CHAPTER - 4

Relational Database Query Language

4.1 Introduction to SQL

The name *SQL* is presently expanded as *Structured Query Language*. Originally, SQL was called SEQUEL (Structured English QUEry Language) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R. A joint effort by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) has led to a standard version of SQL called SQL-86 or SQL1.

The SQL language may be considered one of the major reasons for the commercial success of relational databases. Because it became a standard for relational databases, users were less concerned about migrating their database applications from other types of database systems—for example, network or hierarchical systems—to relational systems. It also has rules for embedding SQL statements into a general-purpose programming language such as Java, COBOL, or C/C++. The later SQL standards (starting with *SQL: 1999*) are divided into a **core** specification plus specialized **extensions**. The core is supposed to be implemented by all RDBMS vendors that are SQL compliant. The extensions can be implemented as optional modules to be purchased independently for specific database applications such as *data mining*, *spatial data*, *temporal data*, *data warehousing*, *online analytical processing (OLAP)*,*multimedia data*, and so on.

SQL uses the terms **table**, **row**, and **column** for the formal relational model terms *relation*, *tuple*, and *attribute*, respectively. We will use the corresponding terms interchangeably. The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, assertions, and triggers). Note that each statement in SQL ends with a semicolon.

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:
  - *Procedural language:* User instructs the system to perform sequence of operation on the database to compete the desired result. Example: relational algebra
  - *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.
- **Data-definition language** (DDL). The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
- **Data-manipulation language** (DML). The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- **Integrity**. The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
- **View definition**. The SQL DDL includes commands for defining views.
- **Transaction control**. SQL includes commands for specifying the beginning and ending of transactions.
- **Embedded SQL** and **dynamic SQL**. Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java.
- **Authorization**. The SQL DDL includes commands for specifying access rights to relations and views.

Query language Vs. 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 can 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.

4.2 Categories of SQL Commands

SQL commands can be roughly divided into three major categories with regard to their functionality. Firstly, there are those used to *create and maintain the database structure*. The second category includes those *commands that manipulate the data* in such structures, and thirdly there are those that *control the use of the database*. To have all this functionality in a single language is a clear advantage over many other systems straightway, and must certainly contribute largely to the rumor of it being easy to use. It's worth naming these three fundamental types of commands for future reference. Those that create and maintain the database are grouped into the class called DDL or ***Data Definition Language*** statements and those used to manipulate data in the tables, of which there are four, are the DML or ***Data Manipulation Language*** commands. To control usage of the data the DCL commands (***Data Control Language***) are used. In SQL, COMMIT (Saves the changes made to database), ROLLBACK (Undo changes to database from the current state database to last commit state), commands go under Data Control Language.

Database Languages

A. Data Definition Language (DDL)

The set of relations in a database must be specified to the system by means of a data-definition language (DDL). The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:

- The schema for each relation.
- The types of values associated with each attribute.
- The integrity constraints.
- The set of indices to be maintained for each relation.
- The security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Basic Types

The SQL standard supports a variety of built-in types, including:

- **char**(*n*): A fixed-length character string with user-specified length *n*. The full form, **character**, can be used instead.
- **varchar**(*n*): A variable-length character string with user-specified maximum length *n*. The full form, **character varying**, is equivalent.
- **int**: An integer (a finite subset of the integers that is machine dependent). The full form, **integer**, is equivalent.
- **smallint**: A small integer (a machine-dependent subset of the integer type).

- **numeric**(*p, d*):A fixed-point number with user-specified precision. The number consists of *p* digits (plus a sign), and *d* of the *p* digits are to the right of the decimal point. Thus, **numeric**(3,1) allows 44.5 to be stored exactly, but neither 444.5 or 0.32 can be stored exactly in a field of this type.
- **real, double precision**: Floating-point and double-precision floating-point numbers with machine-dependent precision.
- **float**(*n*): A floating-point number, with precision of at least *n* digits.

Each type may include a special value called the **null** value. A null value indicates an absent value that may exist but be unknown or that may not exist at all. In certain cases, we may wish to prohibit null values from being entered.

