



# **Base de données II**

## **Concepts et programmation procédurale**

# **Mise en œuvre avec**

## **Oracle SQL**

# Agenda

**Introduction**

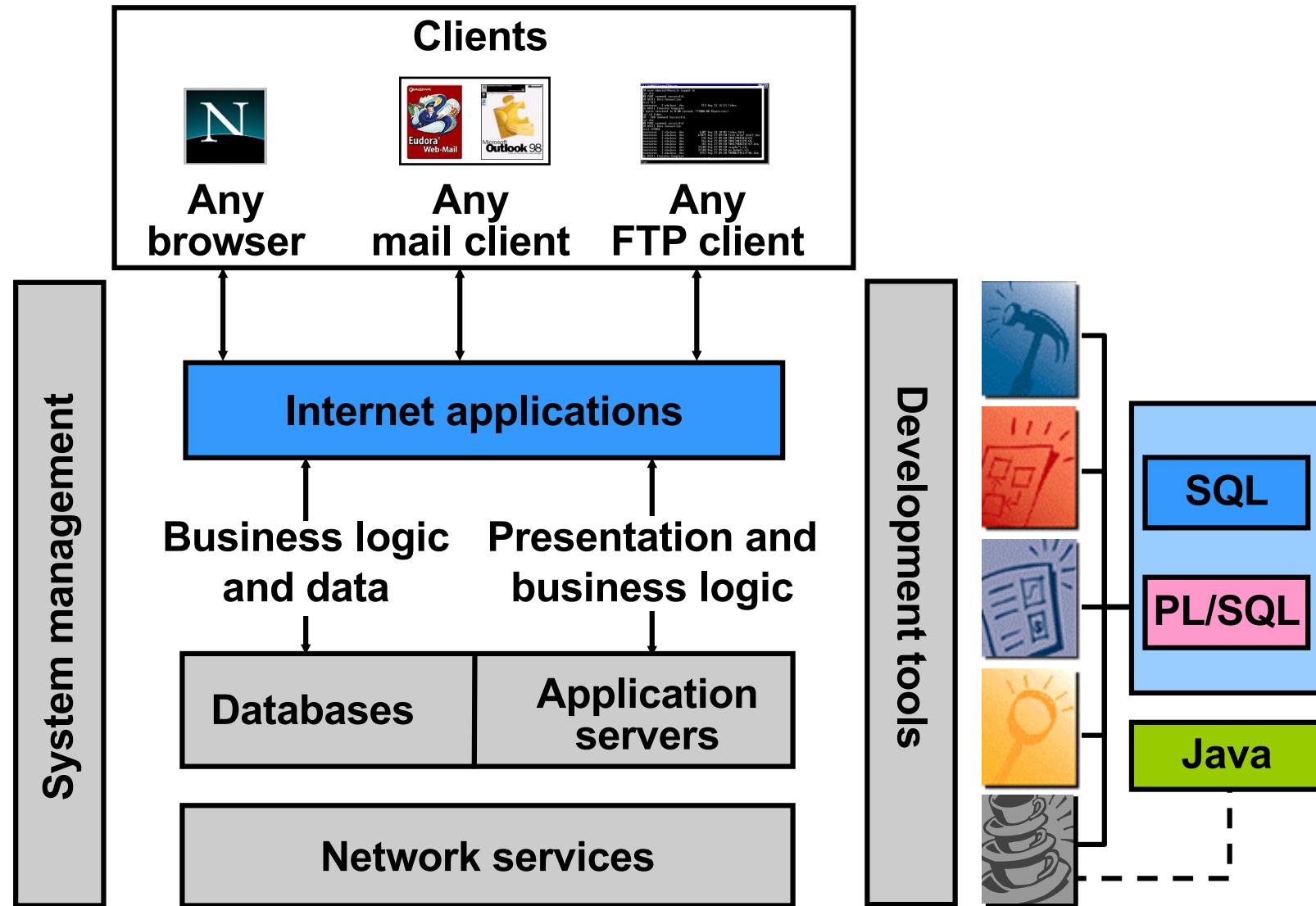
**Consultation de données**

**Manipulation de données (DML)**

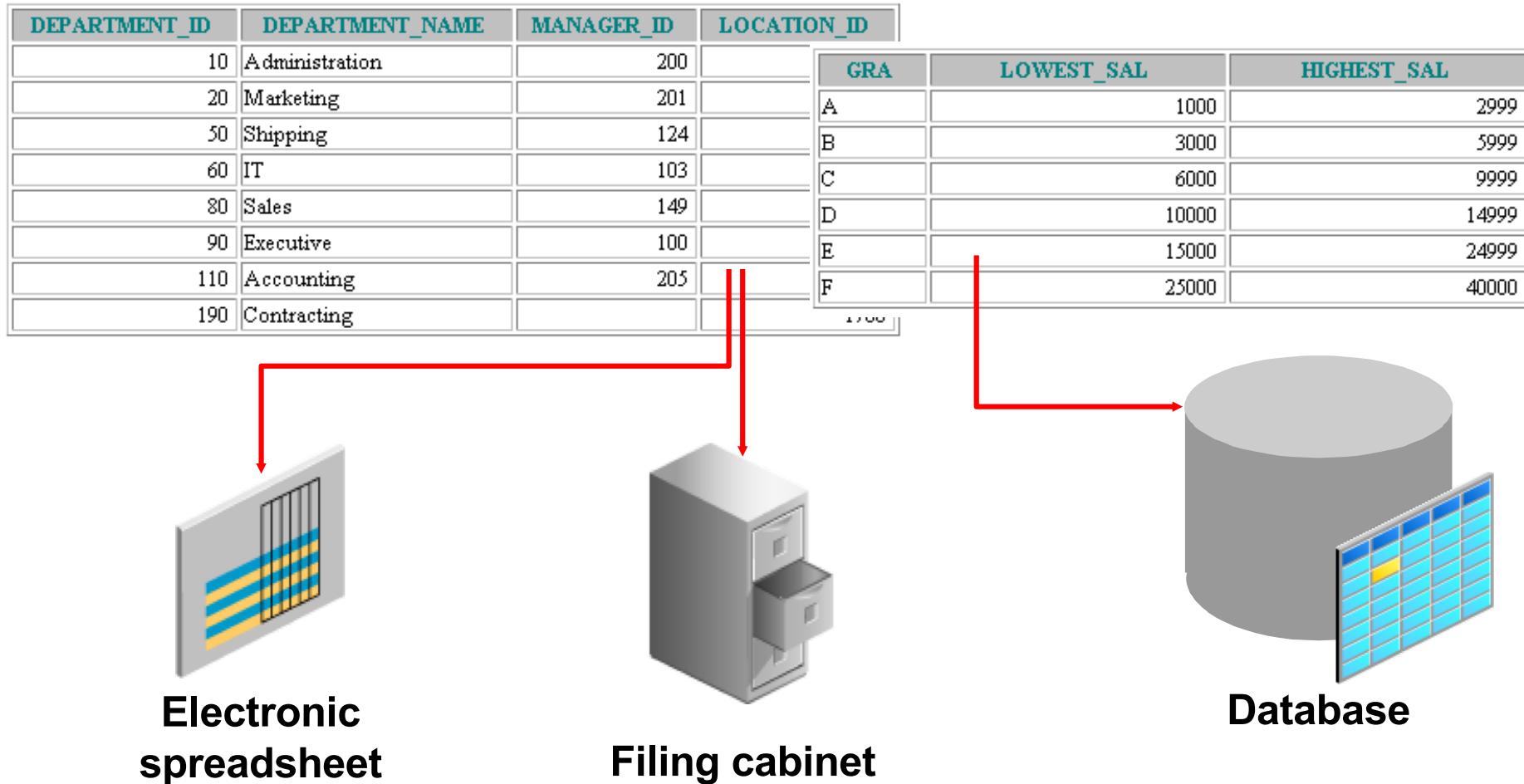
**Définition de données**

# INTRODUCTION

# Oracle Internet Platform



# Stockage de données sur différents supports

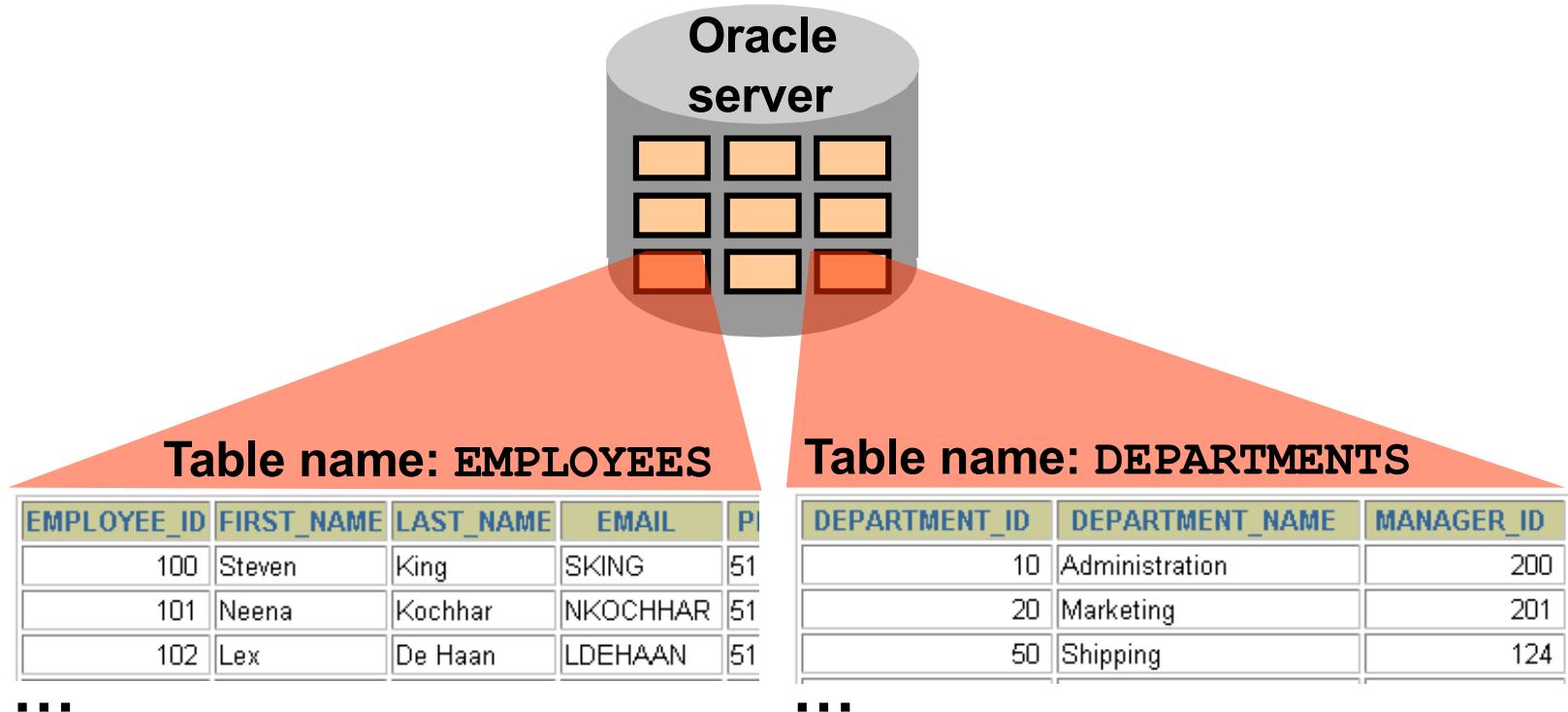


# Base de données Relationelle: Concept

- Dr. E. F. Codd a proposé le model relationel pour les systemes de gestion de base de données en 1970.
- Il est la base de *relational database management system (RDBMS)*.
- Le modèle relationnel se compose des éléments suivants
  - Collection d'objets ou relations
  - Ensemble d'opérateurs qui agissent sur les relations
  - Intégrité des données pour l'exactitude et la cohérence

# Définition d'une base de données relationnelle

**Une base de données relationnelle est un ensemble de relations ou de table à deux dimensions.**



# Lier plusieurs Tables

- Chaque ligne de données dans une table est identifiée par une clé primaire (PK).
- Vous pouvez relier logiquement de données de plusieurs tables à l'aide de clés étrangères (FK).

Table name: EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
174	Ellen	Abel	80
142	Curtis	Davies	50
102	Lex	De Haan	90
104	Bruce	Ernst	60
202	Pat	Fay	20
206	William	Gietz	110

...

Primary key

Foreign key

Table name: DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

Primary key

# Relational Database Terminology

A column as PK

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	SALARY	COMMISSION_PCT	DEPARTMENT_ID
100	King	Steven	24000		90
101	Kochhar	Neena	17000		90
102	De Haan	Lex	17000		90
103	Hunold	Alexander	9000		60
104	Ernst	Bruce	6000		60
107	Lorentz	Diana	4200		60
124	Mourgos	Kevin	5800		50
141	Rajs	Trenna	3500		50
142	Davies	Curtis	3100		50
143	Matos	Randall	2600		50
144	Vargas	Peter	2500		50
149	Zlotkey	Eleni	10500	.2	80
174	Abel	Ellen	11000	.3	80
176	Taylor	Jonathon	8600	.2	80
178	Grant	Kimberely	7000	.15	
200	Whalen	Jennifer	4400		10
201	Hartstein	Michael	13000		20
202	Fay	Pat	6000		20
205	Higgins	Shelley	12000		110
206	Gietz	William	8300		110

A single row

2

A column not a PK

3

null value

4

a foreign key.

5

A field

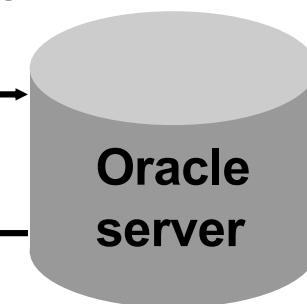
6

# Communiquer avec RDBMS en utilisant SQL

**SQL statement is entered.**

```
SELECT department_name  
FROM departments;
```

**Statement is sent to  
Oracle server.**



# SQL Statements

SELECT  
INSERT  
UPDATE  
DELETE  
MERGE

**Data manipulation language (DML)**

CREATE  
ALTER  
DROP  
RENAME  
TRUNCATE  
COMMENT

**Data definition language (DDL)**

GRANT  
REVOKE

**Data control language (DCL)**

COMMIT  
ROLLBACK  
SAVEPOINT

**Transaction control**

# Les Tables Utilisées dans le cours

## EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	240
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	170
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	170
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	90
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	42
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	58
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	35
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	31

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

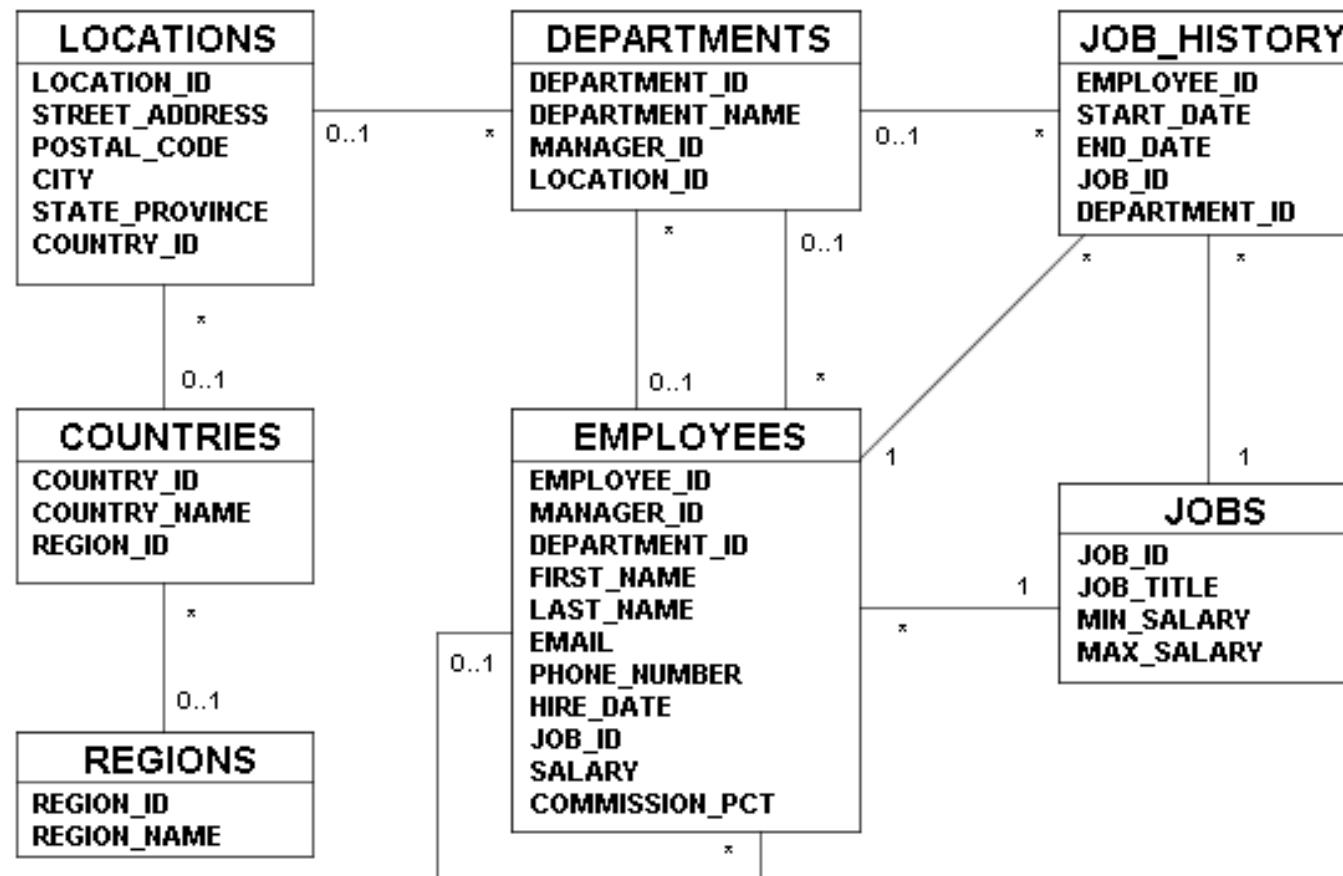
DEPARTMENTS

GRADE	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

JOB\_GRADES

ORACLE®

# Human Resources (hr) Data Set



# Récupération de données avec l'instruction SELECT

# SQL SELECT

**Projection**


**Table 1**

**Selection**


**Table 1**

**Join**


**Table 1**


**Table 2**

# La requête SELECT

```
SELECT * | { [DISTINCT] column|expression [alias],... }  
FROM    table;
```

- **SELECT identifie les colonnes à afficher.**
- **FROM identifie la table contenant ces colonnes.**

# Selectionner toutes les colonnes

```
SELECT *
FROM departments;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

8 rows selected.

# Selectionner des Colonnes Specifiques

```
SELECT department_id, location_id  
FROM departments;
```

DEPARTMENT_ID	LOCATION_ID
10	1700
20	1800
50	1500
60	1400
80	2500
90	1700
110	1700
190	1700

8 rows selected.

# Utilisation des Operateurs

```
SELECT last_name, salary, salary + 300  
FROM employees;
```

LAST_NAME	SALARY	SALARY+300
King	24000	24300
Kochhar	17000	17300
De Haan	17000	17300
Hunold	9000	9300
Ernst	6000	6300
...		

20 rows selected.

# La valeur Null

- Une valeur null est une valeur qui n'est pas disponible, non attribuées, inconnu ou inapplicable.
- Une valeur null n'est pas identique à un zéro ou un espace blanc.

```
SELECT last_name, job_id, salary, commission_pct  
FROM employees;
```

LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
King	AD_PRES	24000	
Kochhar	AD_VP	17000	
...			
Zlotkey	SA_MAN	10500	.2
Abel	SA_REP	11000	.3
Taylor	SA_REP	8600	.2
...			
Gietz	AC_ACCOUNT	8300	

20 rows selected.

# Valeurs null dans les Expressions arithmétiques

Les expressions arithmétiques qui contient une valeur nulle sont évaluées à null.

```
SELECT last_name, 12*salary*commission_pct  
FROM employees;
```

Kochhar	
Bana	
Manzal	TOTALMISSIONS*RAJAS=51
...	
Zlotkey	25200
Abel	39600
Taylor	20640
...	
Gietz	

20 rows selected.

# Utilisation des alias de colonnes

```
SELECT last_name AS name, commission_pct comm  
FROM employees;
```

NAME	COMM
King	
Kochhar	
De Haan	
...	

20 rows selected.

```
SELECT last_name "Name" , salary*12 "Annual Salary"  
FROM employees;
```

Name	Annual Salary
King	288000
Kochhar	204000
De Haan	204000
...	

20 rows selected.

# Opérateur de Concaténation

L'opérateur de concaténation :

- concatène des colonnes ou des strings
- Est représenté par (||)

```
SELECT      last_name || job_id AS "Employees"  
FROM        employees;
```

Employees
KingAD_PRES
KochharAD_VP
De HaanAD_VP
...

20 rows selected.

# Utilisation de la clause WHERE

- Limiter les lignes rentrées à l'aide de la clause WHERE :

```
SELECT * | { [DISTINCT] column|expression [alias],... }  
FROM   table  
[WHERE condition(s) ] ;
```

# Utiliser la condition NULL

Tester si null avec l'opérateur IS NULL.

```
SELECT last_name, manager_id  
FROM employees  
WHERE manager_id IS NULL ;
```

LAST_NAME	MANAGER_ID
King	

# Utiliser la fonction NVL

```
SELECT last_name, salary, NVL(commission_pct, 0),  
       (salary*12) + (salary*12*NVL(commission_pct, 0)) AN_SAL  
FROM employees;
```

LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
King	24000	0	288000
Kochhar	17000	0	204000
De Haan	17000	0	204000
Hunold	9000	0	108000
Ernst	6000	0	72000
Lorentz	4200	0	50400
Mourgos	5800	0	69600
Rajs	3500	0	42000
...			

20 rows selected.

# Expressions Conditionnelles

- Permet l'utilisation de la logique IF-THEN-ELSE dans une requête SQL
- Utilise deux méthodes:
  - L'expression CASE
  - La fonction DECODE

# CASE Expression

Facilite les requêtes conditionnelles en faisant le travail  
d'une Instruction IF-THEN-ELSE :

```
CASE expr WHEN comparison_expr1 THEN return_expr1
            [WHEN comparison_expr2 THEN return_expr2
            WHEN comparison_exprn THEN return_exprn
            ELSE else_expr]
END
```

# Utiliser une expression CASE

## Exemple

```
SELECT last_name, job_id, salary,  
       CASE job_id WHEN 'IT_PROG' THEN 1.10*salary  
                     WHEN 'ST_CLERK' THEN 1.15*salary  
                     WHEN 'SA REP' THEN 1.20*salary  
                     ELSE salary END "REVISED_SALARY"  
FROM employees;
```

LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

# Utiliser la Fonction DECODE

```
DECODE(col|expression, search1, result1  
      [, search2, result2,...,]  
      [, default])
```

```
SELECT last_name, job_id, salary,  
       DECODE(job_id, 'IT_PROG', 1.10*salary,  
              'ST_CLERK', 1.15*salary,  
              'SA REP', 1.20*salary,  
              salary)  
       REVISED_SALARY  
FROM employees;
```

LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
...			
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

ORACLE®

# Exercices

- 1. Produire un rapport qui affiche le nom et le salaire des employés qui gagnent plus de 12 000 \$. Le salaire de l'employé King doit être masqué avec des \*\*\*\*.**
- 2. Modifier la requête pour afficher le nom et le salaire des employés qui gagnent entre 5 000 \$ et 12 000 \$ et sont dans le service 20 ou 50. Étiqueter les colonnes par employé et salaire mensuel, respectivement.**

# Corrigé

1. Produire un rapport qui affiche le nom et le salaire des employés qui gagnent plus de 12 000 \$. Le salaire de l'employé King doit être masqué avec des \*\*\*\*.

```
SELECT last_name, job_id,
       DECODE(last_name, 'King', '*****', salary)
          SALARY
     FROM employees
    Where salary>12000;
```

# Exercices

3. Ecrire une requête qui retourne le nom d'un employé et le niveau de son salaire.

- Le niveau du salaire est
  - 'Low' si le salary<5000
  - 'Medium' si 5000<=salary<10000
  - 'Good' si 10000<=salary<20000 THEN '  
sinon, il est 'Excellent'

# Corrigé

3. Ecrire une requête qui retourne le nom d'un employé et le niveau de son salaire.

- Le niveau du salaire est
  - 'Low' si le salary<5000
  - 'Medium' si 5000<=salary<10000
  - 'Good' si 10000<=salary<20000 THEN '  
sinon, il est 'Excellent'

```
SELECT last_name,salary,  
(CASE WHEN salary<5000 THEN 'Low'  
WHEN salary<10000 THEN 'Medium'  
WHEN salary<20000 THEN 'Good'  
ELSE 'Excellent'  
END) qualified_salary  
FROM employees;
```

# Utiliser la clause ORDER BY

- Trier le résultat avec la clause ORDER BY :
  - ASC: avec un ordre ascendant, par default
  - DESC: avec un ordre descendant

```
SELECT      last_name, job_id, department_id, hire_date
FROM        employees
ORDER BY    hire_date;
```

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
King	AD_PRES	90	17-JUN-87
Whalen	AD_ASST	10	17-SEP-87
Kochhar	AD_VP	90	21-SEP-89
Hunold	IT_PROG	60	03-JAN-90
Ernst	IT_PROG	60	21-MAY-91

...

20 rows selected.

ORACLE®

# Utiliser la clause ORDER BY

- Tri descendant:

```
SELECT      last_name, job_id, department_id, hire_date  
FROM        employees  
ORDER BY    hire_date DESC ;
```

1

- Tri d'un alias:

```
SELECT employee_id, last_name, salary*12 annsal  
FROM      employees  
ORDER BY annsal ;
```

2

- Tri de multiples colonnes:

```
SELECT last_name, department_id, salary  
FROM      employees  
ORDER BY department_id, salary DESC;
```

3

# Exercices

- 1. Créer une requête pour afficher le nom et l'emploi de tous les employés qui n'ont pas de manager.**
- 2. Créer un rapport pour afficher le nom, le salaire et la commission de tous les employés qui gagnent des commissions. Trier les données par ordre décroissant de salaires et de commissions**

# Utiliser Group By

La fonction Group By opèrent sur des ensembles de lignes pour donner un résultat par groupe

EMPLOYEES

DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600
	7000
10	4400

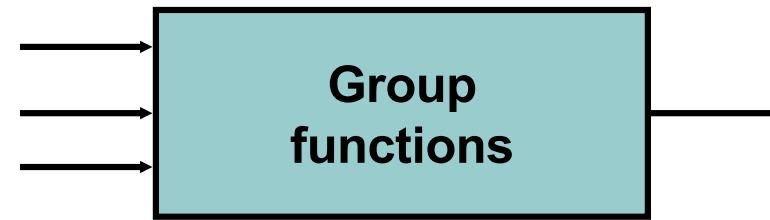
20 rows selected.

Maximum salary in  
EMPLOYEES table

MAX(SALARY)
24000

# Types de Group Functions

- **AVG**
- **COUNT**
- **MAX**
- **MIN**
- **STDDEV**
- **SUM**
- **VARIANCE**



# Group Functions: Syntax

```
SELECT      [column,] group_function(column), ...
FROM        table
[WHERE      condition]
[GROUP BY  column]
[ORDER BY  column] ;
```

# Utiliser les fonctions AVG, SUM, MIN et MAX

Vous pouvez utiliser AVG et SUM pour des données numériques.

```
SELECT AVG(salary), MAX(salary),  
       MIN(salary), SUM(salary)  
FROM   employees  
WHERE  job_id LIKE '%REP%';
```

AVG(SALARY)	MAX(SALARY)	MIN(SALARY)	SUM(SALARY)
8150	11000	6000	32600

Vous pouvez utiliser MIN et MAX pour les types de données date, numérique ou chaîne caractère.

```
SELECT MIN(hire date), MAX(hire date)  
FROM   employees;
```

MIN(HIRE_	MAX(HIRE_
17-JUN-87	29-JAN-00

# Utiliser la fonction COUNT

COUNT (\*) retourne le nombre de lignes dans une table

:

1

```
SELECT COUNT(*)  
FROM employees  
WHERE department_id = 50;
```

COUNT()

5

COUNT (expr) retourne le nombre de lignes avec des valeurs non nulles pour l'expr:

2

```
SELECT COUNT(commission_pct)  
FROM employees  
WHERE department_id = 80;
```

COUNT(COMMISSION\_PCT)

3

# Utiliser le mot clé DISTINCT

- COUNT(DISTINCT expr) retourne le nombre de valeurs non null distinctes de l'expr.
- afficher le nombre de valeurs distinctes de département\_id dans la table EMPLOYEES

```
SELECT COUNT(DISTINCT department_id)  
FROM employees;
```

COUNT(DISTINCTDEPARTMENT_ID)
7

# La Function Group by et les Valeurs Null

Les functions Group by ignorent les valeurs null:

1

```
SELECT AVG(commission_pct)  
FROM employees;
```

AVG(COMMISSION\_PCT)

.2125

La fonction NVL force group functions à inclure les null values:

2

```
SELECT AVG(NVL(commission_pct, 0))  
FROM employees;
```

AVG(NVL(COMMISSION\_PCT,0))

.0425

# La clause GROUP BY : Syntaxe

```
SELECT      column, group_function(column)
FROM        table
[WHERE      condition]
[GROUP BY  group_by_expression]
[ORDER BY  column];
```

**Vous pouvez diviser les lignes d'une table en petits groupes en utilisant la clause GROUP BY.**

# La clause GROUP BY : Exemple

Toutes les colonnes dans la liste de SELECT et qui ne sont pas dans les fonctions de groupe doivent être dans la clause GROUP BY..

```
SELECT department_id, AVG(salary)  
FROM employees  
GROUP BY department_id ;
```

DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

8 rows selected.

# Restreindre les Résultats de Group by avec la Clause HAVING

Lorsque vous utilisez la clause HAVING, le serveur Oracle restreint les groupes comme suit :

- 1. Les lignes sont regroupées.**
- 2. La function group by est appliquée.**
- 3. Les groupes vérifiant la clause HAVING sont affichés.**

```
SELECT      column, group_function
FROM        table
[WHERE      condition]
[GROUP BY  group_by_expression]
[HAVING    group_condition]
[ORDER BY  column];
```

# Restreindre les Résultats de Group by avec la Clause HAVING

```
SELECT      department_id, MAX(salary)
FROM        employees
GROUP BY    department_id
HAVING      MAX(salary)>10000 ;
```

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

# Restreindre les Résultats de Group by avec la Clause HAVING

```
SELECT      job_id, SUM(salary) PAYROLL
FROM        employees
WHERE       job_id NOT LIKE '%REP%'
GROUP BY    job_id
HAVING      SUM(salary) > 13000
ORDER BY    SUM(salary);
```

JOB_ID	PAYROLL
IT_PROG	19200
AD_PRES	24000
AD_VP	34000

# Exercices

Le département HR a besoin des rapports suivants:

- 1.** Trouver le plus haut, le plus bas, la somme et la moyenne des salaires de tous les employés. Étiqueter les colonnes respectivement par Maximum, Minimum, Sum et Average,
- 2.** Modifier la requête pour afficher le minimum, maximum, sum, et average des salaires pour chaque types d'emploi.
- 3.** Écrire une requête pour afficher le nombre de personnes ayant le même travail.

# Exercices

- 1. Créer une requête pour afficher le nombre total d'employés et, pour ce total, le nombre d'employés embauchés en 1995, 1996, 1997 et 1998. Créer des en-têtes de colonnes appropriés.**
- 2. Créer une requête pour afficher les emplois (job), les salaires correspondants basés sur le numéro de département et les sommes de salaires pour ces emplois, pour les départements 20, 50, 80 et 90. Créer des en-têtes de colonnes appropriés.**

# Excercise

```
SELECT COUNT(*) total,  
      SUM(DECODE(TO_CHAR(hire_date, 'YYYY'),1995,1,0))"1995",  
      SUM(DECODE(TO_CHAR(hire_date, 'YYYY'),1996,1,0))"1996",  
      SUM(DECODE(TO_CHAR(hire_date, 'YYYY'),1997,1,0))"1997",  
      SUM(DECODE(TO_CHAR(hire_date, 'YYYY'),1998,1,0))"1998"  
FROM employees;
```

# HIERARCHICAL RETRIEVAL

# Echantillons de la Table EMPLOYEES

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
100	King	AD_PRES	
101	Kochhar	AD_VP	100
102	De Haan	AD_VP	100
103	Hunold	IT_PROG	102
104	Ernst	IT_PROG	103
105	Austin	IT_PROG	103
106	Pataballa	IT_PROG	103
107	Lorentz	IT_PROG	103
108	Greenberg	FI_MGR	101

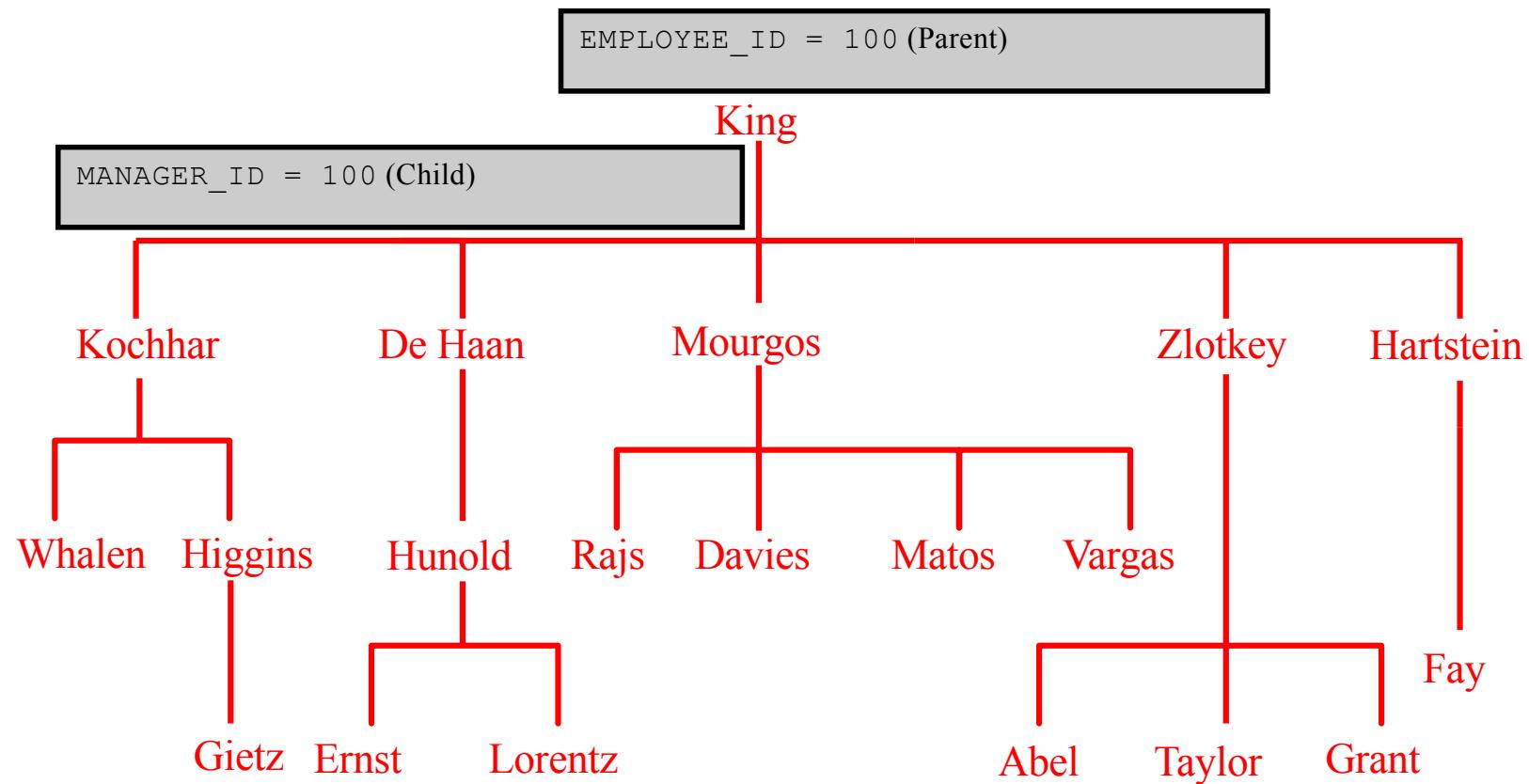
...

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
196	Walsh	SH_CLERK	124
197	Feeney	SH_CLERK	124
198	OConnell	SH_CLERK	124
199	Grant	SH_CLERK	124
200	Whalen	AD_ASST	101
201	Hartstein	MK_MAN	100
202	Fay	MK_REP	201
203	Mavris	HR_REP	101
204	Baer	PR_REP	101
205	Higgins	AC_MGR	101
206	Gietz	AC_ACCOUNT	205

107 rows selected.

ORACLE®

# Organigramme



# Requêtes hiérarchiques

```
SELECT [LEVEL], column, expr...
  FROM table
 [WHERE condition(s)]
 [START WITH condition(s)]
 [CONNECT BY PRIOR condition(s)] ;
```

**WHERE condition:**

```
expr comparison_operator expr
```

Note: ce genre de requête SELECT ne peut pas contenir de jointure

# Parcourir l'arbre

## Starting Point

- Spécifie la condition à remplir
- Accepte n'importe quelle condition valable

```
START WITH column1 = value
```

Pour la table EMPLOYEES, commencer par l'employé Kochhar.

