

Manage Database in MySQL

Creating Database

Before doing anything else with the data, you need to create a database. A database is a container of data. It stores contacts, vendors, customers or any kind of data that you can think of. In MySQL, a database is a collection of objects that are used to store and manipulate data such as tables, database views, triggers, stored procedures, etc.

To create a database in MySQL, you use the `CREATE DATABASE` statement as follows:

```
1 CREATE DATABASE [IF NOT EXISTS] database_name;
```

Let's examine the `CREATE DATABASE` statement in greater detail:

- Followed by the `CREATE DATABASE` statement is database name that you want to create. It is recommended that the database name should be as meaningful and descriptive as possible.
- The `IF NOT EXISTS` is an optional element of the statement. The `IF NOT EXISTS` statement prevents you from an error of creating a new database that already exists in the database server. You cannot have 2 databases with the same name in a MySQL database server.

For example, to create `classicmodels` database, you can execute the `CREATE DATABASE` statement as follows:

```
1 CREATE DATABASE classicmodels;
```

After executing the statement, MySQL returns a message to notify that the new database has been created successfully or not.

Displaying Databases

The `SHOW DATABASE` statement displays all databases in the MySQL database server. You can use the `SHOW DATABASE` statement to check the database that you've created or to see all the databases on the database server before you create a new database, for example:

```
1 SHOW DATABASES;
```

Database
information_schema
classicmodels
mysql

We have three databases in the MySQL database server. The `information_schema` and `mysql` are the default databases that are available when we install MySQL, and the `classicmodels` is the new database that we have created.

Selecting a database to work with

Before working with a particular database, you must tell MySQL which database you want to work with by using the `USE` statement.

```
1 USE database_name;
```

You can select the `classicmodels` sample database using the `USE` statement as follows:

```
1 USE classicmodels;
```

From now all operations such as querying data, create new tables or stored procedures which you perform, will take effects on the current database.

Removing Databases

Removing database means you delete the database physically. All the data and related objects inside the database are permanently deleted and this cannot be undone, therefore it is very important to execute this query with extra cautions.

To delete a database, you use the `DROP DATABASE` statement as follows:

```
1 DROP DATABASE [IF EXISTS] database_name;
```

Followed the `DROP DATABASE` is the database name that you want to remove. Similar to the `CREATE DATABASE` statement, the `IF EXISTS` is an optional part of the statement to prevent you from removing a database that does not exist in the database server.

If you want to practice with the `DROP DATABASE` statement, you can create a new database, make sure that it is created and remove it. Take a look at the following queries:

```
1 CREATE DATABASE IF NOT EXISTS temp_database;
```

```
2 SHOW DATABASES;
```

```
3 DROP DATABASE IF EXISTS temp_database;
```

Understanding MySQL Table Types, or Storage Engines

Summary: in this tutorial, you will learn various **MySQL table types**, or storage engines. It is essential to understand the features of each table type in MySQL so that you can use them effectively to maximize the performance of your databases.

MySQL provides various storage engines for its tables as below:

- MyISAM
- InnoDB
- MERGE
- MEMORY (HEAP)
- ARCHIVE
- CSV
- FEDERATED

Each storage engine has its own advantages and disadvantages. It is crucial to understand each storage engine features and choose the most appropriate one for your tables to maximize the performance of the database. In the following sections we will discuss about each storage engine and its features so that you can decide which one to use.

MyISAM

MyISAM extends the former ISAM storage engine. The MyISAM tables are optimized for compression and speed. MyISAM tables are also portable between platforms and OSes.

The size of MyISAM table can be up to 256TB, which is huge. In addition, MyISAM tables can be compressed into read-only tables to save space. At startup, MySQL checks MyISAM tables for corruption and even repair them in case of errors. The MyISAM tables are not transaction-safe.

Before MySQL version 5.5, MyISAM is the default storage engine when you create a table without explicitly specify the storage engine. From version 5.5, MySQL uses InnoDB as the default storage engine.

InnoDB

The InnoDB tables fully support ACID-compliant and transactions. They are also very optimal for performance. InnoDB table supports foreign keys, commit, rollback, roll-and-forward operations. The size of the InnoDB table can be up to 64TB.

Like MyISAM, the InnoDB tables are portable between different platforms and OSes. MySQL also checks and repair InnoDB tables, if necessary, at startup.

MERGE

A MERGE table is a virtual table that combines multiple MyISAM tables, which has similar structure, into one table. The MERGE storage engine is also known as the MRG_MyISAM engine. The MERGE table does not have its own indexes; it uses indexes of the component tables instead.

Using MERGE table, you can speed up performance in joining multiple tables. MySQL only allows you to perform SELECT, DELETE, UPDATE and INSERT operations on the MERGE tables. If you use DROP TABLE statement on a MERGE table, only MERGE specification is removed. The underlying tables will not be affected.

Memory

The memory tables are stored in memory and used hash indexes so that they are faster than MyISAM tables. The lifetime of the data of the memory tables depends on the up time of the database server. The memory storage engine is formerly known as HEAP.

Archive

The archive storage engine allows you to store a large number of records, which for archiving purpose, into a compressed format to save disk space. The archive storage engine compresses a record when it is inserted and decompress it using *zlib* library as it is read.

The archive tables only allow INSERT and SELECT commands. The archive tables do not support indexes, so reading records requires a full table scanning.

CSV

The CSV storage engine stores data in comma-separated values file format. A CSV table brings a convenient way to migrate data into non-SQL applications such as spreadsheet software.

CSV table does not support NULL data type and read operation requires a full table scan.

FEDERATED

The FEDERATED storage engine allows you to manage data from a remote MySQL server without using cluster or replication technology. The local federated table stores no data. When you query data from a local federated table, the data is pull automatically from the remote federated tables.

MySQL Data Types

Database table contains multiple columns with specific data types such as numeric or string. MySQL provides more data types other than just numeric or string. Each data type in MySQL can be determined by the following characteristics:

- Kind of values it can represent.
- The space that takes up and whether the values are fixed-length or variable-length.
- Does the values of the data type can be indexed.
- How MySQL compares the value of a specific data type.

Numeric Data Types

You can find all SQL standard numeric types in MySQL including exact number data type and approximate numeric data types including integer, fixed-point and floating point. In addition, MySQL also supports BIT data type for storing bit field values. Numeric types can be signed or unsigned except the BIT type. The following table shows you the summary of numeric types in MySQL:

Numeric Types	Description
TINYINT	A very small integer
SMALLINT	A small integer
MEDIUMINT	A medium-sized integer
INT	A standard integer
BIGINT	A large integer
DECIMAL	A fixed-point number
FLOAT	A single-precision floating-point number
DOUBLE	A double-precision floating-point number
BIT	A bit field

String Data Types

In MySQL, string can hold anything from plain text to binary data such as images and files. String can be compared and searched based on pattern matching by using the LIKE operator or regular expression. The following table shows you the string data types in MySQL:

String Types	Description
CHAR	A fixed-length non-binary (character) string
VARCHAR	A variable-length non-binary string
BINARY	A fixed-length binary string
VARBINARY	A variable-length binary string
TINYBLOB	A very small BLOB (binary large object)
BLOB	A small BLOB
MEDIUMBLOB	A medium-sized BLOB
LOBLOB	A large BLOB
TINYTEXT	A very small non-binary string
TEXT	A small non-binary string
MEDIUMTEXT	A medium-sized non-binary string
LONGTEXT	A large non-binary string
ENUM	An enumeration; each column value may be assigned one enumeration member
SET	A set; each column value may be assigned zero or more set members

Date and Time Data Types

MySQL provides types for date and time as well as a combination of date and time. In addition, MySQL also provides timestamp data type for tracking the changes of a row in a table. If you just want to store the year without date and month, you can use YEAR data type. The following table illustrates the MySQL date and time data types:

Date and Time Types	Description
DATE	A date value in 'CCYY-MM-DD' format
TIME	A time value in 'hh:mm:ss' format
DATETIME	A date and time value in 'CCYY-MM-DD hh:mm:ss' format

Date and Time Types	Description
TIMESTAMP	A timestamp value in 'CCYY-MM-DD hh:mm:ss' format
YEAR	A year value in CCYY or YY format

Spatial Data Types

MySQL supports many spatial data types that contain various kind of geometrical and geographical values as shown in the following table:

Spatial Data Types	Description
GEOMETRY	A spatial value of any type
POINT	A point (a pair of X Y coordinates)
LINESTRING	A curve (one or more POINT values)
POLYGON	A polygon
GEOMETRYCOLLECTION	A collection of GEOMETRY values
MULTILINESTRING	A collection of LINESTRING values
MULTIPOINT	A collection of POINT values
MULTIPOLYGON	A collection of POLYGON values

Creating Tables Using MySQL CREATE TABLE Statement

MySQL CREATE TABLE syntax

In order to create a new table within a database, you use the MySQL `CREATE TABLE` statement. The `CREATE TABLE` statement is one of the most complex statement in MySQL.

The following illustrates the syntax of the `CREATE TABLE` statement in the simple form:

```
1 CREATE TABLE [IF NOT EXISTS] table_name(
2     column_list
3 ) engine=table_type
```

Let's examine the syntax in greater detail:

- First, you specify the name of table that you want to create after the `CREATE TABLE` keywords. The table name must be unique within a database. The `IF NOT EXISTS` is an optional part of the statement that allows you to check if the table you are creating already exists in the database. If this is the case, MySQL will ignore the whole statement and it will not create any new table. It is highly recommended that you to use `IF NOT EXISTS` in every `CREATE TABLE` statement for preventing from an error of creating a new table that already exists.
- Second, you specify a list of columns for the table in the `column_list` section. Columns are separated by a comma (,). We will show you how to define columns in more detail in the next section.
- Third, you need to specify the storage engine for the table in the `engine` clause. You can use any storage engine such as InnoDB, MyISAM, HEAP, EXAMPLE, CSV, ARCHIVE, MERGE FEDERATED or NDBCLUSTER. If you don't declare the storage engine explicitly, MySQL will use InnoDB by default.

InnoDB became the default storage engine since MySQL version 5.5. The InnoDB table type brings many benefits of relational database management system such as ACID transaction, referential integrity and crash recovery. In the previous versions, MySQL used MyISAM as the default storage engine.

To define a column for the table in the `CREATE TABLE` statement, you use the following syntax:

```
1 column_name data_type[size] [NOT NULL|NULL] [DEFAULT value]
2 [AUTO_INCREMENT]
```

The most important components of the syntax above are:

- The `column_name` specifies the name of the column. Each column always associates with a specific data type and the size e.g., `VARCHAR(255)`.
- The `NOT NULL` or `NULL` indicates that the column accepts `NULL` value or not.
- The `DEFAULT value` is used to specify the default value of the column.
- The `AUTO_INCREMENT` indicates that the value of column is increased by one whenever a new row is inserted into the table. Each table has one and only one `AUTO_INCREMENT` column.

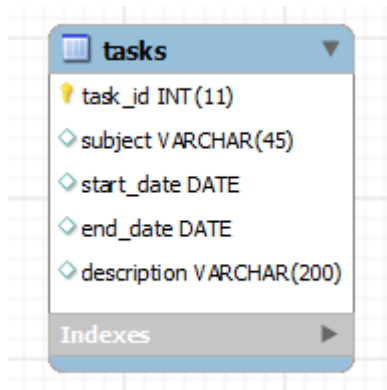
If you want to set particular columns of the table as the primary key, you use the following syntax:

1 PRIMARY KEY (col1,col2,...)

Example of MySQL CREATE TABLE statement

Let's practice with an example of creating a new table named `tasks` in our sample database as follows:

You can use the `CREATE TABLE` statement to create the `tasks` table as follows:



```
1 CREATE TABLE IF NOT EXISTS tasks (
2   task_id int(11) NOT NULL AUTO_INCREMENT,
3   subject varchar(45) DEFAULT NULL,
4   start_date DATE DEFAULT NULL,
5   end_date DATE DEFAULT NULL,
6   description varchar(200) DEFAULT NULL,
7   PRIMARY KEY (task_id)
8 ) ENGINE=InnoDB
```

MySQL Sequence

MySQL create sequence

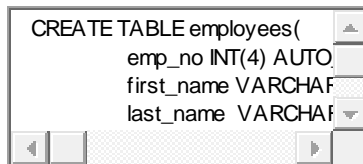
In MySQL, a sequence is a list of integers generated in the ascending order i.e., 1,2,3... Many applications need sequences to generate unique numbers mainly for identification e.g., customer ID in CRM, employee number in HR, equipment number in services management system, etc.

To create a sequence in MySQL automatically, you set the `AUTO_INCREMENT` attribute to a column, which typically is primary key column. The following are rules that you must follow when you use `AUTO_INCREMENT` attribute:

- Each table has only one `AUTO_INCREMENT` column whose data type is typically integer or float which is very rare.
- The `AUTO_INCREMENT` column must be indexed, which means it can be either `PRIMARY KEY` or `UNIQUE` index.
- The `AUTO_INCREMENT` column must have `NOT NULL` constraint. When you set `AUTO_INCREMENT` attribute to a column, MySQL will make it `NOT NULL` for you in case you don't define it explicitly.

MySQL create sequence example

The following example creates `employees` table whose `emp_no` column is `AUTO_INCREMENT` column:



```
1 CREATE TABLE employees(  
2   emp_no INT(4) AUTO_INCREMENT PRIMARY KEY,  
3   first_name VARCHAR(50),  
4   last_name VARCHAR(50)  
5 ) ENGINE = INNODB;
```

How MySQL sequence works

The `AUTO_INCREMENT` column has the following attributes:

- The starting value of an `AUTO_INCREMENT` column is 1 and it is increased by 1 when you insert `NULL` value into the column or when you omit its value in the `INSERT` statement.
- To obtain the last generated sequence number, you use the `LAST_INSERT_ID()` function. You often use the last insert ID for the subsequent statements e.g., insert data into child tables. The last generated sequence is unique across sessions. In other words, if another connection generates a sequence number, from your connection you can obtain it by using the `LAST_INSERT_ID()` function. For more details on `LAST_INSERT_ID()` function, check it out the MySQL `LAST_INSERT_ID()` function tutorial.
- If you insert a new row into a table and specify a value for the sequence column, MySQL will insert the sequence number if the sequence number does not exist in the column or issue an error if it already exists. If you insert a new value that is greater than the next sequence number, MySQL will use the new value as the starting sequence number and generate a

unique sequence number greater than the current one for the next use. This creates gaps in the sequence.

- If you use UPDATE statement to update an `AUTO_INCREMENT` column to a value that already exists, MySQL will issue a duplicate-key error if the column has a unique index. If you update an `AUTO_INCREMENT` column to a value that is larger than the existing values in the column, MySQL will use the next number of the last insert sequence number for the next row e.g., if the last insert sequence number is 3, you update it to 10, the sequence number for the new row is 4. See the example in the below section.
- If you use DELETE statement to delete the last insert row, MySQL may or may not reuse the deleted sequence number depending on the storage engine of the table. A MyISAM table does not reuse the deleted sequence numbers if you delete a row e.g., the last insert id in the table is 10, if you remove it, MySQL still generates the next sequence number which is 11 for the new row. Similar to MyISAM tables, InnoDB tables do use reuse sequence number when rows are deleted.

Once you set `AUTO_INCREMENT` attribute for a column, you can reset auto increment value in various ways e.g., by using `ALTER TABLE` statement.

Let's practice with the MySQL sequence.

