Section 1. Getting started with MySQL

This section helps you get started with MySQL. We will start installing MySQL, downloading a sample database, and loading data into the MySQL server for practicing.

Installing MySQL database server – show you step by step how to install MySQL database server on your computer.

Downloading MySQL sample database – introduce you to a MySQL sample database named classicmodels.

Loading the sample database into your own MySQL database server – walk you through steps of how to load the classicmodels sample database into your MySQL database server for practicing.

Section 2. Querying data

This section helps you learn how to query data from the MySQL database server. We will start with a simple SELECT statement that allows you to query data from a single table.

SELECT – show you how to use simple SELECT statement to query the data from a single table.

SELECT DISTINCT – learn how to use the DISTINCT operator in the SELECT statement to eliminate duplicate rows in a result set.

Section 3. Sorting data

ORDER BY – show you how to sort the result set using ORDER BY clause. The custom sort order with the FIELD function will be also covered.

Section 4. Filtering data

WHERE – learn how to use the WHERE clause to filter rows based on specified conditions.

AND – introduce you to the AND operator to combine Boolean expressions to form a complex condition for filtering data.

OR– introduce you to the OR operator and show you how to combine the OR operator with the AND operator to filter data.

IN – show you how to use the IN operator in the WHERE clause to determine if a value matches any value in a list or a subquery.

BETWEEN – show you how to query data based on a range using BETWEEN operator.

LIKE – provide you with technique to query data based on a specific pattern.

LIMIT – use LIMIT to constrain the number of rows returned by SELECT statement

IS NULL – test whether a value is NULL or not by using IS NULL operator.

Section 5. Joining tables

Table & Column Aliases – introduce you to table and column aliases.

Joins – give you an overview of joins supported in MySQL including inner join, left join, and right join.

INNER JOIN – query rows from a table that has matching rows in another table.

LEFT JOIN – return all rows from the left table and matching rows from the right table or null if no matching rows found in the right table.

RIGHT JOIN – return all rows from the right table and matching rows from the left table or null if no matching rows found in the left table.

CROSS JOIN – make a Cartesian product of rows from multiple tables.

Self-join – join a table to itself using table alias and connect rows within the same table using inner join and left join.

Section 6. Grouping data

GROUP BY – show you how to group rows into groups based on columns or expressions.

HAVING – filter the groups by a specific condition.

ROLLUP – generate multiple grouping sets considering a hierarchy between columns specified in the GROUP BY clause.

Section 7. Subqueries

Subquery – show you how to nest a query (inner query) within another query (outer query) and use the result of the inner query for the outer query.

Derived table – introduce you to the derived table concept and show you how to use it to simplify complex queries.

EXISTS – test for the existence of rows.

Section 8. Common Table Expressions

Common Table Expression or CTE – explain you the common table expression concept and show you how to use CTE for querying data from tables.

Recursive CTE – use the recursive CTE to traverse the hierarchical data.

Section 9. Set operators

UNION and UNION ALL – combine two or more result sets of multiple queries into a single result set.

INTERSECT – show you a couple of ways to simulate the INTERSECT operator.

MINUS – explain to you the SQL MINUS operator and show you how to simulate it.

Section 10. Modifying data in MySQL

In this section, you will learn how to insert, update, and delete data from tables using various MySQL statements.

INSERT – use various forms of the INSERT statement to insert data into a table.

INSERT INTO SELECT – insert data into a table from the result set of a query.

INSERT IGNORE – explain you the INSERT IGNORE statement that inserts rows into a table and ignore rows that cause errors.

UPDATE – learn how to use UPDATE statement and its options to update data in database tables.

UPDATE JOIN – show you how to perform cross table update using UPDATE JOIN statement with INNER JOIN and LEFT JOIN.

DELETE – show you how to use the DELETE statement to delete rows from one or more tables.

ON DELETE CASCADE – learn how to use ON DELETE CASCADE referential action for a foreign key to delete data from a child table automatically when you delete data from a parent table.

DELETE JOIN – show you how to delete data from multiple tables.

REPLACE – learn how to insert or update data depends on whether data exists in the table or not.

Prepared Statement – show you how to use the prepared statement to execute a query.

Section 11. MySQL transaction

Transaction – learn about MySQL transactions, and how to use COMMIT and ROLLBACK to manage transactions in MySQL.

Table locking – learn how to use MySQL locking for cooperating table access between sessions.

Section 12. Managing MySQL databases and tables

This section shows you how to manage the most important database objects in MySQL including database and tables.

Selecting a MySQL database – show you how to use the USE statement to select a MySQL database via the MySQL program and MySQL Workbench.

Managing databases – learn various statements to manage MySQL databases including creating a new database, removing an existing database, selecting a database, and listing all databases.

CREATE DATABASE – show you how to create a new database in MySQL Server.

DROP DATABASE – learn how to delete an existing database.

MySQL storage engines– it is essential to understand the features of each storage engine so that you can use them effectively to maximize the performance of your databases.

CREATE TABLE – show you how to create new tables in a database using CREATE TABLE statement.

MySQL sequence – show you how to use a sequence to generate unique numbers automatically for the primary key column of a table.

ALTER TABLE – learn how to use the ALTER TABLE statement to change existing table's structure.

Renaming table – show you how to rename a table using RENAME TABLE statement.

Removing a column from a table – show you how to use the ALTER TABLE DROP COLUMN statement to remove one or more columns from a table.

Adding a new column to a table – show you how to add one or more columns to an existing table using ALTER TABLE ADD COLUMN statement.

DROP TABLE – show you how to remove existing tables using DROP TABLE statement.

Temporary tables – discuss MySQL temporary table and show you how to manage temporary tables.

TRUNCATE TABLE – show you how to use the TRUNCATE TABLE statement to delete all data in a table fast.

Section 13. MySQL data types

MySQL data types – show you various data types in MySQL so that you can apply them effectively in designing database tables.

INT – show you how to use integer data type.

DECIMAL – show you how to use DECIMAL data type to store exact values in decimal format.

BIT – introduce you BIT data type and how to store bit values in MySQL.

BOOLEAN – explain to you how MySQL handles Boolean values by using TINYINT (1) internally.

CHAR – guide to CHAR data type for storing the fixed-length string.

VARCHAR – give you the essential guide to VARCHAR data type.

TEXT – show you how to store text data using TEXT data type.

DATE – introduce you to the DATE data type and show you some date functions to handle the date data effectively.

TIME – walk you through the features of TIME data type and show you how to use some useful temporal functions to handle time data.

DATETIME – introduce you to the DATETIME data type and some useful functions to manipulate DATETIME values.

TIMESTAMP – introduce you to TIMESTAMP and its features called automatic initialization and automatic update that allows you to define auto-initialized and auto-updated columns for a table.

JSON – show you how to use JSON data type to store JSON documents.

ENUM – learn how to use ENUM data type correctly to store enumeration values.

Section 14. MySQL constraints

NOT NULL constraint – introduce you to the NOT NULL constraint and show you how to declare a NOT NULL column or add a NOT NULL constraint to an existing column.

Primary key constraint – guide you how to use primary key constraint to create the primary key for a table.

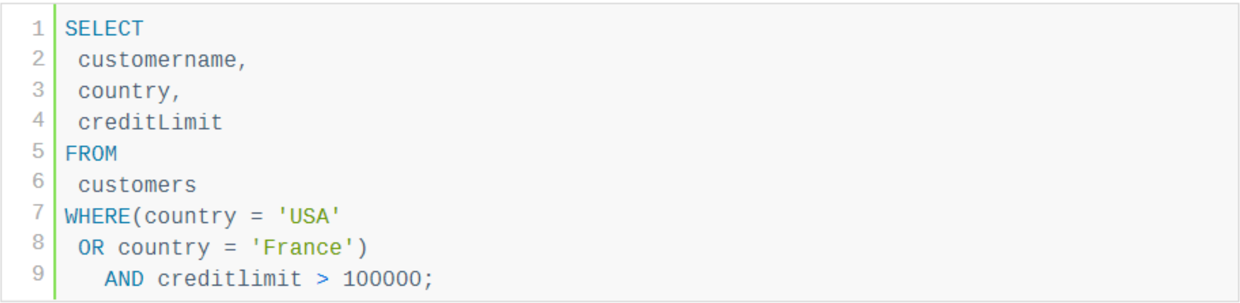
Foreign key constraint – introduce you to the foreign key and show you step by step how to create and drop foreign keys.

UNIQUE constraint – show you how to use UNIQUE constraint to enforce the uniqueness of values in a column or a group of columns in a table.

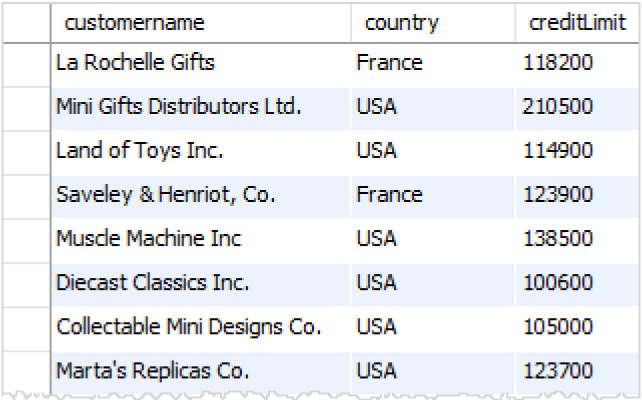
CHECK constraint emulation – walk you through various ways to emulate the CHECK constraint in MySQL.

As you can see the result, the query returns customers who locate in the USA or France.

The following statement returns the customers who locate in the USA or France and have the credit limit greater than 10,000.

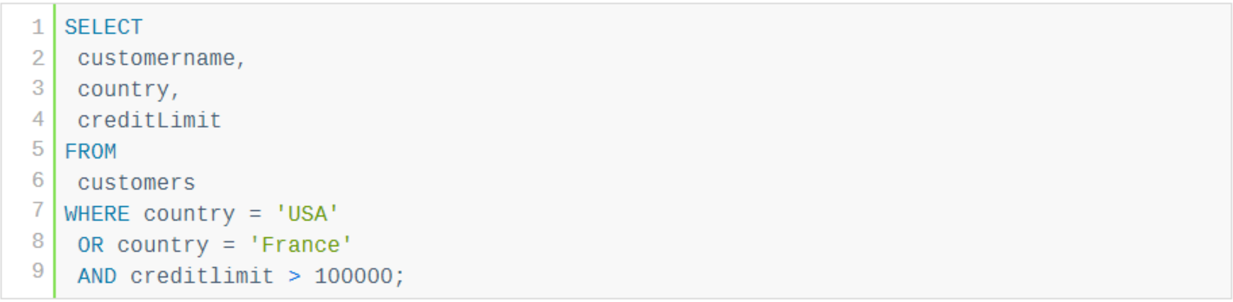


```
SELECT
 customername,
 country,
 creditLimit
FROM
 customers
WHERE(country = 'USA'
 OR country = 'France')
   AND creditlimit > 100000;
```

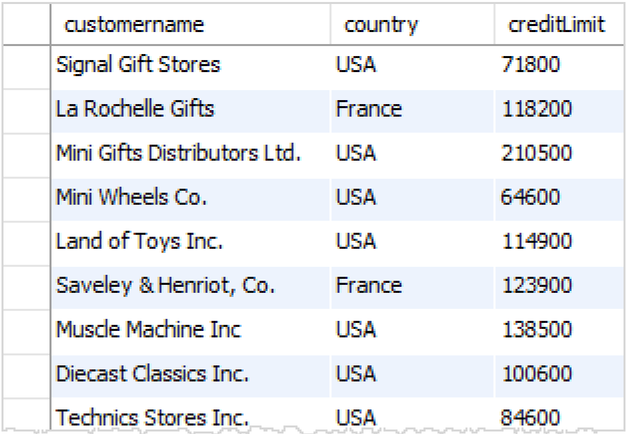


customername	country	creditLimit
La Rochelle Gifts	France	118200
Mini Gifts Distributors Ltd.	USA	210500
Land of Toys Inc.	USA	114900
Saveley & Henriot, Co.	France	123900
Muscle Machine Inc	USA	138500
Diecast Classics Inc.	USA	100600
Collectable Mini Designs Co.	USA	105000
Marta's Replicas Co.	USA	123700

Notice that if you do not use the parentheses, the query will return the customers who locate in the USA or the customers who locate in France with the credit limit greater than 10,000.



```
SELECT
 customername,
 country,
 creditLimit
FROM
 customers
WHERE country = 'USA'
 OR country = 'France'
 AND creditlimit > 100000;
```



customername	country	creditLimit
Signal Gift Stores	USA	71800
La Rochelle Gifts	France	118200
Mini Gifts Distributors Ltd.	USA	210500
Mini Wheels Co.	USA	64600
Land of Toys Inc.	USA	114900
Saveley & Henriot, Co.	France	123900
Muscle Machine Inc	USA	138500
Diecast Classics Inc.	USA	100600
Technics Stores Inc.	USA	84600

MySQL IN

The IN operator allows you to determine if a specified value matches any value in a set of values or returned by a subquery.

The following illustrates the syntax of the IN operator:

```
SELECT
    column1,column2,...
FROM
    table_name
WHERE
 (expr|column_1) IN ('value1','value2',...);
```

Let’s examine the query in more detail:

- Use a column or an expression (expr) with the IN operator in the WHERE clause.
- Separate the values in the list by commas (,).

The IN operator returns 1 if the value of the column_1 or the result of the expr expression is equal to any value in the list, otherwise, it returns 0.

When the values in the list are all constants, MySQL performs the following steps:

- First, evaluate the values based on the type of the column_1 or result of the expr expression.
- Second, sort the values.

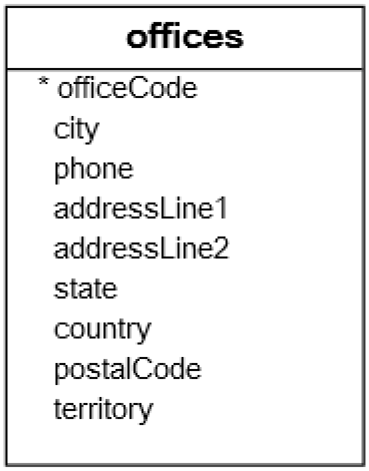
- Third, search for the value using the binary search algorithm. Therefore, a query that uses the IN operator with a list of constants performs very fast.

Note that if the expr or any value in the list is NULL, the IN operator returns NULL.

You can combine the IN operator with the NOT operator to determine if a value does not match any value in a list or a subquery. And you can also use the the IN operator in the WHERE clause of other statements such as UPDATE, and DELETE.

MySQL IN operator examples

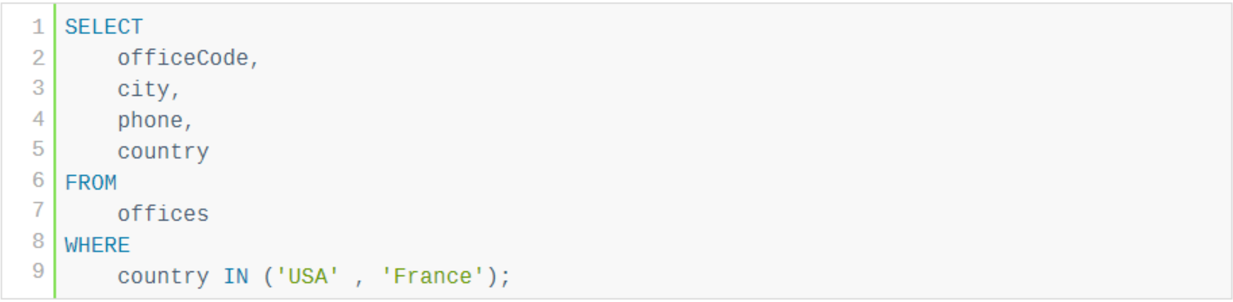
Let’s practice with some examples of using the IN operator. See the following offices table from the sample database:



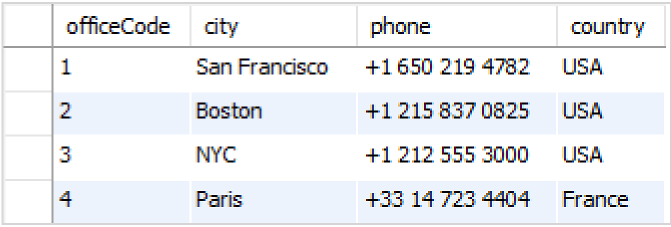
offices

* officeCode
city
phone
addressLine1
addressLine2
state
country
postalCode
territory

If you want to find the offices that locate in the U.S. and France, you can use the IN operator as the following query:

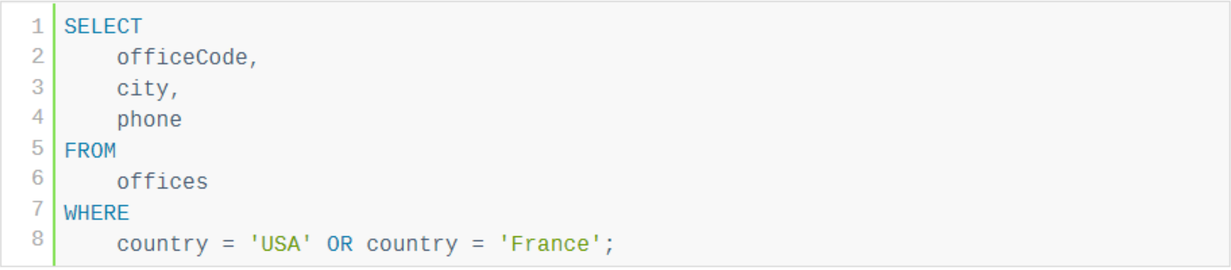


```
SELECT
    officeCode,
    city,
    phone,
    country
FROM
    offices
WHERE
    country IN ('USA' , 'France');
```



officeCode	city	phone	country
1	San Francisco	+1 650 219 4782	USA
2	Boston	+1 215 837 0825	USA
3	NYC	+1 212 555 3000	USA
4	Paris	+33 14 723 4404	France

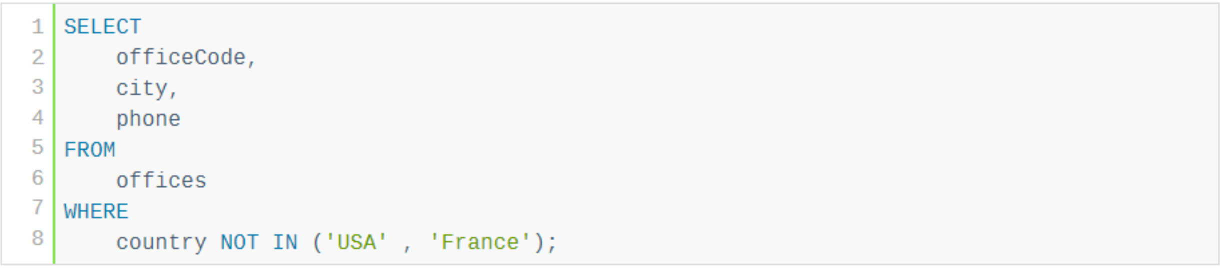
You can achieve the same result with the OR operator as the following query:



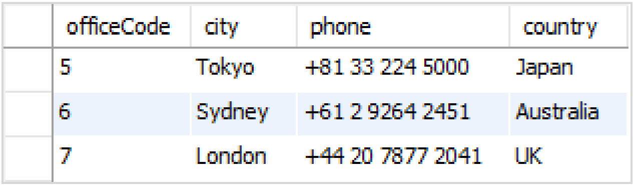
```
SELECT
    officeCode,
    city,
    phone
FROM
    offices
WHERE
    country = 'USA' OR country = 'France';
```

In case the list has many values, you need to construct a very long statement with multiple OR operators. Hence, the IN operator allows you to shorten the query and make it more readable.

To get offices that do not locate in USA and France, you use NOT IN in the WHERE clause as follows:



```
SELECT
    officeCode,
    city,
    phone
FROM
    offices
WHERE
    country NOT IN ('USA' , 'France');
```

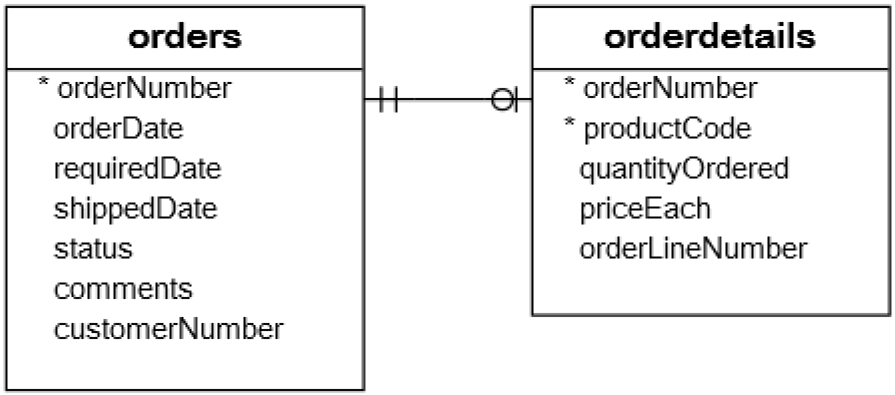


officeCode	city	phone	country
5	Tokyo	+81 33 224 5000	Japan
6	Sydney	+61 2 9264 2451	Australia
7	London	+44 20 7877 2041	UK

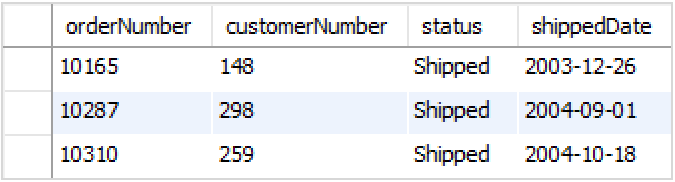
Using MySQL IN with a subquery

The IN operator is often used with a subquery. Instead of providing a list of literal values, the subquery gets a list of values from one or more tables and uses them as the input values of the IN operator.

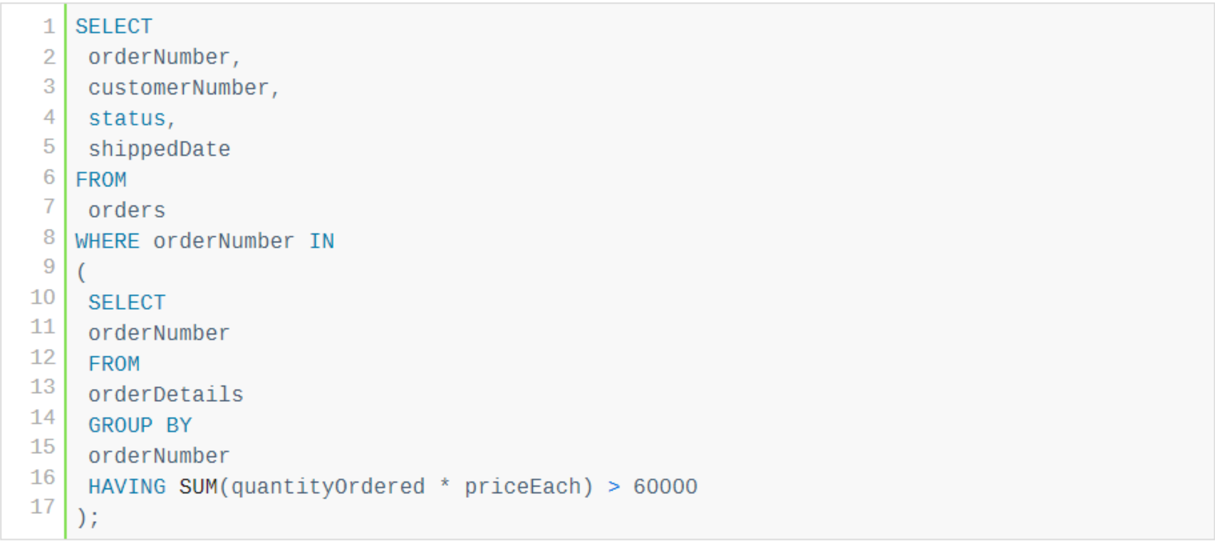
Let’s take a look at the orders and orderDetails tables from the sample database:



For example, if you want to find the orders whose total values are greater than 60,000, you use the IN operator as shown in the following query:



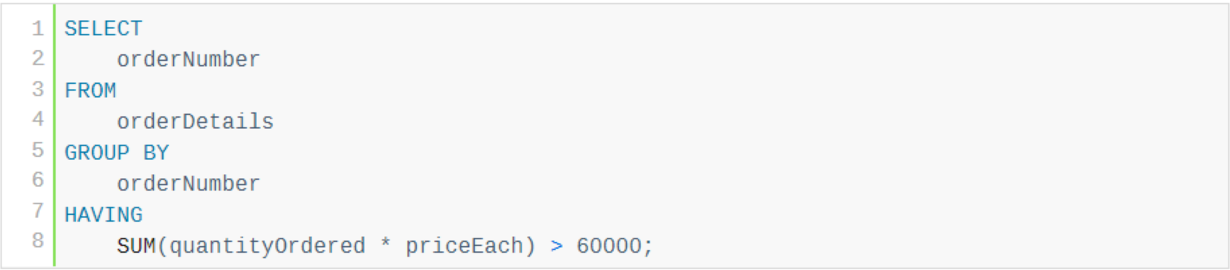
orderNumber	customerNumber	status	shippedDate
10165	148	Shipped	2003-12-26
10287	298	Shipped	2004-09-01
10310	259	Shipped	2004-10-18



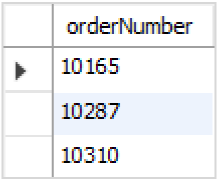
```
SELECT
 orderNumber,
 customerNumber,
 status,
 shippedDate
FROM
 orders
WHERE orderNumber IN
(
 SELECT
 orderNumber
 FROM
 orderDetails
 GROUP BY
 orderNumber
 HAVING SUM(quantityOrdered * priceEach) > 60000
);
```

The whole query above can be broken down into two separate queries.

First, the subquery returns a list of order numbers whose values are greater than 60,000 using the GROUP BY and HAVING clauses:

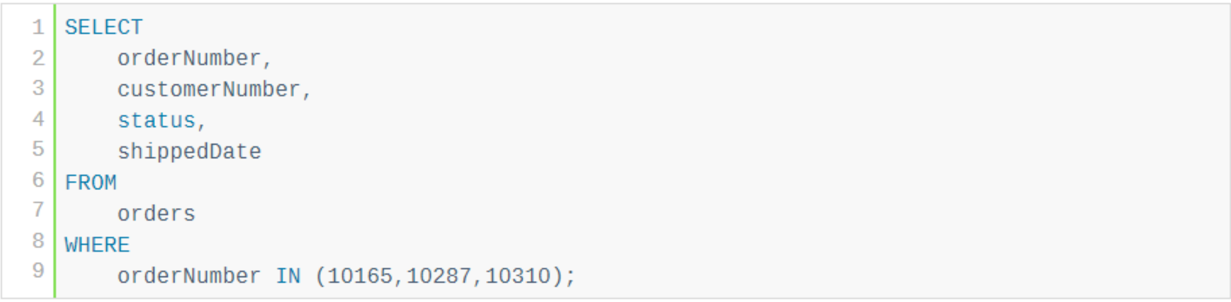


```
SELECT
    orderNumber
FROM
    orderDetails
GROUP BY
    orderNumber
HAVING
    SUM(quantityOrdered * priceEach) > 60000;
```



	orderNumber
▶	10165
	10287
	10310

Second, the outer query uses the IN operator in the WHERE clause to get data from the orders table:

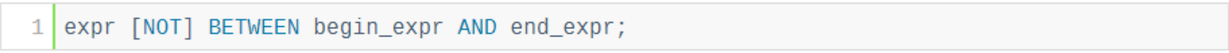


```
SELECT
    orderNumber,
    customerNumber,
    status,
    shippedDate
FROM
    orders
WHERE
    orderNumber IN (10165,10287,10310);
```

MySQL BETWEEN

The BETWEEN operator is a logical operator that allows you to specify whether a value is within a range or not. The BETWEEN operator is often used in the WHERE clause of the SELECT, UPDATE, and DELETE statements.

The following illustrates the syntax of the BETWEEN operator:



```
expr [NOT] BETWEEN begin_expr AND end_expr;
```

The expr is the expression to test in the range defined by begin_expr and end_expr. All three expressions: expr, begin_expr, and end_expr must have the same data type.

The BETWEEN operator returns **true** if the value of the expr is greater than or equal to (>=) the value of begin_expr and less than or equal to (<=) the value of the end_expr, otherwise it returns zero.

The NOT BETWEEN returns **true** if the value of expr is less than (<) the value of the begin expr or greater than the value of the value of end_expr, otherwise it returns 0.

If any expression is **NULL**, the BETWEEN operator returns NULL.

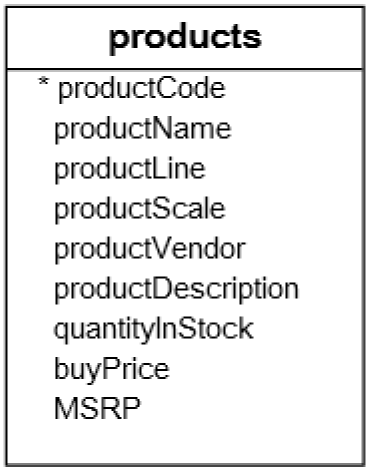
In case you want to specify an exclusive range, you can use the greater than (>) and less than (<) operators.

MySQL BETWEEN operator examples

Let’s practice with some examples of using the BETWEEN operator.

1) Using MySQL BETWEEN with number examples

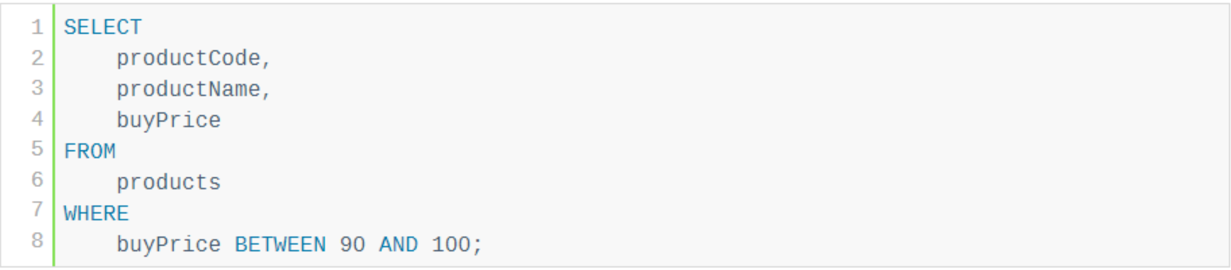
See the following products table in the sample database:



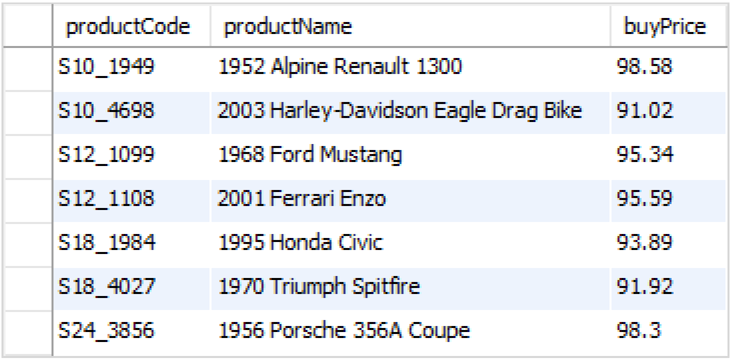
products

* productCode
productName
productLine
productScale
productVendor
productDescription
quantityInStock
buyPrice
MSRP

The following example uses the BETWEEN operator to find products whose buy prices between 90 and 100:

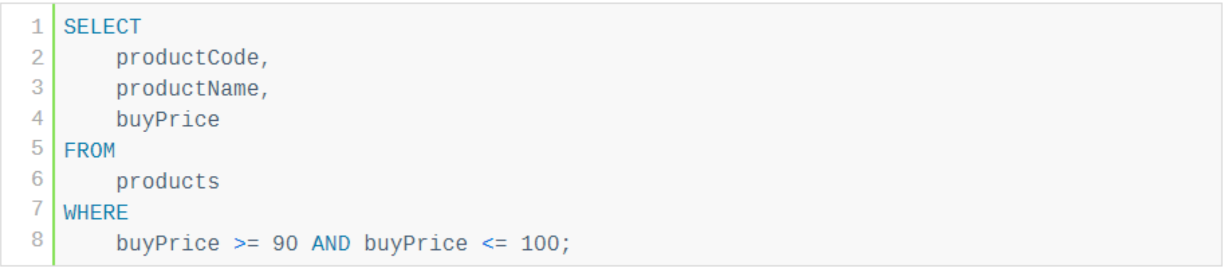


```
SELECT
    productCode,
    productName,
    buyPrice
FROM
    products
WHERE
    buyPrice BETWEEN 90 AND 100;
```



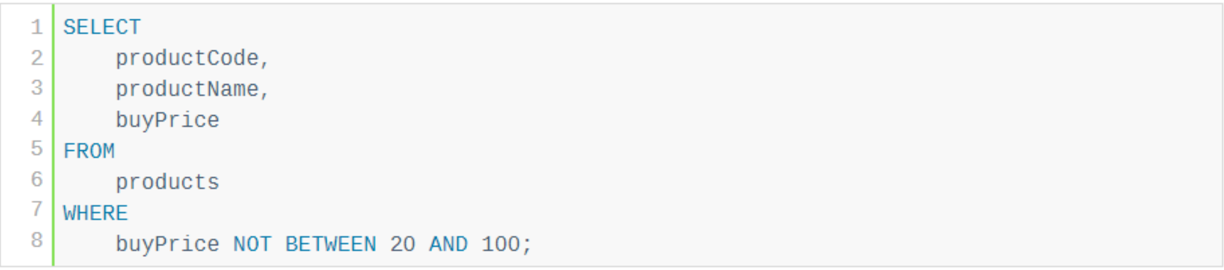
productCode	productName	buyPrice
S10_1949	1952 Alpine Renault 1300	98.58
S10_4698	2003 Harley-Davidson Eagle Drag Bike	91.02
S12_1099	1968 Ford Mustang	95.34
S12_1108	2001 Ferrari Enzo	95.59
S18_1984	1995 Honda Civic	93.89
S18_4027	1970 Triumph Spitfire	91.92
S24_3856	1956 Porsche 356A Coupe	98.3

This query uses the greater than or equal (>=) and less than or equal (<=) operators instead of the BETWEEN operator to get the same result:

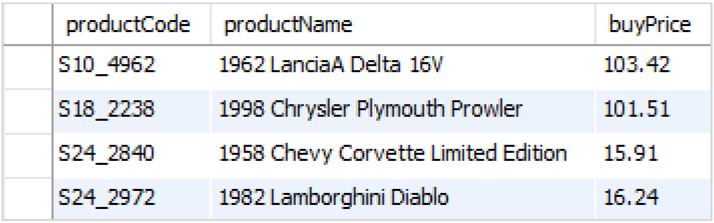


```
SELECT
    productCode,
    productName,
    buyPrice
FROM
    products
WHERE
    buyPrice >= 90 AND buyPrice <= 100;
```

To find the product whose buy price is not between $20 and $100, you combine the BETWEEN operator with the NOT operator as follows:

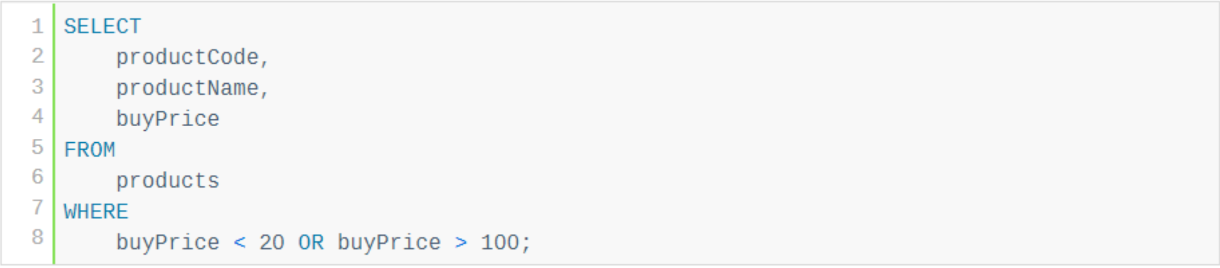


```
SELECT
    productCode,
    productName,
    buyPrice
FROM
    products
WHERE
    buyPrice NOT BETWEEN 20 AND 100;
```



	productCode	productName	buyPrice
	S10_4962	1962 LanciaA Delta 16V	103.42
	S18_2238	1998 Chrysler Plymouth Prowler	101.51
	S24_2840	1958 Chevy Corvette Limited Edition	15.91
	S24_2972	1982 Lamborghini Diablo	16.24

You can rewrite the query above using the less than (>), greater than (>), and logical operators (AND) as the following query:

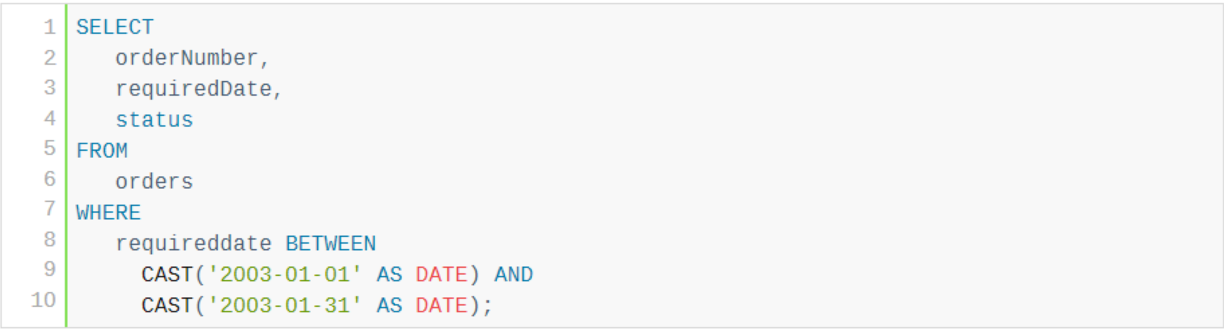


```
SELECT
    productCode,
    productName,
    buyPrice
FROM
    products
WHERE
    buyPrice < 20 OR buyPrice > 100;
```

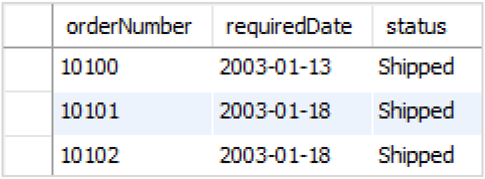
2) Using MySQL BETWEEN with dates example

When you use the BETWEEN operator with date values, to get the best result, you should use the type cast to explicitly convert the type of column or expression to the DATE type.

The following example returns the orders which have the required dates between 01/01/2003 to 01/31/2003:



```
SELECT
   orderNumber,
   requiredDate,
   status
FROM
   orders
WHERE
   requireddate BETWEEN
     CAST('2003-01-01' AS DATE) AND
     CAST('2003-01-31' AS DATE);
```

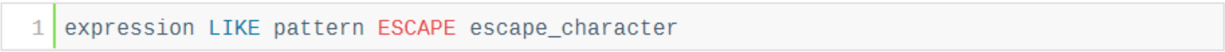


	orderNumber	requiredDate	status
	10100	2003-01-13	Shipped
	10101	2003-01-18	Shipped
	10102	2003-01-18	Shipped

Because the data type of the required date column is DATE so we used the CAST operator to convert the literal strings ‘2003-01-01 ‘and ‘2003-12-31 ‘to the DATE values.

MySQL LIKE

The LIKE operator is a logical operator that tests whether a string contains a specified pattern or not. Here is the syntax of the LIKE operator:



```
expression LIKE pattern ESCAPE escape_character
```

The LIKE operator is used in the WHERE clause of the SELECT, DELETE,
and UPDATE statements to filter data based on patterns.

MySQL provides two wildcard characters for constructing patterns: percentage % and underscore _.

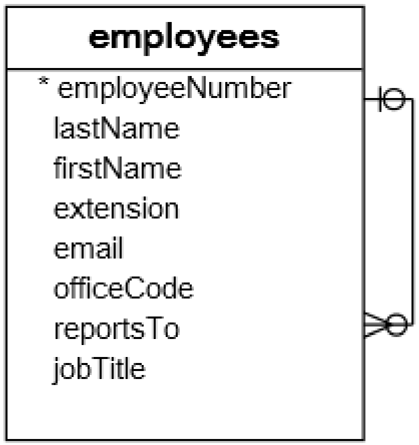
- The percentage (%) wildcard matches any string of zero or more characters.
- The underscore (_) wildcard matches any single character.

For example, s% matches any string starts with the character s such as sun and six.

The se_ matches any string starts with se and is followed by any character such as see and sea.

MySQL LIKE operator examples

Let's practice with some examples of using the LIKE operator. We will use the following employees table from the sample database for the demonstration:

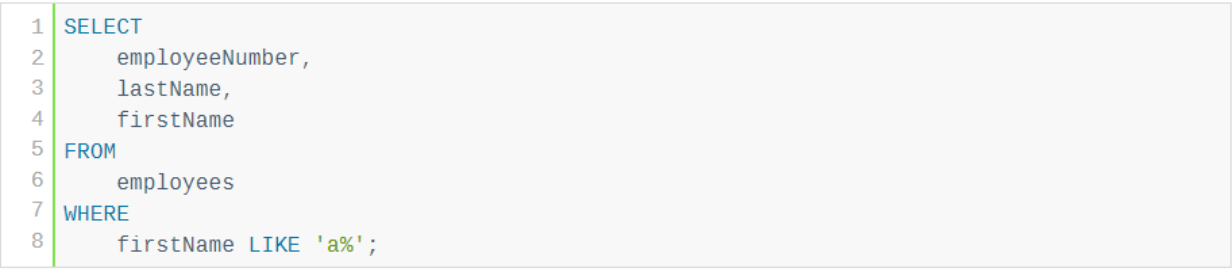


employees

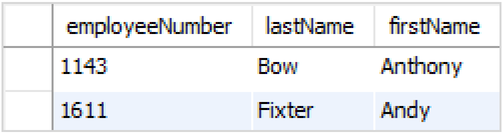
* employeeNumber
lastName
firstName
extension
email
officeCode
reportsTo
jobTitle

A) Using MySQL LIKE with the percentage (%) wildcard examples

This example uses the LIKE operator to find employees whose first names start with a:



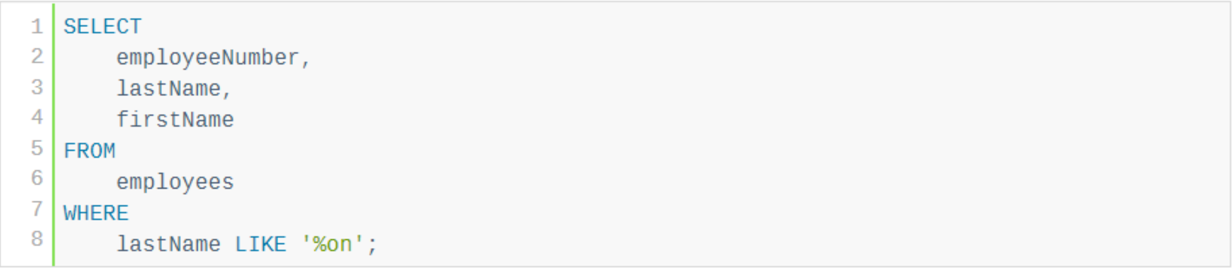
```
SELECT
    employeeNumber,
    lastName,
    firstName
FROM
    employees
WHERE
    firstName LIKE 'a%';
```



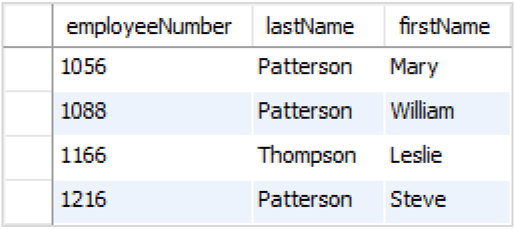
employeeNumber	lastName	firstName
1143	Bow	Anthony
1611	Fixter	Andy

In this example, MySQL scans the whole employees table to find employees whose first names start with the character **a** and are followed by any number of characters.

This example uses the LIKE operator to find employees whose last names end with on e.g., Patterson, Thompson:



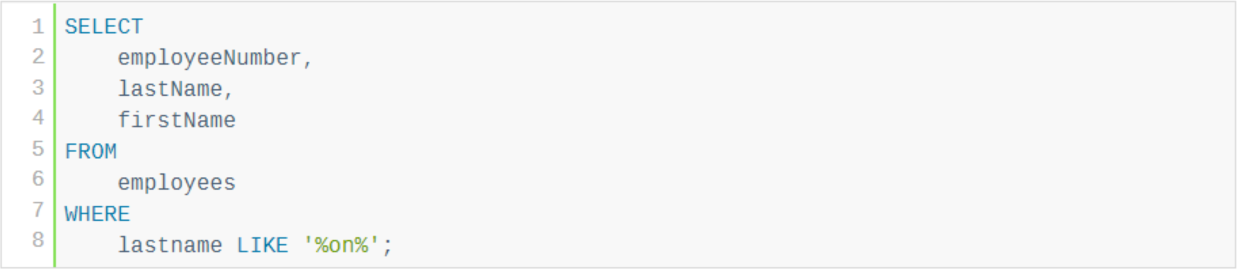
```
SELECT
    employeeNumber,
    lastName,
    firstName
FROM
    employees
WHERE
    lastName LIKE '%on';
```



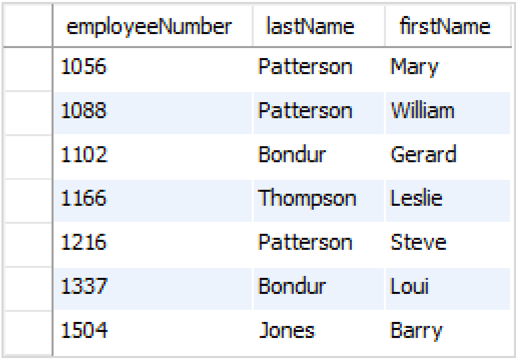
employeeNumber	lastName	firstName
1056	Patterson	Mary
1088	Patterson	William
1166	Thompson	Leslie
1216	Patterson	Steve

If you know the searched string is embedded inside in the middle of a string, you can use the percentage (%) wildcard at the beginning and the end of the pattern.

For example, to find all employees whose last names contain on, you use the following query with the pattern %on%



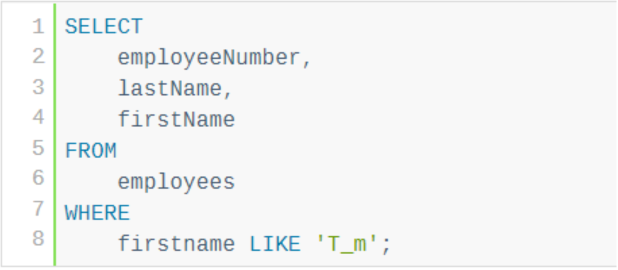
```
SELECT
    employeeNumber,
    lastName,
    firstName
FROM
    employees
WHERE
    lastname LIKE '%on%';
```



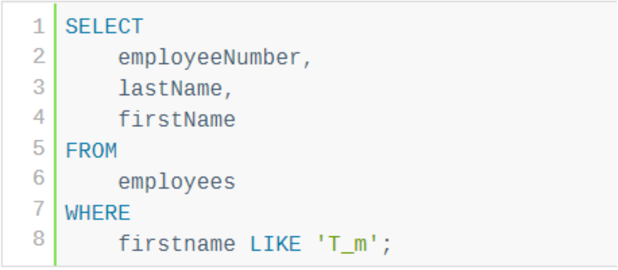
employeeNumber	lastName	firstName
1056	Patterson	Mary
1088	Patterson	William
1102	Bondur	Gerard
1166	Thompson	Leslie
1216	Patterson	Steve
1337	Bondur	Loui
1504	Jones	Barry

B) Using MySQL LIKE with underscore (_) wildcard examples

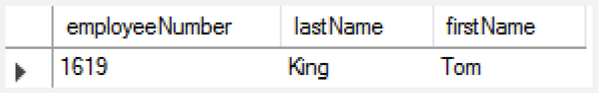
To find employees whose first names start with T, end with m, and contain any single character between e.g., Tom, Tim, you use the underscore (_) wildcard to construct the pattern as follows:



```
SELECT
    employeeNumber,
    lastName,
    firstName
FROM
    employees
WHERE
    firstname LIKE 'T_m';
```



```
SELECT
    employeeNumber,
    lastName,
    firstName
FROM
    employees
WHERE
    firstname LIKE 'T_m';
```

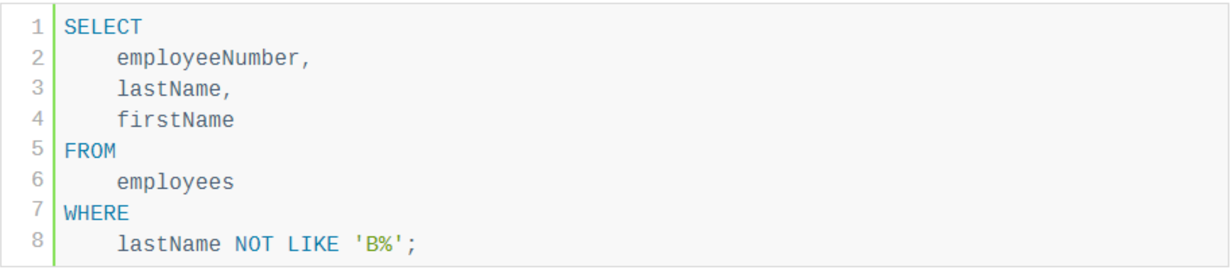


employeeNumber	lastName	firstName
1619	King	Tom

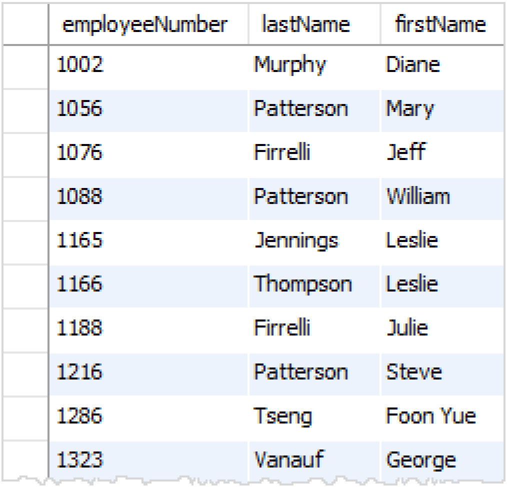
C) Using MySQL LIKE operator with the NOT operator example

The MySQL allows you to combine the NOT operator with the LIKE operator to find a string that does not match a specific pattern.

Suppose you want to search for employees whose last names don’t start with the character B, you can use the NOT LIKE with a pattern as shown in the following query:



```
SELECT
    employeeNumber,
    lastName,
    firstName
FROM
    employees
WHERE
    lastName NOT LIKE 'B%';
```



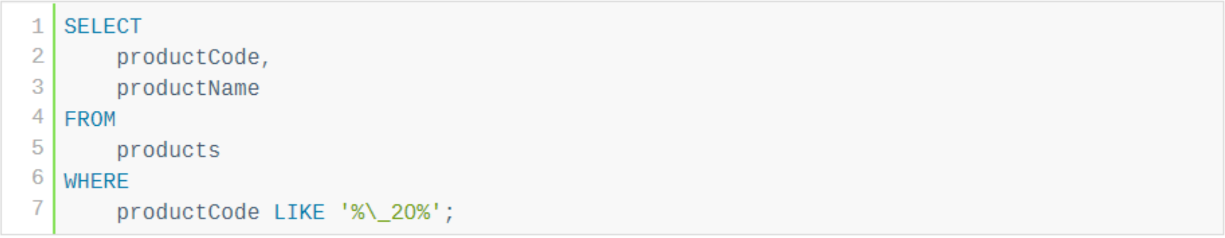
employeeNumber	lastName	firstName
1002	Murphy	Diane
1056	Patterson	Mary
1076	Firrelli	Jeff
1088	Patterson	William
1165	Jennings	Leslie
1166	Thompson	Leslie
1188	Firrelli	Julie
1216	Patterson	Steve
1286	Tseng	Foon Yue
1323	Vanauf	George

> Note that the pattern is not case sensitive, therefore, the b% or B% pattern returns the same result.

MySQL LIKE operator with ESCAPE clause

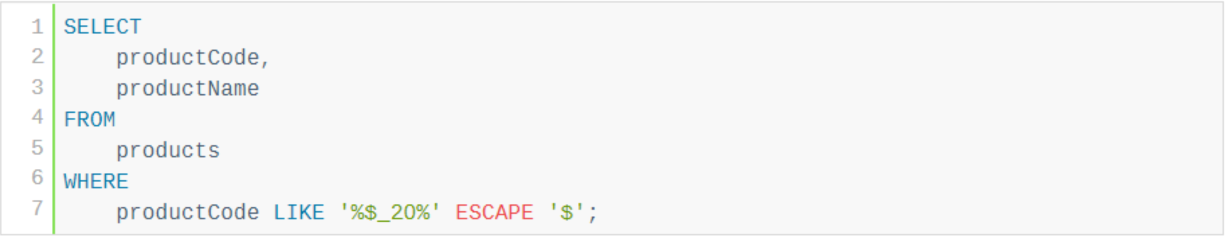
Sometimes the pattern, which you want to match, contains wildcard character e.g., 10%, _20, etc. In this case, you can use the ESCAPE clause to specify the escape character so that MySQL will interpret the wildcard character as a literal character. If you don't specify the escape character explicitly, the backslash character \ is the default escape character.

For example, if you want to find products whose product codes contain string _20, you can use the pattern %_20% as shown in the following query:

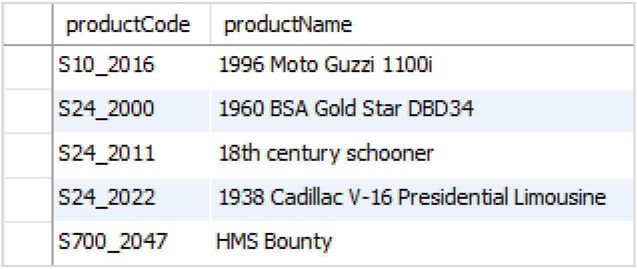


```
SELECT
    productCode,
    productName
FROM
    products
WHERE
    productCode LIKE '%\_20%';
```

Or you can specify a different escape character e.g., $ by using the ESCAPE clause:



```
SELECT
    productCode,
    productName
FROM
    products
WHERE
    productCode LIKE '%$_20%' ESCAPE '$';
```

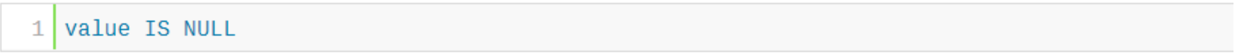


	productCode	productName
	S10_2016	1996 Moto Guzzi 1100i
	S24_2000	1960 BSA Gold Star DBD34
	S24_2011	18th century schooner
	S24_2022	1938 Cadillac V-16 Presidential Limousine
	S700_2047	HMS Bounty

The pattern %$_20% matches any string that contains the _20 string.

MySQL IS NULL

To test whether a value is NULL or not, you use the IS NULL operator. The following shows syntax of the IS NULL operator:

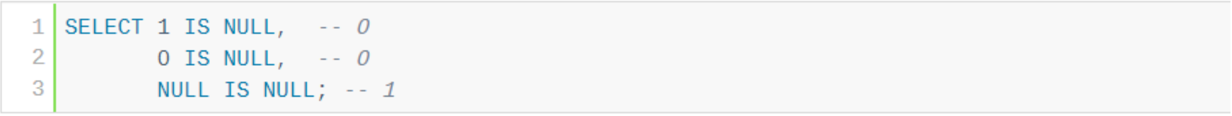


```
value IS NULL
```

If the value is NULL, the expression returns true. Otherwise, it returns false.

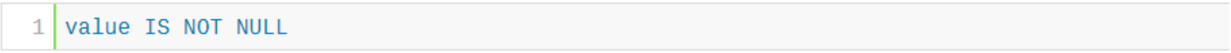
Note that MySQL does not have a built-in BOOLEAN type. It uses the TINYINT(1) to represent the BOOLEAN values i.e., true means 1 and false means 0.

Because the IS NULL is a comparison operator, you can use it anywhere that an operator can be used e.g., in the SELECT or WHERE clause. See the follow example:



```
SELECT 1 IS NULL,  -- 0
       0 IS NULL,  -- 0
       NULL IS NULL; -- 1
```

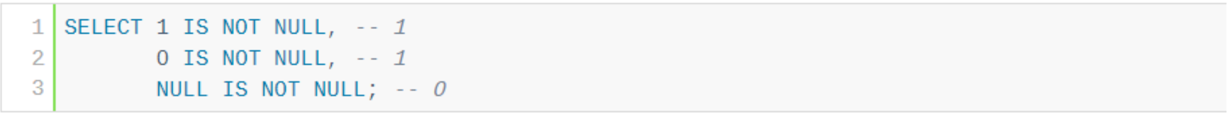
To check if a value is not NULL, you use IS NOT NULL operator as follows:



```
value IS NOT NULL
```

This expression returns true (1) if the value is not NULL. Otherwise, it returns false (0).

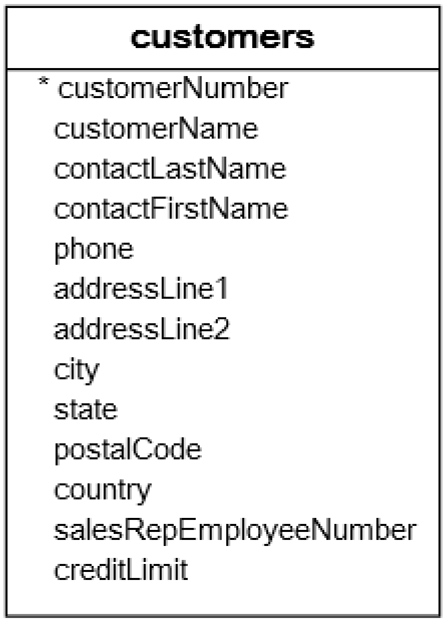
Consider the following example:



```
SELECT 1 IS NOT NULL, -- 1
       0 IS NOT NULL, -- 1
       NULL IS NOT NULL; -- 0
```

MySQL IS NULL examples

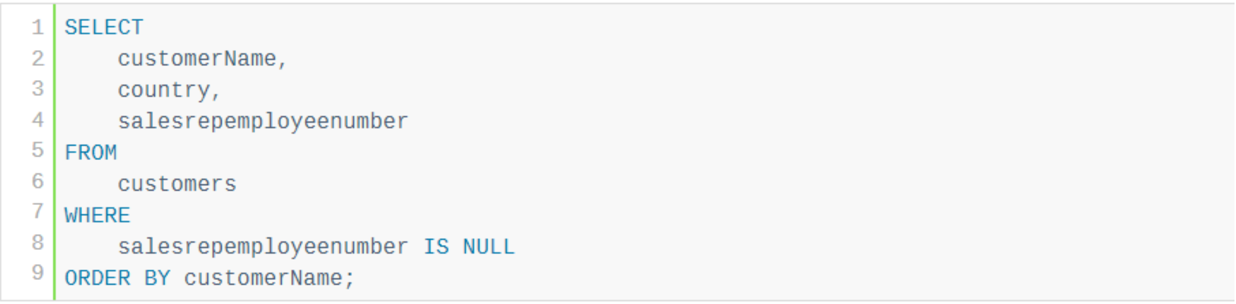
We will use the customers table in the **sample database** for the demonstration.



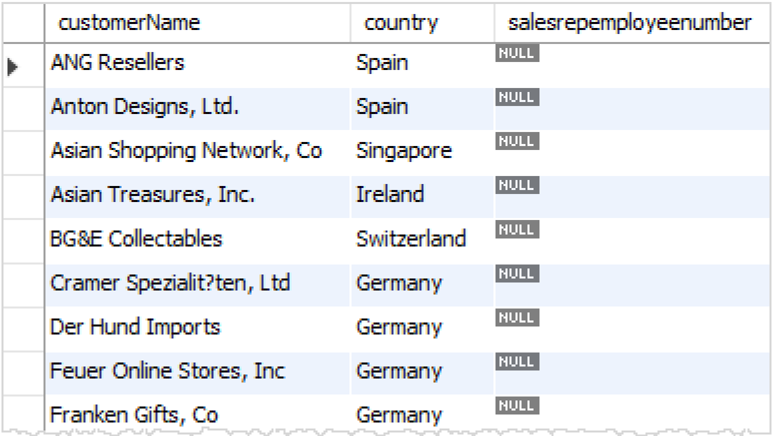
customers

* customerNumber
customerName
contactLastName
contactFirstName
phone
addressLine1
addressLine2
city
state
postalCode
country
salesRepEmployeeNumber
creditLimit

The following query uses the IS NULL operator to find customers who do not have a sales representative:

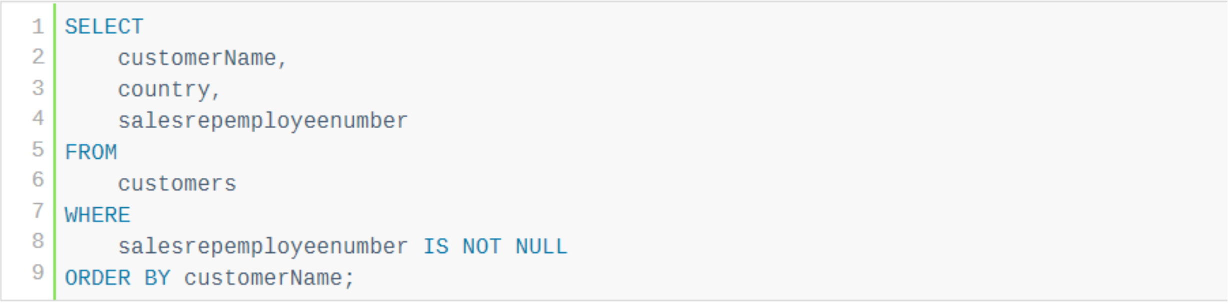


```
SELECT
    customerName,
    country,
    salesrepemployeenumber
FROM
    customers
WHERE
    salesrepemployeenumber IS NULL
ORDER BY customerName;
```

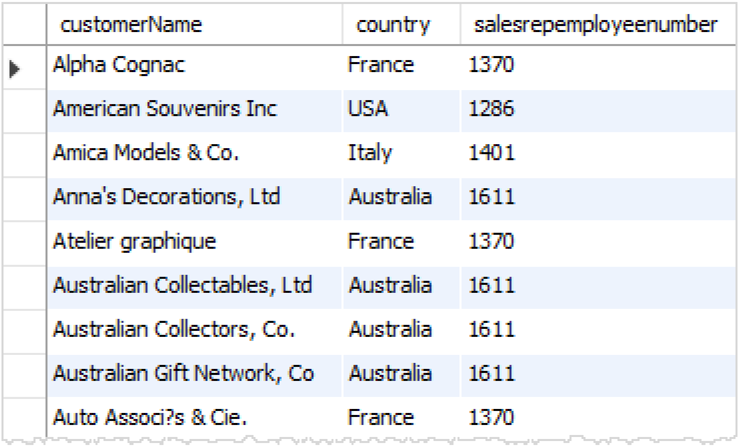


customerName	country	salesrepemployeenumber
ANG Resellers	Spain	NULL
Anton Designs, Ltd.	Spain	NULL
Asian Shopping Network, Co	Singapore	NULL
Asian Treasures, Inc.	Ireland	NULL
BG&E Collectables	Switzerland	NULL
Cramer Spezialit?ten, Ltd	Germany	NULL
Der Hund Imports	Germany	NULL
Feuer Online Stores, Inc	Germany	NULL
Franken Gifts, Co	Germany	NULL

This example uses the IS NOT NULL operator to get the customers who have a sales representative:



```
SELECT
    customerName,
    country,
    salesrepemployeenumber
FROM
    customers
WHERE
    salesrepemployeenumber IS NOT NULL
ORDER BY customerName;
```

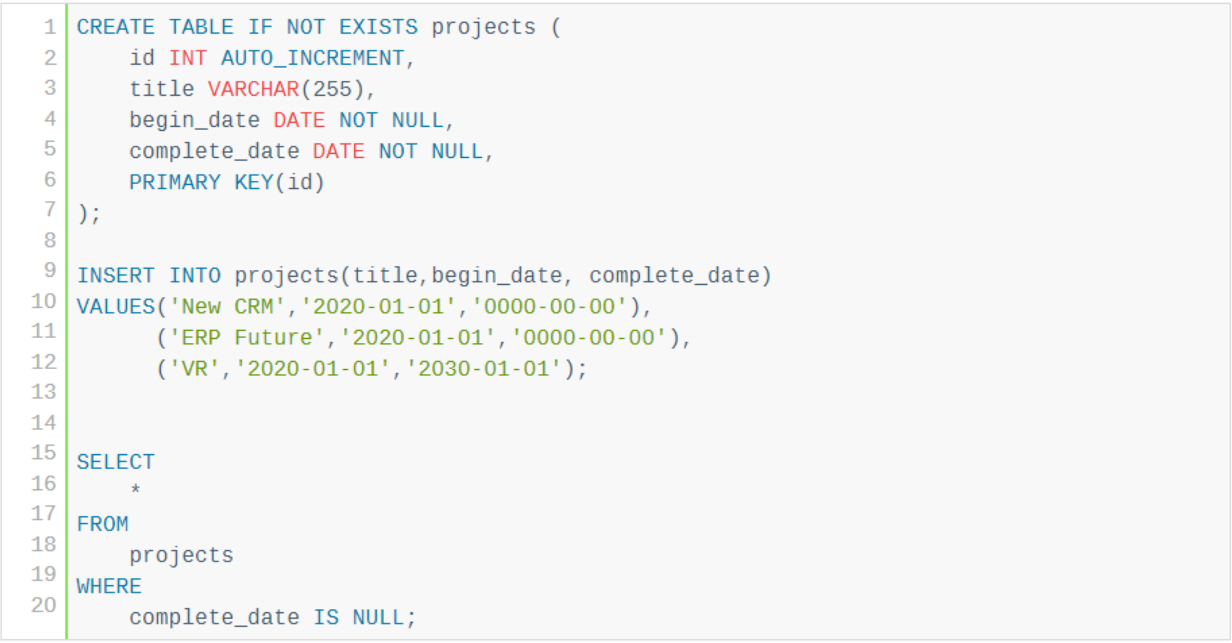


customerName	country	salesrepemployeenumber
Alpha Cognac	France	1370
American Souvenirs Inc	USA	1286
Amica Models & Co.	Italy	1401
Anna's Decorations, Ltd	Australia	1611
Atelier graphique	France	1370
Australian Collectables, Ltd	Australia	1611
Australian Collectors, Co.	Australia	1611
Australian Gift Network, Co	Australia	1611
Auto Associ?s & Cie.	France	1370

MySQL IS NULL's specialized features

To be compatible with ODBC programs, MySQL supports some specialized features of the IS NULL operator.

1) If the DATE or DATETIME column that has a NOT NULL constraint and contains a special date '0000-00-00', you can use the IS NULL operator to find such rows.

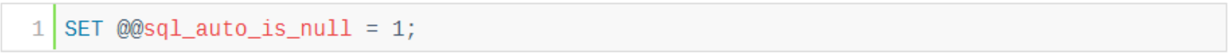


```
CREATE TABLE IF NOT EXISTS projects (
    id INT AUTO_INCREMENT,
    title VARCHAR(255),
    begin_date DATE NOT NULL,
    complete_date DATE NOT NULL,
    PRIMARY KEY(id)
);

INSERT INTO projects(title,begin_date, complete_date)
VALUES('New CRM','2020-01-01','0000-00-00'),
      ('ERP Future','2020-01-01','0000-00-00'),
      ('VR','2020-01-01','2030-01-01');


SELECT
    *
FROM
    projects
WHERE
    complete_date IS NULL;
```

In this example, we created a new table named projects and insert some data into the table. The last query used IS NULL to get the rows whose values in the complete_date value is '0000-00-00'.

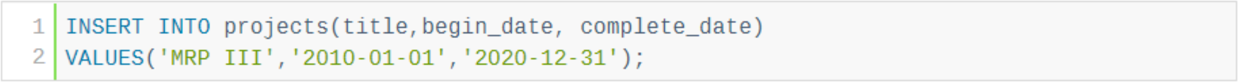
2) If the variable @@sql_auto_is_null is set to 1, you can get the value of a generated column after executing an INSERT statement by using the IS NULL operator. Note that by default the variable @@sql_auto_is_null is 0. Consider the following example:

First, set the variable @@sql_auto_is_null to 1.



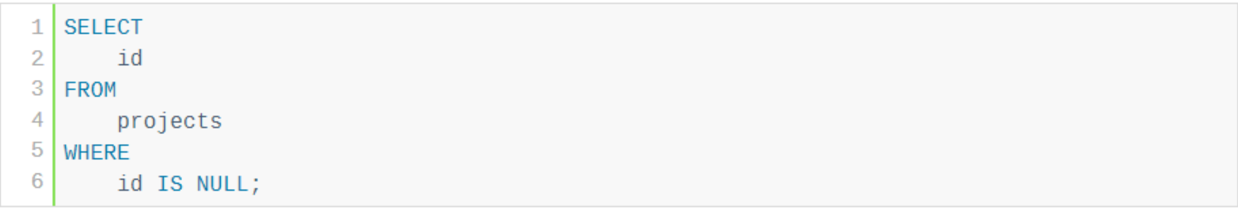
```
SET @@sql_auto_is_null = 1;
```

Second, insert a new row into the projects table:

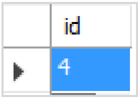


```
INSERT INTO projects(title,begin_date, complete_date)
VALUES('MRP III','2010-01-01','2020-12-31');
```

Third, use the IS NULL operator to get the generated value of the id column:



```
SELECT
    id
FROM
    projects
WHERE
    id IS NULL;
```

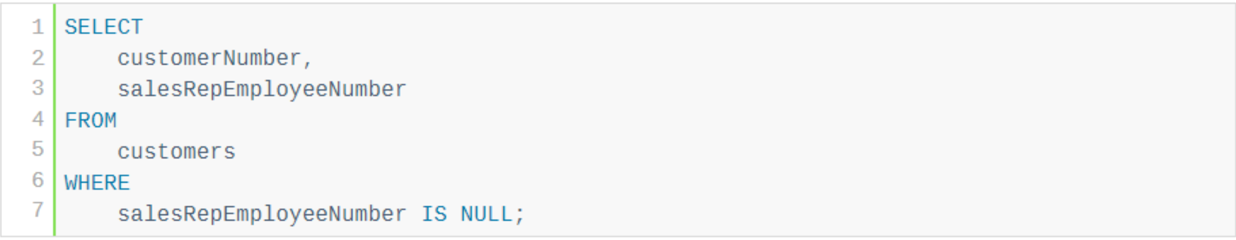


	id
▶	4

MySQL IS NULL Optimization

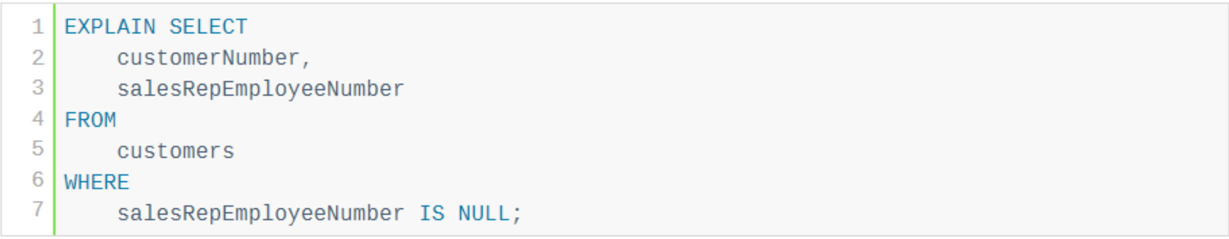
MySQL performs the same optimization for the IS NULL operator in the same way that it does for the equal (=) operator.

