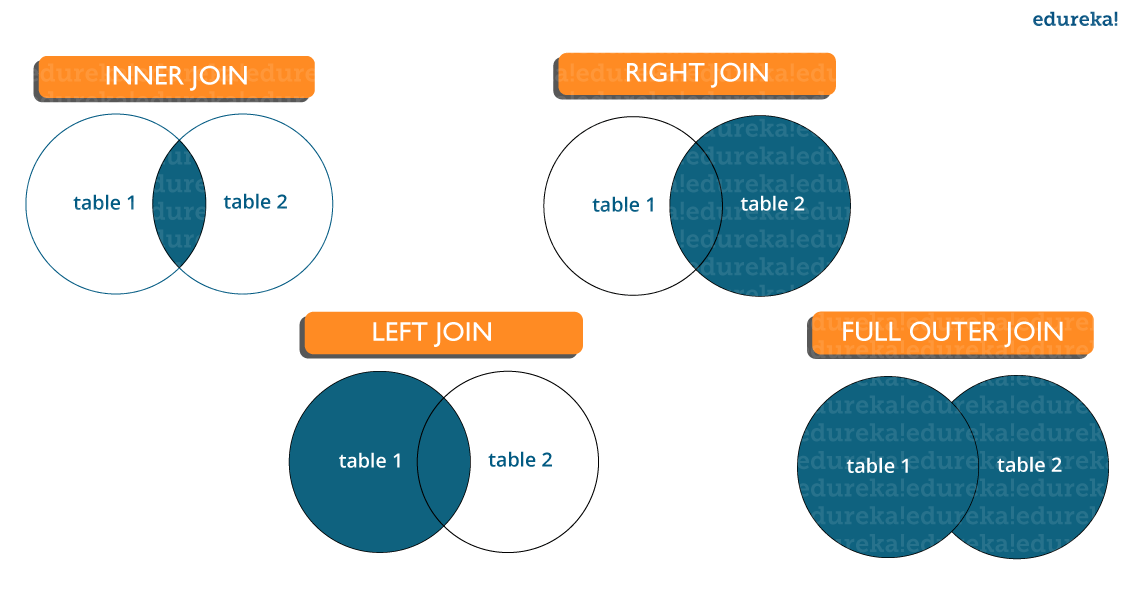
# **What are joins in SQL?**

A JOIN clause is used to combine rows from two or more tables, based on a related column between them. It is used to merge two tables or retrieve data from there. There are 4 types of joins, as you can refer to below:

****

* **Inner join:** [Inner Join in SQL](https://www.edureka.co/blog/sql-joins-types) is the most common type of join. It is used to return all the rows from multiple tables where the join condition is satisfied.
* **Left Join:** Left Join in SQL is used to return all the rows from the left table but only the matching rows from the right table where the join condition is fulfilled.
* **Right Join:** Right Join in SQL is used to return all the rows from the right table but only the matching rows from the left table where the join condition is fulfilled.
* **Full Join:** Full join returns all the records when there is a match in any of the tables. Therefore, it returns all the rows from the left-hand side table and all the rows from the right-hand side table.

# **What is a Primary key?**

* A[Primary key in SQL](https://www.edureka.co/blog/primary-key-in-sql/)is a column (or collection of columns) or a set of columns that uniquely identifies each row in the table.
* Uniquely identifies a single row in the table
* Null values not allowed

# **What are Constraints?**

[Constraints in SQL](https://www.edureka.co/blog/sql-constraints/) are used to specify the limit on the data type of the table. It can be specified while creating or altering the table statement. The sample of constraints are:

1. NOT NULL
2. CHECK
3. DEFAULT
4. UNIQUE
5. PRIMARY KEY
6. FOREIGN KEY

# **What is a Unique key?**

* Uniquely identifies a single row in the table.
* Multiple values allowed per table.
* Null values allowed.

# **What is a Foreign key in SQL?**

* Foreign key maintains referential integrity by enforcing a link between the data in two tables.
* The foreign key in the child table references the primary key in the parent table.
* The [foreign key constraint](https://www.edureka.co/blog/foreign-key-sql/) prevents actions that would destroy links between the child and parent tables.

# **What do you mean by data integrity?**

Data Integrity defines the accuracy as well as the consistency of the data stored in a database. It also defines integrity constraints to enforce business rules on the data when it is entered into an application or a database

**Difference between Locking, Blocking and Deadlocking**

* + **Locking:** Locking occurs when a connection needs access to a piece of data in a database and it locks it for certain use so that no other transaction is able to access it.
  + **Blocking:** Blocking occurs when a transaction tries to acquire an incompatible lock on a resource that another transaction has already locked. The blocked transaction remains blocked until the blocking transaction releases the lock.
  + **Deadlocking:** Deadlocking occurs when two or more transactions have a resource locked, and each transaction requests a lock on the resource that another transaction has already locked. Neither of the transactions here can move forward, as each one is waiting for the other to release the lock.

**What special operators does Oracle provide for dealing with NULLs?**  
  
NVL - Converts a NULL to another specified value, as in:  
my\_var := NVL (your\_var, 'Hello');  
IS NULL and IS NOT NULL  
You can use this syntax to check specificaly to see if a variable's value is NULL or NOT NULL.  
  
