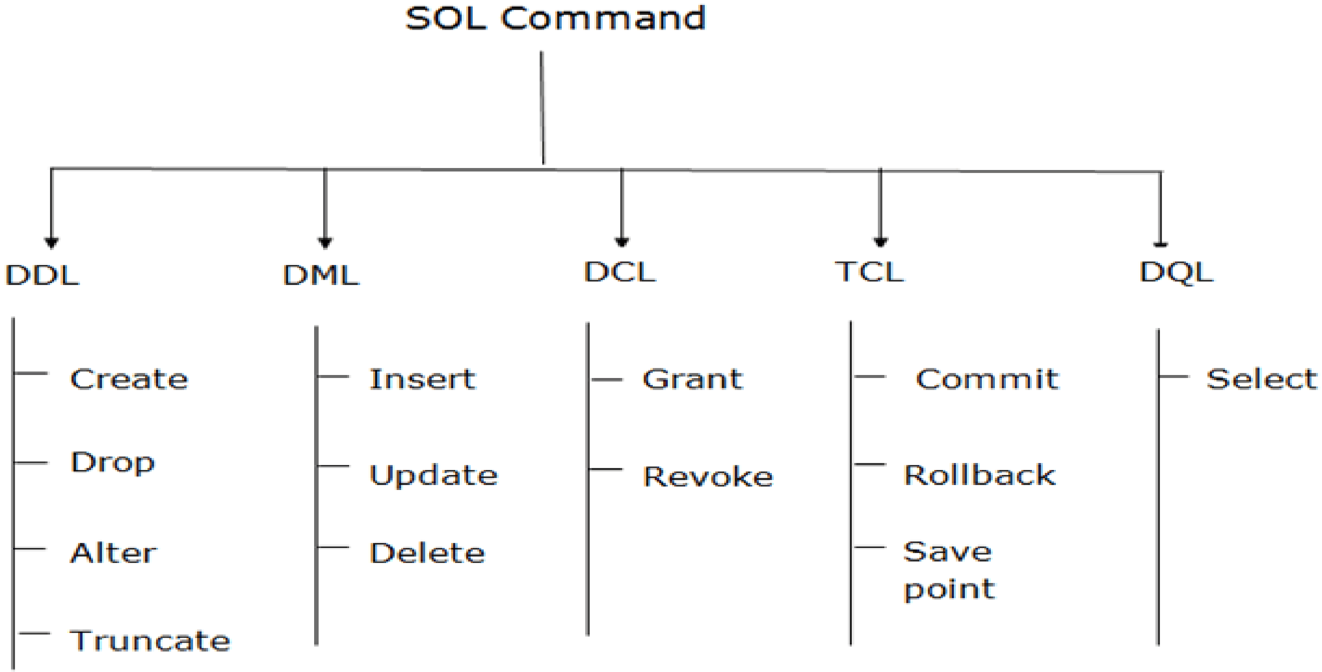
Module 4

Structured Query Language (SQL)

SQL

- SQL stands for **Structured Query Language**. It is used for storing and managing data in relational database management system (RDMS).
- It is a standard language for Relational Database System. It enables a user to **create, read, update and delete** relational databases and tables.
- All the RDBMS like **MySQL**, Informix, Oracle, MS Access use SQL as their standard database language.
- SQL comprises both **data definition and data manipulation languages**. Using the data definition properties of SQL, one can design and modify database schema, whereas data manipulation properties allows SQL to **store and retrieve** data from database.

SQL Commands



SOL Command

DDL
- Create
- Drop
- Alter
- Truncate

DML
- Insert
- Update
- Delete

DCL
- Grant
- Revoke

TCL
- Commit
- Rollback
- Save point

DQL
- Select

SQL commands

- **Data Definition Language(DDL)** – Consists of commands which are used to define the database.
- **Data Manipulation Language(DML)** – Consists of commands which are used to manipulate the data present in the database.
- **Data Query Language(DQL)** Data Query Language is part of the base grouping of SQL sub-languages. Select is DQL .
- **Data Control Language(DCL)** – Consists of commands which deal with the user permissions and controls of the database system.
- **Transaction Control Language(TCL) –** Consist of commands which deal with the transaction of the database.

1. Data Definition Language (DDL)

- DDL changes the **structure of the table** like creating a table, deleting a table, altering a table, etc.
- All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

a. **CREATE:** It is used to create a new database or new table in the database.

Syntax for database creation :

CREATE DATABASE DatabaseName;

Example

CREATE DATABASE Employee;

Syntax for table creation:

1. CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);

Example:

1. CREATE TABLE EMPLOYEE(Name VARCHAR(20), Email VARCHAR(100), DOB DATE);

b. DROP: It is used to delete both the structure and record stored in the table.

Syntax

```
DROP TABLE table_name;
```

Example

```
DROP TABLE EMPLOYEE;
```

c. ALTER: It is used to alter the structure of the database.

This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

Syntax:

To add a new column in the table

```
ALTER TABLE table_name ADD column_name COLUMN-definition;
```

To modify existing column in the table:

```
ALTER TABLE table_name MODIFY(column_definitions....);
```

EXAMPLE

```
ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));
ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR2(20));
```

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

Syntax:

```
TRUNCATE TABLE table_name;
```

Example:

```
TRUNCATE TABLE EMPLOYEE;
```

Rename Command

- **Rename command to rename a column name in sql**
- `ALTER TABLE table_name RENAME COLUMN old_column_name TO new_column_name;`
- **Rename command to rename a table name in sql**

- ALTER TABLE old_table_name RENAME new_table_name

OR

RENAME TABLE old_table_name TO new_table_name;

2. Data Manipulation Language

- DML commands are used to modify the database. It is responsible for all form of changes in the database.
- The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

1. **SELECT**
2. INSERT
3. UPDATE
4. DELETE

1. **SELECT Command**

The SELECT statement is used to select data from a database .

Syntax :

SELECT <Attribute List> FROM <table_name>;

Example :

SELECT * FROM students;

OR

Select with where clause

SELECT <Attribute List> **FROM** Name_of_Table **WHERE** [condition];

SELECT * FROM students where due_fees <=20000;

Note: * denotes all attributes, if you want to select specific attribute you can mention that instead of *

2. INSERT Command

The INSERT INTO statement is used to insert new records in a table.

Syntax :

1. Specify both the column names and the values to be inserted:

```
  INSERT INTO table_name (column1, column2, column3, ...)VALUES (value1, value2, value3, );
```

2. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table.

```
INSERT INTO table_name VALUES (value1, value2, value3, ...);
```

Example:

```
1] INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode,
Country)

VALUES ('Cardinal', 'Tom B. Erichsen', ‘mumbai 21', ‘mumbai', '4006', ‘India');
```

```
2] INSERT INTO Customers (CustomerName, City, Country) VALUES ('Cardinal',
'Stavanger', 'Norway');
```

3. UPDATE Command

This command is used to alter existing table records. Within a table, it modifies data from one or more records. This command is used to alter the data which is already present in a table.

Syntax :

```
UPDATE <table_name> SET <column_name = value>WHERE condition;
```

Example :

```
UPDATE students SET due_fees = 20000 WHERE stu_name = 'Mini';
```

4. DELETE Command

It deletes all archives from a table. This command is used to erase some or all of the previous table’s records. If we do not specify the ‘WHERE’ condition then all the rows would be erased or deleted.

Syntax :

```
DELETE FROM <table_name>WHERE <condition>;
```

Example :

```
DELETE FROM students WHERE stu_id = '001';
```

SET Operations in SQL

- SQL supports few Set operations which can be performed on the table data. These are used to get meaningful results from data stored in the table, under different special conditions.

1. UNION
2. UNION ALL
3. INTERSECT
4. MINUS

1. UNION Operation

UNION is used to **combine** the results of two or more SELECT statements. However it will **eliminate duplicate rows** from its resultset. In case of union, **number of columns** and **datatype** must be same in both the tables, on which UNION operation is being applied.

Table 1

ID	Name
1	abhi
2	adam

Table 2

ID	Name
2	adam
3	Chester

- Union SQL query will be,

```
SELECT * FROM Table1 UNION SELECT * FROM Table2 ;
```

- Output will be,

ID	NAME
1	abhi
2	adam
3	Chester

2. UNION ALL

- This operation is **similar to Union**. But it also **shows** the **duplicate rows**.
- Union All query will be like,

```
SELECT * FROM Table1 UNION ALL SELECT * FROM Table2;
```

- Output of Union All will be,

ID	NAME
1	abhi
2	adam
2	adam
3	Chester

3. INTERSECT

Intersect operation is used to combine two SELECT statements, but it only returns the records which are common from both SELECT statements. In case of **Intersect** the **number of columns** and **datatype** must be same.

Note: MySQL does not support INTERSECT operator.

```
SELECT * FROM Table1 INTERSECT SELECT * FROM Table2;
```

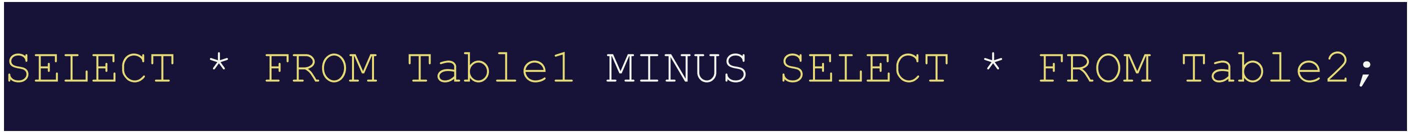
Output of Intersect Operation will be,

ID	NAME
2	adam

4. MINUS

The Minus operation **combines results** of two SELECT statements and return only those in the final result, **which belongs to the first set of the result**.

Minus query will be,

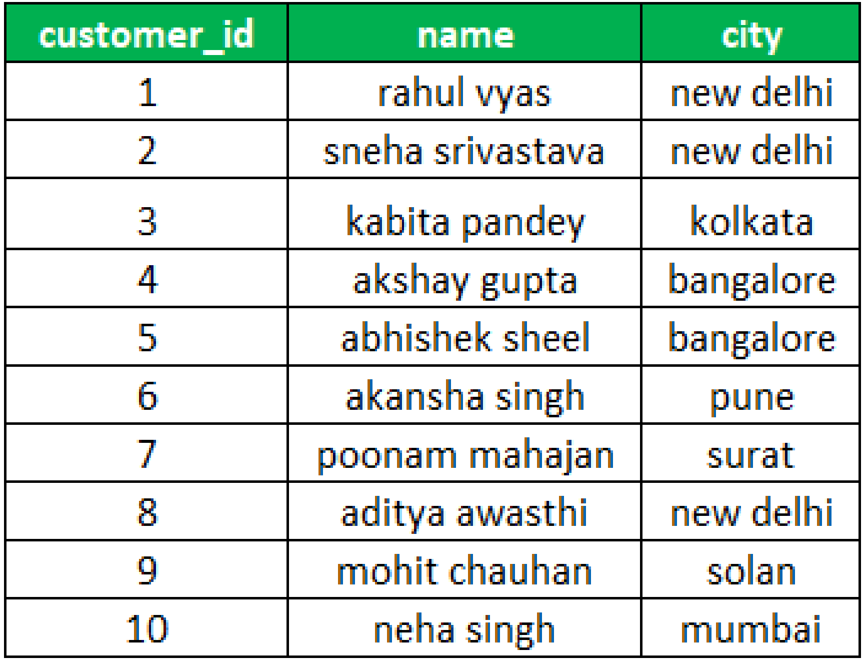


```
SELECT * FROM Table1 MINUS SELECT * FROM Table2;
```

Output of above query will be,

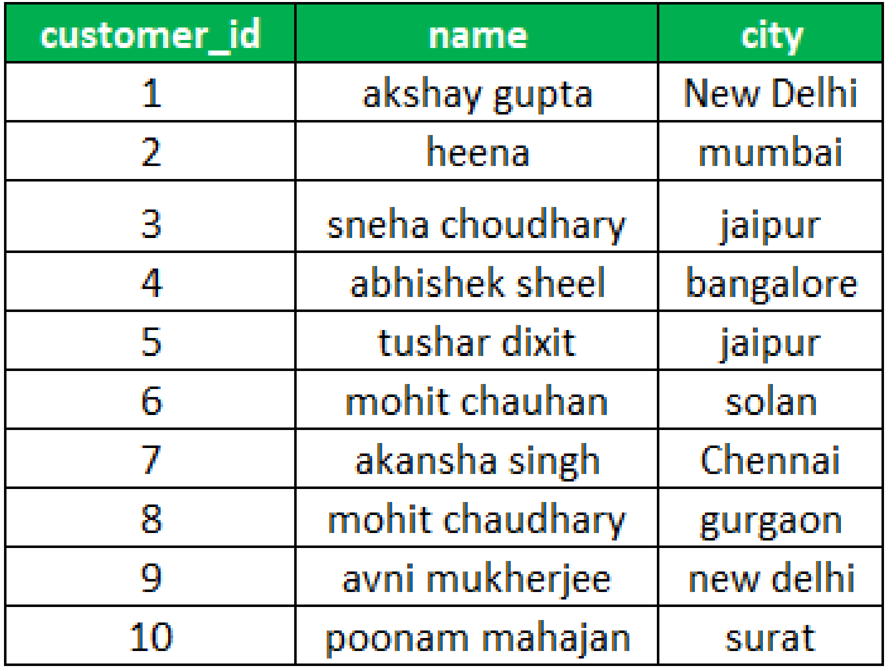
ID	NAME
1	abhi

customers_jan:



customer_id	name	city
1	rahul vyas	new delhi
2	sneha srivastava	new delhi
3	kabita pandey	kolkata
4	akshay gupta	bangalore
5	abhishek sheel	bangalore
6	akansha singh	pune
7	poonam mahajan	surat
8	aditya awasthi	new delhi
9	mohit chauhan	solan
10	neha singh	mumbai

customers_dec



customer_id	name	city
1	akshay gupta	New Delhi
2	heena	mumbai
3	sneha choudhary	jaipur
4	abhishek sheel	bangalore
5	tushar dixit	jaipur
6	mohit chauhan	solan
7	akansha singh	Chennai
8	mohit chaudhary	gurgaon
9	avni mukherjee	new delhi
10	poonam mahajan	surat

- Write SQL Query to find the name of all the customers

```
SELECT name FROM customers_dec
UNION
SELECT name FROM customers_jan;
```

Q. Find all details of all Customers

Q. Find the names of all the customers who registered in December or January.

- SELECT name FROM customers_dec
  UNION ALL
  SELECT name FROM customers_jan;

Q.1 Find all details of Customers who shopped only in December but not January.

