**Oracle database**



**-**A **DATABASE** is a place where you store data.

#Save data for future use

* A data base work on special data structure which has rows and columns called table.

|  |  |  |  |
| --- | --- | --- | --- |
| * Col(Name)/text | * col | Col | * col |
| * rows |  |  |  |
| * Rows |  |  |  |

* Column =attribute ,information /properties about something =particular data=it has a datatype ,every column has a particular datatype that can go in that column= vertical
* Rows =records =horizontally=represent a set of related data ,rows are data with respective attribute
* database = collection of tables(company database){ collection of objects called table, where all the data is stored}



* Tables = has column and rows
* Table has a purpose( employee, departments etc)
* SQL =language used to interact with a database
* SQL is a set of commands that can be sent to database ,sent instructions to the database and ask to perform task (the database interpret the commands and execute them) ( create , delete, modify table, query table)
* Queries : when we write the SQL statements in a way that we are asking the database to give us some data or information back.
* SQL is database independent language.( SQL Standard = ANSI in oracle databases or any other database.
* The shape of database represents the hard disk which are going to connect to the server or our computer
* Oracle is a software installed on any server or machine and that uses the hardware of that machine. Oracle database system is combination of hardware or software.
* Oracle also provide cloud service (Apex)
* Oracle on the cloud means that oracle installed on some remote servers.
* We are going to use the server close to us.
* Apex SQL Command editor
* SQL is not case sensitive
* One query executed at a time

**RETRIEVING DATA USING THE SELECT CLAUSE**

* TO view data in the database
* **-- SELECT \* FROM emp;**
* **-- SELECT \* FROM dept;**
* **SELECT JOB,ename FROM EMP;**
* **SQL statement = select ,** made of keywords{select ,from}
* **SELECT <COLUMN(s)> FROM <TABLE NAME>**
* This is the query select statement makes up the query
* If we want to select all the columns = \* (wild card)
* **DISTINCT KEYWORD** gives unique values
* **E.g.: select DISTINCT job from emp;**

**USING THE WHERE CLAUSE IN A SQL QUERY**

* Filtering the result of the select clause
* **select \* from emp where job ='MANAGER';**
* written in quotes
* where applies a filter condition
* but filter condition needs to be case sensitive.
* AND KEYWORD = CAN be used to add further condition in where clause
* If wrong condition then nothing will get return (data does not exist)
* SELECT FROM WHERE/AND
* Need to use this only other and also use correct column name wise syntax error

**Working – the data from apex is sent to oracle database(server),it checks for the syntax and return possible values on clicking run button( first look at from clause(whether the data is available) ,then goes to where clause(interprets what to filter) and then select (columns) the giving result accordingly).**

**USING OPERATOR IN WHERE CLAUSE**

* Operators are used to compare one column with another or some data attribute
* **= ,equal operator returns equal value**
* SELECT \* FROM EMP WHERE JOB ='SALESMAN';
* **!not operator return value not equal to**
* SELECT \* FROM EMP WHERE JOB ! ='SALESMAN';
* We cannot use same equal to and not equal with AND these will return no data found
* These conditions are applied at record level, we checking every row for the desired result or condition
* **< , less than**
* SELECT \* FROM EMP WHERE JOB!='SALESMAN' AND SAL < 2500;
* **>, larger than**
* SELECT \* FROM EMP WHERE JOB!='SALESMAN' AND SAL > 2500;
* Strings are enclosed in quotes number not.
* **<= ,less than or equal to**
* **>=,greater than or equal to**
* **! =**
* SELECT \* FROM EMP
* WHERE JOB !='MANAGER'
* AND SAL > 2500
* AND DEPTNO = 20;

**Combing where , and & or with operators.**

* Condition applying at same field we cannot use AND (it is logically incorrect)
* **SELECT \* FROM EMP**
* **WHERE JOB = 'CLERK'**
* **AND JOB='SALESMAN';**
* Therefore ,we use OR (if either of the condition is true )
* **SELECT \* FROM EMP**
* **WHERE JOB = 'CLERK' OR**
* **JOB='SALESMAN';**
* With not equal AND is used
* **SELECT \* FROM EMP**
* **WHERE JOB != 'MANAGERS' AND**
* **JOB ! ='SALESMAN'**
* **AND SAL >= 2000;**

**QUERY FILTER -BETWEEN ,IN,NULL**

* IN CLAUSE : reduce the need of using or again and again
* SELECT ename, hiredate FROM EMP
* where DEPTNO IN (20,30);
* the possible values written in bracket
* use single quotes for string
* NOT IN CLAUSE: not one of the values
* SELECT ename, hiredatE FROM EMP
* where ename not in ('FORD','SMITH','ALEN','WARD','MARTIN')
* BETWEEN clause : used to filter based on range of data
* Between val A and val B
* select \* from emp
* where hiredate between '05/01/1981' and '12/09/1982'
* values of A and B are included ,inclusive range
* select \* from emp
* where sal between 1000 and 2000;
* NOT BETWEEN clause: opposite of between
* EMPTY CELLS -NULL value , no data for particular cell, it is not zero
* IS NULL: to identify empty cell
* select \* from emp
* where comm is null;
* IS NOT NULL: opposite of null
* Not inclusive:
* select \* from emp
* where (comm is null
* and sal > 1100 and sal <5000
* and sal != 3000)
* or comm=0;
* <> !=
* If or is used before and then we will not get the desired result

**OPERATOR PRECEDENCE**

* **Therefore,** we need ()
* select \* from emp
* where (comm is null
* and sal > 1100 and sal <5000
* and sal != 1500)
* or comm=0;
* **not work**
* **select \* from emp**
* **where (comm is null or comm =0)**
* **and sal > 1100 and sal < 5000**
* **and sal <> 1500;**

**LIKE OPERATOR**

* to search for records that don’t exactly match but kind of match
* tries to match the pattern
* reduce the need of writing
* **select \* from emp**
* **where job like 'S%';**
* uses a wildcard character -% (to match any number of characters)
* **select \* from emp**
* **where job like '%GER';**

**ALIASING**

* alias is another nickname for particular attribute
* **select ENAME EMPLOYEE , SAL SALARY , COMM COMMISION**
* **FROM emp;**
* If I want to write employee name
* **select ENAME EMPLOYEE\_NAME , SAL SALARY , COMM COMMISION**
* **FROM emp;**
* Or I want it without underscore
* **select ENAME "EMPLOYEE NAME" , SAL SALARY , COMM COMMISION**
* **FROM emp;**
* For alias we use double quote
* Using **as**  keyword is optional but works in the same manner.
* Alias cannot be too long
* **select ENAME as "EMPLOYEE NAME" , SAL AS SALARY , COMM COMMISION**
* **FROM emp;**

**CONCATENATING**

* Combining two attributes together into a sentence
* || pipe operator is used
* Use single quote
* **select 'hello my name is ' || ename as "concatenated value"**
* **from emp**
* **where job ='MANAGER';**
* more than one column
* **select ename ||' makes ' || sal || ' per month '**
* **from emp;**

**ORDERING**

* **ORDER BY :** SORT THE DATA based on particular column
* Order by is ascending by default (smallest to largest) asc keyword
* **select ename, sal from emp**
* **order by ename;**
* to sort in descending order, we can use =DESC keyword
* **select ename, sal from emp**
* **order by sal desc;**
* multiple columns
* **select deptno,ename, sal from emp**
* **order by deptno,sal;**

**SINGLE ROW FUNCTIONS(srf’s)**

* function is predefined program that perform some task
* function have attributes or inputs
* e.g.: sumthese(1,2,3) will return 1+2+3= 6
* single row functions- act on one row at a time
* it will return single value every time
* e.g. we have used || for concatenation , there is also a function that exactly does the same thing **CONCAT(‘HELLO’,’THERE’)>> HELLO THERE**
* **SELECT CONCAT('MY name is ',ename)**
* **from emp;**
* change column name
* **SELECT CONCAT('MY name is ',ename) as sentence**
* **from emp;**
* FUNCTION **UPPER(‘hello’) return HELLO**
* **select upper('vanshika')**
* **from emp;** // all rows of table ,repeating number of time
* **select ‘teesha’ from emp** //will also work in same manner ,print multiple times

**THE DUAL TABLE**

* prevent multiple entries
* single record
* used for testing
* **select lower('ERTeesha')**
* **from dual;**
* **select \* from dual; //** it has single column called dummy
* **select 'pizza' as FOOD ,'fanta' as drink from dual;**
* ****
* Calling function within other function
* **select concat(lower(ename),' is the name') from emp;**

**USING FUNCTION IN WHERE**

* single row functions can also be used to filter condition in where clause
* **select \***
* **from emp**
* **where job=upper('manager');**
* function can not only be used with the select statement but also with the where clause to filter out the result.

