Département Informatique
1ème année

Bases de Données
SQL*PLUS d'ORACLE

Rosine CICCHETTI, Lotfi LAKHAL

Sommaire

Préambule

Développé par IBM Corporation comme support du modèle relationnel, le langage SEQUEL (Structured English QUEry Language) a fait l'objet d'une première implémentation commerciale en 1979 par la société Relational Software (aujourd'hui ORACLE Corporation). SEQUEL est ensuite devenu SQL.

SQL (Structured Query Language) est un langage qui se veut simple (basé sur des mots clefs anglais). C'est un standard ANSI[1]/ISO[2] en matière de langage de manipulation de bases de données relationnelles (une première norme SQL date de 1989, la seconde[3] appelée SQL2 de 1992, la dernière SQL3 de 1999). Malgré cette standardisation, il existe des différences entre les implémentations de SQL proposées par les divers Systèmes de Gestion de Bases de Données (SGBD) : on parle de dialectes SQL.

Un des points forts de SQL est de s'adresser à toutes les catégories d'utilisateurs : développeurs d'applications, gestionnaires ou administrateurs de bases de données (DBA), responsables sécurité mais aussi utilisateurs finals. L'objectif du langage est de leur offrir une interface avec un SGBD et tous les traitements SQL sont des instructions au SGBD. Les différents types d'utilisateurs n'ont pas recours aux mêmes fonctionnalités du langage. Plus précisément, SQL peut être vu comme :

- un *Langage de Définition des Données* (LDD), i.e. qu'il offre les commandes de création et de modification de la structure de la base ;

- un *Langage de Manipulation des Données* (LMD) puisqu'il permet les mises à jour des données d'une base et l'interrogation ;

- un *Langage de Contrôle des Données* (LCD). SQL propose, en effet, les commandes de gestion de la confidentialité des données, la prise en compte des contraintes d'intégrité et les mécanismes de gestion de transactions.

SQL n'est pas un langage de programmation. Il n'est pas procédural mais ensembliste. Lors de la manipulation des données, il permet de gérer des ensembles de tuples " comme un tout " (sans avoir à traiter les données enregistrement par enregistrement comme le font les langages de programmation lors de la manipulation de fichiers). L'exécution de toute requête SQL met en œuvre (de manière transparente pour l'utilisateur) l'optimiseur du SGBD qui détermine la manière la plus efficace d'accéder aux données et de calculer le résultat demandé.

SQL peut être utilisé directement pour manipuler une base de données (SQL interactif) ou être intégré dans un langage de programmation de haut niveau (*embedded SQL*). Dans ce dernier cas, les ordres SQL sont insérés dans le code du langage de programmation et un pré-compilateur en effectue la traduction dans le langage hôte. Tous les SGBD commercialisés proposent de nombreux langages hôtes (de Cobol à Java).

À propos de ce document :

[1] American National Standard Institute.
[2] International Standard Organisation.
[3] SQL2 est la version de référence mais différents SGBD (dont ORACLE) prennent en compte partiellement SQL3.

Ce support de cours a été conçu pour les étudiants en deuxième année de DUT Informatique à l'IUT d'Aix-en-Provence qui connaissent le modèle relationnel et son algèbre. Il est consacré à l'utilisation interactive de l'un des dialectes SQL : SQL*PLUS implémenté dans le SGBD ORACLE (la version de référence pour ce support est : ORACLE8i enterprise version 8.1.7). À partir de la version 8, ORACLE combine les concepts du modèle relationnel et certains concepts objets. Cependant, ce document se limite aux aspects relationnels. De plus, ce support de cours s'adresse à de futurs développeurs d'applications. Parfois évoquées, les tâches des administrateurs et/ou responsables sécurité n'y sont pas détaillées. En outre, bien que faisant partie intégrante des compétences des développeurs d'applications bases de données, l'intégration de SQL*PLUS dans un langage hôte n'est pas abordée dans ce document mais fait l'objet d'un autre polycopié. SQL*PLUS respecte le standard ANSI/ISO 92 mais il se situe au premier niveau de la norme[4] et offre des possibilités supplémentaires (comme la recherche arborescente ou certaines fonctionnalités SQL3).
Merci d'envoyer toute remarque, suggestion ou correction d'erreur à : cicchetti@iut.univ-aix.fr

Vocabulaire et conventions :

Le vocabulaire utilisé en SQL est différent de celui du modèle relationnel. Les associations sont données ci-après. Dans la suite du document, les termes équivalents sont utilisés indifféremment.

Table (*table*)	Relation
Colonne (*column*)	Attribut
Ligne (*row*)	Tuple

Dans ce support, les mots clefs SQL sont indiqués en police `courier`, en majuscules et en gras, les commandes ORACLE sont en `courier`, en minuscules et en gras. Ces mêmes conventions sont utilisées pour les éléments spécifiques d'ORACLE (comme certains noms de fonctions ou les pseudo-colonnes) faisant partie des mots clefs réservés de SQL*PLUS. Les éléments des ordres SQL ou des commandes ORACLE que doit spécifier l'utilisateur sont indiqués entre < et >. L'alternative est représentée par : `|`. Le caractère optionnel d'une clause ou d'un argument est indiqué en l'encadrant de `[` et `]`. Les références bibliographiques sont indiquées entre crochets et renvoient à la bibliographie à la fin de ce document.

La base de données exemple :

La base de données exemple, utilisée dans ce support, propose la gestion très simplifiée d'une compagnie aérienne. Ses relations sont les suivantes (les clefs primaires sont soulignées. Les clefs étrangères sont en gras et en italique, elles font référence aux clefs primaires de même nom) :

PILOTE (NUMPIL, NOMPIL, PRENOMPIL, ADRESSE, SALAIRE, PRIME, DATE_NAI)
AVION (NUMAV, NOMAV, CAPACITE, LOCALISATION)
VOL (NUMVOL, ***NUMPIL***, ***NUMAV***, DATE_VOL, HEURE_DEP, HEURE_ARR, VILLE_DEP, VILLE_ARR).

On suppose qu'un vol, référencé par son numéro NUMVOL, est effectué par un unique pilote, de numéro NUMPIL, sur un avion identifié par son numéro NUMAV.

[4] Il existe trois niveaux croissants de conformité à la norme SQL2 : entry, intermediate, full [8].

Chapitre 1 - Environnement de travail ORACLE

Remarque importante

Si vous disposez d'Oracle Sqldeveloper alors passer au chapitre 2.

1.1 – Session de travail ORACLE

Une session de travail ORACLE démarre avec l'appel du langage SQL*PLUS et, sauf incident, se termine par l'ordre de sortie **`exit`**. Pour pouvoir se connecter sous Oracle, il faut avoir été déclaré comme utilisateur du SGBD et se voir conférer mot de passe et privilèges par l'administrateur (Cf. paragraphe 4.2).

Appel de SQL*PLUS : **`sqlplus /`**
vous êtes directement reconnu comme utilisateur ORACLE, sans avoir à vous identifier ni à donner votre mot de passe. Le prompt SQL> s'affiche.
Remarque : votre nom d'utilisateur ORACLE est de la forme : `OPS$<loggin_unix>`.

Sortie de SQL*PLUS : **`exit`**
Avant de quitter ORACLE, les mises à jour effectuées durant la session de travail sont écrites définitivement dans la base (Cf. Paragraphe 4.1, instruction **`COMMIT`**).

Aide : **`help`**
Cette commande peut être utilisée pour obtenir une aide sur les commandes ORACLE (non SQL). Utilisez **`help index`** pour avoir la liste des sujets traités. Une documentation en ligne est fournie dans [1].

1.2 - Éditeur d'ORACLE

Lorsque vous êtes sous Oracle, vous pouvez directement taper une commande ORACLE qui s'exécutera en utilisant la touche " Entrée " ou bien commencer à saisir un ordre SQL*PLUS. Quand la première ligne de cet ordre est saisie et que vous utilisez " Entrée ", le prompt `2>` apparaît : vous êtes en fait entré sous l'éditeur d'ORACLE qui attend la deuxième ligne de votre ordre. Pour sortir de cet éditeur, il suffit de taper " Entrée " lorsque le prompt affiche un nouveau numéro de ligne.
L'éditeur d'ORACLE est un éditeur ligne donc, évidemment, peu pratique et qu'il faut éviter d'utiliser pour des requêtes longues. Cependant, la connaissance des quelques commandes de base peut vous permettre d'aller rapidement à l'essentiel. En fait, tout ordre SQL est mémorisé dans un espace tampon, le buffer de travail SQL*PLUS. Il est alors possible de modifier le contenu du buffer en utilisant les commandes suivantes :

`l``[`**`ist`**`]`	liste les lignes de l'ordre SQL*PLUS courant, i.e. contenu dans le buffer.
`l``[`**`ist`**`]` `n`	affiche la ligne de numéro `n` qui devient la ligne courante.
`l``[`**`ist`**`]` `m n`	liste les lignes de `m` à `n` (`m` doit toujours être inférieur (ou égal) à `n`) La dernière ligne affichée de numéro `n` devient la ligne courante.
`l``[`**`ist`**`]` `*`	affiche la ligne courante.
`l``[`**`ist`**`]` **`last`**	affiche la dernière ligne.
`n`	la ligne courante devient la ligne `n`.

c[**hange**]/ch1/ch2[/]	remplace, dans la ***ligne courante***, la ***première*** occurrence de la chaîne `ch1` par la chaîne `ch2`. Le dernier caractère " / " de la commande peut généralement être omis mais il est indispensable lorsque `ch1` doit être remplacé par un espace (sinon `ch1` est simplement supprimé). Le caractère de séparation des arguments de la commande est généralement le " / " mais si `ch1` ou `ch2` contiennent ce caractère, tout autre caractère non alphanumérique peut lui être substitué. Il est également possible de substituer non seulement la première occurrence de `ch1` mais aussi tout le texte la précédant ou la suivant sur la même ligne en utilisant " … " avant ou après `ch1`.
a[**ppend**] texte	ajoute un texte donné à la fin de la ligne courante. Pour ajouter le caractère " ; ", il faut le doubler : " ; ; ".
i	passe en mode insertion après la ligne courante. Le prompt affiche le nouveau numéro de ligne.
del	supprime la ligne courante.
del n	supprime la ligne numéro `n`.
del m n	supprime toutes les lignes de `m` à `n` (`n` doit être supérieur (ou égal) à `m`).
del last	supprime la dernière ligne.

Exemple *: L'utilisateur tape la requête suivante puis " Entrée " lorsque le prompt `4` s'affiche.*

```
SQL>  SELECT NUMPIL, NOMPIL, SALAIRE
  2   FROM PILOTE, VOL
  3          WHERE PILOTE.NUMPIL = VOL.NUMPIL AND VILLE_DEP =
     'MARSEILLE'
  4
```

S'il exécute alors la commande `l`, il visualise la requête courante. L'affichage obtenu est le suivant où le caractère " `*` " indique la ligne courante dans le buffer.

```
SQL> l
  1  SELECT NUMPIL, NOMPIL, SALAIRE
  2  FROM PILOTE, VOL
  3* WHERE PILOTE.NUMPIL = VOL.NUMPIL AND VILLE_DEP = 'MARSEILLE'
SQL>
```

Supposons que l'utilisateur veuille modifier les attributs mentionnés dans la clause SELECT en remplaçant NUMPIL par NUMVOL, il doit tout d'abord exécuter la commande suivante pour que la première ligne de la requête devienne la ligne courante :

```
SQL>  1
```

Le système affiche :

```
  1*  SELECT NUMPIL, NOMPIL, SALAIRE
SQL>
```

Il peut alors effectuer la modification voulue par :

```
SQL>  c /PIL/VOL
```

Le système affiche :

```
1*  SELECT NUMVOL, NOMPIL
```

***Exemple** : La requête courante dans le buffer est donnée ci-dessous.*

```
SQL>  SELECT NUMPIL, SALAIRE + PRIME
  2   FROM PILOTE
```

Pour modifier l'expression de calcul et la remplacer par une division arithmétique, les commandes suivantes peuvent être utilisées :

```
SQL>  1
1*  SELECT NUMPIL, SALAIRE + PRIME
SQL>  c * + PRIME*/10
```

*La modification utilisant le caractère de séparation * est exécutée et le système affiche :*

```
1*  SELECT NUMPIL, SALAIRE/10
```

***Exemple** : La requête courante dans le buffer est donnée ci-dessous.*

```
SQL>  SELECT NUMPIL, SALAIRE + PRIME FROM PILOTE
```

La commande suivante supprime tout le texte à partir du symbole +.

```
SQL>  c/+…/ /
1*  SELECT NUMPIL, SALAIRE
```

Sauvegarde d'une requête

Un ordre SQL présent dans le buffer peut être stocké dans un fichier, qui aura automatiquement l'extension .sql, par :

save <nom_fich> **append**	ajoute le contenu du buffer à la fin du fichier <nom_fich> ;
save <nom_fich> [**create**]	effectue la sauvegarde s'il n'existe pas de fichier de même nom ;
save <nom_fich> **rep**[**lace**]	sauvegarde le contenu du buffer dans un fichier existant dont le contenu précédent est écrasé.

Exécution d'une requête

Un ordre SQL*PLUS tapé directement sous l'éditeur d'ORACLE est exécuté, ***en étant toujours en mode insertion*** (le prompt est un numéro de ligne et non pas SQL>), par :

; à la fin de la dernière ligne de l'ordre SQL*PLUS ;

/ seul sur la dernière ligne de l'ordre. Préférez cette deuxième possibilité qui est aussi

valide en PL/SQL[5] (où le " ; " signale la fin de toute instruction) et permet donc, dans un même script, de combiner ordres SQL et programmes en PL/SQL.

Le dernier ordre SQL*PLUS saisi et/ou exécuté sous ORACLE est toujours présent dans le buffer. Vous pouvez l'afficher à nouveau en tapant simplement **`l`**, le modifier en utilisant les commandes données précédemment et vous pouvez l'exécuter par :

/	exécution de l'ordre courant dans le buffer sans affichage de cet ordre ;
`r`[**`un`**]	exécution avec affichage à l'écran de l'ordre SQL*PLUS courant.

Un ordre, préalablement sauvegardé dans un fichier d'extension .sql, peut être chargé dans le buffer par : **`get`** `<nom_fich>`
Il peut alors être visualisé, modifié et exécuté comme indiqué précédemment.

Un ordre stocké dans un fichier d'extension .sql peut être chargé et directement exécuté par :

`r`[**`un`**] `<nom_fich>`	chargement et exécution avec affichage de l'ordre ;
`start` `<nom_fich>`	chargement et exécution sans affichage l'ordre ;
`@<nom_fich>`	chargement et exécution sans affichage l'ordre.

1.3 - Utilisation d'un autre éditeur et des commandes Unix

Lors d'une session de travail, ORACLE vous permet également de faire appel à l'éditeur de votre choix pour créer un fichier contenant un ordre SQL. Ce fichier doit avoir une extension .sql. Pour accéder à cet éditeur tout comme pour exécuter une commande unix (à l'exception de `cd`), en étant dans l'environnement ORACLE, il suffit d'utiliser :

`host` `<commande_unix>|<appel_editeur> <nom_fichier>.sql`

ou simplement

`!``<commande_unix>|<appel_editeur> <nom_fichier>.sql`

Exemple *: les commandes suivantes sont directement saisies sous ORACLE.*

```
SQL> !mkdir TP1
SQL> !vi TP1/req_01.sql
```

□

Lorsque le texte de l'ordre est saisi dans un éditeur externe, terminez votre fichier en tapant / sur une nouvelle ligne. Sauvegardez et quittez l'éditeur, vous êtes à nouveau dans l'environnement ORACLE (le prompt `SQL>` s'affiche). Pour exécuter l'ordre sauvegardé dans votre fichier, il suffit d'utiliser les commandes données précédemment. Elles peuvent échouer si vous avez oublié de donner l'extension `.sql` en sauvegardant votre fichier. Si vous exécutez le contenu d'un fichier créé avec un autre éditeur et ne contenant pas le caractère final déclenchant l'exécution (/), vous vous retrouvez sous l'éditeur d'ORACLE en mode insertion (un numéro de ligne s'affiche en guise de prompt), il suffit alors de taper /.
Sous ORACLE, pour accéder à Unix, utilisez **`host`**, puis **`exit`** pour revenir sous ORACLE.

*1.4 - Quelques commandes ORACLE (non SQL*PLUS)*

Les commandes présentées dans ce paragraphe sont des commandes proposées par le noyau d'ORACLE. À la différence des ordres SQL*PLUS, ces commandes sont exécutées dès que la touche " Entrée " est utilisée (donc elles ne nécessitent pas de caratère final comme le " ; ") et

[5] Langage procédural pour le développement d'applications proposé par ORACLE.

elles ne sont pas mémorisées dans le buffer SQL*PLUS. Toutes ces commandes peuvent être intégrées dans un fichier d'extension .sql contenant un ou plusieurs ordres SQL.

1.4.1 - Sauvegarde des résultats de l'exécution d'un ordre SQL*PLUS

Il peut être nécessaire de conserver les résultats d'un ordre SQL*PLUS par exemple pour les contrôler ou les imprimer. La commande **spo[ol]**, nécessitant en argument le nom d'un fichier de sortie, permet l'écriture dans un fichier de toutes les exécutions ultérieures. ORACLE donne automatiquement au fichier utilisé l'extension `.lst`. Pour arrêter l'écriture dans ce fichier, il faut utiliser **spo[ol] off**.

***Exemple** : L'utilisateur a créé un fichier listepil.sql contenant une requête SQL. Il peut conserver le résultat de l'exécution de cette requête, dans un fichier appelé `sortie`, par :*

```
SQL> spool sortie
SQL> @listepil
SQL> spool off
```

*L'utilisation de **r** `listepil` à la place `@listepil` permet de sauvegarder dans le fichier de sortie le code de l'ordre SQL et ses résultats.* □

1.4.2 - Description de la structure d'une relation

La structure ou intension d'une relation, à laquelle l'utilisateur a accès et dont il connaît le nom, peut être obtenue simplement en utilisant : **desc[ribe]** <nom_table>

1.4.3 - Présentation des résultats

Les résultats d'une requête sont présentés à l'écran avec un format par défaut pas forcément très agréable ni très intelligible : le nom des attributs est utilisé comme intitulé des colonnes et la largeur de ces colonnes correspond à la taille définie pour le type des attributs. Cette largeur étant une taille maximale, l'affichage de chaque tuple peut nécessiter plusieurs lignes à l'écran et rendre le résultat peu lisible. De plus, le nombre de lignes par défaut à afficher par page (écran) est plus réduit que le nombre de lignes qui peuvent être effectivement visualisées. Tous ces paramètres peuvent être ajustés.

Pour modifier les intitulés des colonnes, la commande suivante est utilisée :

col[umn] <nom_attribut> **heading** <nouvel_entete>

où <nouvel_entete> doit être donnée entre apostrophes s'il comporte des espaces.

***Exemple** : attribution de nouveaux intitulés aux attributs NUMPIL et NOMPIL.*

```
SQL> column NUMPIL heading 'Numéro du pilote'
SQL> column NOMPIL heading Nom
```

□

Pour changer la largeur des colonnes, la commande est la suivante :

col[umn] <nom_attribut> **for[mat]** <nouveau_format>

où <nouveau_format> est :

- si l'attribut est numérique
  - une série comprenant autant de 9 que la taille voulue (éventuellement avec des décimales),
  - une série constituée d'autant de 0 que nécessaire (éventuellement avec des décimales) si les zéros non significatifs doivent être affichés,

Ces séries peuvent être précédées d'un **s** pour que les signes + ou - soient affichés. Elles peuvent être précédées ou suivies d'un **l** pour l'utilisation du symbole monétaire local (ou d'un **$** pour le dollar).

- une expression de forme `An` où `n` est le nombre de caractères souhaité pour l'affichage d'un attribut de type alphanumérique ou date.

Lorsque le format d'affichage d'un attribut est modifié, le nouveau format est utilisé pour toute la session de travail et pour les divers attributs dans les différentes relations portant le nom spécifié. Si la valeur d'un attribut est plus grande que la taille indiquée pour l'affichage, plusieurs lignes sont utilisées (attributs alphanumériques) ou des # apparaissent (attributs numériques).

***Exemple** : les commandes suivantes permettent de modifier le format d'affichage.*

```
SQL> column NOMPIL format A15
SQL> column PRENOMPIL heading Prénom format A15
SQL> column SALAIRE format l9999.99
SQL> column PRIME format l000.99
```

□

Pour supprimer, au cours d'une session de travail, toutes les commandes de présentation des colonnes, utilisez : **clear col**[**umns**]. Pour les suspendre puis les ré-appliquer, les commandes sont : **column** <nom_attribut> **off** et **column** <nom_attribut> **on**.

