**1.      To see current user name**   
 Sql> show user;

**4.      How do I eliminate the duplicate rows ?**   
 SQL> delete from table\_name where rowid not in (select max(rowid) from table group by duplicate\_values\_field\_name);   
or   
SQL> delete duplicate\_values\_field\_name dv from table\_name ta where rowid <(select min(rowid)  from table\_name tb where ta.dv=tb.dv);   
Example.   
Table Emp   
Empno Ename   
101               Scott   
102               Jiyo   
103               Millor   
104               Jiyo   
105               Smith   
delete ename from emp a where rowid < ( select min(rowid) from emp b where a.ename = b.ename);   
The output like,   
Empno Ename   
101               Scott   
102               Millor   
103               Jiyo   
104               Smith

**5.      How do I display row number with records?**   
To achive this use rownum pseudocolumn with query, like SQL> SQL> select rownum, ename from emp;   
Output:   
1                    Scott   
2                    Millor   
3                    Jiyo   
4                    Smith

**6.      Display the records between two range**   
select rownum, empno, ename  from emp  where  rowid in   
 (select rowid from emp where rownum <=&upto   
 minus   
 select rowid from emp where rownum<&Start);   
Enter value for upto: 10   
Enter value for Start: 7

   ROWNUM     EMPNO ENAME   
--------- --------- ----------   
        1      7782 CLARK   
        2      7788 SCOTT   
        3      7839 KING   
        4      7844 TURNER

**11.  Find out nth highest salary from emp table**   
 SELECT DISTINCT (a.sal) FROM EMP A WHERE &N = (SELECT COUNT (DISTINCT (b.sal)) FROM EMP B WHERE a.sal<=b.sal);