Q. 2 Find the name and City of customers who shopped only in December but not January.

Q. 3 Find all details of customers who shopped in December and January.

Q. 4 Find all details of customers who registered in December or January.

ANSWERS

- A1:
  ```
  Select * from customers_dec
  MINUS
  Select * from customers_jan;
  ```

- A2:
  ```
  Select name,city from customers_dec
  MINUS
  Select name,city from customers_jan;
  ```

- A3:
  ```
  Select * from customers_dec
  INTERSECT
  Select * from customers_jan;
  ```

- A4:
  ```
  SELECT * FROM customers_dec
  UNION ALL
  SELECT * FROM customers_jan;
  ```

Data Control Language(DCL)

- DCL stands for **Data Control Language.**
- DCL is used to control user access in a database.
- This command is related to the **security issues.**
- Using DCL command, it allows or restricts the user from accessing data in database schema.

DCL commands are used to grant and take back authority from any database user.

1. Grant
2. Revoke

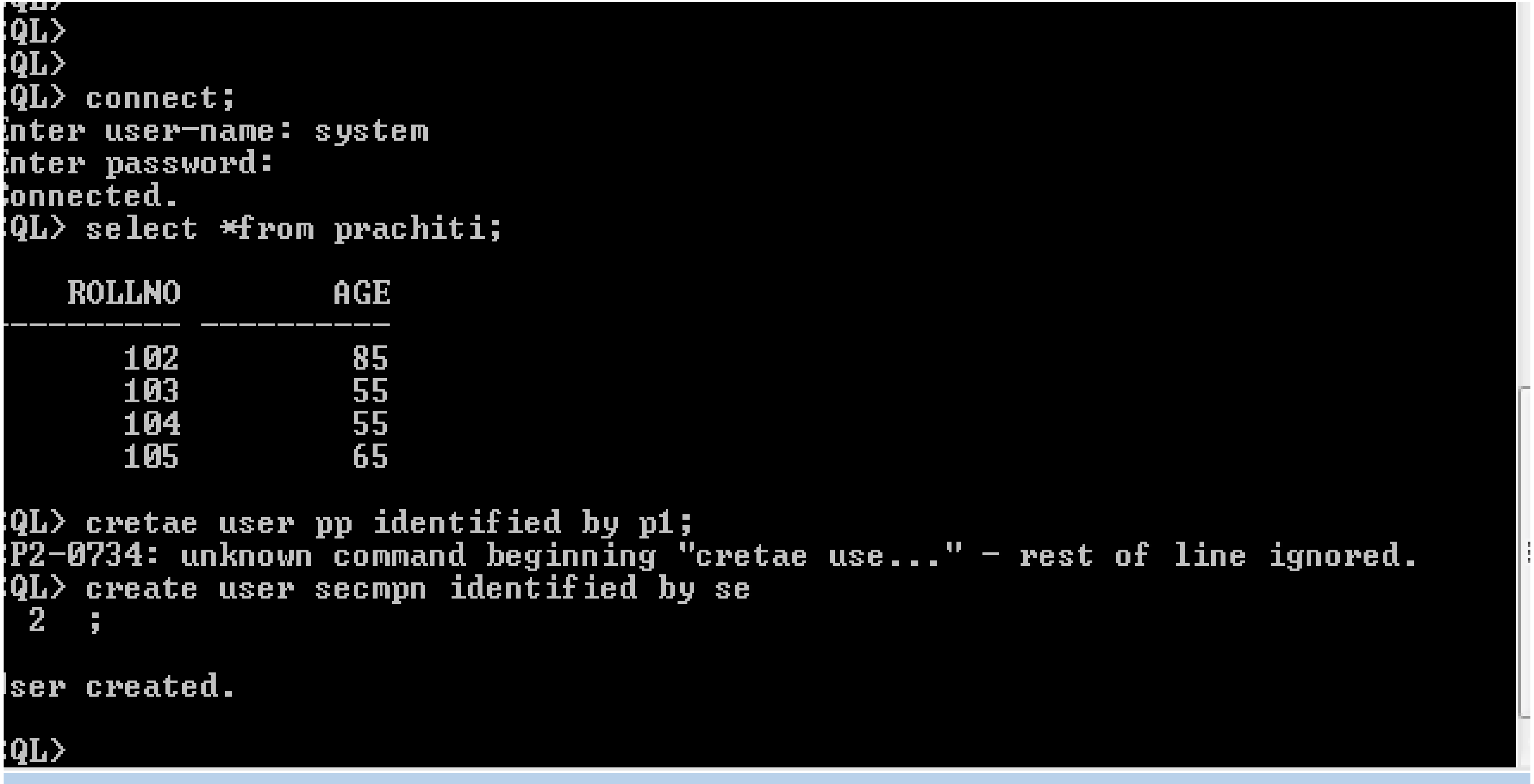
1. Create a new user with password:

   create user user_name identified by admin_password;

   `user_name` is the name of the user you are creating and `admin_password` is the password that you want to assign to the user.

1. Assign the sysdba privilege to the new user:

   grant <privilege> to <user> ;



```
QL>
QL>
QL>
QL> connect;
nter user-name: system
nter password:
onnected.
QL> select *from prachiti;

   ROLLNO        AGE
---------- ----------
       102         85
       103         55
       104         55
       105         65

QL> cretae user pp identified by p1;
P2-0734: unknown command beginning "cretae use..." - rest of line ignored.
QL> create user secmpn identified by se
 2  ;

ser created.

QL>
```

1. GRANT COMMAND

- **GRANT command** gives user's access privileges to the database.
- This command allows specified users to perform specific tasks.

Syntax:

```
GRANT <privilege list>ON <relation name or view name> TO <user/role list>;
```

Example 1 :

```
GRANT ALL ON employee TO ABC [WITH GRANT OPTION] ;
```

Example 2:

```
GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER;
```

2. REVOKE COMMAND

- **REVOKE command** is used to cancel previously granted or denied permissions.
- This command withdraw access privileges given with the GRANT command.
- It takes back permissions from user.

- **Syntax:**

REVOKE <privilege list> ON <relation name or view name> FROM <user name>;

Example 1 :

REVOKE UPDATE ON employee FROM ABC;

Example 2:

REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1;

Transaction Control Language(TCL)

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under **TCL**:

- **COMMIT**
- **ROLLBACK**
- **SAVEPOINT**

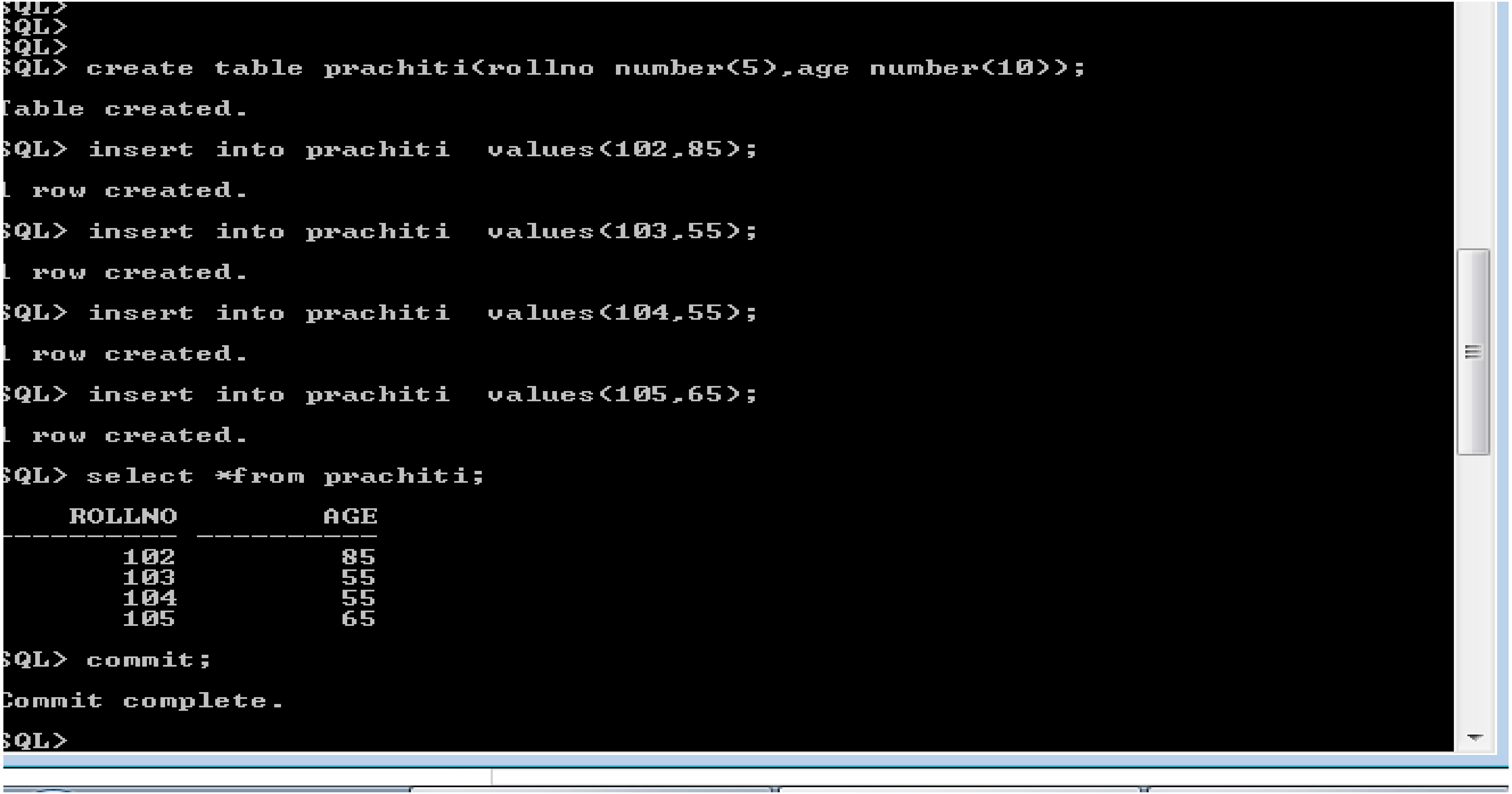
a. Commit: Commit command is used to save all the transactions to the database.

Syntax:

```
COMMIT;
```

Example:

```
DELETE FROM CUSTOMERS  WHERE AGE = 25;
COMMIT;
```



```
SQL>
SQL>
SQL>
SQL> create table prachiti(rollno number(5),age number(10));

Table created.

SQL> insert into prachiti  values(102,85);

1 row created.

SQL> insert into prachiti  values(103,55);

1 row created.

SQL> insert into prachiti  values(104,55);

1 row created.

SQL> insert into prachiti  values(105,65);

1 row created.

SQL> select *from prachiti;

    ROLLNO        AGE
---------- ----------
       102         85
       103         55
       104         55
       105         65

SQL> commit;

Commit complete.

SQL>
```

b. Rollback:

Rollback command is used to undo transactions that have not already been saved to the database.

Note:

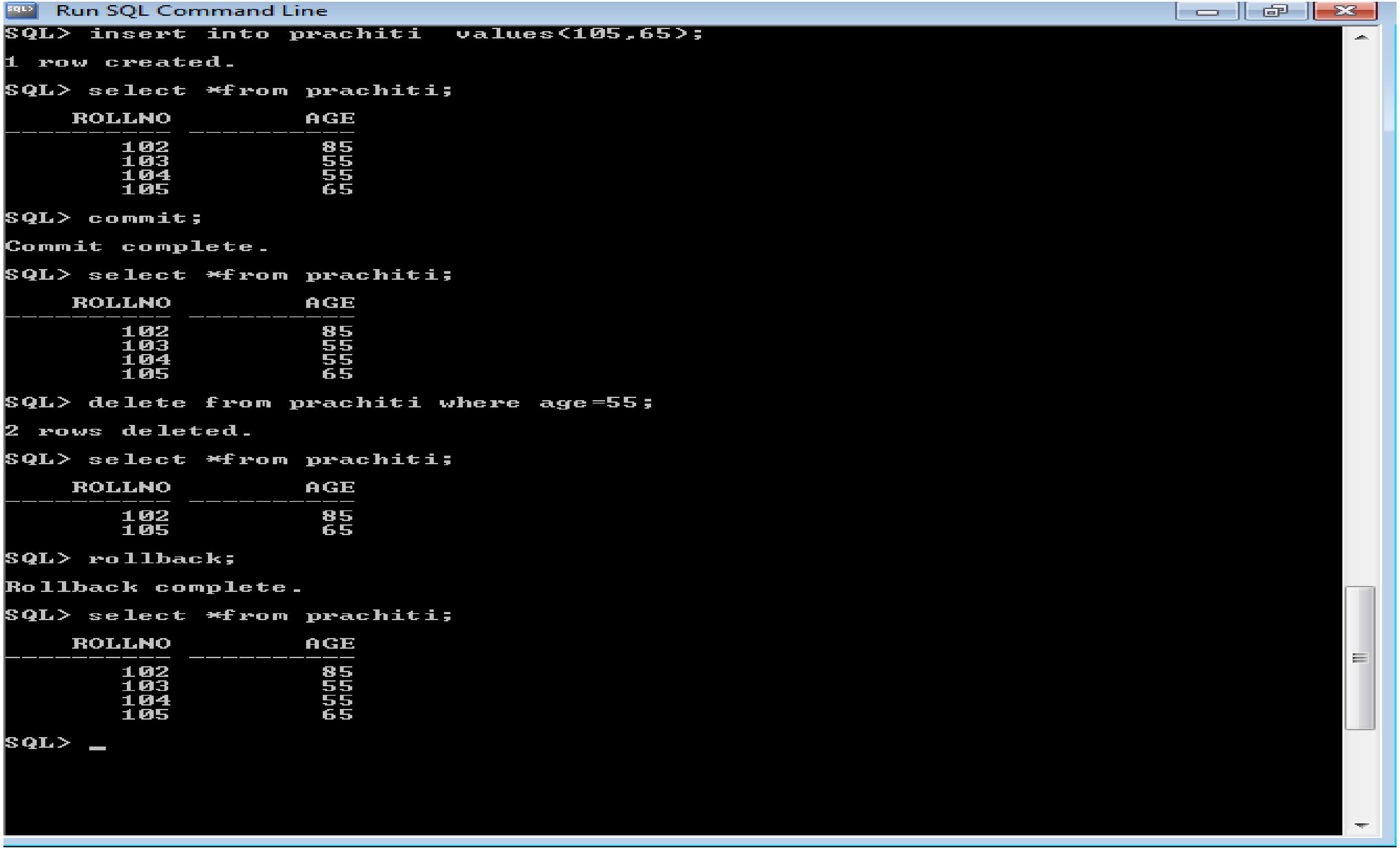
If we have used the UPDATE command to make some changes into the database, and realize that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not committed using the COMMIT command.

