**Here will use mysql-connector-python library to connecting to mysql DB from pythons**

pip install mysql-connector-python

**Connecting to mysql database**

The **connect()** constructor from 'mysql.connector' package creates a connection to the MySQL server and **returns a MySQLConnection object**

*connect(user=?, password=?, port=?, database=?, \*\*kargs)*

import mysql.connector

from mysql.connector.errors import Error

try:

    cnx = mysql.connector.connect(user='root', password='admin',host='127.0.0.1',database='projectalpha')

    print(cnx)

except Error as err:

    print(e)

finally:

    cnx.close()

Output:

<mysql.connector.connection\_cext.CMySQLConnection object at 0x000001C0B5D088E0>

connect() can take below option argument

1. user/username ---- user name for connection to mysql server
2. password/passwd ---- password for connecting to mysql server
3. database(db) ------ database name to use when connecting to mysql server
4. host ---- default 127.0.0.1 ---- host ip addres where mysql server running
5. port --- default 3306 --- port of mysql server
6. autocommit ---- False --- whether to autocommit transaction
7. time\_zone ---- set the time\_zone session variable at connection time
8. buffered ---- if true then created cursor will be buffered cursor
9. raw= Boolean---default to False ---- if False then returned result set are automatically converted into python data types
10. commit=bool----- if true then created cursor will be auto commit.

**MySQLConnection class Methods**

* MySQLConnection.ping() Method
* MySQLConnection.reconnect() Method
* MySQLConnection.reset\_session() Method
* MySQLConnection.rollback() Method
* MySQLConnection.get\_row() Method
* MySQLConnection.get\_rows() Method
* MySQLConnection.get\_server\_info() Method
* MySQLConnection.get\_server\_version() Method
* MySQLConnection.is\_connected() Method

**MySQLConnection.ping() Method**

cnx.ping(reconnect=False, attempts=1, delay=0)

Check whether the connection to the MySQL server is still available.

When reconnect is set to True, one or more attempts are made to try to reconnect to the MySQL server, and these options are forwarded to the reconnect() method. Use the delay argument (seconds) if you want to wait between each retry.

**MySQLConnection.reconnect() Method**

cnx.reconnect(attempts=1, delay=0)

Attempt to reconnect to the MySQL server

The argument attempts specifies the number of times a reconnect is tried.

The delay argument is the number of seconds to wait between each retry

**MySQLConnection.reset\_session() Method**

cnx.reset\_session(user\_variables = None, session\_variables = None)

This method resets the session state by reauthenticating

**MySQLConnection.rollback() Method**

This method sends a ROLLBACK statement to the MySQL server, undoing all data changes from the current transaction. By default, Connector/Python does not autocommit, so it is possible to cancel transactions when using transactional storage engines such as InnoDB.

**MySQLConnection.is\_connected() Method**

Reports whether the connection to MySQL Server is available

This method checks whether the connection to MySQL is available using the ping() method, but unlike ping(), is\_connected() returns True when the connection is available, False otherwise

**MySQLConnection.get\_row() Method**

This method retrieves the next row of a query result set, returning a tuple.

**Cursor in mysql**

A cursor allows you to iterate a set of rows returned by a query and process each row individually.

*Cursor can be created using connection object*.

The MySQLCursor class instantiates objects that can execute operations such as SQL statements. Cursor objects interact with the MySQL server using a MySQLConnection object.

cursor = cnx.cursor([arg=value[, arg=value]...])

This method returns a MySQLCursor() object, or a subclass of it depending on the passed arguments. The returned object is a cursor.CursorBase instance.

Cursor are various types based on arguments passed to cursor() few of them are –

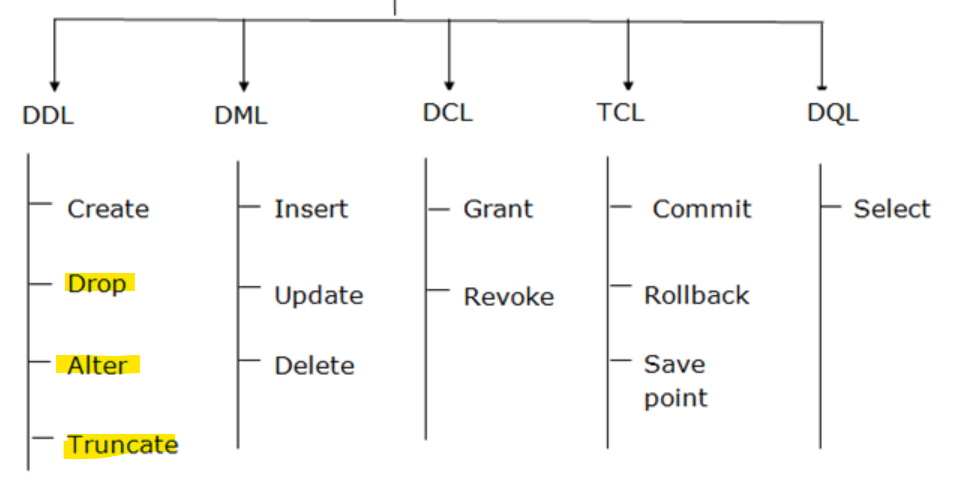
1. “cursor.MySQLCursorBuffered Class” ------ buffered = True
2. “cursor.MySQLCursorRaw Class” ------- raw=True( can be created during connect. creation)
3. “cursor.MySQLCursorBufferedRaw Class” --- buffered = True, raw= True
4. “cursor.MySQLCursorDict Class” --------- dictonary=True
5. “cursor.MySQLCursorNamedTuple Class” --- named\_tupe=True
6. “cursor.MySQLCursorBufferedDict Class” ---- dictionary=True, buffered=True
7. “cursor.MySQLCursorPrepared Class” ------ prepared=True

* If buffered is True, the cursor fetches all rows from the server after an operation is executed. This is useful when queries return small result sets.
* dictionary=True , then creates a MySQLCursorDict cursor that returns rows as dictionaries.
* named\_tuple=True, then creates a MySQLCursorNamedTuple cursor that returns rows as named tuples

**Types of SQL commands**

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL as-

* Data Definition Language (DDL)
* Data Manipulation Language (DML)
* Data Control Language(DCL)
* Transaction Control Language(TCL)
* Data Query Language (DQL)



**Note:**

* IF we execute DML statement is returns(by calling rowcount() ) number of rows effected
* If we execute DQL commands then it gives/returns the result set

**Executing query/sql command**

For executing query of any DB operation we can use **execute()** or **executemany()** based on requirements.

**execute(operation/sql\_query, params=None, multi=False)**

This method executes the given database operation (query or command).

The parameters found in the tuple or dictionary params are bound to the variables in the operation.

Specify variables using %s or %(name)s parameter style (that is, using format or pyformat style).

params ---- it specifies the data what want to insert. If fetching data it’s not required

*execute() returns an iterator if multi is True.*

cursor.execute(operation, params=None, multi=False)

iterator = cursor.execute(operation, params=None, multi=True)

**Note:**

*execute() can insert/update/select only one row at a time, for more rows use executemany()*

Example 1:

Insert data into woodshop\_employee table of projectalpha database, where data is in tuple

import mysql.connector

try:

    cnx = mysql.connector.connect(user='root', password='admin',host='127.0.0.1',database='projectalpha')

    cursor=cnx.cursor()

except Exception as e:

    print(e)

insert\_stmt =   "INSERT INTO woodshophome\_employee (id,first\_name, last\_name) VALUES (%s, %s, %s)"

data = (11,'Brush','for painting')

data=cursor.execute(insert\_stmt, data)

Example 2:

Data is in dictionary style

import mysql.connector

try:

    cnx = mysql.connector.connect(user='root', password='admin',host='127.0.0.1',database='projectalpha')

    cursor=cnx.cursor()

except Exception as e:

    print(e)

select\_stmt = "SELECT \* FROM employees WHERE emp\_no = %(emp\_no)s"

cursor.execute(select\_stmt, { 'emp\_no': 2 })

**executemany(operations/sql\_query, seq\_of\_params)**

This method prepares a database operation (query or command) and executes it against all parameter sequences or mappings found in the sequence seq\_of\_params/data.

