductID": "P5", "Order": "True"}');

SELECT ActivityID, Properties->'$.ClientID', Properties→'$.ProductID',

Properties->'$.Order' FROM Activity;

**Further optimization techniques**

## **Overview**

As you should be aware by now, database optimization techniques are used to improve the performance of your database. In the previous lesson, you learned how to optimize MySQL SELECT queries using a range of different techniques.

However, there are other optimization techniques that can be used to support your database optimization such as transactions, common table expressions and prepared statements.

You may have encountered these already in this lesson. So, let's take a few moments to explore the basics of how these techniques are used to optimize database queries.

## **MySQL Transaction**

As a database engineer, you should use MySQL transaction any time you have a critical activity to complete. If any error occurs during the execution of the query, then you can use this technique to restore your database to the original state.

This is a particularly useful technique to consider when writing a series of related queries that must all execute as intended to achieve an expected outcome.

For example, Lucky Shrub have written a series of queries to assist with new orders. When an order is received, the database first checks that there are enough items available in stock to fulfill the order. Once the order has been processed, the database then updates the number of items remaining in stock.

There are a lot of steps in this process. And some of the changes, or required operations, may fail to occur in the database. If so, then Lucky Shrub can use MySQL transaction to cancel the changes and rollback the current transaction if any statements fail to execute as required.

You can use the MySQL BEGIN or BEGIN WORK aliases to start a MySQL transaction. Although the START TRANSACTION command is the standard way to start a transaction process in MySQL.

Then type the required SELECT statements. If all queries execute as expected, then the COMMIT statement is used to commit the current transaction and make the transaction changes permanently.

However, one or more of these statements may fail to execute as required. If so, then you can use the ROLLBACK statement to roll back the current transaction, cancel the changes made to the database and restore it to its original state.

This process is illustrated in the following abstract syntax:

START TRANSACTION;

SQL statements

ROLLBACK;

## **Common Table Expressions**

Database engineers often need to write complex SQL queries, which are usually difficult to read and maintain.

In this situation, you can use the Common Table Expression (CTE) technique to deconstruct the complex queries into simple blocks of code. With CTE, you can rewrite the query in a way that simplifies it and makes it easier to read and maintain.

The basic abstract syntax of the Common Table Expression is as follows:

WITH CTE\_Name AS (query code)

SELECT \*

FROM CTE\_Name;

In this syntax, you use the WITH clause to start the common table expression, followed by the name of the Common Table Expression. This can be a custom name.

Next, use the AS keyword to associate the query with the related expression name. Finally, you use a SELECT statement to query the common table expression name.

If you require multiple queries, then you just need to associate each query with its related expression name and separate it by a comma as in the following example:

WITH

cte1 AS (query1),

cte2 AS (query2)

SELECT cte1 UNION cte2;

This abstract syntax specifies two common table expressions separated by a comma. Each expression is associated with a query using the AS keyword.

## **Prepared statements**

A query must be parsed by MySQL before it can be executed, which takes time and causes delays.

However, creating prepared statements in MySQL reduces the parsing time for queries. A prepared statement only needs to be parsed by MySQL once, regardless of the number of times it’s used and executed.

A prepared statement in MySQL is a template that you create with an SQL statement. A prepared statement starts with a PREPARE command followed by the statement name. This can be a custom name. It can also include certain unspecified values, used as parameters and labelled with a question mark ?

For example, in the following syntax, the two question marks ? represent two parameters that can be used to insert two different values into the Products table:

**Syntax**

PREPARE InsertProductData ‘INSERT INTO Products VALUES(?, ?)’;

Eg:

PREPARE GetOrderStatement FROM 'SELECT ClientID, ProductID, Quantity, Cost FROM Orders WHERE OrderID = ?';

SET @order\_id = 10;

EXECUTE GetOrderStatement USING @order\_id;

Prepared statements also minimize the bandwidth usage when communicating data with the MySQL server. You only need to pass the parameter’s values to the prepared statement, not the whole SQL statement.

Creating prepared statements also secures your database from SQL injections, a common hacking technique that can be used to destroy your database. A SQL injection attempts to pass malicious code to your database through SQL statements. However, this can be avoided if you use prepared statements when building your SQL queries.

## **Conclusion**

MySQL transaction statements, common table expressions and prepared statements are all examples of effective optimization techniques that can be used to improve the performance of your MySQL database.

A transaction in MySQL contains one or more SQL queries that can be committed permanently to the database, or you can roll back to the original state if any of the executed queries fails.

The Common Table Expression can be used to minimize the complexity of your queries and make them easier to read and maintain.

The prepared statement supports database security and allows you to reuse the same SQL statements repeatedly and more efficiently.

