4.0 Structure Query Language (SQL)

SQL was developed in 1970's in an IBM laboratory "San Jose Research Laboratory" (now the Amaden Research center). SQL is derived from the SEQUEL one of the database language popular during 1970's. SQL established itself as the standard relational database language. Two standard organization (ANSI) and International standards organization (ISO) currently promote SQL standards to industry.
In 1986 ANSI & ISO published an SQL standard called SQL-86. In 1987, IBM published its own corporate SQL standard, the system application Architecture Database Interface (SAA-SQL). In 1989, ANSI published extended standard for SQL called, SQL-89. The next version was SQL-92, and the recent version is SQL: 1999.

4.1 Basic Term and Terminology

Query: is a statement requesting the retrieval of information.
Query language: language through which user request information from database. These languages are generally higher level language than programming language.

The two types of query language are:

(i) Procedural language

- User instructs the system to perform sequence of operation on the database to compete the desired result. Example : relational algebra

ii) Non- procedural language

- User describes the desired information without giving a specific procedure for obtaining that desired information.
- Examples: tuple relational calculus and domain relational calculus.

4.2 Database Languages

Two types of database language
1. Data Definition Language (DDL)
2. Data Manipulation Language (DML)

Data Definition Language

- Specifies the database schema.
- For e.g.: The following statement in SQL defines relation named `student'.

```
CREATE TABLE student
(
  student_id  VARCHAR2(3),
  address      VARCHAR(30)
);
```

The execution of this DDL statement creates the `student' table. It also updates a special set of tables called *data dictionary* or *data directory*. A data dictionary contains metadata, that is data about data. The schema of table is an example of metadata. A database system consults data dictionary or data directory.

Through the set of special type of DDL, called data storage definition language, we may specify the storage structure (like size of database, size of table etc) and access methods.

The DDL allow to enforce constrains in the database. For example: student_id should begin with `S', address could not be null etc.

```
CREATE TABLE STUDENT
(
  student_id   VARCHAR2 (3),
  address      VARCHAR2 NOT NULL,
  CONSTRAINT ch_student_id CHECK (student_id LIKE `S%')
 );
```

The database systems check these constraints every time the database is updated.

Data Manipulation Language:

- A data manipulation language is a language that enables users to access or manipulate the data in database. The data manipulation means :
  - Retrieval of information stored in database.
  - The insertion of new information into database.
  - Deletion of information from database.
  - Modification of data in database.

Two types of data manipulation languages are:

- Procedural DML: User need to specify what data are needed to retrieve (modify) and how to retrieve those data.
- Non Procedural (Declarative DML) : User requires to specify what data are needed to retrieve without specifying how to get (retrieve) those data.
  - Non procedural DML are easier to understand and use than procedural DML, since user does not have to specify now to get data from database.
  - The DML component of SQL is non procedural language.

Example: Consider a simple relational database.

customer_id	customer_name	customer_address
C001	Smith	Kathmandu
C002	John	Bhaktapur
C003	Semi	Lalitpur
C004	Ivan	Kathmandu

The customer table

account_no	balance
A101	900
A102	300
A103	200
A104	700

The account table

customer_id	account_no
C001	A001
C002	A002
C003	A004
C004	A004

The depositor table

Some queries and their equivalent SQL statement

Query: Find the name of customer whose customer_id C001.

```
SELECT customer. customer_name FROM customer
        WHERE customer.customer_id = ` C001';
                OR
SELECT customer_name FROM customer
      WHERE customer_id = ` C001';
```

Note: We don't need to specify the table name while referencing column_name if we are taking column from only one table.

Query: Find the name and balance of the customer.

```
SELECT customer.customer_name,account.balance
        FROM customer,account.balance
                WHERE customer.customer_id= depositor.customer_id
                    AND depositor.acount_no = account.account_no;
```

Problem : Insert record to customer table.

customer_id : C005
customer_name : MICHAEL
address : KATHMANDU

```
INSERT INTO customer (customer_id, customer_name, address)
        VALUES (`C005' , `MICHAEL' , `KATHMANDU') ;
                                OR
INSERT INTO Customer values (`C005' , `MICHAEL' , `KATHMANDU');
```

Note: Column name need not to specify if we are going to insert values for all columns of table.

Query: Delete record from depositor whose customer_id is `C004'.
:

```
DELETE FROM depositor  WHERE Customer_id = `C004';
```

What happen if we execute DELETE statement as below?

```
DELETE FROM depositor;
```

- Deletes all records from the table `depositor'

Problem: If you attempt to delete all records of customer from customer table, what happen ?

- You cannot delete all records, only when "account" and "deposit" tables are empty or only when these table contains records that are not related to the customer.

Query: Increase the balance by 5% in account table whose account no is `A101' or current balance is only 200.

```
UPDATE account
        SET balance = balance + (balance + 0.05)
WHERE account_no = `A1001' OR balance = 200;
```

Note:

Sometimes database languages are also categorized with Data Control Language.

That is

1. Data Definition Language (DDL)
2. Data Control Language (DCL)
3. Data Manipulation (DML)

Data Control language is a language that controls the behavior of database.
In SQL, COMMIT, ROLLBACK, commands go under Data Control Language.

- COMMIT: Saves the changes made to database.
- ROLLBACK: Undo changes to database from the current state database to last commit state.

Question: Why we need query language, even we have formal languages?
(formal languages, relational algebra, relational calculus)

- The formal languages provides concise notation for representing query. But commercial database system requires more user friendly query language. This is a main reason for why we need query languages, even we have formal languages.
- The formal languages form the basis for data manipulation language of DBMS only but DBMS/commercial DBMS also supports data definition capabilities as well as data manipulation capabilities.
- SQL is a most popular and powerful query language. It can do much more than just query database. It ca define structure of data, modify data in database and allow to specify security constraints.
- SQL is a truly non procedural language. It has all features of relational algebra, relational calculus as well as its own powerful features.

4.3 Different parts of SQL Language

1. Data Definition Language (DDL):

SQL DDL provides commands for defining relation schemas, deleting schema, deleting relations and modifying relational schemas.