Syntax:

```
ROLLBACK;
```

Example:

```
DELETE FROM CUSTOMERS  WHERE AGE = 25;

ROLLBACK;
```



```
SQL> insert into prachiti  values(105,65);

1 row created.

SQL> select *from prachiti;

    ROLLNO        AGE
---------- ----------
       102         85
       103         55
       104         55
       105         65

SQL> commit;

Commit complete.

SQL> select *from prachiti;

    ROLLNO        AGE
---------- ----------
       102         85
       103         55
       104         55
       105         65

SQL> delete from prachiti where age=55;

2 rows deleted.

SQL> select *from prachiti;

    ROLLNO        AGE
---------- ----------
       102         85
       105         65

SQL> rollback;

Rollback complete.

SQL> select *from prachiti;

    ROLLNO        AGE
---------- ----------
       102         85
       103         55
       104         55
       105         65

SQL> _
```

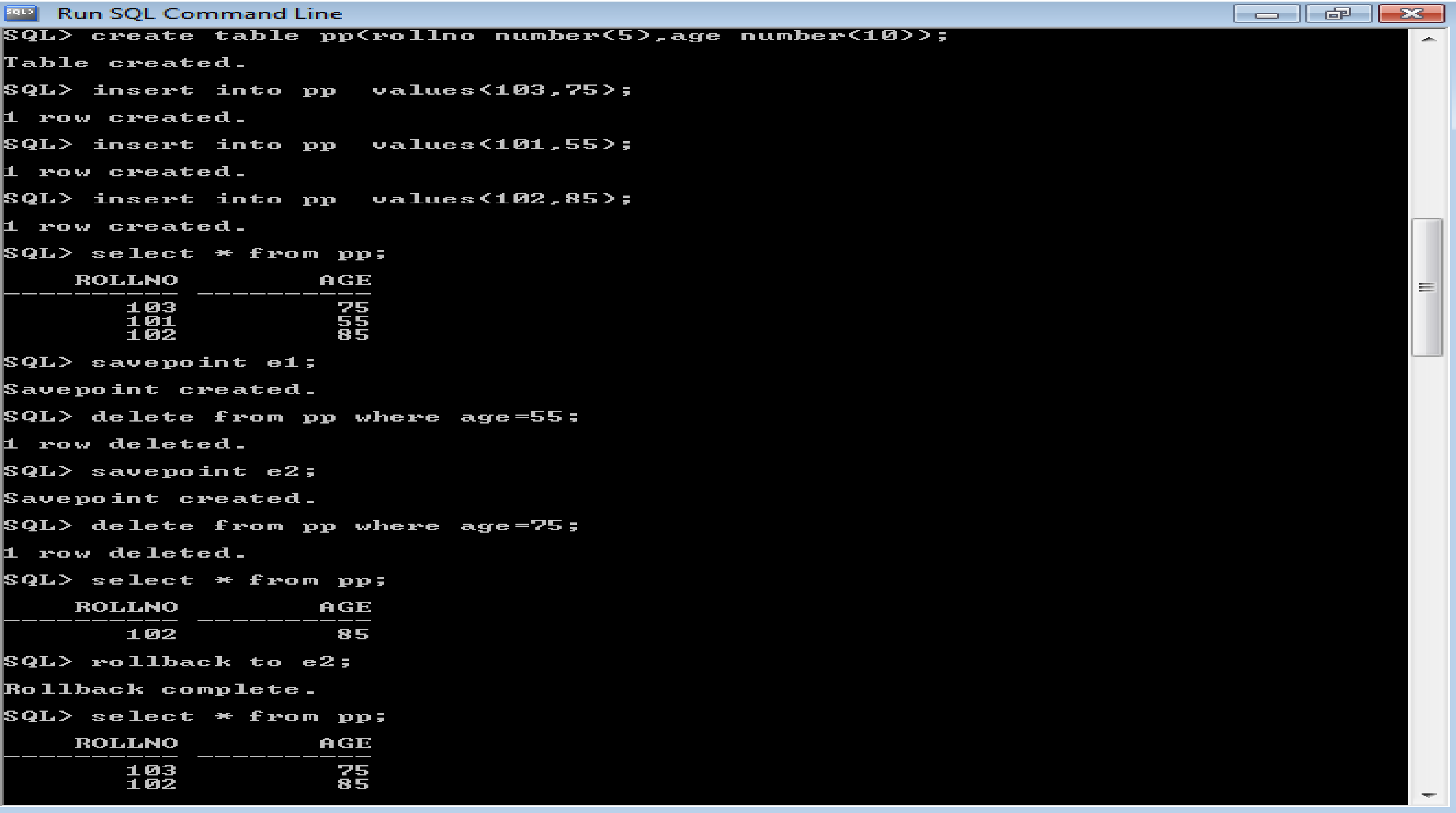
SHOW MORE

3. **SAVEPOINT:** It is used to roll the transaction back to a certain point without rolling back the entire transaction.

4. SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

Syntax:

```
SAVEPOINT SAVEPOINT_NAME;
```



```
SQL> create table pp(rollno number(5),age number(10));

Table created.

SQL> insert into pp  values(103,75);

1 row created.

SQL> insert into pp  values(101,55);

1 row created.

SQL> insert into pp  values(102,85);

1 row created.

SQL> select * from pp;

    ROLLNO        AGE
---------- ----------
       103         75
       101         55
       102         85

SQL> savepoint e1;

Savepoint created.

SQL> delete from pp where age=55;

1 row deleted.

SQL> savepoint e2;

Savepoint created.

SQL> delete from pp where age=75;

1 row deleted.

SQL> select * from pp;

    ROLLNO        AGE
---------- ----------
       102         85

SQL> rollback to e2;

Rollback complete.

SQL> select * from pp;

    ROLLNO        AGE
---------- ----------
       103         75
       102         85
```

Comments in SQL

Comments in SQL

- There are two ways in which you can comment in SQL, i.e. either the Single-Line Comments or the Multi-Line Comments.

1. Single-Line Comments

The single line comment starts with two hyphens (–). So, any text mentioned after (–), till the end of a single line will be ignored by the compiler.

Example:

```
- -Select all:
SELECT * FROM Employee_Info;
```

2. Multi-Line Comments

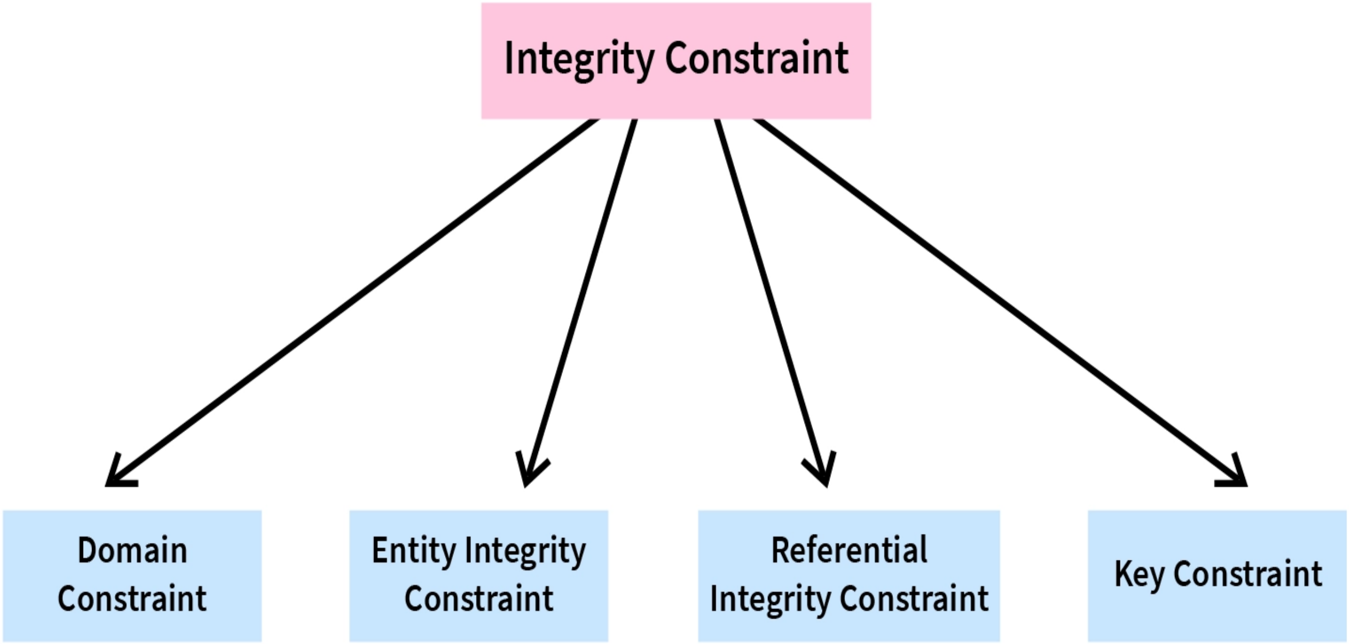
- The Multi-line comments start with **/*** and end with ***/**. So, any text mentioned between /* and */ will be ignored by the compiler.

Example:

```
/*Select all the columns
of all the records
from the Employee_Info table:*/
SELECT * FROM Students;
```

Integrity Constraints

- Integrity constraints are a set of rules. It is used to maintain the quality of information.
- Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
- Thus, integrity constraint is used to guard against accidental damage to the database.



Integrity Constraint

Domain Constraint

Entity Integrity Constraint

Referential Integrity Constraint

Key Constraint

Integrity Constraints in SQL

1. Primary Key Constraints
2. Unique Key Constraints
3. Foreign Key Constraints
4. NOT NULL Constraints
5. Default constraints
6. Check constraints

Integrity Constraints in SQL

- In SQL, integrity constraints are rules that are defined to maintain the accuracy and consistency of data in a database. There are several types of integrity constraints, including:

1. **Primary Key Constraint**: Ensures that each row in a table has a unique identifier, and it cannot contain NULL values. This constraint is used to enforce entity integrity.

Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);
```

2. Unique Constraint:

- Ensures that the values in a specified column or a group of columns are unique across all the rows in a table.
- **Example:**

```
CREATE TABLE Students (
   StudentID INT unique,
   StudentName VARCHAR(50),
);
```

3. Foreign Key Constraint:

- Enforces referential integrity by ensuring that values in a column (or a set of columns) in one table match the values in the primary key of another table. Example:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

4. Check Constraint:

- Specifies a condition that must be met for data to be entered or updated in a column.

Example:

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(50),
    Price DECIMAL(10, 2),
    CHECK (Price >= 0)
);
```

5. Not Null Constraint

Ensures that a column cannot contain NULL values.

Example:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE NOT NULL
);
```

6. Default Constraint:

Specifies a default value for a column **if no value is explicitly provided**.

Example:

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    City VARCHAR(20) DEFAULT ‘Mumbai’
);
```

USE OF *

- To retrieve all the attribute values of the selected tuples, a * is used, which stands for *all the attributes*
  Examples:

```
Q1C:    SELECT *
                FROM            EMPLOYEE
                WHERE  DNO=5

Q1D:    SELECT *
                FROM            EMPLOYEE, DEPARTMENT
                WHERE  DNAME='Research' AND
                DNO=DNUMBER
```

USE OF DISTINCT

- SQL does not treat a relation as a set; duplicate tuples can appear

  To eliminate duplicate tuples in a query result, the keyword **DISTINCT** is used

  For example, the result of Q1may have duplicate SALARY values whereas Q2 does not have any duplicate values

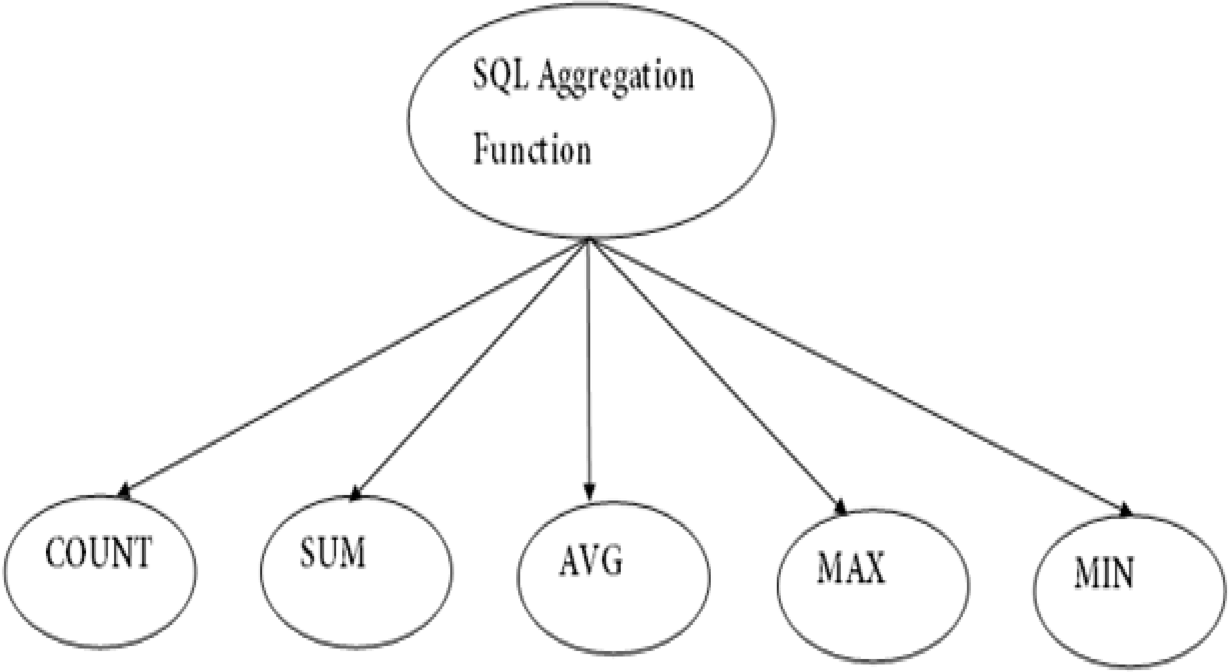
```
      Q1:      SELECT            SALARY
               FROM              EMPLOYEE
Q2    SELECT             DISTINCT SALARY
               FROM              EMPLOYEE
```

Aggregate functions

- Aggregate functions in DBMS take multiple rows from the table and return a value according to the query.
- SQL provides many aggregate functions that include **avg, count, sum, min, max,** etc. An aggregate function ignores NULL values when it performs the calculation, except for the count function.
- All the aggregate functions are used in Select statement.