Les commandes **repf**[**ooter**] 'texte du pied de page' ou **reph**[**eader**] 'texte de l'entête' permettent d'afficher un message à la fin de la dernière page ou au début de la première. Les apostrophes ne sont utiles que si le texte à afficher contient des blancs. De même **tti**[**tle**] [<format>] 'Titre' [<format>] ou **bti**[**tle**] [<format>] 'Titre' [<format>] permettent d'indiquer un titre en haut (**t** pour top) ou en bas (**b** pour bottom) de chaque page. <format> peut prendre une ou plusieurs des valeurs suivantes : **le**[**ft**], [**r**]**ight**, **ce**[**nter**], **bold** ou **skip** n qui introduit un saut de n lignes avant ou après le titre. Les commandes de formatage précédentes peuvent être suspendues par : <nom_commande> **off**, et ré-appliquer en utilisant : <nom_commande> **on**.

Pour éviter l'affichage d'une même valeur d'attribut, l'utilisateur peut utiliser la commande suivante en introduisant éventuellement un saut de ligne(s), lors du changement de valeur de cet attribut : **break on** <nom_attribut> [**skip** n] où n est le nombre de lignes à sauter. Cette commande est intéressante pour améliorer la lisibilité des résultats d'une requête SQL intégrant une clause de tri (Cf. paragraphe 3.1.9). L'annulation de cette commande se fait par : **clear break**.

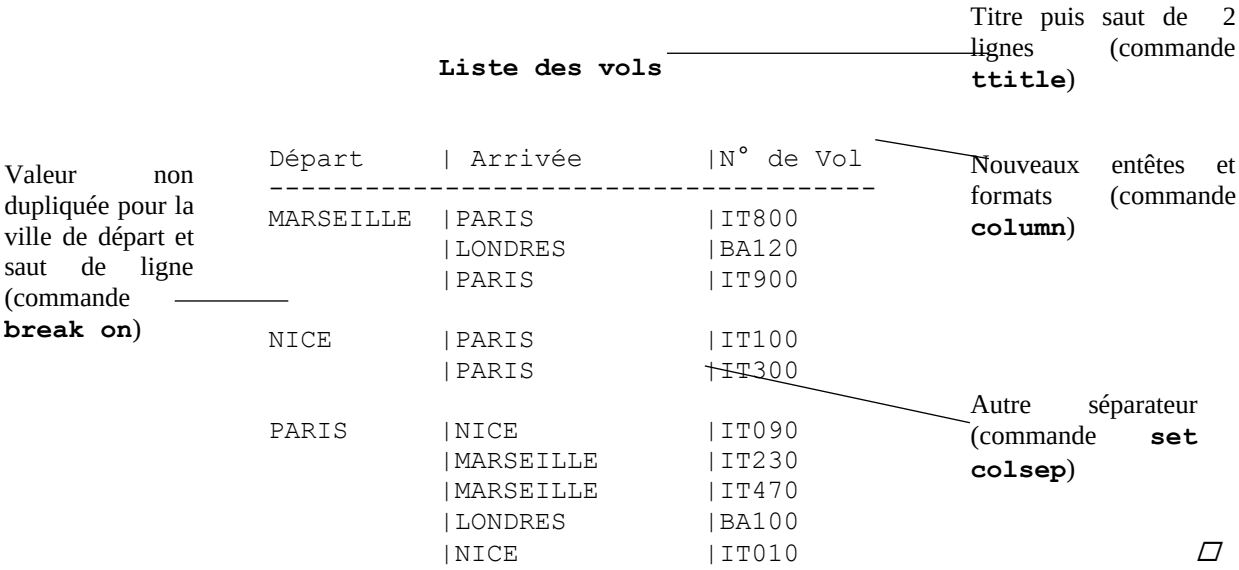
La commande **set** est utilisée pour modifier la valeur de divers paramètres telle que la longueur des pages pour l'affichage (**set pages**[**ize**] n où n est le nombre de lignes par page, le séparateur de colonnes (**set colsep** 'caractere'), l'affichage des valeurs nulles (**set null** 'texte à afficher')...
Elle permet également d'utiliser le caractère spécial " & " dans une chaîne de caractères saisie comme valeur d'un attribut. Ce caractère a une signification particulière pour ORACLE car il introduit des variables dans une requête (Cf. paragraphe 3.1.2.3). Pour intégrer " & " dans une chaîne, il faut au préalable utiliser : **set escape** \. La chaîne 'Dupont & Dupond' peut alors être insérée dans la base par : 'Dupont \& Dupond'.

Sauf suspension ou annulation, toutes les commandes de présentation s'appliquent pour toute la session de travail et ***seulement pour cette session***. Elles peuvent être incluses dans un fichier contenant un ordre SQL.

***Exemple** : les commandes de présentation suivantes sont exécutées.*

```
SQL> set pagesize 25
SQL> set colsep '|'
SQL> ttitle center bold 'Liste des vols' skip 2
SQL> column NUMVOL heading 'N° de Vol' format A10
SQL> column VILLE_DEP heading Départ format A15
SQL> column VILLE_ARR heading Arrivée format A15
SQL> break on VILLE_DEP skip 1
```

L'utilisateur exécute ensuite une requête rendant les villes de départ (triées) et d'arrivée ainsi que le numéro de vol. L'affichage qu'il obtient est le suivant.



1.4.4 - Consultation de paramètres ORACLE

La commande **show** permet de consulter la valeur de diverses variables systèmes, comme le nom de l'utilisateur (**show user**), la version d'ORACLE utilisée (**show rel**[**ease**]), l'entête défini ou le pied de page (**show reph**[**eader**] ou **show repf**[**ooter**]), les titres (**show tti**[**tle**] ou **show bti**[**tle**])…

*1.5 - Commentaires dans le code SQL*PLUS*

Dans un fichier contenant un ordre SQL, les commentaires peuvent être indiqués par :

`--`	précédant le texte du commentaire qui doit s'achever en fin de ligne ;
`/*` et `*/`	encadrant le texte du commentaire.

Chapitre 2 - SQL comme langage de définition des données

Un langage de définition de données doit permettre de manipuler les structures de données qui, dans un SGBD relationnel, sont fondamentalement et d'un point de vue logique des relations. Néanmoins un schéma de base de données ORACLE peut contenir d'autres composants que des relations, e.g. des index, des séquences, des synonymes, des commentaires…
Ce chapitre décrit toutes les capacités de SQL*PLUS pour manipuler l'intension d'une base de données. Il faut, bien entendu, pouvoir définir les relations, i.e. décrire leur schéma et les différentes contraintes d'intégrité auxquelles leurs données seront soumises (Cf. paragraphe 2.2). Il faut aussi pouvoir les faire évoluer au cours du temps pour qu'elles continuent à refléter l'univers réel qu'elles représentent. Diverses opérations sont offertes pour modifier la structure d'une base : ajout ou suppression d'attributs et de contraintes, élimination de relations… (Cf. paragraphe 2.3). Pour optimiser l'accès aux données, des index doivent également être créés (Cf. paragraphe 2.6). Une facilité pour la génération de valeurs de clef primaire est également offerte par la création de séquences (Cf. paragraphe 2.4) ainsi qu'une aide à la documentation des applications par la création de commentaires (Cf. paragraphe 2.5).
Comme tous les SGBD relationnels, ORACLE stocke tous les éléments constituant le schéma de la base dans un dictionnaire de données automatiquement maintenu par le système dès qu'un composant est créé, modifié ou supprimé (Cf. paragraphe 2.7.1).
Évidemment utiliser SQL en tant que langage de définition de données n'est pas donné à tout utilisateur, il faut que ce dernier soit apte à effectuer les manipulations de structure et qu'à ce titre il se voit attribuer certains droits ou privilèges par l'administrateur ORACLE (Cf. paragraphe 4.2).

Les identificateurs ORACLE
Tous les " composants " définis dans une base de données ORACLE (relations, attributs, contraintes, index…) doivent être nommés. Les noms valides sont des chaînes de caractères alphanumériques débutant obligatoirement par un caractère alphabétique. Leur longueur maximale est de 30 caractères, ils peuvent inclure les trois caractères spéciaux suivants : " _ ", " $ ", " # " (les autres sont interdits comme " + " ou " - " réservés pour les opérations arithmétiques). Bien sûr les mots clefs réservés du langage ne peuvent pas servir de nom (par exemple, ne pas utiliser **DATE**, **TYPE**, **GROUP**…). Il est également conseillé de ne pas utiliser de caractères accentués car ils sont interdits dans certaines versions d'ORACLE (anglaise ou américaine en particulier) et peuvent donc nuire à la portabilité des applications. SQL étant insensible à la casse, les mots clefs du langage peuvent être indifféremment saisis en minuscules ou en majuscules (mais par convention, ils apparaissent dans ce support en majuscules). Il en est de même des identificateurs des composants d'une BD. Par exemple, dans un ordre SQL, la relation PILOTE peut être indiquée indifféremment par : `pilote`, `PILOTE`, `Pilote` ou `PiLoTe`… Néanmoins tous les identificateurs des composants d'une base sont stockés en majuscules dans le dictionnaire de données (Cf. paragraphe 2.7.1). C'est pourquoi, dans ce document, ils sont toujours indiqués en majuscules.
Il est évidemment conseillé d'utiliser des identificateurs parlants et explicites.

2.1 - Types syntaxiques des attributs

Dans SQL*PLUS, comme dans la grande majorité des dialectes SQL, le concept de domaine n'est pas pris en compte explicitement[6]. Le concepteur de la base doit donc se contenter, pour la définition des attributs d'une relation, de types syntaxiques. Les plus courants sont donnés dans le tableau 1. Il doit ensuite définir des contraintes de domaine (Cf. paragraphe 2.2).

[6] bien qu'il fasse partie de la norme SQL3 [8].

Type syntaxique	Description
VARCHAR2(n)	Chaînes de caractères de longueur variable (au maximum `n`). La longueur maximale `n` doit être comprise entre 1 et 4000.
CHAR[**(n)**]	Chaînes de caractères de longueur fixe de `n` caractères. La longueur `n` doit être comprise entre 1 (valeur par défaut) et 2000. ***Toute valeur donnée à un attribut de ce type et excédant `n` caractères est tronquée. Toute valeur comportant moins de `n` caractères est complétée par des espaces***.
NUMBER[**(n**[**, m**]**)**]	Valeurs numériques de longueur maximale `n` (sans compter le séparateur décimal qui est le point et les chiffres non significatifs, e.g. la valeur 06.1 a une taille de 2) et de `m` décimales. Bien sûr `n` doit être supérieur ou égal à `m`. Si la taille maximale n'est pas précisée, elle est par défaut de 22 chiffres.
DATE	Les valeurs des attributs de ce type sont des estampilles temporelles, i.e. dates et heures. Dans la version française, le format par défaut est : `JJ/MM/AA HH:MM:SS`, e.g. '10/02/02' ou '15/06/02 08:30:00' sont des valeurs syntaxiquement correctes[7].
LONG	Flots de caractères pouvant aller jusqu'à 2 giga octets de taille. ***Ce type doit être réservé pour le stockage de texte***. Il existe de nombreuses restrictions[8] sur les attributs de type **LONG**. Il ne peut exister qu'au plus un attribut de ce type par relation et il ne peut pas être indexé. Les sélections ne peuvent pas être opérées sur ces attributs, ni les partitionnements, ni les recherches arborescentes, ni les opérations ensemblistes. La seule contrainte d'intégrité admise sur de tels attributs est **NOT NULL**.

Tableau 1 : principaux types syntaxiques d'ORACLE

Outre les types indiqués dans le tableau 1, ORACLE propose divers types permettant le stockage de données non structurées telles que des images, du texte, de la vidéo… Voir [1] pour une liste exhaustive.

Remarques : (a) les valeurs de type alphanumérique, de même que les dates, doivent être saisies entre apostrophes (e.g. 'A340', 'Marseille', '14/12 /04'). Si la valeur saisie contient une apostrophe, il faut doubler le caractère (e.g. saisir 'O''Neill' pour stocker dans la base la valeur O'Neill).
(b) Si un attribut numérique n'admet que des valeurs entières, son type doit être **NUMBER**`(n, 0)` ce qui permet d'interdire la saisie de décimales. Par exemple, un attribut déclaré de type **NUMBER**`(3)` peut admettre les valeurs : 891 ; 0.891 ; 8.91 ou 89.1. Si un attribut est de type **NUMBER**`(3, 0)`, seule la première valeur (891) de la série précédente est admissible.
(c) Au niveau des données stockées dans une base, il y a distinction entre minuscules et majuscules. Par exemple, supposons que toutes les données aient été saisies en majuscules dans la base. La recherche des vols dont la ville de départ est 'marseille' ou 'Marseille' rend systématiquement un résultat vide. Pour éviter les problèmes ultérieurs, pensez à uniformiser les valeurs lors de la saisie ou lors des comparaisons (fonctions **upper**, **lower** ou **initcap**, Cf. Paragraphe 3.1.3.2).

[7] DD-MON-YY HH:MM:SS est le format américain par défaut, où MON correspond aux trois premiers caractères du nom du mois en anglais, e.g. '10-FEB-02' ou '15-JUN-02 08:30:00' sont des valeurs possibles.
[8] D'autres contraintes existent aussi en PL/SQL.

(d) SQL*PLUS ne propose pas de type booléen (contrairement à PL/SQL).

2.2 - Création de relation

Il existe deux façons de créer une nouvelle relation. La première est de définir explicitement son schéma (i.e. ses attributs avec leur type syntaxique et ses contraintes d'intégrité). La relation créée est vide et des tuples devront ensuite y être insérés. La seconde technique consiste à copier tout ou partie d'une relation existante ou, de manière encore plus générale, d'une base de données. En fait, la source de la copie est n'importe quel résultat d'une requête SQL soumise au SGBD. Si ce résultat est vide au moment de la création de la relation, cette dernière sera simplement dotée d'un schéma ou intension. Si le résultat est non vide, la nouvelle relation aura une intension et une extension (i.e. l'ensemble des tuples retournés par la requête). Le nombre maximum d'attributs dans une relation est de 1.000.
La première technique de création doit être utilisée lors du développement d'une nouvelle application ou lors d'ajout de nouvelles informations dans la base. La seconde est utilisée lors de l'évolution d'une base existante (restructuration d'une relation, sans perte de données, par exemple) ou lorsque des relations calculées (à partir d'autres données stockées) sont introduites.

2.2.1 - Création de nouvelles relations

Cette première technique de création de relation consiste à spécifier non seulement les attributs de la nouvelle relation avec leur type mais aussi les contraintes d'intégrité structurelle portant sur ces attributs (intégrité de domaine, de relation, de référence) et les contraintes d'unicité et de présence obligatoire de valeurs. La commande SQL*PLUS de création d'une nouvelle relation doit respecter la syntaxe suivante.

```
CREATE TABLE <nom_table>
(<nom_colonne1> <type1> [DEFAULT <expression1>] [NOT NULL] [UNIQUE]
[,<nom_colonne2> <type2> [DEFAULT <expression2>] [NOT NULL] [UNIQUE]…]
[, <contrainte1>[, <contrainte2>… ]])
```

où :

`<nom_table>` est le nom de la relation à créer (respectant les règles de nommage données précédemment), ce nom doit être unique pour un même créateur ;

`<nom_colonne>` est un nom d'attribut (respectant les mêmes règles). Dans une même relation, les noms des attributs doivent être distincts. Cependant, dans une même base de données, deux attributs peuvent porter le même nom à condition qu'ils appartiennent à deux relations différentes (d'où la convention généralement adoptée de nommer de la même manière clef primaire et clef étrangère associée). De plus deux attributs de même nom peuvent être définis dans deux relations de même nom du moment qu'elles ont été créées par deux utilisateurs différents[9] ;

`<type>` est l'un des types syntaxiques donnés précédemment (Cf. tableau 1) ;

`DEFAULT` `<expression>` permet de spécifier une valeur par défaut pour un attribut. `<expression>` peut être une constante compatible avec le type de l'attribut

[9] Ceci explique que les étudiants puissent travailler en TP sur des " objets " ORACLE de même nom sans ambiguïté ni conflit et ceci s'explique par la structure même du dictionnaire de données (Cf. paragraphe 2.7.1).

ou le résultat de certaines fonctions de calcul horizontal (e.g. **sysdate**, Cf. paragraphe 3.1.3.2). Cependant il n'est pas possible de spécifier dans cette expression le nom d'un autre attribut ;

NOT NULL ces mots clefs permettent d'imposer l'existence de valeurs pour l'attribut indiqué (les valeurs nulles ne sont pas tolérées) ;

UNIQUE permet d'imposer l'unicité des valeurs pour l'attribut précédent (cependant les valeurs nulles sont autorisées). Si cette contrainte porte sur une combinaison d'attributs alors elle est spécifiée indépendamment et après la déclaration des attributs concernés par :
CONSTRAINT <nom_contrainte>
UNIQUE (<colonne1[, colonne2[,colonne3…]]>)

<contrainte> est la spécification d'une contrainte d'intégrité structurelle, portant sur un ou plusieurs attributs de la relation. Cette spécification prend la forme :
CONSTRAINT <nom_contrainte> <spec_contrainte> [<etat>]
où les trois arguments à spécifier sont les suivants :

1. <nom_contrainte> est le nom attribué à la contrainte respectant les conventions des identificateurs ORACLE (ce nom doit être unique dans une relation).

2. <spec_contrainte> peut prendre l'une des formes suivantes en fonction de la nature de la contrainte d'intégrité à exprimer :

   - **PRIMARY KEY**(<attribut1>[, <attribut2>, …])
     l'attribut spécifié (ou la combinaison d'attributs[10]) est déclaré comme la clef primaire de la relation. Ses valeurs doivent exister (pas de valeurs **NULL**) et être uniques. Pour implémenter cette contrainte, ORACLE a recours à un index primaire (Cf. paragraphe 2.6). Une contrainte de clef primaire ne peut pas porter sur des attributs de type **LONG** ;

   - **FOREIGN KEY**(<attribut1>[, <attribut2>, …])
     REFERENCES <nom_relation_associée>(<att1>[, <att2>, …])
     [**ON DELETE CASCADE** | **ON DELETE SET NULL**]
     permet de spécifier une contrainte de référence entre un attribut (ou plusieurs) clef étrangère (<attribut1>[, <attribut2>, …]) et l'attribut associé (ou la combinaison d'attributs) clef primaire dans une relation (<att1>[, <att2>, …]). Il faut donc que ce dernier attribut (ou combinaison) ait été spécifié comme une clef primaire lors de la création de la relation associée ou qu'il ait été déclaré avec la contrainte d'unicité de valeurs (**UNIQUE**), sinon la création de la contrainte de référence échoue.
     Il est possible de spécifier une " autoréférence " : la clef étrangère fait alors référence à la clef primaire de la ***même*** relation.
     La clause facultative **ON DELETE CASCADE** permet une propagation automatique des suppressions de tuples dans la relation contenant la clef étrangère lors des destructions de tuples dans la relation référencée <nom_relation_associee>. En spécifiant **ON DELETE**

[10] 32 attributs au maximum peuvent faire partie d'une clef primaire.

SET NULL, toute valeur de clef étrangère associée à la clef primaire d'un tuple détruit dans <nom_relation_associee> est mise à **NULL**. Une clef étrangère ne peut pas inclure d'attribut de type **LONG**.

- **CHECK**(<nom_attribut | expression> <condition>)
  introduit une contrainte de domaine sur un attribut ou sur le résultat d'un calcul horizontal (Cf. paragraphe 3.1.3). La condition s'exprime de manière similaire aux conditions de sélection (Cf. paragraphe 3.1.2) et peut être :
  - <nom_attribut> θ <valeur> où θ ∈{=, !=, <, <=, >, >=}
  - une combinaison de conditions <nom_attribut> θ <valeur> liées par les opérateurs logiques **AND** ou **OR**
  - <nom_attribut> **IN** (<liste_valeurs>)
  - <nom_attribut> **BETWEEN** <borne_inf> **AND** <borne_sup>
  - <nom_attribut> **LIKE** '<chaine_generique>'

3. <etat> peut être **ENABLE** ou **DISABLE**. Dans le premier cas, la contrainte est active dès la création de la relation et donc toute opération de mise à jour ultérieure doit la vérifier. Dans le second cas, la contrainte est désactivée, le SGBD ne la vérifie donc pas lors des mises à jour ultérieures. L'état initial d'une contrainte peut être modifié (Cf. paragraphe 2.3.3).

Remarques : (a) il est évidemment possible de spécifier plusieurs contraintes d'intégrité de référence ou de domaine dans une même relation.
(b) En revanche, une contrainte de clef primaire est forcément unique dans une relation. Ainsi pour spécifier qu'un attribut (ou une combinaison d'attributs) est une clef candidate (sans être clef primaire), il faut combiner les contraintes d'intégrité **NOT NULL** et **UNIQUE**.
(c) Toute contrainte est forcément nommée. Cet identificateur est soit donné par l'utilisateur lors de la spécification de contrainte, soit généré automatiquement par ORACLE. Pour retrouver le nom d'une contrainte, il faut consulter les tables système (Cf. paragraphe 2.7.1).

