Unit - II

SQL: Queries and Constraints

A.PAVITHRA
ASSISTANT PROFESSOR
DEPARTMENT OF IT
GRIET,HYDERABAD

UNIT-II

- **SQL: Queries and Constraints:** Form of Basic SQL Query, SQL Operators, Set Operators, Nested Queries, Aggregate Operators, NULL values, Integrity Constraints Over Relations, Joins, Introduction to View, Destroying / Altering Tables and Views, Cursors, Triggers and Active Databases.

CONSTRAINTS

- SQL constraints are used to specify rules for the data in a table.
- Constraints are used to limit the type of data that can go into a table.
- This ensures the accuracy and reliability of the data in the table.
- If there is any violation between the constraint and the data action, the action is aborted.
- Constraints can be column level or table level.
- Column level constraints apply to a column, and table level constraints apply to the whole table.

SQL CONSTRAINTS

1. **NOT NULL -** Ensures that a column cannot have a NULL value
2. **UNIQUE -** Ensures that all values in a column are different
3. **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
4. **FOREIGN KEY -** Prevents actions that would destroy links between tables
5. **CHECK -** Ensures that the values in a column satisfies a specific condition
6. **DEFAULT -** Sets a default value for a column if no value is specified
7. **CREATE INDEX -** Used to create and retrieve data from the database very quickly

1.NOT NULL CONSTRAINT

- By default, a column can hold NULL values.
- The NOT NULL constraint enforces a column to NOT accept NULL values.
- This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

- **Example:**
- **SQL NOT NULL on CREATE TABLE:**

```
CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255) NOT NULL, Age int);
```

- **SQL NOT NULL on ALTER TABLE**

```
ALTER TABLE Persons MODIFY Age int NOT NULL;
```

2.UNIQUE CONSTRAINT

- The UNIQUE constraint ensures that all values in a column are different.
- Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint.

- **Example:**
- **SQL UNIQUE Constraint on CREATE TABLE**
- CREATE TABLE Persons (ID int NOT NULL UNIQUE, LastName varchar(255) NOT NULL, FirstName varchar(255), Age int);

(Or)

- CREATE TABLE Persons (ID int NOT NULL,LastName varchar(255) NOT NULL, FirstName varchar(255),Age int,UNIQUE (ID));

(Or)

- CREATE TABLE Persons (ID int NOT NULL,LastName varchar(255) NOT NULL, FirstName varchar(255),Age int,CONSTRAINT UC_Person UNIQUE (ID, LastName));

- **SQL UNIQUE Constraint on ALTER TABLE**
- ALTER TABLE Persons ADD UNIQUE (ID);

  (or)
- ALTER TABLE Persons ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);

- **DROP a UNIQUE Contraint**
- ALTER TABLE Persons DROP INDEX UC_Person;

  (or)
- ALTER TABLE Persons DROP CONSTRAINT UC_Person;

3.PRIMARY KEY CONSTRAINT

- The PRIMARY KEY constraint uniquely identifies each record in a table.
- Primary keys must contain UNIQUE values, and cannot contain NULL values.
- A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

- **Example:**
- **SQL PRIMARY KEY on CREATE TABLE**
- CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL,FirstName varchar(255),Age int, PRIMARY KEY (ID));

  (OR)
- CREATE TABLE Persons (ID int NOT NULL PRIMARY KEY, LastName varchar(255) NOT NULL,FirstName varchar(255),Age int);

  (OR)
- CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL,FirstName varchar(255),Age int, CONSTRAINT PK_Person PRIMARY KEY (ID,LastName));

• SQL PRIMARY KEY on ALTER TABLE

- ALTER TABLE Persons ADD PRIMARY KEY (ID);

  (OR)

- ALTER TABLE Persons ADD CONSTRAINT PK_Person PRIMARY KEY (ID,LastName);

• DROP a PRIMARY KEY Constraint

- ALTER TABLE Persons DROP PRIMARY KEY;

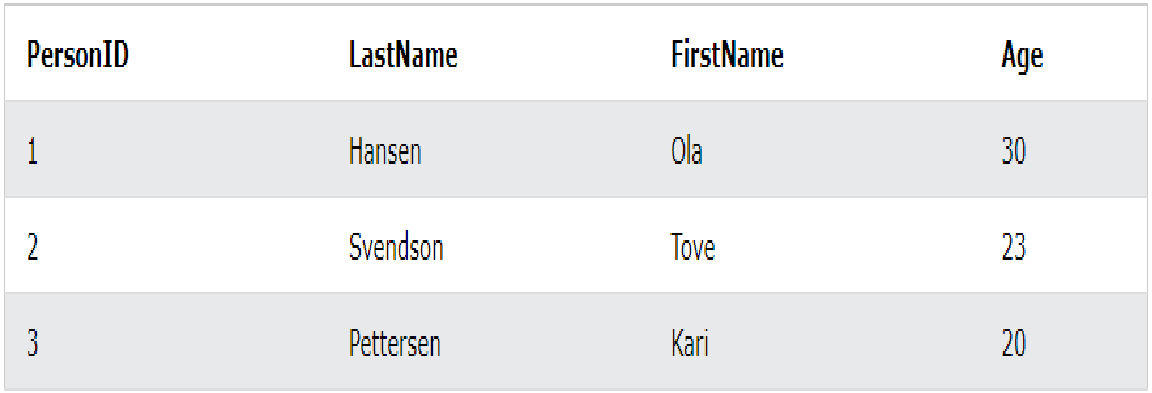
  (OR)

- ALTER TABLE Persons DROP CONSTRAINT PK_Person;

4.FOREIGN KEY CONSTRAINT

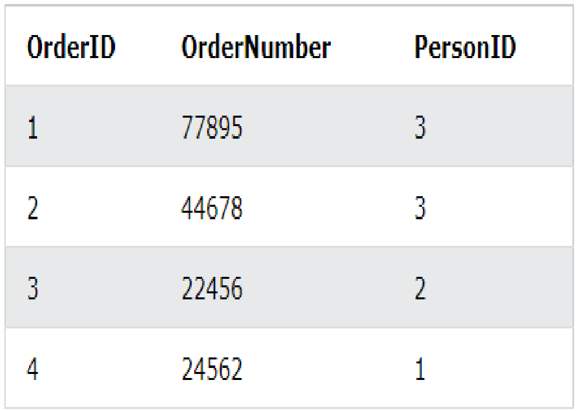
- The **FOREIGN KEY** constraint is used to prevent actions that would destroy links between tables.
- A **FOREIGN KEY** is a field (or collection of fields) in one table, that refers to the **PRIMARY KEY** in another table.
- The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

Person Table



PersonID	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

Orders Table



OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3
3	22456	2
4	24562	1

- **Example:**
- CREATE TABLE Orders (OrderID int NOT NULL, OrderNumber int NOT NULL, PersonID int, PRIMARY KEY (OrderID), FOREIGN KEY (PersonID) REFERENCES Persons(PersonID));

- **SQL FOREIGN KEY on ALTER TABLE**
- ALTER TABLE Orders ADD FOREIGN KEY(PersonID) REFERENCES Persons(PersonID);

- **DROP a FOREIGN KEY Constraint**
- ALTER TABLE Orders DROP FOREIGN KEY FK_PersonOrder;

5.CHECK CONSTRAINT

- The CHECK constraint is used to limit the value range that can be placed in a column.
- If you define a CHECK constraint on a column it will allow only certain values for this column.
- If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.
- **Example:**
- **SQL CHECK on CREATE TABLE**
- CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255), Age int, CHECK (Age>=18));

(OR)

- CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255), Age int CHECK (Age>=18));

- CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL, FirstName varchar(255), Age int, City varchar(255), CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sydney'));

- **SQL CHECK on ALTER TABLE**
- ALTER TABLE Persons ADD CHECK (Age>=18);

  (or)
- ALTER TABLE Persons ADD CONSTRAINT CHK_PersonAge CHECK (Age>=18 AND City='Sydney');

- **DROP a CHECK Constraint**

  ALTER TABLE Persons DROP CHECK CHK_PersonAge;

  (OR)

  ALTER TABLE Persons DROP CONSTRAINT CHK_PersonAge;

6.DEFAULT CONSTRAINT

- The DEFAULT constraint is used to set a default value for a column.
- The default value will be added to all new records, if no other value is specified.
- **SQL DEFAULT on CREATE TABLE**
- CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL,FirstName varchar(255),Age int, City varchar(255) DEFAULT 'Sydney');

