Database Programming with SQL

محمد إبراهيم الدسوقى

محاضر بكلية هندسة وعلوم الحاسب – قسم نظم المعلومات

جامعة سلمان بن عبد العزيز - السعودية – محافظة الخرج

Email : mohamed_eldesouki@hotmail.com

SQL: CREATE TABLE Statement

The basic syntax for a CREATE TABLE statement is:

```
CREATE TABLE table_name
( column1 datatype null/not null,
  column2 datatype null/not null,
  ...
);
```

- Each column must have a datatype.
- The column should either be defined as "null" or "not null“
- if this value is left blank, the database assumes "null" as the default.

A list of general SQL datatypes

Data Type	Syntax	Explanation
Numeric	**number(p,s)**	Where *p* is a precision value; s is a scale value. For example, numeric(6,2) is a number that has 4 digits before the decimal and 2 digits after the decimal.
Character	**char(x)**	Where *x* is the number of characters to store. This data type is space padded to fill the number of characters specified.
Character varying	**varchar2(x)**	Where *x* is the number of characters to store. This data type does NOT space pad.
bit	**bit(x)**	Where *x* is the number of bits to store.
Date	**date**	Stores year, month, and day values.

SQL: CREATE TABLE Statement

For example:

```
CREATE TABLE suppliers
(     supplier_id        number(10)      not null,
      supplier_name      varchar2(50)    not null,
      contact_name       varchar2(50)
);
```

suppliers

supplier_id	supplier_name	contact_name

SQL: CREATE TABLE Statement

For example:

```
CREATE TABLE customers
(       customer_id     number(10)      not null,
        customer_name   varchar2(50)    not null,
        address         varchar2(50),
        city            varchar2(50),
);
```

customers

customer_id	customer_name	address	city

SQL: Drop TABLE Statement

The DROP TABLE statement allows you to remove a table from the database.

-The basic syntax for the DROP TABLE statement is:

DROP TABLE table_name;

For example:

DROP TABLE supplier;

-This would drop table called *supplier*.

SQL :Working With Data

1- Insert into statement

Used to insert new data rows into the Table.

2- Update statement

Used to Modify Existing data values in the Table.

3- Delete statement

Used to Delete Existing data Rows from The Table.

SQL :Insert into statement (1)

The INSERT statement allows you to insert a new data row into a table.

The syntax for the INSERT statement is:

INSERT INTO table_name

VALUES (value-1, value-2, ... value-n);

Here ,You must apply the column order as the table organized

SQL :Insert into statement (1)

For Example

INSERT INTO suppliers

VALUES (100, 'IBM', 'Mr Hassan');

SQL :Insert into statement (2)

The INSERT statement allows you to insert a single record or multiple records into a table.

The syntax for the INSERT into statement is:

INSERT INTO table_name (column-1, column-2, ... column-n)

VALUES (value-1, value-2, ... value-n);

Here ,You can specify the column order as you wish

SQL :Insert into statement (2)

For Example :

```
INSERT INTO Supplier(supplier_name , supplier_id ,
contact_name)
VALUES ('IBM', 100,'Mr Mohamed');
```

SQL :Update statement (1)

The UPDATE statement allows you to update a single record or multiple records in a table.

The syntax for the Update statement is:

UPDATE table

SET column = expression

Here ,You apply the change to all the values stored in this column

SQL :Update statement (1)

For Example:

```
UPDATE supplier
SET name = 'HP'
```

Here , All the values of the name column will be changed to HP

SQL :Update statement (2)

The UPDATE statement allows you to update a single record or multiple records in a table.

The syntax for the Update statement is:

UPDATE table

SET column = expression

Where condition

Here ,You apply the change to all the values stored in this column

The Where Clause

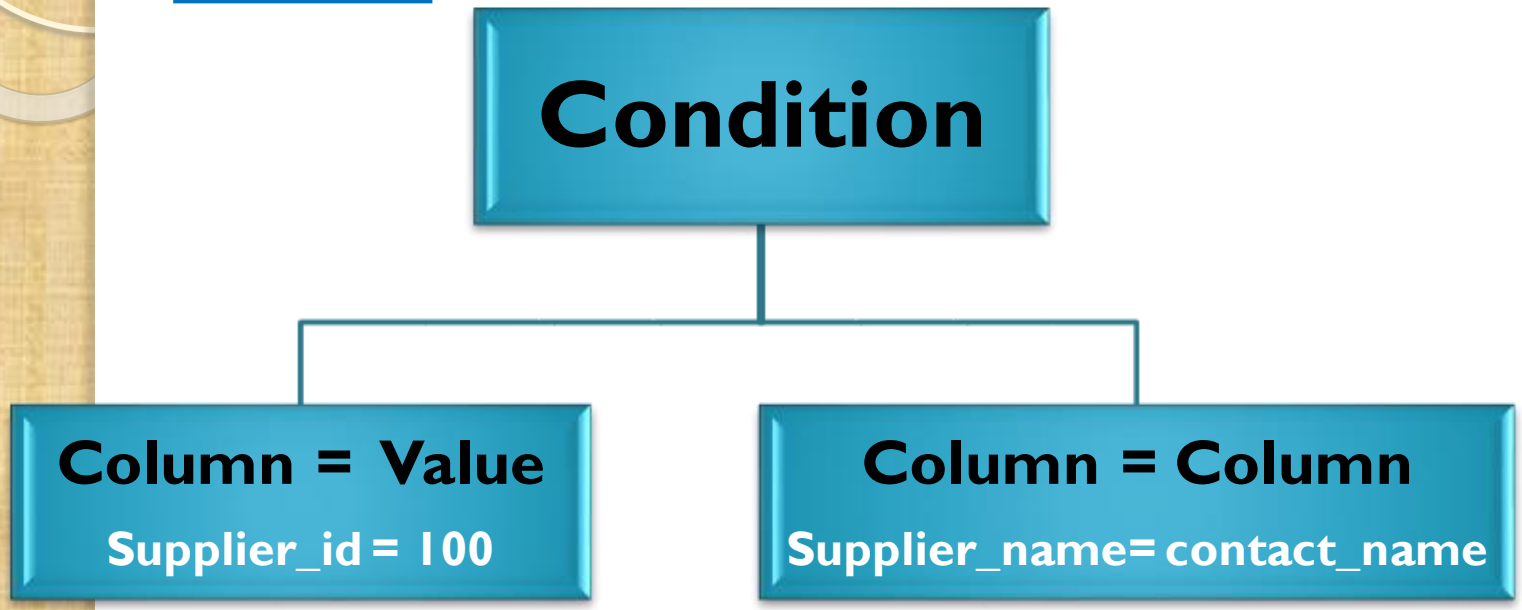
The WHERE clause allows you to filter the results from any SQL statement - insert, update, or delete statement.

The syntax for the Where clause is:

Where <Condition>

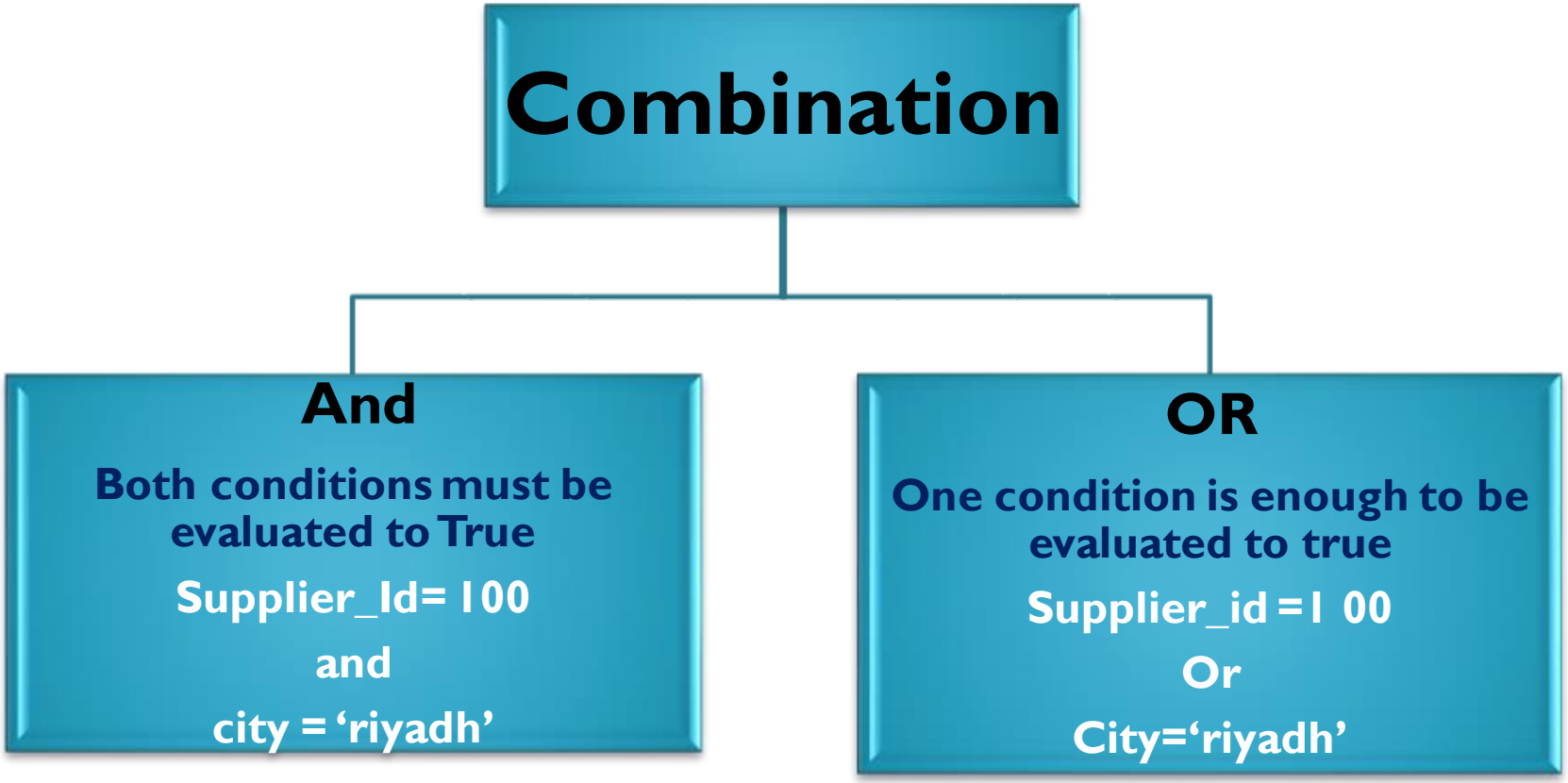
The Where Clause

Where <Condition>



The Where Clause

We can combine more than one condition



SQL :Update statement (1)

For Example:

```
UPDATE supplier
SET supplier_name = 'HP'
Where supplier_name  = 'IBM'
```

Here , only the supplier_name with the value IBM will be changed to HP

SQL :Delete statement (1)

The DELETE statement allows you to delete a single record or multiple records from a table.

The syntax for the Delete statement is:

```
DELETE FROM table_name
```

Here , You Delete all the data rows from the table

SQL :Delete statement (1)

For Example:

```
DELETE FROM Supplier
```

Here , You Delete all the data rows from the supplier table

SQL :Delete statement (2)

The DELETE statement allows you to delete a single record or multiple records from a table.

The syntax for the Delete statement is:

DELETE FROM table_name

Where Condition

Here , You Delete only the data rows which meet the where condition

SQL :Delete statement (2)

For Example:

```
DELETE FROM Supplier
Where supplier_name ='HP'
```

Here , You Delete only the data rows that meet the where condition

Retrieving Data

1- The Select Statement

2- The Where Clause

3- Combine conditions using " And " – "Or" .

4- Using Like , IN , Between and Not

Select Statement (1)

The SELECT statement allows you to retrieve records from one or more tables in your database.

