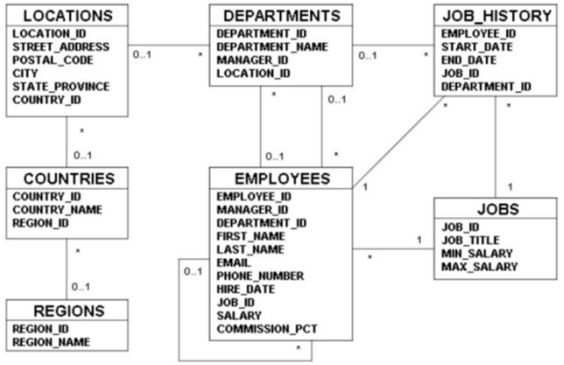
INFO 0505
Bases de données concepts avancés – Partie 2

Exemple de BD Oracle

- La base Oracle HR : classique disponible sous SQL Live via le schéma HR



Compléments SQL

MERGE

- Permet de faire une insertion ou une màj conditionnelle
- Réalise
  - Une màj si le n-upplet existe
  - Une insertion si c'est un nouveau n-upplet
- Evite les màj séparées
- Augmente les performances
- Utilisé dans les applications de data warehouse

Syntaxe

```
MERGE INTO nom_table table_alias
USING (table|view|sub_query) alias
ON (join condition)
WHEN MATCHED THEN
UPDATE SET
col1 = col_val1,
col2 = col2_val
WHEN NOT MATCHED THEN
INSERT (column_list)
VALUES (column_values);
```


Exemple

```
MERGE INTO copy_emp c
USING employees e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
...
c.department_id = e.department_id
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
e.department_id);
```


Jointure

- Syntaxe :

```
SELECT   table1.Attribut1, table2.AttributP
FROM     table1, table2
WHERE    table1.AttributK
CONDITION
table2.AttributM;
```

- Jointure : Condition AdHoc
- Equijointure : =
- Jointure Naturelle : = avec attributs identiques

Jointure

- 2 relations R1 et R2
- R3= R1 ⋈ R2 sous la condition Ak op Bq
- **SQL « Standard »:**

  SELECT …
  FROM R1, R2
  WHERE R1.Ak op R2.Bq
  - Equijointure:
    - op: =
  - Si jointure naturelle:
    - op: =
    - **Et** Ak Ξ Bq

Jointure

- Syntaxe générale SQL2

```
SELECT      table1.att1, table2.att2
FROM  table1
[NATURAL JOIN table2] |
[JOIN table2 USING (attK)] |
[JOIN table2
  ON (table1.attP = table2.attY)]|
[LEFT|RIGHT|FULL OUTER JOIN table2
  ON (table1.attM = table2.attN)]|
[CROSS JOIN table2];
```


Jointure

- **Syntaxe SQL2**

```
SELECT  …
FROM R1 JOIN  R2
ON R1.Ak  op  R2.Bq
```

- Rmq: jointure naturelle possible
- Double jointure :

```
SELECT  …
FROM R1
JOIN  R2
ON R1.Ak  op  R2.Bq
JOIN …
ON…
```


Jointure naturelle

- **Syntaxe SQL 2:**

```
SELECT …
FROM R1
NATURAL JOIN R2;
```


Jointure avec condition

- Exemple

  Tables `EMPLOYEES` et `JOB_GRADES`

```
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;
```

Jointure externe

- Ex. Dans l'exemple suivant, un département qui n'a pas d'employé n'apparaîtra pas :

```
SELECT nomEmploye, nomDept
FROM employe, dept
WHERE employe.numerodept = dept.numero dept
```

- Si on veut qu'il apparaisse, on doit utiliser une jointure externe

Jointure externe

- **Syntaxe SQL 2:**

```
SELECT …
FROM  R1 [NATURAL]
{FULL | LEFT | RIGHT}  OUTER JOIN R2
ON …;
```

- Exemple:

```
select nomEmploye, nomDept
from employe LEFT OUTER JOIN dept
ON employe.numerodept = dept.numerodept
```


Exemples

EMP

NOEMP	NODEPT	SAL
T	1	10 000
J	2	20 000
K	3	15 000

DEPT

NODEPT	NOMDEP
1	Commande
2	Admin
4	Usine

Exemple

- EMP NATURAL JOIN DEPT

NOEMP	NODEPT	SAL	NOMDEP
T	1	10 000	Commande
J	2	20 000	Admin

- EMP NATURAL FULL OUTER JOIN DEPT

NOEMP	NODEPT	SAL	NOMDEP
T	1	10 000	Commande
J	2	20 000	Admin
K	3	15 000	NULL
NULL	4	NULL	Usine

Remarques

- Préfixes de tables
  - Obligatoires si les noms de colonnes sont ambigus
  - Améliorent les performances sauf si les noms de table sont longs → utiliser des alias de table
  - Exemple

```
SELECT *
FROM R1 r, R2 s
WHERE r.Att1=s.Att2
```


Syntaxe USING

- Pour une équijointure
- Syntaxe

```
SELECT …
FROM R1  JOIN  R2
USING (Attribut1)
```


Syntaxe USING

- Alias de colonne obligatoire si ambiguité

```
SELECT r.AttributP, …
FROM R1 r JOIN  R2 s
USING (Attribut1)
```


Syntaxe USING

- Attention : pas d'alias de colonne pour l'attribut du USING

```
SELECT r.AttributK, s.AttributN
FROM R1 r JOIN  R2 s
USING (Attribut1)
WHERE s.Attribut1=…
```

```
SELECT r.AttributK, s.AttributN
FROM R1 r JOIN  R2 s
USING (Attribut1)
WHERE Attribut1=…
```

Restreindre avec

- WHERE ou AND

```
SELECT …
FROM R1
JOIN  R2
ON …
[WHERE|AND]   Condition ;
```


Auto-jointure

- Joindre une table avec elle-même
- Exemple

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


Produit Cartésien

- Généré par une jointure
  - Sans condition
  - Où la condition est non valide
- Clause CROSS JOIN

```
SELECT    table1.att1, table2.att2
FROM      table1
CROSS JOIN table2;
```


Créer des jointures croisées

- La clause CROSS JOIN effectue une jointure croisée entre deux tables
- Cela équivaut au produit cartésien des deux tables

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

	LAST_NAME	DEPARTMENT_NAME
1	Abel	Administration
2	Davies	Administration
3	De Haan	Administration
4	Ernst	Administration
5	Fay	Administration
…		
158	Vargas	Contracting
159	Whalen	Contracting
160	Zlotkey	Contracting

Requêtes imbriquées

- Sous-interrogation ramenant 1 ligne, 1 colonne
  - **WHERE expression op (SELECT …)**
    - où op est un des opérateurs de comparaison =, !=, <, >, <=, >=
  - Exemple :

```
SELECT NOME FROM EMP WHERE SAL >= (SELECT SAL FROM EMP WHERE NOME =
'MERCIER')
```


Requêtes imbriquées

- Sous-interrogation ramenant plusieurs lignes
  - **WHERE *expression op* ALL (SELECT …)**
  - **WHERE *expression op* ANY (SELECT …)**
  - **WHERE *expression* IN (SELECT …)**
  - **WHERE *expression* NOT IN (SELECT …)**
  - où op est un des opérateurs de comparaison =,**!=**, <, >, **<=**, **>=**
  - **ANY** : vrai si la comparaison est vraie pour au moins une des valeurs ramenées par le **SELECT**
  - **ALL** : vrai si la comparaison est vraie pour toutes les valeurs
- Ex. `SELECT NOME, SAL FROM EMP  WHERE SAL > ALL (SELECT SAL FROM EMP WHERE DEPT = 30);`

Requêtes imbriquées

- **La Jointure** s'exprime par deux blocs SFW imbriqués
  - SELECT ... FROM ...
    WHERE Attribut1 IN
    (SELECT Attribut1 FROM... WHERE ...)
- **La Différence** s'exprime aussi par deux blocs SFW imbriqués
  - SELECT ... FROM ...
    WHERE Attribut1 NOT IN
    (SELECT Attribut1 FROM... WHERE ...)

Requêtes imbriquées

- Sous-interrogation ramenant 1 ligne de plusieurs colonnes
  - **WHERE *(expr1, expr2,…) op* (SELECT …)**
  - où *op* est = ou **!=** (mais pas <, >, **<=**, **>=**)
  - Exemple :

```
SELECT NOME, POSTE, SAL FROM EMP
WHERE (POSTE, SAL) =
(SELECT POSTE, SAL FROM EMP
WHERE NOME = 'MERCIER');
```

- Rmq: le select renvoie 1 seule ligne

Requêtes imbriquées

- Sous-interrogation ramenant plusieurs lignes de plusieurs colonnes
  - **WHERE (expr1, expr2,…) op ANY (SELECT …)**
  - **WHERE (expr1, expr2,…) op ALL (SELECT …)**
  - **WHERE (expr1, expr2,…) IN (SELECT …)**
  - **WHERE (expr1, expr2,…) NOT IN (SELECT …)**
  - où op est = ou != (mais pas <, >, <=, >=)
  - Exemple :

```
SELECT NOME, POSTE, SAL FROM EMP
WHERE (POSTE, SAL) IN
(SELECT POSTE, SAL FROM EMP
WHERE DEPT = 10);
```


Requêtes imbriquées

- **La Jointure** s'exprime par deux blocs SFW imbriqués
  - SELECT ... FROM ...
    WHERE Attribut1 IN
    (SELECT Attribut1 FROM... WHERE ...)
- **La Différence** s'exprime aussi par deux blocs SFW imbriqués
  - SELECT ... FROM ...
    WHERE Attribut1 NOT IN
    (SELECT Attribut1 FROM... WHERE ...)