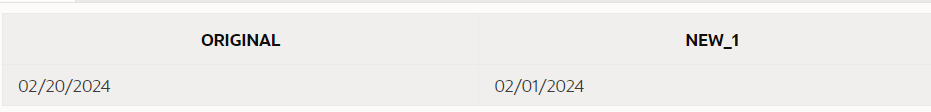
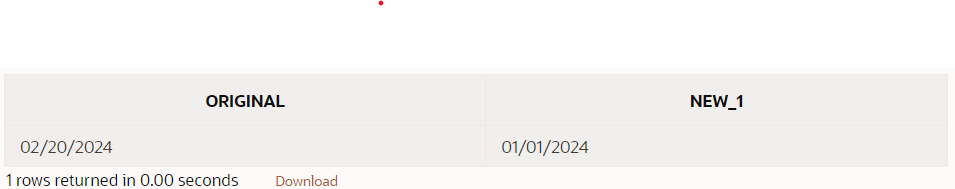
**CHARACTER BASED SRF’S**

* **upper(), lower(),concat()**
* work on character
* **initcap(‘hello there’) returns ‘Hello There’**
* it capitalize the first character of every word in the sentence
* **select initcap('hello yash how are you?') as example**
* **from dual;**
* **length(‘hello’) returns 5**
* return number of character in the sequence
* **select ename, length(ename) as length**
* **from emp**
* **where length(ename)=6;**
* **substr(‘hello’,2,3) returns ell**
* the function is used to extract parts of the string
* the first argument can be a character string, column, word, sentence
* the second argument is the starting position, indicated by the number
* the third argument is the number of character we want to extract
* **select 'hello' ,substr('hello',2,2) from dual; >>el**
* **select 'hello' ,substr('hello',2) from dual;>>ello**
* **lpad(‘day’,6,’$’) return $$$day**
* the function pad the extra character to left of any word or character sequence
* first argument is the word in which we want padding
* second argument is the total width of the string
* third argument is the character we need to add
* **select lpad('hello',10,'\*') from dual; >>** \*\*\*\*\*hello
* **select lpad('hello',10) from dual; >>** hello
* **similarly we have rpad()**
* **ltrim() and rtrim():**
* have two arguments ,the first argument defines the data that need to be trimmed , the second argument contains the character which need to be trimmed
* **select rtrim('helllllllllllllll','l') from dual >>**he
* **select ltrim(' helllllllllllllll ') from dual>>** helllllllllllllll

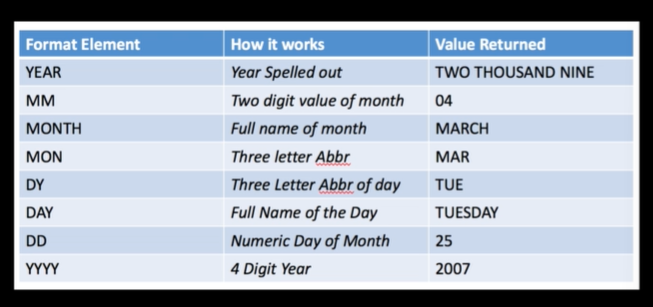
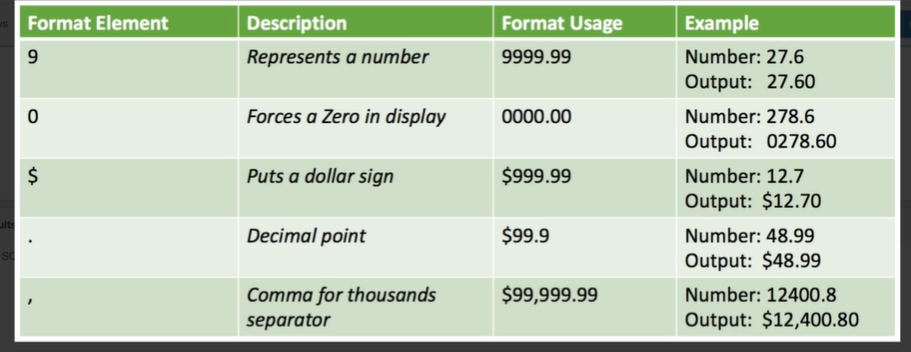
**NUMERIC function**

* number as argument
* ROUND(100.346,2)>>100.35 ( FIRST argument is the number we want to round off , second argument is how many decimal places we want to round about/precision points after the decimal)
* Without giving second argument= we will get whole number
* TRUNC(100.887,2) >>100.88( it truncates the first argument to the number of places defined by the second argument)
* Without giving second argument= we will get whole number

**DATE time function**

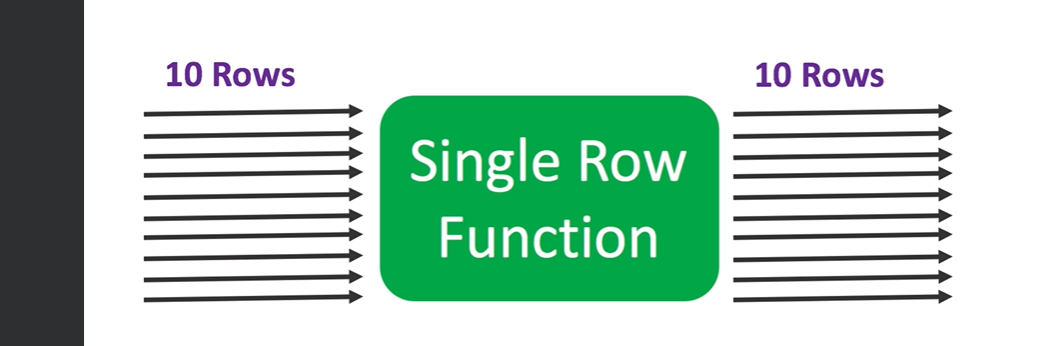
* Sysdate : gives the date where the database is stored
* Systimestamp: time(hours,minutes,seconds,fraction seconds ,am or pm as well as timezone) and date
* Add\_months(‘7/13/2014’,8) >>the first argument is some date and the second argument is a number representing the month to be added the given date
* **select add\_months('11/17/2012',4) from dual;>>** 03/17/2013
* if -4 >> 07/17/2012
* months\_between(dateA,dateB): return months between two dates
* **select months\_between('12/15/2012','12/4/2013') from dual;>>** **-**11.645161290322580645161290322580645161
* smaller date first therefore negative
* **select months\_between('12/15/2013','12/4/2013') from dual; >>** .35483870967741935483870967741935483871
* **trunc(date,’MONTH) >>TRUNCATE unnecessary information**
* **select trunc(systimestamp) from dual;>>** 02/20/2024
* if we use trunc with month or year it would give first of the month or year
* **select trunc(systimestamp) as original ,trunc(systimestamp,'MONTH') as new\_1 from dual;**
* **>>** ****
* **select trunc(systimestamp) as original ,trunc(systimestamp,'YEAR') as new\_1 from dual;**
* ****
* **Single row function can be called in where clause**

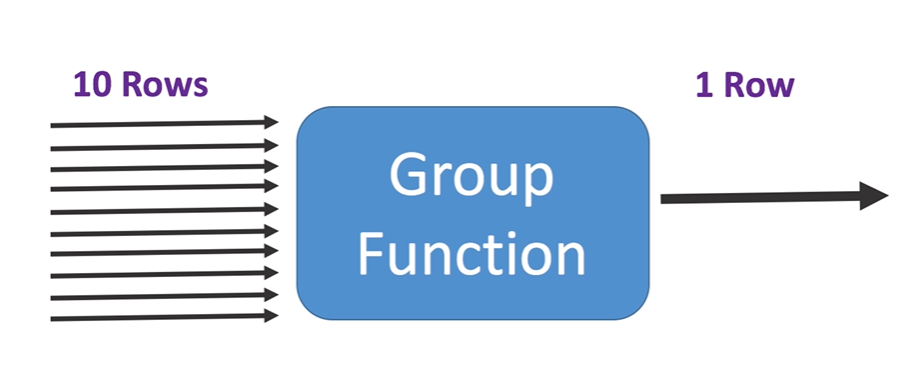
**CONVERSION SRF’S AND DATE FORMATTING**

* **TO\_CHAR(sysdate,’MONTH DD,YYYY’) >>MAY 31,2016**
* **TO\_CHAR(123,’$999.99’)>>$123.00**
* Convert date or number to a string
* Can change the format or the way we want to print
* **select sysdate from dual;//02/21/2024**
* **select to\_char(sysdate,'mm-dd-yyyy') from dual; //02-21-2024**
* **select to\_char(sysdate,'mm/yyyy/dd') from dual; //02/2024/21**
* **select to\_char(sysdate,'month-day-year') from dual; //february -wednesday-twenty twenty-four**
* **select to\_char(sysdate,'day') from dual; //Wednesday**
* **select to\_char(sysdate,'ddth "of" month ,yyyy') from dual; //21st of february ,2024**
* non default elements are wrapped in the double quotes
* ##format numbers
* **select ename as "employee name",to\_char( sal ,'$99,999.99') as "employee salary" from emp;**
* **TO\_date(‘str’,’fmt’)**  it convert string to a date value
* **select to\_date('2012-08-27','yyyy-mm-dd') from dual; // 08/27/2012**
* **select add\_months(to\_date('2012-08-27','yyyy-mm-dd'),2) from dual; //**10/27/2012
* **select to\_date(' 3 of June ,2012','dd "of" Month,YYYY') from dual;//06/03/2012**
* **LAST\_DAY(*d):*** *last\_day is a date function that requires a date as an argument. It returns the last day of the month in which the given date falls. The argument is required for this function to work properly.*
* select last\_day('02-21-2024') from dual; // 02-29-2024
* **NEXT\_DAY(d, c) :**The first argument is the date and the second argument is a text reference to a day of the week. Both arguments are required for this function to work properly. This function returns a valid date representing the first occurrence of the c day following the date represented in d.
* select next\_day('02-21-2024','monday') from dual; //02-26-2024
* ****
* ****