  (OR)
- CREATE TABLE Orders (ID int NOT NULL, OrderNumber int NOT NULL, OrderDate date DEFAULT GETDATE());

- *The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE():*

- **SQL DEFAULT on ALTER TABLE**
- ALTER TABLE Persons MODIFY City DEFAULT 'Sandnes';

- **DROP a DEFAULT Constraint**
- ALTER TABLE Persons ALTER City DROP DEFAULT;

  (OR)
- ALTER TABLE Persons ALTER COLUMN City DROP DEFAULT;

7.CREATE INDEX CONSTRAINT

- The CREATE INDEX statement is used to create indexes in tables.
- Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

- **CREATE INDEX Syntax**
- **Syntax:** CREATE INDEX index_name ON table_name (column1, column2, ...);

- **CREATE UNIQUE INDEX Syntax**
- **Syntax:** CREATE UNIQUE INDEX index_name ON table_name (column1, column2, ...);

- **Example:**
- CREATE INDEX idx_lastname ON Persons (LastName);
- CREATE INDEX idx_pname ON Persons (LastName, FirstName);

- **DROP INDEX Statement**
- **Syntax:**
- DROP INDEX index_name ON table_name;

  (OR)
- DROP INDEX table_name.index_name;

  (OR)
- DROP INDEX index_name;

- **Example:**
- ALTER TABLE table_name DROP INDEX index_name;

SQL Aggregate Functions

- SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
- It is also used to summarize the data.
- We mainly use these functions with the GROUP BY and HAVING clauses of the SELECT statements in the database query languages.

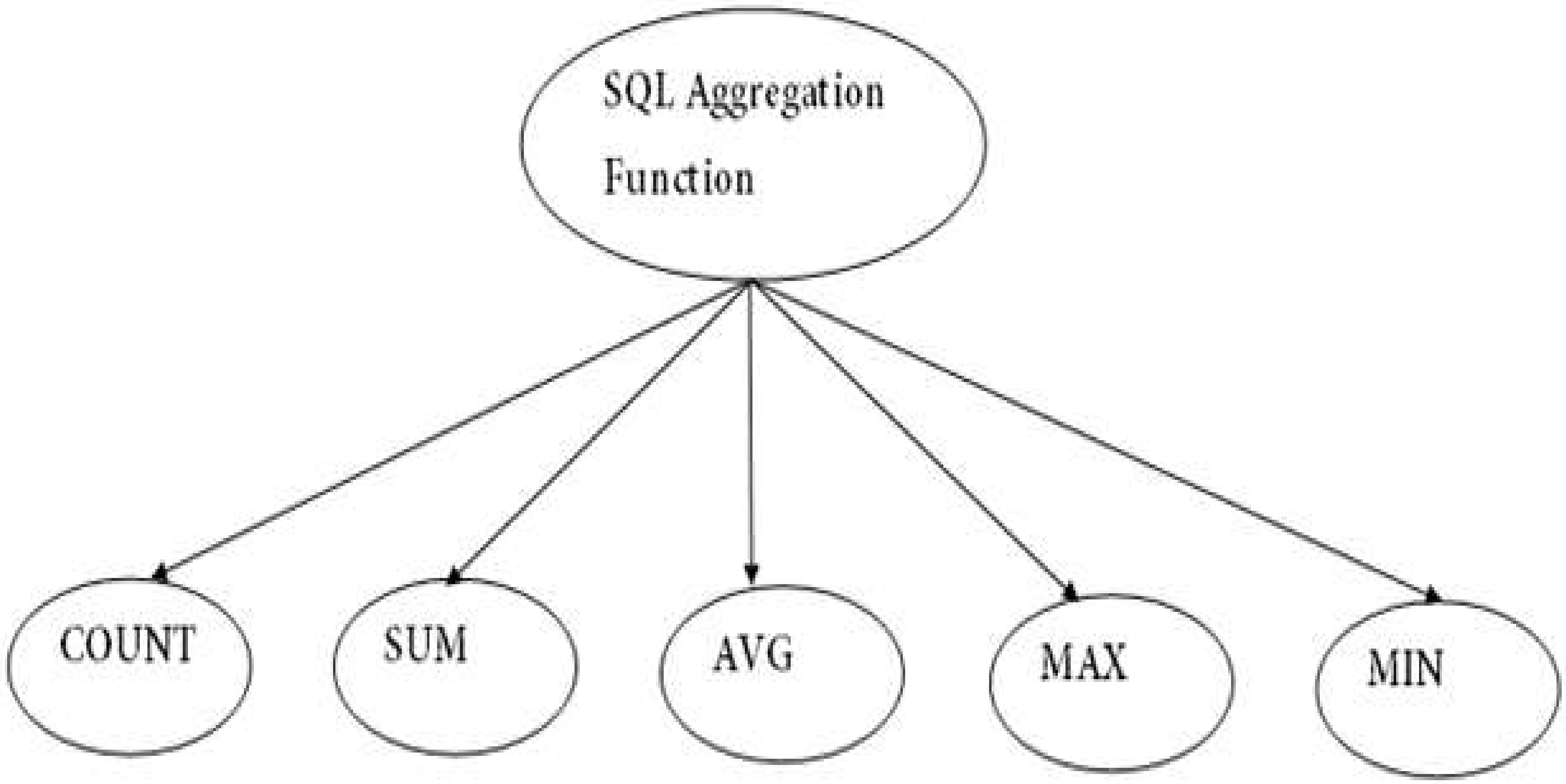
  Syntax:
- **aggregate_function_name(DISTINCT | ALL exp)**

- In this syntax, we can see the following parameters:
- **aggregate_function_name:** It indicates the name of the aggregate function that we want to use.
- **DISTINCT | ALL:** The DISTINCT modifier is used when we want to consider the distinct values in the calculation. The ALL modifiers are used when we want to calculate all values, including duplicates. If we do specify any modifier, all aggregate functions use the ALL modifier by default.
- **exp:** It indicates the table's columns or an expression containing multiple columns with arithmetic operators.

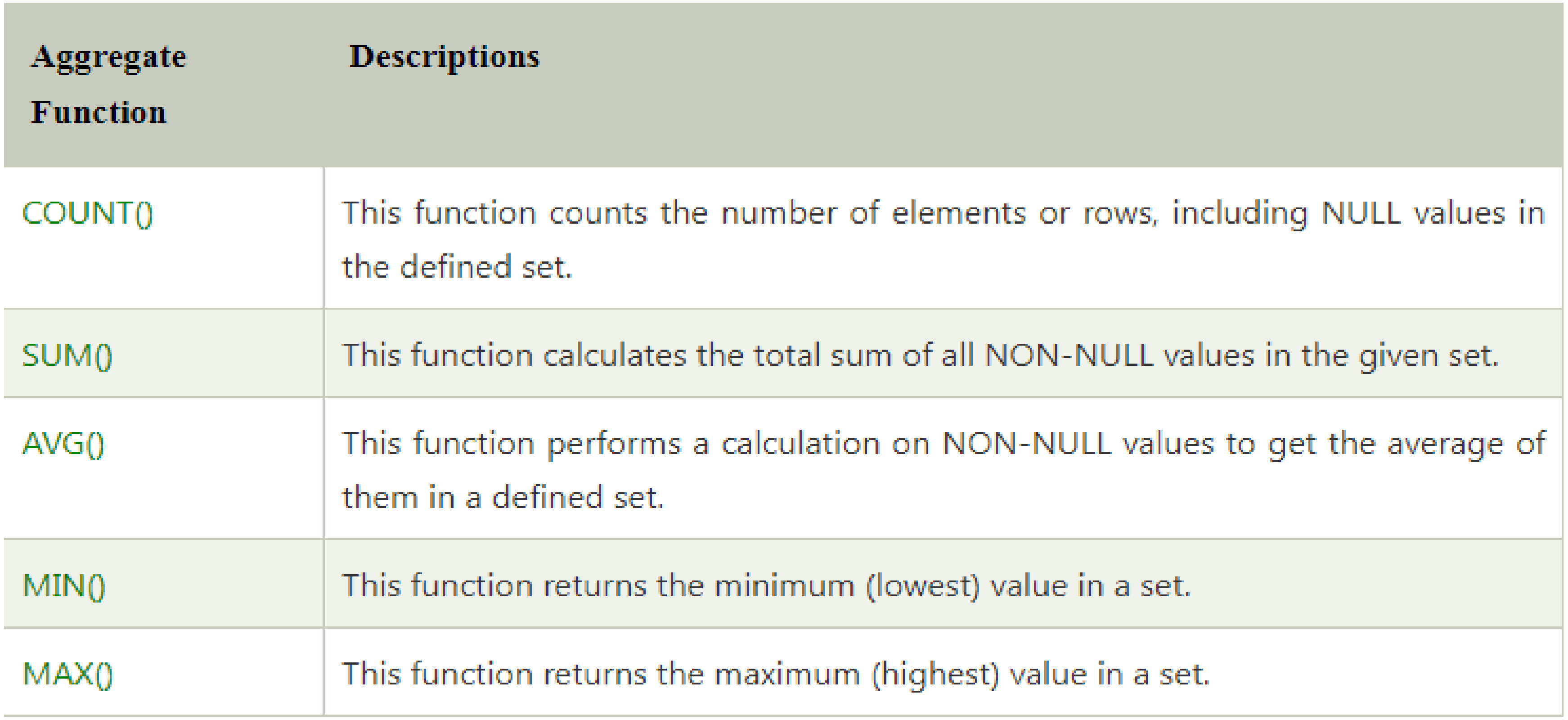
Why we use aggregate functions?