Sous-requêtes corrélées

- C'est une sous requête évaluée pour chaque ligne traitée par la requête principale
- Utilise les valeurs d'une colonne retournée par la requête principale
- La corrélation est obtenue en utilisant un élément de la requête externe dans la sous requête

Sous-requêtes corrélées

- Etapes d'exécution
  - Récupération de la ligne à utiliser (requête externe)
  - Exécution de la requête interne avec la valeur de la ligne récupérée
  - Utilisation de la valeur retournée par la requête interne pour ou non conserver la ligne
  - Itération → à il n'existe plus de lignes

Sous-requêtes corrélées

- Syntaxe
  - SELECT outer1, outer2, …
  - FROM table1 alias1
  - WHERE outer1 operateur (SELECT inner1 FROM table2 alias2
  - WHERE alias1.exp1=alias2.exp2);

Exemples

EMP (**EMPNO**, ENAME, JOB, *MGR*, HIREDATE, SAL, COMM, *DEPTNO*)
DEPT (**DEPTNO**, DNAME, LOC)

```
SELECT ENAME, SAL
FROM EMP
WHERE SAL>= ( SELECT SAL
FROM EMP
WHERE ENAME = 'ADAMS');
```

```
SELECT ENAME, SAL, DEPTNO
FROM EMP
WHERE SAL>= ( SELECT AVG(SAL)
FROM EMP
WHERE DEPTNO=10 GROUP BY DEPTNO);
```

```
SELECT ENAME, SAL, DEPTNO
FROM EMP
WHERE SAL>= ALL ( SELECT SAL  FROM
  EMP
WHERE DEPTNO=30);
```

```
SELECT ENAME, JOB, SAL
FROM EMP
WHERE (JOB,SAL)=( SELECT JOB, SAL
  FROM EMP
WHERE ENAME='TURNER');
```


Exemples

EMP (**EMPNO**, ENAME, JOB, *MGR*, HIREDATE, SAL, COMM, *DEPTNO*)
DEPT (**DEPTNO**, DNAME, LOC)

```
SELECT ENAME, JOB, SAL
FROM EMP
WHERE (SAL,COMM)=(SELECT  SAL,COMM
  FROM EMP
WHERE ENAME='FORD');
```

```
SELECT ENAME, JOB, SAL
FROM EMP
WHERE (JOB,SAL) IN (SELECT  JOB,
  SAL  FROM EMP
WHERE DEPTNO=10);
```

```
SELECT ENAME, JOB, SAL
FROM EMP
WHERE (JOB,SAL)<> ALL(SELECT
  JOB,SAL  FROM EMP
WHERE DEPTNO=10);
```

```
SELECT ENAME
FROM EMP E
WHERE EXISTS ( SELECT NULL FROM
  EMP WHERE MGR=E.EMPNO);
```


Fonctions de groupe

- SELECT ... FROM... WHERE... GROUP BY...
  HAVING ...
- Les fonctions de groupe peuvent apparaître dans une expression du SELECT ou du HAVING
- AVG moyenne, SUM somme, MIN plus petite valeur, MAX plus grande valeur, VARIANCE variance, STDDEV écart type, COUNT(*), nombre de lignes, COUNT(col) nombre de valeurs non nulles dans la colonne, COUNT(DISTINCT col) nombre de valeurs distinctes

GROUP BY

- Permet de regrouper des lignes qui ont les mêmes valeurs pour des expressions
  - **GROUP BY *expression*1, expression2,...**
- Une seule ligne affichée par regroupement de lignes
- Ex. : SELECT dept, count(*) FROM emp GROUP BY dept;
- Dans la liste des expressions du select ne peuvent figurer que des caractéristiques liées aux groupes : des fonctions de groupes, des expressions figurant dans le GROUP BY

HAVING

- Cette clause sert à sélectionner les groupes : **HAVING *prédicat***
- Le prédicat ne peut porter que sur des caractéristiques de groupe
- Ex.

```
SELECT DEPT, COUNT(*) FROM EMP
WHERE POSTE = 'SECRETAIRE'
GROUP BY DEPT
HAVING COUNT(*) > 1;

SELECT DEPT, COUNT(*) FROM EMP
GROUP BY DEPT
HAVING COUNT(*) = MAX(COUNT(*));
```


Order By

- La clause ORDER BY précise l'ordre dans lesquelles les lignes d'un SELECT seront données:

  ORDER BY *expr1* [DESC], *expr2* [DESC], …
- On peut aussi donner le numéro de la colonne qui servira de clé de tri
- Exemples :

```
SELECT DEPT, NOMD FROM DEPT ORDER BY NOMD;
SELECT DEPT, NOMD FROM DEPT ORDER BY 2;
```


Dictionnaire des données et Types Oracle

Dictionnaire des données

- Ensemble des tables, des objets créés et maintenus par le serveur Oracle
- Contient l'information de la BD
- Vues disponibles
  - USER_*
  - ALL_*
  - DBA_*
- Exemples

```
SELECT *
FROM dictionary;

SELECT table_name
FROM user_tables ;

SELECT *
FROM user_catalog ;
```


Vues disponibles

- Vues DBA_
  - Accessible aux personnes disposant du privilège DBA
  - Permet de tout visualiser
- Vues ALL_
  - Accessible à tout le monde
  - Permet de visualiser tous les éléments auxquels l'utilisateur est autorisé à accéder.
- Vues USER_
  - Accessible à tout le monde
  - Permet de visualiser tous les éléments dont l'utilisateur est propriétaire

Type Oracle

Data Type Description	
VARCHAR2(*size*)	Variable-length character data
CHAR(*size*)	Fixed-length character data
NUMBER(*p*,*s*)	Variable-length numeric data
DATE	Date and time values
LONG	Variable-length character data up to 2 gigabytes
CLOB	Character data up to 4 gigabytes
RAW and LONG RAW	Raw binary data
BLOB	Binary data up to 4 gigabytes
BFILE	Binary data stored in an external file; up to 4 gigabytes
ROWID	A 64 base number system representing the unique address of a row in its table

Les dates

- Oracle stocke des dates selon un format numérique interne : siècle, année, mois, jour, heures, minutes et secondes
- Le format par défaut d'affichage de la date est DD-MON-RR
- Les dates Oracle valides appartiennent à l'intervalle 1er janvier 4712 avant J-C - 31 décembre 9999 après J-C

Exemple

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date < '01-FEB-88';
```


Format de date RR

Année en Cours	Date indiquée	Format RR	Format YY
1995	27-OCT-95	1995	1995
1995	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017
2001	27-OCT-95	1995	2095

(documentation Oracle)

		Si l'année à deux chiffres indiquée est :	
		0–49	50–99
Si l'année à deux chiffres indiquée est :	0–49	La date renvoyée appartient au siècle en cours	La date renvoyée appartient au siècle précédant celui en cours
	50–99	La date renvoyée appartient au siècle suivant celui en cours	La date renvoyée appartient au siècle en cours

Type DATE

DataType	Description
`TIMESTAMP`	Date en secondes
`INTERVAL YEAR TO MONTH`	Intervalle en année et mois
`INTERVAL DAY TO SECOND`	Intervalle de jours en heures minutes et secondes

Type DATE

- TIMESTAMP WITH TIME ZONE : inclut une zone temporelle
  - Time Zone : différence, en heures & minutes, entre temps local & UTC
- TIMESTAMP WITH LOCAL TIME ZONE

Type DATE : Exemples

- INTERVAL YEAR [(year_precision)] TO MONTH
- INTERVAL '123-2' YEAR(3) TO MONTH
  - Un intervalle de 123 ans, 2 mois
- INTERVAL '123' YEAR(3)
  - Un intervalle de 123 ans, 0 mois
- INTERVAL '300' MONTH(3)
  - Un intervalle de 300 mois
- INTERVAL '123' YEAR
  - Erreur, car la précision par défaut vaut 2 et '123' est sur 3 car.

Type DATE : Exemples (2)

- INTERVAL DAY TO SECOND stocke une période de temps en termes de jours, heures, minutes, & seconds.
- INTERVAL DAY [(day_precision)] TO SECOND [(fractional_seconds_precision)]
- INTERVAL '4 5:12:10.222' DAY TO SECOND(3)
  - Indique 4 jours, 5 heures, 12 minutes, 10 secondes, and 222 centièmes de seconde
- INTERVAL '7' DAY
  - Indique 7 jours
- INTERVAL '180' DAY(3)
  - Indique 180 jours.

Type DATE : Exemples (3)

- INTERVAL '4 5:12' DAY TO MINUTE
  - Indique 4 jours, 5 heures and 12 minutes.
- INTERVAL '400 5' DAY(3) TO HOUR
  - Indique 400 jours 5 heures.
- INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)
  - Indique 11 heures, 12 minutes, and 10.2222222 secondes.

Fonction SYSDATE

- SYSDATE est une fonction qui renvoie :
  - la date
  - l'heure
  - Exemple

```
SELECT sysdate
FROM   dual;
```

Opérations arithmétiques

- Addition ou soustration de date
- Addition des heures à une date en divisant le nombre d'heures par 24

Opérateurs arithmétiques et dates

- Exemple Oracle

```
SELECT last_name, (SYSDATE-hire_date)/7 AS WEEKS
FROM   employees
WHERE  department_id = 90;
```


Manipulation de dates

- MONTHS_BETWEEN :Nombre de mois entre deux dates
- ADD_MONTHS : Ajout de mois à la date
- NEXT_DAY : Jour suivant la date indiquée
- LAST_DAY : Dernier jour du mois
- ROUND : Arrondi de la date
- TRUNC : Troncature de la date

Exemples Oracle

- `MONTHS_BETWEEN ('01-SEP-95','11-JAN-94')`
  → `19.6774194`
- `ADD_MONTHS ('31-JAN-96',1)`
  → `'29-FEB-96'`
- `NEXT_DAY ('01-SEP-95','FRIDAY')`
  → `'08-SEP-95'`
- `LAST_DAY ('01-FEB-95')`
  → `'28-FEB-95'`

ROUND et TRUNC et dates

- Exemples Oracle :
  - Hypothèse : SYSDATE = '25-JUL-03'

```
ROUND(SYSDATE,'MONTH')
                → 01-AUG-03
ROUND(SYSDATE ,'YEAR')
                → 01-JAN-04
TRUNC(SYSDATE ,'MONTH')
                → 01-JUL-03
TRUNC(SYSDATE ,'YEAR')
                → 01-JAN-03
```


Séquence

- Permet de définir des valeurs par défaut
- Partageable
- Permet de créer des clé primaires sans code additionnel

Séquence

- **Syntaxe:**

```
CREATE SEQUENCE nom_seq
[START WITH  valeur_debut]
[INCREMENT BY valeur_increment]
[MAXVALUE valeur_maxi  |  NOMAXVALUE]
[MINVALUE valeur_mini  |  NOMINVALUE]
[CYCLE | NOCYCLE] [CACHE|NOCACHE]
```

- **Utilisation:**
  - nom_seq.nextval
  - nom_seq.currval
- **Modification:** ALTER
- **Suppression**: DROP

Séquence

- Exemple

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

- Utilisation

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

Séquence

- CACHE:
  - Accès plus rapide aux valeurs
- Trous dans les séquences
  - Si rollback
  - Crash system
  - Si utilisation dans une autre table
- NOCACHE:
  - Accès à la valeur suivante avec la table USER_SEQUENCES

Table temporaire

- Permet de stocker une information particulière

```
SELECT …
INTO  [TEMPORARY | TEMP]  [TABLE]
Nom_nv_table
FROM …
```


Extensions

- INSERT Multitable
- Syntaxe:

```
INSERT [ALL] [condition_insert] [clause_insert_into valeurs]
  (sous_requête)
condition_insert
[ALL] [FIRST]
[WHEN condition THEN] [clause_insert_into valeurs]
[ELSE] [insert_into_clause valeurs]
```


Exemple 1

- Sélectionner EMPLOYEE_ID, HIRE_DATE, SALARY, MANAGER_ID dans EMPLOYES sous la condition EMPLOYEE_ID > à 200.
- Insérer ces valeurs dans les tables SAL_HISTORY et MGR_HISTORY