*This method runs sql query against all set of seq\_of\_params or data that must be specified as list of tuplle*

operations --- this is the sql query

seq\_of\_data --- this is the data as list of tupple

Example1:

import mysql.connector

try:

    cnx = mysql.connector.connect(user='root', password='admin',host='127.0.0.1',database='projectalpha')

    cursor=cnx.cursor()

except Exception as e:

    print(e)

insert\_stmt =   "INSERT INTO woodshophome\_employee (id,first\_name, last\_name) VALUES (%s, %s, %s)"

data = [(12,'Brush','for painting'),(13,'Brush','for painting')]

cursor.executemany(insert\_stmt,data )

cnx.commit()

cursor.close()

cnx.close()

if we run the same query using execute() then we get error as there are multiple data against query

**Bind Variable**

When we need to execute the same query repeatedly, but we just want to substitute the values each time, then we can use bind variables. They improve performance.

Example: This is execute for each value

list\_of\_Id=[100,102,103,104]

for id in list\_of\_Id:

    cur.execute("SELECT \* FROM Computer WHERE CompId=%(c\_id)s",{"c\_id":id})

    for CompId,Make,Model,MYear in cur:

        print(Make,CompId)

Bind variables are not Python variables. They are actually variables used in MySQL. They start with a percentage symbol.

When we use bind variables, the MySQL database will reuse the query thus improving the performance. One caution in using bind for queries is that using bind variables will improve performance only if the query is executed repeatedly for the same connection. If the query is executed in a new connection every time, this will not improve the performance.

**Fetching rows/data from DB**

**We have below methods in cursor class which we can use to retrieve data**

For fetching data/rows from DB we have three methods and all are executed on cursor object.

1. MySQLCursor.fetchall() Method
2. MySQLCursor.fetchmany(size=integer) Method
3. MySQLCursor.fetchone() Method

**Steps to run DQL statement**

* First execute the DQL statement using execute() statement using cursor object
* On cursor object then call fetchall() or fetchmany(n) or fetchone() method

**MySQLCursor.fetchmany(size=integer)**

This method fetches the next set of rows of a query result and returns a list of tuples. If no more rows are available, it returns an empty list

size --- it represents number of rows want to fetch, this default to 1 or if not specified

How to fetch data:

Step 1 --- execute sql statement using cursor, here dataset are not required

Step 2 --- using cursor run execute fetchone() or fetchmany() or fetchall()

query="select \* from woodshophome\_stock"

cursor.execute(query)

data=cursor.fetchmany(size=2) # size=2 means fetch 2 rowss

for each in data:

    print(each)

Output:

(1, 'Chair', 'King design', 0, 10, 'number')

(2, 'Bed', 'queen sze', 0, 12, 'unit')

**MySQLCursor.fetchall() Method**

This method fetches all records from DB or remaining records and returns in list of tuple.

Each value of tuple corresponds to a column data of table.

If no more rows are available, it returns an empty list.

Example 1 ---- Fetch all records from DB

query="select \* from woodshophome\_stock"

cursor.execute(query)

data=cursor.fetchall()

for each in data:

    print(each)

Example 1 ---- Fetch first 2 rows and then fetch remaining records

query="select \* from woodshophome\_stock"

cursor.execute(query)

data=cursor.fetchmany(size=2)

for each in data:

    print(each)

print('now getting all remaning records')

data=cursor.fetchall()

for each in data:

    print(each)

output-

(1, 'Chair', 'King design', 0, 10, 'number')

(2, 'Bed', 'queen sze', 0, 12, 'unit')

now getting all remaning records

(3, 'plywood', 'general purpose', 0, 1000, 'unit')

(4, 'Paint', 'Asian paint', 0, 123,

**Questions**: Print the column data from table

**Method1: ---** Using variable for each column

cur.execute("SELECT \* FROM Computer")

for column1,column2,column3,column4 in cur:

    print(column3,column1)

**Method 2:** --- Using indexing

cur.execute("SELECT \* FROM Computer")

for row in cur:

    print(row[2],row[0])

**MySQLCursor.fetchone() Method**

This method retrieves the next row of a query result set and returns a single sequence, or None if no more rows are available. By default, the returned tuple consists of data returned by the MySQL server, converted to Python objects

*The fetchone() method is used by fetchall() and fetchmany(). It is also used when a cursor is used as an iterator* *as you can see in below example.*

No errors if query is written to fetch more than one row but will fetch only one row.

e.g--- select \* from table\_name ----- this will still fetch one row

**Note:**

We can directly fetch records from cursor and number of data will be based on SQL query.

Records fetched will be stored in cursor object no need to store in any variable

Example:

Below code will run all records from specified table and database

query="select \* from woodshophome\_stock"

cursor.execute(query)

#data=cursor.fetchmany(size=2)

for each in cursor:

    print(each)

Output: it fetched all records

(1, 'Chair', 'King design', 0, 10, 'number')

(2, 'Bed', 'queen sze', 0, 12, 'unit')

(3, 'plywood', 'general purpose', 0, 1000, 'unit')

(4, 'Paint', 'Asian paint', 0, 123, 'unit')

**Some other methods of cursor class**

**MySQLCursor.fetchwarnings()**

fetchwarnings()--- used for fetching warning of last executed query*. To use this methods we should use value of get\_warnings parameter as True while creating connection object.*

**MySQLCursor.stored\_results()**

This method returns a list iterator object that can be used to process result sets produced by a stored procedure executed using the callproc() method.

**Attributes of cursor class**

1. cursor.column\_names ----- This read-only property returns the column names of a result set as sequence of Unicode strings.
2. cursor.description ------- description about table column
3. cursor.lastrowid
4. cursor.rowcount ----- number of rows fetched
5. cursor.with\_rows
6. cursor.statement --------- query executed by cursor

################################################################################

# Exception and Error in MySQL #

################################################################################

MySQL defined many types of exception which can broadly classified as below, it’s available in mysql.connector.errors

Error

|\_\_ InterfaceError

|\_\_ PoolError

|\_\_ DatabaseError

|\_\_ DataError

|\_\_ InternalError

|\_\_ IntegrityError

|\_\_ OperationalError

|\_\_ NotSupportedError

|\_\_ ProgrammingError

**ProgrammingError**

This class contains details of the error code and error message that is generated by MySQL when the query is executed has any errors.

**Example:** Let say in database Cumputer\_1 does not exists but we can below insert statement then it will give error-

import mysql.connector

con = mysql.connector.connect(host="host",user="username",passwd="password",database="database")

cur = con.cursor()

try:

    cur.execute("INSERT INTO Computer\_1 VALUES (100)")

    con.commit()

except mysql.connector.ProgrammingError as e:

    print(e) #Outpyt- 1146 (42S02): Table 'yourdatabase.computer\_1' doesn't exist.

finally:

    con.close()

**errors.Error Exception**

This exception is the base class for all other exceptions. It can be used to catch all errors in a single except statement. This is available in mysql.connector.errors

Error(errno=1146, sqlstate='42S02', msg="Table 'test.spam' doesn't exist")

All parameter are optional and default to None by python.

import mysql.connector

try:

cnx = mysql.connector.connect(user='scott', database='employees')

cursor = cnx.cursor()

cursor.execute("Some Query") # Syntax error in query

cnx.close()

except mysql.connector.Error as err: #initializing error or exception

print("Something went wrong: {}".format(err))

Initializing the exception supports a few optional arguments, namely msg, errno, values and sqlstate. All of them are optional and default to None.