The syntax for the SELECT statement is:

SELECT columns

FROM tables

For example

SELECT supplier_id , Supplier_name

FROM suppliers

Select Statement (2)

SELECT columns

FROM tables

WHERE predicates;

```
SELECT *
FROM suppliers
WHERE city = 'Newark';

SELECT name, city, state
FROM suppliers
WHERE supplier_id > 1000;
```

The Where Clause

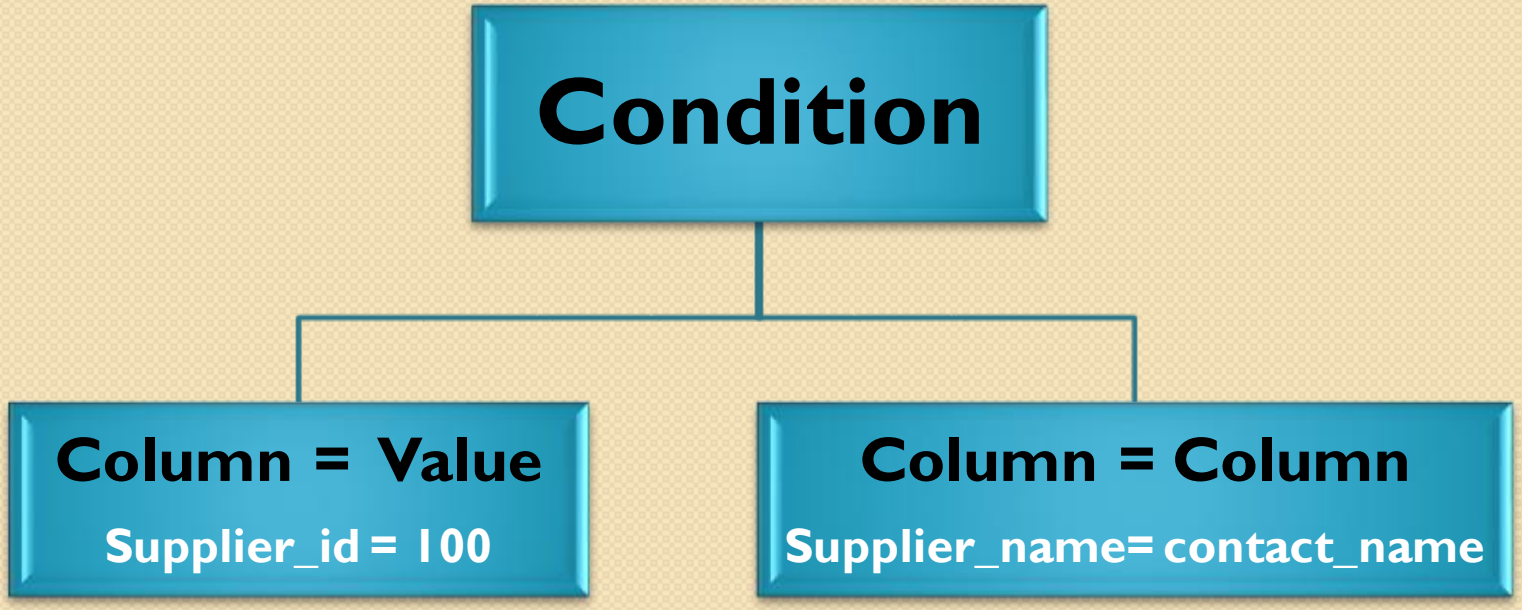
The WHERE clause allows you to filter the results from any SQL statement - insert, update, or delete statement.

The syntax for the Where clause is:

Where <Condition>

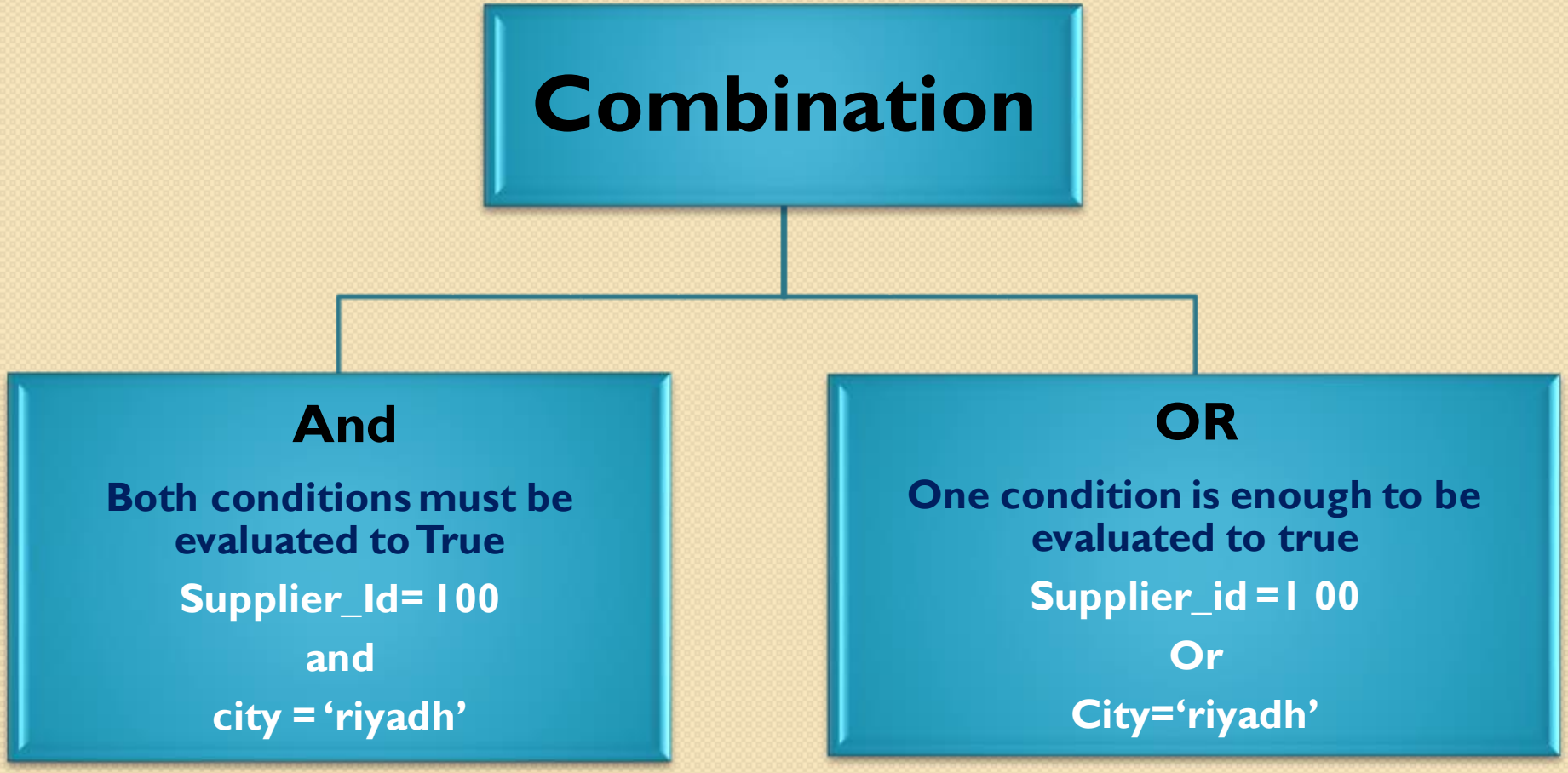
The Where Clause

Where **<Condition>**



The Where Clause

We can combine more than one condition



SQL: "AND" Condition

The syntax for the AND condition is:

```
SELECT columns
FROM tables
WHERE column1 = 'value1' and column2 = 'value2';
```

```
SELECT *
FROM suppliers
WHERE city = 'New York' and type = 'PCs' ;
```

SQL: "OR" Condition

The syntax for the OR condition is:

```
SELECT columns
FROM tables
WHERE column1 = 'value1' or column2 = 'value2';
```

```
SELECT *
FROM suppliers
WHERE city = 'New York' or Type = 'Software';
```

SQL: LIKE Condition

The LIKE condition allows you to use wildcards in the *where* clause of an SQL statement. This allows you to perform pattern matching.

The patterns that you can choose from are:

% *allows you to match any string of any length (including zero length)*

_ *allows you to match on a single character*

Examples using % wildcard

```
SELECT * FROM suppliers
WHERE city like 'new %';
```

```
SELECT * FROM suppliers
WHERE contact_name like '%Ahmed%';
```

SQL: LIKE Condition

Examples using _ wildcard

```
SELECT * FROM suppliers
WHERE contact_name like '_mr';

SELECT * FROM suppliers
WHERE contact_name like '_mr %';
```

SQL: "IN" Function

The IN function helps reduce the need to use multiple *OR* conditions.

The syntax for the IN function is:

```
SELECT columns
FROM tables
WHERE column1 in (value1, value2, .... value_n);
```

SQL: "IN" Function

```
SELECT *
FROM suppliers
WHERE supplier_name in ( 'IBM', 'HP', 'Microsoft');
```

```
SELECT *
FROM suppliers
WHERE supplier_name = 'IBM'
OR supplier_name = 'HP'
OR supplier_name = 'Microsoft';
```

SQL: Not "IN" Function

```
SELECT *
FROM suppliers
WHERE supplier_name  Not In ( 'IBM', 'H P',
'Microsoft');
```

SQL: BETWEEN Condition

The BETWEEN condition allows you to retrieve values within a range.

The syntax for the BETWEEN condition is:

SELECT columns

FROM tables

WHERE column1 between *value1 and value2;*

```
SELECT *
FROM suppliers
WHERE supplier_id between 5000 AND 5010;
```

```
SELECT *
FROM suppliers
WHERE supplier_id >= 5000
AND supplier_id <= 5010;
```

SQL: Not BETWEEN Condition

```
SELECT *
FROM suppliers
WHERE supplier_id not between 5000 and 5500;
```

Retrieve Data from More Than one Table : Join Tables

- A join is used to combine rows from multiple tables.

The Basic Syntax for join tables is

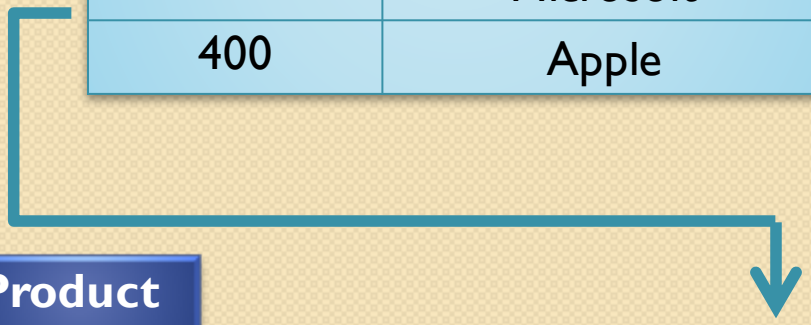
```
Select Columns

From Table1 Join Table2

On Table1.JoinField = Table2.JoinField
```

Supplier

supplier_id	supplier_name
100	IBM
200	HP
300	Microsoft
400	Apple



Product

product_Id	Product_name	sup_id	Price
1	IPAD 2	400	2400
2	IPHONE 4s	400	2500
3	MS Office 2012	300	1600
4	Color Printer	100	1500

Retrieve Data from More Than one Table : Join Tables

For Example:

```
Select Supplier_name , Product_name , Price
From Supplier Join Product
On Supplier.Supplier_id = Product.Sup_id
```

```
Select Supplier_name , Product_name , Price
From Supplier , Product
where Supplier.Supplier_id = Product.Sup_id
```

Customer

Customer_id	Customer_name
100	Mohamed
200	Ahmed
300	Hassan
400	Mostafa

Order

Cust_id	product_Id	Ord_Date
100	1	1/1/2012
200	2	2/5/2011
100	3	10/4/2011
300	4	23/2/2012

Product

product_Id	Product_name	supplier_id
1	IPAD 2	400
2	IPHONE 4s	400
3	MS Office 2012	300
4	Color Printer	100

Join Tables

```
Select Customer_name ,  Product_name , Ord_date
From customer ,Order , product
Where customer_id = cust_id and product.product_id = order.product_id
```

```
Select Customer_name ,  Product_name , Ord_date
From customer , Order , product
Where customer_id = cust_id and product.product_id = order.product_id
and price > 2000
```

Retrieving Data : Aggregate Functions

SQL aggregate functions return a single value, calculated from values in a column

Useful aggregate functions:

SUM(column x) - Returns the sum of the values stored in Column x

Avg(column x) - Returns the Average of the values stored in Column x

Count(column x) - Returns the count of the values stored in Column x

Max(column x) - Returns the Maximum value in the values stored in Column x

Min(column x) - Returns the Minimum value in the values stored in Column x

Retrieving Data : Aggregate Functions

The Basic Syntax for using the Aggregate functions is

```
Select AggregateFunctionName ( columnName)
From Table
Where conditions
```

For Example

```
Select Count (Empno) from emp;

Select Max(sal) from emp;

Select Sum(sal),Avg (sal) from emp;

Select Avg(sal) from emp
Where deptno = 20;
```

Retrieving Data : Group by clause

The GROUP BY clause can be used in a SELECT statement to collect data across multiple records and group the results by one or more columns.