***Exemple** : La commande suivante permet de créer la relation AVION en indiquant simplement ses attributs et leur type ainsi que certaines valeurs par défaut.*

```
CREATE  TABLE  AVION (
    NUMAV NUMBER(4,0),
    NOMAV VARCHAR2(10),
    CAPACITE NUMBER(3,0) DEFAULT 0,
    LOCALISATION VARCHAR2(20) DEFAULT 'PARIS')
```
□

***Exemple** : l'ordre SQL suivant permet de créer la relation PILOTE en spécifiant ses attributs et leur type respectif ainsi que la contrainte d'intégrité de relation (clef primaire). Une contrainte sur la valeur maximale de l'expression SALAIRE + PRIME est également spécifiée.*

```
CREATE  TABLE  PILOTE (
-- déclaration des attributs et de leur type
NUMPIL NUMBER(4,0), NOMPIL VARCHAR2(20), PRENOMPIL VARCHAR2(15),
ADRESSE VARCHAR2(20), SALAIRE NUMBER(7,2), PRIME NUMBER(7,2),

-- déclaration de la contrainte d'intégrité de relation (clef primaire)
```

```
CONSTRAINT CLEF_PRIM_PILOTE PRIMARY KEY (NUMPIL),


-- déclaration d'une contrainte d'intégrité de domaine
CONSTRAINT DOM_SALAIRE_PRIME CHECK (SALAIRE + PRIME <= 10000))
```

Après l'exécution de cet ordre SQL, la commande **`desc`** *PILOTE rend le résultat suivant.*

```
Name                           Null?    Type
------------------------------ -------- ------------
NUMPIL                         Not Null NUMBER(4)
NOMPIL                                  VARCHAR2(20)
PRENOMPIL                               VARCHAR2(15)
ADRESSE                                 VARCHAR2(20)
SALAIRE                                 NUMBER(7,2)
PRIME                                   NUMBER(7,2)
```

NUMPIL étant déclaré comme clef primaire, l'interdiction de valeurs nulles est automatique. Un index primaire est créé par ORACLE. □

***Exemple** : la création de la relation VOL avec ses attributs et ses contraintes est proposée ci-après. On suppose ici que les relations PILOTE et AVION ont été préalablement créées et que les attributs NUMPIL et NUMAV ont été spécifiés comme clefs primaires.*

```
CREATE  TABLE  VOL
(NUMVOL VARCHAR2(7), NUMPIL NUMBER(4,0), NUMAV NUMBER(4,0),
VILLE_DEP VARCHAR2(20), VILLE_ARR VARCHAR2(20), HEURE_DEP NUMBER(4,2),…,

-- Déclaration de la contrainte d'intégrité de relation (clef primaire)
CONSTRAINT CLEF_PRIM_VOL PRIMARY KEY(NUMVOL),

/*  Déclaration  d'une  contrainte  d'intégrité  de  référence  (clef
étrangère) */
CONSTRAINT CLEF_ETR_VOL_PIL FOREIGN KEY(NUMPIL)
REFERENCES PILOTE(NUMPIL),

/* Déclaration de contraintes d'intégrité de domaine */
CONSTRAINT DOM_VILLE_DEP
   CHECK(VILLE_DEP IN ('PARIS', 'LYON', 'LILLE', 'TOULOUSE', …)),
CONSTRAINT DOM_VILLE_ARR
   CHECK(VILLE_ARR IN ('PARIS', 'LYON', 'LILLE', 'TOULOUSE', …)),

CONSTRAINT DOM_HEURE_DEP CHECK(HEURE_DEP BETWEEN 0 AND 23.59) )
```
□

***Exemple** : supposons que la relation VOL ait été définie comme suit.*

```
CREATE  TABLE  VOL
(NUMVOL VARCHAR2(7),…,
/*  Déclaration  d'une  contrainte  d'intégrité  de  référence  (clef
étrangère) avec propagation des suppressions */
CONSTRAINT CLEF_ETR_VOL_AV FOREIGN KEY(NUMAV)
 REFERENCES AVION(NUMAV) ON DELETE CASCADE,…)
```

Toute suppression d'un tuple dans la relation AVION entraîne automatiquement la suppression de tous les vols assurés par cet avion. □

Domaines Sémantiques vs. Contraintes d'Intégrité

La spécification de contraintes structurelles et leur mise en œuvre par le SGBD sont évidemment essentielles pour le développement d'applications bases de données. Elles ne sont pourtant qu'un palliatif à l'absence d'implémentation du concept de domaine sémantique défini par E. F. Codd lors de la proposition du modèle relationnel.

Deux exemples simples permettent d'illustrer les limitations de la spécification de contraintes (par rapport à la définition de domaines) :

- si *n* attributs quelconques (i.e. n'étant ni clef primaire ni clef étrangère) dans le schéma partagent le même domaine sémantique (énuméré ou intervalle), il faut spécifier pour chacun d'eux la même contrainte de domaine " en dur " lors de la création des relations concernées alors que la définition d'un seul domaine sémantique aurait suffi…
- De plus, comme la déclaration de contraintes se fait " en dur ", si la contrainte considérée évolue au cours du temps (e.g. élimination ou ajout de valeurs pour un énuméré, restriction ou extension d'un intervalle de valeurs admissibles), les contraintes pour les *n* attributs concernés doivent être supprimées puis recréées, car ce type de modification de contrainte n'est pas proposé.

Évidemment, ceci ne va pas dans le sens d'une plus grande flexibilité et d'une plus grande facilité de maintenance des applications bases de données.
On peut cependant contourner le problème en " simulant " la notion de domaine mais ***uniquement pour les énumérations***. Pour cela, il suffit de :

- créer une relation `r` réduite à un seul attribut `A` qui sera clef primaire ;
- insérer, pour cet attribut, toutes les valeurs admissibles de l'énumération ;
- déclarer les *n* attributs concernés par la contrainte comme des clefs étrangères référençant la clef primaire `A`.

Cette solution a des avantages. L'énumération est spécifiée une seule fois via des ordres d'insertion pour donner les valeurs de `A` (avec des ordres **`INSERT`**, Cf. paragraphe 3.2.2). De plus, si la contrainte évolue au cours du temps, de simples opérations de mise à jour des données suffisent pour la modifier et ***il n'y a pas à faire de maintenance du code SQL***.

Exemple *: une relation VILLE est créée pour servir de domaine sémantique aux attributs VILLE_DEP, VILLE_ARR, ADRESSE et LOCALISATION.*

```
CREATE  TABLE  VILLE
(VILLE VARCHAR2(30), CONSTRAINT CLEF_PRIM_VILLE PRIMARY KEY(VILLE))
```

La création de la relation VOL devient :

```
CREATE  TABLE  VOL
(NUMVOL VARCHAR2(7), NUMPIL NUMBER(4,0), NUMAV NUMBER(4,0),
VILLE_DEP VARCHAR2(20), VILLE_ARR VARCHAR2(20),…
…
CONSTRAINT CLEF_ETR_VILLE_DEP FOREIGN KEY(VILLE_DEP)
        REFERENCES VILLE(VILLE),
CONSTRAINT CLEF_ETR_VILLE_ARR FOREIGN KEY(VILLE_ARR)
        REFERENCES VILLE(VILLE),
…)
```

Si une nouvelle ville est desservie, il suffit d'insérer une nouvelle valeur dans la relation VILLE et aucune modification du schéma n'est nécessaire. □

2.2.2 - Création de relation par copie

Une nouvelle relation peut être créée en copiant partiellement ou totalement l'intension et l'extension d'une ou de plusieurs relations existantes et/ou en calculant des données non explicitement stockées dans la base. De manière plus précise, le résultat d'une requête d'interrogation est utilisé pour créer le schéma de la nouvelle relation et éventuellement y insérer des tuples. Un pré-requis à la compréhension de ce paragraphe est de connaître la formulation de requêtes (Cf. paragraphe 3). Attention, les contraintes d'intégrité spécifiées dans les relations d'origine ne sont pas répercutées sur la nouvelle relation (à l'exception de **NOT NULL**). La syntaxe utilisée est la suivante.

```
CREATE TABLE <nom_relation> [(<liste_attributs>, <liste_contraintes>)]
AS <specification_requete>
```

Les arguments facultatifs `<liste_attributs>` et éventuellement `<liste_contraintes>` doivent être spécifiés dans les deux cas suivants :

- l'utilisateur souhaite modifier le nom des attributs copiés dans la nouvelle relation ;
- des contraintes d'intégrité doivent être déclarées pour la nouvelle relation.

Dans le premier cas, `<liste_attributs>` donne la liste des attributs, séparés par des virgules, de la nouvelle relation. Pour chaque attribut, il est possible de donner une valeur par défaut comme lors de la création classique de relation (Cf. paragraphe 2.2.1). Chaque nom d'attribut dans cette liste correspond à un attribut ou une expression retourné par la requête de définition et le nouvel attribut a ***le même type*** que l'attribut ou l'expression associé dans la requête. L'ordre de création prend la forme suivante :

```
CREATE TABLE <nom_relation>
(<nom1> [DEFAULT <expr1>] [,<nom2> [DEFAULT <expr2>] …] )
AS SELECT <attribut1 | expression1> [,<attribut2>|<expression2>,… ]…
```

`<nom1>` est utilisé pour renommer le premier argument `<attribut1 | expression1>` de la clause **SELECT** de la requête, `<nom2>` renomme le deuxième argument et ainsi de suite.

Il est également possible d'indiquer que la requête d'interrogation retourne tous les attributs des relations utilisées en remplaçant les arguments de la clause **SELECT** par le caractère `*`. Dans ce cas, il faut que l'utilisateur mentionne autant de nouveaux noms qu'il existe d'attributs dans les relations concernées et il doit respecter l'ordre de création de ces attributs dans les relations.

Si la liste `<nom1>… <nom2>…` est omise, les attributs de la nouvelle relation prennent le même nom que ceux cités dans la clause **SELECT** de la requête. Si jamais la liste des résultats indiqués dans la requête de définition contient des expressions de calcul, il faut leur attribuer un alias, i.e. nommer ces expressions, de manière à ce que les attributs de la nouvelle relation aient tous un nom. Un alias est une chaîne de caractères respectant les règles des identificateurs d'ORACLE. Il est introduit juste après l'attribut ou l'expression qu'il renomme, simplement séparé par un espace : **SELECT** `<attribut | expression> <alias>`.

Lors de la création d'une relation, il est possible de spécifier des contraintes d'intégrité de la même manière que pour une création classique de relation mais en respectant la condition et la restriction suivantes :

- même si leur nom n'est pas modifié, les attributs de la nouvelle relation doivent être spécifiés dans `<liste_attributs>` ;
- les contraintes d'intégrité de référence ne peuvent pas être déclarées. Elles devront être ajoutées ultérieurement par un ordre de modification de relation (Cf. paragraphe 2.3.1).

Enfin, ORACLE propage automatiquement les contraintes **NOT NULL** existant dans la relation originale à la nouvelle relation. Évidemment si les tuples copiés violent une contrainte d'intégrité, la création de la relation échoue.

<specification_requete> est une requête d'interrogation en SQL*PLUS (Cf. chapitre 3). Cette requête peut être simple ou complexe, i.e. inclure des jointures, partitionnements, applications de calculs…

***Exemple** : une copie de la relation PILOTE est réalisée sans modifier le nom des attributs et sans spécification de contrainte.*

```
CREATE  TABLE  COPIE_PILOTE AS SELECT * FROM PILOTE
```
□

***Exemple** : une relation AIRBUS est créée. Ses attributs sont le numéro de l'appareil, son nom et le nombre de vols assurés. Pour créer cette relation, la requête d'interrogation associée est complexe. Elle inclut une jointure et un partitionnement ainsi qu'un calcul vertical. L'ordre suivant intègre la spécification des noms des attributs pour la relation créée.*

```
CREATE  TABLE  AIRBUS(NUMAV, NOMAV, NBRE_VOLS)
AS SELECT AVION.NUMAV, NOMAV, COUNT(*)
FROM AVION, VOL
WHERE AVION.NUMAV = VOL.NUMAV AND NOMAV LIKE 'A%'
GROUP BY AVION.NUMAV, NOMAV
```

Le même résultat peut être obtenu sans spécification de nouveaux noms mais simplement en utilisant un alias pour l'expression de calcul vertical.

```
CREATE  TABLE  AIRBUS
AS SELECT AVION.NUMAV, NOMAV, COUNT(*) NBRE_VOLS
FROM AVION, VOL
WHERE AVION.NUMAV = VOL.NUMAV AND NOMAV LIKE 'A%'
GROUP BY AVION.NUMAV, NOMAV
```
□

***Exemple** : une nouvelle relation est créée en interrogeant la table VOL. Elle contient les numéros des avions et des pilotes qui les conduisent. La spécification de la contrainte impose d'indiquer le nom des nouveaux attributs même s'il n'y a pas de changement par rapport à ceux de la table originale.*

```
CREATE  TABLE  PIL_AVION
(NUMPIL, NUMAV, CONSTRAINT CP_PIL_AVION PRIMARY KEY(NUMPIL, NUMAV))
AS SELECT DISTINCT NUMPIL, NUMAV FROM VOL
```
□

2.2.3 - Notion de schéma de bases de données

La notion de schéma d'une base peut être complètement transparente pour l'utilisateur. En effet, lorsque l'utilisateur se connecte sous ORACLE, il se connecte en fait à une base de données spécifiée par défaut. Néanmoins, un schéma peut être défini par tout utilisateur en ayant le droit. Une telle opération consiste à nommer le nouveau schéma, à définir les relations et les vues (Cf. paragraphe 4.3) appartenant à ce schéma puis à spécifier les privilèges des autres

utilisateurs (Cf. paragraphe 4.2) sur ces relations ou ces vues. Ainsi, lorsqu'une relation est utilisée, son identificateur général est : `[<nom_schéma>.]<nom_relation>`, i.e. si elle appartient à un schéma différent du schéma par défaut, son nom doit être préfixé par le nom de son schéma d'appartenance [1]. Dans un souci de simplification, nous omettons systématiquement ce paramètre dans la suite du document.

2.3 - Modification de la structure d'une base

Contrairement aux opérations de mise à jour des données, les modifications structurelles ne sont pas courantes. En effet, l'implémentation et la mise en œuvre d'une base de données sont le résultat d'un processus long, mené en équipe incluant différentes étapes de contrôle et validation. Une modification structurelle doit résulter d'une évolution du réel, de l'expression de nouveaux besoins, de changements dans les objectifs de l'application… Tant que la base n'est pas opérationnelle (elle est vide ou contient un jeu d'essai facile à re-générer), les opérations du LDD sont aisées à mettre en œuvre. Mais évidemment dès que la base entre dans la phase d'exploitation, les modifications structurelles peuvent être soumises à de nombreuses restrictions (permettant par exemple de garantir la cohérence des données) et délicates à mettre en œuvre. Bien sûr de telles modifications ne sont pas à la portée de n'importe quel utilisateur ORACLE. Elles sont du ressort de l'administrateur ou du concepteur (propriétaire) d'une base particulière et pour les effectuer il faut disposer de certains privilèges (Cf. paragraphe 4.2.2).

Tout ordre du LDD est irréversible dans le sens où il ne peut pas être " défait " ou simplement annulé. Dans le meilleur des cas, il existe un autre ordre permettant de supprimer les effets du premier, dans le pire, il y a perte de données.

Les opérations offertes en SQL*PLUS pour modifier la structure d'une base existante sont les suivantes :

- ajout d'attributs ou de contraintes dans une relation ;
- modification de la définition d'attributs dans une relation ;
- modification de l'état d'une contrainte ;
- suppression de contraintes dans une relation ;
- suppression d'attributs dans une relation ;
- destruction de relation ;
- changement du nom d'une relation et création d'un synonyme pour une relation.

Certaines modifications structurelles ne peuvent pas être directement réalisées, par exemple le changement de nom d'un attribut ou certaines modifications de types. Cependant, il est possible de les mettre en œuvre en créant une relation intermédiaire et en effectuant des copies de tuples (Cf. paragraphe 2.3.8).

2.3.1 – Ajout d'attributs et de contraintes dans une relation

La commande permettant d'ajouter des attributs et/ou des contraintes dans une relation est la suivante :

```
ALTER TABLE <nom_table>
ADD ([<nom_colonne1> <type1>] [DEFAULT <expr1>] [NOT NULL] [UNIQUE]
[,<nom_colonne2> <type2> [DEFAULT <expr2>] [NOT NULL] [UNIQUE]…]
[, <specif_contrainte> …])
```

Un ou plusieurs attributs peuvent être ajoutés dans une relation en respectant une syntaxe similaire à celle de la création classique de relation (Cf. paragraphe 2.2.1).

Exemple *: ajout d'un attribut dans une relation.*

```
ALTER TABLE PILOTE ADD(DATE_NAIS DATE)
```
□

La spécification de nouvelles contraintes se fait de la même manière que lors de la création d'une relation par : **CONSTRAINT** <nom_contrainte> <spec_contrainte> [<etat>] (Cf. paragraphe 2.2.1).

Exemple *: ajout d'un attribut et d'une contrainte de domaine sur cet attribut.*

```
ALTER TABLE AVION ADD
(CAPACITE NUMBER(3,0),
CONSTRAINT DOM_CAPACITE CHECK(CAPACITE BETWEEN 30 AND 600))
```
□

Remarques : (a) l'ajout d'un attribut échoue s'il existe déjà dans la relation un attribut de même nom, de la même manière le nom d'une contrainte doit être unique dans une relation.
(b) L'ajout d'une contrainte d'intégrité peut échouer si la relation concernée contient déjà des tuples ne respectant pas la contrainte en question. Il est, par exemple, impossible d'utiliser **UNIQUE** ou de spécifier comme clef primaire un attribut ayant des valeurs dupliquées. De la même manière l'ajout d'une contrainte **NOT NULL** pour un attribut ayant des valeurs manquantes échoue.
(c) Il est toujours impossible, en une seule opération, d'ajouter un attribut et sa contrainte rétroactive de clef primaire, d'unicité ou d'obligation de valeurs dans une relation contenant déjà des tuples car le nouvel attribut viole systématiquement de telles contraintes. Il faut alors procéder par étapes : créer le nouvel attribut, le renseigner avec des valeurs valides, ajouter la contrainte. Une autre solution est d'ajouter la contrainte mais dans un état **NOVALIDATE**, d'attribuer les valeurs du nouvel attribut puis de réactiver la contrainte.
En effet, si la relation contient déjà des tuples, l'état d'une contrainte peut être **ENABLE**, **DISABLE**, **VALIDATE**, **NOVALIDATE** ou l'une des combinaisons suivantes :

- **ENABLE VALIDATE** : à sa création, la contrainte est active (**ENABLE**) donc toute nouvelle mise à jour doit la respecter. De plus, le mot clef **VALIDATE** indique que toutes les données déjà présentes dans la base avant l'ajout de la contrainte doivent elles aussi la satisfaire ;

- **ENABLE NOVALIDATE** : à sa création, la contrainte est active (**ENABLE**) donc toute nouvelle mise à jour doit la respecter. Cependant, toutes les données déjà présentes dans la base avant la modification ne sont pas soumises à cette contrainte (**NOVALIDATE**) ;

- **DISABLE VALIDATE** : la contrainte est désactivée (**DISABLE**) donc toute nouvelle mise à jour peut ne pas la respecter. Mais toutes les données déjà présentes dans la base avant l'ajout de la contrainte doivent la satisfaire (**VALIDATE**) ;

- **DISABLE NOVALIDATE** : la contrainte n'est pas forcément respectée, ni par les nouvelles mises à jour (**DISABLE**) ni par les données déjà stockées (**NOVALIDATE**) ;

Exemple *: la relation AVION contient des tuples qui doivent être conservés. Cependant la compagnie décide de ne plus acquérir d'appareils dont la capacité est inférieure à 100 places. La contrainte de l'exemple précédent étant préalablement détruite, une nouvelle contrainte de domaine est spécifiée comme suit. Les avions déjà stockés peuvent ne pas la respecter mais*

toute nouvelle insertion doit la satisfaire.

```
ALTER TABLE AVION ADD
(CONSTRAINT DOM_CAPACITE CHECK(CAPACITE >= 100) ENABLE NOVALIDATE)     □
```

2.3.2 – Modification de la définition d'un attribut

Il est possible de modifier le type d'un attribut existant dans une relation, de lui ajouter une contrainte d'obligation de valeurs (**NOT NULL**) et/ou de spécifier une valeur par défaut. La commande à utiliser est la suivante :

```
ALTER TABLE <nom_table>
MODIFY [()<nom_colonne1> [<nouveau_type1>] [DEFAULT <expr1>] [NOT NULL]
  [,<nom_colonne2>[<nouveau_type2>] [DEFAULT <expr2>] [NOT NULL]…] [)]
```

Si les modifications indiquées dans la clause **MODIFY** portent sur un unique attribut, les parenthèses ouvrante (juste après **MODIFY**) et fermante (à la fin de la commande) peuvent être omises. Dans tous les autres cas, elles sont obligatoires.