Error(errno=2006)

Error(errno=2002, values=('/tmp/mysql.sock', 2))

Error(errno=1146, sqlstate='42S02', msg="Table 'test.spam' doesn't exist")

We can get errno, sqlstate, msg on error intnance

Example:

Below script shows how to fetch records from a table

cnx=ctd.DbConnection.get\_connection()

print(cnx)

cursor=cnx.cursor()

sql="select ProductName,UnitsInStock from products order by UnitsInStock DESC"

cursor.execute(sql)

try:

    result=cursor.fetchall()

except mysql.connector.Error as e:

    print(e)

else:

    for each in result:

        print(each)

finally:

    if len(result)>0:

        print('we got some results')

    print('closing connections')

    cursor.close()

    cnx.close()

**Handling Error and Exceptions**

Suppose we gave incorrect table (MyProducts) then for fetching data

cursor=cnx.cursor()

sql="select ProductName,UnitsInStock from MyProducts order by UnitsInStock DESC"

try:

    cursor.execute(sql)

    result=cursor.fetchall()

except mysql.connector.Error as e:

    print(e)

else:

    for each in result:

        print(each)

finally:

    print('closing connections')

    cursor.close()

    cnx.close()

We will get below error on screen

1146 (42S02): Table 'northwind.myproducts' doesn't exist

**errors.DataError Exception**

This exception is raised when there were problems with the data.

Examples:

column set to NULL that cannot be NULL,

out-of-range values for a column,

division by zero, column count does not match value count, and so on.

**errors.DatabaseError Exception**

This exception is the default for any MySQL error which does not fit the other exceptions.

errors.DatabaseError is a subclass of errors.Error.

**errors.IntegrityError Exception**

This exception is raised when the relational integrity of the data is affected.

errors.IntegrityError is a subclass of errors.DatabaseError.

For example, a duplicate key was inserted or a foreign key constraint would fail.

In Below example we are trying insert duplicate key that is example of data integrity error.

cursor=cnx.cursor()

try:

    cursor.execute("CREATE TABLE t1 (id int, PRIMARY KEY (id))")

    cursor.execute("INSERT INTO t1 (id) VALUES (1)")

    cursor.execute("INSERT INTO t1 (id) VALUES (1)")

except mysql.connector.IntegrityError as err: #or #except mysql.connector.Error as err

    print(err)

    cursor.execute('drop table t1')

**errors.InterfaceError Exception**

This exception is raised for errors originating from Connector/Python itself, not related to the MySQL server.

errors.InterfaceError is a subclass of errors.Error.

**errors.InternalError Exception**

This exception is raised when the MySQL server encounters an internal error, for example, when a deadlock occurred.

errors.InternalError is a subclass of errors.DatabaseError.

**errors.NotSupportedError Exception**

This exception is raised when some feature was used that is not supported by the version of MySQL that returned the error. It is also raised when using functions or statements that are not supported by stored routines.

errors.NotSupportedError is a subclass of errors.DatabaseError.

**errors.OperationalError Exception**

This exception is raised for errors which are related to MySQL's operations. For example: too many connections; a host name could not be resolved; bad handshake; server is shutting down, communication errors.

errors.OperationalError is a subclass of errors.DatabaseError.

**errors.PoolError Exception**

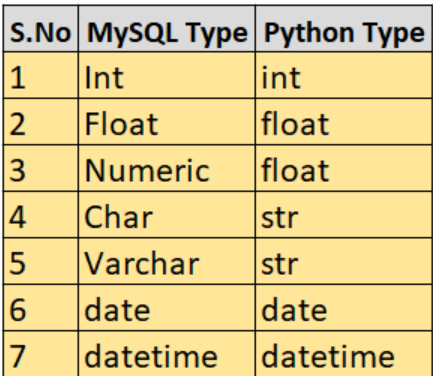
This exception is raised for connection pool errors. errors.PoolError is a subclass of errors.Error.

**errors.ProgrammingError Exception**

This exception is raised on programming errors, for example when you have a syntax error in your SQL or a table was not found

errors.ProgrammingError is a subclass of errors.DatabaseError.

**MySQL to Python data types**



################################################################################

# Database Transaction Management #

################################################################################

The database transaction represents a single unit of work. Any operation which modifies the state of the MySQL database is a transaction

Example-

Want to withdraw money from one account and deposit in to another account, if deposit operations fails then money should be transferred back to account.

Using **ACID properties**, we can study transaction management well. ACID stands for Atomicity, Consistency, isolation, and durability

* Atomicity:

Means all or nothing. Either all transactions are successful or none.

* Consistency:

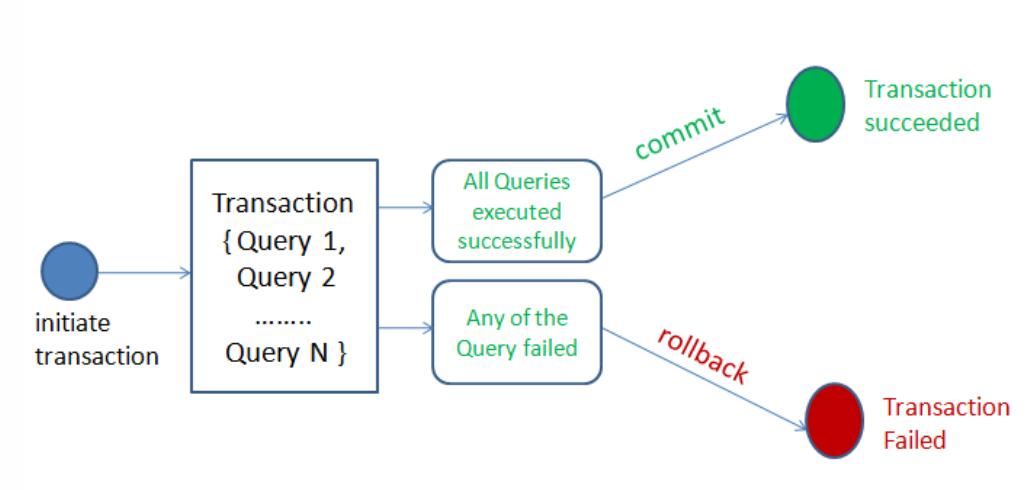
It ensures that the database remains in a consistent state after performing a transaction

* Isolation:

It ensures that the transaction is isolated from other transactions

* Durability:

It means once a transaction has been committed, it persists in the database irrespective of power loss, error, or restart system



In python we use Commit(), rollback() and setAutoCommit() to manage transactions.

################################################################################# Parameterized Query and Prepared statement # ################################################################################

A parameterized query is a query in which placeholders (%s) are used for parameters (column values) and the parameter values supplied at execution time.

We need to pass the following two arguments to a cursor.execute() function to run a parameterized query.

* query
* tuple of parameter values

There are four main reasons to use-

* **Compile Once:** Parameterized query compiled only once. When you use parameterized query, it gets precompiled and stored in a PreparedStatement object. Now, use this object to execute the same statement multiple times efficiently
* **Improves Speed:** If you execute SQL statements repeatedly with a precompiled query, it reduces the execution time.
* **Same Operation with Different Data:** You can use it to execute the same query multiple times with different data. For example, you want to insert 200 rows in a table. In such cases, use parameterized query to repeatedly execute the same operation with a different set of values.
* **It prevents SQL injection attacks.**

**How to use parameterized query**

It’s quite simple and already somewhere in this notes, use execute method with query and data.