Syntax –

```
SELECT <FUNCTION NAME> (<PARAMETER>) FROM <TABLE
NAME>
```


SQL Aggregation Function
COUNT
SUM
AVG
MAX
MIN

AVG Function

This function returns the average value of the numeric column that is supplied as a parameter.

Example: Write a query to select average salary from employee table.

```
Select AVG(salary) from Employee;
```

COUNT Function

The count function returns the number of rows in the result. It does not count the null values.

Example: Write a query to return number of rows where salary > 20000.

```
Select COUNT(*) from Employee where Salary > 20000;
```

Types −

- COUNT(*): Counts all the number of rows of the table including null.

MAX Function

The MAX function is used to find maximum value in the column that is supplied as a parameter. It can be used on any type of data.

Example − Write a query to find the maximum salary in employee table.

```
Select MAX(salary) from Employee;
```

SUM Function

This function sums up the values in the column supplied as a parameter.

Example: Write a query to get the total salary of employees.

```
Select SUM(salary) from Employee;
```

Id	Name	Salary
1	A	80
2	B	40
3	C	60
4	D	70
5	E	60
6	F	Null

- Count(*): Returns total number of records .i.e 6.
- Count(salary): Return number of Non Null values over the column salary. i.e 5.

- Sum():

sum(salary): Sum all Non Null values of Column salary i.e., 310

- Avg():

Avg(salary) = Sum(salary) / count(salary) = 310/5

Figure 3.6
One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

GROUPING

In many cases, we want to apply the aggregate functions to ***subgroups of tuples*** in a relation

Each subgroup of tuples consists of the set of tuples that have the *same value* for the *grouping attribute(s)*

The function is applied to each subgroup independently

SQL has a **GROUP BY**-clause for specifying the grouping attributes, which *must also appear in the SELECT-clause*

```
SELECT column1, column2

FROM table_name

WHERE [ conditions ]

GROUP BY column1, column2
```

GROUPING (contd.)

For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Query:     SELECT  DNO, COUNT (*), AVG (SALARY)
                   FROM  EMPLOYEE
                            GROUP BY DNO
```

In above query the EMPLOYEE tuples are divided into groups- Each group having the same value for the **grouping attribute DNO**

The **COUNT and AVG** functions are applied to each such group of tuples separately

The SELECT-clause includes only the grouping attribute and the functions to be applied on each group of tuples. A join condition can be used in conjunction with grouping

GROUPING (contd.)

Q . For each project, retrieve the project number, project name, and the number of employees who work on that project.

GROUPING (contd.)

Answer:

```
SELECT  PNUMBER, PNAME, COUNT (*)
FROM                PROJECT, WORKS_ON
WHERE PNUMBER=PNO
GROUP BY            PNUMBER, PNAME
```

THE HAVING-CLAUSE

Sometimes we want to retrieve the values of these functions for only those *groups that satisfy certain conditions*

The **HAVING**-clause is used for specifying a selection condition on groups (rather than on individual tuples)

Question: For each project *on which more than two employees work*, retrieve the project number, project name, and the number of employees who work on that project.

Query :

```
SELECT   PNUMBER, PNAME, COUNT(*)
FROM     PROJECT, WORKS_ON
WHERE    PNUMBER=PNO
GROUP BY         PNUMBER, PNAME
HAVING   COUNT (*) > 2
```

ORDER BY

The **ORDER BY** clause is used to sort the tuples in a query result based on the values of some attribute(s).

Query :- Retrieve a list of employees and the projects each works in, ordered by the employee's department, and within each department ordered alphabetically by employee last name.

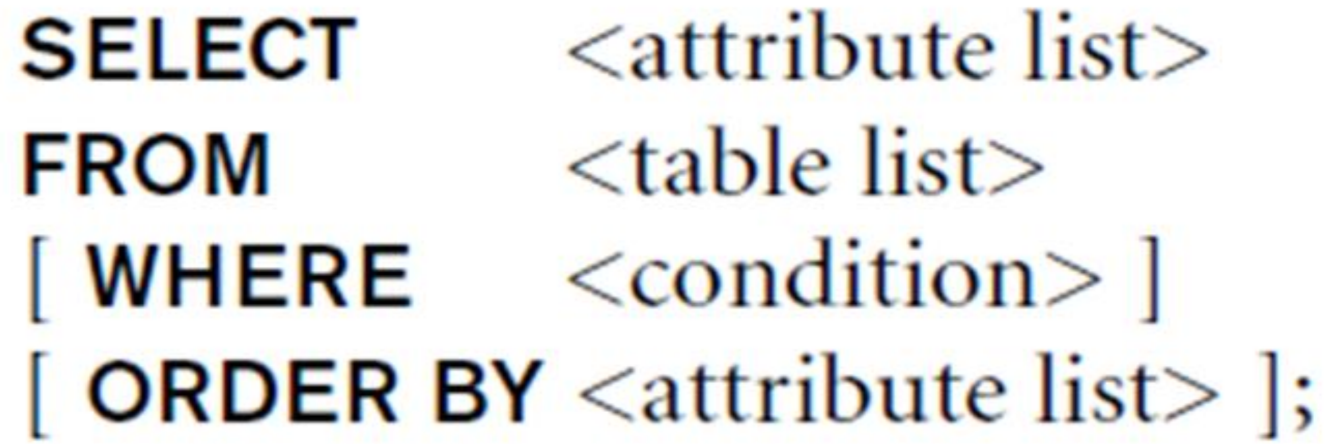
Q:

```
        SELECT  DNAME, LNAME, FNAME, PNAME
        FROM    DEPARTMENT, EMPLOYEE,
        WORKS_ON, PROJECT
                WHERE   DNUMBER=DNO AND SSN=ESSN
                AND PNO=PNUMBER
                ORDER BY DNAME, LNAME
```

Note: The default order is in **ascending order** of values

We can specify the keyword **DESC** if we want a descending order; the keyword ASC can be used to explicitly specify ascending order, even though it is the default

ORDER BY



```
SELECT        <attribute list>
FROM          <table list>
[ WHERE       <condition> ]
[ ORDER BY <attribute list> ];
```

ORDER BY

```
SQL> SELECT * FROM CUSTOMERS
   ORDER BY NAME;
```

This would produce the following result −

```
+----+----------+-----+-----------+
| ID | NAME     | AGE | ADDRESS   |
+----+----------+-----+-----------+-
|  4 | Chaitali |  25 | Mumbai    |
|  5 | Hardik   |  27 | Bhopal    |
|  3 | kaushik  |  23 | Kota      |
|  2 | Khilan   |  25 | Delhi     |
|  6 | Komal    |  22 | MP        |
|  7 | Muffy    |  24 | Indore    |
|  1 | Ramesh   |  32 | Ahmedabad |
   +----+----------+-----+-----------+----------+
```

COMBINATION OF GROUP BY ,HAVING AND ORDER BY

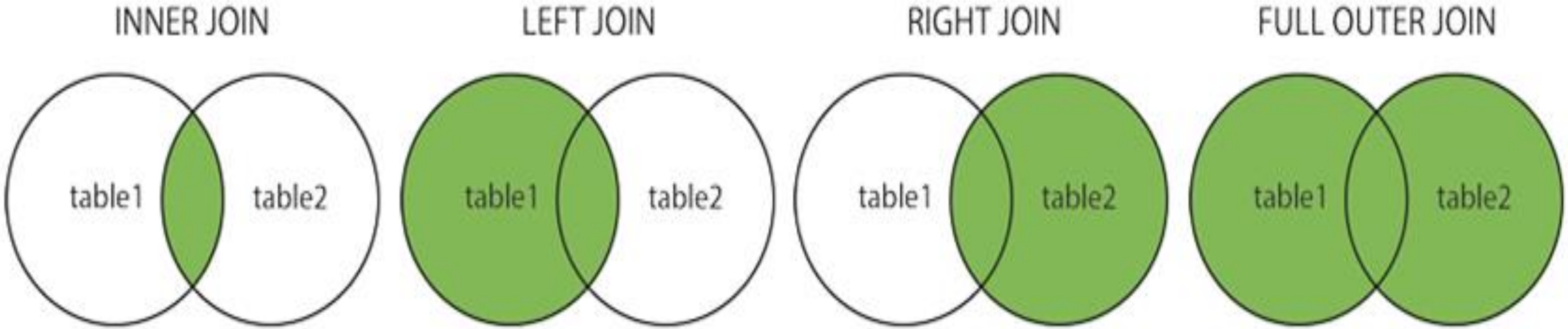
```
SELECT column1, column2

FROM table1

WHERE [ conditions ]

GROUP BY column1, column2

HAVING [ conditions ]

 ORDER BY column1, column2
```

SQL JOIN

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

Different Types of SQL JOINs:

- `(INNER) JOIN`: Returns records that have matching values in both tables.
- `LEFT (OUTER) JOIN`: Returns all records from the left table, and the matched records from the right table
- `RIGHT (OUTER) JOIN`: Returns all records from the right table, and the matched records from the left table
- `FULL (OUTER) JOIN`: Returns all records when there is a match in either left or right table


INNER JOIN
table1
table2
LEFT JOIN
table1
table2
RIGHT JOIN
table1
table2
FULL OUTER JOIN
table1
table2

Equi join

Equi join is the first type of Inner Join.

It joins two or more tables where the specified columns are equal.

In this type of join, we can only use '=' operator in comparing the columns.

Operators like '>', '<' are not allowed in this type of join.

```sql
SELECT table1.column1,table1.column2,table2.column1,...

FROM table1

INNER JOIN table2

ON table1.matching_column = table2.matching_column;
```

Employee

EmpId	EmpName
5	John
2	David
3	Peter
4	Eric

Area

AreaId	AreaName	EmpId
3	New York	1
4	Canada	3
5	Australia	3
6	England	4

Result:

EmpId	EmpName	AreaId	AreaName	EmpId
3	Peter	4	Canada	3
3	Peter	5	Australia	3
4	Eric	6	England	4

```
Select * from Employee emp
 JOIN Area area on area.EmpId = emp.EmpId
```

Natural join

It is same as equijoin but the difference is that in natural join, the common attribute appears only once.

```
Select * from Employee emp

NATURAL JOIN Area area on area.EmpId = emp.EmpId

result:
```

EmpId	EmpName	AreaId	AreaName
3	Peter	4	Canada
3	Peter	5	Australia
4	Eric	6	England

LEFT JOIN:

This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.

Syntax

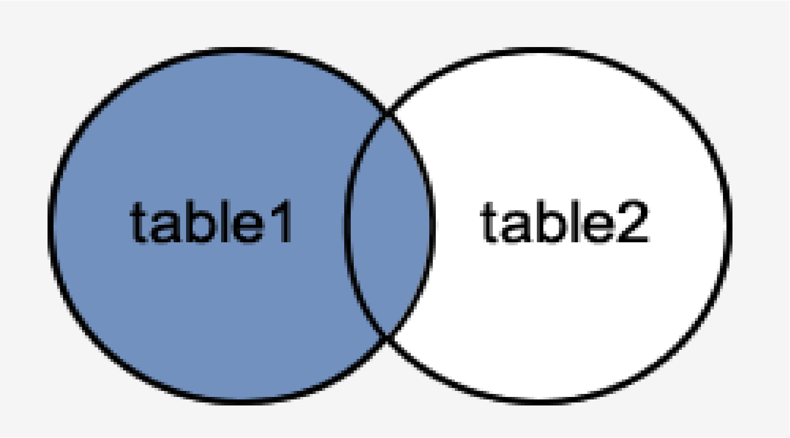
The syntax for the LEFT OUTER JOIN in SQL is:

```
SELECT columns

FROM table1

LEFT [OUTER] JOIN table2

 ON table1.column = table2.column;
```

In some databases, the OUTER keyword is omitted and written simply as LEFT JOIN.

Visual Illustration

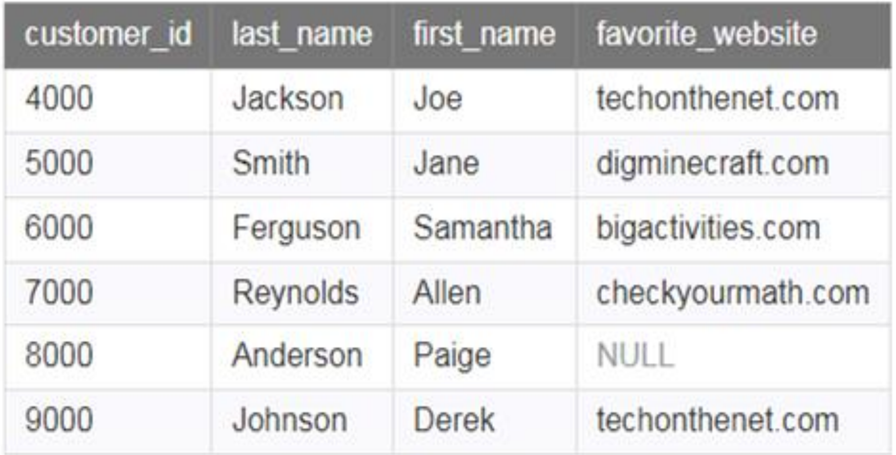
In this visual diagram, the SQL LEFT OUTER JOIN returns the shaded area:



The SQL LEFT OUTER JOIN would return the all records from *table1* and only those records from *table2* that intersect with *table1*.

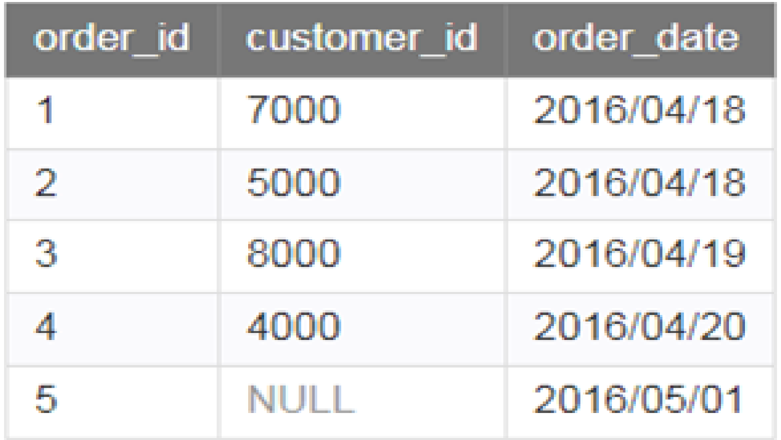
Example

Customer



customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

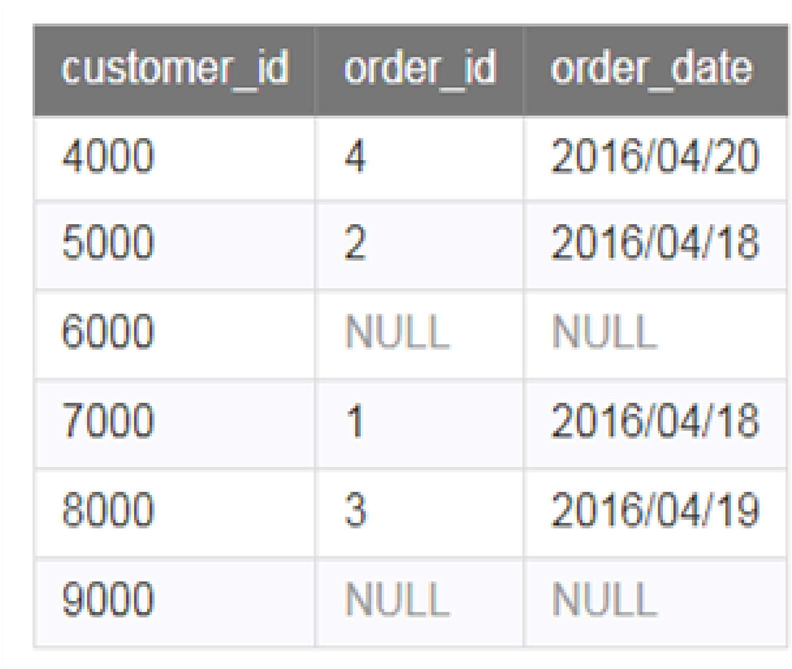
order table



order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

```
SELECT customers.customer_id, orders.order_id, orders.order_date

FROM customers

LEFT OUTER JOIN orders

ON customers.customer_id = orders.customer_id

 ORDER BY customers.customer_id;
```

There will be 6 records selected. These are the results :



customer_id	order_id	order_date
4000	4	2016/04/20
5000	2	2016/04/18
6000	NULL	NULL
7000	1	2016/04/18
8000	3	2016/04/19
9000	NULL	NULL

This LEFT OUTER JOIN example would return all rows from the *customers* table and only those rows from the *orders* table where the joined fields are equal.

If a *customer_id* value in the *customers* table does not exist in the *orders* table, all fields in the *orders* table will display as NULL in the result set. In example , the rows where *customer_id* is 6000 and 9000 would be included with a LEFT OUTER JOIN but the *order_id* and *order_date* fields display NULL.

SQL RIGHT OUTER JOIN

This type of join returns all rows from the RIGHT-hand table specified in the ON condition and **only** those rows from the other table where the joined fields are equal (join condition is met).

Syntax

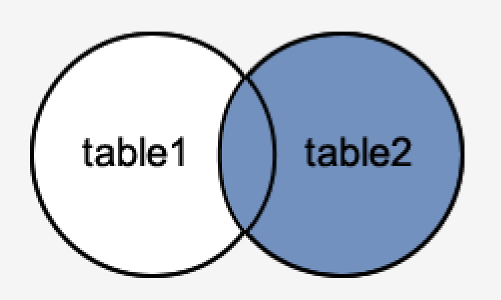
The syntax for the RIGHT OUTER JOIN in SQL is:

```
SELECT columns

FROM table1

RIGHT [OUTER] JOIN table2

 ON table1.column = table2.column;
```

In some databases, the OUTER keyword is omitted and written simply as RIGHT JOIN.

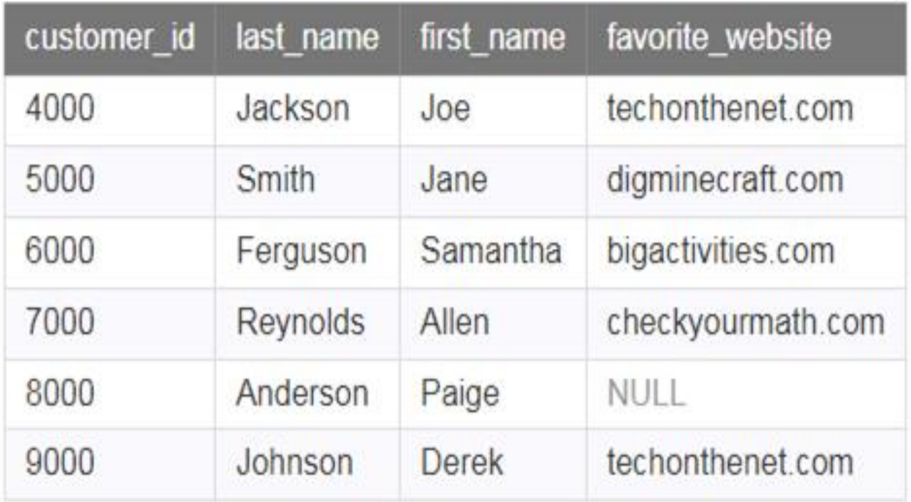
Visual Illustration:

In this visual diagram, the SQL RIGHT OUTER JOIN returns the shaded area:



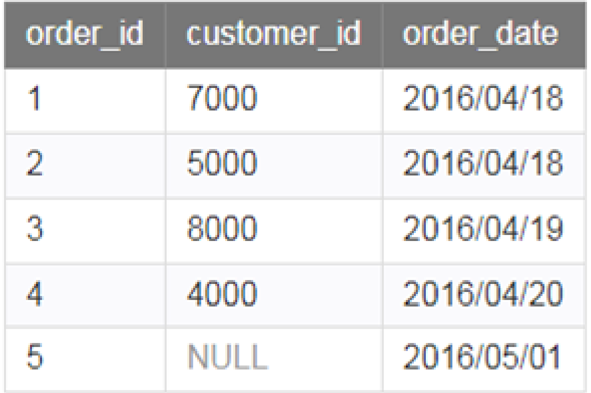
The SQL RIGHT OUTER JOIN would return the all records from *table2* and only those records from *table1* that intersect with *table2*.

Customer



customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

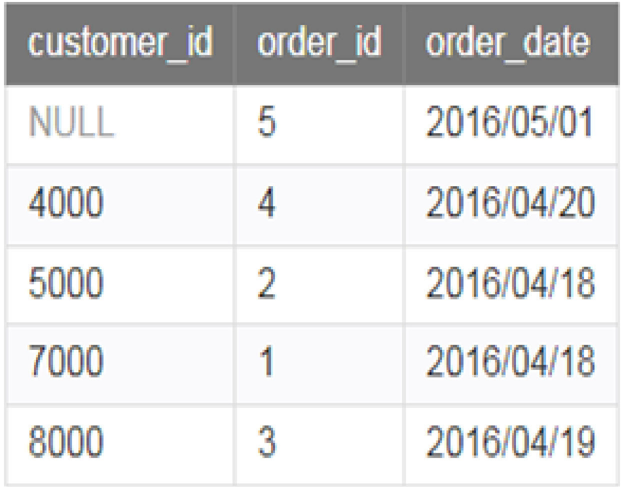
order



order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

```
SELECT customers.customer_id, orders.order_id, orders.order_date

FROM customers

RIGHT OUTER JOIN orders

ON customers.customer_id = orders.customer_id

ORDER BY customers.customer_id;
```

result



customer_id	order_id	order_date
NULL	5	2016/05/01
4000	4	2016/04/20
5000	2	2016/04/18
7000	1	2016/04/18
8000	3	2016/04/19

This RIGHT OUTER JOIN example would return all rows from the *orders* table and only those rows from the *customers* table where the joined fields are equal.

If a *customer_id* value in the *orders* table does not exist in the *customers* table, all fields in the *customers* table will display as NULL in the result set.in example, the row where *order_id* is 5 would be included with a RIGHT OUTER JOIN but the *customer_id* field displays NULL.

SQL FULL OUTER JOIN

This type of join returns all rows from the LEFT-hand table and RIGHT-hand table with NULL values in place where the join condition is not met.

Syntax

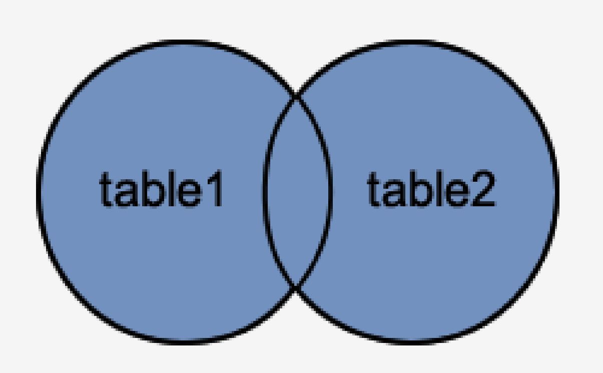
The syntax for the SQL **FULL OUTER JOIN** is:

```
SELECT columns

FROM table1

FULL [OUTER] JOIN table2

 ON table1.column = table2.column;
```

In some databases, the OUTER keyword is omitted and written simply as FULL JOIN.

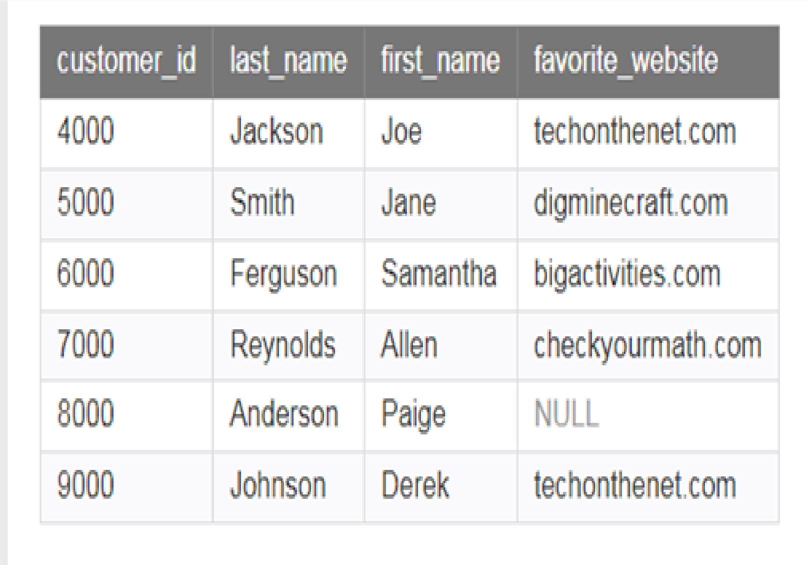
Visual Illustration

In this visual diagram, the SQL FULL OUTER JOIN returns the shaded area:



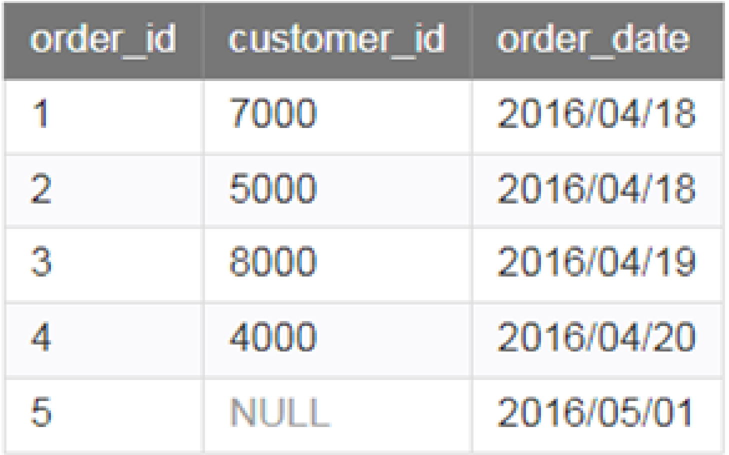
The SQL FULL OUTER JOIN would return the all records from both *table1* and *table2*.

Customer



customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

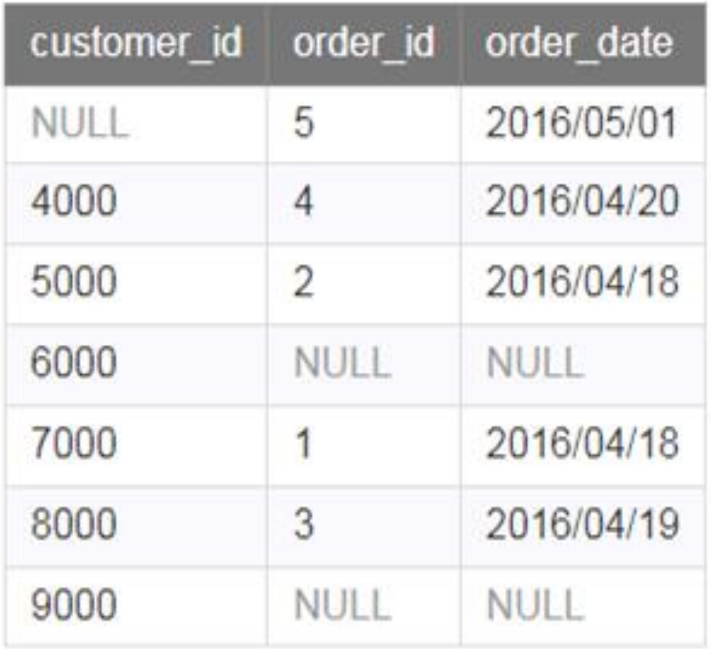
order



order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

```
SELECT customers.customer_id, orders.order_id, orders.order_date

FROM customers

FULL OUTER JOIN orders

ON customers.customer_id = orders.customer_id

 ORDER BY customers.customer_id;
```

result



customer_id	order_id	order_date
NULL	5	2016/05/01
4000	4	2016/04/20
5000	2	2016/04/18
6000	NULL	NULL
7000	1	2016/04/18
8000	3	2016/04/19
9000	NULL	NULL

This FULL OUTER JOIN example would return all rows from the *orders* table and all rows from the *customers* table. Whenever the joined condition is not met, a NULL value would be extended to those fields in the result set. This means that if a *customer_id* value in the *customers* table does not exist in the *orders* table, all fields in the *orders* table will display as NULL in the result set. Also, if a *customer_id* value in the *orders* table does not exist in the *customers* table, all fields in the *customers* table will display as NULL in the result set.

In example the rows where the *customer_id* is 6000 and 9000 would be included but the *order_id* and *order_date* fields for those records contains a NULL value. The row where the *order_id* is 5 would be also included but the *customer_id* field for that record has a NULL value.

Views in SQL

- Views in SQL are considered as a **virtual table**. A view also **contains rows and columns**.
- To create the view, we can select the fields from one or more tables present in the database.
- A view can either have specific rows based on certain condition or all the rows of a table.
- View is a result set of stored query.
- We can have read only or updatable views.

Advantages of Views in SQL

1. Restrict data access
2. To make complex queries
3. To provide data independence
4. To present different views of the same data.

Syntax for Creating Views in SQL

Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
   WHERE [condition];
```

Example:

```
Create view Student1 as
Select * from student
Where student_id>2;
```

Example of Views in SQL

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Example of Views in SQL

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
 FROM  CUSTOMERS;
```

you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

Syntax for Creating Views in SQL

This would produce the following result.

