Oracle

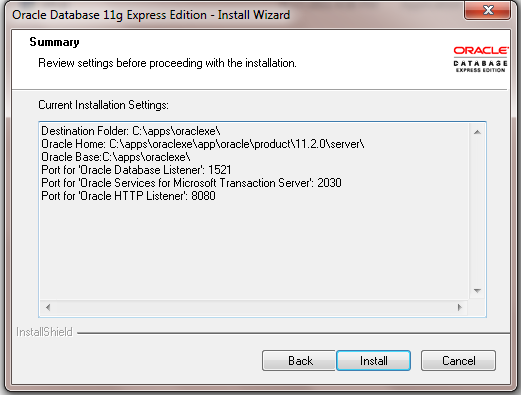
Class 1:

Relational database management systems (RDBMS).

The language, Structured English Query Language (SEQUEL) was developed by IBM Corporation, Inc., to use Codd's model. SEQUEL later became SQL (still pronounced "sequel"). In 1979, RelationalSoftware, Inc.

(now Oracle) introduced the first commercially availableimplementation of SQL. Today, SQL is accepted as the standard RDBMS language.

1. Installation of oracle 11g
2. About oracle



Connect to the oracle database

C:\Users\SURAJ>sqlplus system/suraj

**Note: Oracle is case sensitive means clause or syntaxes are case sensitive but data are not case sensitive.**

DBMS : Database Management System

RDBMS: Database Management System

ORDBMS: Object Database Management System

Eg.

**Oracle**

SYBASE

DB2

**MySql**

MsAccess

Creation of table:

Syntax:

**create table** table\_name

(

Column\_name1 datatype(size),

Column\_name1 datatype(size),

Column\_name1 datatype(size),

Column\_name1 datatype(size)

);

**Data Types**

|  |  |
| --- | --- |
| Oracle data type |  |
| Char |  |
| Varchar |  |
| varchar2 |  |
| Number |  |
| Date | the standard Oracle date format is DD-MON-YY, as follows:  '13-NOV-92' |
| Blob |  |
| Clob |  |
| File |  |

Q. Create a table table\_name as BOOKS and data fields are as below,

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| BID | BNAME | AUTHOR | COST | PUBLISHER | DOP |
| number | varchar2 | varchar2 | number | varchar2 | date |

Answer:

Create table BOOKS

(

bid number(10),

bname varchar2(10),

author varchar2(10),

cost number(10),

publisher varchar2(10),

dop date

);

**Inserting Records in the table**

Syntax:

Only for specific columns:

insert into table\_name (col1, col2, col3, ……)

values(value1, value2, value3, ………..);

For all columns:

insert into table\_name

values (value1, value2, value3, ………..);

Note : Date and character type values must be entered within ‘’.

e.g.

insert into books values (101, 'BOOK1','AUTH1',2000,'PUB1','10-OCT-14');

insert into books values (102, 'BOOK2','AUTH2',2000,'PUB2','10-NOV-14');

**Updating the record in the table:**

**Syntax**

update table\_name set

col1 = value1, col2 = value2

……..

where col1 = value1;

e.g.

update books set

bname = 'bookn1', author = 'authorn1'

where bid = 101;

* update all records

update books set

bname = 'bookn', author = 'authorn';

**Deleting Records:**

Syntax:

delete form table\_name

where condition;

e.g.

delete from books where bid = 101;

Delete all records from table

delete from books;

**Drop the table**

drop table table\_name;

e.g.

drop table books;

**Fast way to delete all records from the table**

**syntax**

truncate table table\_name;

e.g.

truncate table books;

Q. How many records will be affected by one insert statement?

A. only one or zero.

Q. How many records will be affected by one update statement?

A. zero or many

Q. How many records will be affected by one delete statement?

A. zero or many

**Fetching records from table**

Syntax:

select \* or col1,col2, col3,col4

from table

where condition

order by column\_name

group by column\_name1

having condition;

Q1. Display complete information of BOOKS table.

Q2. Display bid, bname, author, cost of all records in the books table.

Q3. Display complete information of books table written by “auth1”.

Q4.Display bname, author, cost of all records of books table whose cost is 2000.

Q5. Display all books information in ascending order by cost.

Q6. Display all books information in descending order by cost.

Q7. Display the information of a book whose bid is 101.

Answers:

1. select \* from books;
2. select bid,bname,author,cost

from books;

1. select \*

from books

where author = 'AUTH1';

1. select bname,author,cost

from books

where cost = 2000;

1. select \*

from books

order by cost asc;

1. select \*

from books

order by cost desc;

1. select \*

from books

where bid = 101;

Class 2:

Revision

1. what is dbms and types of dbms
2. creation of table
3. insertion of record into table
4. update the record in the table
5. delete the record from the table
6. fetch the record from the table

in this class:

1. fetch records in detail
2. sql operators
3. dual table
4. alias
5. sql function or aggregate function
6. use of group by and having
7. constraints
8. joins
9. sub queries
10. create table from another table
11. index
12. sequence
13. view
14. some queries
15. in detail oracle data types
16. sql built in functions
17. insert date data into table in detail

**Sql Operators:**

1. Arithmetic operator (+,-,\*,/)
2. Relational operator (=,<>,<,<=,>,>=)
3. Logical operator (or, and, not)
4. between and operator
5. in, not in operator
6. like, not like operator
7. is null, is not null operator

Questions:

Customer

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| cid | cname | mob | City | dob | Order\_ count | email |
| number | Varchar2 | number | Varchar2 | date | Number | Varchar2 |

**createtable** customer(

cid number,

cname varchar2(20),

mob number(10),

city varchar2(20),

dob **date**,

order\_count number(10),

email varchar2(50)

);

1. Display customers who are staying in Bangalore.
2. Display customers who are staying in Bangalore, Delhi, Ranchi, Patnaand Mumbai.
3. Display customers who are not staying in Kolkata.
4. Display customers who are not staying in Kolkata, Chennai and Pune.
5. Display customers who have placed orders more than 100 orders.
6. Display customers who have placed orders between 10 and 50.
7. Display customers whose name is starting with “Sur” and number of orders placed is more than 20 and staying in “Delhi”.
8. Display customers who are staying in Kolkata or who have placed order less than and equal to 5.
9. Display customers whose mobile is not available.
10. Display customers whose mobile number, city and email are not available.
11. Display customers whose mobile number, city and email are available.
12. Display customers whose email id is with gmail.
13. Display customers whose name must contain “d”.
14. Display customers whose name’s 2nd character should be “m” and name must contain “s”.
15. Display customers whose name doesn’t contain “d”.

Questions with answer:

1. Display customers who are staying in Bangalore.

**select** \* **from** customer **where** city = 'bangalore';

1. Display customers who are staying in Bangalore, Delhi, Ranchi, Patna and Mumbai.

**select** \* **from** customer **where** city **in** ('bangalore', 'delhi', 'ranchi', 'patna', 'mumbai');

1. Display customers who are not staying in Kolkata.

**select** \* **from** customer **where** city <>'kolkata';

1. Display customers whoare not staying in Kolkata, Chennai and Pune.

**select** \* **from** customer **where** city **notin** ('kolkata', 'chennai', 'pune');

1. Display customers who have placed orders more than 100 orders.

**select** \* **from** customer **where** order\_count > 100;

1. Display customers who have placed orders between 10 and 50.

**select** \* **from** customer **where** order\_count **between** 10 **and** 50;

1. Display customers whose name is starting with “Sur” and number of orders placed is more than 20 and staying in “Delhi”.

**select** \* **from** customer **where** cname = 'sur%'**and** order\_count > 20 **and** city = 'delhi';

1. Display customers who are staying in Kolkata or who have placed order less than and equal to 5.

**select** \* **from** customer **where** city = 'kolkata'**or** order\_count <= 5;

1. Display customers whose mobile is not available.

**select** \* **from** customer **where** mob **isnull**;

1. Display customers whose mobile number, city and email are not available.

**select** \* **from** customer **where** city **isnulland** mob **isnulland** email **isnull**;

1. Display customers whose mobile number, city and email are available.

**select** \* **from** customer **where** city **isnotnulland** mob **isnotnulland** email **isnotnull**;

1. Display customers whose email id is with gmail.

**select** \* **from** customer **where** email **like**'%@gmail.com';

1. Display customers whose name must contain “d”.

**select** \* **from** customer **where** cname **like**'%d%';

1. Display customers whose name’s 2nd character should be “m” and name must contain “s”.

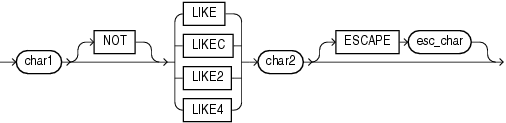
**select** \* **from** customer **where** cname **like**'\_m%'**and** cname **like**'%s%';

1. Display customers whose name doesn’t contain “d”.

**select** \* **from** customer **where** cname **notlike**'%d%';

**LIKE Operator**

The **LIKE** conditions specify a test involving pattern matching. Whereas the equality operator (=) exactly matches one character value to another, the **LIKE** conditions match a portion of one character value to another by searching the first value for the pattern specified by the second.



In this syntax:

* ***char1*** is a character expression, such as a character column, called the **search value**.
* ***char2*** is a character expression, usually a literal, called the **pattern**.
* ***esc\_char*** is a character expression, usually a literal, called the **escape character**.

The pattern can contain special pattern-matching characters:

* An underscore (\_) in the pattern matches exactly one character (as opposed to one byte in a multibyte character set) in the value.
* A percent sign (%) in the pattern can match zero or more characters (as opposed to bytes in a multibyte character set) in the value. The pattern '%' cannot match a null.

The following example searches for employees with the pattern **A\_B** in their name:

**SELECT** last\_name

**FROM** employees

**WHERE** last\_name **LIKE**'%A\\_B%'**ESCAPE**'\';

The **ESCAPE** clause identifies the backslash (\) as the escape character. In the pattern, the escape character precedes the underscore (\_). This causes Oracle to interpret the underscore literally, rather than as a special pattern matching character.

**Use of Logical operator with date data type:**

SQL> select \* from books where dop > '11-oct-14';

BID BNAME AUTHOR COST PUBLISHER DOP

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

102 BOOK2 AUTH2 2000 PUB2 10-NOV-14

SQL> select \* from books where dop < '11-oct-14';

BID BNAME AUTHOR COST PUBLISHER DOP

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

101 BOOK1 AUTH1 2000 PUB1 10-OCT-14

SQL> select \* from books where dop in ('10-oct-2014','10-nov-2014');

BID BNAME AUTHOR COST PUBLISHER DOP

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

101 BOOK1 AUTH1 2000 PUB1 10-OCT-14

102 BOOK2 AUTH2 2000 PUB2 10-NOV-14

SQL> select \* from books where dop between '05-oct-2014' and '20-oct-2014';

BID BNAME AUTHOR COST PUBLISHER DOP

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

101 BOOK1 AUTH1 2000 PUB1 10-OCT-14

SQL> select \* from books where dop = '10-oct-2014';

BID BNAME AUTHOR COST PUBLISHER DOP

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

101 BOOK1 AUTH1 2000 PUB1 10-OCT-14

**Class2:**

* **Date data type in details**

You can insert values into a datetime column in the following ways:

* Insert a character string whose format is based on the appropriate NLS format value
* Insert a literal
* Insert a literal for which implicit conversion is performed
* Use the TO\_TIMESTAMP, TO\_TIMESTAMP\_TZ, or TO\_DATE SQL function

Set the NLS\_DATE\_FORMAT

SQL> ALTER SESSION SET NLS\_DATE\_FORMAT='DD-MON-YYYY HH24:MI:SS';

[**SELECT**](http://www.oracle.com/pls/db10g/search?remark=quick_search&word=SELECT)[**SYSDATE**](http://www.oracle.com/pls/db10g/search?remark=quick_search&word=SYSDATE)[**FROM**](http://www.oracle.com/pls/db10g/search?remark=quick_search&word=FROM) **dual;**

[**ALTER**](http://www.oracle.com/pls/db10g/search?remark=quick_search&word=ALTER) **SESSION** [**SET**](http://www.oracle.com/pls/db10g/search?remark=quick_search&word=SET) **NLS\_DATE\_FORMAT ='DD-MON-YYYY HH24:MI:SS';**

[**SELECT**](http://www.oracle.com/pls/db10g/search?remark=quick_search&word=SELECT)[**SYSDATE**](http://www.oracle.com/pls/db10g/search?remark=quick_search&word=SYSDATE)[**FROM**](http://www.oracle.com/pls/db10g/search?remark=quick_search&word=FROM) **dual;**

Q. How to check the NLS\_DATE\_FORMAT?

SQL> SELECT value FROM v$nls\_parameters WHERE parameter ='NLS\_DATE\_FORMAT';

VALUE

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

DD-MON-RR

SQL>

select \* from NLS\_DATABASE\_PARAMETERS WHERE parameter ='NLS\_DATE\_FORMAT';

SHOW PARAMETER NLS\_DATE\_FORMAT;