For example, MySQL uses the index when it searches for NULL with the IS NULL operator as shown in the following query:

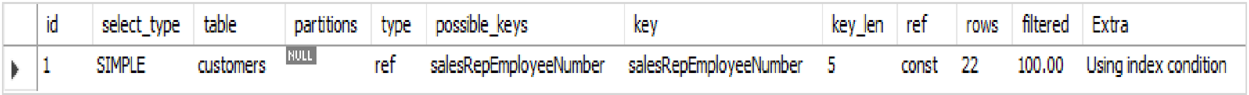


```
SELECT
    customerNumber,
    salesRepEmployeeNumber
FROM
    customers
WHERE
    salesRepEmployeeNumber IS NULL;
```

See the EXPLAIN of the query:

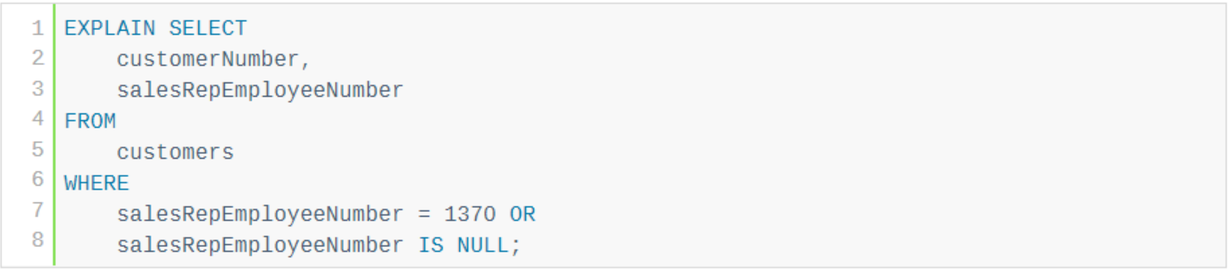


```
EXPLAIN SELECT
    customerNumber,
    salesRepEmployeeNumber
FROM
    customers
WHERE
    salesRepEmployeeNumber IS NULL;
```

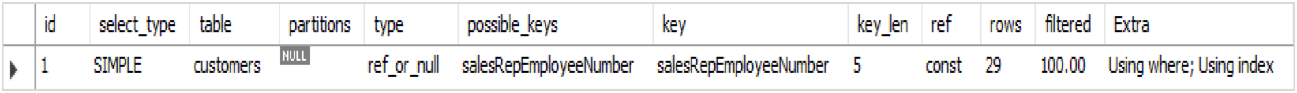


	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	customers	NULL	ref	salesRepEmployeeNumber	salesRepEmployeeNumber	5	const	22	100.00	Using index condition

MySQL can also optimize for the combination col = value OR col IS NULL, see the following example:



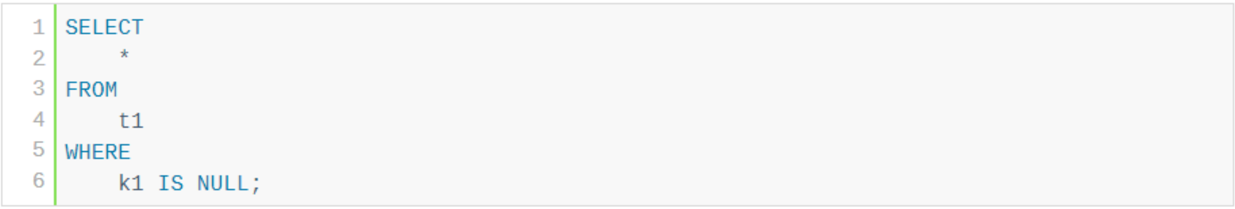
```
EXPLAIN SELECT
    customerNumber,
    salesRepEmployeeNumber
FROM
    customers
WHERE
    salesRepEmployeeNumber = 1370 OR
    salesRepEmployeeNumber IS NULL;
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customers	NULL	ref_or_null	salesRepEmployeeNumber	salesRepEmployeeNumber	5	const	29	100.00	Using where; Using index

In this example, the EXPLAIN shows ref_or_null when the optimization is applied.

If you have a key that is a combination of columns, MySQL can perform optimization for any key part. Suppose there is an index on columns k1 and k2 of the table t1, the following query is leveraging the index:



```
SELECT
    *
FROM
    t1
WHERE
    k1 IS NULL;
```

In this tutorial, you have learned how to use MySQL IS NULL operator to test whether a value is NULL or not.

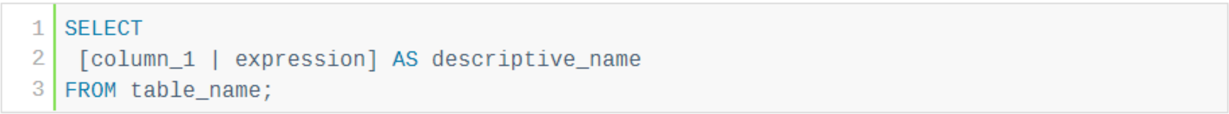
MySQL Alias

MySQL supports two kinds of aliases which are known as **column** alias and **table** alias. Let’s examine each kind of alias in detail.

MySQL alias for columns

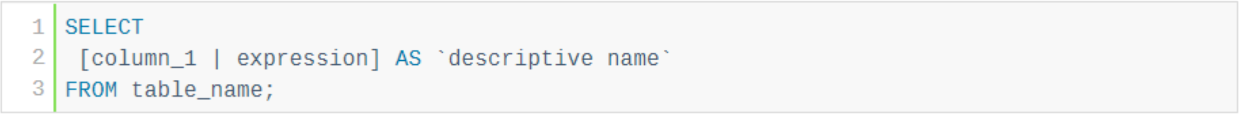
Sometimes the names of columns are so technical that make the query's output very difficult to understand. To give a column a descriptive name, you use a column alias.

The following statement illustrates how to use the column alias:



```
SELECT
 [column_1 | expression] AS descriptive_name
FROM table_name;
```

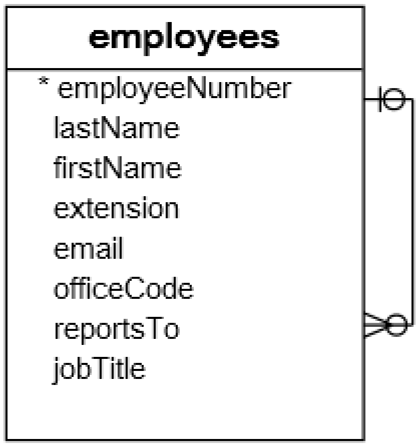
To give a column an alias, you use the AS keyword followed by the alias. If the alias contains space, you must quote it as the following:



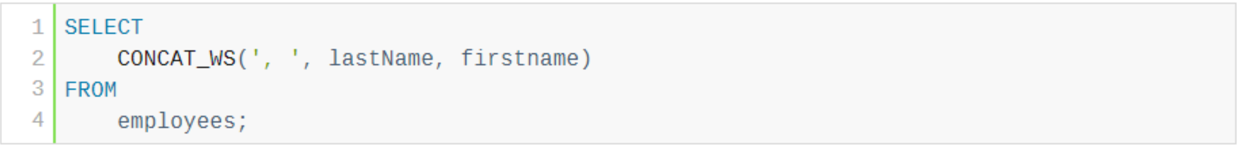
```
SELECT
 [column_1 | expression] AS `descriptive name`
FROM table_name;
```

Because the AS keyword is optional, you can omit it in the statement. Note that you can also give an expression an alias.

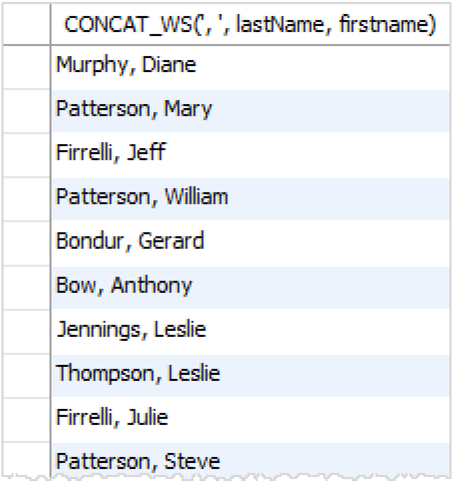
Let's look at the employees table in the **sample database**.



The following query selects first names and last names of employees and combines them to produce the full names. The CONCAT_WS function is used to **concatenate** first name and last name.

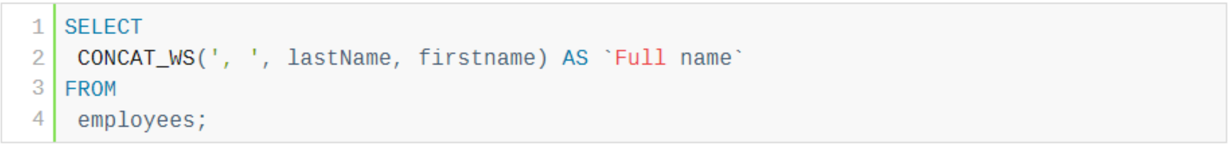


```
SELECT
    CONCAT_WS(', ', lastName, firstname)
FROM
    employees;
```

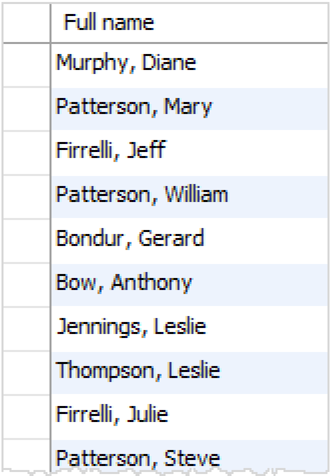


CONCAT_WS(', ', lastName, firstname)
Murphy, Diane
Patterson, Mary
Firrelli, Jeff
Patterson, William
Bondur, Gerard
Bow, Anthony
Jennings, Leslie
Thompson, Leslie
Firrelli, Julie
Patterson, Steve

The column heading is quite difficult to read. You can assign the heading of the output a column alias to make it more readable as the following query:



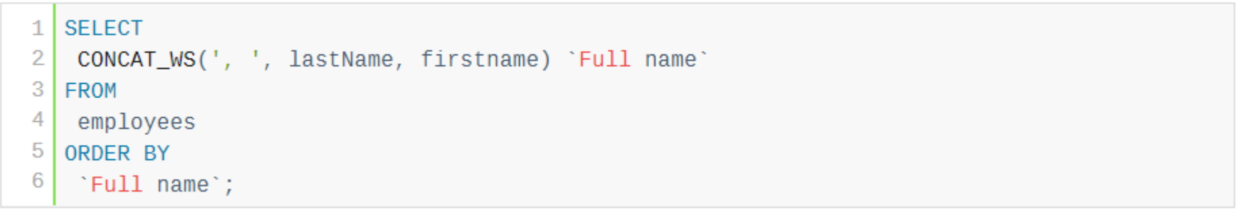
```
SELECT
 CONCAT_WS(', ', lastName, firstname) AS `Full name`
FROM
 employees;
```



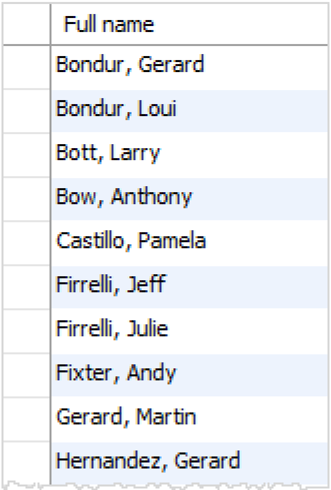
Full name
Murphy, Diane
Patterson, Mary
Firrelli, Jeff
Patterson, William
Bondur, Gerard
Bow, Anthony
Jennings, Leslie
Thompson, Leslie
Firrelli, Julie
Patterson, Steve

In MySQL, you can use the column alias in the ORDER BY, GROUP BY and HAVING clauses to refer to the column.

The following query uses the column alias in the ORDER BY clause to sort the employee's full names alphabetically:

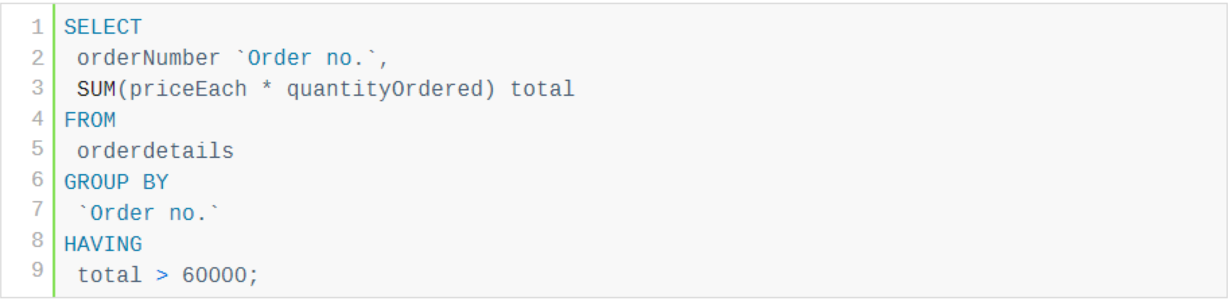


```
SELECT
 CONCAT_WS(', ', lastName, firstname) `Full name`
FROM
 employees
ORDER BY
 `Full name`;
```

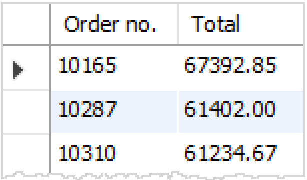


Full name
Bondur, Gerard
Bondur, Loui
Bott, Larry
Bow, Anthony
Castillo, Pamela
Firrelli, Jeff
Firrelli, Julie
Fixter, Andy
Gerard, Martin
Hernandez, Gerard

The following statement selects the orders whose total amount are greater than 60000. It uses column aliases in GROUP BY and HAVING clauses.



```
SELECT
 orderNumber `Order no.`,
 SUM(priceEach * quantityOrdered) total
FROM
 orderdetails
GROUP BY
 `Order no.`
HAVING
 total > 60000;
```

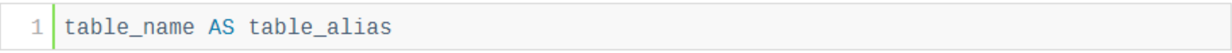


	Order no.	Total
▶	10165	67392.85
	10287	61402.00
	10310	61234.67

Notice that you cannot use a column alias in the WHERE clause. The reason is that when MySQL evaluates the WHERE clause, the values of columns specified in the SELECT clause may not be determined yet.

MySQL alias for tables

You can use an alias to give a table a different name. You assign a table an alias by using the AS keyword as the following syntax:

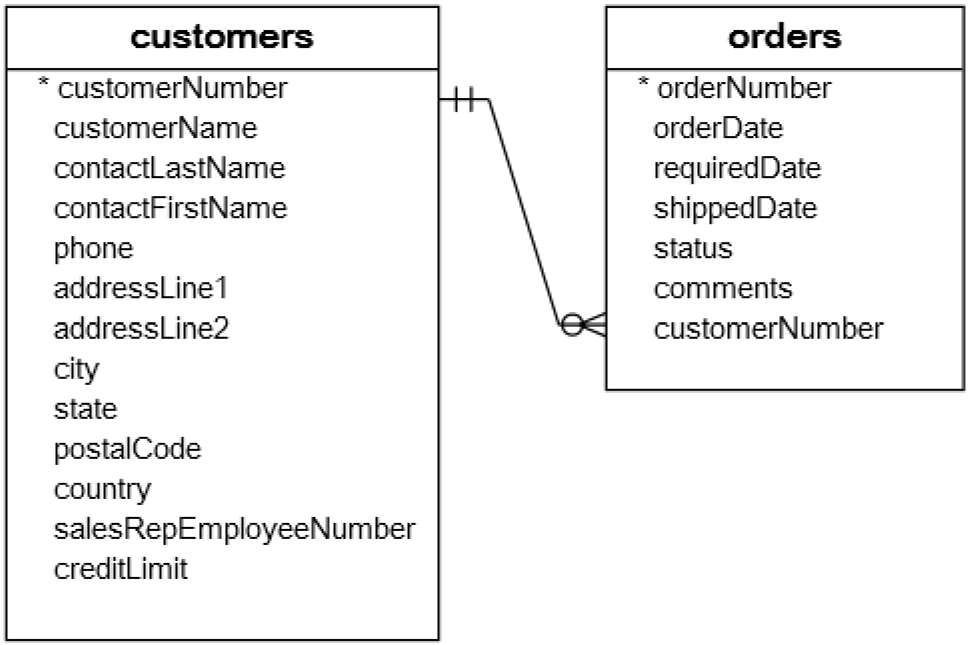


```
table_name AS table_alias
```

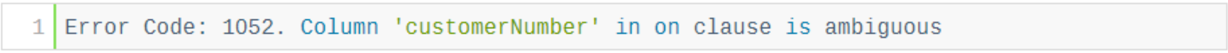
The alias for the table is called table alias. Like the column alias, the AS keyword is optional so you can omit it.

You often use the table alias in the statement that contains INNER JOIN, LEFT JOIN, self-join clauses, and in subqueries.

Let’s look at the customers and orders tables.

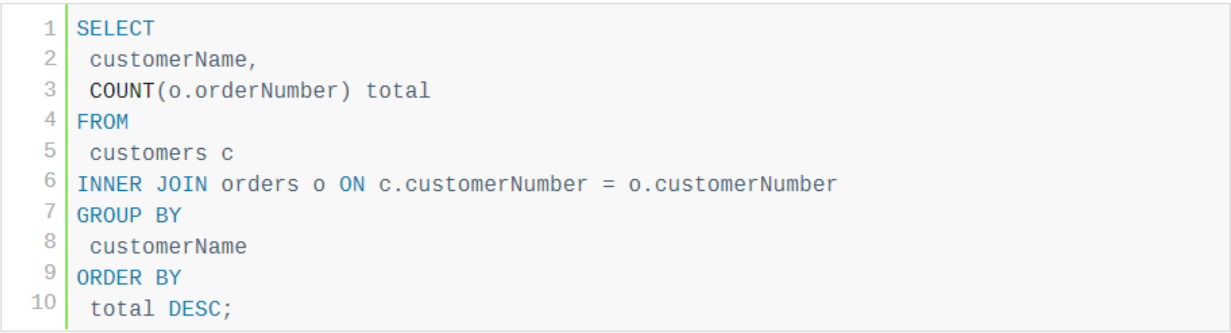


Both tables have the same column name: customerNumber. Without using the table alias to qualify the customerNumber column, you will get an error message like:

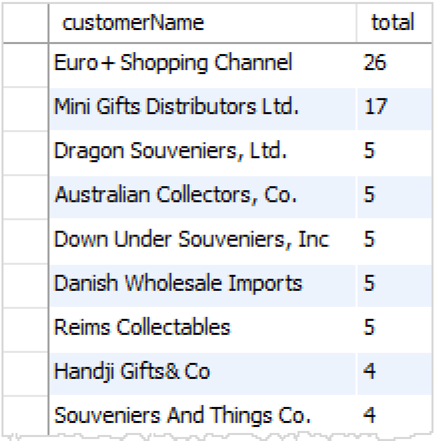


```
Error Code: 1052. Column 'customerNumber' in on clause is ambiguous
```

To avoid this error, you use table alias to qualify the customerNumber column:



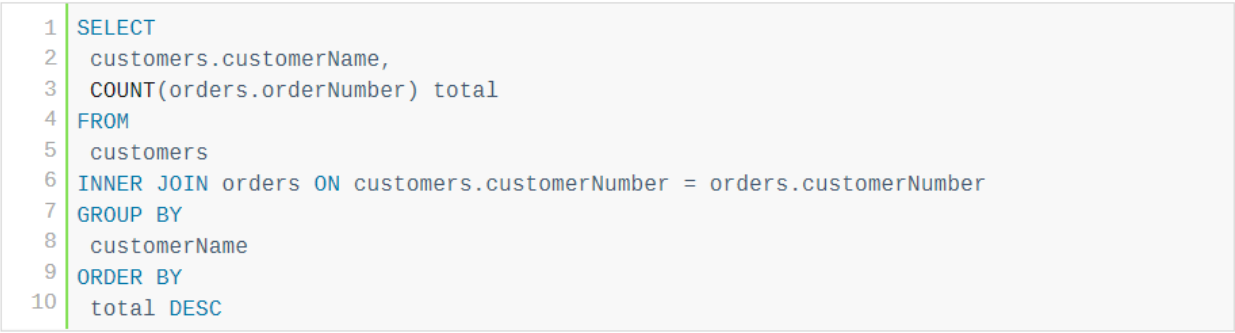
```
SELECT
 customerName,
 COUNT(o.orderNumber) total
FROM
 customers c
INNER JOIN orders o ON c.customerNumber = o.customerNumber
GROUP BY
 customerName
ORDER BY
 total DESC;
```



customerName	total
Euro+ Shopping Channel	26
Mini Gifts Distributors Ltd.	17
Dragon Souveniers, Ltd.	5
Australian Collectors, Co.	5
Down Under Souveniers, Inc	5
Danish Wholesale Imports	5
Reims Collectables	5
Handji Gifts& Co	4
Souveniers And Things Co.	4

The query above selects customer name and the number of orders from the customers and orders tables. It uses c as a table alias for the customers table and o as a table alias for the orders table. The columns in the customers and orders tables are referred to via the table aliases.

If you do not use alias in the query above, you have to use the table name to refer to its columns, which makes the query lengthy and less readable as the following:



```
SELECT
 customers.customerName,
 COUNT(orders.orderNumber) total
FROM
 customers
INNER JOIN orders ON customers.customerNumber = orders.customerNumber
GROUP BY
 customerName
ORDER BY
 total DESC
```

MySQL Join

A relational database consists of multiple related tables linking together using common columns which are known as foreign key columns. Because of this, data in each table is incomplete from the business perspective.

For example, in the **sample database**, we have the orders and orderdetails tables that are linked using the orderNumber column.

To get complete orders' data, you need to query data from both orders and orderdetails table.

And that's why MySQL JOIN comes into the play.

A MySQL join is a method of linking data between one (self-join) or more tables based on values of the common column between tables.

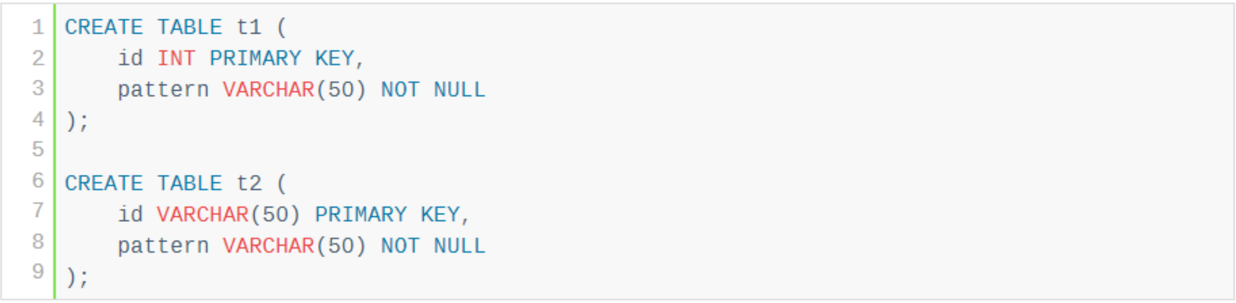
MySQL supports the following types of joins:

- Cross join
- Inner join
- Left join
- Right join

To join tables, you use the CROSS JOIN, INNER JOIN, LEFT JOIN or RIGHT JOIN clause for the corresponding type of join. The join clause is used in the SELECT statement appeared after the FROM clause.

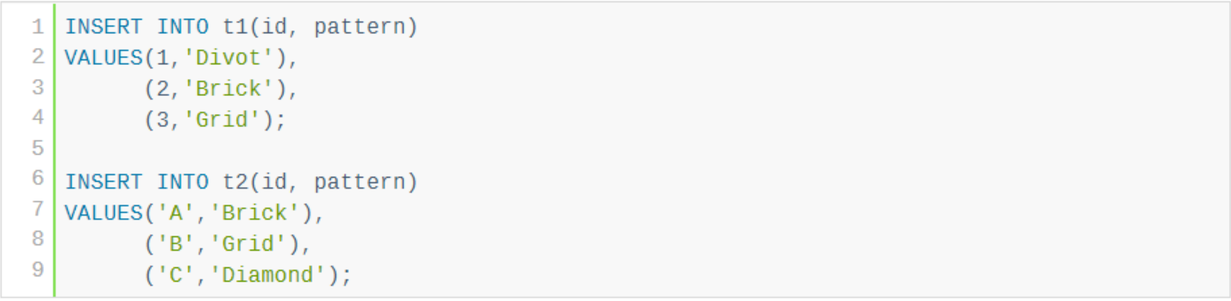
Notice that MySQL does not support full outer join.

To make easy for you to understand each type of join, we will use the t1 and t2 tables with the following structures:



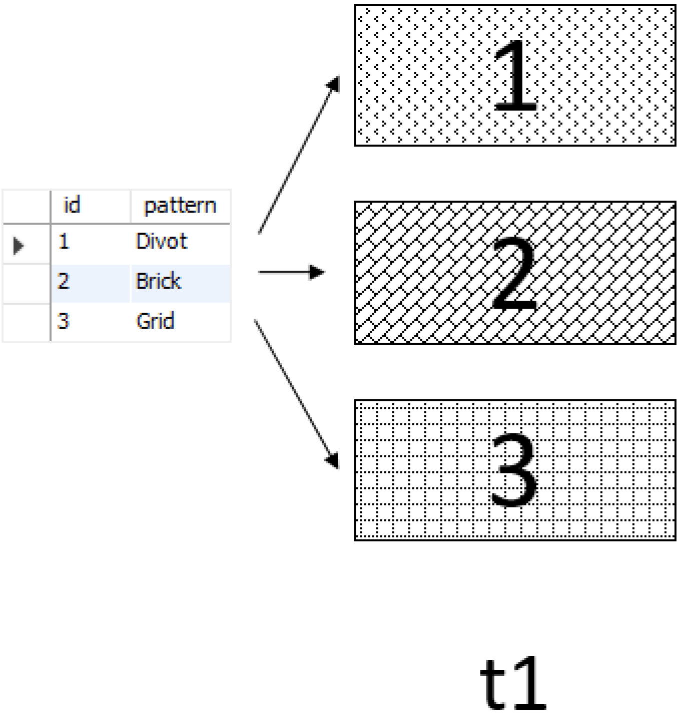
```
CREATE TABLE t1 (
    id INT PRIMARY KEY,
    pattern VARCHAR(50) NOT NULL
);

CREATE TABLE t2 (
    id VARCHAR(50) PRIMARY KEY,
    pattern VARCHAR(50) NOT NULL
);
```

Both t1 and t2 tables have the pattern column, which is also the common column between tables. The following statements insert data into both t1 and t2 tables:



```
INSERT INTO t1(id, pattern)
VALUES(1,'Divot'),
      (2,'Brick'),
      (3,'Grid');

INSERT INTO t2(id, pattern)
VALUES('A','Brick'),
      ('B','Grid'),
      ('C','Diamond');
```

And the pictures below illustrate data from both t1 and t2 tables:



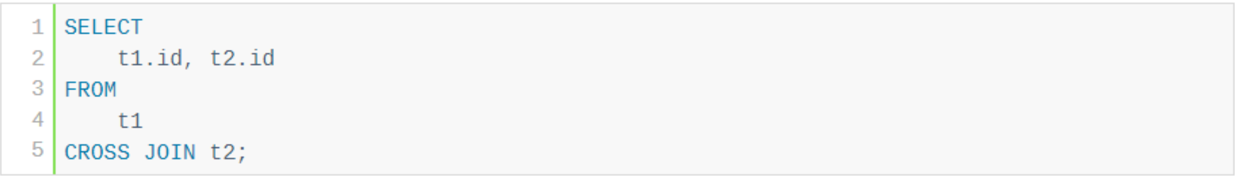
id	pattern
1	Divot
2	Brick
3	Grid

MySQL CROSS JOIN

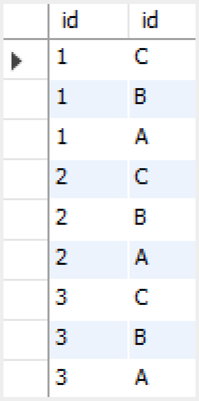
The CROSS JOIN makes a Cartesian product of rows from multiple tables. Suppose, you join t1 and t2 tables using the CROSS JOIN, the result set will include the combinations of rows from the t1 table with the rows in the t2 table.

To perform cross join, you use the CROSS JOIN clause as in the following statement:

The following shows the result set of the query:



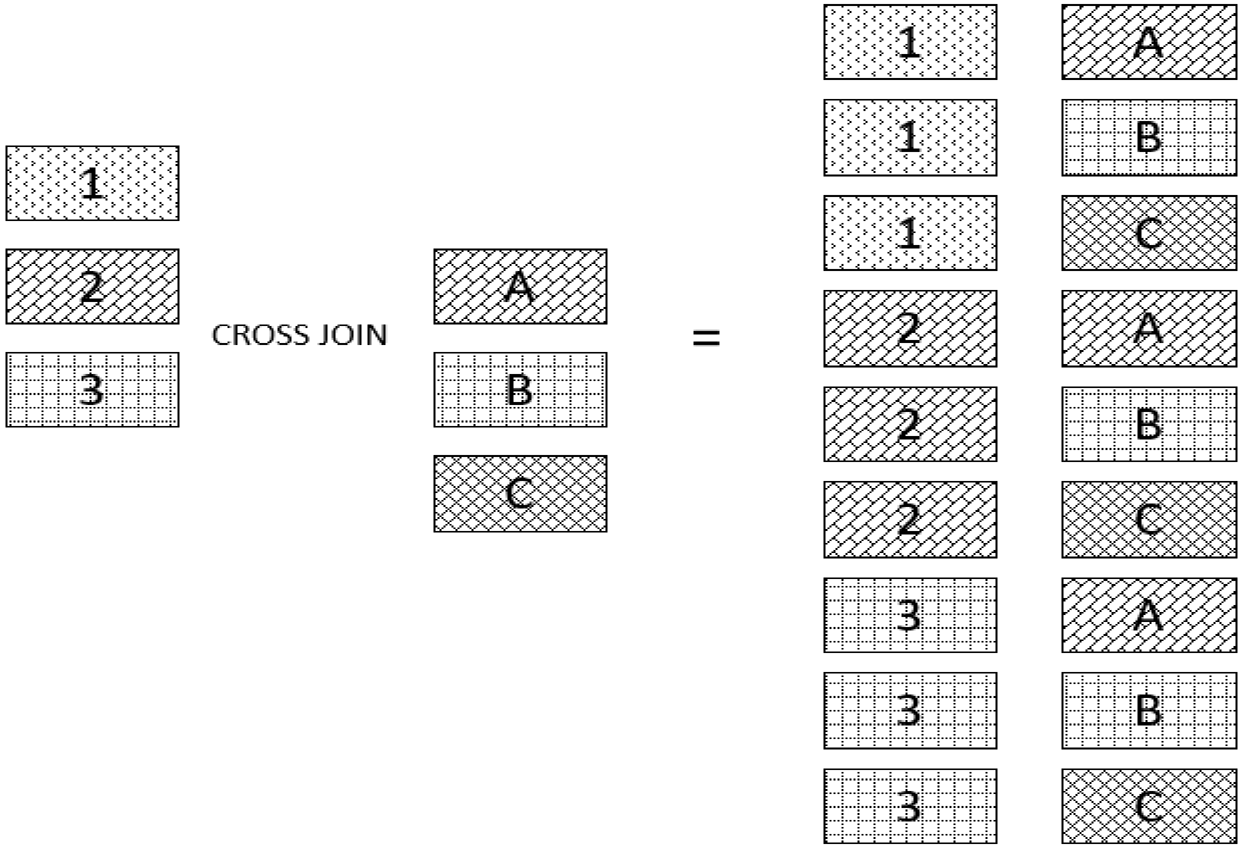
```
SELECT
    t1.id, t2.id
FROM
    t1
CROSS JOIN t2;
```



id	id
1	C
1	B
1	A
2	C
2	B
2	A
3	C
3	B
3	A

As you can see, each row in the t1 table combines with rows in the t2 table to form the Cartesian product.

The following picture illustrates the CROSS JOIN between t1 and t2 tables.

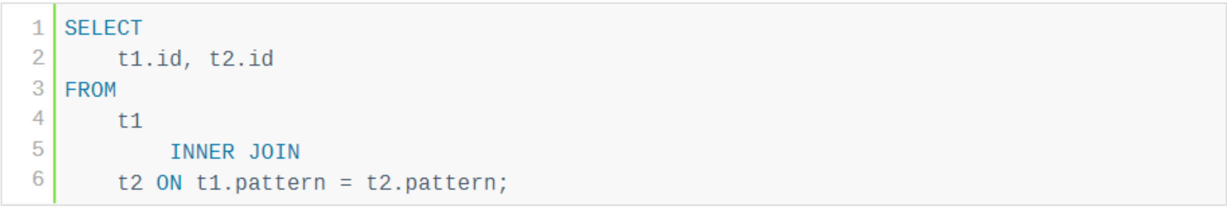


MySQL INNER JOIN

To form an INNER JOIN, you need a condition which is known as a join-predicate. An INNER JOIN requires rows in the two joined tables to have matching column values. The INNER JOIN creates the result set by combining column values of two joined tables based on the join-predicate.

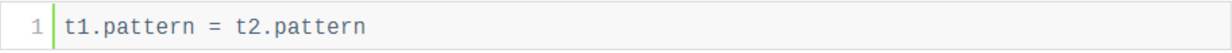
To join two tables, the INNER JOIN compares each row in the first table with each row in the second table to find pairs of rows that satisfy the join-predicate. Whenever the join-predicate is satisfied by matching non-NULL values, column values for each matched pair of rows of the two tables are included in the result set.

The following statement uses the INNER JOIN clause to join t1 and t2 tables:



```
SELECT
    t1.id, t2.id
FROM
    t1
        INNER JOIN
    t2 ON t1.pattern = t2.pattern;
```

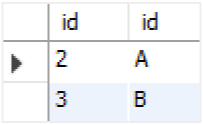
In this statement, the following expression is the join-predicate:



```
t1.pattern = t2.pattern
```

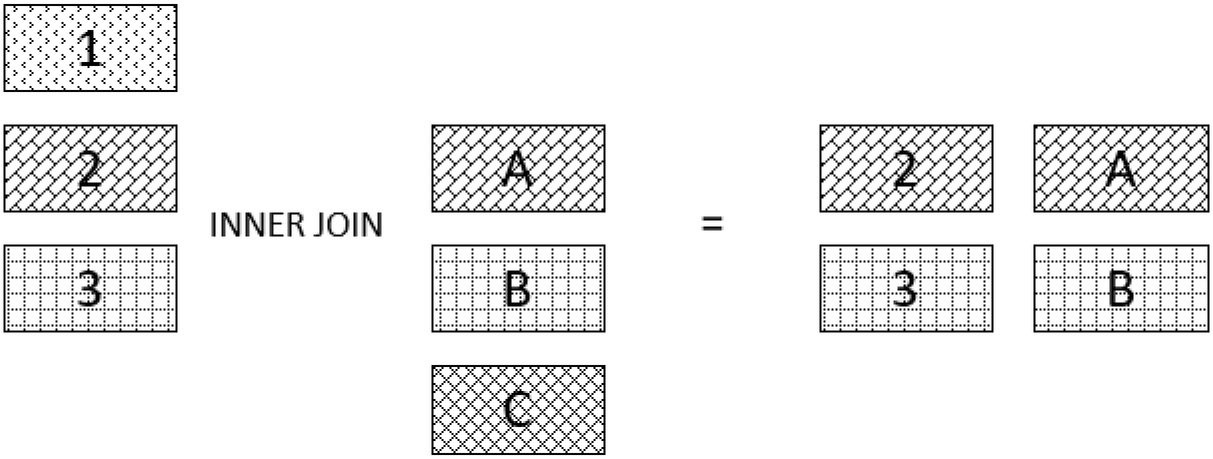
It means that rows in t1 and t2 tables must have the same values in the pattern column to be included in the result.

The following illustrates the result of the query:



id	id
2	A
3	B

The following picture illustrates the INNER JOIN between t1 and t2 tables:



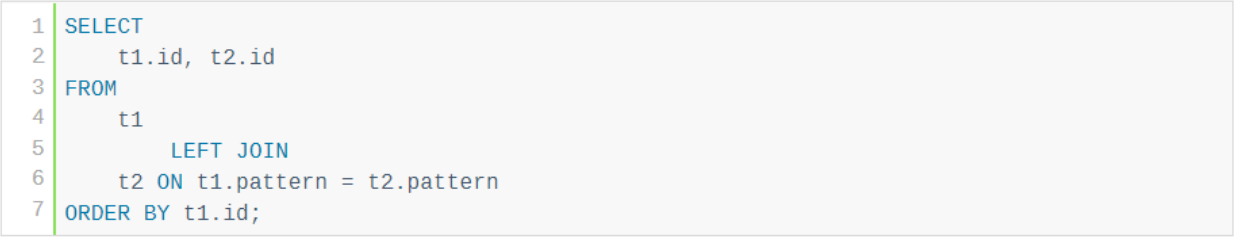
In this illustration, the rows in both tables must have the same pattern to be included in the result set.

MySQL LEFT JOIN

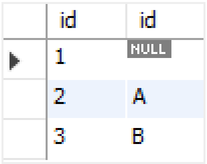
Similar to an INNER JOIN, a LEFT JOIN also requires a join-predicate. When joining two tables using a LEFT JOIN, the concepts of left table and right table are introduced.

Unlike an INNER JOIN, a LEFT JOIN returns all rows in the left table including rows that satisfy join-predicate and rows that do not. For the rows that do not match the join-predicate, NULLs appear in the columns of the right table in the result set.

The following statement uses the LEFT JOIN clause to join t1 and t2 tables:



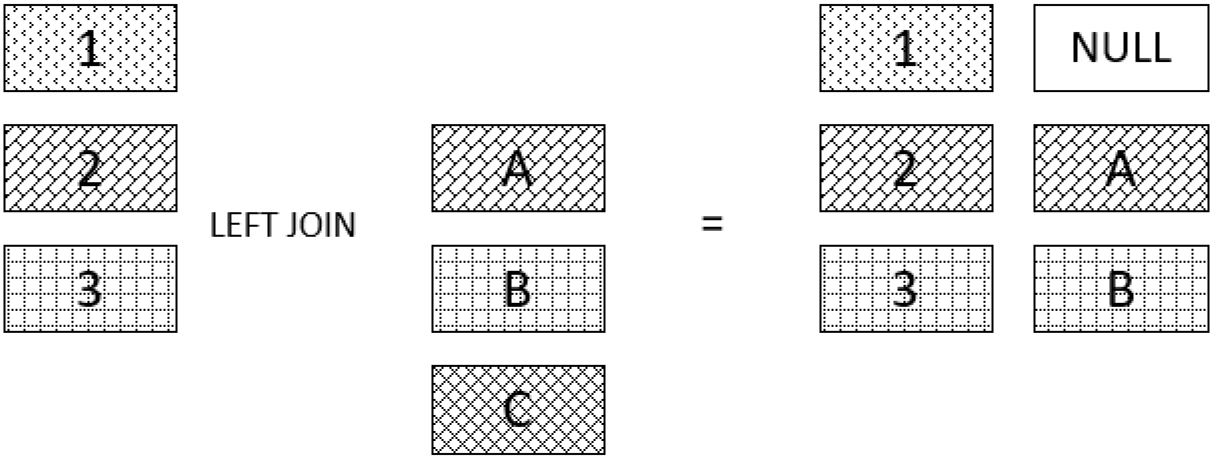
```
SELECT
    t1.id, t2.id
FROM
    t1
        LEFT JOIN
    t2 ON t1.pattern = t2.pattern
ORDER BY t1.id;
```



id	id
1	NULL
2	A
3	B

As you can see, all rows in the t1 table are included in the result set. For the rows in the t1 table (left table) that do not have any matching row in the t2 table (right table), NULLs are used for columns in t2table.

The following picture illustrates the LEFT JOIN between t1 and t2 tables:

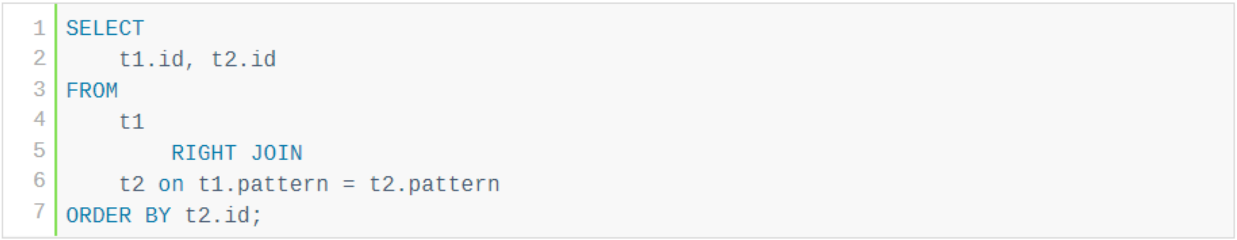


In this illustration, the following rows share the same pattern: (2 and A), (3 and B). The row with id 1 in the t1 table has no matching row in the t2 table, therefore, NULL are used for columns of the t2 table in the result set.

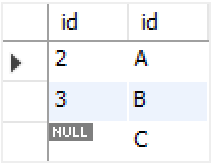
MySQL RIGHT JOIN

A RIGHT JOIN is similar to the LEFT JOIN except that the treatment of tables is reversed. With a RIGHT JOIN, every row from the right table (t2) will appear in the result set. For the rows in the right table that do not have the matching rows in the left table (t1), NULLs appear for columns in the left table (t1).

The following statement joins t1 and t2 tables using RIGHT JOIN:



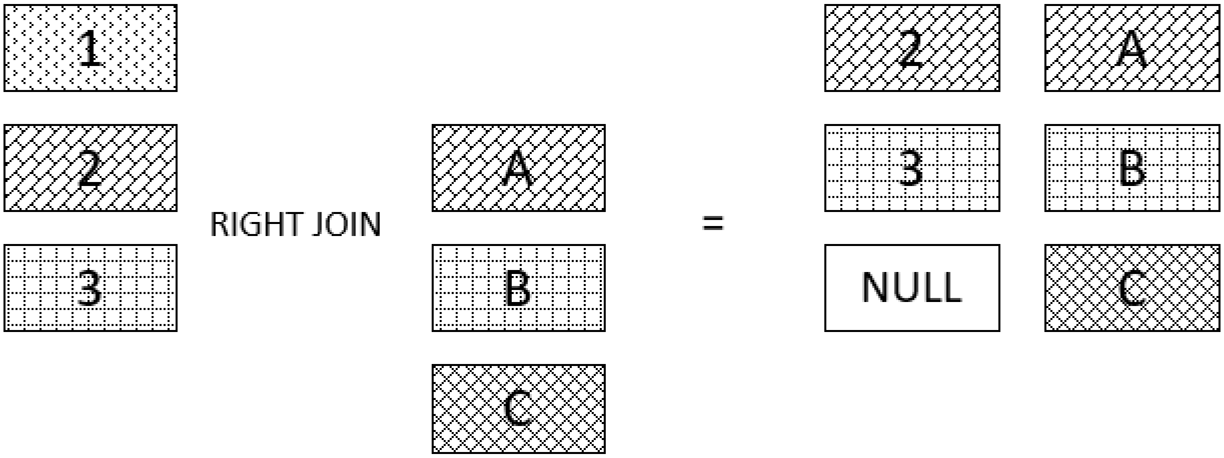
```
SELECT
    t1.id, t2.id
FROM
    t1
        RIGHT JOIN
    t2 on t1.pattern = t2.pattern
ORDER BY t2.id;
```



id	id
2	A
3	B
NULL	C

In this result, all rows from the right table (t2) appear in the result set. For the rows in the right table (t2) that have no matching rows in the left table (t1), NULL appears for columns of the left table (t1).

The following picture illustrates the RIGHT JOIN between t1 and t2 tables:



In this tutorial, you have learned various MySQL join statements including cross join, inner join, left join and right join to query data from two or more tables.

MySQL INNER JOIN

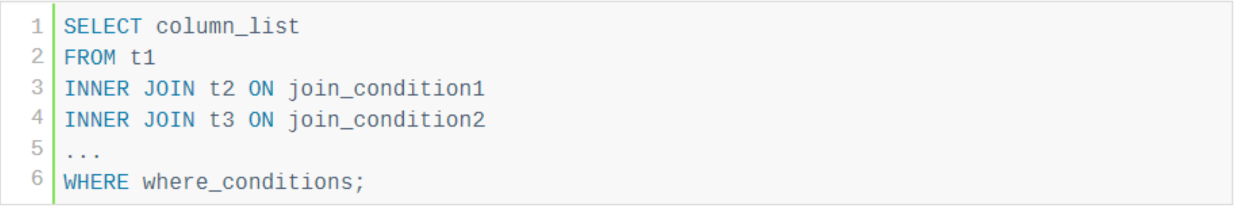
The MySQL INNER JOIN clause matches rows in one table with rows in other tables and allows you to query rows that contain columns from both tables.

The INNER JOIN clause is an optional part of the SELECT statement. It appears immediately after the FROM clause.

Before using the INNER JOIN clause, you have to specify the following criteria:

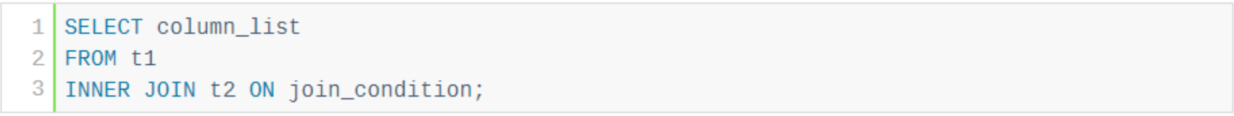
- First, the main table that appears in the FROM clause.
- Second, the table that you want to join with the main table, which appears in the INNER JOIN clause. In theory, you can join a table with many other tables. However, for a better performance, you should limit the number of tables to join.
- Third, the join condition or join predicate. The join condition appears after the ON keyword of the INNER JOIN clause. The join condition is the rule for matching rows in the main table with the rows in the other tables.

The syntax of the INNER JOIN clause is as follows:



```
SELECT column_list
FROM t1
INNER JOIN t2 ON join_condition1
INNER JOIN t3 ON join_condition2
...
WHERE where_conditions;
```

Let’s simplify the syntax above by assuming that we are joining two tables t1 and t2 using the INNER JOIN clause.

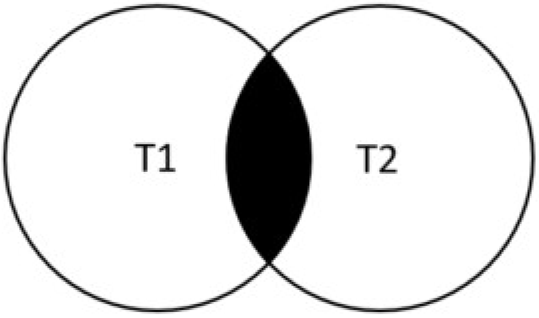


```
SELECT column_list
FROM t1
INNER JOIN t2 ON join_condition;
```

For each row in the t1 table, the INNER JOIN clause compares it with each row of the t2 table to check if both of them satisfy the join condition. When the join condition is met, the INNER JOIN will return a new row which consists of columns in both t1 and t2 tables.

Notice that the rows in both t1 and t2 tables have to be matched based on the join condition. If no match found, the query will return an empty result set. This logic is also applied when you join more than 2 tables.

The following Venn diagram illustrates how the INNER JOIN clause works. The rows in the result set must appear in both tables: t1 and t2 as shown in the intersection part of two circles.



MySQL INNER JOIN Venn diagram

Avoid ambiguous column error in MySQL INNER JOIN

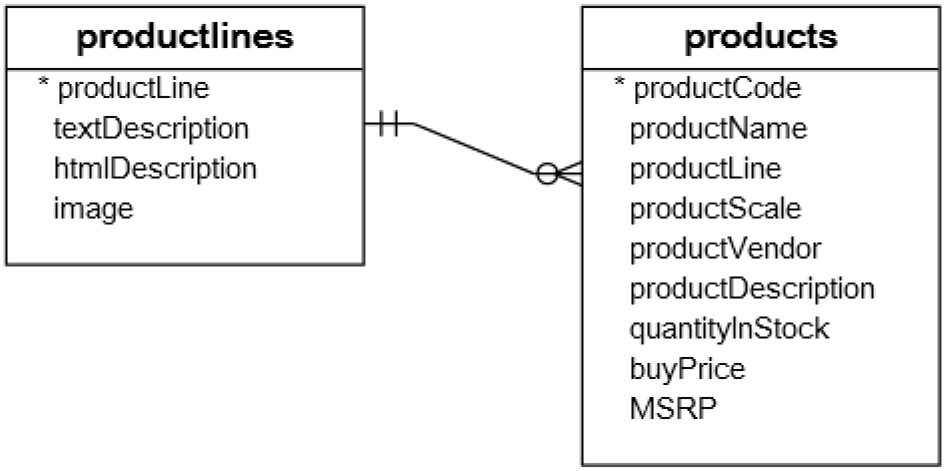
If you join multiple tables that have the same column name, you have to use table qualifier to refer to that column in the SELECT and ON clauses to avoid the ambiguous column error.

For example, if both t1 and t2 tables have the same column named c, you have to refer to the c column using the table qualifiers as t1.c or t2.c in the SELECT and ON clauses.

To save time typing the table qualifiers, you can use table aliases in the query. For example, you can give the verylongtablename table a table’s alias t and refer to its columns using t.column instead of using the verylongtablename.column.

MySQL INNER JOIN examples

Let’s look at the products and productlines tables in the sample database.



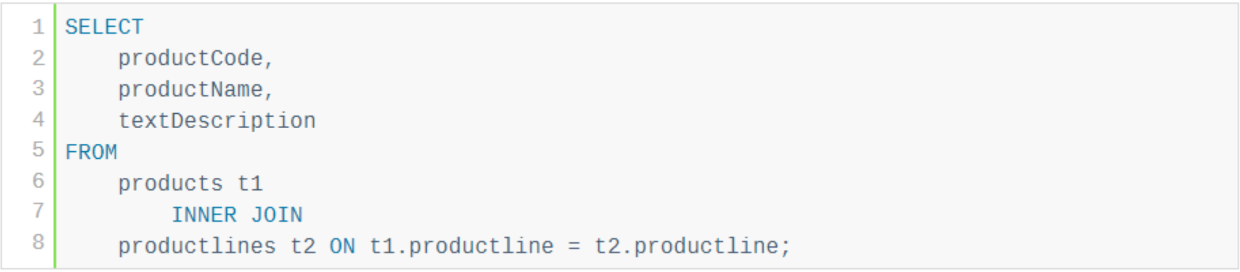
In this diagram, the products table has the productLine column referenced to the productline column of the productlines table. The productLine column in the products table is called a foreign key column.

Typically, you join tables that have foreign key relationships like the productlines and products tables.

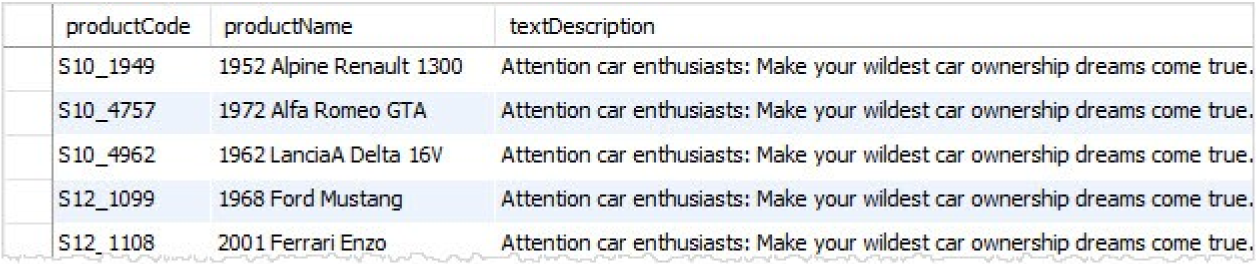
Now, if you want to get

1. The productCode and productName from the products table.
2. The textDescription of product lines from the productlines table.

To do this, you need to select data from both tables by matching rows based on the productline columns using the INNER JOIN clause as follows:

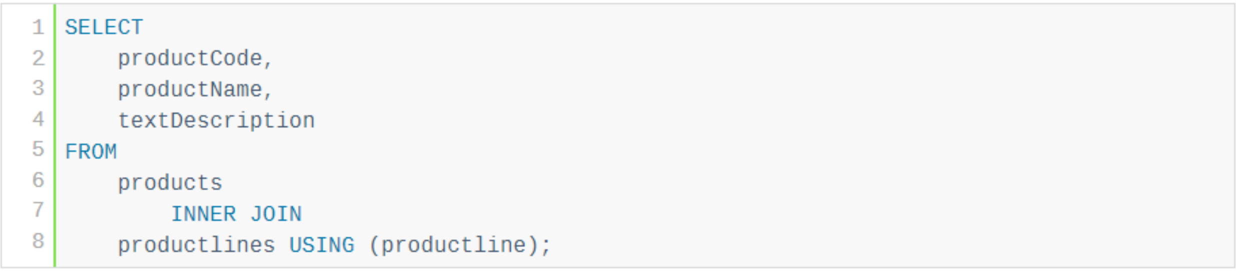


```
SELECT
    productCode,
    productName,
    textDescription
FROM
    products t1
        INNER JOIN
    productlines t2 ON t1.productline = t2.productline;
```



productCode	productName	textDescription
S10_1949	1952 Alpine Renault 1300	Attention car enthusiasts: Make your wildest car ownership dreams come true.
S10_4757	1972 Alfa Romeo GTA	Attention car enthusiasts: Make your wildest car ownership dreams come true.
S10_4962	1962 LanciaA Delta 16V	Attention car enthusiasts: Make your wildest car ownership dreams come true.
S12_1099	1968 Ford Mustang	Attention car enthusiasts: Make your wildest car ownership dreams come true.
S12_1108	2001 Ferrari Enzo	Attention car enthusiasts: Make your wildest car ownership dreams come true.

Because the joined columns of both tables have the same name productline, you can use the following syntax:

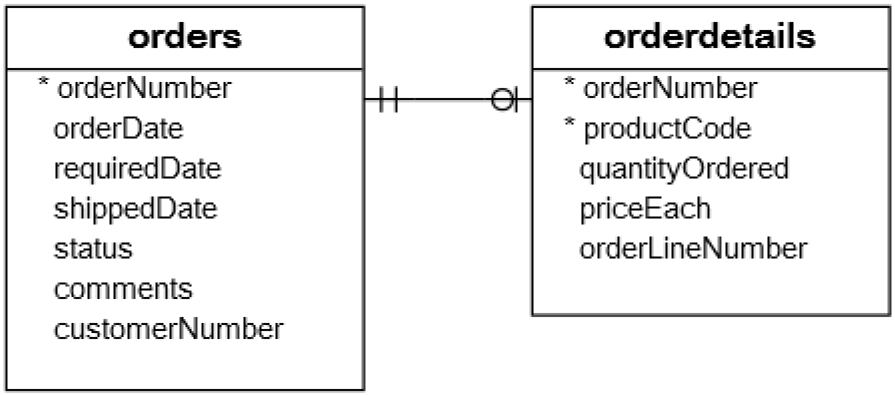


```sql
SELECT
    productCode,
    productName,
    textDescription
FROM
    products
        INNER JOIN
    productlines USING (productline);
```

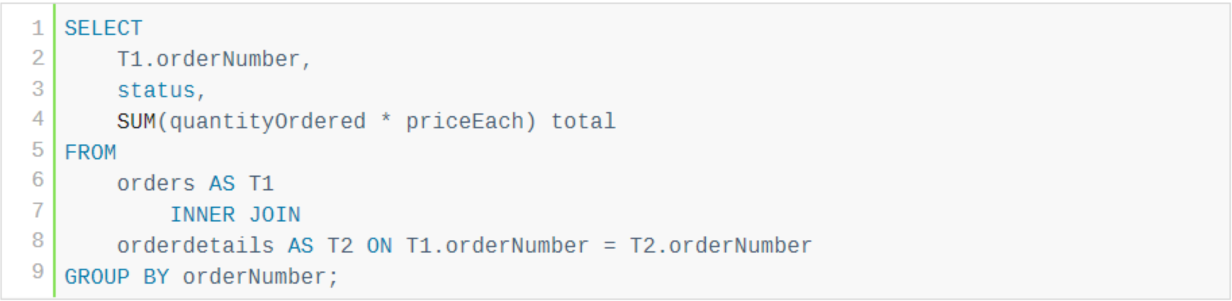
It returns the same result set however with this syntax you don't have to use the table aliases.

MySQL INNER JOIN with GROUP BY clause

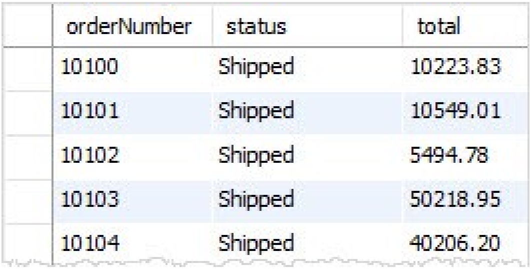
See the following orders and orderdetails tables.



You can get the order number, order status and total sales from the orders and orderdetails tables using the INNER JOIN clause with the GROUP BY clause as follows:

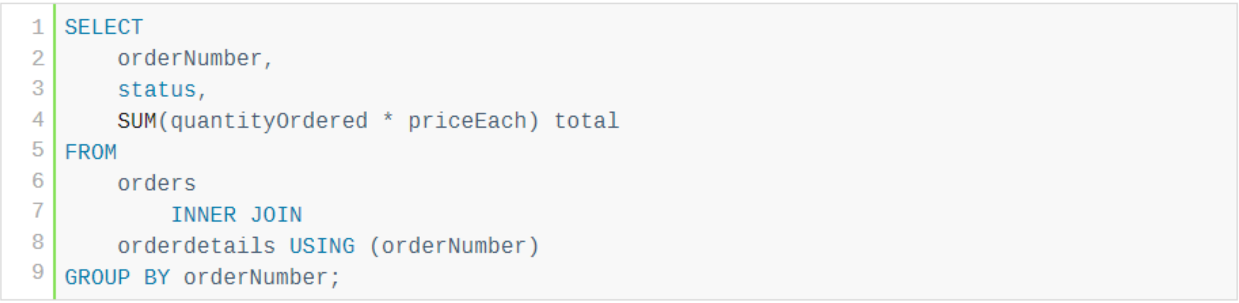


```
SELECT
    T1.orderNumber,
    status,
    SUM(quantityOrdered * priceEach) total
FROM
    orders AS T1
        INNER JOIN
    orderdetails AS T2 ON T1.orderNumber = T2.orderNumber
GROUP BY orderNumber;
```



orderNumber	status	total
10100	Shipped	10223.83
10101	Shipped	10549.01
10102	Shipped	5494.78
10103	Shipped	50218.95
10104	Shipped	40206.20

Similarly, the following query is equivalent to the one above:

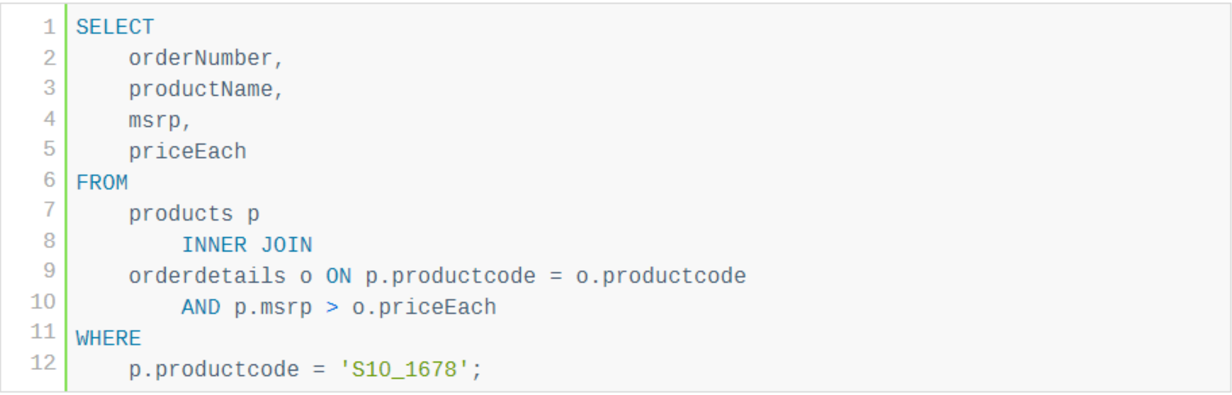


```
SELECT
    orderNumber,
    status,
    SUM(quantityOrdered * priceEach) total
FROM
    orders
        INNER JOIN
    orderdetails USING (orderNumber)
GROUP BY orderNumber;
```

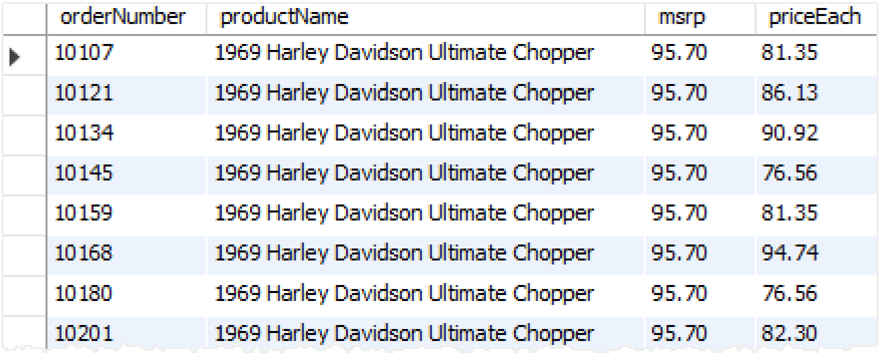
MySQL INNER JOIN using operator other than equal

So far, you have seen that the join predicate used the equal operator (=) for matching rows. In addition, you can use other operators such as greater than (>), less than (<), and not-equal (<>) operator to form the join predicates.

The following query uses a less-than (<) join to find sales prices of the product whose code is S10_1678that are less than the manufacturer’s suggested retail price (MSRP) for that product.



