SQL is the most popular language for adding, accessing and managing content in a database. It is most noted for its quick processing, proven reliability, ease and flexibility of use. **MySQL** is an essential part of almost every open source **PHP** application.

What is a database ?

A database is a separate application that stores a collection of data. Each database has one or more distinct APIs for creating, accessing, managing, searching and replicating the data it holds.

Nowadays, we use relational database management systems (RDBMS) to store and manage huge volume of data. This is called relational database because all the data is stored into different tables and relations are established using primary keys or other keys known as Foreign Keys.

SQL:

Structured Query Language (SQL) is a standard computer language for relational database management and data manipulation.

SQL is used to query, insert, update and modify data. Most relational databases support SQL, which is an added benefit for database administrators (DBAs), as they are often required to support databases across several different platforms.

MySQL is a freely available open source Relational Database Management System (RDBMS) that uses Structured Query Language (**SQL**).

RDBMS:

A Relational DataBase Management System (RDBMS) is a software that −

- Enables you to implement a database with tables, columns and indexes.
- Guarantees the Referential Integrity between rows of various tables.
- Updates the indexes automatically.
- Interprets an SQL query and combines information from various tables.

RDBMS Terminology:

Before we proceed to explain the MySQL database system, let us revise a few definitions related to the database.

- **Database** − A database is a collection of tables, with related data.
- **Table** − A table is a matrix with data. A table in a database looks like a simple spreadsheet.
- **Column** − One column (data element) contains data of one and the same kind, for example the column postcode.
- **Row** − A row (= tuple, entry or record) is a group of related data, for example the data of one subscription.
- **Redundancy** − Storing data twice, redundantly to make the system faster.
- **Primary Key** − A primary key is unique. A key value can not occur twice in one table. With a key, you can only find one row.
- **Foreign Key** − A foreign key is the linking pin between two tables.
- **Compound Key** − A compound key (composite key) is a key that consists of multiple columns, because one column is not sufficiently unique.
- **Index** − An index in a database resembles an index at the back of a book.
- **Referential Integrity** − Referential Integrity makes sure that a foreign key value always points to an existing row.

Types of SQL Commands

The following sections discuss the basic categories of commands used in SQL to perform various functions. These functions include building database objects, manipulating objects, populating database tables with data, updating existing data in tables, deleting data, performing database queries, controlling database access, and overall database administration.

The main categories are

- DDL (Data Definition Language)
- DML (Data Manipulation Language)
- DQL (Data Query Language)
- DCL (Data Control Language)

- Data administration commands
- Transactional control commands

Defining Database Structures

Data Definition Language, DDL, is the part of SQL that allows a database user to create and restructure database objects, such as the creation or the deletion of a table.

Some of the most fundamental DDL commands discussed during following hours include the following:

CREATE TABLE

ALTER TABLE

DROP TABLE

CREATE INDEX

ALTER INDEX

DROP INDEX

CREATE VIEW

DROP VIEW

Manipulating Data

Data Manipulation Language, DML, is the part of SQL used to manipulate data within objects of a relational database.

There are three basic DML commands:

INSERT

UPDATE

DELETE

These commands are discussed in detail during Hour 5, "Manipulating Data."

Selecting Data

Though comprised of only one command, Data Query Language (DQL) is the most concentrated focus of SQL for modern relational database users. The base command is as follows:

SELECT

This command, accompanied by many options and clauses, is used to compose queries against a relational database. Queries, from simple to complex, from vague to specific, can be easily created. The SELECT command is discussed in exhilarating detail during Hours 7 through 16.

A *query* is an inquiry to the database for information. A query is usually issued to the database through an application interface or via a command line prompt.

Data Control Language

Data control commands in SQL allow you to control access to data within the database. These DCL commands are normally used to create objects related to user access and also control the distribution of privileges among users. Some data control commands are as follows:

ALTER PASSWORD

GRANT

REVOKE

CREATE SYNONYM

You will find that these commands are often grouped with other commands and may appear in a number of different lessons throughout this book.

Data Administration Commands

Data administration commands allow the user to perform audits and perform analyses on operations within the database. They can also be used to help analyze system performance. Two general data administration commands are as follows:

START AUDIT

STOP AUDIT

Do not get data administration confused with database administration. *Database administration* is the overall administration of a database, which envelops the use of all levels of commands. *Database administration* is much more specific to each SQL implementation than are those core commands of the SQL language.

Transactional Control Commands

In addition to the previously introduced categories of commands, there are commands that allow the user to manage database transactions.

- COMMIT Saves database transactions
- ROLLBACK Undoes database transactions
- SAVEPOINT Creates points within groups of transactions in which to ROLLBACK
- SET TRANSACTION Places a name on a transaction

Key :

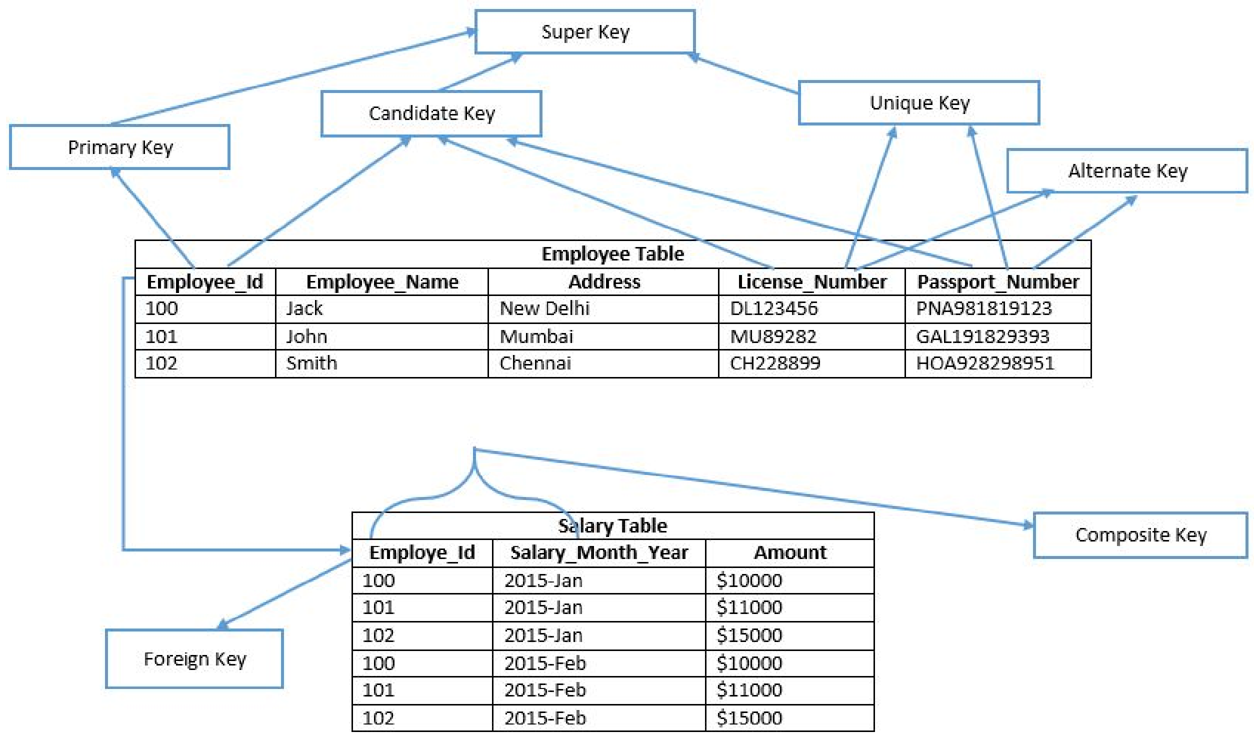
Keys are fields in a table which participate in below activities in RDBMS systems:

1. To create relationships between two tables.
2. To maintain uniqueness in a table.
3. To keep consistent and valid data in database.
4. Might help in fast data retrieval by facilitating indexes on column(s).