**Explain three different rules that apply to NULLs when doing comparisons?**

1. For all operators except for concatenation (||), if a value in an expression is a NULL, that expression evaluates to NULL  
2. NULL is never equal or not equal to another value  
3. NULL is never TRUE or FALSE

# **what is mutating error?when we go to mutating error in oracle?**

It occurs when ever we try to perform either select or dml operations on the same table in the trigger created for same table.

# **Define UNION, MINUS, UNION ALL, INTERSECT?**

**Answer:**

* **MINUS** – returns all distinct rows selected by the first query but not by the second.
* **UNION** – returns all distinct rows selected by either query
* **UNION ALL** – returns all rows selected by either query, including all duplicates.
* **INTERSECT** – returns all distinct rows selected by both queries.

# **Difference between TRUNCATE, DELETE and DROP commands?**

**Answer:**

* **DELETE** removes some or all rows from a table based on the condition. It can be rolled back.
* **TRUNCATE** removes ALL rows from a table by de-allocating the memory pages. The operation cannot be rolled back
* **DROP** command removes a table from the database completely.

# **What do we need to check in Database Testing?**

**Answer: In Database testing, the following thing is required to be tested:**

* Database connectivity
* Constraint check
* Required application field and its size
* Data Retrieval and processing with DML operations
* Stored Procedures
* Functional flow

# **What is Normalization? How many Normalization forms are there?**

**Answer:** Normalization is used to organize the data in such a manner that data redundancy will never occur in the database and avoid insert, update and delete anomalies.

**There are 5 forms of Normalization:**

* **First Normal Form (1NF):** It removes all duplicate columns from the table. It creates a table for related data and identifies unique column values.
* **First Normal Form (2NF):** Follows 1NF and creates and places data subsets in an individual table and defines the relationship between tables using the primary key.
* **Third Normal Form (3NF):** Follows 2NF and removes those columns which are not related through the primary key.
* **Fourth Normal Form (4NF):** Follows 3NF and does not define multi-valued dependencies. 4NF is also known as BCNF.

# **What is the difference between Nested Subquery and Correlated Subquery?**

**Answer:** Subquery within another subquery is called Nested Subquery. If the output of a subquery depends on column values of the parent query table then the query is called Correlated Subquery.

|  |
| --- |
| SELECT adminid(SELEC Firstname+' '+Lastname&nbsp;&nbsp;FROM Employee WHERE  empid=emp. adminid)AS EmpAdminId FROM Employee; |

The result of the query is the details of an employee from the Employee table.

# **What is SQL Sandbox in SQL Server?**

**Answer:** SQL Sandbox is a safe place in the SQL server environment where untrusted scripts are executed. There are 3 types of SQL sandbox:

* **Safe Access Sandbox:** Here a user can perform SQL operations such as creating stored procedures, triggers etc. but cannot have access to the memory as well as cannot create files.
* **External Access Sandbox:** Users can access files without having the right to manipulate the memory allocation.
* **Unsafe Access Sandbox:** This contains untrusted codes where a user can have access to memory.

# **How many types of Privileges are available in SQL?**

**Answer: There are two types of privileges used in SQL, such as**

* **System privilege:** System privilege deals with the object of a particular type and provides users the right to perform one or more actions on it. These actions include performing administrative tasks, ALTER ANY INDEX, ALTER ANY CACHE GROUP CREATE/ALTER/DELETE TABLE, CREATE/ALTER/DELETE VIEW etc.
* **Object privilege:** This allows to perform actions on an object or object of another user(s) viz. table, view, indexes etc. Some of the object privileges are EXECUTE, INSERT, UPDATE, DELETE, SELECT, FLUSH, LOAD, INDEX, REFERENCES etc.

# **What are transactions and their controls?**

**Answer:** A transaction can be defined as the sequence task that is performed on databases in a logical manner to gain certain results. Operations like Creating, updating, deleting records performed in the database come from transactions.

In simple words, we can say that a transaction means a group of SQL queries executed on database records.

**There are 4 transaction controls such as**

* **COMMIT**: It is used to save all changes made through the transaction.
* **ROLLBACK**: It is used to roll back the transaction. All changes made by the transaction are reverted back and the database remains as before.
* **SET TRANSACTION**: Set the name of the transaction.
* **SAVEPOINT:** It is used to set the point where the transaction is to be rolled back.

# **What are aggregate and scalar functions?**

Aggregate functions are used to evaluate mathematical calculation and return single values. This can be calculated from the columns in a table. Scalar functions return a single value based on the input value.

Example -.

Aggregate – max(), count - Calculated with respect to numeric.

Scalar – UCASE(), NOW() – Calculated with respect to strings.

# **What is Online Transaction Processing (OLTP)?**

Online Transaction Processing (OLTP) manages transaction based applications which can be used for data entry, data retrieval and data processing. OLTP makes data management simple and efficient. Unlike OLAP systems goal of OLTP systems is serving real-time transactions.

Example – Bank Transactions on a daily basis.

# **Improve SQL Query Performance**

1. **Use EXISTS** instead of IN to check existence of data.