```
SELECT
    orderNumber,
    productName,
    msrp,
    priceEach
FROM
    products p
        INNER JOIN
    orderdetails o ON p.productcode = o.productcode
        AND p.msrp > o.priceEach
WHERE
    p.productcode = 'S10_1678';
```

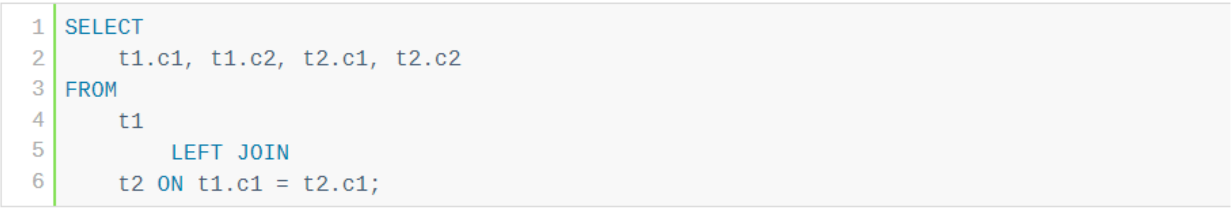


orderNumber	productName	msrp	priceEach
10107	1969 Harley Davidson Ultimate Chopper	95.70	81.35
10121	1969 Harley Davidson Ultimate Chopper	95.70	86.13
10134	1969 Harley Davidson Ultimate Chopper	95.70	90.92
10145	1969 Harley Davidson Ultimate Chopper	95.70	76.56
10159	1969 Harley Davidson Ultimate Chopper	95.70	81.35
10168	1969 Harley Davidson Ultimate Chopper	95.70	94.74
10180	1969 Harley Davidson Ultimate Chopper	95.70	76.56
10201	1969 Harley Davidson Ultimate Chopper	95.70	82.30

MySQL LEFT JOIN

The MySQL LEFT JOIN clause allows you to query data from two or more database tables. The LEFT JOIN clause is an optional part of the SELECT statement, which appears after the FROM clause.

Let’s assume that you are going to query data from two tables t1 and t2. The following statement illustrates the syntax of LEFT JOIN clause that joins the two tables:



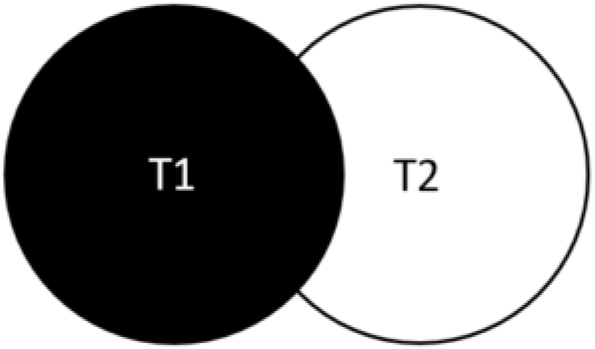
```
SELECT
    t1.c1, t1.c2, t2.c1, t2.c2
FROM
    t1
        LEFT JOIN
    t2 ON t1.c1 = t2.c1;
```

When you join the t1 table to the t2 table using the LEFT JOIN clause, if a row from the left table t1matches a row from the right table t2 based on the join condition (t1.c1 = t2.c1), this row will be included in the result set.

In case the row in the left table does not match with the row in the right table, the row in the left table is also selected and combined with a “fake” row from the right table. The fake row contains NULL for all corresponding columns in the SELECT clause.

In other words, the LEFT JOIN clause allows you to select rows from the both left and right tables that are matched, plus all rows from the left table (t1) even with no matching rows found in the right table (t2).

The following Venn diagram helps you visualize how the LEFT JOIN clause works. The intersection between two circles are rows that match in both tables, and the remaining part of the left circle are rows in the t1 table that do not have any matching row in the t2 table. Hence, all rows in the left table are included in the result set.



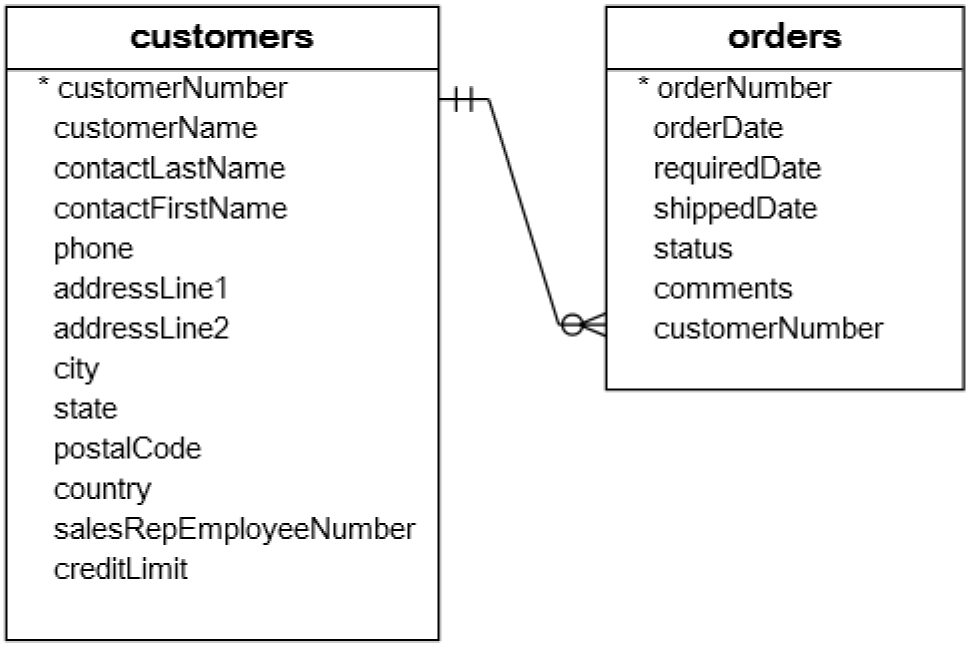
MySQL LEFT JOIN – Venn Diagram

Notice that the returned rows must also match the conditions in the WHERE and HAVING clauses if those clauses are available in the query.

MySQL LEFT JOIN examples

Using MySQL LEFT JOIN clause to join two tables

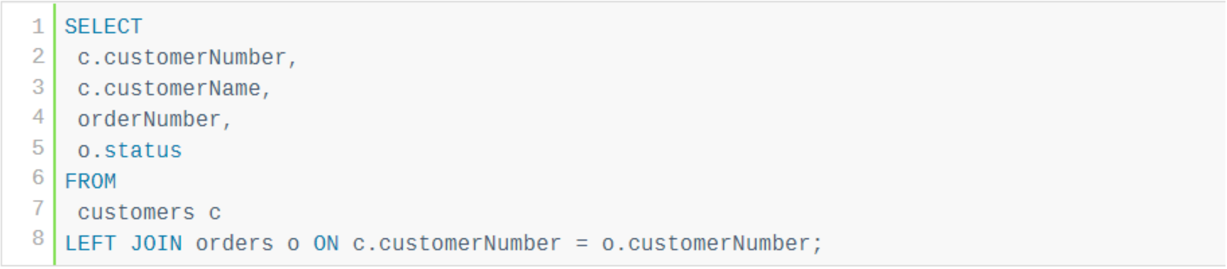
Let’s take a look at the customers and orders tables in the **sample database**.



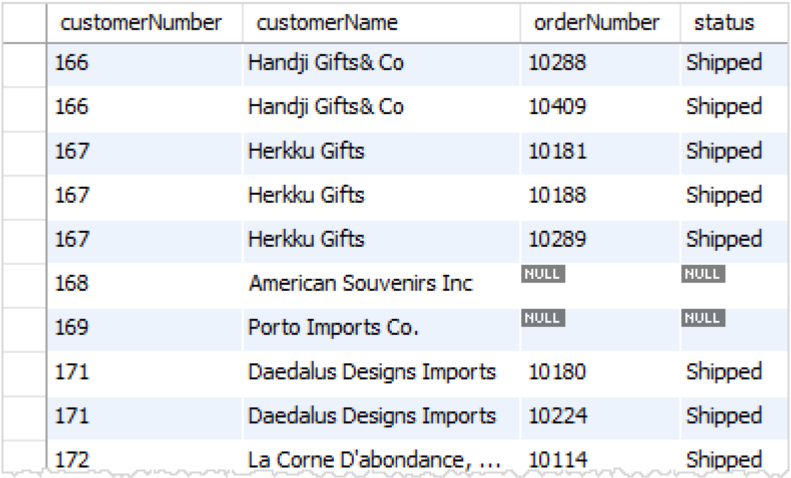
In the database diagram above:

1. Each order in the orders table must belong to a customer in the customers table.
2. Each customer in the customers table can have zero or more orders in the orders table.

To find all orders that belong to each customer, you can use the LEFT JOIN clause as follows:



```
SELECT
 c.customerNumber,
 c.customerName,
 orderNumber,
 o.status
FROM
 customers c
LEFT JOIN orders o ON c.customerNumber = o.customerNumber;
```



customerNumber	customerName	orderNumber	status
166	Handji Gifts& Co	10288	Shipped
166	Handji Gifts& Co	10409	Shipped
167	Herkku Gifts	10181	Shipped
167	Herkku Gifts	10188	Shipped
167	Herkku Gifts	10289	Shipped
168	American Souvenirs Inc	NULL	NULL
169	Porto Imports Co.	NULL	NULL
171	Daedalus Designs Imports	10180	Shipped
171	Daedalus Designs Imports	10224	Shipped
172	La Corne D'abondance, ...	10114	Shipped

The left table is customers, therefore, all customers are included in the result set. However, there are rows in the result set that have customer data but no order data e.g. 168, 169, etc. The order data in these rows are NULL. It means that these customers do not have any order in the orders table.

Because we used the same column name (customerNumber) for joining two tables, we can make the query shorter by using the following syntax:

```
SELECT
 c.customerNumber,
 customerName,
 orderNumber,
 status
FROM
 customers c
LEFT JOIN orders USING (customerNumber);
```

In this statement, the clause

```
USING (customerNumber)
```

is equivalent to

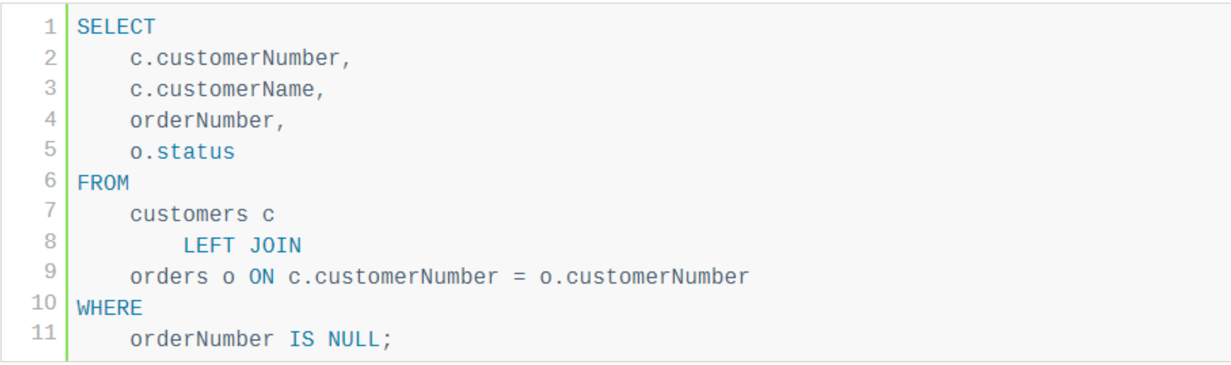
```
ON c.customerNumber = o.customerNumber
```

If you replace the LEFT JOIN clause by the INNER JOIN clause, you get the only customers who have placed at least one order.

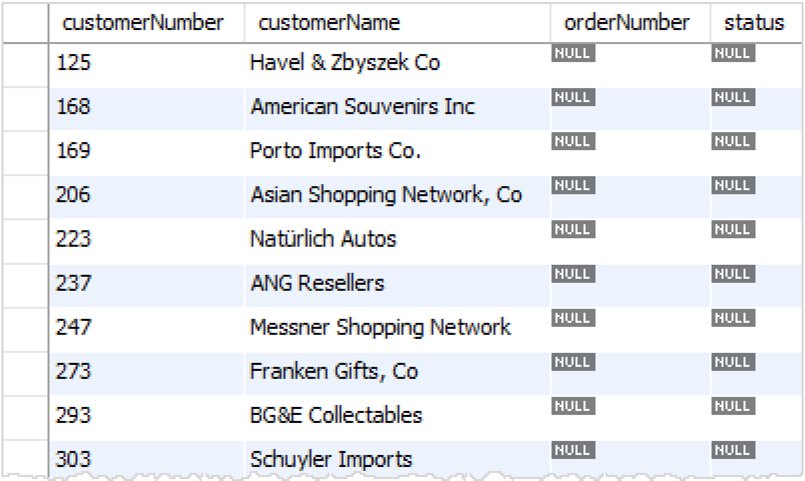
Using MySQL LEFT JOIN clause to find unmatched rows

The LEFT JOIN clause is very useful when you want to find the rows in the left table that do not match with the rows in the right table. To find the unmatching rows between two tables, you add a WHERE clause to the SELECT statement to query only rows whose column values in the right table contains the NULL values.

For example, to find all customers who have not placed any order, you use the following query:



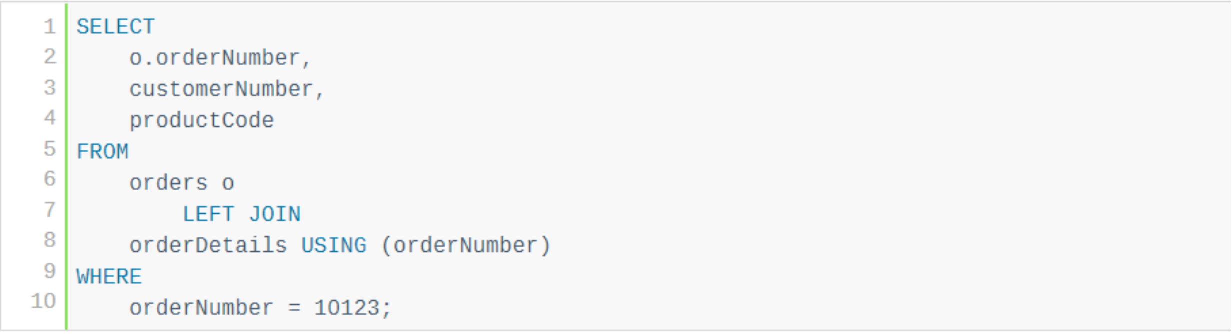
```
SELECT
    c.customerNumber,
    c.customerName,
    orderNumber,
    o.status
FROM
    customers c
        LEFT JOIN
    orders o ON c.customerNumber = o.customerNumber
WHERE
    orderNumber IS NULL;
```



customerNumber	customerName	orderNumber	status
125	Havel & Zbyszek Co	NULL	NULL
168	American Souvenirs Inc	NULL	NULL
169	Porto Imports Co.	NULL	NULL
206	Asian Shopping Network, Co	NULL	NULL
223	Natürlich Autos	NULL	NULL
237	ANG Resellers	NULL	NULL
247	Messner Shopping Network	NULL	NULL
273	Franken Gifts, Co	NULL	NULL
293	BG&E Collectables	NULL	NULL
303	Schuyler Imports	NULL	NULL

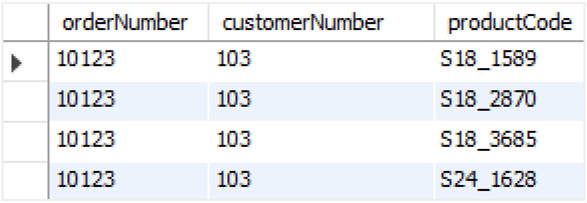
Condition in WHERE clause vs. ON clause

See the following example.



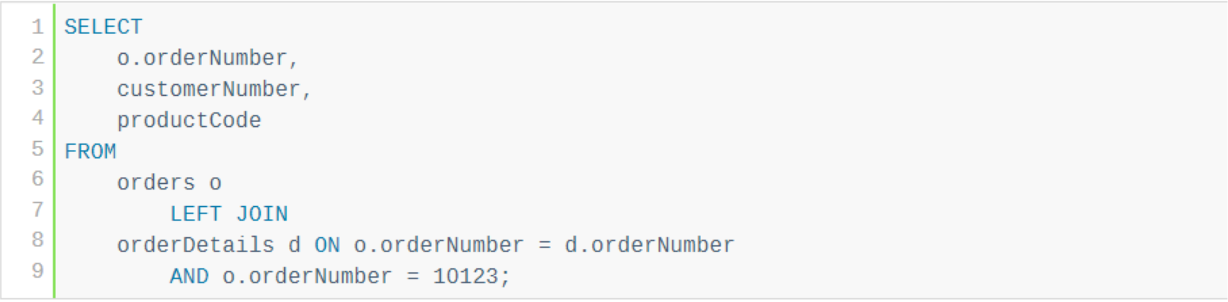
```
SELECT
    o.orderNumber,
    customerNumber,
    productCode
FROM
    orders o
        LEFT JOIN
    orderDetails USING (orderNumber)
WHERE
    orderNumber = 10123;
```

In this example, we used the LEFT JOIN clause to query data from the orders and orderDetails tables. The query returns an order and its detail, if any, for the order 10123.



	orderNumber	customerNumber	productCode
▶	10123	103	S18_1589
	10123	103	S18_2870
	10123	103	S18_3685
	10123	103	S24_1628

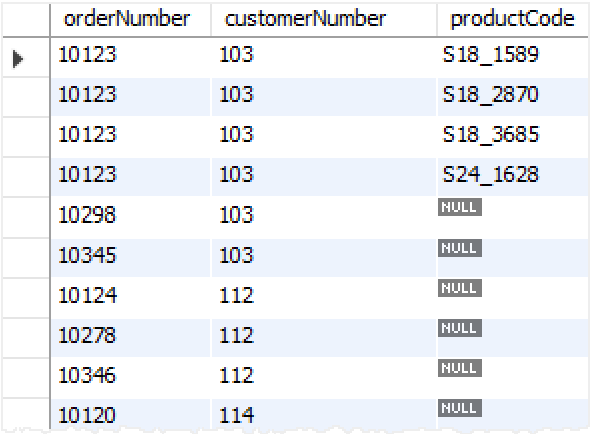
However, if you move the condition from the WHERE clause to the ON clause:



```
SELECT
    o.orderNumber,
    customerNumber,
    productCode
FROM
    orders o
        LEFT JOIN
    orderDetails d ON o.orderNumber = d.orderNumber
        AND o.orderNumber = 10123;
```

It will have a different meaning.

In this case, the query returns all orders but only the order 10123 will have detail associated with it as shown below.



orderNumber	customerNumber	productCode
10123	103	S18_1589
10123	103	S18_2870
10123	103	S18_3685
10123	103	S24_1628
10298	103	NULL
10345	103	NULL
10124	112	NULL
10278	112	NULL
10346	112	NULL
10120	114	NULL

Notice that for INNER JOIN clause, the condition in the ON clause is equivalent to the condition in the WHERE clause.

MySQL RIGHT JOIN

MySQL RIGHT JOIN is similar to LEFT JOIN, except the treatment of table reversed.

The following statement queries data from two tables t1 and t2 using the RIGHT JOIN clause:

```
SELECT
    *
FROM t1
    RIGHT JOIN t2 ON join_predicate;
```

In this statement:

- t1 is the left table and t2 is the right table
- join_predicate is the condition to match rows on the left table (t1) with rows on the right table (t2)

The join_predicate could be in the following form:

```
t1.pk = t2.fk
```

or if the common columns of the two table have the same name, you can use the following syntax:

```
USING (common_column);
```

The following describes how the RIGHT JOIN clause works.

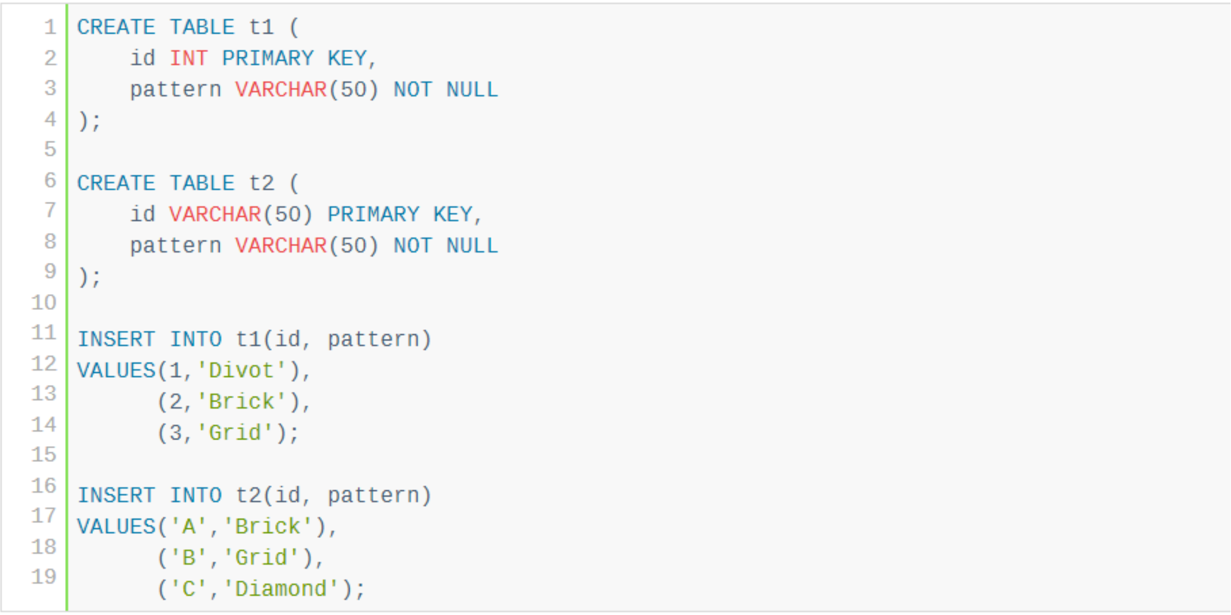
- All rows from the t2 table (right table) will appear at least once in the result set.
- Based on the join_predicate, if no matching row from the t1 table (left table) exists, NULL will appear in columns from the t1 table for the rows that have no match in the t2 table.

It is important to emphasize that RIGHT JOIN and LEFT JOIN clauses are functionally equivalent and they can replace each other as long as the table order is switched.

Note that the RIGHT OUTER JOIN is a synonym for RIGHT JOIN.

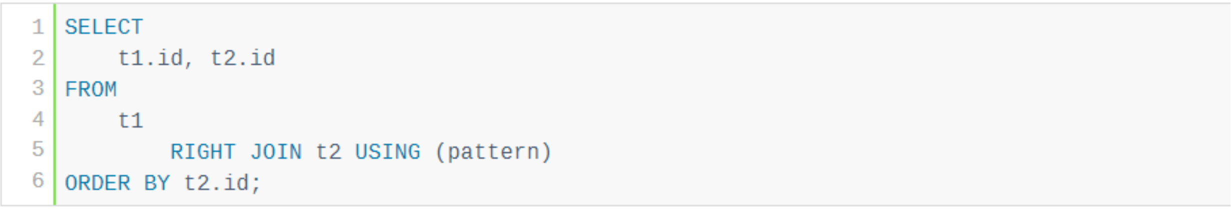
MySQL RIGHT JOIN example

Suppose we have two tables t1 and t2 with the following structures and data:

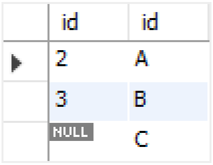


```
CREATE TABLE t1 (
    id INT PRIMARY KEY,
    pattern VARCHAR(50) NOT NULL
);

CREATE TABLE t2 (
    id VARCHAR(50) PRIMARY KEY,
    pattern VARCHAR(50) NOT NULL
);

INSERT INTO t1(id, pattern)
VALUES(1,'Divot'),
      (2,'Brick'),
      (3,'Grid');

INSERT INTO t2(id, pattern)
VALUES('A','Brick'),
      ('B','Grid'),
      ('C','Diamond');
```

The following query joins two tables t1 and t2 using the pattern column:

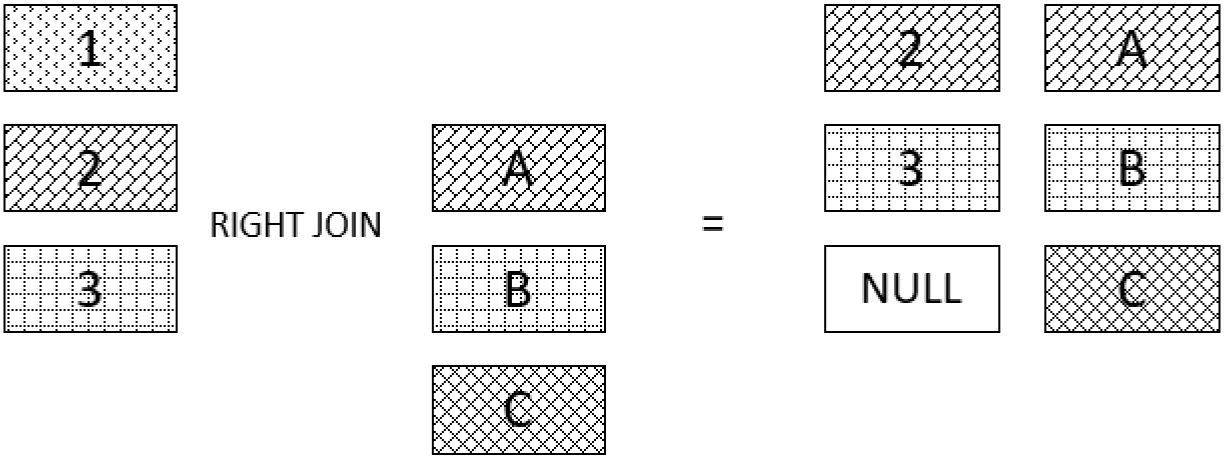


```
SELECT
    t1.id, t2.id
FROM
    t1
        RIGHT JOIN t2 USING (pattern)
ORDER BY t2.id;
```



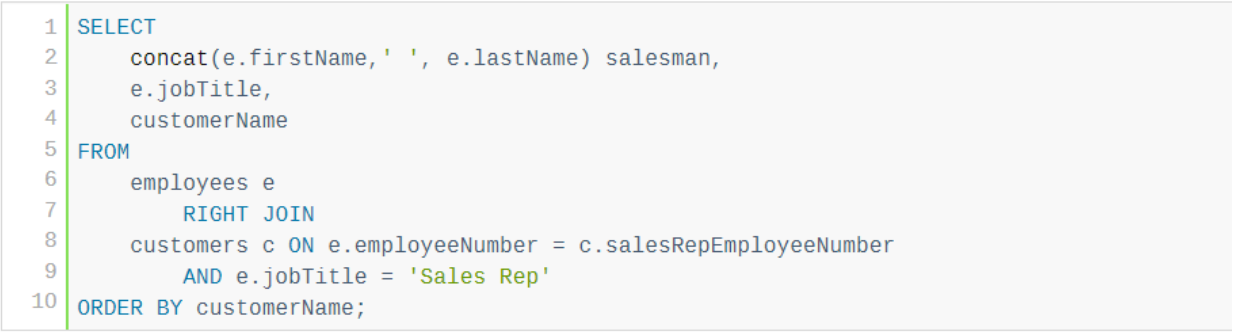
id	id
2	A
3	B
NULL	C

The picture below illustrates the result of the RIGHT JOIN clause:

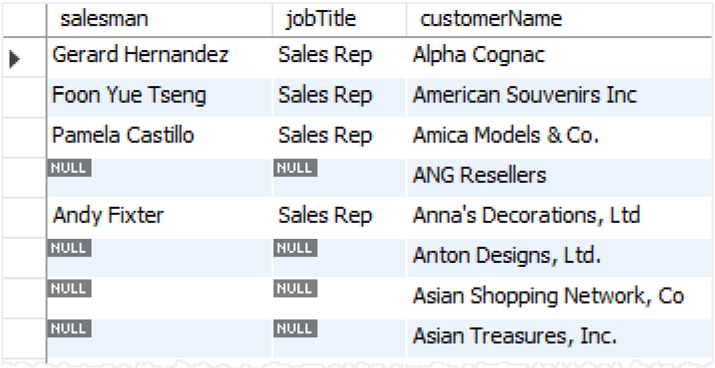


See the following employees and customers in the sample database.

The following query get the sales representatives and their customers:



```
SELECT
    concat(e.firstName,' ', e.lastName) salesman,
    e.jobTitle,
    customerName
FROM
    employees e
        RIGHT JOIN
    customers c ON e.employeeNumber = c.salesRepEmployeeNumber
        AND e.jobTitle = 'Sales Rep'
ORDER BY customerName;
```



salesman	jobTitle	customerName
Gerard Hernandez	Sales Rep	Alpha Cognac
Foon Yue Tseng	Sales Rep	American Souvenirs Inc
Pamela Castillo	Sales Rep	Amica Models & Co.
NULL	NULL	ANG Resellers
Andy Fixter	Sales Rep	Anna's Decorations, Ltd
NULL	NULL	Anton Designs, Ltd.
NULL	NULL	Asian Shopping Network, Co
NULL	NULL	Asian Treasures, Inc.

Because we used RIGHT JOIN, all customers (right table) appears in the result set. We also found that some customers do not have dedicated sales rep indicated by NULL in the salesman column.

In this tutorial, you have learned how to use the MySQL RIGHT JOIN to query data from two or more tables.

MySQL Self Join

In the previous sections, you have learned how to join a table to the other tables using INNER JOIN, LEFT JOIN or CROSS JOIN clause. However, there is a special case that you need join a table to itself, which is known as self join.

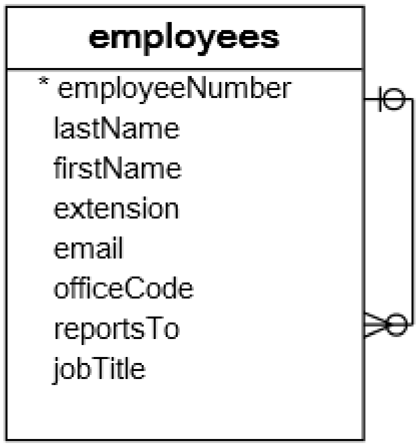
You use the self join when you want to combine rows with other rows in the same table. To perform the self join operation, you must use a table alias to help MySQL distinguish the left table from the right table of the same table in a single query.

MySQL self join examples

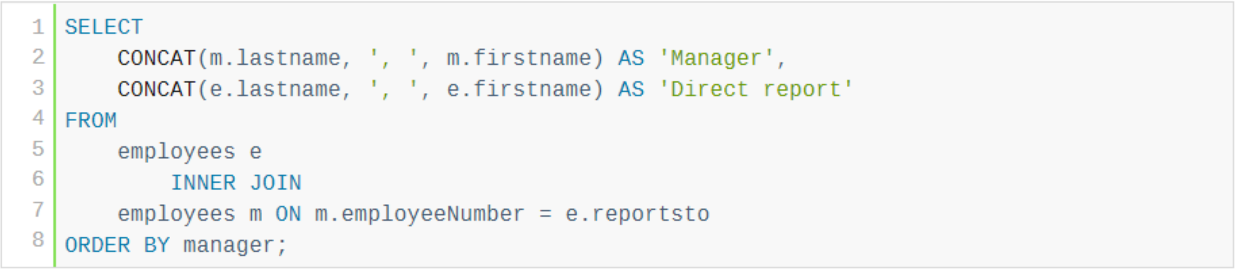
Let’s take a look at the employees table in the sample database.

In the employees table, we store not only employees data but also organization structure data.

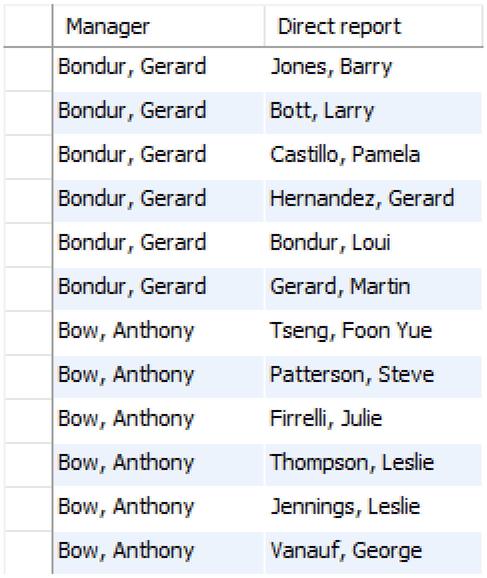
The reportsTo column is used to determine the manager id of an employee.



To get the whole organization structure, you can join the employees table to itself using the employeeNumber and reportsTo columns. The employees table has two roles: one is *Manager* and the other is *Direct Reports.*



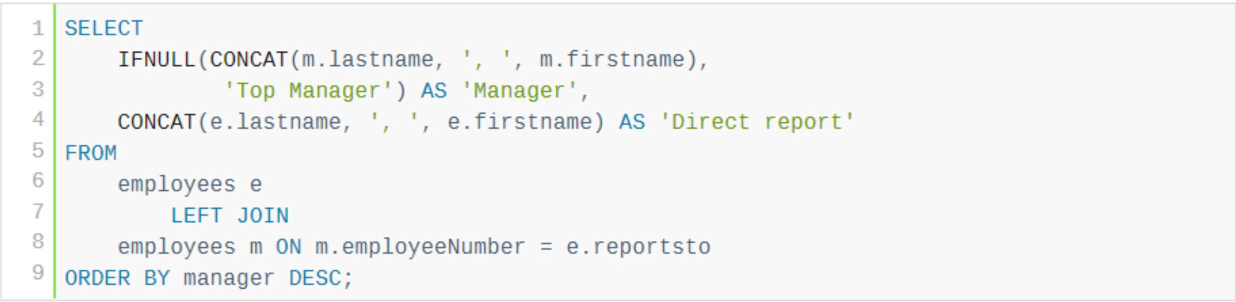
```
SELECT
    CONCAT(m.lastname, ', ', m.firstname) AS 'Manager',
    CONCAT(e.lastname, ', ', e.firstname) AS 'Direct report'
FROM
    employees e
        INNER JOIN
    employees m ON m.employeeNumber = e.reportsto
ORDER BY manager;
```



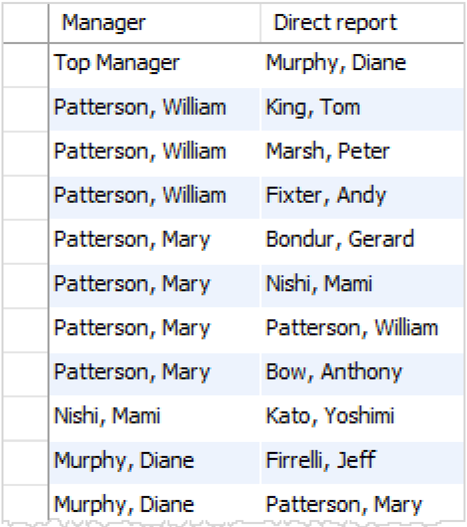
Manager	Direct report
Bondur, Gerard	Jones, Barry
Bondur, Gerard	Bott, Larry
Bondur, Gerard	Castillo, Pamela
Bondur, Gerard	Hernandez, Gerard
Bondur, Gerard	Bondur, Loui
Bondur, Gerard	Gerard, Martin
Bow, Anthony	Tseng, Foon Yue
Bow, Anthony	Patterson, Steve
Bow, Anthony	Firrelli, Julie
Bow, Anthony	Thompson, Leslie
Bow, Anthony	Jennings, Leslie
Bow, Anthony	Vanauf, George

In the above output, you see only employees who have a manager. However, you don’t see the top manager because his name is filtered out due to the INNER JOIN clause. The top manager is the employee who does not have any manager or his manager no is NULL.

Let’s change the INNER JOIN clause to the LEFT JOIN clause in the query above to include the top manager. You also need to use the IFNULL function to display the top manager if the manger’s name is NULL.

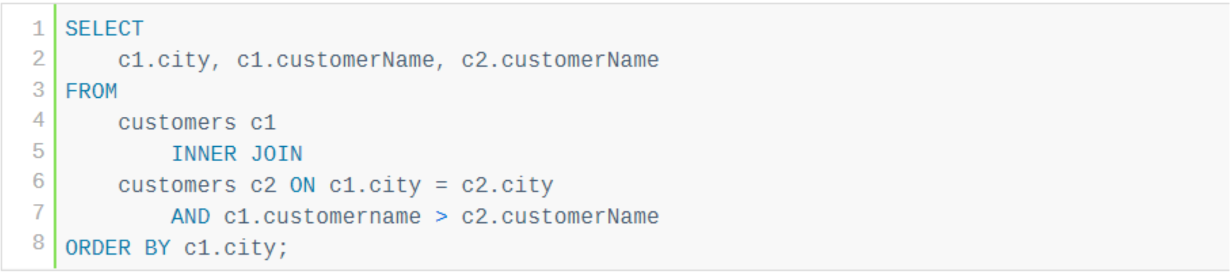


```
SELECT
    IFNULL(CONCAT(m.lastname, ', ', m.firstname),
            'Top Manager') AS 'Manager',
    CONCAT(e.lastname, ', ', e.firstname) AS 'Direct report'
FROM
    employees e
        LEFT JOIN
    employees m ON m.employeeNumber = e.reportsto
ORDER BY manager DESC;
```

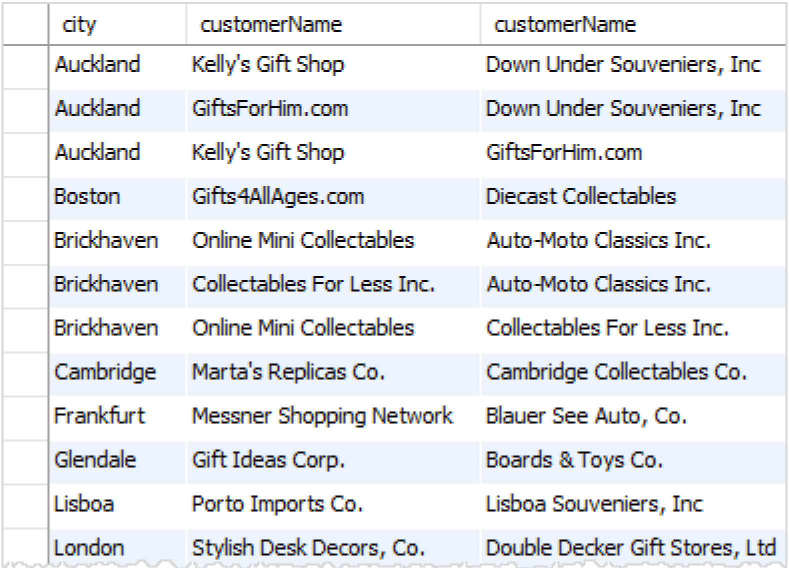


Manager	Direct report
Top Manager	Murphy, Diane
Patterson, William	King, Tom
Patterson, William	Marsh, Peter
Patterson, William	Fixter, Andy
Patterson, Mary	Bondur, Gerard
Patterson, Mary	Nishi, Mami
Patterson, Mary	Patterson, William
Patterson, Mary	Bow, Anthony
Nishi, Mami	Kato, Yoshimi
Murphy, Diane	Firrelli, Jeff
Murphy, Diane	Patterson, Mary

By using the MySQL self join, you can display a list of customers who locate in the same city by joining the customers table to itself.



```
SELECT
    c1.city, c1.customerName, c2.customerName
FROM
    customers c1
        INNER JOIN
    customers c2 ON c1.city = c2.city
        AND c1.customername > c2.customerName
ORDER BY c1.city;
```



city	customerName	customerName
Auckland	Kelly's Gift Shop	Down Under Souveniers, Inc
Auckland	GiftsForHim.com	Down Under Souveniers, Inc
Auckland	Kelly's Gift Shop	GiftsForHim.com
Boston	Gifts4AllAges.com	Diecast Collectables
Brickhaven	Online Mini Collectables	Auto-Moto Classics Inc.
Brickhaven	Collectables For Less Inc.	Auto-Moto Classics Inc.
Brickhaven	Online Mini Collectables	Collectables For Less Inc.
Cambridge	Marta's Replicas Co.	Cambridge Collectables Co.
Frankfurt	Messner Shopping Network	Blauer See Auto, Co.
Glendale	Gift Ideas Corp.	Boards & Toys Co.
Lisboa	Porto Imports Co.	Lisboa Souveniers, Inc
London	Stylish Desk Decors, Co.	Double Decker Gift Stores, Ltd

We joined the customers table to itself with the following join conditions:

- c1.city = c2.city to make sure that both customers have the same city.
- c.customerName > c2.customerName to ensure that we don't get the same customer.

In this tutorial, we have introduced you to MySQL self-join that allows you to join a table to itself by using INNER JOIN or LEFT JOIN clauses.

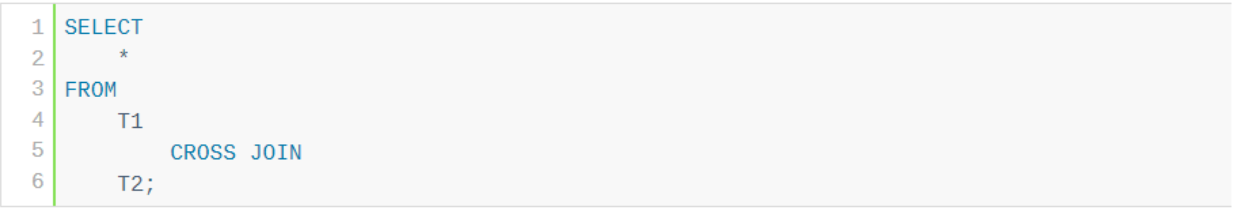
MySQL CROSS JOIN

The CROSS JOIN clause returns the Cartesian product of rows from the joined tables.

Suppose you join two tables using CROSS JOIN. The result set will include all rows from both tables, where each row in the result set is the combination of the row in the first table with the row in the second table. This situation happens when you have no relationship between the joined tables.

The danger thing is that if each table has 1,000 rows, you will get 1,000 x 1,000 = 1,000,000 rows in the result set, which is huge.

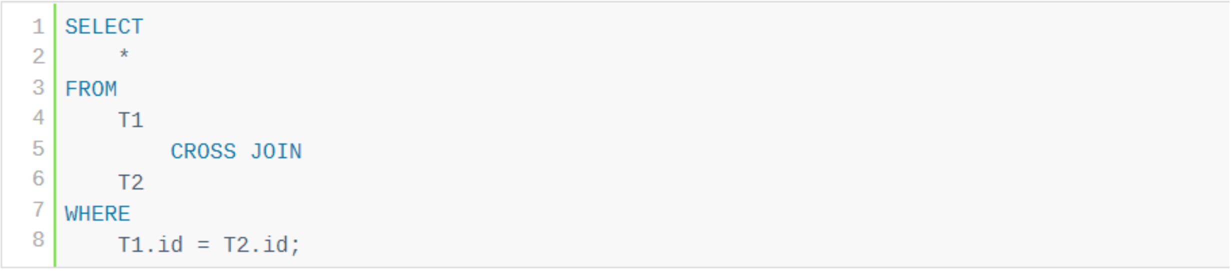
The following illustrates the syntax of the CROSS JOIN clause that joins two tables T1 and T2:



```
SELECT
    *
FROM
    T1
        CROSS JOIN
    T2;
```

Note that different from the INNER JOIN or LEFT JOIN clause, the CROSS JOIN clause does not have the join conditions.

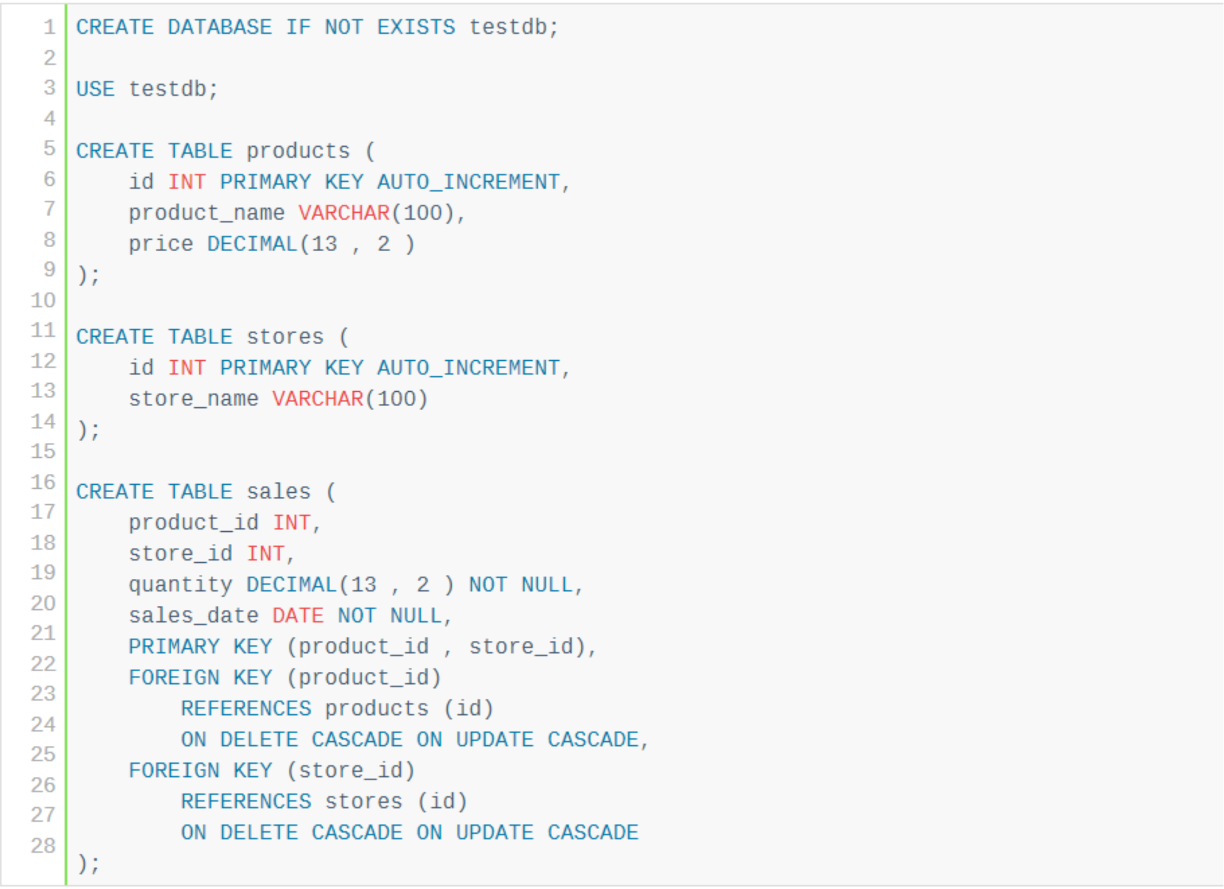
If you add a WHERE clause, in case T1 and T2 has a relationship, the CROSS JOIN works like the INNER JOIN clause as shown in the following query:



```
SELECT
    *
FROM
    T1
        CROSS JOIN
    T2
WHERE
    T1.id = T2.id;
```

MySQL CROSS JOIN clause example

We will use the following testdb database and tables to demonstrate how the CROSS JOIN works.

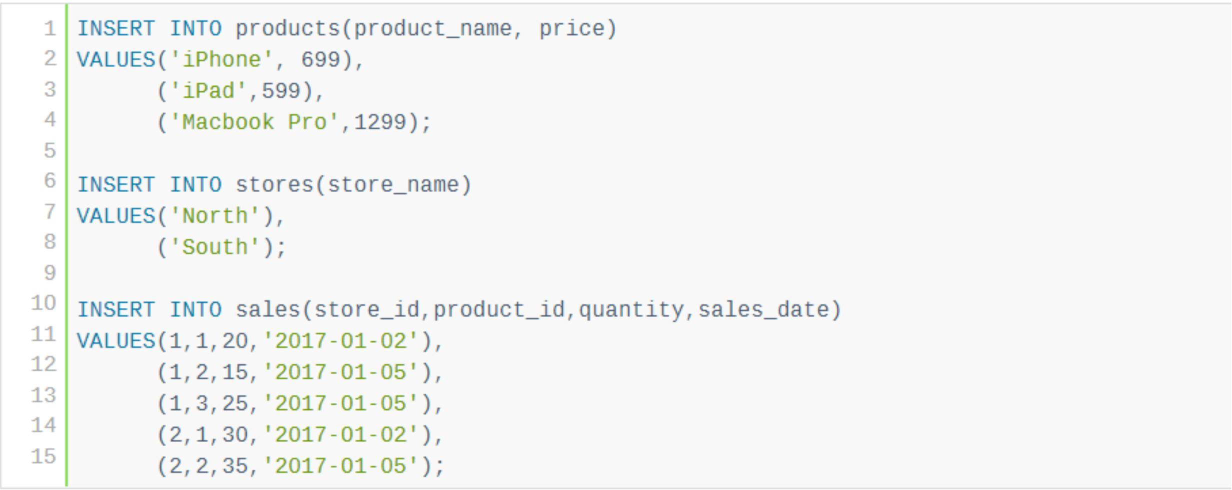


```
CREATE DATABASE IF NOT EXISTS testdb;

USE testdb;

CREATE TABLE products (
    id INT PRIMARY KEY AUTO_INCREMENT,
    product_name VARCHAR(100),
    price DECIMAL(13 , 2 )
);

CREATE TABLE stores (
    id INT PRIMARY KEY AUTO_INCREMENT,
    store_name VARCHAR(100)
);

CREATE TABLE sales (
    product_id INT,
    store_id INT,
    quantity DECIMAL(13 , 2 ) NOT NULL,
    sales_date DATE NOT NULL,
    PRIMARY KEY (product_id , store_id),
    FOREIGN KEY (product_id)
        REFERENCES products (id)
        ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (store_id)
        REFERENCES stores (id)
        ON DELETE CASCADE ON UPDATE CASCADE
);
```

There are three tables involved:

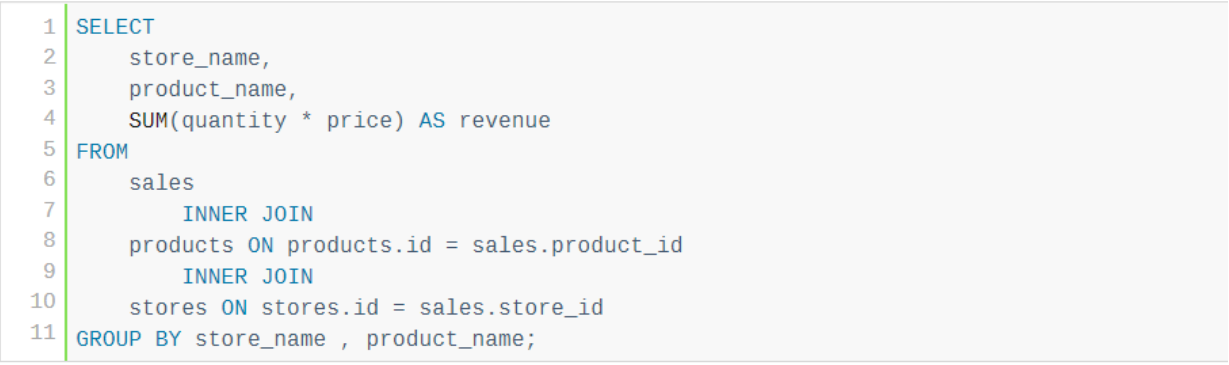
1. The products table contains the products master data that includes product id, product name, and sales price.
2. The stores table contains the stores where the products are sold.
3. The sales table contains the products that sold in a particular store by quantity and date.

Suppose we have three products iPhone, iPad and Macbook Pro which are sold in two stores North and South.

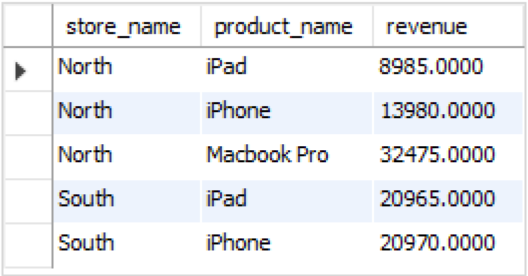


```
INSERT INTO products(product_name, price)
VALUES('iPhone', 699),
      ('iPad',599),
      ('Macbook Pro',1299);

INSERT INTO stores(store_name)
VALUES('North'),
      ('South');

INSERT INTO sales(store_id,product_id,quantity,sales_date)
VALUES(1,1,20,'2017-01-02'),
      (1,2,15,'2017-01-05'),
      (1,3,25,'2017-01-05'),
      (2,1,30,'2017-01-02'),
      (2,2,35,'2017-01-05');
```

To get the total sales for each store and for each product, you calculate the sales and group them by store and product as follows:



```
SELECT
    store_name,
    product_name,
    SUM(quantity * price) AS revenue
FROM
    sales
        INNER JOIN
    products ON products.id = sales.product_id
        INNER JOIN
    stores ON stores.id = sales.store_id
GROUP BY store_name , product_name;
```

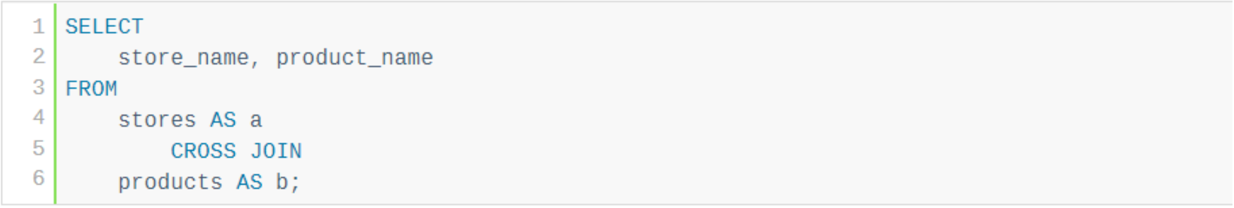


store_name	product_name	revenue
North	iPad	8985.0000
North	iPhone	13980.0000
North	Macbook Pro	32475.0000
South	iPad	20965.0000
South	iPhone	20970.0000

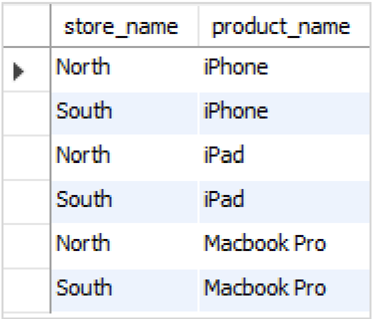
Now, what if you want to know also which store had no sales of a specific product. The query above could not answer this question.

To solve the problem, you need to use the CROSS JOIN clause.

First, you use the CROSS JOIN clause to get the combination of all stores and products:

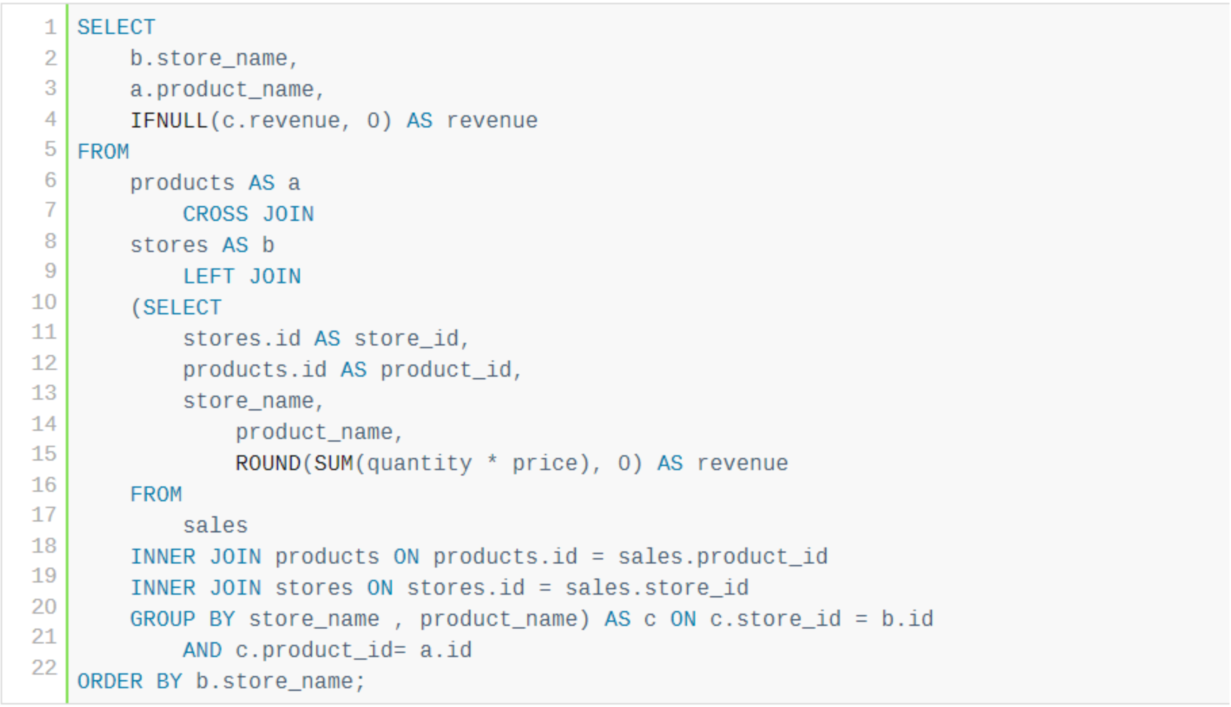


```
SELECT
    store_name, product_name
FROM
    stores AS a
        CROSS JOIN
    products AS b;
```

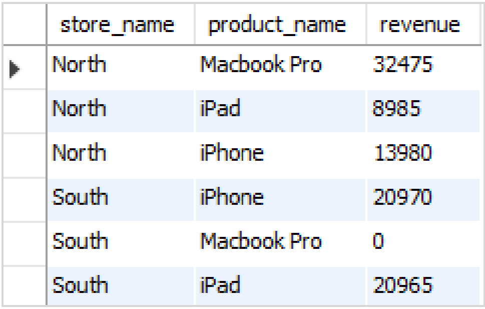


	store_name	product_name
▶	North	iPhone
	South	iPhone
	North	iPad
	South	iPad
	North	Macbook Pro
	South	Macbook Pro

Next, you join the result of the query above with the query that returns the total of sales by store and by product. The following query illustrates the idea:



```
SELECT
    b.store_name,
    a.product_name,
    IFNULL(c.revenue, 0) AS revenue
FROM
    products AS a
        CROSS JOIN
    stores AS b
        LEFT JOIN
    (SELECT
        stores.id AS store_id,
        products.id AS product_id,
        store_name,
            product_name,
            ROUND(SUM(quantity * price), 0) AS revenue
    FROM
        sales
    INNER JOIN products ON products.id = sales.product_id
    INNER JOIN stores ON stores.id = sales.store_id
    GROUP BY store_name , product_name) AS c ON c.store_id = b.id
        AND c.product_id= a.id
ORDER BY b.store_name;
```



store_name	product_name	revenue
North	Macbook Pro	32475
North	iPad	8985
North	iPhone	13980
South	iPhone	20970
South	Macbook Pro	0
South	iPad	20965

Note that the query used the IFNULL function to return 0 if the revenue is NULL (in case the store had no sales).

By using the CROSS JOIN clause this way, you can answer a wide range of questions e.g., find the sales revenue by salesman by month even if a salesman had no sales in a particular month.

MySQL GROUP BY

The GROUP BY clause groups a set of rows into a set of summary rows by values of columns or expressions. The GROUP BY clause returns one row for each group. In other words, it reduces the number of rows in the result set.

You often use the GROUP BY clause with aggregate functions such as SUM, AVG, MAX, MIN, and COUNT. The aggregate function that appears in the SELECT clause provides the information about each group.

The GROUP BY clause is an optional clause of the SELECT statement. The following illustrates the GROUP BY clause syntax:

```
SELECT
    c1, c2,..., cn, aggregate_function(ci)
FROM
    table
WHERE
    where_conditions
GROUP BY c1 , c2,...,cn;
```

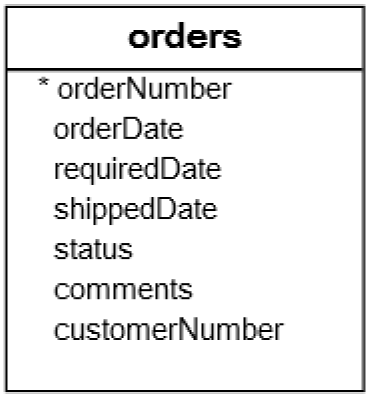
The GROUP BY clause must appear after the FROM and WHERE clauses. Following the GROUP BY keyword is a list of comma-separated columns or expressions that you want to use as criteria to group rows.

MySQL GROUP BY examples

Let’s take some example of using the GROUP BY clause.

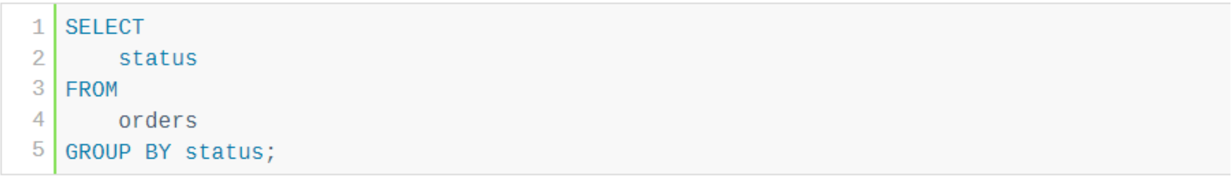
A) Simple MySQL GROUP BY example

Let’s take a look at the orders table in the sample database.

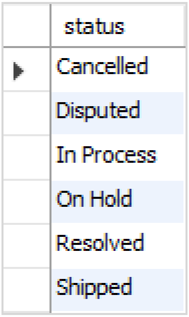


orders
* orderNumber
orderDate
requiredDate
shippedDate
status
comments
customerNumber

Suppose you want to group values of the order’s status into subgroups, you use the GROUP BY clause with the status column as the following query:

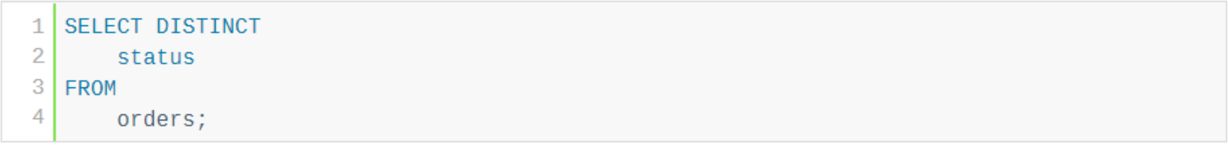


```
SELECT
    status
FROM
    orders
GROUP BY status;
```



status
Cancelled
Disputed
In Process
On Hold
Resolved
Shipped

As you can see, the GROUP BY clause returns unique occurrences of status values. It works like the DISTINCT operator as shown in the following query:

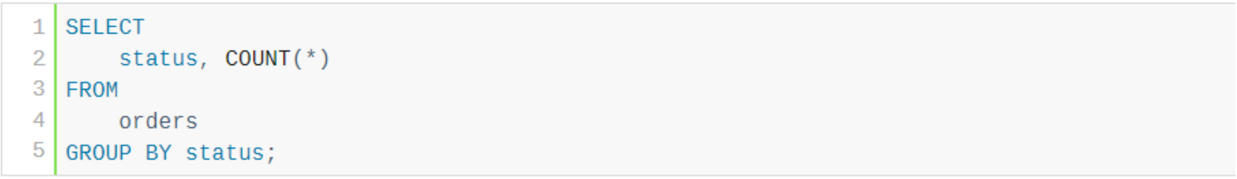


```
SELECT DISTINCT
    status
FROM
    orders;
```

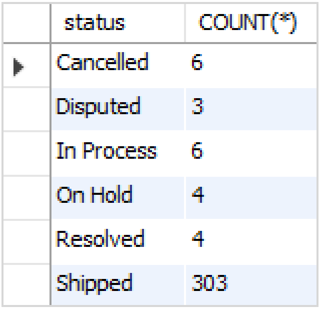
B) Using MySQL GROUP BY with aggregate functions

The aggregate functions allow you to perform the calculation of a set of rows and return a single value. The GROUP BY clause is often used with an aggregate function to perform calculation and return a single value for each subgroup.

For example, if you want to know the number orders in each status, you can use the COUNT function with the GROUP BY clause as follows:

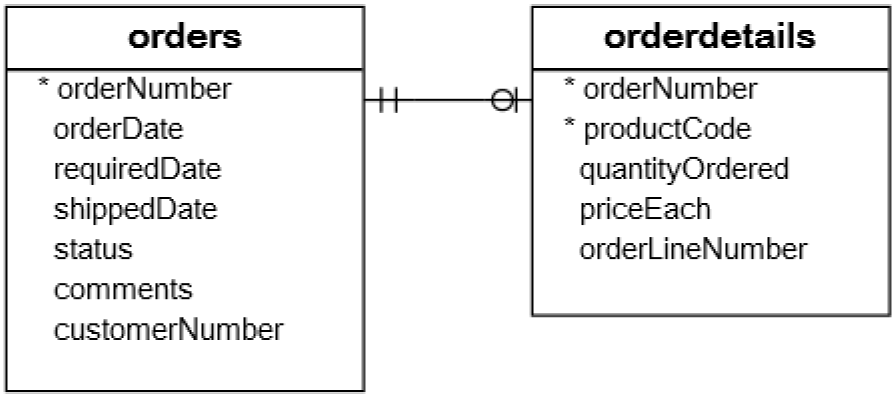


```
SELECT
    status, COUNT(*)
FROM
    orders
GROUP BY status;
```

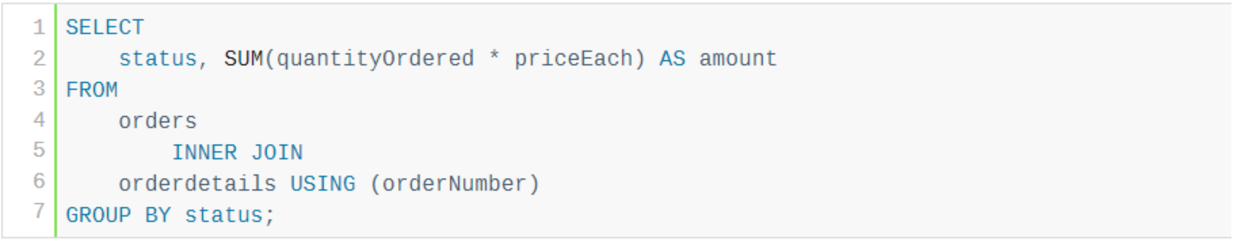


status	COUNT(*)
Cancelled	6
Disputed	3
In Process	6
On Hold	4
Resolved	4
Shipped	303

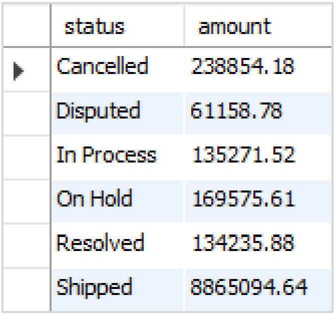
See the following orders and orderdetails table.



To get the total amount of all orders by status, you join the orders table with the orderdetails table and use the SUM function to calculate the total amount. See the following query:

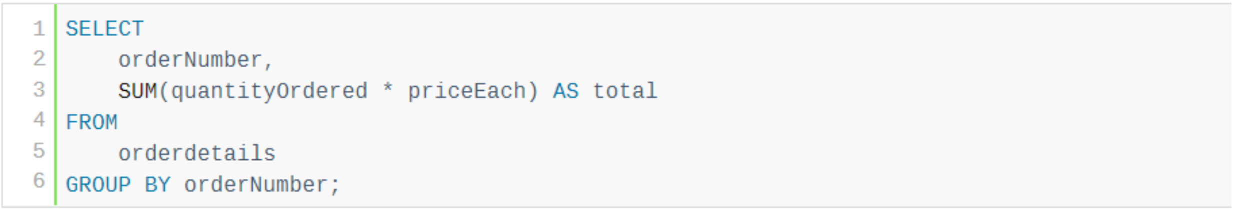


```sql
SELECT
    status, SUM(quantityOrdered * priceEach) AS amount
FROM
    orders
        INNER JOIN
    orderdetails USING (orderNumber)
GROUP BY status;
```

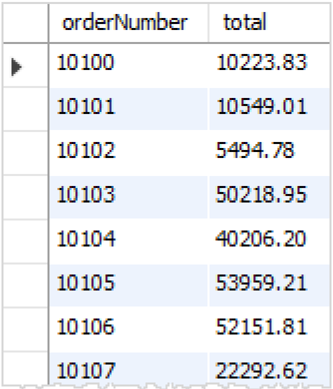


status	amount
Cancelled	238854.18
Disputed	61158.78
In Process	135271.52
On Hold	169575.61
Resolved	134235.88
Shipped	8865094.64

Similarly, the following query returns the order numbers and the total amount of each order.



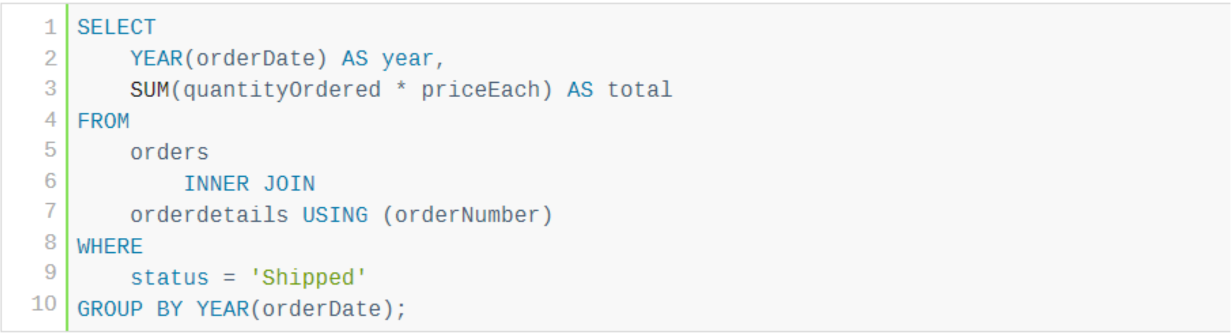
```sql
SELECT
    orderNumber,
    SUM(quantityOrdered * priceEach) AS total
FROM
    orderdetails
GROUP BY orderNumber;
```



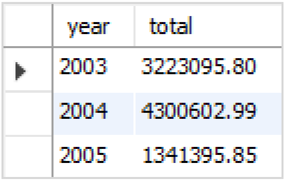
orderNumber	total
10100	10223.83
10101	10549.01
10102	5494.78
10103	50218.95
10104	40206.20
10105	53959.21
10106	52151.81
10107	22292.62

C) MySQL GROUP BY with expression example

In addition to columns, you can group rows by expressions. The following query gets the total sales for each year.



```
SELECT
    YEAR(orderDate) AS year,
    SUM(quantityOrdered * priceEach) AS total
FROM
    orders
        INNER JOIN
    orderdetails USING (orderNumber)
WHERE
    status = 'Shipped'
GROUP BY YEAR(orderDate);
```

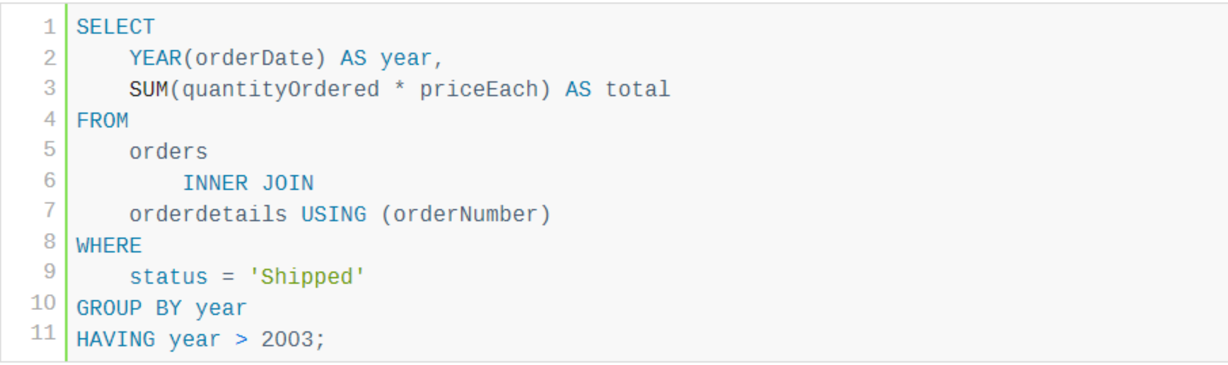


year	total
2003	3223095.80
2004	4300602.99
2005	1341395.85

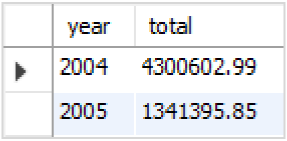
In this example, we used the YEAR function to extract year data from order date (orderDate). We included only orders with shipped status in the total sales. Note that the expression which appears in the SELECT clause must be the same as the one in the GROUP BY clause.

D) Using MySQL GROUP BY with HAVING clause example

To filter the groups returned by GROUP BY clause, you use a HAVING clause. The following query uses the HAVING clause to select the total sales of the years after 2003.