The syntax for the GROUP BY clause is:

SELECT column1, column2,... column_n,aggregate_function (expression)

FROM tables

WHERE predicates

GROUP BY column1, column2,... column_n;

Aggregate_function can be a function such as

Sum, Avg , Max , Min or any other valid Aggregate_function

Retrieving Data : Group by clause

Examples

Display a list of each depart and how many employees assigned to it

```
SELECT deptno , COUNT(*)
FROM emp
GROUP BY deptno;
```

For each depart find the depart no and how many employees who get salary over 1500

```
SELECT deptno , COUNT(*)
FROM emp
Where sal > 1500
GROUP BY deptno;
```

Retrieving Data : Group by clause

Examples

Display a list of each depart and the sum and the average of it's employees saliries.

```
SELECT deptno , sum(sal) , avg (sal)
FROM emp
GROUP BY deptno;
```

Group by clause - Having

The HAVING clause is used in combination with the GROUP BY clause. It can be used in a SELECT statement to filter the records that a GROUP BY returns.

The syntax for the HAVING clause is:

```
SELECT column1, column2, ... column_n, aggregate_function (expression)
FROM tables
WHERE predicates
GROUP BY column1, column2, ... column_n
HAVING condition1 ... condition_n;
```

Group by clause - Having

Examples

Display a list of departments that have more than 3 employees

```
SELECT deptno , Count ( * )
FROM  Emp
GROUP BY deptno
HAVING Count ( * ) > 3 ;
```

Display a list of departments that have at least 2 employees working as SALESMAN

```
SELECT deptno , Count ( * )
FROM  Emp
Where job ='SALESMAN '
GROUP BY deptno
HAVING Count ( * ) > 3 ;
```

Retrieving Data : Order by clause

The Order BY clause allows you to sort the records in your result set. The ORDER BY clause can only be used in SELECT statements.

The syntax for the Order BY clause is:

SELECT columns

FROM tables

WHERE predicates

ORDER BY column **ASC/DESC;**

ASC indicates ascending order. (default)

DESC indicates descending order.

Retrieving Data : Order by clause

For Example :

```
SELECT  empno , ename
FROM emp
ORDER BY ename  DESC;
```

For Example :

```
SELECT  empno , ename
FROM emp
Where deptno = 20
ORDER BY ename  ASC;
```

Retrieving Data : Order by clause

For Example :

```
SELECT  deptno , empno , sal
FROM emp
ORDER BY deptno ASC, sal DESC;
```

Revision

What will be covered ?

1- Working With Tables and Constraints

- Create Table
- Alter Table
- Drop Table

2- Working With Data

- Insert Into Statement
- Update Statement
- Delete Statement

3- Retrieving Data

- Apply Different Queries

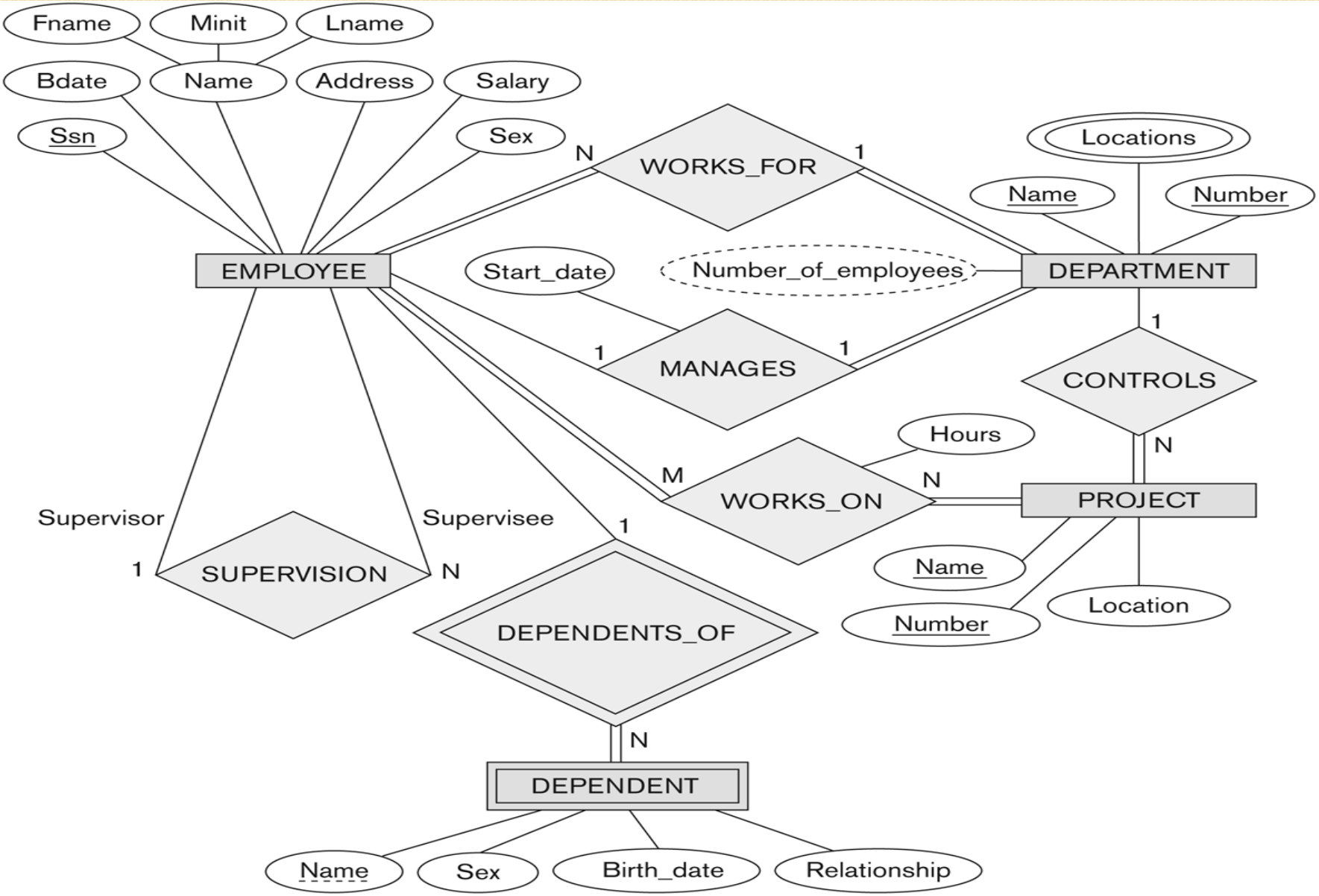
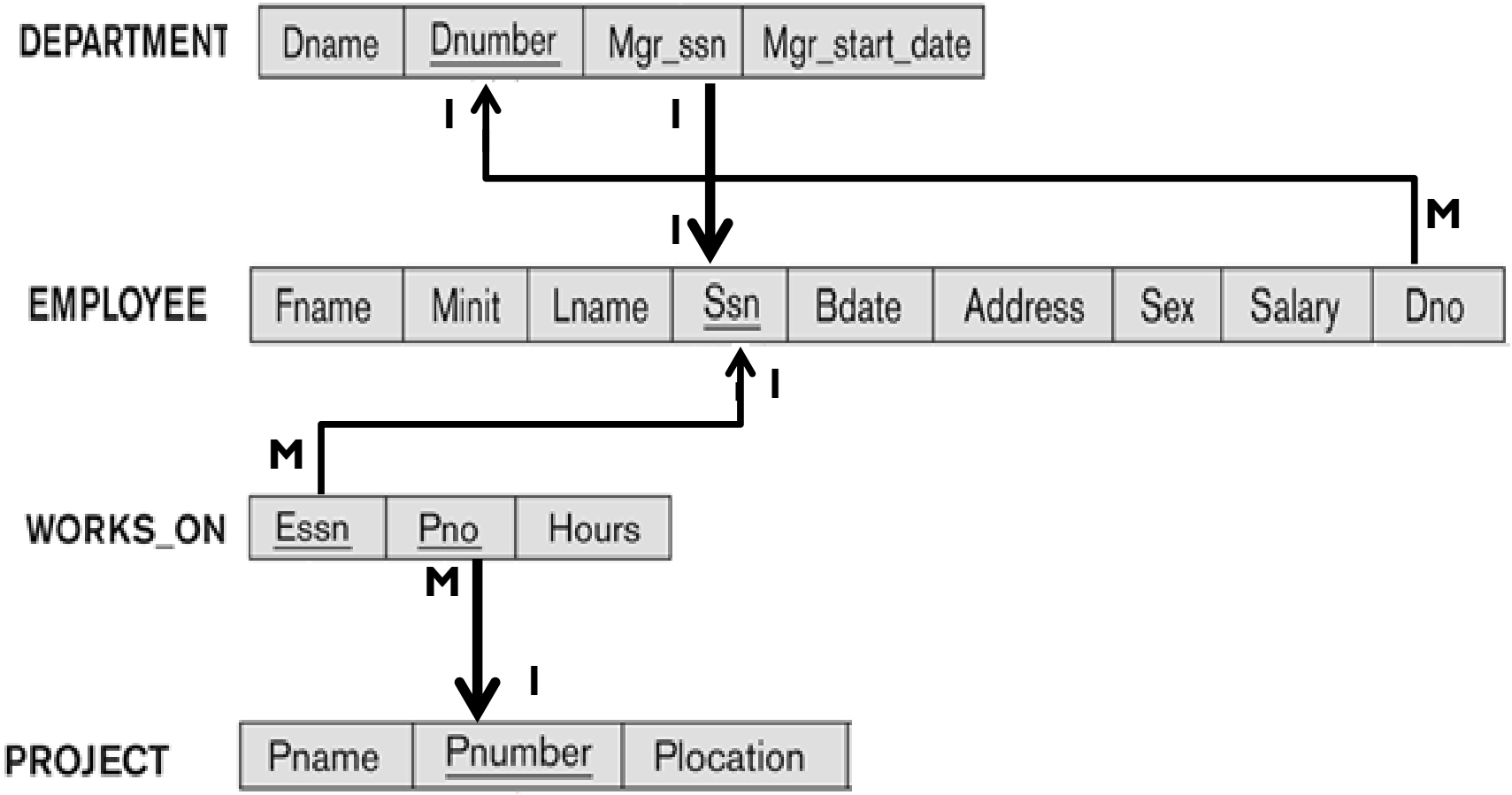


Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

Consider The Following Schema for a company database and answer the following questions.



Q1 : Create all tables of this schema and apply any necessary constraints.

Q2 : Write SQL statements that carry out the following tasks:-

Q2-1 : Alter the employee table to add column city of 20 characters.

Q2-2 : Alter the employee table to apply the following business rule

The minimum salary is 3000 and maximum salary is 10000

Q2-3 : Insert sample data in each table.

Q2-4: display a list of each Employee SSN , Name and Salary.

Q2-5: display a list of Employee SSN , Name and Salary for employees working in depart no 2.

Q2-5: display a list of department name , Employee Name and Salary Sorted by the salary from the highest to the lowest .

Q2 : Write SQL statements that carry out the following tasks:-

Q2-1 : Display a list of employee data for employees with First name that is same as their last name.

Q2-2 : Display a list of employee data for employees with First names starting with "A".