Example:
CREATE, ALTER, DROP

```
CREATE TABLE dept
(
 dept no   NUMBER(2) PRIMARY KEY,
 dname     VARCHAR2(20) NOT NULL
);

CREATE TABLE emp
(
  empno NUMBER(5) PRIMARY KEY,
  deptno NUMBER(3),
  ename VARCHAR2(10) NOT NULL,
  sal NUMBER(5) NOT NULL,
  CONSTRAINT fk_emp_dept FOREIGN KEY(deptno) REFERENCES dept
```

);

```
ALTER TABLE dept ADD (loc VARCHAR2(10));
ALTER TABLE emp MODIFY (empno NUMBER(10));
ALTER TABLE emp ADD UNIQUE (ename);
DROP TABLE emp;
DROP constraint fk_emp_dept;
```

2. Data Manipulation Language (DML):

The SQL DML includes query language based on relational algebra and relational calculus. It includes commands for insert tuples, delete tuples and modify tuples in database.

Example: INSERT, DELETE, UPDATE, SELECT etc. statements.

3. View Definition:

The SQL DDL includes for defining views e.g.

Syntax:

```
CREATE VIEW <view name> AS
  (<query expression>);
```

4. Transaction Control: (Data Control Language):

SQL includes commands for specifying integrity constraints that the data stored in database must satisfy.

5. Embedded SQL and dynamic SQL:

Embedded SQL & dynamic SQL dynamic SQL is that SQL with general purpose programming language; such as C, C++, JAVA, COBAL, PASCAL, FORTRAN.

6. Integrity:

SQL DDL includes commands for specifying integrity constraints that the data stored in database must satisfy.

7. Authorization:

SQL DDL commands used for specifying access right to relation relations and views.

4.4 General overview of SQL:

Though SQL user / programmer / DBA can perform the following task:

- Create database.
- Modify a database structure.
- Add user permissions to database or tables.
- Changes system security.
- Query a database for a retrieval of information.
- Updates the contents of information.

Note: Commands in SQL are not necessarily a question, request to the database. It could be a command to do one of the following.

- Build or delete a table.
- Insert, Modify or delete rows or fields.
- Search several tables for specific information and returns the result in specific order.
- Modify security information.

Note: Commands in SQL are not case-sensitive. But generally conversation is write a keywords as a capital and other should be in small letter.

Data Manipulation Language in SQL:

SQL provides the following basic data manipulation statements: SELECT, UPDATE, DELETE and INSERT.
The select statements:

- The SELECT statement is most commonly used SQL statement. It is only a data retrieval statement in SQL.
- The basic syntax for select statement is

```
SELECT  [DISTINCT / ALL ] <attributes> 1
  FROM <relations> 2
[WHERE <predicate>] 3
```

1. <attribute> -> columns name
2. <relations> -> tables name
3. <predicated> -> conditions

- SELECT, FROM are necessary clause.
- WHERE is optional clause.
- DISTINCT / ALL are optional clause.
- SELECT clause used to list the attributes that required in the result in query.
- FROM clause list the relation/s from where specified attributes are to be selected.
- WHERE clause are used to specify the condition/s while we require retrieving particular data. One or more condition can be specified using where clause by using SQL logical connectives can be any comparison operators <, <=, >, >=, = and < >. SQL also includes BETWEEN comparisons.
- DISTINCT key word is used to eliminate duplicate value.
- ALL key word is used to explicitly allow duplicates.

Example: Assumed simple relational database is as follows.

customer_id	customer_name	customer_address
C001	Smith	Kathmandu
C002	John	Bhaktapur
C003	Semi	Lalitpur
C004	Ivan	Kathmandu

The customer table

account_no	balance
A101	900
A102	300
A103	200
A104	700

The account table

customer_id	account_no
C001	A001
C002	A002
C003	A004
C004	A004

The depositor table

a. Find all customer names.

```
SELECT customer_name FROM customer;
```

b. Find the different customer address (location).

```
SELECT DISTINCT customer_address FROM customer;
```

c. Find all address of customer.

```
SELECT ALL customer_address FROM customer;
```

d. Find customer_id and its corresponding customer name.

```
SELECT customer_id, customer_name FROM customer;
```

e. Find customer detail.

```
SELECT *FROM customer    (* indicates all attributes)
```

f. List name and address of customer who stay in "KATHMANDU".

```
SELECT customer_name, customer_address, FROM customer
        WHERE customer_address = "KATHMANDU";
```

g. List all customer whose name should be "smith" and address should be "Kathmandu".

```
SELECT customer_name FROM customer
        WHERE customer_name = 'smith'
          OR customer_ address  =  'Kathmandu'
```

h. What would be the output if statement like

```
SELECT customer_name FROM customer
        WHERE customer_name FROM customer
          OR customer_ address = 'Kathmandu'
```

i. List account no, balance whose balance is between 200 to 700.

```
    SELECT account_no, balance FROM account
        WHERE balance BETWEEN 200 AND 700;
```

j. What happen if we execute the statement?

```
    SELECT account_no, balance FROM account
        WHERE balance NOT BETWEEN 200 AND 700;
```

k. Write SQL statement for (i) using only AND logical connectives and comparison operatives.

```
  SELECT  account_no, balance FROM account
        WHERE balance <= 700 AND balance >= 200;
```

Note: We can retrieve the information from multiple tables; there should be a common attribute between two tables. i.e., table should be related and we require to join condition.

l. List the customer_id, account_id and balance whose balance is more than 300.

```
SELECT depositor. customer_id, depositor. account_no, account. balance
    FROM depositor, account
        WHERE depositor.account_no = account.account_no
            AND account.balance > 300;
```

m. List all customers and corresponding balance.

```
SELECT c.customer_name, a.balance
            FROM customer   c, account    a, depositor    d
        WHERE d.account_no =a.account_no
            AND d.customer_id =c.customer_id;
```

Renaming attribute and relations:

- In previous example relations customer, account, depositor are renamed respectively c, a and d.

- We can also rename attributes, it is required when we are taking attribute from multiple column or we need arithmetic operations in the statement or when we need to give appropriate name for (column name) attribute name.
- To rename attribute SQL provides as clause or we can simply rename attribute or relation without as clause.

Examples:

a. SELECT account_no as account number FROM account;

OR

SELECT account_no "Account number" FROM account;

b. SELECT account_no, balance, balance+ (balance*0.05) as Account Number, Balance,
Increase salary FROM account;

OR

SELECT account_no "Account number", balance "Balance:,
Balance+(balance*0.05) "Increased salary" FROM account;

c. SELECT c. customer_name, a. balance
FROM customer c, account a, depositor d
WHERE d. account_no = a. account_no
AND d.customer_id = c.customer_id;

OR

SELECT c. customer_name, a. balance
FROM customer as c, account as a, depositor as d
WHERE d.account_no = a.account_no
AND d.customer_id = c.customer_id;

String operations:

String pattern matching operation on SQL can be performed by `like' operator and we can describe the patterns by two spherical character.