SQL Server supports various types of keys, which are listed below:

1. Candidate Key
2. Primary Key
3. Unique Key
4. Alternate Key
5. Composite Key
6. Super Key
7. Foreign Key

Before discussing each type in brief, have a look on the below image used as an an example to define types of keys



Employee Table				
Employee_Id	**Employee_Name**	**Address**	**License_Number**	**Passport_Number**
100	Jack	New Delhi	DL123456	PNA981819123
101	John	Mumbai	MU89282	GAL191829393
102	Smith	Chennai	CH228899	HOA928298951

Salary Table		
Employe_Id	**Salary_Month_Year**	**Amount**
100	2015-Jan	$10000
101	2015-Jan	$11000
102	2015-Jan	$15000
100	2015-Feb	$10000
101	2015-Feb	$11000
102	2015-Feb	$15000

Lets discuss each type in detail:

Candidate Key

Candidate key is a key of a table which can be selected as a primary key of the table. A table can have multiple candidate keys, out of which one can be selected as a primary key.

Example: Employee_Id, License_Number and Passport_Number are candidate keys

Primary Key

Primary key is a candidate key of the table selected to identify each record uniquely in table. Primary key does not allow null value in the column and keeps unique values throughout the column. In above example, Employee_Id is a primary key of Employee table. In SQL Server, by default primary key creates a clustered index on a heap tables (a table which does not have a clustered index is known as a heap table). We can also define a nonclustered primary key on a table by defining the type of index explicitly.

A table can have only one primary key and primary key can be defined in SQL Server using below SQL statements:

1. CRETE TABLE statement (at the time of table creation) – In this case, system defines the name of primary key

2. ALTER TABLE statement (using a primary key constraint) – User defines the name of the primary key

Example: Employee_Id is a primary key of Employee table.

Unique Key

Unique key is similar to primary key and does not allow duplicate values in the column. It has below differences in comparison of primary key:

1. It allows one null value in the column.
2. By default, it creates a nonclustered index on heap tables.

Alternate Key

Alternate key is a candidate key, currently not selected as primary key of the table.

Example: License_Number and Passport_Number are alternate keys.

Composite Key

Composite key (also known as compound key or concatenated key) is a group of two or more columns that identifies each row of a table uniquely. Individual column of composite key might not able to uniquely identify the record. It can be a primary key or candidate key also.

Example: In salary table, Employee_Id and Salary_Month_Year are combined together to identify each row uniquely in Salary table. Independently Employee_Id or Salary_Month_Year column cannot identify each row uniquely. We can create a composite primary key on Salary table using Employee_Id and Salary_Month_Year columns.

Super Key

Super key is a set of columns on which all columns of the table are functionally dependent. It is a set of columns that uniquely identifies each row in a table. Super key may hold some additional columns which are not strictly required to uniquely identify each row. Primary key and candidate keys are minimal super keys or you can say subset of super keys.

In above example, In Employee table, column Employee_Id is sufficient to uniquely identify any row of the table, so that any set of column from Employee table which contains Employee_Id is a super key for Employee Table.

For example: {Employee_Id}, {Employee_Id, Employee_Name}, {Employee_Id, Employee_Name, Address} etc.

License_Number and Passport_Number columns can also identify any row of the table uniquely. Any set of column which contains License_Number or Passport_Number or Employee_Id is a super key of the table.

For example: {License_Number, Employee_Name, Address}, {License_Number, Employee_Name, Passport_Number}, {Passport_Number, Employee_Name, Address, License_Number}, {Passport_Number, Employee_Name}, {Passport_Number, Employee_Id} etc.

Foreign Key

In a relationship between two tables, a primary key of one table is referred as a foreign key in another table. Foreign key can have duplicate values in it and can also keep null values if column is defined to accept nulls.

Example: Employee_Id (primary key of Employee table) is a foreign key in Salary table.

Commands:

```
$ mysql -u root -p
```

```
mysql> INSERT INTO authors (id,name,email) VALUES(3,"Tom","tom@yahoo.com");
```

To create database

```
create database
[databasename];
```

To view all database

```
show
databases;
```

To use particular database

```
use [db
name];
```

To view the list of tables in a database

```
show
```

```
tables;
```

To create a table

```
create table [table_name] (personid int(50) not null auto_increment primary key,
firstname varchar(35), middlename varchar(50), lastname varchar(50) default
‘bato’);
```

To enter records

```
Insert into table_name (col1, col2, col3 ) values (val1, val2, val3);
```

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:

```
Insert into table_name values (value1 , value2, value 3 , …. ) ;
```

To insert multiple records ,

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

To view the table along with data type

```
describe [table
name];
```

To delete a database

```
drop database [database
name];
```

To delete table

```
drop table [table
name];
```

To rename a table :

```
rename table old_name to new_name
```

To select entire table

```
SELECT * FROM [table
name];
```

To display columns in a table

```
show columns from [table
name];
```

To select particular record

```
SELECT * FROM [table name] WHERE [field name] =
"whatever";
```

To use multiple conditions using AND operator

```
SELECT * FROM [table name] WHERE name = "Bob" AND phone_number =
'3444444';
```

To order the records based on the column

```
SELECT * FROM [table name] WHERE name != "Bob" AND phone_number =
'3444444' order by phone_number;
```

To match the name which starts with “Bob” using % operator

```
SELECT * FROM [table name] WHERE name like "Bob%" AND phone_number =
'3444444';
```

To select the distinct values

```
SELECT DISTINCT [column name] FROM [table
name];
```

To order the records in a descending order based on the column

```
SELECT [col1],[col2] FROM [table name] ORDER BY [col2]
DESC;
```

To count number of records

```
SELECT COUNT(*) FROM [table
name];
```

To update the value in a table:

```
UPDATE [table name] SET field name1=”xyz” , field name2 =”abc”   where [field
name] = 'user';
```

To delete a row:

```
DELETE from [table name] where [field name] =
'whatever';
```

To delete all rows in a table

```
DELETE from [table name] ;
```

To add a new column:

```
alter table [table name] add column [new column name]
varchar (20);
```

To change the column name:

```
alter table [table name] change [old column name] [new column name]
```

```
varchar (50);
```

To drop the column:

```
alter table [table name] drop column [column
name];
```

To Change the column size:

```
alter table [table name] modify [column name]
VARCHAR(3
);
```

To make the column unique:

```
alter table [table name] add unique ([column
name]);
```

Delete unique :

```
alter table [table name] drop index [colmn
name];
```

Ref : http://g2pc1.bu.edu/~qzpeng/manual/MySQL%20Commands.htm

Limit clause:

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 integers.

```
SELECT customernumber,
       customername,
       creditlimit
FROM customers
ORDER BY  creditlimit DESC
LIMIT 5;

SELECT customernumber,
       customername,
       creditlimit
FROM customers
```

```
ORDER BY  creditlimit DESC
LIMIT 2 , 5;
```

Limit 5 represent 5 records should be displayed. Limit 2,5 represent minimum of 2 records and maximum upto 5 records.

