**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

try:

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

    print(cnx)

except Exception as e:

    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 resultset 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

**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

**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() Method
3. MySQLCursor.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 represent 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 fetched all records from DB or remaining records and returns in list of tuple.

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,

**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

Error

|\_\_ InterfaceError

|\_\_ PoolError

|\_\_ DatabaseError

|\_\_ DataError

|\_\_ InternalError

|\_\_ IntegrityError

|\_\_ OperationalError

|\_\_ NotSupportedError

|\_\_ ProgrammingError

**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.

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 I 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 kis 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.

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

# 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:

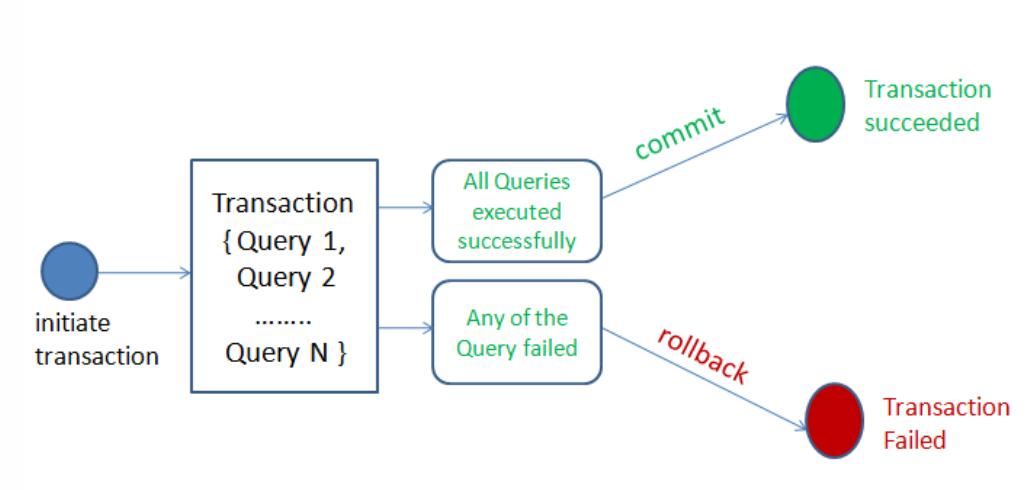
It ensures that the database remains in a consistent state after performing a transaction

* 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.

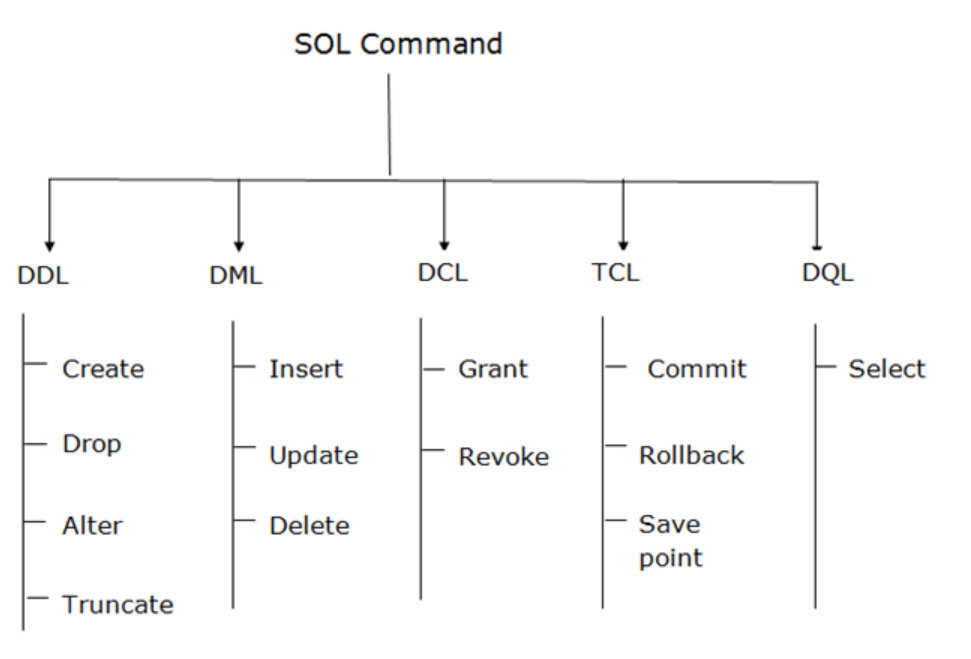
* In the first cursor.execute(query, tuple) Python prepares statement i.e. Query gets compiled
* For subsequent execution calls of cursor.execute(query, tuple), The query gets executed directly with passed parameter values

**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 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 a new attribute.

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

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

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);

INSERT INTO TABLE\_NAME

VALUES (value1, value2, value3, .... valueN);

**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.

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

SELECT MIN(column\_name)

FROM table\_name

WHERE condition;

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.

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:

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.

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

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

**SQL ANY Operator**

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

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;

**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 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);