Langage SQL

Objectifs
Apprentissage du langage SQL
Section
R1.05, R2.06 : IUT GON Campus 3 - Département « Informatique »
Auteur
E.Porcq
Références Bibliographiques
SQL pour Oracle de christian Soutou
Les bases de données Oracle de Rogvt_etapeer Chapuis
Références WEB
http://oracle.developpez.com/guide/developpement/sql/

Pour tester les exemples, il peut être nécessaire d'ajouter " prof. " devant le nom des vues ou des tables

1 La projection en langage SQL

La partie du langage SQL appelée langage d'interrogation (LID ou DIL) est basée sur l'utilisation des opérateurs de l'algèbre relationnelle associés au modèle relationnel. Il existe huit opérateurs.

Quatre sont spécifiques à la manipulation des données organisées selon le modèle relationnel :

- Projection, Restriction (ou Sélection), Jointure et Division.

Quatre correspondent aux opérateurs de la théorie des ensembles :

- Union, Intersection, Différence et Produit cartésien.

1.1 Opérateur de Projection

1.1.1 Définition

L'opérateur de projection permet d'afficher le contenu d'une table en spécifiant les colonnes souhaitées.

La table résultat contient autant de lignes que la table de départ : l'opérateur de projection conserve la cardinalité (nombre de lignes) de la table.

1.1.2 Traduction dans le langage SQL

```
SELECT [DISTINCT] <liste d'attributs>
FROM <nom_table> [t_nom_alias] où
```

- `liste_attributs ::= <attribut> [ [AS] ["]<c_nom_alias> ["] ]`
  `[, attribut [ ["] c_nom_alias ["] ],…]`
  * (tous les attributs de la relation (ou table)

- `attribut ::= [{<nom_table>|<t_nom_alias>}.] {nom_colonne | *} |`
  `<expression> ou <chaîne de caractères>`

- `DISTINCT` * élimine les lignes au résultat identique.
- `t_nom_alias` * " raccourci " de nom de table
- `c_nom_alias` * " raccourci " de nom de colonne

Exemples :

```
select prenom from prof.vt_coureur;
select prenom from vt_coureur co;
select distinct prenom from vt_coureur;
select nom,prenom from vt_coureur;
select * from vt_coureur;
select nom as nom_sponsor from vt_sponsor;
select nom as " Nom du sponsor " from vt_sponsor;
```

1.1.3 **Tri du résultat d'une requête**

Les critères de tri sont spécifiés dans une clause ORDER BY figurant en dernière position de l'ordre SELECT selon la syntaxe :

```
ORDER BY {<nom_col1>|<numéro_col1>|<Expression>}[{ [ASC] | DESC } ]
            [,{<nom_col2>|<numéro_col2>|<Expression>} [{[ASC]|DESC}]]...
```

Exemple :

```
select prenom from prof.vt_coureur
order by prenom;
select distinct prenom from prof.vt_coureur
order by prenom;
select nom,prenom from prof.vt_coureur
order by nom desc, prenom;
```

1.1.4 **Pseudo-colonne ROWNUM et identifiant ROWID**

la pseudo-colonne ROWNUM retourne une valeur numérique entière qui indique l'ordre de sélection de la ligne au moment de l'exécution de la requête. Elle peut être utilisée pour limiter le nombre de réponses lors d'une projection.

Chaque élément de la base de données est enregistré avec un numéro unique déterminé par Oracle : le ROWID.

Exemple :

```
select rownum,nom from prof.vt_coureur;
select rownum,nom from prof.vt_coureur
where rownum < 10;
select nom from prof.vt_coureur
where rownum < 10;
select rowid,rownum,nom from vt_coureur;
```

Rownum n'a pas la performance de limit présent sur d'autres SGBD. Il ne permet pas d'obtenir une partie des résultats en en supprimant les premiers. Cette solution peut être mise en place :

```
select * from
(
  select rownum as nb, nom from
  (
    select * from prof.vt_coureur c order by nom
  )
)
where  nb between 10 and 20;
```

1.1.5 **La clause FETCH**

Depuis la version 12c, une clause permettant de limiter les lignes récupérées a été ajoutée. Cette clause de nomme " Fetch " est considérée comme standard depuis la norme SQL-2008.

Exemple :

```
select * from prof.vt_coureur
order by annee_naissance
FETCH FIRST 10 ROWS ONLY; -- on peut remplacer FIRST par NEXT
```

De plus, si on souhaite garder les enregistrements de même classement que le dernier, on peut remplacer ONLY par WITH TIES

Exemple :

```
select * from prof.vt_coureur
order by annee_naissance
fetch first 5 rows with ties;
```

	N_COUREUR	NOM	PRENOM	ANNEE_NAISSANCE
1	2263	POULIDOR	Raymond	1936
2	2233	BRACKE	Ferdinand	1939
3	2403	VAN SCHIL	Victor	1939
4	2264	GENET	Jean-Pierre	1940
5	2439	PINGEON	Roger	1940
6	2406	RIOTTE	Raymond	1940
7	2213	HOBAN	Barry	1940

Il est possible d'ajouter une limite inférieure avec le mot-clé OFFSET.

Exemple :

```
select * from prof.vt_coureur
order by annee_naissance
OFFSET 10 ROWS
FETCH FIRST 10 ROWS ONLY;
```

Pour finir , on peut remplacer le nombre de ligne par un pourcentage.

Exemple :

```
select * from prof.vt_coureur
order by annee_naissance
FETCH FIRST 1 PERCENT ROWS ONLY;
```

1.2 Opérateur de Sélection (ou de Restriction)

1.2.1 Définition

L'opérateur de sélection, aussi appelé restriction, permet de ne conserver pour un affichage que les lignes de la table qui vérifient une condition de sélection (ou prédicat de sélection), définie sur les valeurs prises par une ou plusieurs colonnes de la table.

Le résultat comporte autant de colonnes que la table de départ : l'opérateur de restriction conserve le degré de la table (nombre de colonnes).

1.2.2 Traduction dans le langage SQL

```
SELECT <liste d'attributs>
FROM <nom_table>
WHERE <prédicat>
[AND … [OR …]]
```

- **Prédicat simple**

=		égal
!= ou <>		différent
>		supérieur
>=		supérieur ou égal
<		inférieur
<=		inférieur ou égal
[NOT] BETWEEN	<nom colonne> BETWEEN expr1 AND expr2	compris entre ... et ...
[NOT] IN	<nom colonne> IN (<expression_1>, <expression_2>...)	est dans (liste)
[NOT] LIKE	<nom colonne> LIKE {'%' | 'chaîne de caractères%' | '%chaîne%' | '___chaîne%'}	ressemble à
le % est le caratère joker qui remplace le reste du texte ('%U' = se termine par U. _ remplace 1 caractère		
IS [NOT] NULL		permet de tester la valeur NULL.
Valeurs NULL : au sens relationnel, une valeur NULL est une valeur non définie.		

```
Exemple :
select * from prof.vt_coureur
where code_cio = 'BEL';
select * from prof.vt_coureur
where code_cio <> 'FRA';
select * from prof.vt_equipe
where annee_creation > 2005;
select * from prof.vt_equipe
where annee_disparition is not null;
select * from prof.vt_etape
where ville_a like 'CA%';
select * from prof.vt_etape
where annee between 1996 and 1998;
```

- **Prédicat composé**
  - Un prédicat composé est constitué de plusieurs prédicats simples ou composés reliés par des opérateurs logiques AND, OR ou NOT.
  - L'opérateur NOT placé devant un prédicat en inverse le sens.
  - L'opérateur AND est prioritaire par rapport à l'opérateur OR.
  - Des parenthèses peuvent être utilisées pour imposer une priorité dans l'évaluation du prédicat.

```
Exemples :
select nom,prenom,annee_naissance,annee_prem from prof.vt_coureur
where annee_naissance = 1997 or annee_naissance = 1996;
select nom,prenom,annee_naissance,annee_prem from prof.vt_coureur
where (annee_naissance = 1997 or annee_naissance = 1996)
and annee_prem = 2018;
```

```
select nom,prenom,annee_naissance,annee_prem from prof.vt_coureur
where annee_naissance between 1994 and 1998;
select nom,prenom,annee_naissance,annee_prem from prof.vt_coureur
where (annee_naissance = 1997 or annee_naissance = 1996)
and annee_prem <> 2018;
select nom,prenom,annee_naissance,annee_prem from prof.vt_coureur where
annee_naissance not between 1995 and 1998;
select nom,prenom,annee_naissance,annee_prem from prof.vt_coureur where
annee_naissance in (1955,1965,1985);
select nom,prenom,annee_naissance,annee_prem from prof.vt_coureur where
annee_naissance > 1995;
select n_equipe,nom,na_sponsor from prof.vt_sponsor
where na_sponsor is null or n_equipe=3;
select * from prof.vt_etape where
ville_d like 'PA%' and annee > 2000;
select nom,prenom from prof.vt_coureur
where nom like '%ERN%';
```

1.2.3 **Expression conditionnelle CASE**

Cette structure de contrôle peut posséder un nombre " infini " de branches conditionnelles. Il faut utiliser cette écriture avec parcimonie car elle est peu performante. 2 formes d'écriture sont accessibles sous Oracle.

Syntaxe générale :

```
select <liste d'attributs>
case
  when <cond1> then <expr1>
  [when <cond2> then <expr2>]
  ...
  [ else <expr3> ]
end
```

```
Exemple :
select n_etape, distance,
case
when distance <10 then 'prologue'
when distance >=10 and distance <100 then 'étape courte'
when distance >=100 then 'étape longue'
end as libelle
from prof.vt_etape
where annee=2010;
```

Syntaxe sur expression :

```
select <liste d'attributs>
case <expression>
  when <valeur1> then <expr1>
  [when <valeur2> then <expr2>]
  ...
  [ else <expr3> ]
end ;
```

```
Exemple :
select n_etape, distance,
case cat_code
  when 'PRO' then 'prologue'
  when 'ETA' then 'étape en ligne'
  when 'CMI' then 'contre la montre individuel'
  when 'CME' then 'contre la montre par équipe'
end  as type_epreuve
from prof.vt_etape
where annee=2010;
```

1.2.4 ***Fonction conditionnelle DECODE***

voir chapitre 1.7.3

1.3 Opérateur de jointure

L'opérateur de jointure permet de mettre en relation plusieurs tables.