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.

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

(or)

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

MySQL IN with subquery

The IN operator is often used with a subquery. Instead of providing a list of constant values, the subquery provides the list of values

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

Here, the subquery returns a list of order numbers that have total greater than 60000 using the GROUP BY and HAVING clauses on the orderDetails table.

Is equal to

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

Is equal to

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

BETWEEN Operator

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

Suppose you want to find products whose buy prices are within the ranges of $90 and $100, you can use the BETWEEN operator as the following query:

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

Is equal to

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

'NOT BETWEEN'

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

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

Is equal to

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

LIKE Operator:

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.

```
SELECT
   employeeNumber, lastName, firstName
FROM
   employees
WHERE
```

```
    firstName LIKE 'a%';
```

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

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

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

ORDER BY clause

When you use the SELECT statement to query data from a table, the result set is not sorted in any orders. To sort the result set, you use the ORDER BY clause. The ORDER BY clause allows you to:

1. Sort a result set by a single column or multiple columns.
2. Sort a result set by different columns in ascending or descending order.

```
SELECT
    contactLastname, contactFirstname
FROM
    customers
ORDER BY contactLastname;
```

```
SELECT
    contactLastname, contactFirstname
FROM
```

```
    customers
ORDER BY contactLastname DESC;
```

```
SELECT
    contactLastname, contactFirstname
FROM
    customers
ORDER BY contactLastname DESC , contactFirstname ASC;
```

In the query above, the ORDER BY clause sorts the result set by the last name in descending order first and then sorts the sorted result set by the first name in ascending order to produce the final result set.

```
SELECT
    ordernumber, orderlinenumber, quantityOrdered * priceEach
FROM
    orderdetails
ORDER BY ordernumber , orderLineNumber, quantityOrdered * priceEach;
```

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:

```
SELECT
    orderNumber, status
FROM
    orders
ORDER BY FIELD(status,
        'In Process',
        'On Hold',
        'Cancelled',
        'Resolved',
        'Disputed',
        'Shipped');
```

MySQL Alias:

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

MySQL alias for columns

Sometimes the names of columns are so technical that make the query’s output very difficult to understand.

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

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

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

MySQL alias for tables

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

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.

```
SELECT
   customers.customerName, COUNT(orders.orderNumber) total
```

```
FROM
   customers
      INNER JOIN
   orders ON customers.customerNumber = orders.customerNumber
GROUP BY customerName
ORDER BY total DESC
```

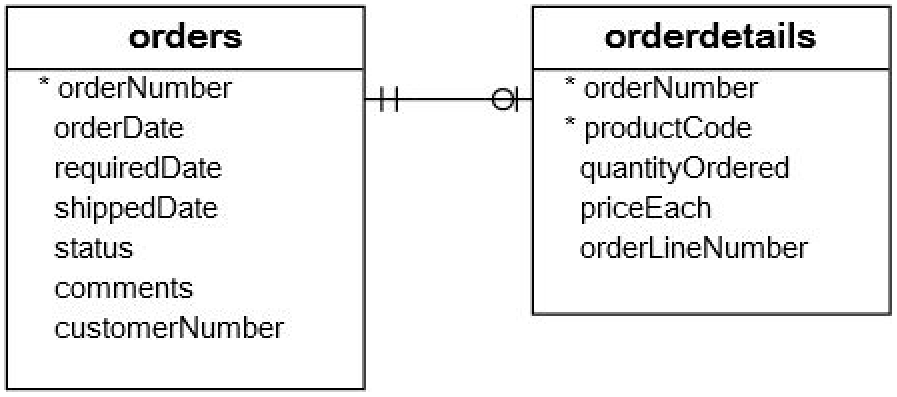
Is equal to

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

MySQL join statements

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

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



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

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

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

MySQL supports the following types of joins:

1. Cross join
2. Inner join
3. Left join
4. Right join

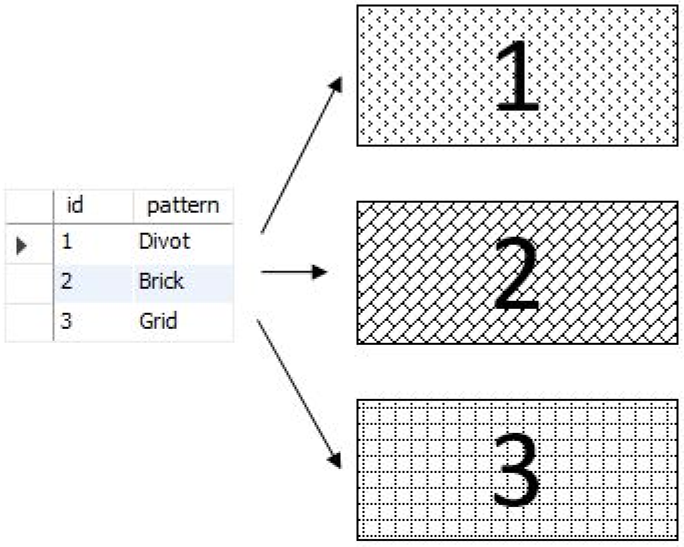
To join tables, you use the CROSS JOIN, INNER JOIN, LEFT JOIN or RIGHT JOIN clause for the corresponding type of join. The join clause is used in the SELECT statement appeared after the FROMclause.

Notice that MySQL does not support full outer join.