# Additional resources

The following resources offer additional reading material on various types of MySQL database optimization techniques:

* [MySQLtutorial overview of MySQL transactions](https://www.mysqltutorial.org/mysql-transaction.aspx)
* [Dev.MySQL tutorial on Common Table Expressions in MySQL](https://dev.mysql.com/doc/refman/8.0/en/with.html)
* [Dev.MySQL overview of the JSON data type](https://dev.mysql.com/doc/refman/8.0/en/json.html)
* [MySQLtutorial guide to PREPARED statements](https://www.mysqltutorial.org/mysql-prepared-statement.aspx)

# **Overview of database analytics**

With data analytics, you can take the data collected during data analysis and convert it into useful information that can inform future business decisions.

**Data analytics**

Converting and processing data is useful and meaningful information.

Data analytics uses data collected during data analysis to make predictions about future behavior or performance.

**Descriptive data analysis**

Presenting data in a descriptive format.

**Exploratory data analysis**

Establishing a relationship between variables.

**Inferential data analysis**

Focusing on a small sample of data to make inferences and draw general conclusions about a larger data population.

**Predictive data analysis**

Using existing or legacy data to identify paradigms and patterns that can be used to make predictions about future performance.

**Causal data analysis**

Exploring cause and effect relationships between different variables.

**Using MySQL for data analysis**

Offers database engineers the tools to perform data analysis on their data.

**Relational database model**

MySQL databases are built using a relational database model. Relational models structured data sets in related tables. And these related tables make it easy to access, retrieve, and analyze related information.

**Advantages of a relational database model**

Orders table <=> Product ID key <=> Products table

MySQL is also a free open source database management system, so there's no cost or intellectual property to consider when managing a database with MySQL. And because of its capacity and accessibility, MySQL is a very widely used database management system. Large numbers of businesses, governments, and other organizations make use of MySQL to collect, store, and process data. This makes it easier for these organizations to communicate data and improve their data analytics. It's free and open source, holds large amounts of data in a relational system, and is widely used across various organizations.

**Communication between MySQL databases**

Lucky Shrub <=> Product table <=> Suppliers

**Disadvantages of relational database model**

MySQL also comes with limitations. MySQL 's capacity to perform data analysis is much more limited than other more advanced data analytics tools. With these other tools, database engineers can perform much more complex data analysis backed by powerful artificial intelligence. MySQL also lacks data visualization feature. Other database analytics tools offer database engineers visualization features like bar charts, graphs, and maps. These tools are a much more effective way of communicating information than just presenting data in the form of tables. With these visualization tools. There are a wide range of benefits to using MySQL as a database management system.

# How MySQL is used with advanced data analytics tools

## **Data analytics overview**

Data analytics is a key concern for all kinds of organizations across a range of sectors including education, government, health, tourism and business.

Data analytics is concerned with collecting, processing, and investigating data to gain deep insights into an organization’s performance. For example, a business can use data analytics to gain insights into their sales patterns, customer behavior or profit margins.

Data analytics provides useful and meaningful information that an organization’s decision makers can leverage to deal with problems and challenges at an early stage. It also helps them to predict the future of their organization.

## **Benefits of using MySQL for data analytics**

There are several benefits to using MySQL for data analytics.

MySQL databases are built using relational models. These models store data sets in related tables, which makes it much easier to access, process and analyze existing data.

It’s also an open-source database management system that can store huge amounts of data. In database analytics, the more data your database stores and processes, the better the results of your analysis.

MySQL is widely used by various types of businesses, social network platforms, e-commerce applications, and government websites. This means that it’s much easier for these organizations to integrate imported data from multiple sources, which will facilitate a rich environment of data to perform data analytics.

## **MySQL and advanced data analytics tools**

MySQL’s ability to perform data analytics is adequate. However, it’s capacity for advanced analytical processing is much more limited than comparable advanced analytical tools

MySQL is more focused on data transaction processing rather than analytics, where records of data are managed in tables within rows and columns through SQL operations such as the INSERT, UPDATE and DELETE operations.

Business owners and managers need to perform sophisticated data analytics operations. So, they often integrate MySQL with other advanced analytics platforms to optimize the benefits of data analytics.

These other platforms provide user-friendly interfaces for dealing with data. They also present analyzed data in a variety of visual formats such as charts, graphs, maps and interactive dashboards.

Generating data or information in these visual formats is a useful method of identifying trends, gaps, patterns and models. It’s much easier to work with than raw data in tables because visual information can be processed by the human brain much faster.

MySQL can be integrated with many advanced analytical tools including the two most popular platforms Tableau and Power BI. Tableau is the most popular analytical platform in the world. Power BI is an important interactive data visualization tool developed by Microsoft. Its primary focus is on business intelligence.

Both Tableau and Power BI are advanced visualization tools that are used to access, connect and analyze data from various sources. They can then convert analyzed data into interactive dashboards.

These platforms also contain built-in connectors that facilitate easy integration, visualization and analysis of MySQL data by supporting the Select, Extract, Transform and Load (SETL) mechanism.

These tools can be used to easily identify MySQL data to be analyzed. They extract the selected data, transform it into a suitable format and load the data onto the platform to perform different types of data analytics.