Bien sûr si l'ordre formulé est : **ALTER TABLE** <nom_table> **MODIFY** <nom_colonne1>, aucune modification structurelle n'est réalisée.
Si une valeur par défaut est ajoutée pour un attribut, l'expression associée doit respecter le type de l'attribut (ancien ou nouveau si celui ci est modifié) et les éventuelles contraintes.

Si une modification de type et/ou l'ajout d'une contrainte **NOT NULL** sont formulés et que ***la relation concernée est vide*** (aucun tuple), la commande **ALTER** est exécutée par ORACLE sans générer d'erreur. Cependant dès qu'une relation contient au moins un tuple, les modifications sont soumises à diverses restrictions fortes :

- pour un attribut ayant déjà au moins une valeur nulle, une contrainte d'obligation de valeurs ne peut être ajoutée que si elle est déclarée dans l'état **NOVALIDATE**. En effet, l'extension actuelle de la relation ne respectant pas la contrainte, son ajout dans l'état **VALIDATE** échoue systématiquement (Cf. paragraphe 2.3.1). En revanche, si l'attribut est renseigné pour tous les tuples, l'ajout de la contrainte **NOT NULL** est réalisé sans problème ;
- le type **VARCHAR2** peut être substitué à **CHAR** et réciproquement sans générer aucune erreur dans les cas suivants :
  - la taille maximale des valeurs admissibles de l'attribut n'est pas changée ou est accrue,
  - la taille maximale des valeurs de l'attribut est réduite et toutes les valeurs actuelles de l'attribut concerné sont **NULL** ;
- tout remplacement d'un type par un autre n'ayant pas le même sur-type est rejeté sauf si l'attribut considéré n'admet, au moment de la modification, que des valeurs **NULL**.

Quel que soit le contenu d'une relation, une modification d'attribut n'échoue jamais si la taille des valeurs admissibles est étendue sans changer le type.

S'il existe déjà des tuples dans la relation et que le nouveau type attribué à un attribut n'est pas un sur-type de l'ancien type, il faut procéder en utilisant une table intermédiaire et avoir recours aux fonctions de conversion de types données au paragraphe 3.1.3.2, lors de la copie des tuples de la table originale vers la table intermédiaire (Cf. paragraphe 2.3.8).

***Exemple** : extension à 15 caractères pour les valeurs de l'attribut AVNOM sans changement du type.*

```
ALTER TABLE  AVION  MODIFY (AVNOM VARCHAR2(15))
```
□

Exemple *: supposons que l'attribut NUMVOL initialement doté d'un type numérique soit modifié pour admettre des valeurs alphanumériques. La commande suivante est acceptée uniquement si la relation VOL est vide.*

```
ALTER TABLE  VOL  MODIFY (NUMVOL VARCHAR2(10))
```
□

Exemple *: dans la relation PILOTE, les attributs SALAIRE et PRIME seront renseignés à 0 si une nouvelle insertion ne leur attribue pas de valeur. De plus, les valeurs de l'attribut ADRESSE sont désormais obligatoires.*

```
ALTER TABLE  PILOTE  MODIFY (
SALAIRE DEFAUT 0, PRIME DEFAUT 0,
ADRESSE NOT NULL)
```
□

2.3.3 – Modification de l'état d'une contrainte

Il est possible de modifier l'état d'une contrainte préalablement définie sur une relation en utilisant la commande suivante.

```
ALTER TABLE <nom_table>
MODIFY CONSTRAINT <nom_contrainte> <etat_contrainte>
```

où `<etat_contrainte>` peut être : soit **ENABLE** ou **DISABLE**, soit **VALIDATE** ou **NOVALIDATE**, soit les combinaisons **ENABLE VALIDATE**, **ENABLE NOVALIDATE**, **DISABLE VALIDATE**, ou **DISABLE NOVALIDATE**, Cf. paragraphe 2.3.1.

Exemple *: l'ordre suivant permet de désactiver une contrainte d'intégrité dans la relation AVION.*

```
ALTER TABLE AVION MODIFY CONSTRAINT DOM_CAPACITE DISABLE NOVALIDATE
```
□

2.3.4 – Suppression de contrainte dans une relation

Il est possible, sous certaines réserves, de supprimer des contraintes dans une relation existante mais dans chaque commande, au plus une contrainte peut être éliminée. La commande permettant de supprimer une contrainte dans une relation peut prendre trois formes :

1) **ALTER TABLE** <nom_table> **DROP CONSTRAINT** <nom_contrainte> [**CASCADE**]
   une contrainte nommée par l'utilisateur peut être supprimée en indiquant son nom dans la clause **DROP CONSTRAINT**. Pour retrouver le nom d'une contrainte, il faut consulter les tables système concernées (Cf. paragraphe 2.7.1). S'il s'agit d'une contrainte de clef primaire, la suppression peut échouer si la clef primaire associée est référencée dans la définition d'une contrainte de clef étrangère. L'argument optionnel **CASCADE** permet, dans le cas de la destruction d'une contrainte de clef primaire, de propager la suppression à toutes les contraintes de référence liant la clef primaire concernée à ses clefs étrangères.

2) **ALTER TABLE** <nom_table> **DROP UNIQUE**(<nom_attribut>) [**CASCADE**]
   une contrainte d'unicité de valeurs peut être éliminée en indiquant le mot clef **UNIQUE** et le nom de l'attribut sur lequel elle porte. Si cet attribut est référencé par une contrainte de clef étrangère, l'argument optionnel **CASCADE** permet de propager la suppression à toutes les

contraintes de référence liant l'attribut concerné à des clefs étrangères.

3) **ALTER TABLE** <nom_table> **DROP PRIMARY KEY** [**CASCADE**]
cette dernière formulation ne concerne que la suppression de la contrainte de clef primaire. Si cette clef primaire est utilisée pour spécifier une ou des contraintes de référence, l'argument **CASCADE** permet de propager la suppression à toutes ces contraintes de référence.

Exemple *: la contrainte de domaine sur la capacité des avions est supprimée.*

```
ALTER TABLE AVION DROP CONSTRAINT DOM_CAPACITE
```
□

Exemple *: les commandes suivantes sont équivalentes et éliminent la contrainte de clef primaire de la relation PILOTE ainsi que la contrainte de référence associée définie lors de la création de la relation VOL.*

```
ALTER TABLE PILOTE DROP CONSTRAINT CLEF_PRIM_PILOTE CASCADE

ALTER TABLE PILOTE DROP PRIMARY KEY CASCADE
```
□

2.3.5 – Suppression d'attribut dans une relation

Tous les SGBD n'offrent pas cette fonctionnalité. À partir de la version 8, ORACLE permet la destruction d'attributs[11]. Cependant, cette opération peut s'avérer longue si la relation concernée est de taille importante. Pour cette raison, ORACLE offre une alternative à la suppression d'attributs : la possibilité de les marquer comme non utilisés (ils pourront être détruits plus tard, par exemple en dehors des heures d'exploitation de la base de données). ***Attention, cet ordre n'est pas réversible : l'état d'un attribut ne peut plus être changé et ses valeurs sont désormais inaccessibles***.

Indiquer qu'un attribut n'est plus utilisé se fait en utilisant la commande donnée ci-après.

```
ALTER TABLE <nom_table> SET UNUSED COLUMN <nom_attribut>
```

Si plusieurs attributs sont concernés, la syntaxe de la commande devient :

```
ALTER TABLE <nom_table> SET UNUSED (<nom_attribut1>[, <nom_attribut2> …])
```

Bien que toujours présents dans la base, les attributs marqués non utilisés sont inaccessibles à tout utilisateur. Ils n'apparaissent plus si la structure de la relation est visualisée par **desc** (mais ils restent présents dans le dictionnaire de données Cf. paragraphe 2.7.1). Leurs valeurs ne sont plus affichées par un ordre **SELECT** * **FROM** <nom_table>. Cependant, comme ils ne sont pas effectivement détruits, ils sont toujours comptabilisés dans le degré[12] de la relation et si l'attribut marqué est de type **LONG**, il n'est pas possible d'ajouter un nouvel attribut de ce type (il y en a un seul maximum par relation). Il est possible de créer dans une relation un attribut portant le même nom qu'un attribut marqué inutilisé.

La suppression effective d'un attribut l'élimine de la relation concernée et libère de l'espace de stockage. ***Cet ordre est irréversible : que l'attribut ait des valeurs ou pas, il est définitivement éliminé de la base***. Plusieurs formulations sont possibles.

1) **ALTER TABLE** <nom_table> **DROP COLUMN** <nom_attribut> [**CASCADE CONSTRAINTS**]
La commande précédente permet d'éliminer un attribut. Si une contrainte de clef primaire

[11] Pour les versions antérieures, il faut procéder en plusieurs étapes comme pour le changement de nom d'attribut (Cf. paragraphe 2.3.8).
[12] i.e. le nombre d'attributs, limité à 1.000.

(ou d'unicité) porte sur cet attribut et qu'il est référencé par une ou plusieurs clefs étrangères, la destruction de l'attribut échoue sauf si l'argument **CASCADE CONSTRAINTS** est précisé. Il permet de propager aux clefs étrangères la suppression de contraintes référentielles.

2) **ALTER TABLE** <nom_table>
DROP (<nom_attribut1>[, <nom_attribut2> …]) [**CASCADE CONSTRAINTS**]
Cette modification de relation permet de supprimer plusieurs attributs. Si une contrainte de clef primaire (ou d'unicité) porte sur l'un des attributs détruits et qu'il est référencé par une ou plusieurs clefs étrangères, l'argument **CASCADE CONSTRAINTS** doit être spécifié pour propager aux clefs étrangères la suppression de contraintes référentielles.

3) **ALTER TABLE** <nom_table> **DROP UNUSED COLUMNS**
Avec cet ordre, tous les attributs marqués comme inutilisés sont éliminés de la relation.

***Exemple** : l'ordre suivant permet l'élimination d'attributs.*

```
ALTER TABLE PILOTE DROP (SALAIRE, PRIME)
```
□

Il n'est pas possible de détruire un attribut (ou plusieurs) et, dans le même ordre **ALTER TABLE**, de marquer comme inutilisé un autre attribut. La combinaison des clauses **ADD, MODIFY** et **DROP** est aussi interdite au sein d'un même ordre **ALTER TABLE**.

2.3.6 – Suppression de relation

La commande permettant la destruction de relation est comme les autres commandes du LDD irréversible. ***Lorsque cet ordre est exécuté, la relation est supprimée sans demande de confirmation, qu'elle contienne des données ou qu'elle soit vide***. La destruction d'une relation entraîne aussi celle des index qui lui ont été associés. Cette commande est :

```
DROP TABLE <nom_table> [CASCADE CONSTRAINTS]
```

La commande **DROP TABLE** peut échouer si la relation à supprimer est " référencée " lors de la définition d'une autre relation (i.e. si sa clef primaire est associée à une clef étrangère). Dans ce cas, il faut détruire la contrainte de référence avant de pouvoir supprimer la relation ou utiliser la clause optionnelle **CASCADE CONSTRAINTS** qui effectue automatiquement la destruction des contraintes de référence.

***Exemple** : pour supprimer la relation AVION et les contraintes de référence définies dans les autres relations du schéma et concernant sa clef primaire NUMAV, l'ordre suivant est exécuté.*

```
DROP TABLE AVION CASCADE CONSTRAINTS
```
□

2.3.7 – Création de synonyme et changement du nom d'une relation

Il est possible d'attribuer un synonyme au nom d'une relation (ou d'autres " objets " ORACLE : vue, séquence… ou même synonyme[12]) de la manière suivante :

```
CREATE [PUBLIC] SYNONYM <nom_synonyme>  FOR  <nom_objet>
```

Lorsqu'un synonyme est créé avec l'option **PUBLIC**, il est accessible à tous les utilisateurs. Sans cette option, le synonyme est privé et peut seulement être utilisé dans le schéma où il est créé (en particulier par l'utilisateur qui l'a créé). <nom_synonyme> doit respecter les règles des identificateurs ORACLE. Dans la clause **FOR**, <nom_objet> est un nom de relation, de vue, de séquence ou de synonyme.

Lorsqu'un synonyme est créé, il peut être utilisé dans tout ordre du LMD (requêtes et opérations de mise à jour des données) mais aussi dans les commandes du LDD suivantes : **COMMENT**, **GRANT**, **REVOKE** (Cf. paragraphes 2.5 et 4.2.2). Pour supprimer un synonyme, l'ordre à utiliser est :

```
DROP SYNONYM <nom_synonyme>
```

Exemple *: les ordres suivants de création de synonymes sont valides.*

```
CREATE PUBLIC SYNONYM LES_PILOTES FOR PILOTE
/
CREATE SYNONYM PIL FOR LES_PILOTES
```
□

Pour renommer une relation (ou plus généralement des " objets " ORACLE : vue, séquence, synonyme privé…[13] (Cf. paragraphes 4.3 et 2.4), l'ordre SQL*PLUS est le suivant :

```
RENAME <ancien_nom>  TO  <nouveau_nom>
```

Bien sûr, l'exécution ultérieure de tout ordre SQL*PLUS utilisant <ancien_nom> déclenchera une erreur. Il faut souligner que tout " objet " ORACLE défini à partir d'un " objet " qui a ensuite été renommé est " marqué " par le système qui lui attribue un état invalide (ce mécanisme est détaillé dans le cadre de la gestion de vues au paragraphe 4.3.4).
Les ordres **CREATE SYNONYM** et **RENAME** ne s'appliquent jamais aux attributs d'une relation. Pour modifier le nom d'un attribut, il faut utiliser la " procédure " décrite au paragraphe 2.3.8.

2.3.8 – Changement de nom d'un attribut ou modification de son type

Modifier le nom d'un attribut dans une relation existante ne peut pas se faire directement via un ordre du LDD. D'autre part, si une relation est non vide, certaines modifications de type ne sont pas autorisées (Cf. paragraphe 2.3.2). Enfin, certains SGBD ne permettent pas la suppression directe d'attributs[14]. Il est toujours possible de mettre en œuvre de tels changements structurels, mais plusieurs ordres doivent être réalisés séquentiellement. Les étapes à suivre sont les suivantes :

1) il faut d'abord procéder à la création d'une nouvelle relation ayant la même structure que la relation devant être modifiée à ceci près que :
   - soit un attribut est de type différent,
   - soit un attribut porte un nom différent,
   - soit un attribut est éliminé, si le SGBD utilisé n'offre pas la fonctionnalité de destruction d'attribut.

   Cette nouvelle relation peut être créée par copie de la relation originale (Cf. Paragraphe 2.2.2) et, si une modification de type est opérée, il faut utiliser les fonctions de conversion

[13] Mais aussi procédures, fonctions ou packages écrits en PL/SQL.
[14] C'est le cas des versions antérieures d'ORACLE.

(Cf. paragraphe 3.1.3.2) dans la spécification de la requête de définition. Il faut également définir les contraintes d'intégrité que doit satisfaire cette nouvelle relation ;

1) si la relation a été créée de manière classique, il faut réaliser l'insertion par copie, dans cette nouvelle relation, des tuples existant dans la relation à modifier (Cf. Paragraphe 3.2.2.2) ;

2) les tuples de la relation originale ayant été " récupérés ", la destruction de cette relation peut être effectuée directement ou en utilisant l'option **CASCADE CONSTRAINTS** pour supprimer les contraintes de référence associées à la clef primaire de la relation (Cf. paragraphe 2.3.6). Une alternative à la destruction de telles contraintes est de les inhiber le temps des modifications structurelles, i.e. de les mettre dans un état **NOVALIDATE** (Cf. paragraphe 2.3.3) ;

3) pour éviter tout problème ultérieur, la nouvelle relation doit être renommée en réutilisant le nom de la relation d'origine (Cf. paragraphe 2.3.7) ;

4) Enfin, si nécessaire, les contraintes d'intégrité de référence associant la clef primaire de la nouvelle relation à des clefs étrangères doivent être :
- soit recréées dans les tables concernées si elles ont été détruites au point 3) de cette procédure de modification ;
- soit remises en œuvre en leur attribuant un état **VALIDATE** (Cf. paragraphe 2.3.3).

***Exemple** : certains attributs de la relation AVION doivent être renommés. La séquence d'ordres suivants permet de réaliser cette modification sans perte de données ni de contraintes.*

```
/* Une nouvelle relation est créée par copie, les attributs concernés
sont renommés et la clef primaire est spécifiée */
CREATE TABLE COPIE_AVION
(NUMAV, NOMAV, CAP, LOC, CONSTRAINT CP_AVION PRIMARY KEY(NUMAV))
AS SELECT NUMAV, NOMAV, CAPACITE, LOCALISATION FROM AVION
/
DROP TABLE AVION CASCADE CONSTRAINTS -- Destruction de la relation AVION
                                     -- avec propagation aux contraintes
/
RENAME COPIE_AVION TO AVION          -- La copie créée est renommée
/

ALTER TABLE VOL ADD
(CONSTRAINT CE_VOL_AVION FOREIGN KEY(NUMAV) REFERENCES AVION(NUMAV))
/* La contrainte de référence liant la relation VOL et la nouvelle
relation AVION est recréée */                                          □
```

2.4 – Création de séquences

Il est possible, dans ORACLE et à condition d'avoir le privilège associé (Cf. paragraphe 4.2.2) de créer des objets qui sont des séquences permettant aux divers utilisateurs de générer des valeurs entières éventuellement uniques (positives ou négatives). De telles séquences sont particulièrement utiles pour générer automatiquement des valeurs de clefs primaires. L'ordre de création est le suivant.

```
CREATE SEQUENCE <nom_sequence>
[START WITH  <valeur_initiale>]
[INCREMENT BY  <valeur_increment>]
```

```
[MAXVALUE  <valeur_maximale> | NOMAXVALUE]
[MINVALUE  <valeur_minimale> | NOMINVALUE]
[CYCLE   | NOCYCLE]
```

Si aucune des clauses optionnelles n'est spécifiée, la séquence débutera par la valeur 1 et sera incrémentée avec un pas de 1 sans limite supérieure. La clause **START WITH** permet de donner la première valeur de la séquence et **INCREMENT BY** permet à l'utilisateur d'indiquer un incrément différent de 1. S'il est négatif, une séquence décroissante de valeurs est obtenue. Pour limiter les valeurs générées, les clauses **MAXVALUE** ou **MINVALUE** peuvent être utilisées pour les séquences respectivement croissantes ou décroissantes. Lorsque cette borne est atteinte, la génération d'une nouvelle valeur produit une erreur. Pour obtenir une séquence sans limitation, **NOMAXVALUE** ou **NOMINVALUE** peuvent être utilisées mais il suffit aussi simplement de ne pas indiquer de clause de borne.
Si une borne est spécifiée pour la séquence, il est possible de re-générer de nouvelles valeurs en recommençant à partir de la valeur initiale. Pour cela la clause **CYCLE** doit être mentionnée ainsi que **MAXVALUE** et **MINVALUE** (par défaut la valeur initiale est 1).

***Exemple** : la séquence suivante produit la série non limitée d'entiers : 2, 12, 22, 32…*

```
CREATE SEQUENCE ID_AVION START WITH 2 INCREMENT BY  10
```

La séquence cyclique suivante démarre à 100, puis produit des entiers incrémentés de 10 à chaque nouvel appel et, lorsque la valeur maximale 500 est atteinte, redémarre à la valeur 100.

```
CREATE SEQUENCE  SEQ03
START WITH  100  INCREMENT BY  10
MAXVALUE  500    MINVALUE  100
CYCLE
```

□

Une séquence peut être supprimée par :

```
DROP SEQUENCE <nom_sequence>
```

L'ordre **ALTER SEQUENCE** permet de modifier les différents paramètres d'une séquence : changer l'incrément, les valeurs minimale et maximale et spécifier un cycle ou pas.

```
ALTER SEQUENCE <nom_sequence>
[INCREMENT BY  <valeur_increment>]
[MAXVALUE  <valeur_maximale> | NOMAXVALUE]
[MINVALUE  <valeur_minimale> | NOMINVALUE]
[CYCLE   | NOCYCLE]
```

Pour modifier la valeur initiale d'une séquence, il faut la détruire et la recréer. Lorsqu'une ***nouvelle valeur maximale est indiquée, il faut qu'elle soit supérieure à la valeur courante de la séquence***, sinon une erreur se produit.

La génération de valeurs pour les séquences se fait indépendamment de la validation ou l'annulation des transactions (Cf. paragraphe 4.1).
Elle est également indépendante des relations de la base de données, si bien qu'une même séquence peut être employée pour une ou plusieurs tables.
Lorsque plusieurs utilisateurs génèrent des valeurs de séquence, chaque valeur transmise à un utilisateur est unique (sauf si un cycle est spécifié). Les valeurs générées par un utilisateur peuvent très bien ne pas être consécutives simplement parce qu'entre temps un autre utilisateur a effectué une génération. Il se peut aussi que tous les entiers générés par tous les utilisateurs ne soient pas consécutifs car certaines transactions les utilisant ont finalement échoué.

