Unit 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

- **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 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 VARCHAR2(20), Email VARCHAR2(100), DOB DATE);

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

Syntax

```
1.  DROP TABLE table_name;
```

Example

```
1.  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

1. ALTER TABLE table_name ADD column_name COLUMN-definition;

To modify existing column in the table:

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

EXAMPLE

1. ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));
2. 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:

```
1.    TRUNCATE TABLE table_name;
```

Example:

```
1.    TRUNCATE TABLE EMPLOYEE;
```

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 * FROM <table_name>;
```

Example :

```
SELECT * FROM students;
```

OR

```
SELECT * FROM students where due_fees <=20000;
```

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';
```

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

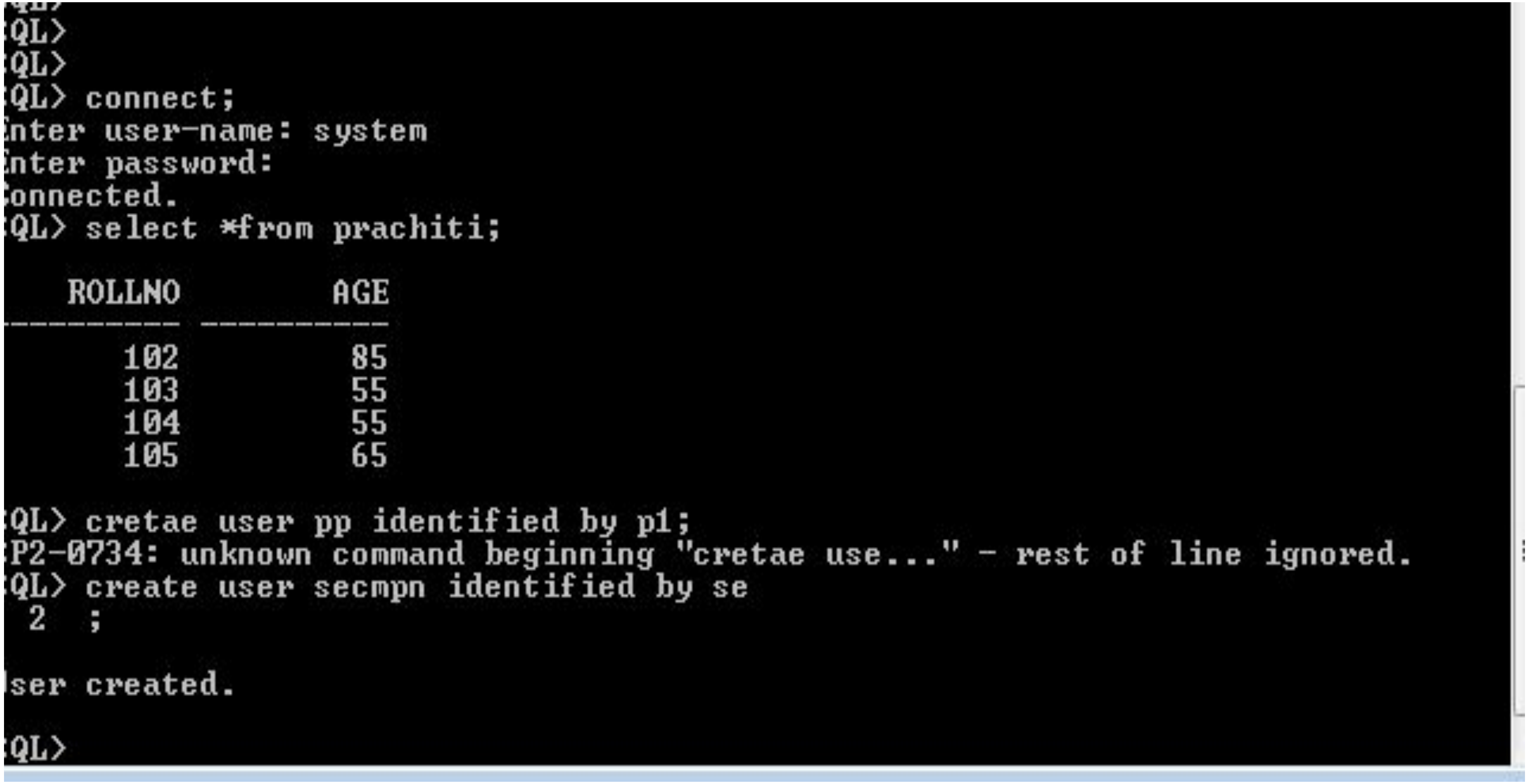
1. Create a new user with password:

   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 Oracle user:

   grant dba to user_name;



```
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

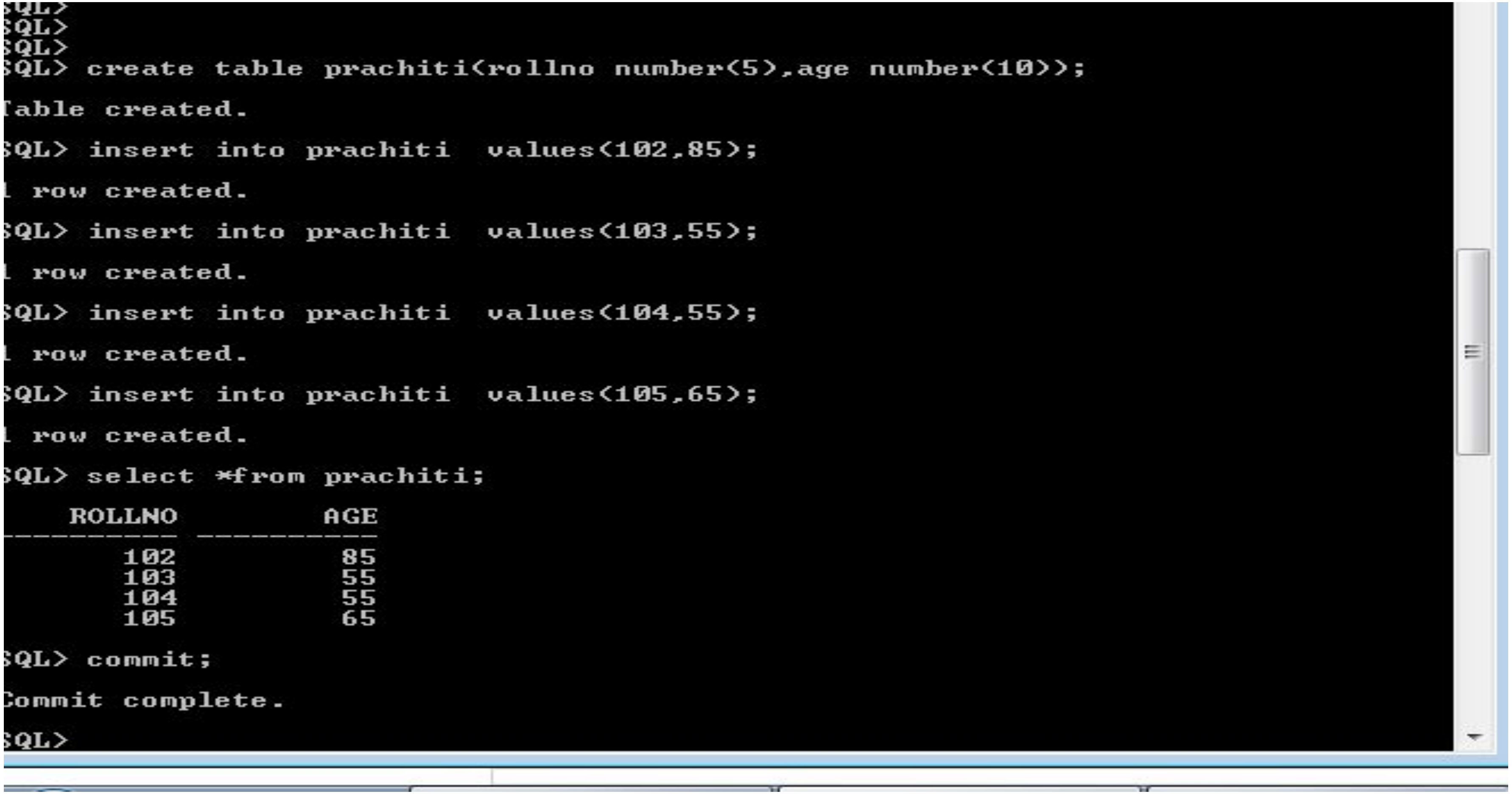
a. Commit: Commit command is used to save all the transactions to the database.

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.

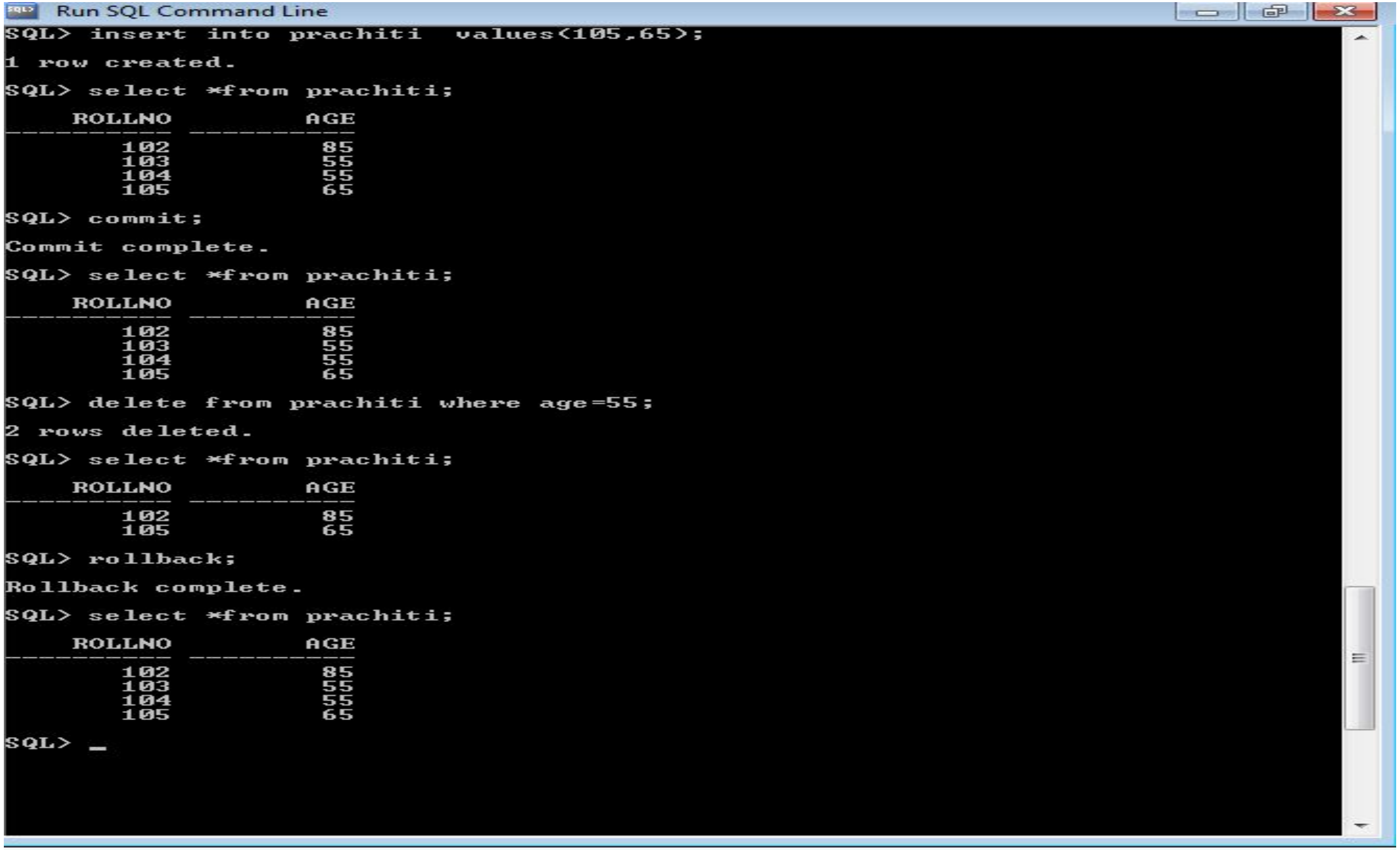
If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.