Enter value for n: 2   
      SAL   
---------   
     3700

**Can a view be updated/inserted/deleted? If Yes - under what conditions ?**

 A View can be updated/deleted/inserted if it has only one base table if the view is based on columns from one or more tables then insert, update and delete is not possible.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | **Explain the difference between a FUNCTION, PROCEDURE and PACKAGE ?**  A function and procedure are the same in that they are intended to be a collection of PL/SQL code that carries a single task. While a procedure does not have to return any values to the calling application, a function will return a single value. A package on the other hand is a collection of functions and procedures that are grouped together based on their commonality to a business function or application.   |  | | --- | | **Explain materialized views and how they are used ?**   Materialized views are objects that are reduced sets of information that have been summarized, grouped, or aggregated from base tables. They are typically used in [data warehouse](javascript:void(0);) or decision support systems. The JobQueue process refresh the materialized view data.  **Explain Database Pessimistic and Optimistic locking?**  Transactional isolation is usually implemented by locking whatever is accessed in a transaction. There are two different approaches to transactional locking:   Pessimistic locking   Optimistic locking        The disadvantage of pessimistic locking is that a resource is locked from the time it is first accessed in a transaction until the transaction is finished, making it inaccessible to other transactions during that time. If most transactions simply look at the resource and never change it, an exclusive lock may be overkill as it may cause lock contention, and optimistic locking may be a better approach. With pessimistic locking, locks are applied in a fail-safe way. In the banking application example, an account is locked as soon as it is accessed in a transaction. Attempts to use the account in other transactions while it is locked will either result in the other process being delayed until the account lock is released, or that the process transaction will be rolled back. The lock exists until the transaction has either been committed or rolled back.        With optimistic locking, a resource is not actually locked when it is first is accessed by a transaction. Instead, the state of the resource at the time when it would have been locked with the pessimistic locking approach is saved. Other transactions are able to concurrently access to the resource and the possibility of conflicting changes is possible. At commit time, when the resource is about to be updated in persistent storage, the state of the resource is read from storage again and compared to the state that was saved when the resource was first accessed in the transaction. If the two states differ, a conflicting update was made, and the transaction will be rolled back. This is very important for application performance to use Optimistic locking.  **Difference in Inner join and  Outer join?**  **Inner Joins**     An inner join (sometimes called a simple join) is a join of two or more tables that returns only those rows that satisfy the join condition.  **Outer Joins**     An outer join extends the result of a simple join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other satisfy the join condition.  To write a query that performs an outer join of tables A and B and returns all rows from A (a left outer join), use the LEFT [OUTER] JOIN syntax in the FROM clause, or apply the outer join operator (+) to all columns of B in the join condition in the WHERE clause. For all rows in A that have no matching rows in B, [Oracle Database](javascript:void(0);) returns null for any select list expressions containing columns of B.  To write a query that performs an outer join of tables A and B and returns all rows from B (a right outer join), use the RIGHT [OUTER] JOIN syntax in the FROM clause, or apply the outer join operator (+) to all columns of A in the join condition in the WHERE clause. For all rows in B that have no matching rows in A, Oracle returns null for any select list expressions containing columns of A.  To write a query that performs an outer join and returns all rows from A and B, extended with nulls if they do not satisfy the join condition (a full outer join), use the FULL [OUTER] JOIN syntax in the FROM clause.  **Why Group by is only used with having clause and  aggregative functions example?**  HAVING is used to perform an action on groups created by GROUP BY similar to that of the WHERE clause on rows in a basic SQL statement. The WHERE clause limits the rows evaluated. The HAVING clause limits the grouped rows returned.  [AVG](http://psoug.org/definition/AVG.htm): returns the variance or variability of an expression.  [COUNT](http://psoug.org/definition/COUNT.htm): returns the number of rows returned by a query.  [FIRST](http://psoug.org/definition/FIRST.htm): returns the first value from an ordered sequence.  [LAST](http://psoug.org/definition/LAST.htm): returns the last value from an ordered sequence.  [MAX](http://psoug.org/definition/MAX.htm): returns the maximum value of a column.  [MIN](http://psoug.org/definition/MIN.htm): returns the minimum value of a column.  [STDDEV](http://psoug.org/definition/STDDEV.htm): returns the standard deviation of a set of numbers.  [SUM](http://psoug.org/definition/SUM.htm): returns the sum or total the values of a column.  [VARIANCE](http://psoug.org/definition/VARIANCE.htm): returns the variance or *variability* of an expression  HAVING can be used only with the SELECT statement. HAVING is typically used in a GROUP BY clause. When GROUP BY is not used, HAVING behaves like a WHERE clause.  A HAVING clause is like a WHERE clause, but applies only to groups as a whole, whereas the WHERE clause applies to individual rows. A query can contain both a WHERE clause and a HAVING clause. The WHERE clause is applied first to the individual rows in the tables . Only the rows that meet the conditions in the WHERE clause are grouped. The HAVING clause is then applied to the rows in the result set. Only the groups that meet the HAVING conditions appear in the query output. You can apply a HAVING clause only to columns that also appear in the GROUP BY clause or in an aggregate function.  Example of HAVING and WHERE in one query:  *SELECT titles.pub\_id, AVG( titles.price) FROM titles INNER JOIN publishers ON titles.pub\_id = publishers.pub\_id WHERE publishers.state = 'CA' GROUP BY titles.pub\_id HAVING  AVG(titles.price)  > 10;*  Sometimes you can specify the same set of rows using either a WHERE clause or a HAVING clause. In such cases, one method is not more or less efficient than the other. The optimizer always automatically analyzes each statement you enter and selects an efficient means of executing it. It is best to use the syntax that most clearly describes the desired result. In general, that means eliminating undesired rows in earlier clauses.  **What is a Stored Procedure?**  A [stored procedure](javascript:void(0);) is a named group of SQL statements that have been previously created and stored in the server database. Stored procedures accept input parameters so that a single procedure can be used over the network by several clients using different input data. And when the procedure is modified, all clients automatically get the new version. Stored procedures reduce network traffic and improve performance. Stored procedures can be used to help ensure the integrity of the database.  e.g. sp\_helpdb, sp\_renamedb, sp\_depends etc.  **What is a Trigger?**  A trigger is a SQL procedure that initiates an action when an event (INSERT, DELETE or UPDATE) occurs. Triggers are stored in and managed by the DBMS. Triggers are used to maintain the referential integrity of data by changing the data in a systematic fashion. A trigger cannot be called or executed; DBMS automatically fires the trigger as a result of a data modification to the associated table. Triggers can be considered to be similar to stored procedures in that both consist of procedural logic that is stored at the database level. Stored procedures, however, are not event-drive and are not attached to a specific table as triggers are. Stored procedures are explicitly executed by invoking a CALL to the procedure while triggers are implicitly executed. In addition, triggers can also execute stored procedures.  *Nested Trigger*: A trigger can also contain INSERT, UPDATE and DELETE logic within itself; so when the trigger is fired because of data modification, it can also cause another data modification, thereby firing another trigger. A trigger that contains data modification logic within itself is called a nested trigger.  **What are the Different Types of Triggers?**  There are two types of Triggers.  1)      DML Trigger  There are two types of DML Triggers      1.Instead of Trigger         Instead of Triggers are fired in place of the triggering action such as an insert, update, or delete.      2. After Trigger         After triggers execute following the triggering action, such as an insert, update, or delete.  2)      DDL Trigger  This type of trigger is fired against Drop Table, Create Table, Alter Table or Login events. DDL Triggers are always After Triggers.  **What is a View?**  A simple view can be thought of as a subset of a table. It can be used for retrieving data as well as updating or deleting rows. Rows updated or deleted in the view are updated or deleted in the table the view was created with. It should also be noted that as data in the original table changes, so does the data in the view as views are the way to look at parts of the original table. The results of using a view are not permanently stored in the database. The data accessed through a view is actually constructed using standard T-SQL select command and can come from one to many different base tables or even other views.  **What is an Index?**  An index is a physical structure containing pointers to the data. Indices are created in an existing table to locate rows more quickly and efficiently. It is possible to create an index on one or more columns of a table, and each index is given a name. The users cannot see the indexes; they are just used to speed up queries. Effective indexes are one of the best ways to improve performance in a database application. A table scan happens when there is no index available to help a query. In a table scan, the SQL Server examines every row in the table to satisfy the query results. Table scans are sometimes unavoidable, but on large tables, scans have a terrific impact on performance.  **What is a Cursor?**  A cursor is a database object used by applications to manipulate data in a set on a row-by-row basis, instead of the typical SQL commands that operate on all the rows in the set at one time.  In order to work with a cursor, we need to perform some steps in the following order:  Declare cursor  Open cursor  Fetch row from the cursor  Process fetched row  Close cursor  Deallocate cursor.  **What is the Difference between a Function and a Stored Procedure?**  UDF can be used in the SQL statements anywhere in the WHERE/HAVING/SELECT section, whereas Stored procedures cannot be. UDFs that return tables can be treated as another rowset. This can be used in JOINs with other tables. Inline UDF’s can be thought of as views that take parameters and can be used in JOINs and other Rowset operations.  **What are Different Types of Join?**  **Cross Join**  A cross join that does not have a WHERE clause produces the Cartesian product of the tables involved in the join. The size of a Cartesian product result set is the number of rows in the first table multiplied by the number of rows in the second table. The common example is when company wants to combine each product with a pricing table to analyze each product at each price.  **Inner Join**      A join that displays only the rows that have a match in both joined tables is known as inner Join. This is the default type of join in the Query and View Designer.  **Outer Join**  A join that includes rows even if they do not have related rows in the joined table is an Outer Join. You can create three different outer join to specify the unmatched rows to be included:  **Left Outer Join:** In Left Outer Join, all the rows in the first-named table, i.e. “left” table, which appears leftmost in the JOIN clause, are included. Unmatched rows in the right table do not appear.  **Right Outer Join:** In Right Outer Join, all the rows in the second-named table, i.e. “right” table, which appears rightmost in the JOIN clause are included. Unmatched rows in the left table are not included.  **Full Outer Join:** In Full Outer Join, all the rows in all joined tables are included, whether they are matched or not.  **Self Join**      This is a particular case when one table joins to itself with one or two aliases to avoid confusion. A self join can be of any type, as long as the joined tables are the same. A self join is rather unique in that it involves a relationship with only one table. The common example is when company has a hierarchal reporting structure whereby one member of staff reports to another. Self Join can be Outer Join or Inner Join.  **Difference in clustered and unclustered index?**  **Non-clustered**  The data is present in arbitrary order, but the logical ordering is specified by the index. The data rows may be spread throughout the table regardless of the value of the indexed column or expression. The non-clustered index tree contains the index keys in sorted order, with the leaf level of the index containing the pointer to the record (page and the row number in the data [page](javascript:void(0);) in page-organized engines; row offset in file-organized engines). In a non-clustered index: The physical order of the rows is not the same as the index order. Typically created on non-primary key columns used in JOIN, WHERE, and ORDER BY clauses. There can be more than one non-clustered index on a database table.  **Clustered**  Clustering alters the data block into a certain distinct order to match the index, resulting in the row data being stored in order. Therefore, only one clustered index can be created on a given database table. Clustered indices can greatly increase overall speed of retrieval, but usually only where the data is accessed sequentially in the same or reverse order of the clustered index, or when a range of items is selected. Since the physical records are in this sort order on disk, the next row item in the sequence is immediately before or after the last one, and so fewer data block reads are required. The primary feature of a clustered index is therefore the ordering of the physical data rows in accordance with the index blocks that point to them. Some databases separate the data and index blocks into separate files, others put two completely different data blocks within the same physical file(s).  **What are the Different Index Configurations a Table can have?**  A table can have one of the following index configurations:  No indexes  A clustered index  A clustered index and many non-clustered indexes  A non-clustered index  Many non-clustered indexes  **What’s the Difference between a Primary Key and a Unique Key?**  Both primary key and unique key enforce uniqueness of the column on which they are defined. But by default, the primary key creates a clustered index on the column, whereas unique key creates a non-clustered index by default. Another major difference is that primary key doesn’t allow NULLs, but unique key allows one NULL only.  **What is Difference between DELETE  and TRUNCATE Commands?**  Delete command removes the rows from a table on the basis of the condition that we provide with a WHERE clause. Truncate will actually remove all the rows from a table, and there will be no data in the table after we run the truncate command.  TRUNCATE  TRUNCATE is faster and uses fewer system and transaction log resources than DELETE. (Read all the points below)  TRUNCATE removes the data by deallocating the data pages used to store the table’s data, and only the [page](javascript:void(0);) deallocations are recorded in the transaction log.  TRUNCATE removes all the rows from a table, but the table structure, its columns, constraints, indexes and so on remains. The counter used by an identity for new rows is reset to the seed for the column.  You cannot use TRUNCATE TABLE on a table referenced by a FOREIGN KEY constraint.  Using T-SQL – TRUNCATE cannot be rolled back unless it is used in TRANSACTION. OR TRUNCATE can be rolled back when used with BEGIN … END TRANSACTION using T-SQL.  TRUNCATE is a DDL Command.  TRUNCATE resets the identity of the table.  DELETE  DELETE removes rows one at a time and records an entry in the transaction log for each deleted row.  DELETE does not reset Identity property of the table.  DELETE can be used with or without a WHERE clause  DELETE activates Triggers if defined on table.  DELETE can be rolled back.  DELETE is DML Command.  DELETE does not reset the identity of the table.  What is the Difference between a HAVING clause and a WHERE clause?  They specify a search condition for a group or an aggregate. But the difference is that HAVING can be used only with the SELECT statement. HAVING is typically used in a GROUP BY clause. When GROUP BY is not used, HAVING behaves like a WHERE clause. Having Clause is basically used only with the GROUP BY function in a query, whereas WHERE Clause is applied to each row before they are part of the GROUP BY function in a query.  What is CHECK Constraint?  A CHECK constraint is used to limit the values that can be placed in a column. The check constraints are used to enforce domain integrity.  What is NOT NULL Constraint?  A NOT NULL constraint enforces that the column will not accept null values. The not null constraints are used to enforce domain integrity, as the check constraints.  What is the difference between UNION and UNION ALL?  UNION   The UNION command is used to select related information from two tables, much like the JOIN command. However, when using the UNION command all selected columns need to be of the same data type. With UNION, only distinct values are selected.  UNION ALL     The UNION ALL command is equal to the UNION command, except that UNION ALL selects all values.  The difference between UNION and UNION ALL is that UNION ALL will not eliminate duplicate rows, instead it just pulls all rows from all the tables fitting your query specifics and combines them into a table. | |   Q 5 - **If the column of the table is having case sensitive data like 'Abc' and 'ABC'  and we want to do** case sensitive search on table. How we can improve performance of case sensitive search? A- In this case we can define case sensitive index on table so that each search is restricted to particular value compare to full table scan. You can force all your data to be the same case by using UPPER() or LOWER():   select \* from my\_table where upper(column\_1) = upper('my\_string'); or   select \* from my\_table where lower(column\_1) = lower('my\_string');  If column\_1 is not indexed on upper(column\_1) or lower(column\_1), as appropriate, this may force a full table scan. In order to avoid this you can create a function-based index.  create index my\_index on my\_table ( lower(column\_1) );  If you're using LIKE then you have to concatenate a % around the string you're searching for.  select \* from my\_table where lower(column\_1) = lower('my\_string') || '%';  **Q 5 - What are Synonyms?**  A Synonyms give you the ability to provide alternate names for database objects. You can alias object names; for example, using the Employee table as Emp. You can also shorten names. This is especially useful when dealing with three and four part names; for example, shortening server.database.owner.object to object.  **Q 9 - Remove duplicates from table?** A:  DELETE FROM TableName  WHERE ID NOT IN (SELECT MAX(ID)  FROM TableName  GROUP BY Column1,  Column2,  Column3  HAVING MAX(ID) IS NOT NULL)  **Q 10 : Finding N'th Maximum salary SQL query?**  // Using Sub query SELECT \*  FROM Employee Emp1  WHERE (N-1) = (  SELECT COUNT(DISTINCT(Emp2.Salary))  FROM Employee Emp2  WHERE Emp2.Salary > Emp1.Salary  )  //Another way to get 2'nd maximum salary Select max(Salary) From Employee e where  e.sal < ( select max(sal) from employee );  **Q 11 - Database Query Optimisation tips with JDBC:**  A : Use prepared statement pooling  Database supports prepared statement pooling for pooled connections, as discussed in the JDBC 3.0 specification, through the TimesTenObservableConnectionDS class. Note that statement pooling is transparent to an application. Use of the PreparedStatement object, including preparing and closing the statement, is no different.  Use arrays of parameters for batch execution  You can improve performance by using groups, referred to as *batches*, of statement executions, calling the addBatch() and executeBatch() methods forStatement or PreparedStatement objects.  A batch can consist of a set of INSERT, UPDATE, DELETE, or MERGE statements. Statements that return result sets, such as SELECT statements, are not allowed in a batch. A SQL statement is added to a batch by calling addBatch() on the statement object. The set of SQL statements associated with a batch are executed through the executeBatch() method.  For PreparedStatement objects, a batch consists of repeated executions of a statement using different input parameter values. For each set of input values, create the batch by using appropriate set*XXX*() calls followed by the addBatch() call. The batch is executed by the executeBatch() method.  // turn off autocommit  conn.setAutoCommit(false);  Statement stmt = conn.createStatement();  stmt.addBatch("INSERT INTO employees VALUES (1000, 'Joe Jones')");  stmt.addBatch("INSERT INTO departments VALUES (260, 'Shoe')");  stmt.addBatch("INSERT INTO emp\_dept VALUES (1000, 260)");  // submit a batch of update commands for execution  int[] updateCounts = stmt.executeBatch();  conn.commit ();  Bulk fetch rows of data  Oracle provides an extension that allows an application to fetch multiple rows of data. For applications that retrieve large amounts of data, fetching multiple rows can increase performance greatly. However, when using Read Committed isolation level, locks are held on all rows being retrieved until the application has received all the data, decreasing concurrency. For more information on this feature, see ["Fetching multiple rows of data"](http://docs.oracle.com/cd/E13085_01/doc/timesten.1121/e13068/writing_app.htm#BABIFHFA).  Use the ResultSet method getString() sparingly  Because Java strings are immutable, the ResultSet method getString() must allocate space for a new string in addition to translating the underlying C string to a Unicode string, making it a costly call.  In addition, you should not call getString() on primitive numeric types, like byte or int, unless it is absolutely necessary. It is much faster to call getInt()on an integer column, for example.  Avoid data type conversions  TimesTen instruction paths are so short that even small delays due to data conversion can cause a relatively large percentage increase in transaction time.  Use the appropriate get*XXX*() method on a ResultSet object for the data type of the data in the underlying database. For example, if the data type of the data is DOUBLE, to avoid data conversion in the [JDBC driver](javascript:void(0);) you should call getDouble(). Similarly, use the appropriate set*XXX*() method on thePreparedStatement object for the input parameter in an SQL statement. For example, if you are inserting data into a CHAR column using aPreparedStatement, you should use setString(). |