## Nullable Columns

Avoid using NOT IN when comparing with nullable columns. Use NOT EXISTS instead.

**Reason:** When NOT IN is used in the query (even if the query doesn’t return rows with the null values), SQL Server will check each result to see if it’s null or not. Using NOT EXISTS will not do the comparison with nulls. Also, NOT EXISTS returns only two states (true or false), while NOT IN can return up to three states (true, false, NULL) and might not give you the result you were expecting.

2. **Avoid \* in SELECT statement**. Give the name of columns which you require.  
3. **Choose appropriate Data Type.** E.g. To store strings use varchar in place of text data type. Use text data type, whenever you need to store large data (more than 8000 characters).  
4. **Avoid nchar and nvarchar** if possible since both the data types takes just double memory as char and varchar.  
5. **Avoid NULL in fixed-length field**. In case of requirement of NULL, use variable-length (varchar) field that takes less space for NULL.  
6. **Avoid Having Clause.** Having clause is required if you further wish to filter the result of an aggregations.  
7. **Create Clustered and Non-Clustered Indexes**.  
8. **Keep clustered index** small since the fields used in clustered index may also used in non-clustered index.  
9. **Most selective columns** should be placed leftmost in the key of a non-clustered index.  
10. **Drop unused Indexes**.  
11. **Better to create indexes on columns that have integer values** instead of characters. Integer values use less overhead than character values.  
12. **Use joins** instead of sub-queries.  
13. **Use WHERE** expressions to limit the size of result tables that are created with joins.  
14. **Use TABLOCKX** while inserting into a table and TABLOCK while merging.  
15. **Use WITH (NOLOCK)** while querying the data from any table.  
16. **Use SET NOCOUNT ON** and use TRY- CATCH to avoid deadlock condition.  
17. **Avoid Cursors since** cursor are very slow in performance.  
18. **Use Table variable** in place of Temp table. Use of Temp tables required interaction with TempDb database which is a time taking task.  
19. **Use UNION ALL** in place of UNION if possible.  
20. **Use Schema name** before SQL objects name.  
21. **Use Stored Procedure** for frequently used data and more complex queries.  
22. **Keep transaction as small as possible** since transaction lock the processing tables data and may results into deadlocks.  
23. **Avoid prefix “sp\_”** with user defined stored procedure name because SQL server first search the user defined procedure in the master database and after that in the current session database.  
24. **Avoid use of Non-correlated Scalar Sub Query**. Use this query as a separate query instead of part of the main query and store the output in a variable, which can be referred to in the main query or later part of the batch.  
25. **Avoid Multi-statement Table Valued Functions (TVFs).** Multi-statement TVFs are more costly than inline TVFs.

26. **Use Same Datatype on JOIN** and WHERE Clauses

This is easier said than done depending on your permissions to make changes to the schema.

**Reason:** When joining or comparing two fields with different datatypes, SQL must do an on-the-fly conversion of the field before it can do a comparison, even if the fields are indexed. If mismatched datatypes are unavoidable, try to cast the larger datatype to the smaller datatype whenever possible.

## Avoid Multiple Joins in a Single Query:

Try to avoid writing a query using multiple joins that include outer joins, cross apply, outer apply and other complex subqueries.

**Reason:** When the joins are complex, the number of possible execution plans exponentially increases. This, in turn, will reduce the choices for the query optimizer to decide the join order and join type. Sometimes, the query optimizer is forced to use nested loop joins, irrespective of the performance consequences for queries with excessively complex cross apply or subqueries. Also, due to time and performance constraint, SQL Engine will try to generate the “good enough” execution plan versus trying to generate the best possible execution plan.

## **Check Indexes**

There should be indexes on all fields used in the WHERE and JOIN portions of the SQL statement.

**Reason:** When fields are not indexed, SQL Server will typically do a full table scan and this may reduce performance. Unless the table is very small, a table scan tends to yield the worst performance out of all the types of database reads.

## **Avoid Using GROUP BY, ORDER BY and DISTINCT**

Avoid using GROUP BY, ORDER BY, and DISTINCT whenever possible.

**Reason:** When using GROUP BY, ORDER BY, or DISTINCT, the SQL Server engine creates a work table and puts the data on the work table. After that, it organizes this data in the work table as requested by the query, and then it returns the final result.

## **Keep Wildcards at the End of Phrases**

Wildcards are useful for creating extremely wide searches, which also runs the risk of being inefficient. Pairing a leading wildcard with an ending wildcard is especially so, as it will query the database for all records for text matching what’s between the two wildcards. If searching for customer names, for instance, %Son% would retrieve both “Sonia” and “Richardson,” whereas Son% would only return “Sonia.”