**SQL operators in detail**

**Arithmetic operator**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| + (unary) | Makes operand positive | SELECT +3 FROM DUAL; |
| - (unary) | Negates operand | SELECT -4 FROM DUAL; |
| / | Division (numbers and dates) | SELECT SAL / 10 FROM EMP; |
| \* | Multiplication | SELECT SAL \* 5 FROM EMP; |
| + | Addition (numbers and dates) | SELECT SAL + 200 FROM EMP; |
| - | Subtraction (numbers and dates) | SELECT SAL - 100 FROM EMP; |

**Relational operator**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| = | Equality test. | SELECT ENAME "Employee" FROM EMP WHERE SAL = 1500; |
| !=, ^=, <> | Inequality test. | SELECT ENAME FROM EMP WHERE SAL ^= 5000; |
| > | Greater than test. | SELECT ENAME "Employee", JOB "Title" FROM EMP WHERE SAL > 3000; |
| < | Less than test. | SELECT \* FROM PRICE WHERE MINPRICE < 30; |
| >= | Greater than or equal to test. | SELECT \* FROM PRICE WHERE MINPRICE >= 20; |
| <= | Less than or equal to test. | SELECT ENAME FROM EMP WHERE SAL <= 1500; |
| IN | "Equivalent to any member of" test. Equivalent to "= ANY". | SELECT \* FROM EMP WHERE ENAME IN ('SMITH', 'WARD'); |
| ANY/ SOME | Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, or >=. Evaluates to FALSE if the query returns no rows. | SELECT \* FROM DEPT WHERE LOC = SOME ('NEW YORK','DALLAS'); |
| NOT IN | Equivalent to "!= ANY". Evaluates to FALSE if any member of the set is NULL. | SELECT \* FROM DEPT WHERE LOC NOT IN ('NEW YORK', 'DALLAS'); |
| ALL | Compares a value with every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, or >=. Evaluates to TRUE if the query returns no rows. | SELECT \* FROM emp WHERE sal>= ALL (1400, 3000); |
| [NOT] BETWEEN*x* and *y* | [Not] greater than or equal to *x* and less than or equal to *y*. | SELECT ENAME, JOB FROM EMP WHERE SAL BETWEEN 3000 AND 5000; |
| EXISTS | TRUE if a sub-query returns at least one row. | SELECT \* FROM EMP WHERE EXISTS (SELECT ENAME FROM EMP WHERE MGR IS NULL); |
| *x* [NOT] LIKE *y*[ESCAPE*z*] | TRUE if *x* does [not] match the pattern *y*. Within *y*, the character "%" matches any string of zero or more characters except null. The character "\_" matches any single character. Any character following ESCAPE is interpreted literally, useful when *y*contains a percent (%) or underscore (\_). | SELECT \* FROM EMP WHERE ENAME LIKE '%E%'; |
| IS [NOT] NULL | Tests for nulls. This is the only operator that should be used to test for nulls. | SELECT \* FROM EMP WHERE COMM IS NOT NULL AND SAL > 1500; |

**Logical operator**

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| NOT | Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN. | SELECT \* FROM EMP WHERE NOT (job IS NULL)  SELECT \* FROM EMP WHERE NOT (sal BETWEEN 1000 AND 2000) |
| AND | Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE; otherwise returns UNKNOWN. | SELECT \* FROM EMP WHERE job='CLERK' AND deptno=10 |
| OR | Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise, returns UNKNOWN. | SELECT \* FROM emp WHERE job='CLERK' OR deptno=10 |

**Question: What is a DUAL Table in Oracle?**  
- This is a single row and single column dummy table provided by oracle. This is used to perform mathematical calculationswithout using a table. Also can be used for any other operations.

Select \* from DUAL;

**Output:**

DUMMY  
-------  
X

Select 777 \* 888 from Dual

**Output:**

777 \* 888  
---------  
689976

**Alias:**

Temporary names for columns or tables

**Column Alias Name**

*Syntax:*

column\_name AS alias\_name

**select** name **AS** Employee\_name

**from** employee;

EMPLOYEE\_NAME

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

Jamil N.Samir

Amani F.Zaki

Jihan H.Walid

Ramy S.Nabil

Joyce A.Eman

Ahmad V.Jabbar

James B.Baher

* 1. When you want to include *spaces or special characters*in alias names, then enclose the alias name in double quotation marks.

Syntax:

Column\_name || ‘ additional string ’ “ alias\_name”

select name || ' has an id of ' || ssn **"**Important information**"**

from employee;

Important information

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

Jamil N.Samir **has an id of** 123456789

Amani F.Zaki **has an id of** 999887777

Jihan H.Walid **has an id of** 987654321

Ahmad V.Jabbar **has an id of** 987987987

James B.Baher **has an id of** 888665555

For E.g. query:

select min(cpd\_id)"Minimum CPD\_ID"

from ( select distinct(cpd\_id) from citizen where rownum < 11 order by cpd\_id desc);

select min(cpd\_id)as Minimum\_CPD\_ID from ( select distinct(cpd\_id) from citizen where rownum < 11 order by cpd\_id desc);

select cpd\_id ||' first name is:'|| FIRST\_NAME ||'sex is: '|| sex "Citizen Information " from citizen where rownum <5;

Citizen Information

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

1872000003 first name is:Mahadoosex is: M

1872000009 first name is:Narjobesex is: F

1872000029 first name is:Coomarsex is: M

1873000004 first name is:Rampersadsex is: M

**Table Alias Name:**

**Syntax**

* table\_name alias\_name

**select** T.item\_id, T.item\_desc

**from** item T;

item\_id item\_desc

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

LA-101 Box, Small

NY-102 Bottle, Large

**Concatenating Character String**

* Oracle Lite provides the CONCAT character function as an alternative to the vertical bar operator.
* Concatenating two character strings results in another character string.
* Oracle Lite preserves trailing blanks in character strings by concatenation, regardless of the strings' datatypes.
* Oracle Lite treats zero-length character strings as nulls. When you concatenate a zero-length character string with another operand the result is always the other operand. A null value can only result from the concatenation of two null strings.

For example:

SELECT CONCAT (CONCAT (ENAME, ' is a '),job) FROM EMP WHERE SAL > 2000;

This returns:

CONCAT(CONCAT(ENAME

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

KING is a PRESIDENT

BLAKE is a MANAGER

CLARK is a MANAGER

JONES is a MANAGER

FORD is a ANALYST

SCOTT is a ANALYST

6 rows selected.

**select** cpd\_id ||' first name is:'|| FIRST\_NAME ||'sex is: '|| sex "Citizen Information "**from** citizen **where** rownum <5;

Citizen Information

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

1872000003 first name is: Mahadoosex is: M

1872000009 first name is: Narjobesex is: F

1872000029 first name is: Coomarsex is: M

1873000004 first name is:Rampersadsex is: M

Question: How to add comments for the column of the table?

-- Create table

**create table** t

(

col1 number

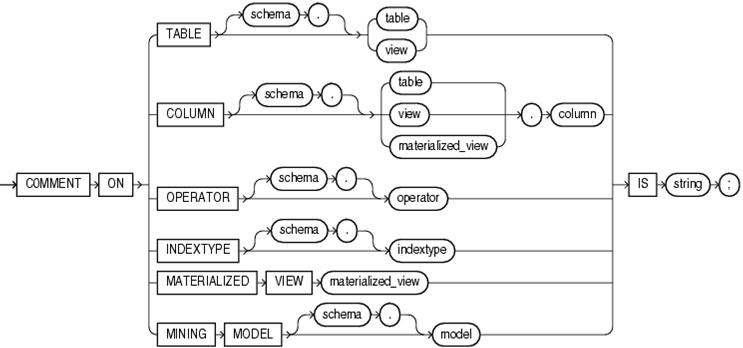
);

-- Add comments to the columns

comment **on column** t.col1 **is** 'this is a comment';

You can't specify the column comment in the table creation DDL.

 Syntax for comment:



Oracle Built in Functions

There are two types of functions in Oracle.  
**1) Single Row Functions:**

Single row or Scalar functions return a value for every row that is processed in a query.

**2) Group Functions:**

These functions group the rows of data based on the values returned by the query.

These are also known as aggregate function.

There are four types of **single row functions**. They are:   
**1) Numeric Functions:** These are functions that accept numeric input and return numeric values.   
**2) Character or Text Functions:** These are functions that accept character input and can return both character and number values.   
**3) Date Functions:** These are functions that take values that are of datatype DATE as input and return values of datatype DATE, except for the MONTHS\_BETWEEN function, which returns a number.  
**4) Conversion Functions:** These are functions that help us to convert a value in one form to another form. For Example: a null value into an actual value, or a value from one datatype to another datatype like NVL, TO\_CHAR, TO\_NUMBER, TO\_DATE etc.

**Group Function or Aggregate Function:**

Some aggregate functions:

1. AVG() - Returns the average value
2. COUNT() - Returns the number of rows
3. MAX() - Returns the largest value
4. MIN() - Returns the smallest value
5. SUM() – Returns the sum
6. FIRST() - Returns the first value
7. LAST() - Returns the last value

Some more aggregate functions:

| **Name** | **Description** |
| --- | --- |
| [**AVG()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_avg) | Return the average value of the argument |
| [**BIT\_AND()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_bit-and) | Return bitwise and |
| [**BIT\_OR()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_bit-or) | Return bitwise or |
| [**BIT\_XOR()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_bit-xor) | Return bitwise xor |
| [**COUNT(DISTINCT)**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_count-distinct) | Return the count of a number of different values |
| [**COUNT()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_count) | Return a count of the number of rows returned |
| [**GROUP\_CONCAT()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_group-concat) | Return a concatenated string |
| [**MAX()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_max) | Return the maximum value |
| [**MIN()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_min) | Return the minimum value |
| [**STD()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_std) | Return the population standard deviation |
| [**STDDEV\_POP()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_stddev-pop) | Return the population standard deviation |
| [**STDDEV\_SAMP()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_stddev-samp) | Return the sample standard deviation |
| [**STDDEV()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_stddev) | Return the population standard deviation |
| [**SUM()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_sum) | Return the sum |
| [**VAR\_POP()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_var-pop) | Return the population standard variance |
| [**VAR\_SAMP()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_var-samp) | Return the sample variance |
| [**VARIANCE()**](https://docs.oracle.com/cd/E17952_01/refman-5.1-en/group-by-functions.html#function_variance) | Return the population standard variance |

Aggregate functions return a single result row based on groups of rows, rather than on single rows. Aggregate functions can appear in select lists and in **ORDER** **BY** and **HAVING** clauses. They are commonly used with the **GROUP** **BY**clause in a **SELECT** statement, where Oracle Database divides the rows of a queried table or view into groups. In a query containing a **GROUP** **BY** clause, the elements of the select list can be aggregate functions, **GROUP** **BY**expressions, constants, or expressions involving one of these. Oracle applies the aggregate functions to each group of rows and returns a single result row for each group.

If you omit the **GROUP** **BY** clause, then Oracle applies aggregate functions in the select list to all the rows in the queried table or view. You use aggregate functions in the **HAVING** clause to eliminate groups from the output based on the results of the aggregate functions, rather than on the values of the individual rows of the queried table or view.

the **DISTINCT** average of 1, 1, 1, and 3 is 2. The **ALL** average is 1.5. If you specify neither, then the default is **ALL**.

All aggregate functions except **COUNT**(\*), **GROUPING**, and **GROUPING\_ID** ignore nulls.

You can nest aggregate functions. For example, the following example calculates the average of the maximum salaries of all the departments in the sample schema **hr**:

SELECT AVG(MAX(salary))

FROM employees

GROUP BY department\_id;

AVG(MAX(SALARY))

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

10926.3333

SELECT MAX(salary)

FROM employees

GROUP BY department\_id;

This calculation evaluates the inner aggregate (**MAX**(**salary**)) for each group defined by the **GROUP** **BY** clause (**department\_id**), and aggregates the results again.

SELECT MAX(salary) FROM employees;

**Question: Where and How Can I Use Functions?**

**Answer:**

In below two ways:

1. In SELECT Statement

**SELECT** INITCAP (agency\_desc)

**FROM** agencies ;

1. In WHERE Statement

**SELECT** \* **FROM** sites

**WHERE** SUBSTR(lut\_land\_use\_type, 1) = ‘MOBILE’;

**Group by and having**

* GROUP BY clause is used to divide the table into various groups depending on the specified column or columns.
* HAVING clause is used to apply the condition on groups.
* Whereas WHERE clause is used to specify the condition on rows of the table.

**SELECT**…

**FROM** ……

**WHERE** ……

**GROUPBY** …..

**HAVING** ….

**ORDERBY** …..;

* Where clause should be before group by clause.
* ORDER BY is use for to order the result in specific order and on specific columns.

…..

**Questions:**

Student

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| sid (number) | sname (varchar2) | feepaid (number) | feebal (number) | city (varchar2) | dor  (date) |

1. Display total number of students available.
2. Display total fee collected.
3. Display total fee collected from Bangalore students.
4. Display student who are paid maximum fee.
5. Display student whose balance fee is minimum.
6. Display student who made registration on 28th July 2015 in Bangalore.