Dans une opération de jointure la condition de restriction est appelée pivot de jointure.

Plusieurs types de jointures sont définies selon la nature de la condition de restriction.

1.3.1 Equijointure

1.3.1.1 Définition

L'équi-jointure est l'opération qui permet de réaliser une liaison logique entre deux tables. Le pivot de jointure exprime alors l'égalité de valeurs entre deux colonnes des deux tables. C'est la jointure la plus utilisée.

1.3.1.2 Syntaxe

- l'équijointure relationnelle avec la clause WHERE (SQL1)

```
SELECT <liste d'attributs>
FROM  <nom_table1> [<nom_alias1>], <nom_table2> [<nom_alias2>]
WHERE <pivot>
avec pivot::= [{<nom_table1>|<nom_alias1>}.]<nom_colonne> =
      [{<nom_table2>|<nom_alias2>}.]<nom_colonne>
```

- l'équijointure SQL2 avec la clause JOIN … ON

```
SELECT <liste d'attributs>
FROM  <nom_table1> [<nom_alias1>]
[INNER] JOIN  <nom_table2> [<nom_alias2>]
ON    [{<nom_table1>|<nom_alias1>}.]<nom_colonne> =
[{<nom_table2>|<nom_alias2>}.]<nom_colonne>
[AND  [{nom_table1|nom_alias1}.]<nom_colonne> =
[{<nom_table2>|<nom_alias2>}.]<nom_colonne>]
```

- l'équijointure SQL2 avec la clause JOIN … USING

```
SELECT <liste d'attributs>
FROM  <nom_table1> [<nom_alias1>]
[INNER] JOIN  <nom_table2> [<nom_alias2>]
USING  (<col1> [,<col2>…])
```

- l'équijointure SQL2 avec la clause NATURAL JOIN

```
SELECT <liste d'attributs>
FROM  <nom_table1> [<nom_alias1>]
NATURAL JOIN  <nom_table2> [<nom_alias2>]
```

```
Exemple :
select *
from prof.vt_coureur,prof.vt_temps
where vt_coureur.n_coureur = vt_temps.n_coureur;

select * from prof.vt_coureur
join prof.vt_temps on vt_coureur.n_coureur = vt_temps.n_coureur;

select * from prof.vt_coureur
join prof.vt_temps using(n_coureur);

select * from prof.vt_coureur
natural join prof.vt_temps;
```

1.3.2 Jointure et restriction simultanées

Il est possible d'exprimer par la rédaction d'un seul ordre SELECT, la jointure entre deux tables ainsi que des opérations de sélection sur les tables :

- La jointure SQL1

```
SELECT <liste d'attributs>
FROM  <nom_table1>, <nom_table2>
WHERE <pivot>
AND   <condition de restriction sur table1>
AND   <condition de restriction sur table2>
```

- La jointure SQL2 avec la clause JOIN … ON

```
SELECT <liste d'attributs>
FROM  <nom_table1>
JOIN  <nom_table2> ON <pivot>
WHERE <condition de restriction sur table1>
AND   <condition de restriction sur table2>
```

- La jointure SQL2 avec la clause JOIN … USING

```
SELECT      <liste d'attributs>
FROM  <nom_table1>
JOIN  <nom_table2> USING   (<col1> [,<col2>…])
WHERE <condition de restriction sur table1>
AND   <condition de restriction sur table2>
```

1.3.3 **Thétajointure**

1.3.3.1 Définition

Une thétajointure est une jointure dont l'expression du pivot utilise des opérateurs autre que l'égalité, tel que : <, <=, >, >=, != ou <>. Elle est utilisée dans des cas très particuliers.

1.3.3.2 Syntaxe

```
SELECT liste d'attributs
FROM  <nom_table1>, <nom_table2>
WHERE <pivot>
```

avec pivot : `[{<nom_table1>|<nom_alias1>}.]<nom_colonne>` **<opérateur>**
`[{<nom_table2>|<nom_alias2>}.]<nom_colonne>`

1.3.4 **Auto jointure**

1.3.4.1 Définition

Une auto-jointure est la jointure d'une table à elle-même. Une même table peut figurer plusieurs fois dans un ordre SELECT. SQL construit la requête selon les mêmes règles que précédemment en considérant plusieurs copies de la même table. Il est alors obligatoire d'utiliser des ***alias*** de table pour différencier les différentes copies. Elle permet en général de comparer des lignes d'une même table.

Une auto jointure possède une égalité, une inégalité et un distinct. Elle ne projette que les propriétés que d'une seule instance de la table

1.3.4.2 Syntaxe

- en jointure SQL1

```
SELECT distinct <liste d'attributs de alias1>
FROM  <nom_table1> <nom_alias1>, <nom_table1> <nom_alias2> [, ...]
WHERE  <nom_alias1>.<attribut1> = <nom_alias2>.<attribut1>
AND <nom_alias1>.<attribut2> <> <nom_alias2>.<attribut2>
[AND …]
```

- en jointure SQL2

```
SELECT distinct <liste d'attributs de alias1>
FROM  <nom_table1> <nom_alias1>
JOIN <nom_table1> <nom_alias2>
ON <nom_alias1>.<attribut1> = <nom_alias2>.<attribut1>
WHERE <nom_alias1>.<attribut2> <> <nom_alias2>.<attribut2>
[AND …]
```

Exemple :

```
select distinct d1.nom from prof.vt_directeur d1, prof.vt_directeur d2
where d1.nom = d2.nom
and d1.n_directeur <> d2.n_directeur;

select distinct d1.nom from prof.vt_directeur d1
join prof.vt_directeur d2 on d1.nom = d2.nom
where d1.n_directeur <> d2.n_directeur;
```

1.3.5 **Jointures multiples**

On peut, de la même manière, réaliser une jointure de plus de deux tables. Le processus consiste alors à joindre, de façon interne deux tables, puis à partir de la table intermédiaire ainsi créée, à réaliser la jointure avec la table suivante jusqu'à l'obtention du résultat final :

- Jointure et clause WHERE

```
SELECT <liste d'attributs>
FROM <nom_table1>, <nom_table2>, ... <nom_tablen>
WHERE <pivot1> AND <pivot...> AND <pivot_n>
[AND  <condition de restriction sur table1>]
[AND  <condition de restriction sur table2>]
[...]
```

- La jointure et la clause JOIN … ON

```
SELECT <liste d'attributs>
FROM  <nom_table1>,
JOIN <nom_table2> ON <pivot1>
JOIN  …
JOIN  <nom_tablen> ON <pivotn>
[WHERE condition de restriction sur table1]
[AND  condition de restriction sur table2]
[...]
```

Exemple :

```
select spo.nom,dir.nom,prenom
from vt_sponsor spo
join vt_parti_equipe eqa using(n_equipe,n_sponsor)
join vt_directeur dir on dir.n_directeur = eqa.n_pre_directeur;
```

1.3.6 **Jointure externe**

1.3.6.1 Définition

Dans l'exécution d'une opération de jointure, les lignes des tables qui ne vérifient pas la condition exprimée par le pivot de jointure, ne sont pas affichées dans le résultat.

Une jointure externe (OUTER JOIN) favorise une table, appelée table dominante, par rapport à l'autre table subordonnée. Les lignes de la table dominante sont affichées même si la condition de jointure n'est pas réalisée.

1.3.6.2 Syntaxe SQL1

L'opérateur **(+)** placé après le nom d'une des deux colonnes qui composent le pivot de jointure, indique la table subordonnée (où manquent des éléments). On peut aussi réaliser une jointure complète (ou bilatérale) en faisant l'union de 2 jointures externes unilatérales.

- pivot de jointure

[{nom_table_1|t_nom_alias_1}.]nom_colonne = [{nom_table_2|t_nom_alias_2}.]nom_colonne (+)

- syntaxe

```
SELECT .......
FROM <nom_table1> <nom_alias1>, <nom_table2> <nom_alias2> [, ...]
WHERE <pivot>
[AND …]
```

```
Exemple :
select nom,prenom,c_typeaban from vt_coureur cou
join vt_abandon aba on cou.n_coureur = aba.n_coureur
order by c_typeaban desc, nom, prenom;
```

1.3.6.3 Syntaxe SQL2

En SQL2, les mots-clés " LEFT ", " RIGHT " et " FULL " associés au " JOIN … ON " remplacent les (+) qui étaient « fastidieux » à poser. Les mots-clés LEFT ou RIGHT sont associés à la table qui contient tous les éléments.

- La jointure externe et la clause JOIN … ON

```
SELECT .......
FROM <nom_table1> <nom_alias1>
{left|right} [outer] join <nom_table2> <nom_alias2> on <pivot classique>
```