1. Percent (%) : matches any substrings.
2. underscore (_): matches any characters.

Example:

`I%' matches any string beginning with I.
IVAN -> valid / match.
INDIA -> valid / match
NEPALI -> invalid / does not match.

` % VAN% match any string containing `VAN' as substring.
IVAN, Mr IVAN, DEVAN, DEVANGAR are all valid.
`---' matches any strings of exactly three character.
`---%' matches any string of atleast three character.

Moreover,

Like `ab\%cd%' matches all string beginning with "ab%cd".
Like `ab\\cd%' matches all string beginning with "ab\cd".

Problems:

List those customers whose name begin with character `S'

```
SELECT customer_name FROM customer.
        WHERE customer_name LIKE `S%';
```

Note:

customer_name like `S%H'

- List all customer name whose name begin with `S' and end with `H'

customer_name like`----'%N'

- List all customer name whose name must contain at least five character and end with character `N'

Ascending and descending records in SQL:

- ORDER BY clause used for ascending or descending records(or list items).
- To specify sort order we may specify desc for descending order or asc ascending orders. By default order by clause list item in ascending order.
- Moreover, ordering can be performed on multiple attributes.

Example: 1

```
SELECT distinct customer_name FROM customer ORDER BY customer_name;
```

- Lists name of customer in alphabetic order by customer name.

Example: 2

```
SELECT DISTINCT customer_name FROM customer ORDER BY customer_name
DESC;
```

- Lists name of customer in descending alphabetic order.

Note: select from customer order by 2;

Here 2 indicates second column in table "customer". This SQL statement is equivalent to first example's SQL statement.

Example: 3

Suppose want list account information in descending order by balance but if say some balance are same and in such case if we want to order account information by order_no in ascending order then we have order record by performing ordering on multiple attributes. The SQL statement likes,

```
SELECT *FROM account
    ORDER BY balance DESC, account ASC;
```

Set operation

- basic set operation are union(u), intersection(n) and difference(_). These operation also can be performed by using union, intersection and minus (except) clause respectively.

The union operation

- the union operation can be perform by using union clause.

Example: consider two reactions

client_id	name
C001	Ammit
C002	Ajay
C003	Rohit
C004	Ammit

The table client

supplier_id	name	city
S001	ASHOK	Kathmandu
S002	MANISH	Bhaktapur
S003	MANOJ	Kathmandu
S004	MANISH	Bhaktapur

The table supplier

List the id and name of the client and supplier who stay in city 'Kathmandu'.

```
Select supplier_id  "ID", name "Name" from supplier
        where city = 'Kathmandu'
UNION
SELECT client_id "ID", name "Name" from client
       where city = 'Kathmandu'
```

- Proceed as follow:
  Output from 1st SQL statement

ID	Name
S001	Ashok
S002	Manoj

Output from 2nd SQL statement

ID	Name
C001	Ammit
C002	Ammit

Hence, the resulting output is

ID	Name
C001	Ammit
C002	Ammit
C003	Ashok
C004	Manoj

Note: if we retrieve only one column say name without duplicate name then corresponding statement like

```
Select name from supplier where city = 'Kathmandu'
UNION
Select name from client where city = 'Kathmandu';
```

- this is unlike select clause, union operation automatically eliminates duplicates. If we want to retain all duplicates, we must replace union all.

Example:
select name from supplier where city = 'Kathmandu'.

The output would be

Name
Ammit
Ammit
Ashok
Manoj

The intersection operation

- the intersection operation can be performed by using INTERSECT clause
- consider relation as follow:

SALESMAN

salesman_id	name	city
S001	Manish	Kathmandu
S002	Manoj	Lalitpur
S003	Ammit	Bhaktapur
S004	Rabin	Kathmandu

The salesman table

order_no	Order_date	salesman_id
0001	10-JAN-98	S001
0002	12-FEB-98	S002
0003	13-FEB-98	S001
0004	18-MAR-98	S001
0005	19-MAR-98	S002

The sales_order table

Retrieve salesman name who stay in Kathmandu and who must sales at least order.

```
SELECT salesman_id, name
    from salesman
        where city = 'Kathmandu'

INTERSECT
 SELECT salesman. salesman_id
    from salesman, sales_order
        Where salesman_id = sales_order. salesman_id;
```

salesman_id	name
S001	Manish
S004	Rabin

=

salesman_id	name
S001	Manish
S002	Manoj
S001	Manish
S002	Manoj

The resulting output is

salesman_id	name
S001	Manish

- The INTERSECT operation also automatically eliminates duplicates. So, here only one record is delayed in output. If we want to retain all duplicates we must replace INTERSECT by INTERSECT ALL.

```
SELECT salesman_id, name from SELECT salesman
    where city = 'Kathmandu'
INTERSECT ALL
SELECT salesman. salesman_id, salesman name
        from salesman, sales_order
WHERE salesman. salesman_id = sales_order salesman_id;
```

The difference operation

The difference operation can be perform in SQL by using except or minus clause.

Example: in previous example, find the salesman_id, name who stay in Kathmandu but they do not sales any order.

```
SELECT salesman_id, name from salesman
   where city = 'Kathmandu'
EXCEPT
SELECT salesman. salesman_id, salesman name
   from salesman, sales_order
        Where salesman. salesman_id = sales_order. salesman_id;
                  OR
SELECT salesman_id, name from salesman
     where city = 'Kathmandu'
MINUS
SELECT salesman. salesman_id, salesman name
       FROM  salesman, sales_order
            Where salesman salesman_id = sales_order salesman_id;
```

The output is

salesman_id	name
S004	Rabin

NOTE: Except operation also automatically eliminates duplicates so it want to return all duplicates, we must write EXCEPT ALL instead of EXCEPT.