```
...START WITH last_name = 'Kochhar'
```

Note: Les clauses CONNECT BY PRIOR et START WITH ne sont pas des standards ANSI SQL.

# Parcourir l'arbre

```
CONNECT BY PRIOR column1 = column2
```

## Parcourir de haut en bas

```
... CONNECT BY PRIOR employee_id = manager_id
```

### Direction

Top down → Column1 = Parent Key  
Column2 = Child Key

Bottom up → Column1 = Child Key  
Column2 = Parent Key

## Ou encore

```
... CONNECT BY employee_id = PRIOR manager_id
```

# Parcourir l'arbre : De bas en haut

```
SELECT employee_id, last_name, job_id, manager_id  
FROM   employees  
START  WITH employee_id = 101  
CONNECT BY PRIOR manager_id = employee_id ;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
101	Kochhar	AD_VP	100
100	King	AD_PRES	

Note: La clause CONNECT BY ne peut pas contenir de sous-requêtes!

# Parcourir l'arbre : De haut en bas

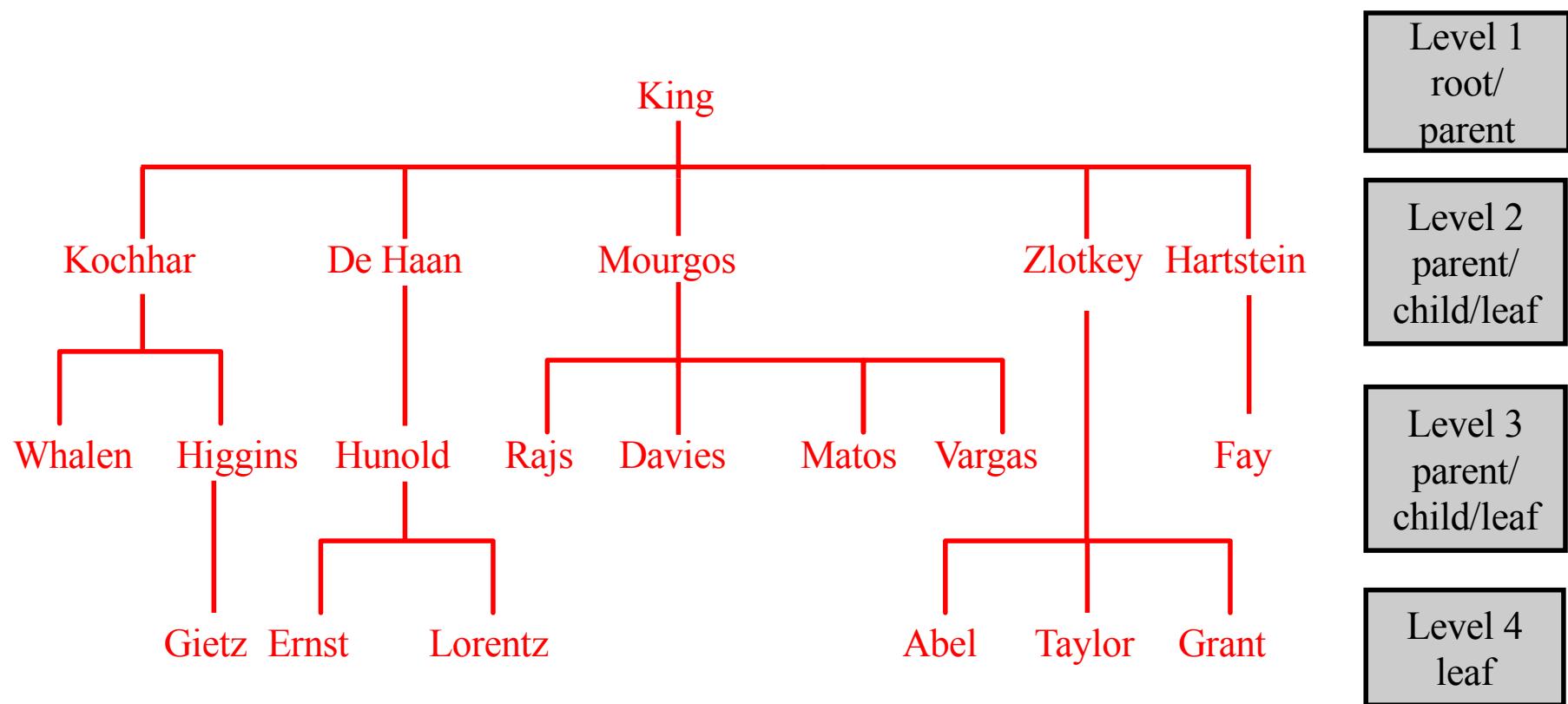
```
SELECT  last_name||' reports to '||  
PRIOR  last_name "Walk Top Down"  
FROM    employees  
START   WITH last_name = 'King'  
CONNECT BY PRIOR employee_id = manager_id ;
```

## Walk Top Down

King reports to
King reports to
Kochhar reports to King
Greenberg reports to Kochhar
Faviet reports to Greenberg
Chen reports to Greenberg
...

108 rows selected.

# Ranger les lignes avec LEVEL Pseudocolumn



# Mise en forme des rapports hiérarchiques à l'aide LEVEL and LPAD

Afficher la hiérarchie avec une mise en retrait des différents niveaux.

```
COLUMN org_chart FORMAT A12
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2, '_')
       AS org_chart
FROM   employees
START WITH first_name='Steven' AND last_name='King'
CONNECT BY PRIOR employee_id=manager_id
```

ORG_CHART
King
Kochhar
Greenberg
Faviet
Chen
Sciarrino
Urman
Popp
Whalen
Mavris
Baer
Higgins
Gietz

ORACLE®

# Exercices

- Produire un rapport affichant un organigramme du département de Mr Mourgos. Afficher les noms, les salaires et les département IDs de ces employés
- Produire un organigramme de la société.  
Commencez par la personne du plus haut niveau.

# **Consulter plusieurs tables**

# Obtention de données à partir de plusieurs Tables

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
...		
202	Fay	20
205	Higgins	110
206	Gietz	110

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700



EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
...		
102	90	Executive
205	110	Accounting
206	110	Accounting

# Types de Jointures

**Tous les types de jointures compatibles avec le standard SQL:1999 incluant se qui suit:**

- **Cross joins**
- **Natural joins**
- **USING clause**
- **Full (or two-sided) outer joins**
- **Arbitrary join conditions for outer joins**

# Jointure de Tables à l'aide SQL:1999 Syntaxe

Utiliser les jointures pour interroger les données de plusieurs tables :

```
SELECT      table1.column, table2.column
FROM        table1
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
    ON (table1.column_name = table2.column_name)] |
[LEFT|RIGHT|FULL OUTER JOIN table2
    ON (table1.column_name = table2.column_name)] |
[CROSS JOIN table2];
```

# Créer Natural Joins

- La clause NATURAL JOIN se base sur toutes les colonnes dans les deux tables ayant le même nom.
- elle sélectionne les lignes des deux tables qui ont des valeurs égales pour toutes les colonnes ayant les mêmes noms.
- Si les colonnes ayant le même nom ont des types de données différents, une erreur est renvoyée.

# Récupération des enregistrements avec Natural Joins

```
SELECT department_id, department_name,  
       location_id, city  
FROM   departments  
NATURAL JOIN locations ;
```

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
60	IT	1400	Southlake
50	Shipping	1500	South San Francisco
10	Administration	1700	Seattle
90	Executive	1700	Seattle
110	Accounting	1700	Seattle
190	Contracting	1700	Seattle
20	Marketing	1800	Toronto
80	Sales	2500	Oxford

8 rows selected.

# Récupération des Enregistrements avec Natural Joins et la Clause WHERE

```
SELECT department_id, department_name,  
       location_id, city  
  FROM departments  
NATURAL JOIN locations  
 WHERE department_id IN (20, 50);
```

**Des restrictions supplémentaires sur une jointure naturelle sont implémentées à l'aide d'une clause WHERE.**

# Récupération des Enregistrements avec la Clause USING

- Si plusieurs colonnes portent le même nom, ayant des types de données différents, la clause NATURAL JOIN peut être remplacée par une clause USING pour spécifier les colonnes qui doivent être utilisées pour cette equijoin.
- La clause USING peut être utilisée pour spécifier uniquement les colonnes qui doivent être utilisées pour une equijoin.
- NATURAL JOIN et USING sont mutuellement exclusive.

# Creating Joins with the USING Clause

EMPLOYEES

EMPLOYEE_ID	DEPARTMENT_ID
200	10
201	20
202	20
124	50
141	50
142	50
143	50
144	50
103	60
104	60
107	60
149	80
174	80
176	80

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
60	IT
60	IT
60	IT
80	Sales
80	Sales
80	Sales

Foreign key

Primary key

# Récupération des enregistrements avec la clause USING

```
SELECT employees.employee_id, employees.last_name,  
       departments.location_id, department_id  
  FROM employees JOIN departments  
USING (department_id);
```

EMPLOYEE_ID	LAST_NAME	LOCATION_ID	DEPARTMENT_ID
200	Whalen	1700	10
201	Hartstein	1800	20
202	Fay	1800	20
124	Mourgos	1500	50
141	Rajs	1500	50
142	Davies	1500	50
144	Vargas	1500	50
143	Matos	1500	50

...  
19 rows selected.

# Récupération des Enregistrements avec la Clause ON

```
SELECT e.employee_id, e.last_name, e.department_id,  
       d.department_id, d.location_id  
  FROM employees e JOIN departments d  
 WHERE e.department_id = d.department_id;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500
...				

19 rows selected.

# Self-Joins Using the ON Clause

EMPLOYEES (WORKER)

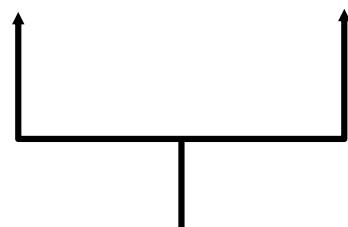
EMPLOYEE_ID	LAST_NAME	MANAGER_ID
100	King	
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103
107	Lorentz	103
124	Mourgos	100

...

EMPLOYEES (MANAGER)

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos

...



**MANAGER\_ID in the WORKER table is equal to  
EMPLOYEE\_ID in the MANAGER table.**

# Self-Joins Using the ON Clause

```
SELECT e.last_name emp, m.last_name mgr
FROM   employees e JOIN employees m
ON     (e.manager_id = m.employee_id);
```

EMP	MGR
Hartstein	King
Zlotkey	King
Mourgos	King
De Haan	King
Kochhar	King
...	

19 rows selected.

# Nonequi joins

EMPLOYEES

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000
Ernst	6000
Lorentz	4200
Mourgos	5800
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500
Zlotkey	10500
Abel	11000
Taylor	8600
...	

20 rows selected.

JOB\_GRADES

GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

← Salary in the EMPLOYEES table must be between lowest salary and highest salary in the JOB\_GRADES table.

# Récupération des Enregistrements avec Nonequi joins

```
SELECT e.last_name, e.salary, j.grade_level  
FROM employees e JOIN job_grades j  
ON e.salary  
BETWEEN j.lowest_sal AND j.highest_sal;
```

LAST_NAME	SALARY	GRA
Matos	2600	A
Vargas	2500	A
Lorentz	4200	B
Mourgos	5800	B
Rajs	3500	B
Davies	3100	B
Whalen	4400	B
Hunold	9000	C
Ernst	6000	C
...		

20 rows selected.

# Outer Joins

DEPARTMENTS

DEPARTMENT_NAME	DEPARTMENT_ID
Administration	10
Marketing	20
Shipping	50
IT	60
Sales	80
Executive	90
Accounting	110
Contracting	190

8 rows selected.

EMPLOYEES

DEPARTMENT_ID	LAST_NAME
90	King
90	Kochhar
90	De Haan
60	Hunold
60	Ernst
60	Lorentz
50	Mourgos
50	Rajs
50	Davies
50	Matos
50	Vargas
80	Zlotkey

20 rows selected.

Aucun employé pour le département 190;  
*Une INNER JOIN (NATURAL JOIN, USING, ou ON ) n'affichera pas ce département*

## LEFT OUTER JOIN

Cette requête récupère toutes les lignes de la table EMPLOYEES, qui est la table de gauche, même s'il n'a aucune correspondance dans la table DEPARTMENTS.

```
SELECT e.last_name, e.department_id, d.department_name  
FROM employees e LEFT OUTER JOIN departments d  
ON (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing

...

De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		

20 rows selected.

L'employé Grant figure bien parmi les résultats grâce à une *OUTER JOIN*

## RIGHT OUTER JOIN

Cette requête récupère toutes les lignes de la table DEPARTMENTS, qui est la table de droite, même s'il n'a aucune correspondance dans la table EMPLOYEES.

```
SELECT e.last_name, e.department_id, d.department_name  
FROM employees e RIGHT OUTER JOIN departments d  
ON (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
Davies	50	Shipping
...		
Kochhar	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
	190	Contracting

20 rows selected.

le département 190 figure bien parmi les résultats grâce à une *OUTER JOIN*

## FULL OUTER JOIN

```
SELECT e.last_name, d.department_id, d.department_name  
FROM employees e FULL OUTER JOIN departments d  
ON (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
...		
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		
	190	Contracting

21 rows selected.

# Récupération des Enregistrements avec Cross Joins

- CROSS JOIN produit le produit cartésien de deux tables

```
SELECT last_name, department_name  
FROM employees  
CROSS JOIN departments ;
```

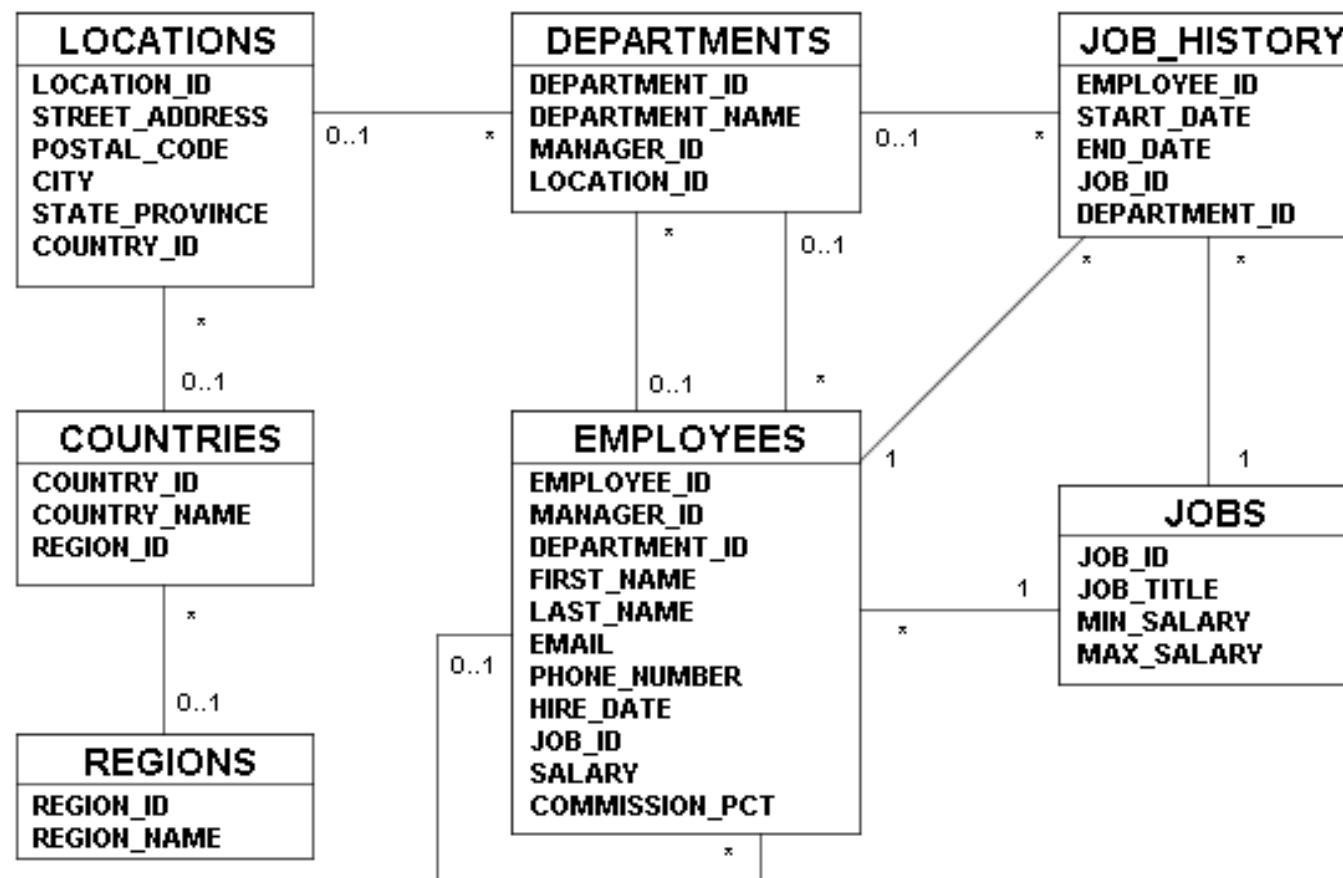
LAST_NAME	DEPARTMENT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration
■ ■ ■	

160 rows selected.

# Exercices

- 1. Écrire une requête pour produire les adresses de tous les départements. Utiliser les tables LOCATIONS et COUNTRIES. Afficher la location ID, street address, city, state or province, et country dans le résultat. Utiliser une NATURAL JOIN pour produire ces résultats. tout d'abord afficher la structure de la table LOCATION**
- 2. créer une requête qui affiche le nom, le job, le nom du département, le salaire et le nom du grade de tous les employés.**

# Human Resources (hr) Data Set



## Exercices

- 3.** Produire un rapport des salariés de Toronto. Afficher le nom, le job, numéro de département et le nom du département pour tous les employés qui travaillent à Toronto.
- 4.** Produire un rapport qui affiche le nom d'un employé son Num ainsi que le nom de son manager et le Num de ce dernier. Étiqueter les colonnes par Employee, Emp#, Manager, and Mgr#, respectivement.

# **Exercices**

- 5. Afficher les noms et les dates d'embauche pour tous les employés qui ont été embauchés avant leurs managers, ainsi que les noms et les dates d'embauche de leurs managers.**

# Sous-requête : Syntaxe

```
SELECT      select_list
FROM        table
WHERE       expr operator
            (SELECT      select_list
             FROM       table) ;
```

- La sous-requête (requête interne) s'exécute avant la requête principale (requête externe).
- Le résultat de la sous-requête est utilisé par la requête principale.

## Sous-requête : Exemple

```
SELECT last_name, salary  
FROM   employees    11000  
WHERE  salary >  
       (SELECT salary  
        FROM   employees  
        WHERE  last_name = 'Abel');
```

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hartstein	13000
Higgins	12000

# **Manipulation de données**

# Data Manipulation Language: DML

- **Une instruction DML est utilisée lorsque vous :**
  - Ajoutez des lignes dans une table
  - Modifiez des lignes existantes dans une table
  - Supprimez les lignes existantes d'une table
- **Une *transaction* se compose d'une collection d'instructions DML qui forment une unité logique de travail.**

# Ajouter une nouvelle ligne à une Table

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

70 Public Relations

100

1700

Nouvelle  
ligne

Insérer la nouvelle ligne  
dans la table  
DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

70 Public Relations

100

1700

# INSERT : Synataxe

- **INSERT statement:**

```
INSERT INTO    table [(column [, column...])]  
VALUES          (value [, value...]);
```

- **Avec cette syntaxe, une seule ligne est insérée à la fois.**

# Insérer une nouvelle ligne

- Insérer une nouvelle ligne contenant les valeurs pour chaque colonne.
- Lister les colonnes et puis les valeurs correspondantes
- Par défaut, toutes les colonnes d'une table sont considérées

```
INSERT INTO departments(department_id,  
                      department_name, manager_id, location_id)  
VALUES (70, 'Public Relations', 100, 1700);  
1 row created.
```

- Encadrer les valeurs de caractère et de la date entre apostrophes.

# Copie des lignes d'une autre Table

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
SELECT employee_id, last_name, salary, commission_pct
FROM employees
WHERE job_id LIKE '%REP%';
```

4 rows created.

- Ne pas utiliser la clause VALUES.

# UPDATE : Syntaxe

- **Modifier des données existantes avec UPDATE**

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

- **Peut modifier plusieurs lignes à la fois.**

# Mettre à jour les lignes d'une Table

```
UPDATE employees  
SET department_id = 70  
WHERE employee_id = 113;  
1 row updated.
```

- Toutes les lignes de la table sont modifiées si vous omettez la clause WHERE :

```
UPDATE copy_emp  
SET department_id = 110;  
22 rows updated.
```

# Mise à jour des lignes en à partir d'une autre Table

```
UPDATE copy_emp
SET department_id = (SELECT department_id
                      FROM employees
                      WHERE employee_id = 100)
WHERE job_id          = (SELECT job_id
                         FROM employees
                         WHERE employee_id = 200);
1 row updated.
```

# DELETE

## Syntaxe:

```
DELETE [FROM]    table
[WHERE          condition] ;
```

- **Exemple**

```
DELETE FROM departments
WHERE department_name = 'Finance';
1 row deleted.
```

- **Toutes les lignes dans la table sont supprimées si vous omettez la clause WHERE :**

```
DELETE FROM copy_emp;
22 rows deleted.
```

## TRUNCATE

- Supprime toutes les lignes d'une table, laissant la structure de la table intacte.

### Attention!

- est une instruction de langage (DDL) de définition de données plutôt qu'une instruction DML ; ne peut pas être facilement annulée par un ROLLBACK
- Ne déclenche pas les delete triggers de la table.
- Syntaxe:

```
TRUNCATE TABLE table_name;
```

- Exemple:

```
TRUNCATE TABLE copy_emp;
```

# **LES TRANSACTIONS**

# Les Transactions

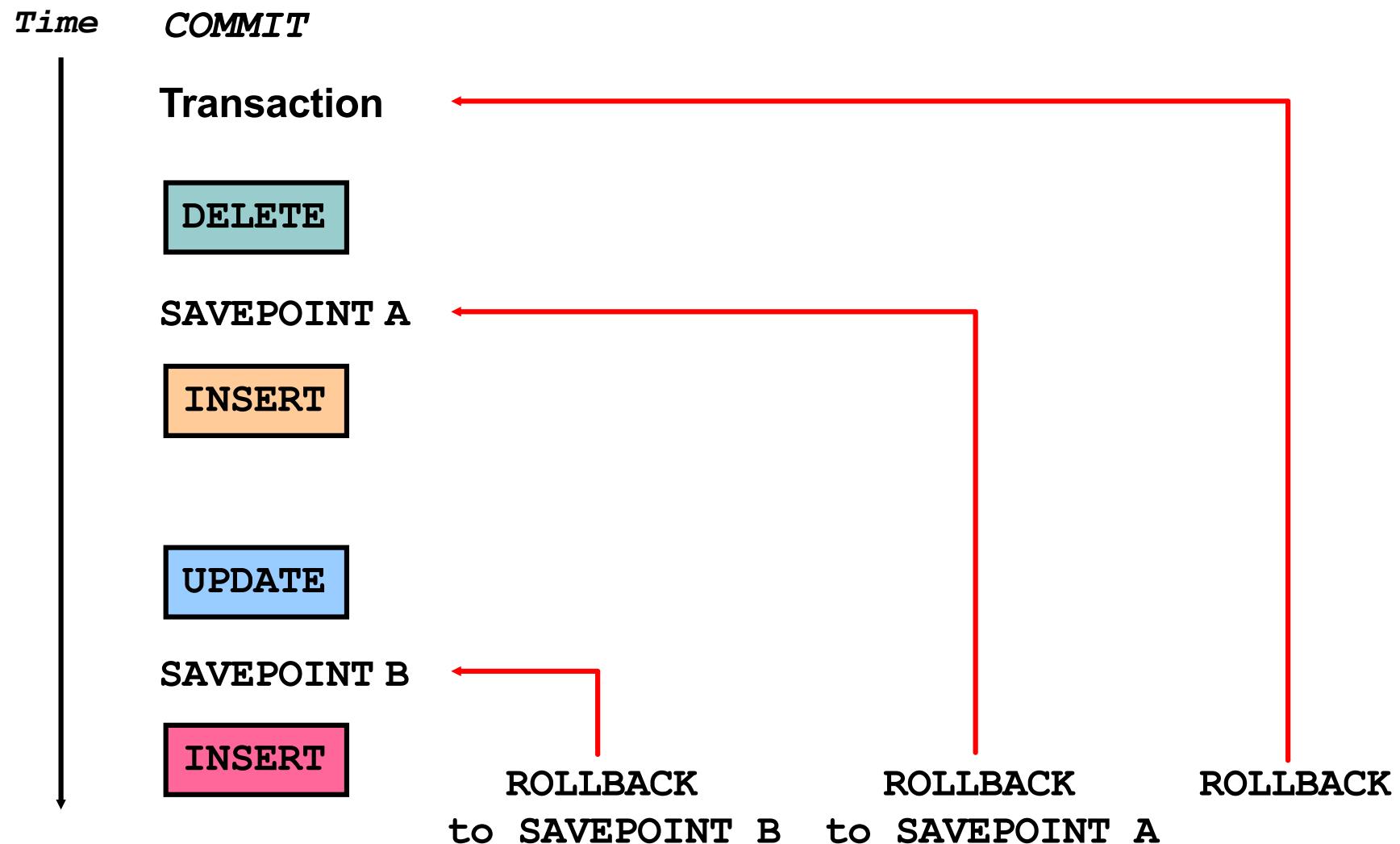
**Une transaction de base de données est constituée de l'une des opérations suivantes**

- **Un ensemble d'instructions DML qui constituent une logique de MAJ de la BD**
- **Une instruction DDL pour la définition de données**
- **Une instruction DCL pour le contrôle des données**

# Les Transactions

- **Commence lors de la première instruction DML**
- **Se termine par l'un des événements suivants**
  - Un COMMIT ou ROLLBACK
  - Une instruction DDL ou DCL (Attention! Dans ce cas un commit automatique est effectué).
  - L'utilisateur quitte le système
  - Le système sort inopinément

# Contrôler une Transactions



**Note:** SAVEPOINT is not ANSI standard SQL.

# Annuler les changements avec ROLLBACK

- Créer un marqueur dans une transaction en cours en utilisant l'instruction SAVEPOINT.
- Annuler les chagements depuis ce marqueur à l'aide de l'instruction ROLLBACK TO SAVEPOINT.

```
UPDATE...  
SAVEPOINT update_done;  
Savepoint created.  
INSERT...  
ROLLBACK TO update_done;  
Rollback complete.
```

# Validation des données avec COMMIT

- Apportez les modifications

```
DELETE FROM employees  
WHERE employee_id = 99999;  
1 row deleted.
```

```
INSERT INTO departments  
VALUES (290, 'Corporate Tax', NULL, 1700);  
1 row created.
```

- Validez les modifications :

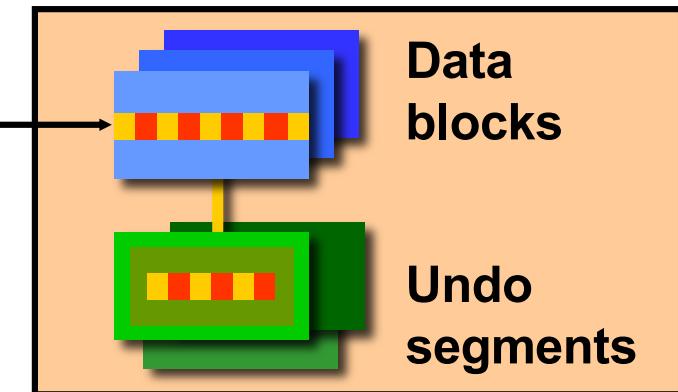
```
COMMIT;  
Commit complete.
```

# Lecture cohérente

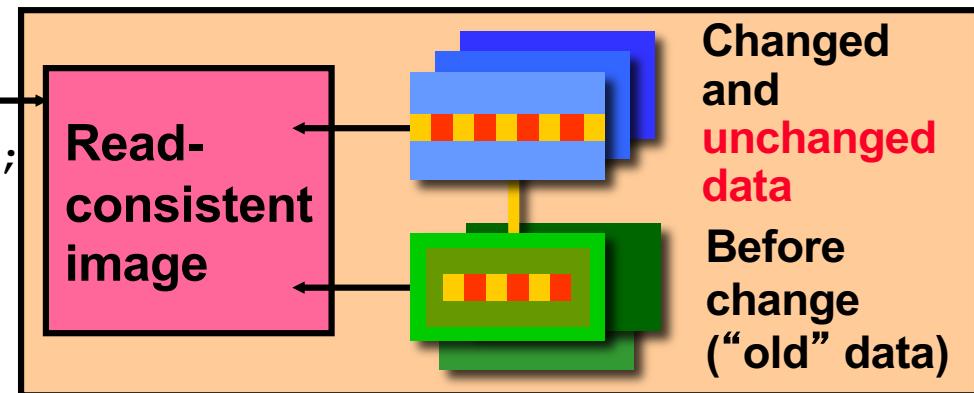
User A



```
UPDATE employees  
SET salary = 7000  
WHERE last_name = 'Grant';
```



```
SELECT *  
FROM userA.employees;
```



User B

# **Utilisation de DDL pour la définition de données**

# Database Objects

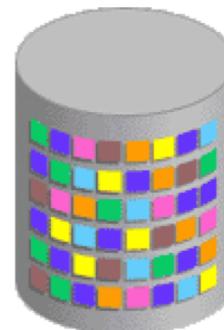
Object	Description
Table	L'unité basique de stockage. Elle est composée de lignes de données
View	Représente logiquement un sous-ensemble de données d'une ou de plusieurs tables.
Sequence	Génère une séquence de valeurs numériques
Index	Améliore la performance des requêtes.
Synonym	Donne des alternatives à des noms d'objets.

# CREATE TABLE

- Vous devez avoir :
  - Le privilège CREATE TABLE
  - Un espace de stockage

```
CREATE TABLE [schema.] table  
    (column datatype [DEFAULT expr] [, . . .]);
```

- Vous spécifiez :
  - Le nom de la Table
  - Column name, column data type, and column size



# Creation d'une Tables

- Créer une table.