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

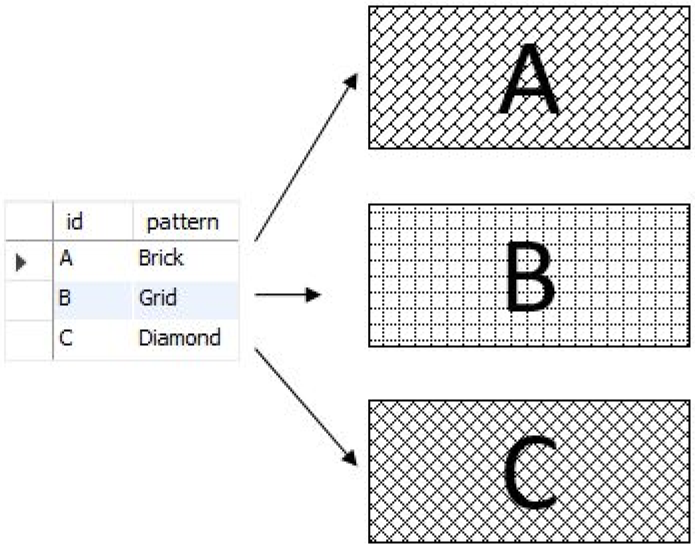
```
CREATE TABLE t2 (
   id VARCHAR(50) PRIMARY KEY,
   pattern VARCHAR(50) NOT NULL
);
```

Below illustrate data from both t1 and t2 tables:



id	pattern
1	Divot
2	Brick
3	Grid

t1



id	pattern
A	Brick
B	Grid
C	Diamond

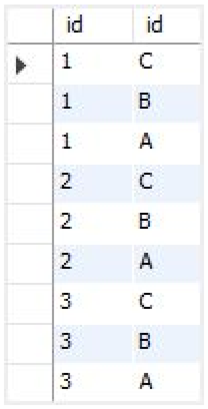
t2

MySQL CROSS JOIN

The CROSS JOIN makes a Cartesian product of rows from multiple tables. Suppose, you join t1 and t2 tables using the CROSS JOIN, the result set will include the combinations of rows from the t1table with the rows in the t2 table.

```
SELECT
   t1.id, t2.id
FROM
   t1
```

The following shows the result set of the query:



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

MySQL INNER JOIN

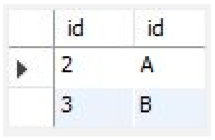
To form anINNER JOIN, you need a condition which is known as a join-predicate. An INNER JOINrequires rows in the two joined tables to have matching column values. The INNER JOIN creates the result set by combining column values of two joined tables based on the join-predicate.

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

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

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

The following illustrates the result of the query:



id	id
2	A
3	B

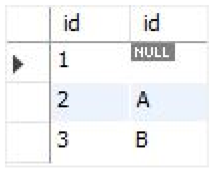
MySQL LEFT JOIN

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

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

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

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



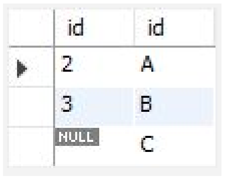
id	id
1	NULL
2	A
3	B

MySQL RIGHT JOIN

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

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

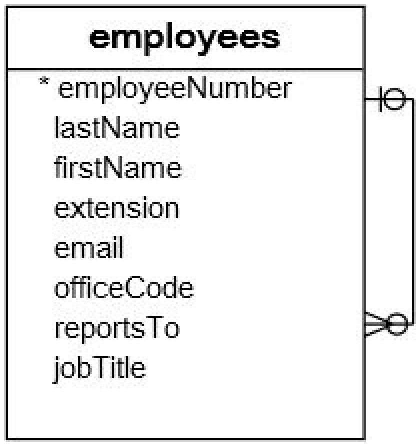
```
SELECT
   t1.id, t2.id
FROM
   t1
      RIGHT JOIN
   t2 on t1.pattern = t2.pattern
ORDER BY t2.id;
```



	id	id
▶	2	A
	3	B
	NULL	C

MySQL self join examples

In the employees table, we store not only employees data but also organization structure data. The reports to column is used to determine the manager id of an employee



.

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

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

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

GROUP BY clause

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

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

```
SELECT status
From orders
GROUP BY status;
```

Is equal to

```
SELECT DISTINCT status

FROM orders;
```

GROUP BY with aggregate functions

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

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

```
SELECT status, COUNT(*)

FROM  orders

GROUP BY status;
```

```
SELECT orderNumber, SUM(quantityOrdered * priceEach) AS total
```

```
FROM orderdetails

GROUP BY orderNumber;
```

GROUP BY with HAVING clause

To filter the groups returned by GROUP BY clause, you use a [HAVING](#) clause.

```
SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country

HAVING COUNT(CustomerID) > 5;
```

The GROUP BY clause: MySQL vs. standard SQL

Standard SQL does not allow you to use an alias in the GROUP BY clause, however, MySQL supports this. The following query extracts the year from the order date and counts the orders per year. The year is used as an alias of the expression YEAR(orderDate) and it is used as an alias in the GROUP BYclause too. This query is invalid in standard SQL.

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

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

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

MySQL HAVING clause

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

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

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

```
SELECT ordernumber, SUM(quantityOrdered) AS itemsCount,  SUM(quantityOrdered*priceeach)
AS total

FROM orderdetails
GROUP BY ordernumber
HAVING total > 1000;
```

Sub Query:

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

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

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

subquery with EXISTS and NOT EXISTS

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

```
SELECT customerName
```

```
FROM customers
WHERE EXISTS (
   SELECT priceEach * quantityOrdered
   FROM orderdetails
   WHERE priceEach * quantityOrdered > 10000
   GROUP BY orderNumber
)
```

MySQL UNION operator

MySQL UNION operator allows you to combine two or more result sets of queries into a single result set.

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

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

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

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

CREATE TABLE t2 (
    id INT PRIMARY KEY
);

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

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

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

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

```
Id 1,2,3,4
```

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

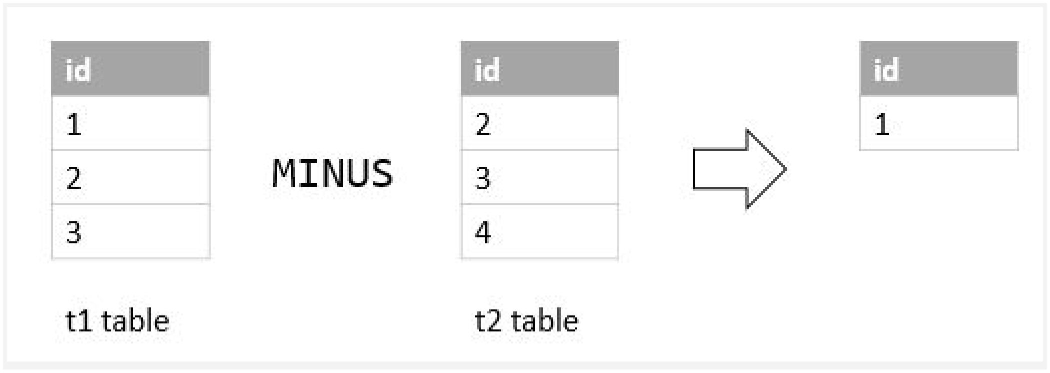
```
Id 1,2,3,2,3,4
```

SQL MINUS operator

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

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

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



SQL INTERSECT operator

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

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

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

Copy data from one table to another:

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

```
CREATE TABLE tasks_1 LIKE tasks;
```

Second, insert data from the tasks table into the tasks_1 table using the following INSERT statement:

```
INSERT INTO tasks_1
SELECT * FROM tasks;
```

LAST_INSERT_ID ()

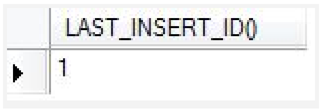
```
CREATE TABLE tbl (
    id INT AUTO_INCREMENT PRIMARY KEY,
    description VARCHAR(250) NOT NULL
);
```

Second, insert a new row into the tbl table.

```
INSERT INTO tbl(description) VALUES('MySQL last_insert_id');
```

Third, use the MySQL LAST_INSERT_ID function to get the last insert id that MySQL has been generated.

```
SELECT LAST_INSERT_ID();
```



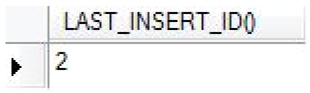
```
LAST_INSERT_ID()
1
```

It's important to note that if you insert multiple rows into a table using a single INSERT statement, the LAST_INSERT_ID function returns the last insert id of the first row.