**CONCLUDING SRF’S & NULL /NULL IF**

* NVL is a substitution function in SQL that replaces null values with a string in the results of a query. NVL takes two arguments:
* expr1: The name of the expression to be evaluated
* expr2: The value that the function returns when the first argument evaluates to NULL
* **select ename , job ,sal , NVL(comm,0)**
* **from emp**
* **where empno in(7839,7698,7566,7654);**
* ****
* **select ename , job ,sal , NVL(to\_char(comm),'No data found')**
* **from emp**
* **where empno in(7839,7698,7566,7654);**
* Nullif is a function that compares two expressions and returns NULL if they are equal. If the expressions are not equal, NULLif returns the first expression
* NULLIF(ARG1,ARG2)
* select ename, length(ename) , nullif(length(ename),5) from emp;
* e.g : select ename, length(ename) , nvl(nullif(to\_char(length(ename)),to\_char(5)),'length is 5') from emp;

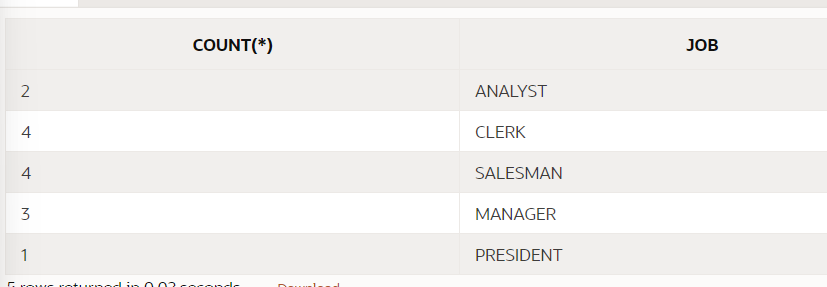
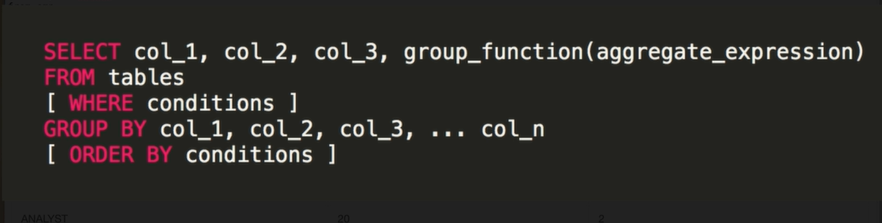




**Grouping function**

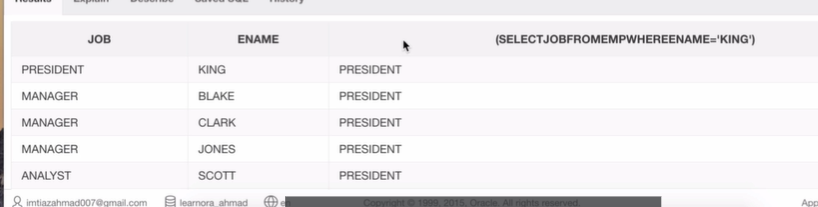
* take output many rows but return a single value as output
* output is single scaler value
* max() – return maximum value out of all >> select max(sal) from emp;
* min() – return min value >> select min(sal) from emp;
* sum() – return aggregate value>> select sum(sal) from emp;
* avg()-return average value >> select avg(sal) from emp;
* count() -return number of records>> select count(sal) from emp;
* count() does not count null values
* **select count(\*) from emp; //see number of records**

**GROUP BY AND HAVING**

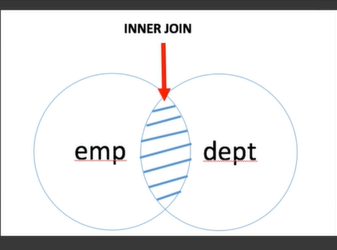
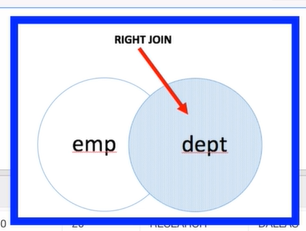
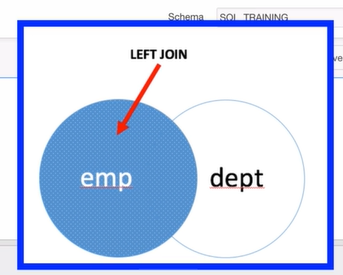
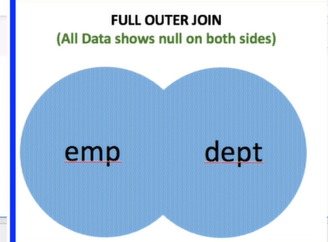
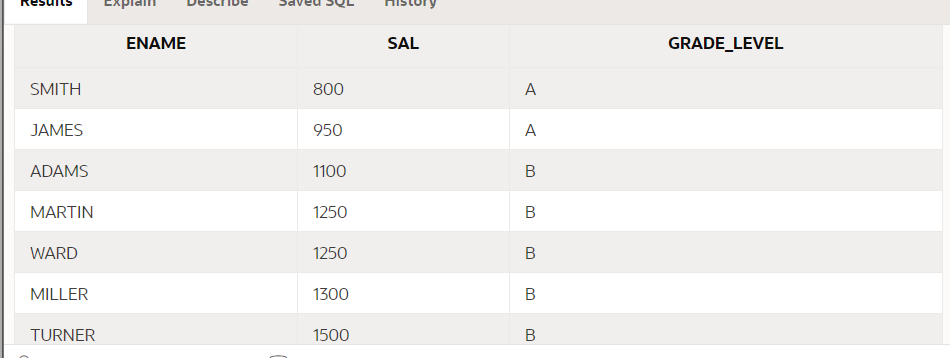
* what if we want to calculate average salary of per job – on way is to use where clause , but then we have to write multiple queries for each job title
* e.g. select avg(sal) from emp where job=’manager’
* it would not be feasible if we have 100 job titles
* other way :is to use GROUP BY clause
* **select avg(sal),job**
* **from emp**
* **group by job;**
* for each job it will give average
* if we remove avg , then it will give us all the individual job
* **select count(\*),job**
* **from emp**
* **group by job;**
* ****
* How to filter in group by -Having
* **select job**
* **from emp**
* **group by job**
* **having count(\*)=2;**
* No grouping function in where clause
* Only single row function allowed in where clause
* **Select from where group by having order by**
* Group by multiple columns
* **select deptno,job, count(\*)**
* **from emp**
* **group by job,deptno**
* **order by deptno;**
* So when there's a grouping function involved in the select statement and there are other expressions as well, other columns in the Select Statement as well that are not grouping functions, those columns better also be in the group by clause.
* And another important thing is you can have multiple columns in the group by clause, right?
* But those don't necessarily have to be in the select portion.
* ****
* till now we have done single table queries

**MULTI TABLE QUERIES**

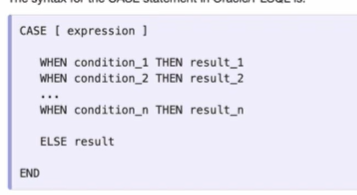
**Select within select**

* nested queries : select statement within the select statement
* the result of subquery is placed in main query to act upon
* the nested query is calculated first
* nested query can be in different part of select statement
* **select \* from dept**
* **where deptno<(select deptno from dept where deptno=30)**
* **AND dname='ACCOUNTING';**
* Can relate multiple tables
* **select \* from emp**
* **where deptno =(**
* **select deptno from dept where loc='CHICAGO'**
* **)**
* nested query does not work with equal sign when subquery return multiple values
* therefore we use in
* **select \* from emp where deptno in(select deptno from dept where deptno in (10,20))**
* also in subquery we will have only one column
* **select \* from emp where deptno in (select deptno, ,loc, dname from dept where deptno in (10,20))**
* // this will not work since in clause work for list of values
* select \* from (select \* from emp); // in from clause interpreted as table
* Select statement in select clause:
* **select job ,ename ,(select job from emp) from emp** , it returns error
* because the nested query contains multi row subquery ,only single row subquery is allowed
* **select job ,ename ,(select job from emp where ename='KING') from emp**
* where clause nested statement should have single scaler value
* ****