# ORACLE/PLSQL: JOINS

Learn how to use **JOINS** (inner and outer) in Oracle with syntax, visual illustrations, and examples.

## DESCRIPTION

Oracle **JOINS** are used to retrieve data from multiple tables. An Oracle JOIN is performed whenever two or more tables are joined in a SQL statement.

There are 4 different types of Oracle joins:

* Oracle INNER JOIN (or sometimes called simple join)
* Oracle LEFT OUTER JOIN (or sometimes called LEFT JOIN)
* Oracle RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)
* Oracle FULL OUTER JOIN (or sometimes called FULL JOIN)

So let's discuss Oracle JOIN syntax, look at visual illustrations of Oracle JOINS, and explore Oracle JOIN examples.

## INNER JOIN (SIMPLE JOIN)

Chances are, you've already written a statement that uses an Oracle INNER JOIN. It is the most common type of join. Oracle INNER JOINS return all rows from multiple tables where the join condition is met.

### SYNTAX

The syntax for the Oracle **INNER JOIN** is:

SELECT columns

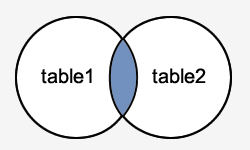
FROM table1

INNER JOIN table2

ON table1.column = table2.column;

### VISUAL ILLUSTRATION

In this visual diagram, the Oracle INNER JOIN returns the shaded area:



The Oracle INNER JOIN would return the records where table1 and table2 intersect.

### EXAMPLE

Here is an example of an Oracle INNER JOIN:

SELECT suppliers.supplier\_id, suppliers.supplier\_name, orders.order\_date

FROM suppliers

INNER JOIN orders

ON suppliers.supplier\_id = orders.supplier\_id;

This Oracle INNER JOIN example would return all rows from the suppliers and orders tables where there is a matching supplier\_id value in both the suppliers and orders tables.