Syntax:

```
1.   ROLLBACK;
```

Example:

```
1.   DELETE FROM CUSTOMERS  WHERE AGE = 25;

2.   ROLLBACK;
```



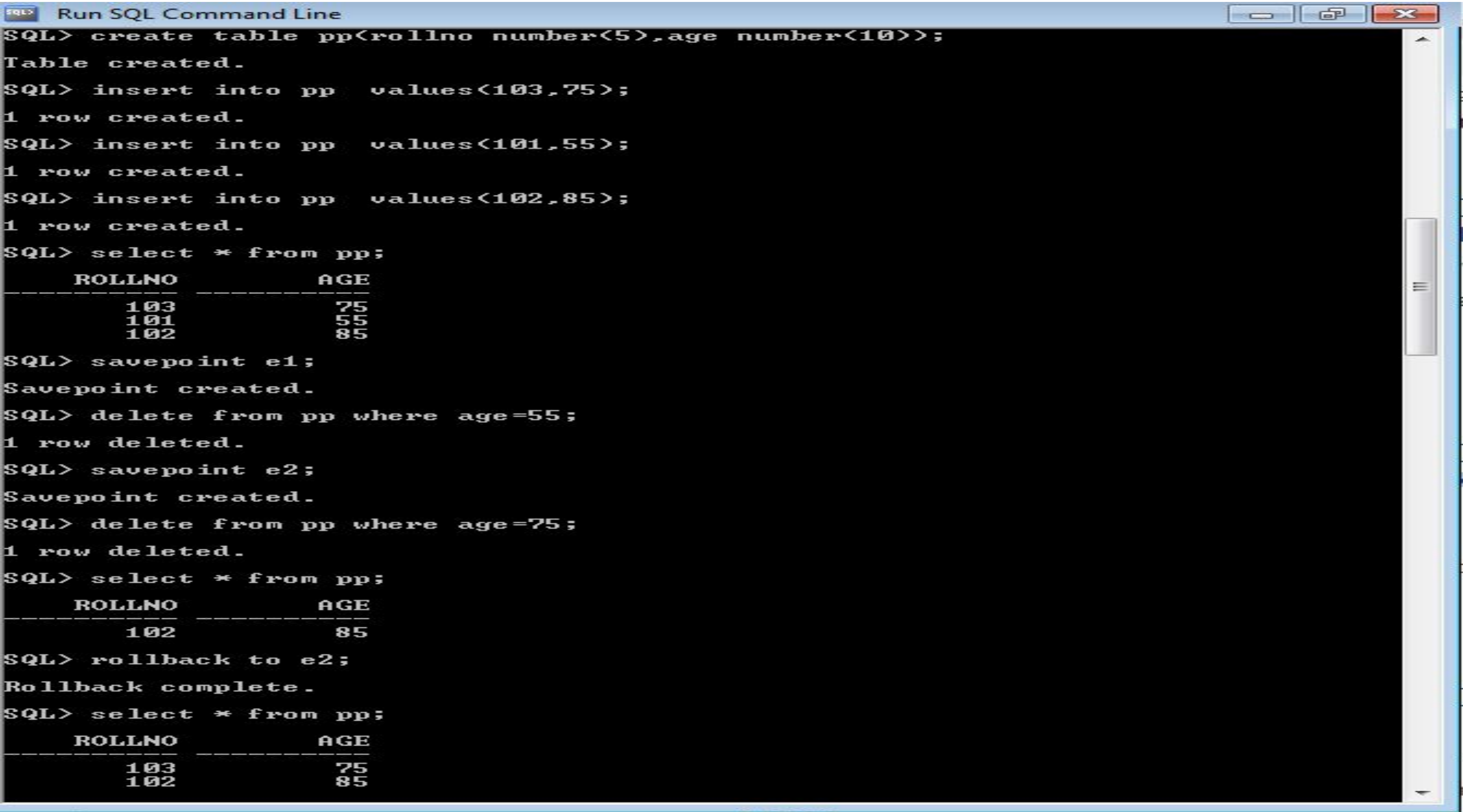
Run SQL Command Line

```
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> _
```

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:

1. SAVEPOINT SAVEPOINT_NAME;



Run SQL Command Line

```
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

- 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

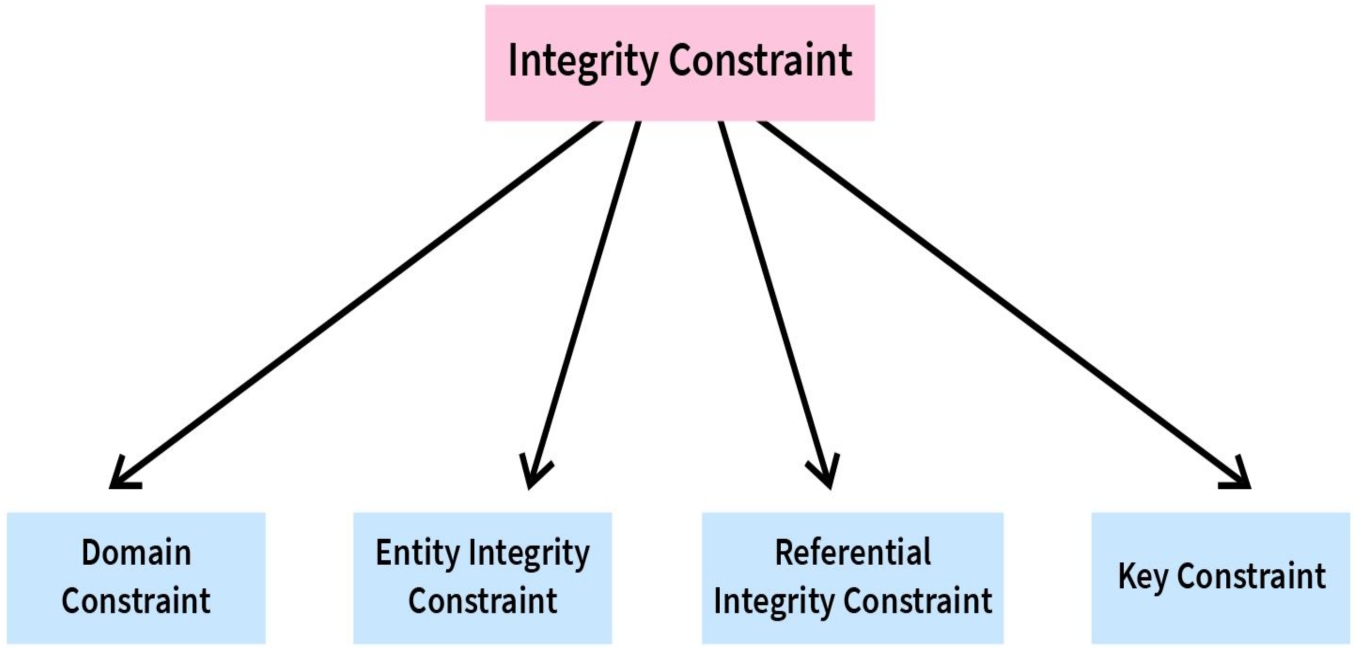
- The Multi-line comments start with **/*** and end with ***/**. So, any text mentioned between /* and */ will be ignored by the compiler.

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

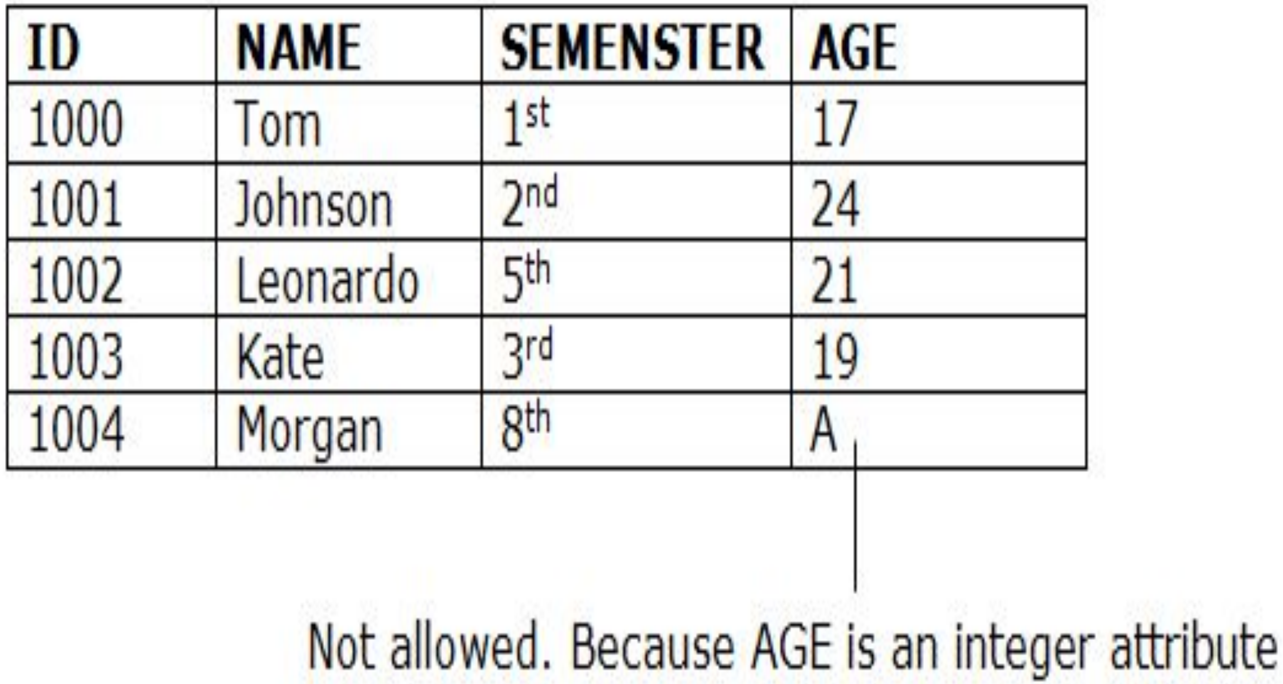
There are four types of integrity constraints in DBMS:

1. Domain Constraint
2. Entity Constraint
3. Referential Integrity Constraint
4. Key Constraint

Domain constraints

- Domain constraints can be defined as the definition of a valid set of values for an attribute.
- The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

Example:



ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1004	Morgan	8th	A

Not allowed. Because AGE is an integer attribute

Consider a Student's table having Roll No, Name, Age, Class of students.

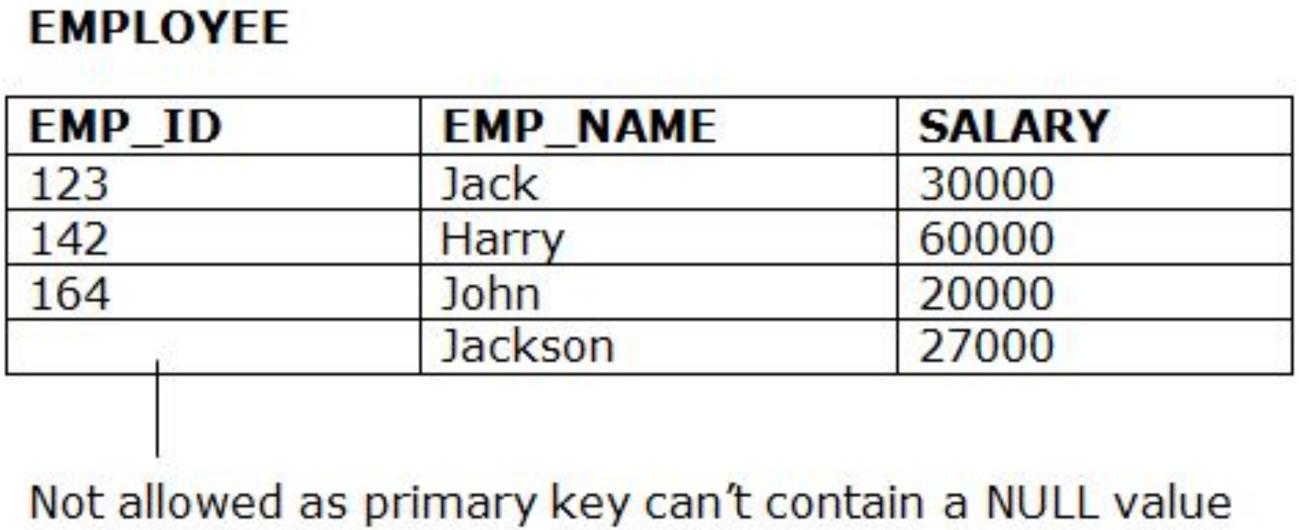
Roll No	Name	Age	Class
101	Adam	14	6
102	Steve	16	8
103	David	8	4
104	Bruce	18	12
105	Tim	6	xyd

In the above student's table, the value xyd in the last row last column violates the domain integrity constraint because the Class attribute contains only integer values while xyd is a character.

2. Entity integrity constraints

- The entity integrity constraint states that primary key value can't be null.
- This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
- A table can contain a null value other than the primary key field.

Example:



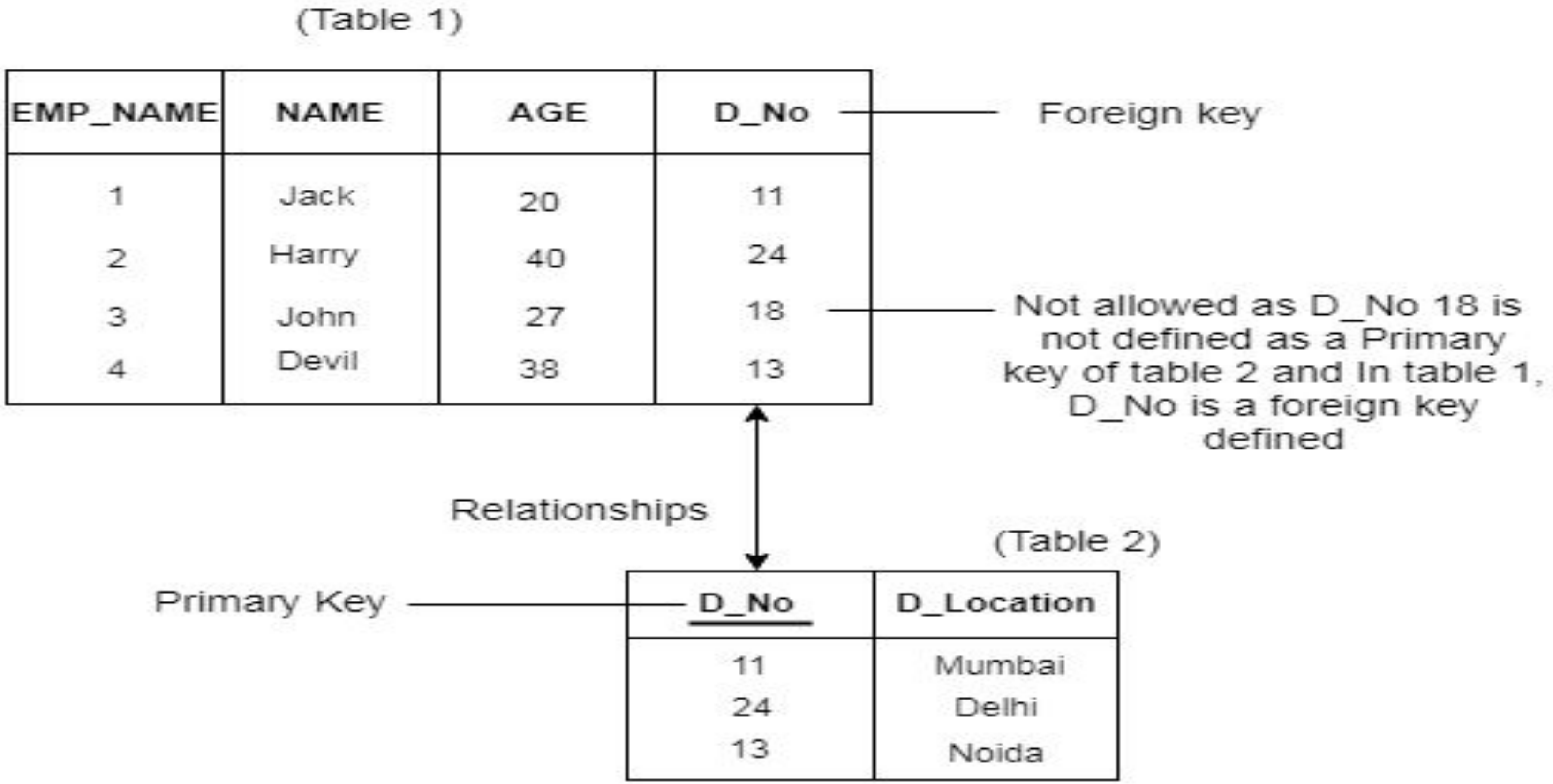
EMPLOYEE

EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

Not allowed as primary key can't contain a NULL value

3. Referential Integrity Constraints

- A referential integrity constraint is specified between two tables.
- In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.



(Table 1)

EMP_NAME	NAME	AGE	D_No
1	Jack	20	11
2	Harry	40	24
3	John	27	18
4	Devil	38	13

Foreign key

Not allowed as D_No 18 is not defined as a Primary key of table 2 and In table 1, D_No is a foreign key defined

Relationships

(Table 2)

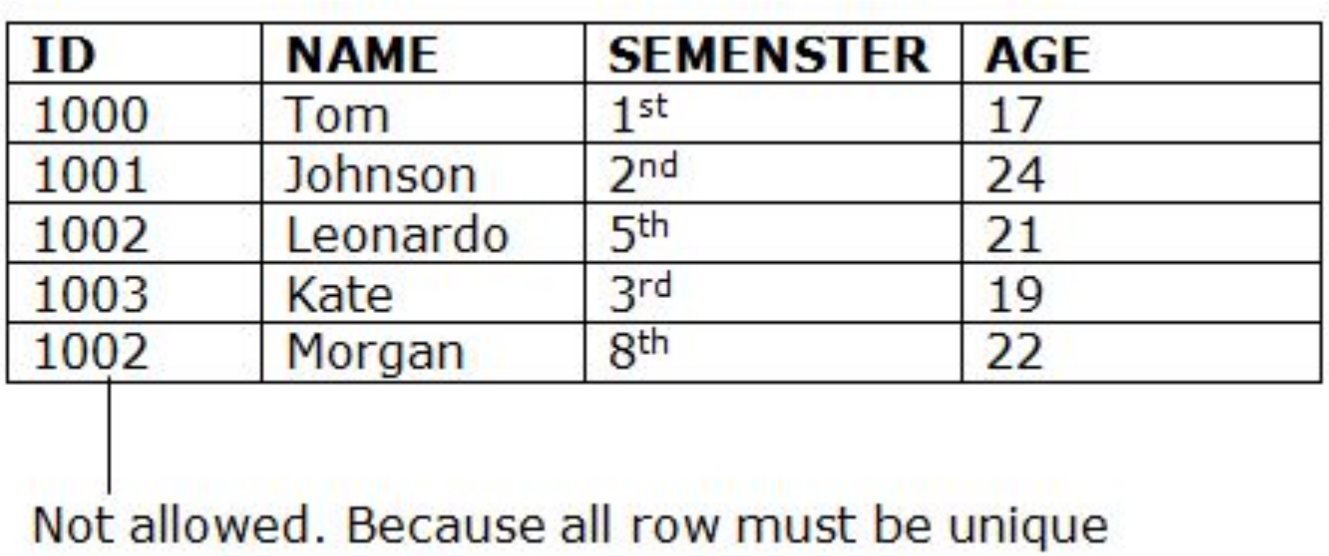
Primary Key

D_No	D_Location
11	Mumbai
24	Delhi
13	Noida

4. Key constraints

- Keys are the entity set that is used to identify an entity within its entity set uniquely.
- An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

Example:



ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1002	Morgan	8th	22

Not allowed. Because all row must be unique

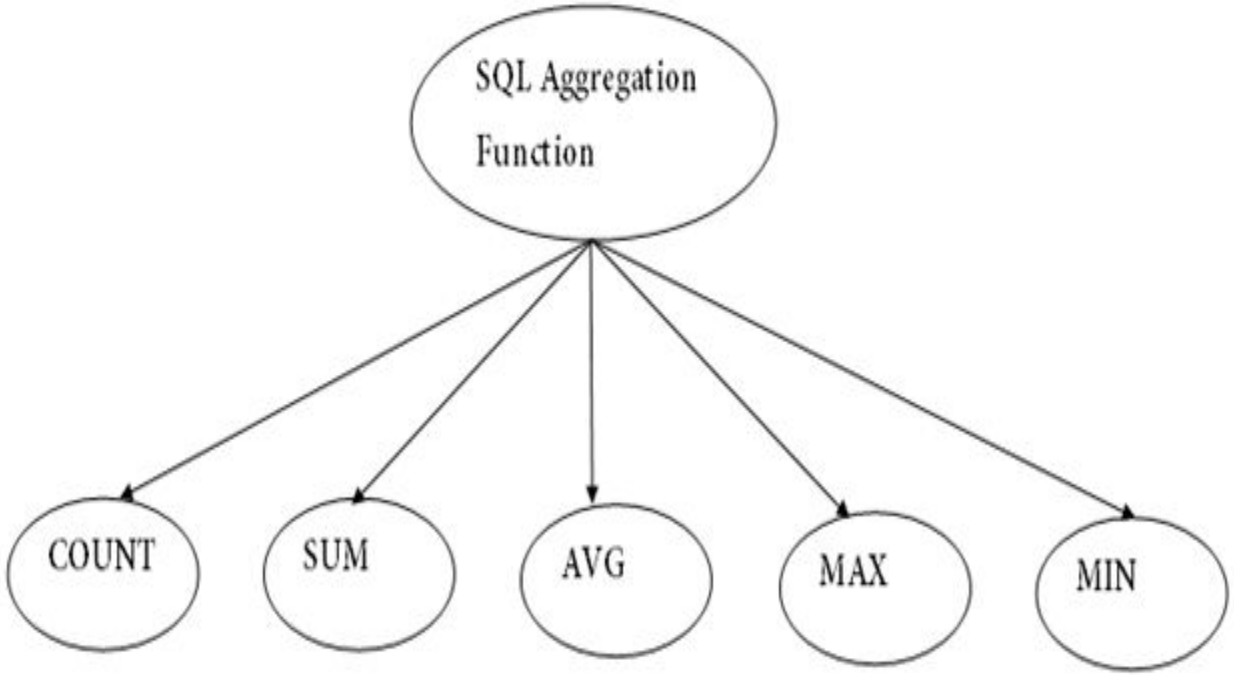
Aggregate functions

Aggregate functions in DBMS take multiple rows from the table and return a value according to the query.

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

ORDER BY clause

The SQL ORDER BY clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default.

Syntax

The basic syntax of the ORDER BY clause is as follows −

```
SELECT column-list

FROM table_name

[WHERE condition]

 [ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

Example

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 |
 +----+----------+-----+-----------+----------+
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 |
  +----+----------+-----+-----------+----------+
SQL> SELECT * FROM CUSTOMERS
   ORDER BY NAME DESC;
```

This would produce the following result −

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

GROUP BY clause

The SQL GROUP BY clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

```
SELECT column1, column2

FROM table_name

WHERE [ conditions ]

GROUP BY column1, column2

 ORDER BY column1
```

Example

Consider the CUSTOMERS table is 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 |
+----+----------+-----+-----------+----------+
```

If you want to know the total amount of the salary on each customer, then the GROUP BY query would be as follows.