**Answers:**

1. Display total number of students available.

**select** count(\*) "Total Student Count"**from** student;

1. Display total fee collected.

**select** sum (feepaid) "Total Fee Collected" **from** student;

1. Display total fee collected from Bangalore students.

**select** sum(feepaid) "Total Fee Collected From Bangalore" **from** student **where** city = 'bangalore';

1. Display the maximum paid fee.

**Select** max(feepaid) "Max Fee" **from** student;

1. Display minimum balance fee in Bangalore.

**Select** min(feebal) "Min Fee" **from** student **where** city = 'bangalore';

1. Display average fee paid on 28th July 2015 in Bangalore.

**Select** avg(feepaid) "Total Fee Paid" **from** student **where** dor = '28-JUL-2015';

1. Display the total fee collection fee daily wise.

**select** sum(feepaid) "Total Fee paid", dor "date"

**from** student

**groupby** dor;

1. Display the total fee collection city wise.

**select** sum(feepaid) "Total Fee paid", city "city"

**from** student

**groupby** city;

1. Display the total fee collection only on 28th July 2015 in Bangalore.

**select** sum(feepaid) "Total Fee paid", dor "date"

**from** student

**group by** dor

**having** dor = '28-JUL-2015';

1. Display the total fee collection city wise only for Bangalore and Mysore.

**select** sum(feepaid) "Total Fee paid", city "city"

**from** student

**group by** city

**having** city **in** ('bangalore','maysore');

**Database Constraint:**

Following are the database constraints:

1. not null
2. unique
3. primary key
4. foreign key
5. check
6. default

**Note: null means in oracle is not specified any value, it doesn’t mean 0 or empty space or any by default literals.**

|  |  |  |
| --- | --- | --- |
| **Constraint type** | **Description** | **Apply at** |
| not null | * If you want to stop the null value for any column you can specify not null constraint for that column. | Only at column level |
| Unique | * To avoid duplicate values for the column we can specify the unique constraint. * Unique constraint may have multiple null values. | Column level and table level |
| primary key | * Primary key constraint is used to identify a row uniquely in the table. * Primary key is the combination of not null and unique constraint. * One table can have only one primary key. It could be simple primary key (PK with one column) or composite primary key(PK with multiple column). | Either at column level or table level  (at only one place) |
| foreign key | * To establish the relationship between two tables we use foreign key. * Primary key of one table can be used as the foreign key in another table. * The table which contains the foreign key is called as child table. * The table which contains the primary key is called as parent table. * Insert first in parent then in child. * Delete first from child then from parent. | Either at column level or table level |
| check | * To perform any business condition we use check constraint. | Either at column level or table level |
| default | * To provide the default value to specific column, if there is no any value provided to it. | Only at column level |

**Syntax of constraint:**

Column level constraint

column\_name data\_type (size) [**constraint** constraint\_name] **constrainttype**

column\_name data\_type (size) [**constraint** constraint\_name] **references** parent\_table (column\_name)

Table level constraint

[**constraint** constraint\_name] constraint\_type (column\_name,… , …..)

[**constraint** constraint\_name] **foreignkey** (column\_name) **references** parent\_table(column\_name)

**For not null and default:**

column\_name data\_type (size) **defaultvalue**

column\_name data\_type (size) **notnull**

**To add the constraint in the table:**

**altertable** table\_name

**add** [**constraint** constraint\_name]

constraint\_type (column\_name, …., …)

For check constraint:

**altertable** table\_name **add** [**constraint** constraint\_name] **check** (condition)

For not null

**altertable** table\_name

modify ( column\_name data\_type (size) **notnull**)

For default

**altertable** table\_name

modify column\_name **defaultvalue**

**For Foreign key**

**altertable** child\_table\_name **addconstraint** constraint\_name **foreignkey**(column\_name) **references** parent\_table\_name)(column\_name)

**To drop a constraint**

**altertable** table\_name **dropconstraint** constraint\_name;

**altertable** table\_name **altercolumn** column\_name **dropdefault**;

**Example:**

country varchar2(20) **constraint** d\_person\_coun **default**'india'

Error report:

SQL Error: ORA-02253: constraint specification not allowed here

02253. 00000 - "constraint specification not allowed here"

\*Cause: Constraint specification is not allowed here in the statement.

\*Action: Remove the constraint specification from the statement.

**Constraint at column level**

**Parent table**

**createtable** person (

pid number(10) **constraint** pk\_person **primarykey**,

pname varchar2(20) **notnull**,

email varchar2(50) **constraint** uk\_person\_email **unique**,

mob number(10) **constraint** uk\_person\_mob **unique**,

dob **datenotnull**,

status varchar2(1) **constraint** ck\_person\_sts **check** (status **in** ('A','I') ),

country varchar2(20) **default**'india'

);

**Child table**

**createtable** usr2(

usrid number(10) **constraint** pk\_usr2 **primarykey**,

username varchar2(20) **constraint** uk\_usr2\_uname **unique**,

pass varchar2(20) **constraint** ck\_usr2\_pass **check** (pass **isnotnulland** pass **like**'%@%'),

person\_id number(10) **constraint** fk\_person\_id **references** person(pid)

);

Insert into person

**insertinto** person

**values**(1001,'pname1', 'pname1@abc.com', 9876543321, '12-jul-1987','A','sri lanka');

**insertinto** person(pid, pname, email, mob, dob,status)

**values**(1002,'pname2', 'pname1@abc.com', 9876543391, '12-jul-1987','A');

person

1001 pname1 pname1@abc.com 9876543321 12-JUL-87 A sri lanka

1002 pname2 pname2@abc.com 9876543391 12-JUL-87 A india

**insertinto** usr2

**values**(10001, 'username1','pass@1', 1001);

**insertinto** usr2

**values**(10002, 'username2','pass@2', 1002);

usr2

10001 username1 pass@1 1001

10002 username2 pass@2 1002

**Constraint at table level**

**createtable** person1 (

pid number(10),

pname varchar2(20) **notnull**,

email varchar2(50),

mob number(10),

dob **datenotnull**,

status varchar2(1) ,

country varchar2(20) **default**'india',

**constraint** pk\_person1 **primarykey**(pid),

**constraint** uk\_person1\_email **unique**(email),

**constraint** uk\_person1\_mob **unique**(mob),

**constraint** ck\_person1\_sts **check** (status **in** ('A','I') )

);

**createtable** usr3(

usrid number(10) ,

username varchar2(20) ,

pass varchar2(20) ,

person\_id number(10) ,

**constraint** pk\_usr3 **primarykey**(usrid),

**constraint** uk\_usr3\_uname **unique**(username),

**constraint** ck\_usr3\_pass **check** (pass **isnotnulland** pass **like**'%@%'),

**constraint** fk\_person1\_id **foreignkey** (person\_id) **references** person(pid)

);

Constraints in detail:

|  |  |  |
| --- | --- | --- |
| **Constraint Type** | **Definition** | **Restrictions** |
| Not Null |  | **Restrictions on NOT NULL Constraints NOT** **NULL** constraints are subject to the following restrictions:   * You cannot specify **NULL** or **NOT** **NULL** in a view constraint. * You cannot specify **NULL** or **NOT** **NULL** for an attribute of an object. Instead, use a **CHECK** constraint with the **IS** [**NOT**] **NULL** condition. |
| Unique |  | **Restrictions on Unique Constraints**Unique constraints are subject to the following restrictions:   * None of the columns in the unique key can be of LOB, **LONG**, **LONG** **RAW**, **VARRAY**, **NESTED** **TABLE**, **OBJECT**, **REF**, **TIMESTAMP** **WITH** **TIME** **ZONE,** or user-defined type. However, the unique key can contain a column of **TIMESTAMP** **WITH** **LOCAL** **TIME** **ZONE**. * A composite unique key cannot have more than 32 columns. * You cannot designate the same column or combination of columns as both a primary key and a unique key. * You cannot specify a unique key when creating a subview in an inheritance hierarchy. The unique key can be specified only for the top-level (root) view. |
| Primary Key |  | * No primary key value can appear in more than one row in the table. * No column that is part of the primary key can contain a null.   When you create a primary key constraint:  ■ Oracle Database uses an existing index if it contains a unique set of values before  enforcing the primary key constraint. The existing index can be defined as unique  or nonunique. When a DML operation is performed, the primary key constraint is  enforced using this existing index.  ■ If no existing index can be used, then Oracle Database generates a unique index.  When you drop a primary key constraint:  ■ If the primary key was created using an existing index, then the index is not  dropped.  ■ If the primary key was created using a system-generated index, then the index is  dropped.  **Restrictions on Primary Key Constraints**Primary constraints are subject to the following restrictions:   * A table or view can have only one primary key. * None of the columns in the primary key can be LOB, **LONG**, **LONG** **RAW**, **VARRAY**, **NESTED** **TABLE**, **BFILE**, **REF**, **TIMESTAMP** **WITH** **TIME** **ZONE**, or user-defined type. However, the primary key can contain a column of **TIMESTAMP** **WITH** **LOCAL** **TIME** **ZONE**. * The size of the primary key cannot exceed approximately one database block. * A composite primary key cannot have more than 32 columns. * You cannot designate the same column or combination of columns as both a primary key and a unique key. * You cannot specify a primary key when creating a subview in an inheritance hierarchy. The primary key can be specified only for the top-level (root) view. |
| Check | CREATE TABLE divisions  (div\_no NUMBER CONSTRAINT check\_divno  CHECK (div\_no BETWEEN 10 AND 99)  DISABLE,  div\_name VARCHAR2(9) CONSTRAINT check\_divname  CHECK (div\_name = UPPER(div\_name))  DISABLE,  office VARCHAR2(10) CONSTRAINT check\_office  CHECK (office IN ('DALLAS','BOSTON',  'PARIS','TOKYO'))  DISABLE  salary NUMBER(7,2),  CONSTRAINT check\_sal CHECK (salary \* commission\_pct <= 5000)  ); | **Restrictions on Check Constraints**Check constraints are subject to the following restrictions:   * You cannot specify a check constraint for a view. However, you can define the view using the **WITH** **CHECK** **OPTION** clause, which is equivalent to specifying a check constraint for the view. * The condition of a check constraint can refer to any column in the table, but it cannot refer to columns of other tables. * Conditions of check constraints cannot contain the following constructs: * Subqueries and scalar subquery expressions * Calls to the functions that are not deterministic (**CURRENT\_DATE**, **CURRENT\_TIMESTAMP**, **DBTIMEZONE**, **LOCALTIMESTAMP**, **SESSIONTIMEZONE**,**SYSDATE**, **SYSTIMESTAMP**, **UID**, **USER**, and **USERENV**) * Calls to user-defined functions * Dereferencing of **REF** columns (for example, using the **DEREF** function) * Nested table columns or attributes * The pseudocolumns **CURRVAL**, **NEXTVAL**, **LEVEL**, or **ROWNUM** * Date constants that are not fully specified |
| Foreign Key |  | **Restrictions on Foreign Key Constraints**Foreign key constraints are subject to the following restrictions:   * None of the columns in the foreign key can be of LOB, **LONG**, **LONG** **RAW**, **VARRAY**, **NESTED** **TABLE**, **BFILE**, **REF**, **TIMESTAMP** **WITH** **TIME** **ZONE**, or user-defined type. However, the primary key can contain a column of **TIMESTAMP** **WITH** **LOCAL** **TIME** **ZONE**. * The referenced unique or primary key constraint on the parent table or view must already be defined. * A composite foreign key cannot have more than 32 columns. * The child and parent tables must be on the same database. To enable referential integrity constraints across nodes of a distributed database, you must use database triggers. See [CREATE TRIGGER](http://docs.oracle.com/cd/B28359_01/server.111/b28286/statements_7004.htm#i2235611). * If either the child or parent object is a view, then the constraint is subject to all restrictions on view constraints. See ["View Constraints"](http://docs.oracle.com/cd/B28359_01/server.111/b28286/clauses002.htm#i1002565). * You cannot define a foreign key constraint in a **CREATE** **TABLE** statement that contains an **AS** ***subquery*** clause. Instead, you must create the table without the constraint and then add it later with an **ALTER** **TABLE** statement. |
| Default |  |  |
| **DISABLE Clause** Specify DISABLE to disable the integrity constraint. Disabled  integrity constraints appear in the data dictionary along with enabled constraints. If  you do not specify this clause when creating a constraint, then Oracle automatically  enables the constraint.  **ENABLE Clause**Specify **ENABLE** if you want the constraint to be applied to the data in the table  **Restriction on the ENABLE Clause**You cannot enable a foreign key that references a disabled unique or primary key. | | |

**JOINS**

* Join is used to fetch data from multiple tables.
* A [SQL](https://en.wikipedia.org/wiki/SQL) join clause combines [records](https://en.wikipedia.org/wiki/Row_(database)) from two or more [tables](https://en.wikipedia.org/wiki/Table_(database)) in a relational [database](https://en.wikipedia.org/wiki/Database).
* types join
  + - Inner Join (Simple join)
      * Equi join
      * Natural join
    - Outer Join
      * Left outer join
      * Right outer join
      * Full outer join
    - Self join
    - Cartesian product or cross join
* ANSI-standard SQL specifies five types of JOIN:
  + - INNER
    - LEFT OUTER
    - RIGHT OUTER
    - FULL OUTER
    - CROSS
    - NATURAL

Syntax of Join queries:

**select** a.col1, a.col2, b.col1

**from** table1 a join\_type table2 b

**on** a.col1 = b.col1;

**select** a.col1, a.col2, b.col1

**from** table1 a join\_type table2 b

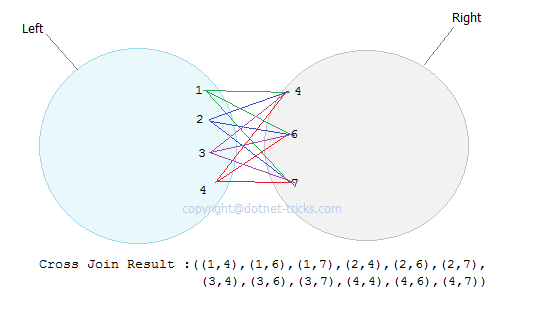
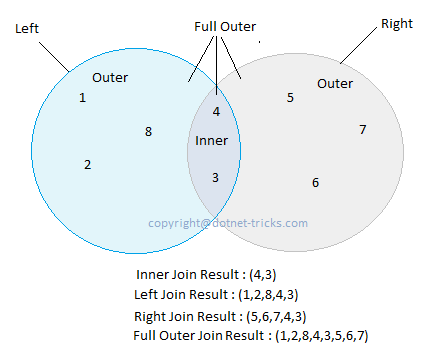
**using** (column\_name);

When both table have same column name on which join condition has to established, then we should use **using** otherwise we should use **on**.