Après la création d'une séquence, il est possible de manipuler sa valeur courante et la prochaine valeur générée en utilisant les pseudo-colonnes[15] **CURRVAL** et **NEXTVAL**, en les préfixant par le nom de la séquence. **CURRVAL** et **NEXTVAL** peuvent être uniquement utilisés dans la partie LMD de SQL*PLUS. Leur utilisation est détaillée et illustrée par des exemples aux paragraphes 3.1.11 (utilisation lors de la formulation de requêtes) et 3.2.4 (utilisation lors des mises à jour).

2.5 – Commentaires

Les commentaires, tels qu'ils ont été vus jusqu'à présent (introduits par `--` ou encadrés par `/*` et `*/` Cf. paragraphe 1.5), peuvent être insérés dans le code de tout ordre SQL et ils sont simplement sauvegardés dans le fichier .sql contenant ce code. Outre cette possibilité, ORACLE permet d'intégrer des commentaires dans la base de données elle-même (plus précisément dans le dictionnaire de données Cf. paragraphe 2.7.1). Ces commentaires ne peuvent être associés qu'à des relations (ou des vues) et des attributs. Suivant le cas, l'une des commandes suivantes est utilisée.

```
COMMENT ON TABLE <nom_table> IS '<texte_commentaire>'

COMMENT ON COLUMN <nom_table>.<nom_attribut> IS '<texte_commentaire>'
```

Pour une relation ou un attribut, il y a au plus un commentaire. `<texte_commentaire>` peut comporter jusqu'à 4.000 caractères et s'il contient une apostrophe elle doit être doublée. Pour supprimer un commentaire, il faut utiliser :

```
COMMENT ON TABLE <nom_table> IS ''

COMMENT ON COLUMN <nom_table>.<nom_attribut> IS ''
```

L'avantage de ce type de commentaires est triple :
- faisant partie de la base, ils sont stockés de manière pérenne et centralisée par le SGBD,
- leur accès en écriture est sécurisé : seuls les utilisateurs autorisés peuvent les modifier,
- ils sont consultables par tous les utilisateurs concernés qui n'ont évidemment pas accès aux scripts SQL de tel ou tel développeur.

Ainsi donc les deux types de commentaires proposés sont nécessaires et complémentaires pour la documentation d'une application base de données. Les premiers destinés aux développeurs expliquent le code source SQL. Les seconds apportent à tous les utilisateurs des informations structurelles sur la base.

***Exemple** : les commentaires suivants sont intégrés dans la base.*

```
COMMENT TABLE AVION IS 'Tous les avions de la compagnie'
/
COMMENT COLUMN AVION.CAPACITE IS
'Pour tous les nouveaux appareils insérés, la capacité doit être
supérieure à 100'
```
□

2.6 - Index sur les relations

[15] Les pseudo-colonnes sont des objets ORACLE se comportant comme des attributs mais qui ne sont pas effectivement des attributs de relations. Il est possible de les interroger mais jamais de les mettre à jour directement.

Les index sont des structures de données, physiquement et logiquement indépendantes des données stockées dans la base, qui permettent un accès direct aux tuples des relations. Ainsi, l'utilisation d'index doit minimiser le temps de lecture des données et améliorer le temps d'exécution des requêtes. En effet, pour retrouver certains tuples dans une relation, le SGBD peut exploiter la présence d'un index et éviter de balayer la totalité de son extension. Les index sont définis sur un ou plusieurs attributs qui ne peuvent pas être de type **LONG**.

***Exemple** : supposons qu'il soit très souvent nécessaire de retrouver les vols assurés par tel ou tel pilote. Un index sur l'attribut NUMPIL de la relation VOL, illustré par la figure 1, permet d'optimiser ces recherches.*

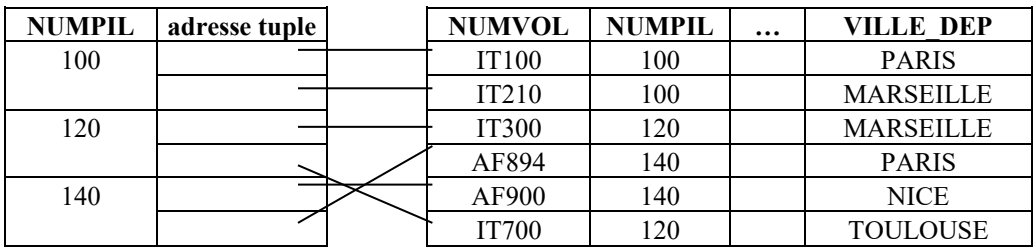


Figure 1 : index sur la relation VOL selon l'attribut NUMPIL □

Lorsque des insertions, modifications ou suppressions de tuples sont effectuées sur une relation indexée, le SGBD répercute automatiquement ces opérations de mise à jour sur les index concernés. Il n'y a pas de limite au nombre d'index pouvant être créés sur une relation, mais s'ils sont nombreux, les opérations de mise à jour qui nécessitent la réorganisation des index sont plus coûteuses. De plus les index nécessitent de l'espace de stockage supplémentaire.

Lorsqu'un index est créé sur un attribut (ou plusieurs) n'admettant que des valeurs uniques, on dit que cet index est ***primaire***. Lorsqu'il porte sur plusieurs attributs, un index est qualifié de ***composite***.

2.6.1 - Création d'index

La création d'index peut être réalisée pour un ou plusieurs attributs fréquemment utilisés dans des opérations de sélection ou de jointure. Par défaut, la structure utilisée par ORACLE pour les index est un arbre équilibré ou B-arbre (B pour Balanced). La commande de création d'index propose diverses options, certaines destinées à l'administrateur de la base. Une version simplifiée est donnée ci-après (pour connaître toutes les options de cet ordre, consultez les références [1] et [11]).

```
CREATE [UNIQUE | BITMAP] INDEX  <nom_index>
ON <nom_table> (<nom_colonne1>[, <nom_colonne2> …])
```

où :

UNIQUE ce mot-clef indique que l'index créé est un index primaire, i.e. l'attribut (ou la combinaison d'attributs) indexé doit avoir des valeurs uniques. Si ce n'est pas le cas, la création d'index échoue. De plus, l'insertion de valeurs dupliquées pour les attributs indexés devient impossible.
Remarque : lors de la spécification des contraintes **PRIMARY KEY** ou **UNIQUE** pour un ou plusieurs attributs ORACLE génère automatiquement un index primaire pour le(s) attribut(s) concerné(s). Il ne faut pourtant pas faire de confusion entre la création de contraintes qui relève du niveau logique et celle d'index qui relève du niveau physique.

BITMAP cette option permet d'utiliser une structure de type bitmap (à la place d'un arbre équilibré). Cette option est impossible pour les index primaires et doit être réservée aux attributs dont la cardinalité est faible et aux applications où il y a peu de transactions concurrentes (Cf. paragraphe 4.1).

<nom_index> ce nom doit respecter les conventions des identificateurs SQL*PLUS et être unique dans une relation.

<nom_table> est le nom de la relation à indexer.

(<nom_colonne1>[, <nom_colonne2> …]) indique la liste des attributs indexés. Un attribut de la relation doit au moins être mentionné. Il n'est pas possible de créer plus d'un index sur un attribut. En revanche, si une combinaison d'attributs est indexée, plusieurs index peuvent être créés sur la même liste d'attributs à condition que ces attributs soient mentionnés dans des ordres différents. Les deux ordres suivants permettent la création de deux index composites sur la même relation et les mêmes attributs mais organisés différemment.

```
CREATE INDEX  <nom_index1>
ON <nom_table> (<nom_colonne1>, <nom_colonne2>)

CREATE INDEX  <nom_index2>
ON <nom_table> (<nom_colonne2>, <nom_colonne1>)
```

Pour un index de type arbre équilibré, 32 attributs peuvent être mentionnés lors de la création d'index. Le nombre maximal d'attributs pour un index de type bitmap est de 30.

***Exemple** : les recherches d'information sur un pilote à partir de son nom sont fréquentes, de même que celles des vols en fonction des villes de départ et d'arrivée. Les index créés permettent d'optimiser ces recherches.*

```
CREATE INDEX  NOM_PILOTE  ON PILOTE (NOMPIL)
/
CREATE INDEX  VILLES_VOL  ON VOL (VILLE_DEP, VILLE_ARR)
```

□

2.6.2 - Utilisation d'index

Une fois qu'ils ont été créés, les index sont automatiquement utilisés par le système et de manière transparente pour l'utilisateur. Plus précisément, ORACLE tire profit des index pour évaluer des conditions de sélection simples et pour effectuer des jointures. Un attribut fréquemment projeté comme résultat de requêtes mais sur lequel les sélections sont rares est un mauvais candidat pour créer un index.
Pour les requêtes effectuant l'extraction d'un petit ensemble de tuples de la base de données, les index permettent une exécution beaucoup plus efficace, ce qui n'est pas le cas pour des requêtes restituant un très grand nombre de tuples.

Pour définir des index permettant d'optimiser au mieux une application, le développeur et l'administrateur doivent faire une analyse précise de son utilisation : type de requêtes formulées, fréquence de ces requêtes, taille des résultats mais aussi nature et fréquence des

opérations de mise à jour.

Les cas dans lesquels la définition d'index est particulièrement profitable sont les suivants :

- les attributs fréquemment utilisés dans des conditions de sélection simples doivent être indexés. Les index ne sont pas utilisés par ORACLE pour certaines conditions : celles incluant l'opérateur de comparaison `!=` ou `<>` , l'opérateur de concaténation de chaînes `||` et les prédicats **`IS NULL`**, **`NOT IN`** `(<liste_valeurs>)`, **`LIKE`** `'<chaine_generique>'` (Cf. paragraphe 3.1.2) ou celles dans lesquelles les attributs indexés participent à une expression de calcul horizontal ou vertical (Cf. paragraphes 3.1.3 et 3.1.4). Ainsi les conditions suivantes inhibent les index [12] : `<nom_attribut> + 0 > <seuil>` ou `<nom_fonction>(<nom_attribut>) > <seuil>` ;
- les attributs fréquemment utilisés pour effectuer des jointures doivent être indexés. Typiquement les attributs clefs étrangères sont de bons candidats pour l'indexation lorsque des requêtes multi-relations sont fréquentes ;
- lorsque plusieurs attributs d'une relation font l'objet de conditions de sélection dans une même requête, un index composite créé sur cet ensemble d'attributs s'avère plus efficace que plusieurs index sur chacun des attributs. De plus si tous les attributs concernés par une requête font partie d'un index composite, il n'y a pas d'accès à la relation elle-même, ORACLE calcule le résultat simplement à partir de l'index ;
- lorsque plusieurs requêtes utilisent des attributs communs pour les sélections ou les jointures, elles peuvent tirer profit d'un même index composite. En effet, lorsqu'un index est créé selon une séquence d'attributs `A1, A2, A3 … An`, il peut être utilisé par toute requête incluant des conditions sur la totalité des attributs de cette séquence ou sur les attributs participant à tout préfixe de cette séquence (e.g. $A_1, A_2, A_3 \ldots A_{n-1}$; $A_1, A_2, A_3 \ldots A_{n-2}$; A_1, A_2, A_3 ; A_1, A_2 ; A_1). Toute requête incluant simplement une condition sur A_i (où $i \neq 1$) ne peut pas tirer profit de l'index composite, d'où ***l'importance de l'ordre de spécification des attributs indexés*** ;
- les index composites portant sur des attributs dont les combinaisons de valeurs sont peu dupliquées sont très efficaces.

Dans les diverses situations examinées, il faut également tenir compte :

- de la cardinalité des attributs indexés. S'ils ont très peu de valeurs différentes, un index de type arbre équilibré est peu performant et une structure de type bitmap doit être préférée ;
- de la fréquence des mises à jour. Pour des attributs très volatiles, i.e. très souvent mis à jour, la création d'index doit être soigneusement étudiée car elle pénalise l'exécution de ces opérations.

En conclusion, optimiser l'accès aux données en utilisant des index consiste à trouver le meilleur compromis entre efficacité des requêtes, coût d'exécution des mises à jour et espace de stockage nécessaire.

2.6.3 - Gestion d'index

Il n'est pas possible de modifier les attributs utilisés dans un index (il faut le détruire et le recréer). Un index peut être renommé en utilisant la commande suivante.

```
ALTER INDEX <nom_index> RENAME TO <nouveau_nom>
```

La suppression d'un index se fait comme indiqué dans l'ordre suivant. L'espace occupé par l'index est alors libéré.

```
DROP INDEX <nom_index>
```

2.7 - Consultation de la structure d'une base

Pour visualiser la définition de la structure d'une relation, la commande ORACLE à utiliser est la suivante :

```
desc[ribe] <nom_table>
```

Elle rend la liste des différents attributs de la relation avec leur type et les éventuelles contraintes sur l'interdiction de valeurs nulles.

Outre les attributs des diverses relations, la structure d'une base de données englobe de nombreux composants comme, par exemple, les contraintes d'intégrité. Les données décrivant ces composants sont appelées " méta-données " car ce sont des " données sur les données ". En effet elles apportent une information structurelle sur les données effectivement stockées dans la base. Par exemple, une contrainte d'intégrité de domaine indique les valeurs admissibles pour un attribut.

Les SGBD basés sur le modèle relationnel préservent l'ensemble des méta-données, encore appelé ***dictionnaire*** de la base, en utilisant les mêmes structures de stockage que pour la base elle-même, i.e. des relations. Il en résulte :

- une uniformité de la vision logique des données et des méta-données (les mêmes concepts sont utilisés, ceux de relations, attributs, clefs primaires et étrangères, contraintes…),
- et ***la possibilité de les manipuler à travers un seul et même langage***. Que l'utilisateur interroge les données d'une base, que le développeur consulte sa structure ou que le système mette à jour les méta-données, il s'agit toujours de formuler des ordres SQL.

2.7.1 - Le dictionnaire de données

Les relations contenant des méta-données sont appelées ***tables système*** car elles sont automatiquement maintenues par le SGBD. ***L'utilisateur peut les interroger mais en aucun cas les mettre à jour directement***. Appartenant au système (qui en est seul propriétaire), elles ne sont modifiables que par le système. Aucun utilisateur (même l'administrateur) n'a le droit de modifier leurs données. En revanche, toutes les commandes des utilisateurs portant sur la structure de la base sont automatiquement répercutées sur les tables système par le SGBD.

Exemple *: toutes les relations constituant la base (i.e. stockant les données mais aussi les méta-données) sont décrites dans une relation que nous appelons la " Table des Tables ". Plus précisément, chaque tuple de cette table apporte des informations sur une table de données ou une table système : son nom, son créateur…*
La figure 2 illustre quelques tuples de l'extension de la Table des Tables.

NOM_TABLE	PROPRIETAIRE	…	…
TABLE DES TABLES	SYSTEM	…	
…	…	…	
PILOTE	USER1	…	
AVION	USER1		
VOL	USER1		

Figure 2 : exemple d'extension de la Table des Tables

Si l'utilisateur effectue la suppression d'une relation (e.g. **DROP TABLE** VOL*), le tuple correspondant est automatiquement détruit dans la table système. Si une nouvelle relation est créée un nouveau tuple la décrivant est inséré, par le SGBD, dans la Table des Tables.* □

Chaque fois qu'un utilisateur exécute un ordre valide du LDD, le système met à jour une ou plusieurs tables système. Selon l'ordre donné par l'utilisateur, le SGBD peut insérer, supprimer ou modifier des tuples dans les tables système. ***Lors des insertions ou modifications dans les tables système, ORACLE utilise systématiquement les majuscules pour toutes les valeurs de type alphanumérique***. Alors qu'une chaîne de caractères identificateur d'un objet ORACLE peut être donnée indifféremment en minuscules ou majuscules lorsqu'un ordre SQL est formulé sur une base de données, elle doit impérativement être donnée en majuscules et évidemment apparaître entre deux apostrophes quand elle représente une valeur stockée dans le dictionnaire de données. Si ce n'est pas le cas, la requête formulée est exécutée mais rend un résultat vide (la condition de sélection n'étant jamais satisfaite).

Le nom des tables système dépend du SGBD utilisé. Dans ORACLE, le dictionnaire est constitué de différentes relations système et d'un ensemble de très nombreuses vues[16] accessibles à l'utilisateur et parfois pourvues de synonymes. Pour les différentes méta-données stockées, il existe généralement un ensemble de trois vues portant le même nom mais avec un préfixe différent.

Les vues préfixées par :

- ALL_ décrivent tous les composants d'une base auquel un utilisateur a accès ;
- USER_ informent l'utilisateur sur les composants de son environnement privé, i.e. les composants dont il est propriétaire. Ces vues proposent un sous-ensemble pertinent pour l'utilisateur des informations données par les vues de préfixe ALL_ ;
- DBA_ sont réservées à l'administrateur de la base de données (DBA) qui peut ainsi consulter les méta-données de tous les utilisateurs.

Le dictionnaire de données est le référentiel unique qui décrit tous les objets logiques et physiques d'une base ORACLE [3]. Le tableau 2 décrit les principales tables système d'ORACLE. Pour chacune de ces tables, une liste non exhaustive d'attributs est indiquée (résultat de la commande **desc** pour la relation concernée). Il est intéressant de remarquer les attributs ne pouvant accepter de valeurs nulles : ils font partie de la clef primaire de la table système considérée. Ainsi, s'expliquent les conditions d'unicité portant sur les noms de relations, d'attributs, de contraintes… Par exemple, une relation est identifiée par son nom et l'utilisateur qui l'a créée. Un attribut est identifié par son nom et l'identifiant de sa relation d'où le fait que le nom d'un attribut doit être unique dans une relation mais peut être utilisé dans des relations différentes…
Dans les tables système répertoriées dans le tableau 2, un attribut OWNER indique le créateur de " l'objet ". Il prend comme valeur le loggin ORACLE de l'utilisateur qui en est propriétaire. Par exemple, toutes les contraintes que vous avez définies peuvent être retrouvées dans ALL_CONSTRAINTS en formulant la sélection suivante : `OWNER = 'OPS$<loggin_unix>'` ou `OWNER LIKE '%<loggin_unix>'` où `<loggin_unix>` est votre nom d'utilisateur Unix donné en majuscules. Cet attribut prend la valeur SYSTEM pour tous les tuples générés automatiquement et décrivant les tables système. À chacune des tables précédentes correspond une vue préfixée par USER_. Les vues `ALL_<nom_vue>` et `USER_<nom_vue>` ont

[16] Une vue est une table virtuelle (non physiquement stockée) qui s'interroge comme une relation (Cf. Paragraphe 4.3).

exactement la même structure à ceci près que l'attribut OWNER n'apparaît pas dans `USER_<nom_vue>` puisque évidemment il s'agit de l'utilisateur courant. Le tableau 3 répertorie les principales vues de l'utilisateur.