# **SQL Query**

**Table – EmployeeDetails**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EmpId** | **FullName** | **ManagerId** | **DateOfJoining** | **City** |
| 121 | John Snow | 321 | 01/31/2014 | Toronto |
| 321 | Walter White | 986 | 01/30/2015 | California |
| 421 | Kuldeep Rana | 876 | 27/11/2016 | New Delhi |

**Table – EmployeeSalary**

|  |  |  |  |
| --- | --- | --- | --- |
| **EmpId** | **Project** | **Salary** | **Variable** |
| 121 | P1 | 8000 | 500 |
| 321 | P2 | 10000 | 1000 |
| 421 | P1 | 12000 | 0 |

**Write an SQL query to fetch the different projects available from the EmployeeSalary table.**

**SELECT DISTINCT(Project)**

**FROM EmployeeSalary;**

**Write an SQL query to fetch the count of employees working in project ‘P1’.**

**SELECT COUNT(\*)**

**FROM EmployeeSalary**

**WHERE Project = 'P1';**

**Write an SQL query to find the maximum, minimum, and average salary of the employees.**

**SELECT Max(Salary),**

**Min(Salary),**

**AVG(Salary)**

**FROM EmployeeSalary;**

**Write an SQL query to find the employee id whose salary lies in the range of 9000 and 15000.**

**SELECT EmpId, Salary**

**FROM EmployeeSalary**

**WHERE Salary BETWEEN 9000 AND 15000;**

**Write an SQL query to fetch those employees who live in Toronto and work under manager with ManagerId – 321.**

**SELECT EmpId, City, ManagerId**

**FROM EmployeeDetails**

**WHERE City='Toronto' AND ManagerId='321';**

**Write an SQL query to fetch all those employees who work on Project other than P1.**

**SELECT EmpId**

**FROM EmployeeSalary**

**WHERE NOT Project='P1';**

Or

**SELECT EmpId**

**FROM EmployeeSalary**

**WHERE Project <> 'P1';**

**Write an SQL query to fetch all the EmpIds which are present in either of the tables – ‘EmployeeDetails’ and ‘EmployeeSalary’.**

**SELECT EmpId FROM EmployeeDetails**

**UNION**

**SELECT EmpId FROM EmployeeSalary;**

**Write an SQL query to fetch common records between two tables.**

**SELECT \* FROM EmployeeSalary**

**INTERSECT**

**SELECT \* FROM ManagerSalary;**

Or

**SELECT \***

**FROM EmployeeSalary**

**WHERE EmpId IN**

**(SELECT EmpId from ManagerSalary);**

**Write an SQL query to fetch records that are present in one table but not in another table.**

**SELECT \* FROM EmployeeSalary**

**MINUS**

**SELECT \* FROM ManagerSalary;**

Or

**SELECT EmployeeSalary.\***

**FROM EmployeeSalary**

**LEFT JOIN**

**ManagerSalary USING (EmpId)**

**WHERE ManagerSalary.EmpId IS NULL;**

**Write an SQL query to fetch the employee full names and replace the space with ‘-’.**

**SELECT REPLACE(FullName, ' ', '-')**

**FROM EmployeeDetails;**

**Write an SQL query to update the employee names by removing leading and trailing spaces.**

**UPDATE EmployeeDetails**

**SET FullName = LTRIM(RTRIM(FullName));**

**Fetch all the employees who are not working on any project.**

**SELECT EmpId**

**FROM EmployeeSalary**

**WHERE Project IS NULL;**

**Write an SQL query to find the current date-time.**

**MySQL:- SELECT NOW();**

**SQL Server- SELECT getdate();**

**Oracle-SELECT SYSDATE FROM DUAL;**

**Write an SQL query to fetch all the Employees details from EmployeeDetails table who joined in the Year 2020.**

**SELECT \* FROM EmployeeDetails**

**WHERE DateOfJoining BETWEEN '2020/01/01'**

**AND '2020/12/31';**

Or

**SELECT \* FROM EmployeeDetails**

**WHERE YEAR(DateOfJoining) = '2020';**

**Write an SQL query to fetch all the Employees who are also managers from the EmployeeDetails table.**

**we have to use Self-Join as the requirement wants us to analyze the EmployeeDetails table as two tables. We will use different aliases ‘E’ and ‘M’ for the same EmployeeDetails table.**