* **For cross join or Cartesian join**

**select** a.col1, a.col2, b.col1

**from** table1 a table2 b;



Explanation with example of all join types

**CREATETABLE** department

(

DepartmentID **INT**,

DepartmentName **VARCHAR**(20)

);

**CREATETABLE** employee

(

LastName **VARCHAR**(20),

DepartmentID **INT**

);

**INSERTINTO** department **VALUES**(31, 'Sales');

**INSERTINTO** department **VALUES**(33, 'Engineering');

**INSERTINTO** department **VALUES**(34, 'Clerical');

**INSERTINTO** department **VALUES**(35, 'Marketing');

**INSERTINTO** employee **VALUES**('Rafferty', 31);

**INSERTINTO** employee **VALUES**('Jones', 33);

**INSERTINTO** employee **VALUES**('Heisenberg', 33);

**INSERTINTO** employee **VALUES**('Robinson', 34);

**INSERTINTO** employee **VALUES**('Smith', 34);

**INSERTINTO** employee **VALUES**('Williams', **NULL**);

**CROSS JOIN**

* CROSS JOIN returns the [Cartesian product](https://en.wikipedia.org/wiki/Cartesian_product) of rows from tables in the join.
* In other words, it will produce rows which combine each row from the first table with each row from the second table.

**SELECT** \*

**FROM** employee **CROSSJOIN** department;

Or,

**SELECT** \*

**FROM** employee, department;

Result:

|  |  |  |  |
| --- | --- | --- | --- |
| **Employee.LastName** | **Employee.DepartmentID** | **Department.DepartmentName** | **Department.DepartmentID** |
| Rafferty | 31 | Sales | 31 |
| Jones | 33 | Sales | 31 |
| Heisenberg | 33 | Sales | 31 |
| Smith | 34 | Sales | 31 |
| Robinson | 34 | Sales | 31 |
| Williams | NULL | Sales | 31 |
| Rafferty | 31 | Engineering | 33 |
| Jones | 33 | Engineering | 33 |
| Heisenberg | 33 | Engineering | 33 |
| Smith | 34 | Engineering | 33 |
| Robinson | 34 | Engineering | 33 |
| Williams | NULL | Engineering | 33 |
| Rafferty | 31 | Clerical | 34 |
| Jones | 33 | Clerical | 34 |
| Heisenberg | 33 | Clerical | 34 |
| Smith | 34 | Clerical | 34 |
| Robinson | 34 | Clerical | 34 |
| Williams | NULL | Clerical | 34 |
| Rafferty | 31 | Marketing | 35 |
| Jones | 33 | Marketing | 35 |
| Heisenberg | 33 | Marketing | 35 |
| Smith | 34 | Marketing | 35 |
| Robinson | 34 | Marketing | 35 |
| Williams | NULL | Marketing | 35 |

* The results of a cross join can be filtered by using a WHERE clause which may then produce the equivalent of an inner join.

**Self Join**

A **self join** is a join of a table to itself. This table appears twice in the FROM clause and is followed by table aliases that qualify column names in the join condition. To perform a self join, Oracle combines and returns rows of the table that satisfy the join condition.

For example the following query returns employee names and their manager names for whom they are working.

Select e.empno, e.ename, m.ename  “Manager” from emp e,

emp m where e.mgrid=m.empno

### Example[[edit](https://en.wikipedia.org/w/index.php?title=Join_(SQL)&action=edit&section=13)]

A query to find all pairings of two employees in the same country is desired. If there were two separate tables for employees and a query which requested employees in the first table having the same country as employees in the second table, a normal join operation could be used to find the answer table. However, all the employee information is contained within a single large table.[[10]](https://en.wikipedia.org/wiki/Join_(SQL)#cite_note-10)

Consider a modified Employee table such as the following:

|  |  |  |  |
| --- | --- | --- | --- |
| Employee Table | | | |
| **EmployeeID** | **LastName** | **Country** | **DepartmentID** |
| 123 | Rafferty | Australia | 31 |
| 124 | Jones | Australia | 33 |
| 145 | Heisenberg | Australia | 33 |
| 201 | Robinson | United States | 34 |
| 305 | Smith | Germany | 34 |
| 306 | Williams | Germany | NULL |

An example solution query could be as follows:

SELECTF.EmployeeID,F.LastName,S.EmployeeID,S.LastName,F.Country

FROMEmployeeFINNERJOINEmployeeSONF.Country=S.Country

WHEREF.EmployeeID<S.EmployeeID

ORDERBYF.EmployeeID,S.EmployeeID;

Which results in the following table being generated.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Employee Table after Self-join by Country | | | | |
| **EmployeeID** | **LastName** | **EmployeeID** | **LastName** | **Country** |
| 123 | Rafferty | 124 | Jones | Australia |
| 123 | Rafferty | 145 | Heisenberg | Australia |
| 124 | Jones | 145 | Heisenberg | Australia |
| 305 | Smith | 306 | Williams | Germany |

The condition F.EmployeeID < S.EmployeeID excludes pairings where the EmployeeID of the first employee is greater than or equal to the EmployeeID of the second employee. In other words, the effect of this condition is to exclude duplicate pairings and self-pairings.

**Inner Join**

* 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.
* The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied by matching non-NULL values, column values for each matched pair of rows of A and B are combined into a result row.
* The result of the join can be defined as the outcome of first taking the [Cartesian product](https://en.wikipedia.org/wiki/Cartesian_product) (or [Cross join](https://en.wikipedia.org/wiki/Join_%28SQL%29#Cross_join)) of all records in the tables (combining every record in table A with every record in table B) and then returning all records which satisfy the join predicate. Actual SQL implementations normally use other approaches, such as [hash joins](https://en.wikipedia.org/wiki/Hash_join) or [sort-merge joins](https://en.wikipedia.org/wiki/Sort-merge_join), since computing the Cartesian product is slower and would often require a prohibitively large memory space to store.
* SQL specifies two different syntactical ways to express joins: "explicit join notation" and "implicit join notation". Although "implicit join notation" was deprecated in 1992, and its use is not considered a best practice, database systems still support it.
* The "explicit join notation" uses the **JOIN** keyword, optionally preceded by the **INNER** keyword, to specify the table to join, and the **ON** keyword to specify the predicates for the join, as in the following example:

**SELECT** \*

**FROM** employee

**INNERJOIN** department **ON** employee.DepartmentID = department.DepartmentID;

* The "implicit join notation" simply lists the tables for joining, in the FROM clause of the SELECT statement, using commas to separate them. Thus it specifies a [cross join](https://en.wikipedia.org/wiki/Join_%28SQL%29#Cross_join), and the WHERE clause may apply additional filter-predicates (which function comparably to the join-predicates in the explicit notation).

**SELECT** \*

**FROM** employee, department

**WHERE** employee.DepartmentID = department.DepartmentID;

|  |  |  |  |
| --- | --- | --- | --- |
| **Employee.LastName** | **Employee.DepartmentID** | **Department.DepartmentName** | **Department.DepartmentID** |
| Robinson | 34 | Clerical | 34 |
| Jones | 33 | Engineering | 33 |
| Smith | 34 | Clerical | 34 |
| Heisenberg | 33 | Engineering | 33 |
| Rafferty | 31 | Sales | 31 |

**Equi Join:**

An **equi-join** is a specific type of comparator-based join, that uses only [equality](https://en.wikipedia.org/wiki/Equality_(mathematics)) comparisons in the join-predicate. Using other comparison operators (such as <) disqualifies a join as an equi-join.

SELECT \*

FROM employee JOIN department

ON employee.DepartmentID = department.DepartmentID;

We can write equi-join as below,

SELECT \*

FROM employee, department

WHERE employee.DepartmentID = department.DepartmentID;

If columns in an equi-join have the same name, [SQL-92](https://en.wikipedia.org/wiki/SQL-92) provides an optional shorthand notation for expressing equi-joins, by way of the USING construct:[[6]](https://en.wikipedia.org/wiki/Join_(SQL)#cite_note-6)

SELECT \*

FROM employee INNER JOIN department USING (DepartmentID);

## Non Equi Joins.

Non equi joins is used to return result from two or more tables where exact join is not possible.

For example we have emp table and salgrade table. The salgrade table contains grade and their low salary and high salary. Suppose you want to find the grade of employees based on their salaries then you can use NON EQUI join.

select e.empno, e.ename, e.sal, s.grade from emp e, salgrade s where e.sal between s.lowsal and s.hisal

**Natural Join**

A [natural join](https://en.wikipedia.org/wiki/Natural_join) is a type of equi-join where the **join** predicate arises implicitly by comparing all columns in both tables that have the same column-names in the joined tables. The resulting joined table contains only one column for each pair of equally named columns. In the case that no columns with the same names are found, a [cross join](https://en.wikipedia.org/wiki/Cross_join) is performed.

SELECT \*

FROM employee NATURAL JOIN department;

**Outer Join:**

An **outer join** does not require each record in the two joined tables to have a matching record. The joined table retains each record—even if no other matching record exists. Outer joins subdivide further into left outer joins, right outer joins, and full outer joins, depending on which table's rows are retained (left, right, or both).

**Left Outer Join**

* The result of a *left outer join* (or simply **left join**) for tables A and B always contains all records of the "left" table (A), even if the join-condition does not find any matching record in the "right" table (B).
* A **left outer join** returns all the values from an inner join plus all values in the left table that do not match to the right table, including rows with NULL (empty) values in the link field.
* Example of a left outer join (the **OUTER** keyword is optional)

**SELECT** \*

**FROM** employee

**LEFTOUTERJOIN** department **ON** employee.DepartmentID = department.DepartmentID;

**Oracle supports the deprecated syntax:**

**SELECT** \*

**FROM** employee, department

**WHERE** employee.DepartmentID = department.DepartmentID(+);

|  |  |  |  |
| --- | --- | --- | --- |
| **Employee.LastName** | **Employee.DepartmentID** | **Department.DepartmentName** | **Department.DepartmentID** |
| Jones | 33 | Engineering | 33 |
| Rafferty | 31 | Sales | 31 |
| Robinson | 34 | Clerical | 34 |
| Smith | 34 | Clerical | 34 |
| *Williams* | NULL | NULL | NULL |
| Heisenberg | 33 | Engineering | 33 |

**Right Outer Join**

* The result of a *right outer join* (or simply **right join**) for tables A and B always contains all records of the "right" table (A), even if the join-condition does not find any matching record in the "left" table (B).
* A **right outer join** returns all the values from an inner join plus all values in the right table that do not match to the left table, including rows with NULL (empty) values in the link field.
* Example of a right outer join (the **OUTER** keyword is optional)

**SELECT** \*

**FROM** employee **RIGHTOUTERJOIN** department

**ON** employee.DepartmentID = department.DepartmentID;

**Oracle supports the deprecated syntax:**

**SELECT** \*

**FROM** employee, department

**WHERE** employee.DepartmentID(+) = department.DepartmentID;

|  |  |  |  |
| --- | --- | --- | --- |
| **Employee.LastName** | **Employee.DepartmentID** | **Department.DepartmentName** | **Department.DepartmentID** |
| Smith | 34 | Clerical | 34 |
| Jones | 33 | Engineering | 33 |
| Robinson | 34 | Clerical | 34 |
| Heisenberg | 33 | Engineering | 33 |
| Rafferty | 31 | Sales | 31 |
| NULL | NULL | *Marketing* | *35* |

### **Full outer join**

* A **full outer join** combines the effect of applying both left and right outer joins.
* A **right outer join** returns all the values

from an inner join

plus all values in the right table that do not match to the left table, including rows with NULL (empty) values in the link field

plus all values in the left table that do not match to the right table, including rows with NULL (empty) values in the link field.