```
SELECT
    YEAR(orderDate) AS year,
    SUM(quantityOrdered * priceEach) AS total
FROM
    orders
        INNER JOIN
    orderdetails USING (orderNumber)
WHERE
    status = 'Shipped'
GROUP BY year
HAVING year > 2003;
```

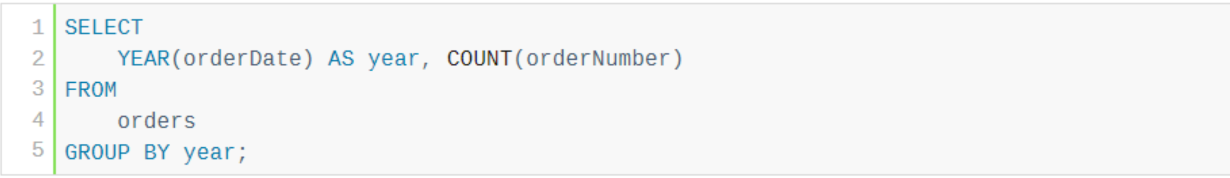


year	total
2004	4300602.99
2005	1341395.85

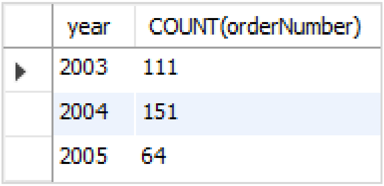
The GROUP BY clause: MySQL vs. standard SQL

Standard SQL does not allow you to use an alias in the GROUP BY clause, however, MySQL supports this.

For example, the following query extracts the year from the order date. It first uses year as an alias of the expression YEAR(orderDate) and then uses the year alias in the GROUP BY clause. This query is not valid in standard SQL.

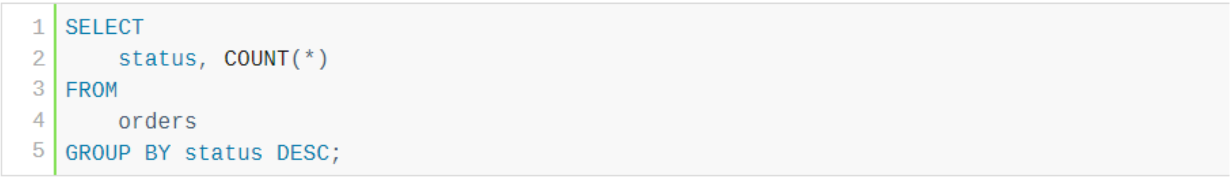


```
SELECT
    YEAR(orderDate) AS year, COUNT(orderNumber)
FROM
    orders
GROUP BY year;
```

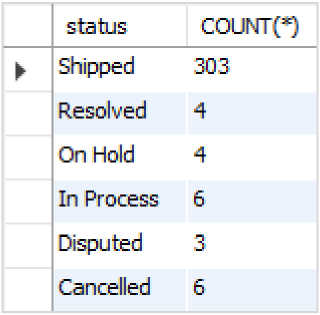


year	COUNT(orderNumber)
2003	111
2004	151
2005	64

MySQL also allows you to sort the groups in ascending or descending orders while the standard SQL does not. The default order is ascending. For example, if you want to get the number of orders by status and sort the status in descending order, you can use the GROUP BY clause with DESC as the following query:



```
SELECT
    status, COUNT(*)
FROM
    orders
GROUP BY status DESC;
```



status	COUNT(*)
Shipped	303
Resolved	4
On Hold	4
In Process	6
Disputed	3
Cancelled	6

Notice that we used DESC in the GROUP BY clause to sort the status in descending order. We could also specify explicitly ASC in the GROUP BY clause to sort the groups by status in ascending order.

In this tutorial, we have shown you how to use the MySQL GROUP BY clause to group rows into subgroups based on values of columns or expressions.

MySQL HAVING

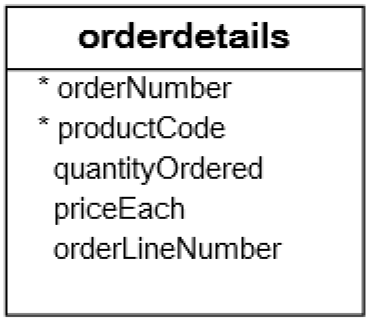
The HAVING clause is used in the SELECT statement to specify filter conditions for a group of rows or aggregates.

The HAVING clause is often used with the GROUP BY clause to filter groups based on a specified condition. If the GROUP BY clause is omitted, the HAVING clause behaves like the WHERE clause.

Notice that the HAVING clause applies a filter condition to each group of rows, while the WHERE clause applies the filter condition to each individual row.

MySQL HAVING clause examples

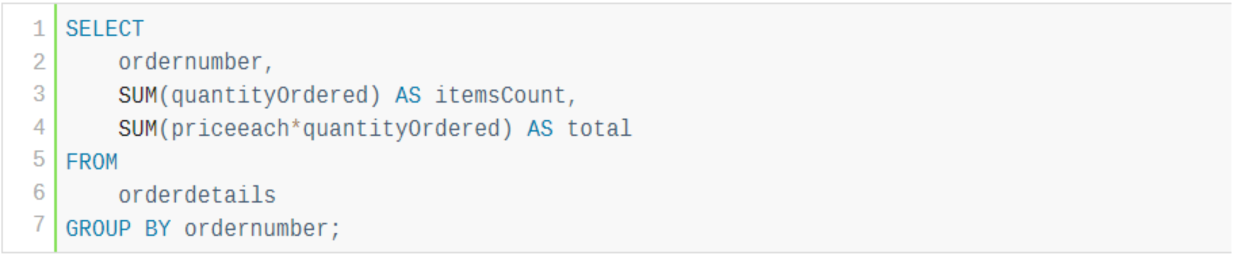
Let’s take some examples of using the HAVING clause to see how it works. We will use the orderdetails table in the sample database for the demonstration.



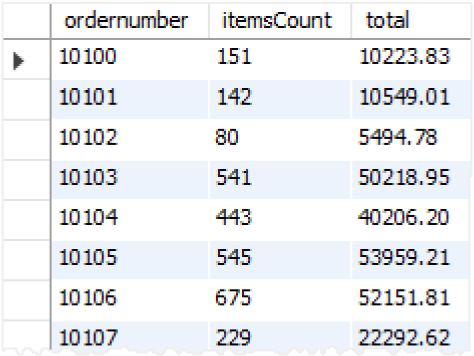
orderdetails

* orderNumber
* productCode
quantityOrdered
priceEach
orderLineNumber

You can use GROUP BY clause to get order numbers, the number of items sold per order, and total sales for each:

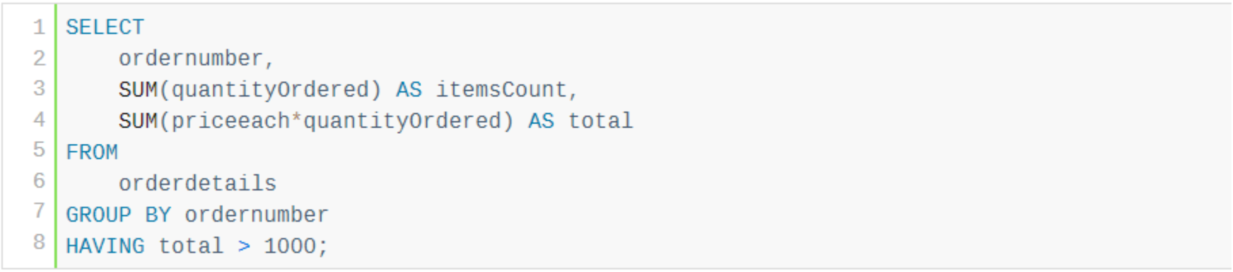


```
SELECT
    ordernumber,
    SUM(quantityOrdered) AS itemsCount,
    SUM(priceeach*quantityOrdered) AS total
FROM
    orderdetails
GROUP BY ordernumber;
```

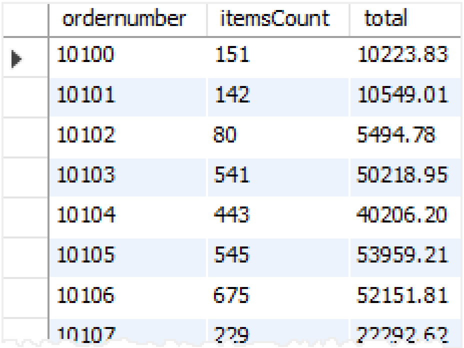


ordernumber	itemsCount	total
10100	151	10223.83
10101	142	10549.01
10102	80	5494.78
10103	541	50218.95
10104	443	40206.20
10105	545	53959.21
10106	675	52151.81
10107	229	22292.62

Now, you can find which order has total sales greater than 1000 by using the HAVING clause as follows:

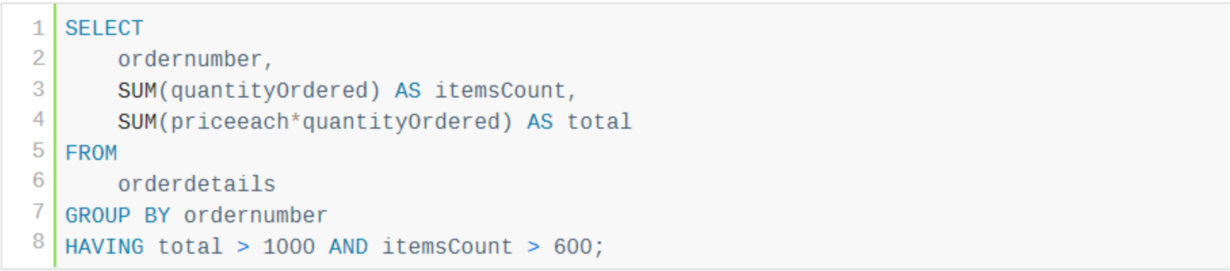


```
SELECT
    ordernumber,
    SUM(quantityOrdered) AS itemsCount,
    SUM(priceeach*quantityOrdered) AS total
FROM
    orderdetails
GROUP BY ordernumber
HAVING total > 1000;
```

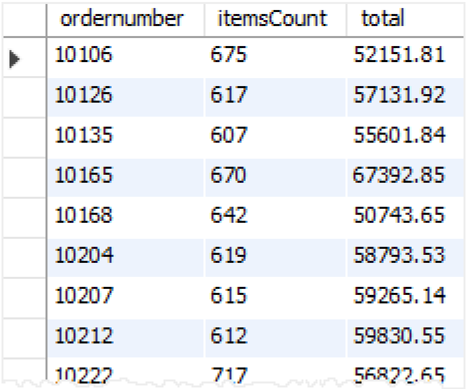


ordernumber	itemsCount	total
10100	151	10223.83
10101	142	10549.01
10102	80	5494.78
10103	541	50218.95
10104	443	40206.20
10105	545	53959.21
10106	675	52151.81
10107	229	22292.62

You can construct a complex condition in the HAVING clause using logical operators such as OR and AND. Suppose you want to find which orders have total sales greater than 1000 and contain more than 600items, you can use the following query:

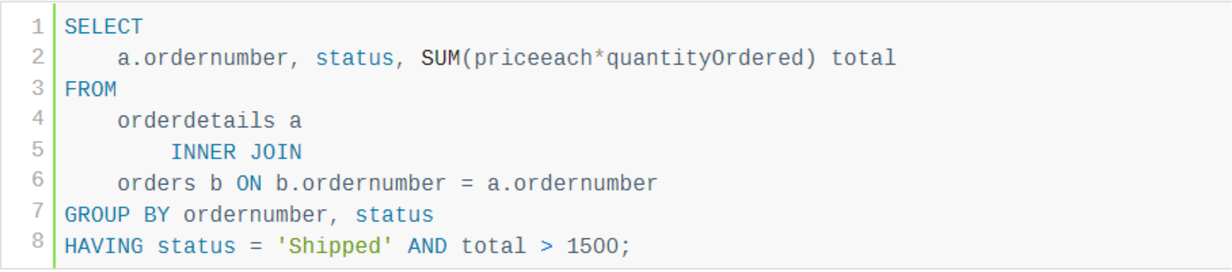


```
SELECT
    ordernumber,
    SUM(quantityOrdered) AS itemsCount,
    SUM(priceeach*quantityOrdered) AS total
FROM
    orderdetails
GROUP BY ordernumber
HAVING total > 1000 AND itemsCount > 600;
```

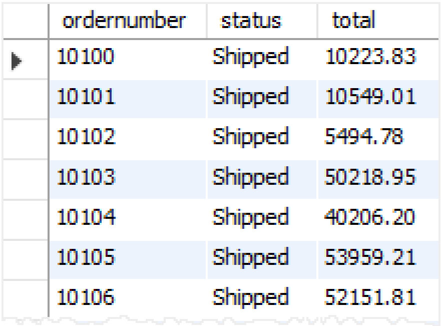


ordernumber	itemsCount	total
10106	675	52151.81
10126	617	57131.92
10135	607	55601.84
10165	670	67392.85
10168	642	50743.65
10204	619	58793.53
10207	615	59265.14
10212	612	59830.55
10222	717	56822.65

Suppose you want to find all orders that have shipped and total sales greater than 1500, you can join the orderdetails table with the orders table using the INNER JOIN clause and apply a condition on status column and total aggregate as shown in the following query:



```
SELECT
    a.ordernumber, status, SUM(priceeach*quantityOrdered) total
FROM
    orderdetails a
        INNER JOIN
    orders b ON b.ordernumber = a.ordernumber
GROUP BY ordernumber, status
HAVING status = 'Shipped' AND total > 1500;
```



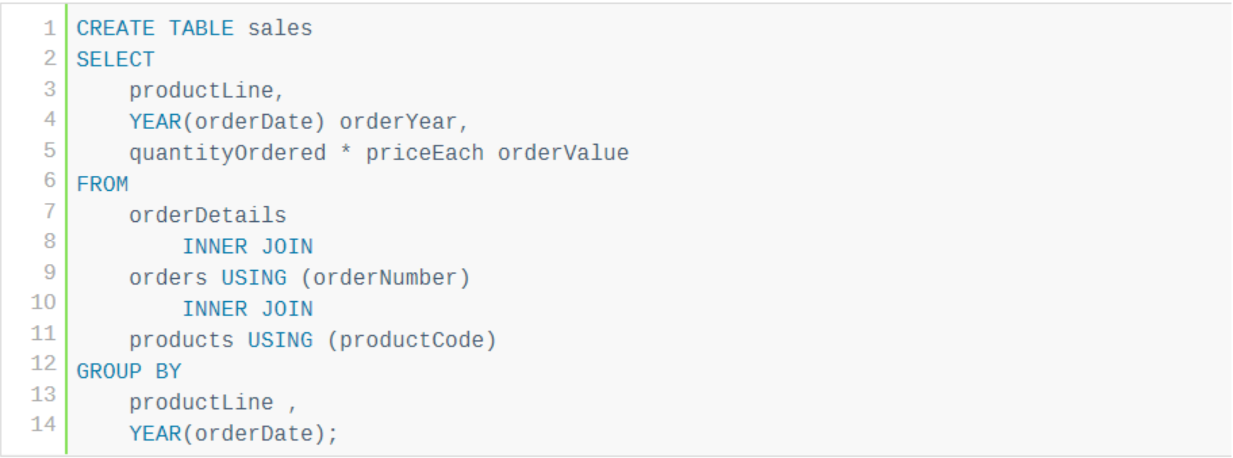
ordernumber	status	total
10100	Shipped	10223.83
10101	Shipped	10549.01
10102	Shipped	5494.78
10103	Shipped	50218.95
10104	Shipped	40206.20
10105	Shipped	53959.21
10106	Shipped	52151.81

The HAVING clause is only useful when you use it with the GROUP BY clause to generate the output of the high-level reports. For example, you can use the HAVING clause to answer statistical questions like finding the number orders this month, this quarter, or this year that have total sales greater than 10K.

In this tutorial, you have learned how to use the MySQL HAVING clause with the GROUP BY clause to specify filter conditions for groups of rows or aggregates.

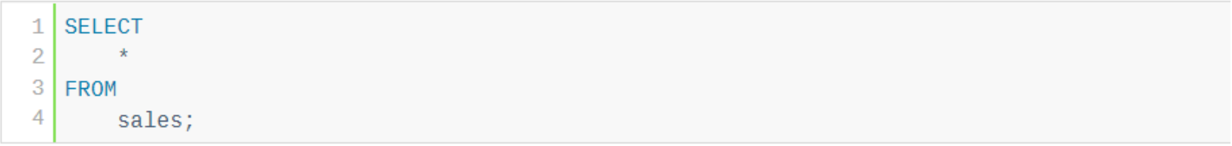
MySQL ROLLUP

The following statement creates a new table named sales that stores the order values summarized by product lines and years. The data comes from the products, orders, and orderDetails tables in the sample database.

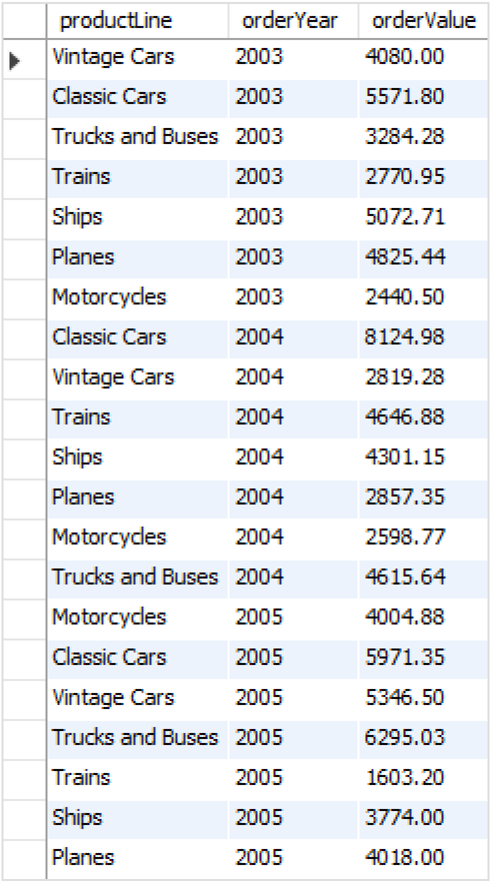


```sql
CREATE TABLE sales
SELECT
    productLine,
    YEAR(orderDate) orderYear,
    quantityOrdered * priceEach orderValue
FROM
    orderDetails
        INNER JOIN
    orders USING (orderNumber)
        INNER JOIN
    products USING (productCode)
GROUP BY
    productLine ,
    YEAR(orderDate);
```

The following query returns all rows from the sales table:



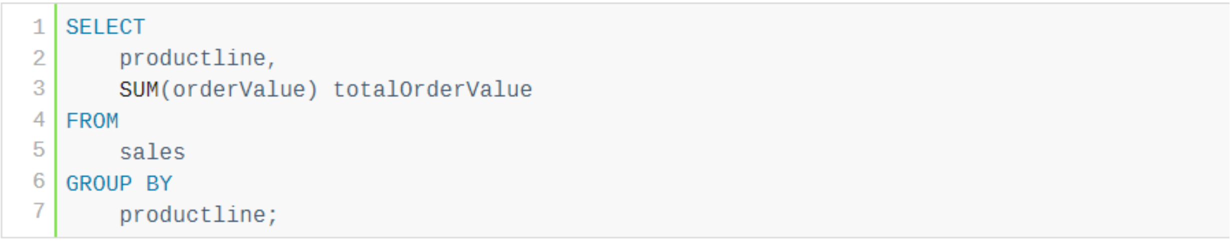
```sql
SELECT
    *
FROM
    sales;
```



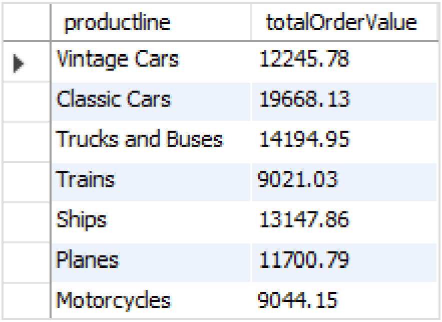
productLine	orderYear	orderValue
Vintage Cars	2003	4080.00
Classic Cars	2003	5571.80
Trucks and Buses	2003	3284.28
Trains	2003	2770.95
Ships	2003	5072.71
Planes	2003	4825.44
Motorcycles	2003	2440.50
Classic Cars	2004	8124.98
Vintage Cars	2004	2819.28
Trains	2004	4646.88
Ships	2004	4301.15
Planes	2004	2857.35
Motorcycles	2004	2598.77
Trucks and Buses	2004	4615.64
Motorcycles	2005	4004.88
Classic Cars	2005	5971.35
Vintage Cars	2005	5346.50
Trucks and Buses	2005	6295.03
Trains	2005	1603.20
Ships	2005	3774.00
Planes	2005	4018.00

MySQL ROLLUP

A grouping set is a set of columns to which you want to group. For example, the following query creates a grouping set denoted by (productline)

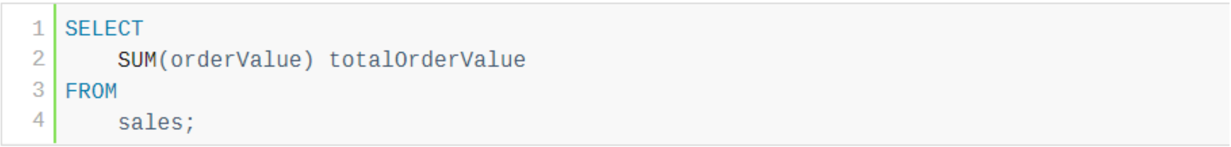


```
SELECT
    productline,
    SUM(orderValue) totalOrderValue
FROM
    sales
GROUP BY
    productline;
```

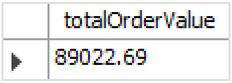


productline	totalOrderValue
Vintage Cars	12245.78
Classic Cars	19668.13
Trucks and Buses	14194.95
Trains	9021.03
Ships	13147.86
Planes	11700.79
Motorcycles	9044.15

The following query creates an empty grouping set denoted by ():

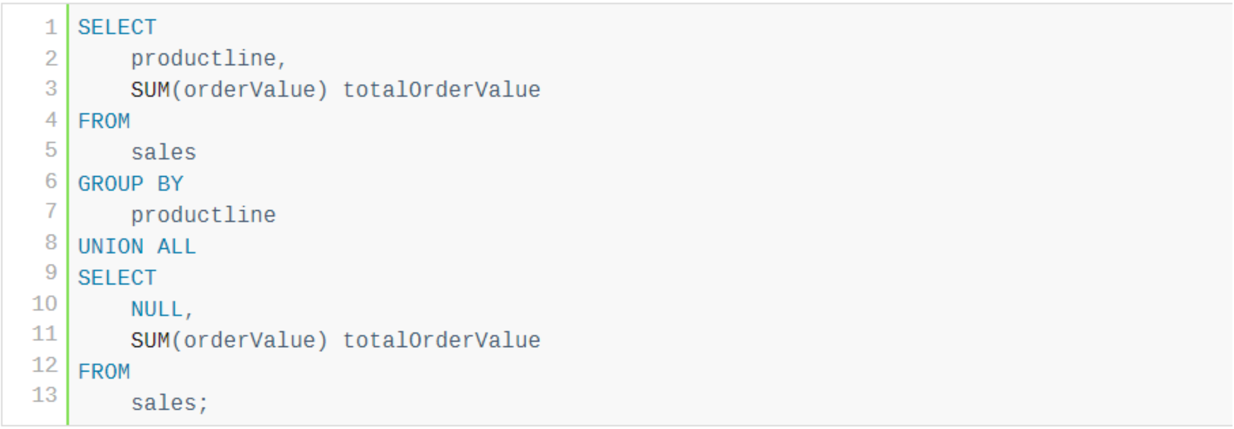


```
SELECT
    SUM(orderValue) totalOrderValue
FROM
    sales;
```



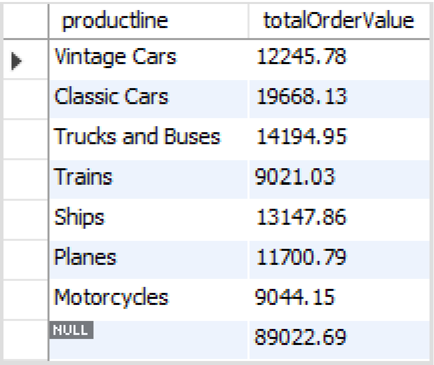
totalOrderValue
89022.69

If you want to generate two or more grouping sets together in one query, you may use the UNION ALL operator as follows:



```
SELECT
    productline,
    SUM(orderValue) totalOrderValue
FROM
    sales
GROUP BY
    productline
UNION ALL
SELECT
    NULL,
    SUM(orderValue) totalOrderValue
FROM
    sales;
```

Here is the query output:



productline	totalOrderValue
Vintage Cars	12245.78
Classic Cars	19668.13
Trucks and Buses	14194.95
Trains	9021.03
Ships	13147.86
Planes	11700.79
Motorcycles	9044.15
NULL	89022.69

Because the UNION ALL requires all queries to have the same number of columns, we added NULL in the select list of the second query to fulfill this requirement.

The NULL in the productLine column identifies the grand total super-aggregate line.

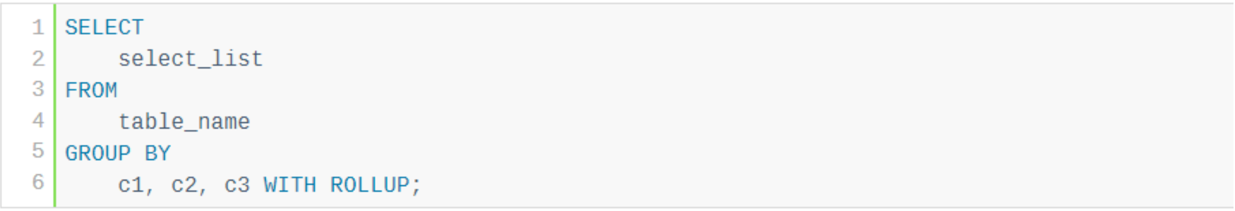
This query is able to generate the total order values by product lines and also the grand total row.

However, it has two problems:

1. The query is quite lengthy.
2. The performance of the query may not be good since the database engine has to internally execute two separate queries and combine the result sets into one.

To solve those issues, you can use the ROLLUP clause.

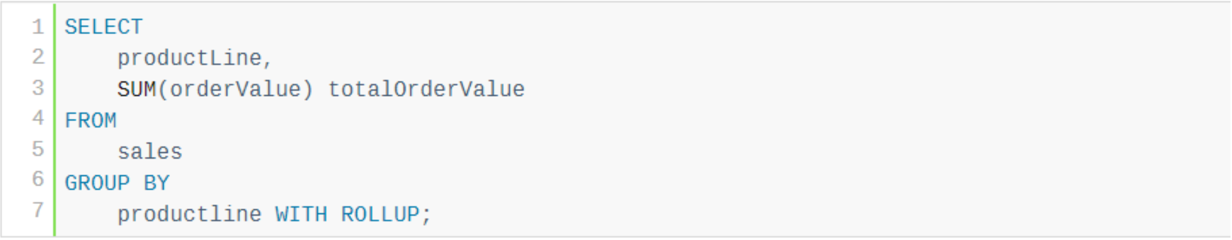
The ROLLUP clause is an extension of the GROUP BY clause with the following syntax:



```
SELECT
    select_list
FROM
    table_name
GROUP BY
    c1, c2, c3 WITH ROLLUP;
```

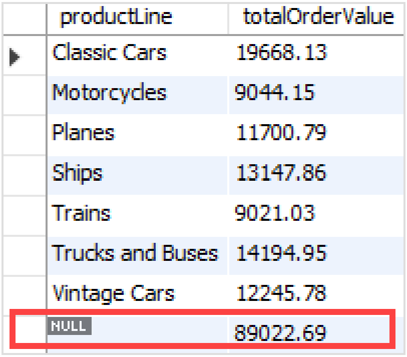
The ROLLUP generates multiple grouping sets based on the columns or expression specified in the GROUP BY clause.

See the following query:



```
SELECT
    productLine,
    SUM(orderValue) totalOrderValue
FROM
    sales
GROUP BY
    productline WITH ROLLUP;
```

Here is the output:

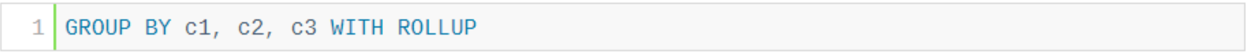


productLine	totalOrderValue
Classic Cars	19668.13
Motorcycles	9044.15
Planes	11700.79
Ships	13147.86
Trains	9021.03
Trucks and Buses	14194.95
Vintage Cars	12245.78
NULL	89022.69

As clearly shown in the output, the ROLLUP clause generates not only the subtotals but also the grand total of the order values.

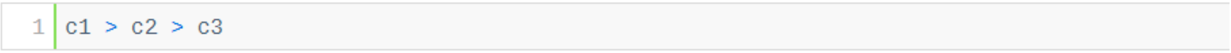
If you have more than one column specified in the GROUP BY clause, the ROLLUP clause assumes a hierarchy among the input columns.

For example:



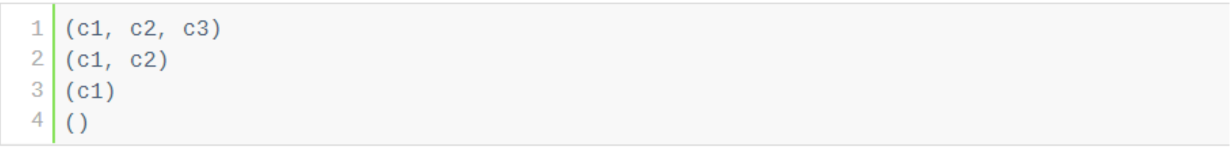
```
GROUP BY c1, c2, c3 WITH ROLLUP
```

The ROLLUP assumes that there is the following hierarchy:



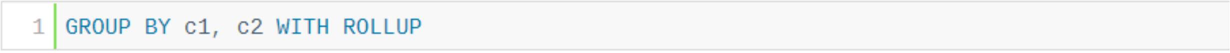
```
c1 > c2 > c3
```

And it generates the following grouping sets:



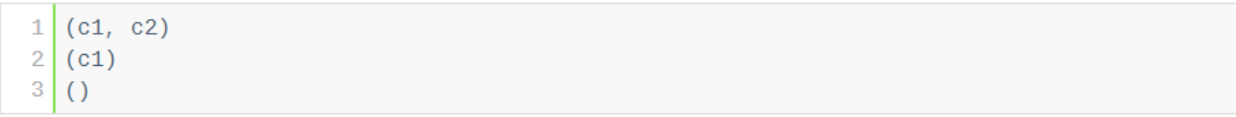
```
(c1, c2, c3)
(c1, c2)
(c1)
()
```

And in case you have two columns specified in the GROUP BY clause:



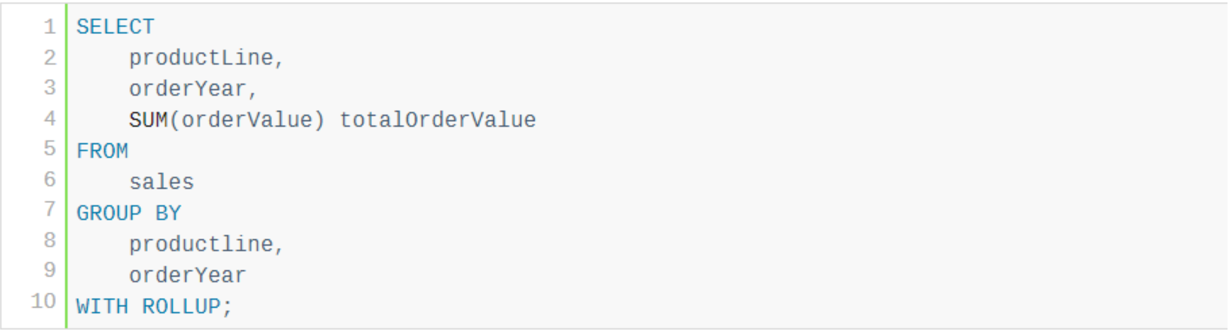
```
GROUP BY c1, c2 WITH ROLLUP
```

then the ROLLUP generates the following grouping sets:

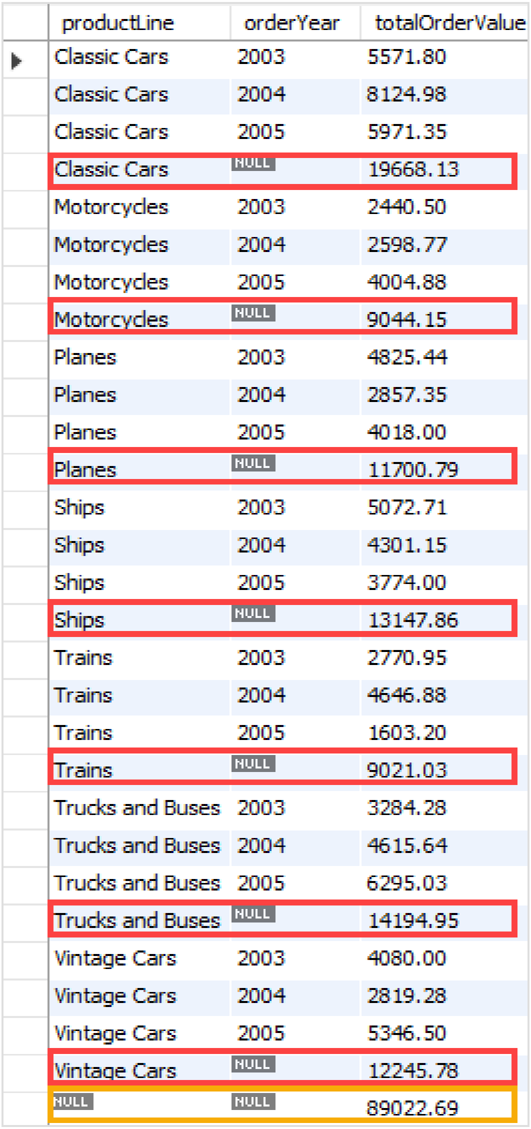


```
(c1, c2)
(c1)
()
```

See the following query example:



```
SELECT
    productLine,
    orderYear,
    SUM(orderValue) totalOrderValue
FROM
    sales
GROUP BY
    productline,
    orderYear
WITH ROLLUP;
```



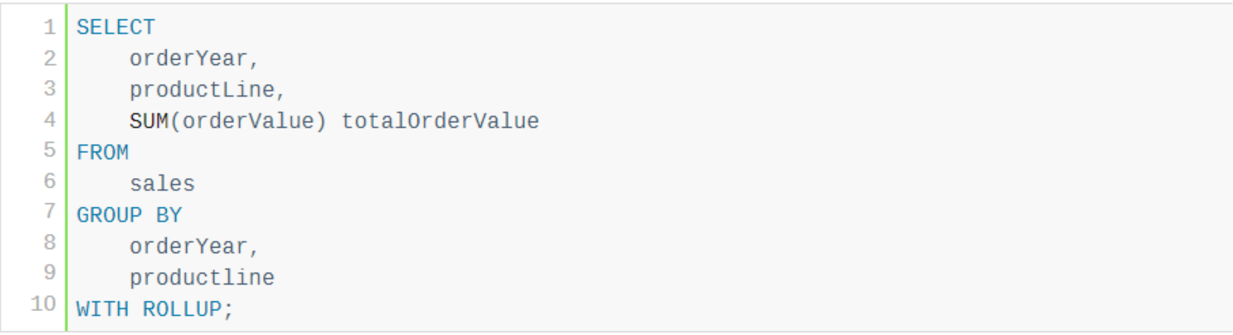
productLine	orderYear	totalOrderValue
Classic Cars	2003	5571.80
Classic Cars	2004	8124.98
Classic Cars	2005	5971.35
Classic Cars	NULL	19668.13
Motorcycles	2003	2440.50
Motorcycles	2004	2598.77
Motorcycles	2005	4004.88
Motorcycles	NULL	9044.15
Planes	2003	4825.44
Planes	2004	2857.35
Planes	2005	4018.00
Planes	NULL	11700.79
Ships	2003	5072.71
Ships	2004	4301.15
Ships	2005	3774.00
Ships	NULL	13147.86
Trains	2003	2770.95
Trains	2004	4646.88
Trains	2005	1603.20
Trains	NULL	9021.03
Trucks and Buses	2003	3284.28
Trucks and Buses	2004	4615.64
Trucks and Buses	2005	6295.03
Trucks and Buses	NULL	14194.95
Vintage Cars	2003	4080.00
Vintage Cars	2004	2819.28
Vintage Cars	2005	5346.50
Vintage Cars	NULL	12245.78
NULL	NULL	89022.69

The ROLLUP generates the subtotal row every time the product line changes and the grand total at the end of the result.

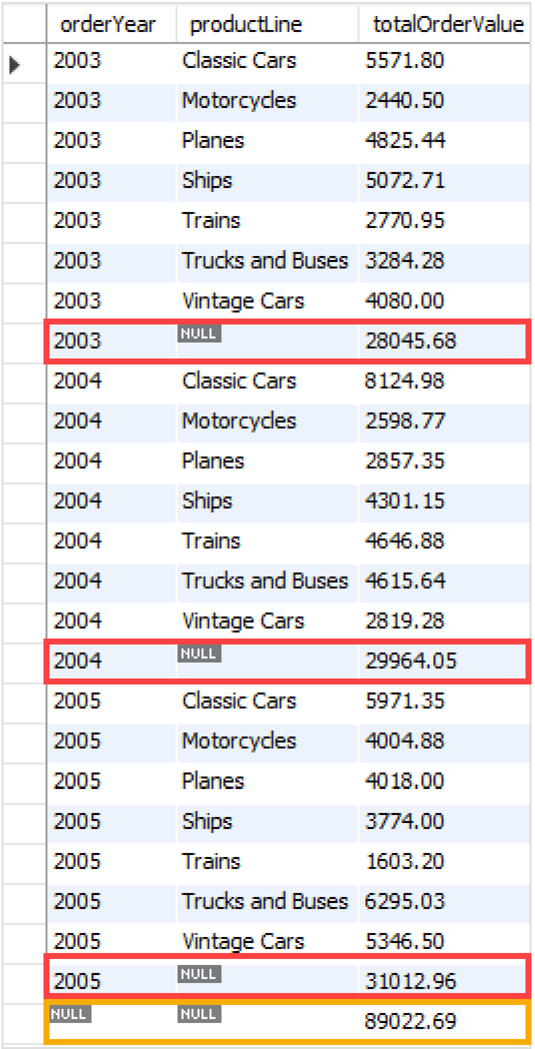
The hierarchy in this case is:

```
productLine > orderYear
```

If you reverse the hierarchy, for example:



```sql
SELECT
    orderYear,
    productLine,
    SUM(orderValue) totalOrderValue
FROM
    sales
GROUP BY
    orderYear,
    productline
WITH ROLLUP;
```



orderYear	productLine	totalOrderValue
2003	Classic Cars	5571.80
2003	Motorcycles	2440.50
2003	Planes	4825.44
2003	Ships	5072.71
2003	Trains	2770.95
2003	Trucks and Buses	3284.28
2003	Vintage Cars	4080.00
2003	NULL	28045.68
2004	Classic Cars	8124.98
2004	Motorcycles	2598.77
2004	Planes	2857.35
2004	Ships	4301.15
2004	Trains	4646.88
2004	Trucks and Buses	4615.64
2004	Vintage Cars	2819.28
2004	NULL	29964.05
2005	Classic Cars	5971.35
2005	Motorcycles	4004.88
2005	Planes	4018.00
2005	Ships	3774.00
2005	Trains	1603.20
2005	Trucks and Buses	6295.03
2005	Vintage Cars	5346.50
2005	NULL	31012.96
NULL	NULL	89022.69

The ROLLUP generates the subtotal every time the year changes and the grand total at the end of the result set.

The hierarchy in this example is:

```
orderYear > productLine
```

GROUPING() function

To check whether NULL in the result set represents the subtotals or grand totals, you use the GROUPING()function.

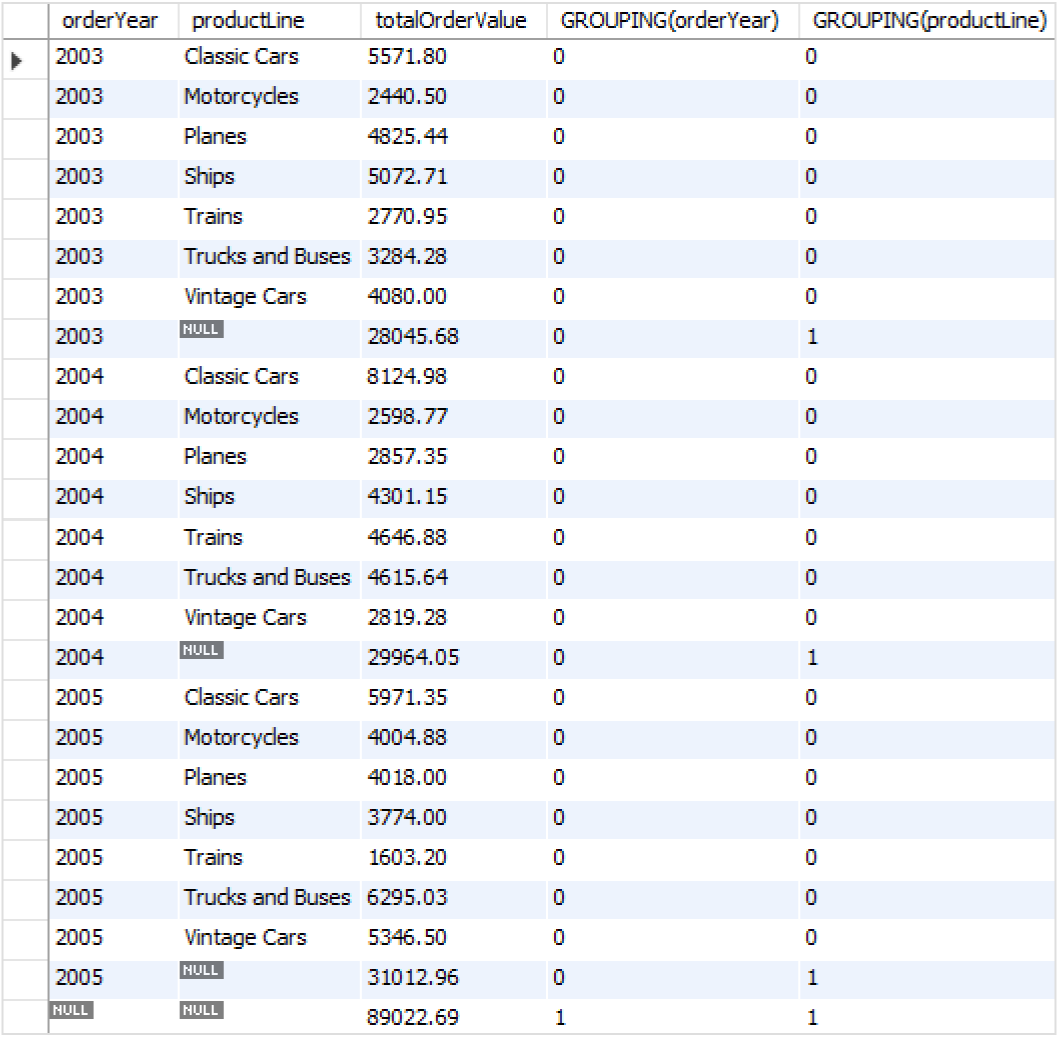
The GROUPING() function returns 1 when NULL occurs in a supper-aggregate row, otherwise, it returns 0.

The GROUPING() function can be used in the select list, HAVING clause, and (as of MySQL 8.0.12) ORDER BY clause.

Consider the following query:

```
SELECT
    orderYear,
    productLine,
    SUM(orderValue) totalOrderValue,
    GROUPING(orderYear),
    GROUPING(productLine)
FROM
    sales
GROUP BY
    orderYear,
    productline
WITH ROLLUP;
```

The following picture shows the output:



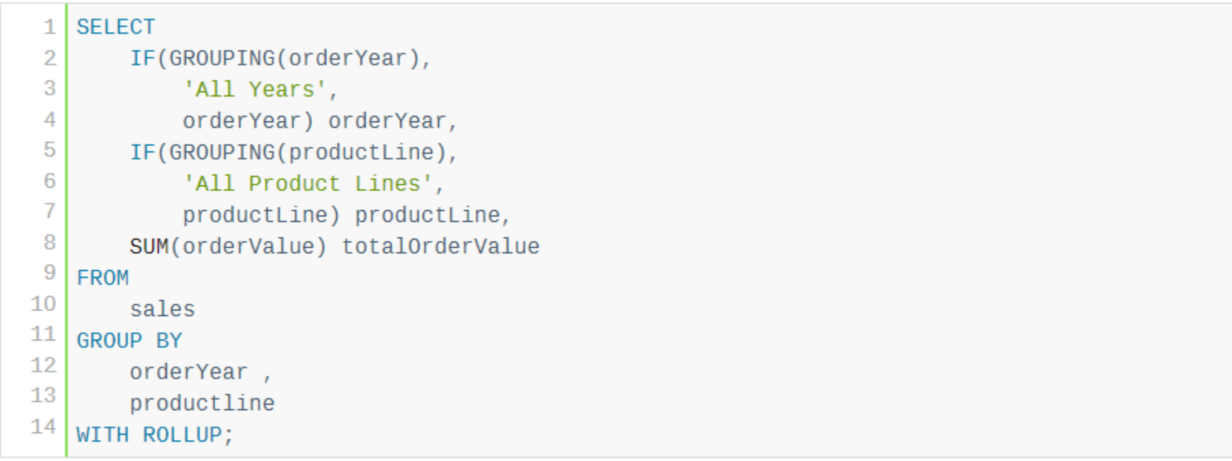
orderYear	productLine	totalOrderValue	GROUPING(orderYear)	GROUPING(productLine)
2003	Classic Cars	5571.80	0	0
2003	Motorcycles	2440.50	0	0
2003	Planes	4825.44	0	0
2003	Ships	5072.71	0	0
2003	Trains	2770.95	0	0
2003	Trucks and Buses	3284.28	0	0
2003	Vintage Cars	4080.00	0	0
2003	NULL	28045.68	0	1
2004	Classic Cars	8124.98	0	0
2004	Motorcycles	2598.77	0	0
2004	Planes	2857.35	0	0
2004	Ships	4301.15	0	0
2004	Trains	4646.88	0	0
2004	Trucks and Buses	4615.64	0	0
2004	Vintage Cars	2819.28	0	0
2004	NULL	29964.05	0	1
2005	Classic Cars	5971.35	0	0
2005	Motorcycles	4004.88	0	0
2005	Planes	4018.00	0	0
2005	Ships	3774.00	0	0
2005	Trains	1603.20	0	0
2005	Trucks and Buses	6295.03	0	0
2005	Vintage Cars	5346.50	0	0
2005	NULL	31012.96	0	1
NULL	NULL	89022.69	1	1

The GROUPING(orderYear) returns 1 when NULL in the orderYear column occurs in a super-aggregate row, 0 otherwise.

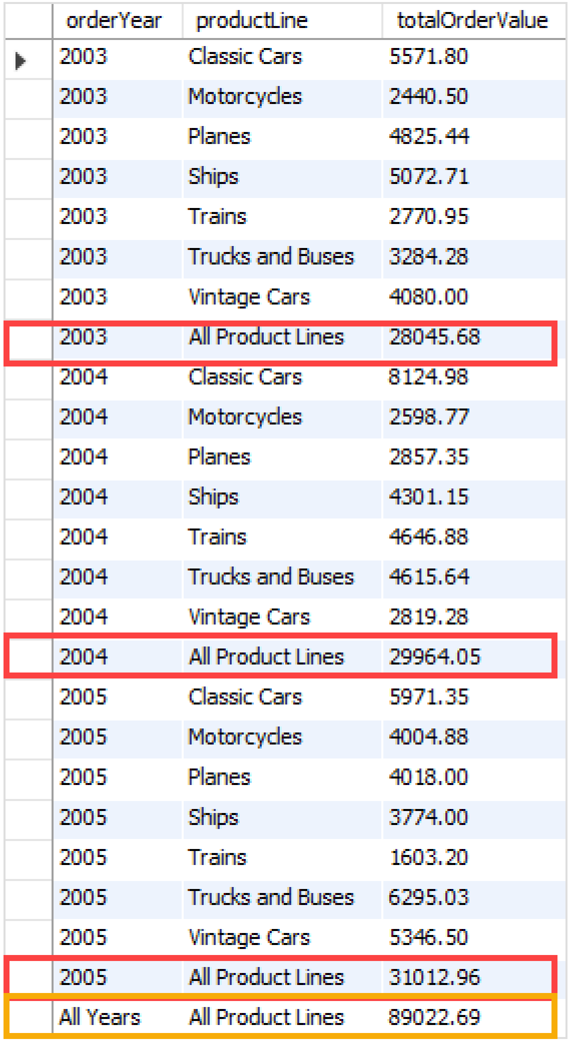
Similarly, the GROUPING(productLine) returns 1 when NULL in the productLine column occurs in a super-aggregate row, 0 otherwise.

We often use GROUPING() function to substitute meaningful labels for super-aggregate NULL values instead of displaying it directly.

The following example shows how to combine the IF() function with the GROUPING() function to substitute labels for the super-aggregate NULL values in orderYear and productLine columns:



```
SELECT
    IF(GROUPING(orderYear),
        'All Years',
        orderYear) orderYear,
    IF(GROUPING(productLine),
        'All Product Lines',
        productLine) productLine,
    SUM(orderValue) totalOrderValue
FROM
    sales
GROUP BY
    orderYear ,
    productline
WITH ROLLUP;
```



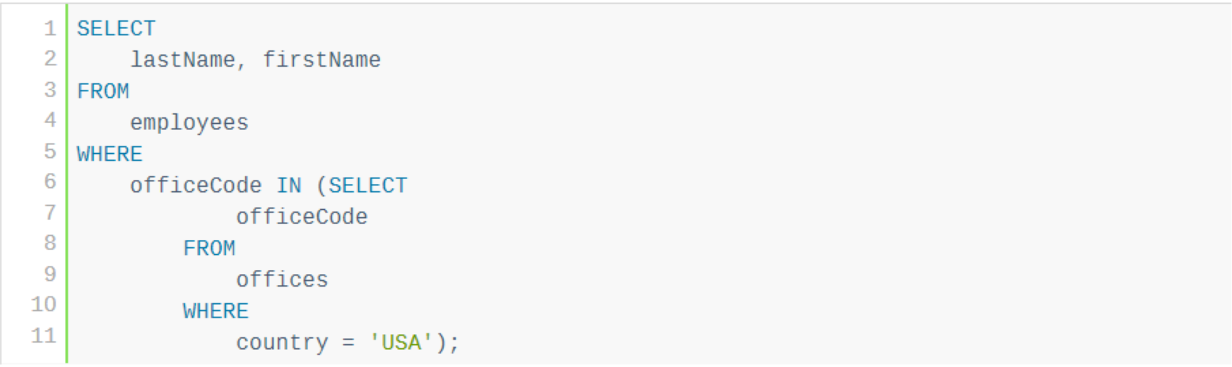
orderYear	productLine	totalOrderValue
2003	Classic Cars	5571.80
2003	Motorcycles	2440.50
2003	Planes	4825.44
2003	Ships	5072.71
2003	Trains	2770.95
2003	Trucks and Buses	3284.28
2003	Vintage Cars	4080.00
2003	All Product Lines	28045.68
2004	Classic Cars	8124.98
2004	Motorcycles	2598.77
2004	Planes	2857.35
2004	Ships	4301.15
2004	Trains	4646.88
2004	Trucks and Buses	4615.64
2004	Vintage Cars	2819.28
2004	All Product Lines	29964.05
2005	Classic Cars	5971.35
2005	Motorcycles	4004.88
2005	Planes	4018.00
2005	Ships	3774.00
2005	Trains	1603.20
2005	Trucks and Buses	6295.03
2005	Vintage Cars	5346.50
2005	All Product Lines	31012.96
All Years	All Product Lines	89022.69

MySQL Subquery

A MySQL subquery is a query nested within another query such as SELECT, INSERT, UPDATE or DELETE. In addition, a MySQL subquery can be nested inside another subquery.

A MySQL subquery is called an inner query while the query that contains the subquery is called an outer query. A subquery can be used anywhere that expression is used and must be closed in parentheses.

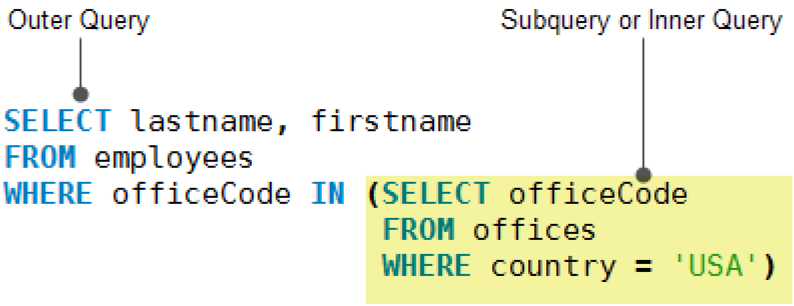
The following query returns employees who work in the offices located in the USA.



```
SELECT
    lastName, firstName
FROM
    employees
WHERE
    officeCode IN (SELECT
            officeCode
        FROM
            offices
        WHERE
            country = 'USA');
```

In this example:

- The subquery returns all *office codes* of the offices located in the USA.
- The outer query selects the last name and first name of employees who work in the offices whose office codes are in the result set returned by the subquery.

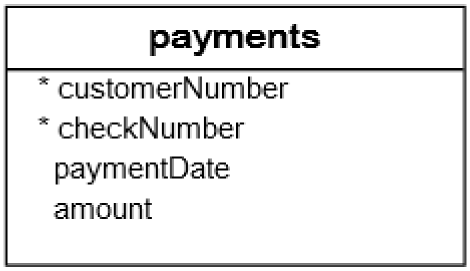


```
SELECT lastname, firstname
FROM employees
WHERE officeCode IN (SELECT officeCode
                     FROM offices
                     WHERE country = 'USA')
```

When the query is executed, the subquery runs first and returns a result set. Then, this result set is used as an input of the outer query.

MySQL subquery in WHERE clause

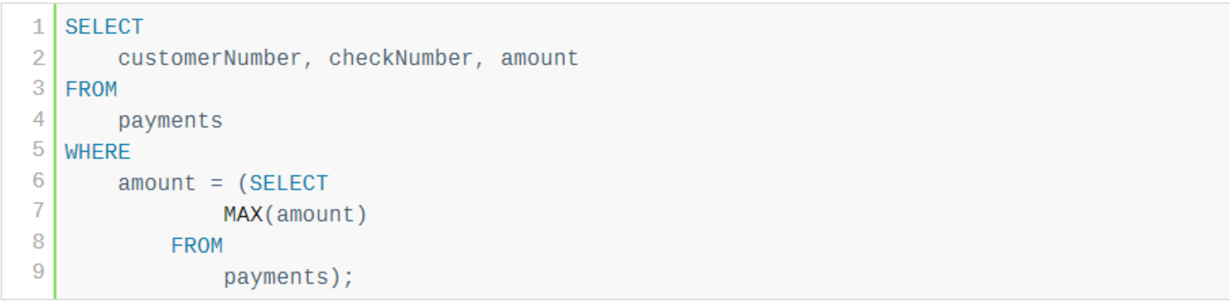
We will use the payments table in the sample database for the demonstration.



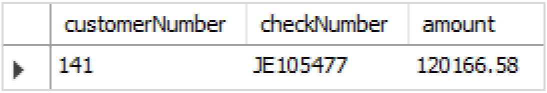
MySQL subquery with comparison operators

You can use comparison operators e.g., =, >, <, etc., to compare a single value returned by the subquery with the expression in the WHERE clause.

For example, the following query returns the customer who has the maximum payment.



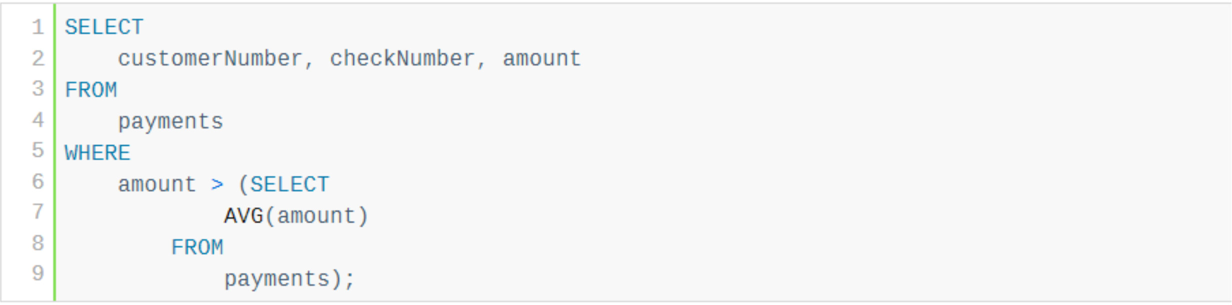
```
SELECT
    customerNumber, checkNumber, amount
FROM
    payments
WHERE
    amount = (SELECT
            MAX(amount)
        FROM
            payments);
```



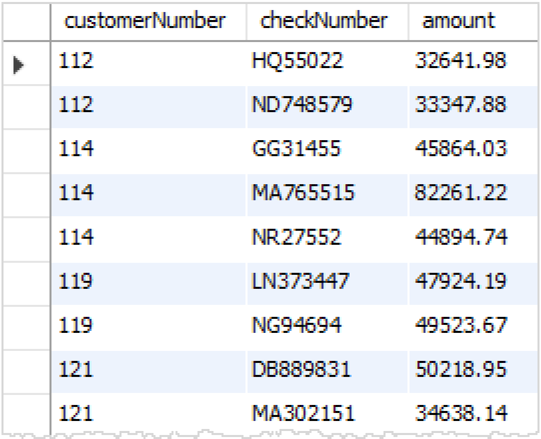
	customerNumber	checkNumber	amount
▶	141	JE105477	120166.58

In addition to the equality operator, you can use other comparison operators such as greater than (>), less than(<), etc.

For example, you can find customers whose payments are greater than the average payment using a subquery. First, use a subquery to calculate the average payment using the AVG aggregate function. Then, in the outer query, query the payments that are greater than the average payment returned by the subquery.



```
SELECT
    customerNumber, checkNumber, amount
FROM
    payments
WHERE
    amount > (SELECT
            AVG(amount)
        FROM
            payments);
```

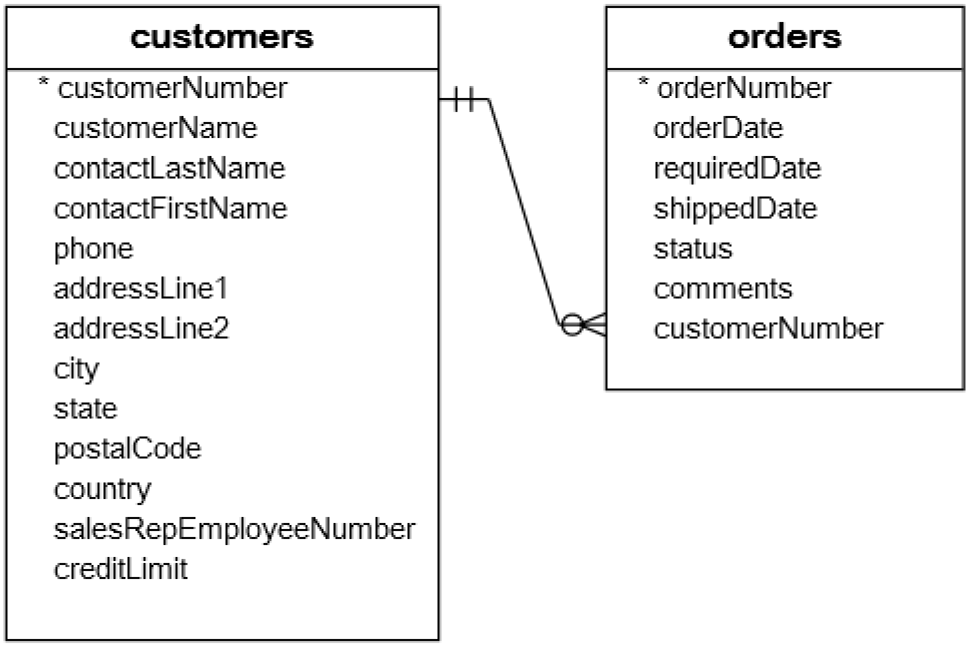


customerNumber	checkNumber	amount
112	HQ55022	32641.98
112	ND748579	33347.88
114	GG31455	45864.03
114	MA765515	82261.22
114	NR27552	44894.74
119	LN373447	47924.19
119	NG94694	49523.67
121	DB889831	50218.95
121	MA302151	34638.14

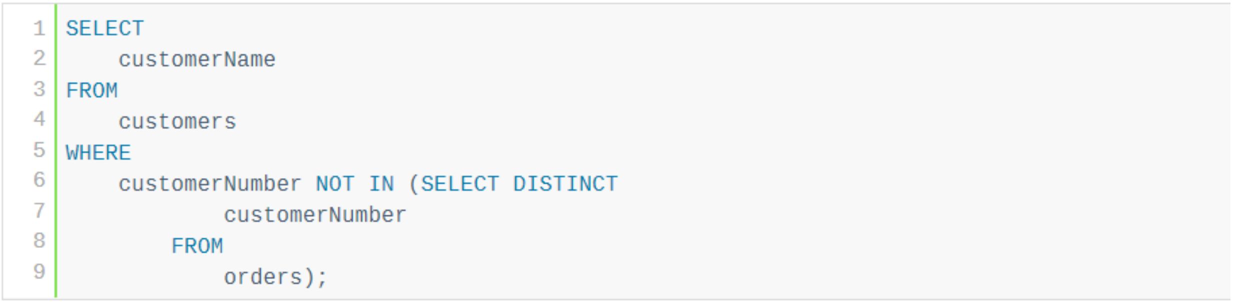
MySQL subquery with IN and NOT IN operators

If a subquery returns more than one value, you can use other operators such as IN or NOT IN operator in the WHERE clause.

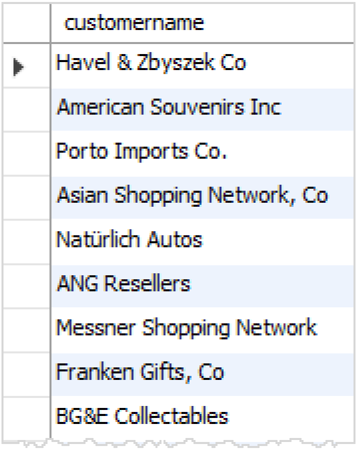
See the following customers and orders tables:



For example, you can use a subquery with NOT IN operator to find the customers who have not placed any orders as follows:



```
SELECT
    customerName
FROM
    customers
WHERE
    customerNumber NOT IN (SELECT DISTINCT
            customerNumber
        FROM
            orders);
```

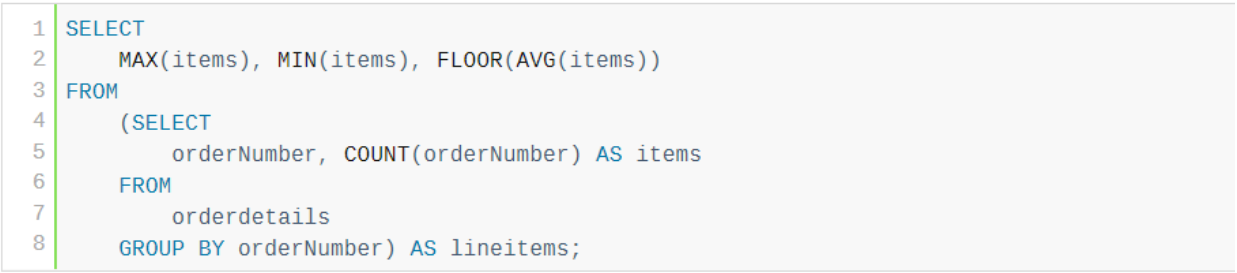


customername
Havel & Zbyszek Co
American Souvenirs Inc
Porto Imports Co.
Asian Shopping Network, Co
Natürlich Autos
ANG Resellers
Messner Shopping Network
Franken Gifts, Co
BG&E Collectables

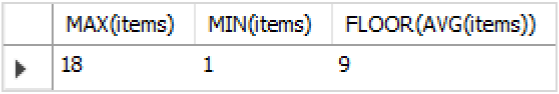
MySQL subquery in the FROM clause

When you use a subquery in the FROM clause, the result set returned from a subquery is used as a temporary table. This table is referred to as a derived table or materialized subquery.

The following subquery finds the maximum, minimum and average number of items in sale orders:



```
SELECT
    MAX(items), MIN(items), FLOOR(AVG(items))
FROM
    (SELECT
        orderNumber, COUNT(orderNumber) AS items
    FROM
        orderdetails
    GROUP BY orderNumber) AS lineitems;
```

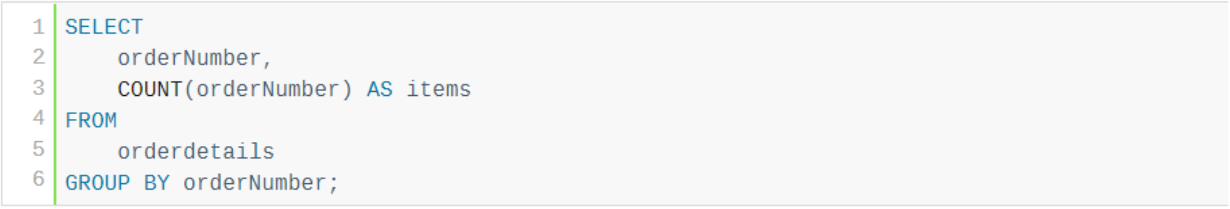


MAX(items)	MIN(items)	FLOOR(AVG(items))
18	1	9

Note that the FLOOR() is used to remove decimal places from the average values of items.

MySQL correlated subquery

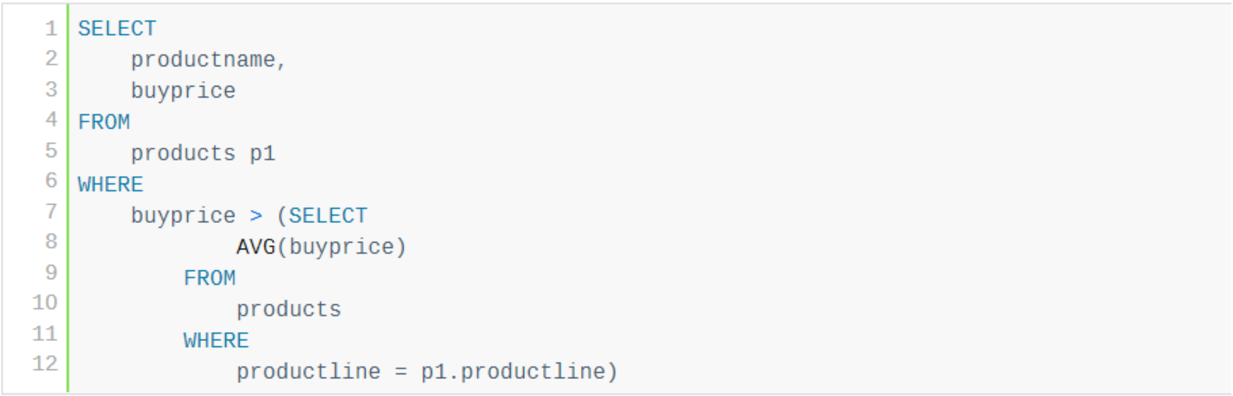
In the previous examples, you notice that a subquery is independent. It means that you can execute the subquery as a standalone query, for example:



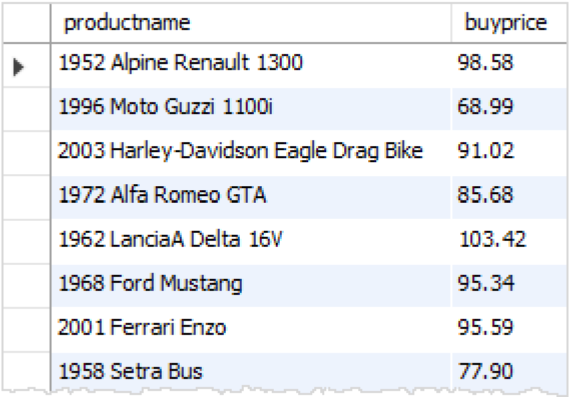
```
SELECT
    orderNumber,
    COUNT(orderNumber) AS items
FROM
    orderdetails
GROUP BY orderNumber;
```

Unlike a standalone subquery, a correlated subquery is a subquery that uses the data from the outer query. In other words, a correlated subquery depends on the outer query. A correlated subquery is evaluated once for each row in the outer query.

In the following query, we select products whose buy prices are greater than the average buy price of all products in each product line.