**SELECT DISTINCT E.FullName**

**FROM EmployeeDetails E**

**INNER JOIN EmployeeDetails M**

**ON E.EmpID = M.ManagerID;**

**Write an SQL query to fetch duplicate records from EmployeeDetails (without considering the primary key – EmpId).**

**SELECT FullName, ManagerId, DateOfJoining, City, COUNT(\*)**

**FROM EmployeeDetails**

**GROUP BY FullName, ManagerId, DateOfJoining, City**

**HAVING COUNT(\*) > 1;**

**Write an SQL query to remove duplicates from a table without using a temporary table.**

**DELETE E1 FROM EmployeeDetails E1**

**INNER JOIN EmployeeDetails E2**

**WHERE E1.EmpId > E2.EmpId**

**AND E1.FullName = E2.FullName**

**AND E1.ManagerId = E2.ManagerId**

**AND E1.DateOfJoining = E2.DateOfJoining**

**AND E1.City = E2.City;**

**Write an SQL query to fetch only odd rows from the table.**

**SELECT \* FROM EmployeeDetails**

**WHERE MOD (EmpId, 2) <> 0;**

**Write an SQL query to create a new table with data and structure copied from another table.**

**CREATE TABLE NewTable**

**SELECT \* FROM EmployeeSalary;**

**Write an SQL query to create an empty table with the same structure as some other table.**

**CREATE TABLE NewTable**

**SELECT \* FROM EmployeeSalary where 1=0;**

**Write an SQL query to find the nth highest salary from table.**  
Ans, Using Top keyword (SQL Server)-

**SELECT TOP 1 Salary**

**FROM (**

**SELECT DISTINCT TOP N Salary**

**FROM Employee**

**ORDER BY Salary DESC**

**)**

**ORDER BY Salary ASC;**

Using limit clause(MySQL)-

**SELECT Salary**

**FROM Employee**

**ORDER BY Salary DESC LIMIT N-1,1;**

**Write SQL query to find the 3rd highest salary from a table without using the TOP/limit keyword.**

**SELECT Salary**

**FROM EmployeeSalary Emp1**

**WHERE 2 = (**

**SELECT COUNT( DISTINCT ( Emp2.Salary ) )**

**FROM EmployeeSalary Emp2**

**WHERE Emp2.Salary > Emp1.Salary**

**)**

For nth highest salary-

**SELECT Salary**

**FROM EmployeeSalary Emp1**

**WHERE N-1 = (**

**SELECT COUNT( DISTINCT ( Emp2.Salary ) )**

**FROM EmployeeSalary Emp2**

**WHERE Emp2.Salary > Emp1.Salary**

**)**

#### **Write an SQL query to show the second highest salary from a table.**

**Select max(Salary) from Worker**

**where Salary not in (Select max(Salary) from Worker);**

#### **Write an SQL query to show the top n (say 10) records of a table.**

**Ans.**

Following MySQL query will return the top n records using the LIMIT method:

**SELECT \* FROM Worker ORDER BY Salary DESC LIMIT 10;**

Following SQL Server query will return the top n records using the TOP command:

**SELECT TOP 10 \* FROM Worker ORDER BY Salary DESC;**

Following Oracle query will return the top n records with the help of ROWNUM:

**SELECT \* FROM (SELECT \* FROM Worker ORDER BY Salary DESC)**

**WHERE ROWNUM <= 10;**

#### **Write an SQL query to show one row twice in results from a table.**

**Ans.**

The required query is:

**select FIRST\_NAME, DEPARTMENT from worker W where W.DEPARTMENT='HR'**

**union all**

**select FIRST\_NAME, DEPARTMENT from Worker W1 where W1.DEPARTMENT='HR';**

#### **Write an SQL query to fetch the departments that have less than five people in it.**

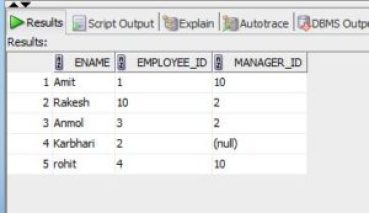
**SELECT DEPARTMENT, COUNT(WORKER\_ID) as 'Number of Workers' FROM Worker GROUP BY DEPARTMENT HAVING COUNT(WORKER\_ID) < 5;**

## Hierarchical Queries

There are 2 types of Hierarchical Queries in oracle: **Connect by ,Start with and prior clause :**

1.Basic hierarchical query

2. Cyclic hierarchical query

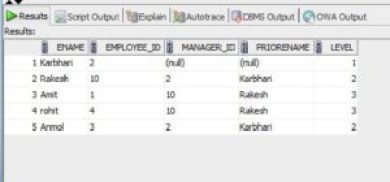


**Write the query which will gives us the records of the Employee and its manager?**

**To show the top to bottom hierarchy :**

select ename, employee\_id, manager\_id, prior ename, level  
from Employee\_M  
connect by prior employee\_id = manager\_id  
start with manager\_id is null;

**Output :**



In above example we need to start with the keyword prior,

1.PRIOR is a unary operator which is used or indicates that “father of” the records or first record.  
2.START WITH clause is used to find out from which record user needs to start the hierarchy. In above example we have started hierarchy with the employee whose manager id is null.  
3.LEVEL is a pseudocolumn which gives user the idea about the depth of the hierarchy.  
4.CONNECT BY gives user the idea about the first sibling record with the specified condition.

**Show the records in hierarchical structure**

**User can show the records into hierarchical structure using following query :**select lpad(‘ ‘,level\*5,’ ‘)||ename, employee\_id, manager\_id, prior ename, level  
from Employee\_M  
connect by prior employee\_id = manager\_id  
start with manager\_id is null  
order siblings by ename;

**Output :**

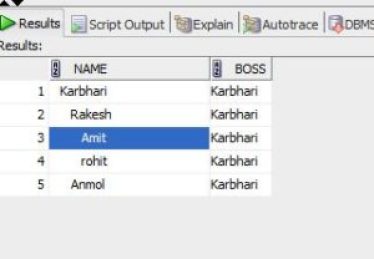


**Query using** [**connect\_by\_root**](https://docs.oracle.com/cd/B19306_01/server.102/b14200/queries003.htm)

This is another important unary operator in oracle which will gives the basic idea about the root of the hierarchy.If you want to see the boss’s hierarchy then user needs to use connect\_by\_root keyword .

select lpad(‘ ‘,level\*3,’ ‘)||ename name,  
connect\_by\_root ename boss  
from Employee\_M  
connect by prior employee\_id = manager\_id  
start with manager\_id is null;