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
   GROUP BY NAME;
```

This would produce the following result −

```
+----------+-------------+
| NAME     | SUM(SALARY) |
+----------+-------------+
| Chaitali |     6500.00 |
| Hardik   |     8500.00 |
| kaushik  |     2000.00 |
| Khilan   |     1500.00 |
| Komal    |     4500.00 |
| Muffy    |    10000.00 |
| Ramesh   |     2000.00 |
```

Example 2:

Student

SUBJECT	YEAR	NAME
C language	**2**	**John**
C language	**2**	Ginny
C language	**2**	Jasmeen
C language	**3**	Nick
C language	**3**	Amara
Java	**1**	Sifa
Java	**1**	dolly

- **Groups based on several columns:** A group of some columns are **GROUP BY column 1**, **column2**, **etc**. Here, we are placing all rows in a group with the similar values of both **column 1** and **column 2**.

Consider the below query:

```
1. SELECT SUBJECT, YEAR, Count (*)
2. FROM Student
3. Group BY SUBJECT, YEAR;
```

Output

SUBJECT	YEAR	Count
C language	2	3
C language	3	2
Java	1	2

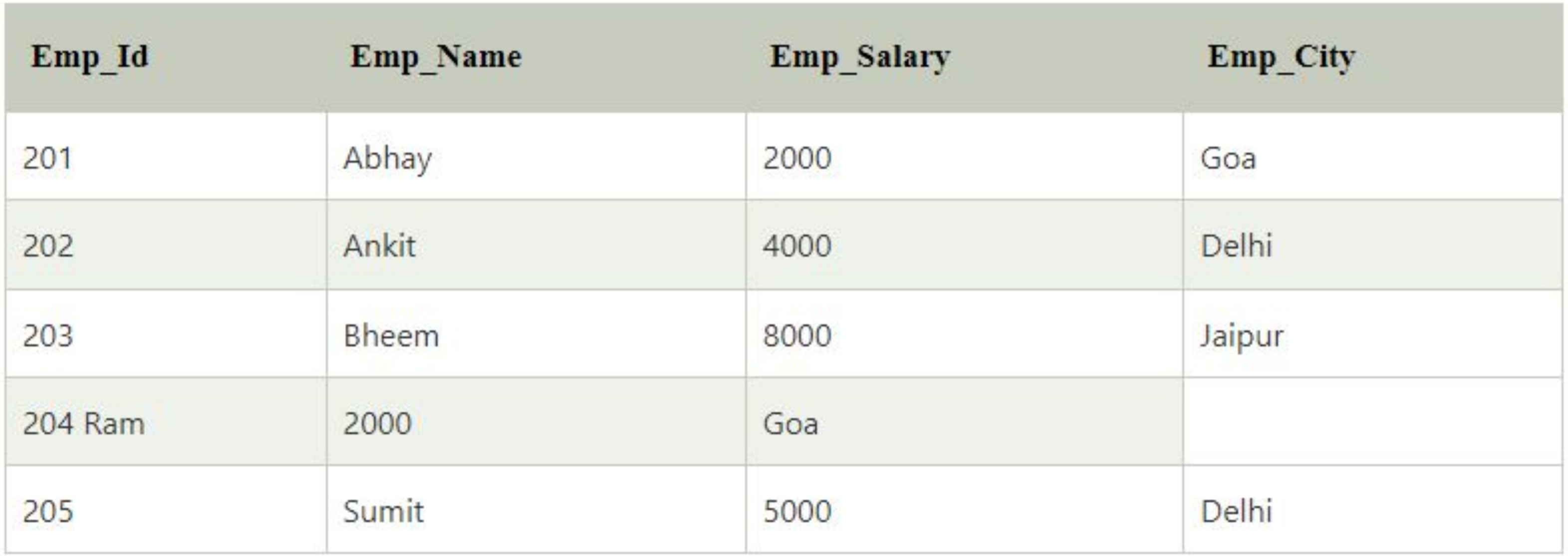
HAVING Clause

The HAVING Clause enables you to specify conditions that filter which group results appear in the results.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

```
SELECT column1, column2
FROM table1
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
 ORDER BY column1, column2
```

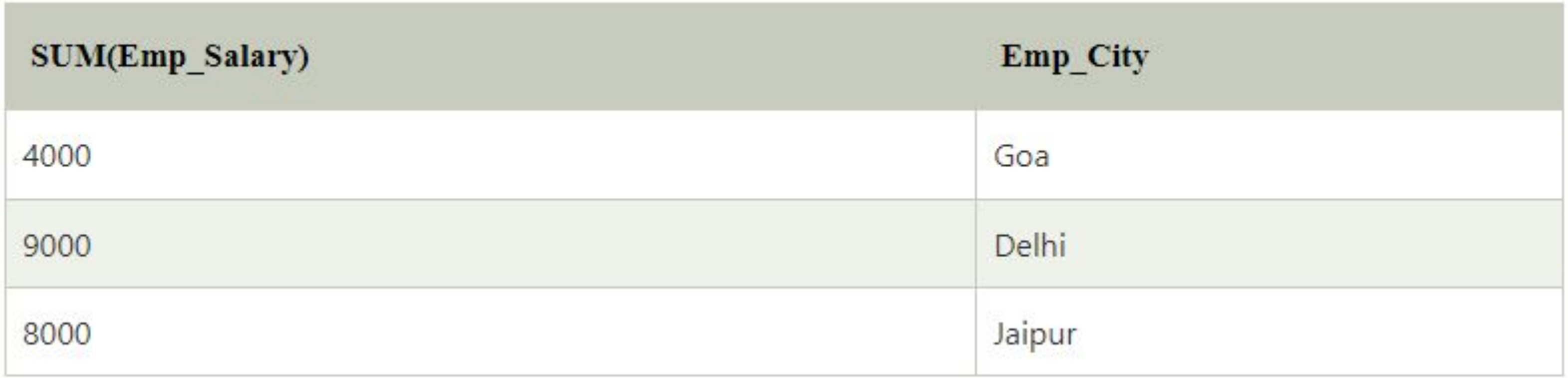
Employee table



Emp_Id	Emp_Name	Emp_Salary	Emp_City
201	Abhay	2000	Goa
202	Ankit	4000	Delhi
203	Bheem	8000	Jaipur
204 Ram	2000	Goa	
205	Sumit	5000	Delhi

If you want to add the salary of employees for each city, you have to write the following query:

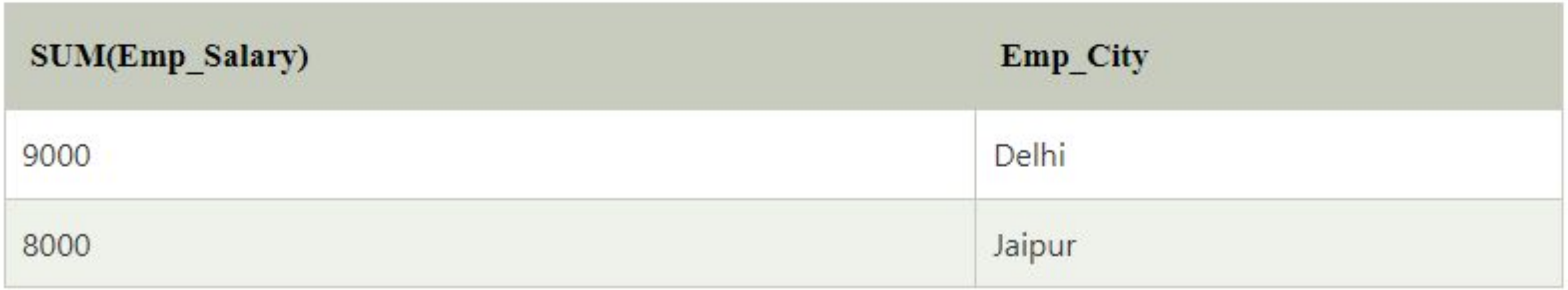
```
SELECT SUM(Emp_Salary), Emp_City FROM Employee GROUP BY
Emp_City;
```



SUM(Emp_Salary)	Emp_City
4000	Goa
9000	Delhi
8000	Jaipur

Now, suppose that you want to show those cities whose total salary of employees is more than 5000. For this case, you have to type the following query with the HAVING clause in SQL:

```
SELECT SUM(Emp_Salary), Emp_City FROM Employee GROUP BY Emp_City HAVING SUM(Emp_Salary)>5000;
```



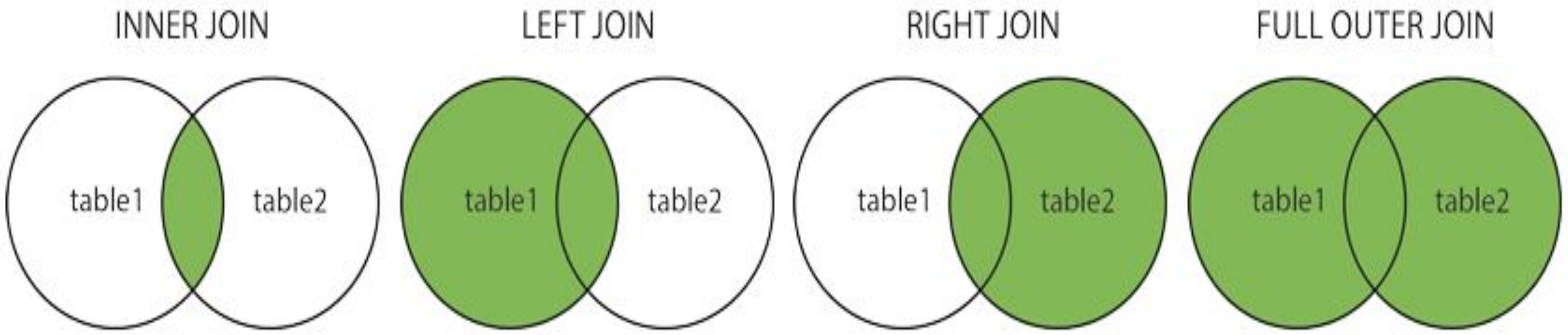
SUM(Emp_Salary)	Emp_City
9000	Delhi
8000	Jaipur

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.

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

FROM table1

INNER JOIN table2

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

EmpId	EmpName
5	John
2	David
3	Peter
4	Eric

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

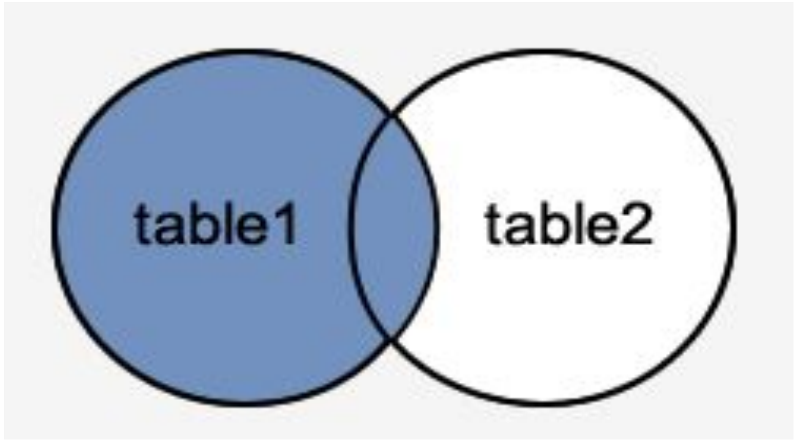
The syntax for the LEFT OUTER JOIN in SQL is:

```
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

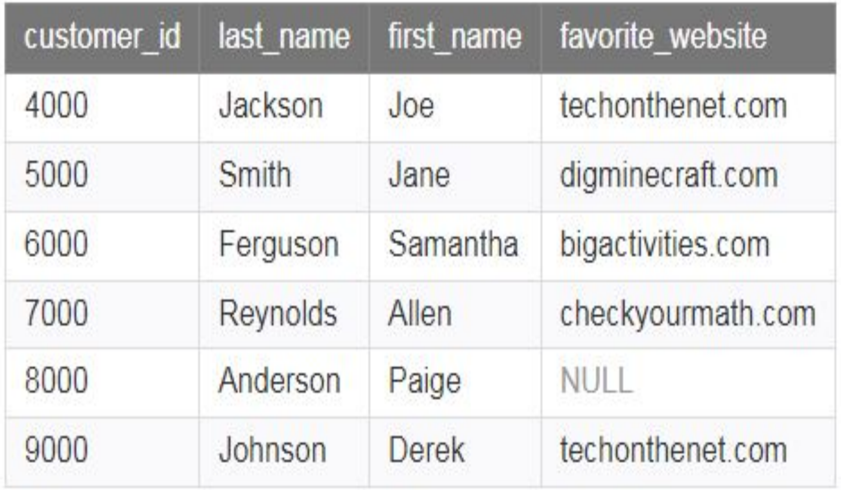
In this visual diagram, the SQL LEFT OUTER JOIN returns the shaded area:



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

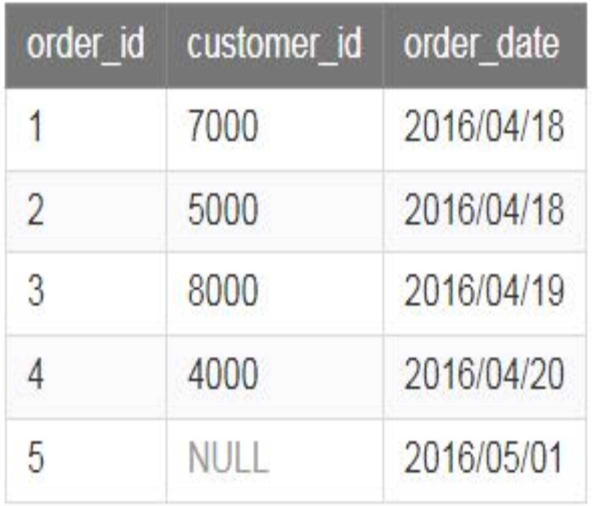
Example

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

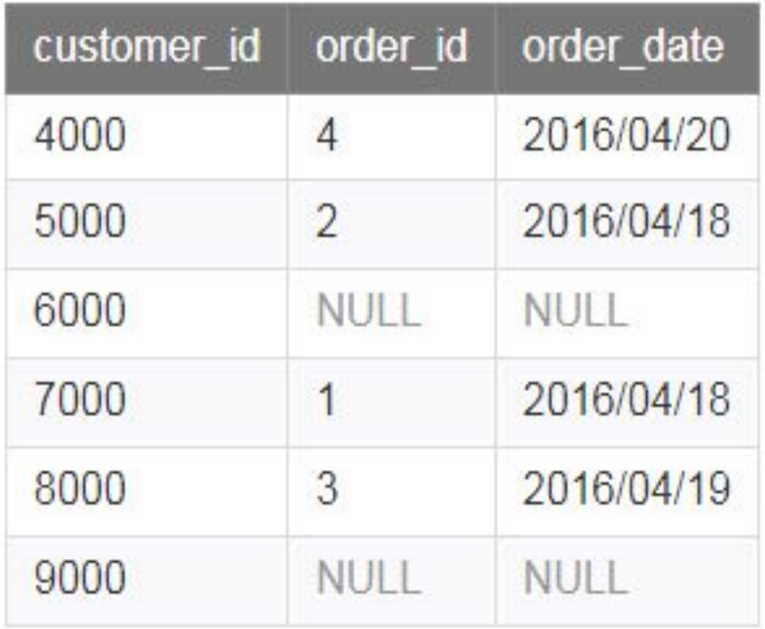
order table



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

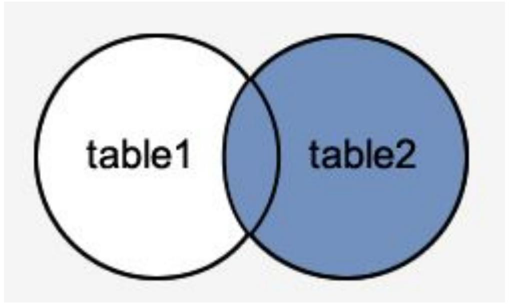
The syntax for the RIGHT OUTER JOIN in SQL is:

```
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:

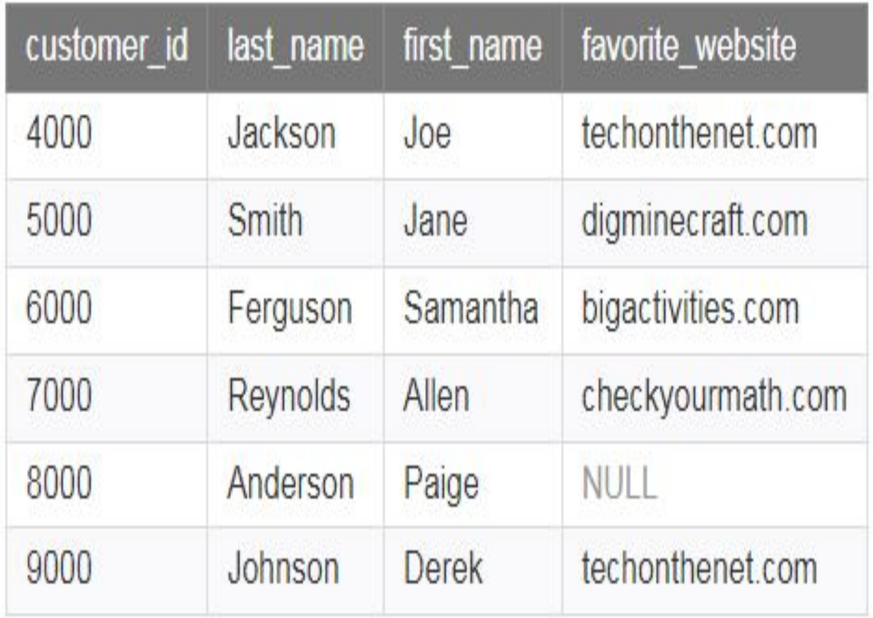
In this visual diagram, the SQL RIGHT OUTER JOIN returns the shaded area:



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

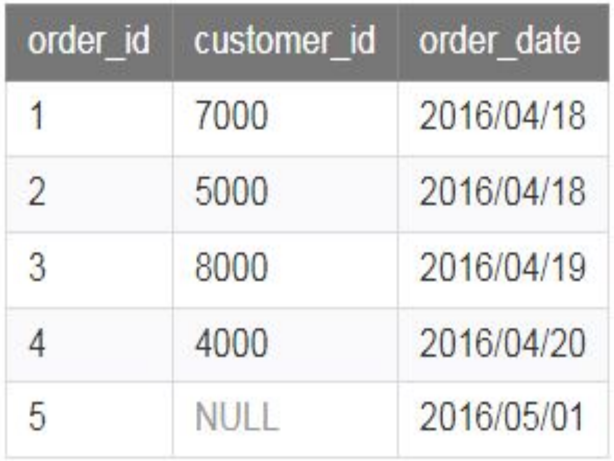
Example

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

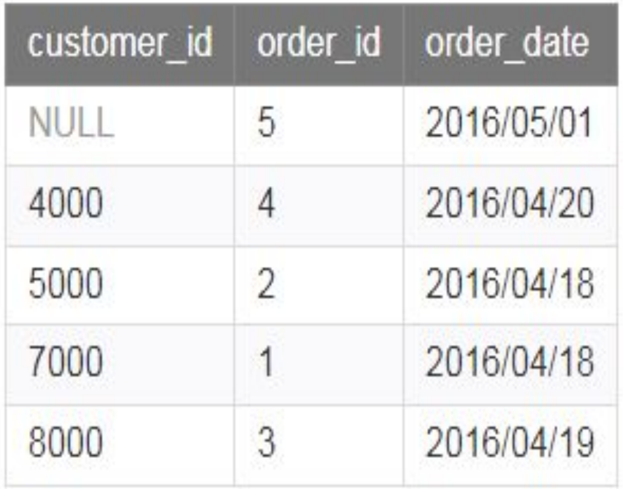
order



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

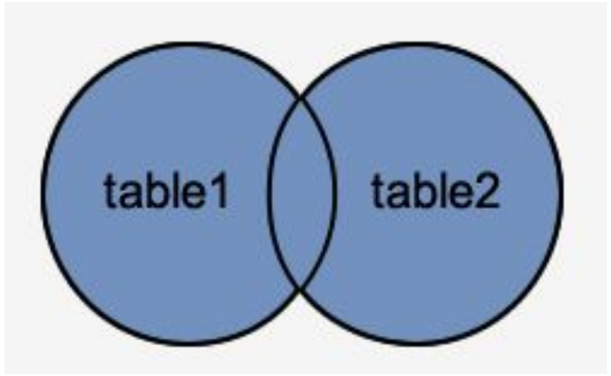
The syntax for the SQL **FULL OUTER JOIN** is:

```
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

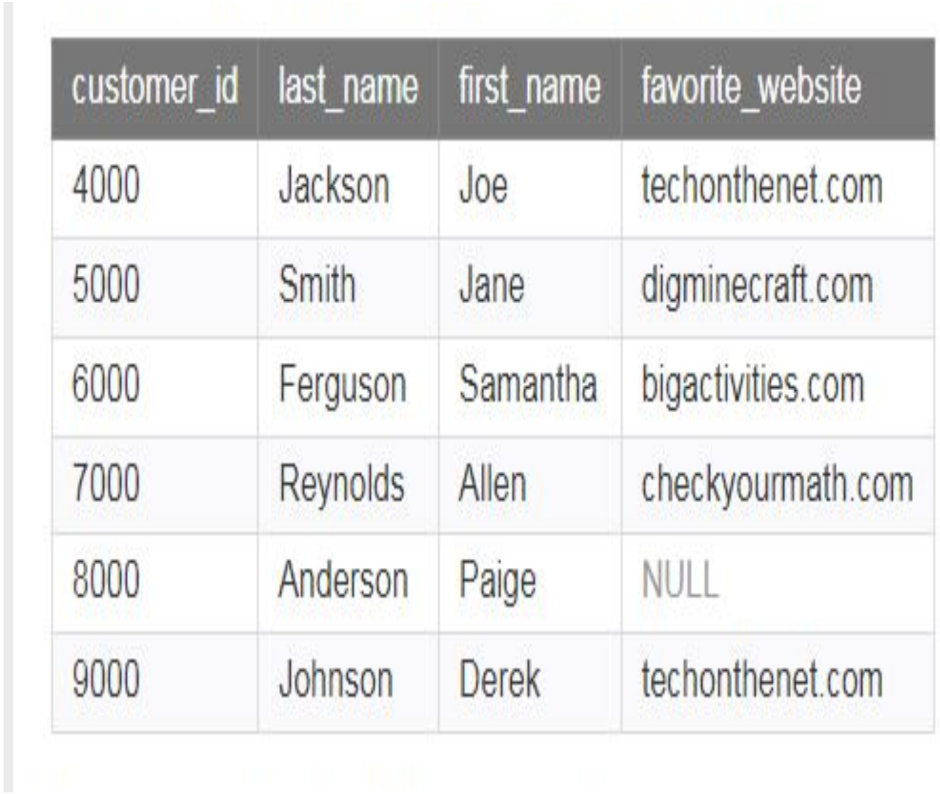
In this visual diagram, the SQL FULL OUTER JOIN returns the shaded area:



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

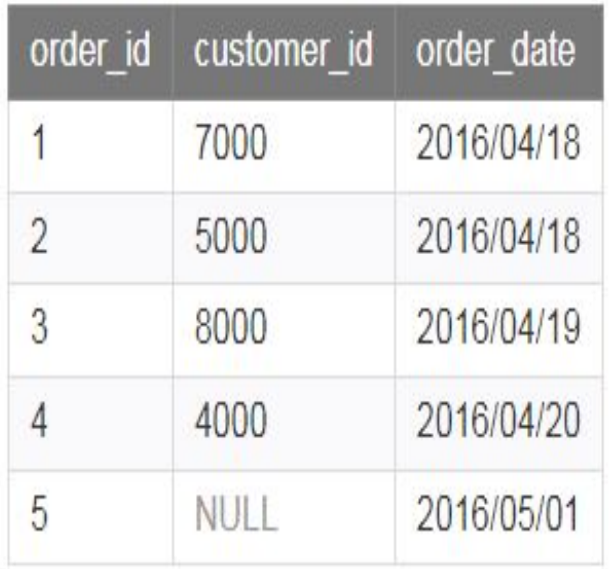
Example

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

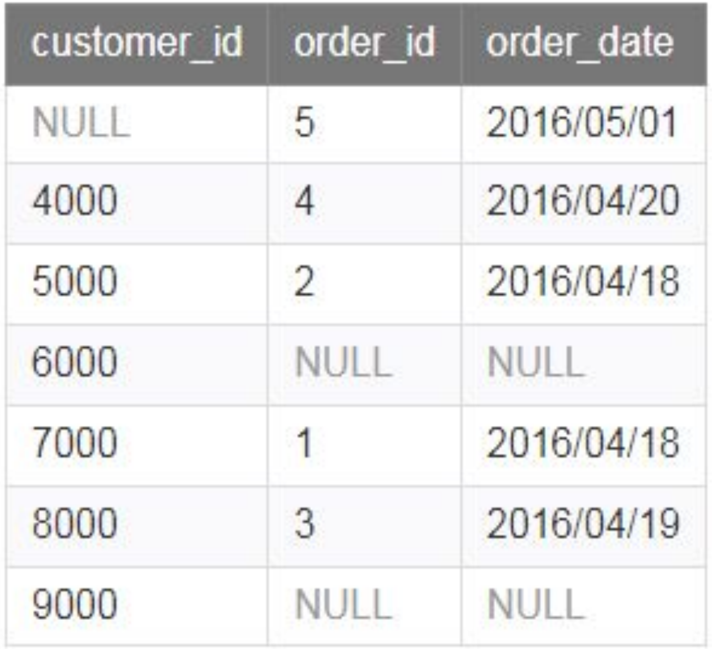
order



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.

Nested and complex queries

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

Rules that subqueries must follow:

1.The SQL Nested Query will be always enclosed inside the parentheses.

2.Nested sub-query can have only one column in select clause.

3.Order by clause is restricted in query which is inner query but outer query or main query can use order by clause.

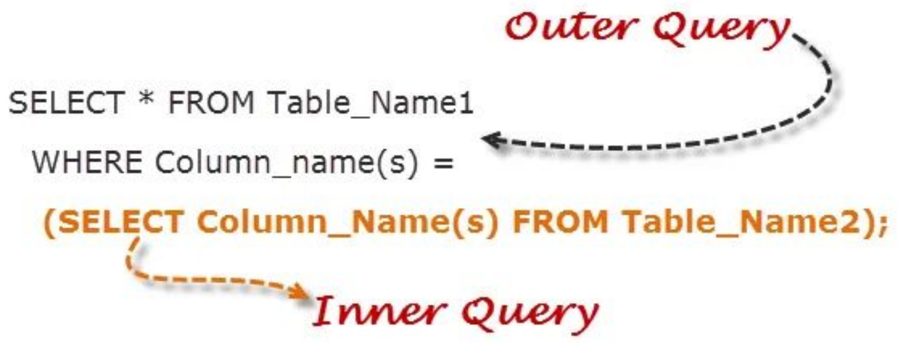
4.User needs to take care of multiple rows operator (IN,ANY) if sub-query will return more than one rows.

5.Between–And Operator can not be used inside the Nested Query.

Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows −

```
SELECT column_name FROM   table1
WHERE  column_name OPERATOR
   (SELECT column_name ]
   FROM table1 where…)
```



```
SELECT * FROM Table_Name1
 WHERE Column_name(s) =
  (SELECT Column_Name(s) FROM Table_Name2);
```

```
SELECT first_name, salary, department_id FROM
employees WHERE salary = (SELECT MIN (salary) FROM
employees);
```

here inner SQL returns only one row i.e. the minimum salary for the company. It in turn uses this value to compare salary of all the employees and displays only those, whose salary is equal to minimum salary.

- SELECT first_name, department_id FROM employees WHERE department_id IN (SELECT department_id FROM departments WHERE LOCATION_ID = 100)

- in the above query, IN matches department ids returned from the sub query,compares it with that in the main query and returns employee's name who satisfy the condition.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
SQL> SELECT *FROM CUSTOMERS WHERE ID IN (SELECT ID FROM CUSTOMERS
WHERE SALARY > 4500) ;
```

This would produce the following result.

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
 +----+----------+-----+---------+----------+
```

Subqueries with the INSERT Statement

- Subqueries also can be used with INSERT statements.
- The INSERT statement uses the data returned from the subquery to insert into another table.
- The selected data in the subquery can be modified with any of the character, date or number functions.

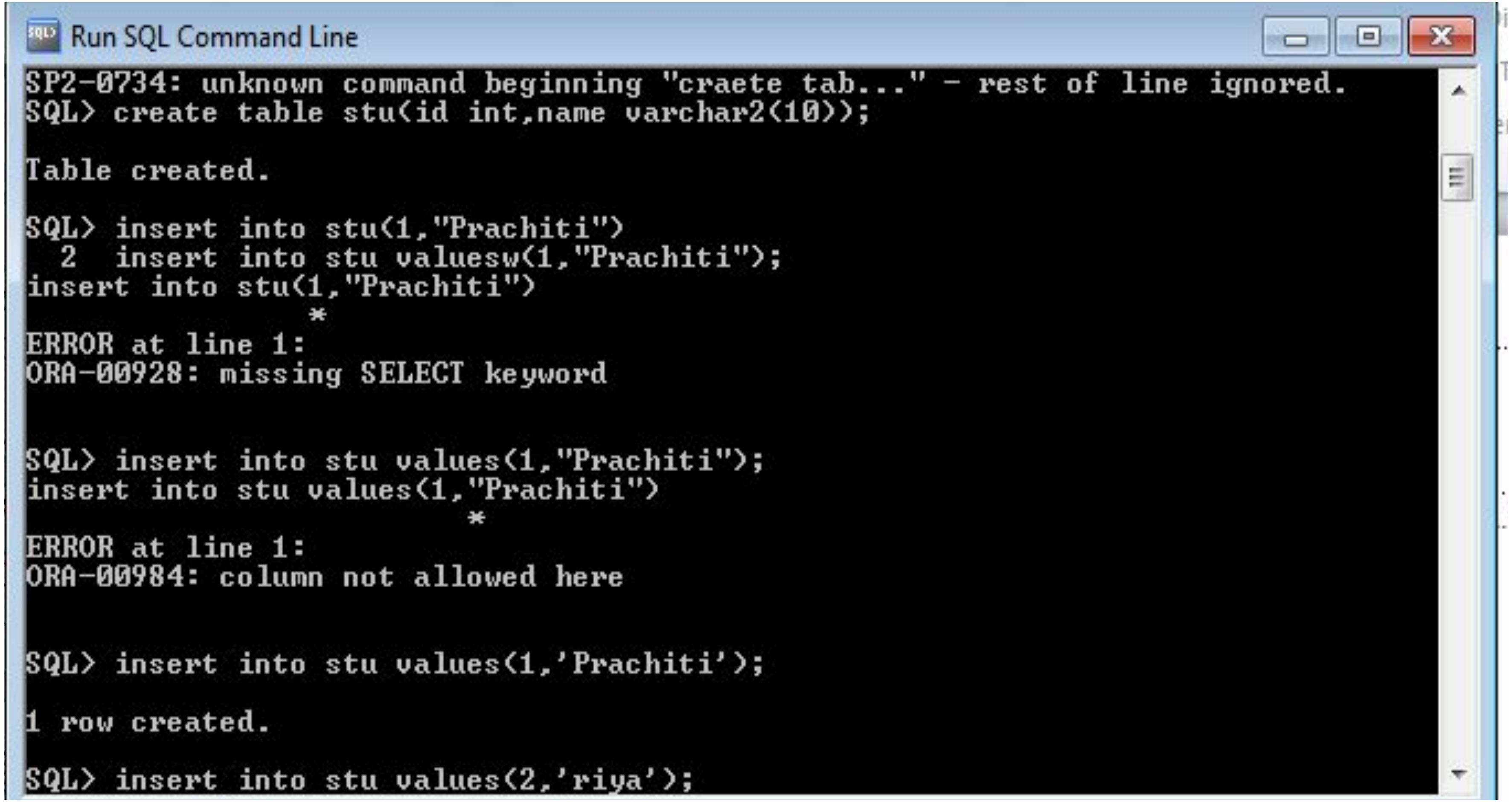
The basic syntax is as follows.

```
INSERT INTO table_name
   SELECT *   FROM table1
      [ WHERE VALUE OPERATOR ]
```

Example

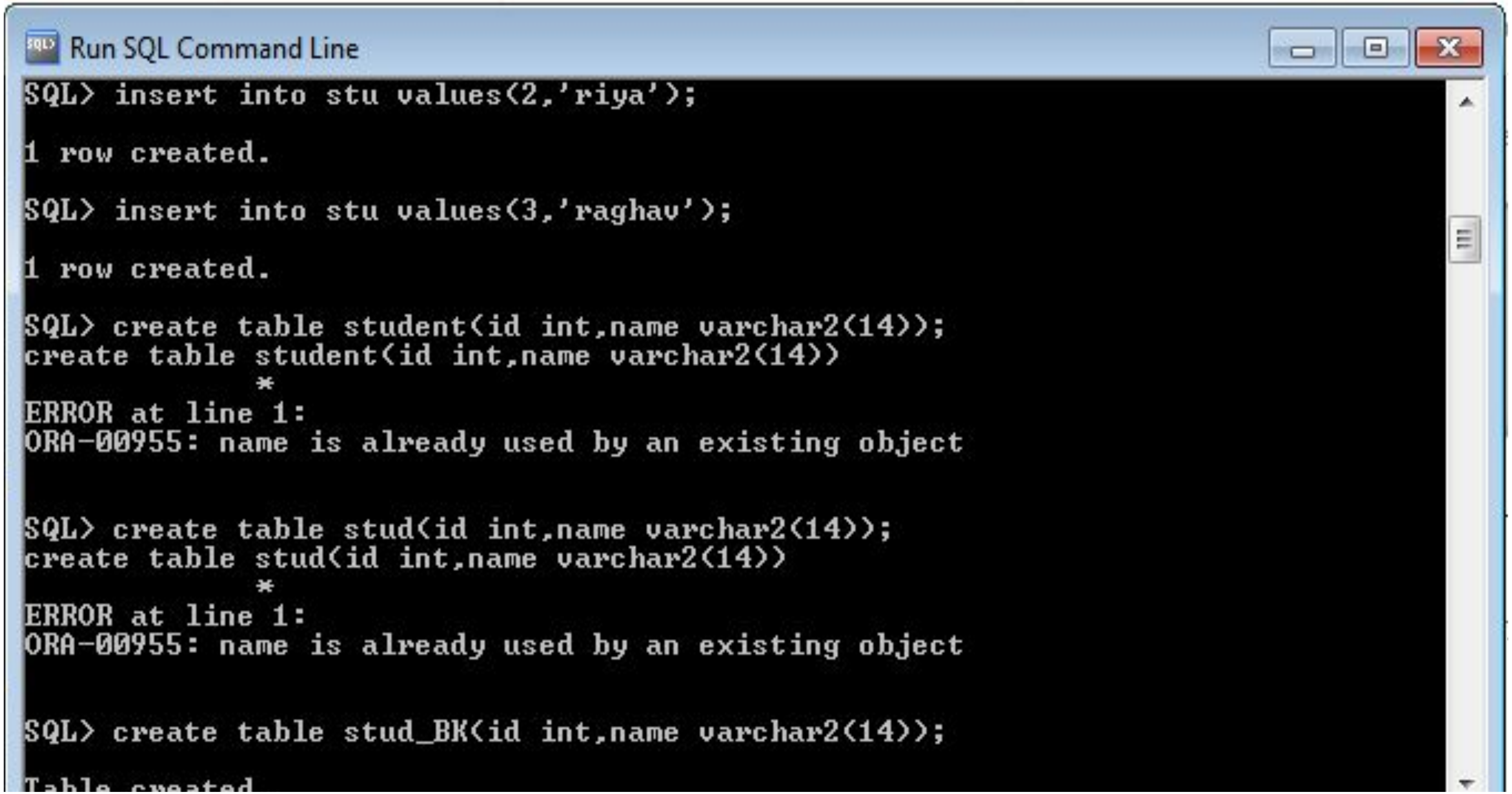
Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy the complete CUSTOMERS table into the CUSTOMERS_BKP table, you can use the following syntax.

```
SQL> INSERT INTO CUSTOMERS_BKP

   SELECT * FROM CUSTOMERS

   WHERE ID IN (SELECT ID
   FROM CUSTOMERS) ;
```