Thank You

Mohamed El Desouki

Lecturer , College of Computer Engineering And Sciences

Salman Bin AbduAlziz University, KSA, Kharj

Email : mohamed_eldesouki@hotmail.com

Selected Topics In SQL / PL SQL

محمد إبراهيم الدسوقى

محاضر بكلية هندسة وعلوم الحاسب – قسم نظم المعلومات

جامعة سلمان بن عبد العزيز - السعودية – محافظة الخرج

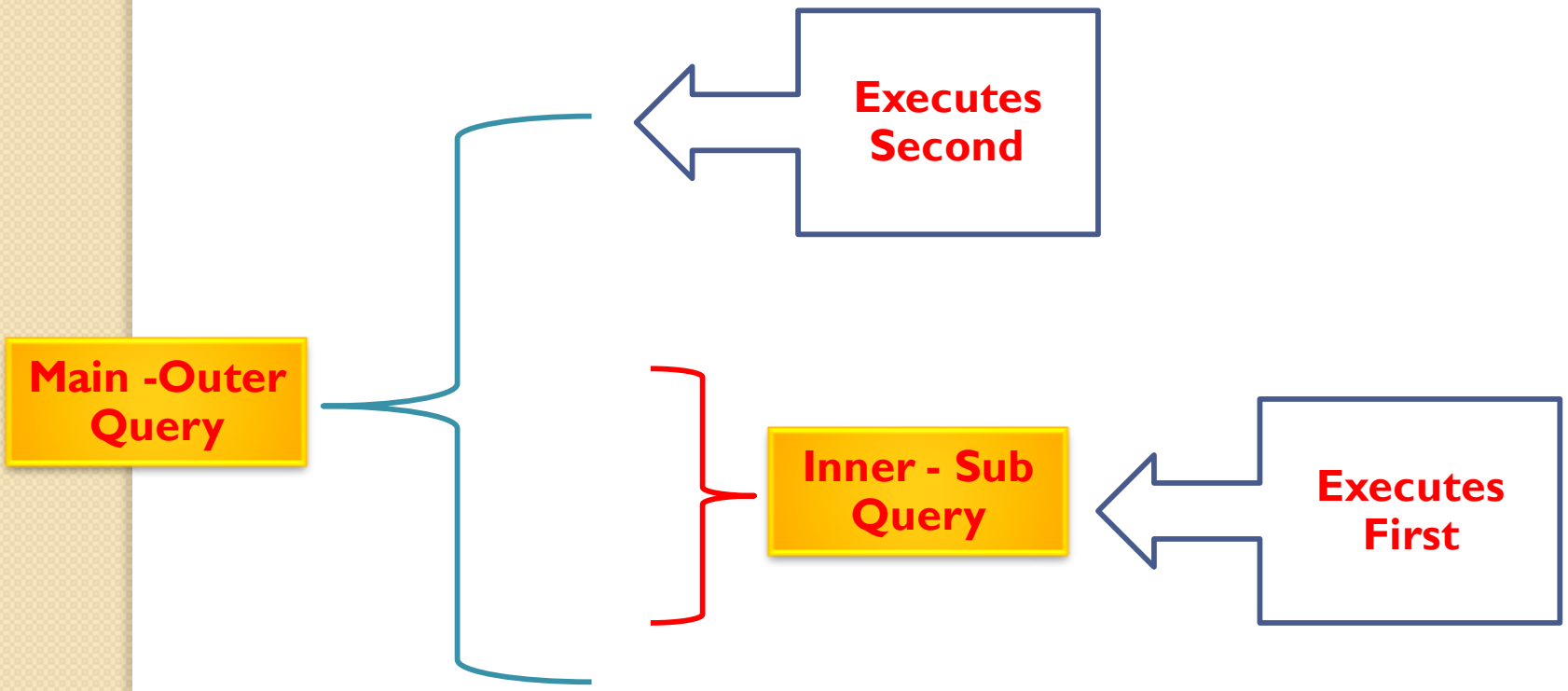
Email : mohamed_eldesouki@hotmail.com

Advanced Topics In SQL / PL SQL

- SQL Sub Queries
- SQL Views
- PL SQL Condition Statements
  1. IF – Then – Else
  2. Case Statement
- PL SQL LOOPs
  1. Loop Statement
  2. For – Loop Statement
  3. While – Loop Statement
- Cursors
- SQL Procedures
- SQL Functions
- SQL Triggers

Subqueries – Nested Queries

- A subquery is a query within a query.
- A subquery is a select statement within another select statement



Subqueries

- **A subquery Can be Categorized as :**

  1- Single Row Subquery : queries that return only a single value.

  Used with **= , <> , >= , <=** Operators.

  2- Multiple Row SubQuery : queries that return more than one row.

  Used with **IN , Exist, Any** Operators.

Subqueries

Using the SubQuery in the <u>WHERE clause</u>

Example: Display the ssn and last name for the employee who Work in the same department that employee named 'Waleed Ahmed' works in.

Here, we Have 2 queries

Query 1 : Retrieve the department that Employee 'Waleed Ahmed' Works on.

```
select dno from employee
where fname ='Waleed' and lname = 'Ahmed' ;
```

4

Query 2 : who are the employees that belong to the department retrieved by Query 1.

```
Select ssn , lname
From Employee Where dno=
```

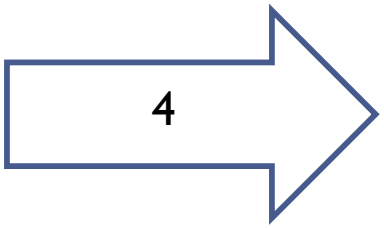
4

Subqueries

Query 1 : Retrieve the department that Employee 'Waleed Ahmed' Works on.

```
select dno from employee
where fname ='Waleed' and lname = 'Ahmed' );
```

Sub Query



Query 2 : who are the employees that belong to the department retrieved by Q1.

```
Select ssn , lname
From Employee Where dno= (
```

Main Query

Subqueries

Using the SubQuery in the WHERE clause

Example: Display the ssn and last name for the employee who got the maximum salary value.

```
Select ssn , lname

From Employee

Where salary = (select max (salary) from employee )
```

Subqueries

Using the SubQuery in the WHERE clause

Example: For All Managers ,Display a list of their last name , first name and ssn

Select ssn , fname , lname

From Employee

where ssn in (select mgr_ssn from department)

Mgr_SSN

Subqueries

Using the SubQuery in the WHERE clause

Example: Display depratment no and name for any department Managed by Female Manager.

Select Dnumber , Dname

From Department

Where Mgr_SSN in (select ssn from employee where sex ='Female');

SSN

Subqueries

Using the SubQuery in the From clause

Example : Display a list of employee last name , first name and ssn

Select ssn , fname , lname

From (select * from Employee)

SSN	Fname	Lname	SEX	Salary
-----	---------	----------	-----	-----
-----	--------	---------	-----	-----
-----	----------	----------	-----	-----
-----	---------	---------	-----	-----

Subqueries

Using the SubQuery in the From clause

Example : For all managers Display a list of last name , first name and the managed department

Select ssn , fname , lname , dname

From Employee e , (select * from department) sub

Where e.ssn= sub.mgr_ssn

dno	Dname	Mgr_SSN
-----	---------	----------
-----	--------	---------
-----	----------	----------
-----	---------	---------

Subqueries

Using the SubQuery in the Select Clause

Example : Display a list of department name and number of working employees in each department.

```
Select DNAME , (select count(*) from employee where
dno = department. dnumber ) as "working employees"
from department
```

Selected Topics In SQL / PL SQL

محمد إبراهيم الدسوقى

محاضر بكلية هندسة وعلوم الحاسب – قسم نظم المعلومات

جامعة سلمان بن عبد العزيز - السعودية – محافظة الخرج

Email : mohamed_eldesouki@hotmail.com

Advanced Topics In SQL / PL SQL

- ~~SQL Sub Queries~~
- SQL Views
- PL SQL Condition Statements
  I. IF – Then – Else
  II. Case Statement
- PL SQL LOOPs
  I. Loop Statement
  II. For – Loop Statement
  III. While – Loop Statement
- Cursors
- SQL Procedures
- SQL Functions
- SQL Triggers

SQL:VIEWS

- A view is a virtual (logical) table. It does not physically exist.
- It is created by a query joining one or more tables.
- A view derives its data from the tables on which it is based
- Views can be based on actual tables or another view also.
- You can Query, Insert, Update and delete from views, just as any other table.

SQL:VIEWS

The syntax for creating a VIEW is:

```
CREATE  or Replace VIEW view_name  AS
SELECT columns
FROM table
WHERE conditions;
```

SQL:VIEWS

For Example : A view based on one table

```
CREATE or Replace VIEW Vw_Managers AS
SELECT empno , ename , job, sal
FROM emp
WHERE  job = 'MANAGER';
```

Query :

```
Select *  from Vw_Managers ;
```

SQL:VIEWS

For Example : A view based on more than one table

```
CREATE or Replace VIEW Accounting_employees AS
SELECT dname ,empno , ename , job
FROM dept , emp
WHERE dept.deptno = emp.deptno
And dname = 'ACCOUNTING'
```

Query :

```
Select * from Accounting_employees ;
```

SQL:VIEWS : Insert , Update , Delete

```
insert into Vw_Managers
values(9876,'desouki','MANAGER',9000);

Update  Vw_Managers
Set  ename = 'Mohamed'
Where  ename = 'desouki';

Delete from  Vw_Managers
Where  ename = 'desouki';
```

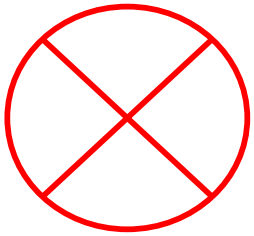
SQL:VIEWS WITH CHECK OPTION

WITH CHECK OPTION creates the view with the constraint that INSERT and UPDATE statements issued against the view are not allowed to create or result in rows that the view cannot select.

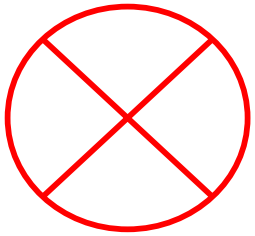
```
CREATE or Replace VIEW Vw_Managers AS
SELECT empno , ename , job, sal  FROM emp
WHERE  job = 'MANAGER'
With check option;
```

SQL: VIEWS : Insert , Update , Delete

```
insert into Vw_Managers
Values(654,'Ahmed','Analyst',9000);
```



```
Update  Vw_Managers
Set  job= 'Clerck' ;
```



Selected Topics In SQL / PL SQL

محمد إبراهيم الدسوقى

محاضر بكلية هندسة و علوم الحاسب – قسم نظم المعلومات

جامعة سلمان بن عبد العزيز- السعودية – محافظة الخرج

Email : mohamed_eldesouki@hotmail.com

Advanced Topics In SQL / PL SQL

- ~~SQL Sub Queries~~
- ~~SQL Views~~
- Introduction to PL / SQL
- PL SQL Condition Statements
  I. IF – Then – Else
  II. Case Statement
- PL SQL LOOPs
  I. Loop Statement
  II. For – Loop Statement
  III. While – Loop Statement
- Cursors
- SQL Procedures
- SQL Functions
- SQL Triggers

Purpose of PL/SQL

•PL / SQL stands for **P**rocedural **L**anguage / SQL

- PL/SQL Designed to overcome SQL's inability to handle control aspects of database interaction.

- Extends SQL by adding procedural language constructs, such as:
  - Variables and types.
  - Control structures (IF-THEN-ELSE statements and loops).
  .....

- Procedural constructs are integrated seamlessly with Oracle SQL, resulting in a structured, powerful language.

- Well-suited for designing complex applications.

PL/SQL Block

- Block is a basic unit in PL/SQL.
- All PL/SQL programs consist of blocks and each block performs a logical function in the program.
- Blocks can be nested within each other or can occur sequentially.

PL/SQL Block Structure

DECLARE

(Declarative section)

BEGIN

(Executable section)

EXCEPTION

(Exception handling section)

END;

/ *(/ at the end of a block tells Oracle to run the block)*

PL/SQL Block Structure

- To enable the messages to be displayed on the screen, you have to execute the following command:

```
Set serveroutput on;
```

- To write any message on the screen, use

```
dbms_output.put_line ( message);
```

PL/SQL Syntax – Variables Declaration

- Variables are declared in the declarative section of the block.

- Variable declaration examples:

```
v_student_id  CHAR(8);
v_lastname    VARCHAR2(25);
v_capacity    NUMBER(3) := 200;
```

(:= is used to initialize variables)

Selected Topics In SQL / PL SQL

محمد إبراهيم الدسوقى

محاضر بكلية هندسة و علوم الحاسب – قسم نظم المعلومات

جامعة سلمان بن عبد العزيز- السعودية – محافظة الخرج

Email : mohamed_eldesouki@hotmail.com

Selected Topics In SQL / PL SQL

- ~~SQL Sub Queries~~
- ~~SQL Views~~
- ~~Introduction to PL / SQL~~
- PL SQL Condition Statements
  1. IF – Then – Else
  2. Case Statement
- PL SQL LOOPs
  1. Loop Statement
  2. For – Loop Statement
  3. While – Loop Statement
- Cursors
- SQL Procedures
- SQL Functions
- SQL Triggers

IF-THEN-ELSE Statement

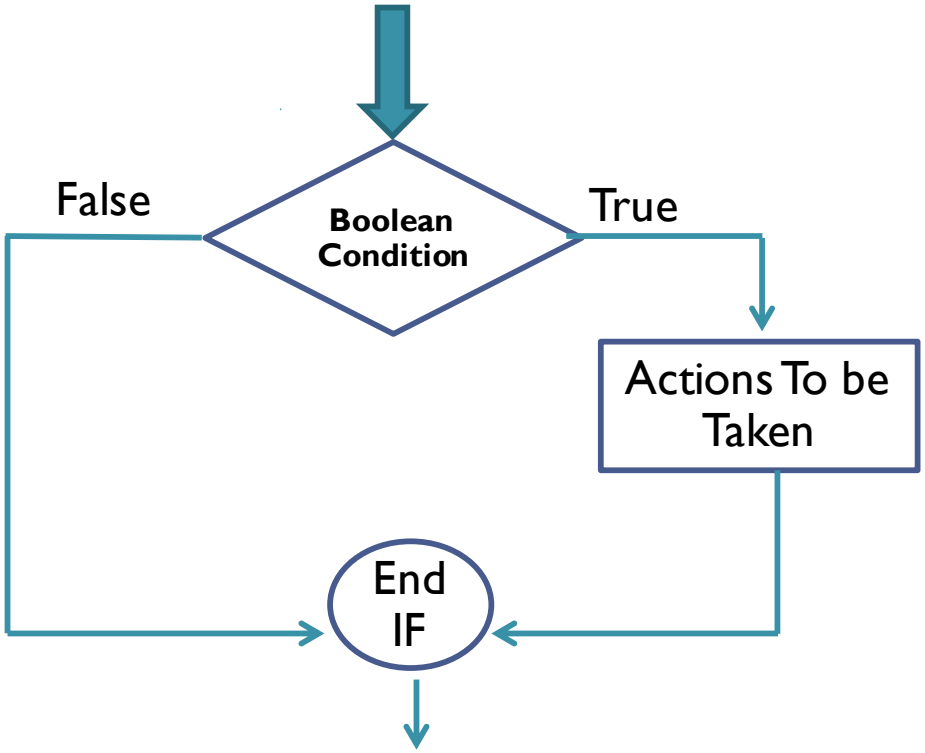
There are three different syntaxes for these types of statements.