**Output:**



**If user wants to see the bosses of employee**

In this case user needs to use keyword **‘SYS\_CONNECT\_BY\_PATH(column\_name,Delimiter)’**

**Query :**

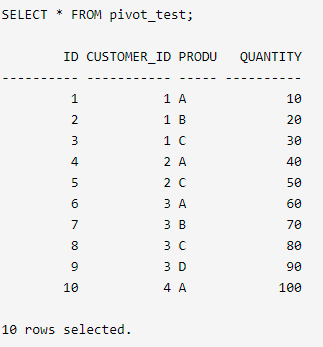
|  |  |
| --- | --- |
| select lpad(‘ ‘,level\*3,’ ‘)||ename name, SYS\_CONNECT\_BY\_PATH(ename,’/’) boss from Employee\_M connect by prior employee\_id = manager\_id start with manager\_id is null; | https://i1.wp.com/www.complexsql.com/wp-content/uploads/2017/11/Example-6-300x186.jpg?resize=331%2C205 |

**Output :**

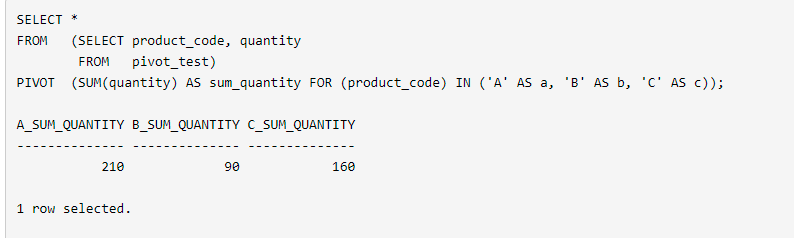
# **PIVOT and UNPIVOT Operators in Oracle db 11g Release**

## **PIVOT**

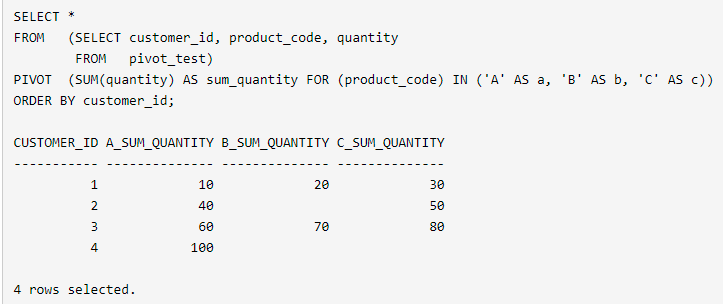
The PIVOT operator takes data in separate rows, aggregates it and converts it into columns. To see the PIVOT operator in action we need to create a test table.



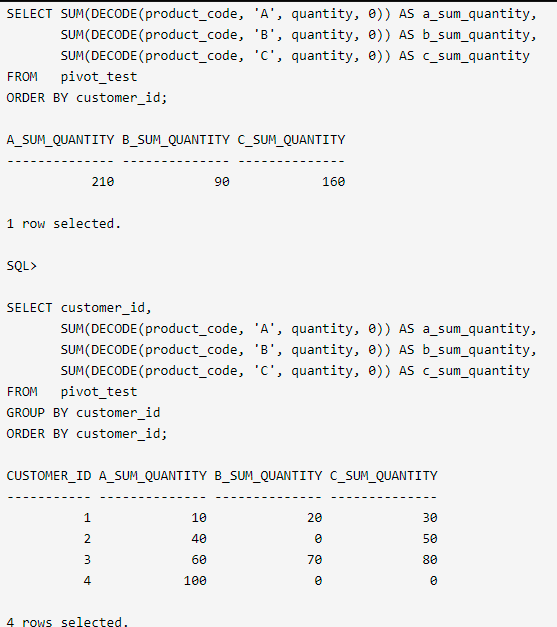
In its basic form the PIVOT operator is quite limited. We are forced to list the required values to PIVOT using the IN clause.



If we want to break it down by customer, we simply include the CUSTOMER\_ID column in the initial select list.

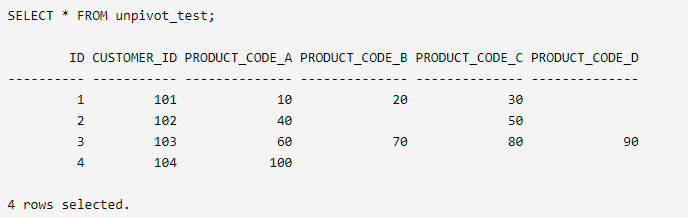


Prior to 11g we could accomplish a similar result using the DECODE function combined with aggregate functions.