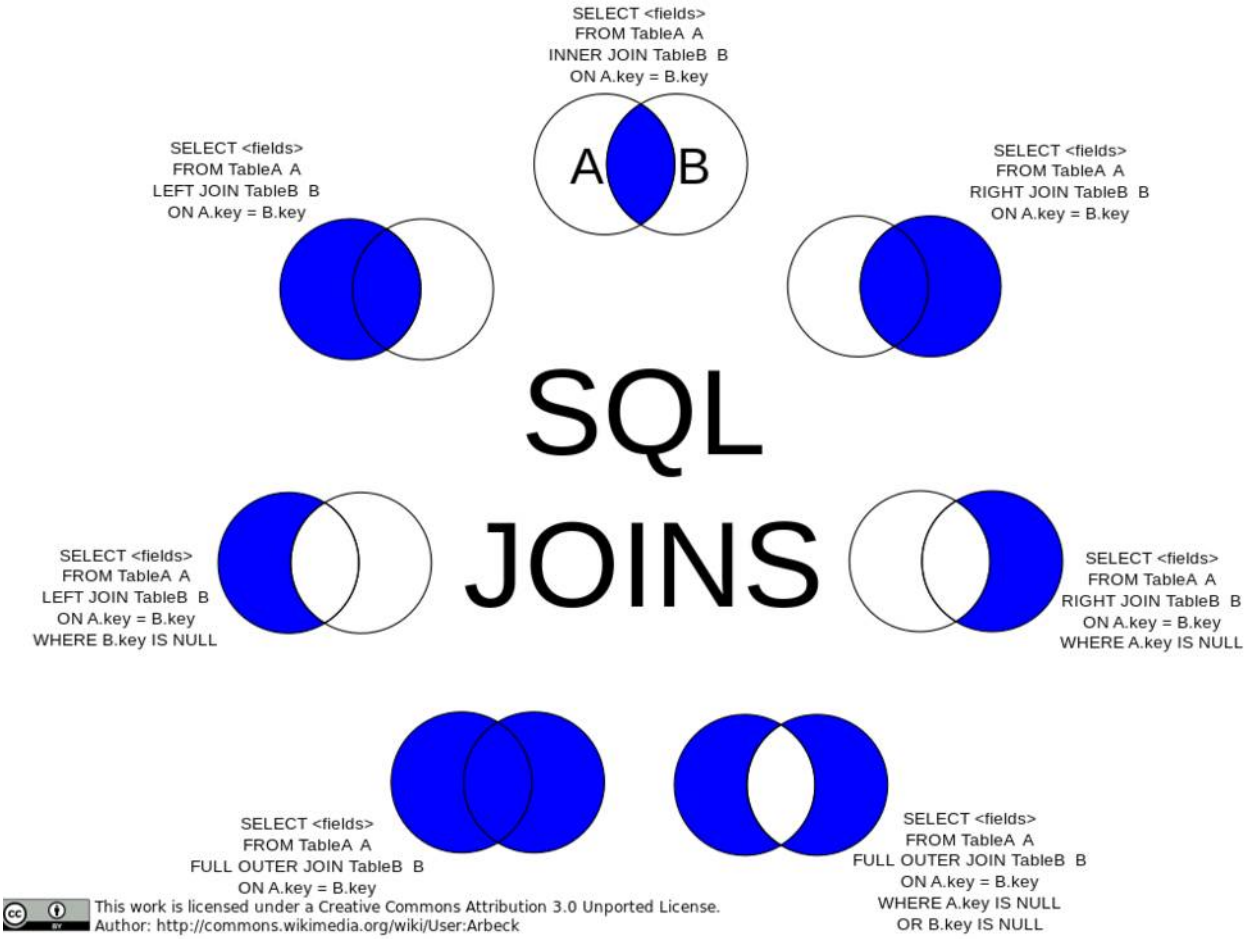
```
Exemple :
select nom,prenom,c_typeaban from vt_coureur cou
left join vt_abandon aba on cou.n_coureur = aba.n_coureur
order by c_typeaban desc, nom, prenom;
```

les jointures en 5 étapes

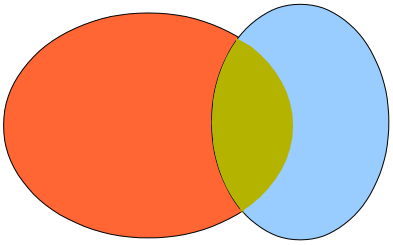
Règle1. Tu utiliseras d'abord une requête sur chaque table concernée
Règle2. Tu réaliseras la jointure entre les tables concernées avec * en projection, sans restriction ni tri
Règle3. Tu ajouteras les restrictions
Règle4. Tu projetteras enfin ensuite les colonnes demandées
Règle5. Tu trieras les résultats obtenus en fonction des besoins

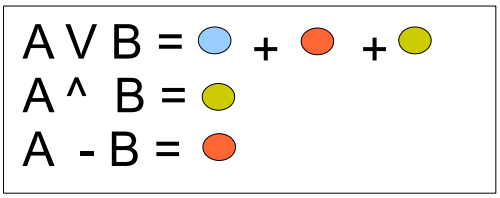
1.4 Opérateurs ensemblistes

Les opérations ensemblistes du SQL sont l'union (V), l'intersection (^) et la différence (-).
On utilise sous Oracle, les opérateurs union, intersect et minus entre deux requêtes projetant le même nombre



de colonnes et leur types doivent être un à un compatibles sinon identiques.
On dit que les deux tables sur lesquelles on travaille doivent avoir le même schéma de relation.
Les doublons sont éliminés
Les colonnes porteront le nom de celles de la première requête
Un seul tri final est autorisé.





1.4.1 **Union**

1.4.1.1 Définition

L'opérateur d'union entre deux relations de même schéma permet de créer une relation résultat contenant **l'ensemble des lignes** des deux relations de départ. Les attributs de même rang des relations de départ doivent être définis avec des **types compatibles**.

1.4.1.2 Syntaxe

- `SELECT <liste_attributs> FROM ... [WHERE …]`
  `UNION`
  `SELECT <liste_attributs> FROM ... [WHERE …]`
  Ou
- `SELECT <liste_attributs> FROM ... [JOIN …] [WHERE ...]`
  `UNION`
  `SELECT <liste_attributs> FROM ... [JOIN …] [WHERE …]`

Exemple :

```
select ville_d from vt_etape
union
select ville_a from vt_etape;

select nom,prenom from vt_coureur
Union
select nom,prenom from vt_directeur;

select nom from vt_coureur
union
select nom from vt_sponsor;

select annee_naissance from vt_coureur
union
select nom from vt_coureur;

select nom,prenom from vt_coureur
union
select nom from vt_directeur;
```

1.4.2 **Intersection**

1.4.2.1 Définition

L'opérateur d'intersection entre deux relations permet de créer une relation résultat composée des lignes qui appartiennent simultanément aux deux tables de départ. Les attributs de même rang des relations de départ doivent être définis avec des types compatibles.

1.4.2.2 Syntaxe

- `SELECT <liste_attributs> FROM ... [WHERE …]`
  `INTERSECT`
  `SELECT <liste_attributs> FROM ... [WHERE …]`
  Ou
- `SELECT <liste_attributs> FROM ... [JOIN …] [WHERE ...]`
  `INTERSECT`
  `SELECT <liste_attributs> FROM ... [JOIN …] [WHERE ...]`

Exemple :

```
select nom from vt_coureur
intersect
select nom from vt_directeur;
```

1.4.3 **Différence**

1.4.3.1 Définition

L'opérateur de différence entre deux tables permet de créer une table résultat composée des lignes qui appartiennent à la première table et pas à la seconde. Les attributs de même rang des tables de départ doivent être définis avec des types compatibles.

1.4.3.2 Syntaxe

- ```
  SELECT <liste_attributs> FROM ... [WHERE …]
  MINUS
  SELECT <liste_attributs> FROM ... [WHERE …]
  Ou
  ```
- ```
  SELECT <liste_attributs> FROM ...   [JOIN  …]   [WHERE ...]
  MINUS
  SELECT <liste_attributs> FROM ...   [JOIN …]    [WHERE ...]
  ```

```
Exemple :
select nom from vt_coureur
minus
select nom from vt_directeur;
```

1.4.4 **Combinaisons de plusieurs opérateurs ensemblistes**

On peut utiliser, dans une même requête, plusieurs opérateurs " UNION ", " INTERSECT " ou " MINUS ", combinés avec des opérations de projections, restrictions ou jointures.

1.4.5 **Règles communes**

Dans une requête utilisant des opérateurs ensemblistes :

- Tous les ordres " SELECT " doivent avoir **le même nombre de colonnes** sélectionnées, et leurs types doivent être **compatibles**. Les conversions éventuelles doivent être faites à l'intérieur de l'ordre SELECT à l'aide des fonctions de conversion (TO_CHAR, TO_DATE, TO_NUMBER...).
- Les doublons sont **éliminés** (DISTINCT implicite).
- Les noms des colonnes « titre du résultat » sont ceux du **premier** ordre " SELECT ".
- La largeur de la colonne affichée est **la plus grande** parmi tous les ordres " SELECT ".
- Si une clause ORDER BY est utilisée, elle doit faire référence au **numéro de la colonne** et non à son nom, car le nom peut être différent dans chacun des ordres " SELECT ".

```
Exemple :
select to_char(date_etape,'DD') from vt_etape
intersect
select to_char(n_etape) from vt_etape;
```

1.4.6 **Opérateur de Produit Cartésien**

1.4.6.1 Définition

Le produit cartésien de deux tables R et S de formats quelconques est une table T ayant pour attributs la concaténation des attributs de R et de S et dont les tuples sont constitués de toutes les combinaisons d'un tuple de R à un tuple de S.

En général, le produit cartésien n'est pas souhaité et est bien souvent le résultat d'une jointure imparfaite.

Il arrive parfois que l'on souhaite distribuer à une table ou à une requête, toutes les données d'une autre table. Dans ce cas, le produit cartésien est la solution.

1.4.6.2 Syntaxe en SQL89

- ```
  SELECT <liste d'attributs> FROM <nom_table1>, <nom_table2>
  ```

1.4.6.3 Syntaxe en SQL2

- ```
  SELECT <liste d'attributs> FROM <nom_table1> cross join <nom_table2>
  ```

1.5 Les vues

1.5.1 Concept de vue

Une vue est un objet de niveau logique qui correspond à une vision logique des données d'une ou plusieurs tables. La notion de vue se rapproche de celle de schéma externe (ANSI) ou de sous-schéma.

Une vue ou " table virtuelle " n'a pas d'existence propre : **aucune donnée n'est associée à la vue**. Seule sa description est stockée **sous forme d'une requête** faisant intervenir des tables ou d'autres vues.

Les vues peuvent être utilisées pour :

- ✗ Répondre à des besoins de **confidentialité**,
- ✗ Maîtriser les mises à jour en assurant des contrôles de cohérence,
- ✗ Offrir **plus de commodités** aux utilisateurs, dans la manipulation des données, en ne leur présentant de façon simplifiée que le sous-ensemble des données qu'ils ont à manipuler,
- ✗ **Sauvegarder des requêtes** dans le dictionnaire de données.

1.5.2 Création d'une vue

La commande `CREATE VIEW` permet de créer une vue en spécifiant la requête constituant la définition de la vue :

- ```
  CREATE [OR REPLACE ] VIEW <nom_vue>
  [ (<nom_colonne1> [,<nom_colonne2>, ...] ) ]
  AS    SELECT <nom_colonneA>[,<nom_colonneB>, …]
  FROM …
  [WHERE …]
  ou
  ```
- ```
  CREATE [OR REPLACE ] VIEW <nom_vue>
  [ (<nom_colonne1> [,<nom_colonne2>, ...] ) ]
  AS    SELECT <nom_colonne1>[,<nom_colonne2>, …]
  FROM …
  [JOIN …]
  [WHERE …]
  ```

La commande REPLACE permet de remplacer le texte d'une vue sans utiliser la commande DROP VIEW <nom_vue>. Les noms de colonne associés à l'ordre CREATE permettent de remplacer les colonnes projetées dans l'ordre SELECT par de nouveaux noms.