Basic Schema Definition

SQL DDL provides commands for defining relation schemas, deleting schema, deleting relations and modifying relational schemas. DDL also allows to specify Integrity constraints, Index on relations, Security and authorization for each relation, and Physical storage structure of each relation. Basic statement in Data Definition Language are CREATE, DROP, and ALTER**.** We define an SQL relation by using the **create table** command. The following command creates a relation *department* in the database.

```
create table department
(     dept_name varchar (20),
      building varchar (15),
      budget numeric (12,2),
      primary key (dept_name)
 );
```

The DDL also 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   VARCHAR (3),
   address        VARCHAR NOT NULL,
   CONSTRAINT ch_student_id CHECK (student_id LIKE `S%')
 );
```

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

create table *department* (*dept name* **varchar** (20), *building* **varchar** (15), *budget* **numeric** (12,2), **primary key** (*dept name*));	**create table** *instructor* (*ID* **varchar** (5), *name* **varchar** (20) **not null**, *dept name* **varchar** (20), *salary* **numeric** (8,2), **primary key** (*ID*), **foreign key** (*dept name*) **references** *department*);	**create table** *section* (*course id* **varchar** (8), *sec id* **varchar** (8), *semester* **varchar** (6), *year* **numeric** (4,0), *building* **varchar** (15), *room number* **varchar** (7), *time slot id* **varchar** (4), **primary key** (*course id*, *sec id*, *semester*, *year*), **foreign key** (*course id*) **references** *course*);
create table *course* (*course id* **varchar** (7), *title* **varchar** (50), *dept name* **varchar** (20), *credits* **numeric** (2,0), **primary key** (*course id*), **foreign key** (*dept name*) **references** *department*);	**create table** *teaches* (*ID* **varchar** (5), *course id* **varchar** (8), *sec id* **varchar** (8), *semester* **varchar** (6), *year* **numeric** (4,0), **primary key** (*ID*, *course id*, *sec id*, *semester*, *year*), **foreign key** (*course id*, *sec id*, *semester*, *year*) **references** *section*, **foreign key** (*ID*) **references** *instructor*);	

Figure. SQL data definition for part of the university database.

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.

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. The `CHECK(P)` clause specifies the predicate P that must satisfy specified condition.

Example: SQL data definition for the simple banking database.

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


CREATE TABLE branch
  ( branch_name  VARCHAR(15),
    branch_city  VARCHAR(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  VARCHAR(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  VARCHAR(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    VARCHAR(20),
  loan_no                CHAR(10),
  CONSTRAINT  fk_ cname
     FOREIGN KEY (customer_name) REFERENCES customer
);
```

Example: Using UNIQUE key and DEFAULT value

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

Example: The `CREATE, ALTER, DROP` operation on database

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

CREATE TABLE emp
(empno NUMBER(5) PRIMARY KEY,
   deptno NUMBER(3),
   ename VARCHAR(10) NOT NULL,
   sal NUMBER(5) NOT NULL,
   CONSTRAINT fk_emp_dept FOREIGN KEY(deptno) REFERENCES
dept
);
ALTER TABLE dept ADD (loc VARCHAR(10));
ALTER TABLE emp MODIFY (empno NUMBER(10));
ALTER TABLE emp ADD UNIQUE (ename);
DROP TABLE emp;
DROP constraint fk_emp_dept;
```

B. 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.

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. For example: INSERT, DELETE, UPDATE, SELECT etc. statements.

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. In other word, 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.

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.

A. SELECT command

The SELECT statement is most commonly used SQL statement. It is only a data retrieval statement in SQL.

Syntax: `SELECT [DISTINCT / ALL] <attributes> FROM <relations> [WHERE <predicate>]`

Here, <attribute> = *columns name*, <relations> = *tables name*, and <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;
```

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

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

OR

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

i. What happen if we execute the statement?