Let's look at some data to explain how the INNER JOINS work:

We have a table called suppliers with two fields (supplier\_id and supplier\_ name). It contains the following data:

|  |  |
| --- | --- |
| **supplier\_id** | **supplier\_name** |
| 10000 | IBM |
| 10001 | Hewlett Packard |
| 10002 | Microsoft |
| 10003 | NVIDIA |

We have another table called orders with three fields (order\_id, supplier\_id, and order\_date). It contains the following data:

|  |  |  |
| --- | --- | --- |
| **order\_id** | **supplier\_id** | **order\_date** |
| 500125 | 10000 | 2003/05/12 |
| 500126 | 10001 | 2003/05/13 |
| 500127 | 10004 | 2003/05/14 |

If we run the Oracle SELECT statement (that contains an INNER JOIN) below:

SELECT suppliers.supplier\_id, suppliers.supplier\_name, orders.order\_date

FROM suppliers

INNER JOIN orders

ON suppliers.supplier\_id = orders.supplier\_id;

Our result set would look like this:

|  |  |  |
| --- | --- | --- |
| **supplier\_id** | **name** | **order\_date** |
| 10000 | IBM | 2003/05/12 |
| 10001 | Hewlett Packard | 2003/05/13 |

The rows for Microsoft and NVIDIA from the supplier table would be omitted, since the supplier\_id's 10002 and 10003 do not exist in both tables. The row for 500127 (order\_id) from the orders table would be omitted, since the supplier\_id 10004 does not exist in the suppliers table.

### OLD SYNTAX

As a final note, it is worth mentioning that the Oracle INNER JOIN example above could be rewritten using the older implicit syntax as follows (but we still recommend using the INNER JOIN keyword syntax):

SELECT suppliers.supplier\_id, suppliers.supplier\_name, orders.order\_date

FROM suppliers, orders

WHERE suppliers.supplier\_id = orders.supplier\_id;

## LEFT OUTER JOIN

Another type of join is called an Oracle LEFT OUTER JOIN. This type of join returns all rows from the LEFT-hand table specified in the ON condition and **only** those rows from the other table where the joined fields are equal (join condition is met).

### SYNTAX

The syntax for the Oracle **LEFT OUTER JOIN** is:

SELECT columns

FROM table1

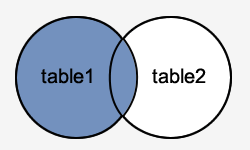
LEFT [OUTER] JOIN table2

ON table1.column = table2.column;

In some databases, the LEFT OUTER JOIN keywords are replaced with LEFT JOIN.

### VISUAL ILLUSTRATION

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



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

### EXAMPLE

Here is an example of an Oracle LEFT OUTER JOIN:

SELECT suppliers.supplier\_id, suppliers.supplier\_name, orders.order\_date

FROM suppliers

LEFT OUTER JOIN orders

ON suppliers.supplier\_id = orders.supplier\_id;

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

If a supplier\_id value in the suppliers table does not exist in the orders table, all fields in the orders table will display as <null> in the result set.

Let's look at some data to explain how LEFT OUTER JOINS work:

We have a table called suppliers with two fields (supplier\_id and name). It contains the following data:

|  |  |
| --- | --- |
| **supplier\_id** | **supplier\_name** |
| 10000 | IBM |
| 10001 | Hewlett Packard |
| 10002 | Microsoft |
| 10003 | NVIDIA |

We have a second table called orders with three fields (order\_id, supplier\_id, and order\_date). It contains the following data:

|  |  |  |
| --- | --- | --- |
| **order\_id** | **supplier\_id** | **order\_date** |
| 500125 | 10000 | 2003/05/12 |
| 500126 | 10001 | 2003/05/13 |

If we run the SELECT statement (that contains a LEFT OUTER JOIN) below:

SELECT suppliers.supplier\_id, suppliers.supplier\_name, orders.order\_date

FROM suppliers

LEFT OUTER JOIN orders

ON suppliers.supplier\_id = orders.supplier\_id;

Our result set would look like this:

|  |  |  |
| --- | --- | --- |
| **supplier\_id** | **supplier\_name** | **order\_date** |
| 10000 | IBM | 2003/05/12 |
| 10001 | Hewlett Packard | 2003/05/13 |
| 10002 | Microsoft | <null> |
| 10003 | NVIDIA | <null> |

The rows for Microsoft and NVIDIA would be included because a LEFT OUTER JOIN was used. However, you will notice that the order\_date field for those records contains a <null> value.

### OLD SYNTAX

As a final note, it is worth mentioning that the LEFT OUTER JOIN example above could be rewritten using the older implicit syntax that utilizes the outer join operator (+) as follows (but we still recommend using the LEFT OUTER JOIN keyword syntax):

SELECT suppliers.supplier\_id, suppliers.supplier\_name, orders.order\_date

FROM suppliers, orders

WHERE suppliers.supplier\_id = orders.supplier\_id(+);

## RIGHT OUTER JOIN

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

### SYNTAX

The syntax for the Oracle **RIGHT OUTER JOIN** is:

SELECT columns

FROM table1

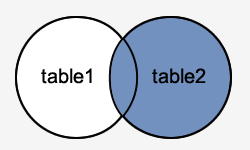
RIGHT [OUTER] JOIN table2

ON table1.column = table2.column;

In some databases, the RIGHT OUTER JOIN keywords are replaced with RIGHT JOIN.

### VISUAL ILLUSTRATION

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



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

### EXAMPLE

Here is an example of an Oracle RIGHT OUTER JOIN:

SELECT orders.order\_id, orders.order\_date, suppliers.supplier\_name

FROM suppliers

RIGHT OUTER JOIN orders

ON suppliers.supplier\_id = orders.supplier\_id;

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

If a supplier\_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.