**JOINS**

* Relate two tables together
* **select \* from emp , dept**
* **where emp.deptno=dept.deptno;**
* **//** it only contain that information which are present in both table , for eg boston is not present in this new table but present in dept table because it is not present in emp table , also it contain two column representing deptno which is the common column
* **select ENAME,JOB,SAL from emp , dept**
* **where emp.deptno=dept.deptno // need to be explicitly specified**
* **and dept.loc='DALLAS'; // to avoid conflicting we can specify the table**
* alias to table
* **select e.ENAME,e.JOB,e.SAL from emp e, dept d**
* **where e.deptno=d.deptno**
* **and d.loc='DALLAS'; // now need to give alias table name only**
* can use nested subquery also -same result , as result of from clause is treated as table
* **select e.ENAME,e.JOB,e.SAL from (select \* from emp) e, dept d**
* **where e.deptno=d.deptno**
* **and d.loc='DALLAS';**
* This increases the flexibility
* **select e.ENAME,e.JOB,e.SAL from (select \* from emp where job in ('MANAGER','CLERK') ) e, (select \* from dept WHERE LOC='DALLAS') d where e.deptno=d.deptno;**
* standard – capitalize all keywords ,table name ,column name in lower case
* ** select \* from emp , dept where emp.deptno= dept.deptno;**
* Inner join return matched records from both the table/ join on common values
* Besides using a equal sign inner join could be performed using keyword INNER JOIN
* E.g.
* **SELECT \***
* **FROM emp e INNER JOIN dept d**
* **ON e.deptno = d.deptno**
* ****
* **RIGHT JOIN KEYWORD IS USED**
* It gives all the data from the right table + the matched column
* **SELECT \***
* **FROM emp e RIGHT JOIN dept d**
* **ON e.deptno = d.deptno**
* Old syntax : SELECT \*
* FROM emp, dept
* WHERE emp.deptno(+)= dept.deptno
* **LEFT JOIN KEYWORD**
* All the records from left table +matched column
* **SELECT \***
* **FROM emp e LEFT JOIN dept d**
* **ON e.deptno = d.deptno**
* Old syntax:
* SELECT \*
* FROM emp, dept
* WHERE emp.deptno = dept.deptno(+)
* ****
* **FULL OUTER JOIN**
* Join all the records from both the table
* **SELECT \***
* **FROM emp e FULL OUTER JOIN dept d**
* **ON e.deptno = d.deptno**
* Subquery slows the process but its flexible
* **SELECT e.\* , d.\*** // all columns
* **FROM (SELECT \* FROM dept) d LEFT OUTER JOIN (SELECT \* FROM emp WHERE job ='SALESMAN') e**
* **ON e.deptno=d.deptno**
* EXISTS CONDITION
* WHERE EXISTS( select \* from ...)
* if something is returned from the subquery the condition is met
* The EXISTS operator is used to test for the existence of any record in a subquery.
* The EXISTS operator returns TRUE if the subquery returns one or more records
* SELECT \*
* FROM emp
* WHERE EXISTS (select 'random' from dual)
* The 'NOT EXISTS' operator in SQL Server will check the Subquery for rows existence. If there are no rows then it will return TRUE, otherwise FALSE.
* SELECT \*
* FROM emp
* WHERE NOT EXISTS (select 'random' from dual) // no data found
* Although the EXISTS and NOT EXISTS are not very efficient , it is used in CORRELATED SUBQUERY.
* Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.
* A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a **SELECT**, **UPDATE**, or **DELETE** statement.
* SELECT d.\*
* FROM dept d
* WHERE EXISTs (SELECT \* FROM emp WHERE d.deptno=emp.deptno)
* SELECT d.\* // return boston
* FROM dept d
* WHERE NOT EXISTs (SELECT \* FROM emp WHERE d.deptno=emp.deptno)
* You can not refer to the variable inside nested query in the outer query
* SELF JOIN – join the table by itself
* **SELECT e1.ENAME as Employee\_name ,e2.ENAME as Manager\_name**
* **from emp e1**
* **inner join emp e2**
* **on e1.MGR=e2.empno**
* **SELECT e1.ENAME as Employee\_name ,e2.ENAME as Manager\_name**
* **from emp e1**
* **left join emp e2**
* **on e1.MGR=e2.empno**
* **CROSS JOIN/CARTESIAN JOIN/NO JOIN :** if you do not specify the join ,each row of the table will be joined other row of the table
* **select \* from emp,dept; 14\*4=56**
* cross join keyword could also be used : **select \* from emp cross join dept**;
* **Natural Join :** it is inner join without specifying on condition and duplicate column of match result , it knows about the matching column
* **select \* from emp**
* **natural join dept;**
* **USING clause : it** is used to specify the columns we want to join on
* **select \* from emp**
* **inner join dept**
* **using(deptno);**
* it also give one column
* if the column name are different in both tables , then we have to specify both the column names (col1,col2)
* **EQUI JOIN :** an inner join , with = sign /equality operator
* **select \* from emp , dept**
* **where emp.deptno=dept.deptno;**
* **NON EQUI JOIN :** other operators
* **select e.ename, e.sal ,j.grade\_level**
* **from emp e JOIN job\_grade j**
* **on e.sal between j.lowest\_sal and j.highest\_sal**
* ****

**CASE STATEMENT**

* Functionality of an IF-THEN-ELSE statement
* ****
* **SELECT ename, job , (CASE job**
* **WHEN 'PRESIDENT' THEN 'BIG SHOT'**
* **WHEN 'MANAGER' THEN 'DECIDES THE PAY'**
* **WHEN 'ANALYST' THEN 'GOOD AT MATH'**
* **WHEN 'CLERK' THEN 'HARD WORKING'**
* **ELSE 'NO COMMENT'**
* **END) AS "COMMENT"**
* **FROM emp;**
* Here it is like equality checking
* **SELECT ename, sal ,(CASE**
* **WHEN sal>5000 THEN 'RICH'**
* **WHEN 1000>=sal and sal<=5000 THEN 'MIDDLE class'**
* **ELSE 'GAREEB'**
* **END ) AS comment\_1**
* **FROM emp;**

**ANALYTIC FUNCTION USING OVER AND PARTION WITH ORDER BY**

**OVER CLAUSE WITH PARTITION BY :**

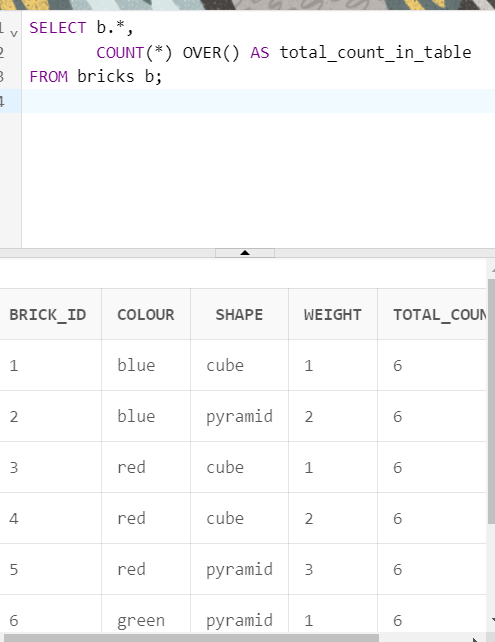
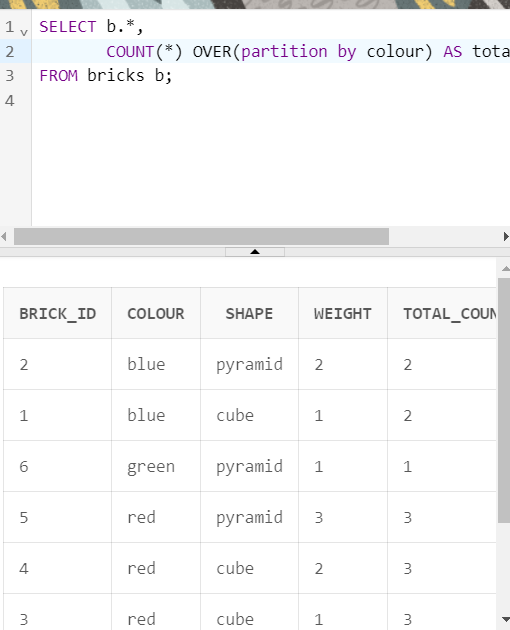
Aggregate and analytic functions both enable you to do a calculation over many rows. Aggregate functions squash the output to one row per group. For example the following counts the total rows in the table. It returns one row:

- select count(\*) from bricks;

Adding the over clause converts it to an analytic. This preserves the input rows. So you get all six, each with the value six:

- select count(\*) over () from bricks; In SQL, the OVER() clause is used in conjunction with window functions to perform calculations across a set of rows related to the current query row. When you use OVER(), you're essentially specifying the window of rows over which the window function will operate.

* Without any parameter over takes whole table
* We can use the **SQL PARTITION BY** clause with the **OVER** clause to specify the column on which we need to perform aggregation

select b.\*,

sum(weight) OVER (partition by shape) sum\_by\_shape,

sum(weight) OVER (partition by colour) sum\_by\_colour,

sum(weight) OVER (partition by shape) max\_weight\_by\_shape,

max(weight) OVER (partition by colour) max\_weight\_by\_colour

from bricks b;

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **BRICK\_ID** | **COLOUR** | **SHAPE** | **WEIGHT** | **SUM\_BY\_SHAPE** | **SUM\_BY\_COLOUR** | **MAX\_WEIGHT\_BY\_SHAPE** | **MAX\_WEIGHT\_BY\_COLOUR** |
| 4 | red | cube | 2 | 4 | 6 | 4 | 3 |
| 1 | blue | cube | 1 | 4 | 3 | 4 | 2 |
| 3 | red | cube | 1 | 4 | 6 | 4 | 3 |
| 5 | red | pyramid | 3 | 6 | 6 | 6 | 3 |
| 2 | blue | pyramid | 2 | 6 | 3 | 6 | 2 |
| 6 | green | pyramid | 1 | 6 | 1 | 6 | 1 |