First, insert two new employees into the `employees` table:

```
1 INSERT INTO employees(first_name,last_name)
2 VALUES('John','Doe'),
3 ('Mary','Jane');
```

Second, select data from the `employees` table:

```
1 SELECT * FROM employees;
```

	emp_no	first_name	last_name
	1	John	Doe
	2	Mary	Jane

Third, delete the second employee whose `emp_no` is 2:

```
1 DELETE FROM employees
2 WHERE emp_no = 2;
```

	emp_no	first_name	last_name
	1	John	Doe

Fourth, insert a new employee:

```
1 INSERT INTO employees(first_name,last_name)
2 VALUES('Jack','Lee');
```

emp_no	first_name	last_name
1	John	Doe
3	Jack	Lee

Because the storage engine of the `employees` table is InnoDB, it does not reuse the deleted sequence number. The new row has `emp_no` 3.

Fifth, update an existing employee with `emp_no` 3 to 1:

```
1 UPDATE employees
2 SET first_name = 'Joe',
3   emp_no = 1
4 WHERE emp_no = 3;
```

MySQL issued an error of duplicate entry for the primary key. Let's fix it:

```
1 UPDATE employees
2 SET first_name = 'Joe',
3   emp_no = 10
4 WHERE emp_no = 3;
```

emp_no	first_name	last_name
1	John	Doe
10	Joe	Lee

Sixth, insert a new employee after updating the sequence number to 10:

```
1 INSERT INTO employees(first_name,last_name)
2 VALUES('Wang','Lee');
```

emp_no	first_name	last_name
1	John	Doe
4	Wang	Lee
10	Joe	Lee

The next sequence number of the last insert is 4, therefore MySQL use 4 for the new row instead of 11.

MySQL Primary Key

Introduction to MySQL primary key

MySQL Primary Key



A primary key is a column or a set of columns that uniquely identifies each row in the table. The following are the rules that you must follow when you define a primary key for a table:

- A primary key must contain unique values. If the primary key consists of multiple columns, the combination of values in these columns must be unique.
- A primary key column cannot contain `NULL` values. It means that you have to declare the primary key column with `NOT NULL` attribute. If you don't, MySQL will force the primary key column as `NOT NULL` implicitly.
- A table has only one primary key.

Because MySQL works faster with integers, the primary key column's type should be an integer type e.g., `INT` or `BIGINT`. You can choose a smaller integer type such as `TINYINT`, `SMALLINT`, etc., however you should make sure that the range of values of the integer type for the primary key is sufficient for storing all possible rows that the table may have.

A primary key column often has `AUTO_INCREMENT` attribute that generates a unique sequence for the key automatically. The the primary key of the next row is greater than the previous one.

MySQL creates an index named `PRIMARY` with `PRIMARY` type for the primary key in a table.

Defining MySQL PRIMARY KEY Constraints

MySQL allows you to to create a primary key by defining a primary key constraint when you create or modify the table.

Defining MySQL PRIMARY KEY constraints using CREATE TABLE statement

MySQL allows you to create the primary key when you create the table by using the `CREATE TABLE` statement. To create a `PRIMARY KEY` constraint for the table, you specify the `PRIMARY KEY` in the primary key column's definition.

The following example creates `users` table whose primary key is `user_id` column:

```

1 CREATE TABLE users(
2   user_id INT AUTO_INCREMENT PRIMARY KEY,
3   username VARCHAR(40),
4   password VARCHAR(255),
5   email VARCHAR(255)
6 );

```

You can also specify the `PRIMARY KEY` at the end of the `CREATE TABLE` statement as follows:

```

1 CREATE TABLE roles(
2   role_id INT AUTO_INCREMENT,
3   role_name VARCHAR(50),
4   PRIMARY KEY(role_id)
5 );

```

In case the primary key consists of multiple columns, you must specify them at the end of the `CREATE TABLE` statement. You put a coma-separated list of primary key columns inside parentheses followed the `PRIMARY KEY` keywords.

```

1 CREATE TABLE userroles(
2   user_id INT NOT NULL,
3   role_id INT NOT NULL,
4   PRIMARY KEY(user_id,role_id),
5   FOREIGN KEY(user_id) REFERENCES users(user_id),
6   FOREIGN KEY(role_id) REFERENCES roles(role_id)
7 );

```

Besides creating the primary key that consists of `user_id` and `role_id` columns, the statement also created two foreign key constraints.

Defining MySQL PRIMARY KEY constraints using ALTER TABLE statement

If a table, for some reasons, does not have a primary key, you can use the `ALTER TABLE` statement to add a column that has all necessary primary key's characteristics to the primary key as the following statement:

```

1 ALTER TABLE table_name
2 ADD PRIMARY KEY(primary_key_column);

```

The following example adds the `id` column to the primary key.

First, create `t1` table without defining the primary key.

```

1 CREATE TABLE t1(
2   id int,
3   title varchar(255) NOT NULL
4 );

```

Second, add the `id` column to primary key of the `t1` table.

```

1 ALTER TABLE t1
2 ADD PRIMARY KEY(id);

```

PRIMARY KEY vs. UNIQUE KEY vs. KEY

A `KEY` is a synonym for `INDEX`. You use `KEY` when you want to create an index for a column or a set of column that is not a part of a primary key or unique key.

A `UNIQUE` index creates a constraint for a column whose values must be unique. Unlike the `PRIMARY` index, MySQL allows `NULL` values in the `UNIQUE` index. A table can also have multiple `UNIQUE` indexes.

For example, the `email` and `username` of user in the `users` table must be unique. You can define `UNIQUE` indexes for the `email` and `username` column as the following statement:

Add a `UNIQUE` index for the `username` column.

```

1 ALTER TABLE users
2 ADD UNIQUE INDEX username_unique (username ASC) ;

```

Add a `UNIQUE` index for the `email` column.

```

1 ALTER TABLE users
2 ADD UNIQUE INDEX email_unique (email ASC) ;

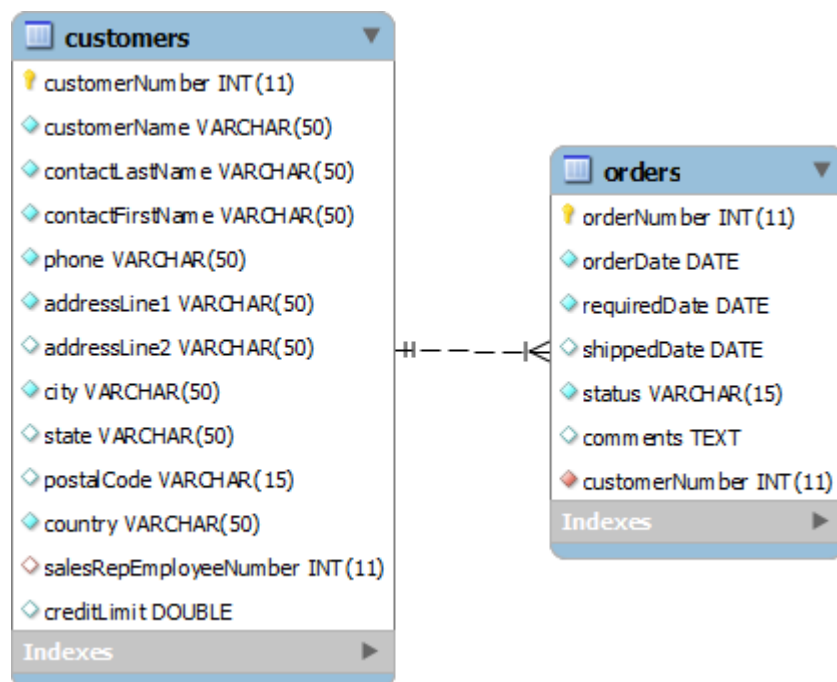
```

MySQL Foreign Key

Introduction to MySQL foreign key

A foreign key is a field in a table that matches a field of another table. A foreign key places constraints on data in the related tables that, which enables MySQL to maintain referential integrity.

Let's take a look at the following database diagram in the sample database.



We have two tables: `customers` and `orders`. Each customer has zero or more orders and each order belongs to only one customer. The relationship between `customers` table and `orders` table is one-to-many, and it is established by a foreign key in the `orders` table specified by the `customerNumber` field. The `customerNumber` field in the `orders` table relates to the `customerNumber` primary key field in `customers` table.

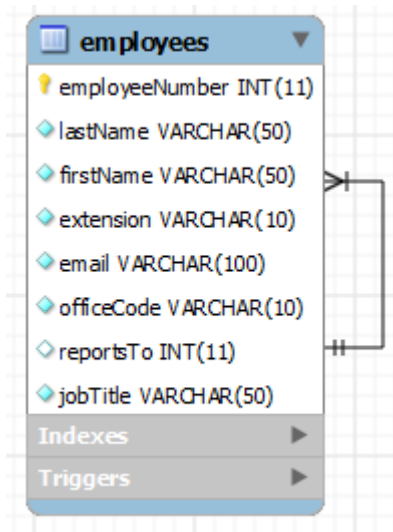
The `customers` table is called *parent table* or *referenced table*, and the `orders` table is known as *child table* or *referencing table*.

A foreign key has not only one column but also a set of columns. The columns in the child table often refer to the primary key columns in the parent table.

A table may have more than one foreign key, and each foreign key in the child table can have a different parent table.

A row in the child table must contain values that exist in the parent table e.g., each order record in the `orders` table must have a `customerNumber` that exists in the `customers` table. Multiple orders can refer to the same customer therefore this relationship is called one (customer) to many (orders), or one-to-many.

Sometimes, the child and parent table is the same table. The foreign key refers back to the primary key of the table e.g., the following `employees` table :



The `reportTo` column is a foreign key that refers to the `employeeNumber` column which is the primary key of the `employees` table to reflect the reporting structure between employees i.e., each employee reports to another employee and an employee can have zero or more direct reports.

The `reportTo` foreign key is also known as *recursive* or *self-referencing* foreign key.

Foreign keys enforce referential integrity that helps you maintain the consistency and integrity of the data automatically. For example, you cannot create an order for a non-existent customer.

In addition, you can set up a cascade on delete action for the `customerNumber` foreign key so that when you delete a customer in the `customers` table, all the orders associated with the customer are also deleted. This saves you time and efforts of using multiple DELETE statements or a DELETE JOIN statement.

The same as deletion, you can also define a cascade on update action for the `customerNumber` foreign key to perform cross-table update without using multiple UPDATE statements or an UPDATE JOIN statement.

In MySQL, the InnoDB storage engine supports foreign keys so that you must create InnoDB tables in order to use foreign key constraints.

MySQL create table foreign key

MySQL create foreign key syntax

The following syntax illustrates how to define a foreign key in a child table in CREATE TABLE statement.

```
1 CONSTRAINT constraint_name
```

2 FOREIGN KEY foreign_key_name (columns)

3 REFERENCES parent_table(columns)

4 ON DELETE action

5 ON UPDATE action

Let's examine the syntax in greater detail:

- The `CONSTRAINT` clause allows you to define constraint name for the foreign key constraint. If you omit it, MySQL will generate a name automatically.
- The `FOREIGN KEY` clause specifies the columns in the child table that refer to primary key columns in the parent table. You can put a foreign key name after `FOREIGN KEY` clause or leave it to let MySQL to create a name for you. Notice that MySQL automatically creates an index with the `foreign_key_name` name.
- The `REFERENCES` clause specifies the parent table and its columns to which the columns in the child table refer. The number of columns in child table and parent table specified in the `FOREIGN KEY` and `REFERENCES` must be the same.
- The `ON DELETE` clause allows you to define what happens to the records in the child table when the records in the parent table are deleted. If you omit the `ON DELETE` clause and delete a record in the parent table that has records in the child table refer to, MySQL will reject the deletion. In addition, MySQL also provides you with actions so that you can have other options such as `ON DELETE CASCADE` that lets MySQL to delete records in the child table that refer to a record in the parent table when the record in the parent table is deleted. If you don't want the related records in the child table to be deleted, you use the `ON DELETE SET NULL` action instead. MySQL will set the foreign key column values in the child table to `NULL` when the record in the parent table is deleted, with a condition that the foreign key column in the child table must accept `NULL` values. Notice that if you use `ON DELETE NO ACTION` or `ON DELETE RESTRICT` action, MySQL will reject the deletion.
- The `ON UPDATE` clause enables you to specify what happens to the rows in the child table when rows in the parent table are updated. You can omit the `ON UPDATE` clause to let MySQL to reject any update to the rows in the child table when the rows in the parent table are updated. The `ON UPDATE CASCADE` action allows you to perform cross-table update, and the `ON UPDATE SET NULL` action resets the values in the rows in the child table to `NULL` values when the rows in the parent table are updated. The `ON UPDATE NO ACTION` or `ON UPDATE RESTRICT` actions reject any updates.

MySQL create table foreign key example

The following example creates a `dbdemo` database and two tables: `categories` and `products`. Each category has one or more products and each product belongs to only one category. The `cat_id` field in the `products` table is defined as a foreign key with `UPDATE ON CASCADE` and `DELETE ON RESTRICT` actions.

1 CREATE DATABASE IF NOT EXISTS dbdemo;

2

3 USE dbdemo;

```

4
5 CREATE TABLE categories(
6   cat_id int not null auto_increment primary key,
7   cat_name varchar(255) not null,
8   cat_description text
9 ) ENGINE=InnoDB;
10
11 CREATE TABLE products(
12   prd_id int not null auto_increment primary key,
13   prd_name varchar(355) not null,
14   prd_price decimal,
15   cat_id int not null,
16   FOREIGN KEY fk_cat(cat_id)
17   REFERENCES categories(cat_id)
18   ON UPDATE CASCADE
19   ON DELETE RESTRICT
20 )ENGINE=InnoDB;

```

MySQL add foreign key

MySQL add foreign key syntax

To add a foreign key to an existing table, you use the ALTER TABLE statement with the foreign key definition syntax above:

```

1 ALTER table_name
2 ADD CONSTRAINT constraint_name
3 FOREIGN KEY foreign_key_name(columns)
4 REFERENCES parent_table(columns)
5 ON DELETE action
6 ON UPDATE action

```

MySQL add foreign key example

Now, let's add a new table named `vendors` and change the `products` table to include the vendor id field:

```
1 USE dbdemo;
2
3 CREATE TABLE vendors(
4   vdr_id int not null auto_increment primary key,
5   vdr_name varchar(255)
6 )ENGINE=InnoDB;
7
8 ALTER TABLE products
9 ADD COLUMN vdr_id int not null AFTER cat_id;
```

To add a foreign key to the `products` table, you use the following statement:

```
1 ALTER TABLE products
2 ADD FOREIGN KEY fk_vendor(vdr_id)
3 REFERENCES vendors(vdr_id)
4 ON DELETE NO ACTION
5 ON UPDATE CASCADE;
```

Now, the `products` table has two foreign keys, one refers to the `categories` table and another refers to the `vendors` table.

MySQL drop foreign key

You also use the `ALTER TABLE` statement to drop foreign key as the following statement:

```
1 ALTER TABLE table_name
2 DROP FOREIGN KEY constraint_name
```

In the statement above:

- First, you specify the table name from which you want to remove the foreign key.
- Second, you put the constraint name after the `DROP FOREIGN KEY` clause.

Notice that `constraint_name` is the name of the constraint specified when you created or added the foreign key to the table. If you omit it, MySQL generates a constraint name for you.