**Working of a Parameterized Query**

The first time you pass a SQL query statement to the cursor’s execute() method, it creates the prepared statement, for subsequent phase of operations it is not compiled but used from prepared statement.

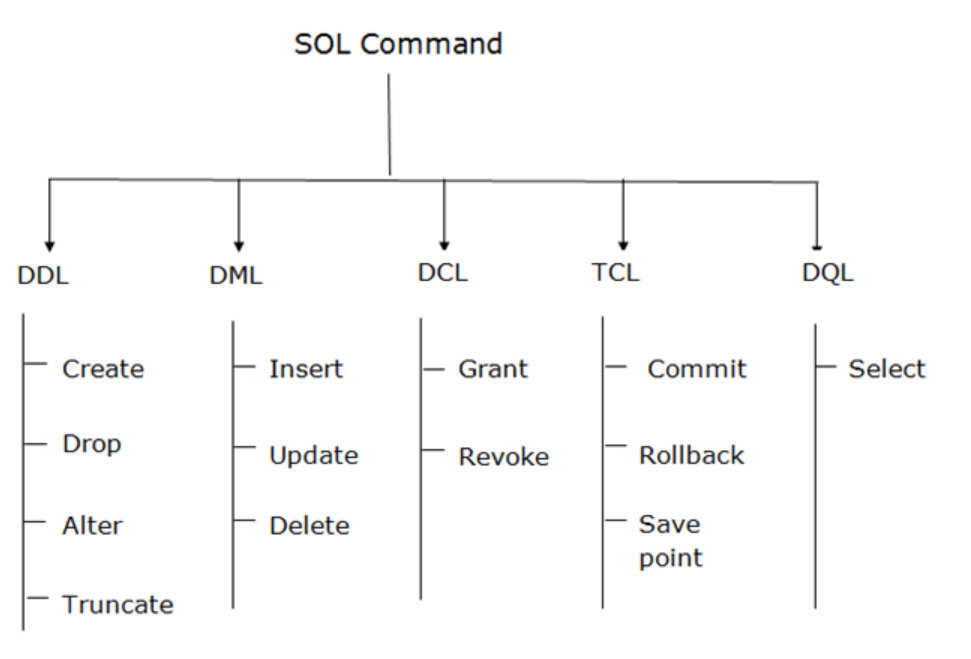
* In the first cursor.execute(query, tuple) Python prepares statement i.e. Query gets compiled
* For subsequent execution calls of cursor.execute(query, tuple), The query gets executed directly with passed parameter values

**How to use prepared statement.**

Just use prepared=True in cursor creation that will create a prepared cursor object.

Remaining all is same as what we do in normal execution or parameterized query.

################################################################################# SQL COMMANDS # ################################################################################



**a. Create**: It's used to create table in database

CREATE TABLE TABLE\_NAME (COLUMN\_NAME DATATYPES[,....]);

**b. DROP:** It is used to delete both the structure (delete column, constraints, table, database) and record stored in the table.

DROP TABLE tbl\_name

**c. ALTER:** It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add/delete new attribute.

ALTER TABLE table\_name ADD column\_name COLUMN-definition;

ALTER TABLE MODIFY (COLUMN DEFINITION....);

ALTER TABLE tbl\_name

[alter\_option| ADD CONSTRAINT]

**alter option**

there are many alter option available, few of them are-

1. Add [column]
2. Add [constraint]
3. Drop [constraint]
4. Drop [column]
5. Drop Primary Key

Example:

ALTER TABLE STU\_DETAILS ADD(ADDRESS VARCHAR2(20));

ALTER TABLE STU\_DETAILS MODIFY (NAME VARCHAR2(20));

**d. TRUNCATE**: It is used to delete all the rows from the table and free the space containing the table.

TRUNCATE TABLE table\_name;

**Data Manipulation Language**

Insert

Update

Delete

**a. INSERT:** The INSERT statement is a SQL query. It is used to insert data into the row of a table.

INSERT INTO TABLE\_NAME

(col1, col2, col3,.... col N)

VALUES (value1, value2, value3, .... valueN); ---- Syntax, **add data for specified columns**

INSERT INTO TABLE\_NAME

VALUES (value1, value2, value3, .... valueN); ----- Syntax, **add data for all columns**

**b. UPDATE:** This command is used to update or modify the value of a column in the table.

UPDATE table\_name SET [column\_name1= value1,...column\_nameN = valueN] [WHERE CONDITION]

**c. DELETE:** It is used to remove one or more row from a table.

DELETE FROM table\_name [WHERE condition];

**Data Control Language**

This is used to revoke, grant access of any user.

**Transaction Control Language**

Comit

Rollback

SavePoint

Some SQL keywords

**SQL MIN() and MAX() Functions**

The MIN() function returns the smallest value of the selected column/numbers.

The MAX() function returns the largest value of the selected column/numbers.

MIN(expression) ------- Syntax

MAX(expression) ------- Syntax

SELECT MIN(column\_name)

FROM table\_name

WHERE condition; -------- Syntax

Example:

SELECT MAX(Price) AS LargestPrice

FROM Products;

**SQL COUNT(), AVG() and SUM() Functions**

The COUNT() function returns the number of rows that matches a specified criterion.

The AVG() function returns the average value of a numeric column.

The SUM() function returns the total sum of a numeric column.

SELECT COUNT(column\_name)

FROM table\_name

WHERE condition;

SELECT AVG(column\_name)

FROM table\_name

WHERE condition;

SELECT SUM(column\_name)

FROM table\_name

WHERE condition;

**SQL IN Operator**

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

SELECT column\_name(s)

FROM table\_name

WHERE column\_name IN (value1, value2, ...);

SELECT column\_name(s)

FROM table\_name

WHERE column\_name IN (SELECT STATEMENT);

**SQL BETWEEN Operator**

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included in range.

SELECT column\_name(s)

FROM table\_name

WHERE column\_name BETWEEN value1 AND value2;

**Example :** Select everything from Products table whose price is between 10 and 20.

SELECT \* FROM Products

WHERE Price BETWEEN 10 AND 20;

**SQL HAVING Clause**

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

SELECT column\_name(s)

FROM table\_name

WHERE condition

GROUP BY column\_name(s)

HAVING condition

ORDER BY column\_name(s);

**Example**

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers: ---- W3school

SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country

HAVING COUNT(CustomerID) > 5;

**SQL EXISTS Operator**

The EXISTS operator is used to test for the existence of any record in a subquery.

The EXISTS operator returns TRUE if the subquery returns one or more records.

SELECT column\_name(s)

FROM table\_name

WHERE EXISTS

(SELECT column\_name FROM table\_name WHERE condition);

**SQL ANY and ALL Operators**

The ANY and ALL operators allow you to perform a comparison between a single column value and a range of other values.

ALL means that the condition will be true only if the operation is true for all values in the range.

The ALL command returns true if all of the subquery values meet the condition.

The word ALL, which must follow a comparison operator, means “return TRUE if the comparison is TRUE for ALL of the values in the column that the subquery returns.”

SELECT s1 FROM t1 WHERE s1 > ALL (SELECT s1 FROM t2);

**SQL ANY Operator**

ANY means that the condition will be true if the operation is true for any of the values in the range.

returns a boolean value as a result

returns TRUE if ANY of the subquery values meet the condition

SELECT column\_name(s)

FROM table\_name

WHERE column\_name operator ANY

(SELECT column\_name

FROM table\_name

WHERE condition);

**SQL ALL Operator**

returns a boolean value as a result

returns TRUE if ALL of the subquery values meet the condition

is used with SELECT, WHERE and HAVING statements

SELECT column\_name(s)

FROM table\_name

WHERE column\_name operator ALL

(SELECT column\_name

FROM table\_name

WHERE condition);

**SQL SELECT INTO Statement**

The SELECT INTO statement copies data from one table into a new table (if not existing then will create)

SELECT column1, column2, column3, ...