Syntax #1: IF-THEN

IF condition THEN

{...statements to be executed...}

END IF;



IF-THEN

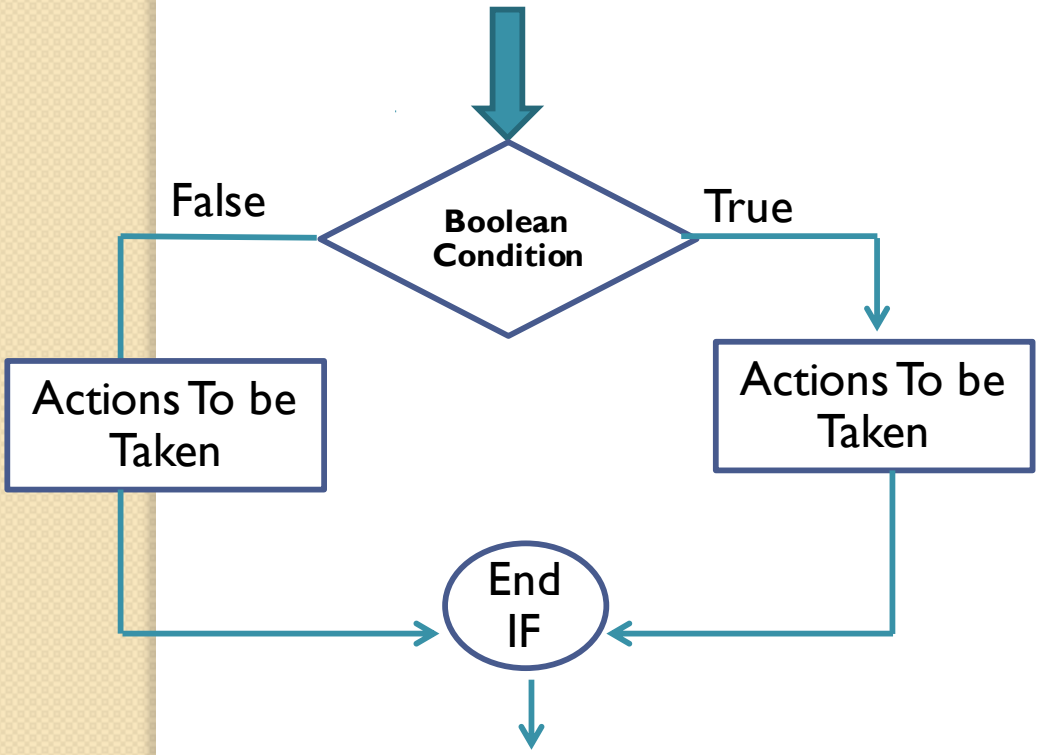
Example:

```
declare
stdmarks    number (2);
passmark number(2);
begin
passmark :=60;
stdmarks :=20;
if stdmarks > passmark then
dbms_output.put_line(Congatulations, go to the next level');
end if;‘
end;
```

IF-THEN-ELSE Statement

Syntax #1: IF-THEN - ELSE

```
IF condition THEN
 {...statements to be executed...}
ELSE
 {...statements to be executed...}
END IF;
```



IF-THEN-ELSE Statement

Example:

```
declare
stdmarks    number (2);
passmark number(2);
begin
passmark :=60;
stdmarks :=20;
if stdmarks > passmark then
dbms_output.put_line(Congatulations, go to the next level');
Else
dbms_output.put_line(Sorry, you have to get extra ‘ ||
passmark – stdmarks || ‘ marks to pass’);
end if;‘
end;
```

IF-THEN-ELSIF Statement

Syntax #1: IF-THEN - ELSIF

```
IF  condition 1       THEN
 {...statements to be executed...}
ELSIF condition 2  THEN
 {...statements to be executed...}
 ELSIF condition 3  THEN
 {...statements to be executed...}
ELSE
 {...statements to be executed...}
END IF;
```

Example: **Print out the student grade according to the following rules**

Marks between 91 – 100 >>> A

Marks between 81 – 90 >>> B

Marks between 71 – 80 >>> C

Marks between 61 – 70 >>> D

Otherwise F

And – Or Conditions

Print out the student grade according to the following rules

Marks between 91 – 100 >>> A

Marks between 81 – 90 >>> B

Marks between 71 – 80 >>> C

Marks between 61 – 70 >>> D

Otherwise F

Selected Topics In SQL / PL SQL

محمد إبراهيم الدسوقى

محاضر بكلية هندسة و علوم الحاسب – قسم نظم المعلومات

جامعة سلمان بن عبد العزيز- السعودية – محافظة الخرج

Email : mohamed_eldesouki@hotmail.com

Selected Topics In SQL / PL SQL

- ~~SQL Sub Queries~~
- ~~SQL Views~~
- ~~Introduction to PL / SQL~~
- ~~PL SQL Condition Statements~~
  1. ~~IF – Then – Else~~
  2. Case Statement
- PL SQL LOOPs
  1. Loop Statement
  2. For – Loop Statement
  3. While – Loop Statement
- Cursors
- SQL Procedures
- SQL Functions
- SQL Triggers

Case Statement

You can use the Case statement within any SQL statement. The case statement has the functionality of an IF-THEN-ELSIF statement.

The syntax for the case statement is:

1st Form : Simple Case Statement

May be
Variable , Expression,
Function

```
CASE [ Selector]
WHEN  condition_1 THEN   Sequence of Statements
WHEN  condition_2 THEN  Sequence of Statements
...
WHEN condition_n THEN  Sequence of Statements
ELSE  Sequence of Statements
END Case ;
```

Case Statement : Simple

Print out the student grade according to the following rules

Marks 95 >>> A
Marks 90 >>> B
Marks 85 >>> C

Otherwise F

```
declare
v_mark number;
begin
v_mark :=90;
case v_mark
when 95 then dbms_output.put_line('You Got A');
when 90 then dbms_output.put_line('You Got B');
when 85 then dbms_output.put_line('You Got C');
else
dbms_output.put_line ('Sorry ,You Got F');
end case;
end;
```

Case Statement : Searched Case

Print out the student grade according to the following rules

Marks between 91 – 100 >>> A
Marks between 81 – 90 >>> B
Marks between 61 – 70 >>> D

Otherwise F

```
declare
v_mark number;
begin
v_mark :=85;
case
when v_mark between 91 and 100 then dbms_output.put_line('You Got A');
when v_mark between 81 and 90  then dbms_output.put_line('You Got B');
when v_mark between 61 and 70  then dbms_output.put_line('You Got D');
else
dbms_output.put_line ('Sorry , You Got F');
end case;
end;
```

Case Statement : Searched Case

```
declare
v_mark number :=85;
v_Level number:= 3;
v_Major char (15) := 'SW Engineering';
begin
case
when v_mark between 91 and 100 then dbms_output.put_line('You Got A');
when v_level = 4 then dbms_output.put_line('You  Are in Second Year');
when v_Major ='Inform Systems'  then dbms_output.put_line('You are an Analyst');
else
dbms_output.put_line ('Non of the conditions is true');
end case;
end;
```

Case Statement : Within Select Statement

Display a list of empno , ename and where he is working according to the following facts

Deptno 10 >>> 'Accounting'
Deptno 20 >>> 'Marketing'
Deptno 10 >>> 'Sales'

```
select empno ,ename, deptno,
case deptno
when 10 then 'You work in Accounting'
when 20 then 'You work in Marketing'
when 30 then 'You work in Sales'
else 'Not Assigned To department'
end
from emp
```

Case Statement

```
Select table_name,
CASE
WHEN owner = 'SYS' THEN 'The owner is SYS'
WHEN owner = 'SCOTT' THEN 'The owner is SCOTT'
ELSE 'The owner is another value'
END
from all_tables
where owner ='SCOTT' or Owner ='SYS' ;
```

```
select supplier_id,
CASE
WHEN supplier_name = 'IBM' and supplier_type = 'Hardware' THEN 'North office'
WHEN supplier_name = 'IBM' and supplier_type = 'Software' THEN 'South office'
END
from suppliers;
```

Selected Topics In SQL / PL SQL

محمد إبراهيم الدسوقى

محاضر بكلية هندسة و علوم الحاسب – قسم نظم المعلومات

جامعة سلمان بن عبد العزيز- السعودية – محافظة الخرج

Email : mohamed_eldesouki@hotmail.com

Selected Topics In SQL / PL SQL

- ~~SQL Sub Queries~~
- ~~SQL Views~~
- ~~Introduction to PL / SQL~~
- ~~PL SQL Condition Statements~~
  1. ~~IF – Then – Else~~
  2. ~~Case Statement~~
- PL SQL LOOPs
  1. Loop Statement
  2. For – Loop Statement
  3. While – Loop Statement
- Cursors
- SQL Procedures
- SQL Functions
- SQL Triggers

Loops: 1- Loop Statement

You would use a LOOP statement when you are not sure how many times you want the loop body to execute and you want the loop body to execute at least once.

The LOOP statement is terminated when it encounters either an EXIT statement or when it encounters an EXITWHEN statement that evaluated to TRUE.

The syntax for the LOOP statement is:

LOOP

{.statements.}

ExitWhen (Condition)

END LOOP;

Loops: 1- Loop Statement

-> Print out the word ‘Hello’ Five times.

```
declare
Counter number :=0;
begin

      loop

     dbms_output.put_line ('Hello');
     counter:=counter +1;

     exit when (counter =5);

     end loop;

end;
/
```

Loops: 1- Loop Statement

-> Print out the word ‘Hello’ Five times.

```
declare
Counter number := 0 ;
begin

     loop

    dbms_output.put_line ('Hello');
    counter:=counter +1;

    If counter =5 then
    Exit ;
    End If;

    end loop;

end;
/
```

Loops: 1- Loop Statement

-> Print out the numbers from 1 to 10.

```
declare
counter number :=1;
begin
      loop

     dbms_output.put_line (counter);
     counter:=counter +1;
     exit when (counter > 10);

     end loop;
end;
/
```

Selected Topics In SQL / PL SQL

محمد إبراهيم الدسوقى

محاضر بكلية هندسة و علوم الحاسب – قسم نظم المعلومات

جامعة سلمان بن عبد العزيز- السعودية – محافظة الخرج

Email : mohamed_eldesouki@hotmail.com

Selected Topics In SQL / PL SQL

- ~~SQL Sub Queries~~
- ~~SQL Views~~
- ~~Introduction to PL / SQL~~
- ~~PL SQL Condition Statements~~
  1. ~~IF – Then – Else~~
  2. ~~Case Statement~~
- PL SQL LOOPs
  1. ~~Loop Statement~~
  2. While – Loop Statement
  3. For – Loop Statement
- Cursors
- SQL Procedures
- SQL Functions
- SQL Triggers

Loops: 2- While Loop Statement

You would use a WHILE Loop when you are not sure how many times you will execute the loop body.

Since the WHILE condition is evaluated before entering the loop, it is possible that the loop body may ***not*** *execute even once.*

The syntax for the While Loop is:

WHILE (Boolean condition)

LOOP

{.statements.}

END LOOP;

Loops: 2- While Loop Statement

-> Print out the word 'Hello' Five times.

```
declare
Counter number :=1;
begin
While (counter <=5 )
      loop

     dbms_output.put_line ('Hello');
     counter:=counter +1;

     end loop;

end;
/
```

Loops: 2- While Loop Statement

-> Print out the numbers from 1 to 10.

