ORACLE DATA TYPE:

1. Char
The char datatype consist character as alpha, numeric and alpha-numeric.
Capacity :
255 bytes Default and minimum size is 1 byte.
Syntax :
Fieldname Char(size)

2. Varchar
The varchar datatype consist character as alpha, numeric and alpha-numeric.
Capacity :
2000 bytes
Syntax :
Fieldname Varchar(size)

3. Varchar2
The varchar2 datatype consist character as alpha, numeric and alpha-numeric.
Capacity :
4000 bytes
Syntax :
Fieldname Varchar2(size)

4. Number
The number datatype consists only numeric.
Capacity :
38 digit
Syntax :
Fieldname number(p) p – precision
Fieldname number(p,s) p – precision, s – scale
Example:
Fieldname Number(7,2) 50000.20 7– precision and 2 – scale

5. Date
The date datatype consist only date in proper oracle format.
Format : dd-mon-yyyy
Syntax: Date

6. Blob
BLOB data type same as BFILE data type to store unstructured binary object into Operating System file. BLOB type fully supported transactions are recoverable and replicated Capacity.
Capacity :
the blob capacity is system dependent
Syntax :
Fieldname blob(size)

7. Raw & Long Raw
The RAW and LONG RAW data types are for storing binary data or byte strings e.g., the content of documents, sound files, and video files.

The RAW data type can store up to 2000 bytes while the LONG RAW data type can store up to 2GB.

8. CLOB
CLOB stands for character large object. You use CLOB to store single-byte or multibyte characters with the maximum size is 4 gigabytes
Note that CLOB supports both fixed-with and variable-with character sets.

9. NCLOB
NCLOB is similar to CLOB except that it can store the Unicode characters.

10. LONG
This data type is used to store large text data up to the maximum size of 2GB. These are mainly used in the data dictionary.
LONG data type is used to store character set data.

ROWID Pseudocolumn

For each row in the database, the ROWID pseudocolumn returns the address of the row. Oracle Database rowid values contain information necessary to locate a row:
- The data object number of the object
- The data block in the data file in which the row resides
- The position of the row in the data block (firstrow is 0)
- The data file in which the row resides (first file is 1). The file number is relative to the tablespace.

Usually, a rowid value uniquely identifies a row in the database. However, rows in different tables that are stored together in the same cluster can have the same rowid.
Values of the ROWID pseudocolumn have the data type ROWID or UROWID. Refer to "Rowid Data Types" and "UROWID Data Type" for more information. Rowid values have several important uses:
- They are the fastest way to accessa single row.
- They can show you how the rows in a table are stored.
- They are unique identifiers for rows in a table.

You should not use ROWID as the primary key of a table. If you delete and reinsert a row with the Import and Export utilities, for example, then its rowid may change. If you delete a row, then Oracle may reassign its rowid to a new row inserted later. Although you can use the ROWID pseudocolumn in the SELECT and WHERE clause of a query, these pseudocolumn values are not actually stored in the database. You cannot insert, update, or delete a value of the ROWID pseudocolumn.

Example This statement selects the address of all rows that contain data for employees in department 20:
SELECT ROWID, last_name FROM employees WHERE department_id = 20;

DUAL Table

DUAL is a table automatically created by Oracle Database along with the data dictionary. DUAL is in the schema of the user SYS but is accessible by the name DUAL to all users. It

has one column, DUMMY, defined to be VARCHAR2(1), and contains one row with a value X. Selecting from the DUAL table is useful for computing a constant expression with the SELECTstatement. Because DUAL has only one row, the constant is returned only once. Alternatively, you can select a constant, pseudocolumn, or expression from any table, but the value will be returned as many times as there are rows in the table.
An example of using the DUAL table would be:

SELECT SYSDATE FROM DUAL;

This would return the system's current date to your screen.

SYSDATE
03/JUL/16

Functions

Scalar functions (in contrast to data functions) do not aggregate data or compute aggregate statistics. Instead, scalar functions compute and substitute a new data value for each value associated with a data item.

String Scalar Functions	**Description**
Ascii (s)	Returns an ASCII numeric representation of string *s*. Example: Ascii ("AZ") = 65
Concat (s1, s2)	Returns text strings *s1* and *s2* concatenated. Example: Concat ("interactive","reporting") = interactivereporting
Initcap (s)	Returns string *s* with the first letter of each word capitalized, and remaining characters in lower case. Example: Initcap ("santa fe") = Santa Fe
Instr (s1,s2,n,m)	Returns position of *m*th occurrence of string *s2* in string *s1*, beginning at position number *n*. If n is negative, the count is made backwards from the end of *s1*. If no values are found, 0 is returned. Examples: `Instr ("Mississippi",'s',5,2) = 7` Instr (City, 'a', -2, 1)
Length (s)	Returns character count of string *s*. Example: Length ("Pittsburgh") = 10
Lower (s)	Returns string *s* in lower case. Example: Lower ("CD-Rom") = cd-rom
Ltrim (s1,s2)	Trims string *s1* from the left, up to the first character not included in string *s2*. Example: Ltrim ("Mr. Jones", "Mr. ") = Jones
Replace (s1,s2,s3)	Returns string item *s1* with all occurrences of string *s2* replaced by string *s3*. The default for *s3* deletes each occurrence of *s2*. Example: Replace (customer,"Mrs.","Ms.") = replaces Mrs. with Ms. for all values of customer containing `Mrs.'

Rtrim (s1,s2)	Trims column string *s1* from the right, up to the first character not included in string *s2*. Example: Rtrim ("Columbus, OH",", OH") = Columbus
Substr (s,n,m)	Returns a portion of string *s*, *m* characters long, beginning at numeric position *n*. The default action for *m* includes all remaining characters. Example: Substr ("312/989-9989","1","3") = "312"
Upper (s)	Returns string *s* in upper case. Example: Example: Upper ("st.") = ST.