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

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.

j. List the customer_id, account_no 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;
```

k. 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;
```

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
    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';
```

Query: Deletes all records from the table `depositor'

```
DELETE FROM depositor;
```

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: 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.

Renaming attribute and relations:

- 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. The pattern includes combination of wildcard characters and regular characters. Some of the wildcard characters are:

Wildcard	Description
%(Percent)	Represent any string of zero or more characters
_(Underscore)	Represent any single character
[]	Represent any single character within specified range
[^]	Represent any single character not within the specified range

E.g.

a. Display all the records from tbl_Marks for which name start with 'M'.

SELECT * from tbl_Marks WHERE name LIKE 'M%'

b. Display all the records from tbl_Marks for which name ends with 'ma'

SLECT * FROM tbl_Marks WHERE name LIKE '%ma'

c. Display all the records from tlb_Marks for which the address is three letters and ends with 'rt'.

SELECT * FROM tbl_Marks WHERE address LIKE '_rt'

d. Display all the records from tbl_Marks for which the name begins with 'a', 'x','y' or 'z'.

SELECT * FROM tbl_Marks WHERE name LIKE '[axyz]%'

e. Display all the records from tbl_Marks for which the name begins with any letter from 'p' to 'w' and ends with 'eta'

SELECT * FROM tbl_Marks WHERE name LIKE '[p-w]eta'

f. Desplay all the records from tbl_Marks for which the name begins with 'd' and not having 'c' as the second letter and ends with 'a'.

SELECT * FROM tbl_Marks where name LIKE'd[^c]%a'

Sorting (*ascending* and *descending*) records in SQL

The 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*** for ascending orders. By default order by clause list item in ascending order. Moreover, ordering can be performed on multiple attributes.

Example: Lists name of customer in alphabetic order by customer name.

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

Example: Lists name of customer in descending alphabetic order.

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

Note: `select from customer order by 2;` Here 2 indicates second column in table "customer".

Example: 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

The set operators combines the results from two or more queries into a single result set. The basic set operation are union (∪), intersection (∩) and difference (-). These operation also can be performed by using union, intersection and minus (except) clause respectively.

A. The union operation: The union operation can be perform by using union clause. For example: consider two relations.

The table client

client_id	name
C001	Ammit
C002	Ajay
C003	Rohit
C004	Ammit

The table supplier

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

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'
```

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

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';
```

The UNION clause is unlike SELECT clause, union operation automatically eliminates duplicates. If we want to retain all duplicates, we must replace UNION ALL.

For example: `select name from supplier where city = 'Kathmandu'.`

The output would be:

Name
Ammit
Ammit
Ashok
Manoj

B. The intersection operation

The intersection operation can be performed by using INTERSECT clause. For ***example***: consider relation as follow:

The salesman table

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

The sales_order 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

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

Output of first SQL statement

salesman_id	name
S001	Manish
S002	Manoj
S001	Manish
S002	Manoj

Output of second SQL statement

salesman_id	name
S001	Manish

Resulting output of SQL statement

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;
```

C. 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:

- AVERGE: AVG
- MAXIMUM: MAX
- MINIMUM: MIN
- TOTAL: SUM
- 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.

A. AVG: It returns average of n, ignoring null values.

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

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 cannot 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;
```

B. MIN: It returns minimum value of n.

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

Example: find the minimum balance of each branch.

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

C. MAX: It returns maximum value of n

Syntax: MAX (<DISTINCT ALL>; n)

Example: find the maximum balance in each branch.

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

D. COUNT: It returns numbers of rows where n is not null.

Syntax: COUNT (<DISTINCT ALL>; n)

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. Acc_number
     GROUP BY branch_name;
```

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

Example: 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;
```

Example: 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.

Example: 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 acc_number = account acc_number
       Depositor customer_name = customer customer_name
              Customer_city = 'KATHMANDU'
    GROUP BY depositor, customer_name
    HAVING COUNT (DISTINCT depositor acc_number)> = 3;
```

E. SUM: It return sum of value of n

Syntax: SUM ([DICTINCT/ ALL] n)

Example: find total loan amount for each branch

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

F. 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`

  Here, 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 query or Subquery or Inner query

- SQL provides subquery facility. A *subquery* is a SQL statement that appears inside another SQL statement. It is also called *nested subqueries* or simply nested query.
- Subqueries are nested inside the WHERE clause of SELECT, INSERT, UPDATE, and DELETE statement.
- The query that represents the parent query is called outer query and the query that represents the subquery is called inner query. The database engine first executes the inner query and then outer query to calculate the result set.
- Subquery use to perform tests for set membership, make set comparisons and determine set cardinality.