Suppose the AUTO_INCREMENT column has current value as 1 and you insert 3 rows into the table. When you use the LAST_INSERT_ID function to get the last insert id, you will get 2 instead of 4.

```
INSERT INTO tbl(description)
VALUES('record 1'),
    ('record 2'),
    ('record 3');
```

```
SELECT LAST_INSERT_ID();
```



```
LAST_INSERT_ID()
2
```

Primary key :

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

1. 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.
2. A primary key column cannot contain NULL values. It means that you have to declare the primary key column with the NOT NULL attribute. If you don't, MySQL will force the primary key column as NOT NULL implicitly.
3. A table has only one primary key.

Because MySQL works faster with integers, the data type of the primary key column should be the integer e.g., INT, BIGINT.You can choose a smaller integer type: 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 the AUTO_INCREMENT attribute that generates a unique sequence for the key automatically. The primary key of the next row is greater than the previous one.

```
CREATE TABLE users(
  user_id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(40),
  password VARCHAR(255),
  email VARCHAR(255)
);
```

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

```
CREATE TABLE roles(
  role_id INT AUTO_INCREMENT,
  role_name VARCHAR(50),
  PRIMARY KEY(role_id)
);
```

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.

```
CREATE TABLE userroles(
  user_id INT NOT NULL,
  role_id INT NOT NULL,
  PRIMARY KEY(user_id,role_id),
  FOREIGN KEY(user_id) REFERENCES users(user_id),
  FOREIGN KEY(role_id) REFERENCES roles(role_id)
);
```

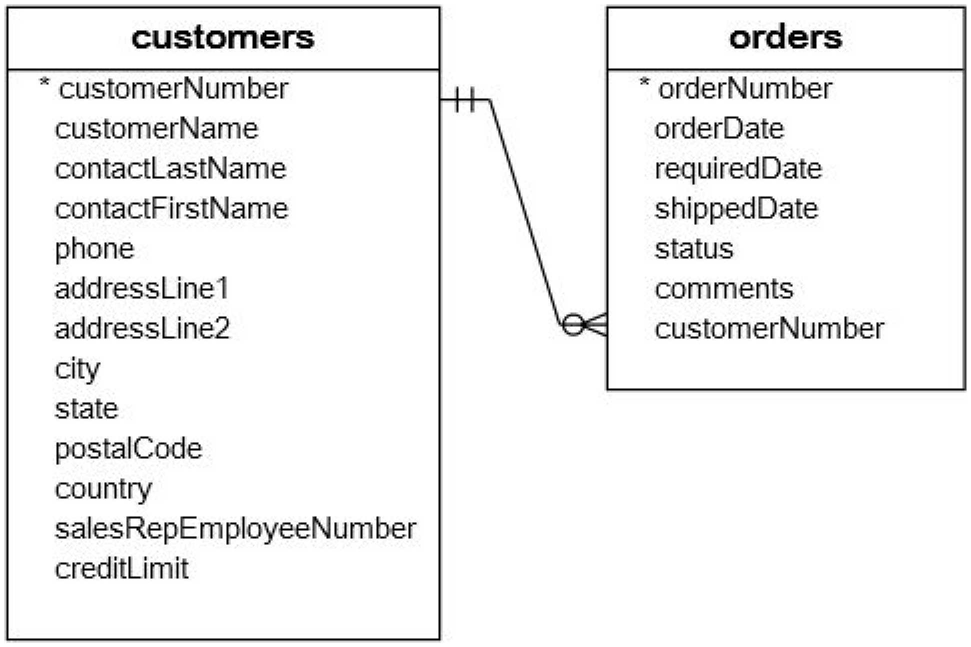
Defining MySQL PRIMARY KEY constraints using ALTER TABLE statement

ALTER TABLE table_name ADD PRIMARY KEY(primary_key_column);

Foreign key

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

Let’s take a look at the following database diagram



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 the customers table.

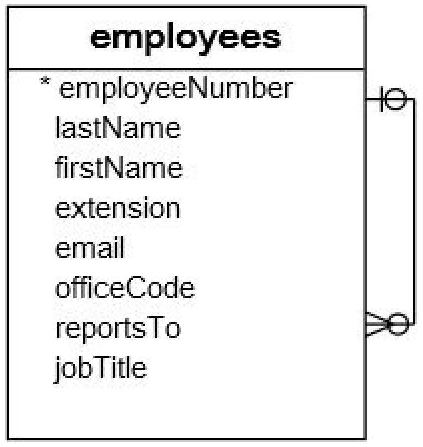
The customers table is called *parent table* or *referenced table*, and the orders table is known as *child table* or *referencing table*.

A foreign key can be a column or 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 may refer to 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 tables are the same. 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. We have a specific tutorial on the self-join to help you query data against this kind of table.

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 the 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.

```
Create table employee(empId int(10) not null auto_increment primary key, firstName varchar(50)
, lastName varchar(50), status varchar(50), salary int(10) );
```

```
create table skill_set (id int not null auto_increment primary key, language varchar(50),
expert_level varchar(50), empId int ,foreign key (empId) references employee(empId) on delete
cascade );
```

```
insert into skill_set(language,expert_level,empId) values("c,c++","intermediate",102);
```

```
 insert into skill_set(language,expert_level,empId)
values("java","beginner",103),("html","advanced",104),("python","beginner",107);
```

Dropping foreign key:

```
alter table table_name drop foreign key key_name;
```

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

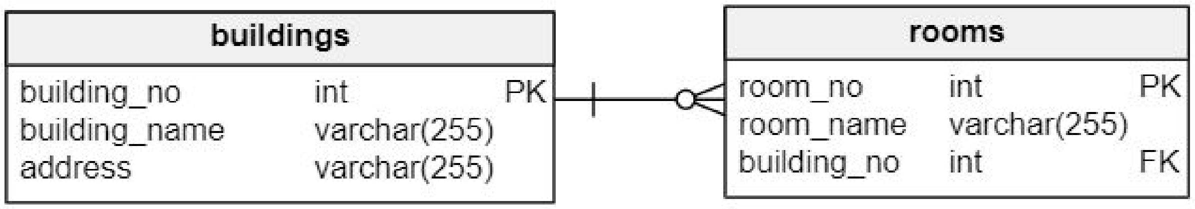
```
show create table table_name;
```

MySQL ON DELETE CASCADE example

Let's take a look at an example of using MySQL ON DELETE CASCADE .