Exemples :

```
create or replace view vCoureur_belge as
select * from vt_app_nation where code_cio='BEL';

create or replace view vCoureur_allemand (num, lenom, leprenom) as
select n_coureur,nom,prenom
from vt_app_nation
join vt_coureur using(n_coureur)
where code_cio='GER';
```

Attention : certaines pratiques non conseillées d'ailleurs habituellement peuvent empêcher de créer une vue.

- Une projection sélectionnant 2 colonnes de même nom empêche la création d'une vue. Cela peut-être le cas par exemple avec une jointure " join on " ou SQL1

```
create or replace view vCoureur_allemand  as
select * from vt_app_nation ap
join vt_coureur co on ap.n_coureur = co.n_coureur
where code_cio='GER';
00957. 00000 -  "duplicate column name"
```

- Une projection sélectionnant le résultat d'une fonction d'agrégation sans alias provoquera une erreur.

```
create or replace view v_nb_coureurs  as
select count(*) from vt_coureur;
```

00998. 00000 - "must name this expression with a column alias"

1.5.3 Utilisation d'une vue

1.5.3.1 Interrogation "à travers une vue"

Une vue peut être référencée dans un ordre SELECT en lieu et place d'une table de la base.
Tout se passe comme s'il existait une table mais en réalité cette table est virtuelle et est recomposée à chaque appel de la vue.

Exemples :

```
select * from vCoureur_allemand;
select * from vCoureur_allemand
where lenom like 'A%';
```

1.5.3.2 Mise à jour "à travers une vue" : *A éviter car il y a de nombreuses conditions à respecter.*

Certaines vues peuvent être l'objet de mise à jour par les instructions " insert ", " update ", " delete " mais avec de nombreuses restrictions. Voici les restrictions imposées par Oracle sur la requête de la vue afin que celle-ci soit modifiable :

- pas d'opérateurs ensemblistes
- pas de fonction d'agrégation
- pas de clause group by ou order by
- pas de sous-requête
- pas de collection dans un select (objet-relationnel).

Si la vue comporte une jointure, les instructions LMD sont très restreintes et ne peuvent concerner qu'une seule table de base telle que la clé primaire de la table reste une clé pour la vue.

Exemple :

création d'une vue modifiable (v_article_fournisseur1)

```
drop view v_article_fournisseur1;
create view v_article_fournisseur1  as
select fo.fo_nom, ar.fo_numero,ar_numeroar_nom,ar_poids,ar_couleur,ar_stock,ar_pa,ar_pv
from cdi_article ar
join cdi_fournisseur fo on ar.fo_numero = fo.fo_numero;
```

création d'une vue non modifiable (v_article_fournisseur2)

```
drop view v_article_fournisseur2;
create view v_article_fournisseur2  as
select fo.fo_nom, fo.fo_numero,ar_numero,ar_nom,ar_poids,ar_couleur,ar_stock,ar_pa,ar_pv
from cdi_article ar
join cdi_fournisseur fo on ar.fo_numero = fo.fo_numero;
```

insertion dans une vue modifiable (v_article_fournisseur1)

```
insert into v_article_fournisseur1
(fo_numero,ar_numero, ar_nom,ar_poids,ar_couleur,ar_stock,ar_pa,ar_pv)
values ('F06','A85','GOMME','25','BLANC',20,1,2);

insert into v_article_fournisseur2
(fo_numero,ar_numero, ar_nom,ar_poids,ar_couleur,ar_stock,ar_pa,ar_pv)
values ('F06','A86','GOMME','25','BLANC',20,1,2);
ORA-01779: impossible de modifier une colonne correspondant à une table non protégée par clé

select * from user_updatable_columns where lower(table_name) like 'v_article_fournisseur%'
and column_name like '%NUMERO%';

OWNER   TABLE_NAME                COLUMN_NAME  UPDATABLE INSERTABLE DELETABLE
CDI     V_ARTICLE_FOURNISSEUR1    FO_NUMERO    YES       YES        YES
CDI     V_ARTICLE_FOURNISSEUR1    AR_NUMERO    YES       YES        YES
CDI     V_ARTICLE_FOURNISSEUR2    FO_NUMERO    NO        NO         NO
CDI     V_ARTICLE_FOURNISSEUR2    AR_NUMERO    YES       YES        YES
```

1.5.3.3 Suppression d'une vue

Une vue peut être détruite par la commande suivante :

- `DROP VIEW <nom_vue>;`

1.5.3.4 Renommer une vue

On peut renommer une vue par la commande suivante :

- `RENAME <ancien_nom> to <nouveau_nom>;`

1.6 SQL et les requêtes complexes

1.6.1 Les sous-requêtes

Le langage SQL étend la notion de valeur de référence au résultat d'une requête. Ainsi l'expression d'une clause WHERE peut prendre la forme :

- `SELECT... WHERE <attribut> <opérateur> (SELECT ...);`

Ou

- `SELECT... JOIN… WHERE <attribut> <opérateur> (SELECT ...);`

On dit alors que la requête, dont le résultat sert de valeur de référence dans le prédicat est une requête imbriquée ou une sous-requête.

Il est possible d'imbriquer plusieurs requêtes, le résultat de chaque requête imbriquée servant de valeur de référence dans la condition de sélection de la requête de niveau **supérieur** appelée, requête principale. Le nombre de niveaux d'imbrication est théoriquement illimité.

Il existe plusieurs types de requêtes imbriquées, selon que :

- ✗ la sous-requête ne renvoie **qu'une** ligne,
- ✗ la sous-requête renvoie **plusieurs** lignes

et que :

- ✗ la sous-requête est **indépendante** de la requête principale,
- ✗ la sous-requête est **synchronisée** avec la requête principale.

1.6.2 La sous-requête indépendante, renvoyant une seule ligne

Pour que la requête s'exécute correctement, il faut que la sous-requête retourne une ligne et une seule. Si la sous-requête ne renvoie rien ou si elle renvoie plusieurs lignes, SQL génère une erreur.

- `SELECT... WHERE <attribut> = (SELECT ...);`

Ex : Nationalité du coureur dont le n° de coureur est 260.

```
SELECT * FROM vt_nation WHERE code_cio =
(     SELECT code_cio FROM vt_app_nation WHERE n_coureur = 260 );
```

Dans cet exemple, la sous-requête est évaluée en premier, puis le résultat est utilisé pour exécuter l'interrogation principale.

1.6.3 La sous-requête indépendante, renvoyant plusieurs lignes

Une sous-requête de ce type s'utilise lorsque la condition de sélection fait référence à une liste de valeurs. La condition de sélection utilise un opérateur **IN** ou un opérateur simple (=, !=, <>, <, >, <=, >=) suivi de ALL ou ANY.

- ✗ **IN** : la condition est vraie si elle est vérifiée pour une des valeurs renvoyées par la sous-requête.
- ✗ **ANY** : la comparaison sera vraie si elle est vraie pour au moins une des valeurs renvoyées par la sous-requête.
- ✗ **ALL** : la comparaison sera vraie si elle est vraie pour chacune des valeurs renvoyées par la sous-requête.

Remarques :

- `SELECT... FROM…WHERE <attributs> IN (SELECT ...)`
  `l'opérateur IN est équivalent à l'opérateur =ANY`
- `SELECT... FROM…WHERE <attributs> NOT IN (SELECT ...)`
  `l'opérateur NOT IN est équivalent à l'opérateur !=ALL`

Exemples :

```
-- nom des coureurs qui ont participé
select nom from vt_coureur
where n_coureur in
(   select n_coureur from vt_parti_coureur );
-- coureurs qui n'ont jamais abandonné
select * from vt_parti_coureur
where n_coureur not in
(   select n_coureur from vt_abandon );
```

1.6.4 Sous-requête avec plusieurs colonnes

Il est possible de comparer le résultat d'un ordre SELECT renvoyant plusieurs colonnes à une liste d'attributs. La liste de colonnes figure alors entre parenthèses à gauche de l'opérateur de comparaison.

- `SELECT … FROM… WHERE (<col1>, <col2>, … ) <opérateur> (SELECT col1, col2, … )`

Exemple :

```
-- coureurs qui n'ont pas abandonné en 2008
select * from vt_parti_coureur where annee = 2008 and(n_coureur,annee) not in
(   select n_coureur,annee from vt_abandon );
```

1.6.5 Sous requête synchronisée avec la requête principale

SQL sait également traiter une sous-requête faisant référence à une colonne de la table de l'interrogation principale. Le traitement dans ce cas est plus complexe, car il faut évaluer la sous-requête pour chaque ligne traitée par la requête principale. On dit alors que la sous-requête est **synchronisée** avec la requête principale. La sous-requête est évaluée pour chaque ligne de la requête principale. Une ou plusieurs colonnes de la requête principale sont citées dans une ou plusieurs jointures dans la ou les sous-requêtes.

- ```
  SELECT ...
  FROM <nom_table> <nom_alias>, ....
  WHERE ...    <opérateur>
  (
        SELECT ...
        FROM <nom_table>
        WHERE <nom_colonne> <opérateur> <nom_alias>.<nom_colonne>
  )
  ```

ou

- ```
  SELECT ...
  FROM <nom_table> <nom_alias>                    [JOIN…]
  WHERE ...    <opérateur>
  (
        SELECT ...
        FROM <nom_table>
        [JOIN…]
        WHERE <nom_colonne> <opérateur> <nom_alias>.<nom_colonne>
  )
  ```

La synchronisation entre la requête principale et la sous-requête est indiquée par l'utilisation, dans la sous-requête, d'une colonne d'une table de la requête principale.

Exemple :

```
-- coureurs qui ont abandonné en 2009
select * from vt_parti_coureur co
where annee=2009 and n_coureur in
(
  select n_coureur from vt_abandon where annee=co.annee
);
```

1.6.6 Cas particulier de l'opérateur "EXISTS" (requête synchronisée)

L'opérateur EXISTS permet de construire un prédicat évalué à vrai si la sous-requête renvoie au moins une ligne.

- `SELECT ...  FROM… [JOIN …] WHERE EXISTS (SELECT ...  )`

Cet opérateur n'a d'intérêt qu'avec l'emploi de requêtes synchronisées.