Numeric Scalar Functions	**Description**
Avg (numbers, break_col, break_value)	Returns the average (arithmetic mean) of values in a number column. The average includes null values when calculating the arithmetic mean. Avg example
chr (n)	Returns string converted from ASCII numeric code *n*. Example: Chr (65) = A
Count (numbers, break_col, break_value)	Counts and returns the number of rows in a column. Count example
Sum (numbers, break_col, break_value)	Returns the total of a column of numbers. Sum

Date Scalar Functions	**Description**
Add_Months (d,n)	Adds *n* months to date *d*. Example: Add_Months (`5/13/99',4) = 9/13/99
Last_Day (d)	Returns date of the last day of the month containing date *d*. Example: Last_Day (`12/6/99') = 12/31/99
Months_Between (d1,d2)	Returns the number of months between dates *d1* and *d2* as a real number (fractional value). Example: Months_Between (`12/5/99','5/6/99') = 6.9677
Next_Day (d,s)	Returns the date of the first weekday *s* after date *d*. If s is omitted, add one day to *d*. Example: NextDay (`12/16/99',"Monday") = 12/22/99
Sysdate	Returns the current system date and time for each record in item *c*. Example: Sysdate = 2/11/96 19:54:36

To_Date (s)	Returns date type in place of date-string *s*. This function does not change the data, but rather the item data type. The results can be computed mathematically. Example: `ToDate ("10/12/96") = 10/12/96` Note: See Functions for Returning the Day of the Week for information on how to return the day of the week on which a given date falls.
Systimestamp	The SYSTIMESTAMP function is used to get the system date, including fractional seconds and time zone, of the system on which the database resides.

Math Scalar Functions	**Description**
Abs (n)	Returns the absolute value of number *n*. Example: Abs (-3) = 3
Ceil (n)	Returns the smallest integer value greater than or equal to number *n*. Example: Ceil (5.6) = 6
Cos (n)	Returns cosine of number *n* radians. Example: Cos (0.5) = .8778
Count (c)	Returns the number of row values in *c* (including nulls). Example: Count (units) = tally of rows in units
Exp (n)	Returns *e* (2.718) raised to exponential power *n*. Example: Exp (4) = 54.598
Mod (n,m)	Returns the integer remainder of number *n* divided by number *m*. If *m* is larger, the default value is *n*. Example: Mod (6,2) = 0
Power (n,m)	Returns number n raised to exponential power *m*. Example: Power (10,5) = 100,000
Round (n,m)	Returns number *n* rounded to m decimal places. The default value for *m* is 0. Example: Round (5.6178,2) = 5.62
Sign (n)	Returns indicator of -1, 0, or 1 if number *n* is variously negative, 0, or positive. Example: Sign (-4) = -1
Sin (n)	Returns sine of number *n* radians. Example: Sin (86) = -0.923
Sqrt (n)	Returns square root of number *n*. Example: Sqrt (81) = 9
Tan (n)	Returns tangent of number *n* radians. Example: Tan (30) = -6.405
Trunc (n,m)	Returns number *n* truncated to number m decimal places. The default value for *m* is 0. Example: Trunc (56.0379,2) = 56.03

Practice Queries of different functions

create table student1(rno number primary key,nm varchar(10),dob date,m1 number,m2 number,total number,per number(5,2));

insert into student1(rno,nm,dob,m1,m2) values(1,'bob','23-feb-1991',90,95);

insert into student1(rno,nm,dob,m1,m2) values(2,'peter','05-jan-1989',70,65);

insert into student1(rno,nm,dob,m1,m2) values(3,'rehan','3-mar-2001',12,7);

insert into student1(rno,nm,dob,m1,m2) values(4,'aryan','30-dec-1997',60,65);

update student1 set total=m1+m2;

update student1 set per=(total*100)/205;

create table customer(cid number primary key,nm varchar(10),occupation varchar(15),age number);

create table orders(oid number primary key,cid number,constraint fine foreign key(cid) references customer(cid),prod_nm varchar(15),order_date date);

insert into customer values(101,'Peter','Engineer',32);

insert into customer values(102,'Joseph','Developer',30);

insert into customer values(103,'John','Leader',28);

insert into customer values(104,'Stephen','Scientist',45);

insert into customer values(105,'Suzi','Carpenter',26);

insert into customer values(106,'Bob','Actor',25);

insert into customer values(107,null,null,null);

insert into orders values(1,101,'Laptop','10-jan-2020');

insert into orders values(2,103,'Desktop','10-jan-2020');

insert into orders values(3,106,'Iphone','10-jan-2020');

insert into orders values(4,104,'Mobile','10-jan-2020');

insert into orders values(5,102,'TV','10-jan-2020');

1) select sysdate from dual;
2) select add_months(order_date,5) as addmonths from orders;
3) select last_day(order_date) as lastdate from orders;
4) select months_between(order_date,'01-jan-2022') as monthmid from orders;
5) select next_day(order_date,'monday') as monthmid from orders;
6) select to_char(order_date) as new from orders;
7) select concat(oid, to_char(order_date)) as new from orders;
8) select trunc(per,1) from student1;

9) select trunc(5.6178,2) from dual;
10) select round(per,1) from student1;
11) select round(5.6178,1) from dual;
12) select to_date('13-jan-23','dd-mon-yy') from dual;

INDEX

Indexes are **special lookup tables** that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.

An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

The CREATE INDEX Command

The basic syntax of a **CREATE INDEX** is as follows.

CREATE INDEX index_name ON table_name (COLUMN_NAME);

Single-Column Indexes

A single-column index is created based on only one table column. The basic syntax is as follows.

CREATE INDEX index_name
ON table_name (column_name);

CREATE INDEX index_name
ON table_name (column_name1, column_name2, column_name3);

Unique Indexes

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows.

CREATE UNIQUE INDEX index_name on table_name (column_name);

Composite Indexes

A composite index is an index on two or more columns of a table. Its basic syntax is as follows.

CREATE INDEX index_name
On table_name (column1, column2);

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

The DROP INDEX Command
An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because the performance may either slow down or improve.
The basic syntax is as follows − DROP INDEX index_name;
You can check the INDEX Constraint chapter to see some actual examples on Indexes.

When should indexes be avoided?
Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered.

- Indexes should not be used on smalltables.
- Tables that have frequent, large batch updates or insertoperations.
- Indexes should not be used on columns that contain a high number of NULLvalues.
- Columns that are frequently manipulated should not beindexed.