* We can also have **order by** in over()
* The order by clause enables you to compute running totals. For example, the following sorts the rows by brick\_id. Then shows the total number of rows and sum of the weights for rows with a brick\_id less than or equal to that of the current row:
* select b.\*,
* sum ( weight ) over (
* order by brick\_id
* ) running\_weight
* from bricks b;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **BRICK\_ID** | **COLOUR** | **SHAPE** | **WEIGHT** | **RUNNING\_WEIGHT** |
| 1 | blue | cube | 1 | 1 |
| 2 | blue | pyramid | 2 | 3 |
| 3 | red | cube | 1 | 4 |
| 4 | red | cube | 2 | 6 |
| 5 | red | pyramid | 3 | 9 |
| 6 | green | pyramid | 1 | 10 |

select b.\*,

count(\*) over (

order by brick\_id

) running\_total\_count\_of\_records,

sum ( weight ) over (

order by brick\_id

) running\_weight

from bricks b;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **BRICK\_ID** | **COLOUR** | **SHAPE** | **WEIGHT** | **RUNNING\_TOTAL\_COUNT\_OF\_RECORDS** | **RUNNING\_WEIGHT** |
| 1 | blue | cube | 1 | 1 | 1 |
| 2 | blue | pyramid | 2 | 2 | 3 |
| 3 | red | cube | 1 | 3 | 4 |
| 4 | red | cube | 2 | 4 | 6 |
| 5 | red | pyramid | 3 | 5 | 9 |
| 6 | green | pyramid | 1 | 6 |  |
|  |  |  |  |  |  |

* we can also use **order by and partition together**
* select b.\*,
* sum(weight) over (
* partition by colour order by brick\_id
* ) running\_weight\_by\_colour,
* sum(weight) over (
* partition by colour
* ) weight\_by\_colour
* from bricks b;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **BRICK\_ID** | **COLOUR** | **SHAPE** | **WEIGHT** | **RUNNING\_WEIGHT\_BY\_COLOUR** | **WEIGHT\_BY\_COLOUR** |
| 1 | blue | cube | 1 | 1 | 3 |
| 2 | blue | pyramid | 2 | 3 | 3 |
| 6 | green | pyramid | 1 | 1 | 1 |
| 3 | red | cube | 1 | 1 | 6 |
| 4 | red | cube | 2 | 3 | 6 |
| 5 | red | pyramid | 3 | 6 | 6 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**select b.brick\_id, b.weight,**

**round ( avg ( weight ) over (**

**order by brick\_id**

**), 2 ) running\_average\_weight**

**from bricks b**

**order by brick\_id; // by default ascending**

**Windowing Clause**

When you use order by, the database adds a default windowing clause of:

range between unbounded preceding

and current row

This means: Include all the rows with a ***value*** less than or equal to that of the current row.This can lead to the function including values from rows after the current!

For example, there are several rows with the same weight. So when you sort by this, all rows with the same weight have the same running count and weight:

select b.\*,

count(\*) over (

order by weight

) running\_total,

sum ( weight ) over (

order by weight

) running\_weight

from bricks b

order by weight;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **BRICK\_ID** | **COLOUR** | **SHAPE** | **WEIGHT** | **RUNNING\_TOTAL** | **RUNNING\_WEIGHT** |
| 1 | blue | cube | 1 | 3 | 3 |
| 3 | red | cube | 1 | 3 | 3 |
| 6 | green | pyramid | 1 | 3 | 3 |
| 4 | red | cube | 2 | 5 | 7 |
| 2 | blue | pyramid | 2 | 5 | 7 |
| 5 | red | pyramid | 3 | 6 | 10 |

Usually this isn't what you want. Normally running totals should only include values from previous rows in the data set.To do this, you must specify a windowing clause of

rows between unbounded preceding

and current row

select b.\*,

count(\*) over (

order by weight

rows between unbounded preceding and current row

) running\_total,

sum ( weight ) over (

order by weight

rows between unbounded preceding and current row

) running\_weight

from bricks b

order by weight;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **BRICK\_ID** | **COLOUR** | **SHAPE** | **WEIGHT** | **RUNNING\_TOTAL** | **RUNNING\_WEIGHT** |
| 1 | blue | cube | 1 | 1 | 1 |
| 3 | red | cube | 1 | 2 | 2 |
| 6 | green | pyramid | 1 | 3 | 3 |
| 4 | red | cube | 2 | 4 | 5 |
| 2 | blue | pyramid | 2 | 5 | 7 |
| 5 | red | pyramid | 3 | 6 | 10 |

**SLIDING WINDOW = CURRENT ROW + PREVIOUS ROW(ANY NUMBER 1,2…..)**

* select b.\*,
* sum ( weight ) over (
* order by weight
* rows between 1 preceding and current row
* ) running\_weight
* from bricks b
* order by weight;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **BRICK\_ID** | **COLOUR** | **SHAPE** | **WEIGHT** | **RUNNING\_WEIGHT** |
| 1 | blue | cube | 1 | 1 |
| 3 | red | cube | 1 | 2 |
| 6 | green | pyramid | 1 | 2 |
| 4 | red | cube | 2 | 3 |
| 2 | blue | pyramid | 2 | 4 |
| 5 | red | pyramid | 3 | 5 |

**FILTERING ANALYTIC FUNCTION**

* traditional group by use having but over uses where condition and subquery for filtering
* **select \* from (**
* **select b.\*,**
* **count(\*) over ( partition by colour ) colour\_count**
* **from bricks b**
* **)**
* **where colour\_count >= 2;**

**MORE ANALYTIC FUNCTIONS**

The analytic functions rank, dense\_rank and row\_number all return an increasing counter, starting at one.

* Rank - Rows with the same value in the order by have the same rank. The next row after a tie has the value N, where N is its position in the data set.
* Dense\_rank - Rows with the same value in the order by have the same rank, but there are no gaps in the ranks
* Row\_number - each row has a new value
* **select brick\_id, weight,**
* **row\_number() over ( order by weight ) rn,**
* **rank() over ( order by weight ) rk,**
* **dense\_rank() over ( order by weight ) dr**
* **from bricks;**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **BRICK\_ID** | **WEIGHT** | **RN** | **RK** | **DR** |
| 1 | 1 | 1 | 1 | 1 |
| 3 | 1 | 2 | 1 | 1 |
| 6 | 1 | 3 | 1 | 1 |
| 4 | 2 | 4 | 4 | 2 |
| 2 | 2 | 5 | 4 | 2 |
| 5 | 3 | 6 | 6 | 3 |

* **Previous and Next Values**
* Lag and lead enable you to get values from rows backwards and forwards in your results.
* **select b.\*,**
* **lag ( shape ) over ( order by brick\_id ) prev\_shape,**
* **lead ( shape ) over ( order by brick\_id ) next\_shape**
* **from bricks b;**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **BRICK\_ID** | **COLOUR** | **SHAPE** | **WEIGHT** | **PREV\_SHAPE** | **NEXT\_SHAPE** |
| 1 | blue | cube | 1 | - | pyramid |
| 2 | blue | pyramid | 2 | cube | cube |
| 3 | red | cube | 1 | pyramid | cube |
| 4 | red | cube | 2 | cube | pyramid |
| 5 | red | pyramid | 3 | cube | pyramid |
| 6 | green | pyramid | 1 | pyramid | - |

* **First and Last Values**
* You can get the first or last value in an ordered set with first\_value and last\_value**:**
* **select b.\*,**
* **first\_value ( weight ) over (**
* **order by brick\_id**
* **) first\_weight\_by\_id,**
* **last\_value ( weight ) over (**
* **order by brick\_id**
* **) last\_weight\_by\_id**
* **from bricks b;**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **BRICK\_ID** | **COLOUR** | **SHAPE** | **WEIGHT** | **FIRST\_WEIGHT\_BY\_ID** | **LAST\_WEIGHT\_BY\_ID** |
| 1 | blue | cube | 1 | 1 | 1 |
| 2 | blue | pyramid | 2 | 1 | 2 |
| 3 | red | cube | 1 | 1 | 1 |
| 4 | red | cube | 2 | 1 | 2 |
| 5 | red | pyramid | 3 | 1 | 3 |
| 6 | green | pyramid | 1 | 1 | 1 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

* Note the result of first\_value stays the same. But for last\_value it changes for each row. This is because the default windowing clause stops at the current row. To find the value from the last row in the data set, you change the end of the window to "unbounded following". For example:
* **select b.\*,**
* **first\_value ( weight ) over (**
* **order by brick\_id**
* **) first\_weight\_by\_id,**
* **last\_value ( weight ) over (**
* **order by brick\_id**
* **range between current row and unbounded following**
* **) last\_weight\_by\_id**
* **from bricks b;**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **BRICK\_ID** | **COLOUR** | **SHAPE** | **WEIGHT** | **FIRST\_WEIGHT\_BY\_ID** | **LAST\_WEIGHT\_BY\_ID** |
| 1 | blue | cube | 1 | 1 | 1 |
| 2 | blue | pyramid | 2 | 1 | 1 |
| 3 | red | cube | 1 | 1 | 1 |
| 4 | red | cube | 2 | 1 | 1 |
| 5 | red | pyramid | 3 | 1 | 1 |
| 6 | green | pyramid | 1 | 1 |  |