Exemples :

```
select * from vt_coureur co
where exists
(   select 1 from vt_abandon where annee=2009 and n_coureur=co.n_coureur );
```

1.6.7 Autre type de requêtes complexes

On utilise des sous-requêtes de façons différentes. Cette fois-ci la sous requête remplace :

- un attribut : select (select ...) from <table>
- une table : select <attributs> from (select …)
- une condition : select <attributs> from <table> where (select …)

Exemples :

```
select co.n_coureur, nom, prenom,
  (select count(*) from vt_parti_coureur pa where pa.n_coureur=co.n_coureur) as nb
from vt_coureur co
order by nb desc;

select n_coureur from (select * from vt_coureur where rownum<5);

select * from vt_temps
where annee=2008 and
( select min(distance) from vt_etape ) < 40;
```

3 règles d'or des sous-requêtes

Règle1. Tu utiliseras les mêmes colonnes dans le select de la sous-requête qu'avant le [not] in de la requête supérieure

Règle2. Tu placeras en requête principale uniquement les tables pour lesquelles on projette des colonnes

Règle3. Tu placeras seulement une table par sous-requête

1.7 Expressions et fonctions

1.7.1 Définition et utilisation

Une expression est une combinaison de variables (contenu d'une colonne), de constantes et de fonctions au moyen d'opérateurs.

Les fonctions sont des procédures prenant une valeur dépendant de leurs arguments, qui eux-mêmes sont des expressions. Les expressions peuvent figurer :

- En tant que colonne résultat d'un SELECT,
- Dans une clause WHERE,
- Dans une clause ORDER BY.

Il existe trois types d'expressions correspondant chacune à un type de donnée Oracle.

- Les expressions arithmétiques,
- Les expressions "chaîne de caractères",
- Les expressions "date".

À chaque type correspondent des opérateurs et des fonctions spécifiques. Oracle autorise les mélanges de types dans les expressions et effectuera les conversions nécessaires : dans une expression mélangeant dates et chaînes de caractères, les chaînes de caractères seront converties en date, et dans une expression mélangeant nombres et chaînes de caractères, les chaînes de caractères seront converties en nombre.

1.7.2 La fonction NULL VALUE - NVL

Une valeur NULL en SQL est une valeur non définie. Un prédicat comportant une comparaison avec une expression ayant la valeur NULL prendra toujours la valeur faux. La fonction NVL permet de remplacer une valeur NULL par une valeur significative. NVL (chaîne, valeur_significative)

Exemple : `select nvl(moyenne,-1) from vt_etape order by moyenne desc;`

1.7.3 Fonction conditionnelle DECODE

Cette fonction permet de façon assez compacte d'enchaîner des expressions conditionnelles.

Syntaxe :

```
select <liste d'attributs>
decode (<expr>, <val1>, <résult1> {, <vali>, <résulti>} [, <défaut>])
```

Si la valeur d'<expr> est <val1> alors on aura <résult1> etc ...

Exemple :

```
select nom,prenom ,annee_naissance,
decode (code_cio, 'FRA','Français','BEL','Belge','ITA','Italien','Autre' ) as
nationalité
from vt_app_nation
join vt_coureur using(n_coureur);
```

1.7.4 Expressions arithmétiques

1.7.4.1 Opérateurs

Une expression arithmétique peut contenir : des noms de colonnes, des constantes, des fonctions arithmétiques combinées au moyen des opérateurs suivants : +, -, *, /.

La priorité des opérateurs (identiques à la priorité des opérateurs en mathématiques)

1.7.4.2 Fonctions

Les fonctions arithmétiques intégrées à Oracle sont les suivantes :

ABS (n)	valeur absolue de n
CEIL (n)	prend la valeur de l'entier directement supérieur ou égal à n
COS(n)	calcule le cosinus de n (en radian)
EXP(n)	calcule l'exponentielle de n
FLOOR (n)	prend la valeur de la partie entière de n (entier directement inférieur ou égal à n)
GREATEST (n1, n2, ...)	prend la valeur la plus grande des expressions arguments
LEAST (n1, n2, ...)	prend la valeur la plus petite des expressions arguments
LN(n)	Calcule le logarithme népérien de n
LOG(m,n)	Calcule le logarithme de base m de n
MOD (n, m)	n modulo m (reste de la division de n par m)
POWER (m,n)	élève m à la puissance n
ROUND (n, [, d])	arrondit n à dix puissances -d (par défaut, d = 0)
SIGN (n)	vaut 1 si n est positif, 0 si n=zéro et -1 si n est négatif
SIN(n)	calcule le sinus de n (en radian)
SQRT (n)	racine carrée de n (prend la valeur NULL si n est négatif)
TAN(n)	calcule la tangente de n (en radian)
TRUNC (n, [, d])	tronque n à dix puissances -d (par défaut, d = 0)

1.7.5 **Expressions sur chaines de caractères**

1.7.5.1 **Opérateurs**

Il existe un opérateur sur chaînes de caractères : la concaténation. Cet opérateur se note au moyen de deux caractères '|' (barre verticale). Le résultat d'une concaténation est une chaîne de caractères obtenue en mettant bout à bout les deux chaînes de caractères en argument ;

exemple : `select nom||' '||prenom from vt_coureur ;`

1.7.5.2 **Fonctions**

Les fonctions sur chaînes de caractères intégrées à Oracle sont les suivantes :

LENGTH (chaîne)	Renvoie la longueur de la chaîne
SUBSTR (chaîne, pos [, long])	extrait de la chaîne "chaîne" une sous-chaîne de longueur "long" commençant en position "pos" de la chaîne. Le paramètre long facultatif
INSTR (chaîne, sous-chaîne [, pos [, n]])	Renvoie la position de la sous-chaîne dans la chaîne (les positions sont numérotées à partir de 1). Zéro signifie que la sous-chaîne n'a pas été trouvée dans la chaîne. La recherche commence à la position "pos" de la chaîne (paramètre facultatif : vaut 1 par défaut). Une valeur négative de "pos" signifie une position rapport à la fin de la chaîne. Le dernier caractère "n" permet de rechercher la énième occurrence de la sous-chaîne dans la chaîne. Ce paramètre vaut 1 par défaut. ex : position du deuxième "E" dans les emplois
UPPER (chaîne)	Convertit les minuscules en majuscules
LOWER (chaîne)	convertit les majuscules en minuscules
INITCAP (chaîne)	Met chaque initiale de mot en majuscule et les autres lettres en minuscules.) Sont considérés comme séparateur de mots, tous les caractères qui ne sont pas des lettres
SOUNDEX (chaîne	Cette fonction calcule une valeur phonétique qui peut être (comparaison phonétique) comparée dans un prédicat à la valeur phonétique d'une autre chaîne. L'on peut ainsi, par exemple, utiliser comme critère de recherche des mots dont on ne connaît pas l'orthographe exacte
LPAD (chaîne, long [, car])	Complète (ou tronque) la chaîne "chaîne" à la longueur "long". LPAD (chaîne, long [, car]) chaîne est complétée à gauche ou à droite par le caractère (ou la chaîne de caractères) "car".
RPAD (chaîne, long [, car])	
LTRIM (chaîne, car)	Supprime les caractères à l'extrémité gauche de la chaîne "chaîne" tant qu'ils appartiennent à l'ensemble de caractères "car".
RTRIM (chaîne, car)	A une fonction analogue, les caractères étant supprimés à l'extrémité droite de la chaîne. ex : supprimer les "A" et les "M" en fin de noms.
TRANSLATE(chaîne,car_source, car _cible)	car source et car cible sont des ensembles de caractères. La fonction remplace dans chaîne chaque caractère de la liste car_source par celui de même position dans car_cible.de ex : remplacer les 'Ê' et les 'È' par des 'E' dans les noms
REPLACE(chaîne,chaîne source, chaîne cible)	La fonction remplace dans chaîne chaque séquence de la liste chaîne source par la chaîne cible.
TO_CHAR (nombre, format)	Cette fonction permet de convertir un nombre en chaîne de caractères en fonction d'un format, "nombre" est une expression de type numérique, "format est une chaîne de caractères pouvant contenir des caractéristiques.
TO_NUMBER (chaîne)	Convertit la chaîne de caractères en numérique.
ASCII (chaîne)	Est un nombre qui représente le code (ASCII ou EBCDIC selon la machine hôte) du premier caractère de la chaîne.
CHR (n)	Est le caractère dont le code (ASCII ou EBCDIC) est égal à l'expression numérique n.

exemples :

```
select dump(nom,17) from vt_coureur where n_coureur<100;

drop table test;
create table test ( txt varchar2(3));

insert into  test values ('1');
insert into  test values ('1A');
insert into  test values ('1AA');
insert into  test values ('1B');
insert into  test values ('1BB');

select * from test;

select rtrim(txt,'AB') from test ;
```

1.7.6 **Expressions de type DATE**

1.7.6.1 **Opérateurs**

Au moyen des opérateurs arithmétiques + et -, il est possible de construire les expressions suivantes :
1) date +/- nombre. Le résultat est une date obtenue en ajoutant le nombre de jours "nombre" à la date "date"
2) datel - date2. Le résultat est le nombre de jours entre les deux dates.
Attention, ces calculs ne fonctionnent pas avec des types " texte ".
3) to_char(date1,format) permet d'extraire une partie d'une date ; format peut contenir une partie de dd mm yyyy hh mi ss. Exemple : to_char(sysdate,'yyyy') permet d'extraire l'année
4) to_date(txt1,format) permet de créer une donnée de type date

1.7.6.2 **Fonctions**

Conversions d'une date en chaîne de caractères - `TO_CHAR (date, format)` où le format indique quelle partie de la date doit apparaître. Le format est une combinaison des codes suivants :

- sec (siècle avec signe) ; ce (siècle), SY.YYY (année avec signe et virgule) ; Y.YYY (année avec virgule) ; YYYY (année) ; YYY (3 derniers chiffres de l'année) ; YY (2 derniers chiffres de l'année) ; Y ; Q (numéro du trimestre dans l'année) ; WW (numéro de la semaine dans l'année) ; W (numéro de la semaine dans le mois) ; MM (numéro du mois) ; DDD (numéro du jour dans l'année) ; DD (numéro du jour dans le mois) ; D (numéro du jour dans la semaine) ; HH ou HH12 (heure sur 12 heures) ; HH24 (heure sur 24 heures) ; Ml (minutes) ; SS (secondes) ; SSSSS (secondes après minuit) ; J (jour du calendrier julien).