SQL JOIN

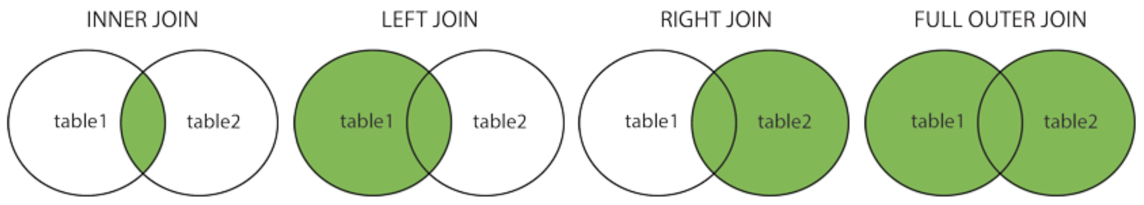
A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Different Types of SQL JOINs
Here are the different types of the JOINs in SQL:

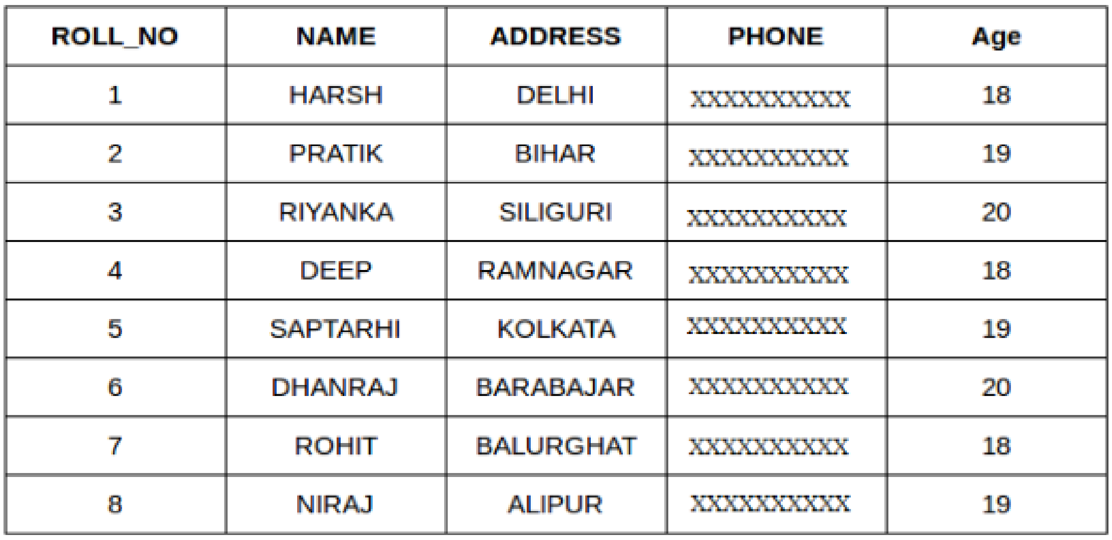
- (INNER) JOIN: Returns records that have matching values in both tables
- LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
- FULL (OUTER)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

SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are as follows:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

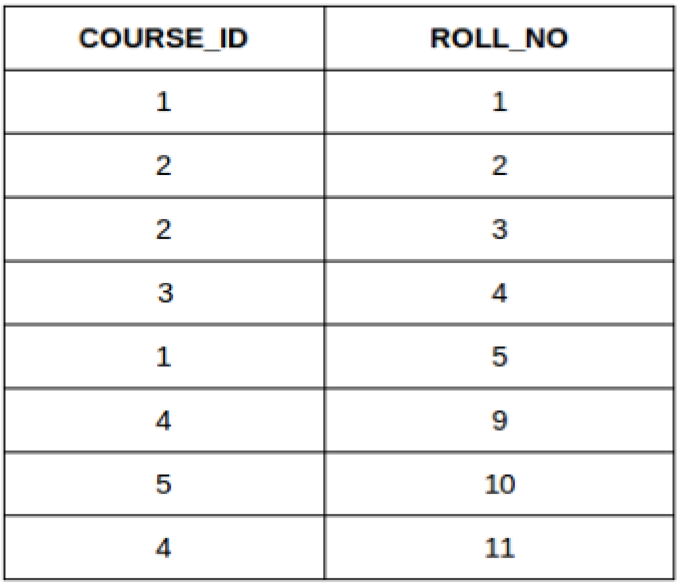


Consider the two tables below:
Student



ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

StudentCourse



COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

The simplest Join is INNER JOIN.
A. INNER JOIN
The INNER JOIN keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

Syntax: SELECT table1.column1,table1.column2,table2.column1,.... FROM table1 INNER JOIN table2 ON table1.matching_column = table2.matching_column;

table1: First table. **table2**: Second table **matching_column**: Column common to both the tables.

Note: *We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.*

Example Queries(INNER JOIN) This query will show the names and age of students enrolled in different courses.
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student INNER JOIN StudentCourse ON Student.ROLL_NO = StudentCourse.ROLL_NO;

Output:

COURSE_ID	NAME	Age
1	HARSH	18
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18
1	SAPTARHI	19

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

Syntax: SELECT table1.column1,table1.column2,table2.column1,.... FROM table1 LEFT JOIN table2 ON table1.matching_column = table2.matching_column;

table1: First table. table2: Second table matching_column: Column common to both the tables.

***Note**: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.*

Example Queries (LEFT JOIN): SELECT Student.NAME,StudentCourse.COURSE_ID FROM Student LEFT JOIN StudentCourse ON StudentCourse.ROLL_NO = Student.ROLL_NO;

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*

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

Syntax: SELECT table1.column1,table1.column2,table2.column1,.... FROM table1 RIGHT JOIN table2 ON table1.matching_column = table2.matching_column;

table1: First table. table2: Second table matching_column: Column common to both the tables.