```
CREATE TABLE dept
  (deptno      NUMBER(2),
   dname       VARCHAR2(14),
   loc         VARCHAR2(13),
   create_date DATE DEFAULT SYSDATE);
```

Table created.

- Vérifier la création

```
DESCRIBE dept
```

Name	Null?	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)
CREATE_DATE		DATE

# Inclure des Contraintes

- Les contraintes forcent l'application des règles de gestion au niveau de la table.
- Types de contraintes:
  - NOT NULL
  - UNIQUE
  - PRIMARY KEY
  - FOREIGN KEY
  - CHECK



# Definir les Contraintes

- **Syntaxe:**

```
CREATE TABLE [schema.]table  
    (column datatype [DEFAULT expr]  
     [column_constraint],  
      ...  
     [table_constraint] [, . . . ]);
```

- **Au niveau d'une colonne**

```
column [CONSTRAINT constraint_name] constraint_type,
```

- Au niveau de la table**

```
column, ...  
      [CONSTRAINT constraint_name] constraint_type  
      (column, . . . ),
```

# Definir les Contraintes

- Au niveau d'une colonne

```
CREATE TABLE employees (
    employee_id  NUMBER(6)
        CONSTRAINT emp_emp_id_pk PRIMARY KEY,
    first_name    VARCHAR2(20),
    ...);
```

1

- Au niveau de la table

```
CREATE TABLE employees (
    employee_id  NUMBER(6),
    first_name    VARCHAR2(20),
    ...
    job_id        VARCHAR2(10) NOT NULL,
    CONSTRAINT emp_emp_id_pk
        PRIMARY KEY (EMPLOYEE_ID));
```

2

# La Contrainte FOREIGN KEY

Définie au niveau de la table

```
CREATE TABLE employees (
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
    department_id    NUMBER(4),
    .
    .
    .
    CONSTRAINT emp_dept_fk FOREIGN KEY (department_id)
        REFERENCES departments(department_id),
    CONSTRAINT emp_email_uk UNIQUE(email));
```

# La Contrainte FOREIGN KEY

Définie au niveau de la colonne

```
CREATE TABLE employees (
    employee_id      NUMBER(6) ,
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25) ,
    salary           NUMBER(8,2) ,
    commission_pct   NUMBER(2,2) ,
    hire_date        DATE NOT NULL,
    department_id    NUMBER(4) CONSTRAINT emp_dept_fk
                      REFERENCES departments(department_id),
    .
    .
    .
    CONSTRAINT emp_email_uk UNIQUE(email));
```

# CREATE TABLE: Exemple

```
CREATE TABLE employees
( employee_id      NUMBER(6)
  CONSTRAINT emp_employee_id PRIMARY KEY
, first_name        VARCHAR2(20)
, last_name         VARCHAR2(25)
  CONSTRAINT emp_last_name_nn NOT NULL
, email             VARCHAR2(25)
  CONSTRAINT emp_email_nn    NOT NULL
  CONSTRAINT emp_email_uk    UNIQUE
, phone_number      VARCHAR2(20)
, hire_date         DATE
  CONSTRAINT emp_hire_date_nn NOT NULL
, job_id            VARCHAR2(10)
  CONSTRAINT emp_job_nn     NOT NULL
, salary            NUMBER(8,2)
  CONSTRAINT emp_salary_ck   CHECK (salary>0)
, commission_pct    NUMBER(2,2)
, manager_id        NUMBER(6)
, department_id     NUMBER(4)
  CONSTRAINT emp_dept_fk    REFERENCES
                           departments (department_id));
```

# Création d'une Table à l'aide d'une sous-requête

```
CREATE TABLE dept80
AS
SELECT employee_id, last_name,
       salary*12 ANNSAL,
       hire_date
  FROM employees
 WHERE department_id = 80;
```

Table created.

```
DESCRIBE dept80
```

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
ANNSAL		NUMBER
HIRE_DATE	NOT NULL	DATE

# **Créer d'autres Objets de la BD**

# Database Objects

Object	Description
Table	L'unité basique de stockage. Elle est composée de lignes de données
View	Représente logiquement un sous-ensemble de données d'une ou de plusieurs tables.
Sequence	Génère une séquence de valeurs numériques
Index	Améliore la performance des requêtes.
Synonym	Donne des alternatives à des noms d'objets.

# C'est quoi une vue

## EMPLOYEES table

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	Kirg	SKING	515.123.4567	17-JUN-87	AD_FRES	2400
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	1700
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	1700
103	Alexander	Humold	AHUMOLD	590.423.4567	03-JAN-90	IT_PROG	900
104	Bruce	Emart	BERNST	590.423.4668	21-MAY-91	IT_PROG	600
107	Diana	Lorentz	DLORENTZ	590.423.5667	07-FEB-99	IT_PROG	420
124	Kavar	Mourgos	INMOURGOS	650.123.5234	16-NOV-99	ST_MAN	580
141	Trenna	Rac	TRAIS	650.121.8009	17-OCT-95	ST_CLERK	350
142	Curtis	Davies	CDAVIES	650.121.2894	29-JAN-97	ST_CLERK	310
143	Randall	Matos	RMATOD	800.121.3074	16-MAR-90	ST_CLERK	200
EMPLOYEE_ID	LAST_NAME		SALARY				
	149	Zlotkey	10500	JUL-96	ST_CLERK	250	
	174	Abel	11000	JAN-00	SA_MAN	1050	
	176	Taylor	0600	MAY-96	SA REP	1100	
	178	Kimberly	0700	MAR-98	SA REP	860	
	200	Jennifer	515.123.4444	24-MAY-99	SA REP	700	
	201	Michael	JWHALEN	515.123.5555	17-SEP-87	AD_ASST	440
	202	Pat	MHARTSTE	515.123.6666	17-FEB-96	MK_MAN	1300
	205	Shelley	PFAY	603.123.6666	17-AUG-97	MK REP	600
	206	William	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	1200
			WGIEZ	515.123.8181	07-JUN-94	AC_ACCOUNT	830

20 rows selected.

ORACLE®

# Créer une Vue

## Syntaxe

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
  [(alias[, alias]...)]
  AS subquery
  [WITH CHECK OPTION [CONSTRAINT constraint]]
  [WITH READ ONLY [CONSTRAINT constraint]];
```

- **Exemple:** Créez la vue EMPVU80, qui contient les détails des employés dans le département 80 :

```
CREATE VIEW empvu80
  AS SELECT employee_id, last_name, salary
    FROM employees
   WHERE department_id = 80;
```

**View created.**

- Décrire la structure de la vue

```
DESCRIBE empvu80
```

# Consulter une vue

```
SELECT *  
FROM salvu50;
```

ID_NUMBER	NAME	ANN_SALARY
124	Mourgos	69600
141	Rajs	42000
142	Davies	37200
143	Matos	31200
144	Vargas	30000

# Simple Views and Complex Views

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operations through a view	Yes	Not always

# Modifier une vue

- Avec l'utilisation de CREATE OR REPLACE VIEW.

```
CREATE OR REPLACE VIEW empvu80
  (id_number, name, sal, department_id)
AS SELECT employee_id, first_name || ' '
          || last_name, salary, department_id
     FROM employees
    WHERE department_id = 80;
```

**View created.**

# Création d'une vue complexe

Créer une vue complexe qui contient des fonctions de groupe et un jointure:

```
CREATE OR REPLACE VIEW dept_sum_vu
  (name, minsal, maxsal, avgsal)
AS SELECT      d.department_name, MIN(e.salary) ,
                MAX(e.salary), AVG(e.salary)
  FROM        employees e JOIN departments d
  ON          (e.department_id = d.department_id)
 GROUP BY    d.department_name;
```

**View created.**

# Règles pour l'exécution des Opérations DML sur une vue

- Vous pouvez généralement effectuer des opérations DML sur des vues simples. 
- Vous ne pouvez pas supprimer des lignes si la vue contient les éléments suivants :
  - Group functions
  - A GROUP BY clause
  - The DISTINCT keyword
  - The pseudocolumn ROWNUM keyword

# Règles pour l'exécution des Opérations DML sur une vue

**Vous ne pouvez pas modifier les données dans une vue si elle contient :**

- **Group functions**
- **A GROUP BY clause**
- **The DISTINCT keyword**
- **The pseudocolumn ROWNUM keyword**
- **Columns defined by expressions**

# Règles pour l'exécution des Opérations DML sur une vue

**Vous ne pouvez pas ajouter de données via une vue si elle inclut :**

- **Group functions**
- **A GROUP BY clause**
- **The DISTINCT keyword**
- **The pseudocolumn ROWNUM keyword**
- **Columns defined by expressions**
- **NOT NULL columns in the base tables that are not selected by the view**

# Using the WITH CHECK OPTION Clause

- Vous pouvez assurer que les opérations DML effectuées sur une de vue restent dans le domaine de la vue en utilisant la clause WITH CHECK OPTION clause:

```
CREATE OR REPLACE VIEW empvu20
AS SELECT *
  FROM employees
 WHERE department_id = 20
  WITH CHECK OPTION CONSTRAINT empvu20_ck ;
```

View created.

- Toute tentative visant à modifier le numéro de département pour toute ligne de la vue échoue parce qu'il viole contrainte la WITH CHECK OPTION.

# Suppression d'une vue

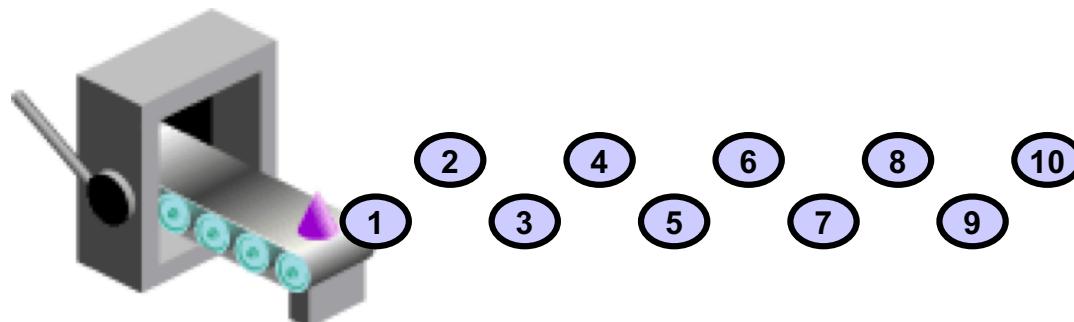
**Vous pouvez supprimer une vue sans perte de données parce qu'une vue est basée sur des tables sous-jacentes dans la base de données.**

```
DROP VIEW view;
```

```
DROP VIEW empvu80;  
View dropped.
```

# Les Sequences

- Générer automatiquement des numéros uniques
- Est un objet partageable
- Peut être utilisé pour créer une valeur de clé primaire
- Représente des codes artificiels et non des codes métiers



# CREATE SEQUENCE :

## Syntaxe

```
CREATE SEQUENCE sequence
    [INCREMENT BY n]
    [START WITH n]
    [ {MAXVALUE n | NOMAXVALUE} ]
    [ {MINVALUE n | NOMINVALUE} ]
    [ {CYCLE | NOCYCLE} ]
    [ {CACHE n | NOCACHE} ] ;
```

## Exemple

```
CREATE SEQUENCE dept_deptid_seq
    INCREMENT BY 10
    START WITH 120
    MAXVALUE 9999
    NOCACHE
    NOCYCLE;
```

Sequence created.

## **NEXTVAL et CURRVAL Pseudocolumns**

- **NEXTVAL** retourne la valeur suivante de la séquence disponible. Elle retourne une valeur unique, chaque fois qu'il est référencé, même pour des utilisateurs différents.
- **CURRVAL** Obtient la valeur actuelle de la séquence.

# Utiliser une Sequence

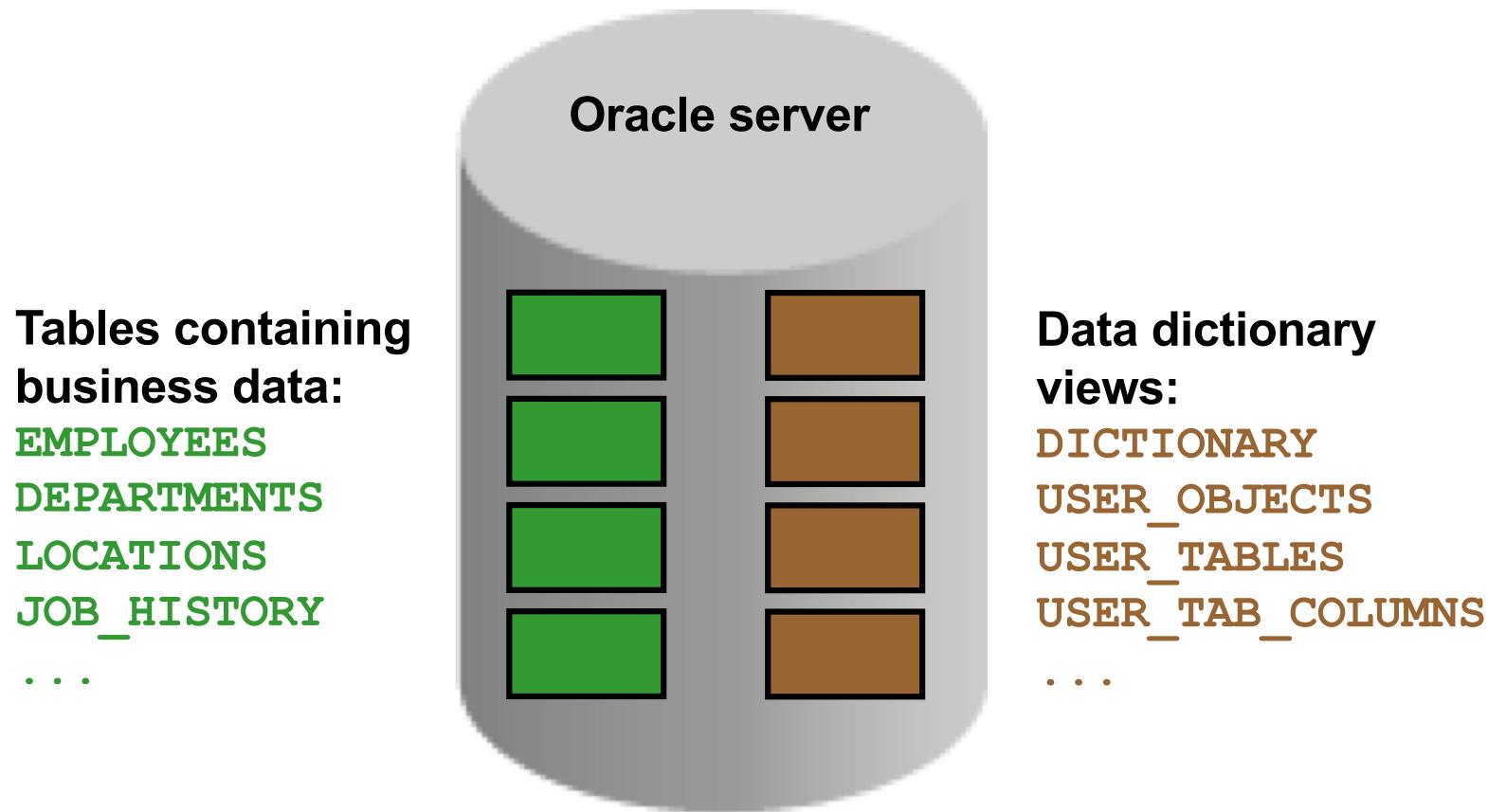
```
INSERT INTO departments(department_id,  
                      department_name, location_id)  
VALUES      (dept_deptid_seq.NEXTVAL,  
                  'Support', 2500);  
  
1 row created.
```

- Afficher la valeur actuelle de la séquence  
**DEPT\_DEPTID\_SEQ :**

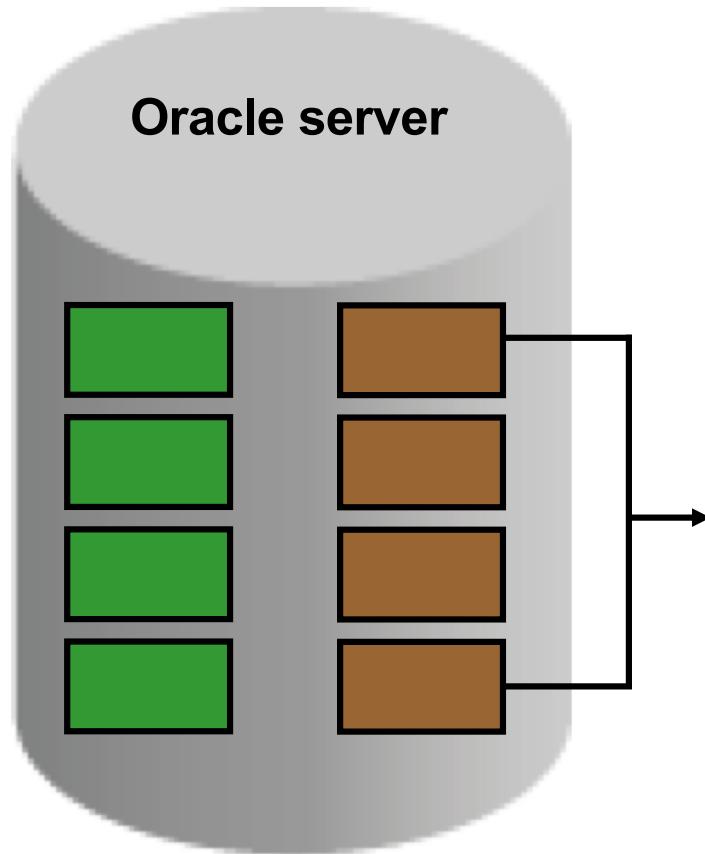
```
SELECT      dept_deptid_seq.CURRVAL  
FROM        dual;
```

# **Utilisation du dictionnaire de données**

# The Data Dictionary



# Data Dictionary Structure



→

- Consists of:**
- **Base tables**
  - **User-accessible views**

# Data Dictionary Structure

## View naming convention:

View Prefix	Purpose
USER	User's view (what is in your schema; what you own)
ALL	Expanded user's view (what you can access)
DBA	Database administrator's view (what is in everyone's schemas)
V\$	Performance-related data

# Comment utiliser les vues du dictionnaire

Commencez par DICTIONARY. Il contient les noms et les descriptions des vues du dictionnaire.

**DESCRIBE DICTIONARY**

Name	Null?	Type
TABLE_NAME		VARCHAR2(30)
COMMENTS		VARCHAR2(4000)

```
SELECT *
FROM   dictionary
WHERE  table_name = 'USER_OBJECTS';
```

TABLE_NAME	COMMENTS
USER_OBJECTS	Objects owned by the user

## USER\_OBJECTS View

```
SELECT object_name, object_type, created, status  
FROM user_objects  
ORDER BY object_type;
```

OBJECT_NAME	OBJECT_TYPE	CREATED	STATUS
REG_ID_PK	INDEX	10-DEC-03	VALID
...			
DEPARTMENTS_SEQ	SEQUENCE	10-DEC-03	VALID
REGIONS	TABLE	10-DEC-03	VALID
LOCATIONS	TABLE	10-DEC-03	VALID
DEPARTMENTS	TABLE	10-DEC-03	VALID
JOB_HISTORY	TABLE	10-DEC-03	VALID
JOB_GRADES	TABLE	10-DEC-03	VALID
EMPLOYEES	TABLE	10-DEC-03	VALID
JOBS	TABLE	10-DEC-03	VALID
COUNTRIES	TABLE	10-DEC-03	VALID
EMP_DETAILS_VIEW	VIEW	10-DEC-03	VALID

# Table Information

## **USER\_TABLES:**

```
DESCRIBE user_tables
```

Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
TABLESPACE_NAME		VARCHAR2(30)
CLUSTER_NAME		VARCHAR2(30)
IOT_NAME		VARCHAR2(30)

```
SELECT table_name  
FROM user_tables;
```

TABLE_NAME
JOB_GRADES
REGIONS
COUNTRIES
LOCATIONS
DEPARTMENTS
...

# Column Information

## USER\_TAB\_COLUMNS:

```
DESCRIBE user_tab_columns
```

Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME	NOT NULL	VARCHAR2(30)
DATA_TYPE		VARCHAR2(106)
DATA_TYPE_MOD		VARCHAR2(3)
DATA_TYPE_OWNER		VARCHAR2(30)
DATA_LENGTH	NOT NULL	NUMBER
DATA_PRECISION		NUMBER
DATA_SCALE		NUMBER
NULLABLE		VARCHAR2(1)
COLUMN_ID		NUMBER
DEFAULT_LENGTH		NUMBER
DATA_DEFAULT		LONG

...

# Column Information

```
SELECT column_name, data_type, data_length,  
       data_precision, data_scale, nullable  
  FROM user_tab_columns  
 WHERE table_name = 'EMPLOYEES';
```

COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION	DATA_SCALE	NUL
EMPLOYEE_ID	NUMBER	22	6	0	N
FIRST_NAME	VARCHAR2	20			Y
LAST_NAME	VARCHAR2	25			N
EMAIL	VARCHAR2	25			N
PHONE_NUMBER	VARCHAR2	20			Y
HIRE_DATE	DATE	7			N
JOB_ID	VARCHAR2	10			N
SALARY	NUMBER	22	8	2	Y
COMMISSION_PCT	NUMBER	22	2	2	Y
MANAGER_ID	NUMBER	22	6	0	Y
DEPARTMENT_ID	NUMBER	22	4	0	Y

# Constraint Information

- **USER\_CONSTRAINTS** describes the constraint definitions on your tables.
- **USER\_CONS\_COLUMNS** describes columns that are owned by you and that are specified in constraints.

```
DESCRIBE user_constraints
```

Name	Null?	Type
OWNER	NOT NULL	VARCHAR2(30)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)
CONSTRAINT_TYPE		VARCHAR2(1)
TABLE_NAME	NOT NULL	VARCHAR2(30)
SEARCH_CONDITION		LONG
R_OWNER		VARCHAR2(30)
R_CONSTRAINT_NAME		VARCHAR2(30)
DELETE_RULE		VARCHAR2(9)
STATUS		VARCHAR2(8)
...		

# Constraint Information

```
SELECT constraint_name, constraint_type,  
       search_condition, r_constraint_name,  
       delete_rule, status  
FROM   user_constraints  
WHERE  table_name = 'EMPLOYEES';
```

CONSTRAINT_NAME	CON	SEARCH_CONDITION	R_CONSTRAINT_NAME	DELETE_RULE	STATUS
EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL			ENABLED
EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL			ENABLED
EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL			ENABLED
EMP_JOB_NN	C	"JOB_ID" IS NOT NULL			ENABLED
EMP_SALARY_MIN	C	salary > 0			ENABLED
EMP_EMAIL_UK	U				ENABLED
EMP_EMP_ID_PK	P				ENABLED
EMP_DEPT_FK	R		DEPT_ID_PK	NO ACTION	ENABLED
EMP_JOB_FK	R		JOB_ID_PK	NO ACTION	ENABLED
EMP_MANAGER_FK	R		EMP_EMP_ID_PK	NO ACTION	ENABLED

# Constraint Information

```
DESCRIBE user_cons_columns
```

Name	Null?	Type
OWNER	NOT NULL	VARCHAR2(30)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
POSITION		NUMBER

```
SELECT constraint_name, column_name  
FROM user_cons_columns  
WHERE table_name = 'EMPLOYEES' ;
```

CONSTRAINT_NAME	COLUMN_NAME
EMP_EMAIL_UK	EMAIL
EMP_SALARY_MIN	SALARY
EMP_JOB_NN	JOB_ID
EMP_HIRE_DATE_NN	HIRE_DATE

...

ORACLE®

# View Information

1

**DESCRIBE user\_views**

Name	Null?	Type
VIEW_NAME	NOT NULL	VARCHAR2(30)
TEXT_LENGTH		NUMBER
TEXT		LONG

2

**SELECT DISTINCT view\_name FROM user\_views;**

VIEW_NAME
EMP_DETAILS_VIEW

3

**SELECT text FROM user\_views  
WHERE view\_name = 'EMP DETAILS VIEW' ;**

TEXT
SELECT e.employee_id, e.job_id, e.manager_id, e.department_id, d.location_id, l.country_id, e.first_name, e.last_name, e.salary, e.commission_pct, d.department_name, j.job_title, l.city, l.state_province, c.country_name, r.region_name FROM employees e, departments d, jobs j, locations l, countries c, regions r WHERE e.department_id = d.department_id AND d.location_id = l.location_id AND l.country_id = c.country_id AND c.region_id = r.region_id AND j.job_id = e.job_id WITH READ ONLY

# Sequence Information

```
DESCRIBE user_sequences
```

Name	Null?	Type
SEQUENCE_NAME	NOT NULL	VARCHAR2(30)
MIN_VALUE		NUMBER
MAX_VALUE		NUMBER
INCREMENT_BY	NOT NULL	NUMBER
CYCLE_FLAG		VARCHAR2(1)
ORDER_FLAG		VARCHAR2(1)
CACHE_SIZE	NOT NULL	NUMBER
LAST_NUMBER	NOT NULL	NUMBER

# Sequence Information

- Verify your sequence values in the USER\_SEQUENCES data dictionary table.

```
SELECT    sequence_name, min_value, max_value,  
          increment_by, last_number  
FROM      user_sequences;
```

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
LOCATIONS_SEQ	1	9900	100	3300
DEPARTMENTS_SEQ	1	9990	10	280
EMPLOYEES_SEQ	1	1.0000E+27	1	207

- The LAST\_NUMBER column displays the next available sequence number if NOCACHE is specified.



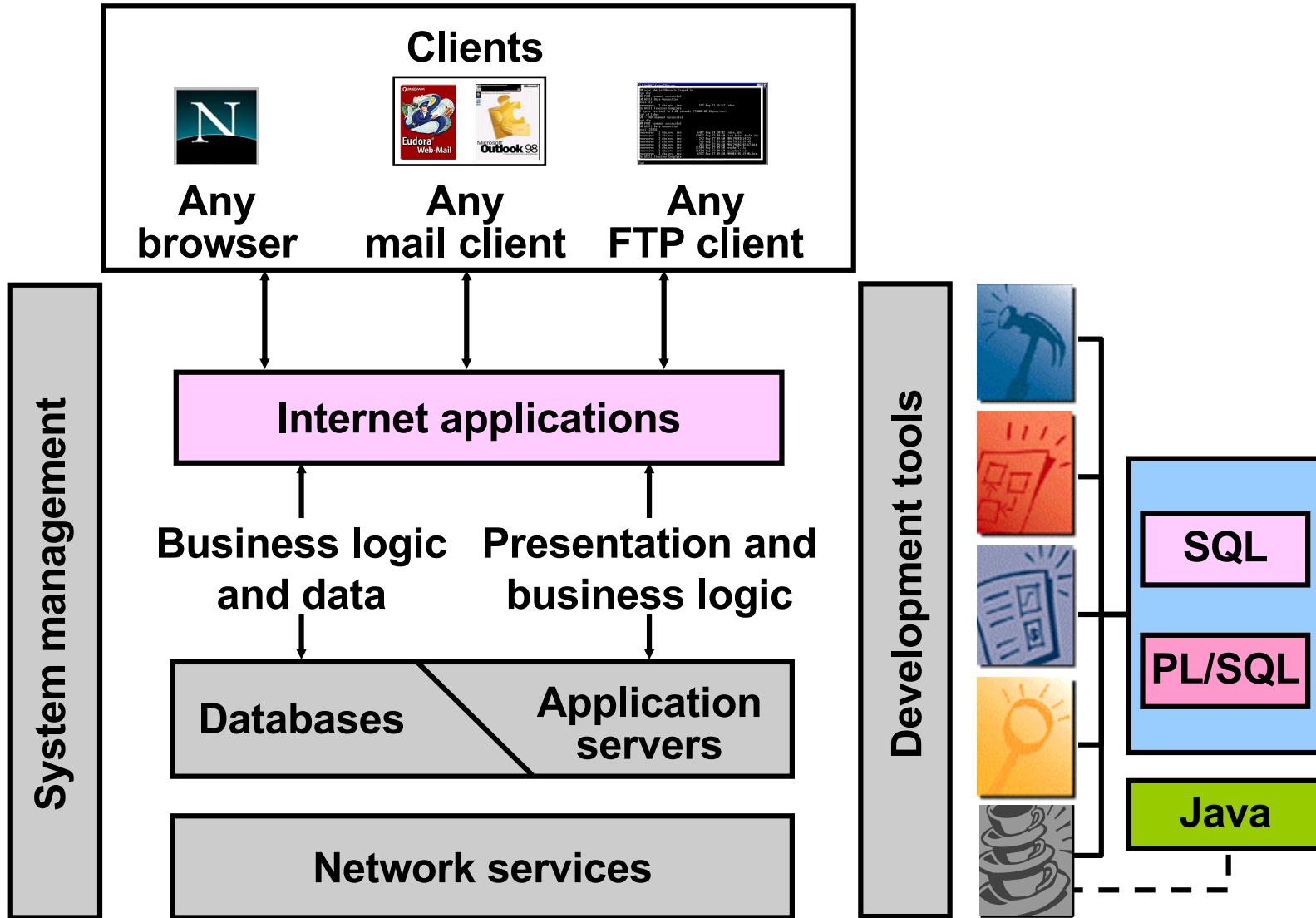
**Base de données II**  
**Concepts et programmation procédurale**

**Mise en oeuvre avec PL/SQL**  
**Présenté par Pr. B. El Asri**

# Objectifs

- Comprendre ce que PL/SQL fournit comme extensions de programmation à SQL
- Écrire du code PL/SQL en interface avec la base de données
- Concevoir des blocks PL/SQL qui s'exécutent efficacement
- Utiliser des constructions PL/SQL pour le traitement conditionnel et de boucle
- Gérer les erreurs d'exécution
- Décrire les fonctions et procédures stockées

# Oracle Internet Platform



# Généralités

## Procedural language for SQL

Oracle : PL/SQL

PostgreSQL : PL/pgSQL

SQL Server : Transact-SQL

DB2 : SQLPL

MySQL PL/mysql depuis 5.0

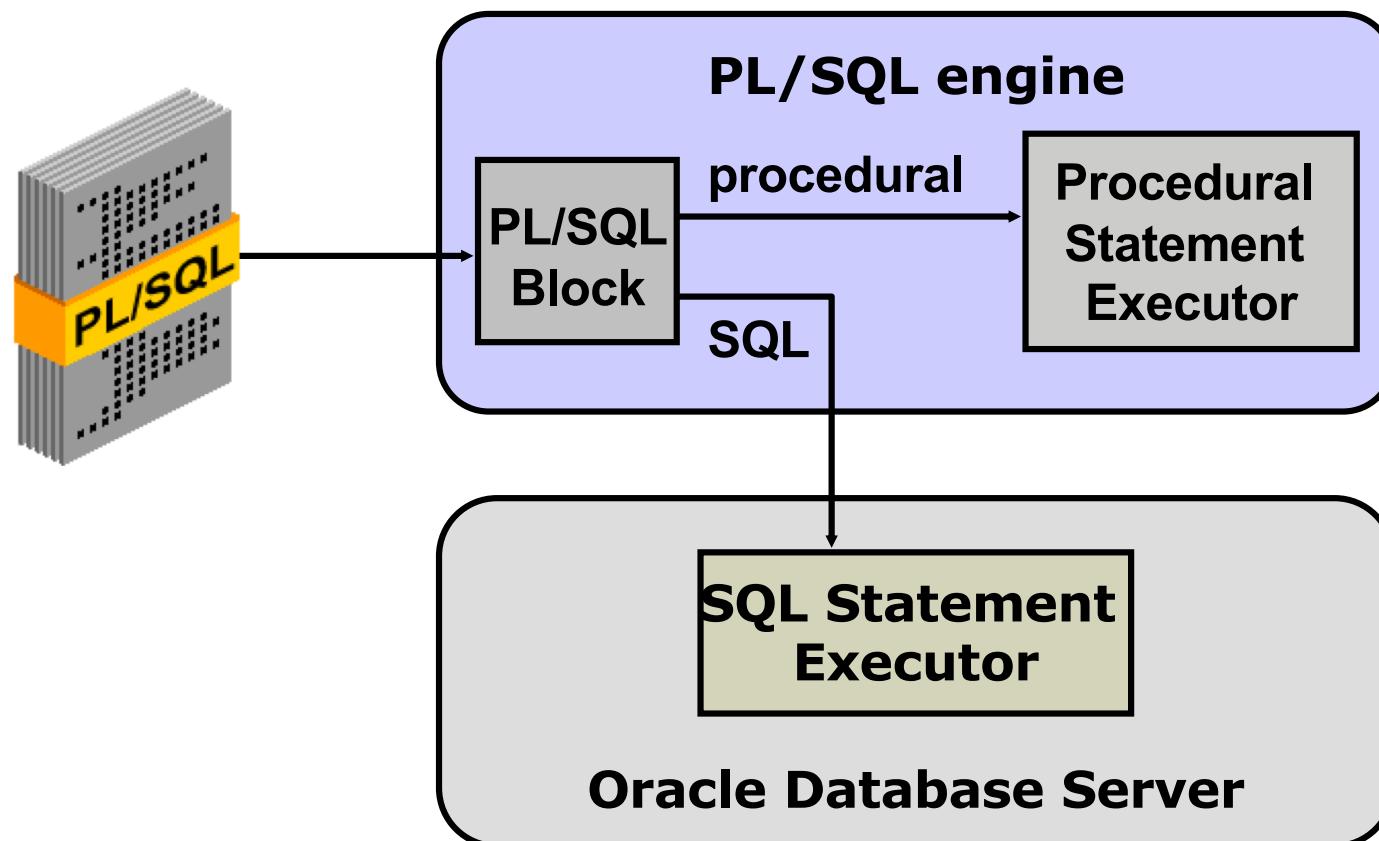
# Généralités

## PL/SQL : Langage procédural

- Extension de SQL
- Déclaration de variables et de constantes
- Définition de sous-programmes
- Gestion des erreurs à l'exécution (exceptions)
- Manipulation de données avec SQL