Nom des tables système	Description	Quelques attributs
ALL_COL_COMMENTS	répertorie les commentaires concernant les attributs.	OWNER NOT NULL VARCHAR2(30) TABLE_NAME NOT NULL VARCHAR2(30) COLUMN_NAME NOT NULL VARCHAR2(30) COMMENTS VARCHAR2(4000)
ALL_CONS_COLUMNS	donne des informations supplémentaires sur les contraintes d'intégrité, en particulier le nom de l'attribut sur lequel elles portent.	OWNER NOT NULL VARCHAR2(30) CONSTRAINT_NAME NOT NULL VARCHAR2(30) TABLE_NAME NOT NULL VARCHAR2(30) COLUMN_NAME VARCHAR2(4000) POSITION NUMBER …
ALL_CONSTRAINTS	stocke toutes les contraintes d'intégrité spécifiées sur les relations, en particulier leur nom, la relation sur laquelle elles s'appliquent. Les valeurs de l'attribut CONSTRAINT_TYPE sont : P (pour **PRIMARY KEY**), C (pour **CHECK**), R (pour **REFERENCES**) pour les contraintes sur les relations et pour les vues : O (pour **READ ONLY**) et V (pour **WITH CHECK OPTION**).	OWNER NOT NULL VARCHAR2(30) CONSTRAINT_NAME NOT NULL VARCHAR2(30) CONSTRAINT_TYPE VARCHAR2(1) TABLE_NAME NOT NULL VARCHAR2(30) SEARCH_CONDITION LONG R_OWNER VARCHAR2(30) R_CONSTRAINT_NAME VARCHAR2(30)
ALL_INDEXES	décrit tous les index définis dans la base, y compris sur les tables système.	OWNER NOT NULL VARCHAR2(30) INDEX_NAME NOT NULL VARCHAR2(30) INDEX_TYPE VARCHAR2(27) TABLE_OWNER NOT NULL VARCHAR2(30) TABLE_NAME NOT NULL VARCHAR2(30) TABLE_TYPE CHAR(5) UNIQUENESS VARCHAR2(9) COMPRESSION VARCHAR2(8)
ALL_OBJECTS	décrit tous les objets existant dans la base, y compris tous les objets système. L'attribut OBJECT_TYPE peut prendre les valeurs : TABLE, SYNONYME, VIEW, SEQUENCE…	OWNER NOT NULL VARCHAR2(30) OBJECT_NAME NOT NULL VARCHAR2(30) SUBOBJECT_NAME VARCHAR2(30) OBJECT_ID NOT NULL NUMBER DATA_OBJECT_ID NUMBER OBJECT_TYPE VARCHAR2(18) CREATED NOT NULL DATE …
ALL_SEQUENCES	décrit toutes les séquences définies dans la base.	SEQUENCE_OWNER NOT NULL VARCHAR2(30) SEQUENCE_NAME NOT NULL VARCHAR2(30) MIN_VALUE NUMBER MAX_VALUE NUMBER INCREMENT NOT NULL NUMBER CYCLE_FLAG CHAR(1) …
ALL_SYNONYMS	décrit tous les synonymes créés dans la base, y compris sur les tables système.	OWNER NOT NULL VARCHAR2(30) SYNONYM_NAME NOT NULL VARCHAR2(30) TABLE_OWNER NOT NULL VARCHAR2(30) TABLE_NAME NOT NULL VARCHAR2(30) …
ALL_TABLES	Répertorie des données sur toutes les relations (y compris les tables système). De nombreux attributs concernent des paramètres physiques.	OWNER NOT NULL VARCHAR2(30) TABLE_NAME NOT NULL VARCHAR2(30) TABLESPACE_NAME VARCHAR2(30) CLUSTER_NAME VARCHAR2(30) IOT_NAME VARCHAR2(30) PCT_FREE NUMBER PCT_USED NUMBER …
ALL_TAB_COMMENTS	répertorie les commentaires sur les relations.	OWNER NOT NULL VARCHAR2(30) TABLE_NAME NOT NULL VARCHAR2(30) COMMENTS VARCHAR2(4000)…

Tableau 2 : quelques tables système d'ORACLE

ALL_TAB_COLUMNS	stocke tous les attributs de toutes les relations existant dans	OWNER NOT NULL VARCHAR2(30) TABLE_NAME NOT NULL VARCHAR2(30) COLUMN_NAME NOT NULL VARCHAR2(30)

	la base, y compris ceux des tables système.	DATA_TYPE VARCHAR2(106)[17] DATA_TYPE_MOD VARCHAR2(3) DATA_TYPE_OWNER VARCHAR2(30) DATA_LENGTH NOT NULL NUMBER DATA_PRECISION NUMBER NULLABLE VARCHAR2(1)
ALL_UNUSED_COL_TABS	répertorie le nombre d'attributs marqués inutilisés par table.	OWNER NOT NULL VARCHAR2(30) TABLE_NAME NOT NULL VARCHAR2(30) COUNT NUMBER
ALL_USERS	décrit tous les utilisateurs identifiés.	USERNAME NOT NULL VARCHAR2(30) USER_ID NOT NULL NUMBER CREATED NOT NULL DATE
ALL_VIEWS	décrit toutes les vues définies dans la base. L'attribut TEXT correspond à la requête de définition de la vue.	OWNER NOT NULL VARCHAR2(30) VIEW_NAME NOT NULL VARCHAR2(30) TEXT_LENGTH NUMBER TEXT LONG TYPE_TEXT_LENGTH NUMBER TYPE_TEXT VARCHAR2(4000) …

Tableau 2 (suite) : quelques tables système d'ORACLE

Nom des vues utilisateur	Description
USER_COL_COMMENTS	répertorie les commentaires sur les attributs des relations de l'utilisateur.
USER_CONS_COLUMNS	donne des informations supplémentaires sur les contraintes d'intégrité, en particulier le nom de l'attribut sur lequel elles portent.
USER_CONSTRAINTS	stocke toutes les contraintes d'intégrité des relations de l'utilisateur.
USER_INDEXES	décrit tous les index de l'utilisateur
USER_OBJECTS	décrit tous les objets de l'utilisateur.
USER_SEQUENCES	décrit toutes les séquences de l'utilisateur
USER_SYNONYMS	décrit tous les synonymes de l'utilisateur
USER_TABLES	décrit toutes les relations de l'utilisateur.
USER_TAB_COLUMNS	stocke tous les attributs des relations de l'utilisateur.
USER_TAB_COMMENTS	donne les commentaires sur les relations de l'utilisateur.
ALL_UNUSED_COL_TABS	répertorie le nombre d'attributs marqués inutilisés.
USER_USERS	décrit l'utilisateur courant.
USER_VIEWS	décrit les vues de l'utilisateur.

Tableau 3 : principales vues utilisateur d'ORACLE

Enfin, Oracle propose quelques synonymes (Cf. paragraphe 2.3.7) pour les vues préfixées par USER_ les plus fréquemment utilisées. Leur nom abrégé est facile d'utilisation. Ces synonymes sont répertoriés dans le tableau 4.

COLS	synonyme de USER_COLUMNS. Il est encore possible d'utiliser l'ancien synonyme COL des versions antérieures d'ORACLE mais ceci est déconseillé (nombre d'attributs très restreint et maintenance dans les versions ultérieures d'ORACLE non garantie).
IND	synonyme de USER_INDEXES.
OBJ	synonyme de USER_OBJECTS.
SEQ	synonyme de USER_SEQUENCES.
SYN	synonyme de USER_SYNONYMS.
TABS	synonyme de USER_TABLES. Il est encore possible d'utiliser l'ancien synonyme TAB mais ceci est déconseillé (nombre d'attributs très restreint et maintenance dans les versions ultérieures d'ORACLE non garantie).

Tableau 4 : les synonymes de vues utilisateur d'ORACLE

Toutes les vues système se consultent exactement de la même manière que les relations de données, i.e. en formulant des requêtes SQL (Cf. chapitre 3). Leur structure peut être obtenue

[17] Les valeurs de cet attribut sont : VARCHAR2, CHAR, NUMBER, DATE… (Cf. Tableau 1).

par **desc** <nom_table_systeme>.

***Exemple** : quels sont les attributs (nom et type) de la relation VOL dont le type est alphanumérique ?*

```
SELECT COLUMN_NAME, DATA_TYPE
FROM USER_TAB_COLUMNS
WHERE TABLE_NAME = 'VOL' AND DATA_TYPE LIKE '%CHAR%'
```
□

***Exemple** : quelles sont les contraintes définies sur la relation VOL ?*

```
SELECT *
FROM USER_CONSTRAINTS
WHERE TABLE_NAME = 'VOL'
```
□

***Exemple** : sur quels attributs sont définies les contraintes de la relation VOL ?*

```
SELECT COLUMN_NAME
FROM USER_CONS_COLUMNS
WHERE TABLE_NAME = 'VOL'
```
□

***Exemple** : Considérons l'ordre SQL suivant :*

```
ALTER TABLE AVION ADD
(CAPACITE NUMBER(3,0),
CONSTRAINT DOM_CAPACITE CHECK(CAPACITE BETWEEN 30 AND 600))
```

Son exécution déclenche l'insertion d'un nouveau tuple dans la table ALL_TAB_COLUMNS décrivant le nouvel attribut CAPACITE. Pour conserver la contrainte de domaine, deux nouveaux tuples sont créés d'une part dans la table ALL_CONSTRAINTS et d'autre part dans la table ALL_CONS_COLUMNS. □

***Exemple** : quel est le nom des séquences et des synonymes créés dans la base par tous les utilisateurs ?*

```
SELECT OBJECT_NAME, OBJECT_TYPE
FROM  ALL_OBJECTS
WHERE OBJECT_TYPE IN ('SEQUENCE', 'SYNONYM')
```
□

2.7.2 - La table système DUAL et les pseudo-colonnes

Hormis certains paramètres très spécifiques accessibles par les commandes ORACLE non SQL (Cf. paragraphe 1.4), toute donnée ne peut être restituée à l'utilisateur qu'en formulant une requête SQL*PLUS. Or toute requête doit obligatoirement inclure le nom de la (ou des) relation(s) sur laquelle (lesquelles) elle s'applique (Cf. introduction du paragraphe 3.1). Dans certains cas particuliers où l'utilisateur souhaite obtenir un résultat calculé par le système et dont la valeur est constante ou indépendante des tuples stockés dans la base, l'obligation de spécifier un nom de relation dans une requête peut être gênante car le résultat est obtenu autant de fois qu'il y a de tuples dans la relation et l'élimination des duplicats (Cf. paragraphe 3.1.1)

n'est pas possible.
Pour éviter ces inconvénients, ORACLE propose une table système particulière réduite à un tuple et un attribut de type **VARCHAR2**(1), qui ne contient qu'une valeur : 'X'. Cette relation appelée DUAL est toujours présente parmi les tables système. Elle peut être utilisée pour obtenir des informations systèmes qui sont soit le résultat de certaines fonctions (e.g. **sysdate** qui rend la date du jour, Cf. paragraphe 3.1.3.2), soit les valeurs de pseudo-colonnes. Ces dernières ont le même comportement que des attributs mais ne sont pas des attributs de relations de la base de données.

Les pseudo-colonnes dans ORACLE sont :

- **CURRVAL** permet d'accéder à la valeur courante d'une séquence (Cf. paragraphes 2.4, 3.1.11 et 3.2.4),
- **NEXTVAL** permet de générer une nouvelle valeur de séquence (Cf. paragraphes 2.4, , 3.1.11 et 3.2.4),
- **LEVEL** indique le numéro d'un niveau dans les requêtes arborescentes (Cf. paragraphe 3.1.16.2),
- **ROWID** retourne l'adresse d'un tuple stocké dans la base de données (Cf. paragraphe 3.1.11),
- **ROWNUM** correspond à l'ordre selon lequel les tuples résultats d'une requête ont été sélectionnés par ORACLE (Cf. paragraphe 3.1.11),
- **USER** rend le nom de l'utilisateur courant (Cf. paragraphe 3.1.11).

Exemple *: Considérons la requête suivante rendant la date du jour :*

```
SELECT sysdate FROM DUAL
```

La date du jour pourrait être rendue en utilisant n'importe quelle relation dans le **FROM** *de la requête précédente mais elle serait répétée autant de fois que le nombre de tuples de la relation.* □

Des exemples d'utilisation de la table DUAL et des pseudo-colonnes sont donnés aux paragraphes 3.1.11 et 3.2.4.

Chapitre 3 - SQL comme Langage de Manipulation des Données

En tant que LMD, SQL permet aux utilisateurs d'exprimer des requêtes d'interrogation, des plus simples aux plus complexes, et d'effectuer des opérations de mise à jour sur les données d'une base.

3.1 - Interrogation des données

Sous sa forme la plus simple, une requête d'interrogation SQL est un bloc composé de différentes clauses respectant un certain ordre. La liste (partielle pour ORACLE) de ces clauses est donnée ci-dessous dans l'ordre devant être suivi.

```
SELECT     <liste_resultat>
FROM       <liste_relations>
[WHERE     <liste_conditions>]
[GROUP  BY <liste_attributs_de_partitionnement>
[HAVING    <liste_conditions_de_partitionnement>]]
[ORDER  BY <liste_attributs_a_trier>]
```

Dans la clause **SELECT**, la liste des résultats attendus pour la requête doit être mentionnée. Si la requête doit rendre plusieurs résultats, ils doivent être séparés par des virgules. Un résultat peut être :

- un nom d'attribut (Cf. paragraphe 3.1.1) ;
- une expression de calcul horizontal (Cf. paragraphe 3.1.3) ;
- l'application d'une fonction agrégative (Cf. paragraphe 3.1.4) ;
- une pseudo-colonne (Cf. paragraphe 3.1.11).

La clause **FROM** permet de spécifier la ou les tables nécessaires pour que le SGBD effectue la requête. Ces tables peuvent être des relations de la base, des vues ou des tables système.

Seules ces deux premières clauses sont obligatoires dans un bloc SQL. Soulignons que sauf cas particuliers et très exceptionnels, si plusieurs relations sont mentionnées dans la clause **FROM**, il faut spécifier une clause **WHERE** pour exprimer les jointures entre les différentes relations (Cf. paragraphes 3.1.6 et 3.1.7). La clause **WHERE** permet également d'exprimer des conditions de sélection des tuples résultats de la requête (Cf. paragraphe 3.1.2).

La clause **GROUP BY** permet d'effectuer des requêtes plus complexes qui sont expliquées au paragraphe 3.1.13. Elle peut être combinée à la clause **HAVING** (Cf. paragraphes 3.1.14 et 3.1.17) mais cette dernière ne peut être utilisée que si un **GROUP BY** a été préalablement introduit dans le bloc SQL.

Enfin, la clause **ORDER BY** introduit une opération de tri des résultats (Cf. paragraphe 3.1.9).

Toute clause apparaît une et une seule fois (si elle est obligatoire) ou au plus une fois (si elle est optionnelle) dans ***tout bloc*** SQL.
Une interrogation ou requête SQL peut être constituée d'un ou de plusieurs blocs. Dans ce dernier cas, les divers blocs peuvent être combinés de différentes manières :

- le résultat attendu résulte d'une opération ensembliste (union, intersection, différence) sur

les résultats de plusieurs blocs (Cf. paragraphe 3.1.10).

- Des résultats intermédiaires doivent être calculés par des blocs imbriqués dans la requête globale (pour ces blocs imbriqués, on parle de sous-requêtes). Le lien entre bloc imbriqué et bloc de niveau supérieur peut être spécifié dans les clauses **WHERE** (Cf. paragraphe 3.1.7) ou **HAVING** (Cf. paragraphe 3.1.14). De plus, bloc de niveau supérieur et bloc imbriqué peuvent être évalués par le SGBD de manière indépendante ou pas. Deux blocs d'une même requête sont dits indépendants si le premier qui doit être exécuté par le SGBD l'est sans que rien ne soit connu de l'exécution du second. Deux blocs sont corrélés si leurs exécutions sont forcément liées.

Exprimer des requêtes en SQL ne présente pas de difficulté syntaxique particulière, cependant une requête peut être syntaxiquement correcte et le système peut retourner des résultats alors que la requête est sémantiquement fausse. De plus, une requête sémantiquement incorrecte peut rendre des résultats justes au jour `j` et se révéler fausse ultérieurement, simplement parce qu'entre temps les données de la base ont changé.

Il existe généralement plusieurs manières de formuler une interrogation et ces alternatives peuvent avoir des temps d'exécution variables à cause des stratégies utilisées par l'optimiseur de requêtes, de la taille des relations, de la possibilité d'utiliser des index…

3.1.1 – Résultat d'une requête et expression des projections

Les résultats d'une requête sont les attributs, pseudo-colonnes ou expressions indiqués dans la clause **SELECT** du premier bloc SQL de la requête.

La forme la plus simple d'une requête de consultation consiste à rechercher toutes les données stockées dans une table. La syntaxe correspondante est la suivante :

```
SELECT * FROM <nom_table>
```

où `<nom_table>` est le nom de la relation (ou de la vue) à consulter. Le caractère `*` signifie qu'aucune projection n'est réalisée, i.e. tous les attributs de la relation font partie du résultat.

Exemple *: donnez toutes les informations sur les pilotes.*

```
SELECT * FROM PILOTE
```
□

Dans la pratique, il est rare que l'on souhaite consulter tous les attributs d'une relation. On se limite souvent à quelques-uns de ces attributs. Dans ce cas, la clause **SELECT** prend la forme suivante : **SELECT** `<colonne1>[, <colonne2>,…]`

Exemple *: donnez le nom et l'adresse des pilotes.*

```
SELECT NOMPIL, ADRESSE
FROM PILOTE
```
□

Le résultat d'une requête est affiché sous forme tabulaire[18]. L'entête de ce résultat est composé des noms de colonnes dans l'ordre de leur apparition dans la clause **SELECT**, ou dans l'ordre de leur définition lors de la création de la table si le caractère * est mentionné dans **SELECT**. Cependant, il est possible de choisir des entêtes différents pour les noms de colonnes en les précisant dans la clause **SELECT**.
Ces entêtes sont appelées alias. La syntaxe de la clause **SELECT** devient alors :

```
SELECT <nom_colonne> [<alias>],…
```

Si <alias> est composé de plusieurs mots (séparés par des blancs), il faut utiliser des guillemets.

***Exemple** : donnez l'identificateur et le nom de chaque pilote.*

```
SELECT NUMPIL "Numéro du pilote", NOMPIL Nom_du_pilote
FROM PILOTE
```

□

Contrairement à l'algèbre relationnelle, l'opération de projection en ***SQL n'élimine pas automatiquement les duplicats***, c'est à l'utilisateur de le préciser, en indiquant le mot clef **DISTINCT** dans la clause **SELECT** avant les attributs projetés. Si plusieurs attributs sont mentionnés dans le **SELECT**, les combinaisons dupliquées de valeurs ne seront pas affichées.

La clause **SELECT** devient :

```
SELECT DISTINCT <colonne1>[, <colonne2>,…] FROM <nom_table>
```

***Exemple** : Considérons l'extension de la relation VOL donnée dans la figure 3.*

NUMVOL	NUMPIL	…	VILLE_DEP	…
IT100	100		MARSEILLE	
IT210	101		PARIS	
IT300	100		MARSEILLE	
AF894	103		MARSEILLE	
AF900	103		PARIS	
IT700	105		PARIS	

Figure 3 : une instance de la relation VOL

Sur cette relation, la requête " quelles sont toutes les villes de départ de vol ? " est formulée de la manière suivante :

```
SELECT VILLE_DEP  FROM VOL
```

Le résultat obtenu est donné par la figure 4.

VILLE_DEP
MARSEILLE
PARIS
MARSEILLE
MARSEILLE
PARIS
PARIS

Figure 4 : Résultat sans élimination de duplicats

[18] Pour chaque attribut, ORACLE réserve pour l'affichage un espace dont la taille est déterminée par celle du type de l'attribut. Le résultat obtenu n'est donc pas forcément très lisible. Pour améliorer sa clarté, utilisez les commandes de formatage proposées par ORACLE (Cf. paragraphe 1.4.3).

Pour éliminer les duplicats dans ce résultat, la requête doit être formulée de la manière suivante :

```
SELECT DISTINCT VILLE_DEP FROM VOL
```

Le résultat est alors celui proposé par la figure 5.

VILLE_DEP
MARSEILLE
PARIS

Figure 5 : Résultat avec élimination de duplicats □

3.1.2 – Expression des sélections

L'opérateur de sélection s'exprime en spécifiant dans la clause **WHERE** une ou des conditions. La clause **WHERE** vient après la clause **FROM**. Elle est facultative.
Une condition est composée de trois termes :

- un nom d'attribut ou une expression,
- un opérateur de comparaison parmi : =, <> ou != pour " différent de ", >, >=, <, <=. Quelques prédicats de comparaison sont également proposés,
- une constante ou le résultat d'un calcul (il est possible de spécifier un nom de colonne mais dans ce cas, il s'agit d'une jointure prédicative et non d'une sélection, Cf. Paragraphe 3.1.6) .

Remarque : les constantes alphanumériques ainsi que les dates doivent être données encadrées par le caractère " ' " (sinon le système les interprète comme des noms d'attribut et génère une erreur).

Qu'elles que soient les conditions exprimées dans la clause WHERE (conditions de sélection ou conditions de jointure), elles sont toujours vérifiées par les tuples résultats de la requête.

Exemple *: donner le nom des pilotes qui habitent à Nice.*

```
SELECT NOMPIL
FROM PILOTE
WHERE ADRESSE = 'NICE'
```
□

Exemple *: donner le nom et l'adresse des pilotes qui gagnent plus de 6.000 €.*

```
SELECT NOMPIL
FROM PILOTE
WHERE SALAIRE > 6000
```
□

En plus des opérateurs de comparaison classiques, SQL dispose des prédicats spécifiques suivants :

- `<nom_attribut> IS NULL`
  teste si la valeur d'une colonne est une valeur nulle (manquante).

  Exemple *: rechercher le nom des pilotes dont l'adresse est inconnue.*

```
SELECT NOMPIL
FROM PILOTE
WHERE ADRESSE  IS  NULL                                                    □
```

- <nom_attribut> **IN** (<liste_valeurs>)
  vérifie si la valeur d'un attribut coïncide avec l'une des valeurs de la liste.