Problem: consider a relation schema as follow

```
branch(#branch_name, branch_city, assets)
account(#account_number, branch_name, balance)
customer(#customer_name,  customer_street, customer_city)
depositor(customer_name, account_number)
loan(#loan_number, branch_name, amount)
brrower(customer_name, loan_number)
```

1. Find all customer who have a loan, account or both at the bank .

```
SELECT customer nameFrom depositor
 UNION
SELECT customer name  From borrower;
```

2. Find all customers who have both loan and account at the bank.

```
SELECT DISTINCT customer name from depositor
     INTERSECT
SELECT DISTINCT customer name from borrower;
```

3. Find all customers who have account but no loan at the bank.

```
SELECT DISTINCT customer name from depositor
     EXCEPT
SELECT customer name from borrower;
```

Aggregate Function

Aggregate functions are those functions that take a set of values as input and return a single value. SQL consist many built in aggregate function. Some are:

1. AVERGE: AVG
2. MAXIMUM: MAX
3. MINIMUM: MIN
4. TOTAL: SUM
5. COUNT: COUNT

The input to AVG and SUM must be a set of numbers and other aggregate function can be operate by non numeric data types, it may be strings, not necessary numbers.

AVG:

Syntax: AVG (<DISTINCT\ALL>; n)

- returns average of n, ignoring null values.

Example: find the average balance in Kathmandu branch.

```
SELECT AVG (balance)  From account
   WHERE branch_name = 'KATHMANDU';
```

There would be a situation that we may have to use aggregate function not only a single set of tuples we may have to use with group of set of tuples, we can specify this by using GROUP BY clause in SQL. That is, group by clause specifies group rows based on distinct values that exist in specified column when we use GROUP BY clause we can not use WHERE clause to specify condition, we must have to use HAVING clause to specify the condition. That is, GROUP BY and HAVING clause. But GROUP BY or HAVING clause act on record sets rather than individual records.

Example: find the average account balance at each branch

```
SELECT branch_name , avg (balance)
    FROM  account
GROUP BY branch_name;
```

MIN:

Syntax: MIN (<DISTINCT\ALL>; n)

- returns minimum value of n.

Example: find the minimum balance of each branch.

```
SELECT branch_name, min (balance) "Minimum Balance"
   FROM account
GROUP BY branch_name;
```

MAX:

Syntax: MAX (<DISTINCT ALL>; n)

- returns maximum value of n

Example: find the maximum balance in each branch.

```
SELECT branch_name, max (balance) "Maximum Balance"
      FROM account
```

GROUP BY branch_name;

COUNT:

Syntax: COUNT (<DISTINCT ALL>; n)

- returns numbers of rows where n is not null.

NOTE: COUNT (*)

- returns numbers of rows in the table, including duplicates and those with nulls.

Example: find the numbers of depositor for each branch.

NOTE: each depositor may have numbers of account so we must count depositor only once thus we write query as below:

```
SELECT branch_name, count (DISTINCT Customer_name)
    From depositor, account
        WHERE depositor account number = account. Account_number
            GROUP BY branch_name;
```

NOTE: if we need to specify the condition (predicates) after GROUP BY clause, we need having clause.

PROBLEM: find only those branches where the average account balance is more than 1200.

```
SELECT branch_name, AVG (balance) FROM account
 GROUP BY branch_name having AVG (balance) > 1200;
```

PROBLEM: find the no. of customer in customer table.

```
SELECT COUNT (*) FROM Customer;
```

NOTE: If a WHERE clause and HAVING clause both appears in the same query, SQL executes predicate in the WHERE clause first and if it satisfied the only it executes predicate of GROUP BY clause.

PROBLEM: Find the average balance for each customer who lives in KATHMANDU and has at least three accounts.

```
SELECT depositor customer_name, AVG (balance)
    FROM depositor, account , customer
        WHERE depositor account_number = account account_number
            Depositor customer_name = customer customer_name
            Customer_city = 'KATHMANDU'
GROUP BY depositor, customer_name
   HAVING COUNT (DISTINCT depositor account_number) > = 3;
```

SUM

Syntax: SUM ([DICTINCT/ ALL] n)

- return sum of value of n

Example: find total loan amount for each branch

```
SELECT branch_name, SUM (amount)
    FROM loan
GROUP BY branch_name;
```

NULL VALUES

- In SQL, NULL values all to indicate absence of information for the value of attribute.
- SQL provides special key word NULL in a predicate to test for NULL values. Not NULL in predicate use to test absence of NULL values.

Example: find the balance of each branch whose balance is not empty.

```
SELECT branch_name, balance
        FROM account
WHERE balance is NOT NULL;
```

Example: List the account number, branch name and balance whose balance is empty.

```
SELECT *FROM account
    WHERE balance is NULL;
```

- The feature of SQL that handles NULL values has important application but some time it gives unpredictable result. For example, an arithmetic expression involving (+. _ , * or /), if any of the input values is NULL value (except IS NULL Q IS NOT NULL).
- If NULL values exist in the processing of aggregate operation, it makes process complicated.
  Example: SELECT AVG (amount) FROM loan;
- This query calculates the average loan amount that is not empty so if there exist empty amount then calculated average amount is not valid. Except COUNT(*) function all aggregate function ignores NULL values in input.
- The COUNT ( ) function return if count value is empty and all other aggregate function returns NULL if it found empty value.

Example: consider a table 'emp' as below:

Empno	Sal	Comm
10	100	
20	200	50
30	300	20

Suppose the SQL statement are as below

a. SELECT COUNT (*) FROM emp;
   returns count is equal to 3
   That is count(*)

   3

b. SELECT COUNT (comm.) FROM emp WHERE empno = 10;
   returns count(comm.)

   0

c. SELECT SUM(COMM) FROM emp WHERE empno = 10;
   return sum(comm.)

   (nothing)

d. SELECT SAL+COMM FROM emp WHERE empno = 10;

```
return          sal+comm.
               -------------
                 (nothing)
```

- here the result is unpredictable. In this case result should be 100. To handle such unpredictable situation SQL provides NVL ( ) function
  Example: SELECT sal+nvl(comm,0)/100

- nvl function returns 0 when comm is found empty. If user do not specify the value for any column (attribute) SQL place null values in these columns. The null value is different from zero. That is null value is not equivalent to value zero.
- A NULL value will evaluate to NULL in any expression.
  Example: NULL multiply by 10 is NULL.
- If the column has a NULL value, SQL ignores the unique Foreign key, check constraints that are attached to the column.
- If any field define as NOT NULL, it does not allow to ignore this field, user must insert value, that is NOT NULL is itself a constraint while it specify in table.

Nested Subqueries

- SQL provides sub_query facility. A sub_query is a SQL statement that appears inside another SQL statement. It is also called nested sub_queries or simply nested query.
- Sub_query use to perform tests for set membership, make set comparisons and determine set cardinality.
- Sub_query can be used with SELECT, INSERT, UPDATE and DELETE statement.