```
INSERT ALL
INTO sal_history VALUES(EMPID,HIREDATE,SAL)
INTO mgr_history VALUES(EMPID,MGR,SAL)
SELECT employee_id EMPID, hire_date HIREDATE,
salary SAL, manager_id MGR
FROM employees
WHERE employee_id > 200;
```

Exemple 2

- Sélectionner EMPLOYEE_ID, HIRE_DATE, SALARY, MANAGER_ID dans EMPLOYES sous la condition EMPLOYEE_ID > à 200.
  - Si le salaire est > à 10 000→vers la table SAL_HISTORY
  - Si MANAGER_ID > 200→ vers la table MGR_HISTORY

```
INSERT ALL
WHEN SAL > 10000 THEN
INTO sal_history VALUES(EMPID,HIREDATE,SAL)
WHEN MGR > 200 THEN
INTO mgr_history VALUES(EMPID,MGR,SAL)
SELECT employee_id EMPID,hire_date HIREDATE,
salary SAL, manager_id MGR
FROM employees
WHERE employee_id > 200;
```


Exemple 3

- Donner DEPARTMENT_ID , SUM(SALARY) & MAX(HIRE_DATE) dans la table EMPLOYEES
  - Si SUM(SALARY) > 25 000 alors insérer dans une table SPECIAL_SAL, en utilisant un INSERT multitable conditionnel
  - Si la première clause WHEN est Vrai, la clause WHEN suivante pour ce tuple ne doit pas être évaluée
  - Pour les autres tuples qui ne satisfont pas le 1er WHEN insérer dans 1 table HIREDATE_HISTORY_00 ou HIREDATE_HISTORY_99 ou HIREDATE_HISTORY basée sur la valeur de l'attribut HIRE_DATE en utilisant 1 INSERT multitable conditionnel

Exemple 3 (2)

```
INSERT FIRST
  WHEN SAL > 25000 THEN
    INTO special_sal VALUES(DEPTID, SAL)
  WHEN HIREDATE like ('%00%') THEN
    INTO hiredate_history_00 VALUES(DEPTID,HIREDATE)
  WHEN HIREDATE like ('%99%') THEN
    INTO hiredate_history_99 VALUES(DEPTID,HIREDATE)
   ELSE
    INTO hiredate_history VALUES(DEPTID, HIREDATE)
SELECT department_id DEPTID, SUM(salary) SAL, MAX(hire_date) HIREDATE
FROM employees    GROUP BY department_id;
```


Présentation données

Opérateur de concaténation

- Permet de :
  - Concaténer des colonnes ou des chaînes de caractères avec d'autres colonnes sous la forme d'une nouvelle chaîne de caractères
  - Syntaxe : ||
- Exemple

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


Littéraux

- Une chaîne de caractères ou une date peut être présente dans un SELECT si elle est entre simples quotes
- Les nombres sont affichés sans nécessité de reformatage

Délimitation avec (q)

- Permet de définir son propre délimiteur
- Exemple :

```
SELECT Attribut1 || q'[ Chaine_delimitateur ]'   || Attribut2 AS
"EnteteDeColonne"
```


Variables de substitution

- Permet de paramétrer les requêtes (WHERE, ORDER BY, SELECT complet, Noms de table) avec des valeurs temporaires saisie par l'utilisateur
- Syntaxe : & ou &&

Exemple 1

```
SELECT Att1, Att2, Att3, Att4
FROM   R
WHERE  Att4 = &NomVariable ;
```

→ Boîte de dialogue de saisie avec nom de la variable (ici EMPLOYEE_NUM)

Substitution

- Pour la saisie des chaînes de caractères et des dates :
  - Rajouter des apostrophes
    - Syntaxe :

```
SELECT…
FROM…
WHERE Att4='&NomVariable'
```

  - Rajouter des apostrophes à la saisie

Substitution

- Noms de colonne, expressions et texte
- Syntaxe :

```
SELECT liste_attributs, &NomAttribut
FROM Table
WHERE  &condition
ORDER BY &attribut
```


Substitution

- avec && si réutilisation de la valeur de la variable
- Syntaxe :

```
SELECT Liste_Attribut, &&NomVariable
FROM table
ORDER BY &NomVariable
```

Utilisation de variable “environnement”

- DEFINE : création et affectation d’une valeur à une variable
- UNDEFINE : suppression d’une variable
- Syntaxe :

```
DEFINE  NomVariable
SELECT Liste_Attribut
FROM…
WHERE Attribut1=&NomVariable
```

- VERIFY : verification de la requête avec la valeur remplacée

Prise en charge des expressions régulières (2)

Expressions régulières ?

- Recherche et manipulation des chaînes de caractères avec des modèles particuliers
- Un SELECT permet de vérifier des informations stockées dans la BD
- Formée avec
  - des métacaractères pour indiquer les algorithmes de recherche
  - des littéraux pour les caractères qu’on recherche

Fonctions d'expression régulière

- REGEXP_LIKE : equivalent au LIKE. Fonctionne sur une colonne de type caractère (à inclure au WHERE)
- REGEXP_REPLACE : remplacement des occurrences d’un modèle par un autre
- REGEXP_INSTR : recherche dans une chaine un modèle particulier et renvoie sa position
- REGEXP_SUBSTR : extrait une sous-chaine correspondant à l’E.R. fournie
- REGEXP_COUNT : renvoie le nb d’occurrences du modèle trouvé dans la chaine

Métacaractères

- Caractères générique,
- Nb d'occurrence d'un caractère
- Répétition du caractère
- …

Métacaractère

- Exemple : l'expression régulière ^(f|ht)tps?:$ recherche les éléments suivants à partir du début de la chaîne :
  - Les littéraux f ou ht
  - Le littéral t
  - Le littéral p, éventuellement suivi du littéral s
  - Le littéral deux-points ":" à la fin de la chaîne

Description des méta. (doc ORACLE)

Syntaxe	Description
.	Correspond à un caractère quelconque dans le jeu de caractères pris en charge, à l'exception de NULL
+	Correspond à une ou plusieurs occurrences
?	Correspond à zéro ou une occurrence
*	Correspond à un nombre quelconque d'occurrences (zéro ou plus) de la sous-expression précédente
{m}	Correspond exactement à *m* occurrences de l'expression précédente
{m, }	Correspond à *m* occurrences au moins de la sous-expression précédente
{m,n}	Correspond à *m* occurrences au moins, mais pas à plus de *n* occurrences de la sous-expression précédente
[...]	Correspond à n'importe quel caractère unique de la liste entre crochets
|	Correspond à l'une des alternatives
(...)	Considère l'expression entre parenthèses comme une unité. La sous-expression peut être une chaîne de littéraux ou une expression complexe contenant des opérateurs.

Description des métacaractères

Syntaxe	Description
^	Correspond au début d'une chaîne
$	Correspond à la fin d'une chaîne
\	Considère le métacaractère suivant de l'expression comme un littéral
\n	Correspond à la *n*ème (1–9) sous-expression précédente d'une quelconque expression entre parenthèses. Les parenthèses entraînent la mémorisation d'une expression ; une référence arrière (backreference) y est associée.
\d	Un chiffre
[:class:]	Correspond à n'importe quel caractère appartenant à la classe de caractères POSIX indiquée ex. [:upper:]
[^:class:]	Correspond à n'importe quel caractère unique *ne* figurant pas dans la liste entre crochets

Syntaxe

- **REGEXP_LIKE (source_char, pattern [,match_option]**
- **REGEXP_COUNT (source_char, pattern [, position [, occurrence [, match_option]]])**
- **REGEXP_REPLACE(source_char, pattern [,replacestr [, position [, occurrence [, match_option]]]])**

Syntaxe

- **REGEXP_SUBSTR (source_char, pattern [, position [, occurrence [, match_option [, subexpr]]]])**
- **REGEXP_INSTR (source_char, pattern [, position [, occurrence [, return_option [, match_option [, subexpr]]]]])**
- **Paramètres**
  - `Occurrence` : quelle occurrence du modèle est recherchée
  - `return_option` :
    - 0 : Renvoie la position du 1er caractère de l'occurrence (par défaut)
    - 1 : Renvoie la position du caractère suivant l'occurrence
  - `match_parameter` :
    - "`c`" : Mise en correspondance avec distinction Maj/Min (par défaut)
    - "`i`": Mise en correspondance sans distinction Maj/Min
    - "`n`" : Autorise l'opérateur de correspondance avec n'importe quel caractère
    - "`m`" : Chaîne source sur plusieurs lignes

REGEXP_LIKE

- **REGEXP_LIKE(source_char, pattern [, match_parameter])**
- Exemple

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)(e|a)n$');
```


REGEXP_REPLACE

- **REGEXP_REPLACE(source_char, pattern [,replacestr [, position [, occurrence [, match_option]]]])**
- Exemple

```
SELECT REGEXP_REPLACE(phone_number, '\.','-') AS phone FROM
employees;
```

REGEXP_INSTR

- **REGEXP_INSTR (source_char, pattern [, position [, occurrence [, return_option [, match_option]]]])**
- Exemple

```
SELECT street_address,
REGEXP_INSTR(street_address,'[[:alpha:]]') AS First_Alpha_Position
FROM  locations;
```


REGEXP_SUBSTR

- **REGEXP_SUBSTR (source_char, pattern [, position [, occurrence [, match_option]]])**
- Exemple

```
SELECT REGEXP_SUBSTR(street_address , ' [^ ]+ ') AS Road
FROM locations;
```


Sous-expressions

```
SELECT
  REGEXP_INSTR
  ('0123456789',      -- source char or search value
  '(123)(4(56)(78))', -- regular expression patterns
  1,                  -- position to start searching
  1,                  -- occurrence
  0,                  -- return option
  'i',                -- match option (case insensitive)
  1)             -- sub-expression on which to search     "Position"
FROM dual;
```

Résultat : 2

Exemple

- Le séquençage de l'ADN
- On recherche dans l'ADN des souris un sous-modèle spécifique qui identifie une protéine nécessaire à l'immunité → Résultat 315

```
SELECT
    REGEXP_INSTR('ccacctttccctccactcctcacgttctcacctgtaaagcgtccctccctcatccccatgccccttaccctgcaggg
    tagagtaggctagaaaccagagagctccaagctccatctgtggagaggtgccatccttgggctgcagagagaggagaatttgccccaaagctgc
    ctgcagagcttcaccacccctagtctcacaaagccttgagttcatagcatttcttgagttttcaccctgcccagcaggacactgcagcacccaa
    agggcttcccaggagtagggttgccctcaagaggctcttgggtctgatggccacatcctggaattgttttcaagttgatggtcacagccctgag
    gcatgtaggggcgtggggatgcgctctgctctgctctcctctcctgaacccctgaaccctctggctaccccagagcacttagagccag',
                    '(gtc(tcac)(aaag))',
                     1, 1, 0, 'i',
                     1) "Position"
FROM dual;
```