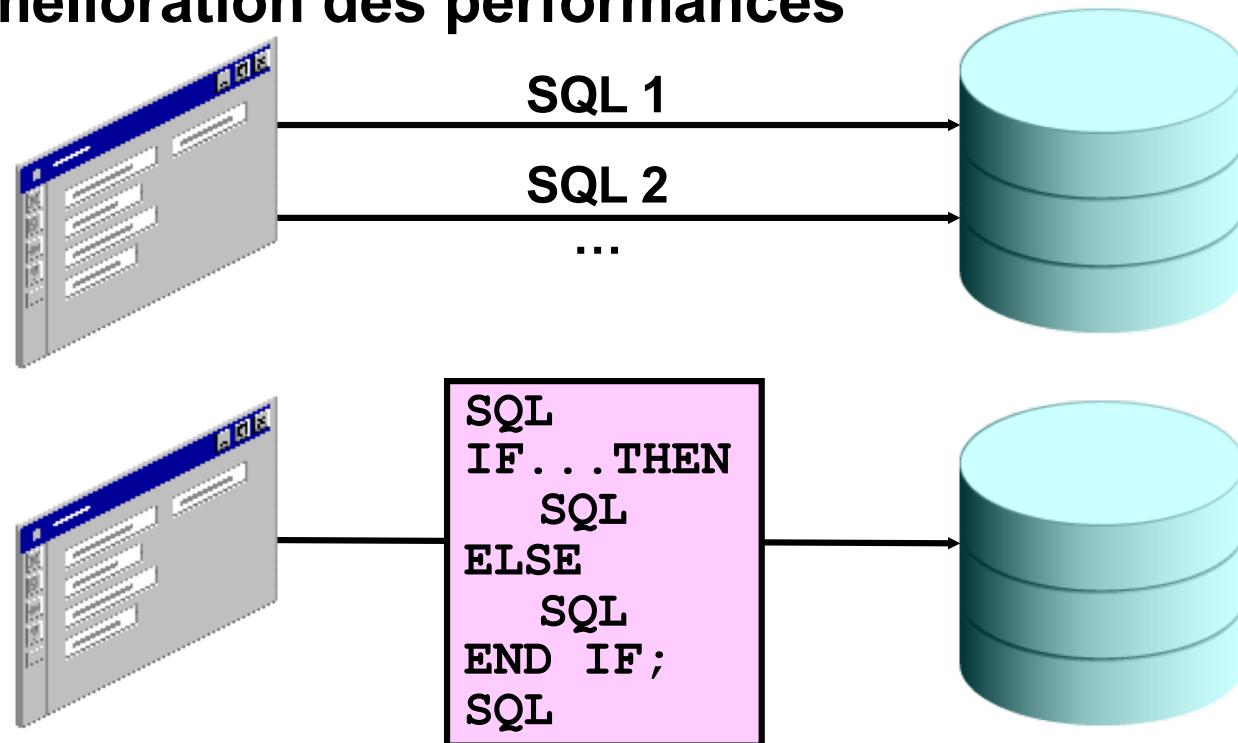


# L'environnement PL/SQL



# Les avantages de PL/SQL

- Intégration des constructions procédurales avec SQL
- Amélioration des performances



# PL/SQL: Structure d'un Block

- **DECLARE (optional)**
  - Variables, cursors, user-defined exceptions
- **BEGIN (mandatory)**
  - SQL statements
  - PL/SQL statements
- **EXCEPTION (optional)**
  - Actions to perform when errors occur
- **END; (mandatory)**



# Block Types

## Anonymous

```
[DECLARE]  
  
BEGIN  
    --statements  
  
[EXCEPTION]  
  
END ;
```

## Procedure

```
PROCEDURE name  
IS  
BEGIN  
    --statements  
  
[EXCEPTION]  
  
END ;
```

## Function

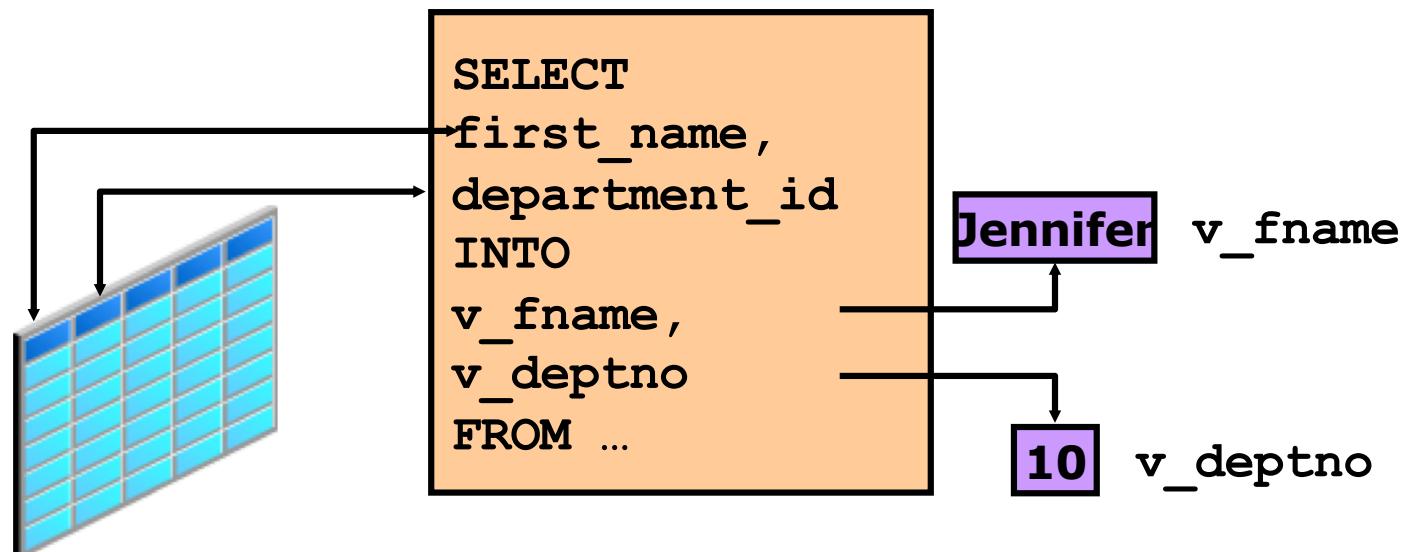
```
FUNCTION name  
RETURN datatype  
IS  
BEGIN  
    --statements  
    RETURN value;  
[EXCEPTION]  
  
END ;
```

# **LES VARIABLES PL/SQL**

# Les Variables

Les Variables peuvent être utilisées pour :

- le stockage temporaire des données
- la manipulation des valeurs stockées
- la réutilisabilité



# Les variables

PL/SQL gère deux types de variables

- Les variables PL/SQL
  - Scalar
  - Composite
  - Reference
  - Large object (LOB)
- Les variables non PL/SQL
  - Les variables champs écrans FORMS
  - Les variables de lien (“ bind ” variables -variables SQL).
  - Les variables du langage hôte dans les langages PRO.
  - Elles sont toujours préfixées de ':' lors de leur utilisation.
  - Les variables PL/SQL déclarées dans les packages sont toujours préfixées du nom du package lors de leur utilisation.

# Declarer et Initialiser des Variables PL/SQL

## Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]  
[ := | DEFAULT expr] ;
```

## Exemples:

```
DECLARE  
    v_hiredate      DATE;  
    v_deptno        NUMBER(2) NOT NULL := 10;  
    v_location       VARCHAR2(13) := 'Atlanta';  
    c_comm           CONSTANT NUMBER := 1400;
```

# Declarer et Initialiser des Variables PL/SQL

1

```
DECLARE
    v_myName VARCHAR2(20);
BEGIN
    DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myName);
    v_myName := 'John';
    DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myName);
END;
/
```

2

```
DECLARE
    v_myName VARCHAR2(20) := 'John';
BEGIN
    v_myName := 'Steven';
    DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myName);
END;
/
```

# Comment déclarer et initialiser des Variables PL/SQL

- Suivre les conventions d'affectation de noms.
- Utiliser des identificateurs significatifs pour les variables.
- Initialiser des variables désignés comme non NULL et constante.
- Initialiser des variables avec l'opérateur d'assignation (`:=`) ou le mot clé `DEFAULT` :

```
v_myName VARCHAR2 (20) := 'John' ;
```

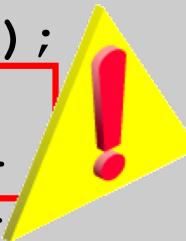
```
v_myName VARCHAR2 (20) DEFAULT 'John' ;
```

- Déclarer chaque identificateur sur une ligne à part pour faciliter la maintenance de code et sa lisibilité.

# Comment déclarer et initialiser des Variables PL/SQL

- Évitez d'utiliser des noms de colonne en tant qu'identificateurs.

```
DECLARE
    employee_id NUMBER(6);
BEGIN
    SELECT      employee_id
    INTO        employee_id
    FROM        employees
    WHERE       last_name = 'Kochhar';
END;
/
```



- Utiliser la contrainte NOT NULL lorsque la variable doit recevoir une valeur.

# Types basiques

- **CHAR [ (maximum\_length) ]**
- **VARCHAR2 (maximum\_length)**
- **NUMBER [ (precision, scale) ]**
- **BINARY\_INTEGER**
- **PLS\_INTEGER**
- **BOOLEAN**
- **BINARY\_FLOAT**
- **BINARY\_DOUBLE**

# L' Attribut %TYPE

- **Est utilisé pour déclarer une variable selon :**
  - une définition de colonne de base de données
  - le type d'une variable déjà déclaré
- **Est préfixé par:**
  - le nom de la table et de la colonne
  - le nom de la variable déjà déclaré

# L' Attribut %TYPE

## Syntaxe

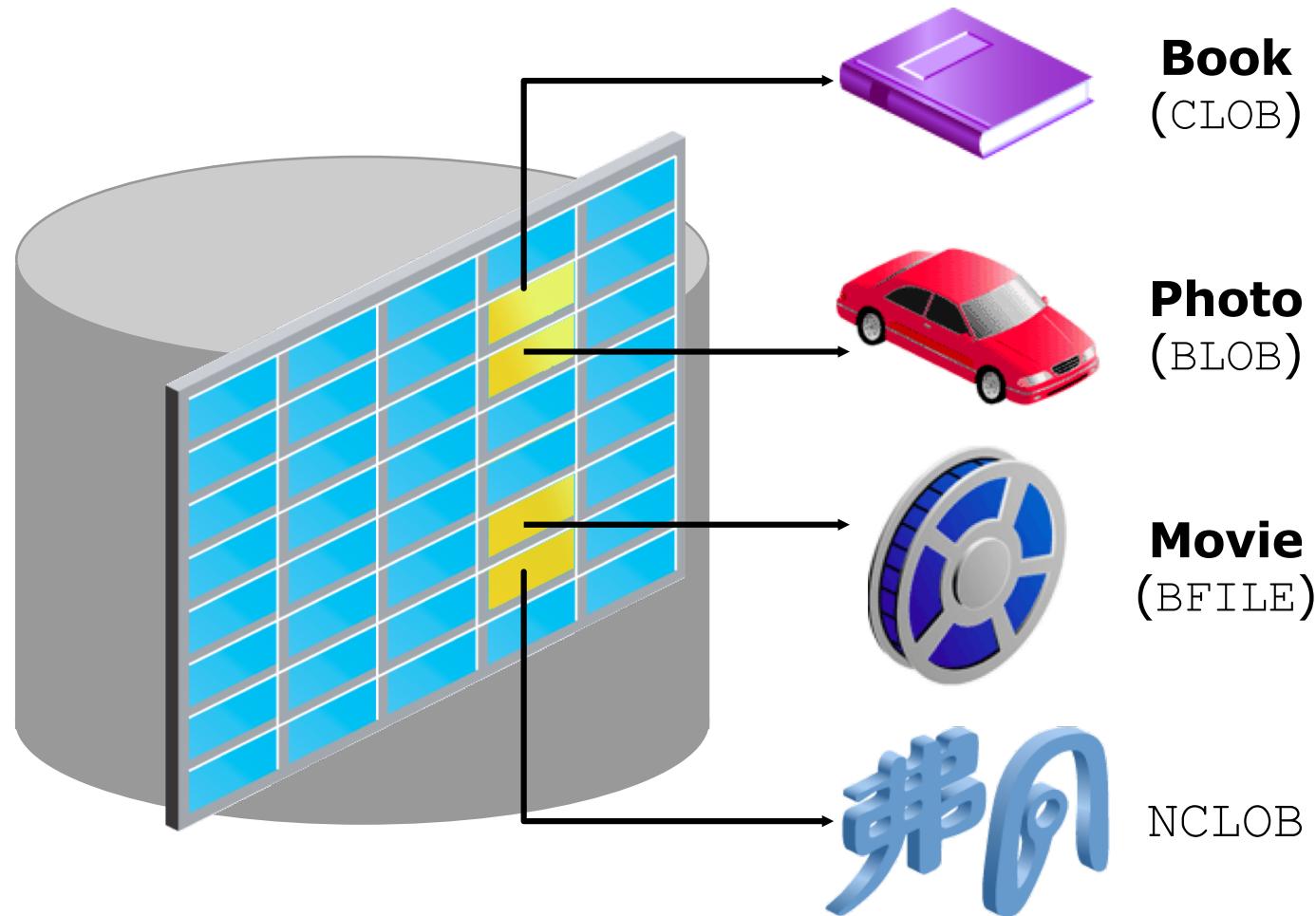
```
identifier      table.column_name%TYPE;
```

## Exemples

```
...  
emp_lname      employees.last_name%TYPE;  
...
```

```
...  
balance        NUMBER(7,2);  
min_balance    balance%TYPE := 1000;  
...
```

# LOB Data Type Variables



# Composite Data Types

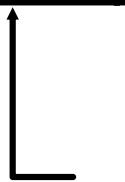
TRUE	23-DEC-98	ATLANTA	
------	-----------	---------	---

**PL/SQL table structure**

1	SMITH
2	JONES
3	NANCY
4	TIM

**PL/SQL table structure**

1	5000
2	2345
3	12
4	3456



VARCHAR2

PLS\_INTEGER



NUMBER

PLS\_INTEGER

# Composite Data Types

- Peut contenir plusieurs valeurs
- Peut être de deux types:
  - PL/SQL records
  - PL/SQL collections
    - INDEX BY tables or associative arrays
    - Nested table
    - VARRAY

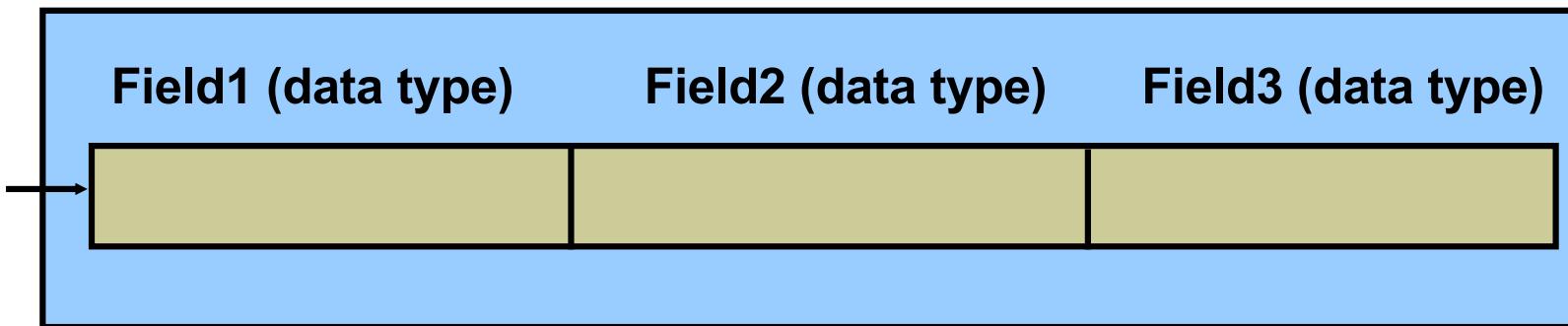
## %ROWTYPE Attribute

- Déclare une variable selon une collection de colonnes dans une table de base de données ou une vue.
- Préfixer %ROWTYPE avec la table de base de données ou la vue.
- Les champs dans l'enregistrement prennent les noms et les types de données des colonnes de la table ou la vue.

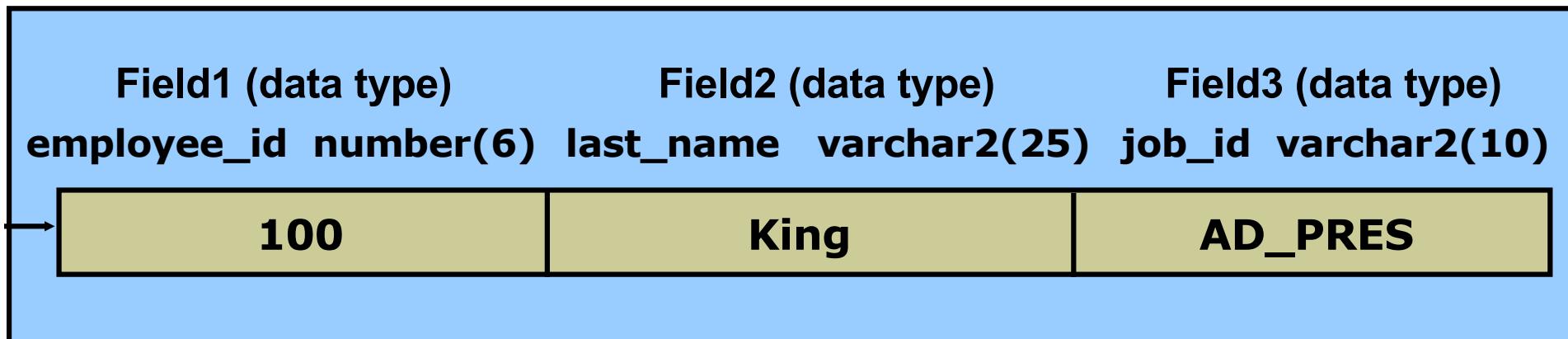
Syntaxe:

```
DECLARE  
    identifier reference%ROWTYPE;
```

# PL/SQL Record Structure



## Example:



# Créer un PL/SQL Record

## Syntaxe:

1

```
TYPE type_name IS RECORD  
      (field_declaration[, field_declaration]...);
```

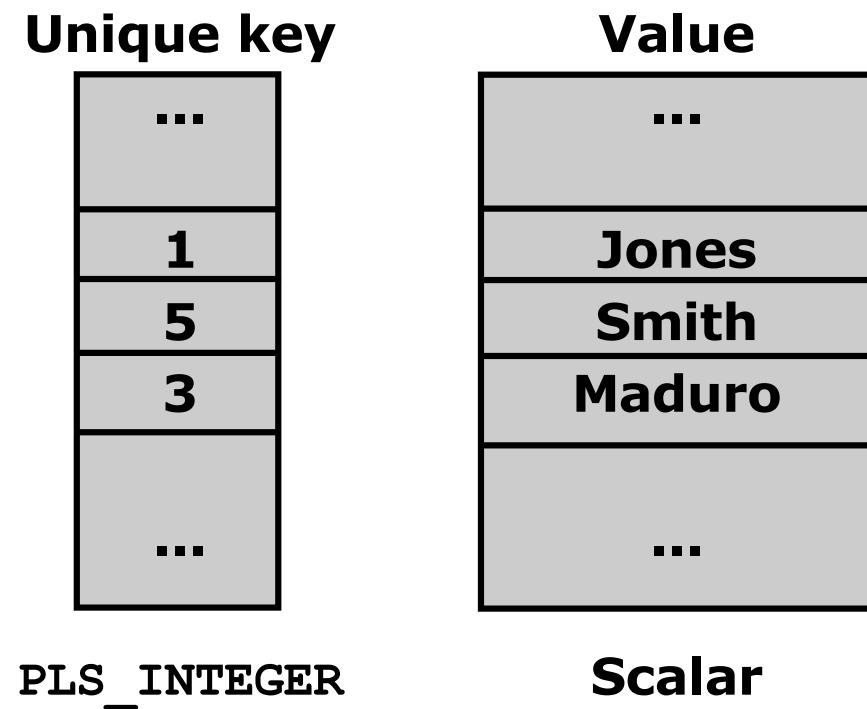
2

```
identifier type_name;
```

*field\_declaration*:

```
field_name {field_type | variable%TYPE  
           | table.column%TYPE | table%ROWTYPE}  
    [ [NOT NULL] { := | DEFAULT } expr]
```

# INDEX BY Table Structure



# Création d'un INDEX BY Table

```
DECLARE
    TYPE ename_table_type IS TABLE OF
        employees.last_name%TYPE
        INDEX BY PLS_INTEGER;
    TYPE hiredate_table_type IS TABLE OF DATE
        INDEX BY PLS_INTEGER;
    ename_table      ename_table_type;
    hiredate_table   hiredate_table_type;
BEGIN
    ename_table(1)      := 'CAMERON';
    hiredate_table(8)   := SYSDATE + 7;
    IF ename_table.EXISTS(1) THEN
        INSERT INTO ...
        ...
END;
/
```

	ENAME	HIREDT
1	CAMERON	23-FEB-09

ORACLE®

# **ECRIRE DES BLOCKS PL/SQL**

# Commenter le Code

- Préfixer les commentaires monolignes avec deux traits d'Union (--).
- Placer les commentaires de plusieurs lignes entre les symboles /\* et \*/.

Exemple:

```
DECLARE
  ...
  v_annual_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the
     monthly salary input from the user */
  v_annual_sal := monthly_sal * 12;
  --The following line displays the annual salary
  DBMS_OUTPUT.PUT_LINE(v_annual_sal);
END;
/
```

# Utiliser les Functions SQL dans le code PL/SQL: Examples

- Obtenir la longueur d'une chaîne :

```
v_desc_size INTEGER(5);  
v_prod_description VARCHAR2(70) := 'You can use this  
product with your radios for higher frequency';  
  
-- get the length of the string in prod_description  
v_desc_size := LENGTH(v_prod_description);
```

- Obtenir le nombre de mois, que l'employé a travaillé :

```
v_tenure := MONTHS_BETWEEN (CURRENT_DATE, v_hiredate);
```

# La Conversion de type

- Convertit les données en types de données comparables
- Peut être de deux types:
  - Implicit conversion
  - Explicit conversion
- Des Fonctions:
  - TO\_CHAR
  - TO\_DATE
  - TO\_NUMBER
  - TO\_TIMESTAMP

# Conversion de Type

1

```
date_of_joining DATE := '02-Feb-2000';
```

2

```
date_of_joining DATE := 'February 02,2000';
```

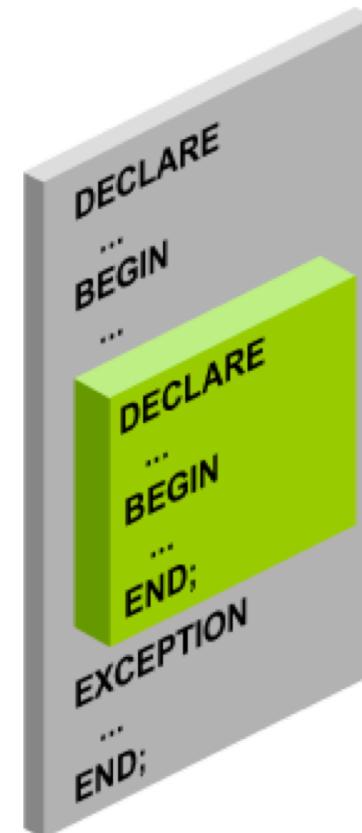
3

```
date_of_joining DATE := TO_DATE ('February  
02,2000','Month DD, YYYY');
```

# Blocks Imbriqués

Les blocs PL/SQL peuvent être imbriqués.

- Une section exécutable (BEGIN ... END) peut contenir des blocs imbriqués.
- Une section d'exception peut contenir des blocs imbriqués.



# Nested Blocks: Example

```
DECLARE
    v_outer_variable VARCHAR2(20) := 'GLOBAL VARIABLE';
BEGIN
    DECLARE
        v_inner_variable VARCHAR2(20) := 'LOCAL VARIABLE';
    BEGIN
        DBMS_OUTPUT.PUT_LINE(v_inner_variable);
        DBMS_OUTPUT.PUT_LINE(v_outer_variable);
    END;
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

anonymous block completed  
LOCAL VARIABLE  
GLOBAL VARIABLE  
GLOBAL VARIABLE

# Visibilité et Porté des Variables

```
DECLARE
    v_father_name VARCHAR2(20) :='Patrick';
    v_date_of_birth DATE:='20-Apr-1972';
BEGIN
    DECLARE
        v_child_name VARCHAR2(20) :='Mike';
        v_date_of_birth DATE:='12-Dec-2002';
    BEGIN
1        DBMS_OUTPUT.PUT_LINE('Father''s Name: '||v_father_name);
        DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
        DBMS_OUTPUT.PUT_LINE('Child''s Name: '||v_child_name);
    END;
2        DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
    END;
/
```

# Visibilité et Porté des Variables

```
BEGIN <<outer>>
DECLARE
    v_father_name VARCHAR2(20) := 'Patrick';
    v_date_of_birth DATE := '20-Apr-1972';
BEGIN
    DECLARE
        v_child_name VARCHAR2(20) := 'Mike';
        v_date_of_birth DATE := '12-Dec-2002';
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Father''s Name: ' || v_father_name);
        DBMS_OUTPUT.PUT_LINE('Date of Birth: '
                            || outer.v_date_of_birth);
        DBMS_OUTPUT.PUT_LINE('Child''s Name: ' || v_child_name);
        DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
    END;
END;
END outer;
```

# Visibilité et Porté des Variables: Exemple

```
BEGIN <>outer>>
DECLARE
    v_sal      NUMBER(7,2) := 60000;
    v_comm     NUMBER(7,2) := v_sal * 0.20;
    v_message  VARCHAR2(255) := ' eligible for commission';
BEGIN
    DECLARE
        v_sal      NUMBER(7,2) := 50000;
        v_comm     NUMBER(7,2) := 0;
        v_total_comp  NUMBER(7,2) := v_sal + v_comm;
    BEGIN
        v_message := 'CLERK not' || v_message;
        outer.v_comm := v_sal * 0.30;
    END;
    v_message := ' SALESMAN' || v_message;
END;
END outer;
/
```

The diagram illustrates variable visibility across different scopes. A purple circle labeled '1' points to the assignment statement `outer.v_comm := v_sal * 0.30;`, which is visible because `v_comm` is declared at the same level as the assignment. A purple circle labeled '2' points to the assignment statement `v_message := ' SALESMAN' || v_message;`, which is also visible because `v_message` is declared at the same level.

# Exercice

```
DECLARE
  v_weight      NUMBER(3) := 600;
  v_message     VARCHAR2(255) := 'Product 10012';
BEGIN
  DECLARE
    v_weight  NUMBER(3) := 1;
    v_message VARCHAR2(255) := 'Product 11001';
    v_new_locn VARCHAR2(50) := 'Europe';
  BEGIN
    v_weight := v_weight + 1;           v_weight at position 1 =?
    v_new_locn := 'Western '||v_new_locn;  v_new_locn at position 1 = ?
  1  →
    END;
    v_weight := v_weight + 1;           v_weight at position 2 =?
    v_message := v_message || ' is in stock';
    v_new_locn := 'Western ' || v_new_locn; v_message at position 2 =?
  2  →
    END;
/
  v_new_locn at position 2 =?
```

# **Interaction avec le serveur de base de données Oracle**

# SELECT dans PL/SQL

Récupérer des données de la base de données avec une instruction SELECT.

Syntaxe :

```
SELECT  select_list
INTO    {variable_name[, variable_name] ...
        | record_name}
FROM    table
[WHERE  condition] ;
```

# SELECT dans PL/SQL

- La clause INTO est requise.
- Les requêtes ne doivent retourner qu'une seule ligne.

```
DECLARE
    v_fname VARCHAR2 (25);
BEGIN
    SELECT first_name INTO v_fname
    FROM employees WHERE employee_id=200;
    DBMS_OUTPUT.PUT_LINE(' First Name is : ' || v_fname);
END;
/
```

```
anonymous block completed
First Name is : Jennifer
```

# Récupération de données par PL/SQL: Exemple

Recuperer `hire_date` et `salary` pour un employé.

```
DECLARE
    v_emp_hiredate    employees.hire_date%TYPE;
    v_emp_salary       employees.salary%TYPE;
BEGIN
    SELECT    hire_date, salary
    INTO      v_emp_hiredate, v_emp_salary
    FROM     employees
    WHERE    employee_id = 100;
END;
/
```

# Récupération de données par PL/SQL

**Retourne la somme des salaires pour tous les employés d'un département spécifié.**

**Exemple:**

```
DECLARE
    v_sum_sal    NUMBER(10,2);
    v_deptno     NUMBER NOT NULL := 60;
BEGIN
    SELECT SUM(salary) -- group function
    INTO v_sum_sal
    FROM employees
    WHERE department_id = v_deptno;
    DBMS_OUTPUT.PUT_LINE ('The sum of salary is ' || v_sum_sal);
END;
```

```
anonymous block completed
The sum of salary is 28800
```

# Récupérer des données : Exemple

Déclarer des variables pour stocker le nom, l'emploi et le salaire d'un nouvel employé.

```
DECLARE
    type t_rec is record
        (v_sal number(8),
         v_minsal number(8) default 1000,
         v_hire_date employees.hire_date%type,
         v_rec1 employees%rowtype);
        v_myrec t_rec;
BEGIN
    v_myrec.v_sal := v_myrec.v_minsal + 500;
    v_myrec.v_hire_date := sysdate;
    SELECT * INTO v_myrec.v_rec1
        FROM employees WHERE employee_id = 100;
    DBMS_OUTPUT.PUT_LINE(v_myrec.v_rec1.last_name || ' ' ||
        to_char(v_myrec.v_hire_date) || ' '|| to_char(v_myrec.v_sal));
END;
```

anonymous block completed  
King 16-FEB-09 1500

ORACLE®

# Récupérer des données pour un %ROWTYPE Attribute: Exemple

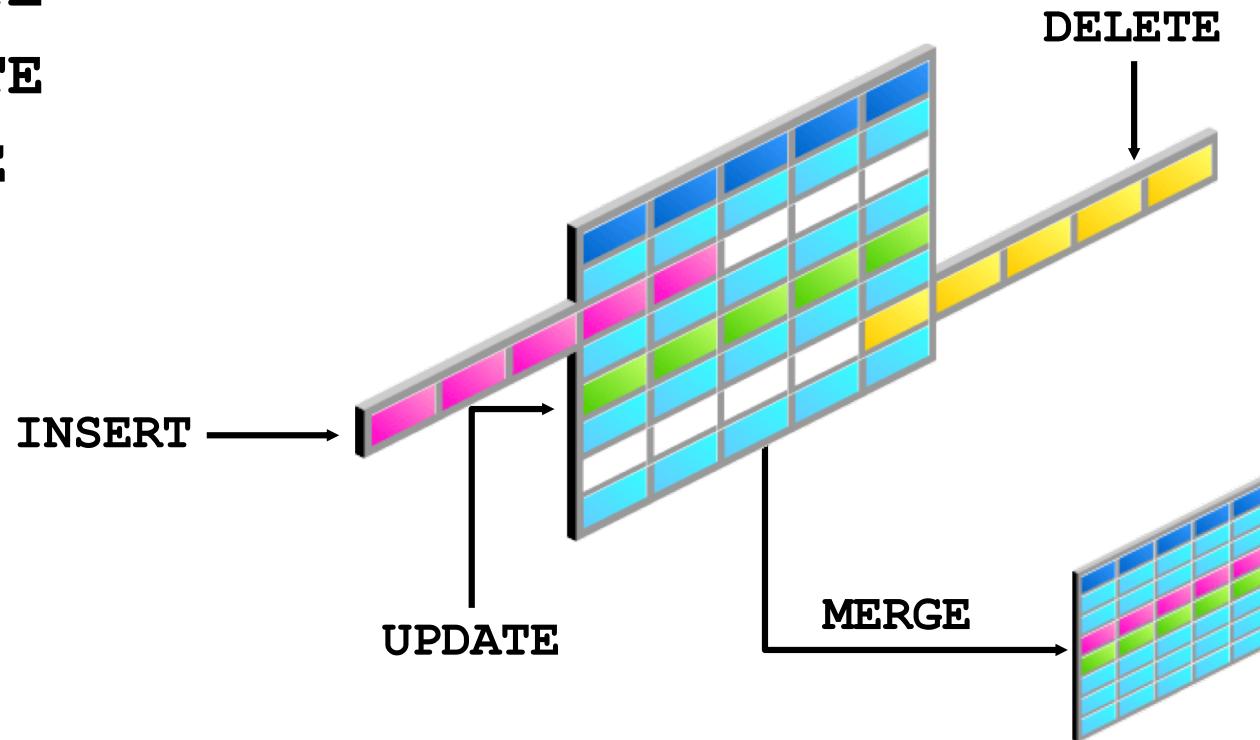
```
DECLARE
    v_employee_number number:= 124;
    v_emp_rec    employees%ROWTYPE;
BEGIN
    SELECT * INTO v_emp_rec FROM employees
    WHERE employee_id = v_employee_number;
    INSERT INTO retired_emps(empno, ename, job, mgr,
                           hiredate, leavedate, sal, comm, deptno)
    VALUES (v_emp_rec.employee_id, v_emp_rec.last_name,
            v_emp_rec.job_id, v_emp_rec.manager_id,
            v_emp_rec.hire_date, SYSDATE,
            v_emp_rec.salary, v_emp_rec.commission_pct,
            v_emp_rec.department_id);
END;
/
```

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-FEB-09	5800	(null)	50

# Utiliser PL/SQL pour Manipuler des Données

Apporter des modifications aux tables de base de données à l'aide de commandes DML :

- **INSERT**
- **UPDATE**
- **DELETE**
- **MERGE**



# Inserer des données : Exemple

```
BEGIN
  INSERT INTO employees
  (employee_id, first_name, last_name, email,
  hire_date, job_id, salary)
  VALUES(employees_seq.NEXTVAL, 'Ruth', 'Cores',
  'RCORES',CURRENT_DATE, 'AD_ASST', 4000);
END;
/
```

# Insérer un enregistrement en utilisant %ROWTYPE

```
...
DECLARE
    v_employee_number number:= 124;
    v_emp_rec retired_emps%ROWTYPE;
BEGIN
    SELECT employee_id, last_name, job_id, manager_id,
    hire_date, hire_date, salary, commission_pct,
    department_id INTO v_emp_rec FROM employees
    WHERE employee_id = v_employee_number;
    INSERT INTO retired_emps VALUES v_emp_rec;
END;
/
SELECT * FROM retired_emps;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124 Mourgos	ST_MAN	100	16-NOV-99	16-NOV-99	5800	(null)	50

# Mettre à jour des données: Exemple

```
DECLARE
    sal_increase      employees.salary%TYPE := 800;
BEGIN
    UPDATE      employees
    SET          salary = salary + sal_increase
    WHERE        job_id = 'ST_CLERK';
END ;
/
```

```
anonymous block completed
FIRST_NAME      SALARY
-----
Julia           4000
Irene           3500
James           3200
Steven          3000
```

```
...
Curtis          3900
Randall         3400
Peter           3300
20 rows selected
```

ORACLE®

# Updating a Row in a Table by Using a Record

```
SET VERIFY OFF
DECLARE
    v_employee_number number:= 124;
    v_emp_rec retired_emps%ROWTYPE;
BEGIN
    SELECT * INTO v_emp_rec FROM retired_emps
    where empno = v_employee_number;
    v_emp_rec.leavedate:=CURRENT_DATE;
    UPDATE retired_emps SET ROW = v_emp_rec WHERE
        empno=v_employee_number;
END ;
/
SELECT * FROM retired_emps;
```

#	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-FEB-09	5800	(null)	50

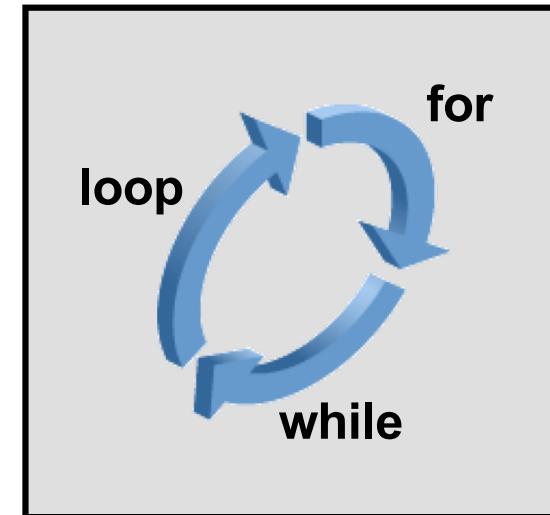
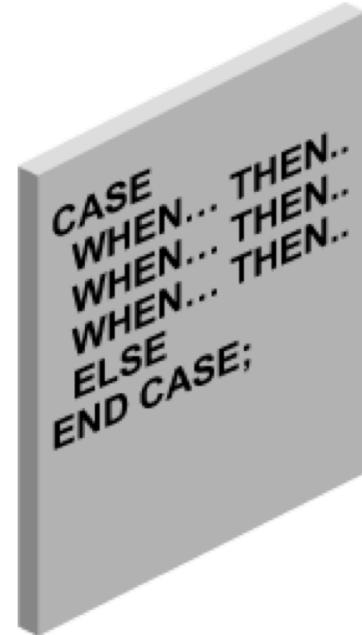
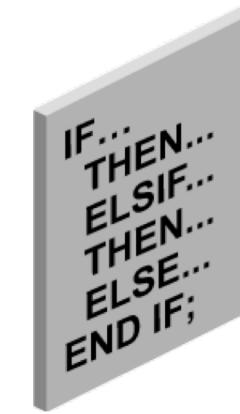
ORACLE®

# Supprimer des données: Exemple

```
DECLARE
    deptno    employees.department_id%TYPE := 10;
BEGIN
    DELETE FROM employees
    WHERE department_id = deptno;
END ;
/
```

# **STRUCTURES DE CONTRÔLE**

# Structures de contrôle et boucle



# IF Statement

## Syntax:

```
IF condition THEN  
  statements;  
[ELSIF condition THEN  
  statements;]  
[ELSE  
  statements;]  
END IF;
```

# Simple IF Statement

```
DECLARE
    v_myage  number:=31;
BEGIN
    IF v_myage  < 11
    THEN
        DBMS_OUTPUT.PUT_LINE(' I am a child ');
    END IF;
END ;
/
```

anonymous block completed

# IF THEN ELSE Statement

```
DECLARE
v_myage  number:=31;
BEGIN
IF v_myage  < 11
THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
END IF;
END;
/
```

anonymous block completed  
I am not a child

# IF ELSIF ELSE Clause

```
DECLARE
    v_myage number:=31;
BEGIN
    IF v_myage < 11 THEN
        DBMS_OUTPUT.PUT_LINE(' I am a child ');
    ELSIF v_myage < 20 THEN
        DBMS_OUTPUT.PUT_LINE(' I am young ');
    ELSIF v_myage < 30 THEN
        DBMS_OUTPUT.PUT_LINE(' I am in my twenties');
    ELSIF v_myage < 40 THEN
        DBMS_OUTPUT.PUT_LINE(' I am in my thirties');
    ELSE
        DBMS_OUTPUT.PUT_LINE(' I am always young ');
    END IF;
END;
/
```

anonymous block completed  
I am in my thirties

# NULL Value in IF Statement

```
DECLARE
    v_myage  number;
BEGIN
    IF v_myage  < 11 THEN
        DBMS_OUTPUT.PUT_LINE(' I am a child ');
    ELSE
        DBMS_OUTPUT.PUT_LINE(' I am not a child ');
    END IF;
END;
/
```

```
anonymous block completed
I am not a child
```

# CASE Expressions

- Une expression CASE sélectionne un résultat et le retourne.
- Pour sélectionner le résultat, l'expression CASE utilise des expressions. La valeur renvoyée par ces expressions est utilisée pour sélectionner une ou plusieurs alternatives.