## **Conclusion**

MySQL is used by many kinds of organizations to maintain huge amounts of data in relational databases. It’s easy to access, integrate, process and analyze. However, MySQL's ability to perform data analytics is very limited compared to other advanced data analytics tools. These other tools can perform much more effective analysis using artificial intelligence and data visualization.

# Additional resources

The following resources provide additional reading material on performing data analysis in MySQL:

* [Data analysis in MySQL – Operators, Joins and more in relational databases](https://towardsdatascience.com/data-analysis-in-mysql-operators-joins-and-more-in-relational-databases-26c0a968e61e)
* [Basics of MySQL knowledge required for Data Science](https://medium.com/analytics-vidhya/basics-mysql-knowledge-required-for-data-science-98530539076a)

**Data analysis in MySQL using SQL queries**

Collecting data using SQL queries

**Join →** Join two tables together.

**Subquery →** create a query within a query.

→ is a query that can be nested inside another query such as SELECT, INSERT, UPDATE and DELETE statements.

**Views →** create virtual tables.

**Functions →** Code that performs an operation and returns a result.

**Operators →** Keyword used to filter specific data from database tables.

SQL queries → Data analysis → Data analytics

**Lucky Shrub and data analysis**

**Syntax**

**ANY**

SELECT ProductID FROM Products\_price WHERE ProductID = ANY (SELECT ProductID FROM Orders WHERE Quantity >= 20);

INNER JOIN

SELECT Orders.ClientID, Orders.Cost, Clients.FullName, Clients.ContactNumber FROM Orders INNER JOIN Clients ON(Orders.ClientID = Clients.ClientID) AND (Date BETWEEN '2020-09-10' AND '2020-09-20');

VIEW

CREATE VIEW TopProducts AS SELECT Products.ProductName, Orders.Quantity, Orders.Cost FROM Orders INNER JOIN Products ON Orders.ProductID = Products.ProductID ORDER BY Orders.Cost DESC LIMIT 5;

Execute

SELECT \* FROM TopProducts;

**Emulating the Full Outer JOIN in MySQL**

**Full OUTER JOIN**

A join used to return all records from two tables, including those that don’t have a match.

**UNION ALL**

Returns duplicate records.

**UNION**

The UNION operator returns unique records.

**Full OUTER JOIN with UNION ALL operators**

SELECT columns FROM table1 LEFT JOIN table2 ON table1.CommonColumn = table2.CommonColumn UNION ALL SELECT columns FROM table1 RIGHT JOIN table2 ON table1.CommonColumn = table CommonColumn.CommonColumn

Now that you've scripted the LEFT JOIN you need to create the RIGHT JOIN. But you also need to combine these JOINS using a method that returns all duplicate records. It's at this point in your syntax that you can add the UNION ALL operator. Once you've added the operator, create the RIGHT JOIN. As you should already know, the syntax is almost the same as the LEFT JOIN statement. The key difference is that you must use a RIGHT JOIN clause. When executed, the UNION ALL statement returns all duplicate records should any exist. But what if you only want unique records to retrieve unique records only, you can use the UNION operator. The syntax is largely the same. You need to script a RIGHT JOIN and a LEFT JOIN statement as before. But this time, place the UNION operator in-between the two statements instead of UNION ALL. Then when executed, the statement only returns unique records.

**Extracting data from multiple tables with JOINS**

**Lucky Shrub and the Full OUTER JOIN**

SELECT Clients.ClientID, Clients.FullName, Clients.ContactNumber, Orders.OrderID, Orders.Cost, Orders.Date FROM Clients LEFT JOIN Orders ON Clients.ClientID = Orders.ClientID UNION SELECT Clients.ClientID, Clients.FullName, Clients.ContactNumber, Orders.OrderID, Orders.Cost, Orders.Date FROM Clients RIGHT JOIN Orders ON Clients.ClientID = Orders.ClientID;

# Additional resources

The following resources provide additional reading material on database analytics:

* [Types of data analysis](_blank)
* [Pros and cons of using MySQL for analytical reporting](https://www.sisense.com/blog/pros-cons-using-mysql-analytical-reporting/)

# Next steps

Congratulations! You've taken another step toward improving your knowledge, skills, and qualifications. By completing this course, you've acquired and practiced the core skills needed to use advanced topics in MySQL.

In the next course, you'll develop a working knowledge of advanced MySQL topics, where you will gain further insight into the world of database clients.

The following resources are additional reading materials that you can review for further insight on Advanced MySQL topics:

* [Overview from Dev.MySQL of optimization in MySQL](https://dev.mysql.com/doc/refman/8.0/en/optimization.html)
* [A guide from Dev.MySQL on procedures and functions in MySQL](https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html)
* [Article from TowardsDataScience providing an overview of data analysis in MySQL](https://towardsdatascience.com/data-analysis-in-mysql-operators-joins-and-more-in-relational-databases-26c0a968e61e)