* Example of a full outer join (the **OUTER** keyword is optional):

**SELECT** \*

**FROM** employee **FULLOUTERJOIN** department

**ON** employee.DepartmentID = department.DepartmentID;

|  |  |  |  |
| --- | --- | --- | --- |
| **Employee.LastName** | **Employee.DepartmentID** | **Department.DepartmentName** | **Department.DepartmentID** |
| Smith | 34 | Clerical | 34 |
| Jones | 33 | Engineering | 33 |
| Robinson | 34 | Clerical | 34 |
| *Williams* | NULL | NULL | NULL |
| Heisenberg | 33 | Engineering | 33 |
| Rafferty | 31 | Sales | 31 |
| NULL | NULL | *Marketing* | *35* |

References for join:

<http://www.oracle-dba-online.com/sql/join_queries.htm>

<https://en.wikipedia.org/wiki/Join_(SQL)>

**Sub Queries:**

* Sub query in the “from” clause => inline view
* Sub query in the “where” clause => nested sub query => 255 level can be used in where clause.
* Correlated sub query:

When a nested sub query references a column from a table referred to a parent statement.

* Correlated sub query execution:
  + 1. First row of the outer query fetched.
    2. Inner query is processed using the outer query’s value.
    3. Outer query is processed using the inner query’s value
    4. Process is continued until the outer query is done.
* Find nth highest:

**select** \* **from** account acc

**where** n = (**select**count(**distinct** bal) **from** account **where** acc.bal <= bal);

* Find nth lowest

**select** \* **from** account acc

**where** n = (**select**count(**distinct** bal) **from** account **where** acc.bal >= bal);

* Nested sub query execution:

Inner most subquery will be executed first and then based on the result the outer subquery will be executed.

* **2nd highest**

**select** \* **from** (**select** \* **from** account **where** balance< (**select**max(balance) **from** account) **orderby** balance **desc**) **where** rownum = 1;

**Create table from another table:**

* With all columns

**createtable** cust1 **asselect** \* **from** customer;

* With selected columns

**createtable** cust2 **asselect** cid,cname,status **from** customer;

* With some condition

**createtable** cust1 **asselect** \* **from** customer where cid <100001;

* Without data

**createtable** cust1 **asselect** \* **from** customer where 1=2;

**Index:**

* Order list of element belongs to one column or combination of two or more columns.

**Simple index**

**createindex** i1

**on** customer(email);

**Composite index**

**createindex** i2

**on** customer (cname, status, age);

* When you create index the data get stored and to find the element better search algorithm works. (speed up the select queries).
* For primary key by default index is created.
* One table can have multiple index.

**Dropping index**

**dropindex** i1; (**dropindex** index\_name;)

**Rename index**

**alterindex** index\_name

rename to new\_index\_name;

**Sequence:**

**create** sequence sequenc\_name

start **with** int\_value

minvalue int\_value

maxvalue int\_value

increment **by** int\_value

cache **value**;

currval => get current value of sequence

nextval => get next value of sequence

for eg:

**create** sequence cid\_seq start **with** 1001 increment **by** 1;

**insertinto** customer **values** (cid\_seq.nextval, 'suraj', 'Bangalore');

* **Dropping sequence**

**drop** sequence sequence\_name;

**View**

* View is the logical entity derived from other tables.
* A view doesn’t necessarily exist in physical form. It’s considered a virtual table.

**createview** view\_name

**as** select\_statement;

**for eg:**

**createview** acc\_view

**asselect** sid, feepaid **from** student;

**Dropping View:**

**dropview** view\_name;

**PL/SQL** (Procedural Language/Structured Query Language)

* **PL/SQL** stands for **Procedural Language** extension of SQL.
* PL/SQL offers the way to write the program to perform some conditional checks, loop statement, writing function, procedure and triggers.
* pl/sql is a stored procedure language specific to Oracle. Different databases use different languages for stored procedures.
* Elements of PL/SQL
  + - PL/SQL Variables
    - PL/SQL Constants
    - PL/SQL blocks
    - If statement
    - Loop statement
    - Stored procedure
    - Triggers

**PL/SQL Variables:**

**variable\_name data\_type(size);**

eg:

a number;

**variable\_name data\_type(size) := value ;**

eg:

b number(10) := 20;

**PL/SQL Constants:**

**Variable\_name constant data\_type(size) := value;**

Eg:

c constant number(10) := 34;

**PL/SQL blocks**

**declare**

-- pl/sql variables

-- pl/sql constant

**begin**

--statements

--statements

--statements

**end**;

/

**Question:**

Write a pl/sql block for the following tasks:

* 1. Declare 3 pl/sql variables.
  2. Assign values to first to variables.
  3. Find the sum and store in 3rd value.
  4. Display the sum.

**Answer:**

sql>**set**serveroutput **on**;

**declare**

a number(5) := 200;

b number(5) := 408;

c number(6);

**begin**

c := a + b;

dbms\_output.put\_line('Sum of a and b =' || c);

**end**;

/

Sum of a and b =608

PL/SQL procedure successfully completed.

**SQL> ed**

* Whatever currently available in the oracle buffer, will display in the notepad.
* You can modify and execute query.
* The query will be copied to the command prompt.

**If Statements**

**Syntax:**

* **===**

if (condition) **then**

-- st1

-- st2

-- st3

**end** if;

* ===

if (condition) **then**

-- st1

else

-- st2

-- st3

**end** if;

* ===

if (condition) **then**

-- st1

elsif(condition) **then**

-- st2

else

-- st3

**end** if;

Example:

Question:

Declare three variables which has been assigned by marks and then find average marks. If average is greater equal to than 80 then display the result as Excellent, if average is between 80 to 70 then display the result as good, if average is between 70 to 60 then display result as satisfactory otherwise display result as not satisfactory.

Answer:

**declare**

m1 number(3);

m2 number(3);

m3 number(3);

mavg number(3);

**begin**

m1:= 60;

m2:= 60;

m3:= 60;

mavg := (m1+m2+m3)/3;

dbms\_output.put\_line('Average marks:' || mavg);

if(mavg >=80) **then**

dbms\_output.put\_line('Result:' || 'EXCELLENT');

elsif(mavg **between** 70 **and** 79) **then**

dbms\_output.put\_line('Result:' || 'GOOD');

elsif(mavg **between** 60 **and** 69) **then**

dbms\_output.put\_line('Result:' || 'SATISFACTORY');

else

dbms\_output.put\_line('Result:' || 'NOT SATISFACTORY');

**end** if;

**end**;

/

Average marks:60

Result:SATISFACTORY

PL/SQL procedure successfully completed.

Loop Statements:

1. Basic loop:

Syntax:

**loop**

**--s1;**

**--s2;**

**exit when(condition)**

**end loop;**

1. While Loop:

Syntax:

**while (condition)**

**loop**

**--s1;**

**--s2;**

**end loop**;

1. For loop

Syntax:

**for variable in minvalue .. maxvalue**

**loop**

**--st1**

**--st2**

**end loop;**

1. For loop in reverse order:

Syntax:

**for variable in reverse minvalue .. maxvalue**

**loop**

**--st1**

**--st2**

**end loop;**

**Note:**

SQL> begin

2 var number(2):=1;

3 end;

4 /

var number(2):=1;

\*

ERROR at line 2:

ORA-06550: line 2, column 6:

PLS-00103: Encountered the symbol "NUMBER" when expecting one of the following:

:= . ( @ % ;

The symbol ":=" was substituted for "NUMBER" to continue.

ORA-06550: line 2, column 15:

PLS-00103: Encountered the symbol "=" when expecting one of the following:

. ( \* % & = - + ; < / > at in is mod remainder not rem

<an exponent (\*\*)><> or != or ~= >= <= <> and or like like2

like4 likec between || multiset membe

**declare**

var number(2);

**begin**

var :=1;

**end**;

Loop in detail:

There are three types of loops in pl/sql

• Simple Loop  
• While Loop  
• For Loop

|  |  |  |
| --- | --- | --- |
| **Simple Loop** | **While Loop** | **For Loop** |
| **LOOP**  **statements;**  **EXIT;**  **{or EXIT WHEN condition;}**  **END LOOP;** | **WHILE <condition>**  **LOOP statements;**  **END LOOP;** | **FOR counter IN val1..val2**  **LOOP statements;**  **END LOOP;** |
| 1) Initialise a variable before the loop body. 2) Increment the variable in the loop. 3) Use a EXIT WHEN statement to exit from the Loop. If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once. | 1) Initialise a variable before the loop body. 2) Increment the variable in the loop. 3) EXIT WHEN statement and EXIT statements can be used in while loops but it's not done often. | val1 - Start integer value.  val2 - End integer value.  1) The counter variable is implicitly declared in the declaration section, so it's not necessary to declare it explicitly. 2) The counter variable is incremented by 1 and does not need to be incremented explicitly. 3) EXIT WHEN statement and EXIT statements can be used in FOR loops but it's not done often. |

Example:

1.

DECLARE

-- variable declaration

message varchar2(30):= 'Printing from 1 to 10';

i number(10,0) := 1;

BEGIN

dbms\_output.put\_line(message);

**loop**

dbms\_output.put\_line(i);

i := i +1;

**exit when** i >10;

**end loop;**

END;

/

**Output:**

SQL> @example;

Printing from 1 to 10

1

2

3

4

5

6

7

8

9

10

PL/SQL procedure successfully completed.

2.

DECLARE

msg varchar(30) := 'Printing using while loop';

dis varchar(4) := '\*';

i number(10,0) := 1;

BEGIN

DECLARE

j number(10,0) := 1;

BEGIN

dbms\_output.put\_line(MSG);

WHILE i <= 10 LOOP

dbms\_output.put\_line(dis || i );

i := i + 1;

END LOOP;

END;

END;

/

**Output:**

SQL> @example;

Printing using while loop

\*1

\*2

\*3

\*4

\*5

\*6

\*7

\*8

\*9

\*10

PL/SQL procedure successfully completed.

3.

DECLARE

BEGIN

FOR counter IN 1 .. 10

LOOP

DBMS\_OUTPUT.PUT\_LINE(COUNTER);

END LOOP;

END;

/

**OUTPUT:**

SQL> @example;

1

2

3

4

5

6

7

8

9

10

PL/SQL procedure successfully completed.

**4. Real time example of FOR loop use.**

begin

for rec in (select \* from USER\_TAB\_COLUMNS where COLUMN\_NAME='LAST\_UPDATED\_DATE') loop

execute immediate 'update ' || rec.TABLE\_NAME || ' set LAST\_UPDATED\_DATE=CREATED\_DATE where LAST\_UPDATED\_DATE is null';

end loop;

end;

/

**5. Reverse For loop**

BEGIN

FORnoIN REVERSE 1..5

LOOP

DBMS\_OUTPUT.PUT\_LINE('Iteration : '||no);

ENDLOOP;

END;

/

Iteration : 5

Iteration : 4

Iteration : 3

Iteration : 2

Iteration : 1

PL/SQL procedure successfully completed.

**Stored procedure**

**CREATE** [**OR**REPLACE] **PROCEDURE** [**SCHEMA**..] procedure\_name

[ (parameter [,parameter]) ]

**IS**

[ declaration\_section

variable declarations;

**constant** declarations;

]

**BEGIN**

[ executable\_section

PL/SQL execute/subprogram body

]

[EXCEPTION]

[exception\_section

PL/SQL Exception block

]

**END** [procedure\_name];

/

Example:

Table: Books

SQL> edit pro1

**CREATEor**REPLACE**PROCEDURE** pro1(**noin** number,temp out books%rowtype)

**IS**

**BEGIN**

**SELECT** \* **INTO** temp **FROM** books **WHERE** bid = **no**;

**END**;

/

SQL> @pro1

Procedure created.

--call the procedure through another pl/sql block.

SQL> edit pro

**DECLARE**

temp books%rowtype;

**no** number := &**no**;

**BEGIN**

pro1(**no**,temp);

dbms\_output.put\_line(temp.bid||' '||

temp.bname||' '||

temp.cost||' '||

temp.publisher||' ');

**END**;

/

SQL> @pro

Enter value for no: 1001

old 3: no number := &no;

new 3: no number := 1001;

1001 xyz 500 ccc

PL/SQL procedure successfully completed.

**Example of IN OUT parameter:**

SQL> edit pro2

**createor**replace**procedure** pro2(n1 **in** number, n2 **in** out number)

**is**

n3 number;

**begin**

n3 := n1+n2;

n2:= n3;

**end**;

/

SQL> @pro2

Procedure created.

SQL> edit pro3

**declare**

n1 number := &n1;

n2 number := &n2;

**begin**

pro2(n1,n2);

dbms\_output.put\_line('sum =' || n2);

**end**;

/

SQL> @pro3

Enter value for n1: 11

old 2: n1 number := &n1;

new 2: n1 number := 11;

Enter value for n2: 33

old 3: n2 number := &n2;

new 3: n2 number := 33;

sum =44

PL/SQL procedure successfully completed.