  Exemple *: rechercher toutes les informations sur les avions de nom A320, A330 et A340.*

```
SELECT *
FROM AVION
WHERE NOMAV  IN ('A320', 'A330', 'A340')                                   □
```

- <nom_attribut> **BETWEEN** v1 **AND** v2
  teste si la valeur d'un attribut est comprise entre les valeurs v1 et v2 avec v1 <= valeur <= v2. Attention si l'inégalité est vérifiée mais que les valeurs v1 et v2 sont inversées dans le prédicat, ORACLE ne génère pas de message d'erreur mais rend un résultat vide.
  L'attribut concerné peut être de type numérique, date ou alphabétique. Dans ce dernier cas, une valeur est inférieure à une autre si elle la précède dans l'ordre alphabétique. Pour les dates, v1 est la date la plus ancienne et v2 la plus récente.

  Exemple *: quel est le nom des pilotes qui gagnent entre 6.000 et 7.000 € ?*

```
SELECT NOMPIL
FROM PILOTE
WHERE SALAIRE  BETWEEN  6000  AND  7000                                   □
```

- <nom_attribut> **LIKE** 'chaine_générique'
  teste si la valeur d'un attribut alphanumérique est équivalente à une chaîne de caractère obtenue à partir de la chaîne générique. Dans cette dernière, le symbole " % " remplace une série de caractères quelconque, y compris le vide alors que le symbole " _ " remplace un caractère.

  Exemple *: quelle est la capacité des avions de type Airbus (leur nom commence par A) ?*

```
SELECT CAPACITE
FROM AVION
WHERE NOMAV  LIKE  'A%'                                                    □
```

3.1.2.1 - Conditions négatives

Tous les prédicats spécifiques peuvent être mis sous forme négative en les combinant avec l'opérateur de négation **NOT**. Nous obtenons alors **IS NOT NULL**, **NOT IN**, **NOT LIKE** et **NOT BETWEEN**. De plus la négation de toute condition peut être obtenue en faisant précéder la condition de **NOT**.

Exemple *: quels sont les noms des avions différents de A310, A320, A330 et A340 ?*

```
SELECT NOMAV
FROM AVION
WHERE NOMAV NOT IN ('A310', 'A320', 'A330', 'A340')                        □
```

***Exemple** : quels sont les noms des avions différents de B747 ?*

```
SELECT NOMAV
FROM AVION
WHERE NOT NOMAV = 'B747'
```
□

3.1.2.2 - Combinaisons de conditions

La condition dans la clause **WHERE** peut être une condition composite, i.e. une combinaison de conditions reliées par les opérateurs logiques **AND** et **OR**. L'opérateur **AND** est prioritaire et, si nécessaire, le parenthésage doit être utilisé pour spécifier un ordre d'évaluation particulier.

***Exemple** : quels sont les vols au départ de Nice desservant Paris ?*

```
SELECT *
FROM  VOL
WHERE VILLE_DEP = 'NICE' AND VILLE_ARR = 'PARIS'
```
□

***Exemple** : quels sont les pilotes niçois et marseillais gagnant plus de 6.000 € ?*

```
SELECT *
FROM  PILOTE
WHERE (ADRESSE = 'NICE' OR ADRESSE = 'MARSEILLE')
      AND  SALAIRE > 6000
```
□

***Exemple** : quels sont les pilotes n'étant ni niçois ni marseillais et gagnant plus de 6.000 € ?*

```
SELECT *
FROM PILOTE
WHERE NOT (ADRESSE = 'NICE'   OR  ADRESSE = 'MARSEILLE')
      AND  SALAIRE > 6000
```
□

3.1.2.3 - Constantes de sélection passées en paramètres

Dans ORACLE, il est possible de paramétrer une requête interactive de sélection, de manière à préciser la constante de qualification uniquement au moment de l'exécution de la requête. Pour cela, il suffit dans le texte de la requête de substituer aux différentes constantes de sélection des symboles de variables qui doivent systématiquement commencer par le caractère " `&` ". À l'exécution d'une requête paramétrée, le système demande à l'utilisateur de saisir la valeur des diverses constantes dans l'ordre où elles apparaissent dans la requête.

Remarque : si la constante de sélection est alphanumérique ou de type date, on peut spécifier `'&var'` dans la requête, l'utilisateur n'aura plus qu'à saisir la valeur sans avoir besoin de l'encadrer d'apostrophes, sinon il devra taper `'valeur'`.

***Exemple** : quels sont les vols au départ d'une ville avant une certaine heure ?*

```
SELECT *
FROM  VOL
WHERE VILLE_DEP = '&dep' AND HEURE_ARR < &harr
```

Lors de l'exécution de cette requête, le système demande à l'utilisateur de lui indiquer une ville de départ et une heure d'arrivée. □

Remarque : si une requête utilise plusieurs fois la même constante de sélection paramétrée, l'utilisateur doit à l'exécution saisir sa valeur autant de fois que la constante apparaît dans la requête.

***Exemple** : quels sont les vols au départ ou à l'arrivée d'une ville ?*

```
SELECT *
FROM  VOL
WHERE VILLE_DEP = '&ville' OR VILLE_ARR = '&ville'
```

À l'exécution de la requête, le système demande deux fois la valeur de la variable `&var1`. □

3.1.3 – Calculs horizontaux

Par calcul horizontal, on entend un calcul effectué pour chacun des tuples manipulés par la requête. Ce calcul peut faire intervenir des opérateurs ou des fonctions proposés par le SGBD et peut être formulé dans les clauses **WHERE** et **SELECT**.

Ces expressions de calcul horizontal sont évaluées puis affichées pour chaque tuple appartenant au résultat (formulation dans le **SELECT**) ou évaluées et testées pour chaque tuple potentiellement résultat (formulation dans le **WHERE**).

La formulation des expressions de calcul horizontal utilise :
- des noms d'attributs,
- des constantes,
- des opérateurs ou des fonctions.

Si un index (Cf. paragraphe 2.6) a été défini sur un attribut et que ce dernier est argument ou opérande dans un calcul horizontal, le SGBD ne peut pas utiliser l'index pour améliorer les performances de la requête.

3.1.3.1 - Les opérateurs

Les opérateurs dépendent du type syntaxique de leurs opérandes. Si ce type est :
- numérique, les opérateurs arithmétiques classiques `+`, `-`, `*`, et `/` peuvent être utilisés ;
- alphanumérique, un seul opérateur, celui de concaténation noté `||` est offert (e.g. : `c1 || c2` correspond concaténation des chaînes `c1` et `c2`) ;
- date, `+` et `-` peuvent être utilisés pour ajouter ou soustraire un nombre de jours à une date. L'opérateur `-` peut être utilisé entre deux dates et rend le nombre de jours entre les deux dates arguments.

***Exemple** : donnez le revenu mensuel des pilotes marseillais.*

```
SELECT NUMPIL, NOMPIL, SALAIRE + PRIME
FROM  PILOTE
WHERE ADRESSE = 'MARSEILLE'
```
□

***Exemple** : quels sont les pilotes qui avec une augmentation de 10% de leur prime gagnent moins de 6.000 € ? Donnez leur numéro, leurs revenus actuel et augmenté.*

```
SELECT      NUMPIL, SALAIRE + PRIME, SALAIRE + (PRIME*1.1)
FROM        PILOTE
WHERE       SALAIRE + (PRIME*1.1) < 6000
```
□

Exemple *: Donnez le numéro et l'age des pilotes ayant moins de 40 ans. L'alias Age qui est introduit pour le calcul ne sert qu'à l'affichage.*

```
SELECT      NUMPIL, sysdate - DATE_NAI Age
FROM        PILOTE
WHERE       (sysdate - DATE_NAI)/365 < 40
```
□

3.1.3.2 - Les fonctions de calcul horizontal

Les fonctions disponibles dépendent du SGBD utilisé (leur nom, leurs paramètres, leur fonctionnement peuvent varier d'un système à un autre). Sous ORACLE, de nombreuses fonctions sont proposées[19], les principales[20] sont répertoriées ci-après en les classant par type.
Attention, utilisées dans la clause **WHERE** d'un requête et appliquées à un attribut indexé, ces fonctions interdisent l'utilisation de l'index associé.

Fonctions sur les chaînes de caractères _____

ascii(ch)	renvoie le code ASCII du caractère donné en argument ou celui du premier caractère de la chaîne ch.
chr(n)	effectue la conversion en caractère d'un code ASCII argument.
initcap(ch)	met en majuscule l'initiale de chaque mot d'une chaîne ch donnée.
instr(ch, ssch [,pos] [,n])	recherche une sous-chaîne ssch dans la chaîne ch et si la sous-chaîne est trouvée, renvoie la position de son premier caractère. L'argument optionnel pos indique la position de départ de la recherche dans la chaîne (par défaut 1) et n indique de rechercher la n^{ième} occurrence de la sous-chaîne (par défaut 1).
length(ch)	retourne la longueur de la chaîne argument.
lower(ch)	met en minuscules la chaîne argument.
lpad(ch, long [, ssch])	complète la chaîne ch à gauche par ssch autant de fois que nécessaire pour atteindre la longueur long. Si ssch est omis, des espaces sont utilisés par défaut.
ltrim(ch[, ssch])	supprime d'une chaîne ch tous les caractères présents dans ssch en partant de la gauche et s'arrête dès qu'un caractère non présent dans ssch est rencontré. Si ssch est omis, le caractère espace est pris par défaut.
replace(ch, ssch1 [, ssch2])	remplace dans une chaîne ch toutes les occurrences de ssch1 par ssch2. Si ssch2 est omis, la fonction supprime toutes les occurrences de ssch1.
rpad(ch, long [, ssch])	complète la chaîne ch à droite par ssch jusqu'à atteindre la longueur long. Si ssch est omis, l'espace est utilisé par défaut.
rtrim(ch[, ssch])	supprime d'une chaîne ch tous les caractères présents dans ssch en partant de la droite et s'arrête dès qu'un caractère non présent dans ssch est rencontré. Si ssch est omis, le caractère espace est pris par défaut.
soundex(ch)	renvoie une représentation phonétique de la chaîne argument.
substr(ch, pos [, long])	extrait une sous-chaîne de ch à partir de la position pos. La longueur de la sous-chaîne extraite est long. Si cette longueur est omise, il y a extraction de tous les caractères à partir de pos.

[19] Elles sont implémentées en PL/SQL comme des fonctions stockées (Cf. polycopié : Le langage PL/SQL d'ORACLE).
[20] Une liste exhaustive de ces fonctions est donnée dans [1].

translate(ch, ssch1, ssch2)	remplace chaque occurrence de ssch1 par ssch2 dans la chaîne ch.
upper(ch)	met en majuscules la chaîne argument.

***Exemple** : quels sont les nom et prénom des pilotes domiciliés à Marseille ? Si on ne sait pas comment les valeurs ont été saisies dans la base (minuscule et/ou majuscules), la requête suivante rend les résultats escomptés. De plus les prénoms sont affichés en minuscules avec l'initiale en capitale et les noms sont donnés en majuscules.*

```
SELECT initcap(PRENOMPIL), upper(NOMPIL)
FROM PILOTE
WHERE upper(ADRESSE) = 'MARSEILLE'
```
□

Fonctions sur les dates

add_months(date_a, nbmois)	renvoie une date en augmentant ou en diminuant l'argument date_a du nombre de mois donné nbmois (qui peut être négatif).
last_day(date_a)	donne la date du dernier jour du mois dans lequel se situe date_a.
months_between(d1, d2)	calcule le nombre de mois entre d1 et d2. Si la date d1 est antérieure à d2, le nombre renvoyé est négatif.
next_day(date_a, jour)	donne la date du prochain jour de la semaine après date_a. L'argument jour est le nom d'un jour (e.g. Lundi).
sysdate	renvoie la date du jour et l'heure courantes.

***Exemple** : quels sont les vols (numéro, horaire) au départ de Paris et à destination de Marseille programmés ce jour ?*

```
SELECT NUMVOL, HEURE_DEP, HEURE_ARR
FROM VOL
WHERE VILLE_DEP = 'PARIS' AND VILLE_ARR = 'MARSEILLE'
      AND DATE_VOL = sysdate
```
□

Fonctions sur les nombres

abs(n)	renvoie la valeur absolue d'un nombre n.
ceil(n)	retourne l'entier directement supérieur ou égal à n.
cos(n)	donne le cosinus de l'angle n.
floor(n)	renvoie la partie entière de n.
mod(n, m)	effectue n modulo m.
power(m, n)	calcule m à la puissance n.
round(n [, nbdeci])	arrondit, au nombre supérieur le plus proche, n tronqué au nombre de décimales nbdeci. Si ce dernier n'est pas donné (par défaut 0), rend l'entier immédiatement supérieur à n. Si nbdeci est négatif, le nombre rendu est immédiatement inférieur à n.
sign(n)	renvoie 1 si n > 0, 0 si n = 0 et -1 si n < 0.
sin(n)	donne le sinus de l'angle n.
sqrt(n)	retourne la racine carrée de n (**NULL** si n < 0).
tan(n)	donne la tangente de l'angle n.
trunc(n[, m])	tronque n à m décimales après le point décimal. Si m est négatif, la troncature se fait avant le point décimal. Si m est omis, c'est la valeur entière de n qui est retournée.

***Exemple** : donner la partie entière des salaires des pilotes.*

```
SELECT NOMPIL, floor(SALAIRE) "Salaire : partie entière"
FROM PILOTE
```
□

Fonctions de conversion de type

to_char(comp[, format])	convertit un nombre ou une date comp en chaîne de caractères. Un format peut être précisé.
to_date(ch[, format])	effectue la conversion d'une chaîne de caractères en date.
to_number(ch[, format])	convertit la chaîne ch en valeur numérique.

La fonction **to_number** convertit une chaîne en nombre selon le format éventuellement spécifié. Elle peut être appliquée à des attributs de type **CHAR** ou **VARCHAR2**. Le format de conversion donne un " patron " permettant au système d'interpréter la chaîne. Il est composé d'une série de 0 et/ou de 9, et peut comporter les caractères de séparation de milliers et décimal " , " et " . " ainsi que le caractère " S " (valeur signée) ou " L " (monnaie locale) .

***Exemple** : les formats suivants sont valides et indiqués avec un exemple de résultat obtenu.*

```
fonction de conversion                    Nombre rendu

to_number('2899', '9999')                 2899
to_number('+2899', 'S9999')               2899
to_number('2,899.9', '9,999.9')           2899,9
to_number('F2,899.9', 'L9,999.9')         2899,9
to_number('+882,899,999', 'S999,899,999') 882899999
```
□

La fonction **to_char** admet format comme argument optionnel. Si l'argument de la fonction est numérique, les formats donnés pour la fonction **to_number** peuvent être utilisés.

***Exemple** : les formats suivants sont valides et indiqués avec un exemple de résultat obtenu.*

```
fonction de conversion                    Chaîne rendue

to_char(2899, '9999')                     2899
to_char(2899, 'S9999')                    +2899
to_char(2899.9, '9,999.9')                2,899.9
to_char(2899.9, 'L9,999.9')               F2,899.9
to_char(882899021, 'S999,899,999L')       +882,899,999F
```
□

***Exemple** : la requête suivante est utilisée pour une mise en forme des résultats.*

```
SELECT NUMVOL||' - ' ||to_char(NUMAV, '9999')
FROM VOL
```

Le résultat est une liste de chaînes de caractères de la forme suivante.

```
NUMVOL||' - ' ||to_char(NUMAV, '9999')
--------------------------------------
IT100 - 10
IT200 - 14
IT400 - 15
```
□

Lorsque la fonction **to_char** est appliquée à des attributs de type date, le format peut combiner les éléments donnés dans le tableau 5. Ces mêmes éléments de format peuvent être utilisés avec la fonction **to_date** appliquée sur des chaînes de caractères.

DDD	numéro du jour dans l'année (de 1 à 366)
DD	numéro du jour dans le mois (de 1 à 31)
D	numéro du jour dans la semaine (de 1 à 7)
day	nom du jour de la semaine en minuscules (e.g. lundi)
DAY	nom du jour de la semaine en majuscules (e.g. LUNDI)
Day	nom du jour de la semaine en minuscules avec l'initiale en majuscule (e.g. Lundi)
MM	numéro du mois dans l'année (de 1 à 12)
MONTH	nom du mois en majuscules (e.g. MARS)
month	nom du mois en minuscules (e.g. mars)
Month	nom du mois en minuscules avec initiale en majuscule (e.g. Mars)

MON	nom abrégé du mois en majuscules (e.g. MAR)
mon	nom abrégé du mois en minuscules (e.g. mar)
Mon	nom abrégé du mois en minuscules avec initiale en majuscule (e.g. Mar)
YYYY	année sur 4 chiffres
YY	deux derniers chiffres de l'année
HH24	heure de la journée (de 0 à 23)
HH	heure de la journée (de 1 à 12)
AM ou PM	indicateur méridien (AM : matin, PM après-midi)
MI	minutes (de 0 à 59)
SS	secondes (de 0 à 59)

Tableau 5 : éléments courants de format pour la conversion de dates

Exemple *: pour les conversion de dates en chaînes de caractères, les formats suivants sont valides et indiqués avec un exemple de résultat obtenu. On suppose que* `date_argument` *est, selon le format par défaut, 22/04/03.*

```
fonction de conversion                          Chaîne rendue

to_char(date_arg, 'DD Month YYYY')              '22 Avril 2003'
to_char(date_arg, 'Day DD MONTH YY')            'Mardi 22 AVRIL 03'
to_char(date_arg, 'day D')                      'mardi 2'
to_char(date_arg, 'DD-MM-YY HH24:MI:SS')        '22-04-03 13:18:25'
to_char(date_arg, 'DD/MM/YY HH12-MI')           '22/04/03 1-18'
to_char(date_arg, 'DD/MM/YY HH12-MI AM')        '22/04/03 1-18 PM'
```

Si les chaînes de la seconde colonne ci-dessus sont passées en argument à la fonction **to_date***, avec le format associé indiqué dans la première colonne ci-dessus, le résultat est une date. Si aucun format n'est spécifié, c'est le format par défaut qui est utilisé. On a par exemple :*

```
fonction de conversion                          Date rendue

to_date('22 Avril 2003', 'DD Month YYYY')       '22 Avril 2003'
to_date('Mardi 22 AVRIL 03', 'Day DD MONTH YY') 'Mardi 22 AVRIL 03'
to_date('22 Avril 2003')                        '22/04/03'
to_date('Avril 2003')                           '01/04/03'[21]      □
```

[21] Le premier jour du mois est pris par défaut.

Exemple *: imaginons que l'attribut DATE_NAI est été défini comme ayant le type CHAR(8) et que ses valeurs aient été saisies avec le format usuel des dates. La requête suivante permet d'obtenir la date mais aussi le jour de la semaine.*

```
SELECT NUMPIL, NOMPIL, to_date(DATE_NAI, 'Day DD MONTH YY')
FROM PILOTE
```
□

Autres fonctions _____

greatest(val1, val2[,val3…])	renvoie la plus grande valeur dans la liste d'arguments donnés. Ces derniers peuvent être numériques, alphabétiques ou de type date.
least(val1, val2[,val3…])	retourne la plus petite valeur de la liste. Les arguments peuvent être numériques, alphabétiques ou de type date.
nvl(val1, val2)	permet de substituer la valeur val2 à val1, au cas où cette dernière est une valeur nulle. Les arguments peuvent être numériques, alphabétiques ou de type date.
nvl2(val1, val2, val3)	permet de substituer la valeur val2 à val1 quand cette dernière n'est pas une valeur nulle. Si val1 est une valeur nulle, alors val3 lui est substituée Les arguments peuvent être numériques, alphabétiques ou de type date. Si val2 et val3 ne sont pas du même type, Oracle effectue automatiquement une conversion. Le type de la valeur retournée est toujours celui de val2.

Exemple *: la requête suivante permet d'afficher la chaîne spécifiée si la localisation d'un avion est manquante.*

```
SELECT NUMAV, NOMAV, nvl(LOCALISATION,'localisation inconnue')
FROM AVION
```
□

3.1.3.3 - Expressions de " IF … THEN … ELSE "

ORACLE permet d'utiliser deux formes d'expressions simulant l'instruction conditionnelle des langages procéduraux. La première permet d'exprimer des conditions dont le comparateur est l'égalité. La seconde peut utiliser tous les comparateurs. Il est possible d'imbriquer ces expressions.

decode(val, rech1, res1 [rech2, res2, [rech3, res3, …]], [valdefaut])	Oracle effectue tout d'abord la comparaison de val et rech1 et, s'il y a égalité, le résultat renvoyé est res1. Dans le cas contraire, la comparaison de val et rech2 est réalisée. S'il y a égalité, res2 est retourné sinon val est comparé à rech3 et ainsi de suite. Si toutes les comparaisons échouent, c'est valdefaut qui est retournée si cette valeur est spécifiée sinon une valeur nulle est restituée. Le nombre total d'arguments est fixé à 255.
case when cond1 **then** expr1 [**when** cond2 **then** expr2 [**when** cond3 **then** expr3 …]] [**else** expr] **end**	évalue cond1 et, si la condition est satisfaite, renvoie expr1. Sinon la deuxième condition est évaluée et, si elle est vraie, expr2 est retournée. Si aucune des conditions n'est vérifiée, le résultat sera expr si **else** est spécifié et **NULL** sinon. En fait cette fonction permet de simuler des **IF** imbriqués pour des conditions dont le comparateur est quelconque.