INTO newtable [IN externaldb]

FROM oldtable

WHERE condition;

**SQL INSERT INTO SELECT Statement**

The INSERT INTO SELECT statement copies data from one table and inserts it into another table.

The INSERT INTO SELECT statement requires that the data types in source and target tables matches.

INSERT INTO table2

SELECT \* FROM table1

WHERE condition;

INSERT INTO table2 (column1, column2, column3, ...)

SELECT column1, column2, column3, ...

FROM table1

WHERE condition;

# [INSERT INTO vs SELECT INTO](https://stackoverflow.com/questions/6947983/insert-into-vs-select-into)

They do different things. Use INSERT INTO SELECT when the table exists. Use SELECT INTO when it does not.

**SQL SELECT DISTINCT Statement**

The SELECT DISTINCT statement is used to return only distinct (different) values.

Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values

SELECT DISTINCT column1, column2, ...

FROM table\_name;

**SQL subquery**

A subquery is a query within another query. The outer query is called as main query and inner query is called as subquery

The subquery/inner query generally executes first, and its output is used to complete the query condition for the main or outer query

Subquery must be enclosed in parentheses

**ORDER BY command cannot be used in a Subquery**. GROUPBY command can be used to perform same function as ORDER BY command.

Subqueries can be used with SELECT, UPDATE, INSERT, DELETE statements along with expression operator. It could be equality operator or comparison operator such as =, >, =, <= and Like operator.

**GROUP BY**

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country"

The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

SELECT column\_name(s)

FROM table\_name

WHERE condition

GROUP BY column\_name(s)

ORDER BY column\_name(s);

################################################################################# MYSQL keywords #

################################################################################

<https://www.w3schools.com/sql/sql_ref_keywords.asp>

1. ADD CONSTRAINT

The ADD CONSTRAINT command is used to create a constraint after a table is already created. As table was already created then then adding constraint then we need to use it with ALTER TABLE keyword.

ALTER TABLE tbl\_name

[alter\_option| ADD CONSTRAINT]

**alter option**

there are many alter option available, few of them are-

1. Add [column]
2. Add [constraint]
3. Drop [constraint]
4. Drop [column]
5. Drop Primary Key
6. ALTER COLUMN ----- for changing the data type of column.

Example:

ALTER TABLE Persons

ADD CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName);

**Note:**

If we are modifying any property(adding column, keys, deleting) of table then we will have to use ALTER TABLE keyword.

1. ADD

Adds column in existing table. This must be also used with ALTER TABLE keyword.

ALTER TABLE Customers

ADD col\_name data\_type;

1. ALTER /ALTER TABLE / ALTER COLUMN

ALTER command is used to change table or column property of table.

Adds, deletes, or modifies columns in a table, or changes the data type of a column in a table.

1. **ALTER TABLE**

This is used with alter/change the table property/structure. (already above)

1. **ALTER COLUMN**

This is used to change the column data type of given table. Always used with ALTER TABLE.

ALTER TABLE table\_name

ALTER COLUMN col\_name new\_data\_type;

1. ALL

The ALL command returns true if all of the subquery values meet the condition.

The word ALL, which must follow a comparison operator, means “return TRUE if the comparison is TRUE for ALL of the values in the column that the subquery returns.”

SELECT s1 FROM t1 WHERE s1 > ALL (SELECT s1 FROM t2);

1. AND ------ Only includes rows where conditions are TRUE
2. ANY --- Returns true if any of the subquery values meet the condition
3. AS --- this is for giving any alais name
4. ASC ---- Sorts the result set in ascending order
5. BETWEEN

The MYSQL BETWEEN condition specifies how to retrieve values from an expression within a specific range (values will be inclusive). It is used with SELECT, INSERT, UPDATE and DELETE statement.

expression BETWEEN value1 AND value2; ------- syntax

**example**

SELECT \* FROM Products

WHERE Price BETWEEN 10 AND 20;

**Question:**

Write a query to display the salary for all employee whose salary is not in the range $10,000 through $15,000, table name is employees

**Solution 1:**

select salary from employees where salary not between 10000 and 15000;

**Solution 2:**

select salary from employees where salary>15000 or salary<10000;

1. CHECK

The CHECK constraint limits the value that can be placed in a column. It is used with CREATE table option.

CHECK (expr) ----------- Syntax

**Example**

CREATE TABLE Persons (

Age int,

CHECK (Age>=18)

);

1. DELETE

DELETE is a DML statement that removes rows from a table.

The DELETE statement deletes rows from tbl\_name and returns the number of deleted rows. To check the number of deleted rows, call the ROW\_COUNT() function

DELETE FROM tbl\_name [[AS] tbl\_alias]

[WHERE where\_condition | expression]

[ORDER BY ...]

[LIMIT row\_count]

**Example**

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

1. SELECT DISTINCT

The SELECT DISTINCT command returns only distinct (different) values in the result set.

**Example**

SELECT DISTINCT Country FROM Customers;

1. DROP /DROP TABLE / DROP COLUMN /DROP FOREIGN KEY / DROP PRIMARY KEY /DROP CHECK
2. **DROP COLUMN**

DROP COLUMN deletes a column from table, as it is modifying the table then we must use it with ALTER TABLE keyword as mention above.

ALTER TABLE table\_name

DROP COLUMN col\_name; -------- Drop column

ALTER TABLE table\_name

DROP PRIMARY KEY; ------- Drop Primary key

ALTER TABLE Orders

DROP FOREIGN KEY foreign\_key\_name; ---- Drop Foreign key

ALTER TABLE Persons

DROP CHECK check\_statement; ---- Drop check constrains

ALTER TABLE Persons

ALTER coln\_name DROP DEFAULT; --- Drop default value for a given col\_name

1. DROP TABLE

This is used for deleting table.

DROP table tbl\_name ------ Dropping table

1. EXISTS

The EXISTS command tests for the existence of any record in a subquery, and returns true if the subquery returns one or more records

Query where EXISTS (sub query) --------- Syntax

**Example**

SELECT SupplierName

FROM Suppliers

WHERE EXISTS (SELECT ProductName FROM Products WHERE SupplierId = Suppliers.supplierId AND Price < 20);

1. IN

The IN command allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

select statement WHERE col\_name IN (val1, val2,...valn)

1. NOT IN

This operator is used for inverting the IN command.

select statement WHERE col\_name NOT IN (val1, val2,...valn)

**Question**

Write a query to display the all information for all employee whose job is that of a Programmer or a Shipping Clerk, and salary(column name) is not equal to 4,500, 10,000.

Database name is hr , table name is employees.

**Solutions**

select \* from employees where JOB\_ID IN ('IT\_PROG','PU\_CLERK') and salary NOT in (9000.00,6000.00);

**IN vs BETWEEN**

BETWEEN --------- It takes only two value ( inclusive) for specifying range

IN ------- It takes any number of value of any type.

1. TOP, LIMIT and ROWNUM

These commands are used to specify the number or X percent of records to return.

These three works same but are database dependent keyword

LIMIT ------ MYSQL

ROWNUM ---- Oracle

SELECT FROM table\_name

WHERE [condition]

LIMIT number|percent column\_name(s) ----------- Syntax

**LIMIT vs COUNT**

Limit command is used to select specified number of records , count is used to return number of records meeting criteria.

**Example:**

SELECT \* FROM employees

ORDER BY salary DESC LIMIT 3;

**Setting Row Offset in LIMIT Clause**

The LIMIT clause accepts an optional second parameter.

the first parameter specifies the offset of the first row to return i.e. the starting point, whereas the second parameter specifies the maximum number of rows to return

SELECT FROM table\_name

WHERE [condition]

LIMIT offset\_number, number\_of\_records\_to\_fetch column\_name(s) ----------- Syntax

**Example**

SELECT \* FROM employees

ORDER BY salary DESC LIMIT 2, 1;

1. SET

The SET command is used with UPDATE to specify which columns and values that should be updated in a table.