```
SELECT
    productname,
    buyprice
FROM
    products p1
WHERE
    buyprice > (SELECT
            AVG(buyprice)
        FROM
            products
        WHERE
            productline = p1.productline)
```

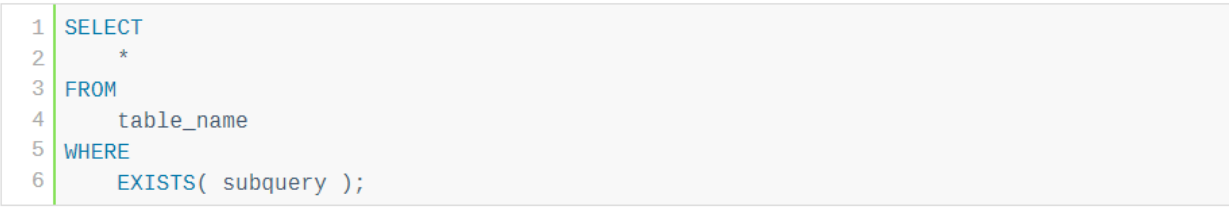


	productname	buyprice
▶	1952 Alpine Renault 1300	98.58
	1996 Moto Guzzi 1100i	68.99
	2003 Harley-Davidson Eagle Drag Bike	91.02
	1972 Alfa Romeo GTA	85.68
	1962 LanciaA Delta 16V	103.42
	1968 Ford Mustang	95.34
	2001 Ferrari Enzo	95.59
	1958 Setra Bus	77.90

The inner query executes for every product line because the product line is changed for every row. Hence, the average buy price will also change. The outer query filters only products whose buy price is greater than the average buy price per product line from the subquery.

MySQL subquery with EXISTS and NOT EXISTS

When a subquery is used with the EXISTS or NOT EXISTS operator, a subquery returns a Boolean value of TRUE or FALSE. The following query illustrates a subquery used with the EXISTS operator:

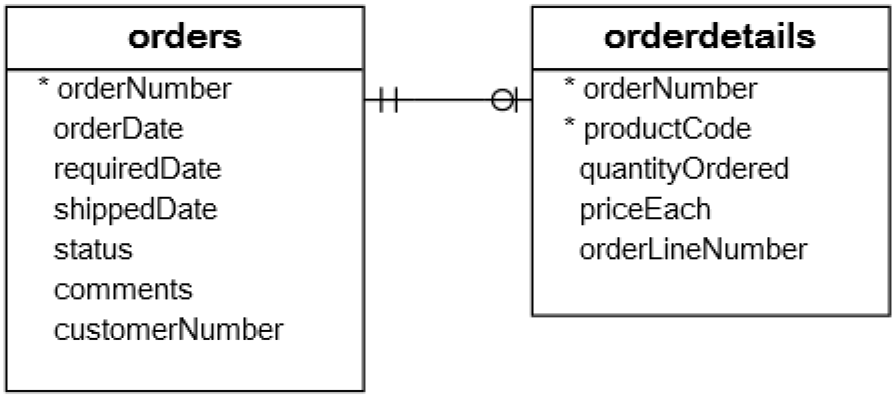


```
SELECT
    *
FROM
    table_name
WHERE
    EXISTS( subquery );
```

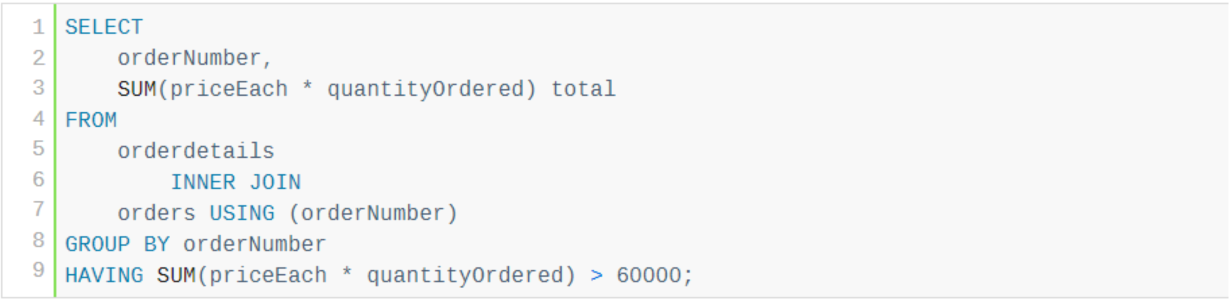
In the query above, if the subquery returns any rows, EXISTS subquery returns TRUE, otherwise, it returns FALSE.

The EXISTS and NOT EXISTS are often used in the correlated subqueries.

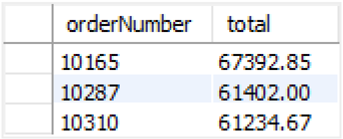
Let’s take a look at the orders and orderDetails table in the sample database:



The following query selects sales orders whose total values are greater than 60K.

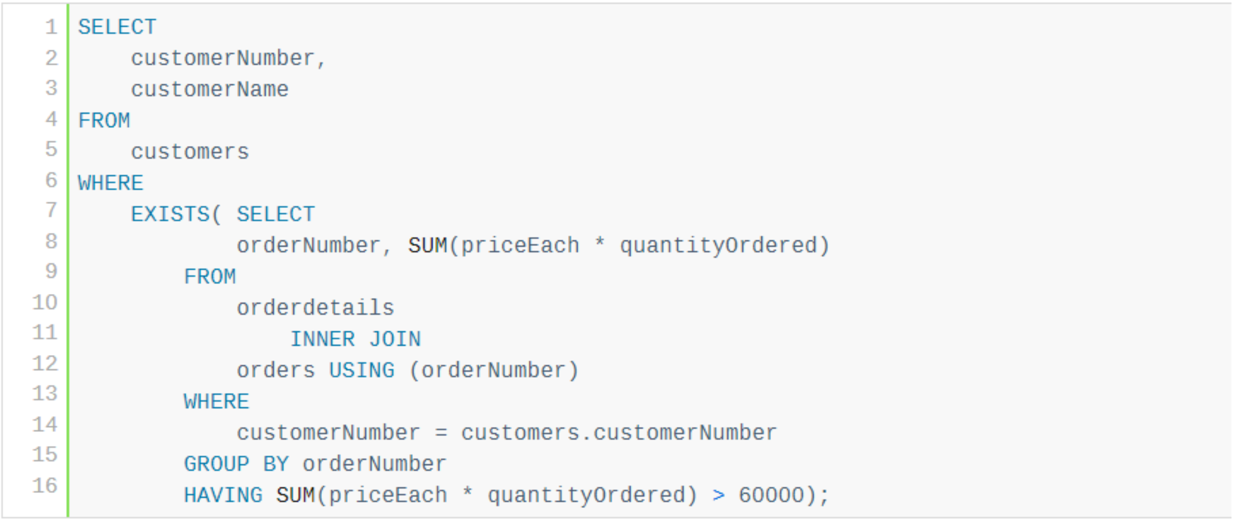


```
SELECT
    orderNumber,
    SUM(priceEach * quantityOrdered) total
FROM
    orderdetails
        INNER JOIN
    orders USING (orderNumber)
GROUP BY orderNumber
HAVING SUM(priceEach * quantityOrdered) > 60000;
```

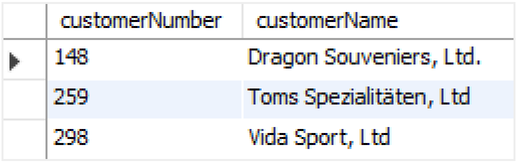


orderNumber	total
10165	67392.85
10287	61402.00
10310	61234.67

It returns 3 rows, meaning that there are 3 sales orders whose total values are greater than 60K. You can use the query above as a correlated subquery to find customers who placed at least one sales order with the total value greater than 60K by using the EXISTS operator:



```
SELECT
    customerNumber,
    customerName
FROM
    customers
WHERE
    EXISTS( SELECT
            orderNumber, SUM(priceEach * quantityOrdered)
        FROM
            orderdetails
                INNER JOIN
            orders USING (orderNumber)
        WHERE
            customerNumber = customers.customerNumber
        GROUP BY orderNumber
        HAVING SUM(priceEach * quantityOrdered) > 60000);
```



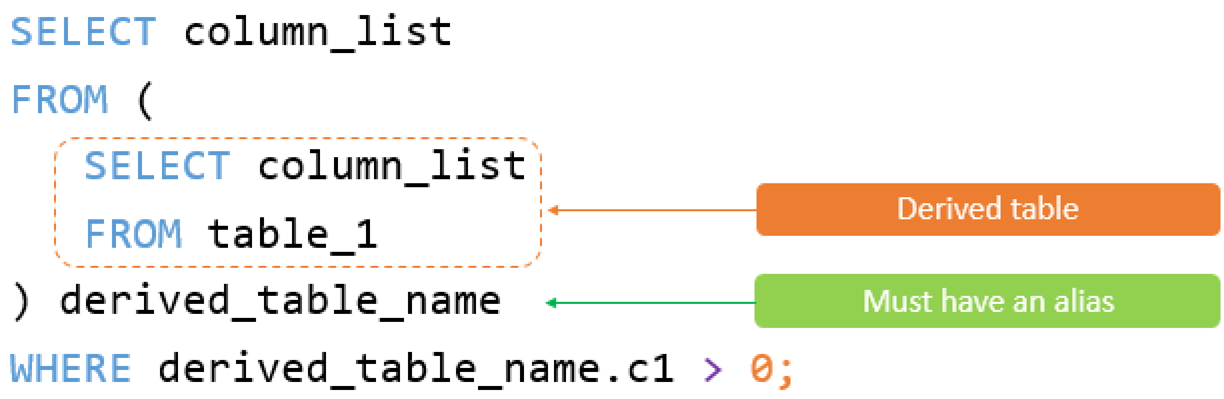
customerNumber	customerName
148	Dragon Souveniers, Ltd.
259	Toms Spezialitäten, Ltd
298	Vida Sport, Ltd

MySQL Derived Table

A derived table is a virtual table returned from a SELECT statement. A derived table is similar to a temporary table, but using a derived table in the SELECT statement is much simpler than a temporary table because it does not require steps of creating the temporary table.

The term derived table and subquery is often used interchangeably. When a stand-alone subquery is used in the FROM clause of a SELECT statement, we call it a derived table.

The following illustrates a query that uses a derived table:



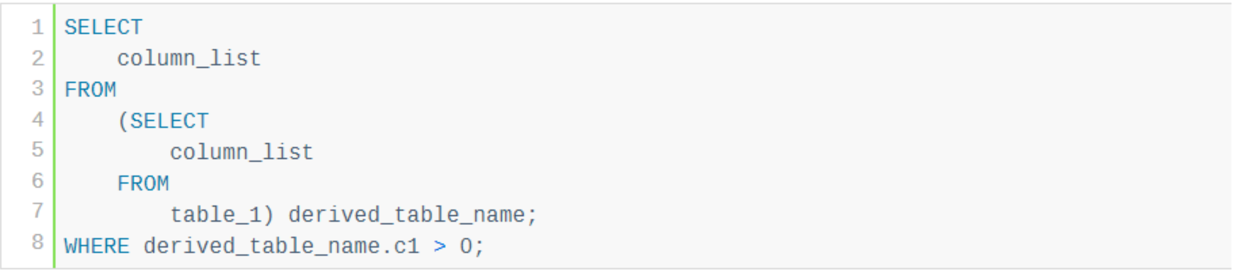
```
SELECT column_list
FROM (
   SELECT column_list
   FROM table_1
) derived_table_name
WHERE derived_table_name.c1 > 0;
```

Note that a stand-alone subquery is a subquery which can execute independently of the statement containing it.

Unlike a subquery, a derived table must have an alias so that you can reference its name later in the query. If a derived table does not have an alias, MySQL will issue the following error:

```
Every derived table must have its own alias.
```

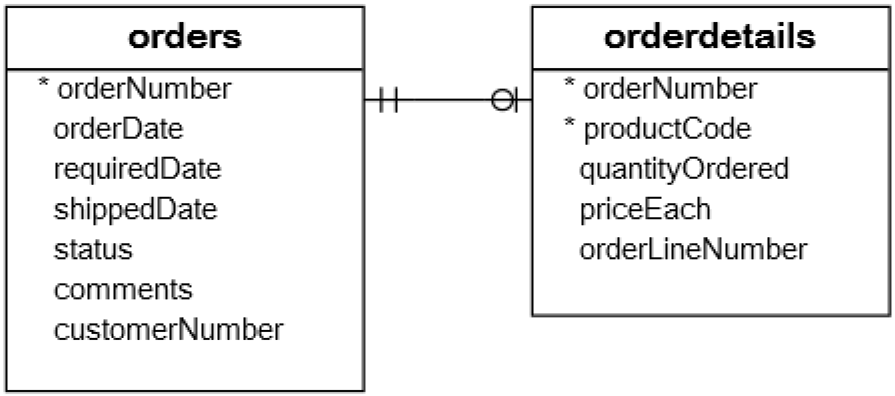
The following illustrates an SQL statement that uses a derived table:

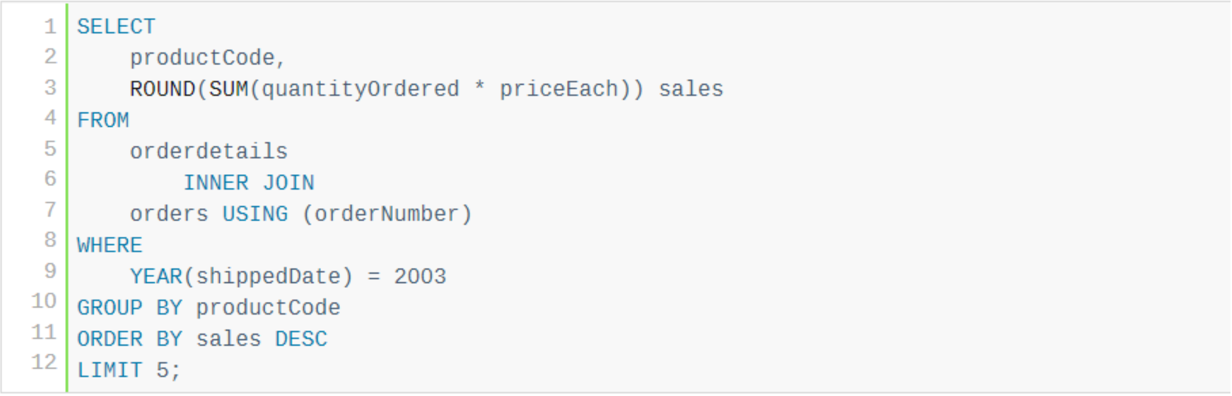


```
SELECT
    column_list
FROM
    (SELECT
        column_list
    FROM
        table_1) derived_table_name;
WHERE derived_table_name.c1 > 0;
```

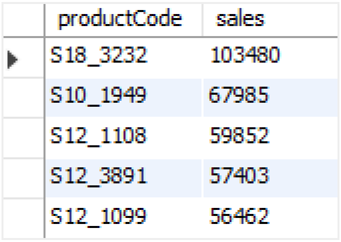
A simple MySQL derived table example

The following query gets the top 5 products by sales revenue in 2003 from the orders and orderdetails tables in the sample database:



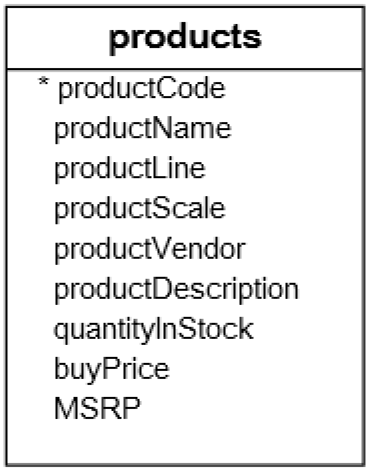


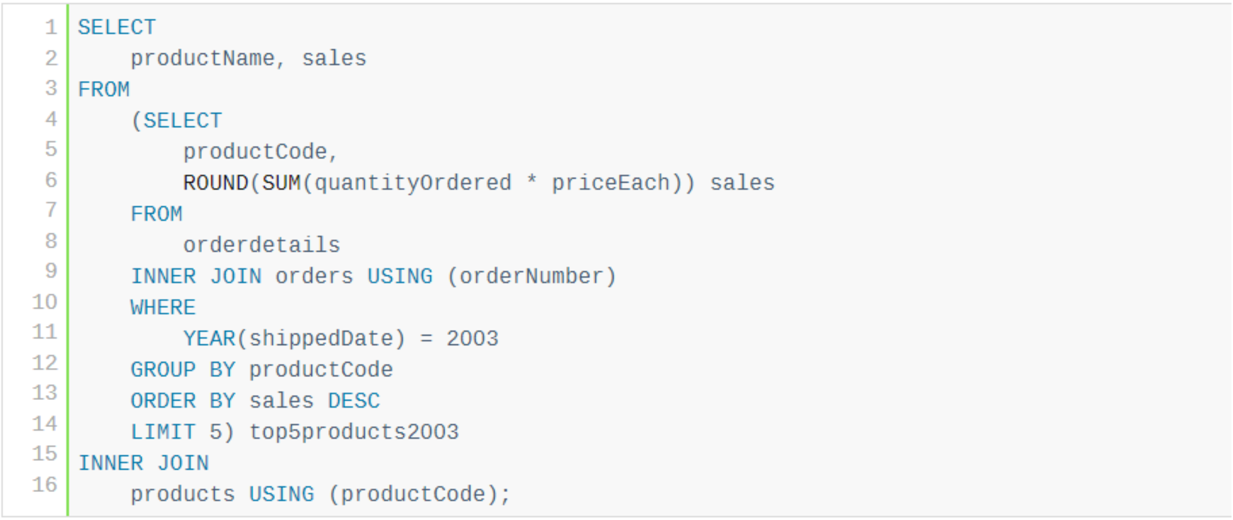
```
SELECT
    productCode,
    ROUND(SUM(quantityOrdered * priceEach)) sales
FROM
    orderdetails
        INNER JOIN
    orders USING (orderNumber)
WHERE
    YEAR(shippedDate) = 2003
GROUP BY productCode
ORDER BY sales DESC
LIMIT 5;
```



productCode	sales
S18_3232	103480
S10_1949	67985
S12_1108	59852
S12_3891	57403
S12_1099	56462

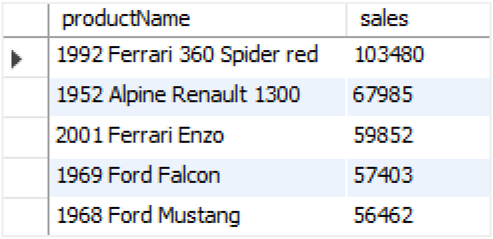
You can use the result of this query as a derived table and join it with the products table as follows:





```
SELECT
    productName, sales
FROM
    (SELECT
        productCode,
        ROUND(SUM(quantityOrdered * priceEach)) sales
    FROM
        orderdetails
    INNER JOIN orders USING (orderNumber)
    WHERE
        YEAR(shippedDate) = 2003
    GROUP BY productCode
    ORDER BY sales DESC
    LIMIT 5) top5products2003
INNER JOIN
    products USING (productCode);
```

The following shows the output of the query above:



productName	sales
1992 Ferrari 360 Spider red	103480
1952 Alpine Renault 1300	67985
2001 Ferrari Enzo	59852
1969 Ford Falcon	57403
1968 Ford Mustang	56462

In this example:

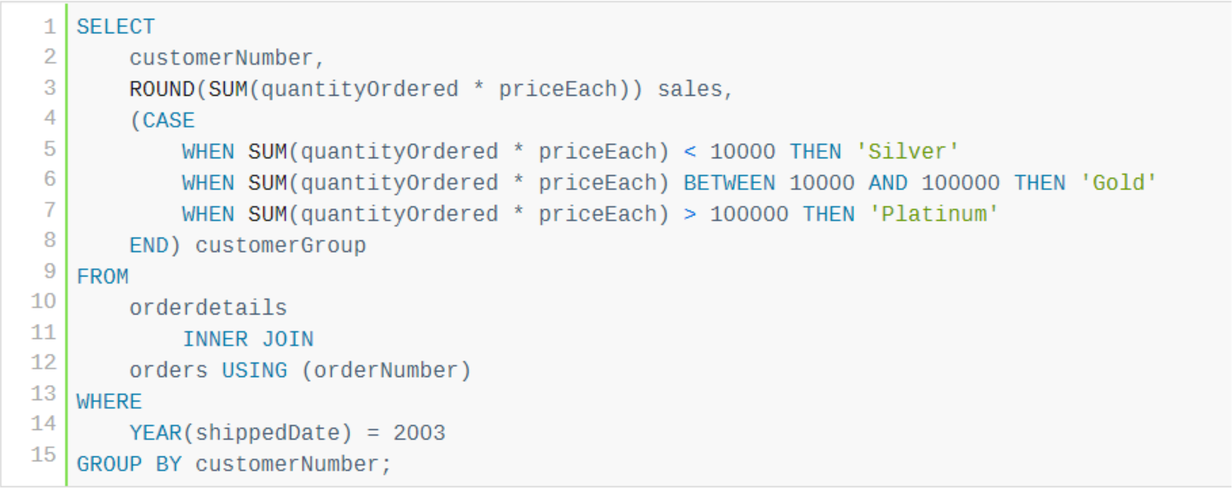
1. First, the subquery executed to create a result set or derived table.
2. Then, the outer query executed that joined the top5product2003 derived table with the products table using the productCode column.

A more complex MySQL derived table example

Suppose you have to classify the customers in the year of 2003 into 3 groups: platinum, gold, and silver. In addition, you need to know the number of customers in each group with the following conditions:

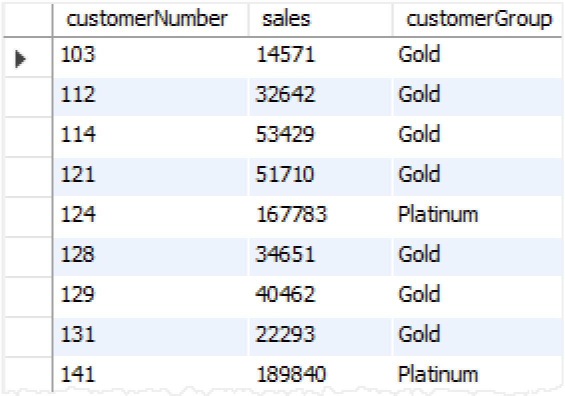
1. Platinum customers who have orders with the volume greater than 100K
2. Gold customers who have orders with the volume between 10K and 100K
3. Silver customers who have orders with the volume less than 10K

To construct this query, first, you need to put each customer into the respective group using CASE expression and GROUP BY clause as follows:



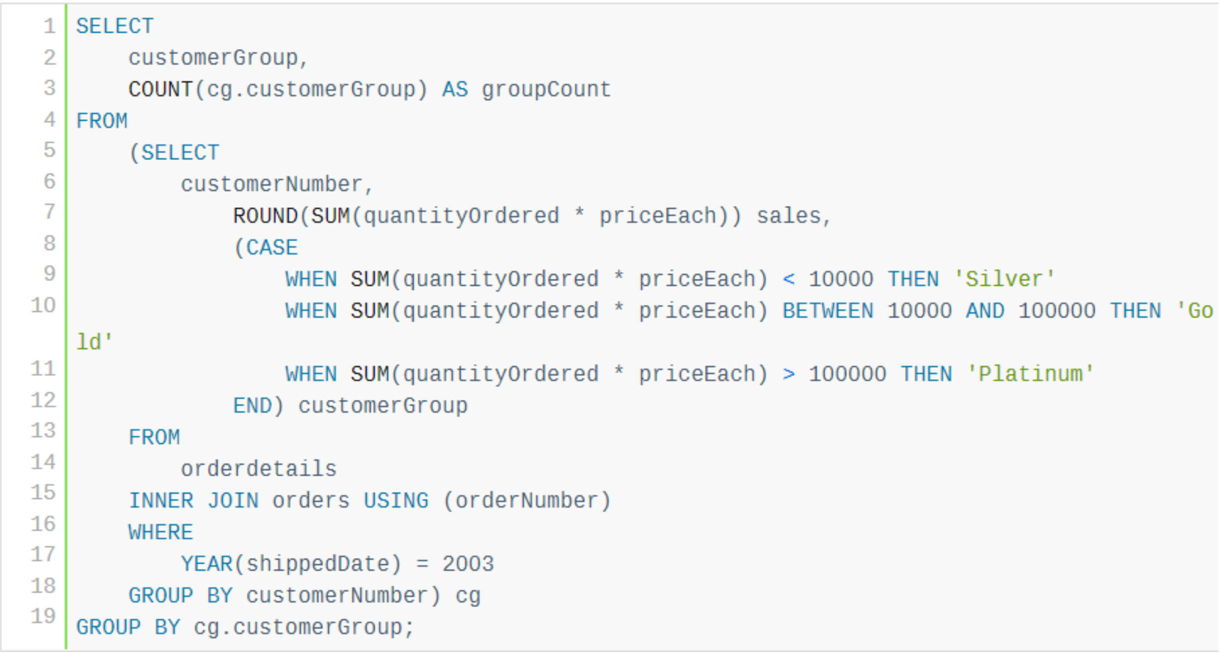
```
SELECT
    customerNumber,
    ROUND(SUM(quantityOrdered * priceEach)) sales,
    (CASE
        WHEN SUM(quantityOrdered * priceEach) < 10000 THEN 'Silver'
        WHEN SUM(quantityOrdered * priceEach) BETWEEN 10000 AND 100000 THEN 'Gold'
        WHEN SUM(quantityOrdered * priceEach) > 100000 THEN 'Platinum'
    END) customerGroup
FROM
    orderdetails
        INNER JOIN
    orders USING (orderNumber)
WHERE
    YEAR(shippedDate) = 2003
GROUP BY customerNumber;
```

The following is the output of the query:



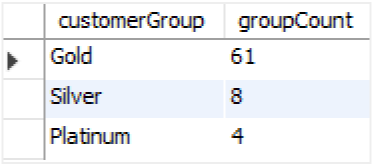
customerNumber	sales	customerGroup
103	14571	Gold
112	32642	Gold
114	53429	Gold
121	51710	Gold
124	167783	Platinum
128	34651	Gold
129	40462	Gold
131	22293	Gold
141	189840	Platinum

Then, you can use this query as the derived table and perform grouping as follows:



```sql
SELECT
    customerGroup,
    COUNT(cg.customerGroup) AS groupCount
FROM
    (SELECT
        customerNumber,
            ROUND(SUM(quantityOrdered * priceEach)) sales,
            (CASE
                WHEN SUM(quantityOrdered * priceEach) < 10000 THEN 'Silver'
                WHEN SUM(quantityOrdered * priceEach) BETWEEN 10000 AND 100000 THEN 'Gold'
                WHEN SUM(quantityOrdered * priceEach) > 100000 THEN 'Platinum'
            END) customerGroup
    FROM
        orderdetails
    INNER JOIN orders USING (orderNumber)
    WHERE
        YEAR(shippedDate) = 2003
    GROUP BY customerNumber) cg
GROUP BY cg.customerGroup;
```

The query returns the customer groups and the number of customers in each.



customerGroup	groupCount
Gold	61
Silver	8
Platinum	4

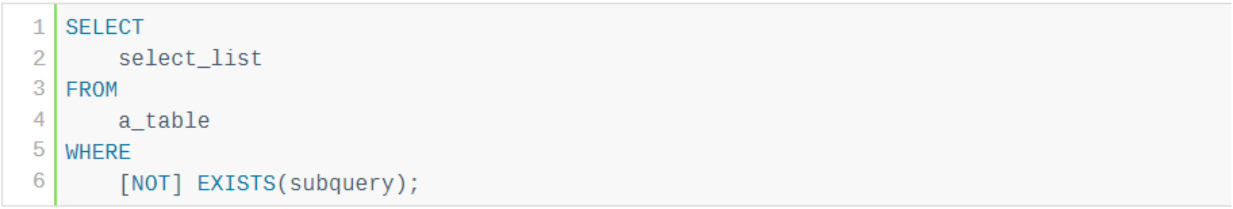
In this tutorial, you have learned how to use the MySQL derived tables which are subqueries in the FROM clause to simplify complex queries.

MySQL EXISTS

The EXISTS operator is a Boolean operator that returns either true or false.

The EXISTS operator is often used the in a subquery to test for an “exist” condition.

The following illustrates the common usage of the EXISTS operator.



```
SELECT
    select_list
FROM
    a_table
WHERE
    [NOT] EXISTS(subquery);
```

If the subquery returns any row, the EXISTS operator returns true, otherwise, it returns false.

In addition, the EXISTS operator terminates further processing immediately once it finds a matching row. Because of this characteristic, you can use the EXISTS operator to improve the performance of the query in some cases.

The NOT operator negates the EXISTS operator. In other words, the NOT EXISTS returns true if the subquery returns no row, otherwise it returns false.

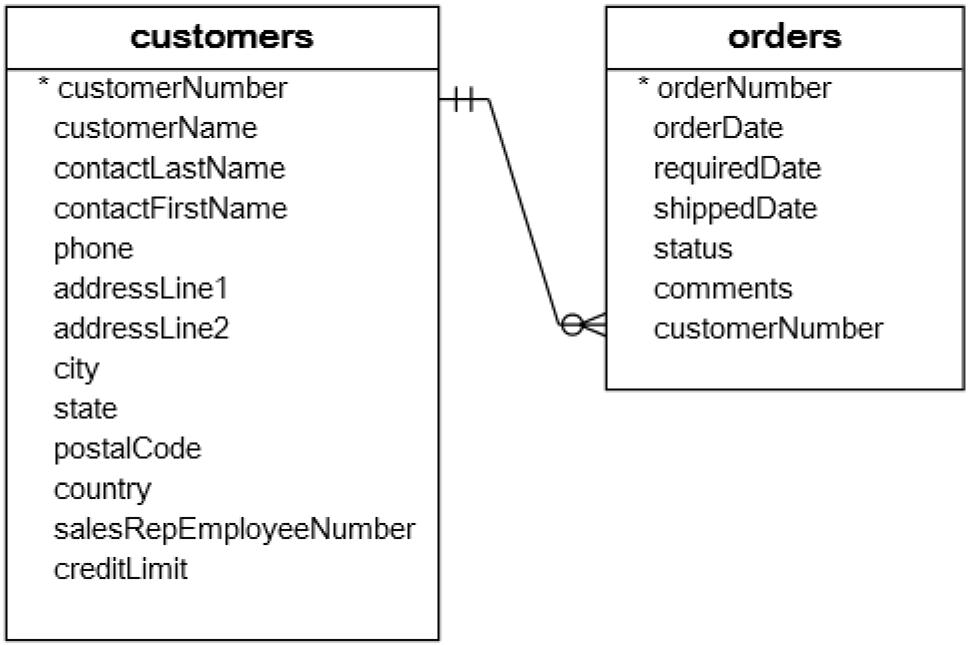
You can use SELECT *, SELECT column, SELECT a_constant, or anything in the subquery. The results are the same because MySQL ignores the select_list that appears in the SELECT clause.

MySQL EXISTS examples

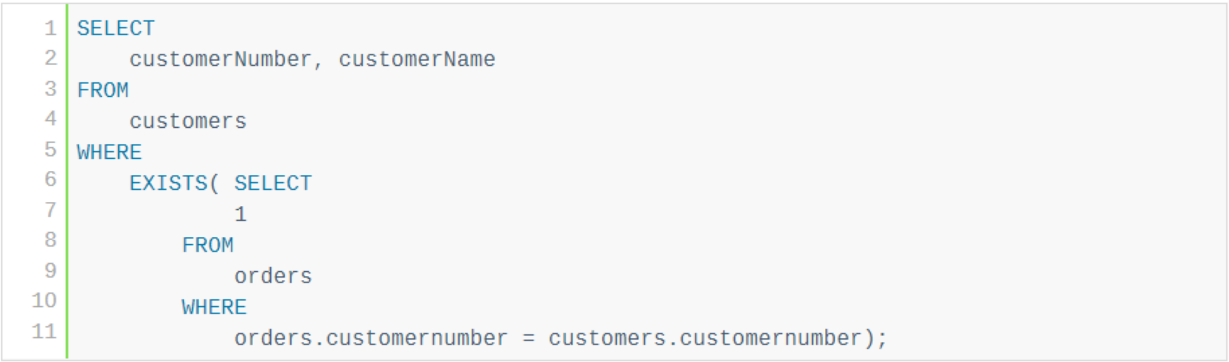
Let’s take some examples of using the EXISTS operator to understand how it works.

MySQL SELECT EXISTS example

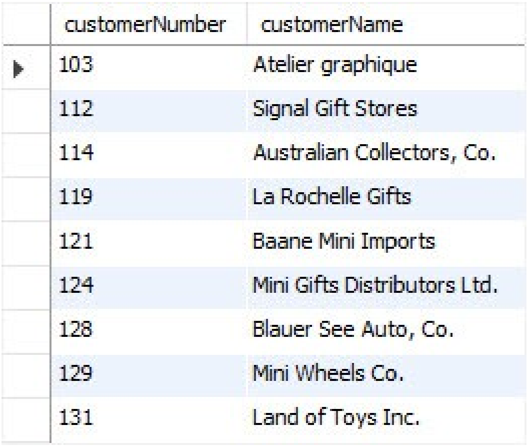
Let’s take a look at the customers and orders tables in the sample database.



Suppose you want to find the customer who has placed at least one sales order, you use the EXISTS operator as follows:



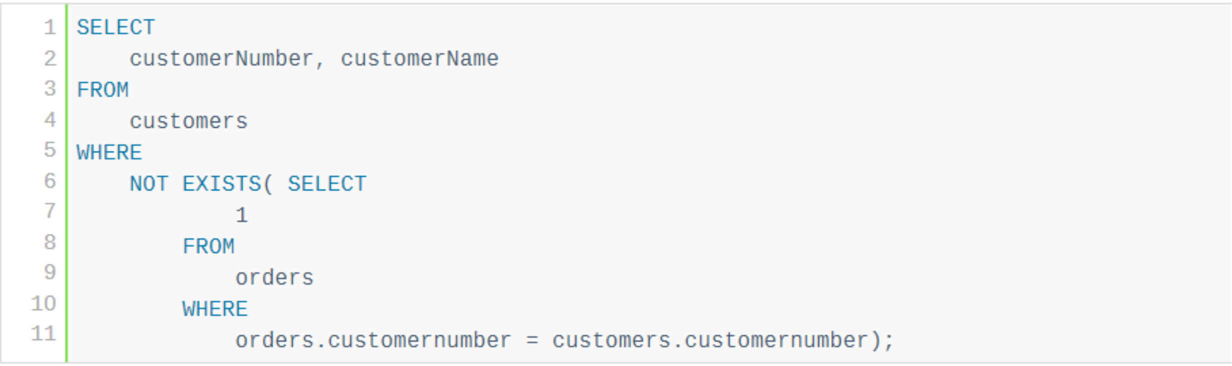
```sql
SELECT
    customerNumber, customerName
FROM
    customers
WHERE
    EXISTS( SELECT
            1
        FROM
            orders
        WHERE
            orders.customernumber = customers.customernumber);
```



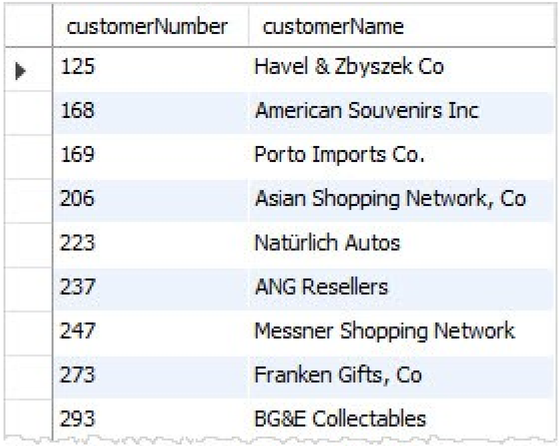
customerNumber	customerName
103	Atelier graphique
112	Signal Gift Stores
114	Australian Collectors, Co.
119	La Rochelle Gifts
121	Baane Mini Imports
124	Mini Gifts Distributors Ltd.
128	Blauer See Auto, Co.
129	Mini Wheels Co.
131	Land of Toys Inc.

For each row in the customers table, the query checks the customerNumber in the orders table. If the customerNumber, which appears in the customers table, exists in the orders table, the subquery returns the first matching row. As the result, the EXISTS operator returns true and stops scanning the orders table. Otherwise, the subquery returns no row and the EXISTS operator returns false.

To get the customer who has not placed any sales orders, you use the NOT EXISTS operator as the following statement:



```
SELECT
    customerNumber, customerName
FROM
    customers
WHERE
    NOT EXISTS( SELECT
            1
        FROM
            orders
        WHERE
            orders.customernumber = customers.customernumber);
```

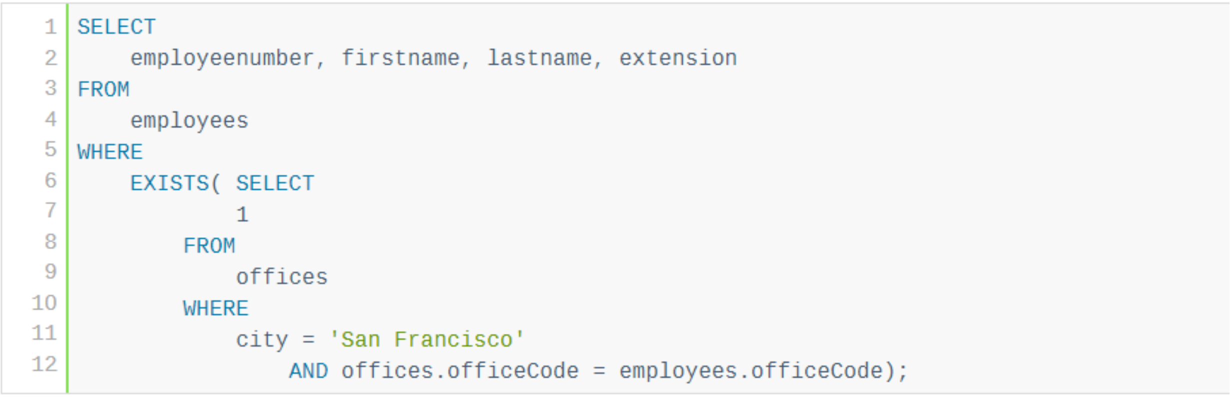


customerNumber	customerName
125	Havel & Zbyszek Co
168	American Souvenirs Inc
169	Porto Imports Co.
206	Asian Shopping Network, Co
223	Natürlich Autos
237	ANG Resellers
247	Messner Shopping Network
273	Franken Gifts, Co
293	BG&E Collectables

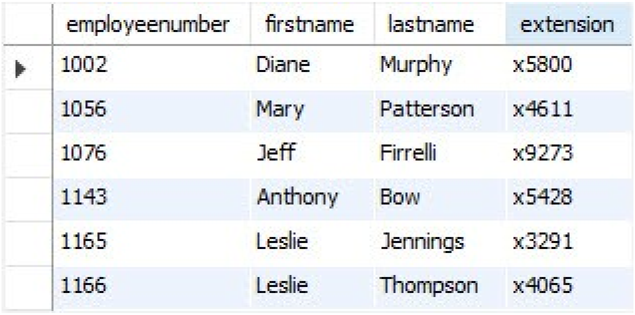
MySQL UPDATE EXISTS example

Assume that you have to update the phone’s extensions of the employees who work at the San Francisco office.

To find employees who work at the San Franciso office, you use the EXISTS operator as the following UPDATE statement:

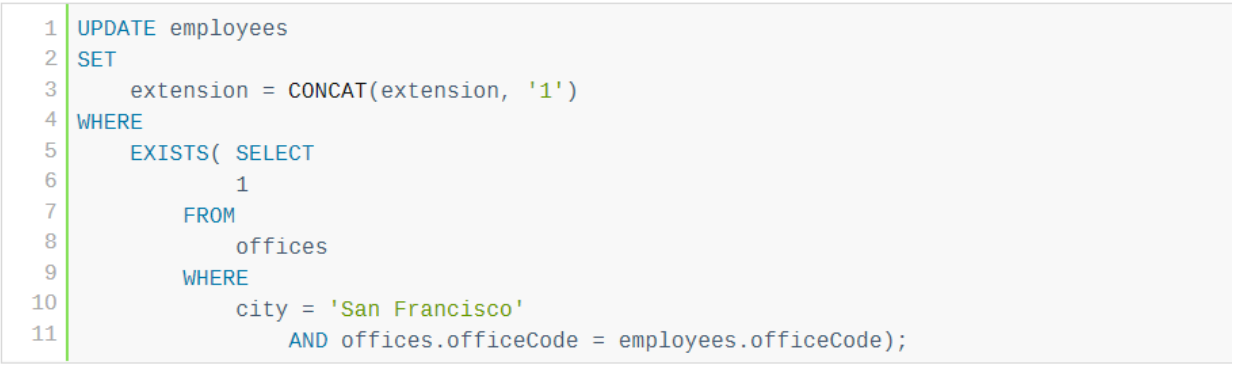


```
SELECT
    employeenumber, firstname, lastname, extension
FROM
    employees
WHERE
    EXISTS( SELECT
            1
        FROM
            offices
        WHERE
            city = 'San Francisco'
                AND offices.officeCode = employees.officeCode);
```



employeenumber	firstname	lastname	extension
1002	Diane	Murphy	x5800
1056	Mary	Patterson	x4611
1076	Jeff	Firrelli	x9273
1143	Anthony	Bow	x5428
1165	Leslie	Jennings	x3291
1166	Leslie	Thompson	x4065

Suppose you want to add the number 5 at every phone's extension of the employees who work at the San Francisco office, you can use the EXISTS operator in WHERE clause of the UPDATE statement as follows:

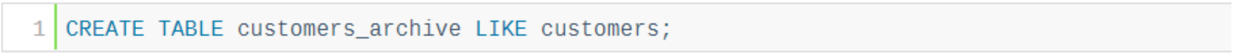


```
UPDATE employees
SET
    extension = CONCAT(extension, '1')
WHERE
    EXISTS( SELECT
            1
        FROM
            offices
        WHERE
            city = 'San Francisco'
                AND offices.officeCode = employees.officeCode);
```

MySQL INSERT EXISTS example

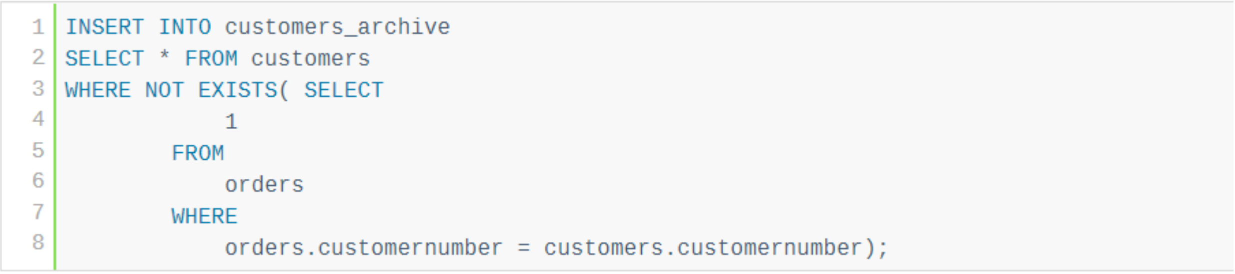
Suppose you want to archive the customers who have not placed any sales order in a separate table. To achieve this, you follow the steps below.

First, create a new table for archiving the customers by copying the structure from the customers table.



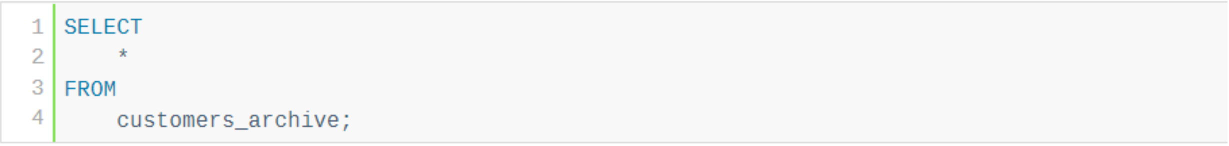
```
CREATE TABLE customers_archive LIKE customers;
```

Second, insert the customers who have not placed any sales order into the customers_archive table using the following INSERT statement.

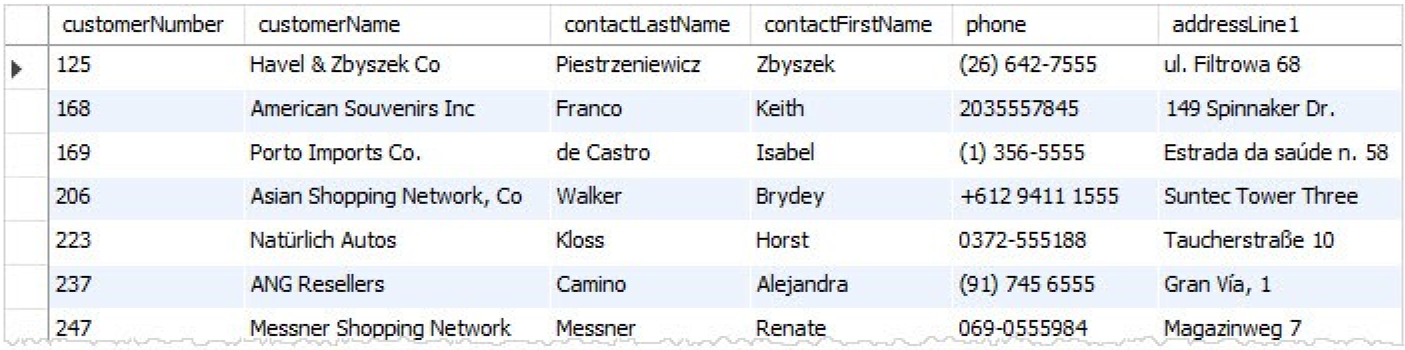


```
INSERT INTO customers_archive
SELECT * FROM customers
WHERE NOT EXISTS( SELECT
            1
        FROM
            orders
        WHERE
            orders.customernumber = customers.customernumber);
```

Third, query data from the customers_archive table to verify the insert operation.



```
SELECT
    *
FROM
    customers_archive;
```

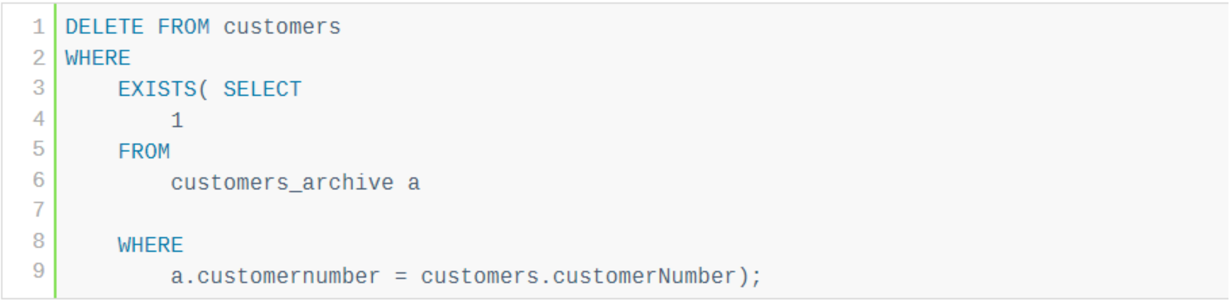


customerNumber	customerName	contactLastName	contactFirstName	phone	addressLine1
125	Havel & Zbyszek Co	Piestrzeniewicz	Zbyszek	(26) 642-7555	ul. Filtrowa 68
168	American Souvenirs Inc	Franco	Keith	2035557845	149 Spinnaker Dr.
169	Porto Imports Co.	de Castro	Isabel	(1) 356-5555	Estrada da saúde n. 58
206	Asian Shopping Network, Co	Walker	Brydey	+612 9411 1555	Suntec Tower Three
223	Natürlich Autos	Kloss	Horst	0372-555188	Taucherstraße 10
237	ANG Resellers	Camino	Alejandra	(91) 745 6555	Gran Vía, 1
247	Messner Shopping Network	Messner	Renate	069-0555984	Magazinweg 7

MySQL DELETE EXISTS example

One final task in archiving the customers data is to delete the customers that exist in the customers_archive table from the customers table.

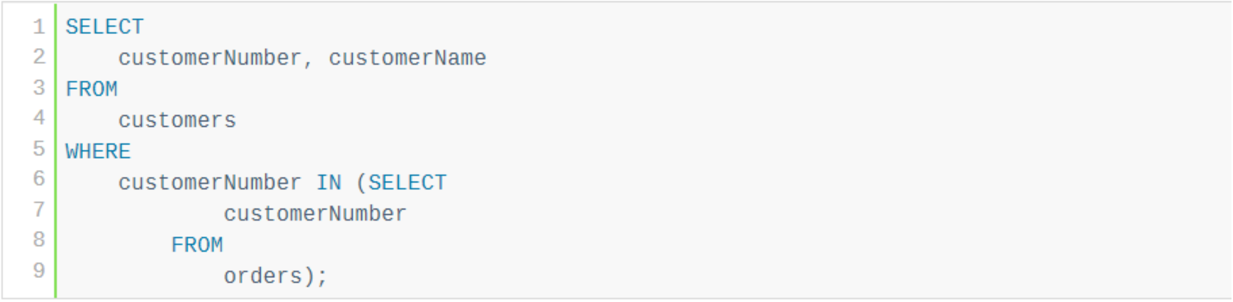
To do this, you use the EXISTS operator in WHERE clause of the DELETE statement as follows:



```
DELETE FROM customers
WHERE
    EXISTS( SELECT
        1
    FROM
        customers_archive a

    WHERE
        a.customernumber = customers.customerNumber);
```

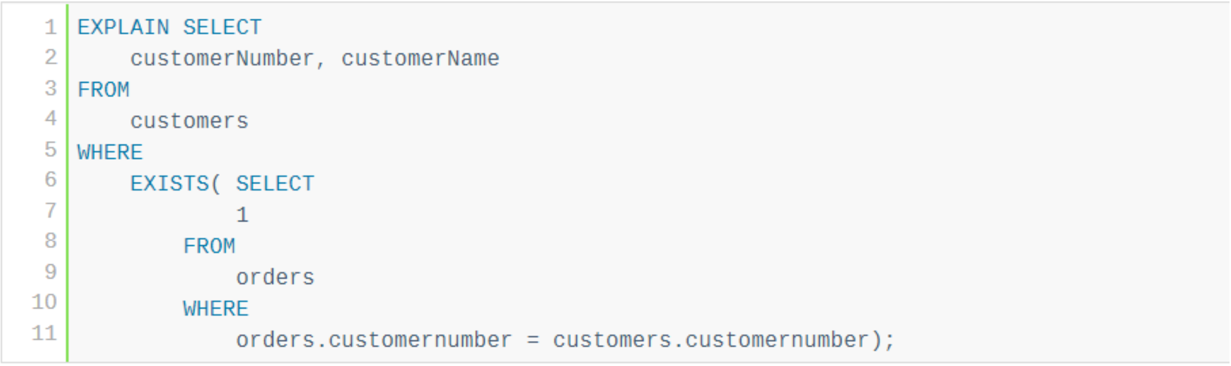
MySQL EXISTS vs. IN

To find the customer who has placed at least one sales order, you can use the IN operator as follows:

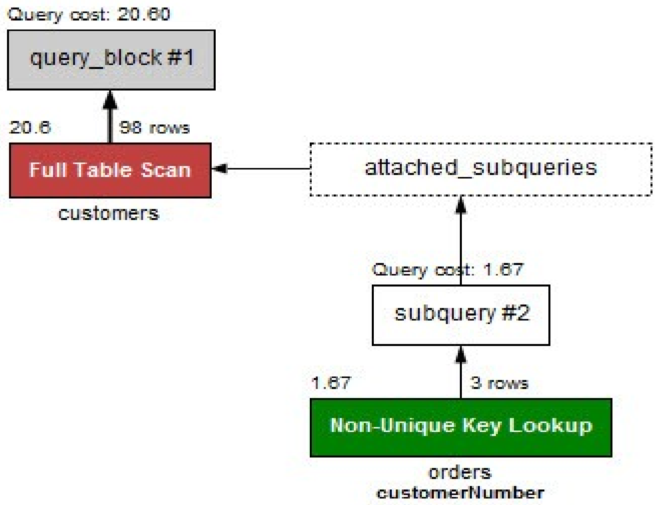


```
SELECT
    customerNumber, customerName
FROM
    customers
WHERE
    customerNumber IN (SELECT
            customerNumber
        FROM
            orders);
```

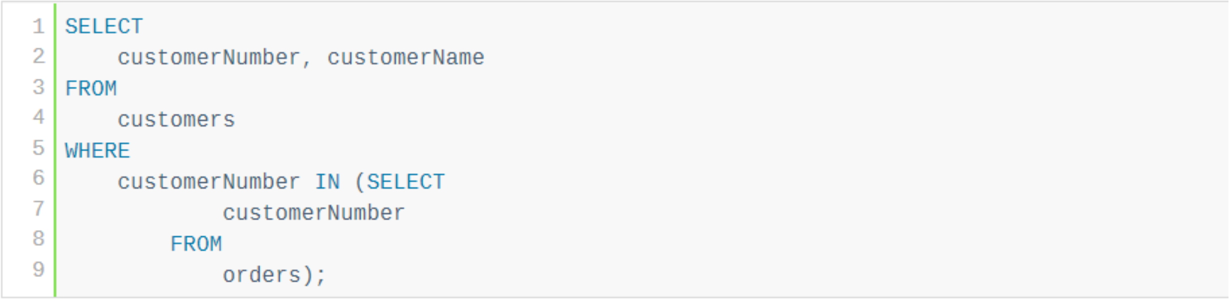
Let's compare the query that uses the IN operator with the one that uses the EXISTS operator by using the EXPLAIN statement.



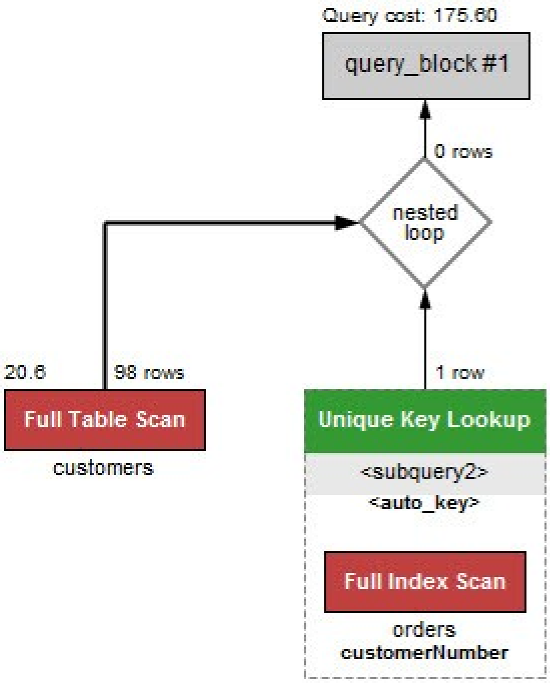
```
EXPLAIN SELECT
    customerNumber, customerName
FROM
    customers
WHERE
    EXISTS( SELECT
            1
        FROM
            orders
        WHERE
            orders.customernumber = customers.customernumber);
```



Now, check the performance of the query that uses the IN operator.



```
SELECT
    customerNumber, customerName
FROM
    customers
WHERE
    customerNumber IN (SELECT
            customerNumber
        FROM
            orders);
```



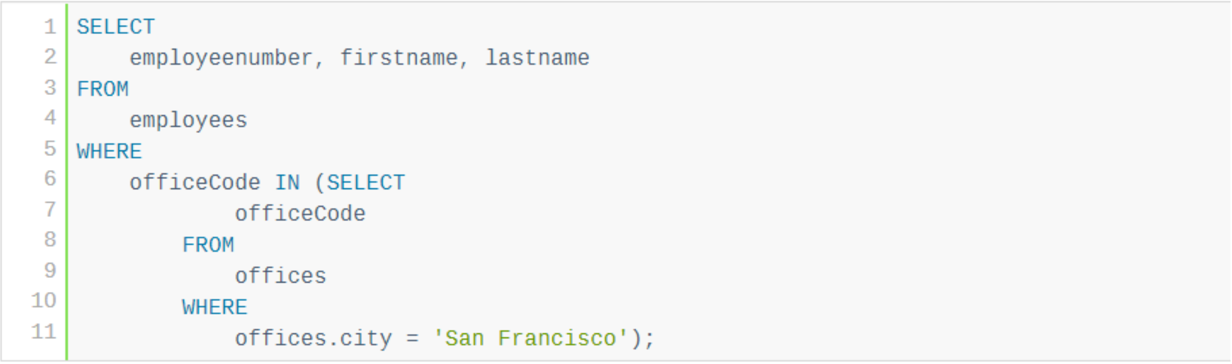
The query that uses the EXISTS operator is much faster than the one that uses the IN operator. The reason is that the EXISTS operator works based on the “at least found” principle. It returns true and stops scanning table once at least one matching row found.

On the other hands, when the IN operator is combined with a subquery, MySQL must process the subquery first and then uses the result of the subquery to process the whole query.

The general rule of thumb is that if the subquery contains a large volume of data, the EXISTS operator provides better performance.

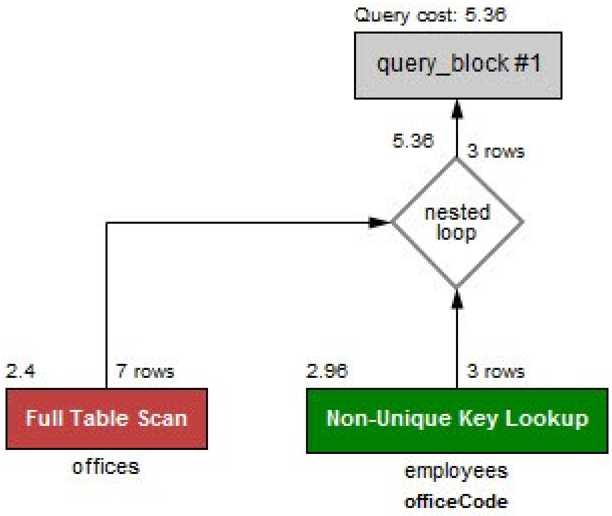
However, the query that uses the IN operator will perform faster if the result set returned from the subquery is very small.

For example, the following statement uses the IN operator selects all employees who work at the office in San Francisco.

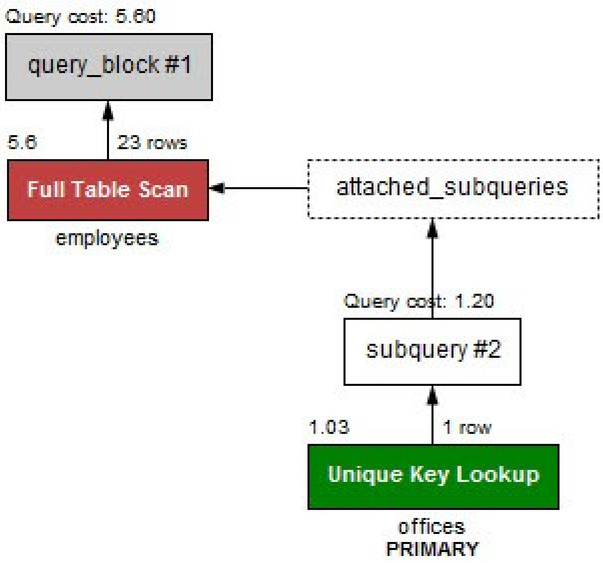


```
SELECT
    employeenumber, firstname, lastname
FROM
    employees
WHERE
    officeCode IN (SELECT
            officeCode
        FROM
            offices
        WHERE
            offices.city = 'San Francisco');
```

Let’s check the performance of the query.



It is a little bit faster than the query that uses the EXISTS operator that we mentioned in the first example. See the performance of the query that uses the EXIST operator below:

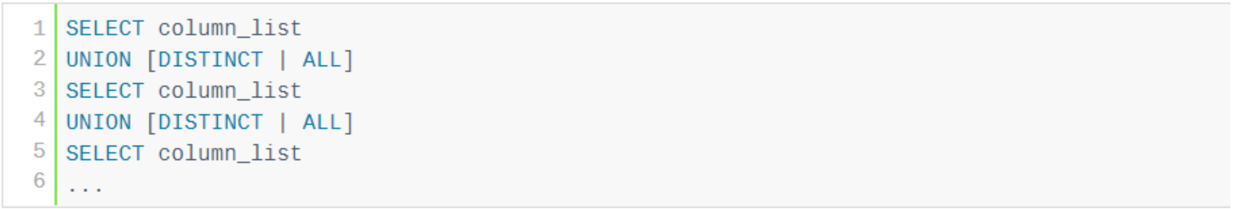


In this tutorial, we have discussed the MySQL EXISTS operator and introduced you to some guidelines for using the EXISTS operator to improve the query's performance.

MySQL UNION

MySQL UNION operator

MySQL UNION operator allows you to combine two or more result sets of queries into a single result set. The following illustrates the syntax of the UNION operator:



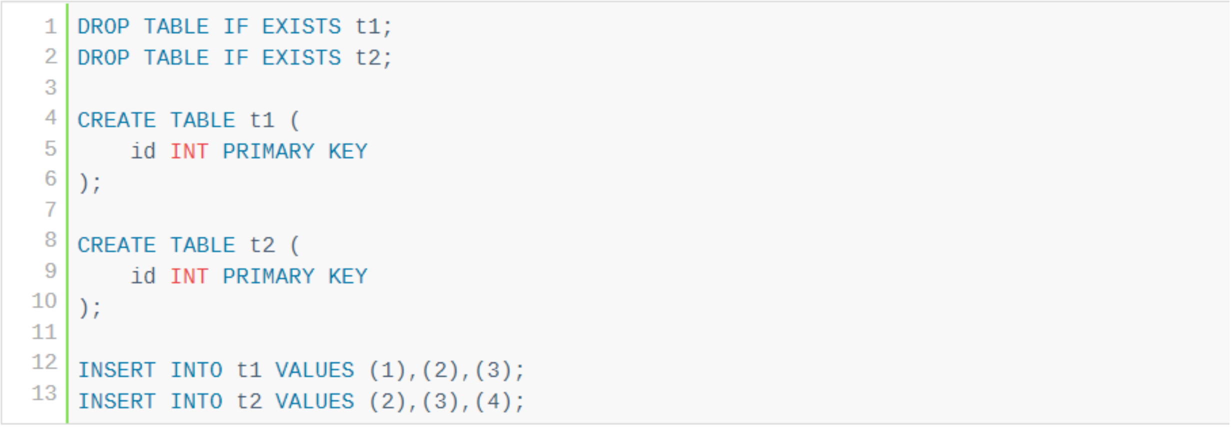
```
SELECT column_list
UNION [DISTINCT | ALL]
SELECT column_list
UNION [DISTINCT | ALL]
SELECT column_list
...
```

To combine result set of two or more queries using the UNION operator, there are the basic rules that you must follow:

- First, the number and the orders of columns that appear in all SELECT statements must be the same.
- Second, the data types of columns must be the same or convertible.

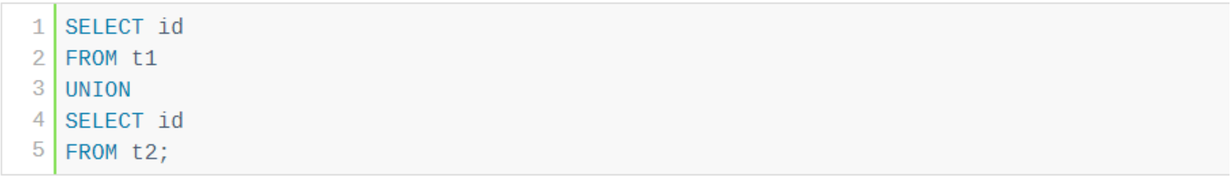
By default, the UNION operator removes duplicate rows even if you don’t specify the DISTINCT operator explicitly.

Let’s see the following sample tables: t1 and t2:



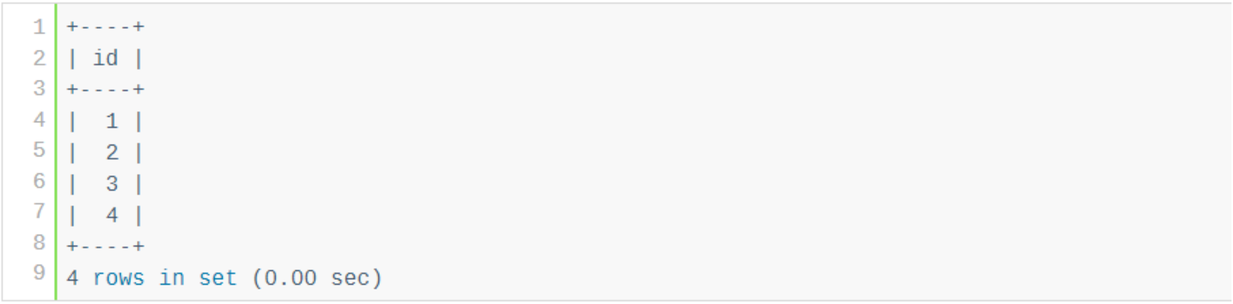
```
DROP TABLE IF EXISTS t1;
DROP TABLE IF EXISTS t2;

CREATE TABLE t1 (
    id INT PRIMARY KEY
);

CREATE TABLE t2 (
    id INT PRIMARY KEY
);

INSERT INTO t1 VALUES (1),(2),(3);
INSERT INTO t2 VALUES (2),(3),(4);
```

The following statement combines result sets returned from t1 and t2 tables:



```
SELECT id
FROM t1
UNION
SELECT id
FROM t2;
```

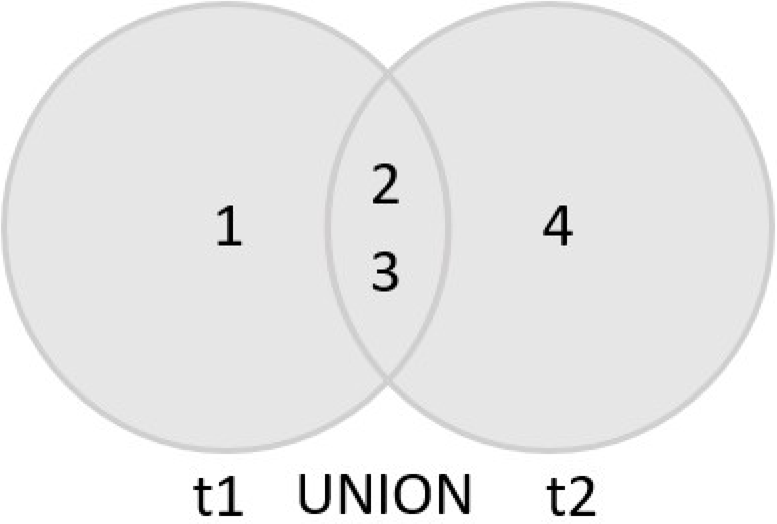
The final result set contains the distinct values from separate result sets returned by the queries:



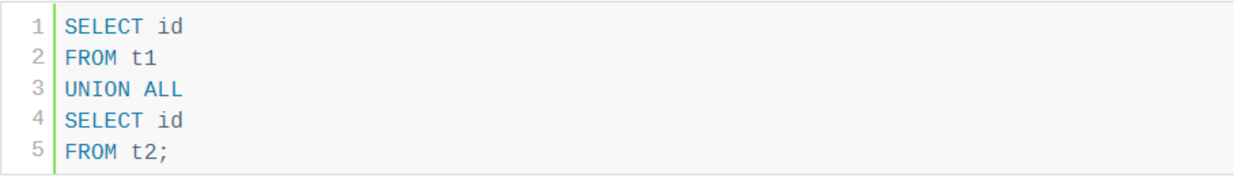
```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
|  4 |
+----+
4 rows in set (0.00 sec)
```

Because the rows with value 2 and 3 are duplicates, the UNION operator removed it and kept only distinct ones.

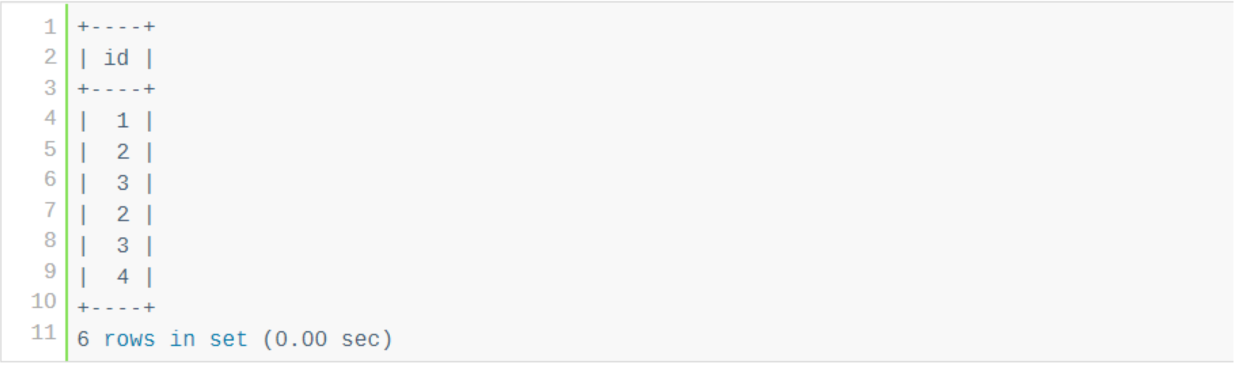
The following Venn diagram illustrates the union of two result sets that come from t1 and t2 tables:



If you use the UNION ALL explicitly, the duplicate rows, if available, remain in the result. Because UNION ALL does not need to handle duplicates, it performs faster than UNION DISTINCT.



```
SELECT id
FROM t1
UNION ALL
SELECT id
FROM t2;
```

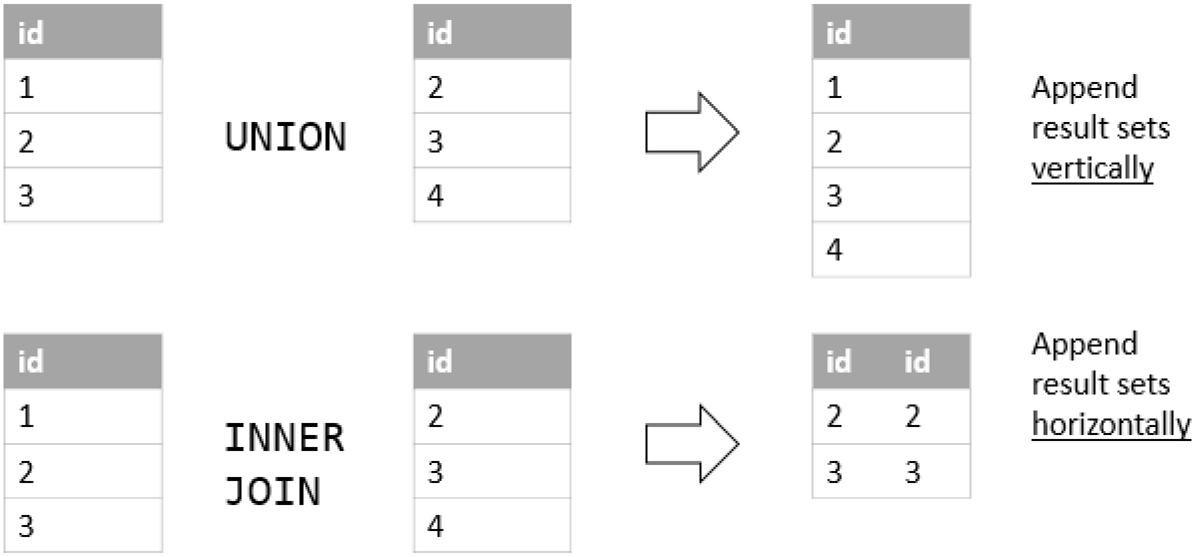


```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
|  2 |
|  3 |
|  4 |
+----+
6 rows in set (0.00 sec)
```

As you can see, the duplicates appear in the combined result set because of the UNION ALL operation.

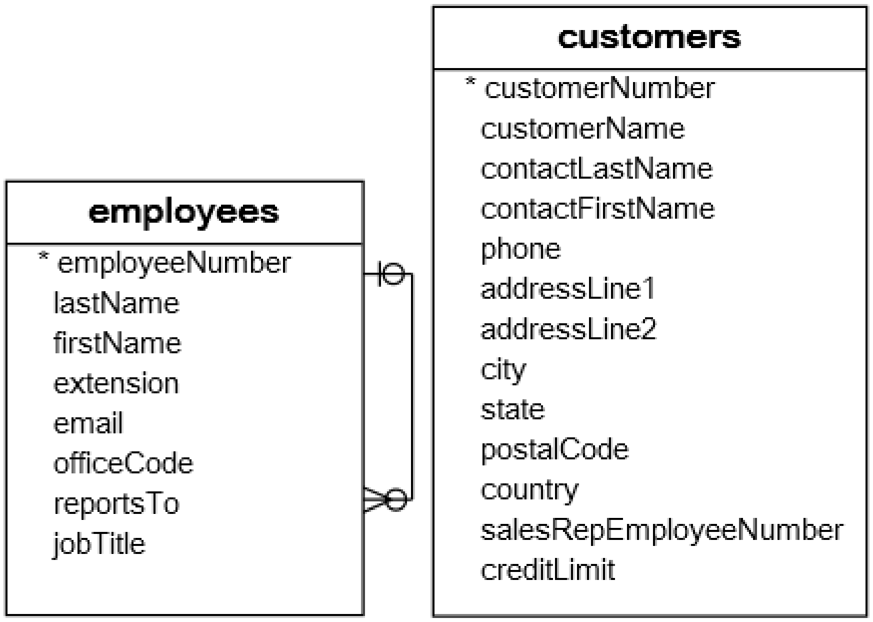
UNION vs. JOIN

A JOIN combines result sets horizontally, a UNION appends result set vertically. The following picture illustrates the difference between UNION and JOIN:

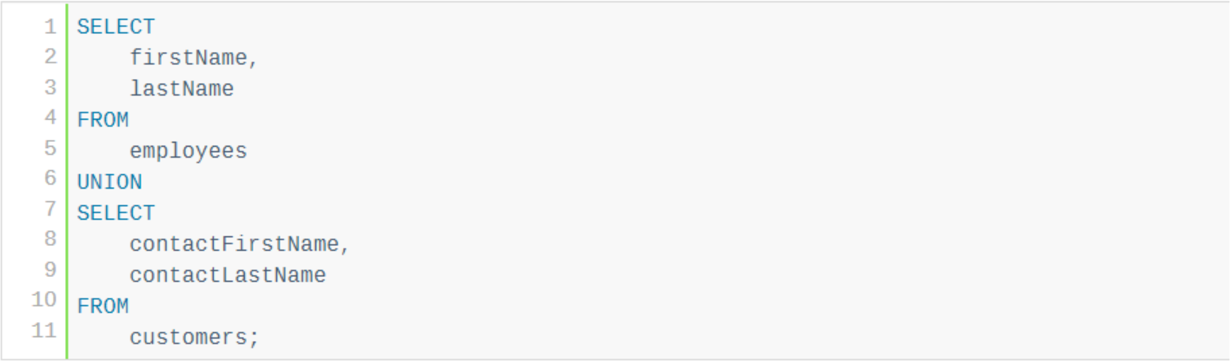

id
1
2
3
UNION
id
2
3
4
id
1
2
3
4
Append result sets vertically
id
1
2
3
INNER JOIN
id
2
3
4
id id
2 2
3 3
Append result sets horizontally

MySQL UNION and column alias examples

We will use the customers and employees tables in the sample database for the demonstration:

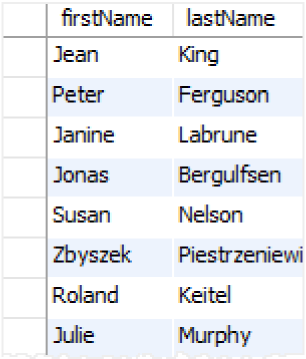


Suppose you want to combine the first name and last name of both employees and customers into a single result set, you can use the UNION operator as follows:



```
SELECT
    firstName,
    lastName
FROM
    employees
UNION
SELECT
    contactFirstName,
    contactLastName
FROM
    customers;
```

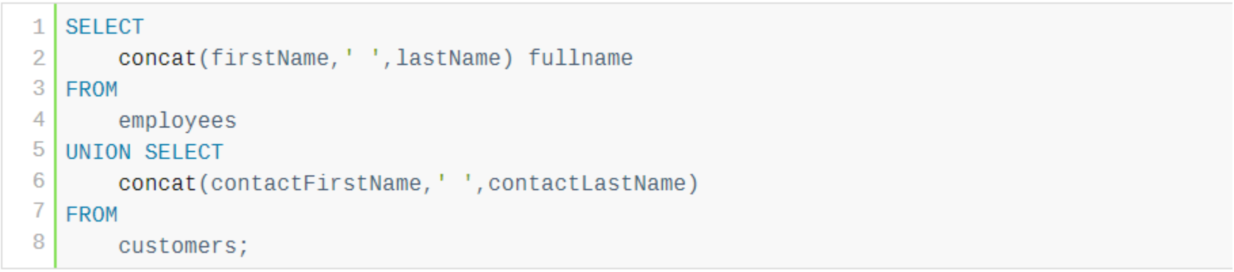
Here is the output:



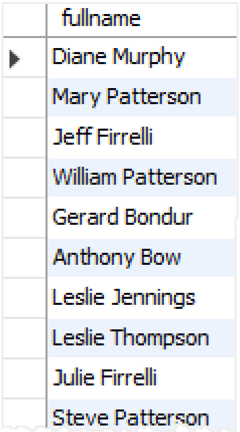
firstName	lastName
Jean	King
Peter	Ferguson
Janine	Labrune
Jonas	Bergulfsen
Susan	Nelson
Zbyszek	Piestrzeniewi
Roland	Keitel
Julie	Murphy

As you can see, the MySQL UNION operator uses the column names of the first SELECT statement for labeling the columns in the output.

If you want to use your own column aliases, you need to specify them explicitly in the first SELECT statement as shown in the following example:



```
SELECT
    concat(firstName,' ',lastName) fullname
FROM
    employees
UNION SELECT
    concat(contactFirstName,' ',contactLastName)
FROM
    customers;
```

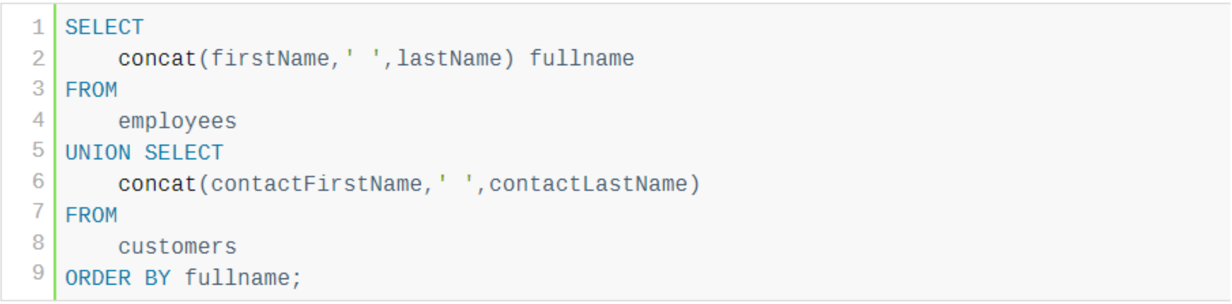


fullname
Diane Murphy
Mary Patterson
Jeff Firrelli
William Patterson
Gerard Bondur
Anthony Bow
Leslie Jennings
Leslie Thompson
Julie Firrelli
Steve Patterson

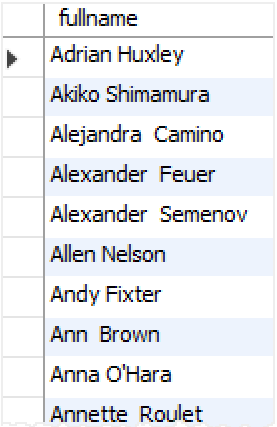
In this example, instead of using the default column label from the first query, we used a column alias fullname for labeling the output.

MySQL UNION and ORDER BY

If you want to sort the result of a union, you use an ORDER BY clause in the last SELECT statement as shown in the following example:



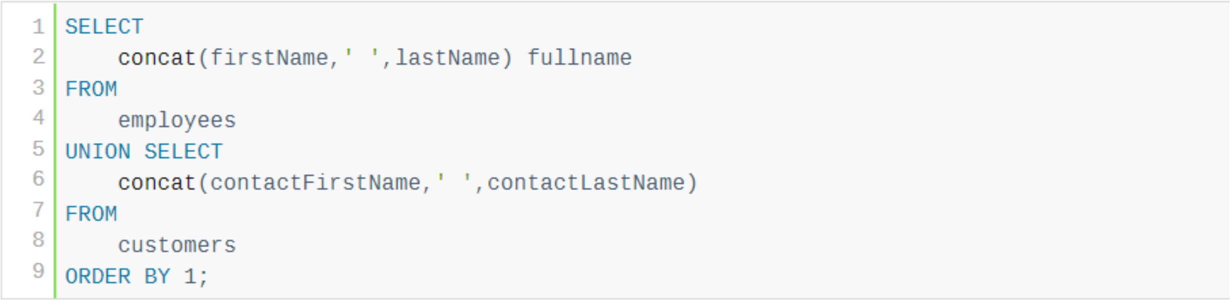
```
SELECT
    concat(firstName,' ',lastName) fullname
FROM
    employees
UNION SELECT
    concat(contactFirstName,' ',contactLastName)
FROM
    customers
ORDER BY fullname;
```



fullname
Adrian Huxley
Akiko Shimamura
Alejandra Camino
Alexander Feuer
Alexander Semenov
Allen Nelson
Andy Fixter
Ann Brown
Anna O'Hara
Annette Roulet

Notice that if you place the ORDER BY clause in each SELECT statement, it will not affect the order of the rows in the final result set.

MySQL also provides you with alternative option to sort a result set based on column position using ORDER BY clause as follows:



```
SELECT
    concat(firstName,' ',lastName) fullname
FROM
    employees
UNION SELECT
    concat(contactFirstName,' ',contactLastName)
FROM
    customers
ORDER BY 1;
```

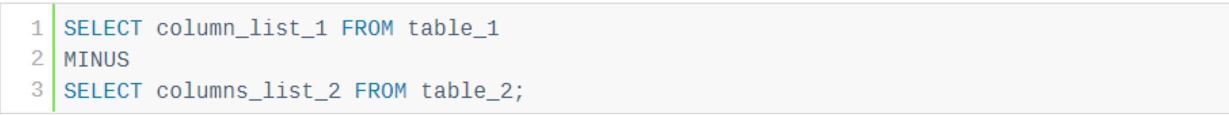
In this tutorial, you have learned how to use MySQL UNION statement to combine data from multiple queries into a single result set.

MySQL MINUS

MINUS is one of three set operations in the SQL standard that includes UNION, INTERSECT, and MINUS.

MINUS compares results of two queries and returns distinct rows from the first query that aren’t output by the second query.

The following illustrates the syntax of the MINUS operator:

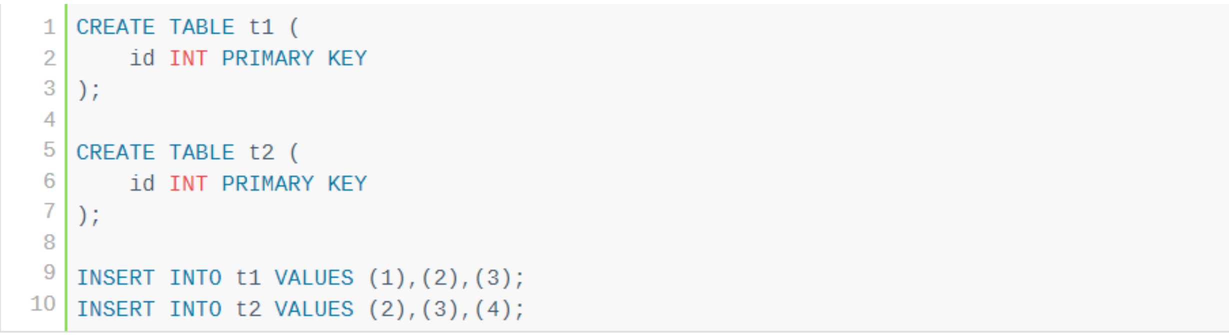


```
SELECT column_list_1 FROM table_1
MINUS
SELECT columns_list_2 FROM table_2;
```

The basic rules for a query that uses MINUS operator are the following:

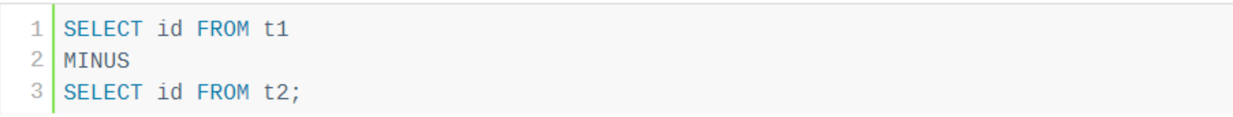
- The number and order of columns in both column_list_1 and column_list_2 must be the same.
- The data types of the corresponding columns in both queries must be compatible.

Suppose we have two tables’ t1 and t2 with the following structure and data:

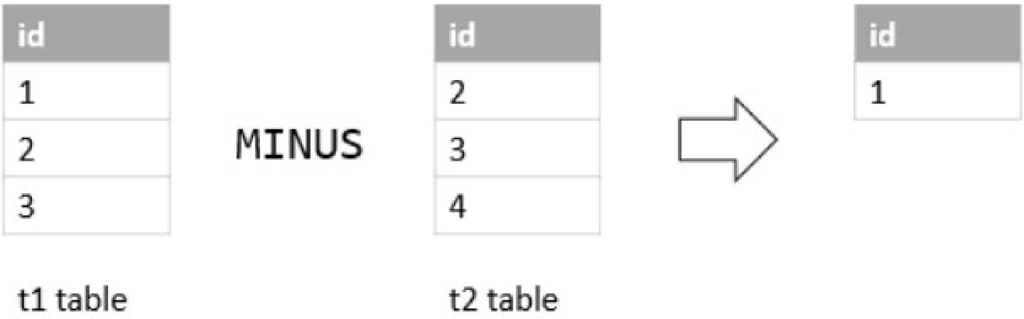


```
CREATE TABLE t1 (
    id INT PRIMARY KEY
);

CREATE TABLE t2 (
    id INT PRIMARY KEY
);

INSERT INTO t1 VALUES (1),(2),(3);
INSERT INTO t2 VALUES (2),(3),(4);
```

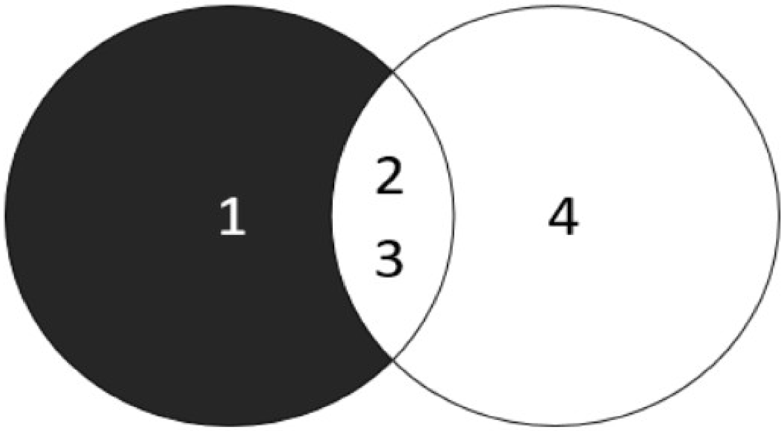
The following query returns distinct values from the query of the t1 table that are not found on the result of the query of the t2 table.