To obtain the generated constraint name of a table, you use the `SHOW CREATE TABLE` statement as follows:

```
1 SHOW CREATE TABLE table_name
```

For example, to see the foreign keys of the `products` table, you use the following statement:

```
1 SHOW CREATE TABLE products
```

The following is the output of the statement:

```
1 CREATE TABLE products (
2   prd_id int(11) NOT NULL AUTO_INCREMENT,
3   prd_name varchar(355) NOT NULL,
4   prd_price decimal(10,0) DEFAULT NULL,
5   cat_id int(11) NOT NULL,
6   vdr_id int(11) NOT NULL,
7   PRIMARY KEY (prd_id),
8   KEY fk_cat (cat_id),
9   KEY fk_vendor(vdr_id),
10
11  CONSTRAINT products_ibfk_2
12  FOREIGN KEY (vdr_id)
13  REFERENCES vendors (vdr_id)
14  ON DELETE NO ACTION
15  ON UPDATE CASCADE,
16
17  CONSTRAINT products_ibfk_1
18  FOREIGN KEY (cat_id)
19  REFERENCES categories (cat_id)
```

20 ON UPDATE CASCADE

21) ENGINE=InnoDB;

The `products` table has two foreign key constraints: `products_ibfk_1` and `products_ibfk_2`.

You can drop the foreign keys of the `products` table by using the following statement:

1 ALTER TABLE `products`

2 DROP FOREIGN KEY `products_ibfk_1`;

3

4 ALTER TABLE `products`

5 DROP FOREIGN KEY `products_ibfk_2`;

MySQL disable foreign key checks

Sometimes, it is very useful to disable foreign key checks e.g., when you load data into the tables that have foreign keys. If you don't disable foreign key checks, you have to load data into a proper order i.e., you have to load data into parent tables first and then child tables, which can be tedious. However if you disable the foreign key checks, you can load data into any orders.

Another example is that, unless you disable the foreign key checks, you cannot drop a table that is referenced by a foreign key constraint. When you drop a table, any constraints that you defined for the table are also removed.

To disable foreign key checks, you use the following statement:

1 SET `foreign_key_checks` = 0

And of course, you can enable it by using the statement below:

1 SET `foreign_key_checks` = 1

Changing Table Structure Using MySQL ALTER TABLE

MySQL ALTER TABLE syntax

The `ALTER TABLE` statement is used to change the structure of existing tables. You can use the `ALTER TABLE` statement to add or drop column, change data type of column, add primary key, rename table and many more. The following illustrates the `ALTER TABLE` statement syntax:

```
1 ALTER TABLE table_name action1[,action2,...]
```

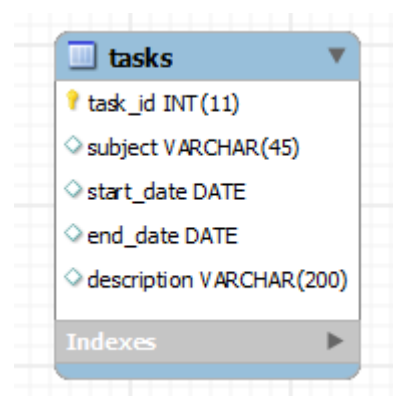
To alter an existing table:

- First, you specify the table name that you want to change after the `ALTER TABLE` keywords.
- Second, you list a set of actions that you want to apply to the table. An action can be anything such as adding a new column, adding primary key, renaming table, etc. The `ALTER TABLE` statement allows you to apply multiple actions in a single `ALTER TABLE` statement, each action is separated by a comma (,).

Let's create a new table for practicing the `ALTER TABLE` statement.

We're going to create a new table named `tasks` in our sample database. The following is the script for creating the `tasks` table.

```
1 CREATE TABLE tasks (
2   task_id INT NOT NULL ,
3   subject VARCHAR(45) NULL ,
4   start_date DATE NULL ,
5   end_date DATE NULL ,
6   description VARCHAR(200) NULL ,
7   PRIMARY KEY (task_id) ,
8   UNIQUE INDEX task_id_UNIQUE (task_id ASC) );
```



Changing columns using MySQL ALTER TABLE statement

Using MySQL ALTER TABLE statement to set auto-increment attribute for a column

Suppose you want the value of the `task_id` column to be increased automatically by one whenever you insert a new record into the `tasks` table. To do this, you use the `ALTER TABLE` statement to set the attribute of the `task_id` column to `AUTO_INCREMENT` as follows:

```
1 ALTER TABLE tasks
2 CHANGE COLUMN task_id task_id INT(11) NOT NULL AUTO_INCREMENT;
You can verify the change by adding some records to the tasks table.
```

```
1 INSERT INTO tasks(subject,
2     start_date,
3     end_date,
4     description)
5 VALUES('Learn MySQL ALTER TABLE',
6     Now(),
7     Now(),
8     'Practicing MySQL ALTER TABLE statement');
9
10 INSERT INTO tasks(subject,
11     start_date,
12     end_date,
13     description)
14 VALUES('Learn MySQL CREATE TABLE',
15     Now(),
16     Now(),
17     'Practicing MySQL CREATE TABLE statement');
```

And you can query data to see if the value of the `task_id` column is increased by 1 each time you insert a new record:

```
1 SELECT task_id, description
2 FROM tasks
```

	task_id	description
▶	1	Practicing MySQL ALTER TABLE statement
	2	Practicing MySQL CREATE TABLE statement

Using MySQL ALTER TABLE statement to add a new column into a table

Because of the new business requirement, you need to add a new column called `complete` to store the percentage of completion for each task in the `tasks` table. In this case, you can use the `ALTER TABLE` to add a new column to the `tasks` table as follows:

```
1 ALTER TABLE tasks
2 ADD COLUMN complete DECIMAL(2,1) NULL
3 AFTER description;
```


Using MySQL ALTER TABLE to drop a column from a table

Suppose you don't want to store the description of tasks in the `tasks` table and you have to remove it. The following statement allows you to remove the `description` column of the `tasks` table:

```
1 ALTER TABLE tasks
2 DROP COLUMN description;
```

Renaming table using MySQL ALTER TABLE statement

You can use the `ALTER TABLE` statement to rename a table. Notice that before renaming a table, you should take a serious consideration to see if the change affects both database and application layers.

The following statement rename the `tasks` table to `work_items`:

```
1 ALTER TABLE tasks
2 RENAME TO work_items;
```

MySQL DROP TABLE – Removing Existing Tables

MySQL DROP TABLE statement syntax

In order to remove existing tables, you use the MySQL `DROP TABLE` statement. The syntax of the `DROP TABLE` is as follows:

```
1 DROP [TEMPORARY] TABLE [IF EXISTS] table_name [, table_name] ...
2 [RESTRICT | CASCADE]
```

The `DROP TABLE` statement removes a table and its data permanently from the database. In MySQL, you can also remove multiple tables using a single `DROP TABLE` statement, each table is separated by a comma (,).

The `TEMPORARY` flag allows you to remove temporary tables only. It is very convenient to ensure that you do not accidentally remove non-temporary tables.

The `IF EXISTS` addition allows you to hide the error message in case one or more tables in the list do not exist. When you use `IF EXISTS` addition, MySQL generates a NOTE, which can be retrieved by using the `SHOW WARNING` statement. It is important to notice that the `DROP TABLE` statement removes all existing tables and issues an error message or a NOTE when you have a non-existent table in the list.

As mentioned above, the `DROP TABLE` statement only removes table and its data. However, it does not remove specific user privileges associated with the table. Therefore if a table with the same name is re-created after that, the existing privileges will apply to the new table, which may pose a security risk.

The `RESTRICT` and `CASCADE` flags are reserved for the future versions of MySQL.

Last but not least, you must have `DROP` privileges for the table that you want to remove.

MySQL DROP TABLE example

We are going to remove the `tasks` table that we created in the previous tutorial on creating tables using `CREATE TABLE` statement. In addition, we also remove a non-existent table to practice with the `SHOW WARNING` statement. The statement to remove the `tasks` table and a non-existent table is as follows:

```
1 DROP TABLE IF EXISTS tasks, nonexistent_table;
```

If you check the database, you will see that the `tasks` table was removed. You can check the `NOTE`, which is generated by MySQL because of non-existent table, by using the `SHOW WARNING` statement as follows:

```
1 SHOW WARNINGS;
```

	Level	Code	Message
►	Note	1051	Unknown table 'nonexistent_table'

MySQL DROP TABLE LIKE

Imagine you have a lot of tables whose names start with `test` in your database and you want to save time by removing all of them using a single `DROP TABLE` statement. Unfortunately, MySQL does not provide the `DROP TABLE LIKE` statement that can remove tables based on pattern matching like the following:

```
1 DROP TABLE LIKE '%pattern%'
```

However, there are some workarounds. We will discuss one of them here for your reference.

Let's start creating `test*` tables for the sake of demonstration.

```
1 CREATE TABLE IF NOT EXISTS test1(
2   id int(11) NOT NULL AUTO_INCREMENT,
3   PRIMARY KEY(id)
4 );
```

5

```
6 CREATE TABLE IF NOT EXISTS test2 LIKE test1;
```

```
7 CREATE TABLE IF NOT EXISTS test3 LIKE test1;
```

```
8 CREATE TABLE IF NOT EXISTS test4 LIKE test1;
```

We've created four tables named `test1`, `test2`, `test3` and `test4` with the same table structure.

Suppose you want to remove all `test*` tables at a time, you can follow the steps below:

First, you declare two variables that accept database schema and a pattern that you want to the tables to match:

```
1 -- set table schema and pattern matching for tables
```

```
2 SET @schema = 'classicmodels';
```

```
3 SET @pattern = 'test%';
```

Next, you need to build a dynamic `DROP TABLE` statement:

```
1 -- build dynamic sql (DROP TABLE tbl1, tbl2...;)
```

```
2 SELECT CONCAT('DROP TABLE ',GROUP_CONCAT(CONCAT(@schema,'.',table_name)),';')
```

```
3 INTO @droplike
```

```
4 FROM information_schema.tables
```

```
5 WHERE @schema = database()
```

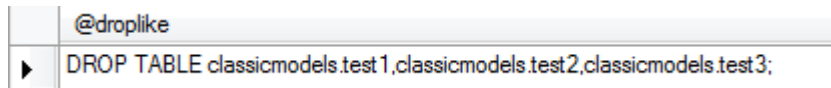
```
6 AND table_name LIKE @pattern;
```

Basically, the query instructs MySQL to go to the `information_schema` table, which contains data on all tables in all databases, and to concatenate all tables in the database `@schema` (`classicmodels`) that matches the pattern `@pattern` (`test%`) with the prefix `DROP TABLE`. The `GROUP_CONCAT` function creates a comma-separated list of tables.

Then, we can display the dynamic SQL to verify if it works correctly:

```
1 -- display the dynamic sql statement
```

```
2 SELECT @droplike;
```



You can see that it works as expected.

After that, you can execute the statement using prepared statement in MySQL as follows:

```
1 -- execute dynamic sql
2 PREPARE stmt FROM @dropcmd;
3 EXECUTE stmt;
4 DEALLOCATE PREPARE stmt;
```

Putting it all together.

```
1 -- set table schema and pattern matching for tables
2 SET @schema = 'classicmodels';
3 SET @pattern = 'test%';
4
5 -- build dynamic sql (DROP TABLE tbl1, tbl2...;)
6 SELECT CONCAT('DROP TABLE ',GROUP_CONCAT(CONCAT(@schema,'.',table_name)),';')
7 INTO @droplike
8 FROM information_schema.tables
9 WHERE @schema = database()
10 AND table_name LIKE @pattern;
11
12 -- display the dynamic sql statement
13 SELECT @droplike;
14
15 -- execute dynamic sql
16 PREPARE stmt FROM @dropcmd;
17 EXECUTE stmt;
18 DEALLOCATE PREPARE stmt;
```

So if you want to drop multiple tables that have a specific pattern in a database, you just use the script above to save time. All you need to do is replacing the *pattern* and the *database schema* in @pattern and @schema variables. If you often have to deal with this task, you can always develop a stored procedure based on the script and reuse the stored procedure in the future.

MySQL Temporary Table

Introduction to MySQL temporary table

In MySQL, a temporary table is a special type of table that allows you to store a temporary result set, which you can reuse several times in a single session. A temporary table is very handy when it is impossible or expensive to query data that requires a single SELECT

statement with JOIN clauses. You often use temporary tables in stored procedures to store immediate result sets for the subsequent uses.

MySQL temporary tables have some additional features:

- A temporary table is created by using `CREATE TEMPORARY TABLE` statement. Notice that the `TEMPORARY` keyword is added between `CREATE` and `TABLE` keywords.
- MySQL drops the temporary table automatically when the session ends or connection is terminated. Of course, you can use the `DROP TABLE` statement to drop a temporary table explicitly when you are no longer use it.
- A temporary table is only available and accessible by the client who creates the table.
- Different clients can create a temporary table with the same name without causing errors because only the client who creates a temporary table can see it. However, in the same session, two temporary tables cannot have the same name.
- A temporary table can have the same name as an existing table in a database. For example, if you create a temporary table named `employees` in the sample database, the existing `employees` table becomes inaccessible. Every query you issue against the `employees` table refers to the `employees` temporary table. When you remove the `employees` temporary table, the permanent `employees` table is available and accessible again. Though this is allowed however it is not recommended to create a temporary table whose name is same as a name of a permanent table because it may lead to a confusion. For example, in case the connection to the MySQL database server is lost and you reconnect to the server automatically, you cannot differentiate between the temporary table and the permanent table. In the worst case, you may issue a `DROP TABLE` statement to remove the permanent table instead of the temporary table, which is not expected.

Create MySQL temporary table

Like the `CREATE TABLE` statement, MySQL provides many options to create a temporary table. To create a temporary table, you just add the `TEMPORARY` keyword to the `CREATE TABLE` statement.

For example, the following statement creates a top 10 customers by revenue temporary table based on the result set of a `SELECT` statement:

```
1 CREATE TEMPORARY TABLE top10customers
2 SELECT p.customerNumber,
3      c.customerName,
4      FORMAT(SUM(p.amount),2) total
5 FROM payments p
6 INNER JOIN customers c ON c.customerNumber = p.customerNumber
7 GROUP BY p.customerNumber
8 ORDER BY total DESC
```

9 LIMIT 10

Now, you can query data from the `top10customers` temporary table as from a permanent table:

```
1 SELECT * FROM top10customers
```

	customerNumber	customerName	total
▶	157	Diecast Classics Inc.	98,509.25
	167	Herkku Gifts	97,562.47
	311	Oulu Toy Supplies, Inc.	95,706.15
	186	Toys of Finland, Co.	95,546.46
	175	Gift Depot Inc.	95,424.63
	205	Toys4GrownUps.com	93,803.30
	282	Souvenirs And Things Co.	91,655.61
	286	Marta's Replicas Co.	90,545.37
	386	L'ordine Souvenirs	90,143.31
	227	Heintze Collectables	89,909.80

Drop MySQL temporary table

You can use the `DROP TABLE` statement to remove temporary tables however it is good practice to use the `DROP TEMPORARY TABLE` statement instead. Because the `DROP TEMPORARY TABLE` removes only temporary tables, not the permanent tables. In addition, the `DROP TEMPORARY TABLE` statement helps you avoid the mistake of removing a permanent table when you name your temporary table the same as the name of the permanent table.

For example, to remove the `top10customers` temporary table, you use the following statement:

```
1 DROP TEMPORARY TABLE top10customers
```

Notice that if you try to remove a permanent table with the `DROP TEMPORARY TABLE` statement, you will get an error message saying that the table you are trying drop is unknown.