Syntax: SELECT <column_name> FROM <table_name> WHERE <column_name> = (SELECT <column_name> FROM <table_name> WHERE <expression>)

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 subquery return more than one values then we required to test whether value written by first query is match/exist or not within values return by subquery.
- 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: Consider the tbl_emp table then perform the following operations:

tbl_emp

Name	Address	Salary
Ram	Pkr	1000
hari	Btl	8000
Shyam	Pkr	5000
Ram	Btl	4000
Sita	Pkr	12000
Rita	Btl	3000

1. Display all the record of employee who have the salary greater than that of Rita

```
SELECT * FROM tbl_emp WHERE salary > (SELECT salary FROM tbl_emp WHERE name = 'rita').
```

Result:

Name	Address	Salary
Hari	Btl	8000
Shyam	Pkr	5000
Ram	Btl	4000
Sita	Pkr	12000

2. Display all the records of employee whose salary is greater than average salary of all employees

```
SELECT * FROM tbl_emp WHERE salary > (SELECT AVG(salary) FROM tbl_emp)
```

Result:

Name	Address	Salary
Hair	Btl	8000
Sita	Pkr	12000

3. Display all the records of employee who have same address as Rita

```
SELECT * FROM tbl_emp WHERE address = (SELECT address FROM tbl_emp WHERE name='rita')
```

Result:

Name	Address	Salary
Hari	Btl	8000
Ram	Btl	4000
Rita	Btl	3000

Note: In the above query, the subquery returns only on value i.e. Btl so it works fine. But if a subquery returns more then one values, we have to use IN keyword.

4. Display address for employes whose address is that of Rita or Sita.

```
SELECT address FROM tbl_emp WHERE address IN (SELECT address FROM tbl_emp WHERE name='Rita' OR
                                              name='Sita')
```

Result:

Address
Pkr
Btl
Pkr
Btl
Pkr
Btl

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 subquery 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 subquery.
- 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 cannot use MAX {AVG (balance)}, since aggregate function cannot 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 nonexistence of values (tuples) in subquery 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 subquery expression to use 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.

Syntax:

```
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

Syntax:
```
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.

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

Note: Delete statement can operate only one relation. It cannot 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-1:
```
INSERT INTO <relation>
     VALUES (<values list>)
```

Syntax-2:
```
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);
```

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

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

Example: Insert infinite number of Tuples.

```
INSERT INTO account
     SELECT *FROM account;
```

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. If P is the number of rows in first table and Q be in second table, then the total number of rows in the result set is P*Q.

   Syntax: `SELECT <column_name> FROM <table-1> CROSS JOIN <table-2>`

Examples: consider two relations

tbl_mobile

Prod_Id	mobile_type	price
2000	Samsung	40000
2001	Apple	50000

tbl_accessories

Cat_id	name	price
1	earphone	1000
2	Mobile sticker	500
3	Mobile Case	700

Find the total price for each mobile device with all the combination of accessories.

```
SELECT      mobile_type,      name,      tbl_mobile.price+
tbl_accessories.price
AS total cost
FROM tbl_mobile CROSS JOIN tbl_accessories
```

Result:

Mobile_type	Name	Total cost
Samsung	Earphone	41000
Samsung	Mobile Sticker	40500
Samsung	Mobile Case	40700
Apple	Earphone	51000
Apple	Mobile Sticker	50500
Apple	Mobile Case	50700

2. **Equi Join**

A equi joins methods is similar to the inner join with joining condition that is based on the equality between values in the common column. The difference is, equi join is used to retrieve all the columns from both the tables but inner join is used to retrieve selected columns from table. So using this join operation results a common column to appear redundancy in the resultant table. The equi join has very important application in commercial database application.

Example: Consider two relations:

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

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.

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

3. **Inner join:** When an inner join is applied, only rows with values satisfying the join condition in the common column are displayed. Records in both tables that do not satisfy the join condition are not displayed.