```
SELECT id FROM t1
MINUS
SELECT id FROM t2;
```



The following Venn diagram illustrates the MINUS operator:



Note that some database systems e.g., Microsoft SQL Server, PostgreSQL, etc., use the EXCEPT instead of MINUS, which have the same function.

MySQL MINUS operator

Unfortunately, MySQL does not support MINUS operator. However, you can use the MySQL join to simulate it.

To emulate the MINUS of two queries, you use the following syntax:

```
SELECT
    column_list
FROM
    table_1
    LEFT JOIN table_2 ON join_predicate
WHERE
    table_2.id IS NULL;
```

For example, the following query uses the LEFT JOIN clause to return the same result as the MINUS operator:

```
SELECT
    id
FROM
    t1
        LEFT JOIN
    t2 USING (id)
WHERE
    t2.id IS NULL;
```

In this tutorial, you have learned about the SQL MINUS operator and how to implement MySQL MINUS operator using LEFT JOIN clause.

MySQL INTERSECT

The INTERSECT operator is a set operator that returns only distinct rows of two queries or more queries.

The following illustrates the syntax of the INTERSECT operator.

```
(SELECT column_list
FROM table_1)
INTERSECT
(SELECT column_list
FROM table_2);
```

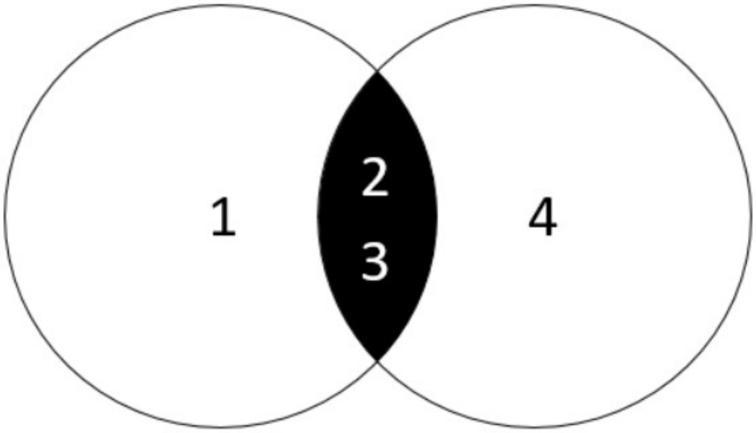
The INTERSECT operator compares the result of two queries and returns the distinct rows that are output by both left and right queries.

To use the INTERSECT operator for two queries, the following rules are applied:

1. The order and the number of columns must be the same.

2. The data types of the corresponding columns must be compatible.

The following diagram illustrates the INTERSECT operator.



The left query produces a result set of (1,2,3).

The right query returns a result set of (2,3,4).

The INTERSECT operator returns the distinct rows of both result sets which include (2,3).

Unlike the UNION operator, the INTERSECT operator returns the intersection between two circles.

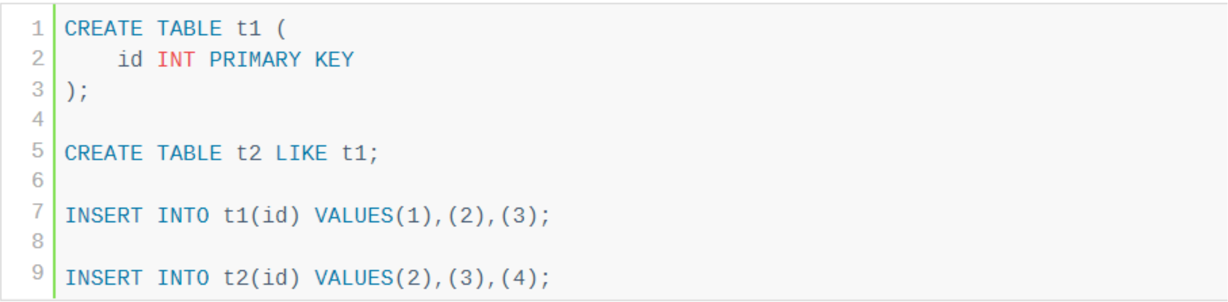
Note that SQL standard has three set operators that include UNION, INTERSECT, and MINUS.

MySQL INTERSECT simulation

Unfortunately, MySQL does not support the INTERSECT operator. However, you can simulate the INTERSECT operator.

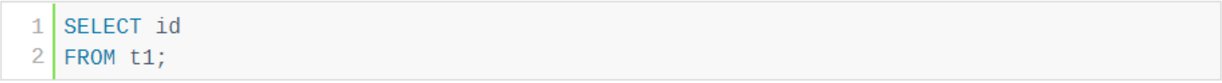
Let’s create some sample data for the demonstration.

The following statements create tables t1 and t2, and then insert data into both tables.

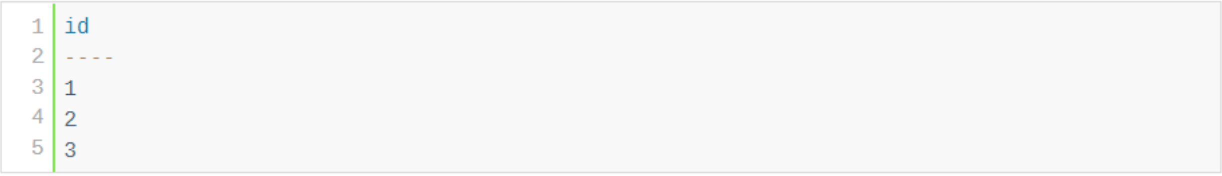


```
CREATE TABLE t1 (
    id INT PRIMARY KEY
);

CREATE TABLE t2 LIKE t1;

INSERT INTO t1(id) VALUES(1),(2),(3);

INSERT INTO t2(id) VALUES(2),(3),(4);
```

The following query returns rows from the t1 table.

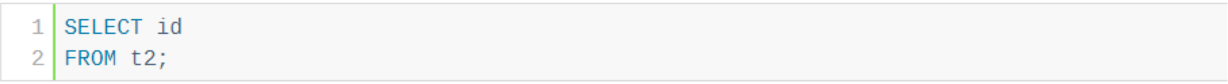


```
SELECT id
FROM t1;
```

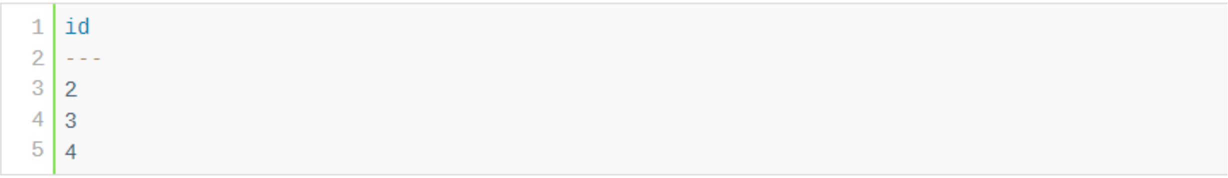


```
id
----
1
2
3
```

The following query returns the rows from the t2 table:



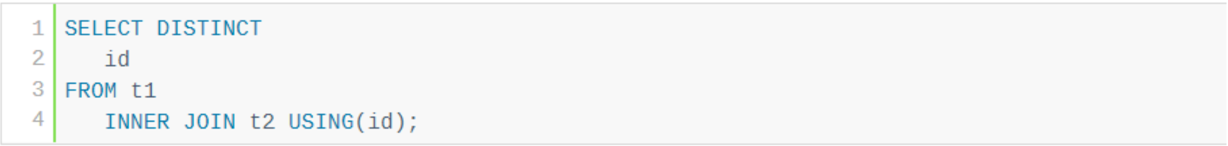
```
SELECT id
FROM t2;
```



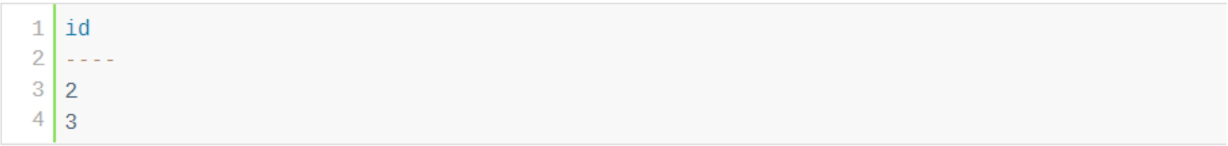
```
id
---
2
3
4
```

Simulate MySQL INTERSECT operator using DISTINCT operator and INNER JOIN clause.

The following statement uses DISTINCT operator and INNER JOIN clause to return the distinct rows in both tables:



```
SELECT DISTINCT
   id
FROM t1
   INNER JOIN t2 USING(id);
```



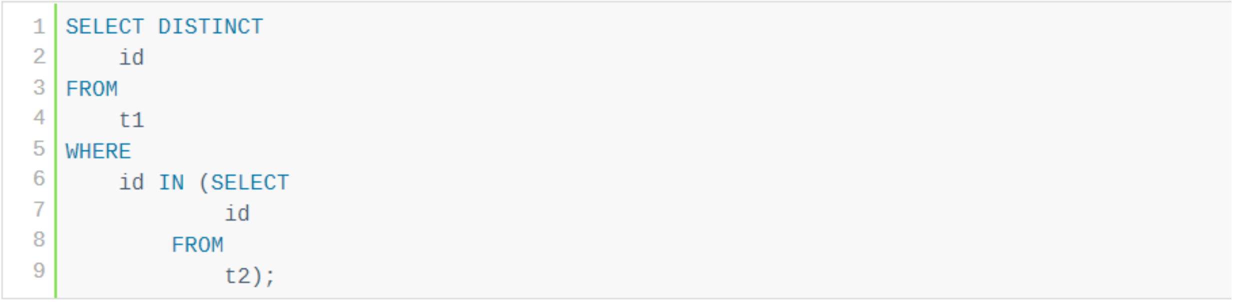
```
id
----
2
3
```

How it works.

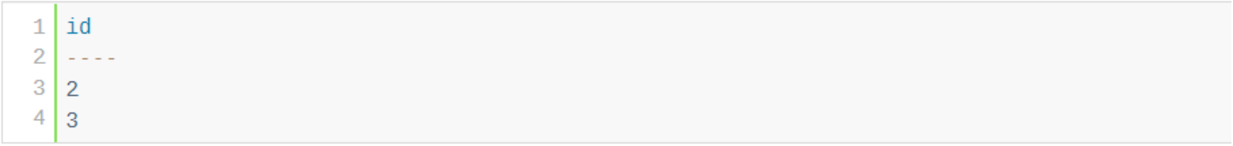
1. The INNER JOIN clause returns rows from both left and right tables.
2. The DISTINCT operator removes the duplicate rows.

Simulate MySQL INTERSECT operator using IN operator and subquery

The following statement uses the IN operator and a subquery to return the intersection of the two result sets.



```
SELECT DISTINCT
    id
FROM
    t1
WHERE
    id IN (SELECT
            id
        FROM
            t2);
```



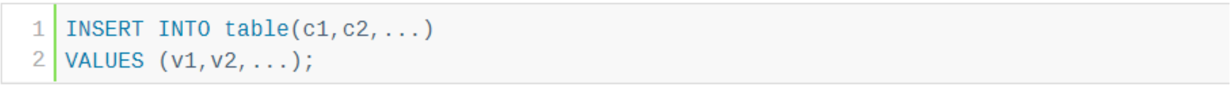
```
id
----
2
3
```

How it works.

1. The subquery returns the first result set.
2. The outer query uses the IN operator to select only values that are in the first result set. The DISTINCT operator ensures that only distinct values are selected.

MySQL Insert

The INSERT statement allows you to insert one or more rows into a table. The following illustrates the syntax of the INSERT statement:



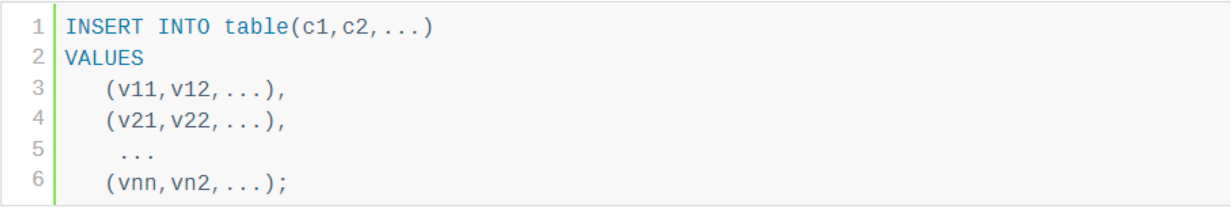
```
INSERT INTO table(c1,c2,...)
VALUES (v1,v2,...);
```

In this syntax,

- First, specify the table name and a list of comma-separated columns inside parentheses after the INSERT INTO clause.
- Then, put a comma-separated list of values of the corresponding columns inside the parentheses following the VALUES keyword.

The number of columns and values must be the same. In addition, the positions of columns must be corresponding with the positions of their values.

To add multiple rows into a table using a single INSERT statement, you use the following syntax:

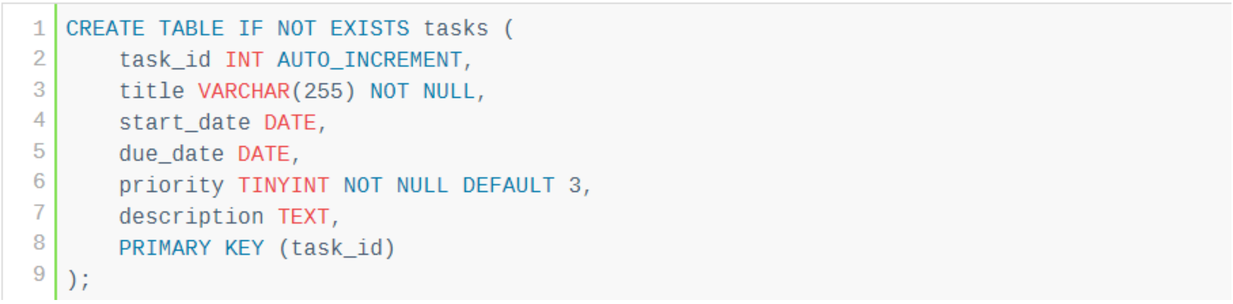


```
INSERT INTO table(c1,c2,...)
VALUES
   (v11,v12,...),
   (v21,v22,...),
    ...
   (vnn,vn2,...);
```

In this syntax, rows are separated by commas in the VALUES clause.

MySQL INSERT examples

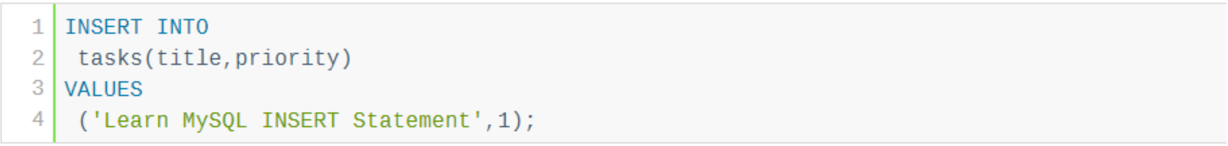
Let’s create a new table named tasks for practicing the INSERT statement.



```
CREATE TABLE IF NOT EXISTS tasks (
    task_id INT AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    start_date DATE,
    due_date DATE,
    priority TINYINT NOT NULL DEFAULT 3,
    description TEXT,
    PRIMARY KEY (task_id)
);
```

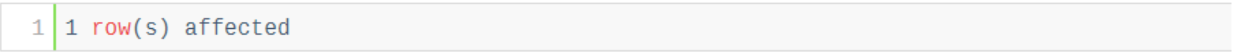
Using simple INSERT statement example

The following statement adds a new row to the tasks table:



```
INSERT INTO
 tasks(title,priority)
VALUES
 ('Learn MySQL INSERT Statement',1);
```

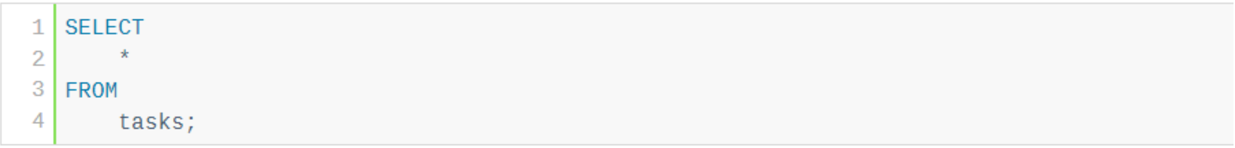
MySQL returns the following message after the statement executed:



```
1 row(s) affected
```

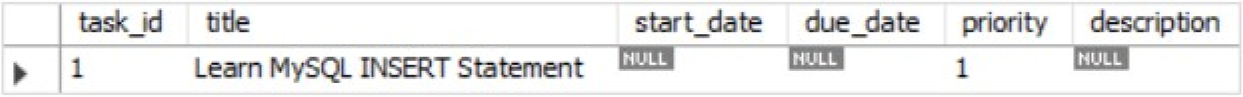
It means that one row has been inserted into the tasks table successfully.

You can verify it by using the following query:



```
SELECT
    *
FROM
    tasks;
```

Here is the output:



	task_id	title	start_date	due_date	priority	description
▶	1	Learn MySQL INSERT Statement	NULL	NULL	1	NULL

In this example, we specified the values for only title and priority columns. For other columns, MySQL uses the default values.

The task_id column is an auto-increment column. It means that MySQL generates a sequential integer whenever a row is added to the table.

The start_date, due_date, and description columns use NULL as the default value, therefore, MySQL uses NULL to insert into those columns if you don't specify their values in the INSERT statement.

Inserting rows using default values

If you want to insert a default value into a column, you have two ways:

- First, ignore both column name and its value in the INSERT statement.
- Second, specify the column name in the INSERT INTO clause and use the DEFAULT keyword in the VALUES clause.

The following example demonstrates how the second way:

```
INSERT INTO
 tasks(title,priority)
VALUES
 ('Understanding DEFAULT keyword in INSERT statement',DEFAULT);
```

In this example, we specified the priority column and the DEFAULT keyword.

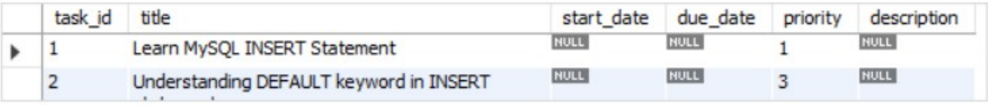
Because the default value for the priority column is 3 as declared in the table definition:

```
priority TINYINT NOT NULL DEFAULT 3
```

MySQL uses the number 3 to insert into the priority column.

The following shows the contents of the tasks table after the insert:

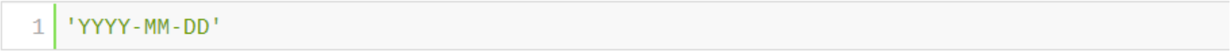
```
SELECT
    *
FROM
    tasks;
```



	task_id	title	start_date	due_date	priority	description
▶	1	Learn MySQL INSERT Statement	NULL	NULL	1	NULL
	2	Understanding DEFAULT keyword in INSERT	NULL	NULL	3	NULL

Inserting dates into the table

To insert a literal date value into a column, you use the following format:

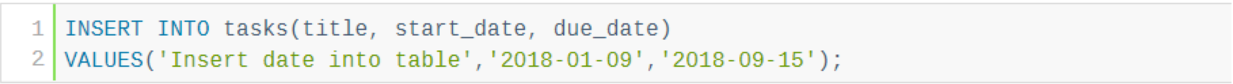


```
'YYYY-MM-DD'
```

In this format:

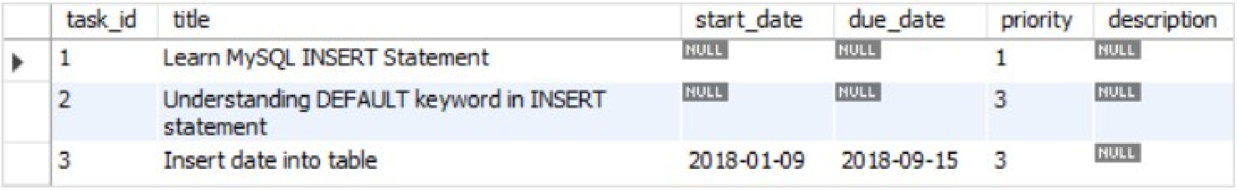
- YYYY represents a four-digit year e.g., 2018.
- MM represents a two-digit month e.g., 01, 02, and 12.
- DD represents a two-digit day e.g., 01, 02, 30.

The following example adds a new row to the tasks table with the start and due date values:



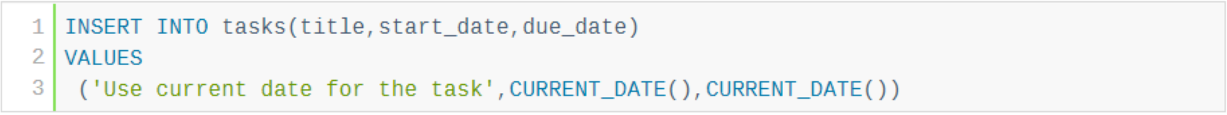
```
INSERT INTO tasks(title, start_date, due_date)
VALUES('Insert date into table','2018-01-09','2018-09-15');
```

The following picture shows the contents of the tasks table after the insert:



	task_id	title	start_date	due_date	priority	description
▶	1	Learn MySQL INSERT Statement	NULL	NULL	1	NULL
	2	Understanding DEFAULT keyword in INSERT statement	NULL	NULL	3	NULL
	3	Insert date into table	2018-01-09	2018-09-15	3	NULL

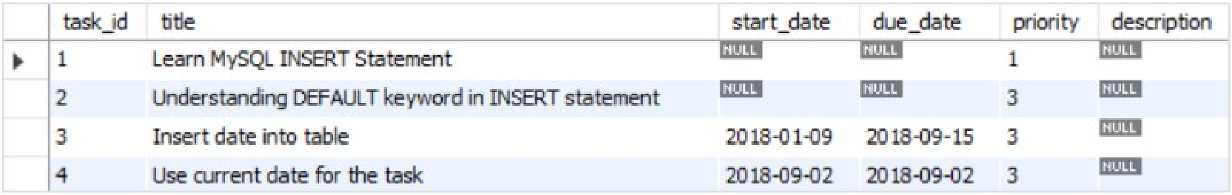
It is possible to use expressions in the VALUES clause. For example, the following statement adds a new task using the current date for start date and due date columns:



```
INSERT INTO tasks(title,start_date,due_date)
VALUES
 ('Use current date for the task',CURRENT_DATE(),CURRENT_DATE())
```

In this example, we used the CURRENT_DATE() function as the values for the start_date and due_date columns. Note that the CURRENT_DATE() function is a date function that returns the current system date.

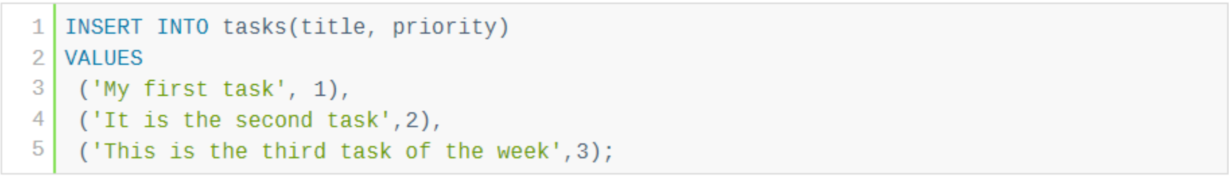
Here are the contents of the tasks table after insert:



	task_id	title	start_date	due_date	priority	description
▶	1	Learn MySQL INSERT Statement	NULL	NULL	1	NULL
	2	Understanding DEFAULT keyword in INSERT statement	NULL	NULL	3	NULL
	3	Insert date into table	2018-01-09	2018-09-15	3	NULL
	4	Use current date for the task	2018-09-02	2018-09-02	3	NULL

Inserting multiple rows example

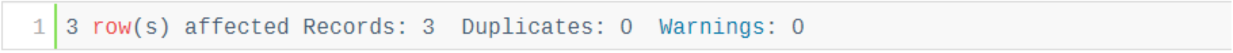
The following statement adds three rows to the tasks table:



```
INSERT INTO tasks(title, priority)
VALUES
 ('My first task', 1),
 ('It is the second task',2),
 ('This is the third task of the week',3);
```

In this example, each row data is specified as a list of values in the VALUES clause.

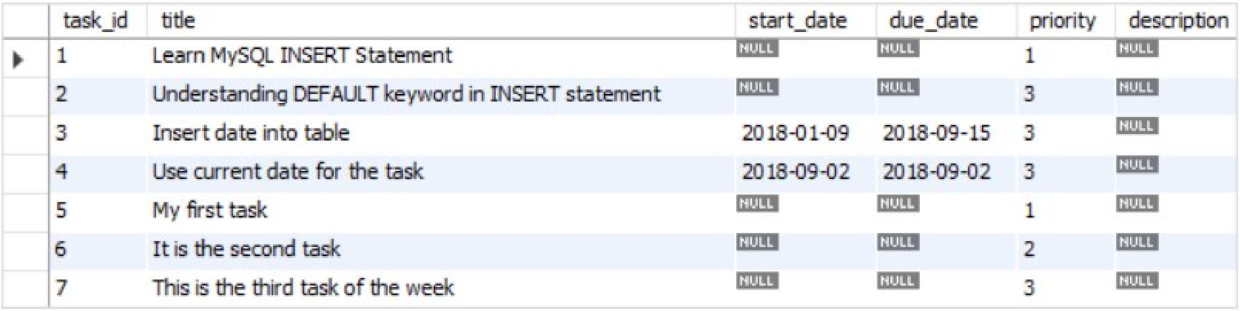
MySQL returns the following message:



```
3 row(s) affected Records: 3  Duplicates: 0  Warnings: 0
```

It means that three rows have been inserted successfully with no duplicates or warnings.

The tasks table has the following data after insert:

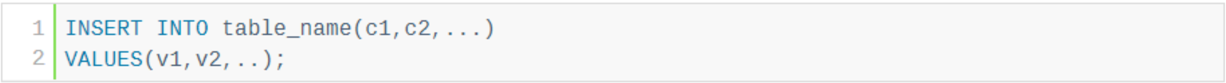


	task_id	title	start_date	due_date	priority	description
▶	1	Learn MySQL INSERT Statement	NULL	NULL	1	NULL
	2	Understanding DEFAULT keyword in INSERT statement	NULL	NULL	3	NULL
	3	Insert date into table	2018-01-09	2018-09-15	3	NULL
	4	Use current date for the task	2018-09-02	2018-09-02	3	NULL
	5	My first task	NULL	NULL	1	NULL
	6	It is the second task	NULL	NULL	2	NULL
	7	This is the third task of the week	NULL	NULL	3	NULL

In this tutorial, you have learned how to use the MySQL INSERT statement to add one or more rows into a table.

MySQL INSERT INTO SELECT

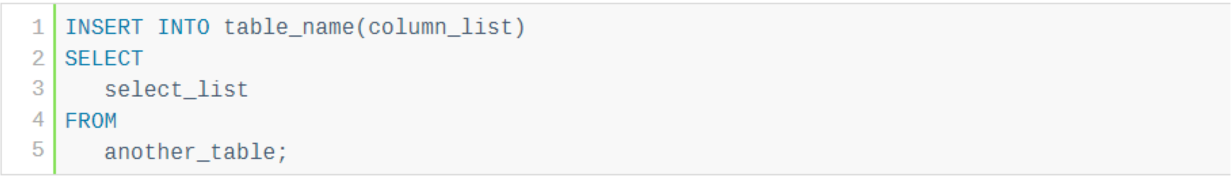
In the previous tutorial, you learned how to add one or more rows into a table using the INSERT statement with a list of column values specified in the VALUES clause.



```
INSERT INTO table_name(c1,c2,...)
VALUES(v1,v2,..);
```

Besides using row values in the VALUES clause, you can use the result of a SELECT statement as the data source for the INSERT statement.

The following illustrates the syntax of the INSERT INTO SELECT statement:



```
INSERT INTO table_name(column_list)
SELECT
   select_list
FROM
   another_table;
```

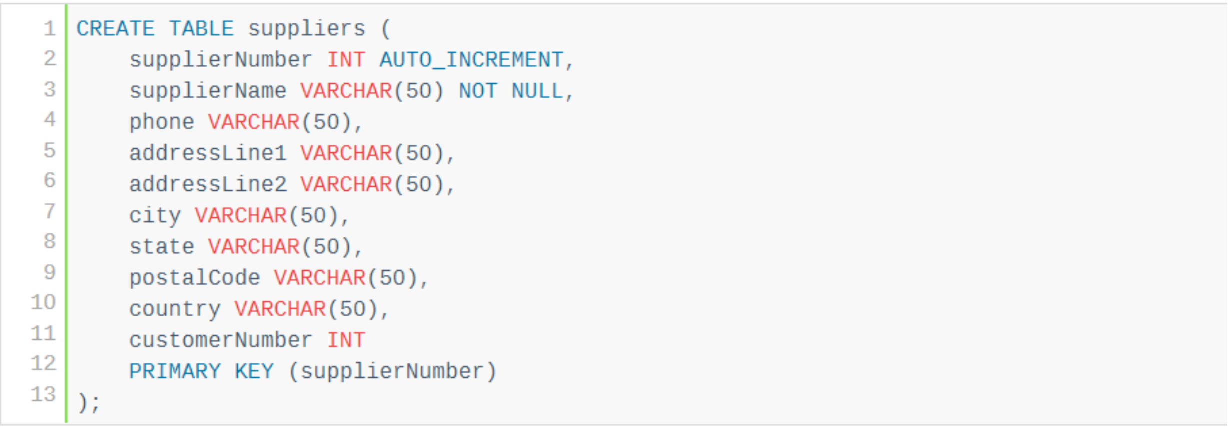
As you can see, instead of using the VALUES clause, you can use a SELECT statement.

The SELECT statement can retrieve data from one or more tables.

The INSERT INTO SELECT statement is very useful when you want to copy data from other tables to a table.

MySQL INSERT INTO SELECT example

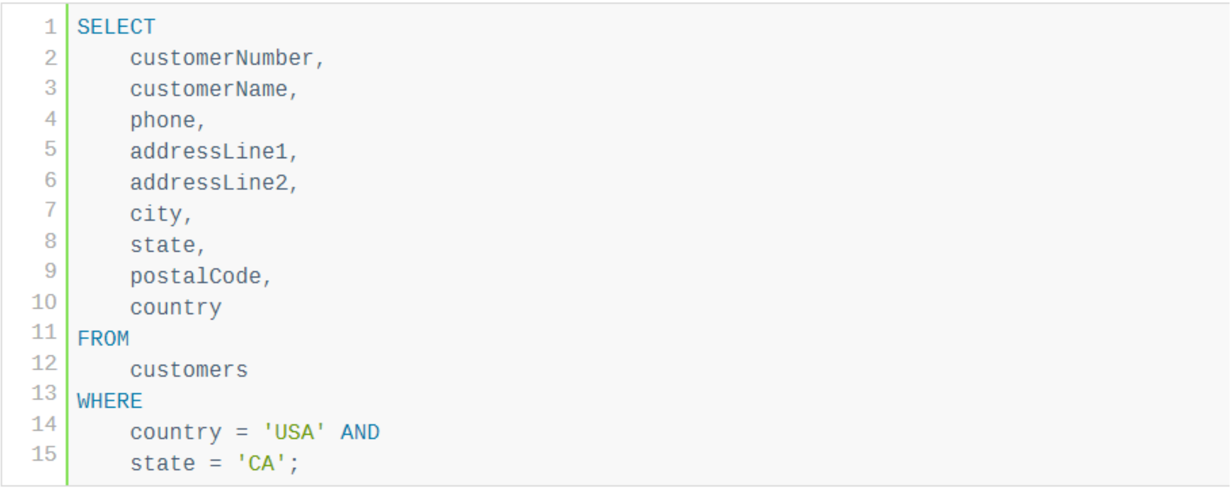
Suppose we have the following suppliers table with the following structure:



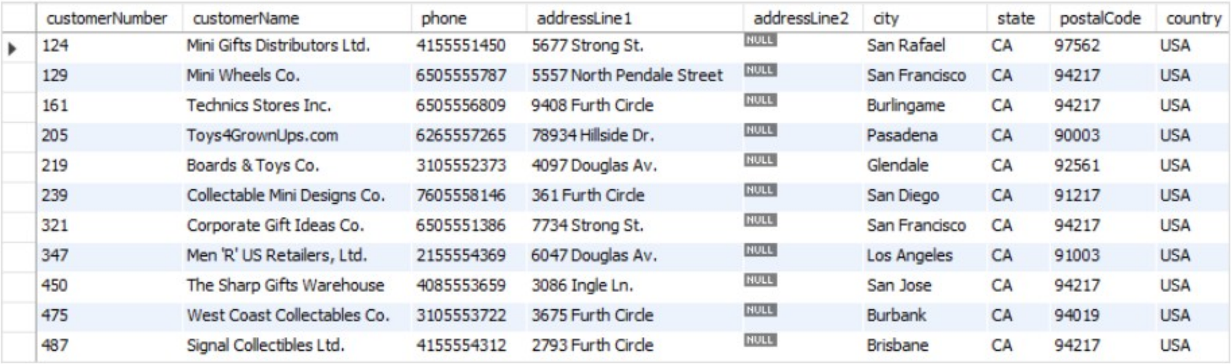
```
CREATE TABLE suppliers (
    supplierNumber INT AUTO_INCREMENT,
    supplierName VARCHAR(50) NOT NULL,
    phone VARCHAR(50),
    addressLine1 VARCHAR(50),
    addressLine2 VARCHAR(50),
    city VARCHAR(50),
    state VARCHAR(50),
    postalCode VARCHAR(50),
    country VARCHAR(50),
    customerNumber INT
    PRIMARY KEY (supplierNumber)
);
```

Note that you will learn how to create a new table in the future tutorial. For now, you just need to execute this statement to create the suppliers table.

Because of the new contracts, all customers from California, USA become the company's suppliers. The following query finds all customers in California, USA:

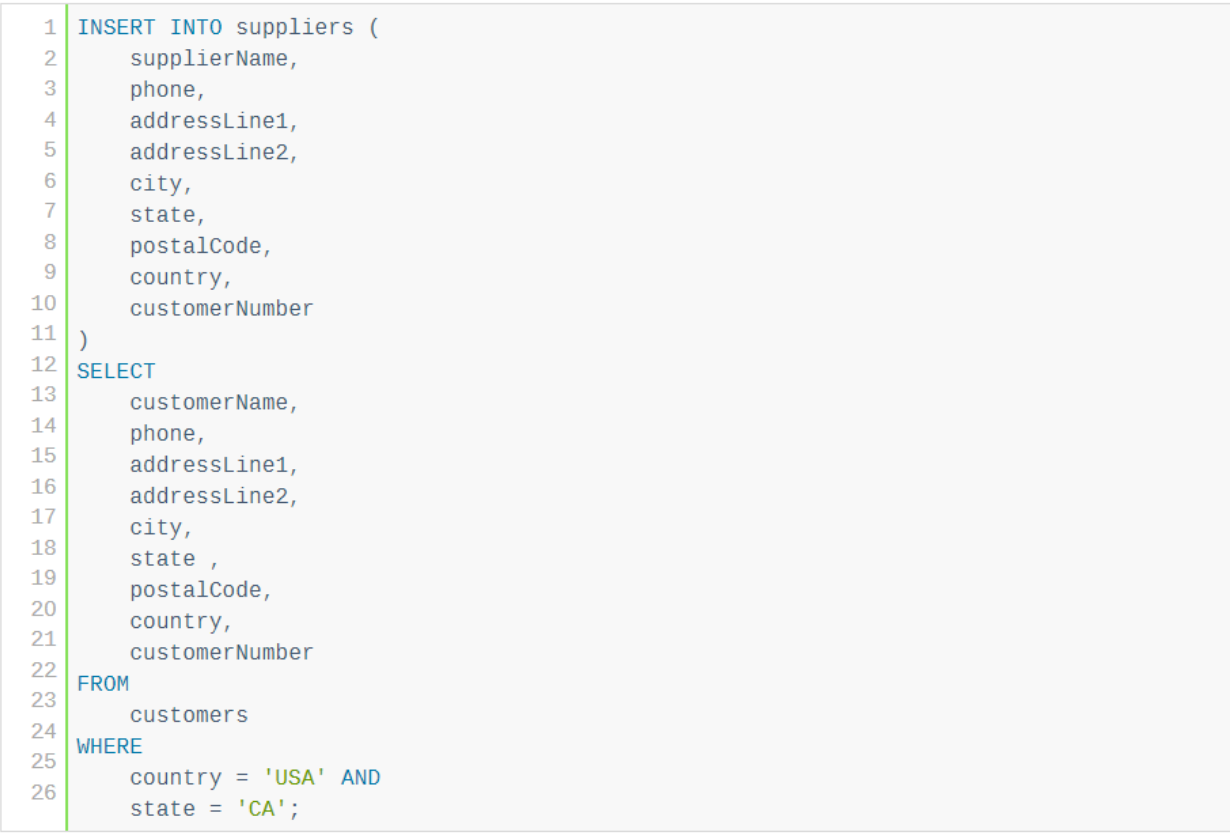


```
SELECT
    customerNumber,
    customerName,
    phone,
    addressLine1,
    addressLine2,
    city,
    state,
    postalCode,
    country
FROM
    customers
WHERE
    country = 'USA' AND
    state = 'CA';
```



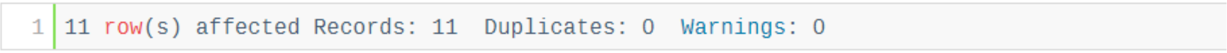
customerNumber	customerName	phone	addressLine1	addressLine2	city	state	postalCode	country
124	Mini Gifts Distributors Ltd.	4155551450	5677 Strong St.	NULL	San Rafael	CA	97562	USA
129	Mini Wheels Co.	6505555787	5557 North Pendale Street	NULL	San Francisco	CA	94217	USA
161	Technics Stores Inc.	6505556809	9408 Furth Circle	NULL	Burlingame	CA	94217	USA
205	Toys4GrownUps.com	6265557265	78934 Hillside Dr.	NULL	Pasadena	CA	90003	USA
219	Boards & Toys Co.	3105552373	4097 Douglas Av.	NULL	Glendale	CA	92561	USA
239	Collectable Mini Designs Co.	7605558146	361 Furth Circle	NULL	San Diego	CA	91217	USA
321	Corporate Gift Ideas Co.	6505551386	7734 Strong St.	NULL	San Francisco	CA	94217	USA
347	Men 'R' US Retailers, Ltd.	2155554369	6047 Douglas Av.	NULL	Los Angeles	CA	91003	USA
450	The Sharp Gifts Warehouse	4085553659	3086 Ingle Ln.	NULL	San Jose	CA	94217	USA
475	West Coast Collectables Co.	3105553722	3675 Furth Circle	NULL	Burbank	CA	94019	USA
487	Signal Collectibles Ltd.	4155554312	2793 Furth Circle	NULL	Brisbane	CA	94217	USA

Now, you need to insert these customers from the customers table into the suppliers table. The following INSERT INTO SELECT statement helps you to do so:



```
INSERT INTO suppliers (
    supplierName,
    phone,
    addressLine1,
    addressLine2,
    city,
    state,
    postalCode,
    country,
    customerNumber
)
SELECT
    customerName,
    phone,
    addressLine1,
    addressLine2,
    city,
    state ,
    postalCode,
    country,
    customerNumber
FROM
    customers
WHERE
    country = 'USA' AND
    state = 'CA';
```

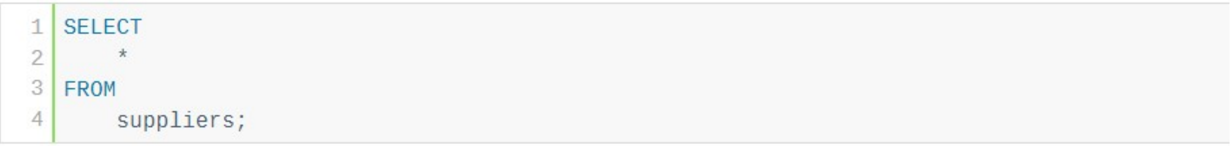
MySQL returned the following message:



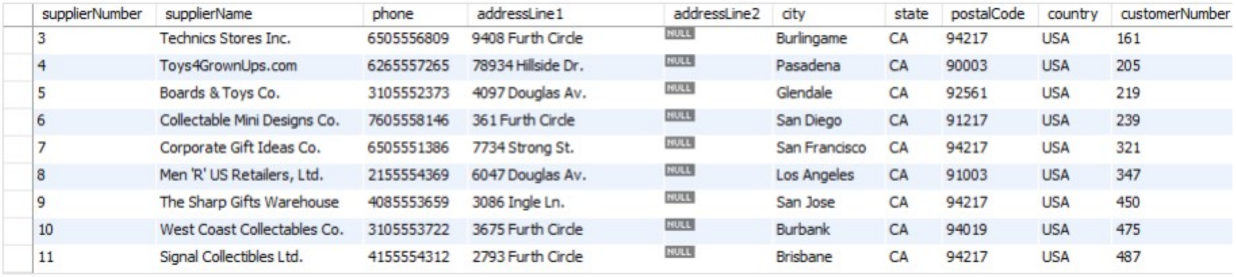
```
11 row(s) affected Records: 11  Duplicates: 0  Warnings: 0
```

It means that 11 rows from the customers table have been inserted into the suppliers table successfully with no duplicates or warnings.

The following query returns the data from the suppliers table after insert:



```
SELECT
    *
FROM
    suppliers;
```



supplierNumber	supplierName	phone	addressLine1	addressLine2	city	state	postalCode	country	customerNumber
3	Technics Stores Inc.	6505556809	9408 Furth Circle	NULL	Burlingame	CA	94217	USA	161
4	Toys4GrownUps.com	6265557265	78934 Hillside Dr.	NULL	Pasadena	CA	90003	USA	205
5	Boards & Toys Co.	3105552373	4097 Douglas Av.	NULL	Glendale	CA	92561	USA	219
6	Collectable Mini Designs Co.	7605558146	361 Furth Circle	NULL	San Diego	CA	91217	USA	239
7	Corporate Gift Ideas Co.	6505551386	7734 Strong St.	NULL	San Francisco	CA	94217	USA	321
8	Men 'R' US Retailers, Ltd.	2155554369	6047 Douglas Av.	NULL	Los Angeles	CA	91003	USA	347
9	The Sharp Gifts Warehouse	4085553659	3086 Ingle Ln.	NULL	San Jose	CA	94217	USA	450
10	West Coast Collectables Co.	3105553722	3675 Furth Circle	NULL	Burbank	CA	94019	USA	475
11	Signal Collectibles Ltd.	4155554312	2793 Furth Circle	NULL	Brisbane	CA	94217	USA	487

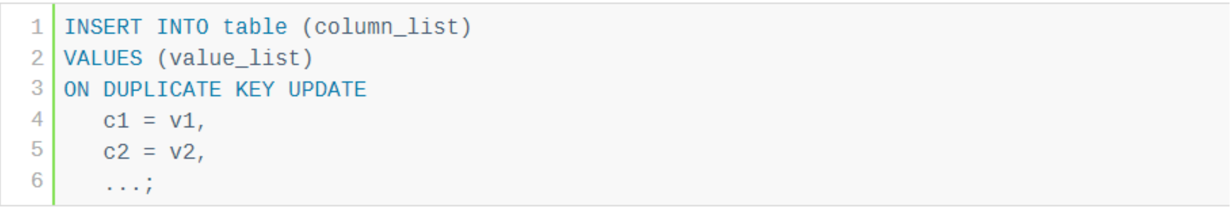
MySQL INSERT ON DUPLICATE KEY UPDATE

The INSERT ON DUPLICATE KEY UPDATE is a MySQL's extension to the SQL standard's INSERT statement.

When you insert a new row into a table if the row causes a duplicate in UNIQUE index or PRIMARY KEY, MySQL will issue an error.

However, if you specify the ON DUPLICATE KEY UPDATE option in the INSERT statement, MySQL will update the existing row with the new values instead.

The syntax of INSERT ON DUPLICATE KEY UPDATE statement is as follows:



```
INSERT INTO table (column_list)
VALUES (value_list)
ON DUPLICATE KEY UPDATE
   c1 = v1,
   c2 = v2,
   ...;
```

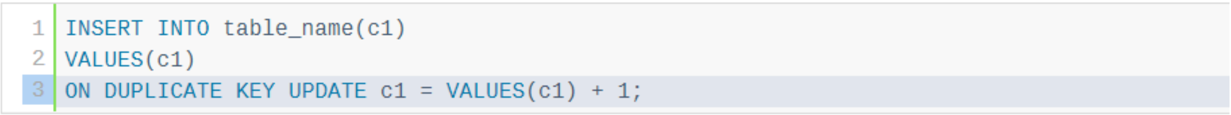
The only addition to the INSERT statement is the ON DUPLICATE KEY UPDATE clause where you specify a list of column-value-pair assignments in case of duplicate.

Basically, the statement first tries to insert a new row into the table. If a duplicate error occurs, it will update the existing row with the value specified in the ON DUPLICATE KEY UPDATE clause.

MySQL returns the number of affected-rows based on the action it performs:

- If the new row is inserted, the number of affected-rows is 1.
- If the existing row is updated, the number of affected-rows is 2.
- If the existing row is updated using its current values, the number of affected-rows is 0.

To use the values from the INSERT clause in the DUPLICATE KEY UPDATE clause, you use the VALUES()function as follows:



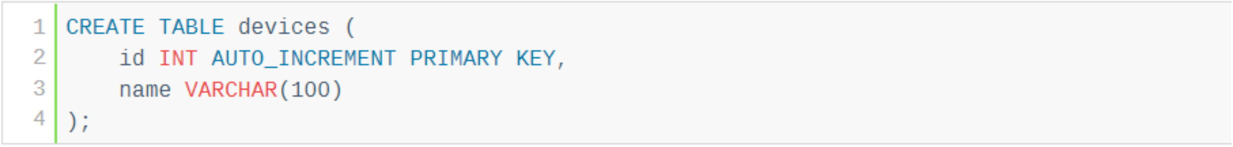
```
INSERT INTO table_name(c1)
VALUES(c1)
ON DUPLICATE KEY UPDATE c1 = VALUES(c1) + 1;
```

The statement above sets the value of the c1 to its current value specified by the expression VALUES(c1) plus 1 if there is a duplicate in UNIQUE index or PRIMARY KEY.

MySQL INSERT ON DUPLICATE KEY UPDATE example

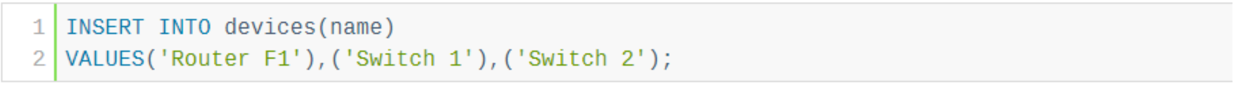
Let’s take a look at an example of using the INSERT ON DUPLICATE KEY UPDATE to understand how it works.

First, create a table named devices to store the network devices.



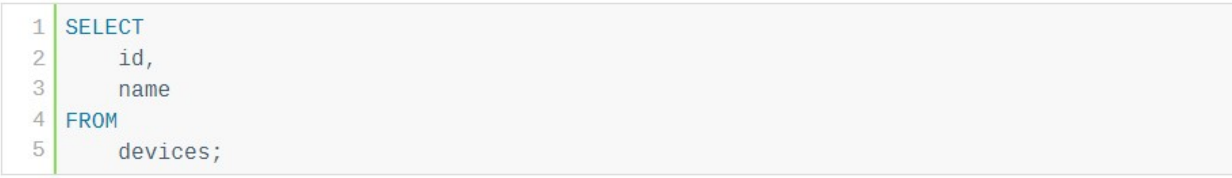
```
CREATE TABLE devices (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100)
);
```

Next, insert rows into the devices table.

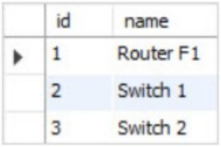


```
INSERT INTO devices(name)
VALUES('Router F1'),('Switch 1'),('Switch 2');
```

Then, query the data from the devices table to verify the insert:



```
SELECT
    id,
    name
FROM
    devices;
```

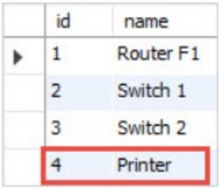


id	name
1	Router F1
2	Switch 1
3	Switch 2

Now, we have three rows in the devices table.

After that, insert one more row into the devices table.

```
INSERT INTO
   devices(name)
VALUES
   ('Printer')
ON DUPLICATE KEY UPDATE name = 'Printer';
```



	id	name
▶	1	Router F1
	2	Switch 1
	3	Switch 2
	4	Printer

Because there is no duplicate, MySQL inserts a new row into the devices table. The statement above has the same effect as the following statement:

```
INSERT INTO devices(name)
VALUES ('Printer');
```

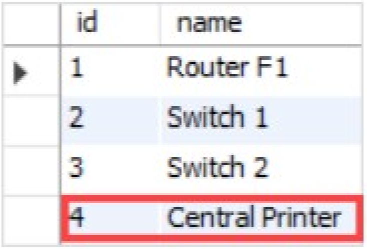
Finally, insert a row with a duplicate value in the id column.

```
INSERT INTO devices(id,name)
VALUES
   (4,'Printer')
ON DUPLICATE KEY UPDATE name = 'Central Printer';
```

MySQL issues the following message:

```
2 row(s) affected
```

Because a row with id 4 already exists in the devices table, the statement updates the name from Printer to Central Printer.



	id	name
▶	1	Router F1
	2	Switch 1
	3	Switch 2
	4	Central Printer

MySQL INSERT IGNORE Statement

When you use the INSERT statement to add multiple rows to a table and if an error occurs during the processing, MySQL terminates the statement and returns an error. As the result, no rows are inserted into the table.

However, if you use the INSERT IGNORE statement, the rows with invalid data that cause the error are ignored and the rows with valid data are inserted into the table.

The syntax of the INSERT IGNORE statement is as follows:

```
INSERT IGNORE INTO table(column_list)
VALUES( value_list),
      ( value_list),
      ...
```

Note that the IGNORE clause is an extension of MySQL to the SQL standard.

MySQL INSERT IGNORE example

We will create a new table called subscribers for the demonstration.

```
CREATE TABLE subscribers (
    id INT PRIMARY KEY AUTO_INCREMENT,
    email VARCHAR(50) NOT NULL UNIQUE
);
```

The UNIQUE constraint ensures that no duplicate email exists in the email column.

The following statement inserts a new row into the subscribers table:

```
INSERT INTO subscribers(email)
VALUES('john.doe@gmail.com');
```

It worked as expected.

Let’s execute another statement that inserts two rows into the subscribers table:

```
INSERT INTO subscribers(email)
VALUES('john.doe@gmail.com'),
      ('jane.smith@ibm.com');
```

It returns an error.

```
Error Code: 1062. Duplicate entry 'john.doe@gmail.com' for key 'email'
```

As indicated in the error message, the email john.doe@gmail.com violates the UNIQUE constraint.

However, if you use the INSERT IGNORE statement instead.

```
INSERT IGNORE INTO subscribers(email)
VALUES('john.doe@gmail.com'),
      ('jane.smith@ibm.com');
```

MySQL returned a message indicating that one row was inserted and the other row was ignored.

```
1 row(s) affected, 1 warning(s): 1062 Duplicate entry 'john.doe@gmail.com' for key 'email'
Records: 2  Duplicates: 1  Warnings: 1
```

To find the detail of the warning, you can use the SHOW WARNINGS command as shown below:

```
SHOW WARNINGS;
```

In conclusion, when you use the INSERT IGNORE statement, instead of issuing an error, MySQL issued a warning in case an error occurs.

If you query data from subscribers table, you will find that only one row was actually inserted and the row that causes the error was not.

MySQL INSERT IGNORE and STRICT mode

When the strict mode is on, MySQL returns an error and aborts the INSERT statement if you try to insert invalid values into a table.

However, if you use the INSERT IGNORE statement, MySQL will issue a warning instead of an error. In addition, it will try to adjust the values to make them valid before adding the value to the table.

Consider the following example.

First, we create a new table named tokens:

```
CREATE TABLE tokens (
    s VARCHAR(6)
);
```

In this table, the column s accepts only string whose lengths are less than or equal to six.

Second, insert a string whose length is seven into the tokens table.

```
INSERT INTO tokens VALUES('abcdefg');
```

MySQL issued the following error because the strict mode is on.

```
Error Code: 1406. Data too long for column 's' at row 1
```

Third, use the INSERT IGNORE statement to insert the same string.