```
+----------+-----+
| name     | age |
+----------+-----+
| Ramesh   |  32 |
| Khilan   |  25 |
| kaushik  |  23 |
| Chaitali |  25 |
| Hardik   |  27 |
| Komal    |  22 |
| Muffy    |  24 |
 +----------+-----+
```

Creating View from multiple tables

Query:

```
CREATE VIEW MarksView AS
SELECT Student_Detail.NAME, Student_Detail.ADDRESS,
Student_Marks.MARKS
FROM Student_Detail, Student_Mark
WHERE Student_Detail.NAME = Student_Marks.NAME;
```

Updating a View

A view can be updated **under certain conditions** which are given below −

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So, if a view satisfies all the above-mentioned rules then you can update that view.

Updating a View

```
SQL > UPDATE CUSTOMERS_VIEW
   SET AGE = 35
    WHERE name = 'Ramesh';
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
 +----+----------+-----+-----------+----------+
```

This would ultimately update the base table **CUSTOMERS** and the **same would reflect in the view** itself.

Drop View

A view can be deleted using the Drop View statement.

Syntax

1. DROP VIEW view_name;

Example:

If we want to delete the View **MarksView**, we can do this as:

1. DROP VIEW MarksView;

Triggers

1. Triggers are the SQL statements that are **automatically executed** when there is any change in the database.
2. The triggers are executed **in response to certain events**(INSERT, UPDATE or DELETE) in a particular table.
3. These triggers help in maintaining the **integrity** of the data by changing the data of the database in a systematic fashion.
4. A trigger is called a special procedure . key distinction between the trigger and procedure is that a trigger is called automatically when a data modification event occurs against a table.

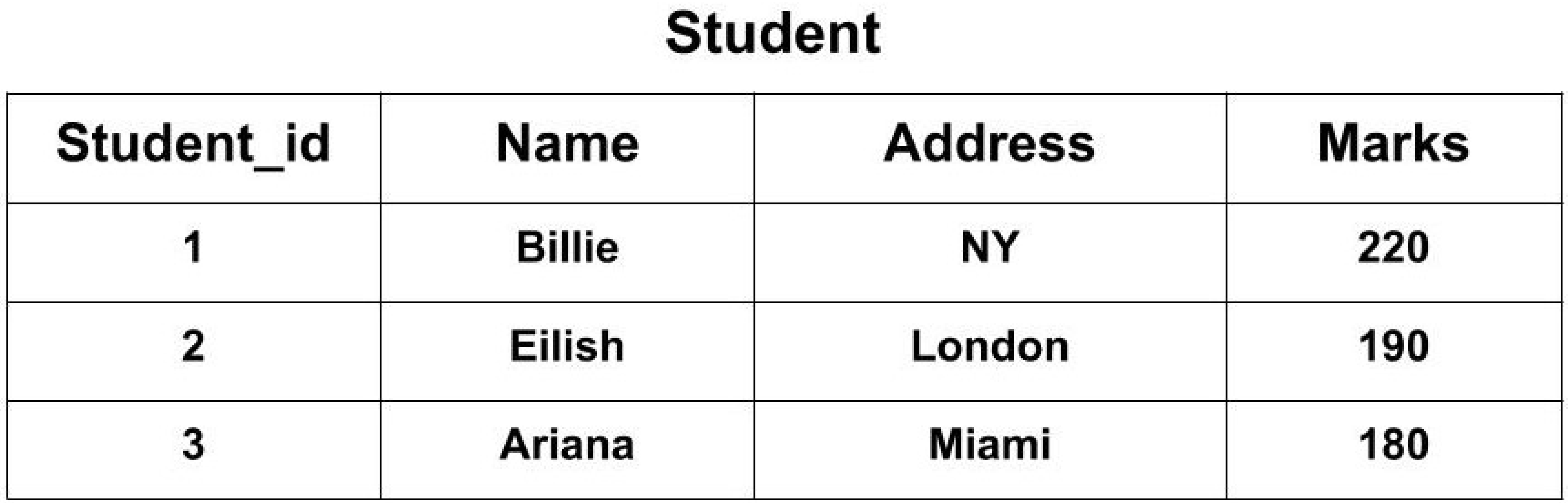
Syntax to Create Triggers

```
create trigger Trigger_name
 (before | after)
[insert | update | delete]
on [table_name]
 [for each row]
[trigger_body]
```

1. **CREATE TRIGGER:** These two keywords specify that a triggered block is going to be declared.
2. **TRIGGER_NAME:** It creates or replaces an existing trigger with the Trigger_name. The trigger name should be unique.
3. **BEFORE | AFTER:** It specifies when the trigger will be initiated i.e. before the ongoing event or after the ongoing event.
4. **INSERT | UPDATE | DELETE**: These are the DML operations and we can use either of them in a given trigger.
5. **ON[TABLE_NAME]:** It specifies the name of the table on which the trigger is going to be applied.
6. **FOR EACH ROW:** Row-level trigger gets executed when any row value of any column changes.
7. **TRIGGER BODY:** It consists of queries that need to be executed when the trigger is called.

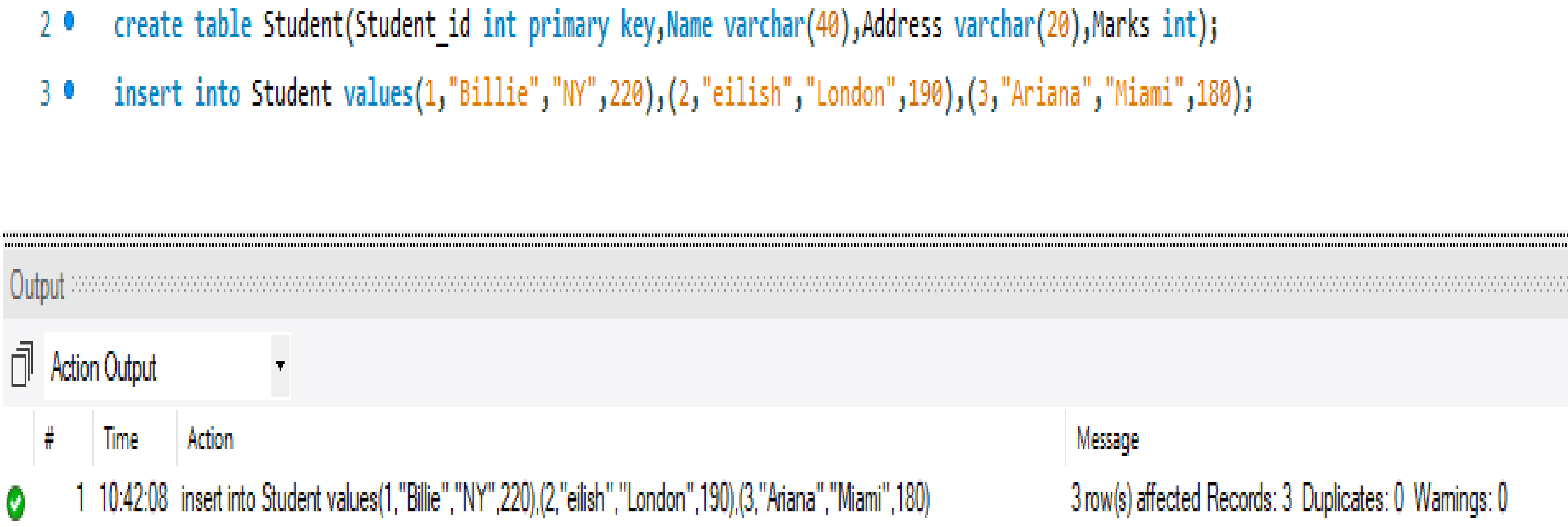
Example of a trigger

- **Example**
- Suppose we have a table named **Student** containing the attributes *Student_id, Name, Address, and Marks.*



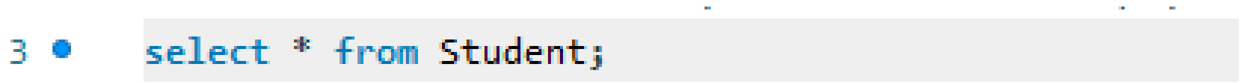
Student

Student_id	Name	Address	Marks
1	Billie	NY	220
2	Eilish	London	190
3	Ariana	Miami	180

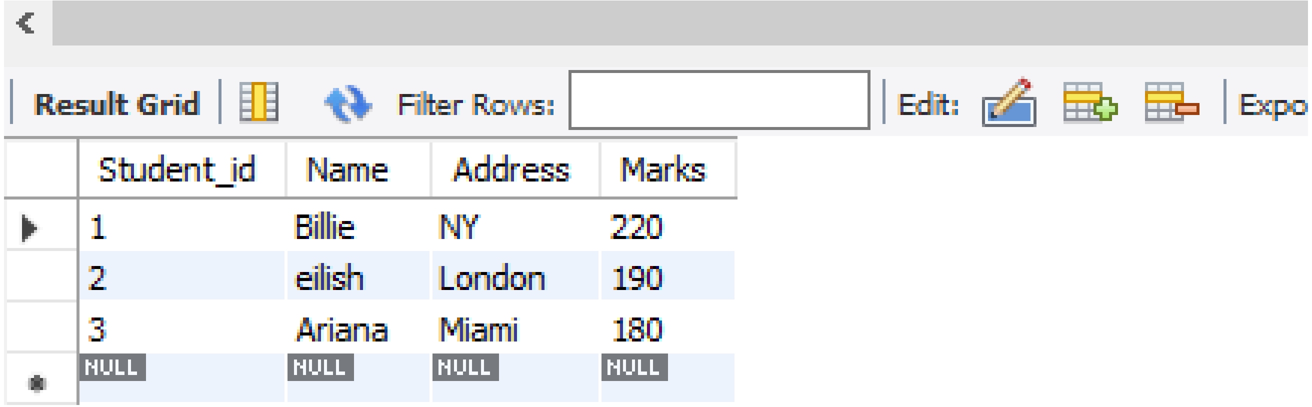


```
create table Student(Student_id int primary key,Name varchar(40),Address varchar(20),Marks int);
insert into Student values(1,"Billie","NY",220),(2,"eilish","London",190),(3,"Ariana","Miami",180);
```

#	Time	Action	Message
1	10:42:08	insert into Student values(1,"Billie","NY",220),(2,"eilish","London",190),(3,"Ariana","Miami",180)	3 row(s) affected Records: 3 Duplicates: 0 Warnings: 0



```
select * from Student;
```

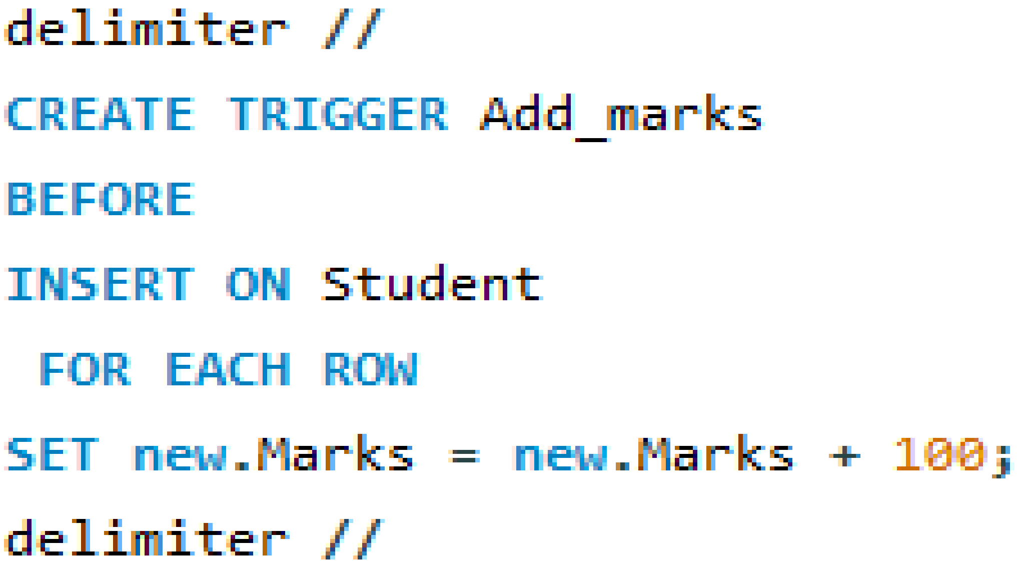


Student_id	Name	Address	Marks
1	Billie	NY	220
2	eilish	London	190
3	Ariana	Miami	180
NULL	NULL	NULL	NULL

Example of a trigger