Example

```
declare
counter number :=1 ;
begin

while (counter <= 10)
LOOP

dbms_output.Put_Line ('counter value reaches ' || counter);
counter := counter + 1;

END LOOP;

end;
```

Selected Topics In SQL / PL SQL

محمد إبراهيم الدسوقى

محاضر بكلية هندسة و علوم الحاسب – قسم نظم المعلومات

جامعة سلمان بن عبد العزيز- السعودية – محافظة الخرج

Email : mohamed_eldesouki@hotmail.com

Selected Topics In SQL / PL SQL

- ~~SQL Sub Queries~~
- ~~SQL Views~~
- ~~Introduction to PL / SQL~~
- ~~PL SQL Condition Statements~~
  1. ~~IF – Then – Else~~
  2. ~~Case Statement~~
- PL SQL LOOPs
  1. ~~Loop Statement~~
  2. ~~While – Loop Statement~~
  3. For – Loop Statement
- Cursors
- SQL Procedures
- SQL Functions
- SQL Triggers

Loops: 3- For Loop Statement

You would use a FOR Loop when you want to execute the loop body a fixed number of times.

The syntax for the FOR Loop is:

```
FOR loop_counter IN lowest_number..highest_number
LOOP
{.statements.}
END LOOP;
```

loop_counter : is declared implicitly within the For Loop

Number of Repetitions : is calculated before executing the For Loop

Loops: 3- For Loop Statement

-> Print out the word 'Hello' Five times.

```
begin
For counter in 1..5
      loop

      dbms_output.put_line ('Hello');

      end loop;

end;
/
```

Loops: 3- For Loop Statement

Example

-> Print out the numbers from 1 to 10.

```
begin
FOR counter IN 1..10

LOOP

dbms_output.Put_Line ('counter value reaches ' || counter);

END LOOP;

end;
/
```

Loops: 2- For Loop Statement with Reverse

FOR loop_counter IN Reverse lowest_number..highest_number

LOOP

{.statements.}

END LOOP;

Example

```
begin
FOR counter IN Reverse 1..10
LOOP
dbms_output.Put_Line ('counter value reaches ' || counter);
END LOOP;
end;
```

Selected Topics In SQL / PL SQL

محمد إبراهيم الدسوقى

محاضر بكلية هندسة و علوم الحاسب – قسم نظم المعلومات

جامعة سلمان بن عبد العزيز- السعودية – محافظة الخرج

Email : mohamed_eldesouki@hotmail.com

Selected Topics In SQL / PL SQL

- ~~SQL Sub Queries~~
- ~~SQL Views~~
- ~~Introduction to PL / SQL~~
- ~~PL SQL Condition Statements~~
  - ~~I. IF – Then – Else~~
  - ~~II. Case Statement~~
- PL SQL LOOPs
  - ~~I. Loop Statement~~
  - ~~II. While – Loop Statement~~
  - ~~III. For – Loop Statement~~
- Cursors
- SQL Procedures
- SQL Functions
- SQL Triggers

What Is Cursor ?

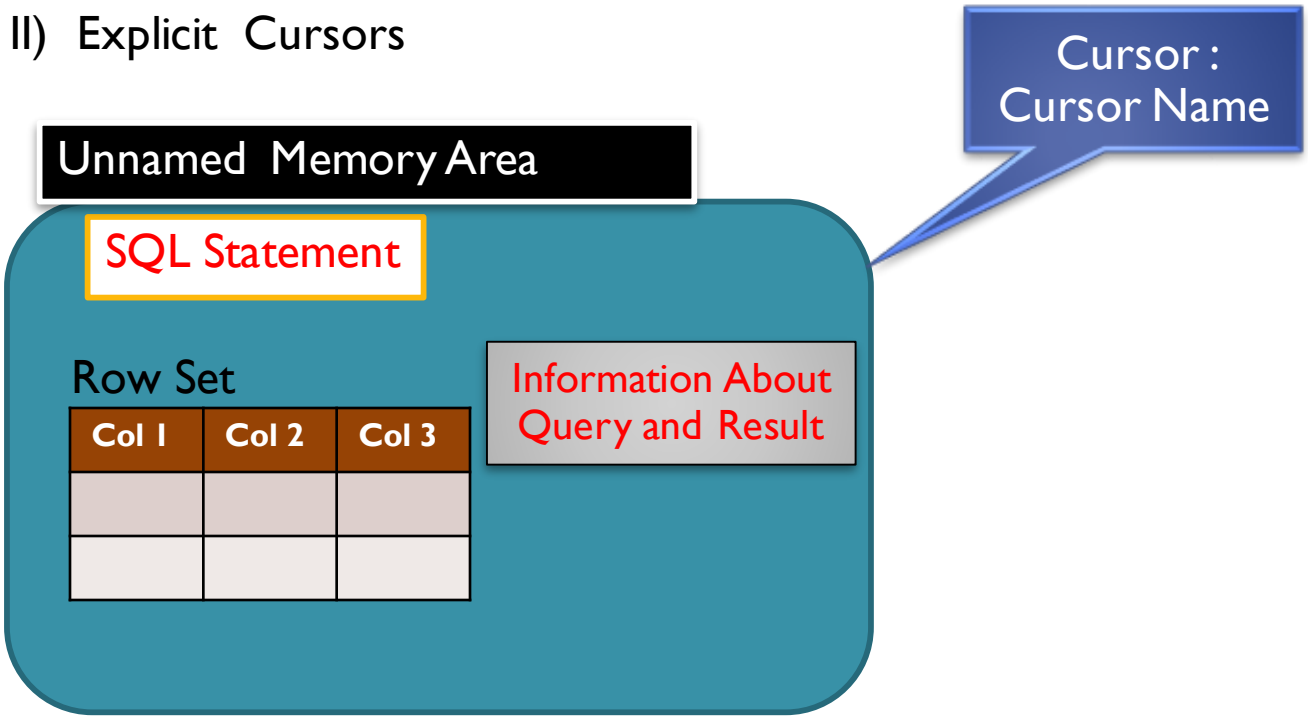
The Cursor is a handle (name or a pointer) for the memory associated with a specific SQL statement.

A cursor is basically an Area allocated by Oracle for executing the SQL Statements.

Oracle Has two basic Types of Cursor :

I) Implicit Cursors

II) Explicit Cursors



Cursors: Implicit Cursors

-An implicit cursor is a cursor which is internally created by Oracle.

-It is associated with any DML statement (Insert , Update , Delete) and with any single row query (query that that returns a single row).

-It is called the SQL cursor.

-It helps us to answer the following Questions :

-Was any rows affected by that DML Query Or Not ?

-How Many rows are affected by that Query?

Example

```
begin
update emp
set comm= sal * .025
where deptno = 10;
end;
```

Query Executed or Not?
How Many Rows Affected?

Cursors: Implicit Cursors

Attributes of implicit (SQL) cursor

Name	Description
%FOUND	Returns TRUE if the most recent DML statement affected one or more rows.
%NOTFOUND	Returns TRUE if there is no rows affected by the recent DML statement .
%ROWCOUNT	Returns number of Rows affected by the most recent DML statement.

Example

```
begin
update emp set  comm = sal * .025 where deptno = 10;

if (sql%found) then
dbms_output.put_line
     ('the query affected ' || to_char(sql%rowcount));
end if;
end;
```

SQL/PL SQL : Explicit Cursors

-An Explicit cursor is a cursor which is declared by the programmer to handle SQL queries (Select Statements) that return more than one row .

- Creation and Manipulation of Explicit cursors Requires a number of manual steps

Steps For Using Cursors

1- Declare a Cursor

2- Open Statement

3- FETCH Statement

4- CLOSE Statement

Explicit Cursors : Declare Statement

1- Declare Statement :

Assigns the user defined cursor to a select statement that returns more than one row

The basic syntax for Declaring a cursor is:

```
CURSOR cursor_name
IS
SELECT_statement;
```

For example,

```
Declare
CURSOR Curs_Emp_Data
 IS
SELECT Empno , Ename , Sal , Deptno
from Emp where  Deptno = 20 ;
```

Explicit Cursor: Open Statement

Once you've declared your cursor, the next step is to open the cursor.

2- Open Statement :

Executes the select statement to create an active set of rows

The basic syntax to OPEN the cursor is:

```
OPEN cursor_name;
```

For example

```
Declare
CURSOR Curs_Emp_Data
 IS
SELECT Empno , Ename , Sal , Deptno
from Emp where  Deptno = 20 ;
Begin

OPEN Curs_Emp_Data;
```

Explicit Cursors: Fetch Statement

3- Fetch Statement :

Used to retrieve a record from the Select Statement Result Set. And populate column values to memory variables. It retrieves one record each time.

The basic syntax for a FETCH statement is:

FETCH cursor_name INTO <list of variables>;

For example

```
Declare
VarEno number; varEname Varchar2(20) ;
 CURSOR Curs_Emp_Data
 IS
SELECT Empno , Ename
from Emp where  Deptno = 20 ;
Begin

OPEN Curs_Emp_Data;
FETCH Curs_Emp_Data  into VarEno , varEname ;
```

Explicit Cursors: Close Statement

4- Close Statement :

Used to release the cursor.

The basic syntax for a Close statement is:

```
Close cursor_name ;
```

For example

```
Declare
VarEno number;  varEnameVarchar2(20) ;
 CURSOR Curs_Emp_Data
 IS
SELECT Empno , Ename
from Emp where  Deptno = 20 ;
Begin

OPEN Curs_Emp_Data;
FETCH Curs_Emp_Data  into VarEno , varEname ;
Close Curs_Emp_Data;
```

%Type and %RowType Attributes

The %TYPE attribute lets you use the datatype of a field instead of hard coding the type names when declaring the variables.

Example

```
Declare
VarEno number (3) ;
varEname Varchar2(10) ;
VarSalary number (5);

CURSOR Curs_Emp_Data
 IS
SELECT Empno , Ename , Sal
from Emp where Deptno = 20 ;
Begin

OPEN Curs_Emp_Data;
FETCH Curs_Emp_Data into VarEno , VarEname, VarSalary ;
Close Curs_Emp_Data;
```

Emp	
Field	**Data Type**
Empno	Number (5)
Ename	Varchar2 (20)
Sal	Number (7,2)

%Type and %RowType Attributes

The syntax of %TYPE attribute

Varname TableName.ColumnName%Type.

Example

```
Declare
VarEno  Emp.Empno%Type ;
varEname Emp.Ename%Type ;
VarSalary Emp.Sal%Type;

CURSOR Curs_Emp_Data
 IS
SELECT Empno  , Ename ,  Sal
from Emp where  Deptno = 20 ;
Begin

OPEN Curs_Emp_Data;
FETCH Curs_Emp_Data  into VarEno , VarEname, VarSalary ;
Close Curs_Emp_Data;
```

Emp	
Field	**Data Type**
Empno	Number (5)
Ename	Varchar2 (20)
Sal	Number (7,2)

%Type and %RowType Attributes

The %RowTYPE attribute provides a record type that represents a row in a database table. The record can store an entire row of data selected from the table or fetched from a cursor.

Example

```
Declare
CURSOR Curs_Emp_Data
 IS
SELECT Empno  , Ename ,  Sal
from Emp where  Deptno = 20 ;

VarEmpData   Emp_Data%RowType;

Begin

OPEN Curs_Emp_Data;
FETCH Curs_Emp_Data  into VarEmpData ;
 dbms_output.put_line (VarEmpData . Ename);
Close Curs_Emp_Data;
```

Emp	
Field	**Data Type**
Empno	Number (5)
Ename	Varchar2 (20)
Sal	Number (7,2)

VarEmpData

Empno	Ename	Sal

Explicit Cursors Attributes

Name	Description
%ISOPEN	Returns True if the Cursor is open , False Otherwise.
%FOUND	Returns TRUE if the Fetch statement finds a row in the cursor.
%NOTFOUND	Returns TRUE if the Fetch statement doesn't find rows in the cursor.
%ROWCOUNT	Identify the row number of the currently fetched row.

Explicit Cursors: Working with Cursor Row Set using Loops.

```
Declare
CURSOR Curs_Emp_Data
 IS
SELECT Empno  , Ename ,  Sal  from Emp
where  Deptno in ( 20 ,30) ;
VarEmpData   Curs_Emp_Data%RowType;
Begin

OPEN Curs_Emp_Data;

Loop
      FETCH Curs_Emp_Data into VarEmpData ;

      Exit When (Curs_Emp_Data%notfound)

      dbms_output.put_line (VarEmpData . Ename ||
                            VarEmpData . Sal);
End Loop;

Close Curs_Emp_Data;
```

Explicit Cursors: Working with Cursor Row Set using Loops.

```
Declare
CURSOR Curs_Emp_Data
 IS
SELECT Empno  , Ename ,  Sal , Comm from Emp
where  Deptno = 20 ;
VarEmpData   Curs_Emp_Data%RowType;
Begin

OPEN Curs_Emp_Data;

Loop
      FETCH Curs_Emp_Data into VarEmpData ;
      Exit when (Curs_Emp_Data%notfound)
      Update Emp
      Set Comm = VarEmpData .Sal * 0.025 ;
End Loop;

Close Curs_Emp_Data;
```

Stored Program Units

Advantages of Using Stored Program Units:

1. It helps to break a program into manageable, well-defined modules.
2. The code is stored in a pre-compiled form which means that its syntax is valid and does not need to be compiled at run-time, thereby saving resources.
3. promote re-usability and easy maintenance.
4. As the Stored Program Units are stored in the database there is no need to transfer the code from the clients to the database server and this results in much less network traffic and improves scalability;
5. Helps to apply security mechanisms on the Database and control Access.