```
INSERT IGNORE INTO tokens VALUES('abcdefg');
```

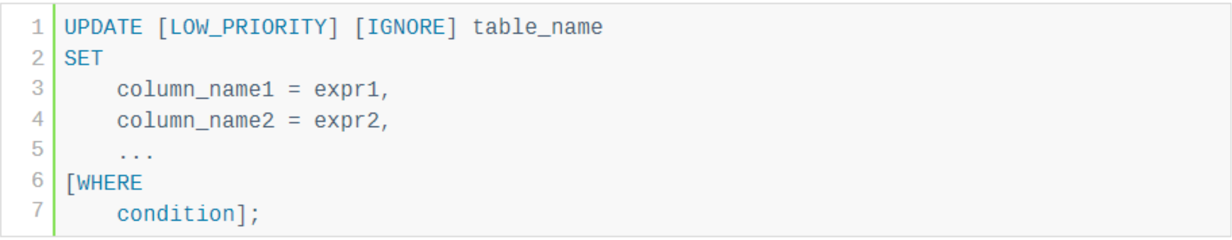
MySQL truncated data before inserting it into the tokens table. In addition, it issues a warning.

In this tutorial, you have learned how to use the MySQL INSERT IGNORE statement to insert rows into a table and ignore error for rows that cause errors.

MySQL UPDATE

You use the UPDATE statement to update existing data in a table. You can also use the UPDATE statement to change column values of a single row, a group of rows, or all rows in a table.

The following illustrates the syntax of the MySQL UPDATE statement:



```
UPDATE [LOW_PRIORITY] [IGNORE] table_name
SET
    column_name1 = expr1,
    column_name2 = expr2,
    ...
[WHERE
    condition];
```

In the UPDATE statement:

- First, specify the table name that you want to update data after the UPDATE keyword.
- Second, the SET clause specifies which column that you want to modify and the new values. To update multiple columns, you use a list of comma-separated assignments. You supply a value in each column's assignment in the form of a literal value, an expression, or a subquery.
- Third, specify which rows to be updated using a condition in the WHERE clause. The WHERE clause is optional. If you omit the WHERE clause, the UPDATE statement will update all rows in the table.

Notice that the WHERE clause is so important that you should not forget. Sometimes, you may want to change just one row; However, you may forget the WHERE clause and accidentally update all rows of the table.

MySQL supports two modifiers in the UPDATE statement.

1. The LOW_PRIORITY modifier instructs the UPDATE statement to delay the update until there is no connection reading data from the table. The LOW_PRIORITY takes effect for

the storage engines that use table-level locking only, for example, MyISAM, MERGE, MEMORY.

2. The IGNORE modifier enables the UPDATE statement to continue updating rows even if errors occurred. The rows that cause errors such as duplicate-key conflicts are not updated.

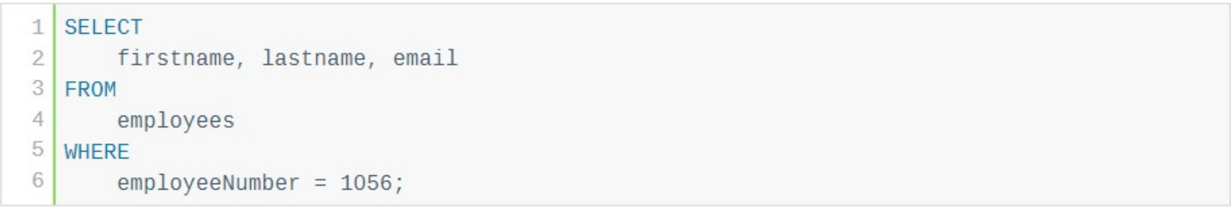
MySQL UPDATE examples

Let's practice the UPDATE statement with some tables in the MySQL sample database.

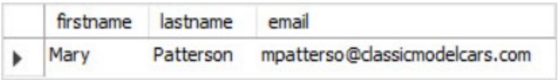
Using MySQL UPDATE to modify values in a single column example

In this example, we are going to update the email of Mary Patterson to the new email mary.patterso@classicmodelcars.com.

First, to ensure the update is successful, we query Mary's email from the employees table using the following SELECT statement:

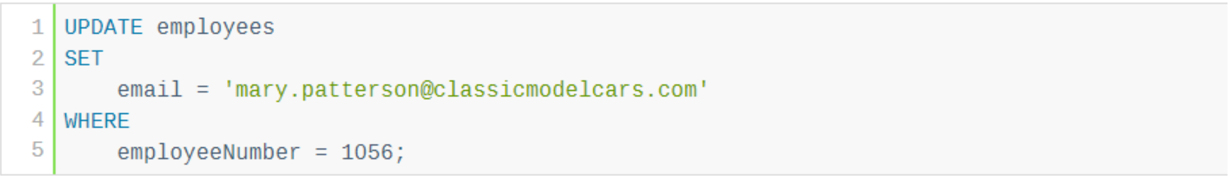


```
SELECT
    firstname, lastname, email
FROM
    employees
WHERE
    employeeNumber = 1056;
```



	firstname	lastname	email
▶	Mary	Patterson	mpatterso@classicmodelcars.com

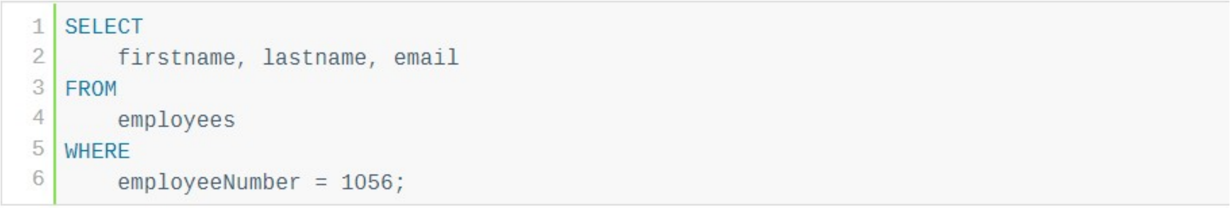
Second, we can update Mary's email to the new email mary.patterson@classicmodelcars.com using the UPDATE statement as shown in the following query:



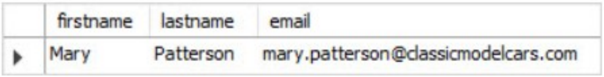
```
UPDATE employees
SET
    email = 'mary.patterson@classicmodelcars.com'
WHERE
    employeeNumber = 1056;
```

Because we just want to update one row, we use the WHERE clause to specify the row using the employee number 1056. The SET clause sets the value of the email column to the new email.

Third, we execute the SELECT statement again to verify the change.



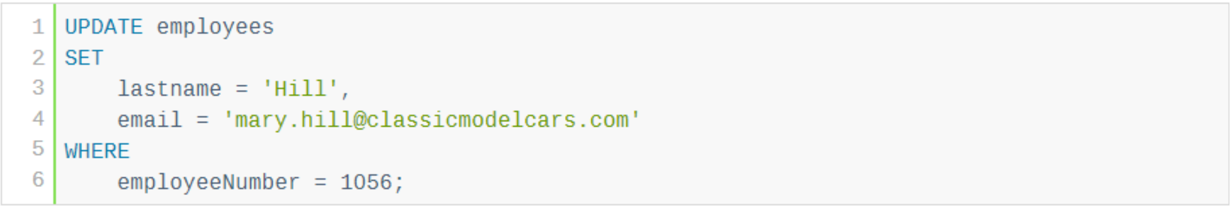
```
SELECT
    firstname, lastname, email
FROM
    employees
WHERE
    employeeNumber = 1056;
```



	firstname	lastname	email
▶	Mary	Patterson	mary.patterson@classicmodelcars.com

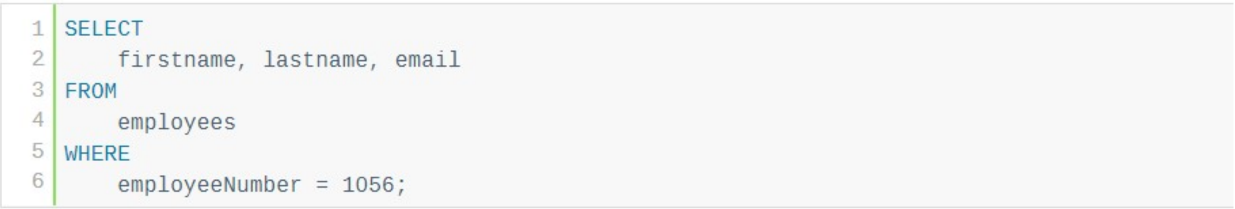
Using MySQL UPDATE to modify values in multiple columns

To update values in the multiple columns, you need to specify the assignments in the SET clause. For example, the following statement updates both last name and email columns of employee number 1056:

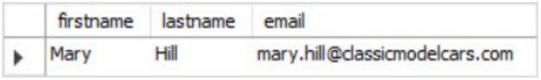


```
UPDATE employees
SET
    lastname = 'Hill',
    email = 'mary.hill@classicmodelcars.com'
WHERE
    employeeNumber = 1056;
```

Let’s check the changes:



```
SELECT
    firstname, lastname, email
FROM
    employees
WHERE
    employeeNumber = 1056;
```

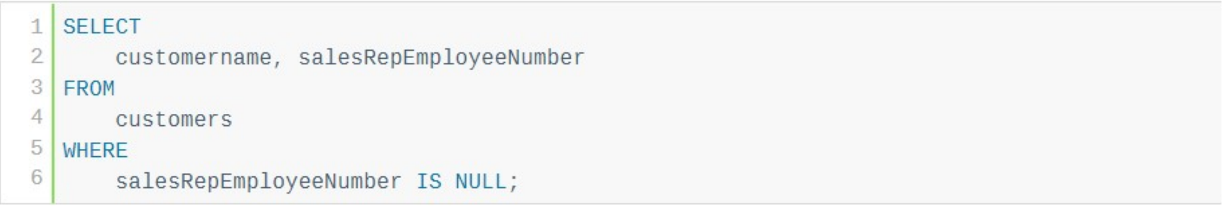


	firstname	lastname	email
▶	Mary	Hill	mary.hill@classicmodelcars.com

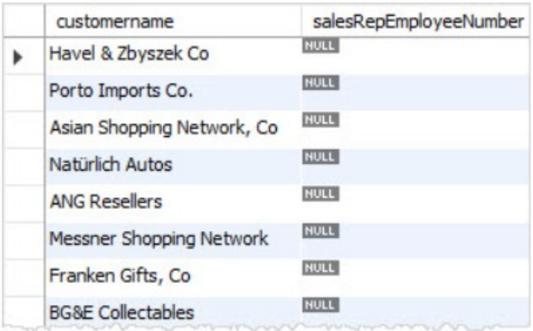
Using MySQL UPDATE to update rows returned by a SELECT statement

You can supply the values for the SET clause from a SELECT statement that queries data from other tables.

For example, in the customers table, some customers do not have any sale representative. The value of the column saleRepEmployeeNumber is NULL as follows:



```
SELECT
    customername, salesRepEmployeeNumber
FROM
    customers
WHERE
    salesRepEmployeeNumber IS NULL;
```

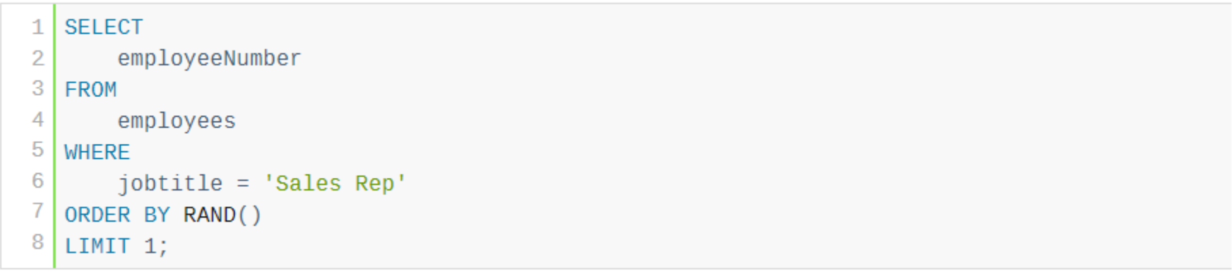


customername	salesRepEmployeeNumber
Havel & Zbyszek Co	NULL
Porto Imports Co.	NULL
Asian Shopping Network, Co	NULL
Natürlich Autos	NULL
ANG Resellers	NULL
Messner Shopping Network	NULL
Franken Gifts, Co	NULL
BG&E Collectables	NULL

We can take a sale representative and update for those customers.

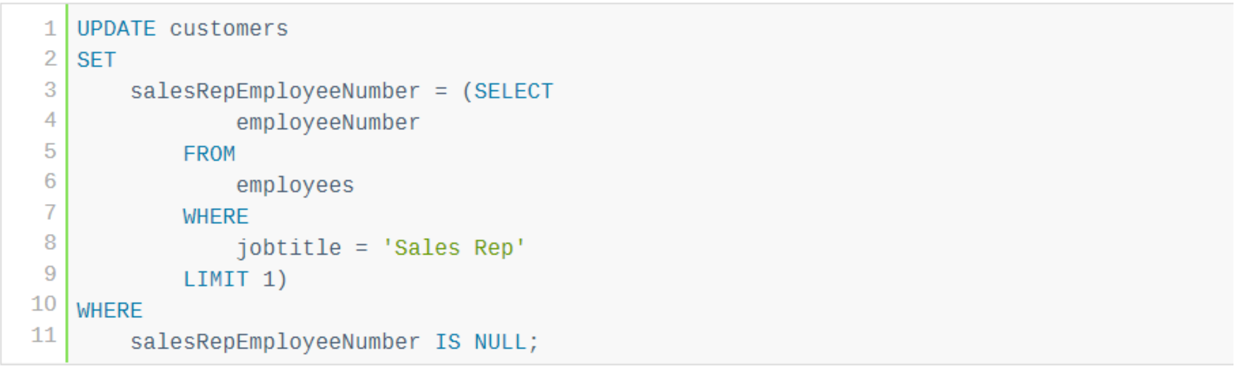
To do this, we can select a random employee whose job title is Sales Rep from the employees table and update it for the employees table.

This query selects a random employee from the employees table whose job title is the Sales Rep.



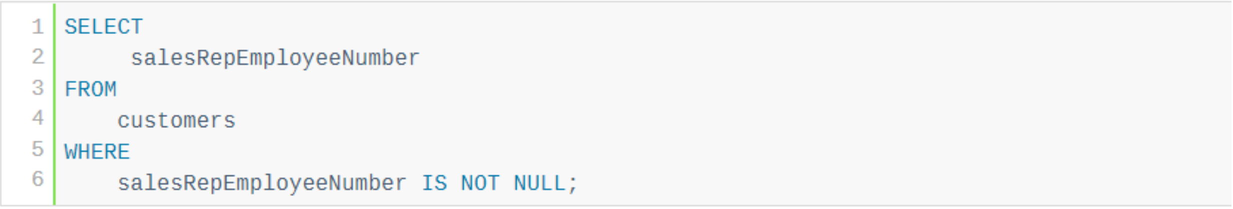
```
SELECT
    employeeNumber
FROM
    employees
WHERE
    jobtitle = 'Sales Rep'
ORDER BY RAND()
LIMIT 1;
```

To update the sales representative employee number column in the customers table, we place the query above in the SET clause of the UPDATE statement as follows:



```
UPDATE customers
SET
    salesRepEmployeeNumber = (SELECT
            employeeNumber
        FROM
            employees
        WHERE
            jobtitle = 'Sales Rep'
        LIMIT 1)
WHERE
    salesRepEmployeeNumber IS NULL;
```

If you query data from the employees table, you will see that every customer has a sales representative. In other words, the following query returns no row.



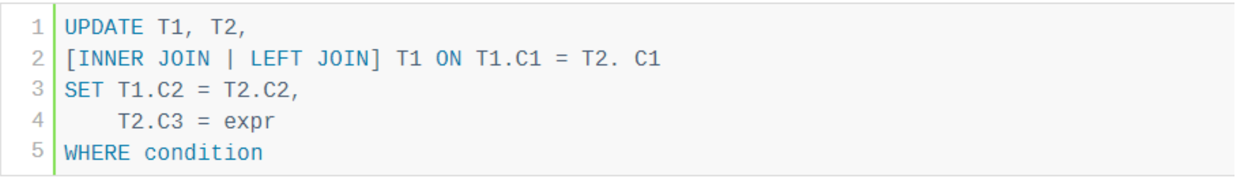
```
SELECT
     salesRepEmployeeNumber
FROM
    customers
WHERE
    salesRepEmployeeNumber IS NOT NULL;
```

MySQL UPDATE JOIN

MySQL UPDATE JOIN syntax

You often use joins to query rows from a table that have (in the case of INNER JOIN) or may not have (in the case of LEFT JOIN) matching rows in another table. In MySQL, you can use the JOIN clauses in the UPDATE statement to perform the cross-table update.

The syntax of the MySQL UPDATE JOIN is as follows:

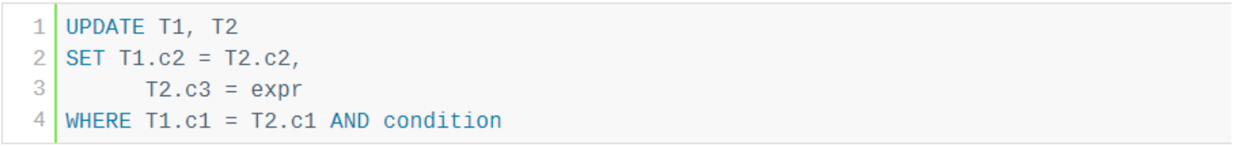


```
UPDATE T1, T2,
[INNER JOIN | LEFT JOIN] T1 ON T1.C1 = T2. C1
SET T1.C2 = T2.C2,
    T2.C3 = expr
WHERE condition
```

Let’s examine the MySQL UPDATE JOIN syntax in greater detail:

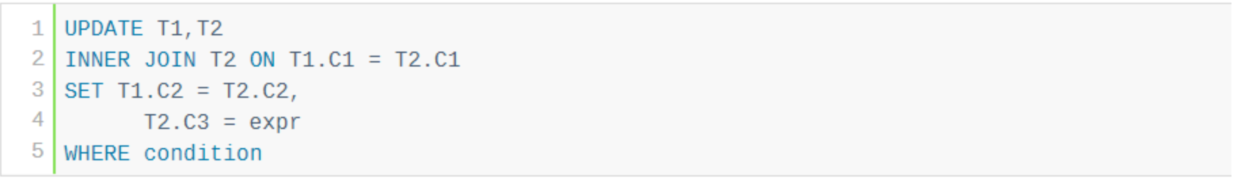
- First, specify the main table (T1) and the table that you want the main table to join to (T2) after the UPDATE clause. Notice that you must specify at least one table after the UPDATE clause. The data in the table that is not specified after the UPDATE clause will not be updated.
- Next, specify a kind of join you want to use i.e., either INNER JOIN or LEFT JOIN and a join predicate. The JOIN clause must appear right after the UPDATE clause.
- Then, assign new values to the columns in T1 and/or T2 tables that you want to update.
- After that, specify a condition in the WHERE clause to limit rows to rows for updating.

If you follow the UPDATE statement tutorial, you will notice that there is another way to update data cross-table using the following syntax:



```
UPDATE T1, T2
SET T1.c2 = T2.c2,
      T2.c3 = expr
WHERE T1.c1 = T2.c1 AND condition
```

This UPDATE statement works the same as UPDATE JOIN with an implicit INNER JOIN clause. It means you can rewrite the above statement as follows:



```
UPDATE T1,T2
INNER JOIN T2 ON T1.C1 = T2.C1
SET T1.C2 = T2.C2,
      T2.C3 = expr
WHERE condition
```

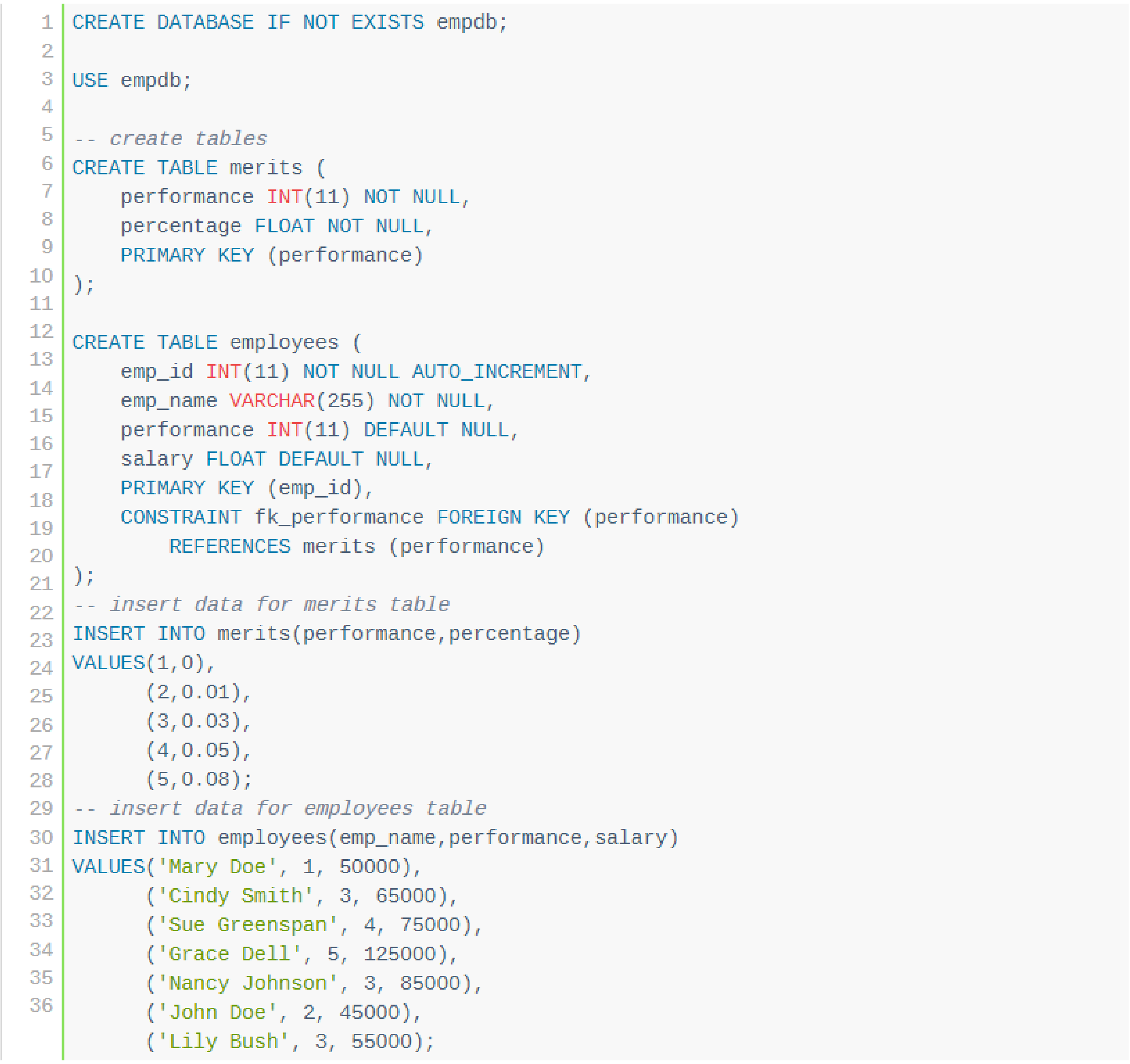
Let's take a look at some examples of using the UPDATE JOIN statement to having a better understanding.

MySQL UPDATE JOIN examples

We are going to use a new sample database named empdb in for demonstration. This sample database consists of two tables:

- The employees table stores employee data with employee id, name, performance, and salary.
- The merits table stores employee performance and merit's percentage.

The following statements create and load data in the empdb sample database:



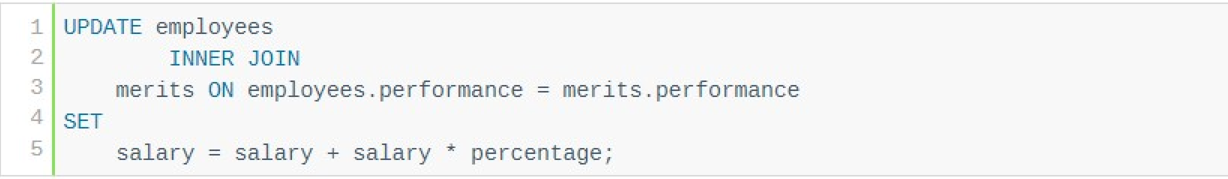
```
CREATE DATABASE IF NOT EXISTS empdb;

USE empdb;

-- create tables
CREATE TABLE merits (
    performance INT(11) NOT NULL,
    percentage FLOAT NOT NULL,
    PRIMARY KEY (performance)
);

CREATE TABLE employees (
    emp_id INT(11) NOT NULL AUTO_INCREMENT,
    emp_name VARCHAR(255) NOT NULL,
    performance INT(11) DEFAULT NULL,
    salary FLOAT DEFAULT NULL,
    PRIMARY KEY (emp_id),
    CONSTRAINT fk_performance FOREIGN KEY (performance)
        REFERENCES merits (performance)
);
-- insert data for merits table
INSERT INTO merits(performance,percentage)
VALUES(1,0),
      (2,0.01),
      (3,0.03),
      (4,0.05),
      (5,0.08);
-- insert data for employees table
INSERT INTO employees(emp_name,performance,salary)
VALUES('Mary Doe', 1, 50000),
      ('Cindy Smith', 3, 65000),
      ('Sue Greenspan', 4, 75000),
      ('Grace Dell', 5, 125000),
      ('Nancy Johnson', 3, 85000),
      ('John Doe', 2, 45000),
      ('Lily Bush', 3, 55000);
```

MySQL UPDATE JOIN example with INNER JOIN clause

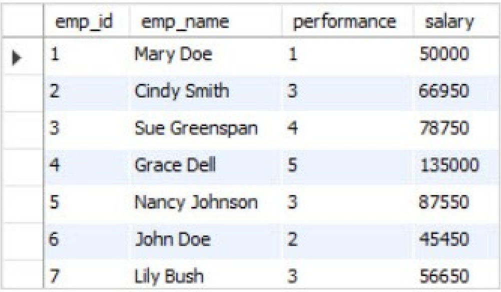
Suppose you want to adjust the salary of employees based on their performance.

The merit's percentages are stored in the merits table, therefore, you have to use the UPDATE INNER JOIN statement to adjust the salary of employees in the employees table based on the percentage stored in the merits table.

The link between the employees and merit tables is the performance field. See the following query:



```
UPDATE employees
        INNER JOIN
    merits ON employees.performance = merits.performance
SET
    salary = salary + salary * percentage;
```



emp_id	emp_name	performance	salary
1	Mary Doe	1	50000
2	Cindy Smith	3	66950
3	Sue Greenspan	4	78750
4	Grace Dell	5	135000
5	Nancy Johnson	3	87550
6	John Doe	2	45450
7	Lily Bush	3	56650

How the query works.

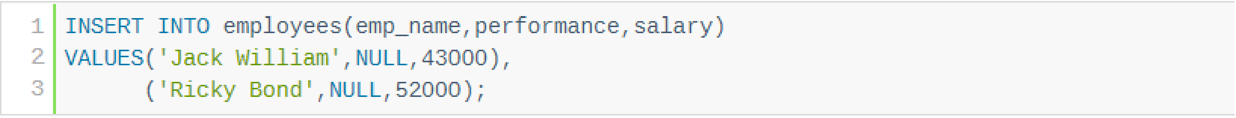
We specify only the employees table after UPDATE clause because we want to update data in the employees table only.

For each row in the employees table, the query checks the value in the performance column against the value in the performance column in the merits table. If it finds a match, it gets the percentage in the merits table and updates the salary column in the employees table.

Because we omit the WHERE clause in the UPDATE statement, all the records in the employees table get updated.

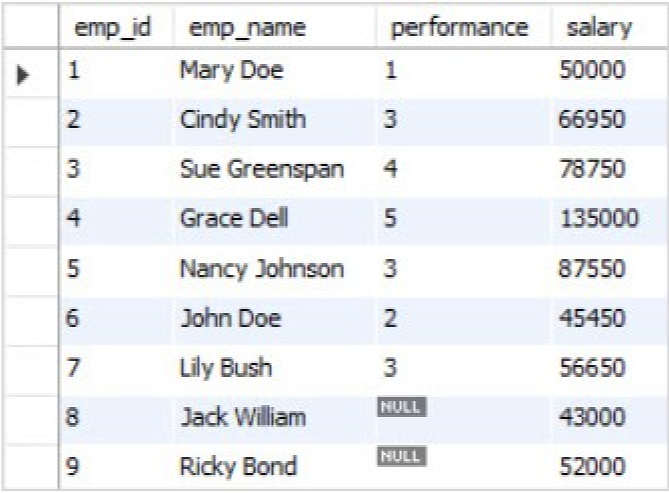
MySQL UPDATE JOIN example with LEFT JOIN

Suppose the company hires two more employees:



```
INSERT INTO employees(emp_name,performance,salary)
VALUES('Jack William',NULL,43000),
      ('Ricky Bond',NULL,52000);
```

Because these employees are new hires so their performance data is not available or NULL.

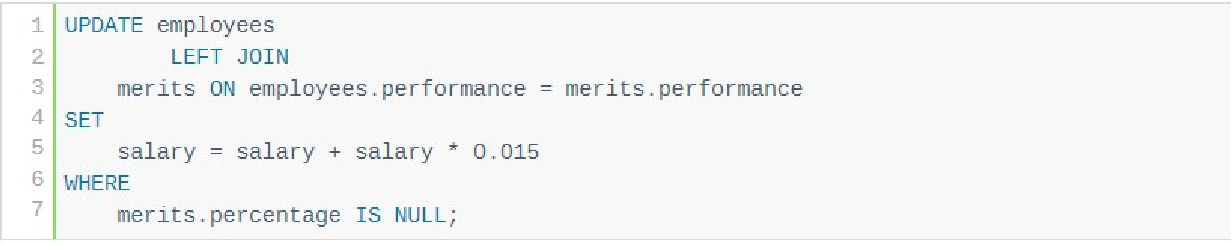


emp_id	emp_name	performance	salary
1	Mary Doe	1	50000
2	Cindy Smith	3	66950
3	Sue Greenspan	4	78750
4	Grace Dell	5	135000
5	Nancy Johnson	3	87550
6	John Doe	2	45450
7	Lily Bush	3	56650
8	Jack William	NULL	43000
9	Ricky Bond	NULL	52000

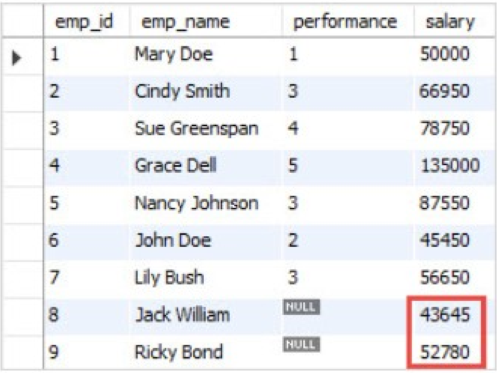
To increase the salary for new hires, you cannot use the UPDATE INNER JOIN statement because their performance data is not available in the merit table. This is why the UPDATE LEFT JOIN comes to the rescue.

The UPDATE LEFT JOIN statement basically updates a row in a table when it does not have a corresponding row in another table.

For example, you can increase the salary for a new hire by 1.5% using the following statement:



```
UPDATE employees
        LEFT JOIN
    merits ON employees.performance = merits.performance
SET
    salary = salary + salary * 0.015
WHERE
    merits.percentage IS NULL;
```



emp_id	emp_name	performance	salary
1	Mary Doe	1	50000
2	Cindy Smith	3	66950
3	Sue Greenspan	4	78750
4	Grace Dell	5	135000
5	Nancy Johnson	3	87550
6	John Doe	2	45450
7	Lily Bush	3	56650
8	Jack William	NULL	43645
9	Ricky Bond	NULL	52780

In this tutorial, we have shown you how to use the MySQL UPDATE JOIN with the INNER JOIN and LEFT JOIN clauses to perform the cross-table update.

MySQL DELETE

To delete data from a table, you use the MySQL DELETE statement. The following illustrates the syntax of the DELETE statement:

```
DELETE FROM table_name
WHERE condition;
```

In this statement:

- First, specify the table from which you delete data.
- Second, use a condition to specify which rows to delete in the WHERE clause. If the row matches the condition, it will be deleted.

Notice that the WHERE clause is optional. If you omit the WHERE clause, the DELETE statement will delete all rows in the table.

Besides deleting data from a table, the DELETE statement returns the number of rows deleted.

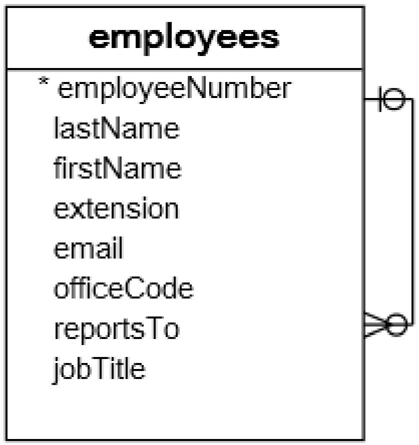
To delete data from multiple tables using a single DELETE statement, you use the DELETE JOIN statement which we will cover in the next lesson.

To delete all rows in a table without the need of knowing how many rows deleted, you should use the TRUNCATE TABLE statement to get better performance.

For a table that has a foreign key constraint, when you delete rows from the parent table, the rows in the child table will be deleted automatically by using the ON DELETE CASCADE option.

MySQL DELETE examples

We will use the employees table in the **sample database** for the demonstration.



Note that once you delete data, it is gone. Therefore, you should backup your database before performing the DELETE statements in the next section.

Suppose you want to delete employees whose the officeNumber are 4, you use the DELETE statement with the WHERE clause as shown in the following query:

```
DELETE FROM employees
WHERE
    officeCode = 4;
```

To delete all rows from the employees table, you use the DELETE statement without the WHERE clause as follows:

```
DELETE FROM employees;
```

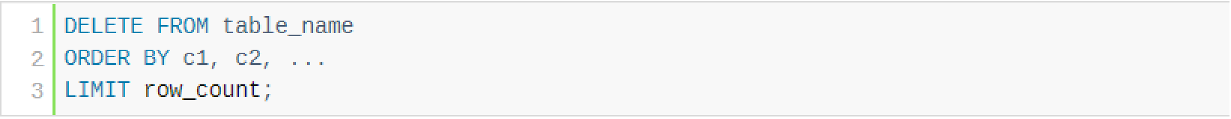
All rows in the employees table deleted.

MySQL DELETE and LIMIT clause

If you want to limit the number of rows to be deleted, you use the LIMIT clause as follows:

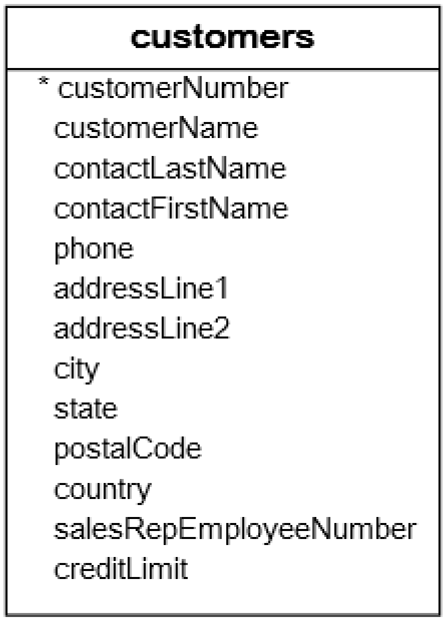
```
DELETE FROM table
LIMIT row_count;
```

Note that the order of rows in a table is unspecified, therefore, when you use the LIMIT clause, you should always use the ORDER BY clause.



```
DELETE FROM table_name
ORDER BY c1, c2, ...
LIMIT row_count;
```

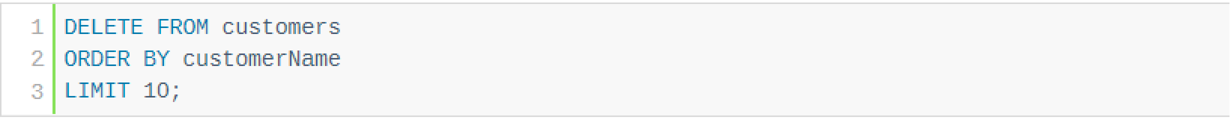
Consider the following customers table in the **sample database**:



customers

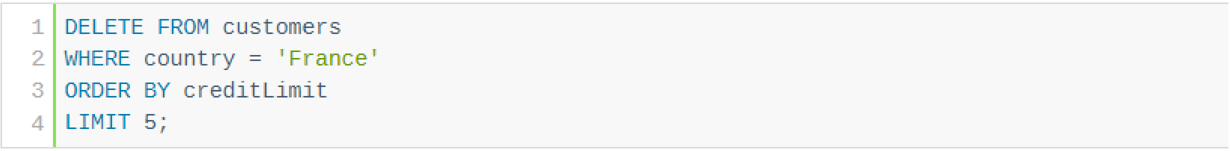
* customerNumber
customerName
contactLastName
contactFirstName
phone
addressLine1
addressLine2
city
state
postalCode
country
salesRepEmployeeNumber
creditLimit

For example, the following statement sorts customers by customer’s names alphabetically and deletes the first 10 customers:



```
DELETE FROM customers
ORDER BY customerName
LIMIT 10;
```

Similarly, the following DELETE statement selects customers in France, sorts them by credit limit in from low to high, and deletes the first 5 customers:



```
DELETE FROM customers
WHERE country = 'France'
ORDER BY creditLimit
LIMIT 5;
```

MySQL DELETE JOIN

In the previous tutorial, you learned how to delete rows of multiple tables by using:

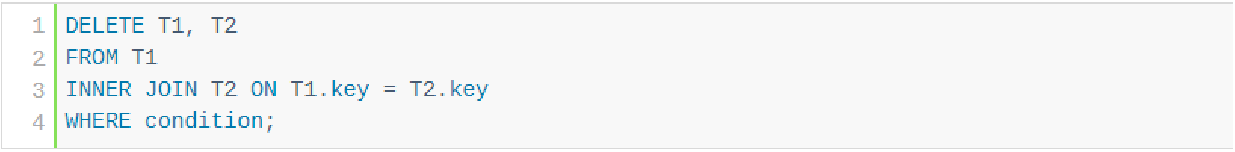
- A single DELETE statement on multiple tables.
- A single DELETE statement on multiple related tables which the child table have an ON DELETE CASCADE referential action for the foreign key.

This tutorial introduces to you a more flexible way to delete data from multiple tables using INNER JOIN or LEFT JOIN clause with the DELETE statement.

MySQL DELETE JOIN with INNER JOIN

MySQL also allows you to use the INNER JOIN clause in the DELETE statement to delete rows from a table and the matching rows in another table.

For example, to delete rows from both T1 and T2 tables that meet a specified condition, you use the following statement:



```
DELETE T1, T2
FROM T1
INNER JOIN T2 ON T1.key = T2.key
WHERE condition;
```

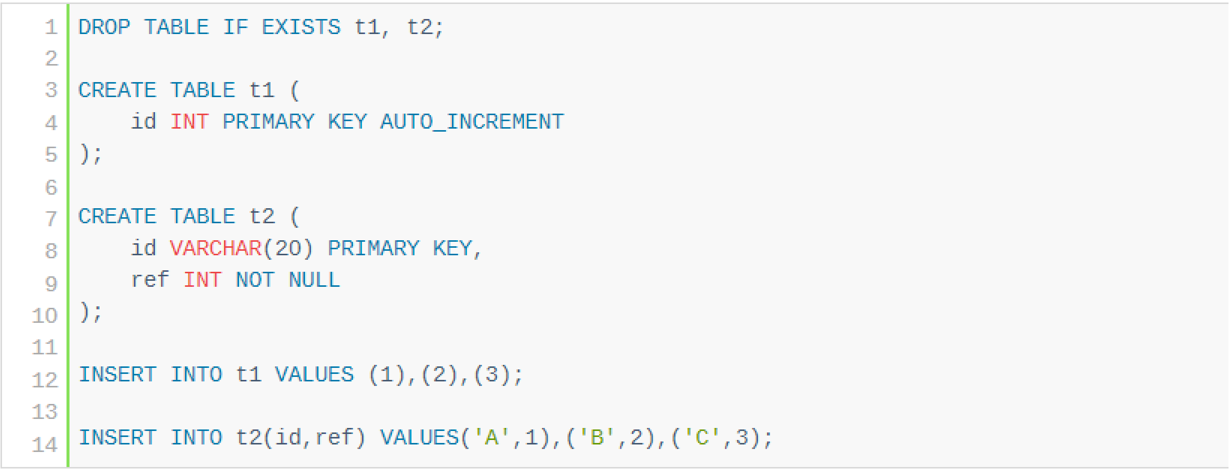
Notice that you put table names T1 and T2 between the DELETE and FROM keywords. If you omit T1 table, the DELETE statement only deletes rows in T2 table. Similarly, if you omitT2 table, the DELETE statement will delete only rows in T1 table.

The expression T1.key = T2.key specifies the condition for matching rows between T1 andT2 tables that will be deleted.

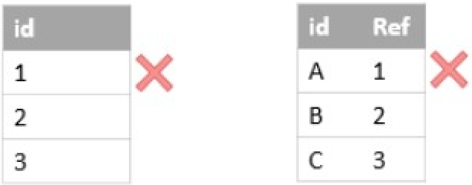
The condition in the WHERE clause determine rows in the T1 and T2 that will be deleted.

MySQL DELETE JOIN with INNER JOIN example

Suppose, we have two tables t1 and t2 with the following structures and data:

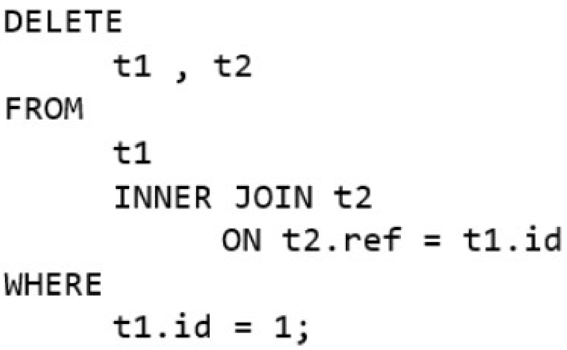


```sql
DROP TABLE IF EXISTS t1, t2;

CREATE TABLE t1 (
    id INT PRIMARY KEY AUTO_INCREMENT
);

CREATE TABLE t2 (
    id VARCHAR(20) PRIMARY KEY,
    ref INT NOT NULL
);

INSERT INTO t1 VALUES (1),(2),(3);

INSERT INTO t2(id,ref) VALUES('A',1),('B',2),('C',3);
```



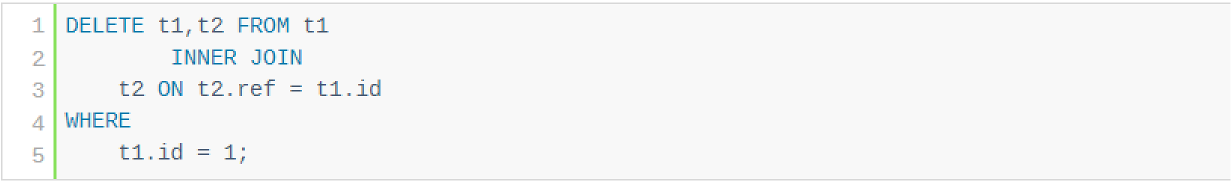
id
1 ✗
2
3

id	Ref
A	1 ✗
B	2
C	3



```sql
DELETE
        t1 , t2
FROM
        t1
        INNER JOIN t2
                ON t2.ref = t1.id
WHERE
        t1.id = 1;
```

The following statement deletes the row with id 1 in the t1 table and also row with ref 1 in the t2 table using DELETE...INNER JOIN statement:



```sql
DELETE t1,t2 FROM t1
        INNER JOIN
    t2 ON t2.ref = t1.id
WHERE
    t1.id = 1;
```

The statement returned the following message:

```
2 row(s) affected
```

It indicated that two rows have been deleted.

MySQL DELETE JOIN with LEFT JOIN

We often use the LEFT JOIN clause in the SELECT statement to find rows in the left table that have or don’t have matching rows in the right table.

We can also use the LEFT JOIN clause in the DELETE statement to delete rows in a table (left table) that does not have matching rows in another table (right table).

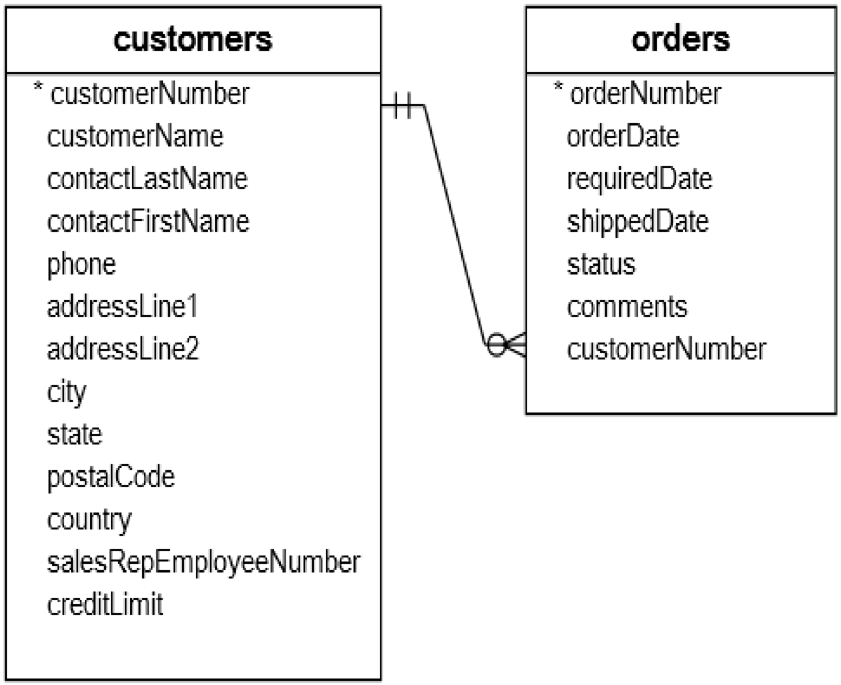
The following syntax illustrates how to use DELETE statement with LEFT JOIN clause to delete rows from T1 table that does not have corresponding rows in the T2 table:

```
DELETE T1
FROM T1
        LEFT JOIN
    T2 ON T1.key = T2.key
WHERE
    T2.key IS NULL;
```

Note that we only put T1 table after the DELETE keyword, not both T1 and T2 tables like we did with the INNER JOIN clause.

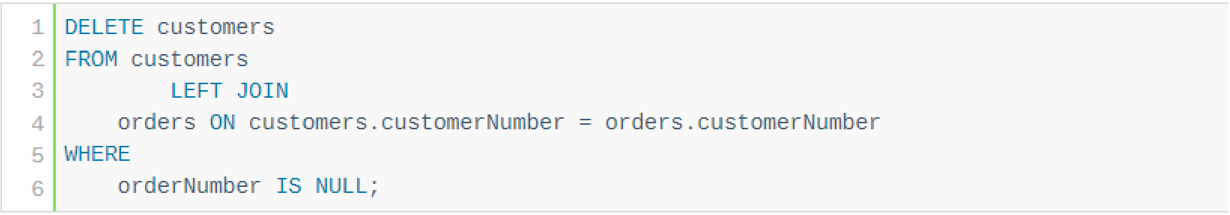
MySQL DELETE JOIN with LEFT JOIN example

See the following customers and orders tables in the sample database:



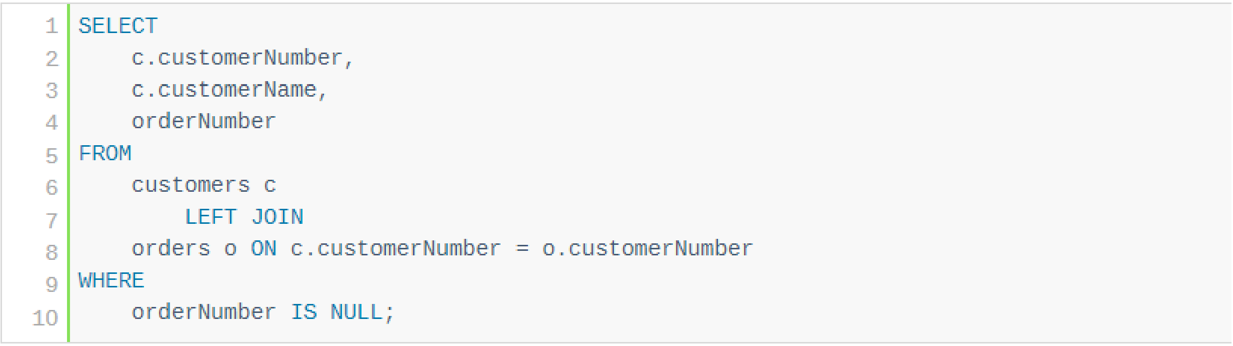
Each customer has zero or more orders. However, each order belongs to one and only one customer.

We can use DELETE statement with LEFT JOIN clause to clean up our customers master data. The following statement removes customers who have not placed any order:



```
DELETE customers
FROM customers
        LEFT JOIN
    orders ON customers.customerNumber = orders.customerNumber
WHERE
    orderNumber IS NULL;
```

We can verify the delete by finding whether customers who do not have any order exists using the following query:



```
SELECT
    c.customerNumber,
    c.customerName,
    orderNumber
FROM
    customers c
        LEFT JOIN
    orders o ON c.customerNumber = o.customerNumber
WHERE
    orderNumber IS NULL;
```

The query returned an empty result set which is what we expected.

In this tutorial, you have learned how to use the MySQL DELETE JOIN statement to delete data from two or more tables.

MySQL ON DELETE CASCADE: Deleting Data From Multiple Related Tables

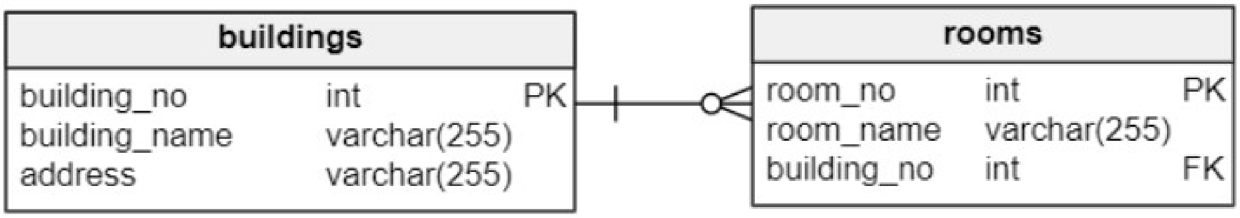
In the previous tutorial, you learned how to delete data from multiple related tables using a single DELETE statement. However, MySQL provides a more effective way called ON DELETE CASCADE referential action for a foreign key that allows you to delete data from child tables automatically when you delete the data from the parent table.

MySQL ON DELETE CASCADE example

Let’s take a look at an example of using MySQL ON DELETECASCADE.

Suppose we have two tables: buildings and rooms. In this database model, each building has one or more rooms. However, each room belongs to one only one building. A room would not exist without a building.

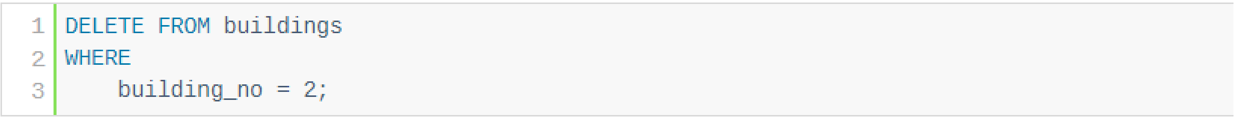
The relationship between the buildings and rooms tables is one-to-many (1:N) as illustrated in the following database diagram:



buildings		
building_no	int	PK
building_name	varchar(255)	
address	varchar(255)	

rooms		
room_no	int	PK
room_name	varchar(255)	
building_no	int	FK

When we delete a row from the buildings table, we also want to delete the rows in the rooms table that references to the rows in the buildings table. For example, when we delete a row with building no. 2 in the buildings table as the following query:

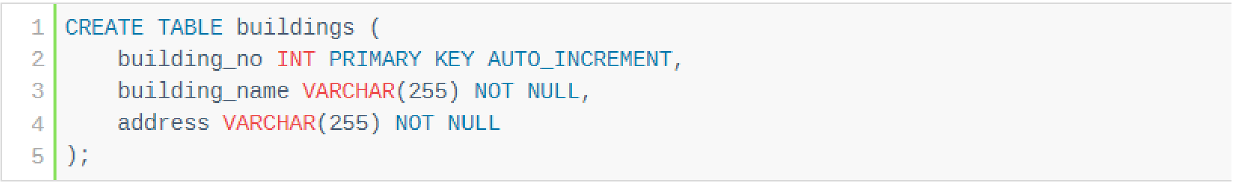


```
DELETE FROM buildings
WHERE
    building_no = 2;
```

We want the rows in the rooms table that refers to building number 2 will be also removed.

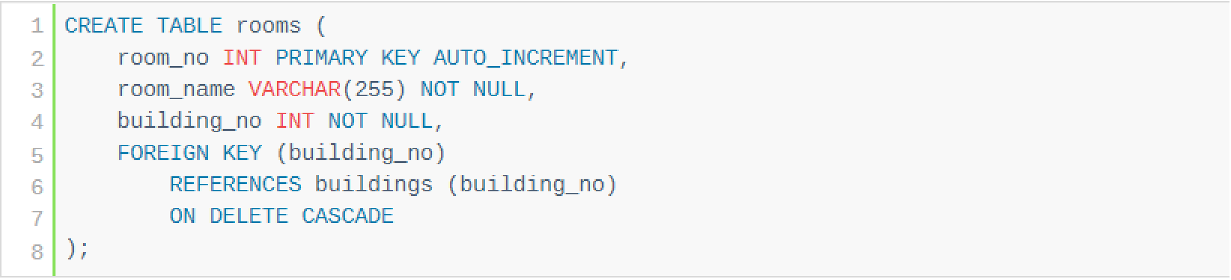
The following are steps that demonstrate how MySQL ON DELETE CASCADE referential action works.

Step 1. Create the buildings table:



```
CREATE TABLE buildings (
    building_no INT PRIMARY KEY AUTO_INCREMENT,
    building_name VARCHAR(255) NOT NULL,
    address VARCHAR(255) NOT NULL
);
```

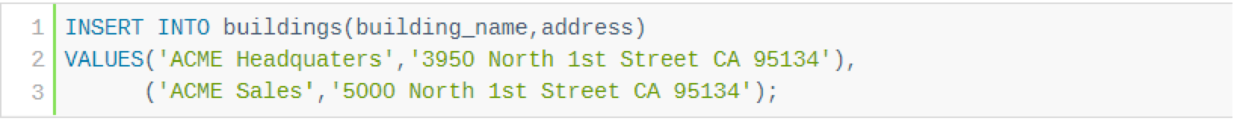
Step 2. Create the rooms table:



```
CREATE TABLE rooms (
    room_no INT PRIMARY KEY AUTO_INCREMENT,
    room_name VARCHAR(255) NOT NULL,
    building_no INT NOT NULL,
    FOREIGN KEY (building_no)
        REFERENCES buildings (building_no)
        ON DELETE CASCADE
);
```

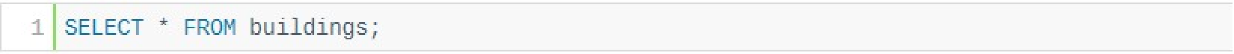
Notice that we add the ON DELETE CASCADE clause at the end of the foreign key constraint definition.

Step 3. Insert data into the buildings table:

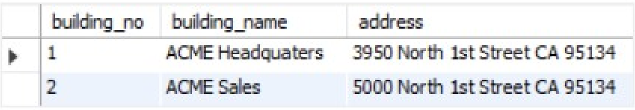


```
INSERT INTO buildings(building_name,address)
VALUES('ACME Headquaters','3950 North 1st Street CA 95134'),
      ('ACME Sales','5000 North 1st Street CA 95134');
```

Step 4. Query data from the buildings table:



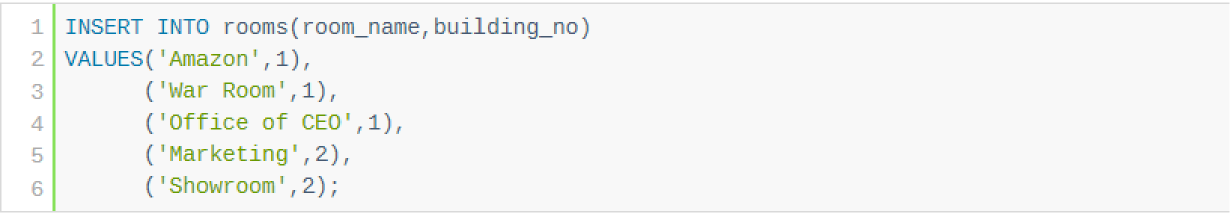
```
SELECT * FROM buildings;
```



building_no	building_name	address
1	ACME Headquaters	3950 North 1st Street CA 95134
2	ACME Sales	5000 North 1st Street CA 95134

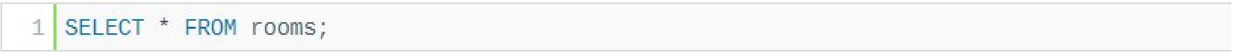
We have two rows in the buildings table.

Step 5. Insert data into the rooms table:

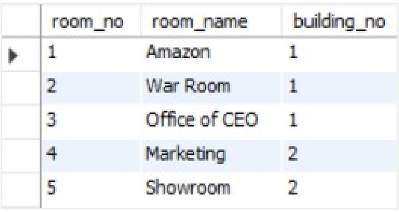


```
INSERT INTO rooms(room_name,building_no)
VALUES('Amazon',1),
      ('War Room',1),
      ('Office of CEO',1),
      ('Marketing',2),
      ('Showroom',2);
```

Step 6. Query data from the rooms table:



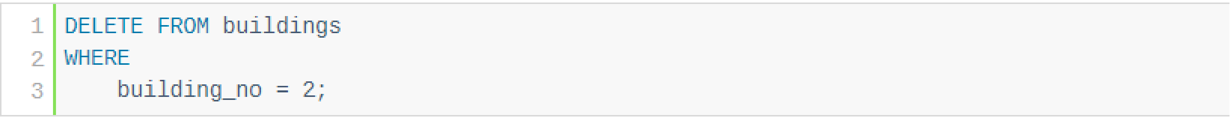
```
SELECT * FROM rooms;
```



room_no	room_name	building_no
1	Amazon	1
2	War Room	1
3	Office of CEO	1
4	Marketing	2
5	Showroom	2

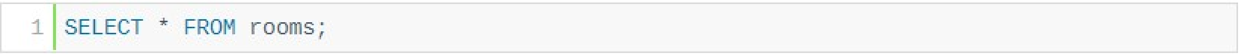
We have 3 rooms that belong to building 1 and 2 rooms that belong to the building 2.

Step 7. Delete the building with building no. 2:

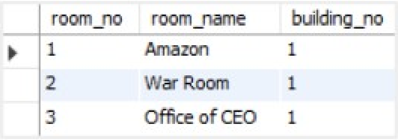


```
DELETE FROM buildings
WHERE
    building_no = 2;
```

Step 8. Query data from rooms table:



```
SELECT * FROM rooms;
```

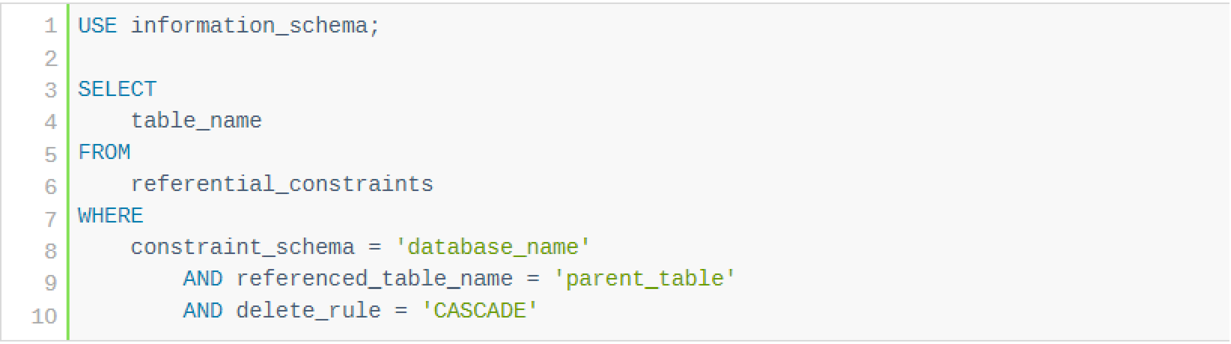


room_no	room_name	building_no
1	Amazon	1
2	War Room	1
3	Office of CEO	1

As you can see, all the rows that reference to building_no 2 were deleted.

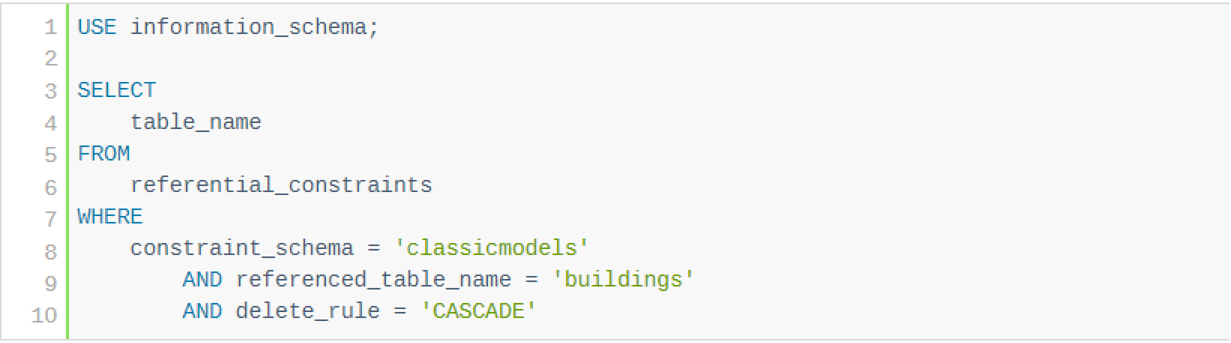
Notice that ON DELETE CASCADE works only with tables with the storage engines support foreign keys e.g., InnoDB. Some table types do not support foreign keys such as MyISAM so you should choose appropriate storage engines for the tables that you plan to use the MySQL ON DELETE CASCADE referential action.

Sometimes, it is useful to know which table is affected by the MySQL ON DELETE CASCADE referential action when you delete data from a table. You can query this data from the referential_constraintsin the information_schema database as follows:

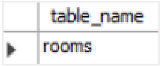


```
USE information_schema;

SELECT
    table_name
FROM
    referential_constraints
WHERE
    constraint_schema = 'database_name'
        AND referenced_table_name = 'parent_table'
        AND delete_rule = 'CASCADE'
```

For example, to find tables that associated with the buildings table with the CASCADE deletion rule in the classicmodels database, you use the following query:



```
USE information_schema;

SELECT
    table_name
FROM
    referential_constraints
WHERE
    constraint_schema = 'classicmodels'
        AND referenced_table_name = 'buildings'
        AND delete_rule = 'CASCADE'
```



	table_name
▶	rooms

In this tutorial, we have shown you step by step how to use the MySQL ON DELETE CASCADE referential action for a foreign key to delete data automatically from the child tables when you delete data from the parent table.

MySQL REPLACE

The MySQL REPLACE statement is a MySQL extension to the standard SQL. The MySQL REPLACE statement works as follows:

- If the new row already does not exist, the MySQL REPLACE statement inserts a new row.
- If the new row already exist, the REPLACE statement deletes the old row first and then inserts a new row. In some cases, the REPLACE statement updates the existing row only.

To determine whether the new row already exists in the table, MySQL uses PRIMARY KEY or UNIQUE KEY index. If the table does not have one of these indexes, the REPLACE statement is equivalent to the INSERT statement.

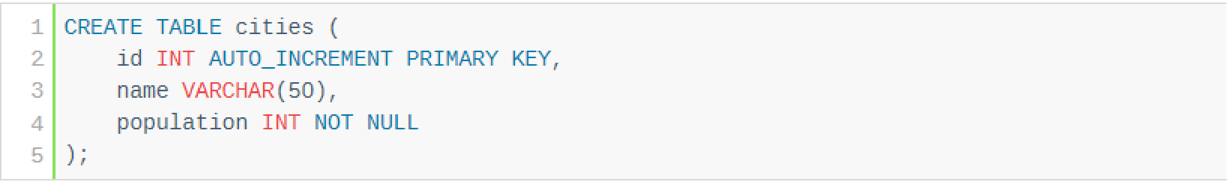
To use the MySQL REPLACE statement, you need to have at least both INSERT and DELETE privileges.

Notice that there is a REPLACE string function which is not the REPLACE statement covered in this tutorial.

MySQL REPLACE statement example

Let’s take a look at an example of using the REPLACE statement to have a better understanding of how it works.

First, create a new table named cities as follows:



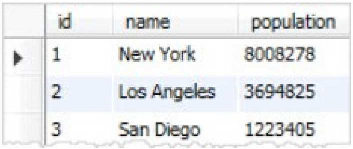
```
CREATE TABLE cities (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50),
    population INT NOT NULL
);
```

Next, insert some rows into the cities table:

```
INSERT INTO cities(name,population)
VALUES('New York',8008278),
   ('Los Angeles',3694825),
   ('San Diego',1223405);
```

We query data from the cities table to verify the insert operation.

```
SELECT
    *
FROM
    cities;
```



id	name	population
1	New York	8008278
2	Los Angeles	3694825
3	San Diego	1223405

We have three cities in the cities table.

Then, suppose we want to update the population of the New York city to 1008256. We can use the UPDATE statement as follows:

```
UPDATE cities
SET
    population = 1008256
WHERE
    id = 1;
```

We query the data from the cities table again to verify the update.

```
SELECT
    *
FROM
    cities;
```

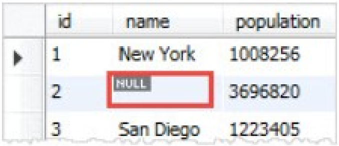
The UPDATE statement updated the data as expected.

After that, use the REPLACE statement to update the population of the Los Angeles city to 3696820.

```
REPLACE INTO cities(id,population)
VALUES(2,3696820);
```

Finally, query the data of the cities table again to verify the replacement.

```
SELECT
    *
FROM
    cities;
```



id	name	population
1	New York	1008256
2	NULL	3696820
3	San Diego	1223405

The name column is NULL now. You may expect that the value of the name column remains intact. However, the REPLACE statement does not behave this way. In this case, the REPLACE statement works as follows:

1. The REPLACE statement first inserts the new row into the cities table with the information provided by the column list. The insertion fails because the row with id 2 already exists in the cities table, therefore, MySQL raises a duplicate-key error.
2. The REPLACE statement then updates the row that has the key specified in the value of the id column. In the normal process, it would delete the old row with conflict id first and then inserts a new row.

We know that the REPLACE statement did not delete the old row and inserted the new row because the value of the id column is 2 instead of 4.

MySQL REPLACE and INSERT

The first form of the REPLACE statement is similar to the INSERT statement except the keyword INSERT is replaced by the REPLACE keyword as follows:

```
REPLACE INTO table_name(column_list)
VALUES(value_list);
```

For example, if you want to insert a new row into the cities table, you use the following query:

```
REPLACE INTO cities(name,population)
VALUES('Phoenix',1321523);
```

Notice that the default values of the columns that do not appear in the REPLACE statement will be inserted into the corresponding columns. In case the column that has the NOT NULL attribute and does not have a default value, and you don’t specify the value in the REPLACE statement, MySQL will raise an error. This is a difference between the REPLACE and INSERT statements. For example, in the following statement, we specify only the value for the name column, not the population column. MySQL raises an error message. Because the population column does not accept a NULL value and we did not specify a default value for it when we defined the cities table.

```
REPLACE INTO cities(name)
VALUES('Houston');
```

This is the error message that MySQL issued:

```
Error Code: 1364. Field 'population' doesn't have a default value
```

MySQL REPLACE and UPDATE

The second form of REPLACE statement is similar to the UPDATE statement as follows:

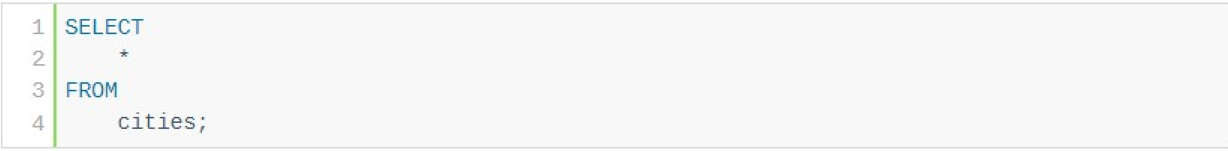
```
REPLACE INTO table
SET column1 = value1,
    column2 = value2;
```

Notice that there is no WHERE clause in the REPLACE statement.

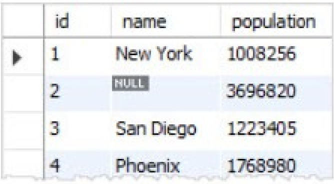
For example, if you want to update the population of the Phoenix city to 1768980, you use the REPLACE statement as follows:

```
REPLACE INTO table
SET column1 = value1,
    column2 = value2;
```

Unlike the UPDATE statement, if you don’t specify the value for the column in the SET clause, the REPLACE statement will use the default value of that column.



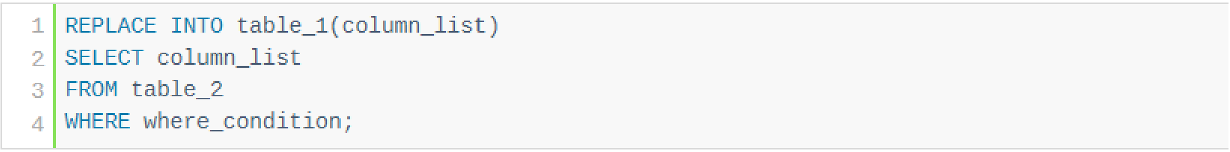
```
SELECT
    *
FROM
    cities;
```



id	name	population
1	New York	1008256
2	NULL	3696820
3	San Diego	1223405
4	Phoenix	1768980

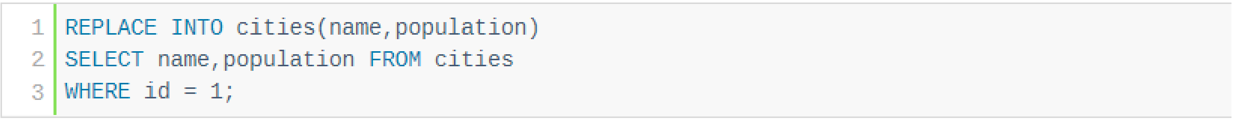
MySQL REPLACE INTO and SELECT

The third form of REPLACE statement is similar to INSERT INTO SELECT statement:



```
REPLACE INTO table_1(column_list)
SELECT column_list
FROM table_2
WHERE where_condition;
```

Suppose, you want to copy the city with id value 1, you use the REPLACE INTO SELECT statement as the following query:



```
REPLACE INTO cities(name,population)
SELECT name,population FROM cities
WHERE id = 1;
```

MySQL REPLACE statement usages

There are several important points you need to know when you use the REPLACE statement:

- If you develop an application that supports not only MySQL database but also other relational database management systems (RDBMS), you should avoid using the REPLACE statement because other RDBMS may not support it. Instead, you can use the combination of the DELETE and INSERT statements within a transaction.
- If you are using the REPLACE statement in the table that has triggers and the deletion of duplicate-key error occurs, the triggers will be fired in the following sequence: BEFORE

INSERT BEFORE DELETE , AFTER DELETE , AFTER INSERT in case the REPLACE statement deletes current row and inserts the new row. In case the REPLACE statement updates the current row, the BEFORE UPDATE and AFTER UPDATE triggers are fired.

In this tutorial, you’ve learned different forms of REPLACE statement to insert or update data in tables.

http://www.mysqltutorial.org/basic-mysql-tutorial.aspx

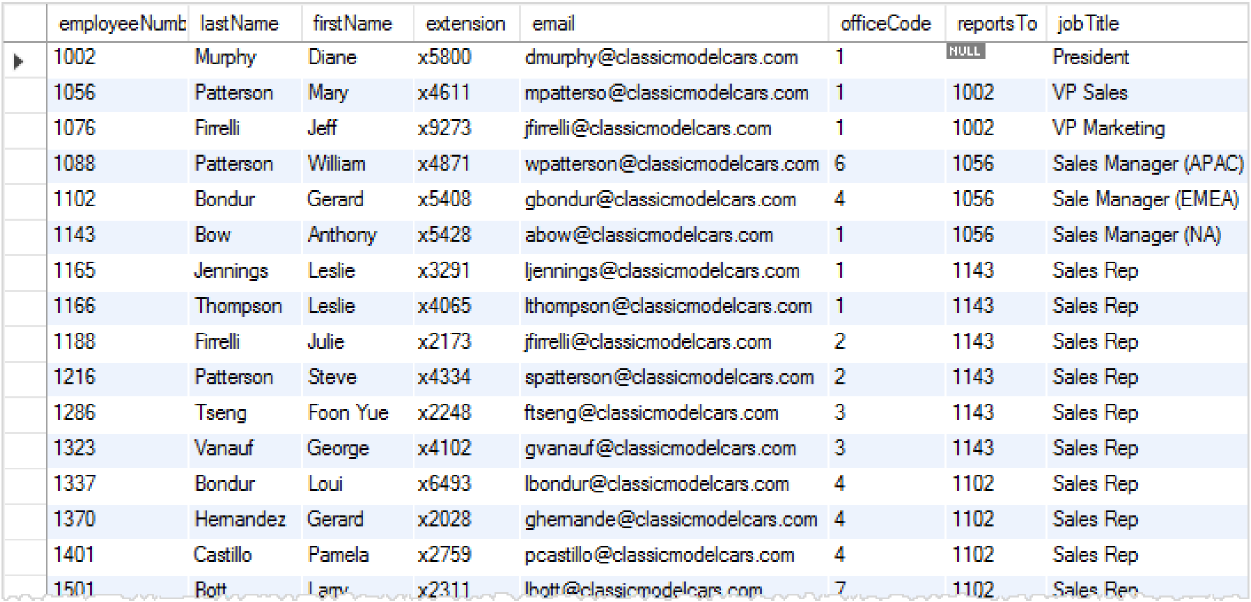
MySQL SELECT

Summary: in this tutorial, you will learn how to use MySQL SELECT statement to query data from tables or views.

Introduction to MySQL SELECT statement

The SELECT statement allows you to get the data from tables or views. A table consists of rows and columns like a spreadsheet. Often, you want to see a subset rows, a subset of columns, or a combination of two. The result of the SELECT statement is called a result set that is a list of rows, each consisting of the same number of columns.

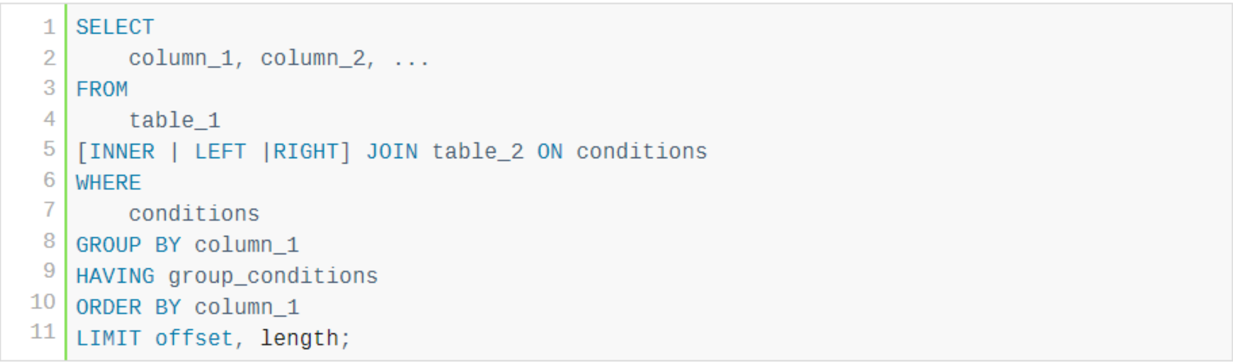
See the following employees table in the sample database. It has eight columns: employee number, last name, first name, extension, email, office code, reports to, job title and many rows.



employeeNumb	lastName	firstName	extension	email	officeCode	reportsTo	jobTitle
1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	1	NULL	President
1056	Patterson	Mary	x4611	mpatterso@classicmodelcars.com	1	1002	VP Sales
1076	Firrelli	Jeff	x9273	jfirrelli@classicmodelcars.com	1	1002	VP Marketing
1088	Patterson	William	x4871	wpatterson@classicmodelcars.com	6	1056	Sales Manager (APAC)
1102	Bondur	Gerard	x5408	gbondur@classicmodelcars.com	4	1056	Sale Manager (EMEA)
1143	Bow	Anthony	x5428	abow@classicmodelcars.com	1	1056	Sales Manager (NA)
1165	Jennings	Leslie	x3291	ljennings@classicmodelcars.com	1	1143	Sales Rep
1166	Thompson	Leslie	x4065	lthompson@classicmodelcars.com	1	1143	Sales Rep
1188	Firrelli	Julie	x2173	jfirrelli@classicmodelcars.com	2	1143	Sales Rep
1216	Patterson	Steve	x4334	spatterson@classicmodelcars.com	2	1143	Sales Rep
1286	Tseng	Foon Yue	x2248	ftseng@classicmodelcars.com	3	1143	Sales Rep
1323	Vanauf	George	x4102	gvanauf@classicmodelcars.com	3	1143	Sales Rep
1337	Bondur	Loui	x6493	lbondur@classicmodelcars.com	4	1102	Sales Rep
1370	Hernandez	Gerard	x2028	ghernande@classicmodelcars.com	4	1102	Sales Rep
1401	Castillo	Pamela	x2759	pcastillo@classicmodelcars.com	4	1102	Sales Rep
1501	Bott	Larry	x2311	lbott@classicmodelcars.com	7	1102	Sales Rep

The SELECT statement controls which columns and rows that you want to see. For example, if you are only interested in the first name, last name, and job title of all employees or you just want to view the information of every employee whose job title is the sales rep, the SELECT statement helps you to do this.

Let’s take look into the syntax of the SELECT statement:



```
SELECT
    column_1, column_2, ...
FROM
    table_1
[INNER | LEFT |RIGHT] JOIN table_2 ON conditions
WHERE
    conditions
GROUP BY column_1
HAVING group_conditions
ORDER BY column_1
LIMIT offset, length;
```

The SELECT statement consists of several clauses as explained in the following list:

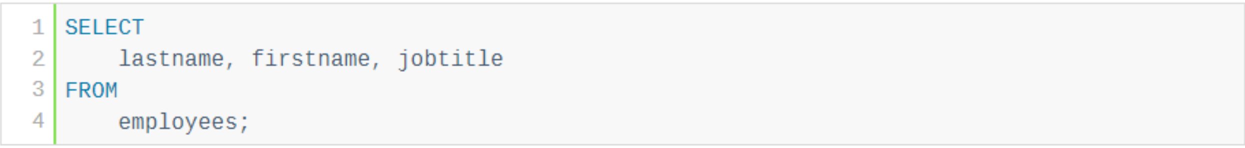
SELECT followed by a list of comma-separated columns or an asterisk (*) to indicate that you want to return all columns.

- FROM specifies the table or view where you want to query the data.
- JOIN gets related data from other tables based on specific join conditions.
- WHERE clause filters row in the result set.
- GROUP BY clause groups a set of rows into groups and applies aggregate functions on each group.
- HAVING clause filters group based on groups defined by GROUP BY clause.
- ORDER BY clause specifies a list of columns for sorting.
- LIMIT constrains the number of returned rows.

The SELECT and FROM clauses are required in the statement. Other parts are optional.

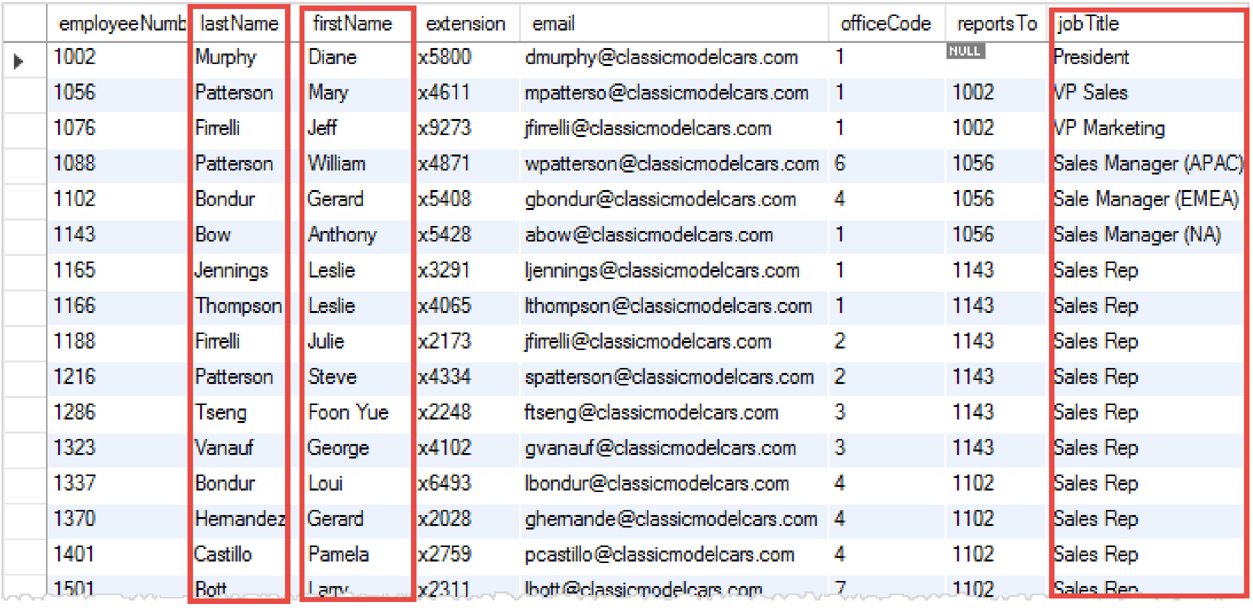
MySQL SELECT statement examples

The SELECT statement allows you to query partial data of a table by specifying a list of comma-separated columns in the SELECT clause. For instance, if you want to view only first name, last name, and job title of the employees, you use the following query:

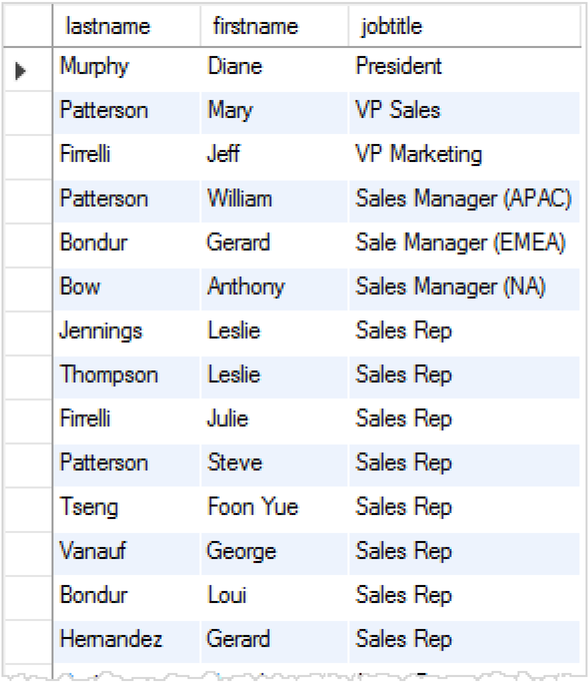