Let's look at some data to explain how RIGHT OUTER JOINS work:

We have a table called suppliers with two fields (supplier\_id and name). It contains the following data:

|  |  |
| --- | --- |
| **supplier\_id** | **supplier\_name** |
| 10000 | Apple |
| 10001 | Google |

We have a second table called orders with three fields (order\_id, supplier\_id, and order\_date). It contains the following data:

|  |  |  |
| --- | --- | --- |
| **order\_id** | **supplier\_id** | **order\_date** |
| 500125 | 10000 | 2013/08/12 |
| 500126 | 10001 | 2013/08/13 |
| 500127 | 10002 | 2013/08/14 |

If we run the SELECT statement (that contains a RIGHT OUTER JOIN) below:

SELECT orders.order\_id, orders.order\_date, suppliers.supplier\_name

FROM suppliers

RIGHT OUTER JOIN orders

ON suppliers.supplier\_id = orders.supplier\_id;

Our result set would look like this:

|  |  |  |
| --- | --- | --- |
| **order\_id** | **order\_date** | **supplier\_name** |
| 500125 | 2013/08/12 | Apple |
| 500126 | 2013/08/13 | Google |
| 500127 | 2013/08/14 | <null> |

The row for 500127 (order\_id) would be included because a RIGHT OUTER JOIN was used. However, you will notice that the supplier\_name field for that record contains a <null> value.

### OLD SYNTAX

As a final note, it is worth mentioning that the RIGHT OUTER JOIN example above could be rewritten using the older implicit syntax that utilizes the outer join operator (+) as follows (but we still recommend using the RIGHT OUTER JOIN keyword syntax):

SELECT orders.order\_id, orders.order\_date, suppliers.supplier\_name

FROM suppliers, orders

WHERE suppliers.supplier\_id(+) = orders.supplier\_id;

## FULL OUTER JOIN

Another type of join is called an Oracle FULL OUTER JOIN. This type of join returns all rows from the LEFT-hand table and RIGHT-hand table with nulls in place where the join condition is not met.

### SYNTAX

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

SELECT columns

FROM table1

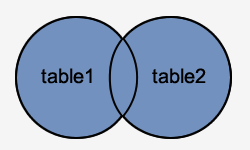
FULL [OUTER] JOIN table2

ON table1.column = table2.column;

In some databases, the FULL OUTER JOIN keywords are replaced with FULL JOIN.

### VISUAL ILLUSTRATION

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



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

### EXAMPLE

Here is an example of an Oracle FULL OUTER JOIN:

SELECT suppliers.supplier\_id, suppliers.supplier\_name, orders.order\_date

FROM suppliers

FULL OUTER JOIN orders

ON suppliers.supplier\_id = orders.supplier\_id;

This FULL OUTER JOIN example would return all rows from the suppliers table and all rows from the orders table and whenever the join condition is not met, <nulls> would be extended to those fields in the result set.

If a supplier\_id value in the suppliers table does not exist in the orders table, all fields in the orders table will display as <null> in the result set. If a supplier\_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.

Let's look at some data to explain how FULL OUTER JOINS work:

We have a table called suppliers with two fields (supplier\_id and name). It contains the following data:

|  |  |
| --- | --- |
| **supplier\_id** | **supplier\_name** |
| 10000 | IBM |
| 10001 | Hewlett Packard |
| 10002 | Microsoft |
| 10003 | NVIDIA |

We have a second table called orders with three fields (order\_id, supplier\_id, and order\_date). It contains the following data:

|  |  |  |
| --- | --- | --- |
| **order\_id** | **supplier\_id** | **order\_date** |
| 500125 | 10000 | 2013/08/12 |
| 500126 | 10001 | 2013/08/13 |
| 500127 | 10004 | 2013/08/14 |

If we run the SELECT statement (that contains a FULL OUTER JOIN) below:

SELECT suppliers.supplier\_id, suppliers.supplier\_name, orders.order\_date

FROM suppliers

FULL OUTER JOIN orders

ON suppliers.supplier\_id = orders.supplier\_id;

Our result set would look like this:

|  |  |  |
| --- | --- | --- |
| **supplier\_id** | **supplier\_name** | **order\_date** |
| 10000 | IBM | 2013/08/12 |
| 10001 | Hewlett Packard | 2013/08/13 |
| 10002 | Microsoft | <null> |
| 10003 | NVIDIA | <null> |
| <null> | <null> | 2013/08/14 |

The rows for Microsoft and NVIDIA would be included because a FULL OUTER JOIN was used. However, you will notice that the order\_date field for those records contains a <null> value.

The row for supplier\_id 10004 would be also included because a FULL OUTER JOIN was used. However, you will notice that the supplier\_id and supplier\_name field for those records contain a <null> value.

### OLD SYNTAX

As a final note, it is worth mentioning that the FULL OUTER JOIN example above could not have been written in the old syntax without using a [**UNION query**](http://www.techonthenet.com/oracle/union.php).

**Average**

SELECT AVG(DISTINCT salary) AS "Avg Salary"

FROM employees

WHERE salary > 25000

SELECT AVG(sales \* 0.10) AS "Average Commission"

FROM orders;

SELECT AVG(sales \* 0.10) AS "Average Commission"

FROM orders;

SELECT department, AVG(sales) AS "Avg sales"

FROM order\_details

GROUP BY department

**Exists**

SELECT \*

FROM customers

WHERE EXISTS (SELECT \*

FROM order\_details

WHERE customers.customer\_id = orders.customer\_id);

**Having**

SELECT department, SUM(sales) AS "Total sales"

FROM order\_details

GROUP BY department

HAVING SUM(sales) > 25000;

**Intersect**

SELECT supplier\_id, supplier\_name

FROM suppliers

WHERE supplier\_id > 500

INTERSECT

SELECT company\_id, company\_name

FROM companies

WHERE company\_name in ( 'Apple', 'Microsoft', 'Oracle' )

ORDER BY 2;

**IS NOT NULL**

SELECT \*

FROM customers

WHERE customer\_name IS NOT NULL;