- The aggregate functions are mainly used to produce the summarized data in economics and finance to represent the economic health or stock and sector performance.
- In the context of business, different organization levels need different information, such as top levels managers interested in knowing whole figures and not the individual details.

SQL Aggregation Function



SQL Aggregation Function



Aggregate Function	**Descriptions**
COUNT()	This function counts the number of elements or rows, including NULL values in the defined set.
SUM()	This function calculates the total sum of all NON-NULL values in the given set.
AVG()	This function performs a calculation on NON-NULL values to get the average of them in a defined set.
MIN()	This function returns the minimum (lowest) value in a set.
MAX()	This function returns the maximum (highest) value in a set.

ORDER BY CLAUSE

- Whenever we want to sort the records based on the columns stored in the tables of the SQL database, then we consider using the ORDER BY clause in SQL.
- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.

Syntax:

SELECT Column Name 1,..........,Column Name N
FROM Table Name
ORDER BY Column Name **ASC/DESC**;

Example: select * from emp **order by** *sal* **desc;**

GROUP BY CLAUSE

- In SQL, The **Group By** statement is used for organizing similar data into groups.
- **Syntax:**

```
SELECT column name, function(column_name)
FROM table_name
WHERE condition
GROUP BY column_name;
Note: function_name: Table name.
Condition: which we used.
```

- **Example:**

```
SELECT D_state, avg(D_salary) AS  salary
FROM developers
GROUP BY D_state
ORDER BY D_state DESC;
```

WHERE CLAUSE

- The WHERE clause in MySQL is used with SELECT, INSERT, UPDATE, and DELETE queries to filter data from the table or relation.
- The **WHERE CLAUSE** places conditions on the selected columns.
- **Syntax:**

  SELECT column_lists,

  FROM table_name

  WHERE conditions

  GROUP BY column_lists;

- **Example:**

SELECT * **FROM** employees **WHERE** working_hour > 9;

HAVING CLAUSE

- HAVING clause in MySQL **used in conjunction with GROUP BY** clause enables us to specify conditions that filter which group results appear in the result.
- This clause places conditions on groups created by the GROUP BY clause.
- It behaves like the WHERE clause when the SQL statement does not use the GROUP BY keyword.
- We can use the aggregate (group) functions such as SUM, MIN, MAX, AVG, and COUNT only with two clauses: SELECT and HAVING.
- **Syntax:**

```
SELECT column_lists,
aggregate_function (expression)
FROM table_name
WHERE conditions
GROUP BY column_lists
HAVING condition;
```

Example:

```
SELECT name, SUM(working_hour) AS "Total working hours"
FROM employees
GROUP BY name
HAVING SUM(working_hour) > 6;
```

1.COUNT FUNCTION

- COUNT function is used to Count the number of rows in a database table.
- It can work on both numeric and non-numeric data types.
- COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table.
- COUNT(*) considers duplicate and Null.
- **Syntax**
- COUNT(*)

  or
- COUNT([ALL|DISTINCT] expression)

Example Table: PRODUCT

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40
Item6	Cpm1	3	25	75
Item7	Com1	5	30	150
Item8	Com1	3	10	30
Item9	Com2	2	25	50
Item10	Com3	4	30	120

- **Example: COUNT()**

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

- **Example: COUNT with WHERE**

```
SELECT COUNT(*)FROM PRODUCT;
WHERE RATE>=20;
```

- **Example: COUNT() with DISTINCT**

```
SELECT COUNT(DISTINCT COMPANY) FROM PRODUCT;
```

- **Example: COUNT() with GROUP BY**

```
SELECT COMPANY, COUNT(*)  FROM PRODUCT
GROUP BY COMPANY;
```

- **Example: COUNT() with HAVING**

```
SELECT COMPANY, COUNT(*)  FROM PRODUCT
GROUP BY COMPANY
HAVING COUNT(*)>2;
```

2.SUM FUNCTION

- Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.
- **Syntax:**
- SUM()

  or
- SUM([ALL|DISTINCT] expression)
- **Example: SUM()**

  SELECT SUM(COST) FROM PRODUCT;

- **Example: SUM() with WHERE**

```
SELECT SUM(COST)  FROM PRODUCT_MAST
WHERE QTY>3;
```

- **Example: SUM() with GROUP BY**

```
SELECT SUM(COST)  FROM PRODUCT_MAST
WHERE QTY>3
GROUP BY COMPANY;
```

- **Example: SUM() with HAVING**

```
SELECT COMPANY, SUM(COST)  FROM PRODUCT
GROUP BY COMPANY
HAVING SUM(COST)>=170;
```

3.AVG FUNCTION

- The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.
- **Syntax**
- AVG()

  or
- AVG([ALL|DISTINCT] expression)
- **Example:**

```
SELECT AVG(COST)  FROM PRODUCT;
```

4.MAX FUNCTION

- MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.
- **Syntax**
- MAX()

  or
- MAX([ALL|DISTINCT] expression)
- **Example:**

  SELECT MAX(RATE) FROM PRODUCT;

5.MIN FUNCTION

- MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.
- **Syntax**
- MIN()

  or
- MIN([ALL|DISTINCT] expression)
- **Example:**

  SELECT MIN(RATE) FROM PRODUCT;

Joins

- **Join in DBMS** is a binary operation which allows you to combine **join** product and selection in one single statement.
- The goal of creating a **join** condition is that it helps you to combine the data from two or more **DBMS** tables.
- The tables in **DBMS** are associated using the primary key and foreign keys.

Joins Types

- **INNER JOIN**
- **NATURAL JOIN**
- **LEFT OUTER JOIN**
- **RIGHT OUTER JOIN**
- **FULL OUTER JOIN**
- **SELF JOIN**

Inner Join

A **Inner Join** is a join operation that joins two tables by their common column with duplicate attributes. It returns rows when there is a match in both tables