**Drop Procedure**

You can drop PL/SQL procedure using DROP PROCEDURE statement,

DROP PROCEDURE procedure\_name;

Drop example:

**SQL>DROP PROCEDURE pro1;**  
  
Procedure dropped.

**Triggers**

<http://plsql-tutorial.com/plsql-procedures.htm>

<http://www.way2tutorial.com/plsql/plsql_procedures.php>

* A trigger is a pl/sql block structure which is fired automatically when a DML statements like Insert, Delete, Update is executed on a database table.
* Syntax of Trigger:

**CREATE** [**OR**REPLACE] TRIGGER trigger\_name

{BEFORE | AFTER | INSTEAD OF }

{**INSERT** [**OR**] | **UPDATE** [**OR**] | **DELETE**}

[OF col\_name]

**ON** table\_name

{FOR EACH ROW | FOR EACH STATEMENT}

[ REFERENCING OLD **AS** O | NEW **AS** N | PARENT **AS** P ]

[ WHEN Condition ]

**DECLARE**

[declaration\_section

variable declarations;

**constant** declarations;

]

**BEGIN**

[executable\_section

PL/SQL execute/subprogram body

]

EXCEPTION

[exception\_section

PL/SQL Exception block

]

**END**;

* *CREATE [OR REPLACE ] TRIGGER trigger\_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
* *{BEFORE | AFTER | INSTEAD OF }* - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
* *{INSERT [OR] | UPDATE [OR] | DELETE}* - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
* *[OF col\_name]* - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
* *[ON table\_name]* - This clause identifies the name of the table or view to which the trigger is associated.
* *[REFERENCING OLD AS o NEW AS n]* - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column\_name or :new.column\_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.
* *[FOR EACH ROW]* - This clause is used to determine whether a trigger must fire when each row gets affected ( i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).
* *WHEN (condition)* - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

For example:

Example1:

SQL> edit trg1

**createor**replace trigger trg1

before

**insert**

**on** books

for each row

**begin**

:new.bname := upper(:new.bname);

**end**;

/

SQL> @trg1

Trigger created.

SQL> insert into books values (1006,'abcdef',3000,'dfdfd','02-apr-15',104);

1 row created.

SQL> select \* from books where bid = 1006;

BID BNAME COST PUBLISHER DOP AUTH\_ID

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

1006 ABCDEF 3000 dfdfd 02-APR-15 104

**Example2:**

SQL> edit trg2

**CREATEor**REPLACE TRIGGER trg2

AFTER

**DELETE**

**ON** books

FOR EACH ROW

**BEGIN**

-- IF (:old.bid = 1006) THEN

-- raise\_application\_error(-20015, 'You can't delete this row');

-- END IF;

dbms\_output.put\_line('deleted Column bid:'|| :old.bid);

**END**;

/

SQL> @trg2

Trigger created.

SQL> delete from books where bid = 1005;

deleted Column bid:1005

1 row deleted.

**Example3:**

SQL> edit trg3

**createor**replace trigger trg3

before

**insertorupdateordelete**

**on** books

for each row

**begin**

if inserting **then**

dbms\_output.put\_line('row inserted with bid:'|| :new.bid);

elsif updating **then**

dbms\_output.put\_line('row updated with bid:' || :new.bid);

elsif deleting **then**

dbms\_output.put\_line('row deleted with bid'|| :old.bid);

**end** if;

**end**;

/

SQL> @trg3

Trigger created.

SQL> insert into books values(1005,'abcdef',3000,'dfdfd','02-apr-15',104);

row inserted with bid:1005

1 row created.

SQL> update books set cost=2000 where bid = 1005;

row updated with bid:1005

1 row updated.

SQL> select \* from books where bid = 1005;

BID BNAME COST PUBLISHER DOP AUTH\_ID

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

1005 ABCDEF 2000 dfdfd 02-APR-15 104

SQL> delete from books where bid = 1005;

row deleted with bid1005

delete coulmn bid:1005

1 row deleted.

* There are two types of triggers based on which the level it is triggered.  
  **1) Row level trigger** - An event is triggered for each row upated, inserted or deleted.   
  **2) Statement level trigger** - An event is triggered for each sql statement executed.
* The following hierarchy is followed when a trigger is fired.  
  **1)** BEFORE statement trigger fires first.  
  **2)** Next BEFORE row level trigger fires, once for each row affected.   
  **3)** Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.  
  **4)** Finally the AFTER statement level trigger fires.
* The below statement shows the structure of the view 'USER\_TRIGGERS'

DESC USER\_TRIGGERS;

* This view stores information about header and body of the trigger.

SELECT \* FROM user\_triggers WHERE trigger\_name = 'trg2';

* drop a trigger using the following command.

DROP TRIGGER trigger\_name;

Question: How do I execute a SQL script file in SQLPlus?

Answer: To execute a script file in SQLPlus, type @ and then the file name.

SQL > @{file}

For example, if your file was called script.sql, you'd type the following command at the SQL prompt:

SQL > @script.sql

The above command assumes that the file is in the current directory. (ie: the current directory is usually the directory that you were located in before you launched SQLPlus.)

If you need to execute a script file that is not in the current directory, you would type:

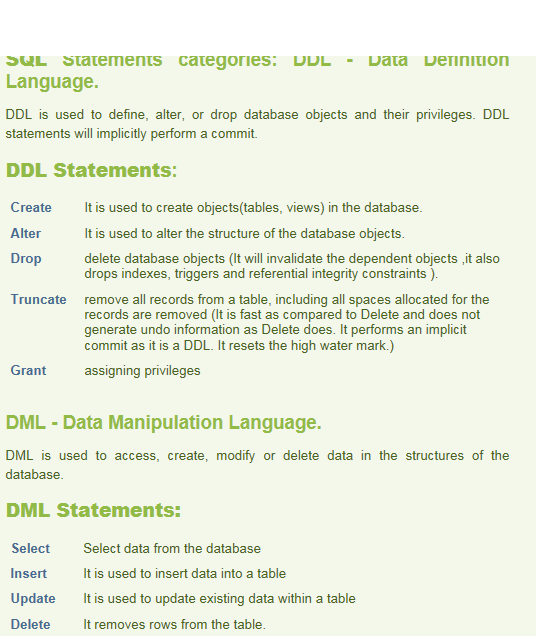
SQL > @{path}{file}

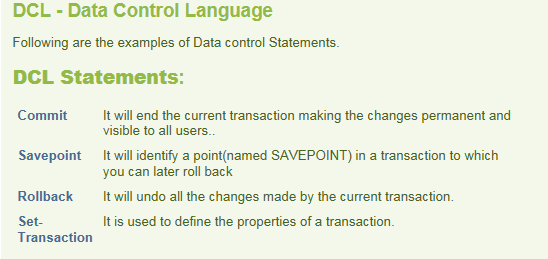
For example:

SQL > @/oracle/scripts/script.sql

This command would run a script file called script.sql that was located in the /oracle/scripts directory.

Types of sql statements





## SQL Command Categories

SQL commands are grouped into four major categories depending on their functionality. They are as follows:

### Data Definition Language (DDL)

These SQL commands are used for creating, modifying, and dropping the structure of database objects. The commands are CREATE, ALTER, DROP, RENAME, and TRUNCATE.

### Data Manipulation Language (DML)

These SQL commands are used for storing, retrieving, modifying, and deleting data. These commands are SELECT, INSERT, UPDATE, and DELETE.

### Transaction Control Language (TCL)

These SQL commands are used for managing changes affecting the data. These commands are COMMIT, ROLLBACK, and SAVEPOINT.

### Data Control Language (DCL)

These SQL commands are used for providing security to database objects. These commands are GRANT and REVOKE.

**Question:**

**oracle query output in excel file**

**SET MARKUP HTML ON ENTMAP ON SPOOL ON PREFORMAT OFF ;**

**SPOOL MY\_TEST\_FILE.xls;**

**SELECT \* FROM EMP;**

**SPOOL OFF;**

**select** \* **from** user\_sequences;

**select** \* **from** all\_sequences;

Both allow joining of "unlimited" tables. The difference is that USING requires the join columns to have the same name:

**select** emp.ename, dept.dname

**from** emp **join** dept **using** (deptno);

The ON version works also when the join columns have different names:

**select** emp.ename, emp2.ename manager\_name

**from** emp **join** emp emp2 **on** (emp.mgr = emp2.empno);

**Question: How to create users in Oracle?**

**Answer:**

**createuser** CPDV2 identified **by** CPDV2\_PASS;

grant **create** session to CPDV2;

grant **create** synonym to CPDV2;

grant **createprocedure** to CPDV2;

grant **createtable** to CPDV2;

grant **createview** to CPDV2;

grant **all** privileges to CPDV2;

grant DBA to CPDV2;

NOTE:

SQL\*Plus: Release 11.2.0.2.0 Production

Copyright (c) 1982, 2010, Oracle. All rights reserved.

Use SQL\*Plus to execute SQL, PL/SQL and SQL\*Plus statements.

Usage 1: sqlplus -H | -V

-H Displays the SQL\*Plus version and the

usage help.

-V Displays the SQL\*Plus version.

Usage 2: sqlplus [ [<option>] [{logon | /nolog}] [<start>] ]

<option> is: [-C <version>] [-L] [-M "<options>"] [-R <level>] [-

-C <version> Sets the compatibility of affected commands to t

version specified by <version>. The version has

the form "x.y[.z]". For example, -C 10.2.0

-L Attempts to log on just once, instead of

reprompting on error.

-M "<options>" Sets automatic HTML markup of output. The optio

have the form:

HTML [ON|OFF] [HEAD text] [BODY text] [TABLE tex

[ENTMAP {ON|OFF}] [SPOOL {ON|OFF}] [PRE[FORMAT]

-R <level> Sets restricted mode to disable SQL\*Plus command

that interact with the file system. The level c

be 1, 2 or 3. The most restrictive is -R 3 whic

disables all user commands interacting with the

file system.

-S Sets silent mode which suppresses the display of

the SQL\*Plus banner, prompts, and echoing of

commands.

<logon> is: {<username>[/<password>][@<connect\_identifier>] | / }

[AS {SYSDBA | SYSOPER | SYSASM}] [EDITION=value]

Specifies the database account username, password and connect

identifier for the database connection. Without a connect

identifier, SQL\*Plus connects to the default database.

The AS SYSDBA, AS SYSOPER and AS SYSASM options are database

administration privileges.

<connect\_identifier> can be in the form of Net Service Name

or Easy Connect.

@[<net\_service\_name> | [//]Host[:Port]/<service\_name>]

<net\_service\_name> is a simple name for a service that reso

to a connect descriptor.

Example: Connect to database using Net Service Name and the

database net service name is ORCL.

sqlplus myusername/mypassword@ORCL

Host specifies the host name or IP address of the database

server computer.

Port specifies the listening port on the database server.

<service\_name> specifies the service name of the database y

want to access.

Example: Connect to database using Easy Connect and the

Service name is ORCL.

sqlplus myusername/mypassword@Host/ORCL

The /NOLOG option starts SQL\*Plus without connecting to a

database.

The EDITION specifies the value for Session Edition.

<start> is: @<URL>|<filename>[.<ext>] [<parameter> ...]

Runs the specified SQL\*Plus script from a web server (URL) or t

local file system (filename.ext) with specified parameters that

will be assigned to substitution variables in the script.

When SQL\*Plus starts, and after CONNECT commands, the site profile

(e.g. $ORACLE\_HOME/sqlplus/admin/glogin.sql) and the user profile

(e.g. login.sql in the working directory) are run. The files may

contain SQL\*Plus commands.

**Change sys password as an administrator:**

sqlplus /nolog  
connect / as sysdba  
Once connected, you can change the SYS password to something you know:   
ALTER USER sys IDENTIFIED BY new\_password;

Set ORACLE\_HOME

**On Windows:**

D:\>set ORACLE\_HOME=C:\oraclexe\app\oracle\product\10.2.0\server

D:\>echo %ORACLE\_HOME%

C:\oraclexe\app\oracle\product\10.2.0\server

D:\>

**On Unix/Linux:**

export ORACLE\_HOME=/app/oracle/product/10.2.0/server

* **check the set ORACLE\_HOME**

**On Windows:** On command prompt, type D:\>echo %ORACLE\_HOME%. If this gives you the directory path, as in the code snippet below, then that means ORACLE\_HOME is set.

D:\>echo %ORACLE\_HOME%

C:\oraclexe\app\oracle\product\10.2.0\server

If ORACLE\_HOME is not set, the output will simply give back %ORACLE\_HOME%, as below. This means you need to set the value, which we’ll see how to in the next section.