Example: find all branch name where depositor account number is ‘A005’ .

```
SELECT branch name FROM account
      WHERE account account_number = (SELECT account_number FROM
depositor
                                          WHERE account number
= ‘A005’);
```

- When sub_query return more than one values the we required to test whether value written by first query is match/exist or not within values return by sub_query.
- IN and NOT IN connectives are useful to test in such condition. That is IN connectives use to test for set membership and NOT IN connectives use to test for absence of set membership.

Example: find those customers who are borrowers from the bank and who are also account holder.

```
SELECT DICTINCT customer_name FROM borrower
   WHERE customer_name IN (SELECT customer_name FROM
depositor);
```

Example: find all customer who do have loan at the bank but do not have an amount at the bank.

```
SELECT DISTINCT customer_name FROM borrower
  WHERE customer_name NOT IN (SELECT customer_name FROM
depositor);
```

Example: list the name of customers who have a loan at the bank and whose name neither SMITH nor JONE

```
SELECT DISTINCT Customer_name FROM borrower
  WHERE customer_name NOT IN ('SMITH', 'JONE');
```

Example: find all customers who have both an account and loan at Kathmandu branch.

```
SELECT DISTINCT Customer_name FROM borrower, loan
  WHERE borrower loan number = loan loan_number and
branch_name = 'KATHMANDU'
      AND (branch_name, customer_name) IN (SELECT
branch_name, customer_name
                FROM depositor account
                  WHERE depositor account_number = account
account_name);
```

SET COMPARISION

Nested sub_query have an ability to compare set.

Example: Find the names of all branches that have assets greater than those of at least one branch located in 'Kathmandu'.

The simple SQL statement is

```
SELECT DISTINCT B1 branch_name FROM branch B1. branch B2
  WHERE B1 assets>B2 assets
    AND B2.branch_city = 'Kathmandu'.
```

The same query can be written by using subquery as

```
SELECT branch_name FROM branch
  WHERE assets > some (select assets FROM branch
                       WHERE branch_city = 'Kathmandu');
```

- here, >some comparison in the where clause of the outer value return by sub_query.
- SQL also allow <some, >=some, =some and < > some comparison
- Not that = some is identical to IN. but < > some is not same as NOT IN.

Example: Find the names of all branches that have an assets value greater than that of each branch in 'KATHMANDU'.

NOTE: The construct > all corresponds to the phase 'greater than all'

```
SELECT branch_name FROM branch
  WHERE assets > all (SELECT assets FROM branch
                      WHERE branch city = 'KATHMANDU')
```

NOTE: SQL also allow <all, <=all, >=all, =all and < >all comparison.

Example: find the branch that has the highest average balance.

NOTE that we can not use MAX {AVG (balance)}, since aggregate function can not be composed in SQL. So we first need to find all average balances and need to nest

it as subquery of another query that finds those branches for which average balance is greater than or equal to all average balances.

```
SELECT branch_name FROM account
    GROUP BY branch_name
  HAVING AVG (balance) >=all (SELECT AVG (balance) FROM account
                                                    GROUP BY
branch_name);
```

TEST EMPTY RELATIONHIP

SQL has a feature for testing whether a subquery return any value or not. The exists construct returns true if subquery returns values.

Example: find all customers who have both an account and loan at the bank.

```
SELECT customer_name FROM borrower
WHERE exists (SELECT *FROM depositor
                    WHERE depositor customer_name = borrower
customer_name);
```

We can test non existence of values (tuples) in sub_query by using not exists construct.

Example: find all customers who have an account at all branches located in 'KATHMANDU'.

```
SELECT DISTINCT d1.customer_name
  FROM depositor as d1
WHERE not exists {(SELECT branch_name FROM branch
                        WHERE branch_city = 'KATHMANDU')
Except
 (SELECT d2. branch_name FROM depositor as d2, account as a
            WHERE d2. Account–no. = a. Account_no.
                AND d1. Customer_name = d2.
Customer_name)};
```

Test for the absence of Duplicate Tuples

SQL has a feature for testing whether the subquery has any duplicate Tuples in its results.
The UNIQUE construct true if a subquery contains no duplicate Tuples.

Example: find all customers who have at most one account at the KATHMANDU branch.

```
SELECT d.customer_name FROM depositor d1
    WHERE UNIQUE (SELECT d2. customer_name FROM account depositor d2
                WHERE d1. customer_name = d2. customer_name
                  AND d2. account_no. = account. Account_no.
                    AND account. Branch_name = 'KATHMANDU');
```

NOTE: using NOT UNIQUE construct, we can test the existence of duplicate tuples.

Example: find all customers who have at least two account at the KATHMANDU branch.

```
SELECT DISTINCT d1.customer_name FROM depositor as d1
      WHERE UNIQUE (SELECT d2. customer_name FROM account depositor d2
                          WHERE d1. customer_name = d2. customer_name
                            AND d2. account_no. = account. Account_no.
                              AND account. Branch_name = 'KATHMANDU');
```

Complex Queries

There are several way of composing query: derived relation and the with clause are ways of composing complex queries.

Derived Relation

- SQL have a feature that it allow sub_query expression to used in the FROM clause.
- If we use such expression then we must give result relation name and we can remove the attributes.

Example: find the average account of those branches where the average account balance is greater than 1200.

```
SELECT branch_name, avg_balance
    FROM {SELECT branch_name, avg (balance) FROM account GROUP BY
branch_name}
AS branch_avg (branch_name, avg_balance)
     WHERE avg_balance > 1200;
```

The with clause

- The with clause introduced in SQL: 1999 and is currently supported by only some database.
- The with clause makes query logic clear.

Example: find all branches where the total account deposit is less than the average deposits at all branches.

```
WITH branch_total (branch_name, value) as
 SELECT branch_name, SUM (balance) FROM account
     GROUP BY branch_name
  WITH branch_total_avg (value) as
      SELECT avg (value) FROM branch_total
   SELECT branch_name FROM branch_total, brnch_total_avg
     WHERE branch_total.value >= branch_total_avg.value;
```

UPDATE STATEMENT

- The UPDATE statement is used to modify one or more records in specified relation. The records to be modify are specified by a predicate in the WHERE clause and new value of the column (s) to be modified is specified by a SET clause.