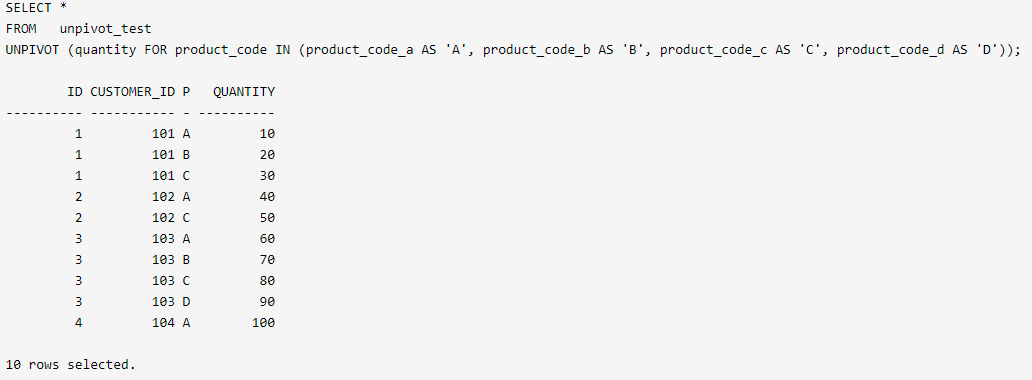


## **UNPIVOT**

The UNPIVOT operator converts column-based data into separate rows. To see the UNPIVOT operator in action we need to create a test table



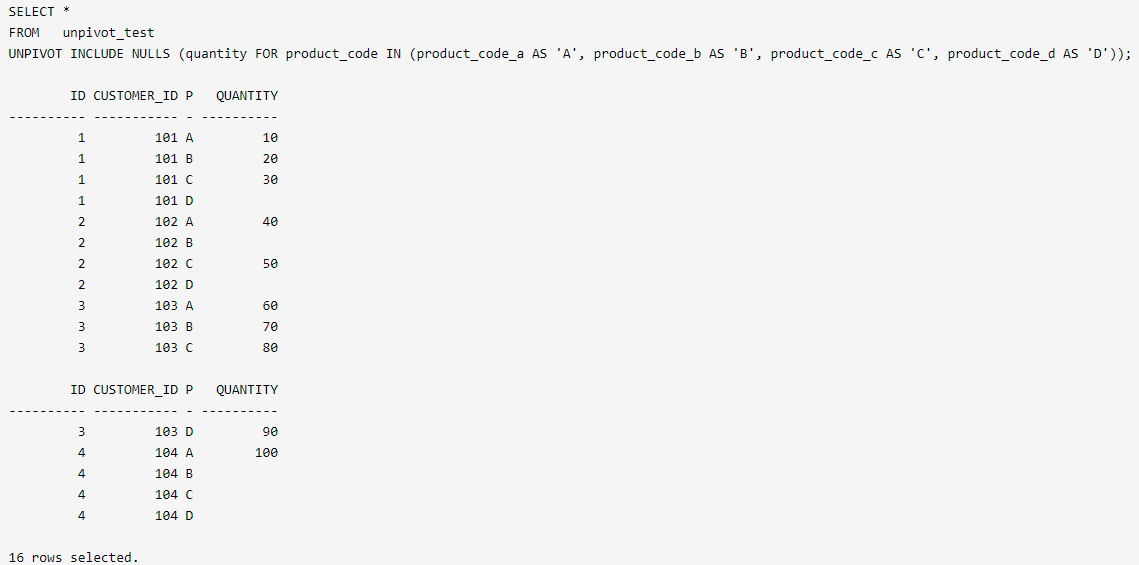
The UNPIVOT operator converts this column-based data into individual rows.



There are several things to note about the query:

* The required column names, in this case QUANTITY and PRODUCT\_CODE, are define in the UNPIVOT clause. These can be set to any name not currently in the driving table.
* The columns to be unpivoted must be named in the IN clause.
* The PRODUCT\_CODE value will match the column name it is derived from, unless you alias it to another value.
* By default the EXCLUDE NULLS clause is used. To override the default behaviour use the INCLUDE NULLS clause.

The following query shows the inclusion of the INCLUDE NULLS clause.



Prior to 11g, we can get the same result using the DECODE function and a pivot table with the correct number of rows. In the following example we use the CONNECT BY clause in a query from dual to generate the correct number of rows for the unpivot operation.

SELECT id,

customer\_id,

DECODE(unpivot\_row, 1, 'A',

2, 'B',

3, 'C',

4, 'D', 'N/A') AS product\_code,

DECODE(unpivot\_row, 1, product\_code\_a,

2, product\_code\_b,

3, product\_code\_c,

4, product\_code\_d, 'N/A') AS quantity

FROM unpivot\_test, (SELECT level AS unpivot\_row FROM dual CONNECT BY level <= 4)

ORDER BY 1,2,3;

ID CUSTOMER\_ID PRO QUANTITY

---------- ----------- --- ----------

1 101 A 10

1 101 B 20

1 101 C 30

1 101 D

2 102 A 40

2 102 B

2 102 C 50

2 102 D

3 103 A 60

3 103 B 70

3 103 C 80

3 103 D 90

4 104 A 100

4 104 B

4 104 C

# **Using the WITH clause to simplify complex SQL**

Oracle SQL can run faster when complex subqueries are replaced with global temporary tables. Starting in Oracle9i release 2, there was an incorporation of a subquery factoring utility implemented the SQL-99 WITH clause. The WITH clause is a tool for materializing subqueries to save Oracle from having to recompute them multiple times.

Use of the SQL WITH clause is very similar to the use of Global Temporary Tables (GTT), a technique that is often employed to improve query speed for complex subqueries. The following are some important notes about the Oracle WITH clause:

 The SQL WITH clause only works on Oracle 9i release 2 and beyond.