```
CREATE TRIGGER Add_marks
BEFORE
INSERT ON Student
 FOR EACH ROW
SET new.Marks = new.Marks + 100;
```

Example of a trigger



```
delimiter //
CREATE TRIGGER Add_marks
BEFORE
INSERT ON Student
 FOR EACH ROW
SET new.Marks = new.Marks + 100;
delimiter //
```

Note: The DELIMITER // can be used to change the statement from semicolon (;) to //. Now you can write multiple statements with semi-colon in a trigger.

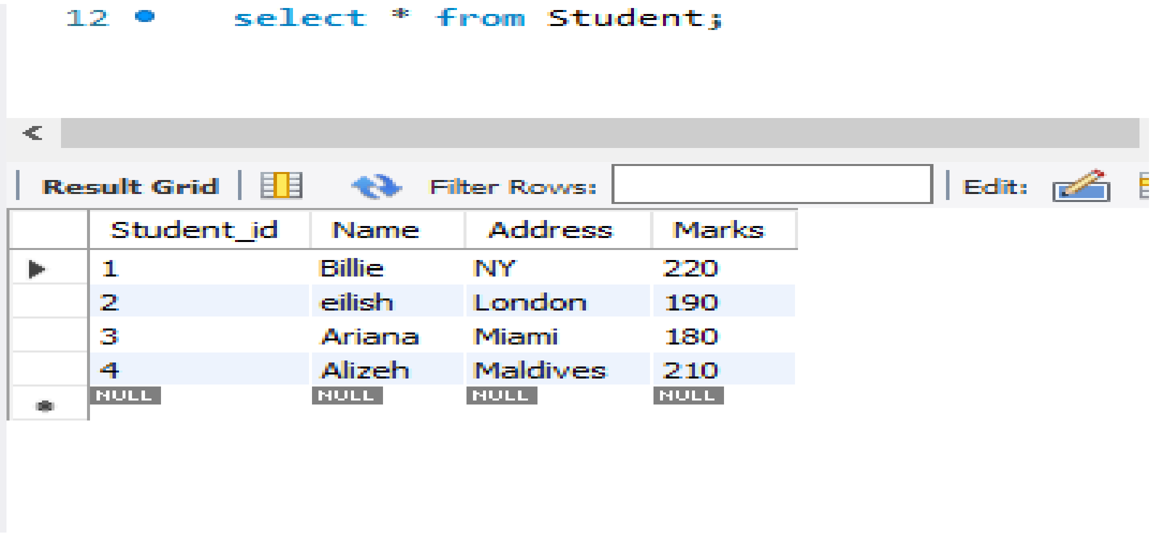
Example of a trigger

The new keyword refers to the row that is getting affected.

After creating the trigger, we will write the **query for inserting a new student** in the database.

```
INSERT INTO Student(Student_id,Name, Address, Marks) VALUES(4,'Alizeh',
'Maldives', 110);
```

To see the final output the query would be



```
select * from Student;
```

Student_id	Name	Address	Marks
1	Billie	NY	220
2	eilish	London	190
3	Ariana	Miami	180
4	Alizeh	Maldives	210
NULL	NULL	NULL	NULL

Note : To **delete trigger** you can use drop trigger trigger_name;

Advantages of Triggers

1. Triggers provide a way to **check the integrity** of the data. When there is a change in the database the triggers can adjust the entire database.
2. Triggers help in keeping **User Interface lightweight**. Instead of putting the same function call all over the application you can put a trigger and it will be executed.

Disadvantages of Triggers

1.Triggers may be difficult to troubleshoot as they execute automatically in the database. If there is some error then it is hard to find the logic of trigger because they are fired before or after updates/inserts happen.

2.The triggers may increase the overhead of the database as they are executed every time any field is updated.

Procedures in SQL

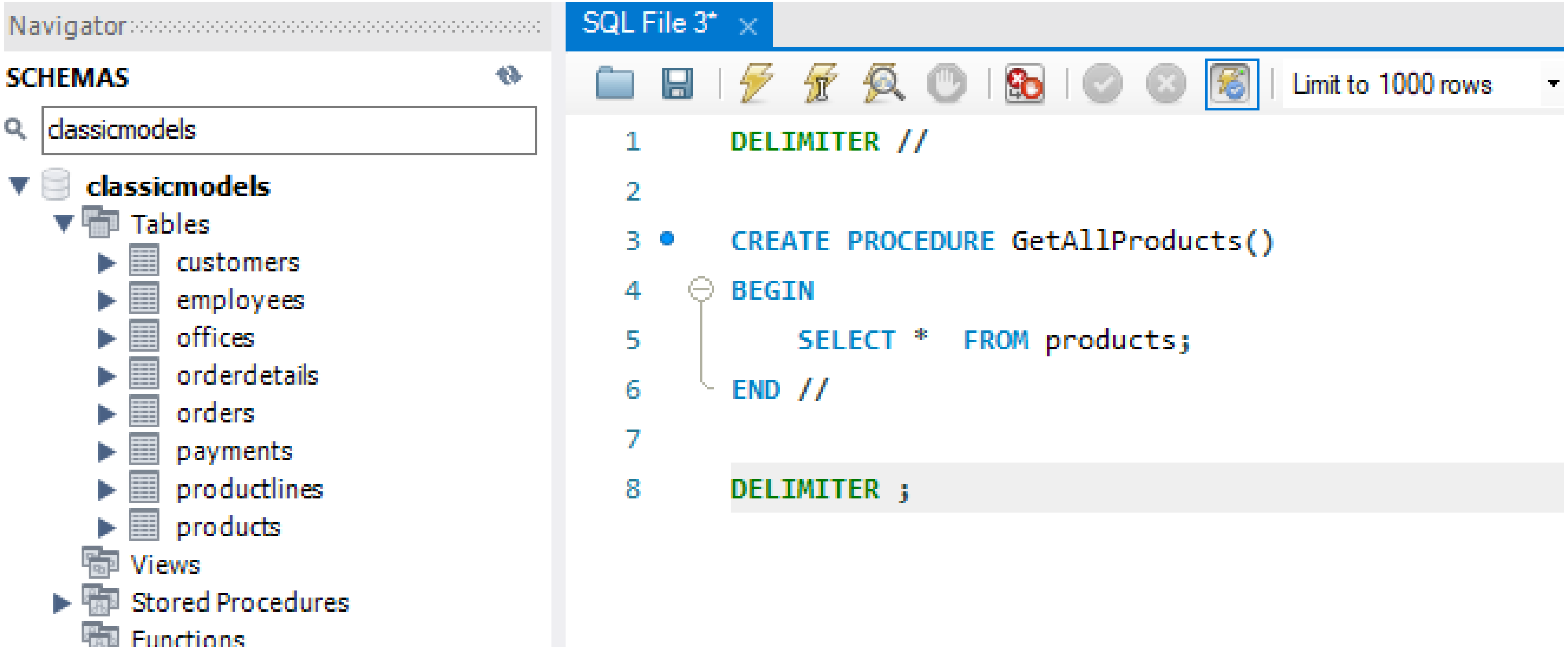
A **stored procedure** is a group of pre-compiled SQL statements (prepared SQL code) that can be reused again and again.

They can be used to perform a wide range of database operations such as inserting, updating, or deleting data, generating reports, and performing complex calculations.

Syntax to create Procedures in SQL

```
DELIMITER &&
CREATE PROCEDURE procedure_name [[IN | OUT | INOUT] parameter_name datatype [, parameter datatype]) ]
BEGIN
    Declaration_section
    Executable_section
END &&
DELIMITER ;
```

Example of Procedures without any parameter in SQL



```
DELIMITER //

CREATE PROCEDURE GetAllProducts()
BEGIN
    SELECT *  FROM products;
END //

DELIMITER ;
```

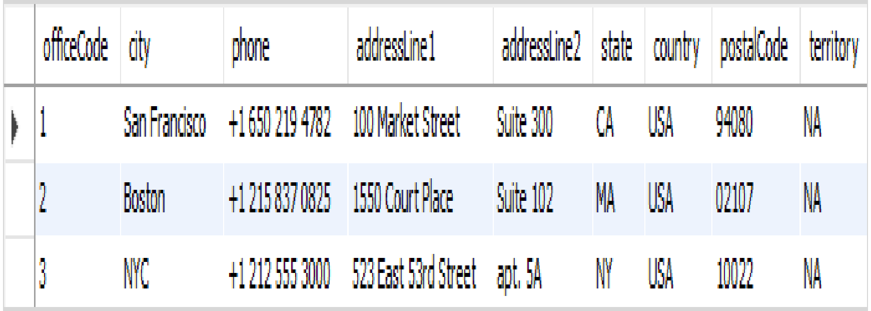
Example of Procedures with one input parameter in SQL

How to create procedure with one input parameter

```
DELIMITER //

CREATE PROCEDURE GetOfficeByCountry(
        IN countryName VARCHAR(255))
BEGIN
        SELECT *
        FROM offices
        WHERE country = countryName;
END //

DELIMITER ;
```

Calling a Procedure

```
CALL GetOfficeByCountry('USA');
```



officeCode	city	phone	addressLine1	addressLine2	state	country	postalCode	territory
1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA
2	Boston	+1 215 837 0825	1550 Court Place	Suite 102	MA	USA	02107	NA
3	NYC	+1 212 555 3000	523 East 53rd Street	apt. 5A	NY	USA	10022	NA

Note:- countryName is the IN parameter

How to execute and delete a procedure

To execute a Stored Procedure use below Syntax

Call Procedure_name;

To delete existing procedure we can use below syntax

DROP PROCEDURE procedure_name ;

To check whether the **process is deleted or not** use below syntax

SHOW PROCEDURE STATUS WHERE db = ‘database_name';

Functions in MySQL

Function can also be created. A function always **returns a value using the return statement**. The function can be used in SQL queries.

Syntax

```
CREATE FUNCTION function_name [ (parameter datatype, parameter datatype]) ]
RETURNS return_datatype
BEGIN
Declaration_section
Executable_section
END;
```

Functions in MySQL

Parameter:

Function_name: name of the function

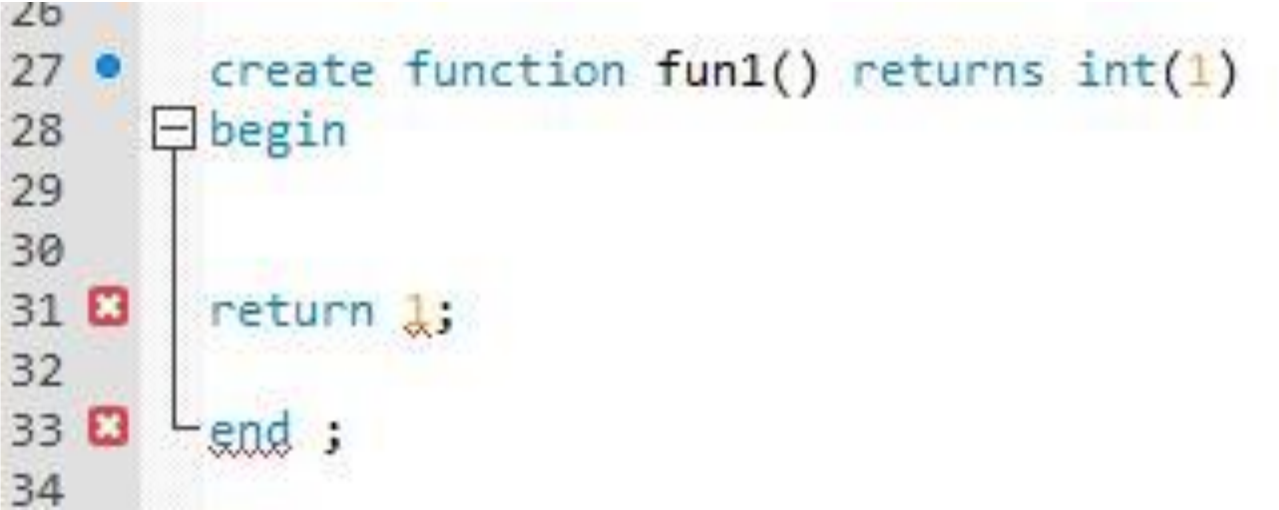
Parameter: number of parameter. It can be one or more than one.

return_datatype: return value datatype of the function

declaration_section: all variables are declared.

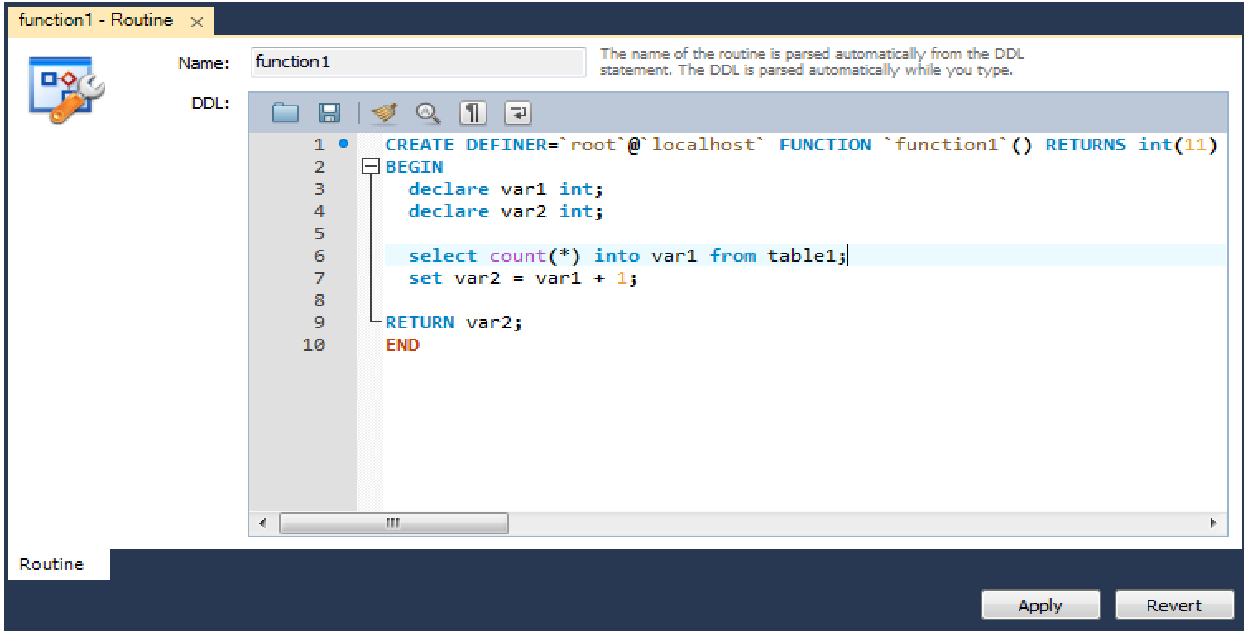
executable_section: code for the function is written here.

Example of Functions in MySQL