Les formats suivants permettent d'obtenir des dates en lettres (en anglais parfois) :

- SYEAR ou YEAR (année en toutes lettres) ; MONTH (nom du mois), MON (nom du mois abrégé sur 3 lettres) ; DAY (nom du jour) ; DY (nom du jour abrégé sur 3 lettres) ; AM ou PM (indication AM ou PM) ; BC ou AD (BC, avant Jésus Christ ou AD, après Jésus Christ-).

Les suffixes suivants modifient la présentation du nombre auquel ils sont accolés : TH (ajout du suffixe ordinal ST, ND, RD ou TH) ; SP (nombre en toutes lettres).

Conversion d'une chaîne de caractères en date : TO_DATE (chaîne, format) permet de convertir une chaîne de caractères en donnée de type date. Le format est identique à celui de la fonction TO_CHAR.

Autres fonctions :

ADD_MONTHS(date,n)	ajoute n mois à date
LAST_DAY(date)	retourne la date du dernier jour du mois de date
MONTHS_BETWEEN(date2, date1)	Calcule le nombre de mois entre date2 et date1
NEXT_DAY(date,nom_du_jour)	Donne la date du prochain jour de la semaine dont le nom est nom_du_jour
ROUND(date[,masque])	Arrondi la date en fonction du masque
SYSDATE	Renvoie la date et l'heure courantes du système d'exploitation hote.

1.8 Le groupement des données

1.8.1 Principe

Le langage SQL offre un mécanisme permettant de travailler sur des valeurs obtenues par regroupement des lignes résultats de l'exécution d'un ordre : SELECT ... FROM ... [WHERE ...].

Soit la requête : `SELECT annee, n_coureur, n_etape, c_typeaban FROM vt_abandon;`

Si on considère que les lignes retournées forment **un seul groupe**, SQL peut calculer le nombre total de coureurs présents dans "abandon" ou le nombre total de coureurs qui ont abandonné avec le type "HD".

On peut aussi obtenir ces valeurs par année en effectuant un groupement des lignes résultats par rapport au critère *année*.

On peut enfin sélectionner l'année et le type d'abandon "AB" pour lesquels il existe plus de 50 abandons dans la base de données en utilisant une condition de sélection sur la valeur nombre d'abandons par type.

1.8.2 Fonctions d'agrégat

SQL offre la possibilité de mettre en œuvre des fonctions qui affectent des calculs sur **l'ensemble** des données d'une colonne pour les lignes appartenant à un même groupe et retournent une valeur unique. À l'exception de COUNT, les fonctions d'agrégation ignorent les valeurs NULL.

Ces fonctions sont :

- AVG(expression) moyenne des valeurs d'une colonne
- SUM(expression) somme des valeurs d'une colonne
- MIN(expression) prend pour valeur la plus petite des valeurs
- MAX(expression) prend pour valeur la plus grande des valeurs
- VARIANCE(expression) variance
- STDDEV(expression) écart-type ou déviation standard
- COUNT(x) dénombre une collection de valeurs
- Cas particulier de la fonction COUNT(x)
  - COUNT(*) nombre de lignes satisfaisant une éventuelle condition
  - COUNT(<attribut>) idem ci-dessus mais ne compte pas les lignes où l'attribut est null
  - COUNT(DISTINCT <attribut>) idem ci-dessus mais ne compte que les lignes l'attribut distincte

Exemple :

```
select avg(distance) from vt_etape;
select sum(distance) from vt_etape where annee=2008;
select count(nom) "nombre de coureur" from vt_coureur;
select count(moyenne) from vt_etape;
select count(*) from vt_etape;
select count(n_etape) from vt_etape;
```

1.8.3 Définition d'un groupe

On appelle groupe un ensemble de lignes, résultat d'une requête, qui ont une valeur commune dans une ou plusieurs colonnes. Cet ensemble de colonnes est appelé **le facteur de groupage**. La définition du facteur de groupage se fait par l'intermédiaire de la clause GROUP BY de l'ordre SELECT.

Cette clause associée à une fonction d'agrégation retourne une ligne par groupe.

Il est possible de subdiviser les lignes résultats d'un ordre SELECT en plusieurs groupes. Un groupe est formé d'un ensemble de lignes ayant une ou plusieurs caractéristiques communes. Il y autant de groupes que de valeurs distinctes projetées.

Exemple :

```
select sum(distance) total_distance from vt_etape
group by annee;

select annee, sum(distance) total_distance from vt_etape
group by annee
order by total_distance desc;

select annee, max(distance), min(distance) from vt_etape
group by annee;

select annee, code_cio_d from vt_etape group by annee ;

select count(*),annee,c_typeaban from vt_abandon
group by annee,c_typeaban
order by annee, c_typeaban;
```

Un ordre SELECT avec une colonne GROUP BY donne une ligne résultat pour chaque groupe. **Les colonnes du GROUP BY sont les colonnes projetées en plus des fonctions d'agrégats.**

La subdivision en groupes peut se faire en plusieurs niveaux, chaque niveau est défini par une expression.

1.8.4 **Cohérence du résultat**

Dans la liste des attributs résultat d'un ordre SELECT avec une clause GROUP BY ne peuvent figurer que des caractéristiques de groupe, c'est-à-dire :

- soit des fonctions de groupe,
- soit des expressions figurant dans la clause GROUP BY.

Exemple :

```
select n_coureur, count(*) from vt_parti_coureur
group by n_coureur;
select n_coureur, jeune, count(*) from vt_parti_coureur
group by n_coureur;
select n_coureur, jeune ,count(*) from vt_parti_coureur
group by n_coureur, jeune;
```

1.8.5 **Sélection de groupe : la clause HAVING** (http://sqlpro.developpez.com/cours/sqlaz/ensembles/)

Il est parfois nécessaire de filtrer les résultats d'une requête en utilisant une fonction d'agrégat. La clause **WHERE agit sur les données des tables** et permet de filtrer ligne après ligne. Mais, en présence de fonction d'agrégat, le filtrage ne porte plus sur la notion de lignes, mais sur une notion de sous-ensemble de la table. En d'autres termes, le filtre, ici, doit porter sur chacun des groupes.

Parfois une sous-requête permet de résoudre le problème.

Exemple :

```
select max(distance) from vt_etape;
select * from vt_etape
where distance = max(distance); -- FAUX
select * from vt_etape
where distance =
(select max(distance) from vt_etape) ;
```

Dans d'autres cas, l'utilisation de sous-requête peut s'avérer compliqué (voire impossible ?).

Exemple : on cherche à trouver des coureurs ayant participé à plus de 7 tours.

```
select n_coureur, count(n_coureur) from vt_parti_coureur
  group by n_coureur
  order by n_coureur;
select n_coureur, count(n_coureur) from vt_parti_coureur
  group by n_coureur
  where count(n_coureur) > 7; FAUX
select n_coureur, count(n_coureur) from vt_parti_coureur p1
where
(
  select count(n_coureur) from vt_parti_coureur p2
  where p1.n_coureur = p2.n_coureur
  group by n_coureur
) > 7
group by n_coureur
order by n_coureur; -- COMPLEXE
```

La clause HAVING agit comme le filtre WHERE, mais permet de filtrer non plus les données, mais les opérations résultant des regroupements, c'est à dire très généralement toute expression de filtre devant introduire un calcul d'agrégation. Presque toujours HAVING s'utilise avec GROUP BY.

SELECT … GROUP BY { <col1> | <expression1>} [,{<col2> | <expression2>} ...]
HAVING <agrégat> <opérateur> {<littéral> | <sous-requête>}

Exemple : même question

```
select n_coureur, count(n_coureur) from vt_parti_coureur
group by n_coureur
having count(n_coureur) > 7
order by n_coureur;
```

1.8.6 **Group by rollup et group by cube (depuis Oracle 8i)**

- L'extension ROLLUP permet de produire des sous-totaux triés à chaque niveau d'un agrégat, jusqu'au total.
- La fonction CUBE, permet de créer toutes les combinaisons de sous-totaux possibles, y compris le total
- Voir Exoplus N°6

1.9 Les fonctions analytiques (pour plus d'informations voir http://lalystar.developpez.com/fonctionsAnalytiques/)

Ces fonctions permettent (depuis Oracle 8i) d'appliquer une fonction (somme, moyenne...) à un ensemble de lignes définies par rapport à la ligne courante ce qui permet de calculer aisément des sommes cumulées, des moyennes mobiles et d'accéder aux valeurs de la ligne précédente.

Les fonctions analytiques opèrent donc sur un ensemble de lignes définies de manière relative; elles se distinguent des fonctions agrégats qui opèrent sur l'ensemble d'un groupe défini par le group by. En ce sens, elles sont d'une granularité plus fine.

En conclusion, elles permettent de réaliser des opérations qu'il serait difficile et coûteux d'écrire en SQL ou PL/SQL.

Syntaxe :

```
select fonc(expres.) over ([clause de partitionnement] [clause d'ordre] [clause de
fenêtrage])
```

• `fonc` est une analytique (pour simplifier, les noms reprennent les fonctions d'agrégats). C'est `over` qui indique qu'il s'agit d'une fonction analytique.

• Clause de partitionnement : est de la forme PARTITION BY ... Elle définit un découpage des données suivant les valeurs des colonnes du PARTITION BY... Chaque valeur définit un groupe logique à l'intérieur duquel est appliquée la fonction analytique. C'est analogue au GROUP BY.

• Clause d'ordre : est de la forme ORDER BY ... [NULLS FIRST|LAST] Elle indique comment les données sont triées à l'intérieur de chaque partition. L'emploi de NULLS FIRST/LAST est facultatif : NULLS FIRST indique que les valeurs nulles apparaissent d'abord tandis que NULL LAST indique que ces valeurs apparaissent à la fin.

• Clause de fenêtrage : indique l'ensemble des lignes sur laquelle doit être appliquée la fonction. Si une clause de fenêtrage est spécifiée, une clause d'ordre doit obligatoirement l'être aussi.