   Syntax: `SELECT <column_name> FROM <table-1> INNER JOIN <table-2> ON <table-1> . <common_field> <join operator> <table-2> . <common_fiel>`

   Where, `table-1` and `table-2` are tables to be joined, `join operator` is the comparison operator like (=), and `common field` is the name of column on which the join is applied.

Note: we can simply use JOIN keyword instead of INNER JOIN keyword in above statement because the inner join is the default join.

4. **Non – Equi join (Outer join)**

The outer join displays the result set containing all the rows from one table and matching rows from another table. An outer join displays NULL for the columns of the related table

where it doesn't find matching records. This method joins tables (relations) based on non-equality so sometime it is also called *non-equi join*. An outer join is of following type:

a. **Left Outer join**: A left oute join returns all the rows from the table specified on left side of LEFT OUTER JOIN keyword and the matching rows from the table specified on the right side. It displays NULL for the columns of the table specified on the right side where it doesn't find matching records.
Syntax: SELECT <column_names> FROM <table1> LEFT OUTER JOIN <table2> ON <table1>.<common_field> <join operator> <table2>.<common_field>
b. Right Outer join: A right outer join returns all the rows from the table specified on the right hand side of the RIGHT OUTER JOIN keyoword and the mathching rows from the table specified on the left side. It displays NULL for the columns of the table specified on the left side where it doesn't find matching records.
Syntax: SELECT <column_names> FROM <table1> RIGHT OUTER JOIN <table2> ON <table1>.<common_field> <join operator> <table2>.<common_field>
c. **Full Outer join:** A full outer join is the combination of left outer join and right outer join which returns all the matching and non matcing rows from both the tables. In case of non matching rows, a NULL values is displayed for the columns for which the data is not available.
Syntax: SELECT <column_names> FROM <table1> FULL OUTER JOIN <table2> ON <table1>.<common_field> <join operator> <table2>.<common_field>

For ***example:*** Let us consider the following two relations.

tbl_filmname

filmId	Filmname	Yearmade
1	Chennai Express	2013
2	3-idiots	2012
4	Titanic	2008

tbl_actor

filmId	Firstname	lastname
1	Sharukh	Khan
1	Deepika	Padukon
2	Aamir	Khan
2	Karina	Kapoor
3	Salman	Khan
3	Katrina	Kaif

a. Join the two table tbl_filmname and tbl_actor using inner join.

SELECT * FROM tbl_filmname JOIN tbl_actor ON tbl_filmname.filmid=tbl_actor.filmid

Result:

filmid	Filmname	Yearmade	filmid	Firstname	Lastname
1	Chennai Express	2013	1	Sharukh	Khan
1	Chennai Express	2013	1	Deepika	Padukon
2	3-idiots	2012	2	Aamir	Khan
2	3-idiots	2012	2	Karina	Kapoor

b. Join the two table tbl_filmname and tbl_actor using left outer join.

SELECT * FROM tbl_filmname LEFT OUTER JOIN tbl_actor ON tbl_filmname.filmid=tbl_actor.filmid

Result:

filmid	Filmname	Yearmade	Filmid	Firstname	lastname
1	Chennai Express	2013	1	Sharukh	Khan
1	Chennai Express	2013	1	Deepika	Padukon
2	3-idiots	2012	2	Aamir	Khan
2	3-idiots	2012	2	Karina	Kapoor
4	Titanic	2008	NULL	NULL	NULL

c. Join the two table tbl_filmname and tbl_actor using Right outer join.

SELECT * FROM tbl_filmname RIGHT OUTER JOIN tbl_actor on tbl_filmname.filmid=tbl_actor.filmid

Result:

filmid	Filmname	Yearmade	Filmid	Firstname	lastname
1	Chennai Express	2013	1	Sharukh	Khan
1	Chennai Express	2013	1	Deepika	Padukon
2	3-idiots	2012	2	Aamir	Khan
2	3-idiots	2012	2	Karina	Kapoor
NULL	NULL	NULL	3	Salman	Khan
NULL	NULL	NULL	3	Katrina	Kaif

d. Join the two table tbl_filmname and tbl_actor using full outer join.