```
CASE selector
  WHEN expression1 THEN result1
  WHEN expression2 THEN result2
  ...
  WHEN expressionN THEN resultN
  [ELSE resultN+1]
END;
/
```

# CASE Expressions: Example

```
SET VERIFY OFF
DECLARE
    v_grade CHAR(1) := UPPER('&grade');
    appraisal VARCHAR2(20);
BEGIN
    appraisal := CASE v_grade
        WHEN 'A' THEN 'Excellent'
        WHEN 'B' THEN 'Very Good'
        WHEN 'C' THEN 'Good'
        ELSE 'No such grade'
    END;
    DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade || '
                           Appraisal '|| appraisal);
END;
/
```

# Searched CASE Expressions

```
DECLARE
    v_grade  CHAR(1) := UPPER('&grade');
    appraisal VARCHAR2(20);
BEGIN
    appraisal := CASE
        WHEN v_grade = 'A' THEN 'Excellent'
        WHEN v_grade IN ('B','C') THEN 'Good'
        ELSE 'No such grade'
    END;
    DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade || '
                           Appraisal ' || appraisal);
END;
/
```

# CASE Statement

```
DECLARE
    v_deptid NUMBER;
    v_deptname VARCHAR2(20);
    v_emps NUMBER;
    v_mngid NUMBER:= 108;
BEGIN
    CASE  v_mngid
        WHEN 108 THEN
            SELECT department_id, department_name
                INTO v_deptid, v_deptname FROM departments
                WHERE manager_id=108;
            SELECT count(*) INTO v_emps FROM employees
                WHERE department_id=v_deptid;
        WHEN 200 THEN
            ...
    END CASE;
    DBMS_OUTPUT.PUT_LINE ('You are working in the '|| deptname ||
        ' department. There are '||v_emps ||' employees in this
        department');
END;
/
```

# Exercice 1:

- **Ecrire un programme permettant**
  - de saisir en entrée le nom d 'un employé
  - de mettre à jour le salaire de cet employé en lui ajoutant :
    - **500 \$ s 'il appartient au département vente**
    - **300 \$ s 'il appartient au département opérateurs**
    - **400 \$ s 'il appartient au département SI**
    - **600 \$ s 'il appartient au département Administrateurs**

**NB:** Faites l'exercice avec

- **IF ..END IF**
- **CASE Expression**
- **CASE Statement**

## **Exercice 2:**

**Écrire un programme permettant  
de saisir en entrée le nom d 'un employé  
de mettre à jour le salaire d 'un employé suivant les conditions  
suivantes:**

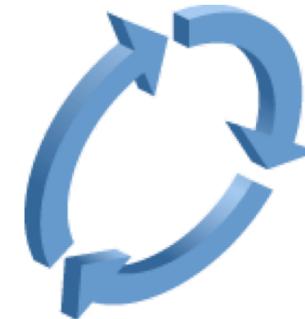
**Si son année d 'entrée est 1990, augmenter son salaire de 50%**

**Si son année d 'entrée est 1991, augmenter son salaire de 25%**

**Si son année d 'entrée est 1992, augmenter son salaire de 10%**

# Traitements itératifs : LOOP Statements

- **Une boucle répète une instruction (ou la séquence d'instructions) plusieurs fois.**
- **Trois types de boucle:**
  - **Basic loop**
  - **FOR loop**
  - **WHILE loop**



# Boucle de base

## Syntaxe:

```
LOOP  
  statement1;  
  . . .  
  EXIT [WHEN condition];  
END LOOP;
```

**Boucle de base qui permet la répétition d'une séquence d'instructions.**

# Basic Loops

## Example:

```
DECLARE
    v_countryid      locations.country_id%TYPE := 'CA';
    v_loc_id          locations.location_id%TYPE;
    v_counter         NUMBER(2) := 1;
    v_new_city        locations.city%TYPE := 'Montreal';
BEGIN
    SELECT MAX(location_id) INTO v_loc_id FROM locations
    WHERE country_id = v_countryid;
    LOOP
        INSERT INTO locations(location_id, city, country_id)
        VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 3;
    END LOOP;
END;
/
```

# La boucle WHILE

## Syntaxe:

```
WHILE condition LOOP  
    statement1;  
    statement2;  
    . . .  
END LOOP;
```

**La boucle WHILE répète les instructions tant que condition est TRUE.**

# WHILE Loops: Example

```
DECLARE
    v_countryid    locations.country_id%TYPE := 'CA';
    v_loc_id        locations.location_id%TYPE;
    v_new_city      locations.city%TYPE := 'Montreal';
    v_counter       NUMBER := 1;
BEGIN
    SELECT MAX(location_id) INTO v_loc_id FROM locations
    WHERE country_id = v_countryid;
    WHILE v_counter <= 3 LOOP
        INSERT INTO locations(location_id, city, country_id)
        VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
        v_counter := v_counter + 1;
    END LOOP;
END;
/
```

## La boucle FOR

- Le nombre d'itérations est connu avant d'entrer dans la boucle.
- Ne pas déclarer le compteur ; Il est déclaré implicitement.

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
    statement1;
    statement2;
    .
    .
    .
END LOOP;
```

# La boucle FOR : Exemple

```
DECLARE
    v_countryid    locations.country_id%TYPE := 'CA';
    v_loc_id        locations.location_id%TYPE;
    v_new_city      locations.city%TYPE := 'Montreal';
BEGIN
    SELECT MAX(location_id) INTO v_loc_id
        FROM locations
        WHERE country_id = v_countryid;
    FOR i IN 1..3 LOOP
        INSERT INTO locations(location_id, city, country_id)
        VALUES((v_loc_id + i), v_new_city, v_countryid );
    END LOOP;
END;
/
```

# Exercice 3

- 2.. Create a PL/SQL block that inserts an asterisk in the stars column for every \$1,000 of the employee's salary.**
- a. In the declarative section of the block,
    - declare a variable `v_empno` of type `emp.employee_id` and initialize it to 176.
    - declare a variable `v_asterisk` of type `emp.stars` and initialize it to `NULL`.
    - create a variable `sal` of type `emp.salary`.
  - b. In the executable section, write logic to append an asterisk (\*) to the string for every \$1,000 of the salary amount. For example, if the employee earns \$8,000, the string of asterisks should contain eight asterisks. If the employee earns \$12,500, the string of asterisks should contain 13 asterisks.
  - c. Update the `stars` column for the employee with the string of asterisks. Commit before the end of the block.

## **Exercice 4**

**Écrire un programme permettant la mise à jour des salaires de tous les employés suivant les conditions de l 'exercice2.**

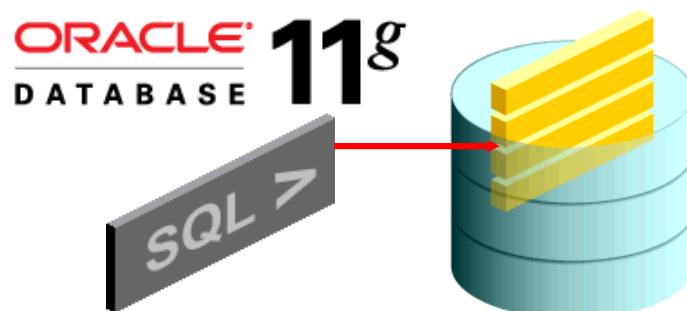
**Remarques ..**

# USING EXPLICIT CURSORS

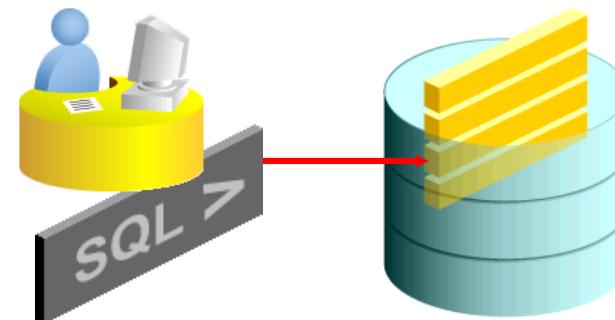
# Cursors

**Chaque instruction SQL exécutée par le serveur Oracle a un curseur individuel associé qui est soit un:**

- **curseur implicite** : Déclaré et géré par le PL/SQL pour toutes les instructions DML et PL/SQL SELECT
- **curseur explicite** : déclaré et géré par le programmeur

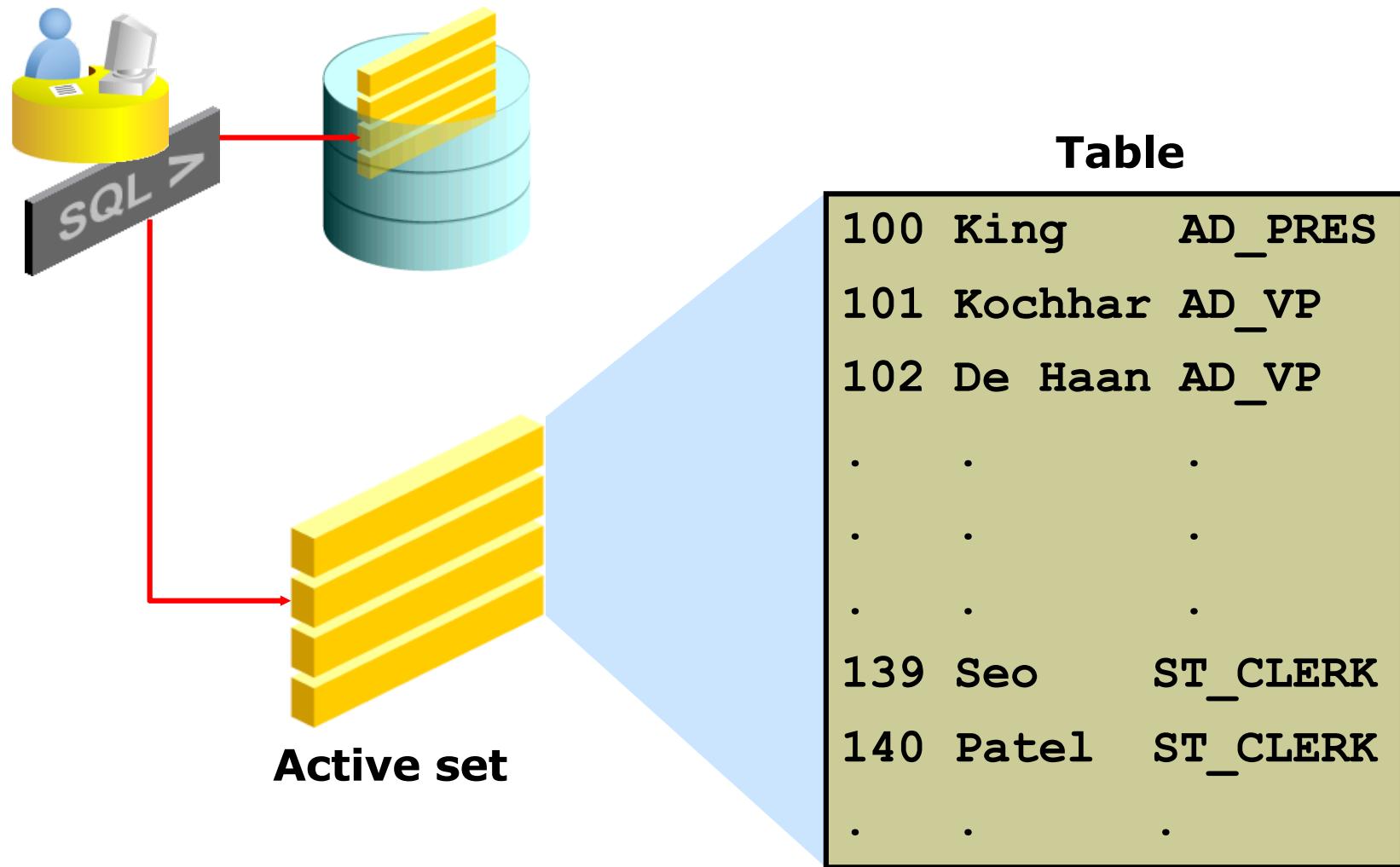


**Implicit cursor**

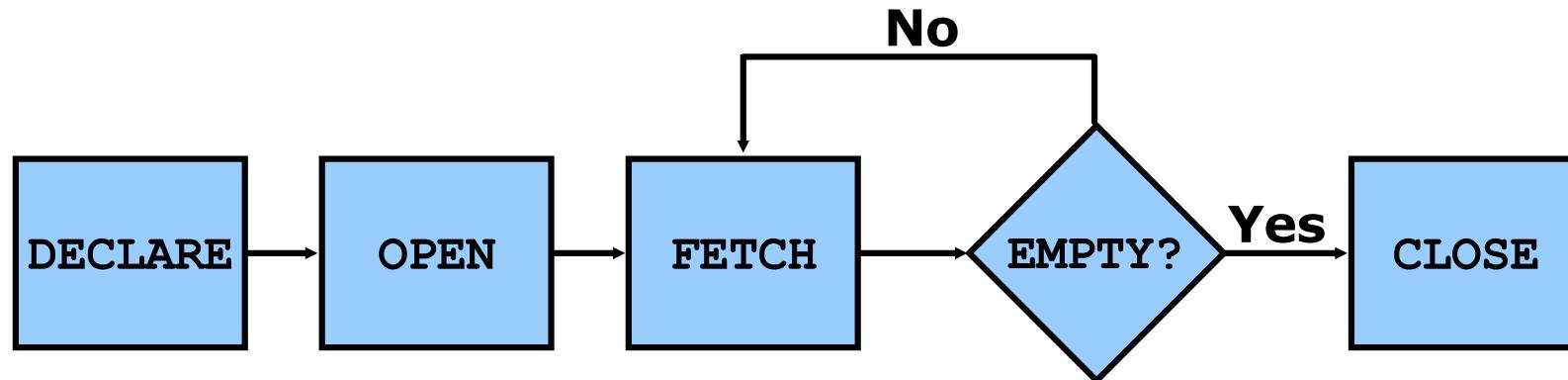


**Explicit cursor**

# Explicit Cursor Operations



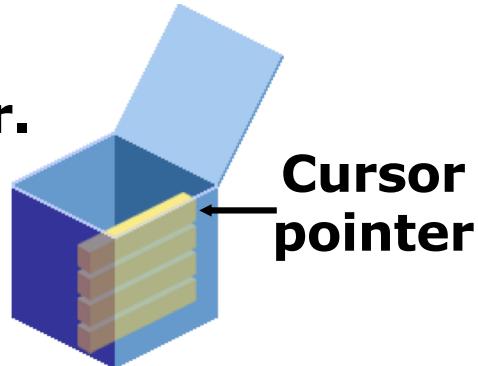
# Controlling Explicit Cursors



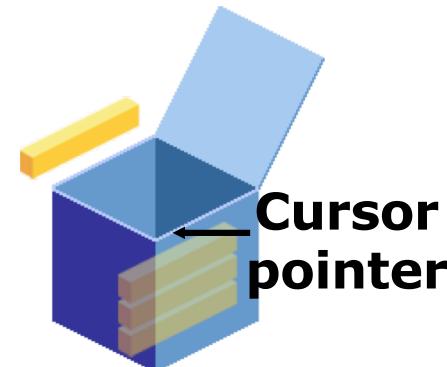
- Create a named SQL area.
- Identify the active set.
- Load the current row into variables.
- Test for existing rows.
- Release the active set.
- Return to FETCH if rows are found.

# Controlling Explicit Cursors

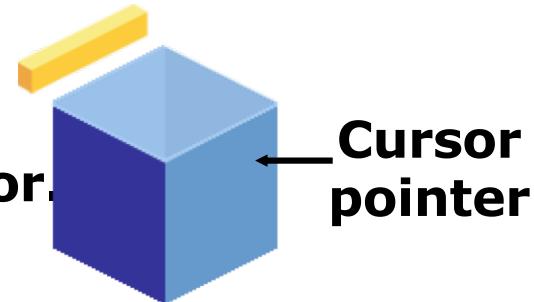
1 Open the cursor.



2 Fetch a row.



3 Close the cursor.



# Declaring the Cursor

## Syntax:

```
CURSOR cursor_name IS  
    select_statement;
```

## Examples:

```
DECLARE  
    CURSOR c_emp_cursor IS  
        SELECT employee_id, last_name FROM employees  
        WHERE department_id =30;
```

```
DECLARE  
    v_locid NUMBER:= 1700;  
    CURSOR c_dept_cursor IS  
        SELECT * FROM departments  
        WHERE location_id = v_locid;  
    ...
```

# Opening the Cursor

```
DECLARE
    CURSOR c_emp_cursor IS
        SELECT employee_id, last_name FROM employees
        WHERE department_id =30;
    ...
BEGIN
    OPEN c_emp_cursor;
```

# Fetching Data from the Cursor

```
DECLARE
  CURSOR c_emp_cursor IS
    SELECT employee_id, last_name FROM employees
    WHERE department_id =30;
    v_empno employees.employee_id%TYPE;
    v_lname employees.last_name%TYPE;
BEGIN
  OPEN c_emp_cursor;
  FETCH c_emp_cursor INTO v_empno, v_lname;
  DBMS_OUTPUT.PUT_LINE( v_empno ||' '||v_lname);
END ;
/
```

```
anonymous block completed
114 Raphaely
```

# Fetching Data from the Cursor

```
DECLARE
    CURSOR c_emp_cursor IS
        SELECT employee_id, last_name FROM employees
        WHERE department_id =30;
        v_empno employees.employee_id%TYPE;
        v_lname employees.last_name%TYPE;
BEGIN
    OPEN c_emp_cursor;
    LOOP
        FETCH c_emp_cursor INTO v_empno, v_lname;
        EXIT WHEN c_emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( v_empno ||' '||v_lname);
    END LOOP;
END ;
/
```

# Closing the Cursor

```
...
LOOP
    FETCH c_emp_cursor INTO empno, lname;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_empno || ' ' || v_lname);
END LOOP;
CLOSE c_emp_cursor;
END;
/
```

# Cursors and Records

Process the rows of the active set by fetching values into a PL/SQL record.

```
DECLARE
    CURSOR c_emp_cursor IS
        SELECT employee_id, last_name FROM employees
        WHERE department_id =30;
        v_emp_record    c_emp_cursor%ROWTYPE;
BEGIN
    OPEN c_emp_cursor;
    LOOP
        FETCH c_emp_cursor INTO v_emp_record;
        EXIT WHEN c_emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( v_emp_record.employee_id
                            ||' '||v_emp_record.last_name);
    END LOOP;
    CLOSE c_emp_cursor;
END;
```

# Cursor FOR Loops

## Syntax:

```
FOR record_name IN cursor_name LOOP  
    statement1;  
    statement2;  
    . . .  
END LOOP;
```

- The cursor FOR loop is a shortcut to process explicit cursors.
- Implicit open, fetch, exit, and close occur.
- The record is implicitly declared.

# Cursor FOR Loops

```
DECLARE
    CURSOR c_emp_cursor IS
        SELECT employee_id, last_name FROM employees
        WHERE department_id =30;
BEGIN
    FOR emp_record IN c_emp_cursor
    LOOP
        DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
        ||' '||emp_record.last_name);
    END LOOP;
END ;
/
```

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

# Explicit Cursor Attributes

**Use explicit cursor attributes to obtain status information about a cursor.**

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	Number	Evaluates to the total number of rows returned so far

## **%ISOPEN Attribute**

- Fetch rows only when the cursor is open.
- Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.

**Example:**

```
IF NOT c_emp_cursor%ISOPEN THEN
    OPEN c_emp_cursor;
END IF;
LOOP
    FETCH c_emp_cursor...
```

# %ROWCOUNT and %NOTFOUND: Example

```
DECLARE
    CURSOR c_emp_cursor IS SELECT employee_id,
        last_name FROM employees;
    v_emp_record  c_emp_cursor%ROWTYPE;
BEGIN
    OPEN c_emp_cursor;
    LOOP
        FETCH c_emp_cursor INTO v_emp_record;
        EXIT WHEN c_emp_cursor%ROWCOUNT > 10 OR
            c_emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( v_emp_record.employee_id
            ||' '||v_emp_record.last_name);
    END LOOP;
    CLOSE c_emp_cursor;
END ; /
```

```
anonymous block completed
198 OConnell
199 Grant
200 Whalen
201 Hartstein
202 Fay
203 Mavris
204 Baer
205 Higgins
206 Gietz
100 King
```

ORACLE®

# Cursor FOR Loops Using Subqueries

There is no need to declare the cursor.

```
BEGIN
    FOR emp_record IN (SELECT employee_id, last_name
                        FROM employees WHERE department_id =30)
    LOOP
        DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
                           ||' '||emp_record.last_name);
    END LOOP;
END ;
/
```

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

# Cursors with Parameters

## Syntax:

```
CURSOR cursor_name
  [ (parameter_name datatype, . . . ) ]
IS
  select_statement;
```

- Passer des valeurs de paramètre à un curseur lorsque le curseur est ouvert et que la requête est exécutée.
- Ouvrir un curseur explicit plusieurs fois avec un actif différent à chaque fois.

```
OPEN cursor_name(parameter_value, . . . . ) ;
```

# Cursors with Parameters

```
DECLARE
  CURSOR    c_emp_cursor (deptno NUMBER) IS
    SELECT employee_id, last_name
    FROM   employees
   WHERE  department_id = deptno;
   ...
BEGIN
  OPEN c_emp_cursor (10);
  ...
  CLOSE c_emp_cursor;
  OPEN c_emp_cursor (20);
  ...
END;
```

```
anonymous block completed
200 Whalen
201 Hartstein
202 Fay
```

# FOR UPDATE Clause

## Syntax:

```
SELECT ...  
FROM      ...  
FOR UPDATE [OF column_reference] [NOWAIT | WAIT n];
```

- Utilise le verrouillage explicite pour refuser l'accès aux autres sessions pendant toute la durée d'une transaction.
- Verrouille les lignes avant la mise à jour ou la suppression.

# WHERE CURRENT OF Clause

## Syntax:

```
WHERE CURRENT OF cursor ;
```

- Utilise le curseur pour mettre à jour ou supprimer la ligne actuelle.
- Inclure la clause FOR UPDATE dans la requête de curseur pour verrouiller les lignes d'abord.
- Utiliser la clause WHERE CURRENT OF pour faire référence à la ligne courante d'un curseur explicit.

```
UPDATE employees  
      SET salary = ...  
 WHERE CURRENT OF c_emp_cursor;
```

# Cursors with Subqueries: Example

```
DECLARE
  CURSOR my_cursor IS
    SELECT t1.department_id, t1.department_name,
           t2.staff
      FROM departments t1, (SELECT department_id,
                                         COUNT(*) AS staff
                                    FROM employees
                                   GROUP BY department_id) t2
     WHERE t1.department_id = t2.department_id
       AND t2.staff >= 3;
  ...

```

# Exercice

- **Ecrire un programme permettant**
  - de mettre à jour le salaire de tous les employés en leurs ajoutant :
    - **500 \$ si l'employé appartient au département vente**
    - **300 \$ s 'il appartient au département opérateurs**
    - **400 \$ s 'il appartient au département SI**
    - **600 \$ s 'il appartient au département Administrateurs**

**NB: utiliser les curseurs explicites et CASE Expression**

# Exercices 1

1. Créez un bloc PL/SQL qui effectue les opérations suivantes:

- a. dans la section déclarative, déclarez une variable `v_deptno` de type numérique et assignez une valeur qui contient l'ID du département.
- b. Déclarer un curseur, `c_emp_cursor`, qui extrait le `last_name`, `salaire` et `manager_id` des employés travaillant dans le département spécifié dans `v_deptno`.
- c. dans la section exécutable, utilisez le curseur FOR loop pour opérer sur les données récupérées.
  - i. Si le salaire de l'employé est inférieur à 5 000, et si l'ID du manager est 101 ou 124, afficher le message de `<<last_name>> Due for a raise.`
  - ii. Dans le cas contraire, afficher le message `<<last_name>> Not due for a raise.`

## Exercice 2

- a. Dans la section déclarative, déclarez un curseur dept\_cursor pour récupérer department\_id et department\_name avec department\_id inférieur à 100, ordonnées par department\_id.
- b. Déclarer un autre curseur, emp\_cursor, qui prend le numéro de département en tant que paramètre et récupère last\_name, job\_id, hire\_date et le salaire des employés dont employee\_id est inférieure à 120 et qui travaillent dans ce département.
- c. Déclarer des variables pour contenir des valeurs extraites de chaque curseur.
- d. Ouvrir dept\_cursor, utilisez une boucle simple et extraire les valeurs dans les variables déclarées. Afficher le numéro et le nom du département .
- e. Pour chaque département, ouvrez emp\_cursor en passant le numéro actuel du département en tant que paramètre. Commencer une autre boucle et extraire les valeurs d'emp\_cursor dans des variables et imprimer tous les détails qui provient de la table employees . *Remarque : Vous pouvez imprimer une ligne après avoir affiché les détails de chaque département. Utilisez des attributs appropriés pour la condition de sortie. En outre, déterminer si un curseur est déjà ouvert avant de l'ouvrir.*
- a. Fermez tous les curseurs et les boucles et puis terminer la section exécutable.

## Exercice 3

Créez un bloc PL/SQL qui détermine les *n* employés les plus rémunérés (n plus grands salaires) et les insère dans une table `top_salaries`.

- a. Dans la section déclarative, déclarez une variable `v_num` de type numérique qui contient un nombre *n* représentant le nombre de salariés de *n* haut salaires de la table `employees`. Par exemple, pour afficher les cinq plus grands salaires, entrez 5.
- b. Déclarer une autre variable `sal` de type `employees.salary`. Déclarer un curseur, `c_emp_cursor`, qui récupère les salaires des employés dans l'ordre décroissant.

**Note:** Make sure you add an exit condition to avoid having an infinite loop.

# Cursor Variables

- **Les variables de curseur agissent comme les pointeurs C. Ils détiennent l'emplacement de la mémoire (adresse) d'un élément au lieu de la valeur de l'élément lui-même.**
- **Dans PL/SQL, un pointeur est déclaré comme REF X, où REF est une abréviation de REFERENCE et X représente une classe d'objets.**
- **Une variable de curseur dispose du type REF CURSOR.**
- **Un CURSOR est statique alors qu'un REF CURSOR est dynamique.**

# Using Cursor Variables

- Vous pouvez utiliser des Ref Cursor pour passer des jeux de résultats de requête entre les blocks PL/SQL.
- PL/SQL peut partager un pointeur vers la zone de travail de requête dans lequel est stocké le jeu de résultats
- Vous pouvez passer la valeur d'une variable de curseur librement d'un champ d'application à un autre.

# Defining REF CURSOR Types

Define a REF CURSOR type:

```
Define a REF CURSOR type
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

Declare a cursor variable of that type:

```
ref_cv ref_type_name;
```

Example:

```
DECLARE
TYPE DeptCurTyp IS REF CURSOR RETURN
departments%ROWTYPE;
dept_cv DeptCurTyp;
```

# Using the OPEN-FOR, FETCH, and CLOSE Statements

- L'instruction **OPEN-FOR**, associe un curseur à une requête multi ligne, exécute la requête, identifie le jeu de résultats et positionne le curseur vers la première ligne du jeu de résultats.
- L'instruction **FETCH** retourne une ligne du résultat de la requête multi lignes, assigne les valeurs des éléments de la liste de sélection à des variables ou des champs dans la clause **INTO**, incrémente le décompte tenu par **% ROWCOUNT** et avance le curseur à la ligne suivante.
- L'instruction **CLOSE** désactive une variable de curseur.

# Opening REF CURSOR

```
OPEN NOM_CURSEUR FOR REQUETE  
[ USING [ IN | OUT | IN OUT ] ARGUMENT[,...] ] ;
```

**USING:** Cette clause permet le paramétrage de la requête SQL dynamique en utilisant une liste des arguments.

**IN ARGUMENT :** L'argument est passé à la requête SQL dynamique lors de son invocation. Il ne peut pas être modifié à l'intérieur de la requête SQL dynamique.

**OUT ARGUMENT :** L'argument est ignoré lors de l'invocation de la requête SQL dynamique. À l'intérieur de celle-ci, l'argument se comporte comme une variable PL/SQL n'ayant pas été initialisée, contenant donc la valeur « NULL » et supportant les opérations de lecture et d'écriture. Au terme de la requête SQL dynamique, il retourne à la valeur affectée.

**IN OUT ARGUMENT:** L'argument combine les deux propriétés « IN » et « OUT ».

# Example of Fetching

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    emp_cv    EmpCurTyp;
    emp_rec   employees%ROWTYPE;
    sql_stmt  VARCHAR2(200);
    my_job    VARCHAR2(10) := 'ST_CLERK';
BEGIN
    sql_stmt := 'SELECT * FROM employees
                WHERE job_id = :j';
    OPEN emp_cv FOR sql_stmt USING my_job;
LOOP
    FETCH emp_cv INTO emp_rec;
    EXIT WHEN emp_cv%NOTFOUND;
    -- process record
END LOOP;
CLOSE emp_cv;
END;
/
```

# Exercice

**Créez le bloc PL/SQL qui permet d'afficher toute les lignes de l'une des tables :**

- EMPLOYES\_1996,
- EMPLOYES\_1997,
- EMPLOYES\_1998, ...
- EMPLOYES\_YYYY

**Dynamiquement, suivant l'année passée en argument,**

- **vous testez que la table existe et vous affichez tous les enregistrements de la table.**
- **Si la table n'existe pas, vous affichez tous les enregistrements de la table EMPLOYES pour l'année qui a été passée en argument.**

# HANDLING EXCEPTIONS

# Example of an Exception

```
DECLARE
    v_lname VARCHAR2(15);
BEGIN
    SELECT last_name INTO v_lname
    FROM employees
    WHERE first_name='John';
    DBMS_OUTPUT.PUT_LINE ('John''s last name is :'
                           ||v_lname);
END;
```

Error report:

```
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
01422. 00000 - "exact fetch returns more than requested number of rows"
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested
```

# Example of an Exception

```
DECLARE
    v_lname VARCHAR2(15);
BEGIN
    SELECT last_name INTO v_lname
    FROM employees
    WHERE first_name='John';
    DBMS_OUTPUT.PUT_LINE ('John''s last name is :'
                           ||v_lname);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE (' Your select statement
                             retrieved multiple rows. Consider using a
                             cursor.');
END;
/
```

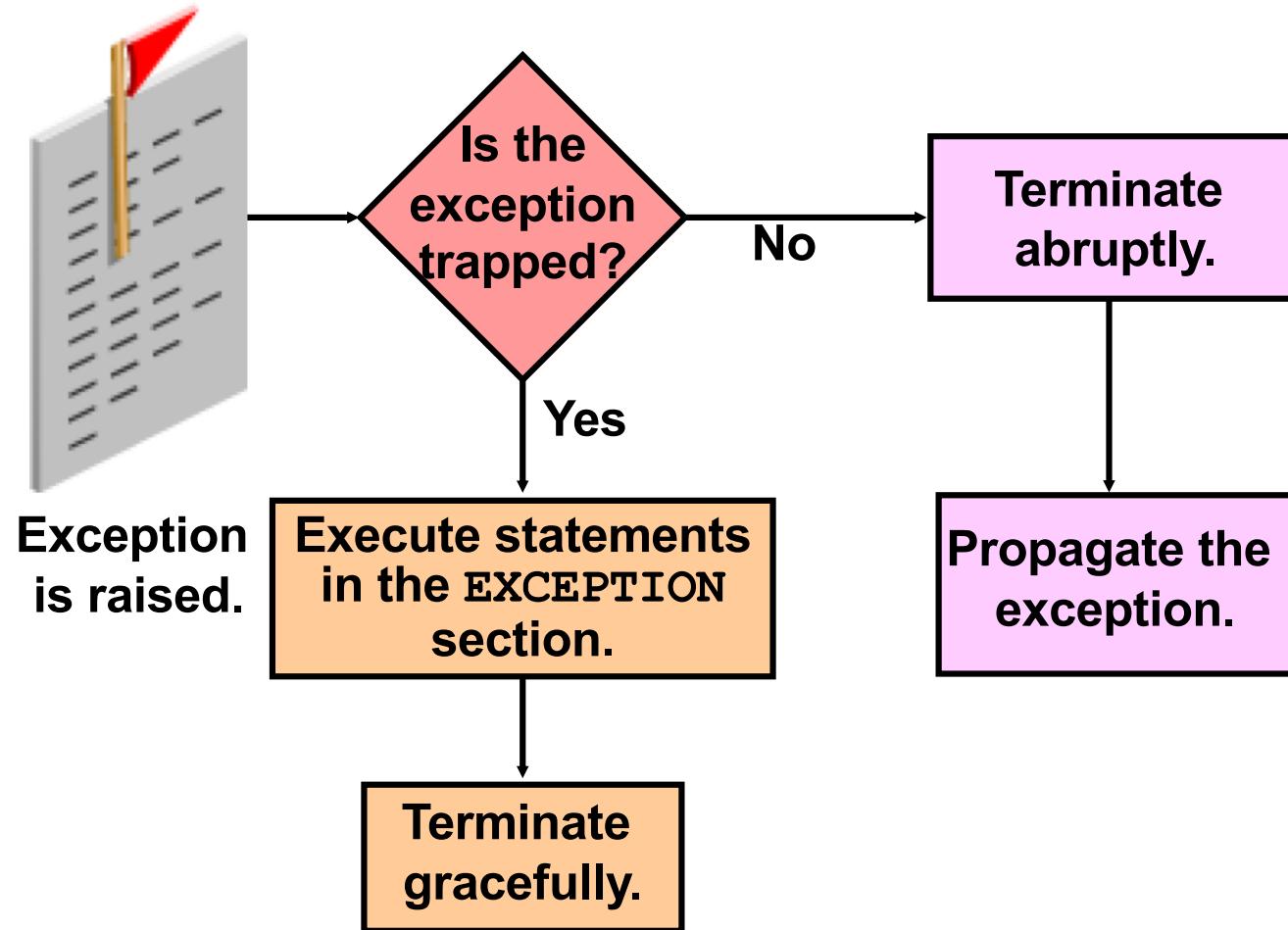
anonymous block completed  
Your select statement retrieved multiple  
rows. Consider using a cursor.