D:\>echo %ORACLE\_HOME%

%ORACLE\_HOME%

**On Unix/Linux:** type

env | grep ORACLE\_HOME

Please check that the Connection String for the database exists in the TNSNAMES.ora File. Usually,  
it is located **$ORACLE\_HOME\ora81\network\ADMIN.**

**C:\apps\oraclexe\app\oracle\product\11.2.0\server\network\ADMIN**

**tnsnames.ora**

XE =

(DESCRIPTION =

(ADDRESS = (PROTOCOL = TCP)(HOST = CLISURAJKUMAR.snshub.org)(PORT = 1521))

(CONNECT\_DATA =

(SERVER = DEDICATED)

(SERVICE\_NAME = XE)

)

)

EXTPROC\_CONNECTION\_DATA =

(DESCRIPTION =

(ADDRESS\_LIST =

(ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC1))

)

(CONNECT\_DATA =

(SID = PLSExtProc)

(PRESENTATION = RO)

)

)

ORACLR\_CONNECTION\_DATA =

(DESCRIPTION =

(ADDRESS\_LIST =

(ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC1))

)

(CONNECT\_DATA =

(SID = CLRExtProc)

(PRESENTATION = RO)

)

)

[ORA-12560: TNS:protocol adaptor error](http://stackoverflow.com/questions/6894558/ora-12560-tnsprotocol-adaptor-error)

**On Windows:**

1. Go to the windows machine that hosts the Oracle database server
2. Go to Start -> Run -> Services.msc in windows. Locate OracleService (here OracleServiceORCL) and click on "Start" to start the oracle database service (if not already running)
3. Once it is up and running, from the command prompt run the following:

tnsping < tnsalias >

(tnsalias entry you can find it in tnsnames.ora file)

INSERT

INTO

births( BTH\_LIVE\_STILL,BTH\_CSO\_CODE,BTH\_YEAR, BTH\_SERIAL\_NO,BTH\_DUP\_NO,BTH\_SEX,BTH\_OLD\_NEW,BTH\_DATE\_CRT,BTH\_USER\_INPUT,BTH\_STATUS,BTH\_PERNO ) VALUES

('Y',12,12,12,12,'F','A', '12-12-12',11,'Y',11112

)

create table CSD.ERRORS\_LOG(

"BTH\_LIVE\_STILL" VARCHAR2(1 BYTE) ,

"BTH\_CSO\_CODE" NUMBER(2,0),

"BTH\_YEAR" NUMBER(4,0),

"BTH\_SERIAL\_NO" NUMBER(4,0) ,

"BTH\_DUP\_NO" NUMBER(2,0) ,

"BTH\_IDNO\_NOT\_UNIQUE" NUMBER(2,0),

"BTH\_PERNO\_NULL" NUMBER(2,0)

);

PL/SQL

1. The PL/SQL language is an extension of SQL language developed by Oracle Corporation and the full name is the Procedural Language/Structured Query Language.
2. Each PL/SQL program consists of SQL and PL/SQL statements which from a PL/SQL block.
3. PL/SQL Block consists of three sections:
   1. The Declaration section (optional).
   2. The Execution section (mandatory).
   3. The Exception (or Error) Handling section (optional).
4. Declaration Section:

The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE.

This section is optional and is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section.

1. **Executable Commands**  
   This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.
2. **Exception Handling**  
   This section starts with the keyword **EXCEPTION**. This section is again optional and contains exception(s) that handle errors in the program
3. Every PL/SQL statement ends with a semicolon **(;)**. PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**.
4. The **end;** line signals the end of the PL/SQL block. To run the code from SQL command line, you may need to type **/** at the beginning of the first blank line after the last line of the code.

DECLARE   
     Variable declaration  
BEGIN   
     Program Execution   
EXCEPTION   
     Exception handling  
END;

/

1. PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

1. The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by PL/SQL compiler. The PL/SQL single-line comments start with the delimiter **--** (double hyphen) and multi-line comments are enclosed by /\* and \*/.

**For** example:

DECLARE

-- variable declaration

message varchar2(20):= 'Hello, World!';

BEGIN

/\*

\* PL/SQL executable statement(s)

\*/

dbms\_output.put\_line(message);

END;

/

Note: By default, most tools do not configure a buffer for dbms\_output to write to and do not attempt to read from that buffer after code executes. Most tools, on the other hand, have the ability to do so. In SQL\*Plus, you'd need to use the command

set serveroutput on [size N|unlimited]

1. syntax to declare a constant:

|  |
| --- |
| constant\_name CONSTANT datatype := VALUE; |

* *constant\_name* is the name of the constant i.e. similar to a variable name.
* The word *CONSTANT* is a reserved word and ensures that the value does not change.
* *VALUE* - It is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

1. The syntax for declaring a variable is:

variable\_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial\_value]

When you provide a size, scale or precision limit with the data type, it is called a constrained declaration. Constrained declarations require less memory than unconstrained declarations. For example:

sales number(10, 2);

name varchar2(25);

address varchar2(100);

1. **Local variables** - variables declared in an inner block and not accessible to outer blocks.

**Global variables** - variables declared in the outermost block or a package.

DECLARE

-- Global variables

num1 number :=95;

num2 number :=85;

BEGIN

dbms\_output.put\_line('Outer Variable num1: '|| num1);

dbms\_output.put\_line('Outer Variable num2: '|| num2);

DECLARE

-- Local variables

num1 number :=195;

num2 number :=185;

BEGIN

dbms\_output.put\_line('Inner Variable num1: '|| num1);

dbms\_output.put\_line('Inner Variable num2: '|| num2);

END;

END;

/

1. The following program assigns values from the table to PL/SQL variables using the SELECT INTO clause of SQL:

DECLARE

c\_id customers.id%type :=1;

c\_name customers.name%type;

c\_addr customers.address%type;

c\_sal customers.salary%type;

BEGIN

SELECT name, address, salary INTO c\_name, c\_addr, c\_sal

FROM customers

WHERE id = c\_id;

dbms\_output.put\_line

('Customer '||c\_name ||' from '|| c\_addr ||' earns '|| c\_sal);

END;

/

1. PL/SQL Literals

PL/SQL supports the following kinds of literals:

* Numeric Literals
* Character Literals
* String Literals
* BOOLEAN Literals
* Date and Time Literals

The following table provides examples from all these categories of literal values.

|  |  |
| --- | --- |
| **Literal Type** | **Example:** |
| Numeric Literals | 050 78 -14 0 +32767 6.6667 0.0 -12.0 3.14159 +7800.00 6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3 |
| Character Literals | 'A' '%' '9' ' ' 'z' '(' |
| String Literals | 'Hello, world!' 'Tutorials Point' '19-NOV-12' |
| BOOLEAN Literals | TRUE, FALSE, and NULL. |
| Date and Time Literals | DATE '1978-12-25'; TIMESTAMP '2012-10-29 12:01:01'; |

PL/SQL language is rich in built-in operators and provides the following types of operators:

* Arithmetic operators
* Relational operators
* Comparison operators
* Logical operators
* String operators

## Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 5 then:

[Show Examples](http://www.tutorialspoint.com/plsql/plsql_arithmetic_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | A + B will give 15 |
| - | Subtracts second operand from the first | A - B will give 5 |
| \* | Multiplies both operands | A \* B will give 50 |
| / | Divides numerator by de-numerator | A / B will give 2 |
| \*\* | Exponentiation operator, raises one operand to the power of other | A \*\* B will give 100000 |

## Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 20, then:

[Show Examples](http://www.tutorialspoint.com/plsql/plsql_relational_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A = B) is not true. |
| != <> ~= | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

## Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either TRUE, FALSE OR NULL.

[Show Examples](http://www.tutorialspoint.com/plsql/plsql_comparison_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| LIKE | The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not. | If 'Zara Ali' like 'Z% A\_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A\_i' returns a Boolean false. |
| BETWEEN | The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b. | If x = 10 then, x between 5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false. |
| IN | The IN operator tests set membership. x IN (set) means that x is equal to any member of set. | If x = 'm' then, x in ('a', 'b', 'c') returns boolean false but x in ('m', 'n', 'o') returns Boolean true. |
| IS NULL | The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL. | If x = 'm', then 'x is null' returns Boolean false. |

## Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produces Boolean results. Assume variable A holds true and variable B holds false, then:

[Show Examples](http://www.tutorialspoint.com/plsql/plsql_logical_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and | Called logical AND operator. If both the operands are true then condition becomes true. | (A and B) is false. |
| or | Called logical OR Operator. If any of the two operands is true then condition becomes true. | (A or B) is true. |
| not | Called logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false. | not (A and B) is true. |

1. PL/SQL Operator precedence:

|  |  |
| --- | --- |
| **Operator** | **Operation** |
| \*\* | Exponentiation |
| +, - | identity, negation |
| \*, / | multiplication, division |
| +, -, || | addition, subtraction, concatenation |
| =, <, >, <=, >=, <>, !=, ~=, ^=,  IS NULL, LIKE, BETWEEN, IN | Comparison |
| NOT | logical negation |
| AND | Conjunction |
| OR | Inclusion |

1. Conditional statement in PL/SQL

1)

IF condition

THEN

 statement 1;

ELSE

 statement 2;

END IF;

2)

IF condition1

THEN

 statement 1;

 statement 2;

ELSIF condtion2 THEN

 statement 3;

ELSE

 statement 4;

END IF;

4)

IF condition1 THEN

ELSE

 IF condition2 THEN

 statement1;

 END IF;

ELSIF condition3 THEN

  statement2;

END IF;

1. Iterative statements in PL/SQL

There are three types of loops in pl/sql

• Simple Loop  
• While Loop  
• For Loop

|  |  |  |
| --- | --- | --- |
| **Simple Loop** | **While Loop** | **For Loop** |
| **LOOP**  **statements;**  **EXIT;**  **{or EXIT WHEN condition;}**  **END LOOP;** | **WHILE <condition>**  **LOOP statements;**  **END LOOP;** | **FOR counter IN val1..val2**  **LOOP statements;**  **END LOOP;** |
| 1) Initialise a variable before the loop body. 2) Increment the variable in the loop. 3) Use a EXIT WHEN statement to exit from the Loop. If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once. | 1) Initialise a variable before the loop body. 2) Increment the variable in the loop. 3) EXIT WHEN statement and EXIT statements can be used in while loops but it's not done often. | val1 - Start integer value.  val2 - End integer value.  1) The counter variable is implicitly declared in the declaration section, so it's not necessary to declare it explicitly. 2) The counter variable is incremented by 1 and does not need to be incremented explicitly. 3) EXIT WHEN statement and EXIT statements can be used in FOR loops but it's not done often. |

Example:

1.

DECLARE

-- variable declaration

message varchar2(30):= 'Printing from 1 to 10';

i number(10,0) := 1;

BEGIN

dbms\_output.put\_line(message);

**loop**

dbms\_output.put\_line(i);

i := i +1;

**exit when** i >10;

**end loop;**

END;

/

**Output:**

SQL> @example;

Printing from 1 to 10

1

2

3

4

5

6

7

8

9

10

PL/SQL procedure successfully completed.

2.

DECLARE

msg varchar(30) := 'Printing using while loop';

dis varchar(4) := '\*';

i number(10,0) := 1;

BEGIN

DECLARE

j number(10,0) := 1;

BEGIN

dbms\_output.put\_line(MSG);

WHILE i <= 10 LOOP

dbms\_output.put\_line(dis || i );

i := i + 1;

END LOOP;

END;

END;

/

**Output:**

SQL> @example;

Printing using while loop

\*1

\*2

\*3

\*4

\*5

\*6

\*7

\*8

\*9

\*10

PL/SQL procedure successfully completed.

3.

DECLARE

BEGIN

FOR counter IN 1 .. 10

LOOP

DBMS\_OUTPUT.PUT\_LINE(COUNTER);

END LOOP;

END;

/

**OUTPUT:**

SQL> @example;

1

2

3

4

5

6

7

8

9

10

PL/SQL procedure successfully completed.