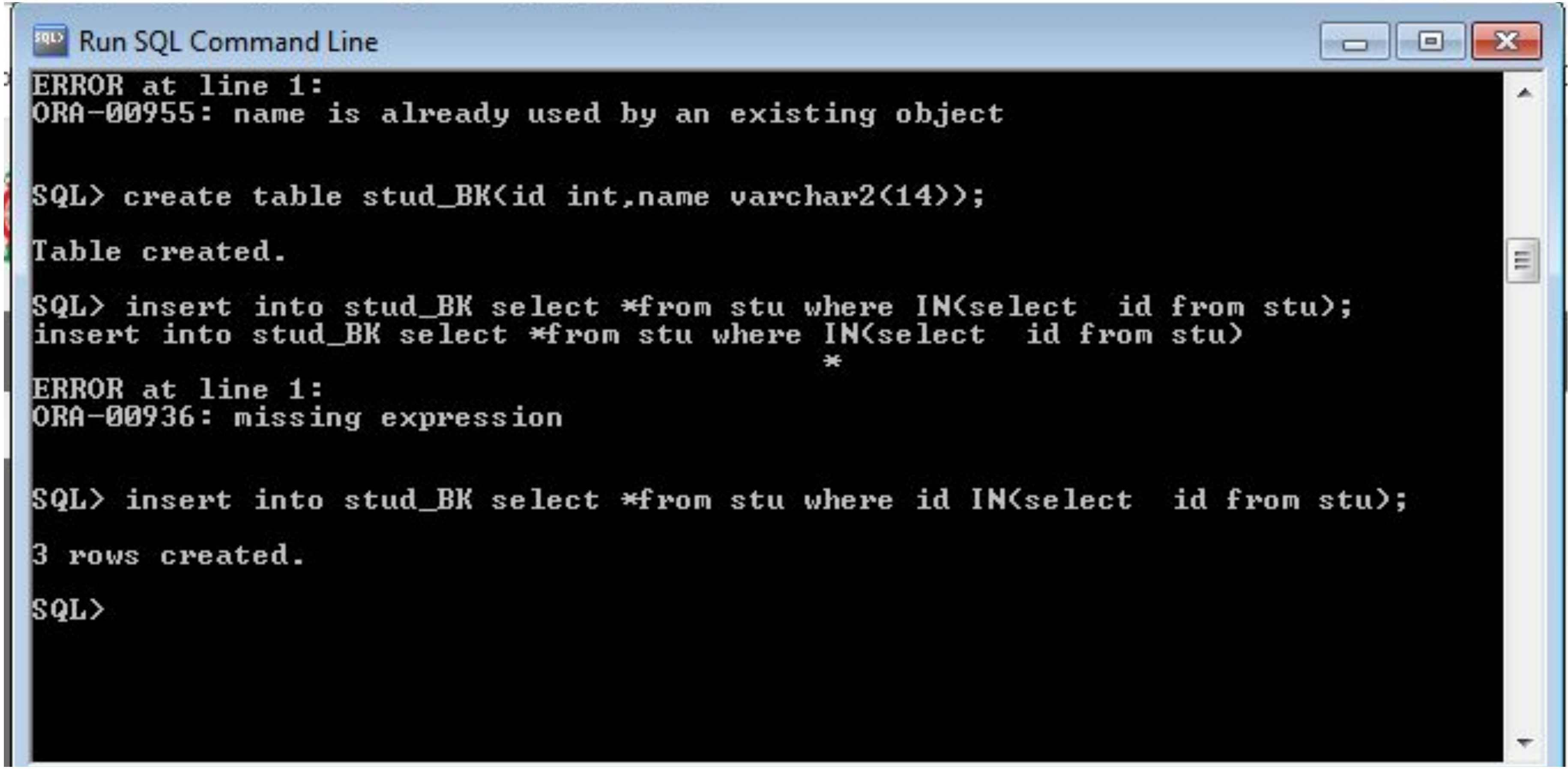
Run SQL Command Line

```
SP2-0734: unknown command beginning "craete tab..." - rest of line ignored.
SQL> create table stu(id int,name varchar2(10));

Table created.

SQL> insert into stu(1,"Prachiti")
  2  insert into stu valuesw(1,"Prachiti");
insert into stu(1,"Prachiti")
                *
ERROR at line 1:
ORA-00928: missing SELECT keyword


SQL> insert into stu values(1,"Prachiti");
insert into stu values(1,"Prachiti")
                         *
ERROR at line 1:
ORA-00984: column not allowed here


SQL> insert into stu values(1,'Prachiti');

1 row created.

SQL> insert into stu values(2,'riya');
```



Run SQL Command Line

```
SQL> insert into stu values(2,'riya');

1 row created.

SQL> insert into stu values(3,'raghav');

1 row created.

SQL> create table student(id int,name varchar2(14));
create table student(id int,name varchar2(14))
             *
ERROR at line 1:
ORA-00955: name is already used by an existing object


SQL> create table stud(id int,name varchar2(14));
create table stud(id int,name varchar2(14))
             *
ERROR at line 1:
ORA-00955: name is already used by an existing object


SQL> create table stud_BK(id int,name varchar2(14));

Table created.
```


Run SQL Command Line

```
ERROR at line 1:
ORA-00955: name is already used by an existing object


SQL> create table stud_BK(id int,name varchar2(14));

Table created.

SQL> insert into stud_BK select *from stu where IN(select  id from stu);
insert into stud_BK select *from stu where IN(select  id from stu)
                                           *
ERROR at line 1:
ORA-00936: missing expression


SQL> insert into stud_BK select *from stu where id IN(select  id from stu);

3 rows created.

SQL>
```

Subqueries with the UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows.

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
    [ WHERE) ]
```

Example

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table. The following example updates SALARY by 0.25 times in the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL> UPDATE CUSTOMERS
   SET SALARY = SALARY * 0.25
   WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
      WHERE AGE >= 27 );
```

Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement .

The basic syntax is as follows.

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
    [ WHERE) ]
```

Example

Assuming, we have a CUSTOMERS_BKP table available which is a backup of the CUSTOMERS table. The following example deletes the records from the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL> DELETE FROM CUSTOMERS
   WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
      WHERE AGE >= 27 );
```

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  2 | Khilan   |  25 | Delhi   |  1500.00 |
|  3 | kaushik  |  23 | Kota    |  2000.00 |
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  6 | Komal    |  22 | MP      |  4500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
 +----+----------+-----+---------+----------+
```

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.

Views, which are a type of virtual tables allow users to do the following −

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views

Database views are created using the CREATE VIEW statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic CREATE VIEW syntax is as follows −

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

Example

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 to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

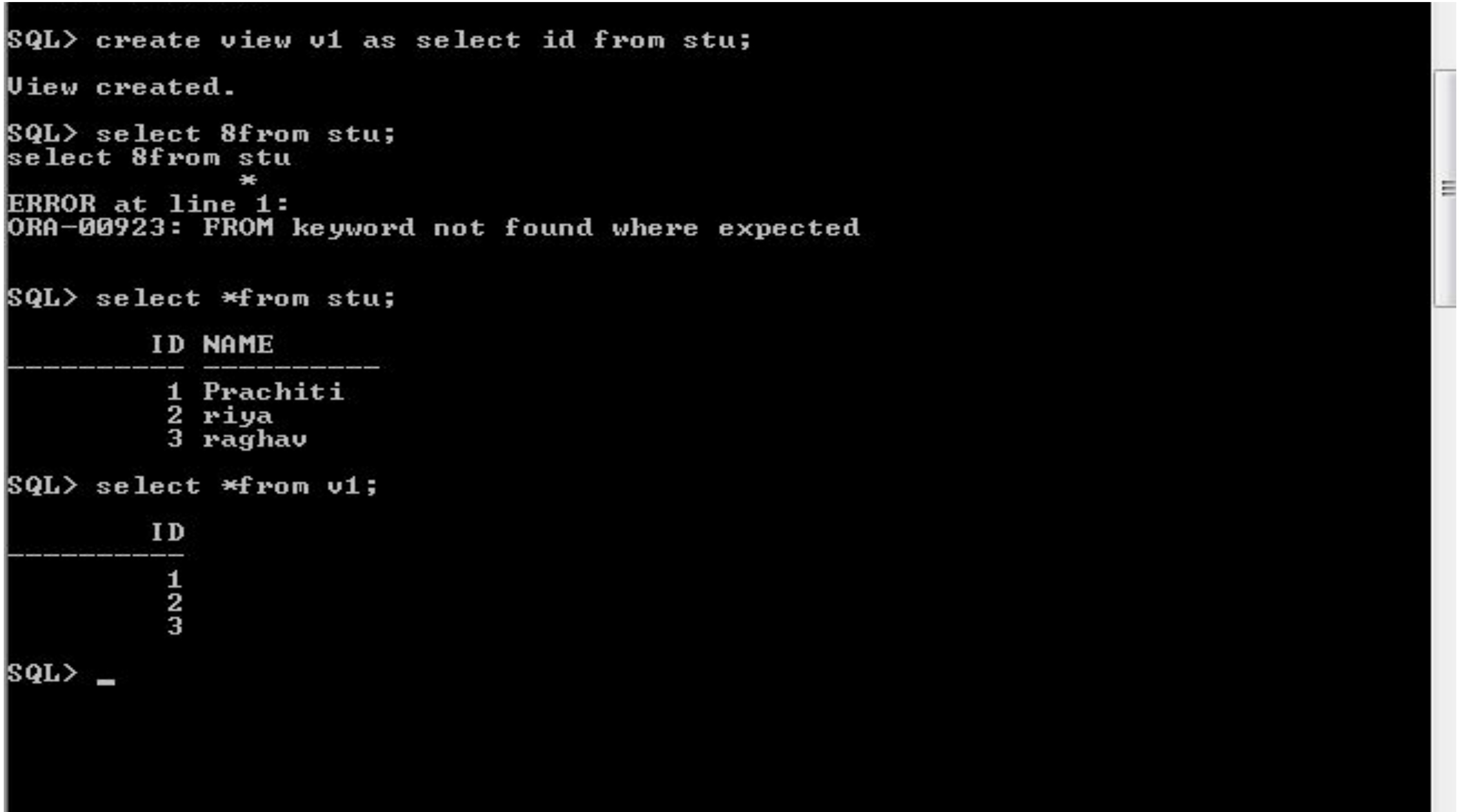
```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
 FROM  CUSTOMERS;
```

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;
```

This would produce the following result.