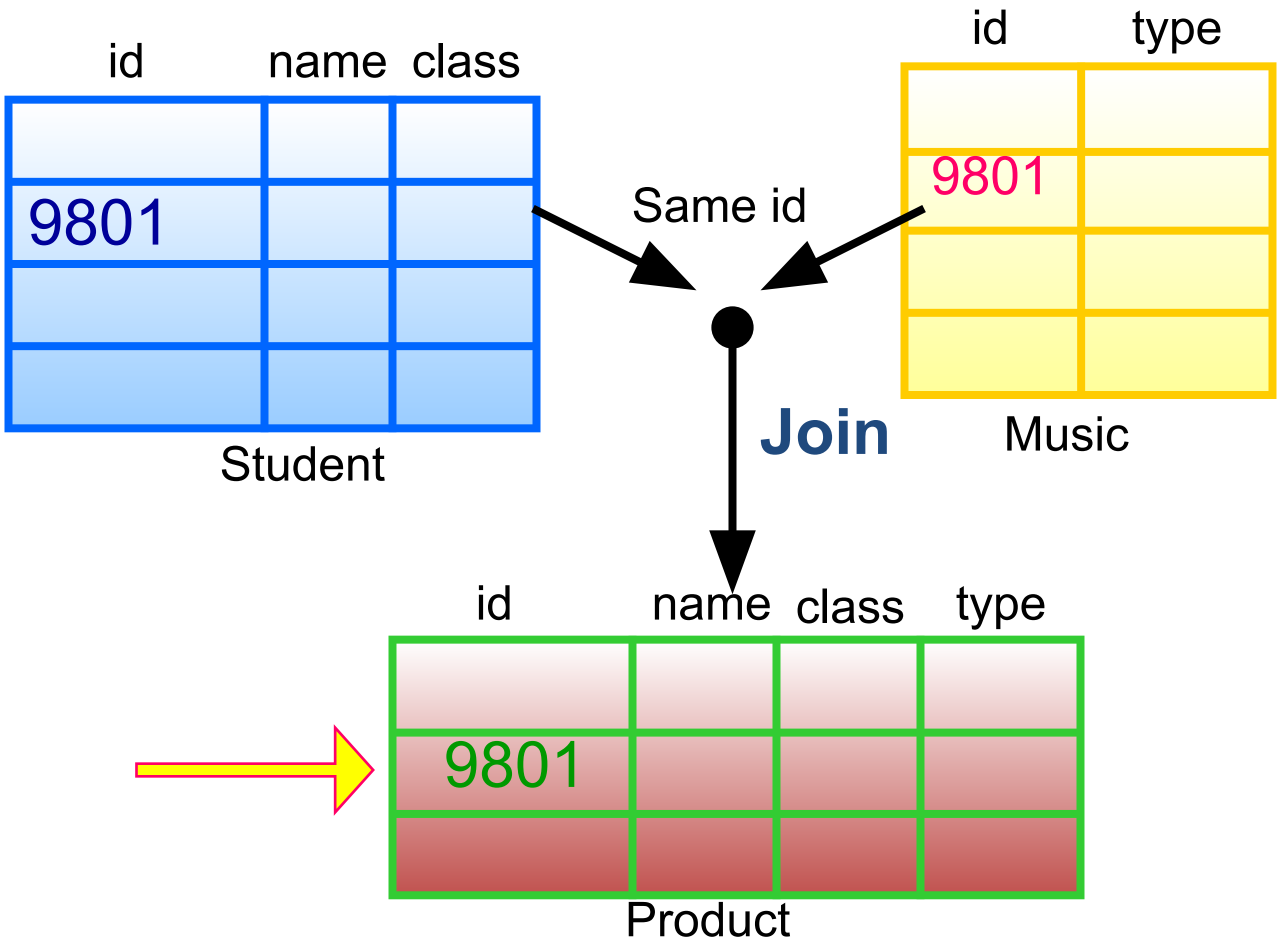
```
Select * from emp e, dept d where e.deptno=d.deptno;
(Or)
Select * from emp e join dept d on e.deptno=d.deptno;
```

Natural Join

A **Natural Join** is a join operation that joins two tables by their common column. Similar to inner join but it removes the duplicate columns in the result.

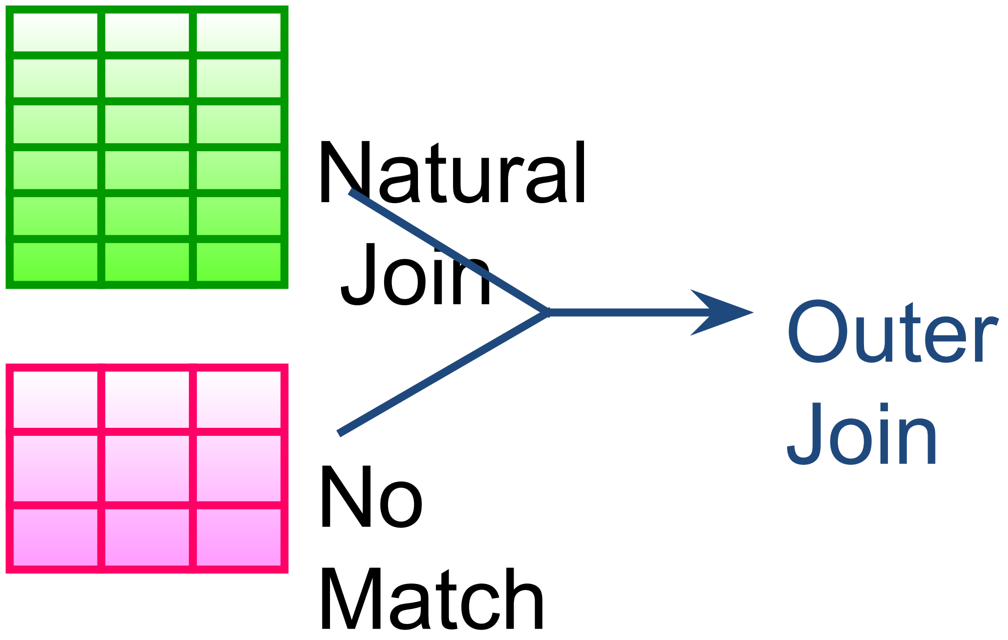
```
Select * from emp natural join dept;
```

Natural Join



Outer Join (Left, Right, Full)

An **Outer Join** is a join operation that includes rows that have a match, plus rows that do not have a match in the other table.



Left Outer Join

A **Left Outer Join** is returns all rows from the left table, even if there are no matches in the right table.

```
Select * from dept d left join emp e on d.
deptno=e.deptno;
                    (or)
Select * from dept d, emp e where d.deptno=e.
deptno(+);
```

Right Outer Join

A **Right Outer Join** is returns all rows from the right table, even if there are no matches in the left table.

```
Select * from emp e right join dept d on e.
deptno=d.deptno;
                    (or)
Select * from emp e, dept d where e.
deptno(+)=d.deptno;
```

Full Outer Join

A **Full Outer Join** is returns rows when there is a match in one of the tables

```
Select * from emp e full join dept d on e.
deptno=d.deptno;
```

Self Join

A **Self Join** is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

We wanted to get a list of employees and their immediate managers:

```
Select e1.ename as Employee, e2.ename as Boss
from emp e1, emp e2 where e1.mgr=e2.empno;
```

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.
- A view is created with the **CREATE VIEW** statement.

Uses of a View

- A good database should contain views due to the given reasons:
- **Restricting data access –**
  Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.
- **Hiding data complexity –**
  A view can hide the complexity that exists in a multiple table join.
- **Simplify commands for the user –**
  Views allows the user to select information from multiple tables without requiring the users to actually know how to perform a join.
- **Store complex queries –**
  Views can be used to store complex queries.
- **Rename Columns –**
  Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to to hide the names of the columns of the base tables.
- **Multiple view facility –**
  Different views can be created on the same table for different users.

1. Creating view

- A view can be created using the **CREATE VIEW** statement.
- We can create a view from a single table or multiple tables.
- **Syntax:**

```
CREATE VIEW view name AS
SELECT column1, column2.....
FROM table name
WHERE condition;
```

Note:

view name: Name for the View

table name: Name of the table

condition: Condition to select rows

a. Creating View from a single table

- In this example, we create a View named Details View from the table Student_Detail.
- **Query:**

```
CREATE VIEW Details View AS
SELECT NAME, ADDRESS
FROM Student Details
WHERE STU_ID < 4;
```

To see the data in the View:

```
SELECT * FROM Details View;
```

b. 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 Marks View from two tables Student_Detail and Student Marks.
- **Query:**

```
CREATE VIEW Marks View AS
SELECT Student_Detail.NAME, Student_Detail. ADDRESS,
Student_Marks.MARKS
FROM Student_Detail, Student Mark
WHERE Student_Detail.NAME = Student_Marks.NAME;
```

To see the data in the View:

```
SELECT * FROM Marks View;
```

2.Updating View

- There are certain conditions needed to be satisfied to update a view.
- If any one of these conditions is **not** met, then we will not be allowed to update the view.
- We can use the **CREATE OR REPLACE VIEW** statement to add or remove fields from a view.
- **Syntax:**

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1,coulmn2,..
FROM table_name
WHERE condition;
```

Example:

```
CREATE OR REPLACE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
    StudentMarks.MARKS,  StudentMarks.AGE
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

- **If we fetch all the data from MarksView now as:**

```
SELECT * FROM MarksView;
```

a. **Inserting a row in a view**:
   We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.

   Syntax:

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

   Note: view_name: Name of the View

b. **Deleting a row from a View**:
   Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view.