Suppose we have two tables:buildings and rooms . In this database model, each building has one or more rooms. However, each room belongs to one only one building. A room would not exist without a building.

The relationship between the buildings and rooms tables is one-to-many (1:N) as illustrated in the following database diagram:



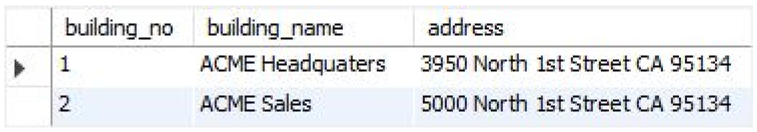
When we delete a row from the buildings table, we also want to delete the rows in the rooms table that references to the rows in the buildings table. For example, when we delete a row with building no. 2 in the buildings table as the following query:

```
delete from buildings where building_no=2;
```

We want the rows in the rooms table that refers to building number 2 will be also removed.

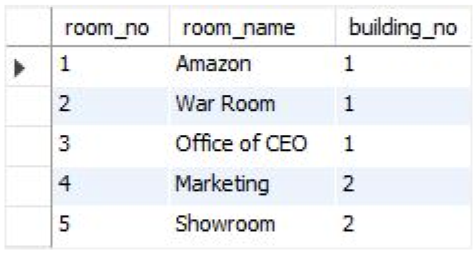
The following are steps that demonstrate how MySQL ON DELETE CASCADE referential action works.

```
CREATE TABLE buildings (
    building_no INT PRIMARY KEY AUTO_INCREMENT,
    building_name VARCHAR(255) NOT NULL,
    address VARCHAR(255) NOT NULL
);

CREATE TABLE rooms (
    room_no INT PRIMARY KEY AUTO_INCREMENT,
    room_name VARCHAR(255) NOT NULL,
    building_no INT NOT NULL,
    FOREIGN KEY (building_no)
        REFERENCES buildings (building_no)
        ON DELETE CASCADE
);

INSERT INTO buildings(building_name,address)
VALUES('ACME Headquaters','3950 North 1st Street CA 95134'),
      ('ACME Sales','5000 North 1st Street CA 95134');
```



building_no	building_name	address
1	ACME Headquaters	3950 North 1st Street CA 95134
2	ACME Sales	5000 North 1st Street CA 95134

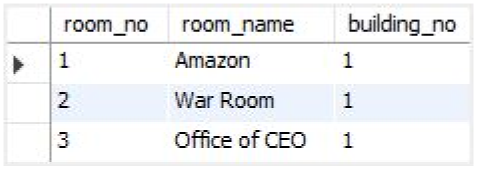
```
INSERT INTO rooms(room_name,building_no)
VALUES('Amazon',1),
      ('War Room',1),
      ('Office of CEO',1),
      ('Marketing',2),
      ('Showroom',2);
```



room_no	room_name	building_no
1	Amazon	1
2	War Room	1
3	Office of CEO	1
4	Marketing	2
5	Showroom	2

Now delete the building with building no. 2:

```
DELETE FROM buildings
WHERE building_no = 2;
```



	room_no	room_name	building_no
▶	1	Amazon	1
	2	War Room	1
	3	Office of CEO	1

Notice that ON DELETE CASCADE works only with tables with the storage engines support foreign keys e.g., InnoDB. Some table types do not support foreign keys such as MyISAM so you should choose appropriate storage engines for the tables that you plan to use the MySQL ON DELETE CASCADE referential action.

MySQL DELETE JOIN with INNER JOIN

MySQL also allows you to use the INNER JOIN clause in the DELETE statement to delete rows from a table and the matching rows in another table.

For example, to delete rows from both T1 and T2 tables that meet a specified condition, you use the following statement:

```
DELETE T1, T2
FROM T1
INNER JOIN T2 ON T1.key = T2.key
WHERE condition;
```

Notice that you put table names T1 and T2 between the DELETE and FROM keywords. If you omit T1 table, the DELETE statement only deletes rows in T2 table. Similarly, if you omitT2 table, the DELETE statement will delete only rows in T1 table.

The expression T1.key = T2.key specifies the condition for matching rows between T1 andT2tables that will be deleted.

The condition in the WHERE clause determine rows in the T1 and T2 that will be deleted.

MySQL DELETE JOIN with INNER JOIN example

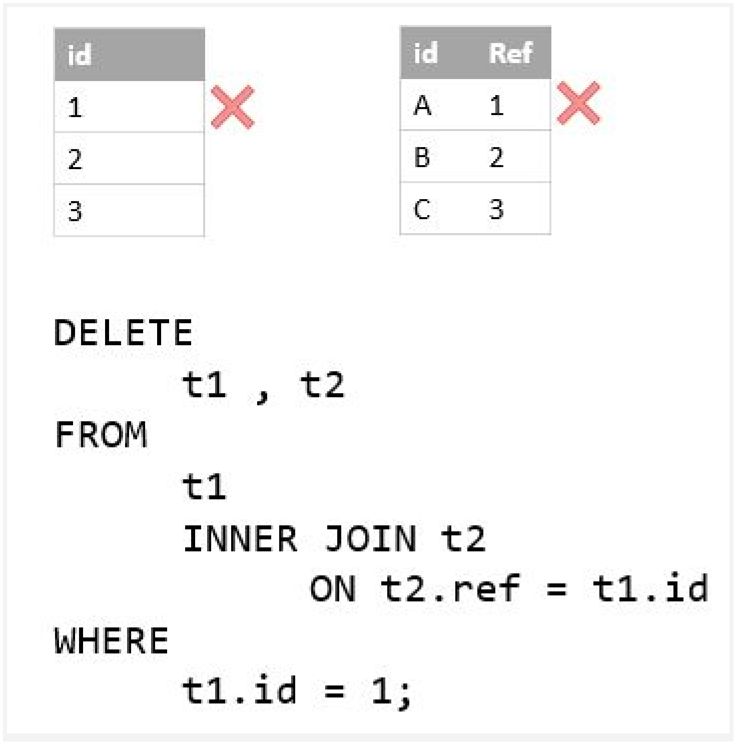
Suppose, we have two tables t1 and t2 with the following structures and data:

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

```
CREATE TABLE t2 (
   id VARCHAR(20) PRIMARY KEY,
   ref INT NOT NULL
);
```

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

```
INSERT INTO t2(id,ref) VALUES('A',1),('B',2),('C',3);
```



id
1
2
3

id	Ref
A	1
B	2
C	3

```
DELETE
      t1 , t2
FROM
      t1
      INNER JOIN t2
            ON t2.ref = t1.id
WHERE
      t1.id = 1;
```

The following statement deletes the row with id 1 in the t1 table and also row with ref 1 in the t2table using DELETE...INNER JOIN statement:

```
DELETE t1,t2 FROM t1
      INNER JOIN
   t2 ON t2.ref = t1.id
WHERE
   t1.id = 1;
```

The statement returned the following message:

It indicated that two rows have been deleted.