**CREATING TABLE AND DESIGN CONSTRAINTS**

* **primary key** uniquely identify every single row , therefore can never contain null value, cannot have duplicate ,primary identifier ,can be one or combination of one or more columns
* good practice to have primary key but not compulsory
* CREATING TABLE
* **CREATE TABLE stores <table name>(**
* **store\_id number not null,**
* **city varchar(50)**
* **<column name> <data type> <constraint>**
* **);**
* **inserting data into the table**
* **INSERT INTO stores(store\_id,city) VALUES (1,'San Francisco');**
* **INSERT INTO stores(store\_id,city) VALUES (2,'New York City');**
* Automatically impied
* faster way?
* **INSERT ALL**
* **INTO stores (store\_id,city) VALUES (4,'Philadelphia')**
* **INTO stores (store\_id,city) VALUES (5,'Boston')**
* **INTO stores (store\_id,city) VALUES (6,'Seattle')**
* **SELECT \* FROM DUAL;**
* Oracle specific syntax
* We cannot insert null value due to -NOT NULL
* But we can again again insert same row , therefore we need primary key
* **CREATE TABLE products(**
* **product\_id number not null,**
* **name varchar2(50),**
* **product\_cost number(5,2), //**(length,no after decimal)
* **product\_retail number(5,2),**
* **product\_type varchar2(10),**
* **store\_id number not null,**
* **CONSTRAINT product\_pk PRIMARY KEY(product\_id)**
* **)**
* Constraint are also kind of object
* **DESCRIBE emp -- get description of any table**
* **INSERT ALL FOR INSERING DATA INTO MULTIPLE TABLES**
* **INSERT ALL**
* **INTO dest\_tbl\_1 (id, name, date\_of) values (EMPNO, ENAME, HIREDATE)**
* **INTO dest\_tbl\_2 (id, name, date\_of) values (EMPNO, ENAME, HIREDATE)**
* **INTO dest\_tbl\_3 (id, name, date\_of) values (EMPNO, ENAME, HIREDATE)**
* **SELECT empno, ename, hiredate**
* **FROM emp**
* Using condition
* **INSERT ALL**
* **WHEN sal <= 1500 THEN**
* **INTO dest\_tbl\_1 (id, name, date\_of) values (EMPNO, ENAME, HIREDATE)**
* **WHEN sal BETWEEN 1501 AND 2500 THEN**
* **INTO dest\_tbl\_2 (id, name, date\_of) values (EMPNO, ENAME, HIREDATE)**
* **WHEN sal > 2500 THEN**
* **INTO dest\_tbl\_3 (id, name, date\_of) values (EMPNO, ENAME, HIREDATE)**
* **SELECT empno, ename, hiredate, sal**
* **FROM emp**
* We can modify the structure of table using ALTER
* To modify the column – change column data type and adding constraint
* **ALTER TABLE products MODIFY name varchar2(50) not null**
* **ALTER TABLE products**
* **MODIFY (product\_cost number(5,2) not null,**
* **product\_retail number(5,2) not null );**
* Describe constraint **DESCRIBE product\_pk**
* Rename column
* **ALTER TABLE products**
* **RENAME COLUMN name to product\_name;**
* Creating a table like existing table
* **CREATE TABLE employees AS**
* **SELECT empno, ename, job, hiredate, sal, comm**
* **FROM emp;**
* ADD COLUMN in table
* **ALTER TABLE employees ADD store\_id number;**
* Cannot contain constraint not null , since when it is added it is blank
* UPDATe RECORDS
* UPDATE your\_table
* Set <column>=<value>
* Where <some criteria>
* **UPDATE employees**
* **SET store\_id = 3**
* **WHERE ename in ('KING', 'BLAKE', 'CLARK')**
* **BEGIN END** // if we want to add multiple insert statements
* MERGE STATEMENT : help us either to insert or update data in the table depending on whether it already exists
* he **MERGE** statement in SQL is a very popular clause that can handle inserts, updates, and deletes all in a single transaction without having to write separate logic for each of these.
* **MERGE INTO existing\_customers c**
* **USING new\_customers n**
* **ON (c.customer\_id = n.customer\_id)**
* **WHEN MATCHED THEN**
* **UPDATE SET**
* **c.first\_name = n.first\_name,**
* **c.last\_name = n.last\_name,**
* **c.address\_state = n.address\_state,**
* **c.email\_address = n.email\_address**
* **DELETE WHERE c.first\_name = 'John'**
* **WHEN NOT MATCHED THEN**
* **INSERT (c.customer\_id, c.first\_name, c.last\_name, c.address\_state, c.email\_address)**
* **VALUES (n.customer\_id, n.first\_name, n.last\_name, n.address\_state, n.email\_address)**
* COMPOSITE KEY
* create table T ( x int,
* y int,
* z int,
* constraint t\_pk primary key (x,y) );
* **modify** alter table t add constraint t\_pk primary key(x,y);
* **SEQUENCE STATEMENTS :** A sequence is exactly what it sounds like.
* It's basically a counter.It keeps track of a given numeric sequence.
* **CREATE SEQUENCE product\_seq**
* **MINVALUE 1**
* **MAXVALUE 99 // CAN BE DEFAULT**
* **START WITH 1**
* **INCREMENT BY 1**
* **CACHE 20; //20 VALUES IN MEMORY /NOCACHE**
* Any value to above
* **SELECT product\_seq.NEXTVAL FROM DUAL; // NUMBER APPEAR AGAIN AND AGAIN AS IT IS CACHE**
* **ALTER SEQUENCE product\_seq NOCACHE;**
* **INSERT INTO products (product\_id,product\_name) VALUES (product\_seq.nextval,'any old product')**

**DELETE , TRUNCATE AND DROP COMMANDS**

**DELETE records**

* **DELETE FROM DEPT WHERE deptno = 40**
* **DELETE FROM DEPT WHERE dname IN ('ACCOUNTING')**
* **DELETE FROM dept //all records**

**DELETE CONSTRAINT**

* **ALTER TABLE emp**
* **DROP CONSTRAINT EMP\_DEPT\_FK**

**DROP TABLE : table will not exist in database**

* **DROP TABLE dept**
* **SELECT \* FROM dept -- table no longer exists**

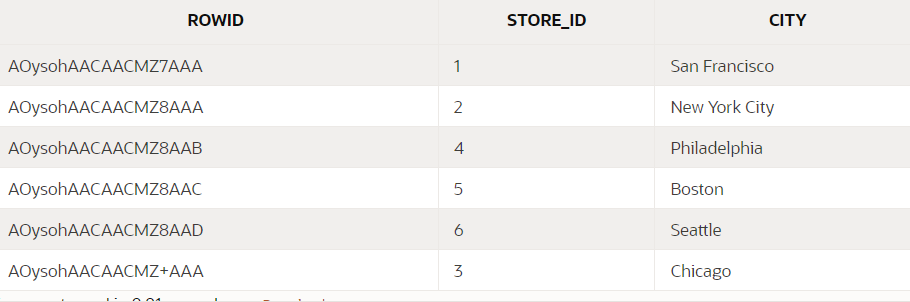
**TRUNCATE TABLE : delete all records from the table , but table exists**

* **TRUNCATE TABLE emp**
* **SELECT \* FROM emp -- no date found**

**INDEXES** : to retrieve/search data faster

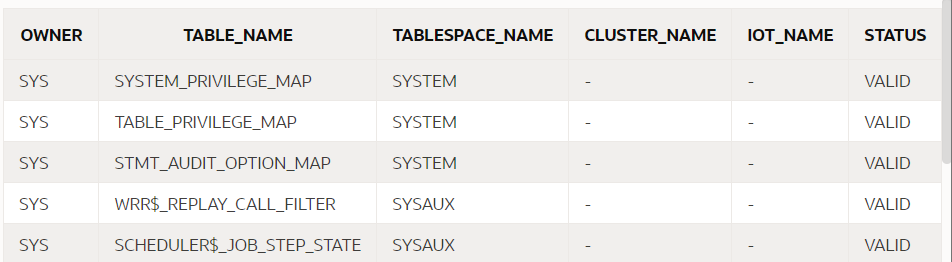
* CREATE [UNIQUE] INDEX index\_name
* ON table\_name (column1,column2….)
* [COMPUTE STATISTICS]
* Index are also db objects
* **CREATE INDEX emp\_name\_idx**
* **ON employees(ename)**
* Where clause works much faster now
* **CREATE INDEX emp\_name\_job\_date\_idx**
* **ON employees (ename, job, hiredate)**
* It is important to note that index takes a lot of space
* A unique index ensures that the indexed fields do not store duplicate values; i.e. enforces uniqueness for the indexed fields.
* CREATE UNIQUE INDEX emp\_job\_idx ON employees (job)
* **Drop index**  DROP INDEX emp\_job\_idx
* **COMPUTE STATISTICS :**WHEN YOU create new index , you can highlight this with the statement
* CREATE INDEX emp\_name\_job\_date\_idx
* ON employees (ename, job, hiredate)
* COMPUTE STATISTICS;
* ALTER INDEX emp\_name\_idx REBUILD COMPUTE STATISTICS

**SYSTEM TABLE,PSEUDO COLUMNS & DELETING DUPLICATES(NEWLY ADDED)**

* ROWID – PSEUDO COLUMN
* You can use this column in any query.Each row in the database has an address,and that address is represented in this string form,
* **SELECT rowid, store\_id, city FROM stores**
* 
* We can use row id to delete duplicate rows
* DELETE FROM stores
* WHERE rowid NOT IN(
* SELECT MIN(rowid)
* FROM STORES
* GROUP BY store\_id,city
* )

**SYSTEM TABLE :** They're basically being managed in the backend.You don't have to get involved with updating these or deleting these or anything like that.You can query these and find out more information about your database, and these are very, very handy to have in your practical career as well.