Types of Stored Program Units

1. **Procedures**
2. **Functions**
3. **Packages**

Stored Program Units

Problem

Write PL/SQL code to update the commission value for all the employees working in a specific department according to the following formulas

1- if the Employee job is 'Manager' , let the commission to be 10 % of the salary value.

2-if the Employee job is 'SALESMAN' , let the commission to be 5 % of the salary value.

3- otherwise , let it 2 % of the salary value.

Stored Procedure

-A **procedure** is a subprogram that performs a specific action.

-A **procedure** is a group of PL/SQL statements that you can call by name.

-Stored Procedure is actually stored in the database.

The syntax for a procedure is:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[ (parameter1 ,parameter2 , parametr n, …..) ]
IS
[declaration_section]
BEGIN
executable_section
[EXCEPTION
exception_section
]
END [procedure_name];
```

Stored Procedure

The syntax for a procedure is:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[ (parameter1 ,parameter2 , parametr n, …..) ]
IS
[declaration_section]
BEGIN
executable_section
[EXCEPTION
exception_section
]
END [procedure_name];
```

Stored Procedure

```
create or replace procedure update_depart_comm
is
cursor Curs_get_Emp is select empno , sal , job from emp where
deptno=10;
var_emp_data Curs_get_Emp%rowtype;
comission number;
begin
open Curs_get_Emp;
loop
fetch Curs_get_Emp into var_emp_data;
exit when (Curs_get_Emp%notfound);
if var_emp_data .job = 'Manager' then
comission := var_emp_data.sal * 0.10;
elsif var_emp_data .job = 'SALESMAN' then
comission := var_emp_data .sal * 0.05;
else
comission := var_emp_data .sal *0.02;
end if;
update emp
set comm = comission
where empno = var_emp_data .empno;
end loop;
close Curs_get_Emp;
end;
```

Stored Procedure With Parameters

There are three types of parameters that can be declared:

IN - The parameter is used to pass a value from the calling program to the procedure. The value of the parameter can not be overwritten by the procedure.

OUT - The parameter is used to pass a value from the procedure to the calling program.

IN OUT - The parameter can be used to pass a value to or from the procedure.

Stored Procedure With Parameters

```
create or replace procedure update_depart_comm ( depart_no in number )
is
cursor Curs_get_Emp is select empno , sal , job from emp where deptno= depart_no;
var_emp_data Curs_get_Emp%rowtype;
comission number;
begin
open Curs_get_Emp;
loop
fetch Curs_get_Emp into var_emp_data;
exit when (Curs_get_Emp%notfound);
if var_emp_data .job = 'Manager' then
comission := var_emp_data.sal * 0.10;
elsif var_emp_data .job = 'SALESMAN' then
comission := var_emp_data .sal * 0.05;
else
comission := var_emp_data .sal *0.02;
end if;
update emp
set comm = comission
where empno = var_emp_data .empno;
end loop;
close Curs_get_Emp;
end;
```

Stored Procedure With Parameters

```
create or replace  procedure update_depart_comm (depart_no in number, comm_sum out
number)
is
cursor Curs_get_Emp is select empno , sal , job from emp where deptno=depart_no ;
var_emp_data Curs_get_Emp%rowtype;
comission number;
begin
open Curs_get_Emp;
loop
fetch Curs_get_Emp into var_emp_data;
exit when (Curs_get_Emp%notfound);
if var_emp_data .job = 'Manager' then
comission := var_emp_data.sal * 0.10;
elsif var_emp_data .job = 'SALESMAN' then
comission := var_emp_data .sal * 0.05;
else
comission := var_emp_data .sal *0.02;
end if;
update emp
set comm = comission
where empno = var_emp_data .empno;
end loop;
close Curs_get_Emp;
select sum(comm) into comm_sum
from emp where deptno = depart_no;
end;
/
```

Stored Procedure With Parameters

```
create or replace procedure update_depart_comm (Param1 in out number)
is
cursor Curs_get_Emp is select empno , sal , job from emp where deptno= Param1 ;
var_emp_data Curs_get_Emp%rowtype;
comission number;
begin
open Curs_get_Emp;
loop
fetch Curs_get_Emp into var_emp_data;
exit when (Curs_get_Emp%notfound);
if var_emp_data .job = 'Manager' then
comission := var_emp_data.sal * 0.10;
elsif var_emp_data .job = 'SALESMAN' then
comission := var_emp_data .sal * 0.05;
else
comission := var_emp_data .sal *0.02;
end if;
update emp
set comm = comission
where empno = var_emp_data .empno;
end loop;
close Curs_get_Emp;

select sum(comm) into Param1
from emp where deptno = Param1 ;
end;
/
```

Stored Procedures

```
DESCRIBE  USER_SOURCE;


SELECT text
 FROM USER_SOURCE
 WHERE name = ' update_depart_comm'
 ORDER BY line;
```

Stored Program Units

Advantages of Using Stored Program Units:

1. It helps to break a program into manageable, well-defined modules.
2. The code is stored in a pre-compiled form which means that its syntax is valid and does not need to be compiled at run-time, thereby saving resources.
3. promote re-usability and easy maintenance.
4. As the Stored Program Units are stored in the database there is no need to transfer the code from the clients to the database server and this results in much less network traffic and improves scalability;
5. Helps to apply security mechanisms on the Database and control Access.

Types of Stored Program Units

1. ~~**Procedures**~~
2. **Functions**
3. **Packages**

Functions

1. A stored function (also called a user function or user defined function) is a set of PL/SQL statements you can call by name.

2. Stored functions are very similar to procedures, except that a function returns a value to the environment in which it is called

The syntax for a function is:

```
CREATE [OR REPLACE] FUNCTION function_name
[ (parameter [,parameter]) ]   RETURN return_datatype

IS | AS
[declaration_section]

BEGIN
executable_section

[EXCEPTION
exception_section]

END [function_name];
```

Functions With Parameters

Problem

Create Function that accepts the hire date the returns The Employee Experience in Years.

```
create or replace function Service_Years ( hdate in date) return number
as
years number;

begin

years:= trunc ( months_between (sysdate,hdate)/12 ,0);
return years ;

end;
/
```

Functions With Parameters

Problem

Create Function that returns the Annual salary for any given Employee Number

```
create or replace function Annual_Sal ( eno in number) return number
as

VarTotal  number;

begin

select sal * 12 into VarTotal from emp where empno = eno ;

return  VarTotal ;

end;
/
```

Functions With Parameters

Problem

Create Function that accepts the hire date and returns The Employee Experience in Years and in weeks.

```
create or replace function Service_Years ( hdate in date, weaks out number)
return number
As
years number;
Begin
years:= trunc ( months_between (sysdate,hdate)/12 ,0);
weaks := trunc ( months_between (sysdate, hdate)*4 ,0);
return years ;
end;
/
```

```
declare
noofweaks number; returnvalue number;
begin
returnvalue:= service_years (to_date('03/08/1981','dd/mm/yyyy')
,noofweaks) ;
dbms_output.put_line (returnvalue);
dbms_output.put_line (noofweaks );
end;
/
```

Restrictions with Stored Functions

1. *when a function is called from within a query or DML statement, the function cannot Have OUT or IN OUT parameters.*

2. *When used in SELECT statement they cannot contain DML –*

3. *When used in UPDATE or DELETE they cannot SELECT or perform DML on the same table*

Packages

•A Package is an encapsulated collection of related procedures, functions, and other program objects stored together in the database.

•A package is a schema object that groups logically related PL/SQL types, items and subprograms.

Packages usually have two parts :

1- **specification :** Is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package

2-**body (Unnecessary):** defines the queries for the cursors and the code for the subprograms.

Packages

Reasons to use packages

1- Modularity: Packages let you encapsulate logically related types, variables, constants, subprograms, cursors, and exceptions in named PL/SQL modules. You can make each package easy to understand, and make the interfaces between packages simple, clear, and well defined. This practice aids application development.

2- Easier Application Design : When designing an application, all you need initially is the interface information in the package specifications. You can code and compile specifications without their bodies. **Next**, you can compile standalone subprograms that reference the packages.

You don't need to fully define the package bodies until you are ready to complete the application.

Packages

Reasons to use packages

3- Better Performance: The first time you invoke a package subprogram, Oracle Database loads the whole package into memory. Subsequent invocations of other subprograms in same the package require no disk I/O.

- Packages prevent cascading dependencies and unnecessary recompiling . if you change the body of a package function, Oracle Database does not recompile other subprograms that invoke the function, because these subprograms depend only on the parameters and return value that are declared in the specification.

Packages

Syntax for creating Package Specs

CREATE [OR REPLACE] PACKAGE package_name
{ IS | AS }

End; [package_name]

Syntax for creating Package Body

CREATE [OR REPLACE] PACKAGE Body package_name
{ IS | AS }

End; [package_name]

Packages

Package Specifications

```
create or replace package hr
is
TYPE  EmpRecTyp IS RECORD (emp_id NUMBER, sal NUMBER);
Function GetEmpCount (dno in number) return number;
Function GetEmpName (empno in number) return varchar2;
procedure UpdateComm (eno in number ,percent in number);
procedure UpdateComm (dno in number);
end;
/
```

Package Body

```
create or replace package body hr
as
function GetEmpCount (dno in number) return number
as
empcount number;
begin
select count(*) into empcount  from emp where deptno=dno;
return empcount;
end;

function GetEmpName(empno in number) return varchar2
as
empname varchar2 (20);
begin
select ename into empname from emp where emp.empno= empno;
return empname ;
end;

procedure UpdateComm (eno in number ,percent in number)
as
begin
update emp
set comm = sal * percent;
end;
```

Referencing Package Contents

To reference the types, items, subprograms, and call specs declared within a package spec, **use dot notation**:

package_name.type_name
package_name.item_name
package_name.subprogram_name

```
Declare
emprec hr.EmpRecTyp;
Empcount  number;
Begin
select empno , sal into emprec
from emp where empno=7499;
dbms_output.put_line(emprec.emp_id);
dbms_output.put_line(emprec.sal);

Empcount := hr.GetEmpCount(10);
dbms_output.put_line(Empcount );
end;
/
```

Selected Topics In SQL / PL SQL

محمد إبراهيم الدسوقى

محاضر بكلية هندسة و علوم الحاسب – قسم نظم المعلومات

جامعة سلمان بن عبد العزيز- السعودية – محافظة الخرج

Email : mohamed_eldesouki@hotmail.com

Selected Topics In SQL / PL SQL

- ~~SQL Sub Queries~~
- ~~SQL Views~~
- ~~Introduction to PL / SQL~~
- ~~PL SQL Condition Statements~~
  - ~~I. IF – Then – Else~~
  - ~~II. Case Statement~~
- PL SQL LOOPs
  - ~~I. Loop Statement~~
  - ~~II. While – Loop Statement~~
  - ~~III. For – Loop Statement~~
- Cursors
- ~~SQL Procedures - Functions~~
- ~~SQL Packages~~
- SQL Triggers

Database Triggers

A **database trigger** is procedural code that is automatically executed in response to certain events(an INSERT, UPDATE, or DELETE) on a particular table or view in a database.

Triggers are similar to stored procedures. However, procedures and triggers differ in the way that they are invoked.

- A procedure is explicitly executed by a user, application, or trigger.

- Triggers are implicitly fired (executed) by Oracle when a triggering INSERT, UPDATE , or DELETE statement is issued.

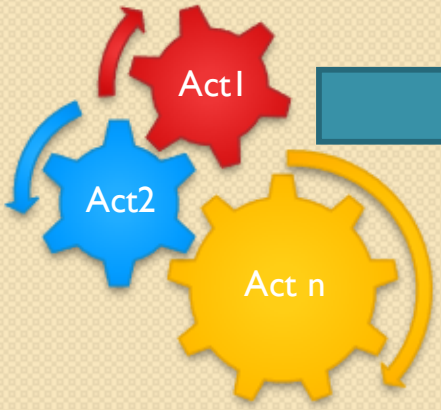
- Triggers do not accept parameters or arguments.