- **Syntax**: DELETE FROM view_name WHERE condition;
- **Note: view_name**: Name of view from where we want to delete rows
  condition: Condition to select rows
- **Example**:
  DELETE FROM DetailsView
   WHERE NAME="Suresh";

If we fetch all the data from DetailsView now as,

- SELECT * FROM DetailsView;

3. Deleting View

- SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.
- **Syntax**

  DROP VIEW view name;
- **Note:**
- **view name**: Name of the View which we want to delete.
- **Example:**
- DROP VIEW Marks View;

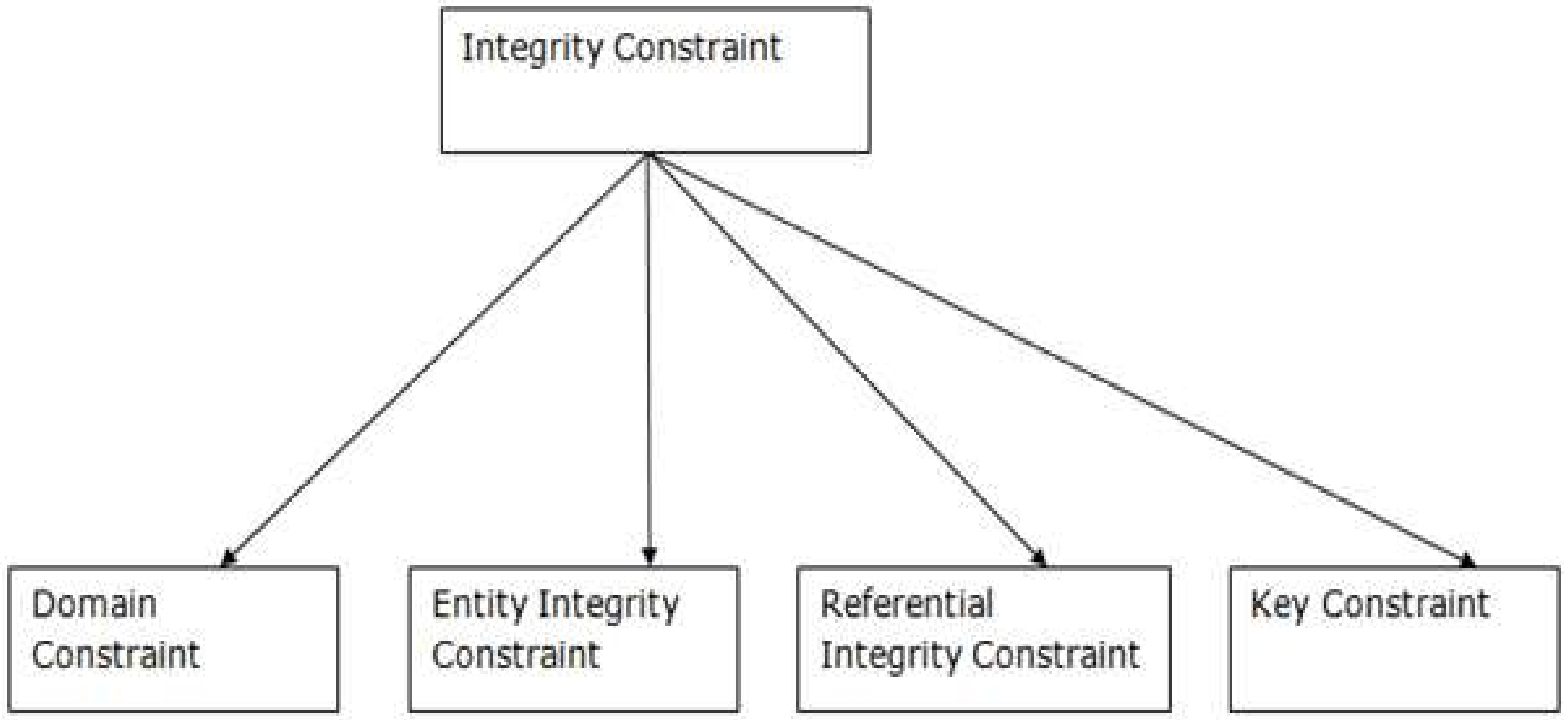
Integrity Constraints Over Relations

- 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.

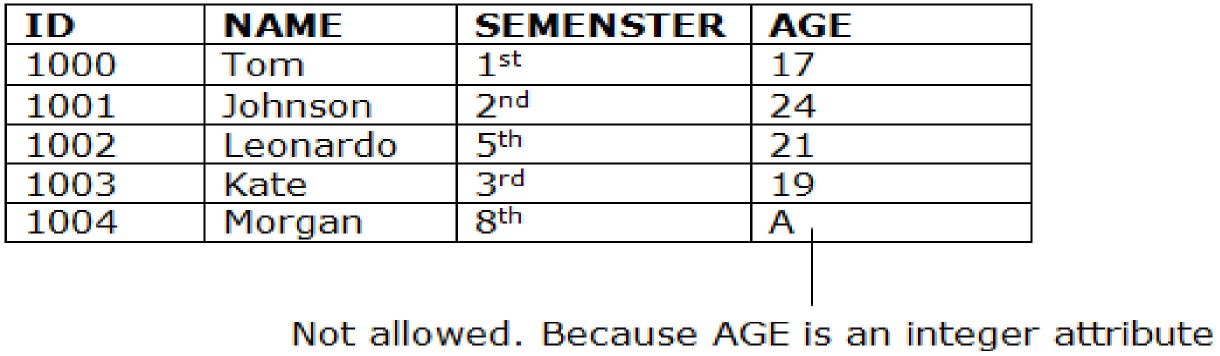
Types of Integrity Constraint



1. 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:

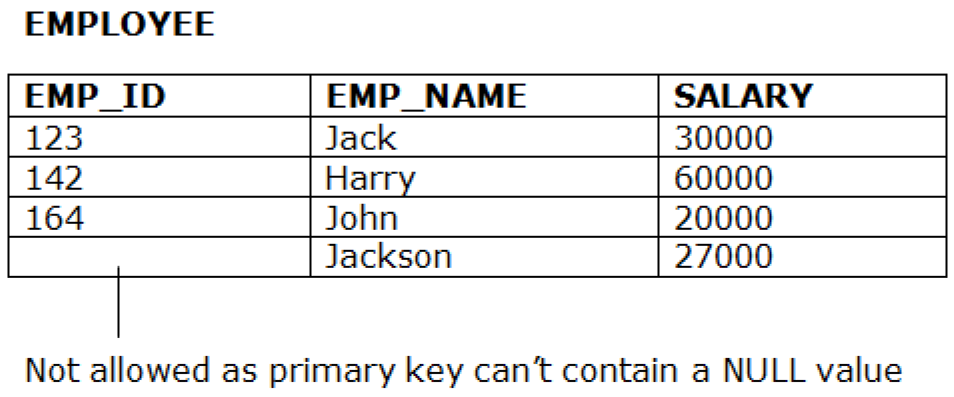


ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1004	Morgan	8th	A

Not allowed. Because AGE is an integer attribute

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:**



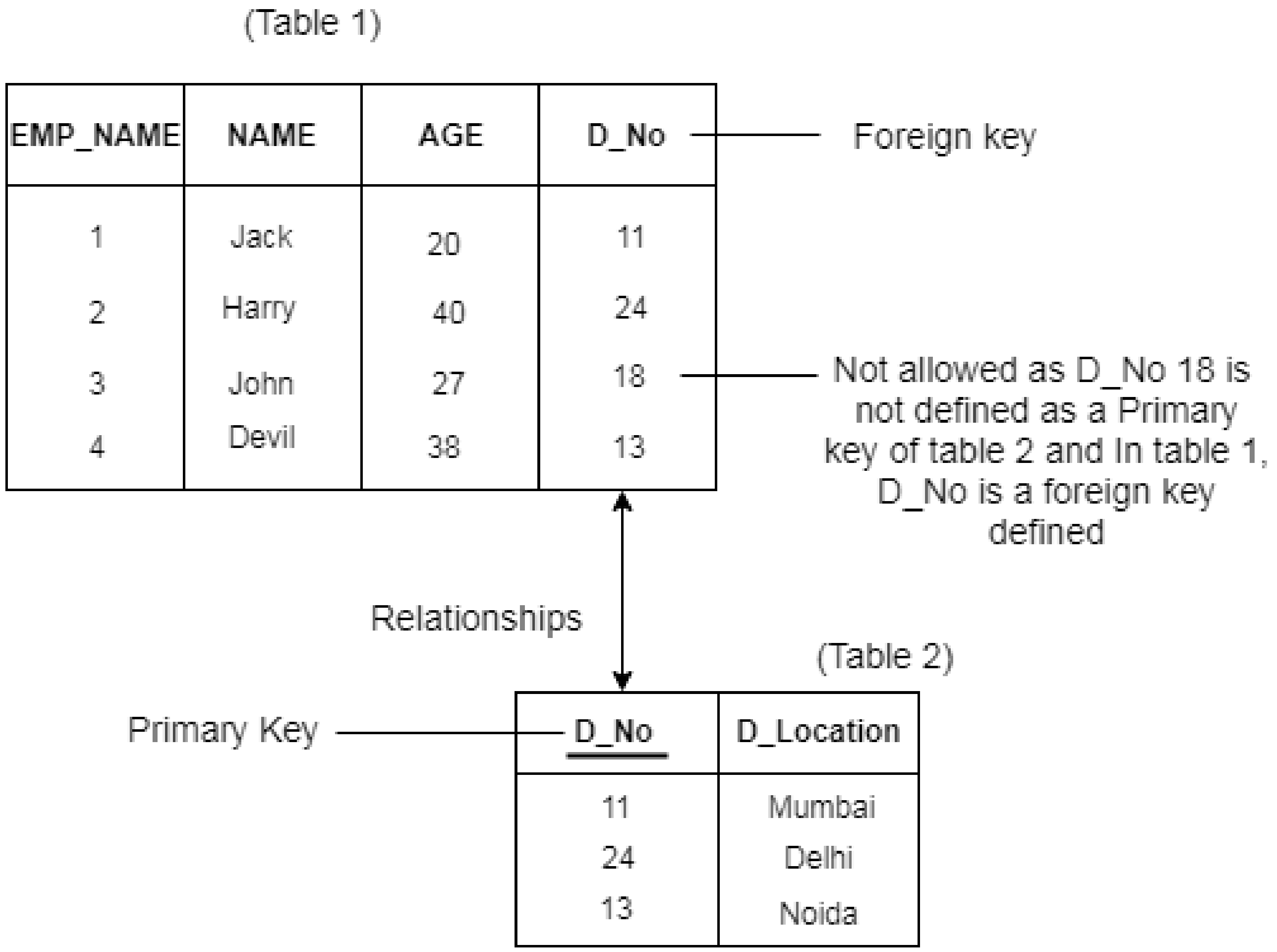
EMPLOYEE

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.

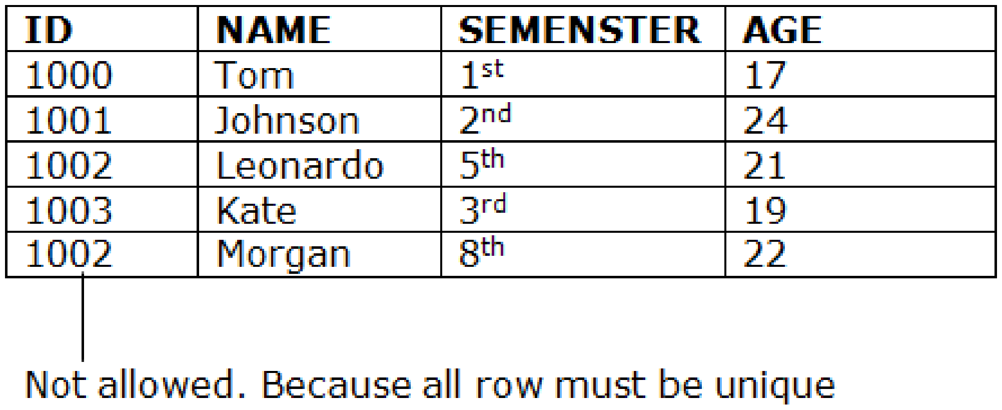

(Table 1)
Foreign key
Not allowed as D_No 18 is not defined as a Primary key of table 2 and In table 1, D_No is a foreign key defined
Relationships
(Table 2)
Primary Key

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

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

DESTROYING/ALTERING TABLES AND VIEWS

- If we decide that we no longer need a base table and want to destroy it (i.e., delete all the rows *and* remove the table definition information).

- **we can use the DROP TABLE command.**
- For example, DROP TABLE Students RESTRICT destroys the Students table unless some view or integrity constraint refers to Students; if so, the command fails.

- **ALTER TABLE modifies the structure of an existing table.**

```
 ALTER TABLE Students
ADD COLUMN maiden-name CHAR(10)
```

Triggers and Active Databases

- A trigger is a procedure that is automatically invoked by the DBMS in response to specified change to the database
- A database that has set of associated trigger is called an active database

It contains 3 parts

- **Event :** a change to the database that activates the trigger.
- **Condition** : A query or test that runs when the trigger is activated.
- **Action** : a procedure that is executed when the trigger is activated and its condition is true.

Use of trigger

- **Triggers may be used for any of the following reasons −**
  - To implement any complex business rule, that cannot be implemented using integrity constraints.
  - Triggers will be used to audit the process. For example, to keep track of changes made to a table.
  - Trigger is used to perform automatic action when another concerned action takes place.

- **Example:**

```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
 on [table_name] [for each row] [trigger_body]
```

- **Explanation of syntax:**
- **create trigger [trigger_name]:** Creates or replaces an existing trigger with the trigger_name.
- **[before | after]:** This specifies when the trigger will be executed.
- **{insert | update | delete}:** This specifies the DML operation.
- **on [table_name]:** This specifies the name of the table associated with the trigger.
- **[for each row]:** This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
- **[trigger_body]:** This provides the operation to be performed as trigger is fired

- **Example:**

```
create trigger stud_marks
before INSERT
on Student for each row
set Student.total = Student.subj1 + Student.subj2
   + Student.subj3, Student.per = Student.total *
   60 / 100;
```

Types of triggers

- The different types of triggers are explained below −
  1. **Statement level trigger**
  2. **Before-triggers**
  3. **After-triggers**
  4. **Row-level triggers**

1. **Statement level trigger –**

   It is fired only once for DML statement irrespective of number of rows affected by statement. Statement-level triggers are the default type of trigger.

2. **Before-triggers –** At the time of defining a trigger we can specify whether the trigger is to be fired before a command like INSERT, DELETE, or UPDATE is executed or after the command is executed.

- Before triggers are automatically used to check the validity of data before the action is performed.
- For instance, we can use before trigger to prevent deletion of rows if deletion should not be allowed in a given case.

3. **After-triggers –** It is used after the triggering action is completed.

- For example, if the trigger is associated with the INSERT command then it is fired after the row is inserted into the table.

4. **Row-level triggers –** It is fired for each row that is affected by DML command.

- For example, if an UPDATE command updates 150 rows then a row-level trigger is fired 150 times whereas a statement-level trigger is fired only for once.

Active Databases

- Active Database is a database consisting of set of triggers.
- These databases are very difficult to be maintained because of the complexity that arises in understanding the effect of these triggers.
- In such database, DBMS initially verifies whether the particular trigger specified in the statement that modifies the database) is activated or not, prior to executing the statement.
- If the trigger is active then DBMS executes the condition part and then executes the action part only if the specified condition is evaluated to true.
- It is possible to activate more than one trigger within a single statement.
- In such situation, DBMS processes each of the trigger randomly. The execution of an action part of a trigger may either activate other triggers or the same trigger that Initialized this action.
- Such types of trigger that activates itself is called as ‘recursive trigger’. The DBMS executes such chains of trigger in some pre-defined manner but it effects the concept of understanding.

Features of Active Database

- It possess all the concepts of a conventional database i.e. data modelling facilities, query language etc.
- It supports all the functions of a traditional database like data definition, data manipulation, storage management etc.
- It supports definition and management of ECA rules.
- It detects event occurrence.
- It must be able to evaluate conditions and to execute actions.
- It means that it has to implement rule execution.

Advantages

- Enhances traditional database functionalities with powerful rule processing capabilities.
- Enable a uniform and centralized description of the business rules relevant to the information system.
- Avoids redundancy of checking and repair operations.
- Suitable platform for building large and efficient knowledge base and expert systems.