SELECT * FROM tbl_filmname FULL OUTER JOIN tbl_actor on tbl_filmname.filmid=tbl_actor.filmid

Result:

filmid	Filmname	Yearmade	Filmid	Firstname	lastname
1	Chennai Express	2013	1	Sharukh	Khan
1	Chennai Express	2013	1	Deepika	Padukon
2	3-idiots	2012	2	Aamir	Khan
2	3-idiots	2012	2	Karina	Kapoor
4	Titanic	2008	NULL	NULL	NULL
NULL	NULL	NULL	3	Salman	Khana
NULL	NULL	NULL	3	Katrina	Kaif

Data Control Language

GRANT and REVOKT commands

GRANT is a command used to provide access right or privilege on the database objects to the users. These commands are generally used by DBA. REVOKE is a command which remove user access right or privilege to the database object.

Syntax:

```
GRANT <Privilege_Name> ON <Object_Name> TO <User_Name>
REVOKE <Privilege_Name> ON <Object_Name> FROM <User_Name>
```

Where, `Privilege_Name` means access right. Some of the access right may be ALL, SELECT, INSERT, UPDATE, DELETE, EXECUTE. `Object_Name` is the name of database object such as TABLE, VIEWS, STORED PROC etc. `User_Name` is the name of the user to whom an access right is being granted.

Example:

- `GRANT SELECT ON tbl_student TO user1;` // give access permission on tbl_student to user1
- `REVOKE SELECT ON tbl_student FROM user1;` //Restrict user1 to access tbl_student

4.4 Stored procedure

When a batch of SQL statements needs to be executed more than once, we need to recreate SQL statement and submit them to the server. This lead to an increase in the overhead, as the server need to compile and create the execution plan for these statements again. Therefore if we need to execute a batch multiple times, we can save it within a stored procedure. A stored procedure is a precompiled object stored in a database data dictionary. A stored procedure is also called as proc, sproc, or sp and is similar to the user defined functions. Stored procedure can invoke DDL and DML statements and can return values.

- *Advantages:* It is faster and have better security because calls are parameterized.
- *Limitations:* Less flexible because we need DBA to make changes.

Creating Stored Procedure

We can create stored procedure using CREATE PROCEDURE statement.

```
Syntax:   CREATE PROCEDURE <procedure_name> AS
          BEGIN
             Sql_statement1
             Sql_statement2
             .
             Sql_statementN
          END

E.g.      CREATE PROCEDURE sp_mobile AS
          BEGIN
                SELECT * FROM tbl_mobile
          END
```

When the CREATE PROCEDURE command is executed, the server compiles the procedure and save it as a database object. The procedure is then available for various applications to execute.

Executing a Stored Procedure

A procedure can be executed using EXECUTE or EXEC statement.

```
Syntax:   EXECUTE | EXEC <Procedure_name>
E.g.      EXEC sp_mobile
```

Altering stored procedure

A stored procedure can be modified by using ALTER PROCEDURE statement.

```
Syntax:   ALTER PROCEDURE <Procedure_name> AS
          BEGIN
                Sql_statement1
                Sql_statement2
                .
                Sql_statementN
          END

E.g       ALTER PROCEDURE  sp_mobile AS
          BEGIN
                SELECT * FROM tbl_mobile  WHERE mobile_type='samsung'
          END
```

Now if we execute the procedure sp_mobile then it will display all the records from tbl_mobile having mobile_type Samsung.

Dropping a stored Procedure

We can destroy a stored procedure form database by using DROP PROCEDURE statement.

```
Syntax:   DROP PROCEDURE <Procedure_name>
E.g.      DROP PROCEDURE sp_mobile
```

Creating a parameterized stored procedure

A parameterized stored procedure is necessary when we need to execute a procedure for different values of variables that are provided at runtime. Parameters are used to pass values to stored procedure during run time. Each parameter has a name and its data type.

Syntax: CREATE PROCEDURE <Procedure_name>
@parameter1 data type1,
@parameter2 data type2,
.
.
@parameterN data typeN
As
BEGIN
Sql_statement1
Sql_statement2
.
Sql_statementN
END

E.g. CREATE PROC sp_mob
@mobile_type varchar(50)
AS
BEGIN
SELECT * FROM tbl_mobile WHERE mobile_type=@mobile_type
END

Now executing the above procedure by using the following statements.
EXECUTE sp_mob @mobile_type='samsung' // displays all the record from tbl_mobile having mobile_type Samsung.
EXECUTE sp_mob @mobile_type='Apple' // displays all the records from tbl_mobile having mobile_type Apple.