***Note**: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.*

Example Queries(RIGHT JOIN): SELECT Student.NAME,StudentCourse.COURSE_ID FROM Student RIGHT JOIN StudentCourse ON StudentCourse.ROLL_NO = Student.ROLL_NO;

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
NULL	4
NULL	5
NULL	4

D. FULL JOIN
FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain *NULL* values.

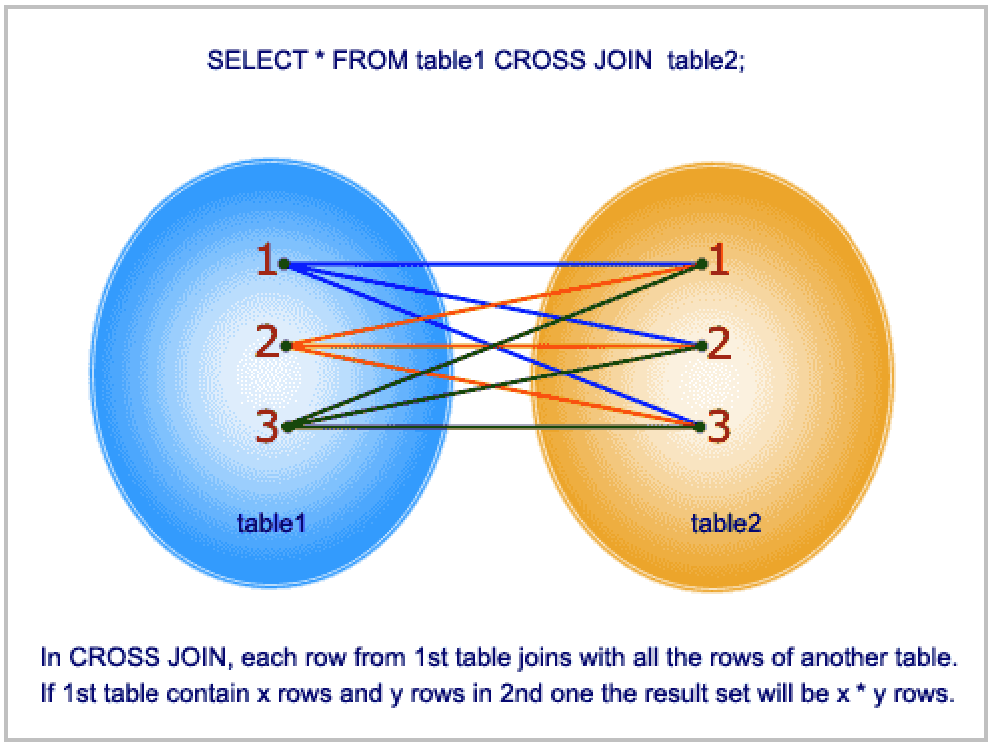
Syntax: SELECT table1.column1,table1.column2,table2.column1,....
FROM table1 FULL JOIN table2 ON table1.matching_column = table2.matching_column;

table1: First table. table2: Second table matching_column: Column common to both the tables.

Example Queries(FULL JOIN): SELECT Student.NAME,StudentCourse.COURSE_ID FROM Student FULL JOIN StudentCourse ON StudentCourse.ROLL_NO = Student.ROLL_NO;

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	NULL
ROHIT	NULL
NIRAJ	NULL
NULL	4
NULL	5
NULL	4

What is Cross Join in SQL?
The SQL CROSS JOIN produces a result set which is the number of rows in the first table multiplied by the number of rows in the second table if no WHERE clause is used along with CROSS JOIN.This kind of result is called as Cartesian Product. If WHERE clause is used with CROSS JOIN, it functions like an INNER JOIN.


SELECT * FROM table1 CROSS JOIN table2;
1
2
3
table1
1
2
3
table2
In CROSS JOIN, each row from 1st table joins with all the rows of another table.
If 1st table contain x rows and y rows in 2nd one the result set will be x * y rows.

An alternative way of achieving the same result is to use column names separated by commas after SELECT and mentioning the table names involved, after a FROM clause.

Syntax: `SELECT * FROM table1 CROSS JOIN table2;`

Example: Here is an example of cross join in SQL between two tables.

Sample table: foods

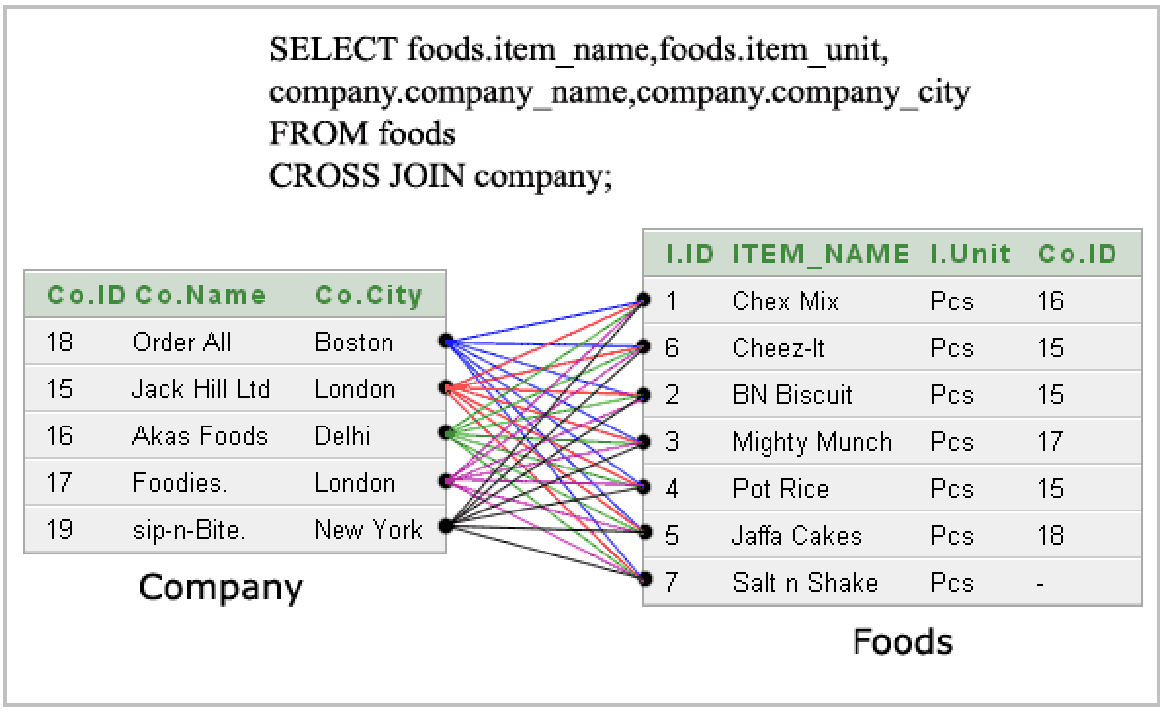
```
+----------+--------------+-----------+------------+
| ITEM_ID  | ITEM_NAME    | ITEM_UNIT | COMPANY_ID |
+----------+--------------+-----------+------------+
| 1        | Chex Mix     | Pcs       | 16         |
| 6        | Cheez-It     | Pcs       | 15         |
| 2        | BN Biscuit   | Pcs       | 15         |
| 3        | Mighty Munch | Pcs       | 17         |
| 4        | Pot Rice     | Pcs       | 15         |
| 5        | Jaffa Cakes  | Pcs       | 18         |
| 7        | Salt n Shake | Pcs       |            |
+----------+--------------+-----------+------------+
```