Note if you develop an application that uses a connection pooling or persistent connections, it is not guaranteed that the temporary tables are removed automatically when your application is terminated. Because the database connection that the application used may be still open and is placed in a connection pool for other clients to reuse it. This means you should always remove the temporary tables that you created whenever you are done with them.

MySQL Managing Database Index

Database index, or just index, helps speed up the retrieval of data from tables. When you query data from a table, first MySQL checks if the indexes exist, then MySQL uses the indexes to select exact physical corresponding rows of the table instead of scanning the whole table.

A database index is similar to an index of a book. If you want to find a topic, you look up in the index first, and then you open the page that has the topic without scanning the whole book.

It is highly recommended that you should create index on columns of table from which you often query the data. Notice that all primary key columns are in the primary index of the table automatically.

If index helps speed up the querying data, why don't we use indexes for all columns? If you create an index for every column, MySQL has to build and maintain the index table. Whenever a change is made to the records of the table, MySQL has to rebuild the index, which takes time as well as decreases the performance of the database server.

Creating MySQL Index

You often create indexes when you create tables. MySQL automatically add any column that is declared as `PRIMARY KEY`, `KEY`, `UNIQUE` or `INDEX` to the index. In addition, you can add indexes to the tables that already have data.

In order to create indexes, you use the `CREATE INDEX` statement. The following illustrates the syntax of the `CREATE INDEX` statement:

```
1 CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
2 USING [BTREE | HASH | RTREE]
3 ON table_name (column_name [(length)] [ASC | DESC],...)
```

First, you specify the index based on the table type or storage engine:

- `UNIQUE` means MySQL will create a constraint that all values in the index must be unique. Duplicate `NULL` value is allowed in all storage engine except `BDB`.
- `FULLTEXT` index is supported only by `MyISAM` storage engine and only accepted on column that has data type is `CHAR`, `VARCHAR` or `TEXT`.
- `SPATIAL` index supports spatial column and is available on `MyISAM` storage engine. In addition, the column value must not be `NULL`.

Then, you name the index and its type after the `USING` keyword such as `BTREE`, `HASH` or `RTREE` also based on the storage engine of the table.

Here are the storage engines of the table with the corresponding allowed index types:

Storage Engine	Allowable Index Types
MyISAM	BTREE, RTREE
InnoDB	BTREE
MEMORY/HEAP	HASH, BTREE
NDB	HASH

Third, you declare table name and a list columns that you want to add to the index.

Example of creating index in MySQL

In the sample database, you can add `officeCode` column of the `employees` table to the index by using the `CREATE INDEX` statement as follows:

```
1 CREATE INDEX officeCode ON employees(officeCode)
```

Removing Indexes

Besides creating index, you can also remove index by using the `DROP INDEX` statement.

Interestingly, the `DROP INDEX` statement is also mapped to `ALTER TABLE` statement. The following is the syntax of removing the index:

```
1 DROP INDEX index_name ON table_name
```

For example, if you want to drop index `officeCode` of the `employees` table, which we have created above, you can execute following query:

```
1 DROP INDEX officeCode ON employees
```

Using MySQL SELECT Statement to Query Data

The MySQL `SELECT` statement allows you to retrieve zero or more rows from tables or views. The `SELECT` statement is the one of the most commonly used queries in MySQL.

The `SELECT` statement returns a result that is a combination of columns and rows, which is also known as a result set.

MySQL SELECT syntax

The following illustrates the syntax of the `SELECT` statement:

```
1 SELECT column_1,column_2...
```


- 2 FROM table_1
- 3 [INNER | LEFT | RIGHT] JOIN table_2 ON conditions
- 4 WHERE conditions
- 5 GROUP BY group
- 6 HAVING group_conditions
- 7 ORDER BY column_1 [ASC | DESC]
- 8 LIMIT offset, row_count

The SELECT statement is composed of several clauses:

- SELECT chooses which columns of the table you want to get the data.
- FROM specifies the table from which you get the data.
- JOIN gets data from multiple table based on certain join conditions.
- WHERE filters rows to select.
- GROUP BY group rows to apply aggregate functions on each group.
- HAVING filters group based on groups defined by GROUP BY clause.
- ORDER BY specifies the order of the returned result set.
- LIMIT constrains number of returned rows.

You will learn about each clause in more detail in the next tutorial. In this tutorial, we are going to focus on the simple form of the SELECT statement.

MySQL SELECT Examples

To select all columns and rows from the `employees` table, you use the following query:

```
1 SELECT * FROM employees
```

	employeeNumber	lastName	firstName	extension	email	officeCode	reportsTo
▶	1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	1	NULL
	1056	Phan	Mary	x4611	mpatterso@classicmodelcars.com	1	1002
	1076	Firelli	Jeff	x9273	jfirelli@classicmodelcars.com	1	1002
	1088	Patterson	William	x4871	wpatterson@classicmodelcars.com	6	1056
	1102	Bondur	Gerard	x5408	gbondur@classicmodelcars.com	4	1056
	1143	Bow	Anthony	x5428	abow@classicmodelcars.com	1	1056
	1165	Jennings	Leslie	x3291	ljennings@classicmodelcars.com	1	1143
	1166	Thompson	Leslie	x4065	lthompson@classicmodelcars.com	1	1143
	1188	Firelli	Julie	x2173	jfirelli@classicmodelcars.com	2	1143

The asterisk (*) notation is a shorthand of selecting all columns in the table.

The `SELECT` statement also allows you to query partial data of a table by specifying a list of comma-separated columns in the `SELECT` clause. For example, if you want to view only `first name`, `last name` and `job title` of the employees, you use the following query:

```
1 SELECT lastname,
2    firstname,
3    jobtitle
```

4 FROM employees

	lastname	firstname	jobtitle
▶	Murphy	Diane	President
	Phan	Mary	VP Sales
	Firelli	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
	Firelli	Julie	Sales Rep
	Patterson	Steve	Sales Rep

Filter Rows Using MySQL WHERE

If you use the [SELECT statement](#) to query the data from a table without the `WHERE` clause, you will get all rows in the table, which sometimes brings more data than you need. The `WHERE` clause allows you to specify exact rows to select based on given conditions e.g., find all customers in the U.S.

The following query selects all customers whose country is U.S. from the `customers` table. We use the `WHERE` clause to filter the customers. In the `WHERE` clause, we compare the values of the `country` column with the `USA` literal string.

```
1 SELECT customerName, city
2 FROM customers
3 WHERE country = 'USA';
```

	customerName	city
▶	Signal Gift Stores	Las Vegas
	Mini Gifts Distributors Ltd.	San Rafael
	Mini Wheels Co.	San Francisco
	Land of Toys Inc.	NYC
	Muscle Machine Inc	NYC
	Diecast Classics Inc.	Allentown
	Technics Stores Inc.	Burlingame

You can form a simple condition like the query above, or a very complex one that combines multiple expressions with logical operators such as `AND` and `OR`. For example, to find all customers in the U.S . and also in the New York city, you use the following query:

```
1 SELECT customerName, city
```

```
2 FROM customers
3 WHERE country = 'USA' AND
4     city = 'NYC';
```

customerName	city
Land of Toys Inc.	NYC
Muscle Machine Inc	NYC
Vitachrome Inc.	NYC
Classic Legends Inc.	NYC
Microscale Inc.	NYC

You can test the condition for not only equality but also inequality. For example, to find all customers whose credit limit is greater than 200.000 USD, you use the following query:

```
1 SELECT customerName, creditlimit
2 FROM customers
3 WHERE creditlimit > 200000;
```

customerName	creditlimit
Mini Gifts Distributors Ltd.	210500
Euro+ Shopping Channel	227600

There are several useful operators that you can use in the **WHERE** clause to form more practical queries such as:

- [BETWEEN](#) selects values within a range of values.
- [LIKE](#) matches value based on pattern matching.
- [IN](#) specifies if the value matches any value in a list.
- **IS NULL** checks if the value is **NULL**

The **WHERE** clause is used not only with the **SELECT** statement but also other SQL statements to filter rows such as [DELETE](#) and [UPDATE](#).

MySQL ORDER BY

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 a specific order. 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:

```
1 SELECT col1, col2,...
2 FROM tbl
3 ORDER BY col1 [ASC|DESC], col2 [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

The following query selects contacts from the `customers` table and sorts the contacts by last name in ascending order.

```
1 SELECT contactLastname,
2     contactFirstname
3 FROM customers
4 ORDER BY contactLastname;
```

	contactLastname	contactFirstname
►	Accorti	Paolo
	Altagar,G M	Raanan
	Andersen	Mel
	Anton	Camen
	Ashworth	Rachel
	Barajas	Miguel
	Benitez	Violeta
	Bennett	Helen

If you want to sort the contact by last name in descending order, you specify the `DESC` after the `contactLastname` column in the `ORDER BY` clause as the following query:

```
1 SELECT contactLastname,
2     contactFirstname
3 FROM customers
4 ORDER BY contactLastname DESC
```

	contactLastname	contactFirstname
▶	Young	Jeff
	Young	Julie
	Young	Mary
	Young	Dorothy
	Yoshido	Juri
	Walker	Brydey
	Victorino	Wendy
	Urs	Braun

If you want to sort the contacts by last name in descending order and first name in ascending order, you specify both `DESC` and `ASC` in the corresponding column as follows:

```

1 SELECT contactLastname,
2     contactFirstname
3 FROM customers
4 ORDER BY contactLastname DESC,
5     contactFirstname ASC;

```

	contactLastname	contactFirstname
▶	Young	Dorothy
	Young	Jeff
	Young	Julie
	Young	Mary
	Yoshido	Juri
	Walker	Brydey
	Victorino	Wendy
	Urs	Braun

In the query above, the `ORDER BY` clause sorts the result set by last name in descending order first, and then sorts the sorted result set by 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. 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 and subtotal.

```

1 SELECT ordernumber,
2     quantityOrdered * priceEach
3 FROM orderdetails
4 ORDER BY ordernumber,
5     quantityOrdered * priceEach

```

	ordernumber	quantityOrdered * priceEach
▶	10100	1660.12
	10100	1729.21
	10100	2754.5
	10100	4080
	10101	1463.8500000000001
	10101	2040.1000000000001
	10101	2701.5
	10101	4343.56

To make the result more readable, you can use a column alias, and sort the result based on the column alias.

```

1 SELECT orderNumber,
2     quantityOrdered * priceEach AS subTotal
3 FROM orderdetails
4 ORDER BY orderNumber,
5     subTotal;
```

	orderNumber	subTotal
▶	10100	1660.12
	10100	1729.21
	10100	2754.5
	10100	4080
	10101	1463.8500000000001
	10101	2040.1000000000001
	10101	2701.5
	10101	4343.56

In the query above, we used `subtotal` as the column alias for the `quantityOrdered * priceEach` expression and sorted the result set based on the `subtotal` alias.

If you use a function that returns a value whose data type is different from the column's and sort the result based on the alias, the `ORDER BY` clause will sort the result set based on the return type of the function, which may not work as expected.

For example, if you use the `DATE_FORMAT` function to format the date values and sort the result set based on the strings returned by the `DATE_FORMAT` function, the order is not always correct. For more information, check it out the example in the `DATE_FORMAT` function tutorial.

MySQL ORDER BY with customer 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. For example, if you want to sort the `orders` based on the following status by the following order:

1. In Process
2. On Hold
3. Cancelled
4. Resolved
5. Disputed
6. 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:

```
1 SELECT orderNumber, status
2 FROM orders
3 ORDER BY FIELD(status, 'In Process',
4                'On Hold',
5                'Cancelled',
6                'Resolved',
7                'Disputed',
8                'Shipped');
```

orderNumber	status
10425	In Process
10424	In Process
10423	In Process
10422	In Process
10421	In Process
10420	In Process
10401	On Hold
10407	On Hold

How to Use MySQL DISTINCT to Eliminate Duplicate Rows

When querying data from a table, you may get duplicate rows. In order to remove the duplicate rows, you use the `DISTINCT` operator in the `SELECT` statement. The syntax of using the `DISTINCT` operator is as follows:

```
1 SELECT DISTINCT columns
2 FROM table_name
3 WHERE where_conditions
```

Let's take a look a simple example of using the `DISTINCT` operator to select the distinct last names of employees from the `employees` table.

First, we query the last names of employees from the `employees` table using the `SELECT` statement as follows:

- 1 SELECT lastname
- 2 FROM employees
- 3 ORDER BY lastname

lastname
Bondur
Bondur
Bott
Bow
Castillo
Firrelli
Firrelli
Fixter
Gerard
Hernandez
Jennings

Some employees has the same last name Bondur, Firrelli, etc. To remove the duplicate last names, you use the `DISTINCT` operator in the `SELECT` clause as follows:

- 1 SELECT DISTINCT lastname
- 2 FROM employees
- 3 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 use the `DISTINCT` operator.

MySQL DISTINCT and NULL values

If a column has `NULL` values and you use the `DISTINCT` operator for that column, MySQL will keep one `NULL` value and eliminate the other because the `DISTINCT` operator treats all `NULL` values as the same value.

For example, in the `customers` table, we have many rows with `state` column has `NULL` values. When we use the `DISTINCT` operator to query states of customers, we will see distinct states plus a `NULL` value as the following query:

```
1 SELECT DISTINCT state
2 FROM customers
```

state
NULL
NV
Victoria
CA
NY
PA
CT
MA
Osaka

MySQL DISTINCT with multiple columns

You can use the `DISTINCT` operator with more than one column. The combination of all columns will be used to define 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:

```
1 SELECT DISTINCT state, city
2 FROM customers
3 WHERE state IS NOT NULL
4 ORDER BY state, city
```

state	city
BC	Tsawassen
BC	Vancouver
CA	Brisbane
CA	Burbank
CA	Burlingame
CA	Glendale
CA	Los Angeles
CA	Pasadena

Without the `DISTINCT` operator, you will get duplicate combination state and city as follows:

```
1 SELECT state, city
2 FROM customers
3 WHERE state IS NOT NULL
4 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

DISTINCT vs. GROUP BY Clause

If you use the `GROUP BY` clause in the `SELECT` statement without using aggregate functions, the `GROUP BY` clause will behave like the `DISTINCT` operator. The following queries produce the same result set:

```
1 SELECT DISTINCT state
2 FROM customers;
3
4 SELECT state
5 FROM customers
6 GROUP BY state;
```

The difference between `DISTINCT` operator and `GROUP BY` clause is that the `GROUP BY` clause sorts the result set whereas the `DISTINCT` operator does not.

MySQL DISTINCT and COUNT aggregate function

The `DISTINCT` operator is used with the `COUNT` function to count unique records in a table. In this case, it ignores the `NULL` values. For example, to count the unique states of customers in the U.S., you use the following query:

```
1 SELECT COUNT(DISTINCT state)
2 FROM customers
3 WHERE country = 'USA';
```

COUNT(DISTINCT state)
8

Using MySQL LIMIT

MySQL LIMIT syntax

The `LIMIT` clause is used in the `SELECT` statement to constrain the number of rows in a result set. The `LIMIT` clause accepts one or two arguments. The values of both arguments must be zero or positive integer constants.

The following illustrates the `LIMIT` clause syntax with 2 arguments:

```
1 SELECT * FROM tbl
2 LIMIT offset, count
```

Let's see what the `offset` and `count` mean in the `LIMIT` clause:

- The `offset` specifies the offset of the first row to return. The `offset` of the first row is 0, not 1.
- The `count` specifies maximum number of rows to return.