MySQL DELETE JOIN with LEFT JOIN

We often use the LEFT JOIN clause in the SELECT statement to find rows in the left table that have or don't have matching rows in the right table.

We can also use the LEFT JOIN clause in the DELETE statement to delete rows in a table (left table) that does not have matching rows in another table (right table).

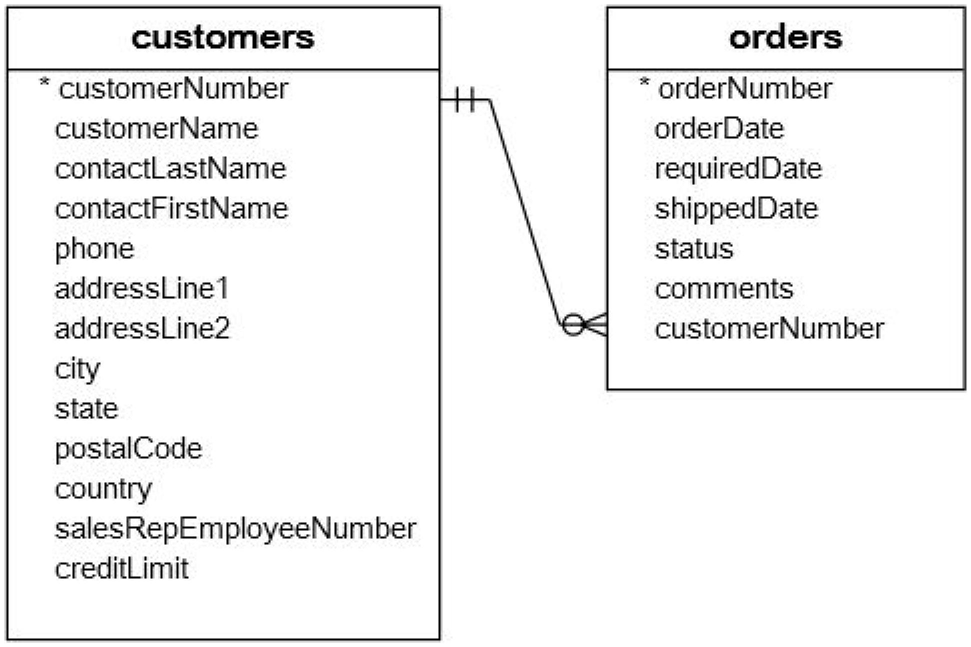
The following syntax illustrates how to use DELETE statement with LEFT JOIN clause to delete rows from T1 table that does not have corresponding rows in the T2 table:

```
DELETE T1
FROM T1
      LEFT JOIN
   T2 ON T1.key = T2.key
WHERE
   T2.key IS NULL;
```

Note that we only put T1 table after the DELETE keyword, not both T1 and T2 tables like we did with the INNER JOIN clause.

MySQL DELETE JOIN with LEFT JOIN example

See the following customers and orders tables in the sample database:



Each customer has zero or more orders. However, each order belongs to one and only one customer.

We can use DELETE statement with LEFT JOIN clause to clean up our customers master data. The following statement removes customers who have not placed any order:

```
DELETE customers
FROM customers
      LEFT JOIN
   orders ON customers.customerNumber = orders.customerNumber
WHERE
   orderNumber IS NULL;
```

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.

Storage engines (underlying software component) are MySQL components, that can handle the SQL operations for different table types to store and manage information in a database. InnoDB is mostly used general-purpose storage engine and as of MySQL 5.5 and later it is the default engine. There are many storage engines available in MySQL and they are used for different purposes.

MySQL provides various storage engines for its tables as below:

1. MyISAM
2. InnoDB
3. MERGE
4. MEMORY (HEAP)
5. ARCHIVE
6. CSV
7. 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 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 operating systems.

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 spaces. At startup, MySQL checks MyISAM tables for corruption and even repairs them in a 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 specifying the storage engine explicitly. 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 optimal for performance. InnoDB table supports foreign keys, commit, rollback, roll-forward operations. The size of an InnoDB table can be up to 64TB.

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

Ref : http://www.mysqltutorial.org/understand-mysql-table-types-innodb-myisam.aspx

----

Stored procedure:

A stored procedure is a segment of declarative SQL statements stored inside the database catalog. A stored procedure can be invoked by triggers, other stored procedures, and applications such as Java, Python, PHP.

A stored procedure that calls itself is known as a recursive stored procedure. Most database management systems support recursive stored procedures. However, MySQL does not support it very well.

MySQL stored procedures advantages :

1. Typically stored procedures help increase the performance of the applications. Once created, stored procedures are compiled and stored in the database. However, MySQL implements the stored procedures slightly different. MySQL stored procedures are compiled on demand. After compiling a stored procedure, MySQL puts it into a cache. And MySQL maintains its own stored procedure cache for every single connection. If an application uses a stored procedure multiple times in a single connection, the compiled version is used, otherwise, the stored procedure works like a query.

2. Stored procedures help reduce the traffic between application and database server because instead of sending multiple lengthy SQL statements, the application has to send only name and parameters of the stored procedure.

3. Stored procedures are reusable and transparent to any applications. Stored procedures expose the database interface to all applications so that developers don’t have to develop functions that are already supported in stored procedures.

4. Stored procedures are secure. The database administrator can grant appropriate permissions to applications that access stored procedures in the database without giving any permissions on the underlying database tables.

Besides those advantages, stored procedures have their own disadvantages, which you should be aware of before using them in your databases.

MySQL stored procedures disadvantages :

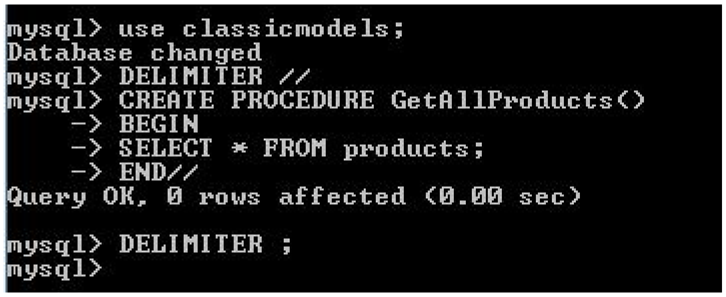
1. If you use many stored procedures, the memory usage of every connection that is using those stored procedures will increase substantially. In addition, if you overuse a large number of logical operations inside store procedures, the CPU usage will also increase because the database server is not well-designed for logical operations.

2. Stored procedure’s constructs are not designed for developing complex and flexible business logic.

3. It is difficult to debug stored procedures. Only a few database management systems allow you to debug stored procedures. Unfortunately, MySQL does not provide facilities for debugging stored procedures.

4. It is not easy to develop and maintain stored procedures. Developing and maintaining stored procedures are often required a specialized skill set that not all application developers possess. This may lead to problems in both application development and maintenance phases.