Sample table: company

```
+------------+---------------+--------------+
| COMPANY_ID | COMPANY_NAME  | COMPANY_CITY |
+------------+---------------+--------------+
| 18         | Order All     | Boston       |
| 15         | Jack Hill Ltd | London       |
| 16         | Akas Foods    | Delhi        |
| 17         | Foodies.      | London       |
| 19         | sip-n-Bite.   | New York     |
+------------+---------------+--------------+
```

To get item name and item unit columns from foods table and company name, company city columns from company table, after a CROSS JOINING with these mentioned tables, the following SQL statement can be used:

SQL Code: `SELECT foods.item_name,foods.item_unit, company.company_name,company.company_city FROM foods,company;`



```
SELECT foods.item_name,foods.item_unit,
company.company_name,company.company_city
FROM foods
CROSS JOIN company;
```

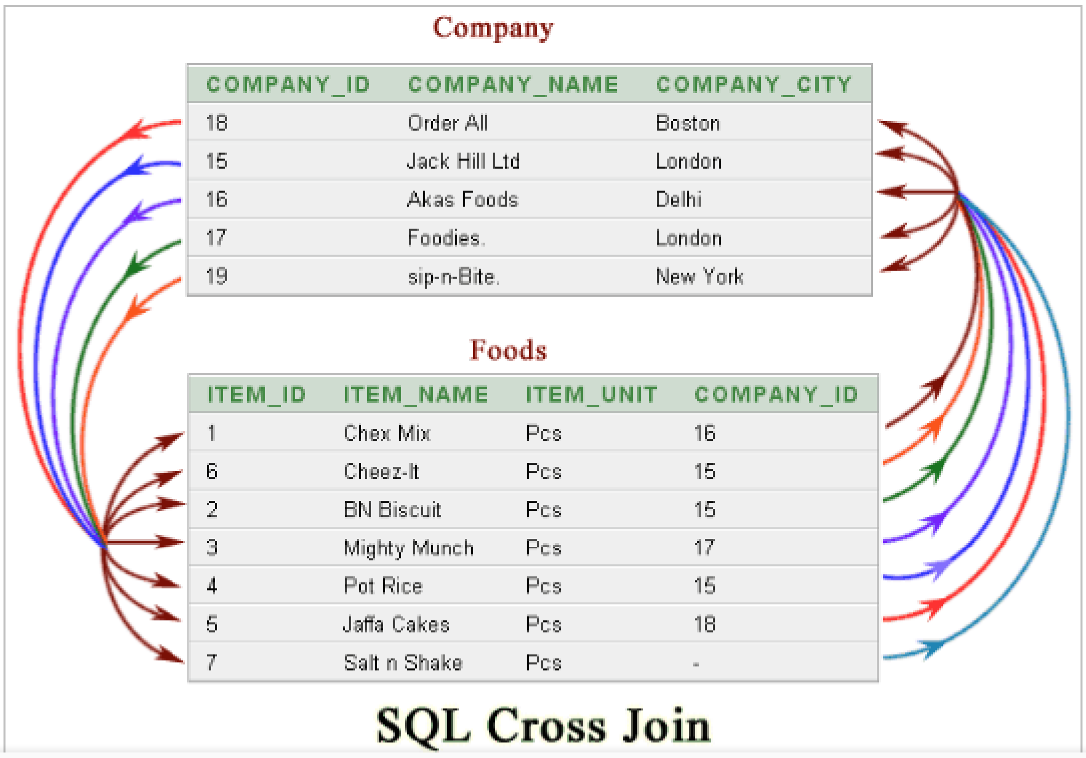
Co.ID	Co.Name	Co.City
18	Order All	Boston
15	Jack Hill Ltd	London
16	Akas Foods	Delhi
17	Foodies.	London
19	sip-n-Bite.	New York

I.ID	ITEM_NAME	I.Unit	Co.ID
1	Chex Mix	Pcs	16
6	Cheez-It	Pcs	15
2	BN Biscuit	Pcs	15
3	Mighty Munch	Pcs	17
4	Pot Rice	Pcs	15
5	Jaffa Cakes	Pcs	18
7	Salt n Shake	Pcs	-

Output:

```
ITEM_NAME       ITEM_UNIT  COMPANY_NAME    COMPANY_CITY
--------------- ---------- --------------- ---------------
Chex Mix        Pcs        Order All       Boston
Cheez-It        Pcs        Order All       Boston
BN Biscuit      Pcs        Order All       Boston
Mighty Munch    Pcs        Order All       Boston
Pot Rice        Pcs        Order All       Boston
Jaffa Cakes     Pcs        Order All       Boston
Salt n Shake    Pcs        Order All       Boston
Chex Mix        Pcs        Jack Hill Ltd   London
Cheez-It        Pcs        Jack Hill Ltd   London
BN Biscuit      Pcs        Jack Hill Ltd   London
Mighty Munch    Pcs        Jack Hill Ltd   London
Pot Rice        Pcs        Jack Hill Ltd   London
Jaffa Cakes     Pcs        Jack Hill Ltd   London
Salt n Shake    Pcs        Jack Hill Ltd   London
Chex Mix        Pcs        Akas Foods      Delhi
Cheez-It        Pcs        Akas Foods      Delhi
BN Biscuit      Pcs        Akas Foods      Delhi
Mighty Munch    Pcs        Akas Foods      Delhi
Pot Rice        Pcs        Akas Foods      Delhi
Jaffa Cakes     Pcs        Akas Foods      Delhi
Salt n Shake    Pcs        Akas Foods      Delhi
Chex Mix        Pcs        Foodies.        London
.........
.........
```

More presentaion of the said output:


Company

COMPANY_ID	COMPANY_NAME	COMPANY_CITY
18	Order All	Boston
15	Jack Hill Ltd	London
16	Akas Foods	Delhi
17	Foodies.	London
19	sip-n-Bite.	New York