Cursors

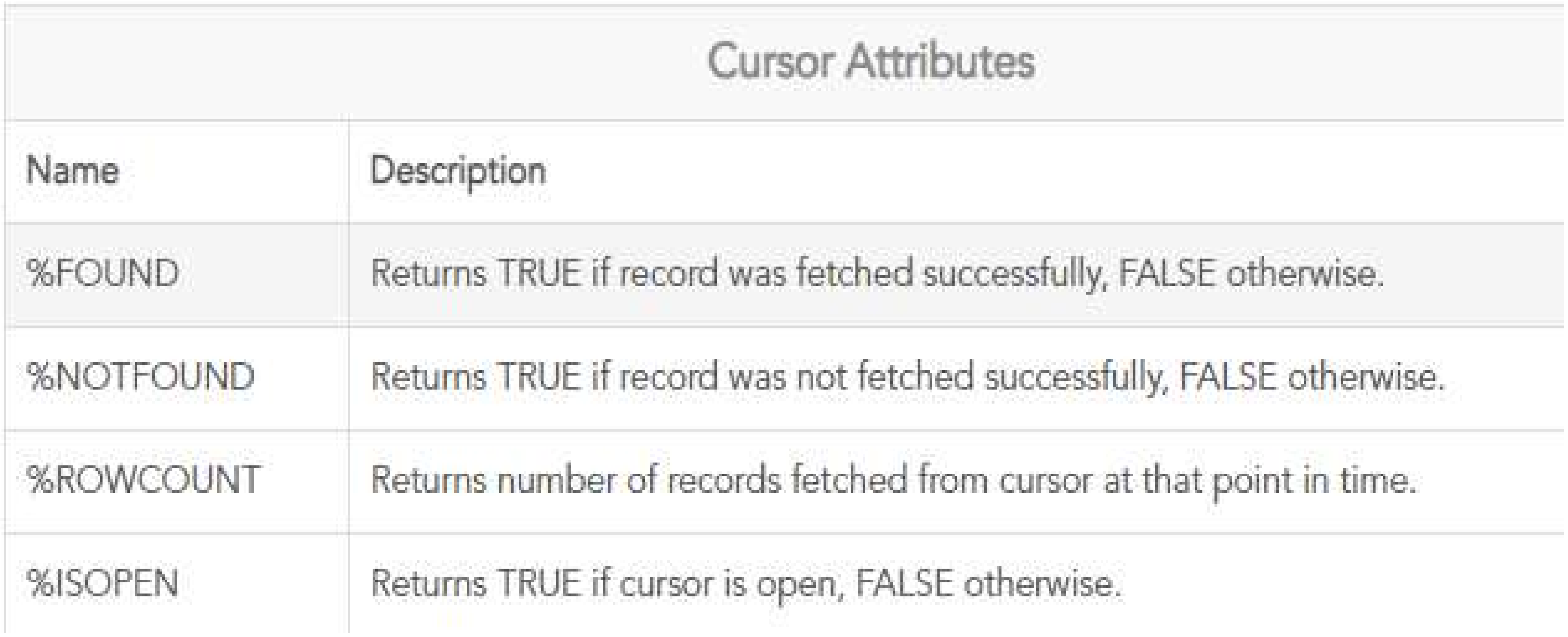
- When an SQL statement is processed, Oracle creates a memory area known as context area.
- A cursor is a pointer to this context area. It contains all information needed for processing the statement.
- In PL/SQL, the context area is controlled by Cursor.
- A cursor contains information on a select statement and the rows of data accessed by it.
- A cursor is used to referred to a program to fetch and process the rows returned by the SQL statement, one at a time.
- There are two types of cursors:

1. **Implicit Cursors**
2. **Explicit Cursors**

1.Implicit cursors

- The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if you don't use an explicit cursor for the statement.
- These are created by default to process the statements when DML statements like INSERT, UPDATE, DELETE etc. are executed.
- Oracle provides some attributes known as Implicit cursor's attributes to check the status of DML operations.
- Some of them are: **%FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.**
- **For example,** When you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected.
- If you run a SELECT INTO statement in PL/SQL block, the implicit cursor attribute can be used to find out whether any row has been returned by the SELECT statement. It will return an error if there no data is selected.

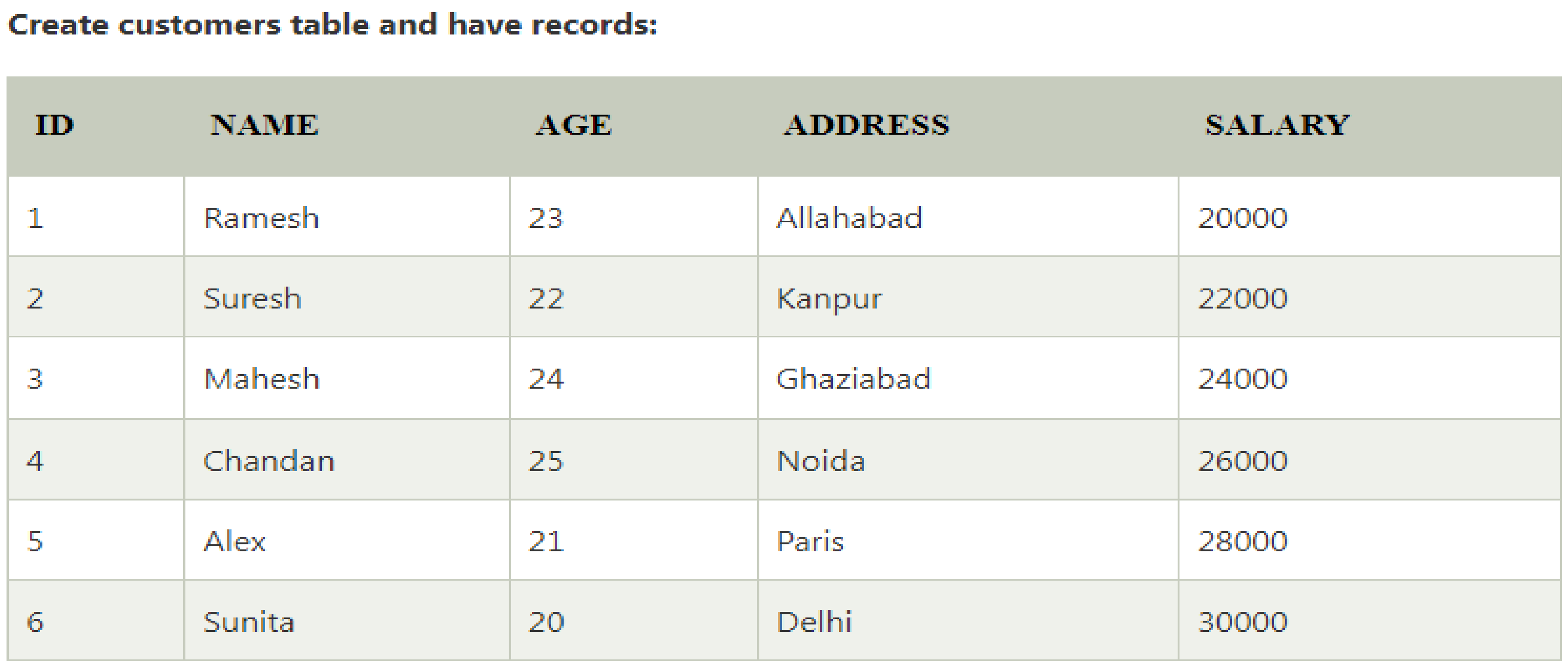
- **The following table specifies the status of the cursor with each of its attribute.**



Cursor Attributes	
Name	Description
%FOUND	Returns TRUE if record was fetched successfully, FALSE otherwise.
%NOTFOUND	Returns TRUE if record was not fetched successfully, FALSE otherwise.
%ROWCOUNT	Returns number of records fetched from cursor at that point in time.
%ISOPEN	Returns TRUE if cursor is open, FALSE otherwise.

PL/SQL Implicit Cursor Example

- Let's execute the following program to update the table and increase salary of each customer by 5000.
- Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:



Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

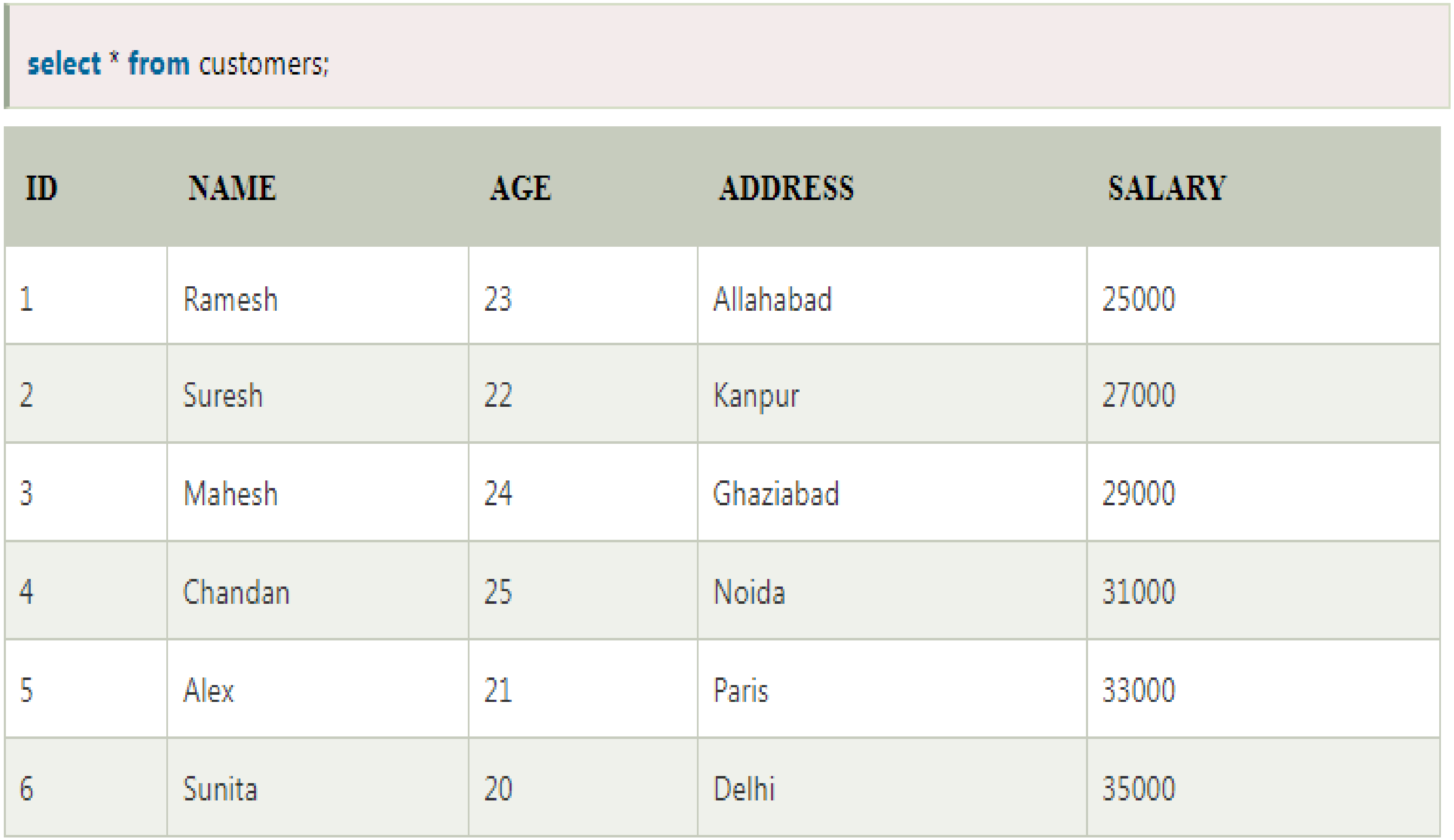
Create procedure:

```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE  customers
   SET salary = salary + 5000;
   IF sql%notfound THEN
      dbms_output.put_line('no customers updated');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers updated ');
   END IF;
END;
/
```

Output:

```
6 customers updated
PL/SQL procedure successfully completed.
```

Result table



```
select * from customers;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	25000
2	Suresh	22	Kanpur	27000
3	Mahesh	24	Ghaziabad	29000
4	Chandan	25	Noida	31000
5	Alex	21	Paris	33000
6	Sunita	20	Delhi	35000

2.Explicit Cursors

- The Explicit cursors are defined by the programmers to gain more control over the context area.
- These cursors should be defined in the declaration section of the PL/SQL block.
- It is created on a SELECT statement which returns more than one row.
- **Syntax:**
- **CURSOR** cursor_name **IS** select_statement;;

Steps to follow explicit cursor

You must follow these steps while working with an explicit cursor.

1. ***Declare the cursor to initialize in the memory.***
2. ***Open the cursor to allocate memory.***
3. ***Fetch the cursor to retrieve data.***
4. ***Close the cursor to release allocated memory.***

1) Declare the cursor:

It defines the cursor with a name and the associated SELECT statement.

Syntax:

```
CURSOR name IS
 SELECT statement;
```

2) Open the cursor:

It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

Syntax:

```
OPEN cursor_name;
```

3) Fetch the cursor:

It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows:

Syntax:

```
FETCH cursor_name INTO variable_list;
```

4) Close the cursor:

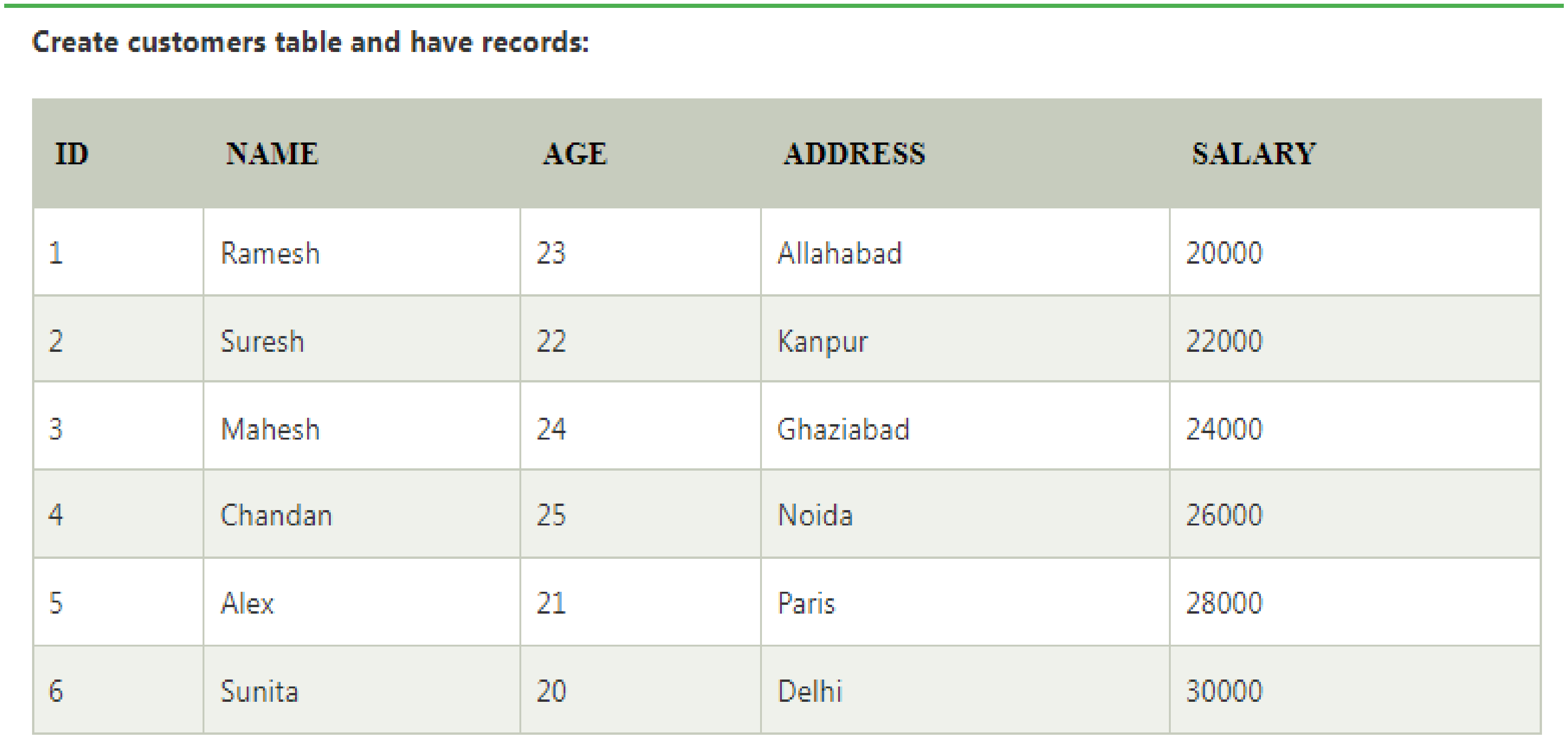
It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.

Syntax:

```
Close cursor_name;
```

PL/SQL Explicit Cursor Example

- Explicit cursors are defined by programmers to gain more control over the context area. It is defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.
- Let's take an example to demonstrate the use of explicit cursor. In this example, we are using the already created CUSTOMERS table.



Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Create procedure:

Execute the following program to retrieve the customer name and address.

```
DECLARE
  c_id customers.id%type;
  c_name customers.name%type;
  c_addr customers.address%type;
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
    FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
/
```

Output:

```
1  Ramesh  Allahabad
2  Suresh  Kanpur
3  Mahesh  Ghaziabad
4  Chandan  Noida
5  Alex  Paris
6  Sunita  Delhi
PL/SQL procedure successfully completed.
```

THANK YOU