UPDATE table\_name

SET var\_name1|col\_name1= vaue

WHERE condition

Example

UPDATE Customers  
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'  
WHERE CustomerID = 1;

**LIKE operator**

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the LIKE operator:

* The percent sign (%) represents zero, one, or multiple characters
* The underscore sign (\_) represents one, single character

**Questions**

Write a query to display the first\_name of all employees who have both "b" and "c" in their first\_name(column name), table name is employees

**Solution**

select first\_name from employees where first\_name like '%b%' and first\_name like '%c%';

**Question**

Write a query to display the last name of employees having 'e' as the third character.

Database name --- hr, table name--- employees.

**Solutions**

SELECT last\_name FROM employees WHERE last\_name LIKE '\_\_e%';

###############################################################################

# FUNCTIONS IN MYSQL #

###############################################################################

Function can be divided into—

1. String function
2. Numerical Function
3. DATE function
4. Server functions

###################################

# String function #

###################################

**Note:**

We can make string comparison, search case sensitive just by adding BINARY. This compares, checks byte-by-byte. Check example in LOCATE function.

1. CHAR\_LENGHT() /LENGTH()

The CHAR\_LENGTH() function return the length of a string (in characters).

CHAR\_LENGTH (string) -------- Syntax

LENGTH (string) ---------- Syntax

string -----Required. The string to count the length for.

**Question:**

Write a query to display the last\_name(column name) of employees(table name) whose names have exactly 6 characters. Database name – hr, table name- employees. ---- w3resource

**Solutions**

select \* from employees where char\_length(first\_name)=6;

1. LOCATE(substring, string, start)

The LOCATE() function returns the position of the first occurrence of a substring in a string.

If the substring is not found within the original string, this function returns 0.

**Note:**

We can make string comparison, search case sensitive just by adding BINARY. This compares, checks byte-by-byte.

**Example:** --- Imp.

select locate('a','Amitabh',1) as position; ---------- output will be 1, b/c default case insensitive.

select locate (BINARY 'a','Amitabh',1) as position; ---------- it’s output will be 5

1. UPPER / UCASE

UPPER() function converts a string to upper-case.

UPPER(text)

Example

SELECT UPPER(CustomerName) AS UppercaseCustomerName

FROM Customers;

1. TRIM

Removes leading and trailing spaces from String. ( Same as python trim() of string)

TRIM(text) ----------

Example:

SELECT TRIM(' SQL Tutorial ') AS TrimmedString;

1. LTRIM / RTRIM

These functions are used to remove spaces from left or right of string. (same in python )

1. STRCMP

The STRCMP() function compares two strings

STRCMP( string1 , string2)

String1 and string2 are two string to be compared

1. SUBSTRING() ---------- this was asked in interview

The SUBSTRING() function extracts a substring from a string (starting at any position).

(Like slicing in python)

SUBSTRING(string, start, length)

String ----- string function where substring will be extracted

Start ---- start index for getting substring. It’s lowest value is 1.

Length ---- number of character to be extracted, if omitted then will get all string from start.

**Example:**

SELECT SUBSTRING(CustomerName, 2, 5) AS ExtractString

FROM Customers; ------------ this was asked in interview

1. SPACE

The SPACE() function returns a string of the specified number of space characters.

**Example**

SELECT SPACE(10);

1. REVERSE

The REVERSE() function reverses a string and returns the result. (same function in python also but we can do by using slicing concept also)

REVERSE(string) -------- Syntax

**Example**

SELECT REVERSE(CustomerName)

FROM Customers;

1. REPLACE

The REPLACE() function replaces all occurrences of a substring within a string, with a new substring.

In python we have re.sub(pattern, replaced\_with, string) or subn() function

REPLACE (string, string\_to\_be\_replaced, new\_string) ------ Syntax

String ---- Required. The original string, in which replacement will be done

String\_to\_be\_replaced ---- Required. The substring to be replaced

New\_string ----- Required. The new replacement substring

1. LOWER / LCASE --------- Converts string into lower case ( same function I python also)

LOWER ( text)

**Example**

SELECT LOWER("SQL Tutorial is FUN!");

10. REPEAT

The REPEAT() function repeats a string as many times as specified.

REPEAT(string, number) -------- Syntax

String ------- string to be repeated. Required

Number ---- number to times to be repeated. Repeated

**Example**

SELECT REPEAT(CustomerName, 2)

FROM Customers;

11. CONCAT

The CONCAT() function adds two or more expressions together.

CONCAT(expression1, expression2, expression3,...)

NOTE:

If any of expression is null then result will be null.

12. CHAR\_LENGTH/CHARACTER\_LENGTH()

The CHARACTER\_LENGTH() function return the length of a string (in characters).

CHARACTER\_LENGTH(string) ------ Syntax

13. LEFT

The LEFT() function extracts a number of characters from a string (starting from left).

LEFT(string, number\_of\_chars) ------ Syntax

string--------Required. The string to extract from.

number\_of\_chars----Required. The number of characters to extract. If this parameter is larger than the number of characters in string, this function will return string

**Example**

SELECT LEFT(CustomerName, 5) AS ExtractString

FROM Customers;

**NOTE:**

This is same as SUBSTRING(string, start, length) where start =0.

14.RIGHT

Same as LEFT but it return from right.

##########################

# Numerical Function #

##########################

1. ABS

The ABS() function returns the absolute (positive) value of a number.

ABS (number)

Example:

SELECT ABS(-243.5);

1. AVG

The AVG() function returns the average value of an expression.

AVG(expression) ------- Syntax

expression ---- Required. A numeric value (can be a field or a formula)

Example

SELECT \* FROM Products

WHERE Price > (SELECT AVG(Price) FROM Products);

1. CEIL

The CEIL() function returns the smallest integer value that is bigger than or equal to a number.

CEIL(number) ----- Syntax

Number ----------- required, A numeric value.

Example:

SELECT CEIL(25);

1. FLOOR

The FLOOR() function returns the largest integer value that is smaller than or equal to a number.

FLOOR(number) ------- Syntax

Example:

SELECT FLOOR(25);

1. DIV

The DIV function is used for integer division (x is divided by y). An integer value is returned.

DIV (x,y) ------- syntax

X --- Required, A value that will be divided

Y --- Required, The divisor

1. GREATEST

The GREATEST() function returns the greatest value of the list of arguments.

GREATEST(arg1, arg2, arg3, ...) ---- Syntax

arg1, arg2….. -------- Required. The list of arguments to be evaluated

**Example**

SELECT GREATEST("w3Schools.com", "microsoft.com", "apple.com");

1. LEAST

The LEAST() function returns the smallest value of the list of arguments.

LEAST(arg1, arg2, arg3, ...) ---- Syntax

arg1, arg2….. -------- Required. The list of arguments to be evaluated

Example:

1. SELECT LEAST(3, 12, 34, 8, 25);
2. SELECT LEAST("w3Schools.com", "microsoft.com", "apple.com");
3. MAX/MIN

The MAX() function returns the maximum value in a set of values.

The MIN() function returns the minimum value in a set of values.

MIN(expression) ------- Syntax

MAX(expression) ------- Syntax

**Example:**

SELECT MIN(Price) AS SmallestPrice FROM Products;

1. ROUND

The ROUND() function rounds a number to a specified number of decimal places.