 Formally, the WITH clause was called subquery factoring.

 The SQL WITH clause is used when a subquery is executed multiple times.

 The ANSI WITH clause is also useful for recursive queries, but this feature has not yet been implemented in Oracle SQL.

WITH dept\_count AS (

SELECT deptno, COUNT(\*) AS dept\_count

FROM emp

GROUP BY deptno)

SELECT e.ename AS employee\_name,

dc.dept\_count AS emp\_dept\_count

FROM emp e

JOIN dept\_count dc ON e.deptno = dc.deptno;

**Q #90) What is CTE?**

**Answer:** A CTE or common table expression is an expression that contains temporary result set which is defined in a SQL statement.

**What is the difference between Local and Global temporary tables?**

**Answer:** If defined inside a compound statement a local temporary table exists only for the duration of that statement but a global temporary table exists permanently in the DB but its rows disappear when the connection is closed.

# **Oracle / PLSQL: LISTAGG Function**

Oracle/PLSQL LISTAGG function concatenates values of the measure\_column for each GROUP based on the order\_by\_clause.

The syntax for the LISTAGG function in Oracle/PLSQL is:

LISTAGG (measure\_column [, 'delimiter'])

WITHIN GROUP (order\_by\_clause) [OVER (query\_partition\_clause)]

Parameters or Arguments

**measure\_column**

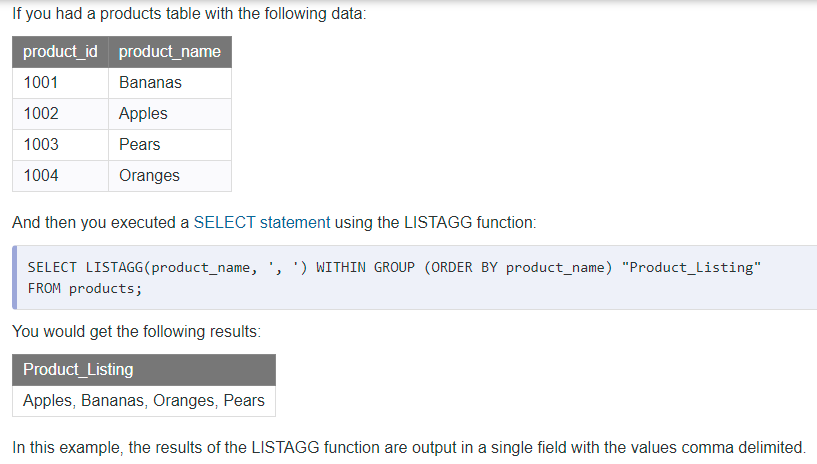
The column or expression whose values you wish to concatenate together in the result set. Null values in the *measure\_column* are ignored.

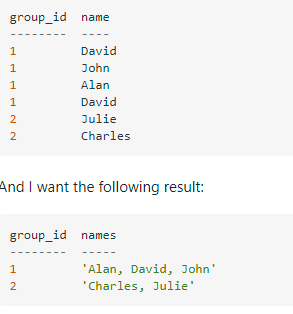
**delimiter**

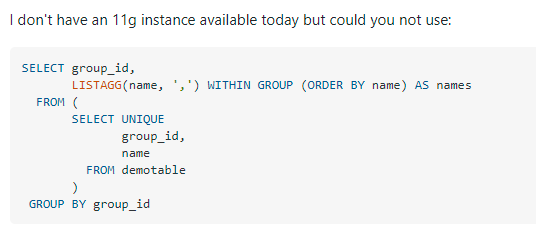
Optional. It is the delimiter to use when separating the *measure\_column* values when outputting the results.

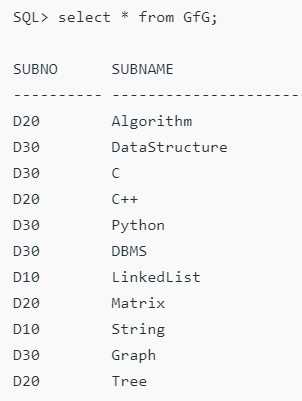
**order\_by\_clause**

It determines the order that the concatenated values (ie: *measure\_column*) are returned.









**Query 1:** Write an SQL query using LISTAGG function to output the subject names in a single field with the values comma delimited.

**SQL> SELECT LISTAGG(SubName, ' , ') WITHIN GROUP (ORDER BY SubName) AS SUBJECTS FROM GfG ;**

Output

SUBJECTS

---------------------------------------------------------------------------

**Algorithm , C , C++ , DBMS , DataStructure , Graph , LinkedList , Matrix , Python , String , Tree**

Write an SQL query to group each subject and show each subject in its respective department separated by comma with the help of LISTAGG function.

**SELECT SubNo, LISTAGG(SubName, ' , ') WITHIN GROUP (ORDER BY SubName) AS SUBJECTS FROM GfG GROUP BY SubNo;**

Output

**SUBNO SUBJECTS**

**------ ----------------------------------------------------------------**

**D10 LinkedList , String**

**D20 Algorithm , C++ , Matrix , Tree**

**D30 C , DBMS , DataStructure , Graph , Python**