The syntax is

```
UPDATE <relation>
    SET <attribute with new value>
[WHERE <predicate>];
```

Example: increase the balance of all branches by 5%

```
UPDATE account
```

```
SET balance = balance * 1.05;
                OR
UPDATE account
        SET balance = (balance) + (balance * 0.05);
```

Example: increase balance of those branches whose current balance is less than or equal to 1000 by 5%

```
UPDATE account
    SET balance = balance * 1.05
  WHERE balance <= 1000;
```

Example: decrease the balance by 5% on accounts whose balance is greater than average.

```
UPDATE account
SET balance = balance_ (balance * 0.05)
WHERE balance > {SELECT average (balance) FROM account};
```

NOTE: SQL provides case construct, which can be perform multiple updates with a single UPDATE statements

The syntax is:

```
Case
When predicate 1 then result 1
When predicate 2 then result 2
When predicate n then result n
        Else result
END
```

Example:

```
UPDATE account
    SET balance = case
                When balance <= 1000then balance *1.05
                else balance *1.06
                end;
```

DELETE STATEMENT

The DELETE statement use to delete one or more records from relations. The records to be deleted are specified by the predicate in the WHERE clause.

The Syntax is:

```
DELETE <relation> [WHERE <Predicate>]
```

Note: Delete statement can operate only one relation. It can not delete records of multiple relation.

Example: Delete all records from loan relation.

```
DELETE FROM loan;
```

Example: Delete all records from account relation whose branch is located in Kathmandu.

```
DELETE FROM account
WHERE branch_name = 'KATHMANDU;
```

Example: Delete all loan with loan amount between 1000 and11500;

```
DELETE FROM account
WHERE branch_name IN (SELECT branch_name FROM branch)
                      WHERE branch_city = 'KATHMANDU');
```

Example: Delete all records of account with balance below the average

```
DELETE FROM account
WHERE balance <(SELECT avg (balance) FROM account);
```

INSERT STATEMENT

The INSERT statement used to insert a new records into a specified relation

Syntax:

```
INSERT INTO <relation>
    VALUES (<values list>)
```

Another form:

```
INSERT INTO <relation> (<target columns>)
    VALUES (<values list>)
```

The attributes values that are going to insert must be match order by corresponding attributes and also must be matched data type of value and corresponding attribute data type.

Example: insert one record in account relation.

```
INSERT INTO account
  VALUES ('A_0009', 'LALITPUR', 1500);
                  OR
   INSERT INTO account (account_no., branch_name, balance )
 VALUES ('A_0009', 'LALITPUR', 1500);
```

we can also insert records on the basis of query

Example: INSERT INTO account

```
        SELECT loan_number = 'KATHMANDU';
```

It inserts the records in account relation taking account_number, brancg_name from loan relation whose branch is located in Kathmandu and for all records balance is constant 500.

Example: INSERT INTO depositor

```
SELECT customer_name, loan_number FROM borrower, loan
 WHERE borrower . loan_number = loan . loan_number
   AND branch_name = 'KATHMANDU';
```

Insert Tuple (customer_name, loan_number) into the depositor relation for each customer who has a loan in Kathmandu branch with loan number.

Example: INSERT INTO account

```
    SELECT *FROM account;
```

Insert infinite number of Tuples.

Joined Relations

- one of the most powerful feature of SQL is its capability to gather and manipulate data from several relations.

- If SQL does not provides this feature we must have to store all the data elements in a single relations for each application. We have to store same data in several relations.
- The join statements of SQL enables to design smaller, more specific relations that are easier to maintain than larger relations.
- There are several methods for joining relations. Some methods are not useful for application and some are very useful.

1. Cross Join

Joins two or more tables without relation between them or without condition in the where clause. The results is the Cartesian product of two or more table's attributes

Examples: consider two relations

Table 1:

row	remarks
1.	Table 1
2.	Table 1

Table 2:

row	remarks
1.	Table 2
2.	Table 2

The cross join statement is

```
SELECT *FROM Table 1,Table 2;
```

Output is:

row	remarks	row	remarks
1	Table 1	1	Table 2
1	Table 1	2	Table 2
2	Table 1	1	Table 2
2	Table 1	2	Table 2

- cross join is normally not useful but it illustrates the basic combining property of all join types.

Equi Join (natural inner join)

Equi joins methods joins relations based on the equality. It has very important application in commercial database application.

Example: consider

Relation : dept

#empno	E name	job	sal	comm	Dept. no
E001	SMITH	Manager	7000	500	10
E002	JONE	Engineer	6000	3000	20
E003	MICLE	Engineer	5000	2000	20
E004	JACK	Accountant	3000	500	40

relation: emp

#dept. no	Depo. name	Loc
10	Management	Kathmandu
20	Technical	Kathmandu
30	Marketing	Bhaktapur
40	Account	Lalitpur

Example: list all employee name, department name and salary

```
SELECT e. ename, d. dname FROM emp e, dept d
      WHERE  e. deptno = d. deptno;
```

Output:

ename	Dname
SMITH	MANAGEMENT
JONE	TECHNICAL
MICLE	TECHNICAL
JACK	ACCOUNT

We can further qualify this query by adding more condition on where clause.

```
Example: SELECT e.ename, d.dname
              FROM emp e , dept d
       WHERE e.deptno = d.deptno
            AND e.sal >5000       ORDER By e.ename;
```

Example: consider relations

Loan_number	Depo. name	Loc
L_170	Management	Kathmandu
L_230	Technical	Kathmandu
L_260	Marketing	Bhaktapur

Relation: loan

Customer_name	Loan_nuber
JONE	L_170
SMITH	L_230
MICLE	L_155

relation: borrower

```
SELECT *FROM loan, borrower
  WHERE loan. Loan_number = borrower.loan_number;
```

Non – Equi join (Outer join)

SQL also supports non equi_join. That is, this method joins tables (relations) based on non equality. But equi_join is far more common than non equi_join

```
e.g. Select e.ename, d.dname, d.dname d.deptno
                FROM emp e, dept d
       WHERE e.deptno > d.deptno;
```

Output:

Ename	dname	dept no
JONE	MANAGEMENT	10
MICLE	MANAGEMENT	10
JACK	MANAGEMENT	10
JACK	TECHNICAL	20
JACK	MARKETING	30

Here, this information is not so useful.