**Table Name : Employee**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **EMPLOYEE\_ID** | **FIRST\_NAME** | **LAST\_NAME** | **SALARY** | **JOINING\_DATE** | **DEPARTMENT** |
| 1 | John | Abraham | 1000000 | 01-JAN-13 12.00.00 AM | Banking |
| 2 | Michael | Clarke | 800000 | 01-JAN-13 12.00.00 AM | Insurance |
| 3 | Roy | Thomas | 700000 | 01-FEB-13 12.00.00 AM | Banking |
| 4 | Tom | Jose | 600000 | 01-FEB-13 12.00.00 AM | Insurance |
| 5 | Jerry | Pinto | 650000 | 01-FEB-13 12.00.00 AM | Insurance |
| 6 | Philip | Mathew | 750000 | 01-JAN-13 12.00.00 AM | Services |
| 7 | TestName1 | 123 | 650000 | 01-JAN-13 12.00.00 AM | Services |
| 8 | TestName2 | Lname% | 600000 | 01-FEB-13 12.00.00 AM | Insurance |

**Table Name : Incentives**

|  |  |  |
| --- | --- | --- |
| **EMPLOYEE\_REF\_ID** | **INCENTIVE\_DATE** | **INCENTIVE\_AMOUNT** |
| 1 | 01-FEB-13 | 5000 |
| 2 | 01-FEB-13 | 3000 |
| 3 | 01-FEB-13 | 4000 |
| 1 | 01-JAN-13 | 4500 |
| 2 | 01-JAN-13 | 3500 |

[**http://a4academics.com/interview-questions/53-database-and-sql/397-top-100-database-sql-interview-questions-and-answers-examples-queries?showall=&limitstart**](http://a4academics.com/interview-questions/53-database-and-sql/397-top-100-database-sql-interview-questions-and-answers-examples-queries?showall=&limitstart)**=**

**Select first 3 characters of FIRST\_NAME from EMPLOYEE**

select substr(FIRST\_NAME,0,3) from employee

**Get position of 'o' in name 'John' from employee table**

Select instr(FIRST\_NAME,'o') from employee where first\_name='John'

**Get FIRST\_NAME from employee table after removing white spaces from right side**

select RTRIM(FIRST\_NAME) from employee

**Get length of FIRST\_NAME from employee table**

select length(FIRST\_NAME) from employee

**Get First\_Name from employee table after replacing 'o' with '$'**

select REPLACE(FIRST\_NAME,'o','$') from employee

**Get First\_Name and Last\_Name as single column from employee table separated by a '**Select FIRST\_NAME|| '\_' ||LAST\_NAME from EMPLOYEE

**Get FIRST\_NAME ,Joining year,Joining Month and Joining Date from employee table\_**

Select FIRST\_NAME, to\_char(joining\_date,'YYYY') JoinYear , to\_char(joining\_date,'Mon'), to\_char(joining\_date,'dd') from EMPLOYEE

**Get employee details from employee table whose joining year is “2013”**

Select \* from EMPLOYEE where to\_char(joining\_date,'YYYY')='2013'

**Get employee details from employee table who joined before January 1st 2013**

Select \* from EMPLOYEE where JOINING\_DATE <to\_date('01/01/2013','dd/mm/yyyy')

**Get Joining Date and Time from employee table**

select to\_char(JOINING\_DATE,'dd/mm/yyyy hh:mi:ss') from EMPLOYEE

**Get difference between JOINING\_DATE and INCENTIVE\_DATE from employee and incentives table**

Select FIRST\_NAME,INCENTIVE\_DATE - JOINING\_DATE from employee a **inner join** incentives B on A.EMPLOYEE\_ID=B.EMPLOYEE\_REF\_ID

**Get database date**

select sysdate from dual

## "SQL Group By Query"

**Get department,total salary with respect to a department from employee table.**

Select DEPARTMENT,sum(SALARY) Total\_Salary from employee group by department

**Get department,total salary with respect to a department from employee table order by total salary descending**

Select DEPARTMENT,sum(SALARY) Total\_Salary from employee group by DEPARTMENT order by Total\_Salary descending

## "SQL Mathematical Operations using Group By"

**Get department,no of employees in a department,total salary with respect to a department from employee table order by total salarydescending**

Select DEPARTMENT,count(FIRST\_NAME),sum(SALARY) Total\_Salary from employee group by DEPARTMENT order by Total\_Salary descending

**Get department wise average salary from employee table order by salaryascending**

select DEPARTMENT,avg(SALARY) AvgSalary from employee group by DEPARTMENT order by AvgSalary asc

**Get department wise maximum salary from employee table order by salaryascending**

select DEPARTMENT,max(SALARY) MaxSalary from employee group by DEPARTMENT order by MaxSalary asc

**Get department wise minimum salary from employee table order by salary ascending**

select DEPARTMENT,min(SALARY) MinSalary from employee group by DEPARTMENT order by MinSalary asc

**Select no of employees joined with respect to year and month from employee table**

select to\_char (JOINING\_DATE,'YYYY') Join\_Year,to\_char (JOINING\_DATE,'MM') Join\_Month,count(\*) Total\_Emp from employee group by to\_char (JOINING\_DATE,'YYYY'),to\_char(JOINING\_DATE,'MM')

**Select department,total salary with respect to a department from employee table where total salary greater than 800000 order by Total\_Salary descending**

Select DEPARTMENT,sum(SALARY) Total\_Salary from employee group by DEPARTMENT having sum(SALARY) >800000 order by Total\_Salary desc

**Select employee details from employee table if data exists in incentive table ?**

select \* from EMPLOYEE where exists (select \* from INCENTIVES)

**Explanation** : Here "exists" statement helps us to do the job of If statement. Main query will get executed if the sub query returns at least one row. So we can consider the sub query as "If condition" and the main query as "code block" inside the If condition. We can use any SQL commands (Joins, Group By , having etc) in sub query. This command will be useful in queries which need to detect an event and do some activity.

**How to fetch data that are common in two query results ?**

select \* from EMPLOYEE where EMPLOYEE\_ID INTERSECT select \* from EMPLOYEE where EMPLOYEE\_ID < 4

**Explanation** : Here "INTERSECT" command is used to fetch data that are common in 2 queries. In this example, we had taken EMPLOYEE table in both the queries.We can apply INTERSECT command on different tables. The result of the above query will return employee details of "ROY" because, employee id of ROY is 3, and both query results have the information about ROY.