REGEXP_SUBSTR : Exemple

```
SELECT
  REGEXP_SUBSTR
①  ('acgctgcactgca', -- source char or search value
②   'acg(.*)gca',    -- regular expression pattern
③   1,               -- position to start searching
④   1,               -- occurrence
⑤   'i',             -- match option (case insensitive)
⑥   1)               -- sub-expression
 "Value"
FROM dual;
```

- Résultat → ctgcact

REGEXP_COUNT

- **REGEXP_COUNT (source_char, pattern [, position [, occurrence [, match_option]]])**

```
SELECT REGEXP_COUNT(
   'ccacctttccctccactcctcacgttctcacctgtaaagcgtccctccctcatccccatgccccc
   ttaccctgcag
   ggtagagtaggctagaaaccagagagctccaagctccatctgtggagaggtgccatccttgggctg
   cagagagaggagaatttgccccaaagctgcctgcagagcttcaccacccttagtctcacaaagcct
   tgagttcatagcatttcttgagttttcaccctgcccagcaggacactgcagcacccaaagggcttc
   ccaggagtagggttgccctcaagaggctcttgggtctgatggccacatcctggaattgttttcaag
   ttgatggtcacagccctgaggcatgtaggggcgtggggatgcgctctgctctgctctcctctcctg
   aacccctgaaccctctggctaccccagagcacttagagccag',
   'gtc') AS Count
FROM dual;
```

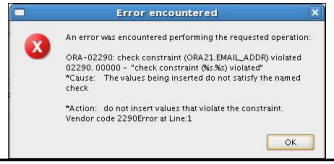
Résultat → 4

E.R. & contraintes CHECK

- Exemples Oracle

```
ALTER TABLE emp8
  ADD CONSTRAINT email_addr
  CHECK(REGEXP_LIKE(email,'@')) NOVALIDATE;
```

```
INSERT INTO emp8 VALUES
  (500,'Christian','Patel','ChrisP2creme.com',
   1234567890,'12-Jan-2004','HR_REP',2000,null,102,40);
```



Fonctions utiles

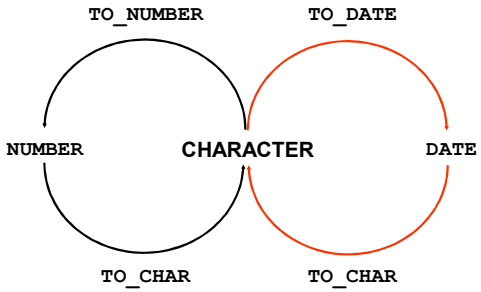
Fonctions de conversion

- Permettent de convertir EXPLICITEMENT les données d'un type vers un autre (par opposition aux conversions IMPLICITES)

Conversion implicite

- De VARCHAR2 ou CHAR → NUMBER
- De VARCHAR2 ou CHAR → DATE
- De NUMBER → VARCHAR2 ou CHAR
- De DATE → VARCHAR2 ou CHAR

Conversion explicite



Fonction TO_CHAR

- Syntaxe : **`TO_CHAR(date, 'format_model')`**
- Avec
  - Date : date à convertir
  - Format_model :
    - Indiqué entre apostrophes
    - Différence entre majuscules & minuscules
    - Comporte un élément fm permettant de supprimer les espaces de complément ou les zéros de début
- Exemple :

```
SELECT employee_id, TO_CHAR(hire_date, 'MM/YY') Month_Hired
FROM   employees
WHERE  last_name = 'Higgins';
```

Eléments du modèle de format de date

Elément	Résultat
YYYY	Année complète en chiffres
YEAR	Année en toutes lettres (en anglais)
MM	Mois sur deux chiffres
MONTH	Mois en toutes lettres
MON	Abréviation à trois lettres du mois
DY	Abréviation à trois lettres du jour de la semaine
DAY	Jour de la semaine en toutes lettres
DD	Jour du mois sous forme numérique

Exemples d'Elements de format

Elément	Description
SCC ou CC	Siècle. Le serveur utilise le préfixe - pour les dates avant J-C.
Années dans les dates YYYY ou SYYYY	Année. Le serveur utilise le préfixe - pour les dates avant J-C.
YYY ou YY ou Y	Dernier(s) chiffre(s) de l'année (3,2 ou 1).
Y,YYY	Année avec une virgule à la position indiquée.
IYYY, IYY, IY, I	Chiffres de l'année (4, 3, 2 ou 1), selon la norme ISO.
SYEAR ou YEAR	Année en toutes lettres. Le serveur utilise le préfixe - pour les dates avant J-C.
BC ou AD	Respectivement avant et après J-C.
B.C. ou A.D.	Respectivement avant et après J-C.
Q	Trimestre de l'année.

Elément	Description
MM	Représentation à deux chiffres du mois.
MONTH	Nom du mois complété avec des espaces pour atteindre une longueur de 9 caractères.
MON	Abréviation à trois lettres du nom du mois.
RM	Mois en chiffres romains.
WW ou W	Semaine de l'année ou du mois.
DDD ou DD ou D	Jour de l'année, du mois ou de la semaine.
DAY	Nom du jour complété avec des espaces afin d'atteindre une longueur de 9 caractères.
DY	Abréviation à trois lettres du nom du jour.
J	Jour julien (nombre de jours depuis le 31 décembre 4713 avant J-C).
IW	Semaines de l'année, selon la norme ISO (1 à 53).

Fonction TO_CHAR avec des dates

```
SELECT last_name,
       TO_CHAR(hire_date, 'fmDD Month YYYY')
       AS HIREDATE
FROM   employees;
```


TO_CHAR avec des nombres

- Syntaxe : **`TO_CHAR(number, 'format_model')`**

Elément	Résultat
9	Représente un nombre
0	Force l'affichage d'un zéro
$	Place un signe dollar flottant
L	Utilise le symbole flottant de la devise locale
.	Affiche un point comme séparateur décimal
,	Affiche une virgule comme séparateur des milliers

Fonction TO_CHAR

- Exemple

```
SELECT TO_CHAR(salary, '$99,999.00') SALARY
FROM   employees
WHERE  last_name = 'Ernst';
```


TO_NUMBER et TO_DATE

- TO_NUMBER : Chaîne de caractères → nombre

```
TO_NUMBER(char[, 'format_model'])
```

- TO_DATE : Chaîne de caractères → date

```
TO_DATE(char[, 'format_model'])
```


TO_CHAR et TO_DATE avec RR

- Exemple Oracle : Trouver les employés embauchés avant 1990, utilisation du format de date RR, qui produit les mêmes résultats que la commande soit exécutée en 1999 ou maintenant :

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')
FROM  employees
WHERE hire_date < TO_DATE('01-Jan-90','DD-Mon-RR');
```


Exemple Oracle 1

```
SELECT last_name,
  UPPER(CONCAT(SUBSTR (LAST_NAME, 1, 8), '_US'))
FROM   employees
WHERE  department_id = 60;
```

Exemple Oracle 2

```
SELECT   TO_CHAR(ROUND((salary/7), 2),'99G999D99',
         'NLS_NUMERIC_CHARACTERS = '',.'' ')
         "Formatted Salary"
FROM employees;
```

...

Fonction NVL

- Convertit une valeur NULL en valeur réelle :
  - Syntaxe : NVL (expr1, expr2)
    - Expr1: source
    - Expr2: cible
  - Exemples :
    - NVL(commission_pct,0)
    - NVL(hire_date,'01-JAN-97')
    - NVL(job_id,'No Job Yet')

Fonction NVL2

- Convertit une valeur NULL en valeur réelle
- Syntaxe : NVL2 (expr1, expr2, *expr3*)
  - *expr1* : valeur source qui peut contenir une valeur NULL.
  - *expr2* : valeur renvoyée si *expr1* <> NULL
  - *expr3* : valeur renvoyée si *expr1* = NULL.

Fonction NULLIF

- Compare deux expressions et retourne NULL si elle sont égales sinon renvoie la première valeur
- Syntaxe : NULLIF (expr1, expr2)
  - *expr1* : doit avoir une valeur non NULL

Fonction COALESCE

- Retourne la première expression non NULL de la liste
- Syntaxe : COALESCE (*expr1*, *expr2, ... exprn)*

Exemples Oracle

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

        SELECT last_name,  salary, commission_pct,
               NVL2(commission_pct,
                    'SAL+COMM', 'SAL') income
        FROM   employees WHERE department_id IN (50, 80);

SELECT first_name, LENGTH(first_name) "expr1",
       last_name,  LENGTH(last_name)  "expr2",
       NULLIF(LENGTH(first_name), LENGTH(last_name)) result
FROM   employees;


        SELECT last_name, employee_id,
        COALESCE(TO_CHAR(commission_pct),TO_CHAR(manager_id),
                 'No commission and no manager')
        FROM employees;
```


Expressions conditionnelles

- Logique IF-THEN-ELSE en SQL.
- Deux fonctions disponibles
  - CASE
  - DECODE

CASE

- Syntaxe

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

Exemple CASE

```
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;
```


Fonction DECODE

- Syntaxe

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


Exemple DECODE

```
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;
```


Exemple Oracle DECODE

- Afficher le taux d'imposition applicable pour chaque employé du département 80 :

```
SELECT last_name, salary,
       DECODE (TRUNC(salary/2000, 0),
                         0, 0.00,
                         1, 0.09,
                         2, 0.20,
                         3, 0.30,
                         4, 0.40,
                         5, 0.42,
                         6, 0.44,
                            0.45) TAX_RATE
FROM   employees
WHERE  department_id = 80;
```

SQL Wrapper

- Conversion du code PL/SQL en code exécutable
- Syntaxe

```
WRAP INAME=input_file_name [ONAME=output_file_name]
```

  - INAME paramètre obligatoire
  - Extension .sql par défaut
  - ONAME optionnel
  - Extension en sortie .plb par défaut

```
WRAP INAME=demo_04_hello.sql
WRAP INAME=demo 04 hello
```


SQL Wrapper : Résultats

- Avant

```
CREATE PACKAGE banking IS
  min_bal := 100;
  no_funds EXCEPTION;
...
END banking;
/
```

- Après

```
CREATE PACKAGE banking
wrapped

012abc463e ...

/
```


SQL Wrapper

- Sur le corps du package
- PAS sur les spécifications du package
- A exécuter après chaque modification sur le code PL/SQL

Packages Oracle

Package DBMS_SCHEDULER

- Gestion des tâches
  - Nom unique
  - QUOI
  - QUAND

Création d'un job

- Fonction CREATE_JOB
  - Nom et type du programme
  - Date et type de répétition
- Nécessite le privilège CREATE JOB