**SELECT \* FROM all\_tables where rownum<10**

****

**SELECT \* FROM all\_tab\_COLUMNS**

**WHERE table\_name = 'EMPLOYEES'**

**AND rownum < 10**

**SELECT \* FROM all\_tab\_COLUMNS**

**WHERE table\_name = 'EMPLOYEES'**

**SELECT \* FROM ALL\_OBJECTS**

**WHERE object\_type = 'TABLE'**

**AND ROWNUM < 50**

**SELECT \* FROM ALL\_OBJECTS**

**WHERE object\_type = 'INDEX'**

**AND lower(object\_name) = 'emp\_name\_idx'**

* **SYNONYMS :** IT is an alternative name for objects
* **CREATE SYNONYM emp\_table -- insufficient privileges**
* **FOR employees**

**VIEWS**

* It is the named query
* **CREATE OR REPLACE VIEW managers\_v**
* **AS SELECT \* FROM employees**
* **WHERE JOB='MANAGER'**
* CAN query view as a table
* **drop view managers\_v**

**UNION**

* The UNION operator is a set operator that combines result sets of two or more [SELECT](https://www.oracletutorial.com/oracle-basics/oracle-select/) statements into a single result set.
* The following illustrates the syntax of the UNION operator that combines the result sets of two queries:
* **SELECT**
* **column\_list\_1**
* **FROM**
* **T1**
* **UNION**
* **SELECT**
* **column\_list\_1**
* **FROM**
* T2;
* In this statement, the column\_list\_1 and column\_list\_2 must have the same number of columns presented in the same order. In addition, the data type of the corresponding column must be in the same data type group such as number or character.
* By default, the UNION operator returns the unique rows from both result sets. If you want to retain the duplicate rows, you explicitly use UNION ALL

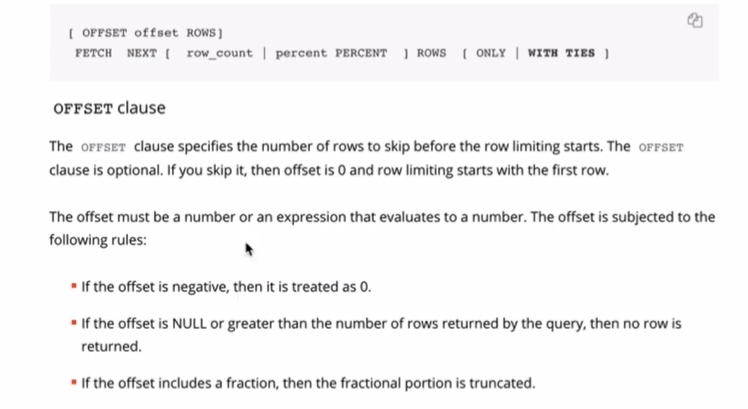
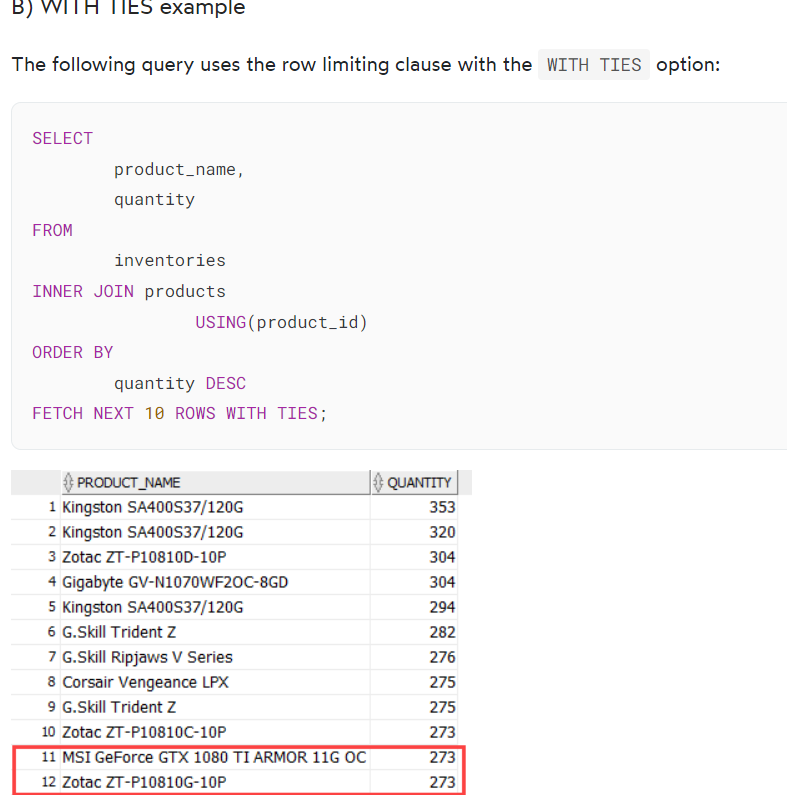
**MINUS**

* The Oracle MINUS operator is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement. Each SELECT statement will define a dataset. The MINUS operator will retrieve all records from the first dataset and then remove from the results all records from the second dataset.
* SELECT expression1, expression2, ... expression\_n
* FROM tables
* [WHERE conditions]
* MINUS
* SELECT expression1, expression2, ... expression\_n
* FROM tables
* [WHERE conditions];
* **There must be same number of expressions in both SELECT statements and have similar data types.**

**GRANTING AND REVOKING PRIVILEGES**

* **Dml – select, update, delete, insert**
* **Ddl – create, drop**
* **Dcl -data control language , granting and revoking privileges**
* **Schema =user**
* **GRANT –** grant privileges to other user to access database
* **GRANT select,update,delete on PRODUCTS TO U1,U2….**
* **REVOKE –** revoke the given privilege
* **revoke select,update,delete from PRODUCTS TO U1,U2….**
* As a database administrator I can create roles and assign them privileges
* CREATE ROLE R1
* Grant p1,p2,p3 to R1
* Show user // it see the user

**FETCH AND OFFSET**

* Oracle does not have limit key word to control on number of rows retrieved
* Instead it has a keyword called fetch
* ****
* **Offset clause** : it specifies the number of rows to skip before the row limiting starts
* ****
* ****
* Even though the query requested 10 rows because it had the WITH TIES option, the query returned two more additional rows. Notice that these two additional rows have the same value in the quantity column as the row 10.

### **Limit by percentage of rows example**

The following query returns the top 5% products with the highest inventory level:

SELECT

product\_name,

quantity

FROM

inventories

INNER JOIN products

USING(product\_id)

ORDER BY

quantity DESC

FETCH FIRST 5 PERCENT ROWS ONLY;

### The inventories table has 1112 rows, therefore, 5% of 1112 is 55.6 which is rounded up to 56 (rows).

### **OFFSET example**

The following query skips the first 10 products with the highest level of inventory and returns the next 10 ones:

SELECT

product\_name,

quantity

FROM

inventories

INNER JOIN products

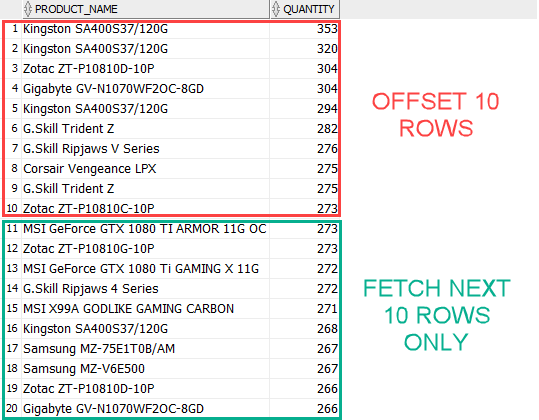
USING(product\_id)

ORDER BY

quantity DESC

OFFSET 10 ROWS

FETCH NEXT 10 ROWS ONLY;Code language: SQL (Structured Query Language) (sql)



* **FOREIGN KEY IS ON MANY SIDE OF TABLE**

**COMMIT ,ROLLBACK AND SAVEPOINT**

* Apex has auto commit(update,delete and insert)
* Tcl (transaction control language)
* **Commit-** make transaction permanent
* **Rollback-** undo ton previous commit or last save point
* **E.g. insert …**
* **Commit;**
* **Insert..**
* **Insert..**
* **Savepoint my\_savepoint; // temporary type ..not commited yet**
* **Update… //have done some mistake**
* **Rollback to my\_savepoint;**
* **Commit;**
* **Rollback ;// alone can do rollback to last commit not to savepoint**
* **Commit are of two types**
* **Implicit : no need to type commit**
* **Explicit : need to commit manually**