Foods

ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_ID
1	Chex Mix	Pcs	16
6	Cheez-It	Pcs	15
2	BN Biscuit	Pcs	15
3	Mighty Munch	Pcs	17
4	Pot Rice	Pcs	15
5	Jaffa Cakes	Pcs	18
7	Salt n Shake	Pcs	-

SQL Cross Join

MySQL Subquery

A subquery in MySQL is a query, which is nested into another SQL query and embedded with SELECT, INSERT, UPDATE or DELETE statement along with the various operators. We can also nest the subquery with another subquery. A subquery is known as the **inner query**, and the query that contains subquery is known as the **outer query**. The inner query executed first gives the result to the outer query, and then the main/outer query will be performed. MySQL allows us to use subquery anywhere, but it must be closed within parenthesis. All subquery forms and operations supported by the SQL standard will be supported in MySQL also.

The following are the rules to use subqueries:
o Subqueries should always use in **parentheses.**

o We can use various comparison operators with the subquery, such as >, <, =, IN, ANY, SOME, and ALL.

o We cannot use the **ORDER BY** clause in a subquery, although it can be used inside the main query.

The following are the advantages of using subqueries:
o The subqueries make the queries in a structured form that allows us to isolate each part of a statement.

o The subqueries provide alternative ways to query the data from the table; otherwise, we need to use complex joins and unions.

o The subqueries are more readable than complex join or union statements.

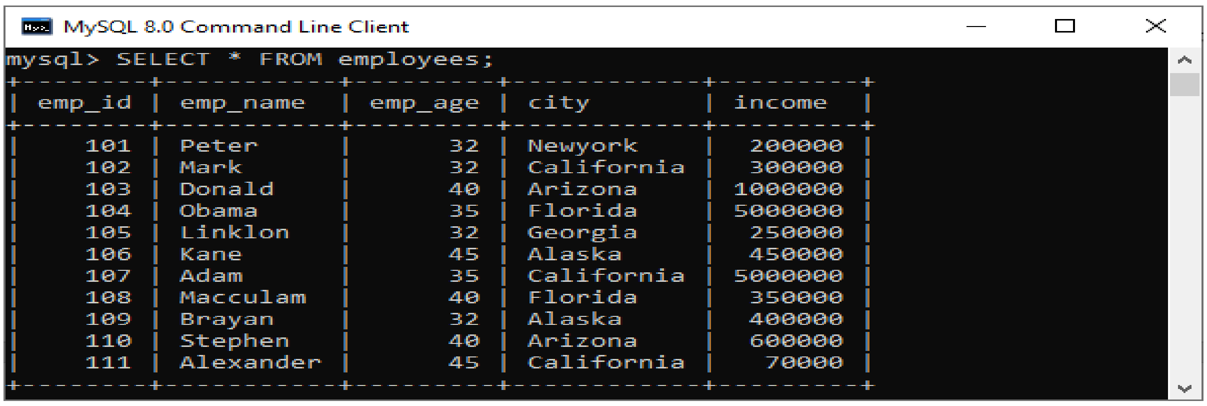
MySQL Subquery Syntax:

The following is the basic syntax to use the subquery in MySQL:

SELECT column_list (s) **FROM** table_name
WHERE column_name OPERATOR
(**SELECT** column_list (s) **FROM** table_name [**WHERE**])

MySQL Subquery Example:

Let us understand it with the help of an example. Suppose we have a table named **"employees"** that contains the following data:


MySQL 8.0 Command Line Client

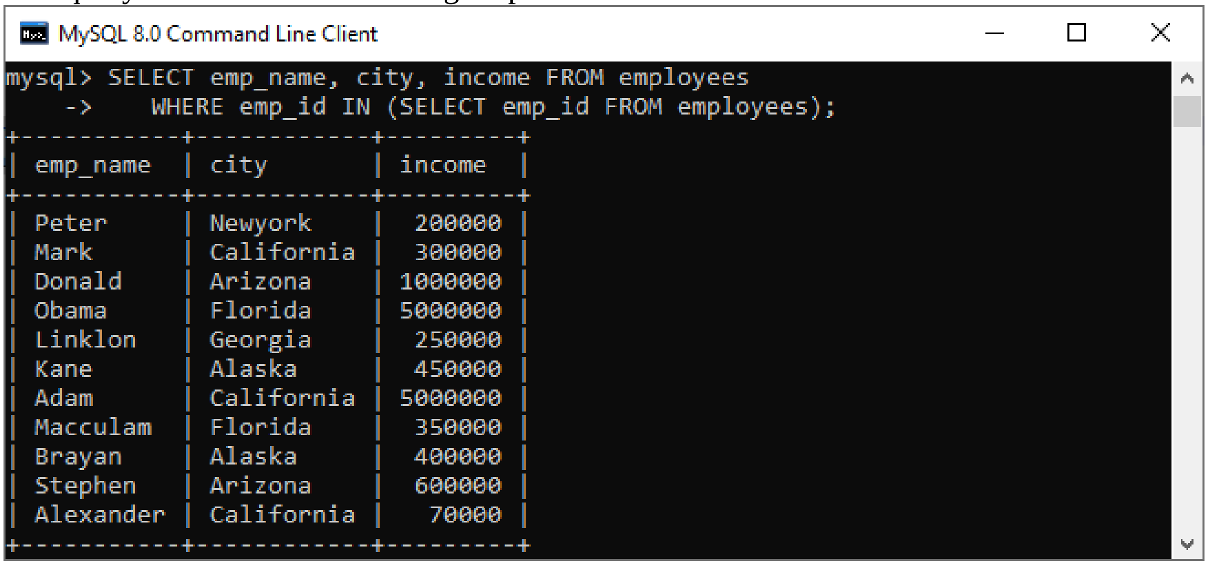
```
mysql> SELECT * FROM employees;
+--------+-----------+---------+------------+---------+
| emp_id | emp_name  | emp_age | city       | income  |
+--------+-----------+---------+------------+---------+
|    101 | Peter     |      32 | Newyork    |  200000 |
|    102 | Mark      |      32 | California |  300000 |
|    103 | Donald    |      40 | Arizona    | 1000000 |
|    104 | Obama     |      35 | Florida    | 5000000 |
|    105 | Linklon   |      32 | Georgia    |  250000 |
|    106 | Kane      |      45 | Alaska     |  450000 |
|    107 | Adam      |      35 | California | 5000000 |
|    108 | Macculam  |      40 | Florida    |  350000 |
|    109 | Brayan    |      32 | Alaska     |  400000 |
|    110 | Stephen   |      40 | Arizona    |  600000 |
|    111 | Alexander |      45 | California |   70000 |
+--------+-----------+---------+------------+---------+
```