ROUND(number, decimals) ------- Syntax

number ------ Required. The number to be rounded.

decimal ------ Optional. The number of decimal places to round number to. If omitted, it returns the integer (no decimals)

**Example**

1.)SELECT ROUND(345.156, 0);

2.)SELECT ProductName, Price, ROUND(Price, 1) AS RoundedPrice FROM Products;

1. SIGN ---- return the sign of number
2. SQRT ---- returns the square root of number
3. SUM --- Calculte sum of number
4. TRUNCATE ---- Truncates a number to the specified number of decimal places

TRUNCATE(number, decimals) ---- Syntax

number ----- Required. The number to be truncated

decimal ---- Required. The number of decimal places to truncate to

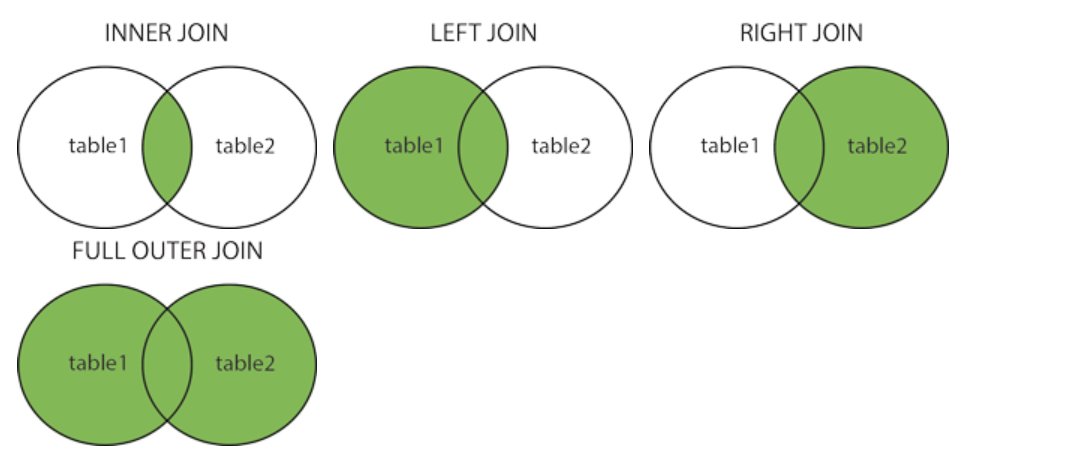
**Example**

1.)SELECT TRUNCATE(345.156, 0);

2.)SELECT TRUNCATE(135.375, 2);

**JOINS**

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.



**INNER join**

The INNER JOIN clause compares each row in the t1 table with every row in the t2 table based on the join condition.

SELECT

select\_list

FROM t1

INNER JOIN t2 ON join\_condition1

INNER JOIN t3 ON join\_condition2

...;

SELECT column\_name(s)  
FROM table1  
INNER JOIN table2ON table1.column\_name = table2.column\_name;

e.g..

SELECT Orders.OrderID, Customers.CustomerName  
FROM Orders  
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;\

**Natural join**

A natural join is a type of join operation that creates an implicit join by combining tables based on columns with the same name and data type. It is similar to the INNER or LEFT JOIN, but we cannot use the ON or USING clause with natural join as we used in them.

**Notes:**

* There is no need to specify the column names to join.
* The resultant table always contains unique columns.
* It is possible to perform a natural join on more than two tables.
* Do not use the ON clause.

SELECT [column\_names | \*]

FROM table\_name1

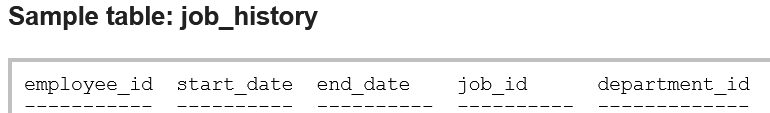
NATURAL JOIN table\_name2;

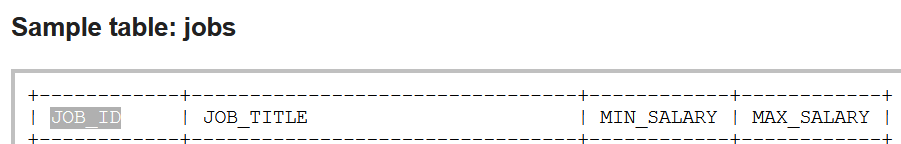
Here we are not specifying the column name on which join operation will be performed, it picked itself.

**Question**

Write a query to find the employee ID, job title, number of days between ending date and starting date for all jobs in department 90 from job history.

<https://www.w3resource.com/mysql-exercises/join-exercises/find-the-employee-id-job-title-number-of-days-between-ending-date-and-starting-date-for-all-jobs-in-department-90-from-job-history.php>





**Method 1:** Using Natural join.

SELECT employee\_id, job\_title, end\_date-start\_date Days FROM job\_history

NATURAL JOIN jobs

WHERE department\_id=90;

**Method 2:** Using INNER join

select employee\_id,job\_title,end\_date-start\_date from job\_history

inner join jobs

on job\_history.JOB\_ID=jobs.JOB\_ID where department\_id=90;

**LEFT / LEFT OUTER join**

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

*In some databases it is called as left outer join*

SELECT column\_name(s)

FROM table1

LEFT JOIN table2

ON table1.column\_name = table2.column\_name;

**Example**

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID

ORDER BY Customers.CustomerName;

**RIGHT join**

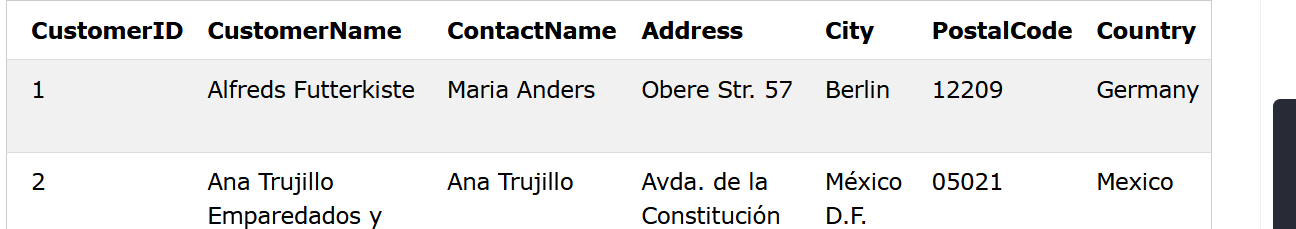
Same as left joins

**Self join**

A self join is a regular join, but the table is joined with itself.

SELECT column\_name(s)  
FROM table1 T1, table1 T2  
WHERE condition;

Let say we have below table structure



**Question**:- Write a SQL query to find customer from same city.

selet A.customer\_name, B.customer\_name

from customer A , customer B

where A.customername<>B.customername

where A.city=B.city

**CROSS join**

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

The result set will include all rows from both tables, where each row is the combination of the row in the first table with the row in the second table. In general, if each table has n and m rows respectively, the result set will have nxm rows.

SELECT \*/column(s) FROM t1

CROSS JOIN t2;

**########################**

**# Natural joins vs Inner join #**

**########################**

Natural join:

Natural Join joins two tables based on same attribute name and datatypes. The resulting table will contain all the attributes of both the table but keep only one copy of each common column.

SELECT \*

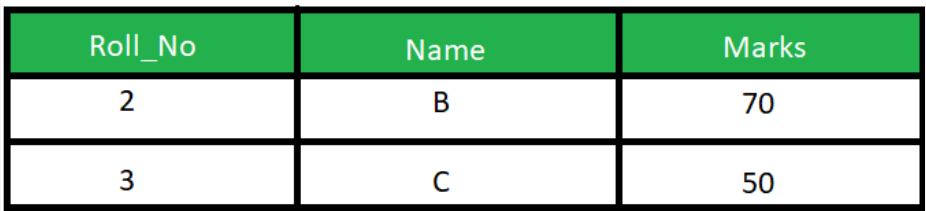
FROM Student S NATURAL JOIN Marks M; ------- Syntax

|  |  |
| --- | --- |
| Table Student | Table Mark |
|  |  |

SELECT \*

FROM Student S NATURAL JOIN Marks M

Result:



**Inner join:**

Inner Join joins two table on the basis of the column which is explicitly specified in the ON clause. The resulting table will contain all the attributes from both the tables including common column also.