Orders

Ord #	Date	Item	Qty
135	12/1	1	5

Event (Insert –Update- Delete)

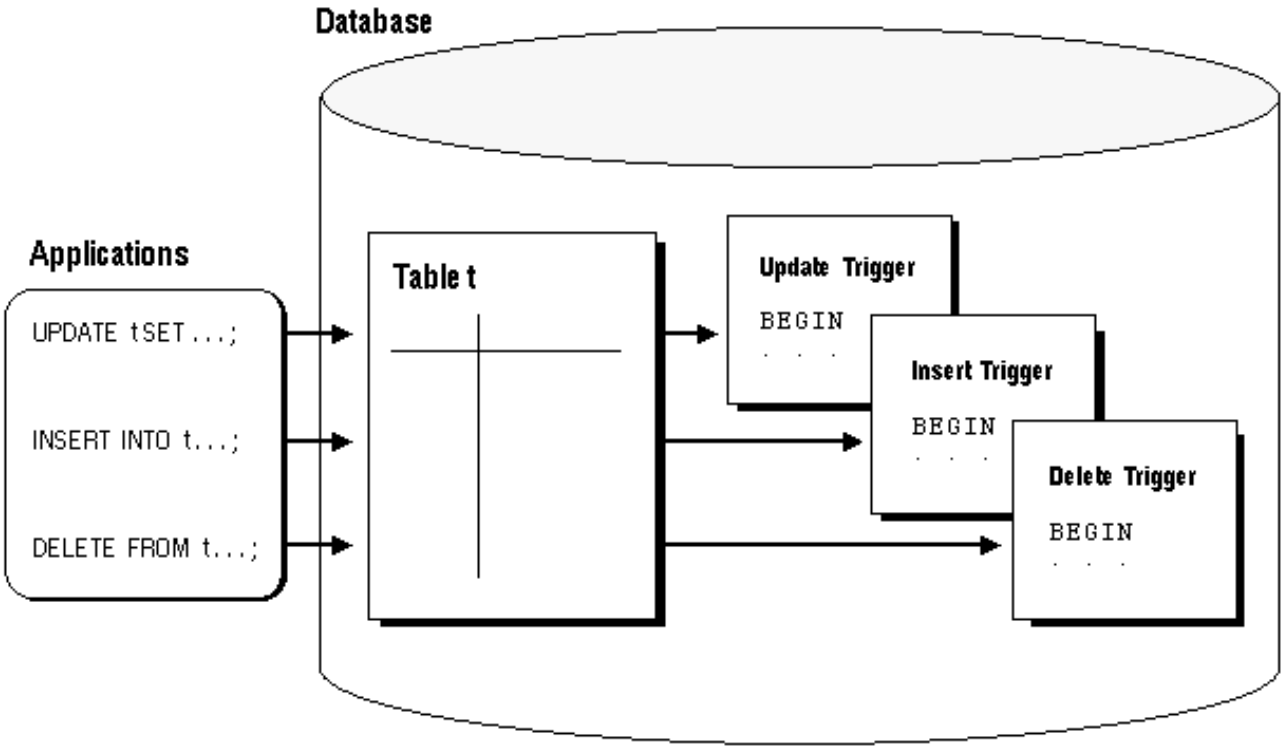
Delete from orders
Where ordno=135;



Inventory

Item #	Qty
1	25
2	30
3	15

Database Triggers



Notice 1 That triggers are stored in the database separately from their associated tables.

Notice 2 Triggers can be defined only on tables, not on views. However, triggers on the base table(s) of a view are fired if an INSERT, UPDATE, or DELETE statement is issued against a view.

How Triggers Are Used ?

Triggers are commonly used to :

-Automatically generate derived column values.

- Prevent invalid transactions.

-Enforce complex security authorizations.

-Enforce complex business rules

-Provide sophisticated auditing

-Gather statistics on table access

Triggers vs. Declarative Integrity Constraints

Triggers and declarative integrity constraints can both be used to constrain data input.

However, triggers and integrity constraints have significant differences.

- A trigger does not apply to data loaded before the definition of the trigger. Therefore, it does not guarantee all data in a table conforms to its rules.

- A constraint applies to existing data in the table and any statement that manipulates the table.

Parts of a Trigger

A trigger has three basic parts:

1- A Triggering Event Or Statement : The SQL statement that causes a trigger to be fired. A triggering event can be an INSERT, UPDATE, or DELETE statement on a table.

A triggering event can specify multiple DML statements, as INSERT OR UPDATE OR DELETE

Note that when the triggering event is an UPDATE statement, you can include a column list to identify which columns must be updated to fire the trigger.

Parts of a Trigger

2- A Trigger Restriction (Optional) : specifies a Boolean (logical) expression that must be TRUE for the trigger to fire.

The trigger action is not executed if the trigger restriction evaluates to FALSE or UNKNOWN

3- A Trigger Action is the procedure (PL/SQL block) that contains the SQL statements and PL/SQL code to be executed when a triggering statement is issued and the trigger restriction evaluates to TRUE.

Types of Triggers

1- Row Level Triggers

A row trigger is fired each time the table is affected by the triggering statement.

For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement.

If a triggering statement affects no rows, a row trigger is not executed at all.

Row triggers are useful if the code in the trigger action depends on data provided by the triggering statement or rows that are affected

Types of Triggers

2- Statement Triggers

A statement trigger is fired once on behalf of the triggering statement, regardless of the number of the number of rows affected.(even if no rows are affected).

For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once, regardless of how many rows are deleted from the table.

Are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected.

For example, if a trigger makes a complex security check on the current time or user, or if a trigger generates a single audit record based on the type of triggering statement, a statement trigger is used.

Trigger Timing

When defining a trigger, you can specify the *trigger timing*. That is,

you can specify whether the **trigger action** is to be executed ***before*** or ***after*** the triggering statement.

BEFORE and AFTER apply to both statement and row triggers.

BEFORE Triggers : Execute the trigger action before the triggering statement.

AFTER Triggers: Execute the trigger action after the triggering statement is executed

Tips in Designing Triggers

1- Use triggers to guarantee that when a specific operation is performed, related actions are performed.

2- Use database triggers only for centralized, global operations that should be fired for the triggering statement, regardless of which user or database application issues the statement.

3- Do not define triggers that duplicate the functionality already built into Oracle. For example, do not define triggers to enforce data integrity rules that can be easily enforced using declarative integrity constraints.

4- Limit the size of triggers (60 lines or fewer is a good guideline). If the logic for your trigger requires much more than 60 lines of PL/SQL code, it is better to include most of the code in a stored procedure, and call the procedure from the trigger.

Creating Triggers

CREATE [OR REPLACE] **TRIGGER** trigger_name

{BEFORE | AFTER } trigger_event

ON table_name

[FOR EACH ROW [WHEN trigger_condition]]

BEGIN

trigger_body

END trigger_name;

Insert , Update , Delete

Row Level Trigger

•**Notice :**

FOR EACH ROW

Specifies the trigger is a row-level trigger.
-If you omit FOR EACH ROW, the trigger is a statement-level trigger .

Accessing Column Values in Row Triggers

Within a trigger body of a row trigger, the PL/SQL code and SQL statements have access to the old and new column values of the current row affected by the triggering statement.

- The new column values are referenced using the **:NEW** qualifier before the column name,

- while the old column values are referenced using the **:OLD** qualifier before the column name.

For Example :

```
create or replace Trigger emp_name_changes
BEFORE INSERT OR UPDATE ON emp
for each row
begin
:new.ename :=Upper(:new.ename);
end;
```

Accessing Column Values in Row Triggers

Orders

Ord #	Date	Item	Qty
135	12/1	1	5

:NEW

135	12/1	1	5

```
Insert into orders
Values  (135,'12/1',1,5);
```

(1)

:NEW.Qty = 12

135	12/1	1	12

:OLD.Qty = 5

```
Update  orders
Set qty = 12 where
order _id =135;
```

(2)

:OLD

```
Delete from orders
Where  ordno=135;
```

(3)

Trigger Examples (**Row Level Trigger**)

Suppose we have the following table :

```
CREATE TABLE orders
( order_id number(5), Item_no number(4), quantity number(4),
  cost_per_unit number(6,2), total_cost number(8,2),
  create_date date
);
```

We want to create trigger that automatically set the total cost , create_date columns.

```
CREATE OR REPLACE TRIGGER orders_before_insert BEFORE INSERT ON orders FOR EACH ROW

BEGIN

  :new.total_cost := :new.quantity * :new.cost_per_unit;
 :new.create_date := sysdate;

END;
```

Trigger Examples (**Row Level Trigger**)

Suppose we have the following table :

```
CREATE TABLE Inventory
( Item_no  number(4)  primary key,  available_qty  number(4) );
```

We want to create trigger that automatically subtract the ordered quantity from the available Quantity.

```
CREATE OR REPLACE TRIGGER orders_After_insert After INSERT
ON orders FOR EACH ROW
BEGIN
Update inventory
Set available_qty = available_qty - :new.quantity
where item_no = :new.item_no;
END;
```

Trigger Examples (**Row Level Trigger**)

```
CREATE OR REPLACE TRIGGER orders_After_insert  After INSERT or Update
 ON orders

FOR EACH ROW

BEGIN

 Update inventory
Set available_qty = available_qty  -  :new.quantity  where  item_no = :new.item_no;

END;
```

Trigger Examples (**Row Level Trigger**)

Write a trigger that prevent the cancelation of orders that is older than 2 days .
And in case of cancelation increase the available quantity in the inventory table with the canceled quantity

```
CREATE OR REPLACE TRIGGER orders_before_delete
 BEFORE Delete ON orders   FOR EACH ROW
DECLARE
   v_daysaftersale  number;
BEGIN
select  floor (sysdate - :old.create_date)  into v_daysaftersale  from dual;
if  ( v_daysaftersale  > 2) then
 raise_application_error  (-20007,'Orders  can be cancelled  within  2 days only');
else
Update inventory
Set available_quantity  = available_quantity  + :old.quantity  where  item_no  =
:old.item_no;
end if;
END;
/
```

Error Handling in Triggers

The Oracle engine provides a procedure named **raise_application_error** that allows programmers to issue user-defined error messages.

Syntax:

RAISE_APPLICATION_ERROR (error_number, message);

Here:

error_number : It is a negative integer in the range-20000 to –20999

Message : It is a character string up to 2048 bytes in length .

This procedure terminates procedure execution,
rolls back any effects of the procedure,
and returns a user-specified error

Trigger Examples (**Statement Level Trigger**)

To Create a trigger so that no operation can be performed on emp table on Sunday

```
CREATE OR REPLACE TRIGGER Orders_SUNDAY
BEFORE INSERT OR UPDATE OR DELETE ON orders
BEGIN
IF RTRIM(UPPER(TO_CHAR(SYSDATE,'DAY')))='SUNDAY' THEN
RAISE_APPLICATION_ERROR(-20022,'NO OPERARTION CAN
BE
PERFORMED ON SUNDAY');
END IF;
End;
```

Trigger Examples (**Statement Level Trigger**)

```
CREATE OR REPLACE TRIGGER Orders_AUDIT AFTER INSERT OR UPDATE OR DELETE
ON Orders

declare
v_user varchar2(20);

BEGIN
select user into v_user from dual;
IF INSERTING THEN
INSERT INTO AUDITOR VALUES(v_user,'INSERT');

ELSIF UPDATING THEN
INSERT INTO AUDITOR VALUES(v_user,'UPDATE');

ELSIF DELETING THEN
INSERT INTO AUDITOR VALUES(v_user,'DELETE');

END IF;

end ;
```

Enabling and Disabling - Dropping Triggers

We can disable / enable the trigger by the following syntax

ALTER TRIGGER <trigger name> DISABLE / ENAMBLE

We can Drop the trigger by the following syntax

DROP TRIGGER trigger_name;

At a weekend retreat, the entity type PERSON has three subtypes: CAMPER, BIKER, and RUNNER. Draw a separate EER diagram segment for each of the following situations:

a. At a given time, a person must be exactly one of these subtypes.

b. A person may or may not be one of these subtypes. However, a person who is one of these subtypes cannot at the same time be one of the other subtypes.

c. A person may or may not be one of these subtypes. On the other hand, a person may be any two (or even three) of these subtypes at the same time.

d. At a given time, a person must be at least one of these subtypes.

A bank has three types of accounts: checking, savings, and loan. Following are the attributes for each type of account:

CHECKING: Acct_No, Date_Opened, Balance, _Service_ Charge

SAVINGS: Acct_No, Date_Opened, Balance, _Interest_Rate

LOAN: Acct_No, Date_Opened, Balance, Interest_Rate, Payment

Assume that each bank account must be a member of exactly one of these subtypes. Using generalization, develop an EER model segment to represent this situation using the traditional EER notation.