When you use `LIMIT` with one argument, this argument will be used to specifies the maximum number of rows to return from the beginning of the result set.

```
1 SELECT * FROM tbl
2 LIMIT count
```

The query above is equivalent to the following query with the `LIMIT` clause that accepts two arguments:

```
1 SELECT * FROM tbl
2 LIMIT 0, count
```

Using MySQL LIMIT to get the first N rows

You can use the `LIMIT` clause to select the first `N` rows in a table as follows:

```
1 SELECT * FROM tbl
2 LIMIT N
```

For example, to select the first 10 customers, you use the following query:

```
1 SELECT customernumber,
2     customername,
3     creditlimit
4 FROM customers
5 LIMIT 10;
```

	customernumber	customername	creditlimit
►	103	Atelier graphique	21000
	112	Signal Gift Stores	71800
	114	Australian Collectors, Co.	117300
	119	La Rochelle Gifts	118200
	121	Baane Mini Imports	81700
	124	Mini Gifts Distributors Ltd.	210500
	125	Havel & Zbyszek Co	0
	128	Blauer See Auto, Co.	59700
	129	Mini Wheels Co.	64600
	131	Land of Toys Inc.	114900

Using MySQL LIMIT to get the highest and lowest values

The `LIMIT` clause often used with `ORDER BY` clause. First, you use the `ORDER BY` clause to sort the result set based on a certain criteria, and then you use `LIMIT` clause to find lowest or highest values.

For example, to select 5 customers who have the highest credit limit, you use the following query:

```

1 SELECT customernumber,
2     customername,
3     creditlimit
4 FROM customers
5 ORDER BY creditlimit DESC
6 LIMIT 5;
```

	customernumber	customername	creditlimit
►	141	Euro+ Shopping Channel	227600
	124	Mini Gifts Distributors Ltd.	210500
	298	Vida Sport, Ltd	141300
	151	Muscle Machine Inc	138500
	187	AV Stores, Co.	136800

And the following query returns 5 customers who have the lowest credit limit:

```

1 SELECT customernumber,
2     customername,
3     creditlimit
4 FROM customers
5 ORDER BY creditlimit ASC
6 LIMIT 5;
```

	customernumber	customername	creditlimit
▶	168	American Souvenirs Inc	0
	481	Raanan Stores, Inc	0
	480	Kremlin Collectables, Co.	0
	273	Franken Gifts, Co	0
	477	Mit Vergnügen & Co.	0

Using MySQL LIMIT to get the N highest values

One of the toughest questions in MySQL is how to select the N highest values in a result set e.g., select the second most expensive product, which you cannot use MAX or MIN functions to answer. However, you can use MySQL LIMIT to answer those kinds of questions.

Let's take a look at the products result set of the following query:

```

1 SELECT productName,
2     buyprice
3 FROM products
4 ORDER BY buyprice DESC;

```

	productName	buyprice
▶	1962 LanciaA Delta 16V	103.42
	1998 Chrysler Plymouth Prowler	101.51
	1952 Alpine Renault 1300	98.58
	1956 Porsche 356A Coupe	98.3
	2001 Ferrari Enzo	95.59
	1968 Ford Mustang	95.34
	1995 Honda Civic	93.89

Our task is to get the highlight product, which is the second most expensive product in the products result set. In order to do so, you use `LIMIT` clause to select 1 row from the second row as the following query: (notice that the offset starts from zero)

```

1 SELECT productName,
2     buyprice
3 FROM products
4 ORDER BY buyprice DESC
5 LIMIT 1, 1

```

	productName	buyprice
▶	1998 Chrysler Plymouth Prowler	101.51

Querying Data with MySQL IN Operator

Introduction to the MySQL IN Operator

The `IN` operator allows you to determine if a specified value matches any one of a list or a subquery. The following illustrates the syntax of the `IN` operator.

```
1 SELECT column_list
2 FROM table_name
3 WHERE (expr|column) IN ('value1','value2',...)
```

In the query above:

- You can use a `column` or an expression (`expr`) with the `IN` operator in the `WHERE` clause of the `SELECT` statement.
- The values in the list must be separated by a comma (,)
- The `IN` operator can also be used in the `WHERE` clause of other statements such as `INSERT`, `UPDATE`, `DELETE`, etc.

The `IN` operator returns 1 if the value of the `column` 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:

- First, MySQL evaluates the values based on the type of the `column` or result of the `expr`.
- Second, MySQL sorts the values.
- Third, MySQL searches for values using binary search algorithm which is very fast.

Therefore a query that uses the `IN` operator with a list of constants will perform very fast.

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.

Let's practice with some examples of using the `IN` operator.

MySQL IN examples

If you want to find out all offices which locates in the U.S. and France, you can use the `IN` operator as the following query:

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

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

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

```

1 SELECT officeCode, city, phone
2 FROM offices
3 WHERE country = 'USA' OR country = 'France'
```

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

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

```

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

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

MySQL IN with subquery

The `IN` operator is often used with a subquery. For example, if you want to find order whose total amount is greater than \$60K, you can use the `IN` operator as the following query:

```

1 SELECT orderNumber,
2     customerNumber,
3     status,
4     shippedDate
5 FROM orders
6 WHERE orderNumber IN (
7     SELECT orderNumber
8     FROM orderDetails
9     GROUP BY orderNumber
10    HAVING SUM(quantityOrdered * priceEach) > 60000)
```

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

MySQL BETWEEN Operator Explained

Introduction to MySQL BETWEEN Operator

The `BETWEEN` operator allows you to specify a range to test. The following illustrates the syntax of the `BETWEEN` operator:

```
1 expr (NOT) BETWEEN begin_expr AND end_expr
```

In the expression above:

- All expressions: `expr`, `begin_expr` and `end_expr` must return values with the same data type.
- The `BETWEEN` operator returns 1 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 `end_expr`, otherwise it returns 0.
- The `NOT BETWEEN` returns 1 if the value of `expr` is less than (`<`) the value of `begin_expr` or greater than the value of `end_expr`, otherwise it returns 0.
- If any expression above is `NULL`, the `BETWEEN` returns `NULL`.

The `BETWEEN` operator is typically used in the `WHERE` clause of `SELECT`, `INSERT`, `UPDATE` and `DELETE` statements.

MySQL BETWEEN examples

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

MySQL BETWEEN with number examples

Suppose you want to find product whose buy price within the range of \$90 and \$100, you can use the `BETWEEN` operator as the following query:

```
1 SELECT productCode,
2    productName,
3    buyPrice
4 FROM products
5 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

You can achieve the same result by using the greater than or equal (\geq) and less than or equal (\leq) operators as the following query:

```
1 SELECT productCode,
2     productName,
3     buyPrice
4 FROM products
5 WHERE buyPrice >= 90 AND buyPrice <= 100
```

To find the product whose buy price is out of the range of \$20 and \$100, you use combine the **BETWEEN** operator with the **NOT** operator as follows:

```
1 SELECT productCode,
2     productName,
3     buyPrice
4 FROM products
5 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

The query above is equivalent to the following query that uses the comparison operators, greater than operator ($>$) and less than operator ($<$) and a logical operator **OR**.

```
1 SELECT productCode,
2     productName,
3     buyPrice
4 FROM products
5 WHERE buyPrice < 20 OR buyPrice > 100
```

MySQL BETWEEN with dates example

When you use the **BETWEEN** operator with date values, to get the best result, you should use the **CAST** function to explicitly convert the type of column or expression to the **DATE** type. For example, to get the orders whose required date is from 01/01/2003 to 01/31/2003, you use the following query:

```
1 SELECT orderNumber,
2     requiredDate,
3     status
4 FROM orders
5 WHERE requireddate
6 BETWEEN CAST('2003-01-01' AS DATE) AND
7     CAST('2003-01-31' AS DATE)
```

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

In the query above, because the data type of the required date column is `DATE` so we used the `CAST` function to convert the literal strings '2003-01-01' and '2003-12-31' to the `DATE` data type.

Using MySQL LIKE Operator to Select Data Based On Patterns

The `LIKE` operator is commonly used to select data based on patterns. Using the `LIKE` operator in appropriate way is essential to increase the query performance.

The `LIKE` operator allows you to select data from a table based on a specified pattern. Therefore the `LIKE` operator is often used in the `WHERE` clause of the `SELECT` statement.

MySQL provides two wildcard characters for using with the `LIKE` operator, the percentage `%` and underscore `_`.

- The percentage (`%`) wildcard allows you to match any string of zero or more characters.
- The underscore (`_`) wildcard allows you to match any single character.

MySQL LIKE examples

Let's practice with some examples of how to use the `LIKE` operator.

MySQL LIKE with percentage (%) wildcard

Suppose you want to search for employee whose first name starts with character 'a', you can use the percentage wildcard (`%`) at the end of the pattern as follows:

```
1 SELECT employeeNumber, lastName, firstName
2 FROM employees
3 WHERE firstName LIKE 'a%'
```

employeeNumber	lastName	firstName
1143	Bow	Anthony
1611	Foxter	Andy

MySQL scans the whole `employees` table to find employee whose first name starts with character 'a' and followed by any number of characters.

To search for employee whose last name ends with 'on' string e.g., `Patterson`, `Thompson`, you can use the `%` wildcard at the beginning of the pattern as the following query:

```

1 SELECT employeeNumber, lastName, firstName
2 FROM employees
3 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 column, 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' string, you can execute following query:

```

1 SELECT employeeNumber, lastName, firstName
2 FROM employees
3 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

MySQL LIKE with underscore(_) wildcard

To find employee whose first name starts with T, ends with m and contains any single character between e.g., Tom, Tim, you use the underscore wildcard to construct the pattern as follows:

```

1 SELECT employeeNumber, lastName, firstName
2 FROM employees
3 WHERE firstname LIKE 'T_m'

```

	employeeNumber	lastName	firstName
▶	1619	King	Tom

MySQL LIKE operator with NOT operator

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

Suppose you want to search for employee whose last name does not start with character ' B', you can use the NOT LIKE with the pattern as the following query:

```
1 SELECT employeeNumber, lastName, firstName
2 FROM employees
3 WHERE lastName NOT LIKE 'B%'
```

	employeeNumber	lastName	firstName
	1002	Murphy	Diane
	1056	Patterson	Mary
	1076	Firelli	Jeff
	1088	Patterson	William
	1165	Jennings	Leslie
	1166	Thompson	Leslie

Notice that the pattern is not case sensitive with the `LIKE` operator therefore the `'b%'` and `'B%'` patterns produce the same result.

MySQL LIKE 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 interprets the wildcard character as 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 product whose product code contains string `_20`, you can perform following query:

```
1 SELECT productCode, productName
2 FROM products
3 WHERE productCode LIKE '%\_20%'
```

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

```
1 SELECT productCode, productName
2 FROM products
3 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 `_20` string.

The `LIKE` operator forces MySQL to scan the whole table to find the matching rows therefore it does not allow the database engine to use index for fast searching. As the result,

the performance of the query that uses the `LIKE` operator degrades when you query data from a table with a large number of rows.

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

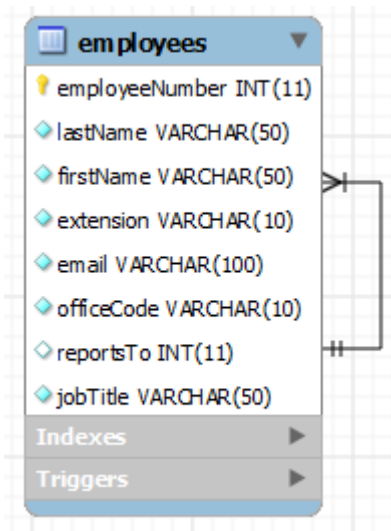
Sometimes the names of columns are so technical that make the query's output difficult to understand. To give a column a descriptive name, you use a column alias. The following illustrates how to use the column alias:

```
1 SELECT [col1 | expression] AS `descriptive name`
2 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 the it as shown in the syntax. 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 combine them to produce the full names. The `CONCAT_WS` function is used to concatenate first name and last name.

```
1 SELECT CONCAT_WS(' ', lastName, firstName)
2 FROM employees;
```

CONCAT_WS(' ', lastName, firstname)
Murphy, Diane
Patterson, Mary
Firelli, Jeff
Patterson, William
Bondur, Gerard
Bow, Anthony
Jennings, Leslie

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:

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

Full name
Murphy, Diane
Patterson, Mary
Firelli, Jeff
Patterson, William
Bondur, Gerard
Bow, Anthony
Jennings, Leslie

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:

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

Full name
Bondur, Gerard
Bondur, Loui
Bott, Larry
Bow, Anthony
Castillo, Pamela
Firelli, Jeff
Firelli, Julie
Foster, Andy

The following statement selects the *order* whose total amount is greater than 60000. It uses column aliases in `GROUP BY` and `HAVING` clauses.

```
1 SELECT orderNumber `Order no.`,
2 SUM(priceEach * quantityOrdered) Total
3 FROM orderDetails
4 GROUP BY `Order no.`
5 HAVING total > 60000;
```

	Order no.	Total
▶	10165	67392.84999999999
	10287	61402
	10310	61234.669999999984

Notice that you cannot use 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

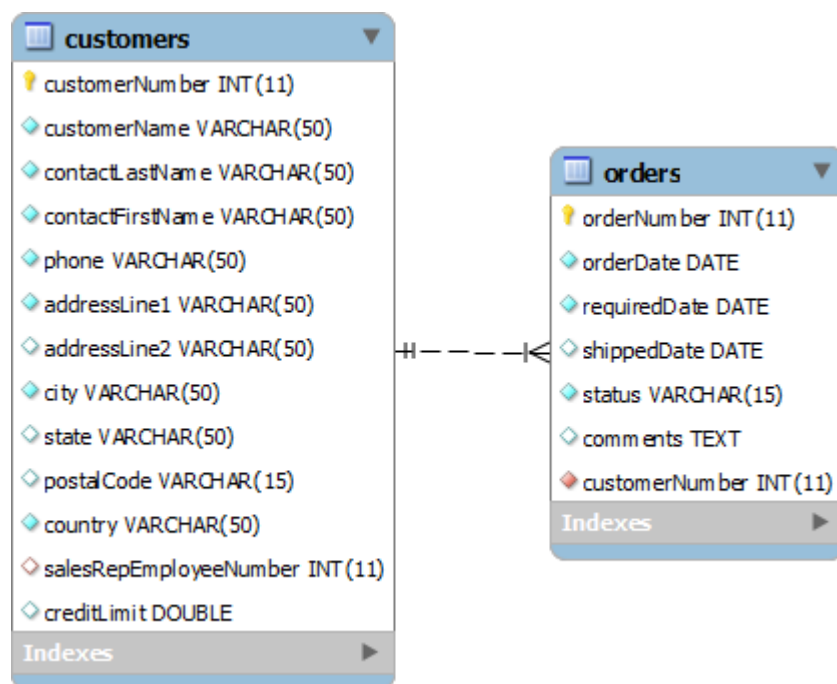
An alias also gives a table a different name. You assign a table an alias by using the `AS` keyword as the following syntax:

```
1 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:



See the following query:

```
1 SELECT customerName,
2 COUNT(o.orderNumber) total
3 FROM customers c
4 INNER JOIN orders o ON c.customerNumber = o.customerNumber
5 GROUP BY customerName
6 ORDER BY total DESC
```

customerName	total
Euro+ Shopping Channel	26
Mini Gifts Distributors Ltd.	17
Australian Collectors, Co.	5
Down Under Souvenirs, Inc	5
Danish Wholesale Imports	5
Reims Collectables	5
Dragon Souvenirs, Ltd.	5

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 is referred 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.