Writing the first MySQL stored procedure



```
mysql> use classicmodels;
Database changed
mysql> DELIMITER //
mysql> CREATE PROCEDURE GetAllProducts()
    -> BEGIN
    -> SELECT * FROM products;
    -> END//
Query OK, 0 rows affected (0.00 sec)

mysql> DELIMITER ;
mysql>
```

Calling stored procedures

In order to call a stored procedure, you use the following SQL command:

```
CALL GetAllProducts();
```

Declaring variables

To declare a variable inside a stored procedure, you use the DECLARE statement as follows:

```
 DECLARE total_sale INT DEFAULT 0;
DECLARE x, y INT DEFAULT 0;
```

Assigning variables

Once you declared a variable, you can start using it. To assign a variable another value, you use the SET statement, for example:

```
DECLARE total_count INT DEFAULT 0;
SET total_count = 10;
```

Variables scope

A variable has its own scope that defines its lifetime. If you declare a variable inside a stored procedure, it will be out of scope when the END statement of stored procedure reached.

-

Trigger:

A SQL trigger is a set of SQL statements stored in the database catalog. A SQL trigger is executed or fired whenever an event associated with a table occurs e.g., insert, update or delete.

A SQL trigger is a special type of stored procedure. It is special because it is not called directly like a stored procedure. The main difference between a trigger and a stored procedure is that a trigger is called automatically when a data modification event is made against a table whereas a stored procedure must be called explicitly.

A trigger can be defined to be invoked either before or after the data is changed by INSERT, UPDATE or DELETE statement.

Before MySQL version 5.7.2, you can to define maximum six triggers for each table.

1. BEFORE INSERT – activated before data is inserted into the table.
2. AFTER INSERT – activated after data is inserted into the table.
3. BEFORE UPDATE – activated before data in the table is updated.
4. AFTER UPDATE – activated after data in the table is updated.
5. BEFORE DELETE – activated before data is removed from the table.
6. AFTER DELETE – activated after data is removed from the table.

However, from MySQL version 5.7.2+, you can define multiple triggers for the same trigger event and action time.

When you use a statement that does not use INSERT, DELETE or UPDATE statement to change data in a table, the triggers associated with the table are not invoked. For example, the TRUNCATE statement removes all data of a table but does not invoke the trigger associated with that table.

There are some statements that use the INSERT statement behind the scenes such as REPLACE statement or LOAD DATA statement. If you use these statements, the corresponding triggers associated with the table are invoked.

Advantage of triggers :

1. Triggers can be used as an alternative method to check the integrity of data.

2. Triggers are standard actions performed by more than one application program. By coding the action once and storing it in the database for future use, if we need to change the functionality or business logic you need to change only the corresponding trigger program instead of each application program.
3. SQL triggers provide an alternative way to run scheduled tasks. You don't have to wait to run the scheduled tasks because the triggers are invoked automatically before or after a change is made to the data in tables.
4. SQL triggers are useful to audit the changes of data in tables.
5. Triggers are also used for calling stored procedures.

Disadvantages of trigger:

1. Programmers don't have full control: Since business rules are hidden, programmers don't have full control on the database.
2. Decrease in performance of the database: By the complex nature of the database programs take more time to execute and there can be hidden performance downtimes.
3. SQL triggers may increase the overhead of the database server.

Trigger limitation:

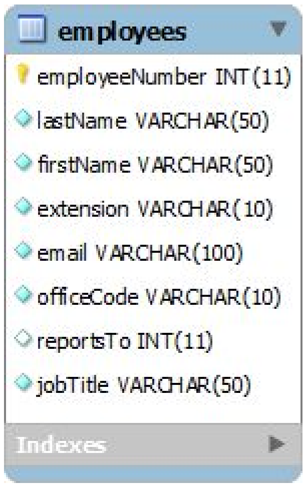
MySQL triggers cannot:

1. Use SHOW, LOAD DATA, LOAD TABLE, BACKUP DATABASE, RESTORE, FLUSH and RETURN statements.
2. Use statements that commit or rollback implicitly or explicitly such as COMMIT , ROLLBACK , START TRANSACTION , LOCK/UNLOCK TABLES , ALTER , CREATE , DROP , RENAME.
3. Use prepared statements such as PREPAREand EXECUTE.
4. Use dynamic SQL statements.

From MySQL version 5.1.4, a trigger can call a stored procedure or stored function, which was a limitation is the previous versions.

MySQL trigger example

Let's start creating a trigger in MySQL to log the changes of the employees table.



```
employeeNumber INT(11)
lastName VARCHAR(50)
firstName VARCHAR(50)
extension VARCHAR(10)
email VARCHAR(100)
officeCode VARCHAR(10)
reportsTo INT(11)
jobTitle VARCHAR(50)
```

First, create a new table named employees_audit to keep the changes of the employee table. The following statement creates the employee_audit table.

```
CREATE TABLE employees_audit (
   id INT AUTO_INCREMENT PRIMARY KEY,
   employeeNumber INT NOT NULL,
   lastname VARCHAR(50) NOT NULL,
   changedat DATETIME DEFAULT NULL,
   action VARCHAR(50) DEFAULT NULL
);
```

Next, create a BEFORE UPDATE trigger that is invoked before a change is made to the employees table.

```
DELIMITER $$
CREATE TRIGGER before_employee_update
   BEFORE UPDATE ON employees
   FOR EACH ROW
BEGIN
   INSERT INTO employees_audit
   SET action = 'update',
         employeeNumber = OLD.employeeNumber,
      lastname = OLD.lastname,
      changedat = NOW();
END$$
DELIMITER ;
```

Inside the body of the trigger, we used the OLD keyword to access employeeNumber and lastnamecolumn of the row affected by the trigger.

Notice that in a trigger defined for INSERT, you can use NEW keyword only. You cannot use the OLDkeyword. However, in the trigger defined for DELETE, there is no new row so you can use the OLDkeyword only. In the UPDATE trigger, OLD refers to the row before it is updated and NEW refers to the row after it is updated.

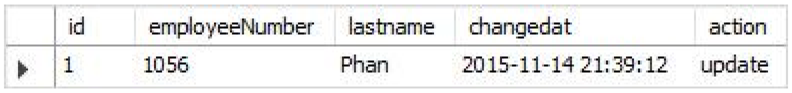
After that, update the employees table to check whether the trigger is invoked.

```
UPDATE employees
SET
   lastName = 'Phan'
WHERE
   employeeNumber = 1056;
```

Finally, to check if the trigger was invoked by the UPDATE statement, you can query the employees_audit table using the following query:

```
SELECT
   *
FROM
   employees_audit;
```

The following is the output of the query:



	id	employeeNumber	lastname	changedat	action
▶	1	1056	Phan	2015-11-14 21:39:12	update

As you see, the trigger was really invoked and it inserted a new row into the employees_audit table.

SQL vs No-SQL :

Ref : https://www.upwork.com/hiring/data/sql-vs-nosql-databases-whats-the-difference/