Following is a simple SQL statement that returns the **employee detail whose id matches in a subquery**:

SELECT emp_name, city, income **FROM** employees

WHERE emp_id IN (**SELECT** emp_id **FROM** employees);

This query will return the following output:


MySQL 8.0 Command Line Client

```
mysql> SELECT emp_name, city, income FROM employees
    ->     WHERE emp_id IN (SELECT emp_id FROM employees);
+-----------+------------+---------+
| emp_name  | city       | income  |
+-----------+------------+---------+
| Peter     | Newyork    |  200000 |
| Mark      | California |  300000 |
| Donald    | Arizona    | 1000000 |
| Obama     | Florida    | 5000000 |
| Linklon   | Georgia    |  250000 |
| Kane      | Alaska     |  450000 |
| Adam      | California | 5000000 |
| Macculam  | Florida    |  350000 |
| Brayan    | Alaska     |  400000 |
| Stephen   | Arizona    |  600000 |
| Alexander | California |   70000 |
+-----------+------------+---------+
```

MySQL Subquery with Comparison Operator

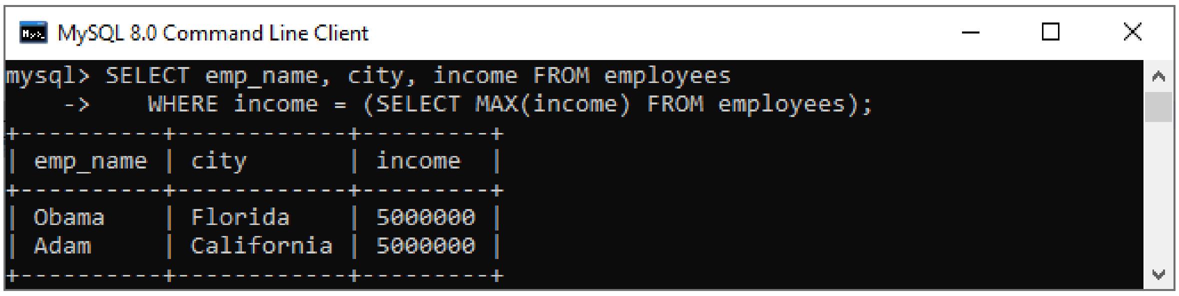
A comparison operator is an operator used to compare values and returns the result, either true or false. The following comparison operators are used in MySQL <, >, =, <>, <=>, etc. We can use the subquery before or after the comparison operators that return a single value. The returned value can be the arithmetic expression or a column function. After that, SQL compares the subquery results with the value on the other side of the comparison operator. The below example explains it more clearly:

Let us see an example of another comparison operator, such as equality (=) to find employee details with **maximum income** using a subquery.

SELECT emp_name, city, income **FROM** employees

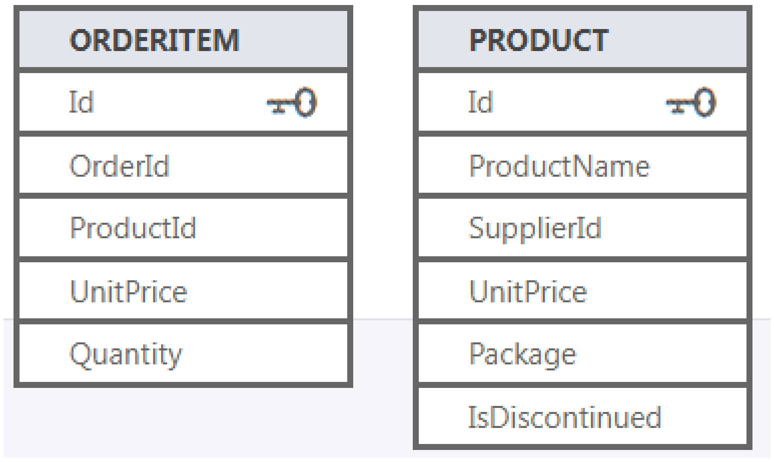
WHERE income = (**SELECT MAX**(income) **FROM** employees);

It will give the output where we can see two employees detail who have maximum income.


MySQL 8.0 Command Line Client

```
mysql> SELECT emp_name, city, income FROM employees
    ->     WHERE income = (SELECT MAX(income) FROM employees);
+----------+------------+---------+
| emp_name | city       | income  |
+----------+------------+---------+
| Obama    | Florida    | 5000000 |
| Adam     | California | 5000000 |
+----------+------------+---------+
```

MySQL Subquery with IN or NOT-IN Operator
If the subquery produces more than one value, we need to use the IN or NOT IN operator with the WHERE clause. Suppose we have a table named **"Student"** and **"Student2"** that contains the following data:



Problem: List products with order quantities greater than 100.
SELECT ProductName
FROM Product
WHERE Id IN (SELECT ProductId
FROM OrderItem
WHERE Quantity > 100)

Subqueries with INSERT statement
After the SELECT statement, the second-best option to use subqueries is with INSERT statements. The selected data in the subquery can be further modified through any characters, number functions, etc.
The INSERT statement uses data returned from the subquery to enter into another table.

Syntax:
INSERT INTO table_name
SELECT *
FROM table_name
WHERE VALUE OPERATOR

To understand more clearly let's take the above mentioned Employee table and also consider that a new employee table i.e. EmployeeNew is available in the database and run the below mentioned query and observe the output.