Combining Result Sets by Using MySQL UNION

MySQL UNION Operator

MySQL `UNION` operator allows you to combine two or more result sets from multiple tables into a single result set. The syntax of the MySQL `UNION` is as follows:

```
1 SELECT column1, column2
2 UNION [DISTINCT | ALL]
3 SELECT column1, column2
4 UNION [DISTINCT | ALL]
5
6 ...
```

There are some rules that you need to follow in order to use the `UNION` operator:

- The number of columns appears in the corresponding `SELECT` statements must be equal.
- The columns appear in the corresponding positions of each `SELECT` statement must have the same data type or at least convertible data type.

By default, the `UNION` operator eliminates duplicate rows from the result even if you don't use `DISTINCT` operator explicitly. Therefore it is said that `UNION` clause is the shortcut of `UNION DISTINCT`.

If you use the `UNION ALL` explicitly, the duplicate rows, if available, remain in the result. The `UNION ALL` performs faster than the `UNION DISTINCT`.

MySQL UNION example

Let's practice with an example of using MySQL `UNION` to get a better understanding.

Suppose you want to combine data from the `customers` and `employees` tables into a single result set, you can `UNION` operator as the following query:

```
1 SELECT customerNumber id, contactLastname name
2 FROM customers
3 UNION
4 SELECT employeeNumber id,firstname name
5 FROM employees
```

Here is the output:

	id	name
	103	Schmitt
	112	King
	114	Ferguson
	119	Labrune
	121	Bergulfsen
	124	Nelson

MySQL UNION without Alias

In the example above, we used the column alias for each column in the `SELECT` statements. What would be the output if we didn't use the column alias? MySQL uses the names of columns in the first `SELECT` statement as the labels for the output.

Let's try the query that combines *customers* and *employees* information without using column alias:

```

1 (SELECT customerNumber, contactLastname
2 FROM customers)
3 UNION
4 (SELECT employeeNumber, firstname
5 FROM employees)
6 ORDER BY contactLastname, customerNumber

```

The result has `customerNumber` and `contactLastname` as the label, which are the names of columns in the first `SELECT` statement.

	customerNumber	contactLastname
	249	Accorti
	481	Altagar,G M
	307	Andersen
	1611	Andy
	1143	Anthony
	465	Anton

MySQL UNION with ORDER BY

If you want to sort the results returned from the query using the `UNION` operator, you need to use `ORDER BY` clause in the last SQL `SELECT` statement. You can put each `SELECT` statement in the parentheses and use the `ORDER BY` clause as the last statement.

Let's take a look at the following example:

```

1 (SELECT customerNumber id,contactLastname name
2 FROM customers)
3 UNION
4 (SELECT employeeNumber id,firstname name

```

```
5 FROM employees)
6 ORDER BY name,id
```

id	name
249	Accorti
481	Altagar,G M
307	Andersen
1611	Andy
1143	Anthony
465	Anton
187	Ashworth
204	Barrias

In the query above, first we combine `id` and `name` of both *employees* and *customers* into one result set using the `UNION` operator. Then we sort the result set by using the `ORDER BY` clause. Notice that we put the `SELECT` statements inside the parentheses and place the `ORDER BY` clause as the last statement.

If you place the `ORDER BY` clause in each `SELECT` statement, it will not affect the order of the rows in the final result produced by the `UNION` operator.

MySQL also provides you with alternative option to sort the result set based on column position using `ORDER BY` clause as the following query:

```
1 (SELECT customerNumber, contactLastname
2 FROM customers)
3 UNION
4 (SELECT employeeNumber,firstname
5 FROM employees)
6 ORDER BY 2, 1
```

MySQL INNER JOIN

Introducing MySQL INNER JOIN clause

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

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

- First, you have to specify the main table that appears in the `FROM` clause.

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

The syntax of the MySQL `INNER JOIN` clause is as follows:

```

1 SELECT column_list
2 FROM t1
3 INNER JOIN t2 ON join_condition1
4 INNER JOIN t3 ON join_condition2
5 ...
6 WHERE where_conditions;
```

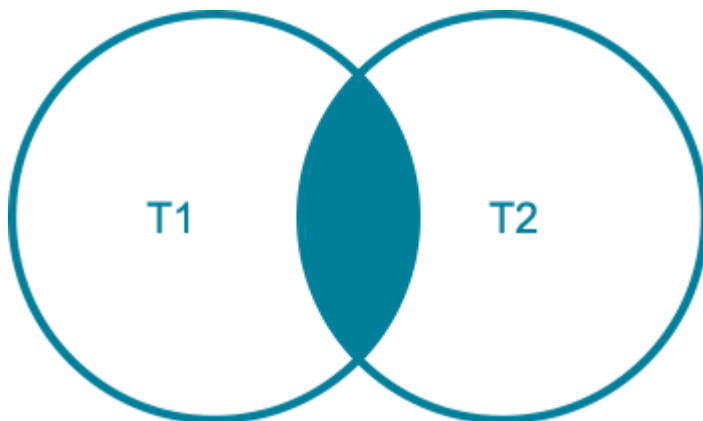
Let's simplify the syntax above by assuming that we are joining two tables `T1` and `T2` using the `INNER JOIN` clause.

For each record in the `T1` table, the MySQL `INNER JOIN` clause compares it with each record of the `T2` table to check if both of them satisfy the join condition. When the join condition is matched, it will return that record that combine columns in either or both `T1` and `T2` tables.

Notice that the records on 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.

The logic is applied if we join more than 2 tables.

The following Venn diagram illustrates how the MySQL `INNER JOIN` clause works.



MySQL `INNER JOIN` Venn Diagram

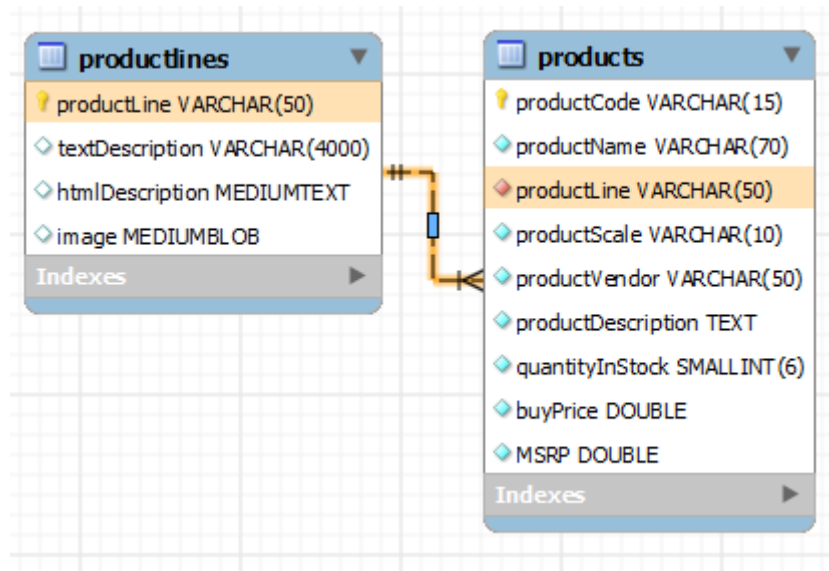
Avoid ambiguous column error in MySQL INNER JOIN

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` clause to avoid ambiguous column error. For example, if both `T1` and `T2` tables have the same column named `C`; in the `SELECT` clause, you have to refer to `C` column using the table qualifiers as `T1.C` or `T2.C`.

To save time typing the table qualifiers, you can use table aliases in the query. For example, you can give the `verylongtablename` table an alias `T` and refer to its columns using `T.column` instead of `verylongtablename.column`.

Examples of using MySQL INNER JOIN clause

Let's take a look at two tables: `products` and `productlines` tables in the sample database.



Now, if you want to get

- The *product code* and *product name* from the `products` table.
- The *text description* of product lines from the `productlines` table.

You need to select data from both tables and match rows by comparing the `productline` column from the `products` table with the `productline` column from the `productlines` table as the following query:

```
1 SELECT productCode,
2    productName,
3    textDescription
4 FROM products T1
5 INNER JOIN productlines T2 ON T1.productline = T2.productline;
```

	productCode	productName	textDescription
▶	S10_1678	1969 Harley Davidson Ultimate Chopper	Our motorcycles are state of the art replicas of classic as well as contemporary motorcycle legend
	S10_1949	1952 Alpine Renault 1300	Attention car enthusiasts: Make your wildest car ownership dreams come true. Whether you are
	S10_2016	1996 Moto Guzzi 1100i	Our motorcycles are state of the art replicas of classic as well as contemporary motorcycle legend
	S10_4698	2003 Harley-Davidson Eagle Drag Bike	Our motorcycles are state of the art replicas of classic as well as contemporary motorcycle legend
	S10_4757	1972 Alfa Romeo GTA	Attention car enthusiasts: Make your wildest car ownership dreams come true. Whether you are
	S10_4962	1962 LanciaA Delta 16V	Attention car enthusiasts: Make your wildest car ownership dreams come true. Whether you are

MySQL INNER JOIN with GROUP BY clause

We 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:

```

1 SELECT T1.orderNumber,
2     status,
3     SUM(quantityOrdered * priceEach) total
4 FROM orders AS T1
5 INNER JOIN orderdetails AS T2 ON T1.orderNumber = T2.orderNumber
6 GROUP BY orderNumber

```

	orderNumber	status	total
▶	10100	Shipped	10223.829999999998
	10101	Shipped	10549.01
	10102	Shipped	5494.78
	10103	Shipped	50218.950000000004
	10104	Shipped	40206.2
	10105	Shipped	53959.21
	10106	Shipped	52151.810000000005

Introducing to 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 we are going to query data from two tables `T1` and `T2`. The following is the syntax of the `LEFT JOIN` clause that joins the two tables:

```

1 SELECT T1.c1, T1.c2,... T2.c1,T2.c2
2 FROM T1
3 LEFT JOIN T2 ON T1.c1 = T2.c1...

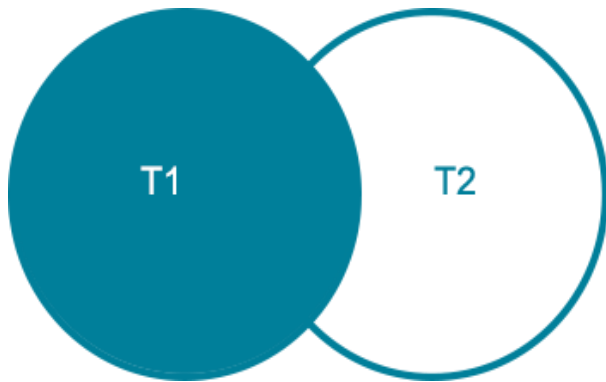
```

When we join the `T1` table to the `T2` table using the `LEFT JOIN` clause, if a row from the left table `T1` matches a row from the right table `T2` based on the join condition (`T1.c1 = T2.c1`), this row is included in the result set. In case the row in the left table does not match 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` values 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 match, plus all rows from the left table (`T1`) even there is no match found for them in the right table (`T2`).

The following Venn diagram helps you visualize how the MySQL `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 matches in the `T2` table. All rows in the left table are included in the result set.



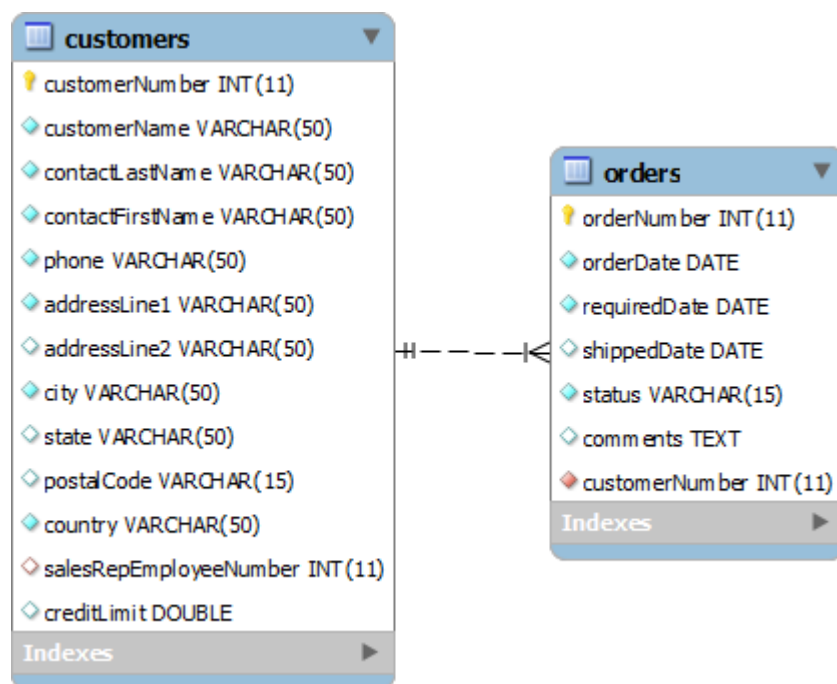
MySQL `LEFT JOIN` – Venn Diagram

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

MySQL `LEFT JOIN` Examples

MySQL `LEFT JOIN` clause – joining 2 tables example

Let's take a look at the `customers` and `orders` tables.



In the database diagram above:

- Each order in the `orders` table must belong to a customer in the `customers` table.
- 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:

```
1 SELECT c.customerNumber,
2    c.customerName,
3    orderNumber,
4    o.status
5 FROM customers c
6 LEFT JOIN orders o ON c.customerNumber = o.customerNumber
```

166	Handji Gifts& Co	10217	Shipped
166	Handji Gifts& Co	10259	Shipped
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 Imp...	10180	Shipped
171	Daedalus Designs Imp...	10224	Shipped

The left table is the `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 those customers do not have any order in the `orders` table.

If you replace the `LEFT JOIN` clause by the `INNER JOIN` clause, you only get the customers who have orders in the `orders` table.

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 unmatched rows between two tables, you add a `WHERE` clause to the `SELECT` statement to select only rows whose column values in the right table contains the `NULL` values.

For example, to find all customers who have not ordered any product, you can use the following query:

```
1 SELECT c.customerNumber,
2     c.customerName,
3     orderNumber,
4     o.status
5 FROM customers c
6 LEFT JOIN orders o ON c.customerNumber = o.customerNumber
7 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

MySQL Self Join

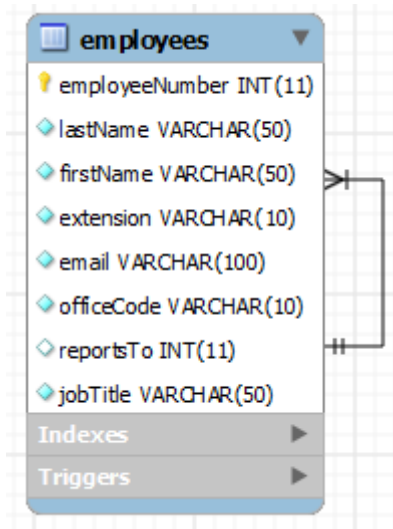
In the previous tutorial, you have learned how to join a table to the other tables using `INNER JOIN`, `LEFT JOIN` or `RIGHT JOIN` statement. However, there is a special case that you can join a table to itself, which is known as self join.

You use self join when you want to combine records with other records 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.

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.



In order to get the whole organization structure, we 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 Report*.