**Get Employee ID's of those employees who didn't receive incentives without using sub query ?**

select EMPLOYEE\_ID from EMPLOYEE  
MINUS  
select EMPLOYEE\_REF\_ID from INCENTIVES

**Explanation** : To filter out certain information we use MINUS command. What MINUS Command odes is that, it returns all the results from the first query, that are not part of the second query. In our example, first three employees received the incentives. So query will return employee id's 4 to 8.

**Select 20 % of salary from John , 10% of Salary for Roy and for other 15 % of salary from employee table**

SELECT FIRST\_NAME, CASE FIRST\_NAME WHEN 'John' THEN SALARY \* .2 WHEN 'Roy' THEN SALARY \* .10 ELSE SALARY \* .15 END "Deduced\_Amount" FROM EMPLOYEE

**Explanation** : Here, we are using "SQL CASE" statement to achieve the desired results. After case statement, we had to specify the column on which filtering is applied. In our case it is "FIRST\_NAME". And in then condition, specify the name of filter like John, Roy etc. To handle conditions outside our filter, use else block where every one other than John and Roy enters.

**Select Banking as 'Bank Dept', Insurance as 'Insurance Dept' and Services as 'Services Dept' from employee table**

SQL Queries in Oracle, SELECT distinct DECODE (DEPARTMENT, 'Banking', 'Bank Dept', 'Insurance', 'Insurance Dept', 'Services', 'Services Dept') FROM EMPLOYEE  
SQL Queries in SQL Server and MySQL, SELECT case DEPARTMENT when 'Banking' then 'Bank Dept' when 'Insurance' then 'Insurance Dept' when 'Services' then 'Services Dept' end FROM EMPLOYEE

**Explanation** : Here "DECODE" keyword is used to specify the alias name. In oracle we had specify, Column Name followed by Actual Name and Alias Name as arguments. In SQL Server and MySQL, we can use the earlier switch case statements for alias names.

**Delete employee data from employee table who got incentives in incentive table**

delete from EMPLOYEE where EMPLOYEE\_ID in (select EMPLOYEE\_REF\_ID from INCENTIVES)

**Explanation** : Trick about this question is that we can't delete data from a table based on some condition in another table by joining them. Here to delete multiple entries from EMPLOYEE table, we need to use Subquery. Entries will get deleted based on the result of Subquery.

**Write a query to rank employees based on their incentives for a month**

select FIRST\_NAME,INCENTIVE\_AMOUNT,DENSE\_RANK() OVER (PARTITION BY INCENTIVE\_DATE ORDER BY INCENTIVE\_AMOUNT DESC) AS Rank from EMPLOYEE a, INCENTIVES b where a.EMPLOYEE\_ID=b.EMPLOYEE\_REF\_ID

**Explanation** : In order to rank employees based on their rank for a month, "DENSE\_RANK" keyword is used. Here partition by keyword helps us to sort the column with which filtering is done. Rank is provided to the column specified in the order by statement. The above query ranks employees with respect to their incentives for a given month.

## "SQL Join"

**Select first\_name, incentive amount from employee and incentives table for those employees who have incentives**

Select FIRST\_NAME,INCENTIVE\_AMOUNT from employee a inner join incentives B on A.EMPLOYEE\_ID=B.EMPLOYEE\_REF\_ID

**Select first\_name, incentive amount from employee and incentives table for those employees who have incentives and incentive amount greater than 3000**

Select FIRST\_NAME,INCENTIVE\_AMOUNT from employee a inner join incentives B on A.EMPLOYEE\_ID=B.EMPLOYEE\_REF\_ID and INCENTIVE\_AMOUNT >3000

**Select first\_name, incentive amount from employee and incentives table for all employes even if they didn't get incentives**

Select FIRST\_NAME,INCENTIVE\_AMOUNT from employee a left join incentives B on A.EMPLOYEE\_ID=B.EMPLOYEE\_REF\_ID

**Select first\_name, incentive amount from employee and incentives table for all employees even if they didn't get incentives and set incentive amount as 0 for those employees who didn't get incentives.**

Select FIRST\_NAME,nvl(INCENTIVE\_AMOUNT,0) from employee a left join incentives B on A.EMPLOYEE\_ID=B.EMPLOYEE\_REF\_ID

**Select first\_name, incentive amount from employee and incentives table for all employees who got incentives using left join**

Select FIRST\_NAME,nvl(INCENTIVE\_AMOUNT,0) from employee a right join incentives B on A.EMPLOYEE\_ID=B.EMPLOYEE\_REF\_ID

**Select max incentive with respect to employee from employee and incentives table using sub query**

select DEPARTMENT,(select nvl(max(INCENTIVE\_AMOUNT),0) from INCENTIVES where EMPLOYEE\_REF\_ID=EMPLOYEE\_ID) Max\_incentive from EMPLOYEE

## "Top N Salary"

**Select TOP N salary from employee table**

select \* from (select \* from employee order by SALARY desc) where rownum <N + 1

**Select 2nd Highest salary from employee table**

select min(salary) from (select \* from (select \* from employee order by SALARY desc) where rownum <3)

**Select Nth Highest salary from employee table**

select min(salary) from (select \* from (select \* from employee order by SALARY desc) where rownum <N + 1)

## "SQL Union"

**Select First\_Name,LAST\_NAME from employee table as separate rows**

select FIRST\_NAME from EMPLOYEE union select LAST\_NAME from EMPLOYEE

**What is the difference between UNION and UNION ALL ?**

Both UNION and UNION ALL is used to select information from structurally similar tables. That means corresponding columns specified in the union should have same data type. For example, in the above query, if FIRST\_NAME is DOUBLE and LAST\_NAME is STRING above query wont work. Since the data type of both the columns are VARCHAR, union is made possible. Difference between UNION and UNION ALL is that , UNION query return only distinct values.