Exemple Oracle

```
DBMS_SCHEDULER.CREATE_JOB (
  job_name           => 'oe.my_job1',
  job_type           => 'PLSQL_BLOCK',
  job_action         => 'BEGIN DBMS_STATS.GATHER_TABLE_STATS("oe",
                       "sales"); END;',
  start_date         => '15-JUL-08 1.00.00AM US/Pacific',
  repeat_interval    => 'FREQ=DAILY',
  end_date           => '15-SEP-08 1.00.00AM US/Pacific',
  enabled            =>  TRUE,
  comments           => 'Gather table statistics');
END;
/
```


Création d'un job

- CREATE_PROGRAM

```
BEGIN
  DBMS_SCHEDULER.CREATE_PROGRAM(
   program_name => 'PROG_NAME',
   program_type => 'PLSQL_BLOCK',
   program_action => 'BEGIN ...; END;');
END;
```

- Surcharge de CREATE_JOB

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   program_name => 'PROG_NAME',
   start_date => SYSTIMESTAMP,
   repeat_interval => 'FREQ=DAILY',
   enabled => TRUE);
END;
```

Création d'un job avec arguments

- CREATE_PROGRAM

```
DBMS_SCHEDULER.CREATE_PROGRAM(
  program_name => 'PROG_NAME',
  program_type => 'STORED_PROCEDURE',
  program_action => 'EMP_REPORT');
```

- Définition d'un argument

```
DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT(
  program_name => 'PROG_NAME',
  argument_name => 'DEPT_ID',
  argument_position=> 1, argument_type=> 'NUMBER',
  default_value => '50');
```

- Création d'un

```
DBMS_SCHEDULER.CREATE_JOB('JOB_NAME', program_name
  => 'PROG_NAME', start_date => SYSTIMESTAMP,
  repeat_interval => 'FREQ=DAILY',
  number_of_arguments => 1, enabled => TRUE);
```


Création d'un job avec arguments

- CREATE_SCHEDULE

```
BEGIN
  DBMS_SCHEDULER.CREATE_SCHEDULE('SCHED_NAME',
   start_date => SYSTIMESTAMP,
   repeat_interval => 'FREQ=DAILY',
   end_date => SYSTIMESTAMP +15);
END;
```

- CREATE_JOB

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   schedule_name => 'SCHED_NAME',
   job_type => 'PLSQL_BLOCK',
   job_action => 'BEGIN ...; END;',
   enabled => TRUE);
END;
```


Intervalle d'un job

- Expression calendaire

```
repeat_interval=> 'FREQ=HOURLY; INTERVAL=4'
repeat_interval=> 'FREQ=DAILY'
repeat_interval=> 'FREQ=MINUTELY;INTERVAL=15'
repeat_interval=> 'FREQ=YEARLY;
                   BYMONTH=MAR,JUN,SEP,DEC;
                   BYMONTHDAY=15'
```

- Expression PL/SQL

```
repeat_interval=> 'SYSDATE + 36/24'
repeat_interval=> 'SYSDATE + 1'
repeat_interval=> 'SYSDATE + 15/(24*60)'
```


Nommage

- Du programme: CREATE_PROGRAM
- De l'ordonnancement: CREATE_SCHEDULE
- Exemple

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   program_name => 'PROG_NAME',
   schedule_name => 'SCHED_NAME',
   enabled => TRUE);
END;
/
```

Gestion des jobs

- Lancement

```
DBMS_SCHEDULER.RUN_JOB('SCHEMA.JOB_NAME');
```

- Arrêt

```
DBMS_SCHEDULER.STOP_JOB('SCHEMA.JOB_NAME');
```

- Arrêt en cours d'exécution

```
DBMS_SCHEDULER.DROP_JOB('JOB_NAME', TRUE);
```


Vues du dico des données

- [DBA | ALL | USER]_SCHEDULER_JOBS
- [DBA | ALL | USER]_SCHEDULER_RUNNING_JOBS
- [DBA | ALL]_SCHEDULER_JOB_CLASSES
- [DBA | ALL | USER]_SCHEDULER_JOB_LOG
- [DBA | ALL | USER]_SCHEDULER_JOB_RUN_DETAILS
- [DBA | ALL | USER]_SCHEDULER_PROGRAMS

SQL Dynamique, Collections et Relationnel Objet

SQL Dynamique Natif

SQL Dynamique Natif

- Support direct dans le PL/SQL
- Exécution de requête dont la structure est inconnue avant l'exécution
- Syntaxe
  - EXECUTE IMMEDIATE
  - OPEN-FOR
  - FETCH
  - CLOSE

Syntaxe

- EXECUTE IMMEDIATE
- Syntaxe :

```
EXECUTE IMMEDIATE dynamic_string
      [INTO {  define_variable
           [, define_variable] ... | record}]
      [USING [IN|OUT|IN OUT]   bind_argument
          [, [IN|OUT|IN OUT]  bind_argument] … ];
```

  - INTO : requête renvoyant une seule ligne et une colonne à stocker
  - USING : arguments

SQL dynamique

- Ordre LDD
  - Exemple : Création de table

```
CREATE PROCEDURE create_table(
  table_name VARCHAR2, col_specs  VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE '||table_name||
                    ' (' || col_specs || ')';
END;
/
```

  - Exemple : Appel

```
BEGIN
  create_table('EMPLOYEE_NAMES',
    'id NUMBER(4) PRIMARY KEY, name VARCHAR2(40)');
END;
/
```


SQL dynamique

- Suppression de lignes

```
CREATE FUNCTION del_rows(table_name VARCHAR2)
RETURN NUMBER IS
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM '||table_name;
  RETURN SQL%ROWCOUNT;
END;

BEGIN DBMS_OUTPUT.PUT_LINE(
  del_rows('EMPLOYEE_NAMES')|| ' rows deleted.');
END;
```

- Insertion d'un

```
CREATE PROCEDURE add_row(table_name VARCHAR2,
   id NUMBER, name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO '||table_name||
        ' VALUES (:1, :2)' USING id, name;
END;
```

Exemple

- Retour d'un tuple

```
CREATE FUNCTION get_emp(emp_id NUMBER)
RETURN employees%ROWTYPE IS
  stmt VARCHAR2(200);
  emprec employees%ROWTYPE;
BEGIN
  stmt := 'SELECT * FROM employees ' ||
          'WHERE employee_id = :id';
  EXECUTE IMMEDIATE stmt INTO emprec USING emp_id;
  RETURN emprec;
END;
/

DECLARE
  emprec employees%ROWTYPE := get_emp(100);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Emp: '||emprec.last_name);
END;
/
```


Exemple

- Retour de plusieurs tuples
  - OPEN-FOR, FETCH, CLOSE

```
CREATE PROCEDURE list_employees(deptid NUMBER) IS
  TYPE emp_refcsr IS REF CURSOR;
  emp_cv  emp_refcsr;
  emprec  employees%ROWTYPE;
  stmt varchar2(200) := 'SELECT * FROM employees';
BEGIN
  IF deptid IS NULL THEN OPEN emp_cv FOR stmt;
  ELSE
    stmt := stmt || ' WHERE department_id = :id';
    OPEN emp_cv FOR stmt USING deptid;
  END IF;
  LOOP
    FETCH emp_cv INTO emprec;
    EXIT WHEN emp_cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emprec.department_id||
                         ' ' ||emprec.last_name);
  END LOOP;
  CLOSE emp_cv;
END;
```


Bloc PL/SQL dynamique

- Exemple

```
CREATE FUNCTION annual_sal(emp_id NUMBER)
RETURN NUMBER IS
  plsql varchar2(200) :=
     'DECLARE '||
     ' emprec employees%ROWTYPE; '||
     'BEGIN '||
     ' emprec := get_emp(:empid); ' ||
     ' :res := emprec.salary * 12; ' ||
     'END;';
  result NUMBER;
BEGIN
 EXECUTE IMMEDIATE plsql
         USING IN emp_id, OUT result;
  RETURN result;
END;
/

EXECUTE DBMS_OUTPUT.PUT_LINE(annual_sal(100))
```


Compilation bloc PL/SQL

- Avec ALTER
  - ALTER PROCEDURE name COMPILE
  - ALTER FUNCTION name COMPILE
  - ALTER PACKAGE name COMPILE SPECIFICATION
  - ALTER PACKAGE name COMPILE BODY
- Exemple

```
CREATE PROCEDURE compile_plsql(name VARCHAR2,
 plsql_type VARCHAR2, options VARCHAR2 := NULL) IS
  stmt varchar2(200) := 'ALTER '|| plsql_type ||
                        ' '|| name || ' COMPILE';
BEGIN
 IF options IS NOT NULL THEN
   stmt := stmt || ' ' || options;
 END IF;
 EXECUTE IMMEDIATE stmt;
END;
/
```

DBMS_SQL Package

Exécution

1. OPEN_CURSOR
2. PARSE
3. BIND_VARIABLE or BIND_ARRAY
4. DEFINE_COLUMN, DEFINE_COLUMN_LONG, or DEFINE_ARRAY
5. EXECUTE
6. FETCH_ROWS or EXECUTE_AND_FETCH
7. VARIABLE_VALUE, COLUMN_VALUE, or COLUMN_VALUE_LONG
8. CLOSE_CURSOR

Exécution

1. OPEN_CURSOR
2. PARSE
3. BIND_VARIABLE or BIND_ARRAY
4. DEFINE_COLUMN, DEFINE_COLUMN_LONG, or DEFINE_ARRAY
5. EXECUTE
6. FETCH_ROWS or EXECUTE_AND_FETCH
7. VARIABLE_VALUE, COLUMN_VALUE, or COLUMN_VALUE_LONG
8. CLOSE_CURSOR

Exemple

- Suppression de tuples

```
CREATE OR REPLACE FUNCTION delete_all_rows
  (table_name VARCHAR2) RETURN NUMBER IS
  csr_id INTEGER;
  rows_del    NUMBER;
BEGIN
  csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(csr_id,
    'DELETE FROM '||table_name, DBMS_SQL.NATIVE);
  rows_del := DBMS_SQL.EXECUTE (csr_id);
  DBMS_SQL.CLOSE_CURSOR(csr_id);
  RETURN rows_del;
END;
/

CREATE table temp_emp as select * from employees;
BEGIN
 DBMS_OUTPUT.PUT_LINE('Rows Deleted: ' ||
delete_all_rows('temp_emp'));
END;
/
```

Exemple 2

- Ordre LMD avec paramètres