```
1 SELECT CONCAT(m.lastname,', ',m.firstname) AS 'Manager',
2     CONCAT(e.lastname,', ',e.firstname) AS 'Direct report'
3 FROM employees e
4 INNER JOIN employees m ON m.employeeNumber = e.reportsTo
5 ORDER BY manager
```

	Manager	Direct report
▶	Bondur, Gerard	Castillo, Pamela
	Bondur, Gerard	Hernandez, Gerard
	Bondur, Gerard	Bondur, Loui
	Bondur, Gerard	Gerard, Martin
	Bondur, Gerard	Jones, Barry
	Bondur, Gerard	Bott, Larry
	Bow, Anthony	Firelli, Julie
	Bow, Anthony	Thompson, Leslie
	Bow, Anthony	Jennings, Leslie

In the above example, we only see employees who have manager. However, we 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 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. We also need to use the `IFNULL` function to display the top manager if the manger's name is `NULL`.

```

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

```

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

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

```

1 SELECT c1.city,
2     c1.customerName,
3     c2.customerName
4 FROM customers c1
5 INNER JOIN customers c2
6 ON c1.city = c2.city AND
7     c1.customername <> c2.customerName
8 ORDER BY c1.city

```

city	customerName	customerName
Auckland	GiftsForHim.com	Down Under Souvenirs, Inc
Auckland	Down Under Souvenirs, Inc	GiftsForHim.com
Auckland	GiftsForHim.com	Kelly's Gift Shop
Auckland	Kelly's Gift Shop	Down Under Souvenirs, Inc
Auckland	Kelly's Gift Shop	GiftsForHim.com
Auckland	Down Under Souvenirs, Inc	Kelly's Gift Shop
Boston	Gifts4AllAges.com	Diecast Collectables
Boston	Diecast Collectables	Gifts4AllAges.com

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.

MySQL GROUP BY

Introducing to MySQL GROUP BY clause

The MySQL GROUP BY clause is used with the SELECT statement to group rows into subgroups by the one or more values of columns or expressions.

The MySQL GROUP BY clause is an optional part of the SELECT statement. It must appear after the FROM or WHERE clause. The MySQL GROUP BY clause consists of the GROUP BY keyword followed by a list of comma-separated columns or expressions.

The following illustrates the MySQL GROUP BY clause syntax:

```
1 SELECT c1,c2,... cn, aggregate_function(expression)
2 FROM table
3 WHERE where_conditions
4 GROUP BY c1, c2, ... cn
```

MySQL GROUP BY Examples

Let's take a look at the `orders` table in the sample database. Suppose you want to group values of the `order status` into subgroups, you use the GROUP BY clause with the `status` column as the following query:

```
1 SELECT status
2 FROM orders
3 GROUP BY status
```

status
Cancelled
Disputed
In Process
On Hold
Resolved
Shipped

You can see that the GROUP BY clause returns unique occurrences of `status` values. It works like the DISTINCT operator as using in the following query:

```
1 SELECT DISTINCT status
2 FROM orders
```

MySQL GROUP BY with aggregate functions

The aggregate functions allow you to perform calculation of a set of records and return a single value. The most common aggregate functions are `SUM`, `AVG`, `MAX`, `MIN` and `COUNT`.

An aggregate functions is often used with the MySQL `GROUP BY` clause to perform calculation on each subgroup and return a single value for each subgroup. For example, if you want to know how many *orders* in each *status*, you can use the `COUNT` function with the `GROUP BY` clause as follows:

```
1 SELECT status, count(*)
2 FROM orders
3 GROUP BY status
```

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

MySQL GROUP BY vs. ANSI SQL GROUP BY

MySQL follows ANSI SQL. However, MySQL gives you more flexibility when using the `GROUP BY` clause:

- In ANSI SQL, you must list all columns that you use in the `SELECT` clause in the `GROUP BY` clause. MySQL does not have this restriction. MySQL allows you to have additional columns in the `SELECT` clause that are not specified in the `GROUP BY` clause.
- MySQL also allows you to sort the group order in which the results are returned. The default order is ascending.

If you want to see the `status` and the number of orders in descending order, you can use the `GROUP BY` clause with `DESC` as the following query:

```
1 SELECT status, count(*)
2 FROM orders
3 GROUP BY status DESC;
```

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

Notice that we use `DESC` in the `GROUP BY` clause to sort the `status` in descending order. You can specify `ASC` explicitly in the `GROUP BY` clause to sort the groups in ascending order.

Introducing MySQL HAVING clause

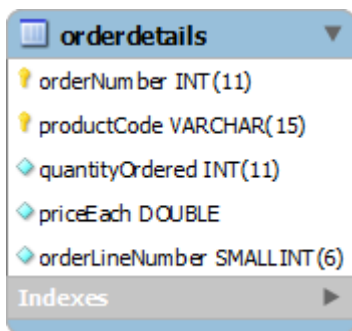
The MySQL `HAVING` clause is used in the `SELECT` statement to specify filter conditions for group of rows or aggregates.

The MySQL `HAVING` clause is often used with the `GROUP BY` clause. When using with the `GROUP BY` clause, you can apply a filter condition to the columns that appear in the `GROUP BY` clause. If the `GROUP BY` clause is omitted, the MySQL `HAVING` clause behaves like the `WHERE` clause. Notice that the MySQL `HAVING` clause applies the condition to each group of rows, while the `WHERE` clause applies the condition to each individual row.

Examples of using MySQL HAVING clause

Let's take a look at an example of using MySQL `HAVING` clause.

We will use the `orderdetails` table in the sample database for the sake of demonstration.



orderdetails
orderNumber INT(11)
productCode VARCHAR(15)
quantityOrdered INT(11)
priceEach DOUBLE
orderLineNumber SMALLINT(6)
Indexes

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

```
1 SELECT ordernumber,
2     SUM(quantityOrdered) AS itemsCount,
3     SUM(priceeach) AS total
4 FROM orderdetails
5 GROUP BY ordernumber
```

	ordernumber	itemsCount	total
▶	10100	151	301.84000000000003
	10101	142	352
	10102	80	138.68
	10103	541	1520.3699999999997
	10104	443	1251.8899999999999
	10105	545	1479.71

Now, we can find which order has total sales greater than \$1000. We use the MySQL HAVING clause on the aggregate as follows:

```

1 SELECT ordernumber,
2    SUM(quantityOrdered) AS itemsCount,
3    SUM(priceeach) AS total
4 FROM orderdetails
5 GROUP BY ordernumber
6 HAVING total > 1000

```

	ordernumber	itemsCount	total
▶	10103	541	1520.3699999999997
	10104	443	1251.8899999999999
	10105	545	1479.71
	10106	675	1427.2800000000002
	10108	561	1432.86
	10110	570	1338.4699999999998

We can construct a complex condition in the MySQL HAVING clause using logical operators such as OR and AND. Suppose we want to find which order has total sales greater than \$1000 and contains more than 600 items, we can use the following query:

```

1 SELECT ordernumber,
2    sum(quantityOrdered) AS itemsCount,
3    sum(priceeach) AS total
4 FROM orderdetails
5 GROUP BY ordernumber
6 HAVING total > 1000 AND itemsCount > 600

```

	ordernumber	itemsCount	total
▶	10106	675	1427.2800000000002
	10126	617	1623.71
	10135	607	1494.86
	10165	670	1794.9399999999996
	10168	642	1472.5
	10204	619	1619.73
	10207	615	1560.08

Suppose we want to find all orders that has shipped and has total sales greater than \$1500, we can join the orderdetails table with the orders table by using the INNER JOIN clause, and apply condition on the status column and the total aggregate as the following query:

```

1 SELECT a.ordernumber,
2    SUM(priceeach) total,
3    status
4 FROM orderdetails a

```

```
5 INNER JOIN orders b ON b.ordernumber = a.ordernumber
6 GROUP BY ordernumber
7 HAVING b.status = 'Shipped' AND
8     total > 1500;
```

	ordernumber	total	status
▶	10103	1520.3699999999997	Shipped
	10122	1598.2699999999995	Shipped
	10126	1623.71	Shipped
	10127	1601.3899999999999	Shipped
	10142	1570.77	Shipped
	10159	1686.9999999999998	Shipped
	10165	1794.9399999999996	Shipped

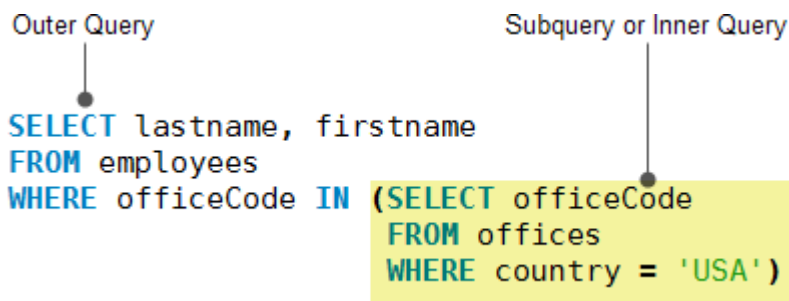
The MySQL HAVING clause is only useful when we use it with the GROUP BY clause to generate the output of the high-level reports. For example, we can use the MySQL HAVING clause to answer some kinds of queries like give me all the orders in this month, this quarter and this year that have total sales greater than 10K.

MySQL Subquery

A MySQL subquery is a query that is nested inside another query such as SELECT, INSERT, UPDATE or DELETE. A MySQL subquery is also can be nested inside another subquery. A MySQL subquery is also called an inner query, while the query that contains the subquery is called an outer query.

Let's take a look at the following subquery that returns employees who locate in the offices in the USA.

- The subquery returns all *offices codes* of the offices that locate in the USA.
- The outer query selects the last name and first name of employees whose office code is in the result set returned from the subquery.



You can use a subquery anywhere an expression can be used. A subquery also must be enclosed in parentheses.

MySQL subquery within a WHERE clause

MySQL subquery with comparison operators

If a subquery returns a single value, you can use comparison operators to compare it with the expression in the WHERE clause. For example, the following query returns the customer who has the maximum payment.

```
1 SELECT customerNumber,
2    checkNumber,
3    amount
4 FROM payments
5 WHERE amount = (
6    SELECT MAX(amount)
7    FROM payments
8 )
```

	customerNumber	checkNumber	amount
►	141	JE105477	120166.58

You can also use other comparison operators such as greater than (>), less than(<), etc. For example, you can find customer whose payment is greater than the average payment. A subquery is used to calculate the average payment by using the `AVG` aggregate function. The outer query selects payments that are greater than the average payment returned from the subquery.

```
1 SELECT customerNumber,
2    checkNumber,
3    amount
4 FROM payments
5 WHERE amount > (
6    SELECT AVG(amount)
7    FROM payments
8 )
```

	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

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. For example, you can use a subquery with `NOT IN` operator to find customer who has not ordered any product as follows:

```

1 SELECT customername
2 FROM customers
3 WHERE customerNumber NOT IN(
4   SELECT DISTINCT customernumber
5   FROM orders
6 )

```

	customername
▶	Havel & Zbyszek Co
	American Souvenirs Inc
	Porto Imports Co.
	Asian Shopping Network, Co
	Natürlich Autos
	ANG Resellers
	Messner Shopping Network

MySQL subquery with EXISTS and NOT EXISTS

When a subquery is used with `EXISTS` or `NOT EXISTS` operator, a subquery returns a Boolean value of `TRUE` or `FALSE`. The subquery acts as an existence check.

In the following example, we select a list of customers who have at least one order with total sales greater than 10K.

First, we build a query that checks if there is at least one order with total sales greater than 10K:

```

1 SELECT priceEach * quantityOrdered
2 FROM orderdetails
3 WHERE priceEach * quantityOrdered > 10000
4 GROUP BY orderNumber

```

	priceEach * quantityOrdered
▶	10286.400000000001
	11503.14
	10460.16
	11170.52
	10723.6
	10072

The query returns 6 records so that when we use it as a subquery, it will return `TRUE`; therefore the whole query will return all customers:

```

1 SELECT customerName
2 FROM customers
3 WHERE EXISTS (
4   SELECT priceEach * quantityOrdered
5   FROM orderdetails

```

```

6 WHERE priceEach * quantityOrdered > 10000
7 GROUP BY orderNumber
8 )

```

	customerName
►	Atelier graphique
	Signal Gift Stores
	Australian Collectors, Co.
	La Rochelle Gifts
	Baane Mini Imports
	Mini Gifts Distributors Ltd.
	Havel & Zbyszek Co
	Blauer See Auto Co

If you replace the `EXISTS` by `NOT EXISTS` in the query, it will not return any record at all.

MySQL subquery in FROM clause

When you use a subquery in the `FROM` clause, the result set returned from a subquery is used as a 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:

```

1 SELECT max(items),
2        min(items),
3        floor(avg(items))
4 FROM
5 (SELECT orderNumber,
6        count(orderNumber) AS items
7 FROM orderdetails
8 GROUP BY orderNumber) AS lineitems

```

Notice that the subquery returns the following result set that is used as a derived table for the outer query.

	max(items)	min(items)	floor(avg(items))
►	18	1	9

MySQL correlated subquery

In the previous examples, we see the subquery itself is independent. It means that you can execute the subquery as a normal query. However a correlated subquery is a subquery that uses the information from the outer query, or we can say that a correlated subquery depends on the outer query. A correlated subquery is evaluated once for each row in the outer query.

In the following correlated subquery, we select products whose buy price is greater than the average buy price of all products for a particular *product line*.

```

1 SELECT productname,
2    buyprice
3 FROM products AS p1
4 WHERE buyprice > (
5    SELECT AVG(buyprice)
6    FROM products
7    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

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.

Simple MySQL INSERT statement

The MySQL INSERT statement allows you to insert data into tables. The following illustrates the syntax of the `INSERT` statement:

```

1 INSERT INTO table(column1,column2...)
2 VALUES (value1,value2,...)

```

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

You need to have the `INSERT` privilege to use the `INSERT` statement.

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

```

1 CREATE TABLE IF NOT EXISTS tasks (
2   task_id int(11) NOT NULL AUTO_INCREMENT,
3   subject varchar(45) DEFAULT NULL,
4   start_date DATE DEFAULT NULL,
5   end_date DATE DEFAULT NULL,
6   description varchar(200) DEFAULT NULL,
7   PRIMARY KEY (task_id)
8 )

```

For example, if you want to insert a new task into the `tasks` table, you use the `INSERT` statement as follows:

```

1 INSERT INTO tasks(subject,start_date,end_date,description)
2 VALUES('Learn MySQL INSERT','2010-01-01','2010-01-02','Start learning..')

```

After executing the statement, MySQL returns a message to indicate how many rows were affected. In this case, one row were affected.

MySQL INSERT – insert multiple rows

In order to insert multiple rows into a table, you use the `INSERT` statement with the following syntax:

```
1 INSERT INTO table(column1,column2...)
2 VALUES (value1,value2,...),
3     (value1,value2,...),
4 ...
```

Each row, which you want to insert into the table, is specified by a comma-separated values inside parentheses. For example, to insert multiple tasks in the `tasks` table, you use the following query:

```
1 INSERT INTO tasks(subject,start_date,end_date,description)
2 VALUES ('Task 1','2010-01-01','2010-01-02','Description 1'),
3     ('Task 2','2010-01-01','2010-01-02','Description 2'),
4     ('Task 3','2010-01-01','2010-01-02','Description 3');
```

3 rows affected. Great!

If you specify the values of the corresponding column for all columns in the table, you can ignore the column list in the `INSERT` statement as follows:

```
1 INSERT INTO table
2 VALUES (value1,value2,...)
and
```

```
1 INSERT INTO table
2 VALUES (value1,value2,...),
3     (value1,value2,...),
4 ...
```

Notice that you don't have to specify the value for auto-increment column e.g., `taskid` column because MySQL generates its values automatically.