ORACLE®

# Handling Exceptions with PL/SQL

- **Une exception est une erreur de PL/SQL qui est déclenchée pendant l'exécution du programme.**
- **Une exception peut être levée :**
  - implicitement par le serveur Oracle
  - explicitement par le programme
- **Une exception peut être traitée :**
  - par une capture et un traitement par le bloc courant
  - en la propageant à l'environnement appelant

# Handling Exceptions



# Exception Types

- Predefined Oracle server
- Non-predefined Oracle server

} Implicitly raised

- User-defined

Explicitly raised

# Trapping Exceptions

## Syntax:

```
EXCEPTION  
  WHEN exception1 [OR exception2 . . .] THEN  
    statement1;  
    statement2;  
    . . .  
  [WHEN exception3 [OR exception4 . . .] THEN  
    statement1;  
    statement2;  
    . . .]  
  [WHEN OTHERS THEN  
    statement1;  
    statement2;  
    . . .]
```

# Guidelines for Trapping Exceptions

- The EXCEPTION keyword starts the exception-handling section.
- Several exception handlers are allowed.
- Only one handler is processed before leaving the block.
- WHEN OTHERS is the last clause.

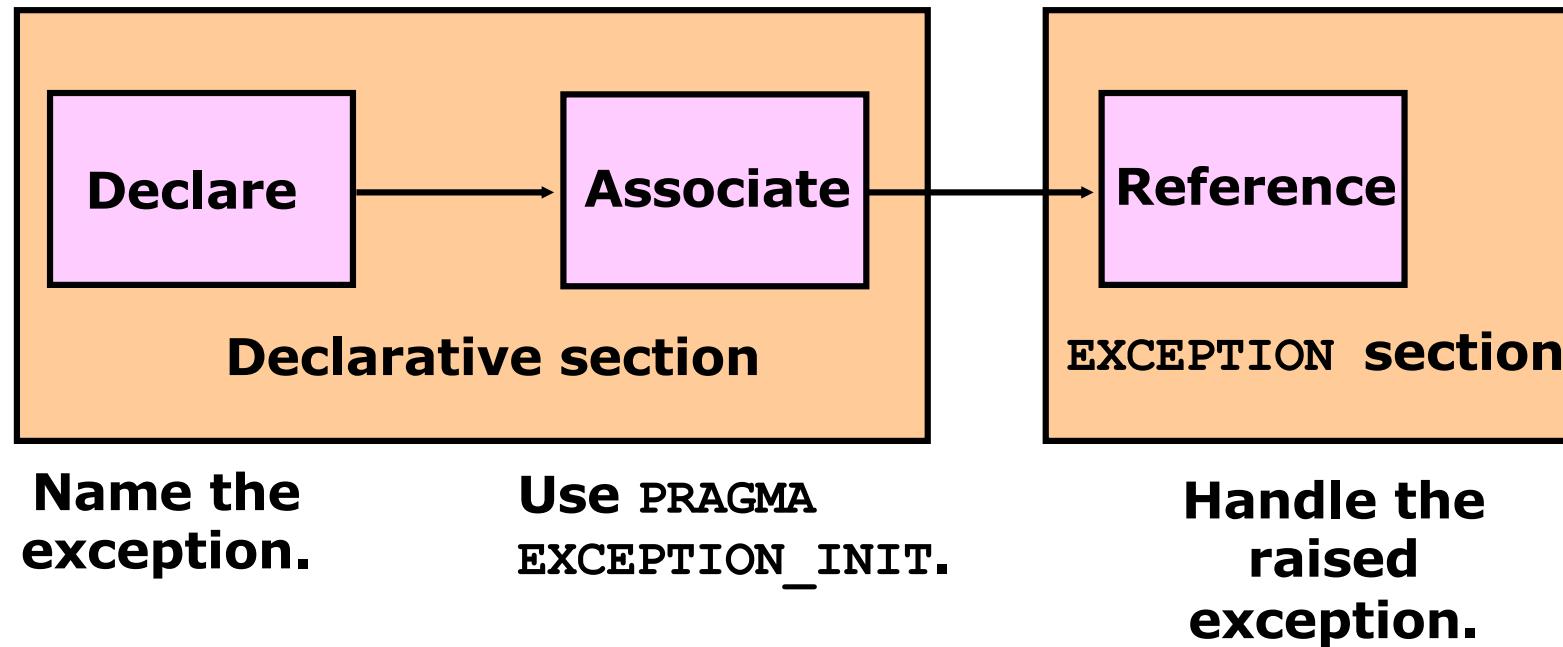
# Trapping Predefined Oracle Server Errors

- Reference the predefined name in the exception-handling routine.
- Sample predefined exceptions:
  - NO\_DATA\_FOUND
  - TOO\_MANY\_ROWS
  - INVALID\_CURSOR
  - ZERO\_DIVIDE
  - DUP\_VAL\_ON\_INDEX

# Trapping Predefined Oracle Server Errors

```
DECLARE
    v_lname VARCHAR2(15);
BEGIN
    SELECT last_name INTO v_lname
    FROM employees
    WHERE first_name='John';
    DBMS_OUTPUT.PUT_LINE ('John''s last name is :'
                          ||v_lname);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE (' Your select statement
retrieved no rows.');
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE (' Your select statement
retrieved multiple rows. Consider using a
cursor.');
END;
/
```

# Trapping Non-Predefined Oracle Server Errors



# Non-Predefined Error

To trap Oracle server error number -01400 (“cannot insert NULL”):

```
DECLARE
    e_insert_excep EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_insert_excep, -01400);
BEGIN
    INSERT INTO departments
        (department_id, department_name) VALUES (280, NULL);
EXCEPTION
    WHEN e_insert_excep THEN
        DBMS_OUTPUT.PUT_LINE('INSERT OPERATION FAILED');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

```
anonymous block completed
INSERT OPERATION FAILED
ORA-01400: cannot insert NULL into ("ORA41"."DEPARTMENTS"."DEPARTMENT_NAME")
```

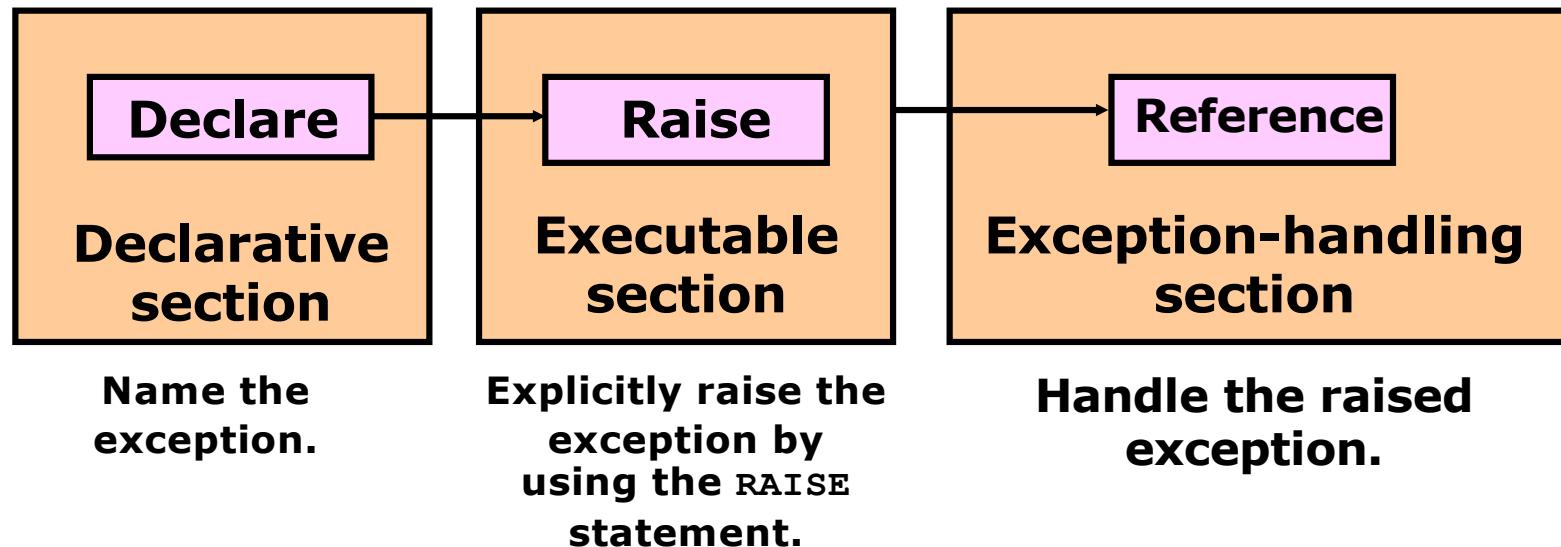
# Functions for Trapping Exceptions

- **SQLCODE:** Returns the numeric value for the error code
- **SQLERRM:** Returns the message associated with the error number

# Functions for Trapping Exceptions

```
DECLARE
    error_code      NUMBER;
    error_message   VARCHAR2(255);
BEGIN
    ...
EXCEPTION
    ...
    WHEN OTHERS THEN
        ROLLBACK;
        error_code := SQLCODE ;
        error_message := SQLERRM ;
        INSERT INTO errors (e_user, e_date, error_code,
                           error_message) VALUES (USER, SYSDATE, error_code,
                           error_message);
END;
/
```

# Trapping User-Defined Exceptions



# Trapping User-Defined Exceptions

```
DECLARE
    v_deptno NUMBER := 500;
    v_name VARCHAR2(20) := 'Testing';
    e_invalid_department EXCEPTION; ← 1
BEGIN
    UPDATE departments
    SET department_name = v_name
    WHERE department_id = v_deptno;
    IF SQL % NOTFOUND THEN
        RAISE e_invalid_department; ← 2
    END IF;
    COMMIT;
EXCEPTION
    WHEN e_invalid_department THEN
        DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
/
```

anonymous block completed  
No such department id.

ORACLE®

# Propagating Exceptions in a Subblock

**Subblocks can handle an exception or pass the exception to the enclosing block.**

```
DECLARE
    . . .
    e_no_rows      exception;
    e_integrity    exception;
    PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
    FOR c_record IN emp_cursor LOOP
        BEGIN
            SELECT . . .
            UPDATE . . .
            IF SQL%NOTFOUND THEN
                RAISE e_no_rows;
            END IF;
        END;
    END LOOP;
EXCEPTION
    WHEN e_integrity THEN . . .
    WHEN e_no_rows THEN . . .
END;
/
```

# **RAISE\_APPLICATION\_ERROR Procedure**

## **Syntax:**

```
raise_application_error (error_number,  
                         message[, {TRUE | FALSE}]);
```

- **Vous pouvez utiliser cette procédure pour émettre des messages d'erreur définis par l'utilisateur de sous-programmes stockées.**
- **Vous pouvez signaler des erreurs de votre application et éviter le déclenchement d'exceptions non gérées.**

# **RAISE\_APPLICATION\_ERROR Procedure**

- **Used in two different places:**
  - Executable section
  - Exception section
- **Returns error conditions to the user in a manner consistent with other Oracle server errors**

# **RAISE\_APPLICATION\_ERROR Procedure**

## **Executable section:**

```
BEGIN  
  ...  
  DELETE FROM employees  
    WHERE manager_id = v_mgr;  
  IF SQL%NOTFOUND THEN  
    RAISE_APPLICATION_ERROR(-20202,  
      'This is not a valid manager');  
  END IF;  
  ...
```

## **Exception section:**

```
  ...  
EXCEPTION  
  WHEN NO_DATA_FOUND THEN  
    RAISE_APPLICATION_ERROR (-20201,  
      'Manager is not a valid employee.');
```

END;

## Another example

```
DECLARE
  e_name EXCEPTION;
  PRAGMA EXCEPTION_INIT (e_name, -20999);
BEGIN
  ...
  DELETE FROM employees
  WHERE last_name = 'Higgins';
  IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20999,'This is not a
      valid last name');
  END IF;
EXCEPTION
  WHEN e_name THEN
    -- handle the error
  ...
END;
/
```

# Exercises

- a. In the declarative section, declare two variables: `v_ename` of type `employees.last_name` and `v_emp_sal` of type `employees.salary`. Initialize the latter to 6000.
- b. In the executable section, retrieve the last names of employees whose salaries are equal to the value in `v_emp_sal`.  
Note: Do not use explicit cursors.  
If the salary entered returns only one row, insert into the messages table the employee's name and the salary amount.
- c. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the messages table the message "No employee with a salary of <salary>."
- d. If the salary entered returns more than one row, handle the exception with an appropriate exception handler and insert into the messages table the message "More than one employee with a salary of <salary>."
- e. Handle any other exception with an appropriate exception handler and insert into the messages table the message "Some other error occurred."

# Exercises

- 2. Use the Oracle server error ORA-02292 (integrity constraint violated – child record found).**
  - a. In the declarative section, declare an exception `e_childrecord_exists`. Associate the declared exception with the standard Oracle server error –02292.**
  - b. In the executable section, display “Deleting department 40....” Include a `DELETE` statement to delete the department with `department_id` 40.**
  - c. Include an exception section to handle the `e_childrecord_exists` exception and display the appropriate message. Sample output is as follows:**

```
anonymous block completed
Deleting department 40.....
Cannot delete this department.
There are employees in this department (child records exist.)
```

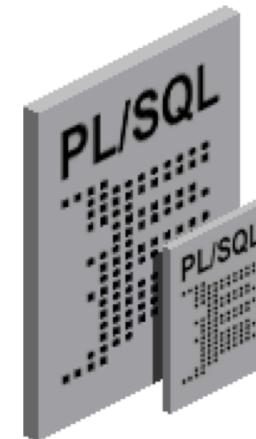
- 3. Rewrite the block to remove all departments who have no employee**

# **CREATING STORED PROCEDURES AND FUNCTIONS**

**ORACLE®**

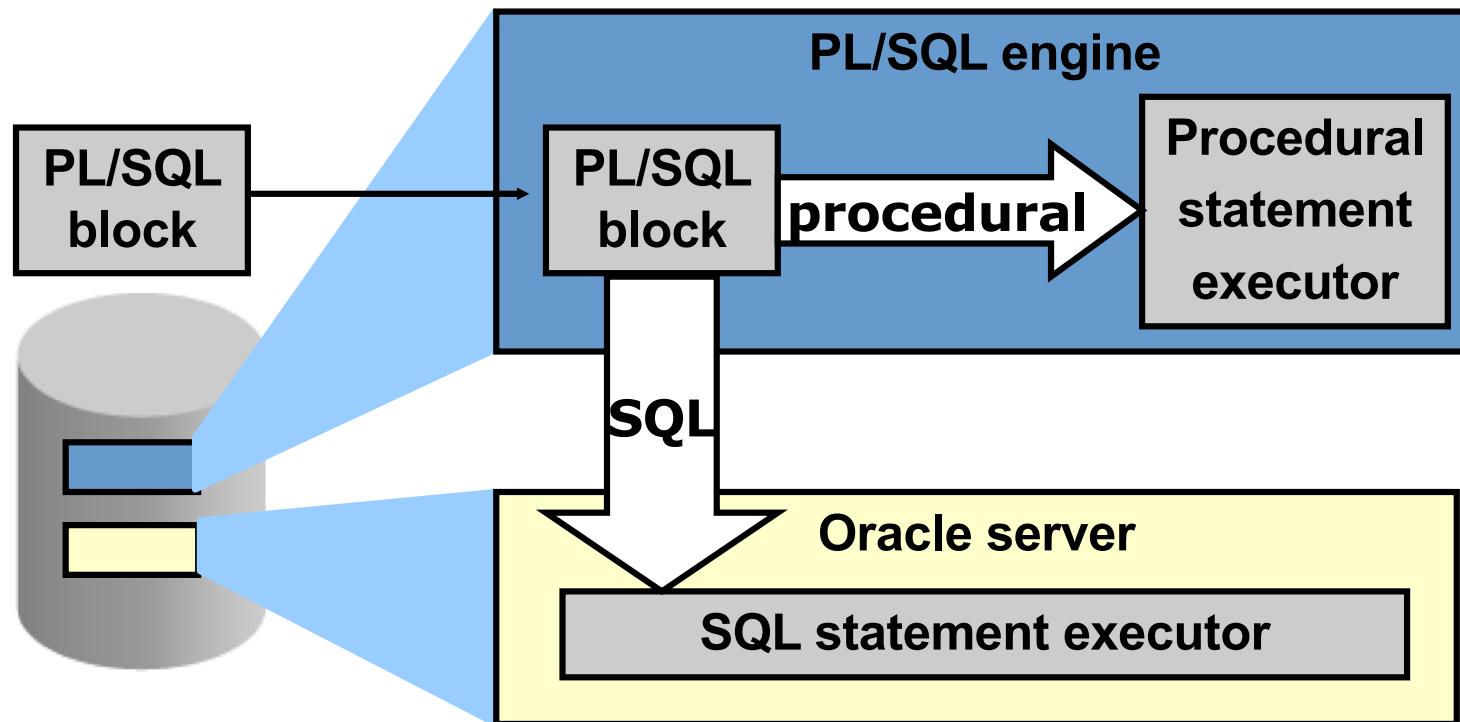
# Procedures and Functions

- Sont des block PL/SQL nommés
- Ont une structure semblable à celle des blocs anonymes :
  - Optional declarative section (without the DECLARE keyword)
  - Mandatory executable section
  - Optional section to handle exceptions



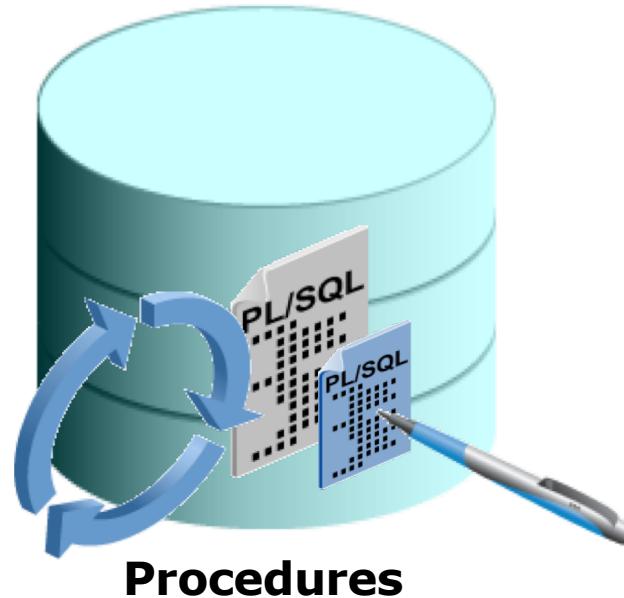
# PL/SQL Execution Environment

The PL/SQL run-time architecture:

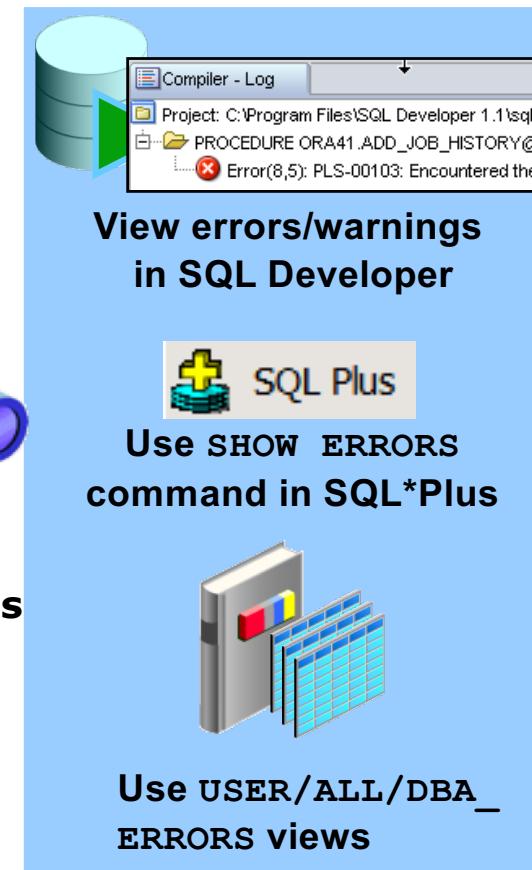
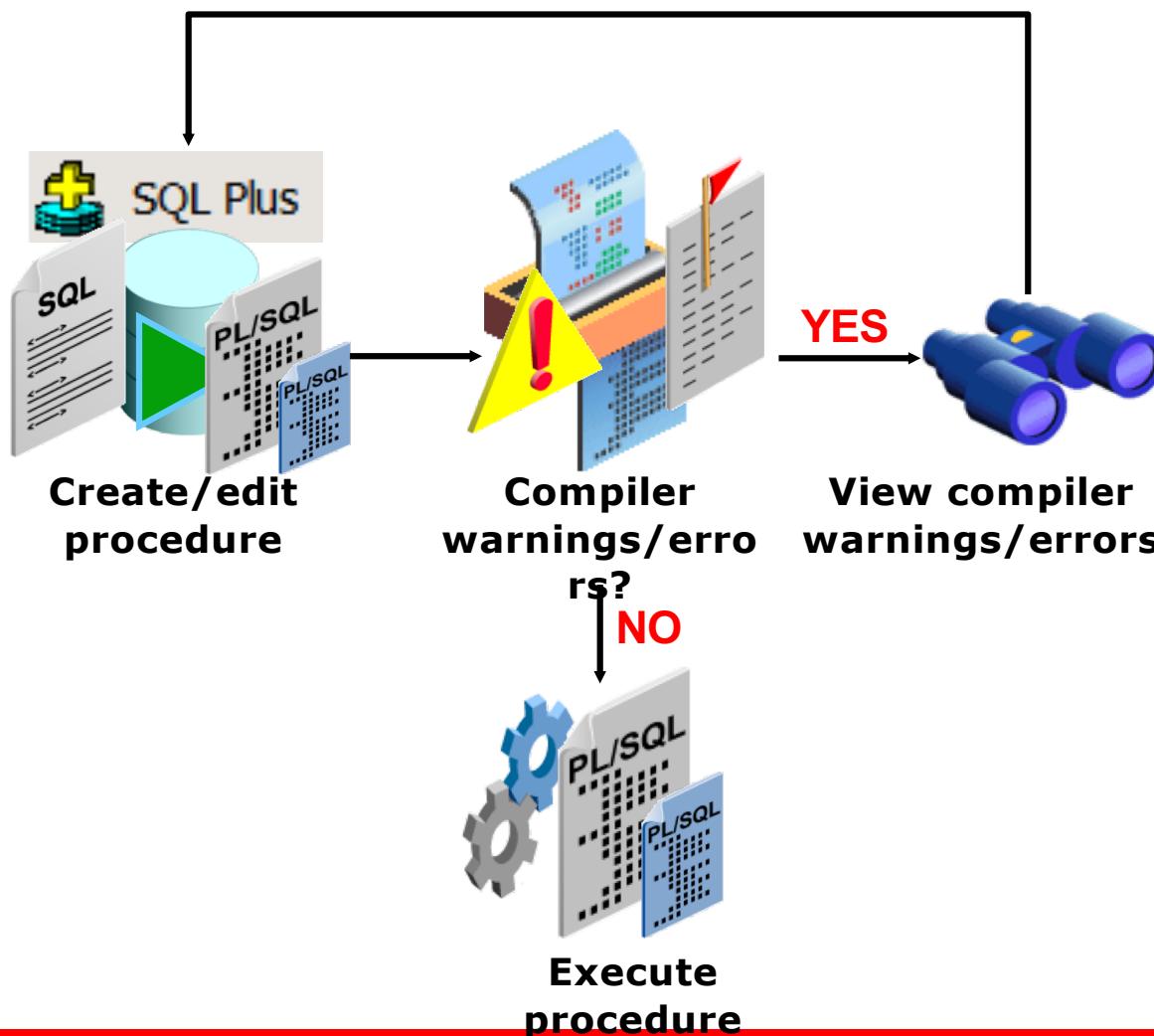


# What Are Procedures?

- Sont un type de sous-programme qui exécutent une action
- Peuvent être stockés dans la base de données comme un objet de schéma
- Promeuvent la réutilisation et la maintenabilité



# Creating Procedures: Overview



# Procedure: Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[ (argument1 [mode1] datatype1,
  argument2 [mode2] datatype2,
  . . .)]
IS|AS
procedure_body;
```

# Procedure: Example

```
....  
CREATE TABLE dept AS SELECT * FROM departments;  
CREATE PROCEDURE add_dept IS  
    v_dept_id dept.department_id%TYPE;  
    v_dept_name dept.department_name%TYPE;  
BEGIN  
    v_dept_id:=280;  
    v_dept_name:='ST-Curriculum';  
    INSERT INTO dept(department_id,department_name)  
    VALUES(v_dept_id,v_dept_name);  
    DBMS_OUTPUT.PUT_LINE(' Inserted '|| SQL%ROWCOUNT  
    ||' row ' );  
END ;
```

# Formal and Actual Parameters

- **Paramètres formels** : les variables locales déclarées dans la liste de paramètres d'une spécification de sous-programme
- **Véritables paramètres (ou arguments)**: valeurs littérales, variables et expressions utilisées dans la liste des paramètres de l'appel de sous-programme

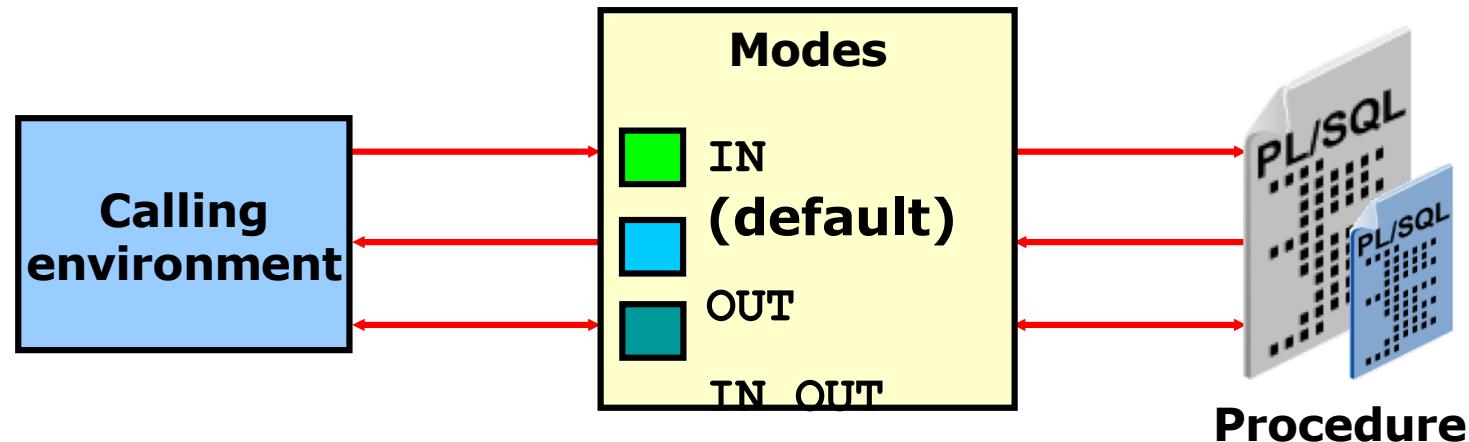
```
-- Procedure definition, Formal parameters
CREATE PROCEDURE raise_sal(p_id NUMBER, p_sal NUMBER) IS
BEGIN
    . . .
END raise_sal;
```

```
-- Procedure calling, Actual parameters (arguments)
v_emp_id := 100;
raise_sal(v_emp_id, 2000)
```

# Procedural Parameter Modes

- Les modes des paramètres sont précisés dans la déclaration des paramètres formels, après le nom du paramètre et avant son type de données.
- Le mode `IN` est la valeur par défaut si aucun mode n'est spécifié.

```
CREATE PROCEDURE proc_name(param_name [mode] datatype)  
...
```



# Passing Actual Parameters: Creating the add\_dept Procedure

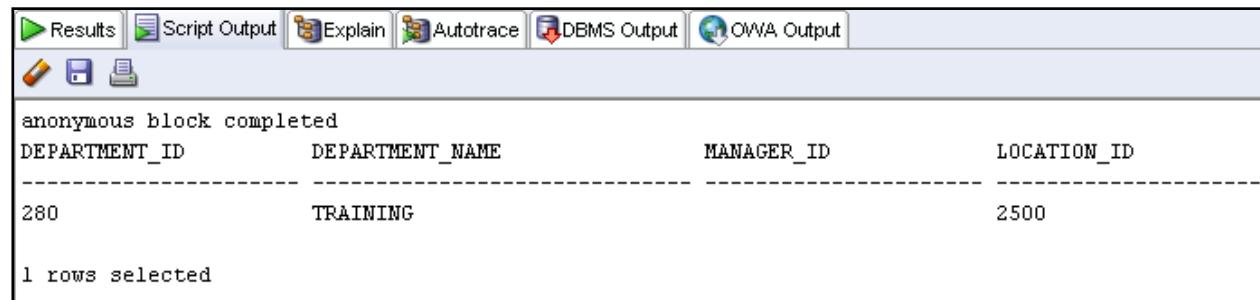
```
CREATE OR REPLACE PROCEDURE add_dept(
    p_name IN departments.department_name%TYPE,
    p_loc IN departments.location_id%TYPE) IS
BEGIN
    INSERT INTO departments(department_id,
                           department_name, location_id)
    VALUES (departments_seq.NEXTVAL, p_name , p_loc );
END add_dept;
/
```



ORACLE®

# Passing Actual Parameters: Examples

```
-- Passing parameters using the positional notation.  
EXECUTE add_dept ('TRAINING', 2500)
```

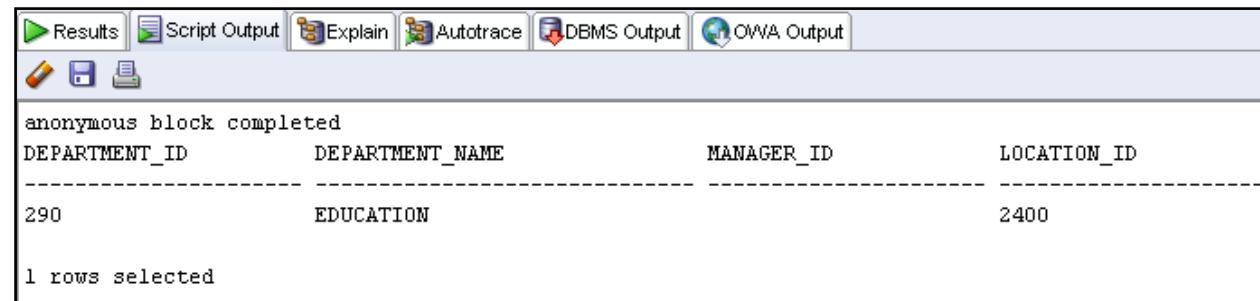


The screenshot shows the Oracle SQL Developer interface with the 'Results' tab selected. An anonymous block has been executed, displaying the output of the 'add\_dept' procedure. The output shows a single row inserted into the DEPARTMENTS table.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	TRAINING		2500