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



```
SQL> create view v1 as select id from stu;

View created.

SQL> select 8from stu;
select 8from stu
            *
ERROR at line 1:
ORA-00923: FROM keyword not found where expected


SQL> select *from stu;

        ID NAME
---------- ----------
         1 Prachiti
         2 riya
         3 raghav

SQL> select *from v1;

        ID
----------
         1
         2
         3

SQL> _
```

Creating View from multiple tables

View from multiple tables can be created by simply include multiple tables in the SELECT statement.

In the given example, a view is created named MarksView from two tables Student_Detail and Student_Marks.

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.

```
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;
```

Deleting Rows into a View

Rows of data can be deleted from a view.

example to delete a record having AGE = 22.

```
SQL > DELETE FROM
CUSTOMERS_VIEW
    WHERE age = 22;
```

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself.

```
+----+----------+-----+-----------+----------
+
| 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
|
|  7 | Muffy    |  24 | Indore    | 10000.00
|
 +----+----------+-----+-----------+-------
 ---+
```

Triggers

- Triggers are the SQL statements that are **automatically executed** when there is any change in the database. The triggers are executed **in response to certain events**(INSERT, UPDATE or DELETE) in a particular table. These triggers help in maintaining the integrity of the data by changing the data of the database in a systematic fashion.

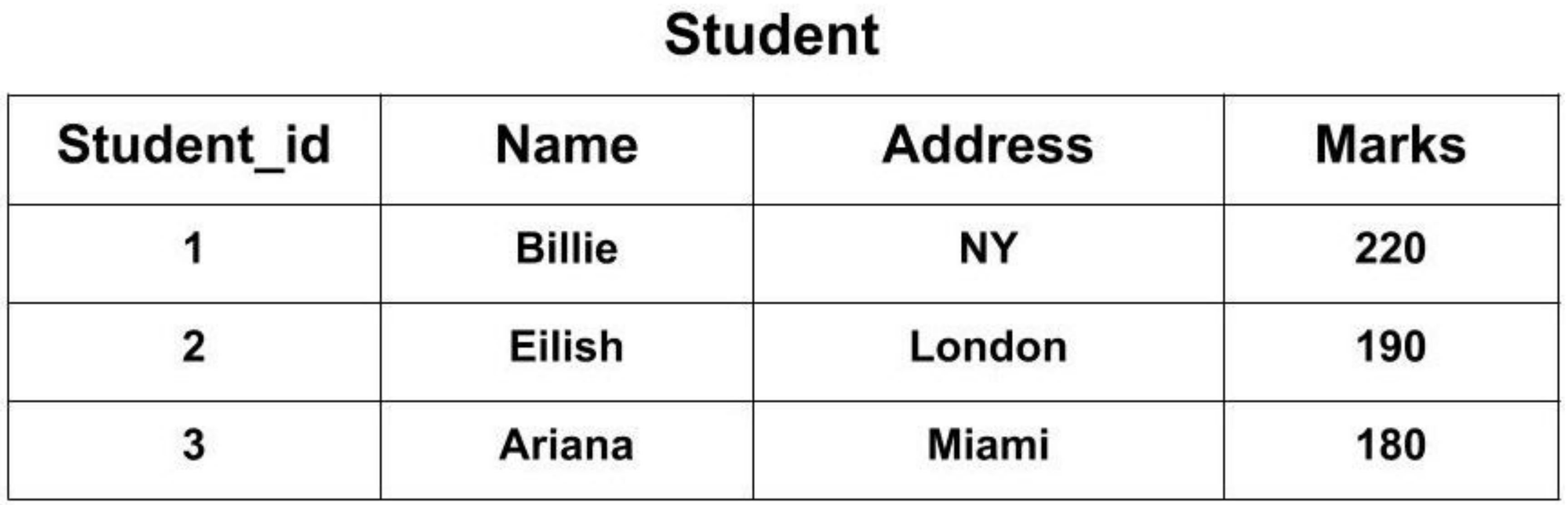
create trigger **Trigger_name**
(before | after)
[insert | update | delete]
on [table_name]
[**for** each row]
[trigger_body]

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.

- Triggers can be broadly classified into Row Level and Statement Level triggers.
- Broadly, these can be differentiated as:

Row Level Triggers	Statement Level Triggers
Row level triggers executes once for each and every row in the transaction.	Statement level triggers executes only once for each single transaction.
Example: If 1500 rows are to be inserted into a table, the row level trigger would execute 1500 times.	Example: If 1500 rows are to be inserted into a table, the statement level trigger would execute only once.

- **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

- Now, we want to create a **trigger** that will add 100 marks to each new row of the *Marks* column whenever a new student is inserted to the table.

- **The SQL Trigger will be:**

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

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

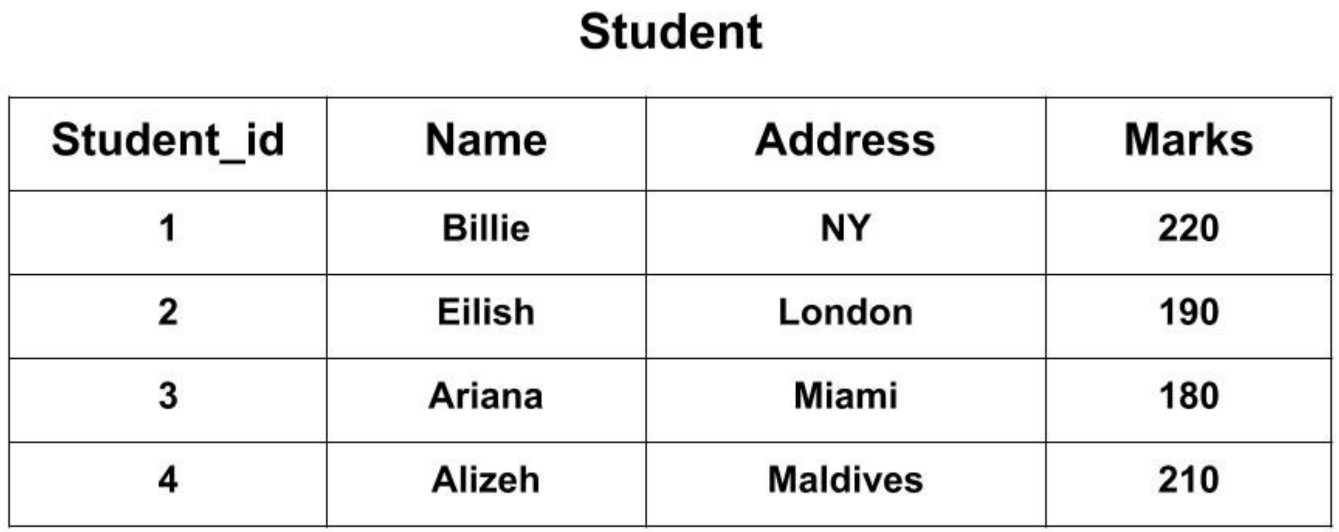
After creating the trigger, we will write the **query for inserting a new student** in the database.

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

*The **Student_id column** is an auto-increment field and will be generated automatically when a new record is inserted into the table.*

To see the final output the query would be:

```
SELECT * FROM Student;
```



Student

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

- **Advantages of Triggers**

Some of the prominent advantages of triggers are as follows:

1. Helps us to automate the data alterations.
2. Allows us to reuse the queries once written.
3. Provides a method to check the data integrity of the database.
4. Helps us to detect errors on the database level.
5. Allows easy auditing of data.

- **Disadvantages of Triggers**

Some of the disadvantages of triggers in SQL are as follows:
1. Increases the overhead costs of the server.
2. Provides only extended validations i.e. not all validations are accessible in SQL triggers.
3. Troubleshooting errors due to triggers is a tedious job.
4. Can cause logical errors in the application even if a slight mistake in query exists.
5. We could lose the original data if we set a wrong trigger by mistake.

SQL Set Operation

The SQL Set operation is used to combine the two or more SQL SELECT statements.

- Types of Set Operation

1. Union
2. UnionAll
3. Intersect
4. Minus

1. Union

- The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
- In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.
- The union operation eliminates the duplicate rows from its resultset.

Syntax

```
1.SELECT column_name FROM table1
2.UNION
3.SELECT column_name FROM table2;
```

Example:
The First table

ID	NAME
1	Jack
2	Harry
3	Jackson

The Second table

ID	NAME
3	Jackson
4	Stephan
5	David

Union SQL query will be:

```
SELECT * FROM First
UNION
SELECT * FROM Second;
```

ID	NAME
1	Jack
2	Harry
3	Jackson
4	Stephan
5	David

2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

- **Syntax:**

```
SELECT column_name FROM table1
UNION ALL
SELECT column_name FROM table2;
```

Example: Using the above First and Second table.

- Union All query will be like:

```
SELECT * FROM First
UNION ALL
SELECT * FROM Second;
```

ID	NAME
1	Jack
2	Harry
3	Jackson
3	Jackson
4	Stephan
5	David

3. Intersect

- It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
- In the Intersect operation, the number of datatype and columns must be the same.
- It has no duplicates and it arranges the data in ascending order by default.

- **Syntax**

```
SELECT column_name FROM table1
INTERSECT
SELECT column_name FROM table2;
```

Example:

Using the above First and Second table.

- Intersect query will be:

```
SELECT * FROM First
INTERSECT
SELECT * FROM Second;
```

ID	NAME
3	Jackson

4. Minus

- It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
- It has no duplicates and data arranged in ascending order by default.

- **Syntax:**

```
SELECT column_name FROM table1
MINUS
SELECT column_name FROM table2;
```

- **Example**

Using the above First and Second table.

- Minus query will be:

```
SELECT * FROM First
MINUS
SELECT * FROM Second;
```

ID	NAME
1	Jack
2	Harry

Set and string operations

- The string is a collection of characters put to use for storing multiple characters. Most of the data present around us is a string like our name, our address, etc.
- We often require modification and access to the strings around us. To make this process easy, SQL has some built-in string functions.

- The built-in functions which take an input string and return an output string are called string functions.

- 1. SQL CHAR_LENGTH()

Helps to calculate the length of a given string.(not containing spaces).

Syntax:

```
Select char_length(col_name) as length_name from tableName;
```

Example: Let us use the CHAR_LENGTH function on the name_emp column.

Query:

```
SELECT CHAR_LENGTH(name) as length_name from dataflair;
```

- 2. ASCII in SQL

Returns the ASCII value of the expression.

Syntax:

```
Select ASCII(col_name) as ascii_name from tableName;
```

Example:

```
select Ascii('a’) from dual;
```

3) Concat

Concatenates two character strings.

```
SELECT  Concat ('abc', 'def') FROM
employee;
```

4)Lower:

Converts a character string to lowercase.

Syntax: Lower(Customer_Name)

5) Upper:

Converts a character string to uppercase.

Syntax: Upper(Customer_Name)

6) Replace

Replaces one or more characters from a specified character expression with one or more other characters.

```
Syntax: Replace('abcd1234', '123', 'zz')
```

7) SUBSTRING

SUBSTRING functions allows you to extract a substring from a string.

```
SUBSTRING( string, start_position, length )
```

- `SELECT SUBSTR('w3resource',3,4) "Substring" FROM DUAL;`

```
output:   reso
```