```
create function fun1() returns int(1)
begin


return 1;

end ;
```

Example of Functions in MySQL



Name: function1

The name of the routine is parsed automatically from the DDL statement. The DDL is parsed automatically while you type.

DDL:

```
CREATE DEFINER=`root`@`localhost` FUNCTION `function1`() RETURNS int(11)
BEGIN
  declare var1 int;
  declare var2 int;

  select count(*) into var1 from table1;
  set var2 = var1 + 1;

RETURN var2;
END
```

Note : We cannot use the DML statements in a function, (functions such as Update, Delete, and Insert).

How to delete a Function in MySQL

Following is the syntax the DELETE FUNCTION statement –

DROP FUNCTION function_name ;

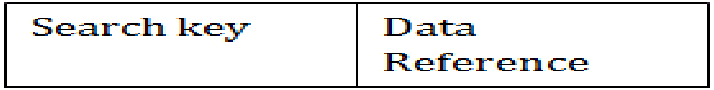
Indexing in DBMS

Indexing mechanisms used to **speed up access to desired data**.

E.g., author catalog in library

Search Key - attribute to set of attributes used to look up records in a file.

An **index file** consists of records (called **index entries**) of the form



Search key	Data Reference

Fig: Structure of Index

Index files are typically much smaller than the original file

Two basic kinds of indices:

Ordered indices: search keys are stored in sorted order

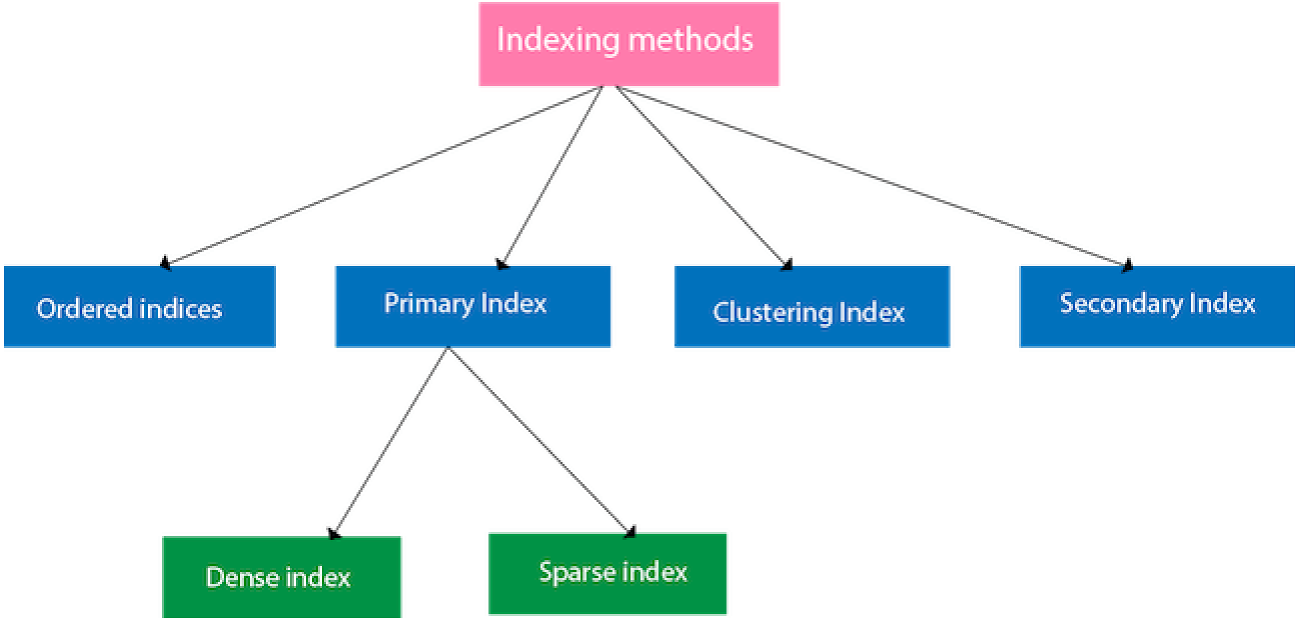
Hash indices: search keys are distributed uniformly across "buckets" using a "hash function".

Indexing in DBMS

Definition - Indexing is a data structure technique to retrieve records from the database files efficiently

Index **-** are the special lookup tables which are available to only database search engine for accessing data.It speedups the searching process.

Types of Indexing



Ordered Indices

Ordered indices

The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.

Primary Index

Primary Index

•

If the index is created on the **basis of the primary key** of the table, then it is known as **primary indexing**.

•These primary keys are **unique to each record** and contain 1:1 relation between the records.

•As primary keys are stored in **sorted order**, the performance of the searching operation is **quite efficient.**

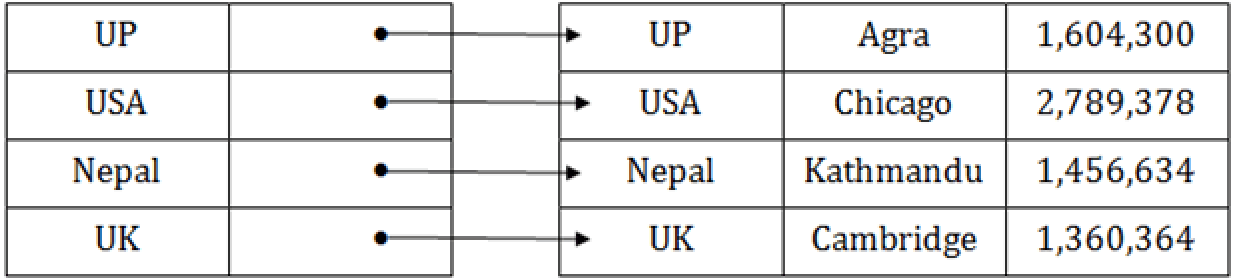
•The primary index can be classified into **two types**: Dense index and Sparse index.

Dense index

- The dense index contains an index record for **every search key value** in the data file. It makes **searching faster**.
- In this, the number of records in the index table is same as the number of records in the main table.

Number of records in index table = number of records in main table

- It needs **more space** to store index record itself. The index records have the search key and a pointer to the actual record on the disk.

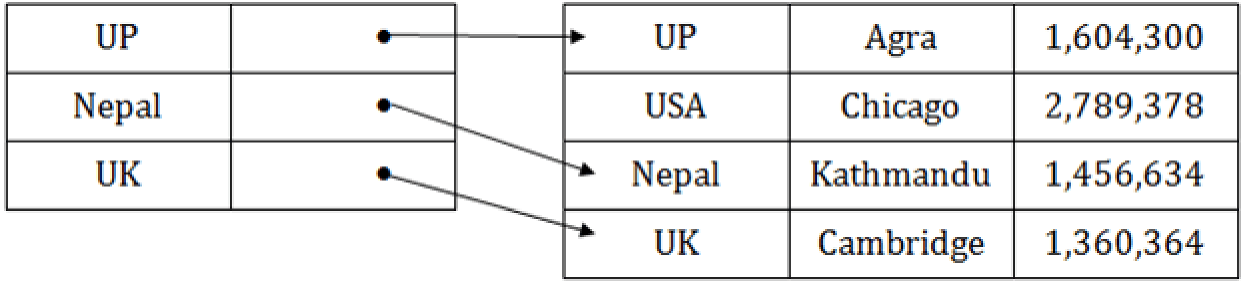


UP	•
USA	•
Nepal	•
UK	•

UP	Agra	1,604,300
USA	Chicago	2,789,378
Nepal	Kathmandu	1,456,634
UK	Cambridge	1,360,364

Sparse index

- In the data file, index record appears **only for a few items**. Each item points to a block.
- In this, instead of pointing to each record in the main table, the index points to the records in the **main table in a gap**.



UP	●
Nepal	●
UK	●

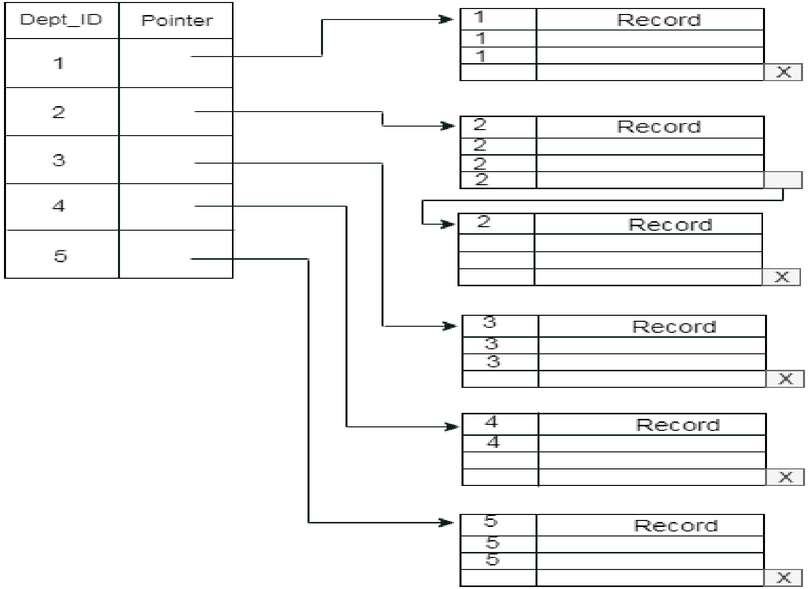
UP	Agra	1,604,300
USA	Chicago	2,789,378
Nepal	Kathmandu	1,456,634
UK	Cambridge	1,360,364

Clustering Index

- A clustered index can be defined as an **ordered data file.** Sometimes the index is created on **non-primary key columns** which may **not be unique** for each record.
- In this case, to identify the record faster, we will **group two or more columns** to get the unique value and **create index** out of them. This method is called a **clustering index.**
- The records which have similar characteristics are grouped, and indexes are created for these group.

Clustering Index

Example: Suppose a company contains several employees in each department. Suppose we use a clustering index, where all employees which belong to the same **Dept_ID** are considered within a **single cluster,** and index pointers point to the **cluster as a whole**. Here **Dept_Id is a non-unique key.**

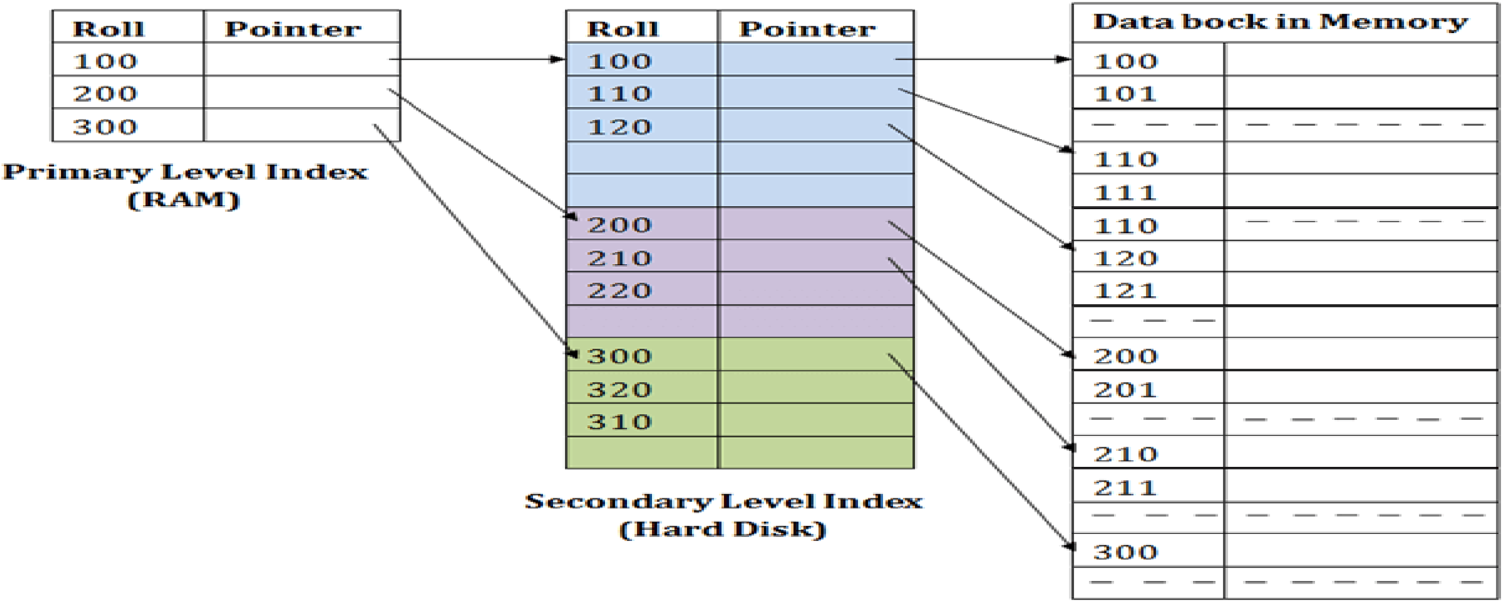


Secondary Index

To reduce the **size of mapping in** sparse indexing , another level of indexing is introduced. In this method, the **huge range for the columns** is selected initially so that the mapping size of the **first level becomes small**. Then each range is further divided into smaller ranges. The mapping of the first level is stored in the primary memory, so that address fetch is faster. The mapping of the **second level** and actual data are stored in the **secondary memory (hard disk).**

Secondary Index

How are you going to find out 111 in disk?



Roll	Pointer
100	
200	
300	

Primary Level Index (RAM)

Roll	Pointer
100	
110	
120	
200	
210	
220	
300	
320	
310	

Secondary Level Index (Hard Disk)

Data bock in Memory	
100	
101	
— — —	— — — — — —
110	
111	
110	— — — — — —
120	
121	
— — —	
200	
201	
— — —	— — — — — —
210	
211	
— — —	— — — — — —
300	
— — —	— — — — — —