```
CREATE PROCEDURE insert_row (table_name VARCHAR2,
 id VARCHAR2, name VARCHAR2, region NUMBER) IS
  csr_id     INTEGER;
  stmt       VARCHAR2(200);
  rows_added NUMBER;
BEGIN
  stmt := 'INSERT INTO '||table_name||
          ' VALUES (:cid, :cname, :rid)';
  csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(csr_id, stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cid', id);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cname', name);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':rid', region);
  rows_added := DBMS_SQL.EXECUTE(csr_id);
  DBMS_SQL.CLOSE_CURSOR(csr_id);
  DBMS_OUTPUT.PUT_LINE(rows_added||' row added');
END;
/
```


SQL dynamique versus DBMS_SQL

- SQL dynamique
  - Plus facile à utiliser
  - Nécessite moins de code
  - Supporte tous les types PL/SQL y compris ceux utilisateur
  - Stocke directement des tuples dans des variables PL/SQL

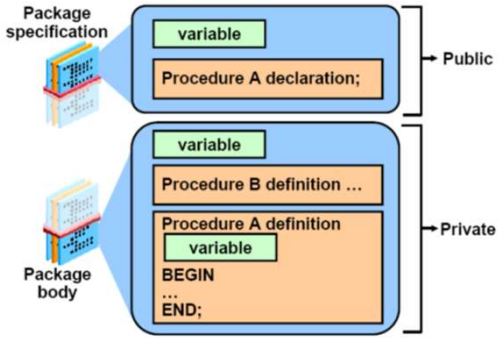
Packages

- Encapsulation des procédures, variables et types de données
- Séparation de la spécification et le corps du package
- Déclaration de procédure « public » et « private »
- Définition de variables persistantes
- Amélioration des performances

Packages

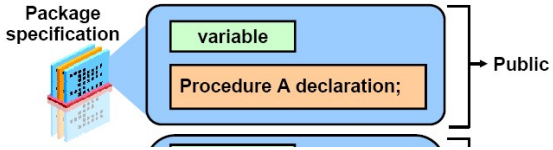
- Composé d'une partie
  - Spécification:
    - Définit l'interface pour les applications, procédures et fonctions disponibles
    - Y sont déclarés les types, les variables, les constantes, les exceptions, les curseurs et les sous-programmes utilisables dans le package
  - Corps:
    - Définit les curseurs et les sous-programmes spécifiés dans la spécification

Vue générale



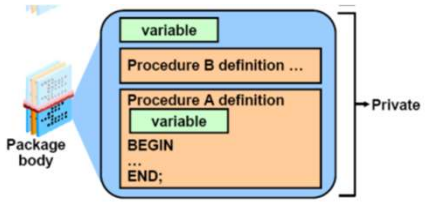
Packages

- Package Specification



Package

- Package body



Variables privée spécifiques au package

Procédures privées

Variables locales spécifiques à la proc.

Syntaxe Package

- Spécification:

```
CREATE [OR REPLACE] PACKAGE nom_package
IS |AS
-- déclaration variables
-- déclaration des types
-- déclaration des cstes, curseurs, public
-- spécifications des s/s prog.
END nom_package;
```

- Corps

```
CREATE [OR REPLACE] PACKAGE BODY nom_package
IS |AS
-- code du package
END nom_package;
```

Package

- Constantes/variables globales
  - Durée de vie des structures de données: pdt la durée de la session
  - Donnée déclarée dans le corps: persiste pendant la session
  - Donnée déclarée dans spécifications: donnée publique
- Curseur ouvert dans une procédure packagée reste ouvert pendant la session
- Les variables peuvent exister au-delà des transactions mais non partagées par plusieurs sessions sinon utiliser package DBMS_PIPE

Constantes variables globales

- Déclarées dans les spécifications d'un package
  - Variables globales
  - Corps inutile!

Exemple Oracle

- Exemple

```
CREATE OR REPLACE PACKAGE global_consts IS
  mile_2_kilo    CONSTANT  NUMBER  :=  1.6093;
  kilo_2_mile    CONSTANT  NUMBER  :=  0.6214;
  yard_2_meter   CONSTANT  NUMBER  :=  0.9144;
  meter_2_yard   CONSTANT  NUMBER  :=  1.0936;
END global_consts;
```

```
BEGIN  DBMS_OUTPUT.PUT_LINE('20 miles = ' ||
       20 * global_consts.mile_2_kilo || ' km');
END;
```

```
CREATE FUNCTION mtr2yrd(m NUMBER) RETURN NUMBER IS
BEGIN
  RETURN (m * global_consts.meter_2_yard);
END mtr2yrd;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(mtr2yrd(1))
```


Curseur packagé

- Défini dans les spécifications du package
  - Comme un curseur classique
  - Seulement avec un entête sans la requête: nécessite un RETURN qui précise les éléments renvoyés par le FETCH

Exemple

- Spécification

  CREATE OR REPLACE PACKAGE nom_package IS
  CURSOR nom_c1[(param1, …] RETURN nom_table%ROWTYPE
  END nom_package;

- Corps

  CREATE OR REPLACE PACKAGE BODY nom_package IS
  CURSOR nom_c1[(param1,…)] RETURN nom_table%ROWTYPE IS
  SELECT .. FROM … WHERE …
  …

Manipulation des packages

- Procédure publique : accessible via nom_package.nom_proc (parametres)
- Dans le package, inutile de nommer le package
- Surcharge possible

Exemple Oracle 2

```
CREATE OR REPLACE PACKAGE comm_pkg IS
  std_comm NUMBER := 0.10;  --initialized to 0.10
  PROCEDURE reset_comm(new_comm NUMBER);
END comm_pkg;
/
```

```
CREATE OR REPLACE PACKAGE BODY comm_pkg IS
  FUNCTION validate(comm NUMBER) RETURN BOOLEAN IS
    max_comm employees.commission_pct%type;
  BEGIN
    SELECT MAX(commission_pct) INTO max_comm
    FROM   employees;
    RETURN (comm BETWEEN 0.0 AND max_comm);
  END validate;
  PROCEDURE reset_comm (new_comm NUMBER) IS BEGIN
    IF validate(new_comm) THEN
      std_comm := new_comm; -- reset public var
    ELSE  RAISE_APPLICATION_ERROR(
            -20210, 'Bad Commission');
    END IF;
  END reset_comm;
END comm_pkg;
```


Exemple Oracle 2

- Appel dans le

```
CREATE OR REPLACE PACKAGE BODY comm_pkg IS ...
  PROCEDURE reset_comm(new_comm NUMBER) IS
  BEGIN
    IF validate(new_comm) THEN
      std_comm := new_comm;
    ELSE ...
    END IF;
  END reset_comm;
END comm_pkg;
```

Exemple Oracle 3

- Package sans corps

```
CREATE OR REPLACE PACKAGE global_consts IS
  mile_2_kilo    CONSTANT  NUMBER  :=  1.6093;
  kilo_2_mile    CONSTANT  NUMBER  :=  0.6214;
  yard_2_meter   CONSTANT  NUMBER  :=  0.9144;
  meter_2_yard   CONSTANT  NUMBER  :=  1.0936;
END global_consts;
```

```
BEGIN  DBMS_OUTPUT.PUT_LINE('20 miles = ' ||
       20 * global_consts.mile_2_kilo || ' km');
END;
```

```
CREATE FUNCTION mtr2yrd(m NUMBER) RETURN NUMBER IS
BEGIN
  RETURN (m * global_consts.meter_2_yard);
END mtr2yrd;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(mtr2yrd(1))
```


Vue package

- Tables USER_SOURCE et ALL_SOURCE
  - Spécifications

```
SELECT text
FROM   user_source
WHERE  name = 'COMM_PKG' AND type = 'PACKAGE';
```

  - Body

```
SELECT text
FROM   user_source
WHERE  name = 'COMM_PKG' AND type = 'PACKAGE BODY';
```


Surcharge

- Paramètres différents (ordre ou nombre ou type)
- Fonctionne avec
  - Sous-programmes de packages
  - Sous-programmes locaux

Exemple

- Spécification du package

```
CREATE OR REPLACE PACKAGE dept_pkg IS
  PROCEDURE add_department(deptno NUMBER,
   name VARCHAR2 := 'unknown', loc NUMBER := 1700);
  PROCEDURE add_department(
   name VARCHAR2 := 'unknown', loc NUMBER := 1700);
END dept_pkg;
/
```

Exemple

- Corps du package

```
CREATE OR REPLACE PACKAGE BODY dept_pkg  IS
  PROCEDURE add_department (deptno NUMBER,
   name VARCHAR2:='unknown', loc NUMBER:=1700) IS
  BEGIN
    INSERT INTO departments(department_id,
      department_name, location_id)
    VALUES  (deptno, name, loc);
  END add_department;

  PROCEDURE add_department (
    name VARCHAR2:='unknown', loc NUMBER:=1700) IS
  BEGIN
    INSERT INTO departments (department_id,
      department_name, location_id)
    VALUES (departments_seq.NEXTVAL, name, loc);
  END add_department;
 END dept_pkg;
/
```


Surcharge du package STANDARD

- Définit l'environnement PL/SQL et les fonctions
- Exemple la fonction TO_CHAR

```
FUNCTION TO_CHAR (p1 DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (p2 NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (p1 DATE, P2 VARCHAR2) RETURN
VARCHAR2;
FUNCTION TO_CHAR (p1 NUMBER, P2 VARCHAR2) RETURN
VARCHAR2;
```

- Surcharge
  - valable localement

Déclaration postérieure

- Exemple de pb

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
  PROCEDURE award_bonus(. . .) IS
  BEGIN

    calc_rating (. . .);    --illegal reference

  END;

  PROCEDURE calc_rating (. . .) IS
  BEGIN
    ...
  END;
END forward_pkg;
/
```


Déclaration postérieure

- Exemple

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
  PROCEDURE calc_rating (...);-- forward declaration

  -- Subprograms defined in alphabetical order

  PROCEDURE award_bonus(...) IS
  BEGIN
    calc_rating (...);        -- reference resolved!
    . . .
  END;

  PROCEDURE calc_rating (...) IS -- implementation
  BEGIN
    . . .
  END;
END forward_pkg;
```

Package

- Bloc d'initialisation
  - Exécuté une fois
  - Initialisation variables public et privée

```
CREATE OR REPLACE PACKAGE taxes IS
  tax   NUMBER;
  ...  -- declare all public procedures/functions
END taxes;
/
CREATE OR REPLACE PACKAGE BODY taxes IS
  ... -- declare all private variables
  ... -- define public/private procedures/functions
BEGIN
  SELECT   rate_value INTO tax
  FROM     tax_rates
  WHERE    rate_name = 'TAX';
END taxes;
/
```


Utilisation des packages

- Possible dans des requêtes SQL ou du LMD
  - Fonctions appelées ne doivent pas à partir
    - D'une requête, finir la transaction courante, faire un savepoint ou faire un rollback vers un savepoint
    - D'une requête, modifier la base si la requête est parallélisée
    - D'un ordre LMD lire ou modifier la table modifiée par la requête

Exemple d'utilisation

```
CREATE OR REPLACE PACKAGE taxes_pkg IS
  FUNCTION tax (value IN NUMBER) RETURN NUMBER;
END taxes_pkg;
/
CREATE OR REPLACE PACKAGE BODY taxes_pkg IS
  FUNCTION tax (value IN NUMBER) RETURN NUMBER IS
    rate NUMBER := 0.08;
  BEGIN
    RETURN (value * rate);
  END tax;
END taxes_pkg;
/
```