- ◦ ROWS indique qu'on définit une fenêtre en termes de ligne, par exemple. ROWS 5 PRECEDING signifie qu'on prend toutes les lignes comprises entre la ligne courante et celles qui précèdent jusqu'à la 5ème incluse.
- ◦ RANGE définit une fenêtre en termes de valeur : par exemple pour avoir les dates à moins de 100 jours d'une date de référence indiquée dans la clause order by on utiliserait RANGE 100 PRECEDING. RANGE ne fonctionne qu'avec des colonnes de type numérique ou date; la colonne de référence est celle indiquée dans le order by, il ne peut y avoir qu'une unique colonne dans le order by.
- ◦ Exemples :
- ◦ ROWS 100 PRECEDING : on prend les lignes comprises entre la ligne courante et celles qui la précèdent jusqu'à la 100e incluse.
- ◦ CURRENT ROW : la fenêtre est réduite à la ligne courante
- ◦ ROWS UNBOUNDED PRECEDING : on prend toutes les lignes depuis le début de la partition jusqu'à la ligne courante incluse.
- ◦ ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW : c'est la même chose que l'exemple précédent
- ◦ ROWS BETWEEN CURRENT ROW AND 5 FOLLOWING : on prend toutes les lignes depuis la ligne courante ainsi que les suivantes jusqu'à la 5e incluse

Exemples :

```
select distinct n_coureur,
round(avg(distance/total_seconde*3600) over (partition by n_coureur),2) as moy ,
round(avg(distance/total_seconde*3600) over(),2) as moy_tot
from vt_temps
join vt_etape using (n_etape,annee)
where annee=2023
order by moy desc;

select n_coureur,distance,
sum (distance) over (partition by n_coureur order by distance) as total_cum_cour,
sum (distance) over (partition by n_coureur ) as total_coureur
from vt_temps
join vt_etape using (n_etape,annee)
where annee=2023
order by n_coureur;
```

1.10 Les requêtes hiérarchiques

Il arrive que les données d'une table correspondent à une structure de données arborescente (graphe sans cycle), Dans ce cas, Oracle propose un mécanisme permettant de projeter de façon intéressante ces données issues d'associations réflexives.

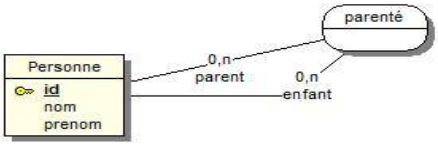
La syntaxe générale est la suivante :

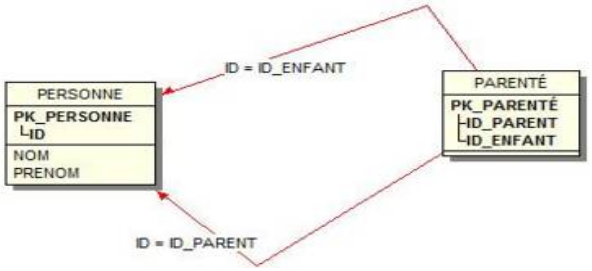
select [level,] <colonne>, <expression>
from <nom table>
[start with <condition1>]
connect by <condition2>

Level : pseudo-colonne désignant le niveau de la donnée dans l'arbre (par rapport à la racine choisie)
Start with : point de départ du parcours
connect by prior : lien d'une donnée avec la donnée associée permettant le parcours

- si <condition 2> = prior colonneSup= colonneInf, on effectue le parcours du bas vers le haut
- si <condition 2> = colonneInf= prior colonneSup, on effectue le parcours du haut vers le bas

Exemple :





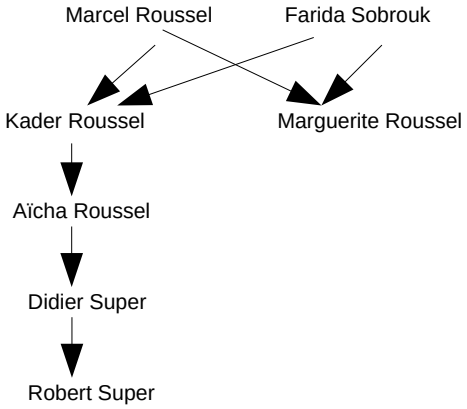

```
drop table personne cascade Constraints;
create table personne
(
idp number(2) primary key,
nom varchar2(50),
prenom varchar2(30)
);
drop table parenté cascade Constraints;
create table parenté
(
id_parent number(2),
id_enfant number(2)
);

insert into personne values (10,'Roussel','Marcel');
insert into personne values (15,'Sobrouk','Farida');
insert into personne values (18,'Roussel','Kader');
insert into personne values (19,'Roussel','Marguerite');
insert into personne values (21,'Roussel','Aïcha');
insert into personne values (25,'Super','Didier');
insert into personne values (29,'Super','Robert');


insert into parenté values (10,18);
insert into parenté values (15,18);
insert into parenté values (10,19);
insert into parenté values (15,19);
insert into parenté values (18,21);
insert into parenté values (21,25);
insert into parenté values (25,29);
insert into parenté values (29,null);

select level, idp, nom, prenom,lpad(' ',4*level-4)||id_enfant as id_enfant
from personne join parenté on personne.idp = parenté.id_parent
start with idp=10
connect by prior id_enfant = idp
order by level;

select level, idp, nom, prenom,lpad(' ',4*level-4)||id_enfant as id_parent
from personne join parenté on personne.idp = parenté.id_parent
start with idp=29
connect by id_enfant = prior idp
order by level;
```



LEVEL	IDP	NOM	PRENOM	ID_ENFANT
1	10	Roussel	Marcel	18
1	10	Roussel	Marcel	19
2	18	Roussel	Kader	21
3	21	Roussel	Aïcha	25
4	25	Super	Didier	29
5	29	Super	Robert	

LEVEL	ID	NOM	PRENOM	ID_PARENT
1	29	Super	Robert	(null)
2	25	Super	Didier	29
3	21	Roussel	Aïcha	25
4	18	Roussel	Kader	21
5	10	Roussel	Marcel	18
5	15	Sobrouk	Farida	18

2 SQL, la définition des données de la base et la mise à jour des données

2.1 La définition des données de la base

La définition des données va permettre de définir les tables, les modifier et les supprimer.

2.1.1 Les types de données

- **VARCHAR2** (lg)Chaîne de caractères de longueur variable dont la précision est de "lg" caractères. La taille maximale possible est de 4000 caractères. Il faut toujours indiquer la taille pour le type "varchar2".
- **CHAR** (lg)Chaîne de caractères de longueur fixe. La taille maximale est de 2000 caractères. La taille par défaut est 1.
- **NUMBER** (p,s). Nombre de précision **p** et d'échelle **s**. La précision **p** s'étend de 1 à 38 (38 chiffres significatifs). L'échelle **s** est le nombre de chiffres à droite du point décimal. Le type NUMBER permet de représenter les nombres négatifs, positifs, à virgule fixe, à virgule flottante. Les nombres peuvent prendre des valeurs comprises entre 10x10-130 et 9.9…9x10125.
- **LONG** Texte de longueur variable jusqu'à 2 gigaoctets. Une seule colonne de table peut avoir ce type. Le type LONG n'est pas utilisable dans les clauses WHERE, ORDER BY, GROUP BY et ne peut pas être indexé.
- **DATE** Les dates valides vont de january 1, 4712 BC à December 3, 4712 AD. Ce type permet de représenter les dates et les heures. Le format de saisie/affichage est défini par le paramètre NLS_DATE_FORMAT situé dans le fichier d'initialisation INIT.ORA. Le format retenu est le format suivant : DD/MM/YY.
- **RAW** (lg)Donnée binaire de longueur variable. La longueur maximale est de 2000 octets. La longueur doit toujours être spécifiée.
- **LONG RAW** Donnée binaire de longueur variable jusqu'à 2 gigaoctets.
- **ROWID** Chaîne hexadécimale représentant l'adresse unique d'une ligne dans sa table. Ce type permet de récupérer les valeurs retournées par la pseudo-colonne ROWID.
- **BLOB** Un grand objet de type binaire (image, vidéo, …). Taille maximale : 4 gigaoctets.
- **CLOB** Un grand objet de type texte. Taille maximale : 4 gigaoctets.

2.1.2 Les tables

2.1.2.1 La création d'une table

- La commande pour créer et définir une table

```
CREATE TABLE [<utilisateur>.]<table>
(
  <nom_col1>      <type>
  [,<nom_col2>    <type>
  [, …]]
);
```

```
Exemples :
create table article (code_article int, couleur char(20));
create table etudiant (numEtu number(3), nom varchar2(30),prenom varchar2(30) );
```

- Création d'une structure de table à partir d'une requête SQL

```
CREATE TABLE [nom_utilisateur.]nom_table
[(<nom_col1> [, <nom_col2> [, …]])]
AS
SELECT …
FROM …
[WHERE {<condition> | 0=1}]
```

(la condition " 0 = 1 "permet d'éliminer automatiquement les lignes retournées par la requête).

```
Exemple :
create table gamelle as select * from bidon where 0=1;
```

2.1.2.2 **La modification de la structure d'une table**

- La commande ALTER TABLE permet :
  - d'ajouter de nouvelles colonnes ADD

```
ALTER TABLE [<nom_utilisateur>.]<nom_table>
ADD (
<Nom_col1> <type (lg)> [NOT NULL]
[, <nom_col2> <type (lg)> [NOT NULL] [, …]]
);
```

  - de modifier des colonnes existantes MODIFY

```
ALTER TABLE [<nom_utilisateur>.]<nom_table>
MODIFY    (
<Nom_col1> [<type (lg)>] [{NULL | NOT NULL}][default expression]
[, <nom_col2> [<type (lg)>] [{NULL | NOT NULL}] [default expression]
[, …]]
);
```

  - de supprimer des colonnes DROP COLUMN

```
ALTER TABLE [nom_utilisateur.]nom_table
DROP COLUMN     (<Nom_colonne>);
```

  - de renommer une colonne RENAME COLUMN

```
ALTER TABLE [nom_utilisateur.]nom_table
RENAME COLUMN <ancien_nom> TO <nouveau_nom>
```

  - de renommer une table RENAME

```
ALTER TABLE [<nom_utilisateur>.]<ancien_nom_table>
RENAME TO <nouveau_nom_table>;
```

Exemples :

```
alter table bidon add couleur char(20);
alter table bidon modify type varchar2(20);
alter table bidon drop column couleur;
alter table bidon rename column type to description;
alter table gamelle rename to bidon;
rename bidon to gamelle;
```

2.1.2.3 **La suppression d'une table**

On utilise la commande DROP TABLE pour supprimer une table ainsi que ses données

- DROP TABLE [<nom_utilisateur>.] <nom_table>

Exemples :

```
drop table bidon;
```

2.1.2.4 **Les colonnes invisibles**

Depuis Oracle 12c, on peut rendre une colonne INVISIBLE. Une requête de type « SELECT * » ne permet pas de consulter les colonnes invisibles, il faut les spécifier dans la requête nominativement, de même la commande DESC sur la table ne donne pas d'informations sur la colonne invisible à moins d'utiliser le paramètre COLINVISIBLE de SQL*Plus. L'information sur les colonnes invisibles est visible dans la vue USER_TAB_COLUMNS.