1 rows selected

```
-- Passing parameters using the named notation.  
EXECUTE add_dept (p_loc=>2400, p_name=>'EDUCATION')
```



The screenshot shows the Oracle SQL Developer interface with the 'Results' tab selected. An anonymous block has been executed, displaying the output of the 'add\_dept' procedure. The output shows a single row inserted into the DEPARTMENTS table using named parameters.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
290	EDUCATION		2400

1 rows selected

# Invoking the Procedure

```
BEGIN  
    add_dept;  
END ;  
/  
SELECT department_id, department_name FROM dept  
WHERE department_id=280;
```

```
anonymous block completed  
Inserted 1 row  
  
DEPARTMENT_ID      DEPARTMENT_NAME  
-----  
280                ST-Curriculum  
  
1 rows selected
```

# Calling Procedures

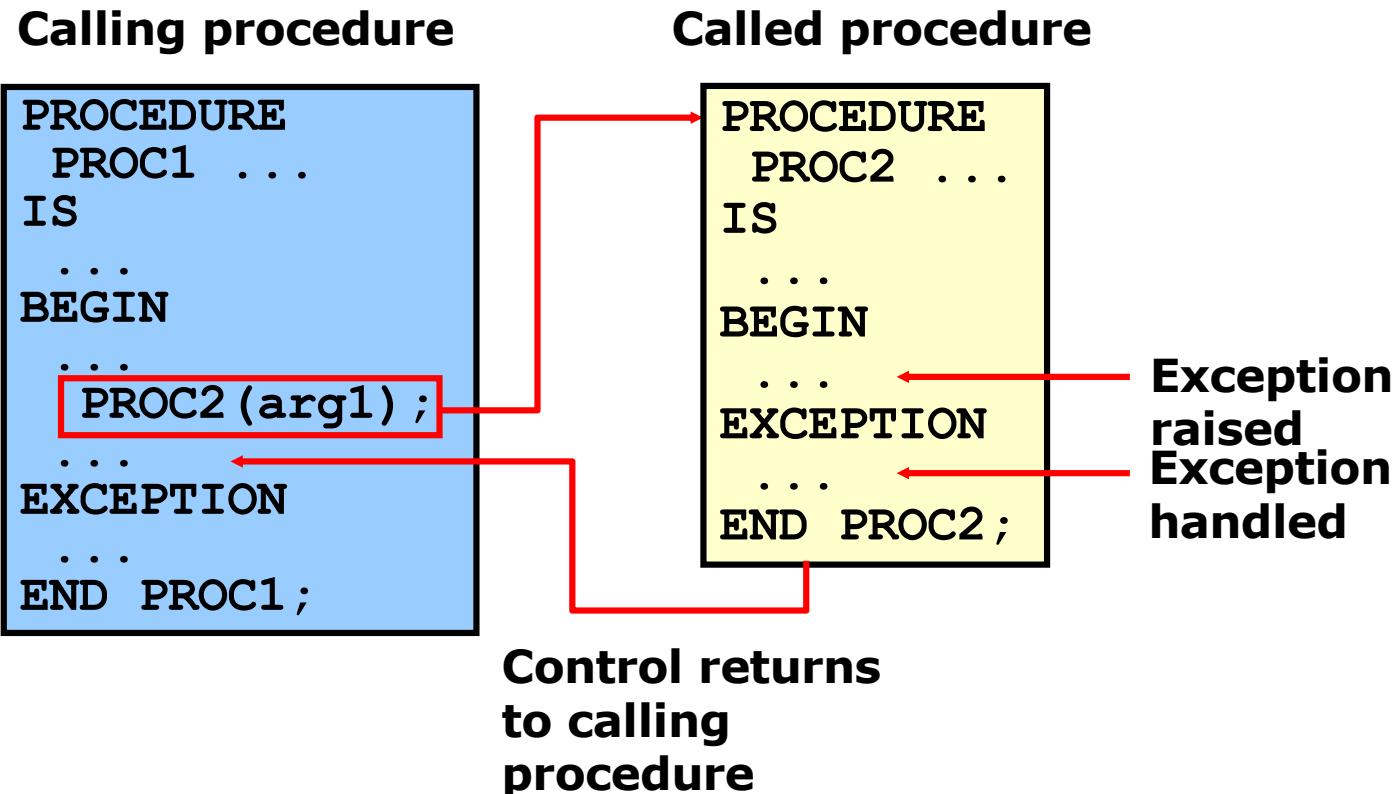
**Vous pouvez appeler des procédures à l'aide de blocs anonymes, une autre procédure ou package.**

```
CREATE OR REPLACE PROCEDURE process_employees
IS
    CURSOR cur_emp_cursor IS
        SELECT employee_id
        FROM   employees;
BEGIN
    FOR emp_rec IN cur_emp_cursor
    LOOP
        raise_salary(emp_rec.employee_id, 10);
    END LOOP;
    COMMIT;
END process_employees;
/
```

```
PROCEDURE process_employees Compiled.
```

ORACLE®

# Handled Exceptions



# Handled Exceptions: Example

```
CREATE PROCEDURE add_department(
    p_name VARCHAR2, p_mgr NUMBER, p_loc NUMBER) IS
BEGIN
    INSERT INTO DEPARTMENTS (department_id,
        department_name, manager_id, location_id)
    VALUES (DEPARTMENTS_SEQ.NEXTVAL, p_name, p_mgr, p_loc);
    DBMS_OUTPUT.PUT_LINE('Added Dept: '|| p_name);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Err: adding dept: '|| p_name);
END;
```

```
CREATE PROCEDURE create_departments IS
BEGIN
    add_department('Media', 100, 1800);
    add_department('Editing', 99, 1800); X
    add_department('Advertising', 101, 1800); ✓
END;
```

# Exceptions Not Handled

Calling procedure

```
PROCEDURE  
  PROC1 ...  
IS  
  ...  
BEGIN  
  ...  
  PROC2 (arg1);  
  ...  
EXCEPTION  
  ...  
END PROC1;
```

Called procedure

```
PROCEDURE  
  PROC2 ...  
IS  
  ...  
BEGIN  
  ...  
EXCEPTION  
  ...  
END PROC2;
```

Exception raised  
Exception not handled

Control returned  
to exception  
section of calling  
procedure

# Exceptions Not Handled: Example

```
SET SERVEROUTPUT ON
CREATE PROCEDURE add_department_noex(
    p_name VARCHAR2, p_mgr NUMBER, p_loc NUMBER) IS
BEGIN
    INSERT INTO DEPARTMENTS (department_id,
        department_name, manager_id, location_id)
    VALUES (DEPARTMENTS_SEQ.NEXTVAL, p_name, p_mgr, p_loc);
    DBMS_OUTPUT.PUT_LINE('Added Dept: '|| p_name);
END;
```

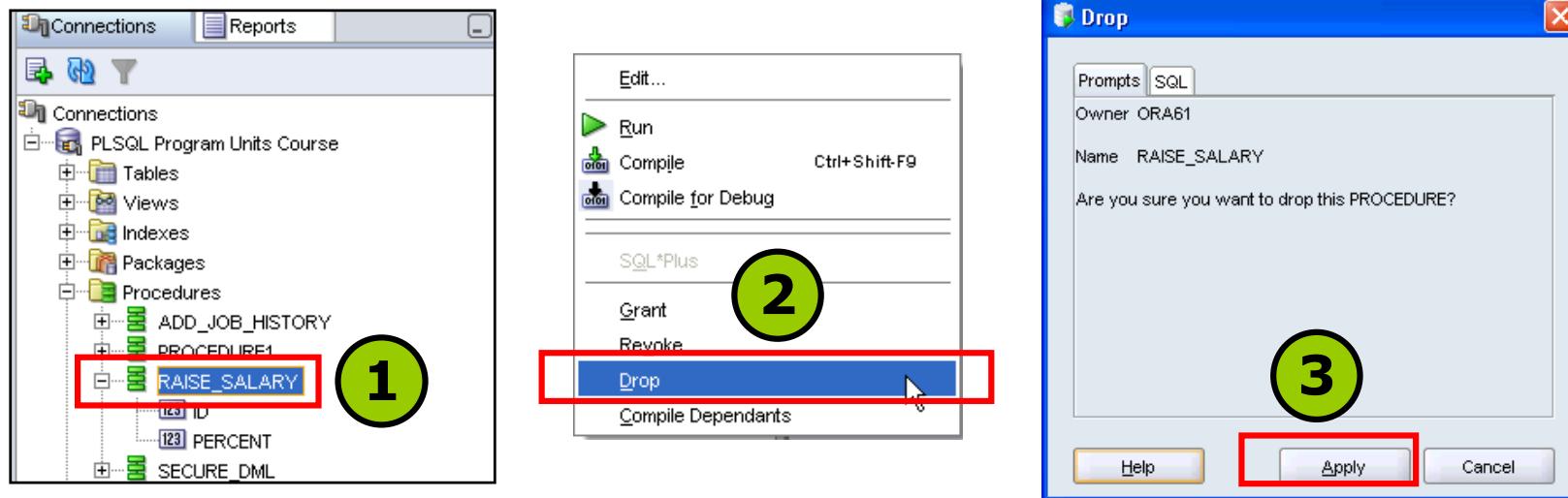
```
CREATE PROCEDURE create_departments_noex IS
BEGIN
    add_department_noex('Media', 100, 1800);
    add_department_noex('Editing', 99, 1800);
    add_department_noex('Advertising', 101, 1800); X
END;
```

# Removing Procedures: Using the DROP SQL Statement or SQL Developer

- Using the DROP statement:

```
DROP PROCEDURE raise_salary;
```

- Using SQL Developer:



# Viewing Procedure Information Using the Data Dictionary Views

```
DESCRIBE user_source
```

DESCRIBE user_source		
Name	Null	Type
NAME		VARCHAR2(30)
TYPE		VARCHAR2(12)
LINE		NUMBER
TEXT		VARCHAR2(4000)
4 rows selected		

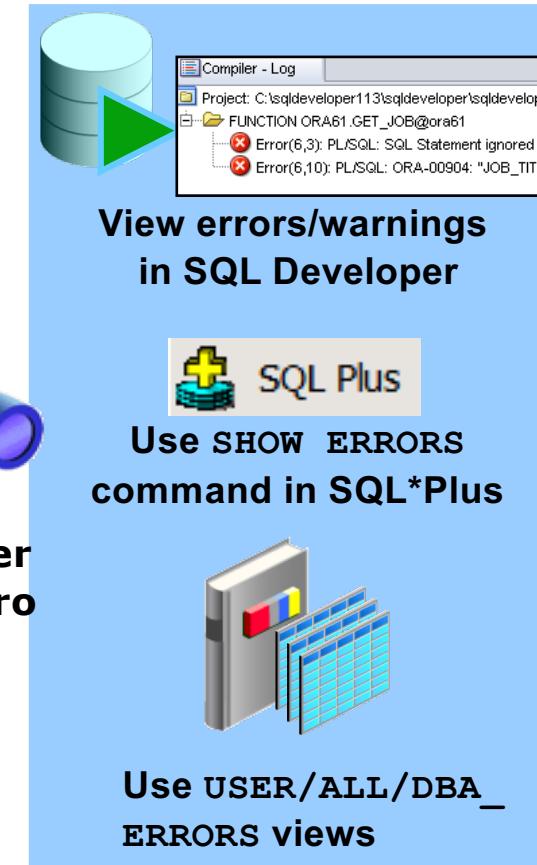
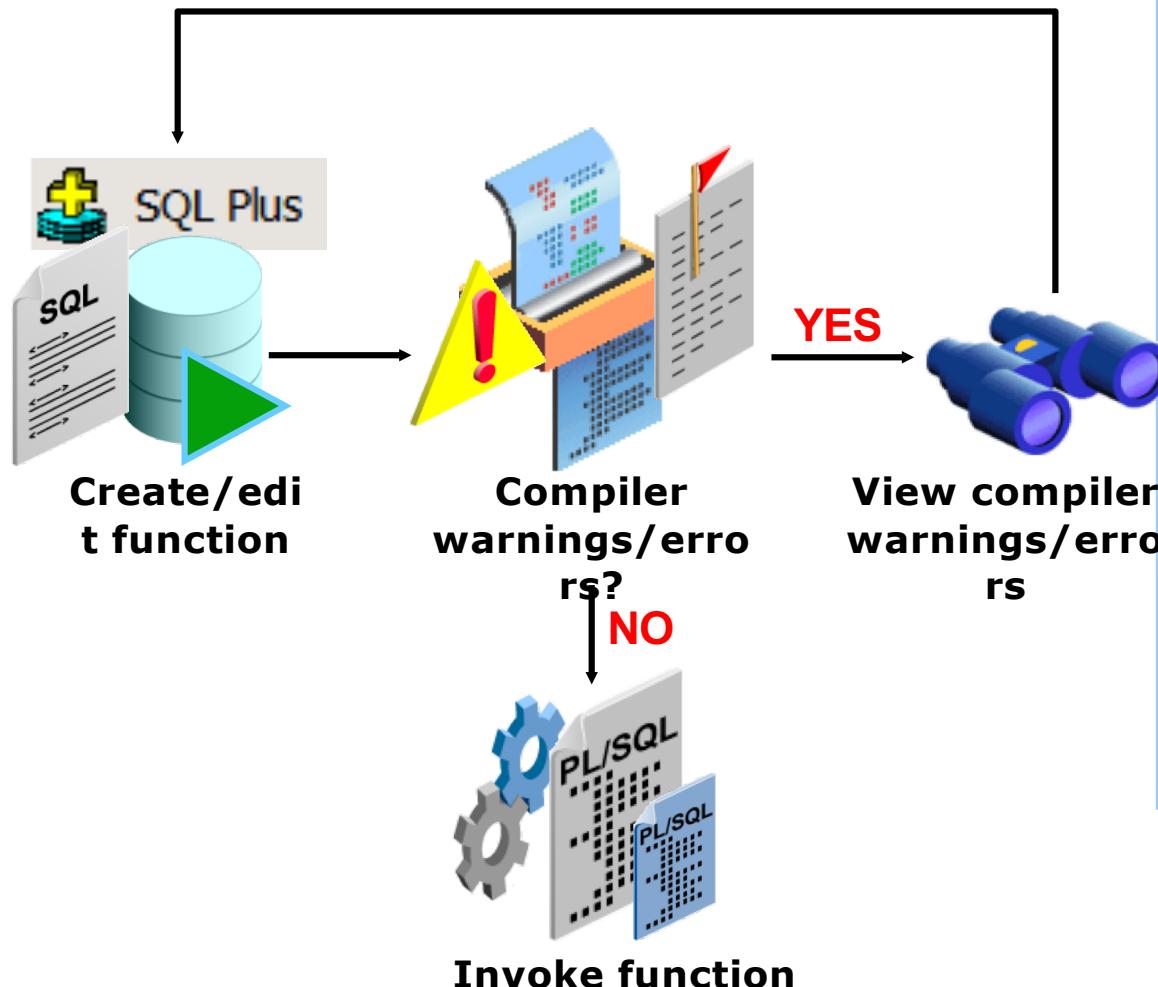
```
SELECT text
FROM   user_source
WHERE  name = 'ADD_DEPT' AND type = 'PROCEDURE'
ORDER BY line;
```

TEXT
1 PROCEDURE add_dept(
2 p_name IN departments.department_name%TYPE,
3 p_loc IN departments.location_id%TYPE) IS
4
5 BEGIN
6 INSERT INTO departments(department_id, department_name, location_id)
7 VALUES (departments_seq.NEXTVAL, p_name, p_loc);
8 END add_dept;

# Exercice

- a. Create a procedure called `CHECK_SALARY` as follows:**
  - i. The procedure accepts two parameters, one for an employee's job ID string and the other for the salary.
  - ii. The procedure uses the job ID to determine the minimum and maximum salary for the specified job.
  - iii. If the salary parameter does not fall within the salary range of the job, inclusive of the minimum and maximum, then it should raise an application exception, with the message “Invalid salary <sal>. Salaries for job <jobid> must be between <min> and <max>”.
- b. Create a procedure called `PROCESS_CHECK_SALARY` to check salary for all employees .**

# Creating and Running Functions: Overview



# Function: Syntax

```
CREATE [OR REPLACE] FUNCTION function_name
[ (argument1 [mode1] datatype1,
  argument2 [mode2] datatype2,
  . . .)]
RETURN datatype
IS|AS
function_body;
```

# Function: Example

```
CREATE FUNCTION check_sal RETURN Boolean IS
v_dept_id employees.department_id%TYPE;
v_empno   employees.employee_id%TYPE;
v_sal      employees.salary%TYPE;
v_avg_sal employees.salary%TYPE;
BEGIN
  v_empno:=205;
  SELECT salary,department_id INTO v_sal,v_dept_id FROM
employees
  WHERE employee_id= v_empno;
  SELECT avg(salary) INTO v_avg_sal FROM employees WHERE
department_id=v_dept_id;
  IF v_sal > v_avg_sal THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN NULL;
END;
```

# Creating Functions

The PL/SQL block must have at least one RETURN statement.

```
CREATE [OR REPLACE] FUNCTION function_name
[ (parameter1 [mode1] datatype1, . . . .) ]
RETURN datatype IS|AS
[local_variable_declarations;
 . . . .]
BEGIN
-- actions;
RETURN expression;
END [function_name];
```

PL/SQL Block

# Passing a Parameter to the Function

```
DROP FUNCTION check_sal;
CREATE FUNCTION check_sal(p_empno employees.employee_id%TYPE)
RETURN Boolean IS
    v_dept_id employees.department_id%TYPE;
    v_sal      employees.salary%TYPE;
    v_avg_sal  employees.salary%TYPE;
BEGIN
    SELECT salary,department_id INTO v_sal,v_dept_id FROM employees
        WHERE employee_id=p_empno;
    SELECT avg(salary) INTO v_avg_sal FROM employees
        WHERE department_id=v_dept_id;
    IF v_sal > v_avg_sal THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END IF;
EXCEPTION
    ...

```

# Invoking the Function with a Parameter

```
BEGIN
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 205');
IF (check_sal(205) IS NULL) THEN
DBMS_OUTPUT.PUT_LINE('The function returned
NULL due to exception');
ELSIF (check_sal(205)) THEN
DBMS_OUTPUT.PUT_LINE('Salary > average');
ELSE
DBMS_OUTPUT.PUT_LINE('Salary < average');
END IF;
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 70');
IF (check_sal(70) IS NULL) THEN
DBMS_OUTPUT.PUT_LINE('The function returned
NULL due to exception');
ELSIF (check_sal(70)) THEN
...
END IF;
END;
/
```

# Using Different Methods for Executing Functions

```
-- As a PL/SQL expression, get the results using host variables

VARIABLE b_salary NUMBER
EXECUTE :b_salary := get_sal(100)
```

```
anonymous block completed
b_salary
-----
24000
```

```
-- As a PL/SQL expression, get the results using a local
-- variable
```

```
DECLARE
    sal employees.salary%type;
BEGIN
    sal := get_sal(100);
    DBMS_OUTPUT.PUT_LINE('The salary is: '|| sal);
END;/
```

```
anonymous block completed
The salary is: 24000
```

# Using Different Methods for Executing Functions

```
-- Use as a parameter to another subprogram
```

```
EXECUTE dbms_output.put_line(get_sal(100))
```

```
anonymous block completed  
24000
```

```
-- Use in a SQL statement (subject to restrictions)
```

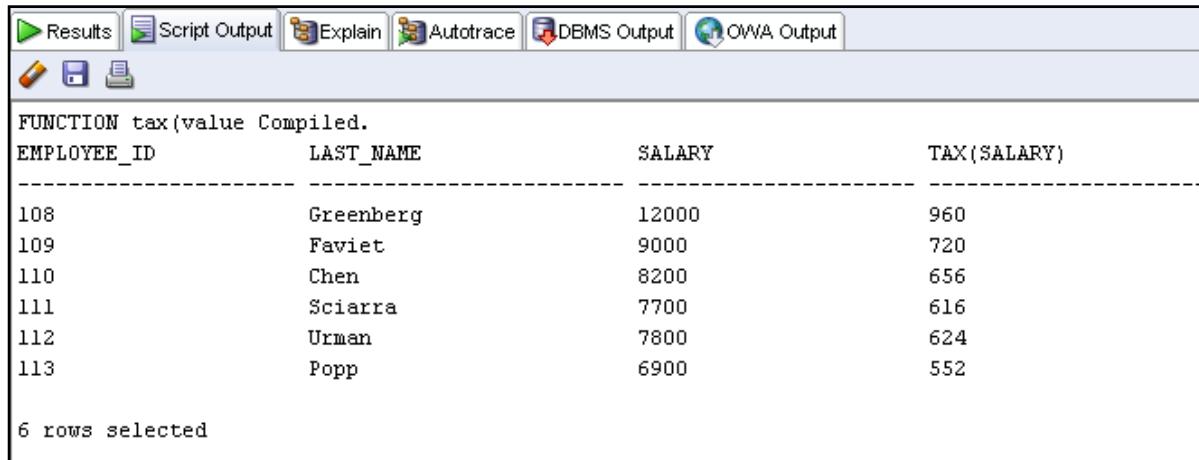
```
SELECT job_id, get_sal(employee_id) FROM employees;
```

JOB_ID	GET_SAL(EMPLOYEE_ID)
SH_CLERK	2600
SH_CLERK	2600
AD_ASST	4400
MK_MAN	13000

```
...  
SH_CLERK 3100  
SH_CLERK 3000  
107 rows selected
```

# Using a Function in a SQL Expression: Example

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
  RETURN NUMBER IS
BEGIN
  RETURN (p_value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 100;
```



The screenshot shows the Oracle SQL Developer interface with the 'Results' tab selected. The query has been executed, displaying the results of the SELECT statement. The results are as follows:

EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
108	Greenberg	12000	960
109	Faviet	9000	720
110	Chen	8200	656
111	Sciarra	7700	616
112	Urman	7800	624
113	Popp	6900	552

6 rows selected

# Viewing Functions Using Data Dictionary Views

```
DESCRIBE USER_SOURCE
```

DESCRIBE user_source		
Name	Null	Type
NAME		VARCHAR2(30)
TYPE		VARCHAR2(12)
LINE		NUMBER
TEXT		VARCHAR2(4000)

4 rows selected

```
SELECT    text
FROM      user_source
WHERE     type = 'FUNCTION'
ORDER BY  line;
```

Results	Script Output	Explain	Autotrace	DBMS Output	OWA
Results:					
TEXT					
1	FUNCTION tax(p_value IN NUMBER)				
2	FUNCTION query_call_sql(p_a NUMBER) RETURN NUMBER IS				
3	FUNCTION get_sal				
4	FUNCTION dml_call_sql(p_sal NUMBER)				
5	RETURN NUMBER IS				
6	RETURN NUMBER IS				
7	(p_id employees.employee_id%TYPE) RETURN NUMBER IS				
8	v_s NUMBER;				

# Exercice

2. Create a function called GET\_ANNUAL\_COMP to return the annual salary computed from an employee's monthly salary and commission passed as parameters.
  - a. Create the GET\_ANNUAL\_COMP function, which accepts parameter values for the monthly salary and commission. Either or both values passed can be NULL, but the function should still return a non-NULL annual salary. Use the following basic formula to calculate the annual salary:  
$$(\text{salary} * 12) + (\text{commission\_pct} * \text{salary} * 12)$$
  - b. Use the function in a SELECT statement against the EMPLOYEES table for employees in department 30.

EMPLOYEE_ID	LAST_NAME	Annual Compensation
114	Raphaely	132000
115	Khoo	37200
116	Baida	34800
117	Tobias	33600
118	Himuro	31200
119	Colmenares	30000

6 rows selected

# **WORKING WITH PACKAGES**

# Overloading Procedures Example: Creating the Package Specification

```
CREATE OR REPLACE PACKAGE dept_pkg IS
    PROCEDURE add_department
        (p_deptno departments.department_id%TYPE,
         p_name  departments.department_name%TYPE := 'unknown',
         p_loc   departments.location_id%TYPE := 1700);

    PROCEDURE add_department
        (p_name  departments.department_name%TYPE := 'unknown',
         p_loc   departments.location_id%TYPE := 1700);
END dept_pkg;
/
```

# Overloading Procedures Example: Creating the Package Body

```
CREATE OR REPLACE PACKAGE BODY dept_pkg IS
PROCEDURE add_department -- First procedure's declaration
(p_deptno departments.department_id%TYPE,
 p_name   departments.department_name%TYPE := 'unknown',
 p_loc    departments.location_id%TYPE := 1700) IS
BEGIN
    INSERT INTO departments(department_id,
                           department_name, location_id)
    VALUES  (p_deptno, p_name, p_loc);
END add_department;

PROCEDURE add_department -- Second procedure's declaration
(p_name   departments.department_name%TYPE := 'unknown',
 p_loc    departments.location_id%TYPE := 1700) IS
BEGIN
    INSERT INTO departments (department_id,
                           department_name, location_id)
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_department;
END dept_pkg; /
```

# Examples of Some Oracle-Supplied Packages

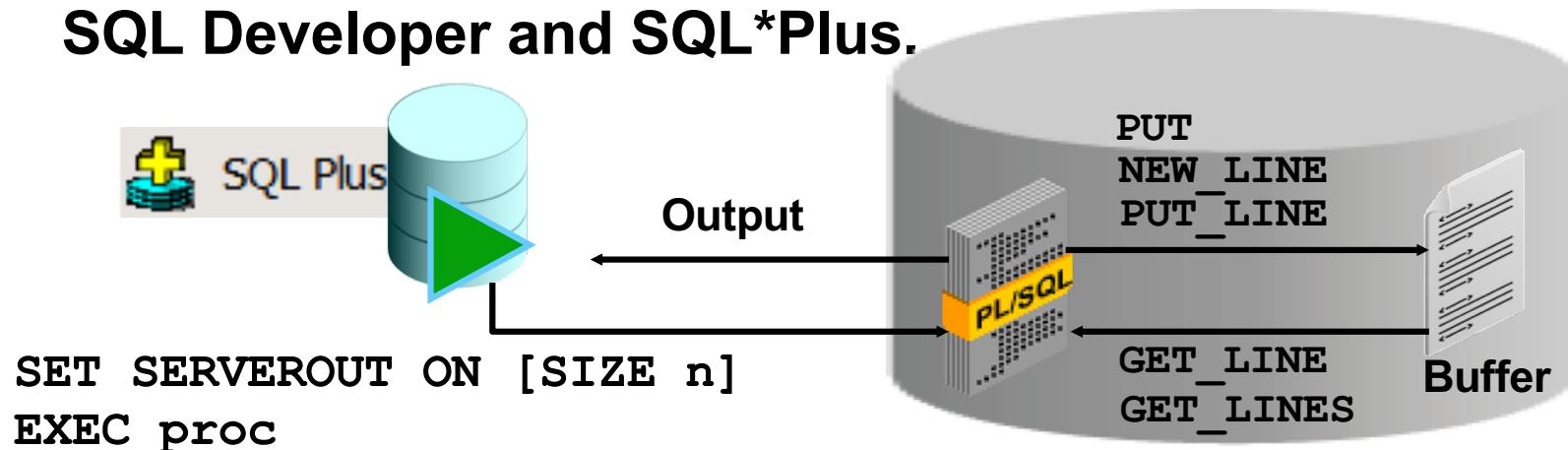
Here is an abbreviated list of some Oracle-supplied packages:

- DBMS\_OUTPUT
- UTL\_FILE
- UTL\_MAIL
- DBMS\_ALERT
- DBMS\_LOCK
- DBMS\_SESSION
- HTP
- DBMS\_SCHEDULER

# How the DBMS\_OUTPUT Package Works

The DBMS\_OUTPUT package enables you to send messages from stored subprograms and triggers.

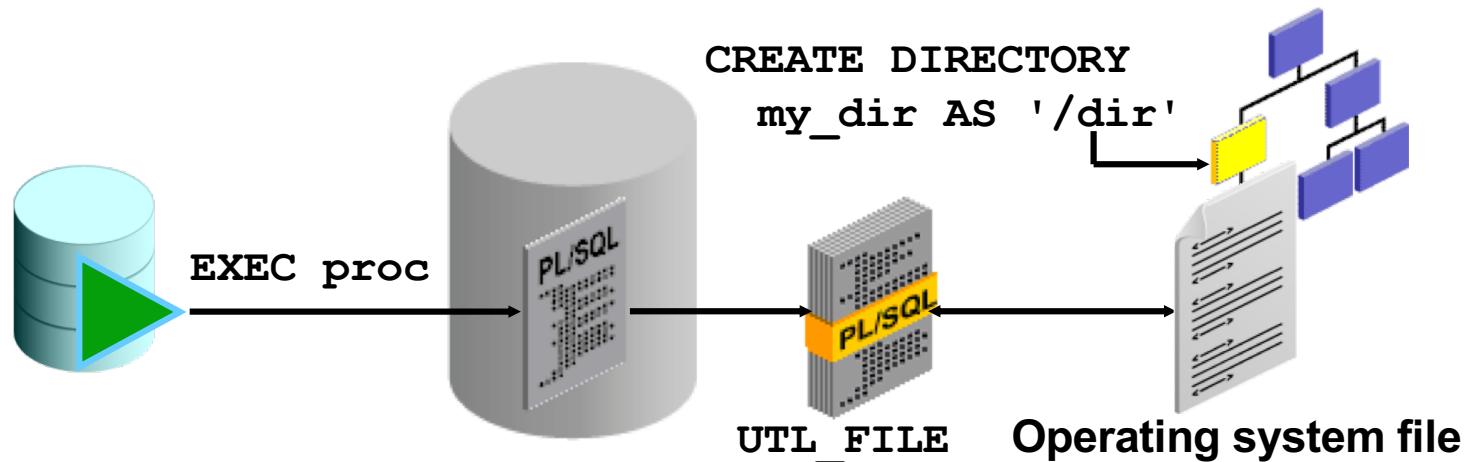
- PUT and PUT\_LINE place text in the buffer.
- GET\_LINE and GET\_LINES read the buffer.
- Messages are not sent until the sending subprogram or trigger completes.
- Use SET SERVEROUTPUT ON to display messages in SQL Developer and SQL\*Plus.



# Using the UTL\_FILE Package to Interact with Operating System Files

The **UTL\_FILE** package extends PL/SQL programs to read and write operating system text files:

- Provides a restricted version of operating system stream file I/O for text files
- Can access files in operating system directories defined by a CREATE DIRECTORY statement



# Using UTL\_FILE: Example

```
CREATE OR REPLACE PROCEDURE sal_status(
    p_dir IN VARCHAR2, p_filename IN VARCHAR2) IS
    f_file UTL_FILE.FILE_TYPE;
CURSOR cur_emp IS
    SELECT last_name, salary, department_id
        FROM employees ORDER BY department_id;
    v_newdeptno employees.department_id%TYPE;
    v_olddeptno employees.department_id%TYPE := 0;
BEGIN
    f_file:= UTL_FILE.FOPEN (p_dir, p_filename, 'W');
    UTL_FILE.PUT_LINE (f_file,
        'REPORT: GENERATED ON ' || SYSDATE);
    UTL_FILE.NEW_LINE (f_file);
    . . .

```

# Using UTL\_FILE: Example

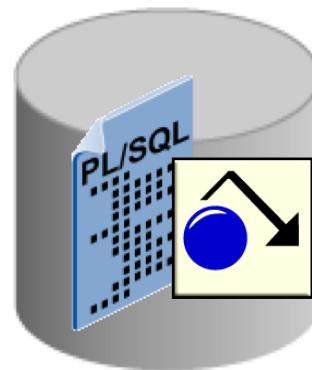
```
    . . .
FOR emp_rec IN cur_emp LOOP
    IF emp_rec.department_id <> v_olddeptno THEN
        UTL_FILE.PUT_LINE (f_file,
            'DEPARTMENT: ' || emp_rec.department_id);
        UTL_FILE.NEW_LINE (f_file);
    END IF;
    UTL_FILE.PUT_LINE (f_file,
        'EMPLOYEE: ' || emp_rec.last_name ||
        ' earns: ' || emp_rec.salary);
    v_olddeptno := emp_rec.department_id;
    UTL_FILE.NEW_LINE (f_file);
END LOOP;
UTL_FILE.PUT_LINE(f_file,'*** END OF REPORT ***');
UTL_FILE.FCLOSE (f_file);