**4. Real time example of FOR loop use.**

begin

for rec in (select \* from USER\_TAB\_COLUMNS where COLUMN\_NAME='LAST\_UPDATED\_DATE') loop

execute immediate 'update ' || rec.TABLE\_NAME || ' set LAST\_UPDATED\_DATE=CREATED\_DATE where LAST\_UPDATED\_DATE is null';

end loop;

end;

/

**5. Reverse For loop**

BEGIN

FORnoIN REVERSE 5..1LOOP

DBMS\_OUTPUT.PUT\_LINE('Iteration : '||no);

ENDLOOP;

END;

/

1. **PL/SQL – Strings**

PL/SQL offers three kinds of strings:

* **Fixed-length strings**: In such strings, programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.
* **Variable-length strings**: In such strings, a maximum length up to 32,767, for the string is specified and no padding takes place.
* **Character large objects (CLOBs)**: These are variable-length strings that can be up to 128 terabytes.
* Oracle database provides numerous string datatypes , like, CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The datatypes prefixed with an 'N' are 'national character set' datatypes, that store Unicode character data.
* If you need to declare a variable-length string, you must provide the maximum length of that string. For example, the VARCHAR2 data type.
* To declare a fixed-length string, use the CHAR datatype. Here you do not have to specify a maximum length for a fixed-length variable. If you leave off the length constraint, Oracle Database automatically uses a maximum length required. So following two declarations below are identical:

red\_flag CHAR(1) := 'Y';

red\_flag CHAR := 'Y';

## String Functions and Operators

PL/SQL offers the concatenation operator (||) for joining two strings. The following table provides the string functions provided by PL/SQL:

|  |  |
| --- | --- |
| **S.N.** | **Function & Purpose** |
| 1 | **ASCII(x);** Returns the ASCII value of the character x. |
| 2 | **CHR(x);** Returns the character with the ASCII value of x. |
| 3 | **CONCAT(x, y);** Concatenates the strings x and y and return the appended string. |
| 4 | **INITCAP(x);** Converts the initial letter of each word in x to uppercase and returns that string. |
| 5 | **INSTR(x, find\_string [, start] [, occurrence]);** Searches for find\_string in x and returns the position at which it occurs. |
| 6 | **INSTRB(x);**  Returns the location of a string within another string, but returns the value in bytes. |
| 7 | **LENGTH(x);**  Returns the number of characters in x. |
| 8 | **LENGTHB(x);**  Returns the length of a character string in bytes for single byte character set. |
| 9 | **LOWER(x);**  Converts the letters in x to lowercase and returns that string. |
| 10 | **LPAD(x, width [, pad\_string]) ;**  Pads x with spaces to left, to bring the total length of the string up to width characters. |
| 11 | **LTRIM(x [, trim\_string]);**  Trims characters from the left of x. |
| 12 | **NANVL(x, value);**  Returns value if x matches the NaN special value (not a number), otherwise x is returned. |
| 13 | **NLS\_INITCAP(x);**  Same as the INITCAP function except that it can use a different sort method as specified by NLSSORT. |
| 14 | **NLS\_LOWER(x) ;**  Same as the LOWER function except that it can use a different sort method as specified by NLSSORT. |
| 15 | **NLS\_UPPER(x);**  Same as the UPPER function except that it can use a different sort method as specified by NLSSORT. |
| 16 | **NLSSORT(x);**  Changes the method of sorting the characters. Must be specified before any NLS function; otherwise, the default sort will be used. |
| 17 | **NVL(x, value);**  Returns value if x is null; otherwise, x is returned. |
| 18 | **NVL2(x, value1, value2);**  Returns value1 if x is not null; if x is null, value2 is returned. |
| 19 | **REPLACE(x, search\_string, replace\_string);**  Searches x for search\_string and replaces it with replace\_string. |
| 20 | **RPAD(x, width [, pad\_string]);**  Pads x to the right. |
| 21 | **RTRIM(x [, trim\_string]);**  Trims x from the right. |
| 22 | **SOUNDEX(x) ;**  Returns a string containing the phonetic representation of x. |
| 23 | **SUBSTR(x, start [, length]);**  Returns a substring of x that begins at the position specified by start. An optional length for the substring may be supplied. |
| 24 | **SUBSTRB(x);**  Same as SUBSTR except the parameters are expressed in bytes instead of characters for the single-byte character systems. |
| 25 | **TRIM([trim\_char FROM) x);**  Trims characters from the left and right of x. |
| 26 | **UPPER(x);**  Converts the letters in x to uppercase and returns that string. |

## Example 1

DECLARE

greetings varchar2(11):='hello world';

BEGIN

dbms\_output.put\_line(UPPER(greetings));

dbms\_output.put\_line(LOWER(greetings));

dbms\_output.put\_line(INITCAP(greetings));

/\* retrieve the first character in the string \*/

dbms\_output.put\_line ( SUBSTR (greetings,1,1));

/\* retrieve the last character in the string \*/

dbms\_output.put\_line ( SUBSTR (greetings,-1,1));

/\* retrieve five characters,

starting from the seventh position. \*/

dbms\_output.put\_line ( SUBSTR (greetings,7,5));

/\* retrieve the remainder of the string,

starting from the second position. \*/

dbms\_output.put\_line ( SUBSTR (greetings,2));

/\* find the location of the first "e" \*/

dbms\_output.put\_line ( INSTR (greetings,'e'));

END;

/

When the above code is executed at SQL prompt, it produces the following result:

HELLO WORLD

hello world

Hello World

h

d

World

ello World

2

PL/SQL procedure successfully completed.

## Example 2

DECLARE

greetings varchar2(30):='......Hello World.....';

BEGIN

dbms\_output.put\_line(RTRIM(greetings,'.'));

dbms\_output.put\_line(LTRIM(greetings,'.'));

dbms\_output.put\_line(TRIM('.'from greetings));

END;

/

When the above code is executed at SQL prompt, it produces the following result:

......Hello World

Hello World.....

Hello World

PL/SQL procedure successfully completed.

1. PL/SQL programming language provides a data structure called the VARRAY, which can store a fixed-size sequential collection of elements of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Each element in a varray has an index associated with it. It also has a maximum size that can be changed dynamically.

The basic syntax for creating a VRRAY type at the schema level is:

CREATEOR REPLACE TYPE varray\_type\_name IS VARRAY(n)of<element\_type>

Where,

* *varray\_type\_name* is a valid attribute name,
* *n* is the number of elements (maximum) in the varray,
* *element\_type* is the data type of the elements of the array.

Maximum size of a varray can be changed using the ALTER TYPE statement.

CREATE Or REPLACE TYPE namearray IS VARRAY(3) OF VARCHAR2(10);

/

Type created.

TYPE namearray IS VARRAY(5) OF VARCHAR2(10);

Type grades IS VARRAY(5) OF INTEGER;

DECLARE

type namesarray IS VARRAY(5) OF VARCHAR2(10);

type grades IS VARRAY(5) OF INTEGER;

names namesarray;

marks grades;

total integer;

BEGIN

names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');

marks:= grades(98, 97, 78, 87, 92);

total := names.count;

dbms\_output.put\_line('Total '|| total || ' Students');

FOR i in 1 .. total LOOP

dbms\_output.put\_line('Student: ' || names(i) || '

Marks: ' || marks(i));

END LOOP;

END;

/

When the above code is executed at SQL prompt, it produces the following result:

Student: Kavita Marks: 98

Student: Pritam Marks: 97

Student: Ayan Marks: 78

Student: Rishav Marks: 87

Student: Aziz Marks: 92

PL/SQL procedure successfully completed.

1. Cursor

Cursor is the work area which Oracle reserves for internal processing of SQL statements.

Cursors can be classified as:

1. Implicit Cursor or Internal Cursor : Manage for Oracle itself or internal process itself.
2. Explicit Cursor or User-defined Cursor : Manage for user/programmer or external processing.
3. Implicit Cursor

Oracle uses implicit cursors for its internal processing. Even if we execute a SELECT statement or DML statement Oracle reserves a private SQL area in memory called cursor.

Implicit cursor scope you can get information from cursor by using session attributes until another SELECT statement or DML statement execute.

Following are implicit cursor attributes,

|  |  |  |
| --- | --- | --- |
| **Cursor Attribute** | **Cursor Variable** | **Description** |
| %ISOPEN | SQL%ISOPEN | Oracle engine automatically open the cursor If cursor open **return TRUE** otherwise **return FALSE.** |
| %FOUND | SQL%FOUND | If SELECT statement return one or more rows or DML statement (INSERT, UPDATE, DELETE) affect one or more rows If affect **return TRUE** otherwise **return FALSE.** If not execute SELECT or DML statement **return NULL.** |
| %NOTFOUND | SQL%NOTFOUND | If SELECT INTO statement return no rows and fire no\_data\_found PL/SQL exception before you can check SQL%NOTFOUND. If not affect the row **return TRUE** otherwise **return FALSE.** |
| %ROWCOUNT | SQL%ROWCOUNT | Return the number of rows affected by a SELECT statement or DML statement (insert, update, delete). If not execute SELECT or DML statement **return NULL.** |

Example:

SQL> create table emp(

2 emp\_name varchar2(20),

3 emp\_dept varchar2(30)

4 );

Table created.

BEGIN

FOR no in 1 .. 5 LOOP

insert into emp values('empname'||no ,'empdept'||no );

END LOOP;

UPDATE emp SET emp\_dept='Web Developer'

WHERE emp\_name='empname1';

IF SQL%FOUND THEN

dbms\_output.put\_line('Updated - If Found');

END IF;

IF SQL%NOTFOUND THEN

dbms\_output.put\_line('NOT Updated - If NOT Found');

END IF;

IF SQL%ROWCOUNT>0 THEN

dbms\_output.put\_line(SQL%ROWCOUNT||' Rows Updated');

ELSE

dbms\_output.put\_line('NO Rows Updated Found');

END IF;

END;

/

SQL> @example;

Updated - If Found

1 Rows Updated

PL/SQL procedure successfully completed.

SQL> select \* from emp;

EMP\_NAME EMP\_DEPT

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

empname1 Web Developer

empname2 empdept2

empname3 empdept3

empname4 empdept4

empname5 empdept5

SQL> drop table emp;

Table dropped.

1. **Explicit Cursor**

User itself to declare the cursor, open cursor to reserve the memory and populate data, fetch the records from the active data set one at a time, apply logic and last close the cursor.

You can not directly assign value to an explicit cursor variable you have to use expression or create subprogram for assign value to explicit cursor variable.

Step for Using Explicit Cursor :

1. Declare cursor
2. Open cursor
3. Loop
4. Fetch data from cursor
5. Exit loop
6. Close cursor

 Declare cursor

Declare explicit cursor has this syntax,

CURSOR cursor\_name [ parameter ] RETURN return\_type;

CURSOR cursor\_name [ parameter ] [ RETURN return\_type ]

IS SELECT STATEMENT;

Declaring explicit cursor example,

CURSOR c RETURN EMP\_DEPT%ROWTYPE; -- Declare c

CURSOR c IS -- Define c,

SELECT \* FROM emp\_information; -- all row return type

CURSOR c RETURN EMP\_DEPT%ROWTYPE IS -- Define c,

SELECT \* FROM emp\_information; -- repeating return type

 Opening Explicit Cursor

DECLARE block you are already declare CURSOR now you can OPEN CURSOR by using following way, and allocate some reserve area for process database query.

OPEN cursor\_name [( cursor\_parameter )];

 Loop

Loop iterate until ROW not found. Once found loop exit control goes next statement (outside loop).

 Fetching data from cursor

Using FETCH statement you can fetch CURSOR data into explicit variable.

FETCH cursor\_name INTO variable;

 Exit loop

 Closing Explicit Cursor

This way you can close opened CURSOR.

CLOSE cursor\_name [( cursor\_parameter )];

Input:

ORDER\_MODE ORDER\_STATUS ORDER\_TOTAL

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

direct 0 163131.3

direct 1 227569.5

direct 2 166169.5

direct 3 206659.4

direct 4 56352.5

direct 5 172586.2

direct 6 115968

direct 7 33617.1

direct 8 545300.5

direct 9 205674.2

direct 10 10601

online 0 25976.7

online 2 103834.4

online 3 56381.7

online 4 700068.1

online 5 183261.2

online 6 90411.8

online 8 322192.5

online 9 57062.4

online 10 225236.7

Output:

ORDER\_MODE STAT\_0 STAT\_1 STAT\_2 STAT\_3 STAT\_4 STAT\_5 STAT\_6 STAT\_7 STAT\_8 STAT\_9 STAT\_10

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

DIRECT 163131.3 227569.5 166169.5 206659.4 56352.5 172586.2 115968 33617.1 545300.5 205674.2 10601

ONLINE 25976.7 103834.4 56381.7 700068.1 183261.2 90411.8 322192.5 57062.4 225236.7

Query:

QL> SELECT UPPER(o.order\_mode) order\_mode

2 ,SUM(DECODE(o.order\_status, 0, o.order\_total)) stat\_0

3 ,SUM(DECODE(o.order\_status, 1, o.order\_total)) stat\_1

4 ,SUM(DECODE(o.order\_status, 2, o.order\_total)) stat\_2

5 ,SUM(DECODE(o.order\_status, 3, o.order\_total)) stat\_3

6 ,SUM(DECODE(o.order\_status, 4, o.order\_total)) stat\_4

7 ,SUM(DECODE(o.order\_status, 5, o.order\_total)) stat\_5

8 ,SUM(DECODE(o.order\_status, 6, o.order\_total)) stat\_6

9 ,SUM(DECODE(o.order\_status, 7, o.order\_total)) stat\_7

10 ,SUM(DECODE(o.order\_status, 8, o.order\_total)) stat\_8

11 ,SUM(DECODE(o.order\_status, 9, o.order\_total)) stat\_9

12 ,SUM(DECODE(o.order\_status, 10, o.order\_total)) stat\_10

13 FROM orders o

14 GROUP BY o.order\_mode

15 ORDER BY 1

ORDER\_MODE STAT\_0 STAT\_1 STAT\_2 STAT\_3 STAT\_4 STAT\_5 STAT\_6 STAT\_7 STAT\_8 STAT\_9 STAT\_10

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

DIRECT 163131.3 227569.5 166169.5 206659.4 56352.5 172586.2 115968 33617.1 545300.5 205674.2 10601

ONLINE 25976.7 103834.4 56381.7 700068.1 183261.2 90411.8 322192.5 57062.4 225236.7

<http://oraclecoder.com/tutorials/three-ways-to-transpose-rows-into-columns-in-oracle-sql--160>