Exemples :

```
drop table testInvisible;
create table testInvisible
(
  num number(3),
  nom varchar2(10),
  date_insert date invisible
);

insert into testInvisible values (1,'Dupont', sysdate); -- interdit
insert into testInvisible values (2,'Dubois'); -- autorisé
insert into testInvisible (num, nom, date_insert) values (3,'Dulong', sysdate); -- autorisé

select * from testInvisible;
```

NUM	NOM
2	Dubois
3	Dulong

```
select num,nom,date_insert from testInvisible;
```

NUM	NOM	DATE_INSERT
2	Dubois	(null)
3	Dulong	21/08/22

```
SET COLINVISIBLE OFF;
desc testInvisible;
```

```
Nom NULL ? Type
--- ------ ------------
NUM        NUMBER(3)
NOM        VARCHAR2(10)
```

```
SET COLINVISIBLE ON;
desc testInvisible;
```

```
Nom                    NULL ? Type
---------------------- ------ ------------
NUM                           NUMBER(3)
NOM                           VARCHAR2(10)
DATE_INSERT (INVISIBLE)       DATE
```

2.1.3 **Les index**

Les index ont pour but principal d'améliorer les performances d'accès aux données. Ils permettent également d'interdire la duplication de valeurs pour certains attributs (clé primaire).

2.1.3.1 **La création d'index**

- La création explicite d'index : La commande CREATE INDEX permet de créer un index construit à partir d'une ou plusieurs colonnes de la table référencée.

```
CREATE [UNIQUE] INDEX <nom_index>
ON <nom_table> (<nom_col1> [ASC | {DESC}] [,<nom_col2> [ASC | {DESC}]…]
```

nom_index : est l'identificateur attribué à l'index
UNIQUE : les valeurs dupliquées sont interdites (clé primaire)
ASC, DESC : indique si l'index est classé par ordre croissant ou décroissant (ASC par défaut).

- La création implicite de l'index : cette création se réalise par l'intermédiaire d'une contrainte d'intégrité de clé primaire (PRIMARY KEY). Dans ce cas, le nom de l'index est identique à celui de la contrainte d'intégrité.
  La contrainte empêche la duplication des valeurs de certains attributs et l'index a pour but principal d'améliorer les performances d'accès aux données. Aussi, si une table a moins de 300 lignes (documents Oracle), il devient inutile de créer des index. Ainsi, la suppression de l'index n'empêchera pas la gestion de la duplication des valeurs qui sera laissée à la contrainte.

2.1.3.2 **La suppression d'index**

```
DROP INDEX <nom_index>
```

2.1.4 **Les synonymes**

Oracle offre la possibilité d'attribuer plusieurs noms à un même objet. Ces noms additionnels sont appelés synonymes. Ils permettent de faire référence aux objets sans avoir besoin de spécifier leurs propriétaires ni leurs localisations. Contrairement aux alias de table, les synonymes sont sauvegardés dans le dictionnaire de données et être consultés avec la vue USER_SYNONYMS.

Oracle permet la création de synonymes pour les objets d'un schéma. Un objet peut être l'un des types suivants :

- table,
- vue,
- séquence,
- fonction,
- procédure,
- package,
- snapshot (vue matérialisée, MVIEW)
- synonyme

2.1.4.1 **La création de synonyme**

CREATE [PUBLIC] SYNONYM [nom schéma.] nom synonyme FOR [nom schéma.} nom objet[@lien base]

Exemple :

```
create materialized view mv1 refresh on commit complete as select * from essai_ville ;
create synonym sv1 for mv1;
```

2.1.4.2 **La suppression de synonymes**

Pour supprimer un synonyme privé ou public, on utilise la commande suivante :

DROP [PUBLIC] SYNONYM [nom schéma.] nom_synonyme

2.1.5 **Les séquences**

Une séquence est un objet qui génère des valeurs entières séquentielles uniques. Ces valeurs sont utilisées pour les clés primaires ou uniques. On accède aux valeurs d'une séquence avec deux peudo-colonnes :

- CURRVAL retourne la valeur courante de la séquence
- NEXTVAL retourne la valeur suivante de la séquence

```
CREATE SEQUENCE [<nom_utilisateur>.]<nom_sequence>
[INCREMENT BY <entier>]
[START WITH <entier>]
[ { MAXVALUE <entier> | [NOMAXVALUE] } ]
[ { MINVALUE <entier> | [NOMINVALUE] } ]
[ { CACHE <entier> | [NOCACHE] } ]
[ { CYCLE | [NOCYCLE] } ]
[ { ORDER | [NOORDER] } ]
```

- `INCREMENT BY` : indique la valeur entière d'incrément. La valeur peut être positive ou négative. La

valeur par défaut est 1.

- `START WITH` : indique la valeur de départ de la séquence. La valeur par défaut est 1.
- `MAXVALUE` : valeur maximale de la séquence (ne pas dépasser 10^{29}-1)
- `MINVALUE` : valeur minimale de la séquence (ne pas dépasser -10^{27}-1)
- `CYCLE` : la séquence boucle et passe de la valeur maximale à la valeur minimale si l'incrément est positif et de la valeur minimale à la valeur maximale si l'incrément est négatif..
- `CACHE` : spécifie le nombre de valeur que le cache doit contenir en préallocation. Le minimum est de 2 et le maximum est de `CEIL(MAXVALUE-MINVALUE)/ABS(INCREMENT)`

Pour connaître le numéro de séquence courant ; le numéro rendu par `CURRVAL` correspond au dernier numéro construit par `NEXTVAL` : `SELECT <nom_sequence>.currval FROM DUAL;`

Lors d'une insertion dans une table, on construit la nouvelle valeur en utilisant `NEXTVAL` :

```
INSERT INTO <nom_table> (<nom colonne numérotant les enregistrements>, …)
VALUES      ( <nom_sequence.NEXTVAL>, … )
```

Pour supprimer une séquence, on utilise la commande `DROP SEQUENCE`

Exemple :

```
create sequence seq_num_coureur start with 1 increment by 5;
insert into vt_coureur (numcoureur,nom,prenom)
values (seq_num_coureur.nextval,'MOREAU','Christophe');
drop sequence seq_num_coureur;
```

Les séquences sont particulièrement intéressantes quand deux tables l'unicité d'une donnée (héritage par exemple).

Elles peuvent être partagées par plusieurs transactions ce qui n'est pas le cas avec « select (max +1) from … »

2.1.6 **Les colonnes auto-incrémentées**

Depuis la version 12c on peut déclarer des colonnes auto-incrémentées en suivant la syntaxe SQL standard.

La syntaxe est : `generated [ always | by default [ on null ] ] as identity [ ( identity_options ) ]`

- Always : il n'est pas permis d'entrer manuellement une donnée dans la colonne de type auto.
- By default: il est permis d'entrer manuellement une donnée dans la colonne de type auto.

Exemples :

```
drop table test1;
create table test1 ( id number(2) generated always as identity , n number(2) );

insert into test1  values (5); -- interdit
insert into test1 (n) values (5);
insert into test1 (n) values (4);
insert into test1 (id,n) values (8,6); -- interdit
select * from test1;
```

ID	N
1	5
2	4

```
drop table test2;
create table test2 ( id number(2) generated by default as identity , n number(2) );

insert into test2  values (5); -- interdit
insert into test2 (n) values (5);
insert into test2 (id,n) values (8,6); -- autorisé
select * from test2;
```

ID	N
1	5
8	6

```
drop table test3;
create table test3 ( id number(2) generated by default as identity  (start with 10 increment by -2 maxvalue 10),
n number(2) );

insert into test3 (n) values (5);
insert into test3 (n) values (7);
select * from test3;
```

ID	N
10	5
8	7

2.2 La mise à jour des données

2.2.1 L'insertion des données

On utilise la commande INSERT pour insérer une ou plusieurs lignes dans une table.

```
INSERT INTO [<nom propriétaire>].<nom_table>
[(<nom col1>,<nom col2>)]
VALUES  {<liste de valeurs> | <requête SQL> }
```

Exemple :

```
insert into vt_coureur (n_coureur, nom, prenom)
values (1505,'JALABERT','Laurent');
```

2.2.2 La mise à jour des données

On utilise la commande UPDATE pour changer les valeurs d'un ou plusieurs attributs dans une ou plusieurs lignes d'une table :

```
UPDATE [<nom_propriétaire>.]<nom_table> [<t_nom_alias>]
SET  <nom de colonne>  = {<expression> | <requête SQL>}
           [, <nom de colonne> = {<expression> | <requête SQL>}
           [, … ]]
[WHERE <condition> ]
```

• SET <nom_col> permet de donner une nouvelle valeur à "nom_col" ;
• WHERE <condition> condition détermine les lignes sur lesquelles va porter la mise à jour. Sinon toutes les lignes sont mises à jour. La condition peut consister en une sous-requête SQL.

```
Exemples :
update vt_coureur set n_coureur=1510
where n_coureur=1505;

update vt_parti_coureur set n_coureur=1510
where n_coureur=1505
and annee=2009;
```

2.3 La suppression des données

2.3.1 L'insertion des données

On utilise la commande DELETE pour effacer une ou plusieurs lignes d'une table :

```
DELETE [FROM] [<nom_propriétaire>.]<nom_table> [WHERE <condition de recherche>]
```

WHERE condition : condition permet de définir les lignes d'une table qui seront effacées ; l'absence de condition va entraîner l'effacement de toutes les lignes. La condition peut consister en une sous-requête SQL.

```
Exemples :
delete from vt_coureur
where n_coureur between 150 and 300 ;

delete from vt_parti_coureur
where n_coureur =
  (select n_coureur from vt_coureur where nom = 'CONTADOR');
```

3) Les transactions (se reporter au cours spécifique)