**LARGE OBJECTS AND INTERVAL DATA TYPES**

## Character Datatypes

The following are the **Character Datatypes** in Oracle/PLSQL:

| Data Type Syntax | Oracle 9i | Oracle 10g | Oracle 11g | Explanation |
| --- | --- | --- | --- | --- |
| char(size) | Maximum size of 2000 bytes. | Maximum size of 2000 bytes. | Maximum size of 2000 bytes. | Where **size** is the number of characters to store. Fixed-length strings. Space padded. |
| nchar(size) | Maximum size of 2000 bytes. | Maximum size of 2000 bytes. | Maximum size of 2000 bytes. | Where **size** is the number of characters to store. Fixed-length NLS string Space padded. |
| nvarchar2(size) | Maximum size of 4000 bytes. | Maximum size of 4000 bytes. | Maximum size of 4000 bytes. | Where **size** is the number of characters to store. Variable-length NLS string. |
| varchar2(size) | Maximum size of 4000 bytes.  Maximum size of 32KB in PLSQL. | Maximum size of 4000 bytes.  Maximum size of 32KB in PLSQL. | Maximum size of 4000 bytes.  Maximum size of 32KB in PLSQL. | Where **size** is the number of characters to store. Variable-length string. |
| long | Maximum size of 2GB. | Maximum size of 2GB. | Maximum size of 2GB. | Variable-length strings. (backward compatible) |
| raw | Maximum size of 2000 bytes. | Maximum size of 2000 bytes. | Maximum size of 2000 bytes. | Variable-length binary strings |
| long raw | Maximum size of 2GB. | Maximum size of 2GB. | Maximum size of 2GB. | Variable-length binary strings. (backward compatible) |

## Numeric Datatypes

The following are the **Numeric Datatypes** in Oracle/PLSQL:

| Data Type Syntax | Oracle 9i | Oracle 10g | Oracle 11g | Explanation |
| --- | --- | --- | --- | --- |
| number(p,s) | Precision can range from 1 to 38. Scale can range from -84 to 127. | Precision can range from 1 to 38. Scale can range from -84 to 127. | Precision can range from 1 to 38. Scale can range from -84 to 127. | Where **p** is the precision and **s** is the scale.  For example, number(7,2) is a number that has 5 digits before the decimal and 2 digits after the decimal. |
| numeric(p,s) | Precision can range from 1 to 38. | Precision can range from 1 to 38. | Precision can range from 1 to 38. | Where **p** is the precision and **s** is the scale.  For example, numeric(7,2) is a number that has 5 digits before the decimal and 2 digits after the decimal. |
| float |  |  |  |  |
| dec(p,s) | Precision can range from 1 to 38. | Precision can range from 1 to 38. | Precision can range from 1 to 38. | Where **p** is the precision and **s** is the scale.  For example, dec(3,1) is a number that has 2 digits before the decimal and 1 digit after the decimal. |
| decimal(p,s) | Precision can range from 1 to 38. | Precision can range from 1 to 38. | Precision can range from 1 to 38. | Where **p** is the precision and **s** is the scale.  For example, decimal(3,1) is a number that has 2 digits before the decimal and 1 digit after the decimal. |
| integer |  |  |  |  |
| int |  |  |  |  |
| smallint |  |  |  |  |
| real |  |  |  |  |
| double precision |  |  |  |  |

## Date/Time Datatypes

The following are the **Date/Time Datatypes** in Oracle/PLSQL:

| Data Type Syntax | Oracle 9i | Oracle 10g | Oracle 11g | Explanation |
| --- | --- | --- | --- | --- |
| date | A date between Jan 1, 4712 BC and Dec 31, 9999 AD. | A date between Jan 1, 4712 BC and Dec 31, 9999 AD. | A date between Jan 1, 4712 BC and Dec 31, 9999 AD. |  |
| timestamp (fractional seconds precision) | ***fractional seconds precision*** must be a number between 0 and 9. (default is 6) | ***fractional seconds precision*** must be a number between 0 and 9. (default is 6) | ***fractional seconds precision*** must be a number between 0 and 9. (default is 6) | Includes year, month, day, hour, minute, and seconds.  For example: timestamp(6) |
| timestamp (fractional seconds precision) with time zone | **fractional seconds precision** must be a number between 0 and 9. (default is 6) | **fractional seconds precision** must be a number between 0 and 9. (default is 6) | **fractional seconds precision** must be a number between 0 and 9. (default is 6) | Includes year, month, day, hour, minute, and seconds; with a time zone displacement value.  For example: timestamp(5) with time zone |
| timestamp (fractional seconds precision) with local time zone | **fractional seconds precision** must be a number between 0 and 9. (default is 6) | **fractional seconds precision** must be a number between 0 and 9. (default is 6) | **fractional seconds precision** must be a number between 0 and 9. (default is 6) | Includes year, month, day, hour, minute, and seconds; with a time zone expressed as the session time zone.  For example: timestamp(4) with local time zone |
| interval year (year precision) to month | **year precision** is the number of digits in the year. (default is 2) | **year precision** is the number of digits in the year. (default is 2) | **year precision** is the number of digits in the year. (default is 2) | Time period stored in years and months.  For example: interval year(4) to month |
| interval day (day precision) to second (fractional seconds precision) | **day precision** must be a number between 0 and 9. (default is 2)  **fractional seconds precision** must be a number between 0 and 9. (default is 6) | **day precision** must be a number between 0 and 9. (default is 2)  **fractional seconds precision** must be a number between 0 and 9. (default is 6) | **day precision** must be a number between 0 and 9. (default is 2)  **fractional seconds precision** must be a number between 0 and 9. (default is 6) | Time period stored in days, hours, minutes, and seconds.  For example: interval day(2) to second(6) |

## Large Object (LOB) Datatypes

The following are the **LOB Datatypes** in Oracle/PLSQL:

| Data Type Syntax | Oracle 9i | Oracle 10g | Oracle 11g | Explanation |
| --- | --- | --- | --- | --- |
| bfile | Maximum file size of 4GB. | Maximum file size of 232-1 bytes. | Maximum file size of 264-1 bytes. | File locators that point to a binary file on the server file system (outside the database). |
| blob | Store up to 4GB of binary data. | Store up to (4 gigabytes -1) \* (the value of the CHUNK parameter of LOB storage). | Store up to (4 gigabytes -1) \* (the value of the CHUNK parameter of LOB storage). | Stores unstructured binary large objects. |
| clob | Store up to 4GB of character data. | Store up to (4 gigabytes -1) \* (the value of the CHUNK parameter of LOB storage) of character data. | Store up to (4 gigabytes -1) \* (the value of the CHUNK parameter of LOB storage) of character data. | Stores single-byte and multi-byte character data. |
| nclob | Store up to 4GB of character text data. | Store up to (4 gigabytes -1) \* (the value of the CHUNK parameter of LOB storage) of character text data. | Store up to (4 gigabytes -1) \* (the value of the CHUNK parameter of LOB storage) of character text data. | Stores [Unicode](https://www.techonthenet.com/unicode/chart.php) data. |
|  |  |  |  |  |

## Rowid Datatypes

The following are the **Rowid Datatypes** in Oracle/PLSQL:

| Data Type Syntax | Oracle 9i | Oracle 10g | Oracle 11g | Explanation |
| --- | --- | --- | --- | --- |
| rowid | The format of the rowid is: BBBBBBB.RRRR.FFFFF  Where BBBBBBB is the block in the database file; RRRR is the row in the block; FFFFF is the database file. | The format of the rowid is: BBBBBBB.RRRR.FFFFF  Where BBBBBBB is the block in the database file; RRRR is the row in the block; FFFFF is the database file. | The format of the rowid is: BBBBBBB.RRRR.FFFFF  Where BBBBBBB is the block in the database file; RRRR is the row in the block; FFFFF is the database file. | Fixed-length binary data. Every record in the database has a physical address or **rowid**. |
| urowid(size) |  |  |  | Universal rowid.  Where **size** is optional. |

* **Large datatypes cannot be used with primary key ,orderby ,group by , distinct**

**USING CASCADE CONSTRAINTS AND ON DELETE CASCADE**

CREATE TABLE products

( product\_id numeric(10) not null,

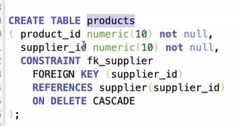
supplier\_id numeric(10) not null,

CONSTRAINT fk\_supplier

FOREIGN KEY (supplier\_id)

REFERENCES supplier(supplier\_id)

);

* A parent table cannot be deleted if it has child table , need to delete child table first
* We can use cascade constraint to delete parent table
* ALTER TABLE supplier DROP COLUMN supplier\_id CASCADE CONSTRAINTS
* CASCADE DELETE – IF PARENT TABLE RECORD DELETE THEN CHILD WILL GET DELETED
* 
* 

**UNUSED COLUMNS**

* unused columns are not physically dropped, but are treated as if they were. They can't be restored, and select statements won't retrieve data from columns marked as unused. NOT ROLLBACK POSSIBLE
* One advantage of setting a column to unused is to reduce possible high database load when dropping a column from a large table. To set a column as unused, you can use the SET UNUSED clause:
* Set a column as unused: ALTER TABLE table\_name SET UNUSED COLUMN column\_name
* Mark a column as unused: Before dropping, you can mark a column as unused by using the Column Usage feature
* **SELECT \* FROM USER\_UNUSED\_COL\_TABS // TABLE CONTAINING UNUSED COLUMNS**

**EXTERNAL TABLES**

* External tables in Oracle allow users to query data stored outside of the database in flat files. The contents of external files are treated as rows in a table in the Oracle database
* External tables are created using the SQL CREATE TABLE... ORGANIZATION EXTERNAL statement. Data files and output files must be located on the server. A directory object is also required that specifies the location from which to read and write files.

**SQL PLUS AND SUBSTITUTION VARIABLES**