EXCEPTION
    WHEN UTL_FILE.INVALID_FILEHANDLE THEN
        RAISE_APPLICATION_ERROR(-20001,'Invalid File.');
    WHEN UTL_FILE.WRITE_ERROR THEN
        RAISE_APPLICATION_ERROR (-20002, 'Unable to write to file');
END sal_status;/
```

# **CREATING TRIGGERS**

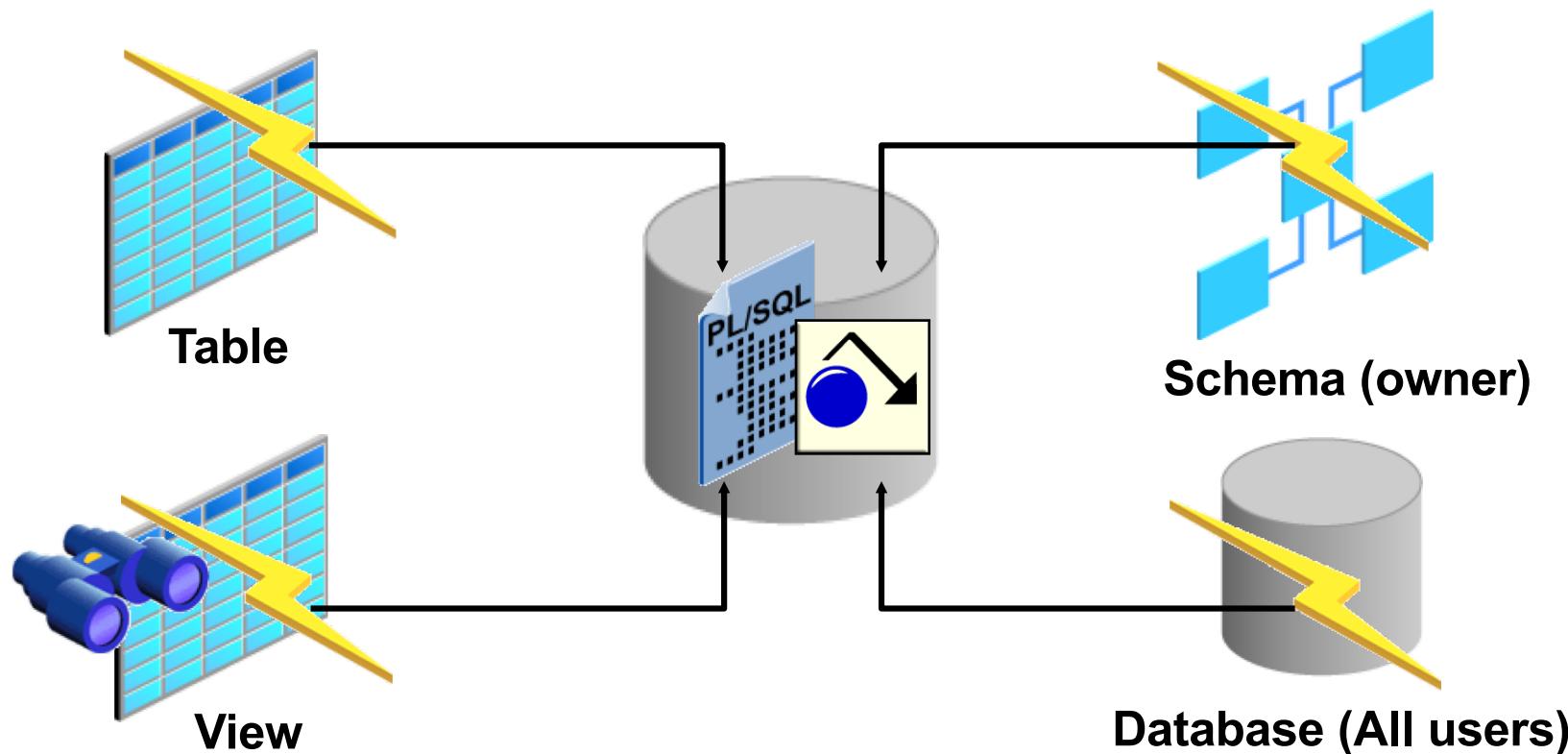
# What Are Triggers?

- A trigger is a PL/SQL block that is stored in the database and fired (executed) in response to a specified event.
- The Oracle database automatically executes a trigger when specified conditions occur.



# Defining Triggers

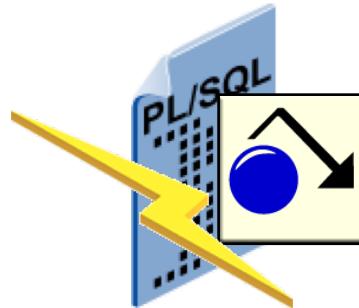
A trigger can be defined on the table, view, schema (schema owner), or database (all users).



# Trigger Event Types

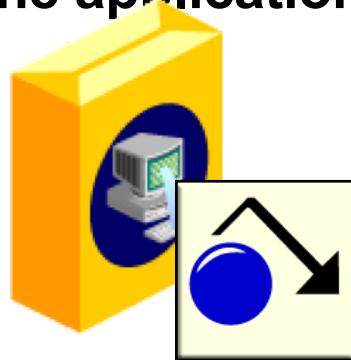
**Vous pouvez écrire des triggers qui se déclenchent lorsqu'une des opérations suivantes se produit dans la base de données :**

- **Une manipulation de la base de données (DML) (DELETE, INSERT, or UPDATE).**
- **Une requête de définition de base de données (DDL) (CREATE, ALTER, or DROP).**
- **Une opération de base de données tels que SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN.**

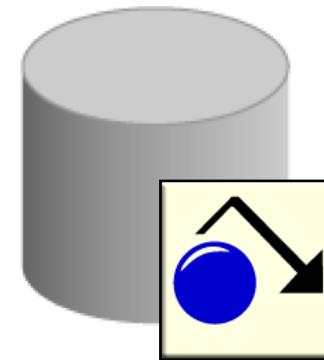


# Application and Database Triggers

- **Database trigger**
  - Se déclenchent chaque fois qu'un événement DML, DLL ou système se produit sur une base de données ou un schéma
- **Application trigger:**
  - Se déclenchant si un événement se produit dans une application particulière



Application Trigger



Database Trigger

# **Business Application Scenarios for Implementing Triggers**

**You can use triggers for:**

- **Security**
- **Auditing**
- **Data integrity**
- **Referential integrity**
- **Table replication**
- **Computing derived data automatically**
- **Event logging**

# Available Trigger Types

- **Simple DML triggers**
  - BEFORE
  - AFTER
  - INSTEAD OF
- **Compound triggers**
- **Non-DML triggers**
  - DDL event triggers
  - Database event triggers

# Trigger Event Types and Body

- Le type de déclencheur détermine quelle instruction DML provoque l'exécution du trigger. Les événements possibles sont :
  - INSERT
  - UPDATE [OF column]
  - DELETE
- Le corps de déclencheur détermine quelle action est exécutée et est un bloc PL/SQL ou un appel d'une procédure

# Creating DML Triggers Using the CREATE TRIGGER Statement

```
CREATE [OR REPLACE] TRIGGER trigger_name  
  timing -- when to fire the trigger  
  event1 [OR event2 OR event3]  
  ON object_name  
  [REFERENCING OLD AS old | NEW AS new]  
  FOR EACH ROW -- default is statement level trigger  
  WHEN (condition) ]]  
  DECLARE]  
  BEGIN  
    ... trigger_body -- executable statements  
  [EXCEPTION . . .]  
  END [trigger_name];
```

timing = BEFORE | AFTER | INSTEAD OF

event = INSERT | DELETE | UPDATE | UPDATE OF column\_list

# Specifying the Trigger Firing (Timing)

**Vous pouvez spécifier le moment de déclenchement quant à l'exécution de l'action avant ou après l'instruction de déclenchement :**

- **BEFORE:** Exécute le corps du déclencheur avant l'événement de déclencheur DML sur une table.
- **AFTER:** Exécuter le corps de déclencheur après l'événement de déclencheur DML sur une table.
- **INSTEAD OF:** Exécuter le corps de déclencheur au lieu de l'instruction de déclenchement. Ceci est utilisé pour les vues qui ne sont pas modifiables.

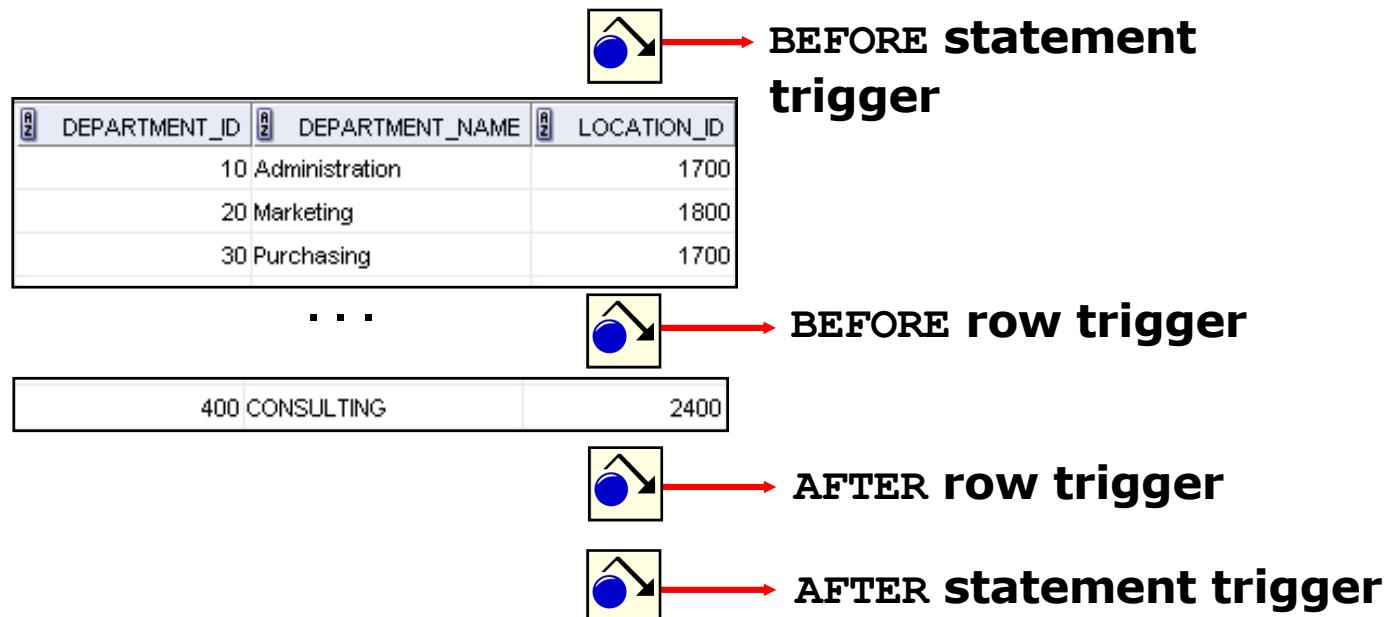
# **Statement-Level Triggers Versus Row-Level Triggers**

<b>Statement-Level Triggers</b>	<b>Row-Level Triggers</b>
Is the default when creating a trigger	Use the FOR EACH ROW clause when creating a trigger.
Fires once for the triggering event	Fires once for each row affected by the triggering event
Fires once even if no rows are affected	Does not fire if the triggering event does not affect any rows

# Trigger-Firing Sequence: Single-Row Manipulation

Use the following firing sequence for a trigger on a table when a single row is manipulated:

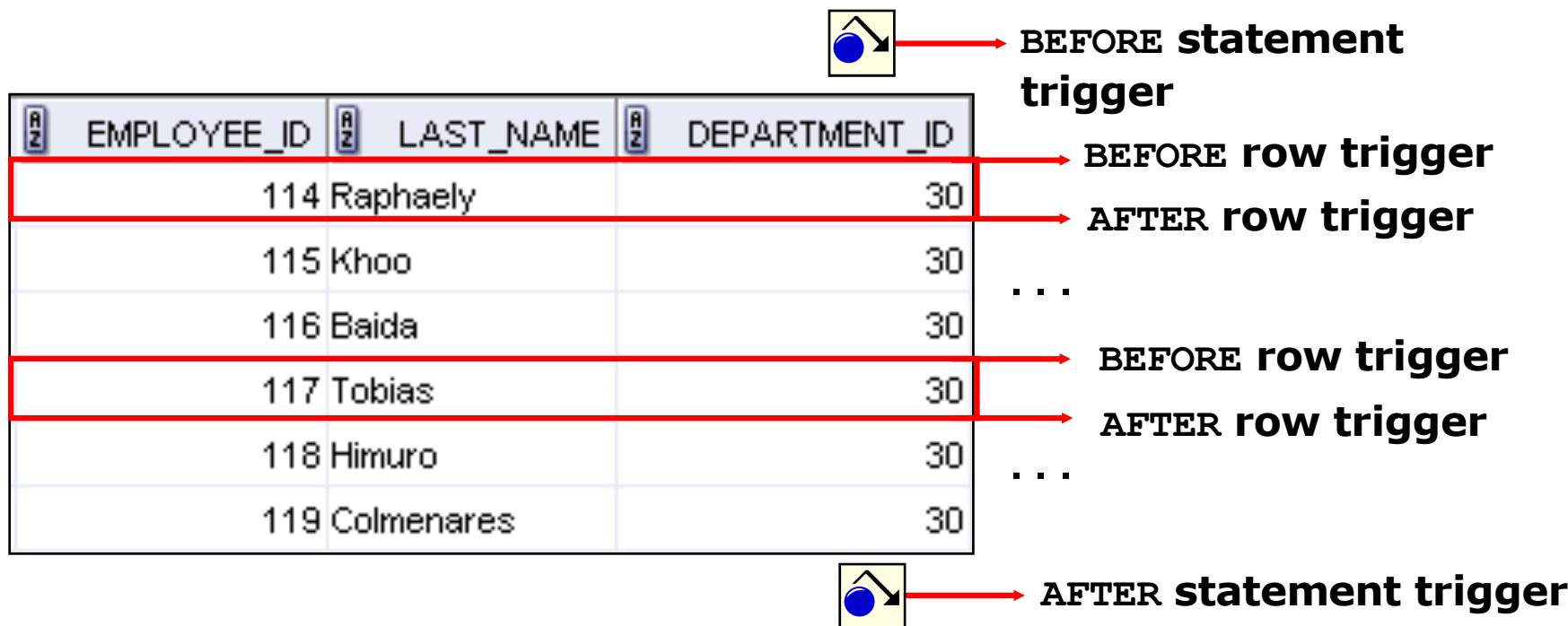
```
INSERT INTO departments
  (department_id, department_name, location_id)
VALUES (400, 'CONSULTING', 2400);
```



# Trigger-Firing Sequence: Multirow Manipulation

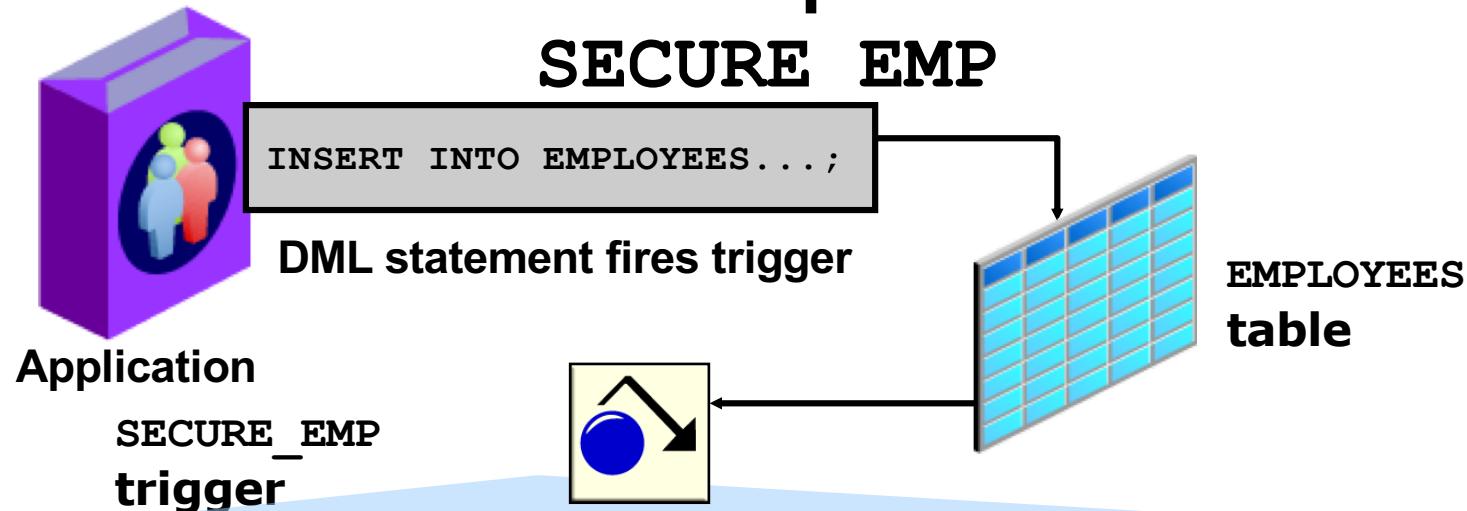
Use the following firing sequence for a trigger on a table when many rows are manipulated:

```
UPDATE employees  
  SET salary = salary * 1.1  
 WHERE department_id = 30;
```



# Creating a DML Statement Trigger

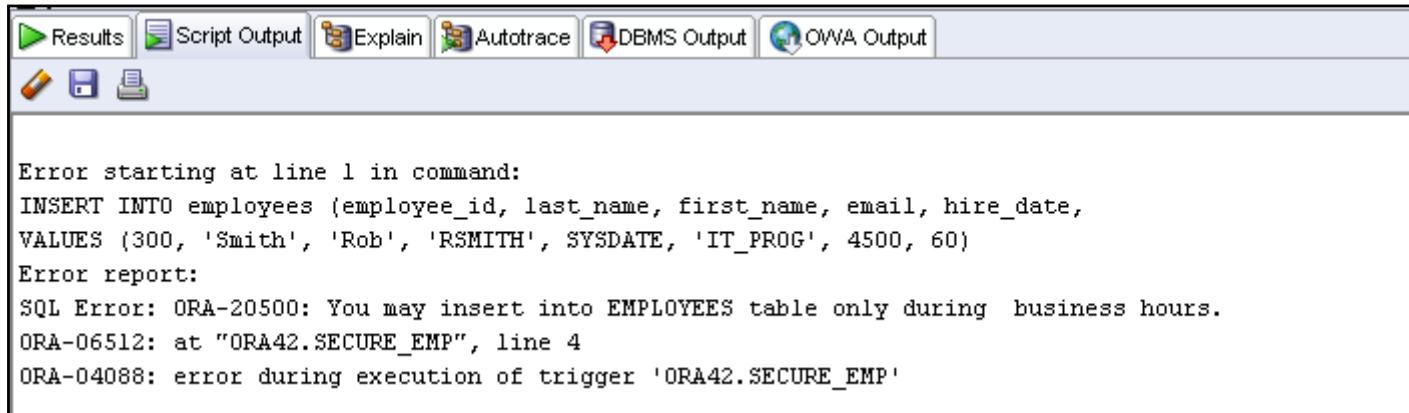
## Example:



```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees
BEGIN
  IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
    (TO_CHAR(SYSDATE,'HH24:MI')
     NOT BETWEEN '08:00' AND '18:00') THEN
    RAISE_APPLICATION_ERROR(-20500, 'You may insert'
    || ' into EMPLOYEES table only during '
    || ' normal business hours.');
  END IF;
END;
```

# Testing Trigger SECURE\_EMP

```
INSERT INTO employees (employee_id, last_name,
    first_name, email, hire_date,
    job_id, salary, department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,
    'IT_PROG', 4500, 60);
```



The screenshot shows the Oracle SQL Developer interface with the 'Results' tab selected. The results window displays the following error message:

```
Error starting at line 1 in command:
INSERT INTO employees (employee_id, last_name, first_name, email, hire_date,
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE, 'IT_PROG', 4500, 60)
Error report:
SQL Error: ORA-20500: You may insert into EMPLOYEES table only during business hours.
ORA-06512: at "ORA42.SECURE_EMP", line 4
ORA-04088: error during execution of trigger 'ORA42.SECURE_EMP'
```

# Using Conditional Predicates

```
CREATE OR REPLACE TRIGGER secure_emp BEFORE
INSERT OR UPDATE OR DELETE ON employees
BEGIN
    IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
        (TO_CHAR(SYSDATE,'HH24')
         NOT BETWEEN '08' AND '18') THEN
        IF DELETING THEN RAISE_APPLICATION_ERROR(
            -20502,'You may delete from EMPLOYEES table'|||
            'only during normal business hours.');
        ELSIF INSERTING THEN RAISE_APPLICATION_ERROR(
            -20500,'You may insert into EMPLOYEES table'|||
            'only during normal business hours.');
        ELSIF UPDATING ('SALARY') THEN
            RAISE_APPLICATION_ERROR(-20503, 'You may '|||
            'update SALARY only normal during business hours.');
        ELSE RAISE_APPLICATION_ERROR(-20504,'You may'|||
            ' update EMPLOYEES table only during'|||
            ' normal business hours.');
        END IF;
    END IF;
END;
```

# Creating a DML Row Trigger

```
CREATE OR REPLACE TRIGGER restrict_salary
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
        AND :NEW.salary > 15000 THEN
        RAISE_APPLICATION_ERROR (-20202,
            'Employee cannot earn more than $15,000.');
    END IF;
END ;/
```

```
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell';
```

```
Error starting at line 1 in command:
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell'
Error report:
SQL Error: ORA-20202: Employee cannot earn more than $15,000.
ORA-06512: at "ORA62.RESTRICT_SALARY", line 4
ORA-04088: error during execution of trigger 'ORA62.RESTRICT_SALARY'
```

# Using OLD and NEW Qualifiers

- When a row-level trigger fires, the PL/SQL run-time engine creates and populates two data structures:
  - OLD: Stores the original values of the record processed by the trigger
  - NEW: Contains the new values
- NEW and OLD have the same structure as a record declared using the %ROWTYPE on the table to which the trigger is attached.

Data Operations	Old Value	New Value
INSERT	NULL	Inserted value
UPDATE	Value before update	Value after update
DELETE	Value before delete	NULL

# Using OLD and NEW Qualifiers: Example

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO audit_emp(user_name, time_stamp, id,
        old_last_name, new_last_name, old_title,
        new_title, old_salary, new_salary)
    VALUES (USER, SYSDATE, :OLD.employee_id,
        :OLD.last_name, :NEW.last_name, :OLD.job_id,
        :NEW.job_id, :OLD.salary, :NEW.salary);
END;
/
```

# Using OLD and NEW Qualifiers: Example Using AUDIT\_EMP

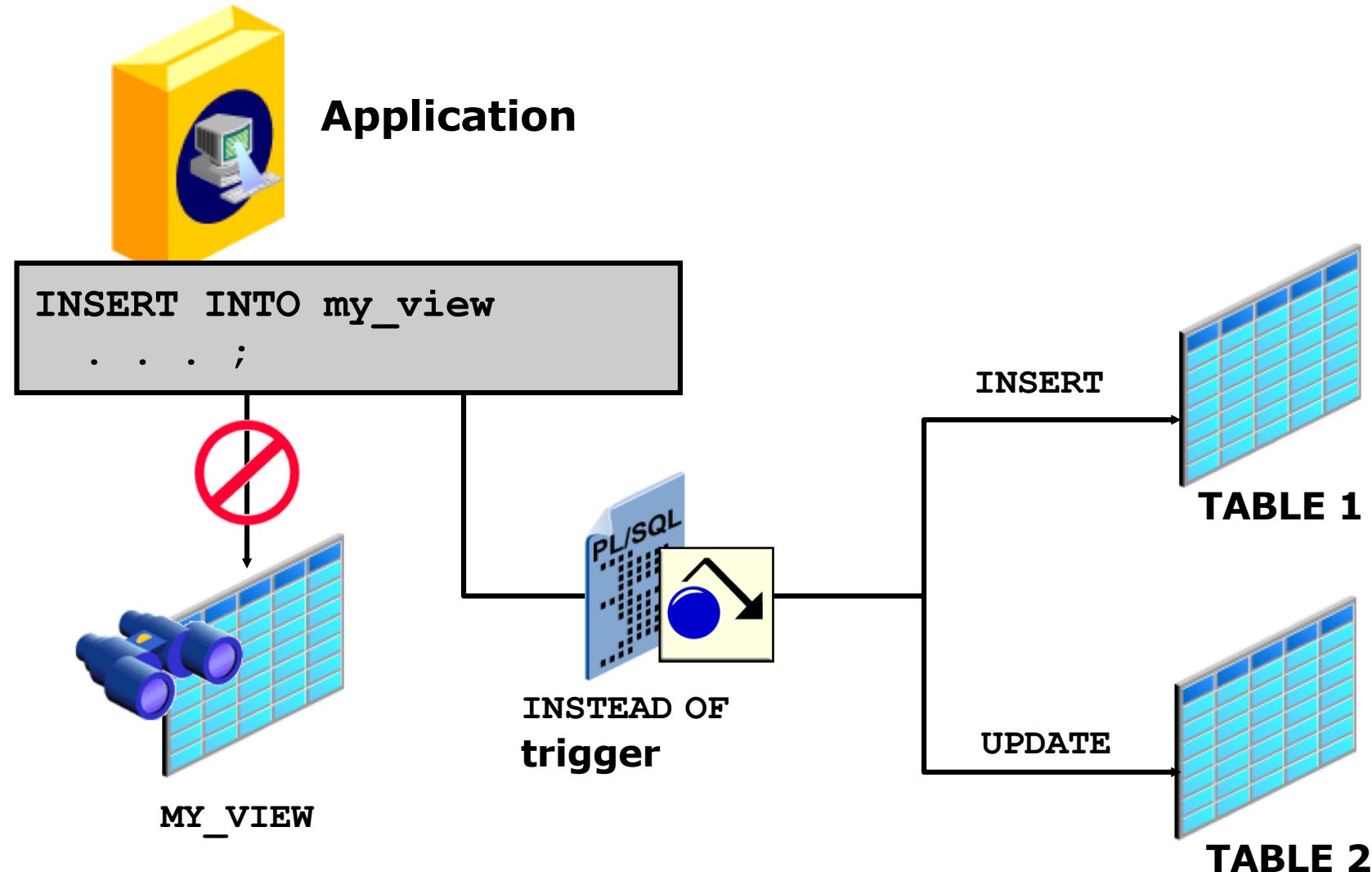
```
INSERT INTO employees (employee_id, last_name, job_id,
salary, email, hire_date)
VALUES (999, 'Temp emp', 'SA REP', 6000, 'TEMPEMP',
TRUNC(SYSDATE));
/
UPDATE employees
  SET salary = 7000, last_name = 'Smith'
 WHERE employee_id = 999;
/
SELECT *
FROM audit_emp;
```

	USER_NAME	TIME_STAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
1	ORA62	27-JUN-07	(null)	(null)	Temp emp	(null)	SA REP	(null)	6000
2	ORA62	27-JUN-07	999	Temp emp	Smith	SA REP	SA REP	6000	7000

# Using the WHEN Clause to Fire a Row Trigger Based on a Condition

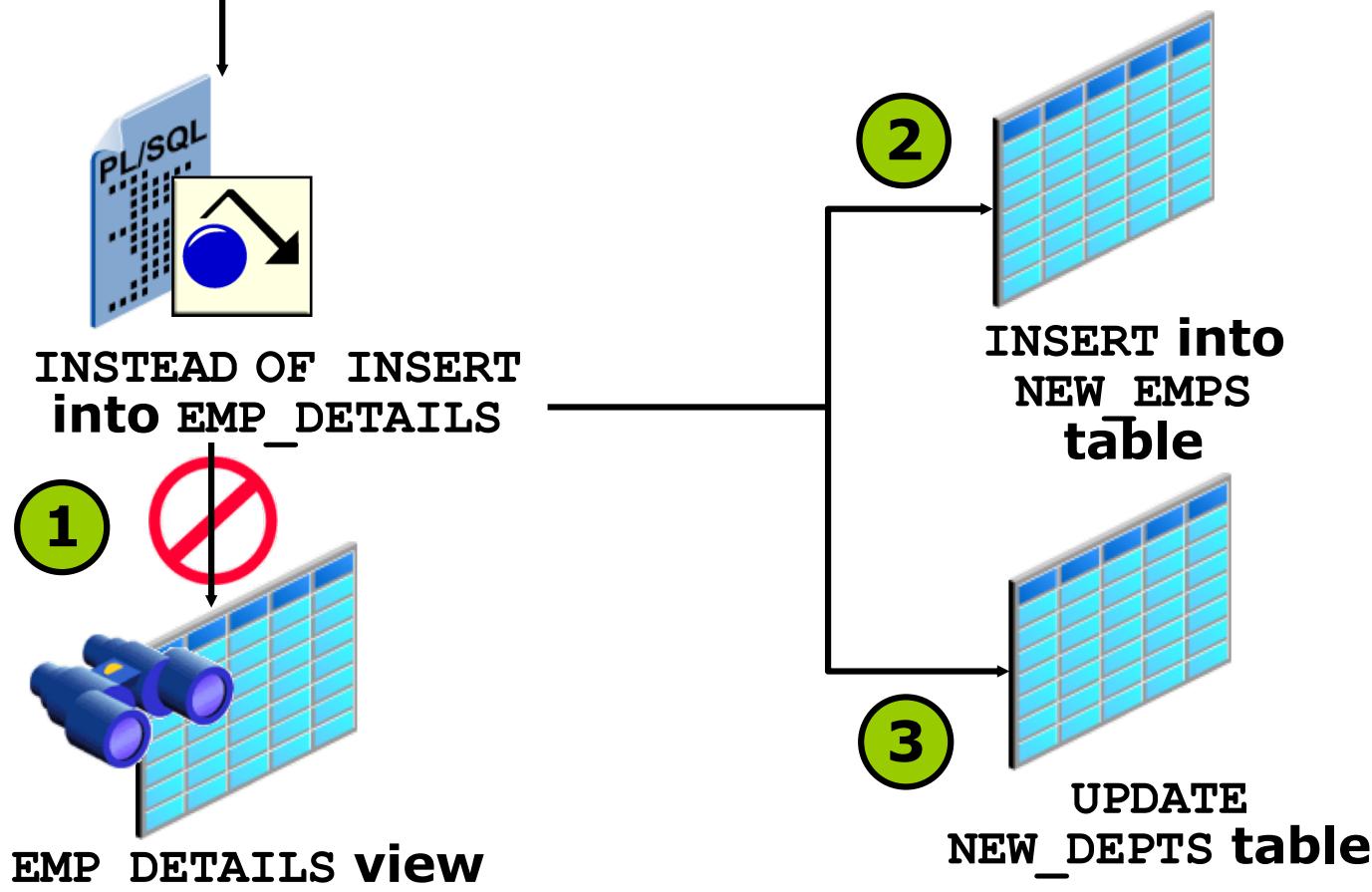
```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.job_id = 'SA_REP')
BEGIN
  IF INSERTING THEN
    :NEW.commission_pct := 0;
  ELSIF :OLD.commission_pct IS NULL THEN
    :NEW.commission_pct := 0;
  ELSE
    :NEW.commission_pct := :OLD.commission_pct+0.05;
  END IF;
END;
/
```

# INSTEAD OF Triggers



# Creating an INSTEAD OF Trigger: Example

```
INSERT INTO emp_details  
VALUES (9001,'ABBOTT',3000, 10, 'Administration');
```



# Creating an INSTEAD OF Trigger to Perform DML on Complex Views

```
CREATE TABLE new_emps AS  
SELECT employee_id, last_name, salary, department_id  
FROM employees;
```

```
CREATE TABLE new_depts AS  
SELECT d.department_id, d.department_name,  
       sum(e.salary) dept_sal  
  FROM employees e, departments d  
 WHERE e.department_id = d.department_id  
GROUP BY d.department_id, d.department_name;
```

```
CREATE VIEW emp_details AS  
SELECT e.employee_id, e.last_name, e.salary,  
       e.department_id, d.department_name, FROM  
      employees e, departments d  
 WHERE e.department_id = d.department_id  
GROUP BY d.department_id, d.department_name;
```

# Creating an INSTEAD OF Trigger to Perform DML on Complex Views

```
CREATE OR REPLACE TRIGGER new_emp_dept
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_details
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        INSERT INTO new_emps
        VALUES (:NEW.employee_id, :NEW.last_name,
                :NEW.salary, :NEW.department_id);
        UPDATE new_depts
        SET dept_sal = dept_sal + :NEW.salary
        WHERE department_id = :NEW.department_id;
    ELSIF DELETING THE
        DELETE FROM new_emps
        WHERE employee_id = :OLD.employee_id;
        UPDATE new_depts
        SET dept_sal = dept_sal - :OLD.salary
        WHERE department_id = :OLD.department_id;
    END IF;
END;
```

```
ELSIF UPDATING ('salary') THEN
    UPDATE new_emps
        SET salary = :NEW.salary
        WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
        SET dept_sal = dept_sal +
                        (:NEW.salary - :OLD.salary)
        WHERE department_id = :OLD.department_id;
ELSIF UPDATING ('department_id') THEN
    UPDATE new_emps
        SET department_id = :NEW.department_id
        WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
        SET dept_sal = dept_sal - :OLD.salary
        WHERE department_id = :OLD.department_id;
    UPDATE new_depts
        SET dept_sal = dept_sal + :NEW.salary
        WHERE department_id = :NEW.department_id;
END IF;
END;
/
```

# The Status of a Trigger

Un déclencheur est défini dans un des deux modes distincts :

- Enabled: The trigger runs its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to true (default).
- Disabled: The trigger does not run its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to true.



# Managing Triggers Using the ALTER and DROP SQL Statements

```
-- Disable or reenable a database trigger:
```

```
ALTER TRIGGER trigger_name DISABLE | ENABLE;
```

```
-- Disable or reenable all triggers for a table:
```

```
ALTER TABLE table_name DISABLE | ENABLE ALL TRIGGERS;
```

```
-- Recompile a trigger for a table:
```

```
ALTER TRIGGER trigger_name COMPILE;
```

```
-- Remove a trigger from the database:
```

```
DROP TRIGGER trigger_name;
```

# Viewing Trigger Information

You can view the following trigger information:

Data Dictionary View	Description
USER_OBJECTS	Displays object information
USER/ALL/DBA_TRIGGERS	Displays trigger information
USER_ERRORS	Displays PL/SQL syntax errors for a trigger

# Using USER\_TRIGGERS

```
DESCRIBE user_triggers
```

Name	Null	Type
TRIGGER_NAME		VARCHAR2(30)
TRIGGER_TYPE		VARCHAR2(16)
TRIGGERING_EVENT		VARCHAR2(227)
TABLE_OWNER		VARCHAR2(30)
BASE_OBJECT_TYPE		VARCHAR2(16)
TABLE_NAME		VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
REFERENCING_NAMES		VARCHAR2(128)
WHEN_CLAUSE		VARCHAR2(4000)
STATUS		VARCHAR2(8)
DESCRIPTION		VARCHAR2(4000)
ACTION_TYPE		VARCHAR2(11)
TRIGGER_BODY		LONG()
CROSSEDITION		VARCHAR2(7)

14 rows selected

```
SELECT trigger_type, trigger_body  
FROM user_triggers  
WHERE trigger_name = 'SECURE_EMP';
```

# Exercises

- 1. Create a trigger called CHECK\_SALARY\_TRG on the EMPLOYEES table that fires before an INSERT or UPDATE operation on each row:**
  - i. The trigger must call the CHECK\_SALARY procedure to carry out the business logic.
  - ii. The trigger should pass the new job ID and salary to the procedure parameters.
- 2. Update the CHECK\_SALARY\_TRG trigger to fire only when the job ID or salary values have actually changed.**
  - a. **Implement the business rule using a WHEN clause to check whether the JOB\_ID or SALARY values have changed.**

**Note:** Make sure that the condition handles the NULL in the OLD.column\_name values if an INSERT operation is performed; otherwise, an an INSERT operation will fail.

# Exercises

3. You are asked to prevent employees from being deleted during business hours.

**Write a statement trigger called  
DELETE\_EMP\_TRG on the EMPLOYEES table to  
prevent rows from being deleted during weekday  
business hours, which are from 9:00 AM through  
6:00 PM.**