```
SELECT taxes_pkg.tax(salary), salary, last_name
FROM   employees;
```


Persistance: Etat du package

- Initialisé quand il est chargé pour la première fois
- Persistant lors de la session utilisateur
  - Sauvegardé dans la UGA (User Global Area)
  - Unique à chaque session
  - Modifié éventuellement par les sous-programmes
  - Pendant la vie d'un sous-programmes si utilisation de la directive PRAGMA SERIALLY_REUSABLE dans les spécifications du package

Exemple

		State for: -Scott-		-Jones-	
Time	Events	STD	MAX	STD	MAX
9:00	Scott> EXECUTE comm_pkg.reset_comm(0.25)	0.10 0.25	0.4	-	0.4
9:30	Jones> INSERT INTO employees(last_name,commission_pct) VALUES('Madonna', 0.8);	0.25	0.4		0.8
9:35	Jones> EXECUTE comm_pkg.reset_comm (0.5)	0.25	0.4	0.1 0.5	0.8
10:00	Scott> EXECUTE comm_pkg.reset_comm(0.6) Err -20210 'Bad Commission'	0.25	0.4	0.5	0.8
11:00	Jones> ROLLBACK;	0.25	0.4	0.5	0.4
11:01	EXIT ...	0.25	0.4	-	0.4
12:00	EXEC comm_pkg.reset_comm(0.2)	0.25	0.4	0.2	0.4

Curseur packagé

- Exemple

```
CREATE OR REPLACE PACKAGE BODY curs_pkg IS
  CURSOR c IS SELECT employee_id FROM employees;
  PROCEDURE open IS
  BEGIN
    IF NOT c%ISOPEN THEN   OPEN c;  END IF;
  END open;
  FUNCTION next(n NUMBER := 1) RETURN BOOLEAN IS
    emp_id employees.employee_id%TYPE;
  BEGIN
    FOR count IN 1 .. n LOOP
      FETCH c INTO emp_id;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Id: ' ||(emp_id));
    END LOOP;
    RETURN c%FOUND;
  END next;
  PROCEDURE close IS BEGIN
    IF c%ISOPEN THEN  CLOSE c;  END IF;
  END close;
END curs_pkg;
```


Curseur packagé

- Exemple Utilisation

```
SET SERVEROUTPUT ON
EXECUTE curs_pkg.open
DECLARE
  more BOOLEAN := curs_pkg.next(3);
BEGIN
  IF NOT more THEN
    curs_pkg.close;
  END IF;
END;
/
RUN  -- repeats execution on the anonymous block
EXECUTE curs_pkg.close
```


Droits sur les procédures

- **authid current_user** : utilisé quand un code PL / SQL s'exécute avec les privilèges de l'utilisateur actuel, et NON l'ID utilisateur qui a créé la procédure
- **authid definer** : s'exécute avec les privilèges de l'utilisateur créateur

Exemple

```
CREATE TYPE adresse_t
AUTHID CURRENT_USER
AS OBJECT (
address_line1 varchar2(80),
address_line2 varchar2(80),
street_name varchar2(30),
street_adress number,
city varchar2(30),
state_or_province varchar2(2),
zip number(5),
zip_4 number(4),
country_code varchar2 (20));
```


Exemple

```
CREATE OR REPLACE PROCEDURE
    MYPROC
    AUTHID DEFINER
AS . . . .
```

```
CREATE TYPE address_t
    AUTHID DEFINER
AS OBJECT (
    address_line1     varchar2(80),
    street_name       varchar2(30),
    street_number     number,
    city              varchar2(30),
    state_or_province varchar2(2),
    country_code varchar2(20));
```


Exemple

```
select
   object_name,
   procedure_name,
   authid
from
   dba_procedures;
```


Collections et Relationnel Objet

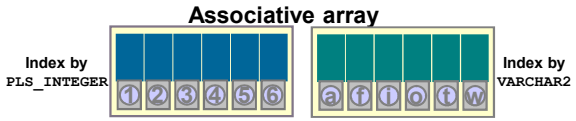
Collections Oracle

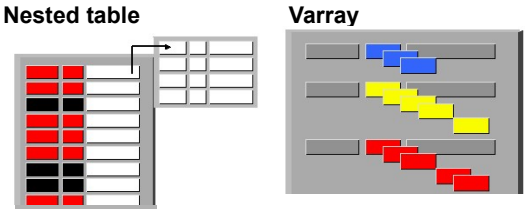
- Un ensemble d'objets tous du même type
- Fonctionnent comme les tableaux
- Peuvent contenir des instances d'un type d'objet particulier ou être « attributs » d'un type d'objet

Différents types

- Les tableaux associatifs
  - Indexés par des chaînes de caractères
  - INDEX BY pls_integer ou BINARY_INTEGER
- Nested tables / Tables imbriquées
- Varrays

Collections (doc Oracle.com)





Tableaux associatifs

- Indexés par des chaînes de caractères
- Structures mémoire plus rapides que des tables classiques

Exemple

- En PL/SQL (string-indexed):

```
TYPE type_name IS TABLE OF  element_type INDEX BY VARCHAR2(size)

CREATE OR REPLACE PROCEDURE report_credit
  (p_last_name customers.cust_last_name%TYPE, p_credit_limit customers.credit_limit%TYPE)
IS
  TYPE  typ_name IS TABLE OF customers%ROWTYPE
    INDEX BY customers.cust_email%TYPE;
  v_by_cust_email   typ_name;
  i VARCHAR2(30);

  PROCEDURE load_arrays IS
  BEGIN
    FOR rec IN  (SELECT * FROM customers WHERE cust_email IS NOT NULL)
      LOOP
        -- Load up the array in single pass to database table.
         v_by_cust_email (rec.cust_email) := rec;
      END LOOP;
  END;
...
```


Exemple : peupler le tableau

```
...
BEGIN
  load_arrays;
  i:= v_by_cust_email.FIRST;
  dbms_output.put_line ('For credit amount of: ' || p_credit_limit);
  WHILE i IS NOT NULL LOOP
    IF v_by_cust_email(i).cust_last_name = p_last_name
    AND v_by_cust_email(i).credit_limit > p_credit_limit
      THEN dbms_output.put_line ( 'Customer '||
        v_by_cust_email(i).cust_last_name || ': ' ||
        v_by_cust_email(i).cust_email || ' has credit limit of: ' ||
        v_by_cust_email(i).credit_limit);
    END IF;
    i := v_by_cust_email.NEXT(i);
  END LOOP;
END report_credit;
/
```

→ **`EXECUTE report_credit('Walken', 1200)`**

Tables imbriquées

- Une table dans une table
- Taille non limitée a priori
- En SQL et en PL/SQL
- Accès direct à des lignes particulières comme dans un tableau

Table de stockage (doc Oracle.com)

ORDID	SUPPLIER	REQUESTER	ORDERED	ITEMS
500	50	5000	30-OCT-07	
800	80	8000	31-OCT-07	

`pOrder` nested table:

NESTED_TABLE_ID	PRODID	PRICE
	55	555
	56	566
	57	577
NESTED_TABLE_ID	PRODID	PRICE
	88	888

Création

- Syntaxe SQL

```
CREATE [OR REPLACE] TYPE type_name AS TABLE OF
Element_datatype [NOT NULL];
```

- Syntaxe PL/SQL:

```
TYPE type_name IS TABLE OF element_datatype
[NOT NULL];
```


Syntaxe

1. Déclaration du type OBJECT
2. Déclaration de la table imbriquée comme colonne

Exemple

```
CREATE TYPE typ_item AS OBJECT  --create object
 (prodid  NUMBER(5),
  price   NUMBER(7,2) )
/

CREATE TYPE typ_item_nst -- define nested table type
  AS TABLE OF typ_item
/
CREATE TABLE pOrder (  -- create database table
     ordid  NUMBER(5),
     supplier     NUMBER(5),
     requester    NUMBER(4),
     orderedDATE,
     items  typ_item_nst)
     NESTED TABLE items STORE AS item_stor_tab
/
```


Peuplement

```
INSERT INTO pOrder
  VALUES (500, 50, 5000, sysdate, typ_item_nst(
     typ_item(55, 555),
     typ_item(56, 566),
     typ_item(57, 577)));

INSERT INTO pOrder
  VALUES (800, 80, 8000, sysdate, typ_item_nst (typ_item (88, 888)));
```

Table imbriquée pOrder

ORDID	SUPPLIER	REQUESTER	ORDERED	ITEMS
500	50	5000	30-OCT-07	
800	80	8000	31-OCT-07	

PRODID	PRICE
55	555
56	566
57	577

PRODID	PRICE
88	888

Utilisation

- Requête classique

```
SELECT * FROM porder;

     ORDID   SUPPLIER  REQUESTER ORDERED
---------- ---------- ---------- ---------
ITEMS(PRODID, PRICE)
--------------------------------------------------------------------
       500         50       5000 31-OCT-07
TYP_ITEM_NST(TYP_ITEM(55, 555), TYP_ITEM(56, 566), TYP_ITEM(57, 577))
       800         80       8000 31-OCT-07
TYP_ITEM_NST(TYP_ITEM(88, 888))
```


Utilisation

- Requête avec la fonction `TABLE`

```
SELECT p2.ordid, p1.*
FROM porder p2, TABLE(p2.items) p1;

     ORDID     PRODID      PRICE
---------- ---------- ----------
       800         88        888
       500         57        577
       500         55        555
       500         56        566
```


Accès aux éléments

- Via leur indice
  - Syntaxe

```
collection_name(subscript)
```

  - Exemple

```
v_with_discount(i)
```

- Accès aux champs : exemple

```
p_new_items(i).prodid
```


Exemple PL/SQL

```
CREATE OR REPLACE PROCEDURE add_order_items
(p_ordid NUMBER, p_new_items typ_item_nst)
IS
  v_num_items      NUMBER;
  v_with_discount  typ_item_nst;
BEGIN
  v_num_items := p_new_items.COUNT;
  v_with_discount := p_new_items;
  IF v_num_items > 2 THEN
  --ordering more than 2 items gives a 5% discount
    FOR i IN 1..v_num_items LOOP
      v_with_discount(i) :=
      typ_item(p_new_items(i).prodid,
               p_new_items(i).price*.95);
    END LOOP;
  END IF;
  UPDATE pOrder
    SET  items = v_with_discount
    WHERE ordid = p_ordid;
END;
```

Utilisation en PL/SQL

```
-- caller pgm:
DECLARE
  v_form_items  typ_item_nst:= typ_item_nst();
BEGIN
  -- let's say the form holds 4 items
  v_form_items.EXTEND(4);
  v_form_items(1) := typ_item(1804, 65);
  v_form_items(2) := typ_item(3172, 42);
  v_form_items(3) := typ_item(3337, 800);
  v_form_items(4) := typ_item(2144, 14);
  add_order_items(800, v_form_items);
END;
```

v_form_items

PRODID	PRICE
1804	65
3172	42
3337	800
2144	14

Données de la table imbriquée

ORDID	SUPPLIER	REQUESTER	ORDERED	ITEMS
500	50	5000	30-OCT-07	
800	80	8000	31-OCT-07	

PRODID	PRICE
1804	65
3172	42
3337	800
2144	14

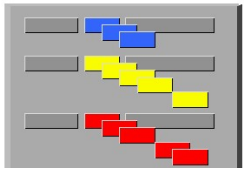
VARRAYS

- En SQL