How from the output? Lets look

```
Case 1:  equi_join
  SELECT e.ename, d.dname, e.deptno "emp deptno", d.deptno "deptno"
    FROM emp e, dept d
  WHERE e.deptno;
```

OUTPUT:

ename	dname	emp deptno	deptno
SMITH	MANAGER	10	10

JONE	TECHNICAL	20	20
MICLE	TECHNCAL	20	20
JACK	ACCOUNT	40	40

REMARKS: output is selected from the Cartesian product of two relations
Emp (e.ename, e.deptno)
Dept (d.dname, d.deptno)
such that both department no is equal only.

Case 2: (Non equi join)

```
SELECT e.ename, d.dname, e.deptno "emp deptno", d.deptno "dept deptno"
    FROM emp e, dept
        WHERE e. dept> d.dept no;
```

Output:

ename	dname	empdeptno	detno
JOHN	Management	20	10
MICHAEL	Management	20	10
JACK	Management	40	10
JACK	Technical	40	20
JACK	Markeing	40	30

..

Remarks:

Output is seleted from the Cartesian product of two relations
emp (e.ename, e.dept no)
Dept (d.dname, d.deptno)
Such that emp tanle of department no is greater than department table of department no.

Case 3 (Non equi join)

```
SELECT e.ename , d.dname  e.dept no "emp deptno"
    FROM emp e ,  dept d
WHERE e. deptno> 10;
```

Output:

ename	dname	Emp deptno	
JONE	MANAGEMENT	20	>10
MICLE	MANAGEMENT	20	>10
JACK	MANAGEMENT	40	>10
JONE	TECHNICAL	20	>10

REMARKS: output is selested from the Cartesian product of two relation
Emp (e.ename, e.deptno)
Dept (e.ename, e.deptno)
Such that emp table of deptno is always greater than 10.

Three types of outer join

1. Left outer join
2. Right outer join
3. Full outer join

Consider two relations

Loan

Loan_no.	Branch_name	amount
L-170	KATHMANDU	3000
L-230	BHAKTAPUR	4000
L-260	LALITPUR	1700

Borrower

Customer_name	Loan_number
JONE	L-170
SMITH	L-230
MICLE	L-155

a. SELECT loan . loan_number, loan . branch_name, loan . amount, borrower . loan_number loan

Left outer join borrower on loan . loan_number = borrower . loan_number

Loan_no.	Branch_name	amount	Customer_name
L-170	KATHMANDU	3000	JONE
L-230	BHAKTAPUR	4000	SMITH
L-260	LALITPUR	1700	null

Remarks: Tuple from the left hand side relation that do not math any Tuple in the right side relation are padded with null.
SELECT loan. loan_number, loan . brach_name, loan.amount, borrower. Customer_name loan
right outer join borrower on loan.loan_number = borrower. Loan_number;

Loan_no.	Branch_name	amount	Loan_no.	Customer_name
L-170	KATHMANDU	3000	L-170	JONE
L-230	BHAKTAPUR	4000	L-230	SMITH
L-155	null	null	null	MICLE

Remarks: Tuples from the right hand-side that do not match any Tuple in the left hand side relation are padded with nulls.

SELECT loan. loan_number, loan. branch_name, loan. amount, borrower customer_name loan
Full outer join burrower on loan.loan_number = borrower. Loan_number;

loan number	branch name	amount	customer name
L - 170	Kathmandu	3000	JONES
L - 230	Bhaktapur	4000	SMITH
L - 260	Lalitpur	1700	NULL
L -150	NULL	NULL	MICHALE

Self join

- In some situation, we may need necessary to join a table to itself as we are joining two separate tables, this is known as self-join
- In a self join two rows from the same table combine to form a result row.
- Example: Consider a relation `employee' as below.

Empno	**name**	**manager no**
E001	Smith	E002
E002	Michale	E005
E003	John	E004
E004	Ivan	
E005	Scott	

Retrieve the names of employees and the names of their respective manager from the employee relation.

```
SELECT emp name , mgr name   "Manager"
      FROM  employee  emp,  employee mgr
            WHERE emp. Manager no = mgr. emp_no;
```

Output:

Name	Manager
Smith	Michael
Michael	Scott
John	Ivan

Process:

EMP

Emp no	name	manager no
E001	Smith	E002
E002	Michael	E005
E003	John	E04

MGR

emp no	name	manager no
E001	Michael	E002
E002	Scott	E005
E003	Ivan	E004

Name	Manager
Smith	Michael
Michael	Scott
John	Ivan

Data definition Language in SQL

- In SQL, DDL specifies set of relations (tables) in a database.
- SQL DDL also allows to specify
  - Integrity constraints.
  - Index on relations
  - Security and authorization for each relation
  - Physical storage structure of each relation

- Basic statement in Data Definition Language are CREATE, DROP, ALTER

Some Domain types in SQL (Data type in SQL)

CHAR(n): fixed length character string with user specified length n.

VARCHAR2(n) : A variable length character string with user specified length n. Full form is character varying and to indicate version of the domain.

NUMBER(n): holds fixed number specified length n.

NUMBER(P,S): holds fixed or floating point numbers

P determines the maximum length of data, and S Determines the number of places to the right of decimal.

If S is not specified then default is zero ; in such case or specified 0, it can not hold floating point number.

INT : An integer , small int

FLOAT(n) : A floating point number , with precision of at least n digits.

DATE : Represents date and time. The standard format is DD-MM-YY

LONG: Used to store variable length character strings containing up to 2 GB.

RAN/ LONG RAW : Used to store binary data such as digitized picture or image. It can contain up to 2 GB.

Schema Definition in SQL

- CREATE TABLE command is usedto create relation

Syntax :

```
CREATE TABLE <relation name>
(       A1D1 , A2D2, ..... , AnDn ,
        [< integrity constraint 1>]
        [< integrity constraint K>]
);
```

- Here, Ai is the name of attributes.
- Di is the domain type (or data type)

And integrity constraint includes:

PRIMARY KEY :

Primary key is an attribute or combination of multiple attributes that uniquely identifies records.

If a primary key is a combination of multiple attributes called composite primary key. A primary key attributes are required NOT NULL AND UNIQUE. That is primary key attribute cannot be left null and it cannot contain duplicate values.

NOT NULL / UNIQUE: Attribute can be specified NOT NULL attribute or unique attribute.

FOREIGN KEY: Any column (attribute) of table (relation) can be specified as a foreign key if it is a common attribute between relations where we are going to establish a relationship.

- In one relation (master table) it should be primary key and in another table (detail table) some attribute should be foreign key.
- Primary key and foreign key together used to establish the relationship between the two relations.
- The concept of primary key and foreign key is very important in RDBMS.