4.5 Query By Example (QBE)

QBE is a graphical query language which is based on the domain relational calculus. A user needs minimal information to get started and the whole language contains relatively few concepts. QBE is especially suited for queries that are not too complex and can be expressed in terms of a few tables. QBE provide a tabular interface that has expressive power of relational calculus in a user friendly form i.e. queries look like table. Here queries are expressed by 'example'.

Example: Let us consider the following three relations.

Sailor (Sid, Sname, age)
Boats (Bid, Bname, color)
Reserve (sid, bid, day)

1. Create example table for printing name and age of all sailors.

Sailors	Sid	Sname	age
		P.N	P.A

In the above table, N and A are variables. The use of such variable is optional. P. indicates the print. Duplicate values are removed by default. To retain duplicate use P.All.

2. Display all the record of sailor

Sailors	Sid	Sname	age
	P.	P.	P.

OR

Sailors	Sid	Sname	age
P.			

3. Display all the records of sailor whose age is equal to 60.

Sailors	Sid	Sname	age
P.			60

Placing a constant 60 means, placing a condition = 60. We can use other comparison operator (>, <, >=, <=, ¬) as well.

4. Display all the records of sailors in ascending order of age.

Sailors	Sid	Sname	age

		P.	P.AO

The P.AO means ascending order and P.DO means descending order.

5. Display the average age of sailors.

Sailors	Sid	Sname	age	
P.		P.	A	P.AVG.A

6. Find sailors who have reserved a boat for 8/24/96 and who are older than 25.
Hind: To find the sailors with a reservation, we have to combine information from the Sailors and the Reservation relations. We do this by placing the same variable in the Sid column of the two example relations.

Sailors	Sid	Sname	age
P.	ID	P.S	>25

Reserve	Bid	Sid	Days
P.		ID	'8/24/96'

7. Display all the name of sailors whose age is greater than 20 and less than 60.

Sailors	Sid	Sname	age
		P.	A

Condition
A > 20 AND A< 60

Simple conditions can be expressed directly in column of the example. For more complex conditions QBE provides a feature called a *condition box*.

8. Display all the name of sailors whose name start with m and age is below 60.

Sailors	Sid	Sname	age
		P.N	A

Condition
N LIKE 'M%' AND A< 60

9. Insert one record to relation sailor.
Hind: I. command is used to insert new record.

Sailors	Sid	Sname	age
I.	11	'Ram'	40

10. Delete all the records whose age is greater than 60.
Hind: D. command is used to delete all or selected records.

Sailors	Sid	Sname	age
D.			> 60

11. Modify the age of sailor with Sid 72 to be 42 years.
Hind: U. command is used to update the existing records.

Sailors	Sid	Sname	age
	72		U.42

12. Increment the age of all sailors by 10.

Sailors	Sid	Sname	age
	72		U.A+10

Note:

Example: Write the relational algebra and SQL command of the following expression where the given relations are as follows:

Loan(loannumber, branchname, amount)
Customer(customername,street,city)
Borrower(customername,loannumber)
Depositer(customername,accountnumber)

1. Find the loan number of each loan of amount greater than Rs 1200.
Relational Algebra: $\prod_{loannumber}(\sigma_{amount > 1200}(Loan))$
SQL; SELECT loannumber FROM Loan WHERE amount > 1200

2. Find all the name of customer who have loan, account or both.
Relational Algebra: $\prod_{customername}(Borrower) \cup \prod_{customername}(Depositer))$
SQL: SELECT customername FROM Borrower UNION SELECT customername FROM Depositer

3. Find all the name of customer who have loan and account at bank.
Relational Algebra: $\prod_{customername}(Borrower) \cap \prod_{customername}(Depositer))$
SQL: SELECT customername FROM Borrower INTERSECT SELECT customername FROM Depositer