MySQL INSERT with SELECT clause

In MySQL, you can specify the values for the `INSERT` statement from a `SELECT` statement. It is very handy because you can clone a table fully or partially by using the `INSERT` and `SELECT` clauses as follows:

```
1 INSERT INTO table_1
2 SELECT c1, c2, FROM table_2;
```

First, we create a new table named `tasks_1` by cloning the structure of the `tasks` table as follows:

```
1 CREATE TABLE tasks_1
```

2 LIKE tasks;

Second, we can insert data into the `tasks_1` table from the `tasks` table by using

`INSERT` statement:

```
1 INSERT INTO tasks_1
```

```
2 SELECT * FROM tasks;
```

Third, we can check the `tasks_1` table to see if we actually clone it from the `tasks` table.

```
1 SELECT * FROM tasks_1;
```

task_id	subject	start_date	end_date	description
1	Learn MySQL INSERT	2010-01-01	2010-01-02	Start learning..
2	Task 1	2010-01-01	2010-01-02	Description 1
3	Task 2	2010-01-01	2010-01-02	Description 2
4	Task 3	2010-01-01	2010-01-02	Description 3

MySQL INSERT with ON DUPLICATE KEY UPDATE

If the new record that you want to insert may cause duplicate value in `PRIMARY KEY` or a `UNIQUE` index, MySQL will issue an error. For example, if you execute the following statement:

```
1 INSERT INTO tasks(task_id,subject,start_date,end_date,description)
```

```
2 VALUES (4,'Task 4','2010-01-01','2010-01-02','Description 4');
```

MySQL will issue an error message:

```
1 Error Code: 1062. Duplicate entry '4' for key 'PRIMARY' 0.016 sec
```

However if you specify `ON DUPLICATE KEY UPDATE` in the `INSERT` statement, MySQL will update the old record, for example:

```
1 INSERT INTO tasks(task_id,subject,start_date,end_date,description)
```

```
2 VALUES (4,'Task 4','2010-01-01','2010-01-02','Description 4')
```

```
3 ON DUPLICATE KEY UPDATE task_id = task_id + 1;
```

MySQL issues a message saying that `2 rows affected`.

Let's check the `tasks` table:

```
1 SELECT * FROM tasks;
```

	task_id	subject	start_date	end_date	description
▶	1	Learn MySQL INSERT	2010-01-01	2010-01-02	Start learning..
	2	Task 1	2010-01-01	2010-01-02	Description 1
	3	Task 2	2010-01-01	2010-01-02	Description 2
	5	Task 3	2010-01-01	2010-01-02	Description 3

A new record was inserted but the old record with `task_id` value 4 was updated. The above query is equivalent to the following query:

```
1 UPDATE tasks
2 SET task_id = task_id+1
3 WHERE task_id = 1;
```

In conclusion, when you use `ON DUPLICATE KEY UPDATE` in the `INSERT` statement with the new record that causes the duplicate value in `PRIMARY KEY` or `UNIQUE` index, MySQL updates the old record and a new record.

Introduction to MySQL UPDATE statement

The `UPDATE` statement is used to update existing data in tables. It can be used to change column values of a single row, a group of rows or all rows in a table.

The following illustrates the MySQL `UPDATE` statement syntax:

```
1 UPDATE [LOW_PRIORITY] [IGNORE] table_name [, table_name...]
2 SET column_name1 = expr1
3   [, column_name2=expr2 ...]
4 [WHERE condition]
```

Let's examine the `UPDATE` statement in greater detail:

- Followed by the `UPDATE` keyword is the name of the table that you want to update data. In MySQL, you can change the data of multiple tables using a single `UPDATE` statement. If the `UPDATE` statement violates any integrity constraint, MySQL does not perform the update and issues an error message.
- The `SET` clause determines the column names of the table and the new values. The new values could be literal values, result of expressions or subqueries.
- The `WHERE` clause determines which rows will be updated. It is an optional element of the `UPDATE` statement. If the `WHERE` clause is omitted, all rows in the table will be updated. The `WHERE` clause is so important that you should not forget. Sometimes, you may want to change just one row of the table; if you forget the `WHERE` clause, the `UPDATE` statement will update all the rows, which is not what you expected.
- The `LOW_PRIORITY` keyword is used to delay the execution until no other connections read data from the table. It is used for controlling the update process in MySQL database server.
- The `IGNORE` keyword is used to execute the update even there is an error occurred during the execution of the `UPDATE` statement. The error in the update process could be duplicate value on a unique column, the new value does not match with the column's data type, etc.

MySQL UPDATE examples

Let's practice with a couple of examples in the MySQL sample database.

MySQL UPDATE a single column in a table

In this example, we are going to update the email of Mary Patterson to the new email `mary.patterso@classicmodelcars.com`.

First, to make sure that we update the email successfully, we query Mary's email using the `SELECT` statement as follows:

```

1 SELECT firstname,
2     lastname,
3     email
4 FROM employees
5 WHERE employeeNumber = 1056

```

firstname	lastname	email
Mary	Patterson	mpatterso@classicmodelcars.com

Second, we can update her current email to the new email

`mary.patterso@classicmodelcars.com` using the `UPDATE` statement as the following query:

```

1 UPDATE employees
2 SET email = 'mary.patterso@classicmodelcars.com'
3 WHERE employeeNumber = 1056

```

Because we just want to update Mary's record so we use the `WHERE` clause to specify the Mary's record ID `1056`. The `SET` clause sets the email column value to the new email.

Third, we execute the `SELECT` statement again to verify the change.

```

1 SELECT firstname,
2     lastname,
3     email
4 FROM employees
5 WHERE employeeNumber = 1056

```

firstname	lastname	email
Mary	Patterson	mary.patterso@classicmodelcars.com

MySQL UPDATE multiple columns

To update multiple columns, you need to specify them in the `SET` clause. The following query updates both mary's last name and email:

```

1 UPDATE employees

```



```
2 SET lastname = 'Hill',
3   email = 'mary.hill@classicmodelcars.com'
4 WHERE employeeNumber = 1056;
Le'ts check the changes:
```

```
1 SELECT firstname,
2       lastname,
3       email
4 FROM employees
5 WHERE employeeNumber = 1056;
```

firstname	lastname	email
Mary	Hill	mary.hill@classicmodelcars.com

MySQL UPDATE from SELECT statement

You can provide the values for the `SET` clause from a `SELECT` statement that selects data from other tables. For example, first we check if is there any customer who does not have sales representative:

```
1 SELECT customername,
2       salesRepEmployeeNumber
3 FROM customers
4 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 any sale representative and update for those customers:

```
1 UPDATE customers
2 SET salesRepEmployeeNumber =
3   (
4     SELECT employeeNumber
5     FROM employees
6     WHERE jobtitle = 'Sales Rep'
7     LIMIT 1
8   )
9 WHERE salesRepEmployeeNumber IS NULL;
```

The `SELECT` statement returns an employee number of the employee whose job title is Sales Rep. The `UPDATE` statement uses this employee number and update it for the customers whose sales representative is not available.

MySQL UPDATE JOIN

MySQL UPDATE JOIN syntax

You often use `JOIN` clauses to query records in a table that have (in case of `INNER JOIN`) or do not have (in case of `LEFT JOIN`) corresponding records in another table. In MySQL, you can use the `JOIN` clauses in the `UPDATE` statement to perform cross-table update.

The syntax of the MySQL `UPDATE JOIN` is as follows:

```
1 UPDATE T1, T2,
2 [INNER JOIN | LEFT JOIN] T1 ON T1.C1 = T2. C1
3 SET T1.C2 = T2.C2,
4   T2.C3 = expr
5 WHERE condition
```

Let's examine the MySQL `UPDATE JOIN` syntax in greater detail:

- First, you 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 is not updated.
- Second, you specify a kind of join you want to use i.e., either `INNER JOIN` or `LEFT JOIN` and a join condition. Notice that the `JOIN` clause must appear right after the `UPDATE` clause.
- Third, you assign new values to the columns in `T1` and/or `T2` tables that you want to update.
- Fourth, the condition in the `WHERE` clause allows you to limit the rows to update.

you notice that there is another way to update data cross-table using the following syntax:

```
1 UPDATE T1, T2
2 SET T1.c2 = T2.c2,
3   T2.c3 = expr
4 WHERE T1.c1 = T2.c1 AND condition
```

This `UPDATE` statement works the same as `UPDATE JOIN` with implicit `INNER JOIN` clause. It means you can rewrite the above statement as follows:

```
1 UPDATE T1,T2
2 INNER JOIN T2 ON T1.C1 = T2.C1
3 SET T1.C2 = T2.C2,
4   T2.C3 = expr
5 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 in these examples. The sample database contains 2 tables:

- `employees` table stores employee data with employee id, name, performance and salary.
- `merits` table stores performance and merit's percentage.

The SQL script for creating and loading data in this sample database is as follows:

```

1 CREATE DATABASE IF NOT EXISTS empdb;
2
3 -- create tables
4 CREATE TABLE merits (
5     performance INT(11) NOT NULL,
6     percentage FLOAT NOT NULL,
7     PRIMARY KEY (performance)
8 );
9
10 CREATE TABLE employees (
11     emp_id INT(11) NOT NULL AUTO_INCREMENT,
12     emp_name VARCHAR(255) NOT NULL,
13     performance INT(11) DEFAULT NULL,
14     salary FLOAT DEFAULT NULL,
15     PRIMARY KEY (emp_id),
16     CONSTRAINT fk_performance
17     FOREIGN KEY(performance)
18     REFERENCES merits(performance)
19 );
20 -- insert data for merits table
21 INSERT INTO merits(performance,percentage)
22 VALUES(1,0),
23     (2,0.01),
24     (3,0.03),
25     (4,0.05),
26     (5,0.08);
27 -- insert data for employees table
28 INSERT INTO employees(emp_name,performance,salary)
29 VALUES('Mary Doe', 1, 50000),
30     ('Cindy Smith', 3, 65000),
31     ('Sue Greenspan', 4, 75000),
32     ('Grace Dell', 5, 125000),
33     ('Nancy Johnson', 3, 85000),
34     ('John Doe', 2, 45000),
35     ('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 `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 `performance` field. See the following query:

- 1 `UPDATE employees`
- 2 `INNER JOIN merits ON employees.performance = merits.performance`
- 3 `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 employee record in the `employees` table, the query checks the its performance value against the performance value in the `merits` table. If it finds a match, it gets the percentage in the `merits` table and update 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:

- 1 `INSERT INTO employees(emp_name,performance,salary)`
- 2 `VALUES('Jack William',NULL,43000),`
- 3 `('Ricky Bond',NULL,52000);`

Because these employees are new hires so their performance data is not available or `NULL`.

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 record in a table when it does not have a corresponding record in another table. For example, you can increase the salary for a new hire by 1.5% using the following statement:

- 1 UPDATE employees
- 2 LEFT JOIN merits ON employees.performance = merits.performance
- 3 SET salary = salary + salary * 0.015;
- 4 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
	15	Jack William	NULL	43645
	16	Ricky Bond	NULL	52780

Using MySQL DELETE to Remove Data from Tables

To remove data from a table, you use the MySQL `DELETE` statement. The MySQL `DELETE` statement allows you to remove records from not only one table but also multiple tables using a single `DELETE` statement.

MySQL delete from one table

To remove data from a single table, you use the following `DELETE` statement:

- 1 `DELETE FROM table`
- 2 `[WHERE conditions] [ORDER BY ...] [LIMIT rows]`

Followed the `DELETE FROM` clause is the table name that you want to delete records.

The `WHERE` clause specifies which rows you want to delete. If a record meets the `WHERE` condition, it is deleted permanently from the table. If you omit the `WHERE` clause, all records in the table are deleted.

The `DELETE` statement returns the number of rows deleted specified by the `ROW_COUNT()` function.

Notice that the `ROW_COUNT()` function returns the number of rows inserted, updated or deleted by the last `INSERT`, `UPDATE` or `DELETE` statement

MySQL delete from a table examples

Suppose you want to remove employees whose `officeNumber` is 4, you use the `DELETE` statement with the `WHERE` clause as the following query:

```
1 DELETE FROM employees
2 WHERE officeCode = 4
```

To delete all employee records from the `employees` table, you use the `DELETE` statement without the `WHERE` clause as follows:

```
1 DELETE FROM employees
```

All employee records in the `employees` table were deleted.

MySQL delete from multiple tables

To delete records from multiple tables, you use one of the following `DELETE` statements:

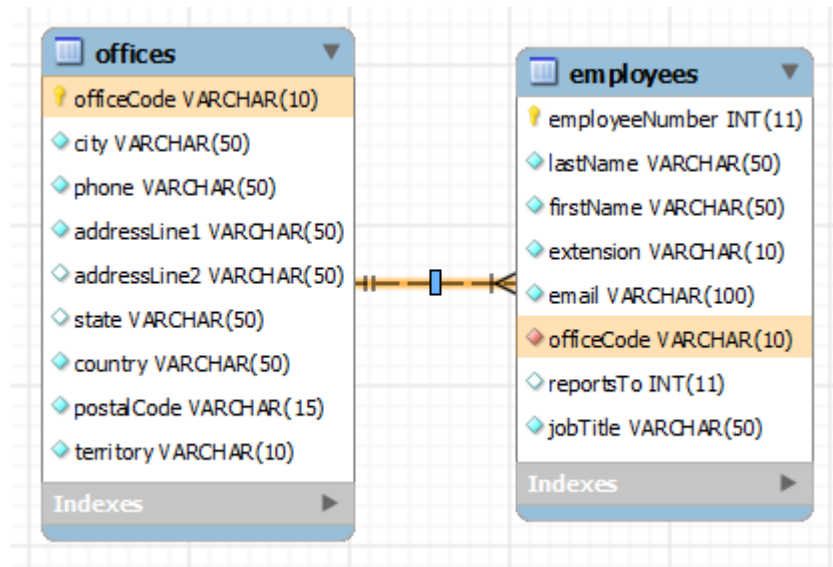
```
1 DELETE table_1, table_2,...
2 FROM table-refs
3 [WHERE conditions]
4
5 DELETE FROM table_1, table_2,...
6 USING table-refs
7 [WHERE conditions]
```

Let's examine each statement in greater detail:

- Both `DELETE` statements delete records from multiple tables i.e., `table_1, table_2,...` specified after the `DELETE` keyword.
- The first `DELETE` statement uses the `FROM` clause while the second one use the `USING` clause.
- The `WHERE` clause is used to determine which record to be deleted.
- Both statements return the number of rows deleted from the multiple tables.

MySQL delete from multiple tables example

Let's take a look at the `offices` and `employees` tables in the MySQL sample database.



Suppose one office is closed and you want to remove all employee records in the `employees` table associated with that office and also the office itself in the `offices` table, you can use the second form of the MySQL `DELETE` statement

The following query removes the office record with the code is 1 in the `offices` table and also all employee records associated with the office 1 in the `employees` table:

```

1 DELETE employees,
2   offices
3 FROM employees,
4   offices
5 WHERE employees.officeCode = offices.officeCode AND
6   offices.officeCode = 1
    
```

You can verify the changes by using the following `SELECT` statements to query data from both `employees` table and `offices` table.

```

1 SELECT * FROM employees
2 WHERE officeCode = 1;
3
4 SELECT * FROM offices
5 WHERE officeCode = 1;
    
```

You can achieve the same effect by using the second form of the MySQL `DELETE` statement as follows:

```

1 DELETE FROM employees, offices
2 USING employees, offices
3 WHERE employees.officeCode = offices.officeCode AND
4   offices.officeCode = 1
    
```