CHECK(P) : Check clause specifies the predicate P that must satisfy specified condition.

Example: SQL data definition for the simple banking database.

```
CREATE TABLE customer
        Customer_ name VARCHAR2(20) NOT NULL,
        Customer location VARCHAR2 (20)
        Constraint PK_Cname Primary key (Customer name));

CREATE TABLE branch
 (
   branch_name VARCHAR 2(15),
   branch_city VARCHAR2(30) DEFAULT " KATHMANDU" ,
   assets NUMBER (5) ,
   CONSTRAINT PK_branch_name PRIMARY KEY (branch_name),
   CONSTRAINT ch_accbal CHECK (balance >=0)
);
```

```
CREATE TABLE depositor
 (
  customer_name  VARCHAR2(20),
  account no  CHAR(10)
  CONSTRAINT fk_depositor_cname
         FOREIGN KEY(customer_name)  REFERENCES  customer,
  CONSTRAINT PK_cname_accno PRIMARY KEY (custome_name, account no)
 );

CREATE TABLE loan
 (
   loan_no CHAR(10) PRIMARY KEY ,
   branch_name  VARCHAR2(15) NOT NULL,
   amount NUMBER (5),
   CONSTRAINT fk_loan_branch_name
              FOREIGN KEY  (branch_name) REFERENCES branch,
   CHECK (amount>=0)
 );
CREATE  TABLE borrower
(
 customer_name    VARCHAR2(20),
 loan_no             CHAR(10),
 CONSTRAINT  fk_ cname  FOREIGN KEY (customer_name) REFERENCES customer
);
.
```

Example of using UNIQUE key and DEFAULT value

```
 e.g.  CREATE TABLE student
       (
         student_id  NUMBER (3) PRIMARY KEY,
         name        CHAR  (20)  UNIQUE,
         degree      CHAR (15)   DEFAULT ` Master' ,
         CHECK   (degree IN ( `Bachelors' , `Master' , `Doctorate'))
       );
```

Drop statement

- Drop table statement used to drop the relation.

Syntax:

```
DROP TABLE  <relation name>;
```

DROP user statement

- DROP USER statement is used to drop the user.

Syntax:

```
DROP USER  <user name>  [USER CASCADE];
```

ALTER TABLE statement

- Alter Table command is used to add or modify attributes to the existing relation.

Syntax:

```
ALTER TABLE <relation> ADD (attribute domain type);
```

```
ALTER TABLE <relation> MODIFY (attribute domain type);
ALTER TABLE <relation>  DROP CONSTRAINT  <constraint_name>;
ALTER TABLE <relation>  DROP  COLOUMN  <coloumn_name>;
```

Examples:

```
ALTER TABLE customer   ADD PRIMARY KEY (customer_name);
ALTER TABLE customer   ADD (customer_adds  VARCHAR2(23));
ALTER TABLE customer   MODIFY (customer_adds  VARCHAR2(32) NOT NULL);
ALTER TABLE customer  DROP PRIMARY KEY;
ALTER TABLE customer   DROP CONSTRAINT fk_cname;
ALTER TABLE customer  DROP COLOUMN customer_adds
```

View

- View is a virtual table, it does not contain actual data it is map to the base table/s.
- When tables are created or populated with data, we may require to prevent all user from accessing all columns of table. So for the data security reasons view are created.
- One alternative solution is to create several table having appropriate no of columns and assigned each user to each table. This provides well data security but it keeps redundant data in tables. So it is not useful practically. So, views are generally created instead of it . It reduces redundant data.
- View can be created from a single table hiding some coloumn/s or from the multiple tables mapping all or some of the columns of the base tables. View is the simple and effective way of hiding columns of tables for security reason.
- When view is referenced then only it holds data, so it reduces redundant data.
- When view is used to manipulate table , the underlying base e table/s are completely invisible. This adds level of data security.
- Since view can be created from multiple tables so it makes easy to query multiple tables because we can simply query views instead of query multiple tables.
- View may be read only or updateable view. Read only view only allow to read data from view. Updatable view allow insert, update and delete on view.
- If view is created from multiple tables it won't be updateable.
- If view is created without primary key and null columns then value/record can be inserted in view.
- The general syntax for view is

```
CREATE VIEW <view name>  AS  < query expression> ;
```

Example: creating view from single table.

```
CREATE VIEW   vw_emp   AS
      SELECT empno,  ename, job FROM emp;
  View coloumn can be renamed as below:
  CREATE VIEW vw_emp  AS
      SELECT empno "Employee no", ename " Employee name"
                                    Job "work" FROM  emp;
```

- Creating view from multiple tables.

```
CREATE VIEW vw_emp_info AS
      SELECT e.empno, e.ename, e.ejob, d.dname
            FROM emp e, dept d
      WHERE e.empno = d.dept no AND  e.sal>1000;
```

Common restrictions on view

- We cannot use delete statement on multiple table view.
- We cannot use insert statement unless all NOT NULL columns On underlying table are included.
- View must be created from single table to allow insert or update on view.

- If we use DISTINCT clause to create views, we cannot update or insert records within that view.

Common application of views

- Provides user security functions.
- Simplifies the constructions of complex queries.
- Summarize data from multiple tables.

Common restrictions on updatable views

- For the views to be updateable, the view definitions must not include.
- Aggregate function.
- DISTINCT, GROUP BY or HAVING clause.
- Sub – queries.
- Constraints, strin or value expression like bal*1.05
- UNION, INTERSECT, OR MINUS/EXCEPT clause.
- View can be destroyed by using DROP VIEW command.

Syntax:

DROP VIEW <view name>;
e.g. DROP view vw_emp;

Transactions:

- A transaction consists of sequence of query / or updateable statements.
- SQL standard specifies, the transaction begins implicitly when SQL statement is executed and one of the following SQL statement must end with transaction commands.

COMMIT:

It commits (save) the current transaction changes on database / table/s by update statements. After the transaction is committed, a new transaction is automatically started.

ROLLBACK:

It rollback (undo) the current transaction. That is, it undo all the update performed by SQL statements. Thus database state is restored to what it was before the first statements of the transaction were executed.

- If program terminates without executing either of the commands commit or rollback. The updates or changes to database are either committed or rollback. This depends upon SQL implementation.
- In many SQL implementations, if transactions are continued and at the same moment if the system is restarted or fails then transaction is rollback.