```
CREATE [OR REPLACE] TYPE type_name AS VARRAY(max_elements) OF
element_datatype [NOT NULL];

TYPE type_name IS VARRAY (max_elements) OF element_datatype [NOT NULL];
```

- En PL/SQL



Mise en oeuvre

1. Définition du type d'objet
2. Déclaration d'une collection de ce type
3. Utilisation SQL ou PL/SQL

Exemple

```
CREATE TYPE typ_Project AS OBJECT(  --create object
   project_no NUMBER(4),
   title      VARCHAR2(35),
   cost       NUMBER(12,2))
/

CREATE TYPE typ_ProjectList AS VARRAY (50) OF typ_Project
        -- define VARRAY type
/

CREATE TABLE department (  -- create database table
   dept_id  NUMBER(2),
   name     VARCHAR2(25),
   budget   NUMBER(12,2),
   projects typ_ProjectList)  -- declare varray as column
/
```

Utilisation

- Ajout de données

```
INSERT INTO department
  VALUES (10, 'Executive Administration', 30000000,
     typ_ProjectList(
     typ_Project(1001, 'Travel Monitor',  400000),
     typ_Project(1002, 'Open World',    10000000)));

INSERT INTO department
  VALUES (20, 'Information Technology', 5000000,
     typ_ProjectList(
     typ_Project(2001, 'DB11gR2', 900000)));
```

Table DEPARTMENT

DEPT_ID	NAME	BUDGET	PROJECTS		
			PROJECT_NO	TITLE	COSTS
10	Executive Administration	30000000	1001.	Travel Monitor	400000
			1002.	Open World	10000000
20	Information Technology	5000000	2001	DB11gR2	900000

Requêtes

- SELECT Classique

```
SELECT * FROM department;

   DEPT_ID NAME                           BUDGET
---------- ------------------------- ----------
PROJECTS(PROJECT_NO, TITLE, COST)
--------------------------------------------------------------------
        10 Executive Administration    30000000
TYP_PROJECTLIST(TYP_PROJECT(1001, 'Travel Monitor', 400000),
TYP_PROJECT(1002, 'Open World', 10000000))
        20 Information Technology       5000000
TYP_PROJECTLIST(TYP_PROJECT(2001, 'DB11gR2', 900000))
```


Requêtes

- SELECT avec fonction TABLE :

```
SELECT d2.dept_id, d2.name, d1.*
FROM department d2, TABLE(d2.projects) d1;

DEPT_ID NAME                      PROJECT_NO TITLE              COST
------- ------------------------- ---------- --------------  --------
     10 Executive Administration       1001 Travel Monitor    400000
     10 Executive Administration       1002 Open World      10000000
     20 Information Technology         2001 DB11gR2           900000
```


Collections en PL/SQL

- Utilisation possible dans les procédures et les fonctions comme paramètres
- Utilisation possible comme type de retour de fonction via le RETURN

Exemple

```
CREATE OR REPLACE PACKAGE manage_dept_proj
AS
  PROCEDURE allocate_new_proj_list
    (p_dept_id NUMBER, p_name VARCHAR2, p_budget NUMBER);
  FUNCTION get_dept_project (p_dept_id NUMBER)
    RETURN typ_projectlist;
  PROCEDURE update_a_project
    (p_deptno NUMBER, p_new_project typ_Project,
     p_position NUMBER);
  FUNCTION manipulate_project (p_dept_id NUMBER)
    RETURN typ_projectlist;
  FUNCTION check_costs (p_project_list typ_projectlist)
    RETURN boolean;
END manage_dept_proj;
```


Initialisation

- 3 posibilités
  - Utilisation d'un constructeur
  - A partir de la BD
  - En assignant directement une autre variable de ce type collection

Initialisation

```
PROCEDURE allocate_new_proj_list
   (p_dept_id NUMBER, p_name VARCHAR2, p_budget NUMBER)
 IS
   v_accounting_project typ_projectlist;
 BEGIN
 -- this example uses a constructor
   v_accounting_project :=
     typ_ProjectList
       (typ_Project (1, 'Dsgn New Expense Rpt', 3250),
        typ_Project (2, 'Outsource Payroll', 12350),
        typ_Project (3, 'Audit Accounts Payable',1425));
   INSERT INTO department
     VALUES(p_dept_id, p_name, p_budget, v_accounting_project);
 END allocate_new_proj_list;
```


Exemple

```
FUNCTION get_dept_project (p_dept_id NUMBER)
   RETURN typ_projectlist
 IS
   v_accounting_project typ_projectlist;
 BEGIN  -- this example uses a fetch from the database
    SELECT  projects INTO  v_accounting_project
      FROM  department WHERE dept_id = p_dept_id;
    RETURN v_accounting_project;
 END get_dept_project;

 FUNCTION manipulate_project (p_dept_id NUMBER)
    RETURN typ_projectlist
 IS
   v_accounting_project typ_projectlist;
   v_changed_list typ_projectlist;
 BEGIN
   SELECT  projects INTO  v_accounting_project  FROM  department
      WHERE dept_id = p_dept_id;
        -- this example assigns one collection to another
   v_changed_list := v_accounting_project;
   RETURN v_changed_list;
 END manipulate_project;
```

Exemple

```
-- sample caller program to the manipulate_project function
DECLARE
  v_result_list typ_projectlist;
BEGIN
  v_result_list := manage_dept_proj.manipulate_project(10);
  FOR i IN 1..v_result_list.COUNT LOOP
    dbms_output.put_line('Project #: '

||v_result_list(i).project_no);
    dbms_output.put_line('Title: '||v_result_list(i).title);
    dbms_output.put_line('Cost: ' ||v_result_list(i).cost);
  END LOOP;

END;
```

```
Project #: 1001
Title: Travel Monitor
Cost: 400000
Project #: 1002
Title: Open World
Cost: 10000000
```


Méthodes

- EXISTS
- COUNT
- LIMIT
- FIRST & LAST
- PRIOR & NEXT
- EXTEND
- TRIM
- DELETE

collection_name.method_name **[(*parameters*)]**

Utilisation

```
FUNCTION check_costs (p_project_list typ_projectlist)
    RETURN boolean
  IS
    c_max_allowed        NUMBER := 10000000;
    i                    INTEGER;
    v_flag               BOOLEAN := FALSE;
  BEGIN
    i := p_project_list.FIRST ;
    WHILE i IS NOT NULL LOOP
      IF p_project_list(i).cost > c_max_allowed then
        v_flag := TRUE;
        dbms_output.put_line (p_project_list(i).title || '
                              exceeded allowable budget.');
        RETURN TRUE;
      END IF;
    i := p_project_list.NEXT(i);
    END LOOP;
    RETURN null;
  END check_costs;
```


Exemple

```
-- sample caller program to check_costs
set serveroutput on
DECLARE
  v_project_list typ_projectlist;
BEGIN
  v_project_list := typ_ProjectList(
    typ_Project (1,'Dsgn New Expense Rpt', 3250),
    typ_Project (2, 'Outsource Payroll', 120000),
    typ_Project (3, 'Audit Accounts Payable',14250000));
  IF manage_dept_proj.check_costs(v_project_list) THEN
    dbms_output.put_line('Project rejected: overbudget');
  ELSE
    dbms_output.put_line('Project accepted, fill out forms.');
  END IF;
END;
```

→ Audit Accounts Payable exceeded allowable budget.
Project rejected: overbudget

V_PROJECT_LIST

PROJECT_NO	TITLE	COSTS
1	Dsgn New Expense Rpt	3250
2	Outsource Payroll	120000
3	Audit Accounts Payable	14250000

Manipuler les éléments

```
PROCEDURE update_a_project
    (p_deptno NUMBER, p_new_project typ_Project, p_position NUMBER)
  IS
     v_my_projects typ_ProjectList;
  BEGIN
    v_my_projects := get_dept_project (p_deptno);
    v_my_projects.EXTEND;   --make room for new project
    /* Move varray elements forward */
    FOR i IN REVERSE p_position..v_my_projects.LAST - 1 LOOP
      v_my_projects(i + 1) := v_my_projects(i);
    END LOOP;
    v_my_projects(p_position) := p_new_project; -- insert new one
   UPDATE department SET projects = v_my_projects  WHERE dept_id = p_deptno;
  END update_a_project;
```


Exemple Oracle

```
-- check the table prior to the update:
SELECT d2.dept_id, d2.name, d1.*
FROM department d2, TABLE(d2.projects) d1;

DEPT_ID NAME                      PROJECT_NO TITLE                        COST
------- ------------------------- ---------- ------------------------------ --
     10 Executive Administration        1001 Travel Monitor             400000
     10 Executive Administration        1002 Open World               10000000
     20 Information Technology          2001 DB11gR2                    900000

-- caller program to update_a_project
BEGIN
  manage_dept_proj.update_a_project(20,
    typ_Project(2002, 'AQM', 80000), 2);
END;

-- check the table after the update:
SELECT d2.dept_id, d2.name, d1.*
FROM department d2, TABLE(d2.projects) d1;

DEPT_ID NAME                      PROJECT_NO TITLE                        COST
------- ------------------------- ---------- ------------------------------ --
     10 Executive Administration        1001 Travel Monitor             400000
     10 Executive Administration        1002 Open World               10000000
     20 Information Technology          2001 DB11gR2                    900000
     20 Information Technology          2002 AQM                         80000
```


Gestion des Exceptions

- Exceptions usuelles avec les collections
  - COLLECTION_IS_NULL
  - NO_DATA_FOUND
  - SUBSCRIPT_BEYOND_COUNT
  - SUBSCRIPT_OUTSIDE_LIMIT
  - VALUE_ERROR

Exemple

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  nums NumList;          -- atomically null
BEGIN
  /* Assume execution continues despite the raised exceptions.
*/
  nums(1) := 1;          -- raises COLLECTION_IS_NULL
  nums := NumList(1,2); -- initialize table
  nums(NULL) := 3        -- raises VALUE_ERROR
  nums(0) := 3;          -- raises SUBSCRIPT_OUTSIDE_LIMIT
  nums(3) := 3;          -- raises SUBSCRIPT_BEYOND_COUNT
  nums.DELETE(1);        -- delete element 1
  IF nums(1) = 1 THEN    -- raises NO_DATA_FOUND
...
```

Utilisation des collections

- Varrays :
  - Moins d'accès disque, + efficace
  - Ordre des éléments préservé

Utilisation des collections

- Tables imbriquées : pour de grandes quantités de données
- Après suppression libérer la mémoire avec DBMS_SESSION.FREE_UNUSED_USER_MEMORY