SELECT \*

FROM A INNER JOIN B ON A.col\_name= B.Col\_name;

**UNION vs UNION ALL**

Both UNION and UNION ALL are used for combining two or more than two result sets using select statements.

UNION ---- it combines two or more result sets and removes duplicated record (default behavior)

UNION ALL --- it combines two or more result sets but don’t remove duplicate rows.

################################################################################

# NoSQL ---- MongoDB #

################################################################################

For connecting to MongoDB we can will be using pymongo library.

Connecting to MongoDB

For connecting to MongoDB we will be using MongoClient class from MongoDB and can connect in below two ways-

client = MongoClient(hostname, 27017) ---- Method 1

client = MongoClient('connection\_string') ---- Method 2

connection\_string = mongodb://hostname:portnumber/

**Getting a Database**

A single instance of MongoDB can support multiple independent databases. When working with PyMongo you access databases using attribute style access or dictionary style on MongoClient instances:

db = client.database\_name --------- Using attribute style

db = client['database\_name'] ------- Using dictionary styke

**Getting a Collection**

We can get collection from on database in attribute or dictionary style on db instance

A collection is a group of documents stored in MongoDB, and can be thought of as roughly the equivalent of a table in a relational database.

collection=db.colluection\_name ------------ Using attribute style

collection=db['collection\_name'] ------------ Using dictionary style

**Getting all database and collection names:**

db\_list=conn.list\_database\_names() ----- To get all Database names in list form

col\_list=db.list\_collection\_names() ---- To get all collection in list form

**Inserting document**

We can insert document using insert\_one() or insert\_many() from collection class.

**collection\_ins.insert\_one(document)**

To insert single document in database

Insert\_one returns a Result object. Using the attribute inserted\_id, we can retrieve the insertion id.

Example:

from pymongo import MongoClient

conn=MongoClient('mongodb://localhost:27017/')

db=conn['ETA']

col=db['student']

document={'Name':'John','Regd\_id':'1','Age':20,'Sex':'M'}

returnval=col.insert\_one(document)

print(returnval.inserted\_id)

**collection\_inst.insert\_many(documents)**

To insert multiple documents in database. Multiple documents in MongoDB is similar to List of dictionary in python.

Example:

from pymongo import MongoClient

conn=MongoClient('mongodb://localhost:27017/')

db=conn['ETA']

col=db['student']

document\_list=[

    {'Name':'Kelley','Regd\_id':'2','Age':21,'Sex':'M'},

    {'Name':'Hannah','Regd\_id':'3','Age':18,'Sex':'F'},

    {'Name':'Ravi','Regd\_id':'4','Age':22,'Sex':'M'},

    {'Name':'Rachel','Regd\_id':'5','Age':26,'Sex':'F'},

    {'Name':'Esther','Regd\_id':'6','Age':19,'Sex':'F'}

]

returnval=col.insert\_many(document\_list)

print(returnval.inserted\_ids)

**Getting document from DB**

For finding data from DB we have below methods in collection class

find\_one() ----- Getting one document from db , **returns pymongo cursor object**

find() ----- To retrive all document , **return pymongo cursor object**

This method returns a single document matching a query (or None if there are no matches)

find\_one() also supports querying on specific elements that the resulting document must match

col.find\_one({'attr':value}) ----- getting data based on some element value

**Note:**

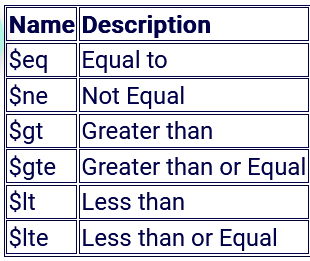
The returned document contains an "\_id", which was automatically added on insert. We can get data using \_id attribute in below way-

collection.find\_one({"\_id": post\_id})

**Retrieving based on condition**

Sometime we need to find the document based on some condition. For this we have below operator available-

E.g- find document for which age is greater then 30 .



col\_inst.find\_one({'attr':{operator:value}})

col\_inst.find({'attr':{operator:value}})

**Example:**

Find all document from ETA database and student collection for which age is greater than 21.

**Solution:**

from pymongo import MongoClient

conn=MongoClient('mongodb://localhost:27017/')

db=conn['ETA']

col=db['student']

curs=col.find({'Age':{'$gt':21}})

for row in curs:

    print(row)

**Using python regex to find document**

We can use python regex to find the document.

curs=col\_int.find({'attr':{'$regex':'pattern'}})

**Projection**

Projection is a technique to restrict the fields/columns in the output.

We need to pass the dictionary of column names as keys and their project value as 0 or 1, where-

1 projects it in the output

0 hides it from the output.

col.find({},{field1:value1,field2:value2…fieldn:valuen})

curs=col.find({},{'\_id':0,'Name':1,'Age':1}) ---- In output it will have Name and Age field

**Sorting the key values/resultset**

We can sort the resultset in below way-

collection\_obj.find({}).sort(Key,Order)

key= field by which it will be sorted

Order = 1 ---- sort in ascending order

order= -1 ---- sort in descending order

**Limiting number of records to fetch**

To limit the number of documents to be displayed, we use limit() method of cursor class. We need to pass the number of documents to be displayed as an argument to limit() method.

limit(n) ----- to get n number of result set

Example: It will fetch only two documents.

from pymongo import MongoClient

conn=MongoClient('mongodb://localhost:27017/')

db=conn['ETA']

col=db['student']

curs=col.find({},{'\_id':0,'Regd\_id':1,'Age':1}).limit(2)

for row in curs:

    print(row)

**Deleting the document**

For deleting the documents, we have below two methods-

col\_int.delete\_one({filed\_name:value})--- Delete the one doc which matched the given condition

col\_int.delete\_many({filed\_name:value}) ---- Delete all doc matching the given condition

Note:

* If not provided with a query, delete\_one will delete the first document from the collection where as delete\_many will delete all the documents.
* Delete\_many will return a delete object.
* Delete object has an attribute deleted\_count which holds the number of records deleted.

Example:

from pymongo import MongoClient

conn=MongoClient('mongodb://localhost:27017/')

db=conn['ETA']

col=db['student']

col.delete\_one({'Regd\_id':'1'})

col.delete\_many({'Regd\_id':'2'})

curs=col.find({})

for row in curs:

    print(row)

**Updating the document**

To update documents, we use update\_one() and update\_many() methods of collection class.

update\_one({arg1},{arg2}) --- updates the first document that matches with the given query.

update\_many(({arg1},{arg2}) --- updates the first document that matches with the given query.

**Note:**

First argument is query and second argument is filed and value that want to update.

Example:-

from pymongo import MongoClient

conn=MongoClient('mongodb://localhost:27017/')

db=conn['ETA']

col=db['student']

#update Age to 25 for doc where Regd\_id=3

col.update\_one({'Regd\_id':'3'},{'$set':{'Age':25}})

#Update the Age to 25 for which Regd\_id is greater than 4

col.update\_many({'Regd\_id':{'$gt':'4'}},{'$set':{'Age':24}})

curs=col.find({},{'\_id':0,'Regd\_id':1,'Age':1})

for row in curs:

    print(row)

Drop a collection

For dropping a collection we can use drop() from collection class

col\_inst.drop() ---- To drop a collection