Example:
INSERT INTO EmployeeNew
SELECT * FROM Employee
WHERE emp_id IN (
SELECT emp_id
FROM Employee);

The final result of the above query i.e. the EmployeeNew table would look like below:

EmployeeNew:

```
121 | Lisa Carol | 27 | Finance |30000 | USA |
122 | Farooq Shaikh | 29 | HR| 20000 | USA |
123 | Nastya Henry | 28 | Technology | 45000 | UK |
124 | Christine Maybach | 30 | Research | 50000 | USA|
125 | Ryan Renolds | 27 | Research | 50000 | Canada |
```

Here, the EmployeeNew table is empty and the above query copies all the columns from the Employee table to the EmployeeNew table. Let's break the query statements to understand clearly.
The INSERT statement specifies that the new data is added to this table i.e. EmployeeNew table. As we are copying all the data from the Employee table to the EmployeeNew table, there is no need to specify the column name in the INSERT statement otherwise you would need to mention the column name in which you want to add the data.

Subqueries with UPDATE Statement
The UPDATE statement is used to modify an existing record within the table. To benefit from the usage of subqueries the WHERE clause should be used with the UPDATE statement otherwise the overall functioning of the table may get disturbed. In the UPDATE statement, the two clauses in which subqueries are most commonly used are SET and WHERE.

Syntax:

```
UPDATE Employee
SET column_name = new_value
WHERE Operator (
SELECT COLUMN_NAME
FROM TABLE_NAME
WHERE ... );
```

To understand more clearly let's take the above mentioned Employee table and also consider that a new employee table i.e. EmployeeNew is available in the database and run the below mentioned query and observe the output.

Example:

```
UPDATE Employee
SET emp_salary = emp_salary * 2
WHERE emp_age IN (
SELECT emp_age
FROM EmployeeNew
WHERE emp_age > 27 );
```

The final result of the above query would look like the below:
Employee:

```
| 121 | Lisa Carol | 27 | Finance | 30000 | USA |
| 122 | Farooq Shaikh | 29 | HR | 40000 | USA |
| 123 | Nastya Henry | 28 | Technology | 90000 | UK |
| 124 | Christine Maybach | 30 | Research | 100000|USA |
| 125 | Ryan Renolds | 27 | Research | 50000 | Canada |**
```

In this example, the Employee table is updated using UPDATE and SET. Here, the SET clause defines the new value for the emp_salary column which is being modified by the

UPDATE statement. As you can see in the subquery, the employees having age greater than 27 are filtered out and this result set is passed on to the outer query. The outer query updates the **emp_salary** column while considering the result set of the subquery i.e. salary of all the Employees having an age greater than 27 will increase by two times as mentioned in the outer query.

Subqueries with DELETE Statement
The subqueries can also be used with the DELETE statements to delete one or multiple records from the table.
Syntax:

```
DELETE FROM TableName
WHERE Operator (
SELECT COLUMN_NAME
FROM TABLE_NAME
WHERE condition );
```

To understand more clearly let's take the above mentioned Employee table and also consider that a new employee table i.e. EmployeeNew is available in the database that is mentioned above in the article and run the below mentioned query and observe the output.

```
DELETE FROM Employee
WHERE emp_dept NOT IN (
SELECT emp_dept
FROM EmployeeNew
WHERE emp_dept = 'Research' );
```

The final result of the above query i.e. the Employee table would look like the below:

```
| 121 | Lisa Carol | 27 | Finance | 30000 | USA |
| 122 | Farooq Shaikh | 29 | HR | 20000 | USA |
| 123 | Nastya Henry | 28 | Technology | 45000 | UK |
```

In this example, the DELETE statement is used to delete the rows from the Employee table with the help of EmployeeNew table. Firstly, the subquery is executed and the employees from the EmployeeNew table whose department is "Research" are fetched out and this result set is passed on to the outer query.

The outer query then deletes the records from the Employee table who is in the Research department. Note that not all the records that are present in the EmployeeNew and Employee table are deleted but only those records or employees who are in the Research department are deleted.

Practice queries for subquery

```
create table customer(cid number primary key,nm varchar(10),occupation varchar(15),age number);

create table orders(oid number primary key,cid number,constraint fine foreign key(cid) references customer(cid),prod_nm varchar(15),order_date date);

insert into customer values(101,'Peter','Engineer',32);
insert into customer values(102,'Joseph','Developer',30);
insert into customer values(103,'John','Leader',28);
insert into customer values(104,'Stephen','Scientist',45);
insert into customer values(105,'Suzi','Carpenter',26);
insert into customer values(106,'Bob','Actor',25);
insert into customer values(107,null,null,null);

insert into orders values(1,101,'Laptop','10-jan-2020');
insert into orders values(2,103,'Desktop','10-jan-2020');
insert into orders values(3,106,'Iphone','10-jan-2020');
insert into orders values(4,104,'Mobile','10-jan-2020');
insert into orders values(5,102,'TV','10-jan-2020');
```

1) Fetch name of customers who had placed oreder --

```
select nm from customer where cid in (select cid from orders);
```

2) Fetch name of customers who had not placed oreder --

```
select nm from customer where cid not in (select cid from orders);

create table empdetails(empid number(4) primary key, empname varchar2(15), gender varchar(8), empage varchar(8), city varchar(20), income number(10));

create table deptdetails(empid number(4), dept varchar2(20));

insert into empdetails values(101,'Jared','male',24,'surat',42000);
insert into empdetails values(102,'Cal','female',32,'vapi',54000);
insert into empdetails values(103,'Saanvi','female',33,'bharuch',25000);
insert into empdetails values(104,'Zeke','male',28,'surat',47000);
insert into empdetails values(105,'Robert','male',30,'vapi',58000);

insert into deptdetails values(101,'Production');
insert into deptdetails values(102,'Marketing');
insert into deptdetails values(103,'Production');
insert into deptdetails values(104,'Accounting');
insert into deptdetails values(105,'Accounting');
```

1)
```
SELECT empname, city, income FROM empdetails
WHERE empid IN (SELECT empid FROM empdetails);
```

2)
```
SELECT * FROM empdetails WHERE empid IN
(SELECT empid FROM empdetails WHERE income > 350000);
```

3)
```
SELECT empname,city,income FROM empdetails WHERE income =
(SELECT MAX(income) FROM empdetails);
```

4)
```
SELECT empname,dept FROM empdetails,deptdetails WHERE
empdetails.empid=deptdetails.empid and income =
(SELECT MAX(income) FROM empdetails);
```

5) insert into deptdetails (empid) select empid from empdetails where empname='Krishna';

6) update deptdetails set empid=(select empid from empdetails where empid=105) where empid=106;

7) delete from deptdetails where empid not in (select empid from empdetails);