```
SELECT
    lastname, firstname, jobtitle
FROM
    employees;
```

Even though the `employees` table has many columns, the `SELECT` statement just returns data of three columns of all rows in the table as highlighted in the picture below:



employeeNumb	lastName	firstName	extension	email	officeCode	reportsTo	jobTitle
1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	1	NULL	President
1056	Patterson	Mary	x4611	mpatterso@classicmodelcars.com	1	1002	VP Sales
1076	Firrelli	Jeff	x9273	jfirrelli@classicmodelcars.com	1	1002	VP Marketing
1088	Patterson	William	x4871	wpatterson@classicmodelcars.com	6	1056	Sales Manager (APAC)
1102	Bondur	Gerard	x5408	gbondur@classicmodelcars.com	4	1056	Sale Manager (EMEA)
1143	Bow	Anthony	x5428	abow@classicmodelcars.com	1	1056	Sales Manager (NA)
1165	Jennings	Leslie	x3291	ljennings@classicmodelcars.com	1	1143	Sales Rep
1166	Thompson	Leslie	x4065	lthompson@classicmodelcars.com	1	1143	Sales Rep
1188	Firrelli	Julie	x2173	jfirrelli@classicmodelcars.com	2	1143	Sales Rep
1216	Patterson	Steve	x4334	spatterson@classicmodelcars.com	2	1143	Sales Rep
1286	Tseng	Foon Yue	x2248	ftseng@classicmodelcars.com	3	1143	Sales Rep
1323	Vanauf	George	x4102	gvanauf@classicmodelcars.com	3	1143	Sales Rep
1337	Bondur	Loui	x6493	lbondur@classicmodelcars.com	4	1102	Sales Rep
1370	Hernandez	Gerard	x2028	ghernande@classicmodelcars.com	4	1102	Sales Rep
1401	Castillo	Pamela	x2759	pcastillo@classicmodelcars.com	4	1102	Sales Rep
1501	Bott	Larry	x2311	lbott@classicmodelcars.com	7	1102	Sales Rep



lastname	firstname	jobtitle
Murphy	Diane	President
Patterson	Mary	VP Sales
Firrelli	Jeff	VP Marketing
Patterson	William	Sales Manager (APAC)
Bondur	Gerard	Sale Manager (EMEA)
Bow	Anthony	Sales Manager (NA)
Jennings	Leslie	Sales Rep
Thompson	Leslie	Sales Rep
Firrelli	Julie	Sales Rep
Patterson	Steve	Sales Rep
Tseng	Foon Yue	Sales Rep
Vanauf	George	Sales Rep
Bondur	Loui	Sales Rep
Hernandez	Gerard	Sales Rep

If you want to get data for all columns in the `employees` table, you can list all column names in the `SELECT` clause. Or you just use the asterisk (*) to indicate that you want to get data from all columns of the table like the following query:

```
SELECT * FROM employees;
```

It returns all columns and rows in the `employees` table.
You should use the asterisk (*) for testing only. In practical, you should list the columns that you want to get data explicitly because of the following reasons:

- The asterisk (*) returns data from the columns that you may not use. It produces unnecessary I/O disk and network traffic between the MySQL database server and application.
- If you explicit specify the columns, the result set is more predictable and easier to manage. Imagine when you use the asterisk (*) and someone changes the table by adding more columns, you will end up with a result set that is different from what you expected.
- Using asterisk (*) may expose sensitive information to unauthorized users.

MySQL DISTINCT

Summary: in this tutorial, you will learn how to use **MySQL DISTINCT** clause with the `SELECT` statement to eliminate duplicate rows in a result set.

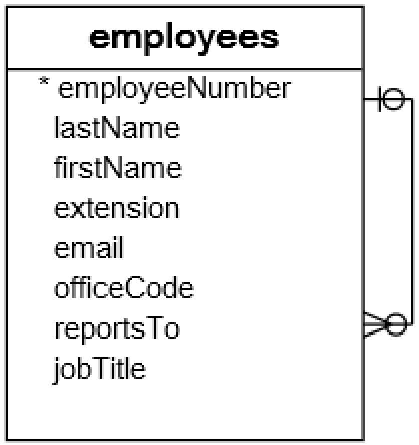
Introduction to MySQL `DISTINCT` clause

When querying data from a table, you may get duplicate rows. In order to remove these duplicate rows, you use the `DISTINCT` clause in the `SELECT` statement.
The syntax of using the `DISTINCT` clause is as follows:

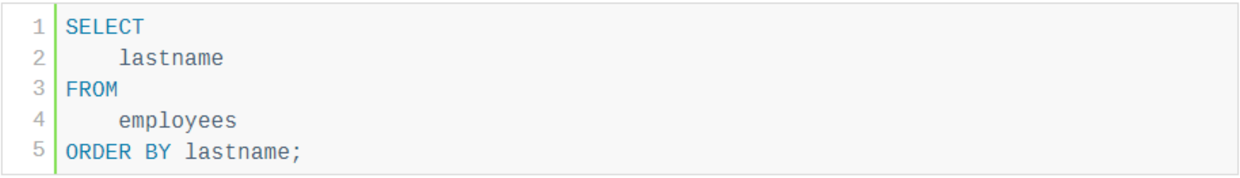
```
SELECT DISTINCT
    columns
FROM
    table_name
WHERE
    where_conditions;
```

MySQL `DISTINCT` examples

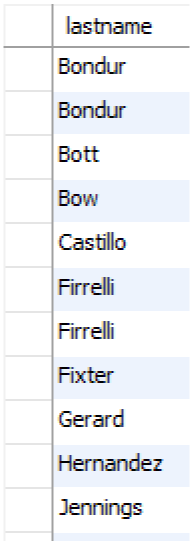
Let’s take a look at a simple example of using the `DISTINCT` clause to select the unique last names of employees from the `employees` table.



First, query the last names of employees from the `employees` table using the `SELECT` statement as follows:

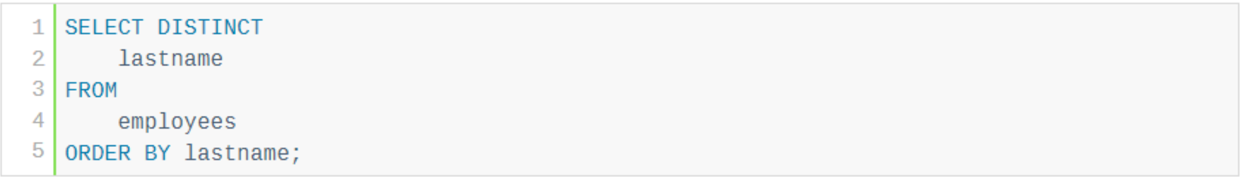


```
SELECT
    lastname
FROM
    employees
ORDER BY lastname;
```

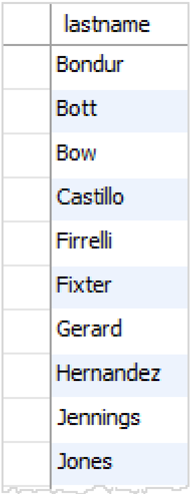


lastname
Bondur
Bondur
Bott
Bow
Castillo
Firrelli
Firrelli
Fixter
Gerard
Hernandez
Jennings

As clearly shown in the output, some employees have the same last name e.g, `Bondur,Firrelli` .
To remove the duplicate last names, you add the `DISTINCT` clause to the `SELECT` statement as follows:



```
SELECT DISTINCT
    lastname
FROM
    employees
ORDER BY lastname;
```



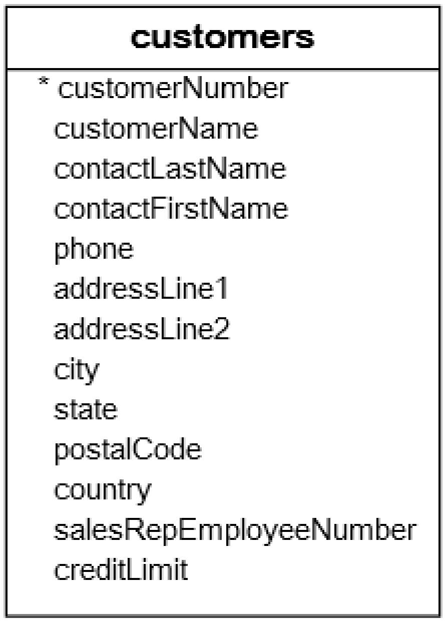
lastname
Bondur
Bott
Bow
Castillo
Firrelli
Fixter
Gerard
Hernandez
Jennings
Jones

The duplicate last names are eliminated in the result set when we used the DISTINCT clause.

MySQL DISTINCT and NULL values

If a column has NULL values and you use the DISTINCT clause for that column, MySQL keeps only one NULL value and eliminates the other because the DISTINCT clause treats all NULL values as the same value.

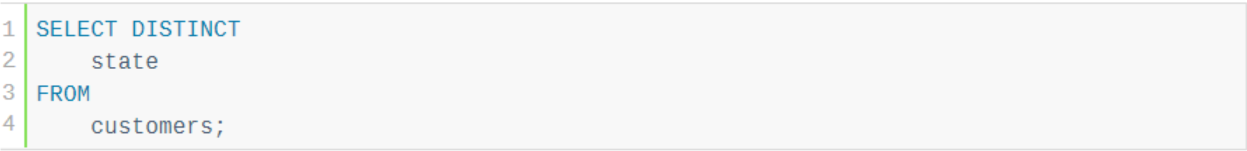
For example, in the customers table, we have many rows whose state column has NULL values.



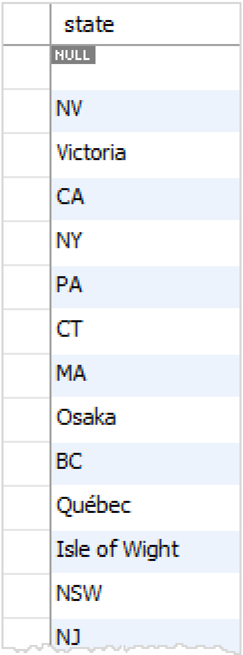
customers

* customerNumber
customerName
contactLastName
contactFirstName
phone
addressLine1
addressLine2
city
state
postalCode
country
salesRepEmployeeNumber
creditLimit

When you use the `DISTINCT` clause to query the customers' states, you will see unique states and `NULL`as the following query:



```
SELECT DISTINCT
    state
FROM
    customers;
```

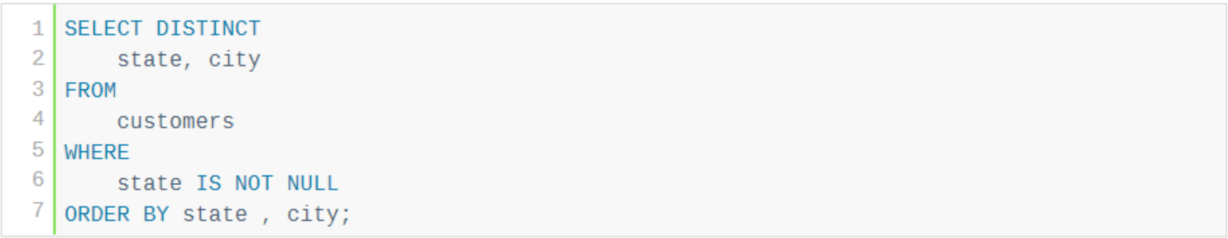


	state
	NULL
	NV
	Victoria
	CA
	NY
	PA
	CT
	MA
	Osaka
	BC
	Québec
	Isle of Wight
	NSW
	NJ

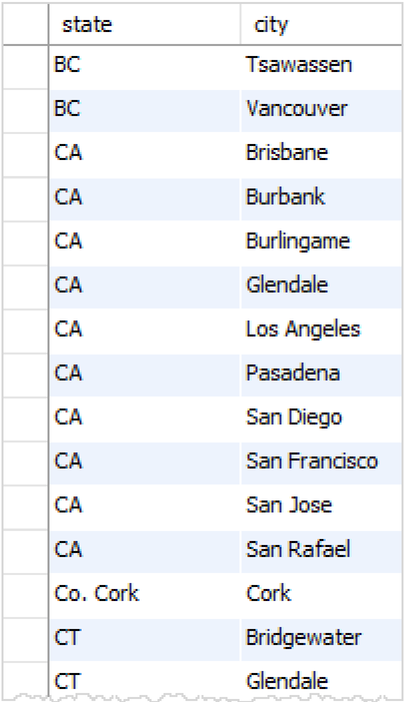
MySQL DISTINCT with multiple columns

You can use the DISTINCT clause with more than one column. In this case, MySQL uses the combination of values in these columns to determine the uniqueness of the row in the result set.

For example, to get the unique combination of city and state from the customers table, you use the following query:

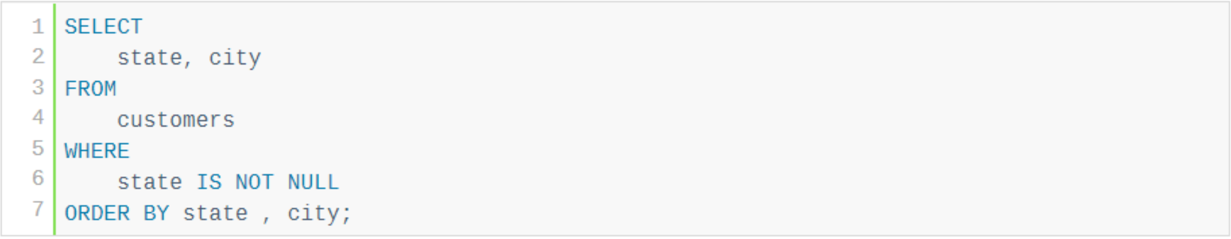


```
SELECT DISTINCT
    state, city
FROM
    customers
WHERE
    state IS NOT NULL
ORDER BY state , city;
```

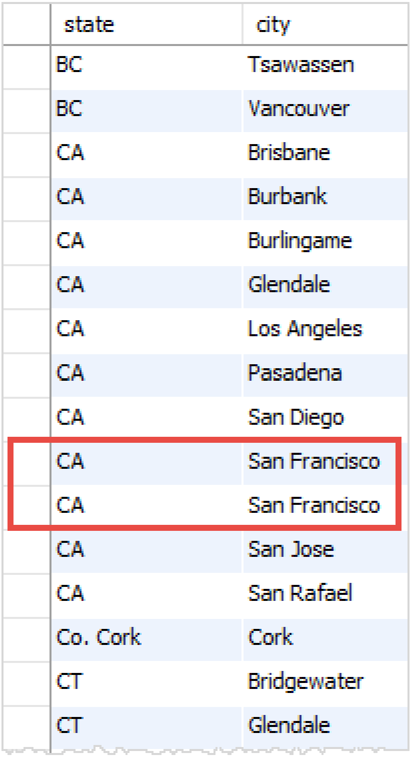


state	city
BC	Tsawassen
BC	Vancouver
CA	Brisbane
CA	Burbank
CA	Burlingame
CA	Glendale
CA	Los Angeles
CA	Pasadena
CA	San Diego
CA	San Francisco
CA	San Jose
CA	San Rafael
Co. Cork	Cork
CT	Bridgewater
CT	Glendale

Without the DISTINCT clause, you will get the duplicate combination of state and city as follows:



```
SELECT
    state, city
FROM
    customers
WHERE
    state IS NOT NULL
ORDER BY state , city;
```



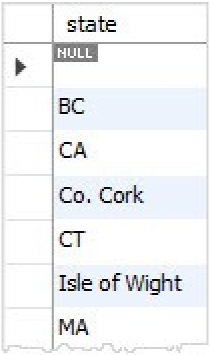
state	city
BC	Tsawassen
BC	Vancouver
CA	Brisbane
CA	Burbank
CA	Burlingame
CA	Glendale
CA	Los Angeles
CA	Pasadena
CA	San Diego
CA	San Francisco
CA	San Francisco
CA	San Jose
CA	San Rafael
Co. Cork	Cork
CT	Bridgewater
CT	Glendale

DISTINCT clause vs. GROUP BY clause

If you use the GROUP BY clause in the SELECT statement without using aggregate functions, the GROUP BYclause behaves like the DISTINCT clause.

The following statement uses the GROUP BY clause to select the unique states of customers from the customers table.

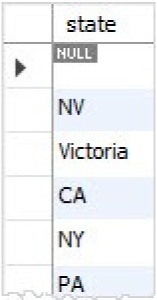
```
SELECT
    state
FROM
    customers
GROUP BY state;
```



state
NULL
BC
CA
Co. Cork
CT
Isle of Wight
MA

You can achieve a similar result by using the DISTINCT clause:

```
SELECT DISTINCT
    state
FROM
    customers;
```



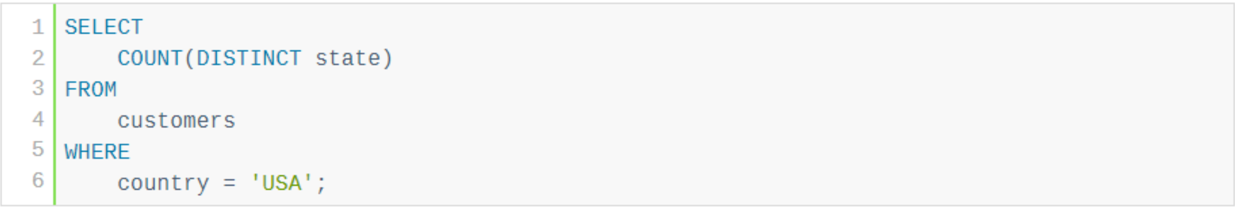
state
NULL
NV
Victoria
CA
NY
PA

Generally speaking, the DISTINCT clause is a special case of the GROUP BY clause. The difference between DISTINCT clause and GROUP BY clause is that the GROUP BY clause sorts the result set whereas the DISTINCT clause does not.
If you add the ORDER BY clause to the statement that uses the DISTINCT clause, the result set is sorted and it is the same as the one returned by the statement that uses GROUP BY clause.

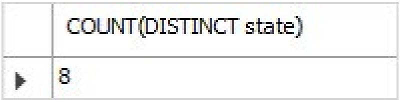
```
SELECT DISTINCT
    state
FROM
    customers
ORDER BY state;
```

MySQL DISTINCT and aggregate functions

You can use the DISTINCT clause with an aggregate function e.g., SUM, AVG, and COUNT, to remove duplicate rows before the aggregate functions are applied to the result set.
For example, to count the unique states of customers in the U.S., you use the following query:



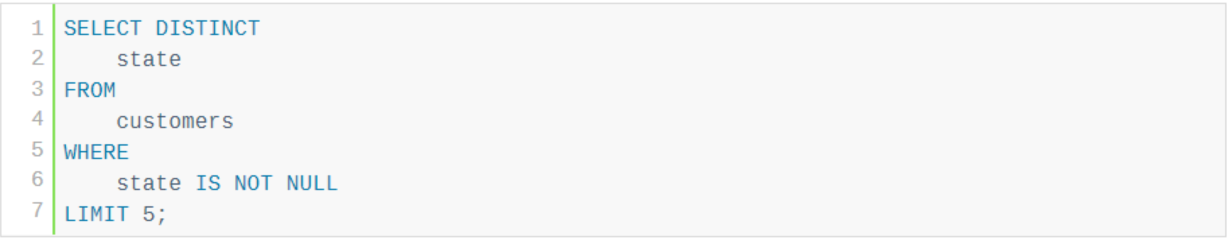
```
SELECT
    COUNT(DISTINCT state)
FROM
    customers
WHERE
    country = 'USA';
```



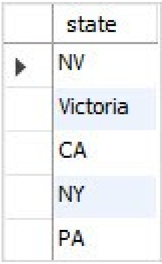
	COUNT(DISTINCT state)
▶	8

MySQL DISTINCT with LIMIT clause

In case you use the DISTINCT clause with the LIMIT clause, MySQL immediately stops searching when it finds the number of unique rows specified in the LIMIT clause.
The following query selects the first five non-null unique states in the customers table.



```
SELECT DISTINCT
    state
FROM
    customers
WHERE
    state IS NOT NULL
LIMIT 5;
```



	state
▶	NV
	Victoria
	CA
	NY
	PA

In this tutorial, we have shown you various ways of using MySQL DISTINCT clause such as eliminating duplicate rows and counting non-NULL values.

MySQL ORDER BY

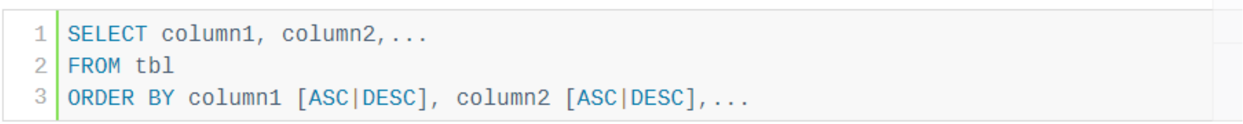
Summary: in this tutorial, you will learn how to sort a result set using **MySQL ORDER BY** clause.

Introduction to MySQL ORDER BY clause

When you use the SELECT statement to query data from a table, the result set is not sorted in any orders. To sort the result set, you use the ORDER BY clause. The ORDER BY clause allows you to:

- Sort a result set by a single column or multiple columns.
- Sort a result set by different columns in ascending or descending order.

The following illustrates the syntax of the ORDER BY clause:

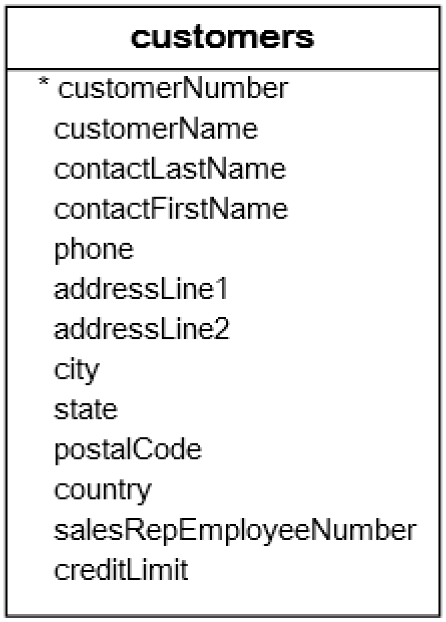


```
SELECT column1, column2,...
FROM tbl
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC],...
```

The ASC stands for ascending and the DESC stands for descending. By default, the ORDER BY clause sorts the result set in ascending order if you don't specify ASC or DESC explicitly. Let's practice with some examples of using the ORDER BY clause.

MySQL ORDER BY examples

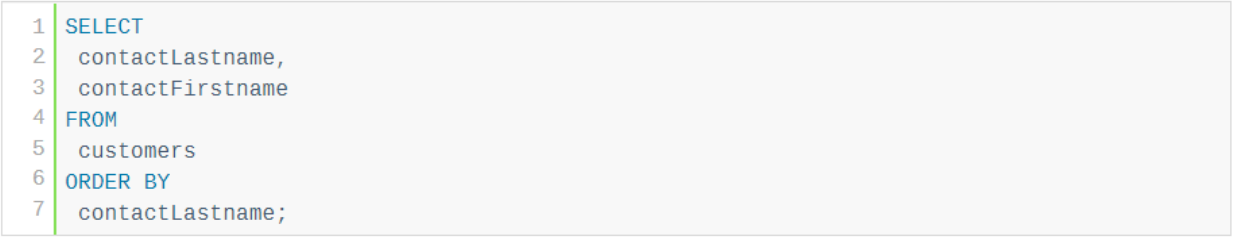
See the following customers table in the sample database.



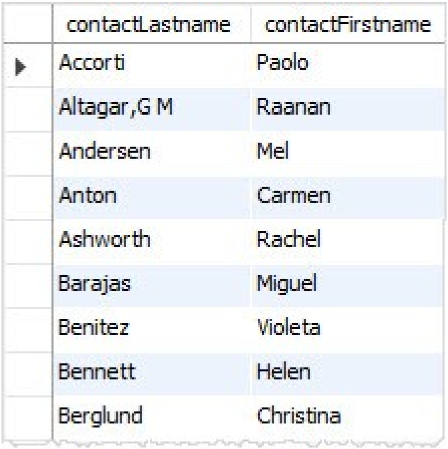
customers

- * customerNumber
- customerName
- contactLastName
- contactFirstName
- phone
- addressLine1
- addressLine2
- city
- state
- postalCode
- country
- salesRepEmployeeNumber
- creditLimit

The following query selects contacts from the `customers` table and sorts the contacts by last name in ascending order.

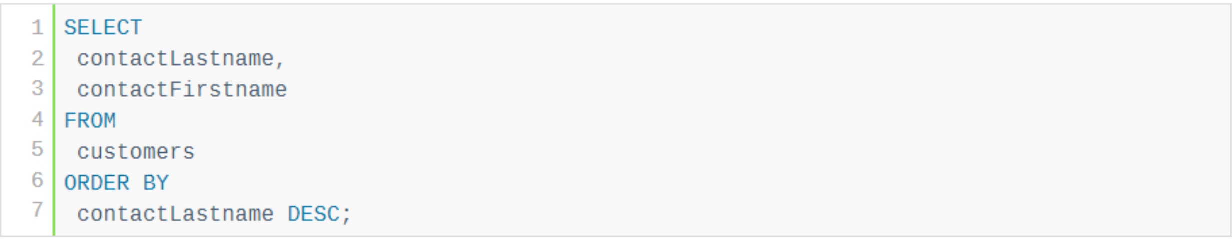


```
SELECT
 contactLastname,
 contactFirstname
FROM
 customers
ORDER BY
 contactLastname;
```

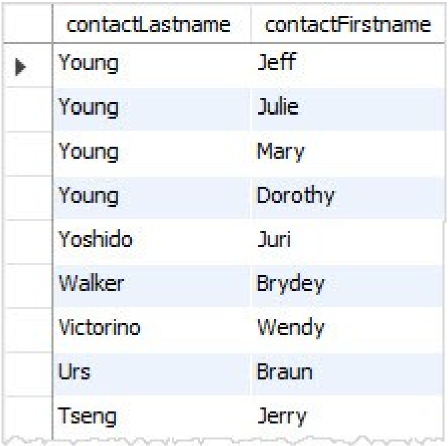


contactLastname	contactFirstname
Accorti	Paolo
Altagar,G M	Raanan
Andersen	Mel
Anton	Carmen
Ashworth	Rachel
Barajas	Miguel
Benitez	Violeta
Bennett	Helen
Berglund	Christina

If you want to sort the contacts by last name in descending order, you specify the `DESC` after the `contactLastname` column in the `ORDER BY` clause as the following query:



```
SELECT
 contactLastname,
 contactFirstname
FROM
 customers
ORDER BY
 contactLastname DESC;
```

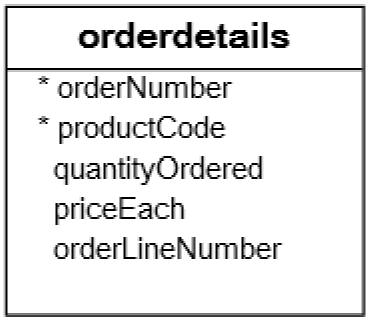


contactLastname	contactFirstname
Young	Jeff
Young	Julie
Young	Mary
Young	Dorothy
Yoshido	Juri
Walker	Brydey
Victorino	Wendy
Urs	Braun
Tseng	Jerry

In the query above, the `ORDER BY` clause sorts the result set by the last name in descending order first and then sorts the sorted result set by the first name in ascending order to produce the final result set.

MySQL ORDER BY sort by an expression example

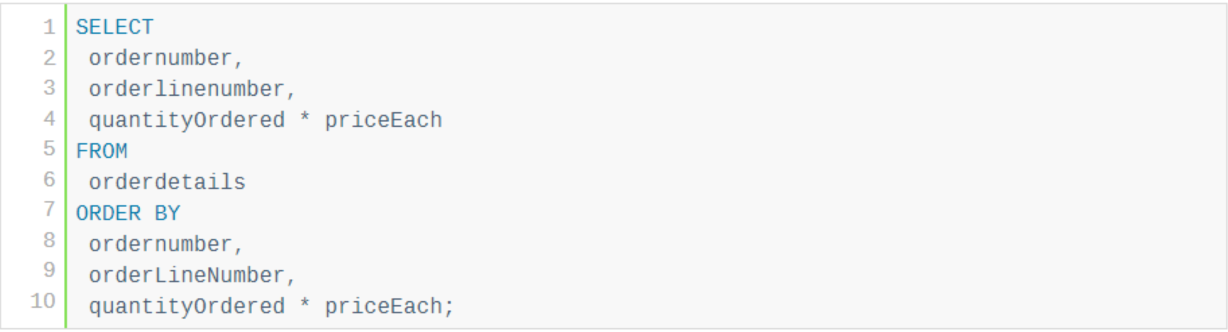
The `ORDER BY` clause also allows you to sort the result set based on an expression. See the following `orderdetails` table.



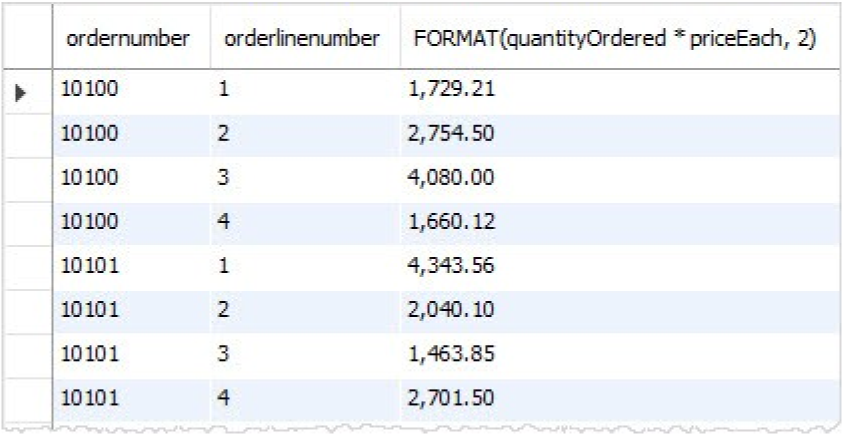
orderdetails

* orderNumber
* productCode
quantityOrdered
priceEach
orderLineNumber

The following query selects the order line items from the orderdetails table. It calculates the subtotal for each line item and sorts the result set based on the order number, order line number, and subtotal.



```
SELECT
 ordernumber,
 orderlinenumber,
 quantityOrdered * priceEach
FROM
 orderdetails
ORDER BY
 ordernumber,
 orderLineNumber,
 quantityOrdered * priceEach;
```



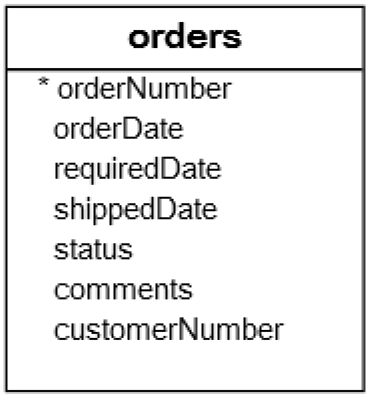
ordernumber	orderlinenumber	FORMAT(quantityOrdered * priceEach, 2)
10100	1	1,729.21
10100	2	2,754.50
10100	3	4,080.00
10100	4	1,660.12
10101	1	4,343.56
10101	2	2,040.10
10101	3	1,463.85
10101	4	2,701.50

We used subtotal as the column alias for the expression quantityOrdered * priceEach and sorted the result set based on the subtotal alias.

MySQL ORDER BY with custom sort order

The ORDER BY clause enables you to define your own custom sort order for the values in a column using the FIELD () function.

See the following orders table.

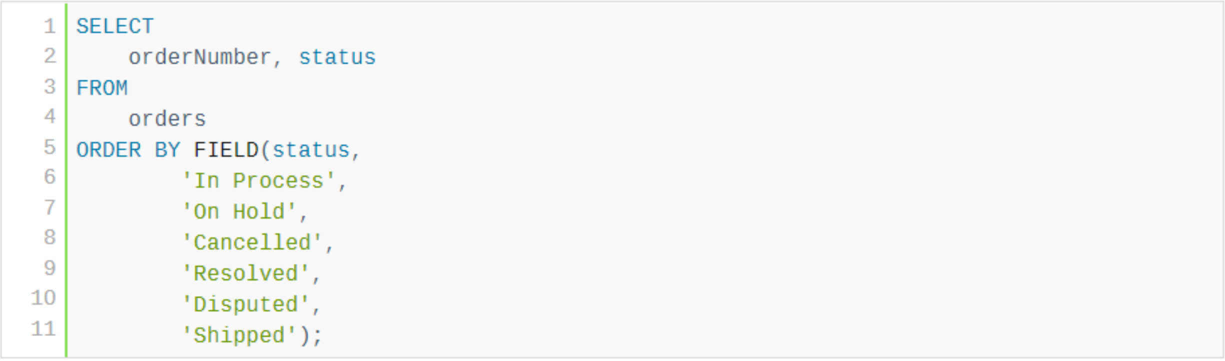


orders
* orderNumber
orderDate
requiredDate
shippedDate
status
comments
customerNumber

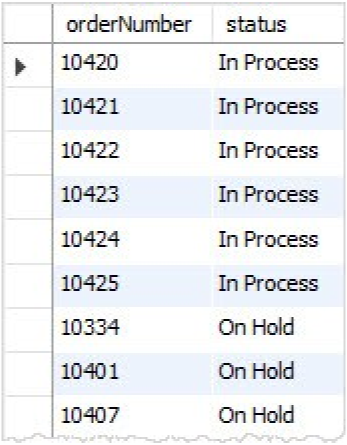
For example, if you want to sort the orders based on the following status by the following order:

- In Process
- On Hold
- Canceled
- Resolved
- Disputed
- Shipped

You can use the FIELD function to map those values to a list of numeric values and use the numbers for sorting; See the following query:



```
SELECT
    orderNumber, status
FROM
    orders
ORDER BY FIELD(status,
        'In Process',
        'On Hold',
        'Cancelled',
        'Resolved',
        'Disputed',
        'Shipped');
```



orderNumber	status
10420	In Process
10421	In Process
10422	In Process
10423	In Process
10424	In Process
10425	In Process
10334	On Hold
10401	On Hold
10407	On Hold

In this tutorial, we've shown you various techniques to sort a result set by using the MySQL ORDER BY clause.

MySQL WHERE

Summary*:* you will learn how to use **MySQL WHERE** clause in the SELECT statement to filter rows in the result set.

Introduction to MySQL WHERE clause

The WHERE clause allows you to specify the search condition for the rows returned by the query. The following shows the syntax of the WHERE clause:

```
SELECT
    select_list
FROM
    table_name
WHERE
    search_condition;
```

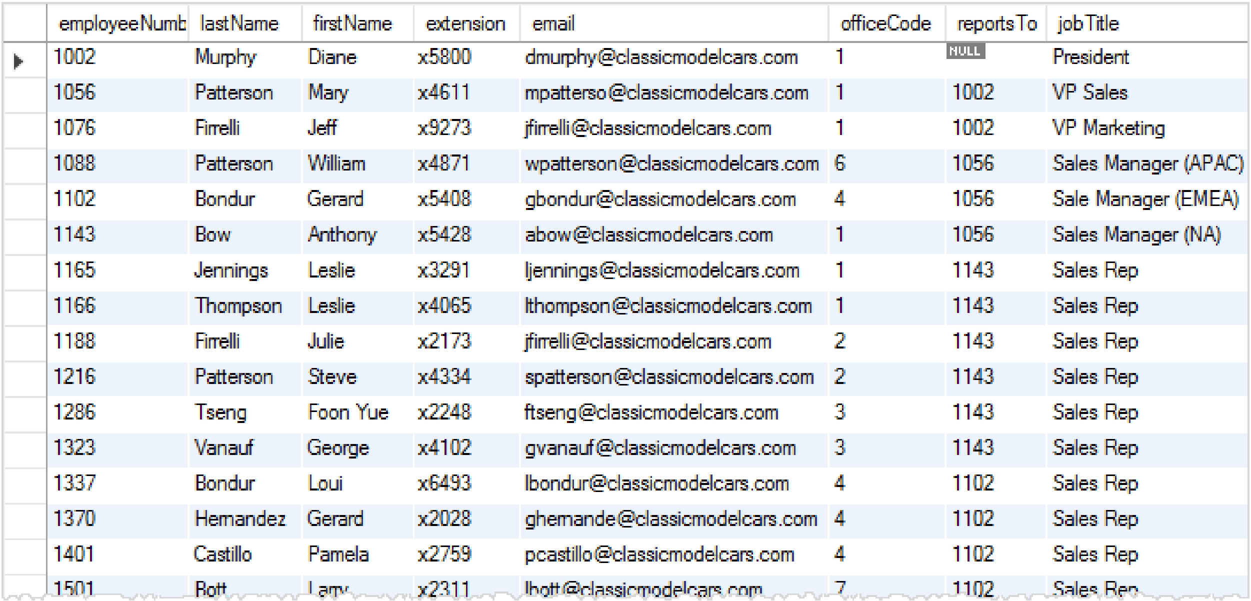
The search_condition is a combination of one or more predicates using the logical operator AND, ORand NOT. In SQL, a predicate is an expression that evaluates to true, false, or unknown.

Any row from the table_name that causes the search_condition to evaluate to true will be included in the final result set.

Besides the SELECT statement, you can use the WHERE clause in the UPDATE and DELETE statement to specify which rows to update and delete.

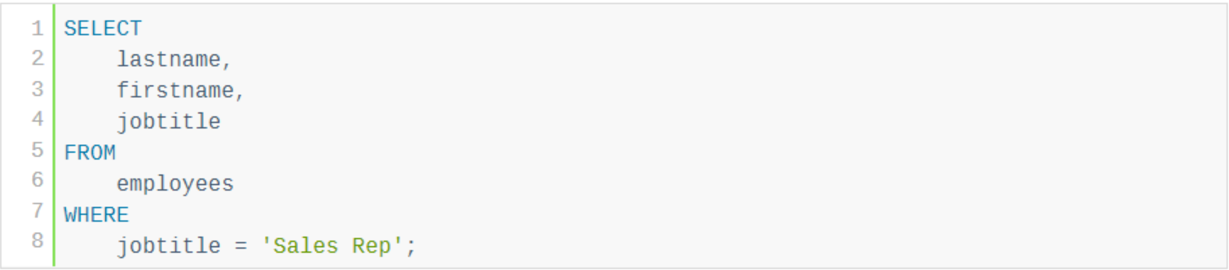
MySQL WHERE clause examples

We will use the employees table from the sample database for the demonstration.

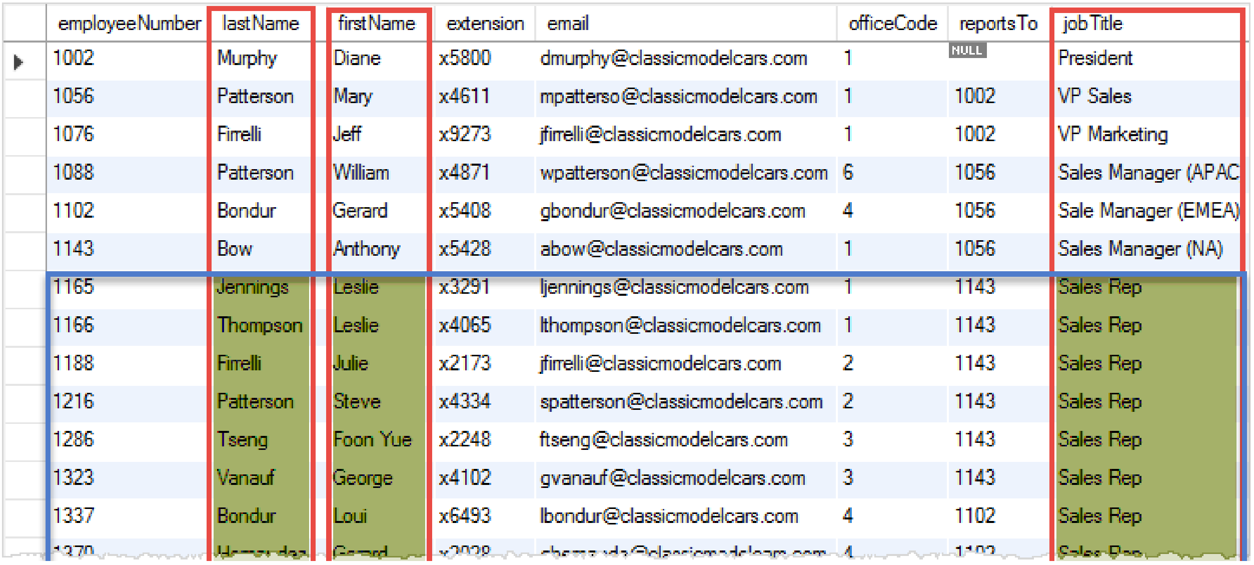


employeeNumb	lastName	firstName	extension	email	officeCode	reportsTo	jobTitle
1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	1	NULL	President
1056	Patterson	Mary	x4611	mpatterso@classicmodelcars.com	1	1002	VP Sales
1076	Firrelli	Jeff	x9273	jfirrelli@classicmodelcars.com	1	1002	VP Marketing
1088	Patterson	William	x4871	wpatterson@classicmodelcars.com	6	1056	Sales Manager (APAC)
1102	Bondur	Gerard	x5408	gbondur@classicmodelcars.com	4	1056	Sale Manager (EMEA)
1143	Bow	Anthony	x5428	abow@classicmodelcars.com	1	1056	Sales Manager (NA)
1165	Jennings	Leslie	x3291	ljennings@classicmodelcars.com	1	1143	Sales Rep
1166	Thompson	Leslie	x4065	lthompson@classicmodelcars.com	1	1143	Sales Rep
1188	Firrelli	Julie	x2173	jfirrelli@classicmodelcars.com	2	1143	Sales Rep
1216	Patterson	Steve	x4334	spatterson@classicmodelcars.com	2	1143	Sales Rep
1286	Tseng	Foon Yue	x2248	ftseng@classicmodelcars.com	3	1143	Sales Rep
1323	Vanauf	George	x4102	gvanauf@classicmodelcars.com	3	1143	Sales Rep
1337	Bondur	Loui	x6493	lbondur@classicmodelcars.com	4	1102	Sales Rep
1370	Hernandez	Gerard	x2028	ghernande@classicmodelcars.com	4	1102	Sales Rep
1401	Castillo	Pamela	x2759	pcastillo@classicmodelcars.com	4	1102	Sales Rep
1501	Bott	Larry	x2311	lbott@classicmodelcars.com	7	1102	Sales Rep

The following query finds the employees whose job title is `Sales Rep`:



```sql
SELECT
    lastname,
    firstname,
    jobtitle
FROM
    employees
WHERE
    jobtitle = 'Sales Rep';
```

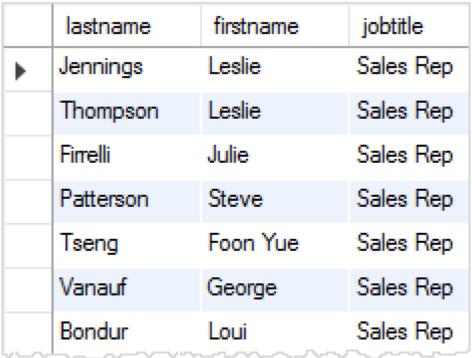


employeeNumber	lastName	firstName	extension	email	officeCode	reportsTo	jobTitle
1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	1	NULL	President
1056	Patterson	Mary	x4611	mpatterso@classicmodelcars.com	1	1002	VP Sales
1076	Firrelli	Jeff	x9273	jfirrelli@classicmodelcars.com	1	1002	VP Marketing
1088	Patterson	William	x4871	wpatterson@classicmodelcars.com	6	1056	Sales Manager (APAC
1102	Bondur	Gerard	x5408	gbondur@classicmodelcars.com	4	1056	Sale Manager (EMEA)
1143	Bow	Anthony	x5428	abow@classicmodelcars.com	1	1056	Sales Manager (NA)
1165	Jennings	Leslie	x3291	ljennings@classicmodelcars.com	1	1143	Sales Rep
1166	Thompson	Leslie	x4065	lthompson@classicmodelcars.com	1	1143	Sales Rep
1188	Firrelli	Julie	x2173	jfirrelli@classicmodelcars.com	2	1143	Sales Rep
1216	Patterson	Steve	x4334	spatterson@classicmodelcars.com	2	1143	Sales Rep
1286	Tseng	Foon Yue	x2248	ftseng@classicmodelcars.com	3	1143	Sales Rep
1323	Vanauf	George	x4102	gvanauf@classicmodelcars.com	3	1143	Sales Rep
1337	Bondur	Loui	x6493	lbondur@classicmodelcars.com	4	1102	Sales Rep
1370	Hernandez	Gerard	x2028	ghernande@classicmodelcars.com	4	1102	Sales Rep

Even though the WHERE clause appears at the end of the statement, MySQL evaluates the expression in the WHERE clause first to select the matching rows. It chooses the rows that have a job title as Sales Rep

```
jobtitle = 'Sales Rep';
```

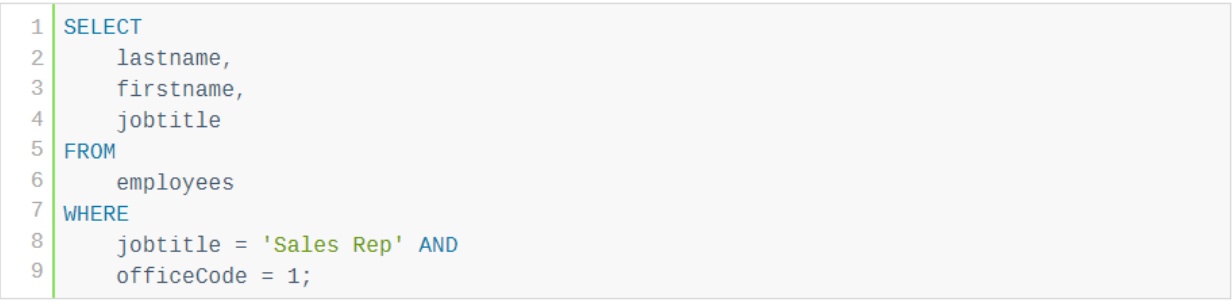
MySQL then selects the columns from the select list in the SELECT clause. The highlighted area contains the columns and rows in the final result set.



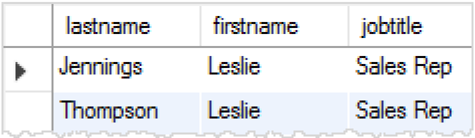
lastname	firstname	jobtitle
Jennings	Leslie	Sales Rep
Thompson	Leslie	Sales Rep
Firrelli	Julie	Sales Rep
Patterson	Steve	Sales Rep
Tseng	Foon Yue	Sales Rep
Vanauf	George	Sales Rep
Bondur	Loui	Sales Rep

You can form a simple condition like the query above, or a very complex one that combines multiple expressions with logical operators.

For example, to find all sales rep in the office code 1, you use the following query:

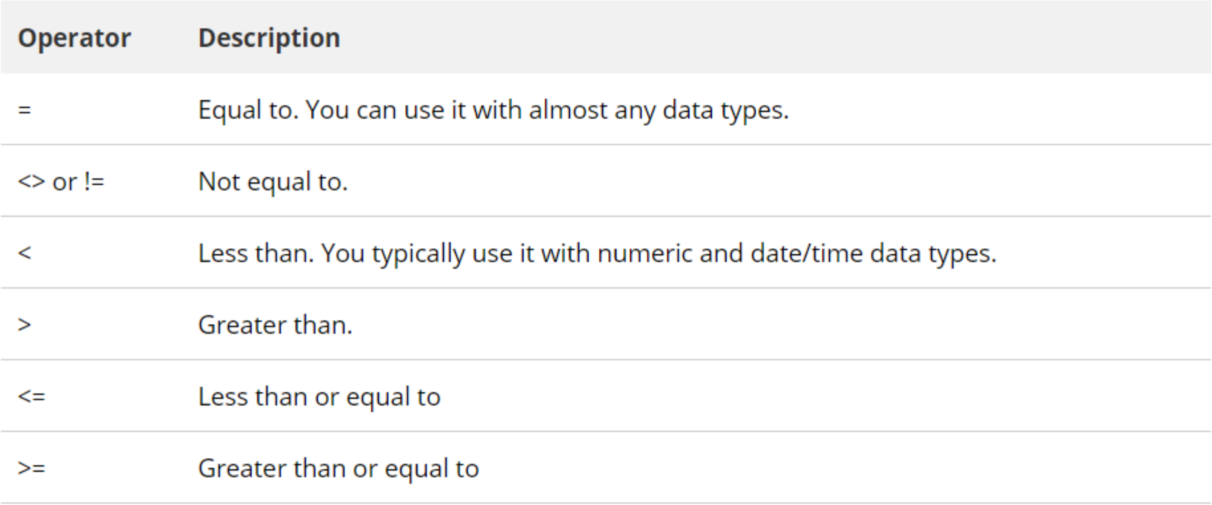


```
SELECT
    lastname,
    firstname,
    jobtitle
FROM
    employees
WHERE
    jobtitle = 'Sales Rep' AND
    officeCode = 1;
```



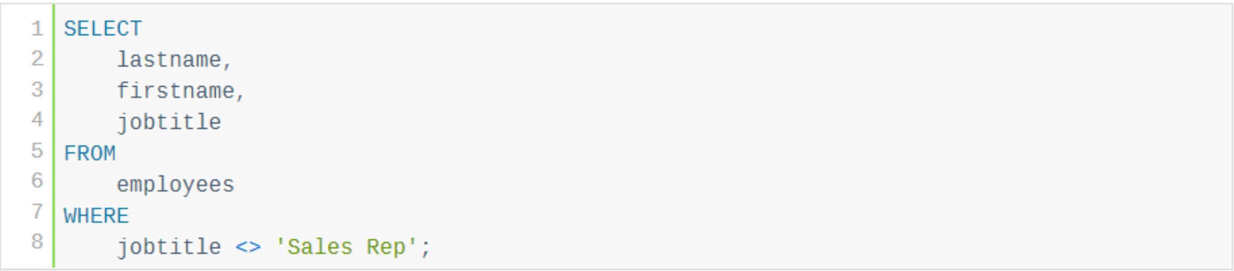
	lastname	firstname	jobtitle
▶	Jennings	Leslie	Sales Rep
	Thompson	Leslie	Sales Rep

The following table shows the comparison operators that you can use to form filtering expressions in the `WHERE` clause.

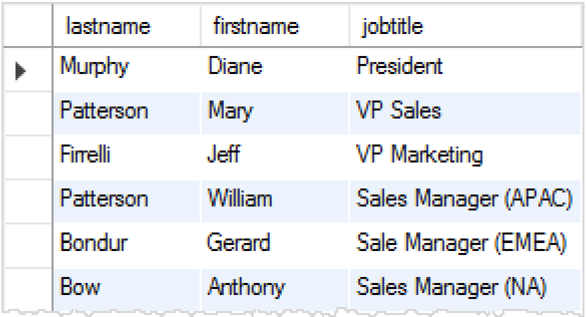


Operator	Description
=	Equal to. You can use it with almost any data types.
<> or !=	Not equal to.
<	Less than. You typically use it with numeric and date/time data types.
>	Greater than.
<=	Less than or equal to
>=	Greater than or equal to

The following query uses the not equal to (<>) operator to find all employees who are not the Sales Rep:

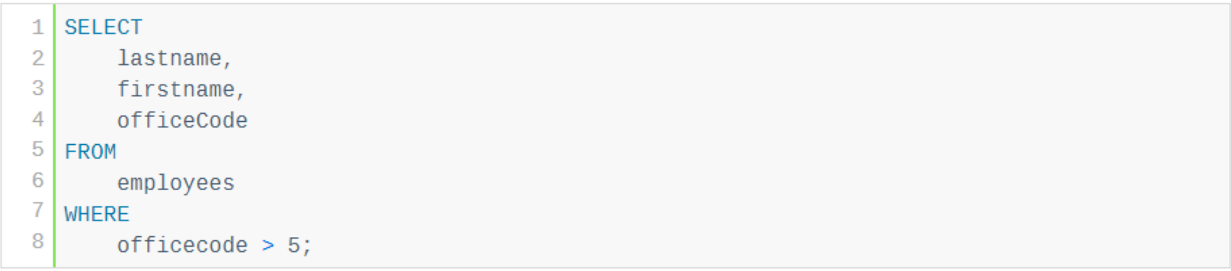


```
SELECT
    lastname,
    firstname,
    jobtitle
FROM
    employees
WHERE
    jobtitle <> 'Sales Rep';
```

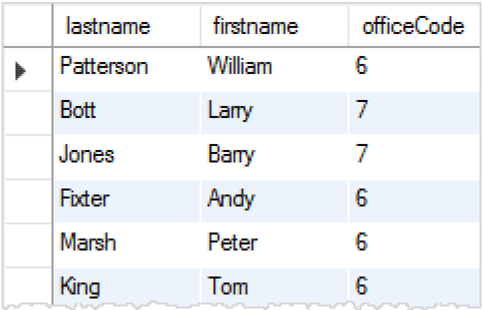


lastname	firstname	jobtitle
Murphy	Diane	President
Patterson	Mary	VP Sales
Firrelli	Jeff	VP Marketing
Patterson	William	Sales Manager (APAC)
Bondur	Gerard	Sale Manager (EMEA)
Bow	Anthony	Sales Manager (NA)

The following query find employees whose office code is greater than 5:

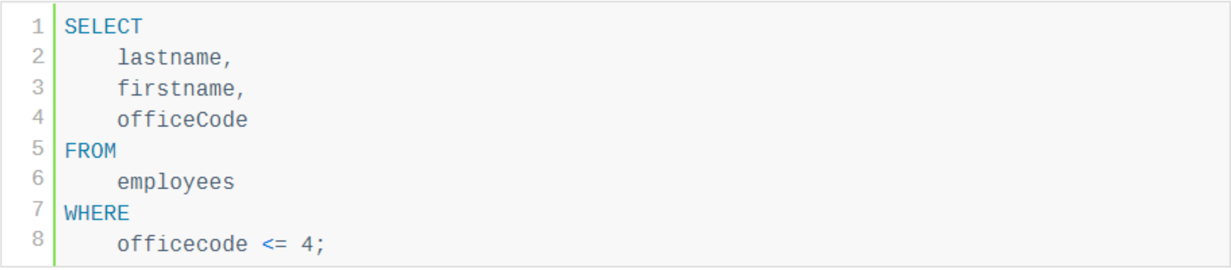


```sql
SELECT
    lastname,
    firstname,
    officeCode
FROM
    employees
WHERE
    officecode > 5;
```

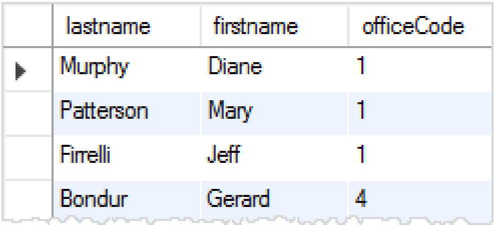


lastname	firstname	officeCode
Patterson	William	6
Bott	Larry	7
Jones	Barry	7
Fixter	Andy	6
Marsh	Peter	6
King	Tom	6

The following query returns employees with office code less than or equal 4 (<=4):



```sql
SELECT
    lastname,
    firstname,
    officeCode
FROM
    employees
WHERE
    officecode <= 4;
```



	lastname	firstname	officeCode
▶	Murphy	Diane	1
	Patterson	Mary	1
	Firrelli	Jeff	1
	Bondur	Gerard	4

More on MySQL WHERE clause…

MySQL provides you with some other operators for using in the WHERE clause to form complex search conditions such as:

- BETWEEN selects values within a range of values.
- LIKE matches values based on pattern matching.
- IN specifies if a value matches any value in a set.
- IS NULL checks if the value is NULL.

In this tutorial, you have learned how to use the MySQL WHERE clause to filter rows based on conditions.

MySQL AND Operator

Summary: in this tutorial, you will learn how to the MySQL AND operator to combine multiple Boolean expressions to filter data.

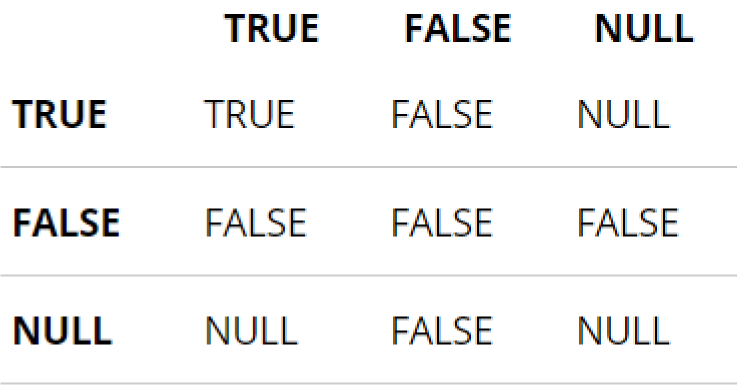
Introduction to MySQL AND operator

The AND operator is a logical operator that combines two or more Boolean expressions and returns true only if both expressions evaluate to true. The AND operator returns false if one of the two expressions evaluate to false.

Here is the syntax of the AND operator:

```
boolean_expression_1 AND boolean_expression_2
```

The following table illustrates the results of the AND operator when combining true, false, and null.

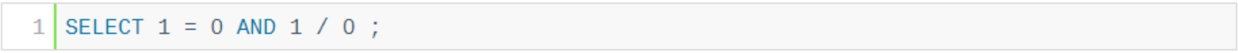


	TRUE	**FALSE**	**NULL**
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

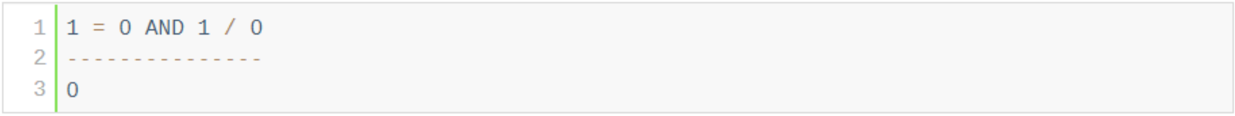
The AND operator is often used in the WHERE clause of the SELECT, UPDATE, DELETE statement to form a condition. The AND operator is also used in join conditions of the INNER JOIN and LEFT JOIN clauses.

When evaluating an expression that has the AND operator, MySQL stops evaluating the remaining parts of the expression whenever it can determine the result. This function is called short-circuit evaluation.

Consider the following example.



```
SELECT 1 = 0 AND 1 / 0 ;
```



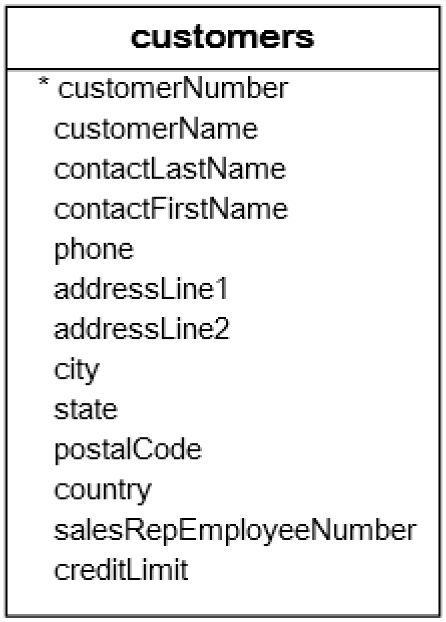
```
1 = 0 AND 1 / 0
---------------
0
```

Note that in MySQL, zero is considered false and non-zero is treated as true.

MySQL only evaluates the first part `1 = 0` of the expression `1 = 0 AND 1 / 0`. Because the expression `1 = 0` returns false, MySQL can conclude the result of the whole expression, which is false. MySQL does not evaluate the remaining part of the expression, which is 1/0; If it did, it would issue an error because of the division by zero error.

MySQL AND operator examples

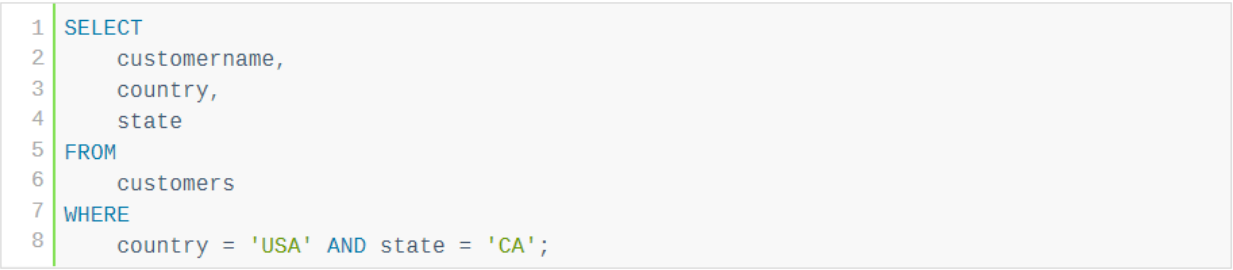
Let’s use the `customers` table in the sample database for the demonstration.



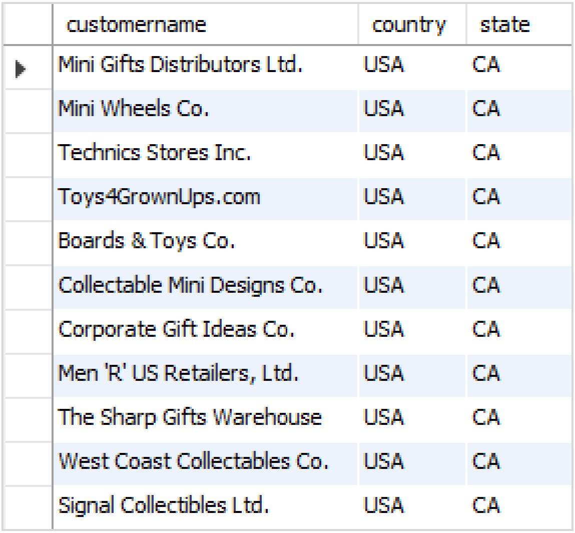
customers

* customerNumber
customerName
contactLastName
contactFirstName
phone
addressLine1
addressLine2
city
state
postalCode
country
salesRepEmployeeNumber
creditLimit

The following statement use the AND operator to find customers who locate in California (CA), USA:



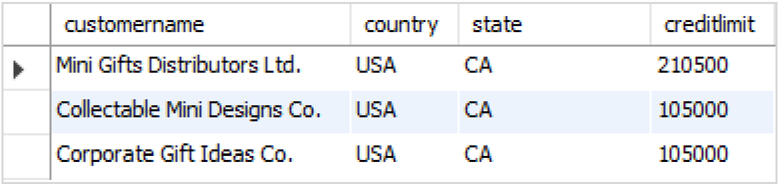
```sql
SELECT
    customername,
    country,
    state
FROM
    customers
WHERE
    country = 'USA' AND state = 'CA';
```



customername	country	state
Mini Gifts Distributors Ltd.	USA	CA
Mini Wheels Co.	USA	CA
Technics Stores Inc.	USA	CA
Toys4GrownUps.com	USA	CA
Boards & Toys Co.	USA	CA
Collectable Mini Designs Co.	USA	CA
Corporate Gift Ideas Co.	USA	CA
Men 'R' US Retailers, Ltd.	USA	CA
The Sharp Gifts Warehouse	USA	CA
West Coast Collectables Co.	USA	CA
Signal Collectibles Ltd.	USA	CA

By using the AND operator, you can combine more than two Boolean expressions. For example, the following query returns the customers who locate in California, USA, and have the credit limit greater than 100K.

```
SELECT
 customername,
 country,
 state,
 creditlimit
FROM
 customers
WHERE country = 'USA'
 AND state = 'CA'
 AND creditlimit > 100000;
```



customername	country	state	creditlimit
Mini Gifts Distributors Ltd.	USA	CA	210500
Collectable Mini Designs Co.	USA	CA	105000
Corporate Gift Ideas Co.	USA	CA	105000

MySQL OR Operator

Summary: in this tutorial, you will learn how to use the MySQL OR operator to combine Boolean expressions for filtering data.

Introduction to the MySQL OR operator

The MySQL OR operator combines two Boolean expressions and returns true when either condition is true.
The following illustrates the syntax of the OR operator.

```
boolean_expression_1 OR boolean_expression_2
```

Both boolean_expression_1 and boolean_expression_2 are Boolean expressions that return true, false, or NULL.
The following table shows the result of the OR operator.

	TRUE	**FALSE**	**NULL**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

MySQL OR short-circuit evaluation

MySQL uses short-circuit evaluation for the OR operator. In other words, MySQL stops evaluating the remaining parts of the statement when it can determine the result.
See the following example.

```
SELECT 1 = 1 OR 1 / 0;
```

```
1 = 1 OR 1 / 0
--------------
1
```

Home / Basic MySQL Tutorial / MySQL OR Operator

MySQL OR Operator

Summary: in this tutorial, you will learn how to use the MySQL OR operator to combine Boolean expressions for filtering data.

Introduction to the MySQL `OR` operator

The MySQL `OR` operator combines two Boolean expressions and returns true when either condition is true.

The following illustrates the syntax of the `OR` operator.

```
boolean_expression_1 OR boolean_expression_2
```

Both `boolean_expression_1` and `boolean_expression_2` are Boolean expressions that return true, false, or NULL.

The following table shows the result of the `OR` operator.

	TRUE	**FALSE**	**NULL**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

MySQL `OR` short-circuit evaluation

MySQL uses short-circuit evaluation for the `OR` operator. In other words, MySQL stops evaluating the remaining parts of the statement when it can determine the result.

See the following example.

```
SELECT 1 = 1 OR 1 / 0;
```

Try It Out

```
1 = 1 OR 1 / 0
--------------
1
```

Because the expression 1 = 1 always returns true, MySQL does not evaluate the 1 / 0. If it did, it would issue an error because of the division by zero error.

Operator precedence

When you use more than one logical operator in an expression, MySQL always evaluates the `OR` operators after the AND operators. This is called operator precedence which determines the order of evaluation of the operators. MySQL evaluates the operator with the higher precedence first.

See the following example.

```
SELECT true OR false AND false;
```

```
true OR false AND false
-----------------------
1
```

How it works

3. First, MySQL evaluates the `AND` operator, therefore the expression `false AND false` returns false.
4. Second, MySQL evaluates the `OR` operator hence the expression `true OR false` returns true.

To change the order of evaluation, you use the parentheses, for example:

```
SELECT (true OR false) AND false;
```

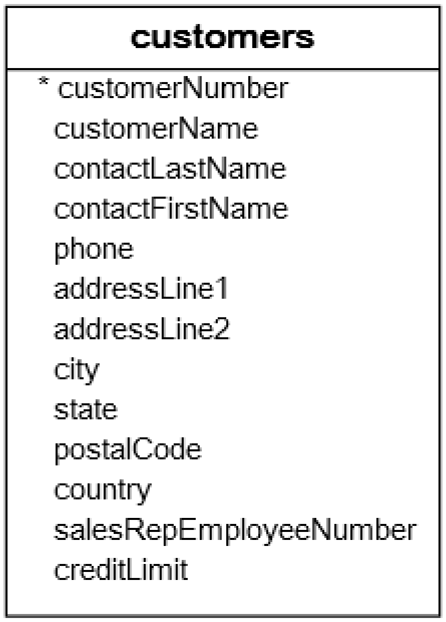
```
(true OR false) AND false
-------------------------
0
```

How it works

3. First, MySQL evaluates the expression in the parenthesis `(true OR false)` returns true
4. Second, MySQL evaluates the remaining part of the statement, `true AND false` returns false.

MySQL `OR` operator examples

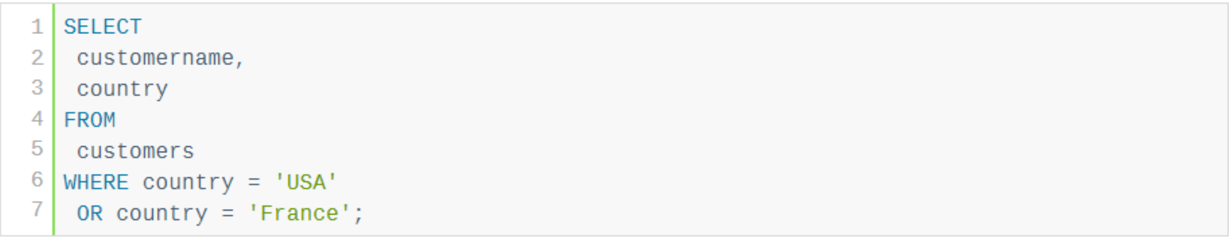
We will use the customers table in the sample database for the demonstration.



customers

* customerNumber
customerName
contactLastName
contactFirstName
phone
addressLine1
addressLine2
city
state
postalCode
country
salesRepEmployeeNumber
creditLimit

For example, to get the customers who locate in the USA or France, you use the `OR` operator in the `WHERE`clause as follows:



```
SELECT
 customername,
 country
FROM
 customers
WHERE country = 'USA'
 OR country = 'France';
```

To learn more, visit https://www.mysqltutorial.org/