Exemple *: si on souhaite connaître les zones géographiques dans lesquelles sont localisés les avions, la requête suivante peut être formulée. Si l'attribut LOCALISATION prend la valeur 'MARSEILLE' alors le résultat de l'expression est 'SUD EST', il en est de même si la valeur de l'attribut est 'NICE'. Si l'attribut prend la valeur 'PARIS' alors le résultat est 'NORD' et ainsi de suite. Si l'attribut ne prend aucune des valeurs spécifiées, la valeur par défaut '******' est retournée.*

```
SELECT decode(LOCALISATION,'MARSEILLE', 'SUD EST',
       'NICE', 'SUD EST',
       'PARIS', 'NORD',
       'LYON', 'CENTRE',
       'LILLE', 'NORD',
       'TOULOUSE', 'SUD OUEST',
       '******'), NUMAV
FROM AVION
```

□

Exemple *: pour les différents pilotes, l'utilisateur désire connaître le type de leur salaire. Si ce dernier est supérieur à 5000 €, le type est 'HAUT', s'il est compris entre 5000 et 3000 € (exclu), le type est 'MOYEN', sinon le type est 'BAS'.*

```
SELECT NUMPIL, NOMPIL,
       case when SALAIRE > 5000 then 'HAUT'
       when SALAIRE > 3000 then 'MOYEN'
       else 'BAS' end
FROM PILOTE
```

□

Exemple *: on souhaite faire une simulation du calcul de la prime selon les règles suivantes : si le salaire est supérieur à 5.000 et que la prime actuelle est supérieure à 1.000, elle est conservée. Si la prime actuelle est inférieure à 1.000, le taux d'augmentation est de 0.05. Si le salaire est inférieur ou égal à 5.000, le taux d'augmentation est de 0.1 si la prime est supérieure à 1.000 et de 0.15 sinon.*

```
SELECT NUMPIL, NOMPIL,
       case when SALAIRE > 5000 then
           case when PRIME > 1000 then PRIME
           else PRIME * 1.05
           end
       else
           case when PRIME > 1000 then PRIME * 1.1
           else PRIME * 1.15
           end
       end
FROM PILOTE
```

□

3.1.3.4 - La gestion de valeurs nulles dans les calculs horizontaux

La présence de valeurs nulles peut poser des problèmes dans l'évaluation des expressions de calcul horizontal utilisant des opérateurs ou des fonctions.

Généralement, si un des arguments du calcul est **NULL**, le résultat est lui aussi **NULL** (il existe quelques exceptions). Par exemple, pour ORACLE, la somme d'une valeur et d'une inconnue est inconnue. Pour éviter tout problème, il convient d'utiliser la fonction **NVL** (décrite ci avant) qui permet de substituer une valeur par défaut aux valeurs nulles éventuelles.

***Exemple** : l'attribut PRIME peut avoir des valeurs nulles. Supposons que la requête donnant le revenu mensuel des pilotes marseillais soit formulée de la façon suivante.*

```
SELECT NUMPIL, NOMPIL, SALAIRE + PRIME
FROM  PILOTE
WHERE ADRESSE = 'MARSEILLE'
```

*Pour chaque pilote dont la prime est une valeur inconnue, le résultat du calcul horizontal est **NULL** (et en aucun cas n'est égal à son salaire). Pour que l'interrogation précédente rende le résultat désiré, la requête doit être formulée comme suit.*

```
SELECT NUMPIL, NOMPIL, SALAIRE + nvl(PRIME,0)
FROM  PILOTE
WHERE ADRESSE = 'MARSEILLE'
```
□

Comme pour les opérateurs, il faut veiller à la gestion des valeurs nulles lors de l'utilisation de fonctions de calcul horizontal. Ainsi, l'opération de conversion de type (via **to_date**, **to_char** ou **to_number**) appliquée à une valeur manquante rend **NULL**. Si la fonction **length** est utilisée sur une valeur manquante, elle ne retourne pas 0 mais **NULL**. Dans les mêmes conditions, les fonctions **least** et **greatest** rendent **NULL**…

L'opérateur de concaténation de chaînes || constitue une exception pour le traitement de valeurs manquantes. En effet, il considère les valeurs nulles " comme rien " et rend donc le résultat attendu. Il ne retourne une valeur nulle que si ***tous*** ses opérandes sont **NULL**.

***Exemple** : considérons la requête suivante.*

```
SELECT NOMPIL ||' '|| PRENOMPIL
FROM  PILOTE
```

Les résultats peuvent être : le nom suivi (et séparé par un espace) du prénom si les deux sont connus ; seulement le nom (suivi d'un espace) si le prénom est inconnu ; seulement le prénom (précédé d'un espace) si le nom est inconnu ou enfin un espace si les deux données sont manquantes pour un pilote. □

En conclusion, si un attribut n'a pas été défini avec une contrainte de présence obligatoire de valeurs (via la clause **NOT NULL**, Cf. paragraphe 2.2), il faut toujours considérer le cas où aucune valeur ne lui est attribuée par un utilisateur et ***systématiquement gérer les valeurs nulles dans les calculs horizontaux***.

3.1.4 – Calculs verticaux (fonctions agrégatives)

Les calculs verticaux s'appliquent sur un ensemble de valeurs ayant la même sémantique (généralement les valeurs d'un attribut) et mettent en jeu des fonctions dites agrégatives car elles résument ou agrègent un ensemble de valeurs en une seule. La syntaxe générale pour l'utilisation de fonction est la suivante :

```
<nom_fonction> ([DISTINCT]<nom_colonne>)
```

Une fonction agrégative peut s'appliquer sur les valeurs d'un attribut pour tous les tuples (satisfaisant la clause **WHERE** si elle existe) ou seulement sur les valeurs distinctes de cet attribut. Dans ce cas, le mot clef **DISTINCT** doit être précisé avant l'attribut argument de la fonction.

Contrairement aux calculs horizontaux, ***le résultat d'une fonction agrégative est évalué une seule fois pour tous les tuples du résultat***[22].

Les fonctions agrégatives portent sur un ensemble de valeurs d'un attribut de type numérique ou aux résultats numériques d'expressions de calculs horizontaux, sauf pour la fonction de comptage **COUNT** pour laquelle le type de l'attribut argument est indifférent et pour les fonctions **MIN** et **MAX** qui peuvent être utilisées sur des attributs numériques, alphanumériques ou de type date. Dans ces deux derniers cas, la plus petite valeur est soit la première valeur dans l'ordre alphabétique[23] soit la date la plus ancienne ; la plus grande est soit la dernière valeur dans l'ordre alphabétique soit la date la plus récente.
Les fonctions agrégatives disponibles sous ORACLE sont les suivantes :

SUM	somme,
AVG	moyenne arithmétique,
COUNT	nombre ou cardinalité,
MAX	valeur maximale,
MIN	valeur minimale,
STDDEV	écart type (*standard deviation*), racine carrée de la variance,
VARIANCE	variance.

Chacune de ces fonctions a comme argument un nom d'attribut ou une expression. Les valeurs nulles ne posent pas de problème d'évaluation dans la mesure où elles sont ignorées. Le résultat d'une de ces fonctions n'est **NULL** que si, pour ***tous les tuples concernés***, l'attribut argument de la fonction agrégative a comme valeur **NULL** sauf pour la fonction **COUNT** qui rend 0.
La fonction **COUNT** peut prendre comme argument le caractère `*`, dans ce cas, elle rend comme résultat le nombre de lignes retournées par le bloc.

Exemple *: quel est le salaire moyen des pilotes Niçois.*

```
SELECT AVG(SALAIRE)
FROM PILOTE
WHERE ADRESSE = 'NICE'
```

Supposons que l'extension de la relation PILOTE soit celle présentée par la figure 6.

NUMPIL	…	SALAIRE	ADRESSE
100		6000	LYON
110		5000	LYON
120		6500	MARSEILLE
130		5800	PARIS
140		6000	PARIS
150		4800	LILLE

Figure 6 : extension des relations PILOTE et AVION

[22] En cas de partitionnement, cette évaluation est faite pour chaque classe d'équivalence de tuples (Cf. paragraphe 3.1.13).
[23] Dans cet ordre alphabétique, les valeurs en majuscules précèdent les valeurs en minuscules.

Le résultat retourné par la requête est la moyenne des six salaires (5683.33). □

***Exemple** : trouver le nombre de vols au départ de Marseille.*

```
SELECT COUNT(VOLNUM)
FROM VOL
WHERE VILLE_DEP = 'MARSEILLE'
```

*Dans cette requête, **COUNT**(*) peut être utilisé à la place de **COUNT**(VOLNUM). En fait, la fonction **COUNT** peut être appliquée à n'importe quel attribut de la relation VOL du moment que toutes ses valeurs sont dénombrées (et pas ses valeurs distinctes).* □

***Exemple** : donnez les capacités minimale et maximale des appareils localisés à Paris.*

```
SELECT MIN(CAPACITE), MAX(CAPACITE)
FROM AVION
WHERE LOCALISATION = 'PARIS'
```
□

Si un attribut a des valeurs dupliquées dans une relation et que le calcul agrégatif ne doit pas tenir compte de ces duplicats, il faut utiliser le mot clef **DISTINCT**.

***Exemple** : combien de destinations sont desservies au départ de Marseille ?*

```
SELECT COUNT(DISTINCT VILLE_ARR)
FROM  VOL
WHERE VILLE_DEP = 'MARSEILLE'
```

*Dans cette requête, il ne s'agit pas de dénombrer le nombre de vols au départ de Marseille mais seulement les villes desservies. En l'absence de **DISTINCT**, la requête rendrait des résultats faux.* □

Il existe des cas où l'utilisation de **DISTINCT** n'est pas pertinente :

- si les fonctions **MIN** ou **MAX** sont employées ;
- si l'attribut, argument de la fonction agrégative, détermine tous les autres attributs utilisés. Si la requête travaille sur une seule relation (comme pour tous les exemples vus jusqu'à présent), compter les valeurs distinctes d'une clef primaire ou d'une clef candidate est évidemment absurde ;
- si des opérations ensemblistes sont utilisées (Cf. paragraphe 3.1.10).

Une erreur classique est de faire figurer, dans un même bloc SQL, des attributs ou expressions ne relevant pas du même niveau de granularité des données. Les valeurs des attributs ou les résultats de calcul horizontaux proposent des données à un niveau de granularité dit détaillé : ces données sont effectivement stockées pour les divers tuples ou calculables pour les tuples.
Avec l'utilisation de fonctions agrégatives, c'est un niveau de granularité dit agrégé ou résumé qui est proposé à l'utilisateur. En effet le résultat d'une fonction agrégative est une synthèse d'un ensemble de valeurs détaillées. Il ne peut être calculé que si l'ensemble des valeurs détaillées a été parcouru par le système.

Il est donc impossible, lors de l'évaluation d'un même bloc SQL, de demander au SGBD d'afficher ou de comparer des valeurs contenues dans un tuple (ou calculables à partir d'un

tuple) et des valeurs synthétisant celles d'un ensemble de tuples.

Ainsi la formulation suivante de la clause **WHERE** est toujours ***erronée*** :

```
WHERE       <nom_attribut1> θ fonction_agregative(<nom_attribut2>)
```

La clause **SELECT** indiquée ci après est correcte si le bloc intègre un **GROUP BY** (Cf. paragraphe 3.1.13) mais, dans tous les autres cas, ***elle est erronée*** :

```
SELECT      <nom_attribut1>, fonction_agregative(<nom_attribut2>)…
```

3.1.5 – Combinaison de calculs

Les calculs verticaux et horizontaux peuvent être combinés au sein d'un même bloc, dans la clause **SELECT**. Examinons les différentes imbrications de calcul :

- il est possible de donner comme argument, à une fonction agrégative, une expression de calcul horizontal : le calcul horizontal est réalisé pour chacun des tuples concernés. Les `n` résultats obtenus sont alors agrégés en utilisant la fonction agrégative.

  Exemple : Quel est le total des revenus des pilotes ?

  ```
  SELECT SUM(SALAIRE + nvl(PRIME,0))
  FROM PILOTE
  ```

  *Bien sûr l'oubli de la fonction **nvl** provoque un résultat faux car, pour tous les pilotes ayant une prime inconnue, le calcul horizontal rend une valeur nulle qui ne sera pas prise en compte par la fonction agrégative **SUM**.* □

- il est possible d'utiliser les résultats de fonctions agrégatives comme opérandes ou arguments d'un calcul horizontal : les fonctions agrégatives sont évaluées en parcourant tous les tuples concernés. Chacune donne un seul résultat s'il n'y a pas de partitionnement et le calcul horizontal est donc effectué une seule fois.

  Exemple : Cumulez le total des salaires et le total des primes ?

  ```
  SELECT SUM(SALAIRE) + SUM(PRIME)
  FROM PILOTE
  ```

  Le résultat de cette requête est bien sûr similaire à celui obtenu pour la précédente si on est sûr qu'il existe, dans la relation PILOTE, au moins un tuple pour lequel la valeur de la prime est connue. Si cette condition n'est pas vraie à tout instant de la vie de la base, alors la requête doit être formulée comme suit.

  ```
  SELECT SUM(SALAIRE) + nvl(SUM(PRIME),0)
  FROM PILOTE
  ```
  □

- Les deux possibilités de combinaisons de calculs, indiquées précédemment, peuvent elles-mêmes être combinées.

  ***Exemple** : quelle est la partie entière du total des revenus des pilotes ?*

  ```
  SELECT floor(SUM(SALAIRE + nvl(PRIME,0)))
  FROM PILOTE
  ```

La fonction agrégative ***SUM*** *permet de cumuler les n résultats obtenus par le calcul horizontal :* `SALAIRE + nvl(PRIME,0)`*. Elle rend donc un seul résultat dont la fonction* ***floor*** *rend la partie entière.* □

- En revanche, il est absurde d'appliquer un calcul vertical sur le résultat d'un calcul vertical, car ce dernier rend une seule valeur. Appliquer, sur cette valeur, un nouveau calcul vertical, destiné à agréger *n* valeurs en une seule, est évidemment non pertinent. De toutes façons, ORACLE ne permet pas de telles imbrications

  Exemple *: considérons la formulation suivante d'une requête.*

```
SELECT MAX(SUM(SALAIRE))
FROM PILOTE
```

  Comme la fonction agrégative ***SUM*** *calcule un seul résultat, la fonction* ***MAX*** *ne peut que rendre ce résultat.*

  La requête suivante, dont le résultat doit être invariablement 1, est elle aussi non pertinente. De toutes façons, ***ORACLE ne permet pas une telle formulation****.*

```
SELECT COUNT(SUM(SALAIRE))
FROM PILOTE
```
□

3.1.6 – Expression des jointures sous forme prédicative

La formulation d'une jointure consiste à indiquer dans un premier temps les tables à fusionner dans la clause **FROM** et ensuite le critère de jointure sous forme de condition dans la clause **WHERE**. L'utilisation des sous-requêtes peut être une forme équivalente (Cf. paragraphe 3.1.7).
Si deux tables sont mentionnées dans la clause **FROM** et qu'***aucune jointure n'est spécifiée, le système effectue le produit cartésien des deux relations*** (chaque tuple de la première est mis en correspondance avec tous les tuples de la seconde), le résultat est donc faux car les liens sémantiques entre relations ne sont pas utilisés[24]. Donc respectez et vérifiez la règle suivante.

Si `n` tables apparaissent dans la clause **FROM**, il faut ***au moins*** `(n-1)` opérations de jointure.

La forme générale d'une requête de jointure prédicative est :

```
SELECT <nom_colonne1> [,<nom_colonne2>…]
FROM  <nom_table1>, <nom_table2> …
WHERE <condition_jointure>
```

où `<condition_jointure>` est de la forme `<attribut1> θ <attribut2>` et `θ` est un opérateur de comparaison. Si l'opérateur utilisé est l'égalité, on parle d'équi-jointure sinon de thêta-jointure. Bien sûr, les attributs de jointure `<attribut1>` et `<attribut2>` doivent être compatibles. Ils appartiennent respectivement à `<nom_table1>` et `<nom_table2>`. Ces deux relations ne sont pas forcément distinctes (Cf. paragraphe 3.1.6.2).

Exemple *: quel est le numéro et le nom des pilotes résidant dans la ville de localisation de*

[24] Il existe des cas où l'on veut réellement effectuer le produit Cartésien de deux relations mais ils relèvent de l'exception.

l'avion n° 33 ?

```
SELECT NUMPIL, NOMPIL
FROM  PILOTE, AVION
WHERE ADRESSE = LOCALISATION AND NUMAV = 33
```

Considérons l'extension des relations proposées par la figure 7. La condition de sélection va extraire un seul tuple de la relation AVION (l'avion n° 33).

NUMPIL	…	ADRESSE
100		LYON
110		LYON
120		MARSEILLE
130		PARIS
140		PARIS
150		LILLE

NUMAV	…	LOCALISATION
10		PARIS
11		MARSEILLE
20		MARSEILLE
24		PARIS
33		PARIS
35		TOULOUSE

Figure 7 : extension des relations PILOTE et AVION

La jointure entre la relation PILOTE et l'avion sélectionné (localisé à Paris) va produire un tuple résultat chaque fois qu'un pilote habite Paris. Le résultat de cette jointure est illustré par la figure 8.

NUMPIL	…	ADRESSE	NUMAV	…	LOCALISATION
130		PARIS	33		PARIS
140		PARIS	33		PARIS

Figure 8 : liste des tuples retournés par la clause ***WHERE***

La projection rend alors les informations demandées sur les pilotes n° 130 et 140. □

Les noms des attributs mentionnés dans les clauses **SELECT** et **WHERE** doivent être uniques dans les relations citées dans la clause **FROM**. Si deux attributs de deux tables différentes ont le même nom, il faut, pour lever l'ambiguïté, les préfixer par le nom des tables (ou les alias des relations quand ils existent).

Exemple *: quel est le numéro et la ville de départ des vols effectués par un B767 ?*

```
SELECT NUMVOL, VILLE_DEP
FROM  VOL, AVION
WHERE NOMAV = 'B767' AND VOL.NUMAV = AVION.NUMAV
```

□

Exemple *: donner le numéro et le nom des pilotes effectuant des vols au départ de Nice sur des Airbus ?*

```
SELECT DISTINCT PILOTE.NUMPIL, NOMPIL
FROM  PILOTE, VOL, AVION
WHERE VILLE_DEP = 'NICE' AND NOMAV LIKE 'A%' AND
      PILOTE.NUMPIL = VOL.NUMPIL  AND VOL.NUMAV = AVION.NUMAV
```

Dans cette requête, **DISTINCT** *est utilisé dans la clause* **SELECT** *car un même pilote peut effectuer plusieurs vols au départ de Nice avec un Airbus.* □

3.1.6.1 - Problème des thêta-jointures

Les thêta-jointures doivent être utilisées avec beaucoup de précautions car l'utilisateur débutant

leur attribue un comportement qu'elles n'ont pas. Pourtant, exactement comme les équi-jointures, un tuple de la première relation génère un résultat ***chaque fois*** que la condition de jointure est satisfaite avec un tuple de la seconde relation opérande.

Exemple *: quels sont les pilotes n'habitant pas la ville de départ d'un vol ? Un pilote fait partie du résultat si sa ville de résidence n'est la ville de départ d'aucun vol. Supposons que pour répondre à cette question, l'utilisateur formule la requête SQL suivante.*

```
SELECT DISTINCT PILOTE.NUMPIL
FROM  PILOTE, VOL
WHERE ADRESSE != VILLE_DEP
```

Cette requête est logiquement fausse *et les chances pour qu'elle rende à l'utilisateur un résultat juste sont infimes. Analysons son exécution sur l'instance des deux relations illustrées par la figure 9.*

NUMPIL	…	ADRESSE
100		LYON
110		LYON
120		MARSEILLE
130		PARIS
140		PARIS
150		LILLE

NUMVOL	…	VILLE_DEP
IT100		PARIS
IT210		MARSEILLE
IT300		MARSEILLE
AF894		PARIS
AF900		NICE
IT700		TOULOUSE

Figure 9 : extension des relations PILOTE et VOL

La thêta-jointure spécifiée calcule un ensemble de tuples qui est illustré par la figure 10 ***: chaque pilote est associé à tous les vols ne partant pas de sa ville de résidence****.*

NUMPIL	…	ADRESSE	NUMVOL	…	VILLE_DEP
100		LYON	IT100		PARIS
100		LYON	IT210		MARSEILLE
100		LYON	IT300		MARSEILLE
100		LYON	AF894		PARIS
100		LYON	AF900		NICE
100		LYON	IT700		TOULOUSE
110		LYON	IT100		PARIS
110		LYON	IT210		MARSEILLE
110		LYON	IT300		MARSEILLE
110		LYON	AF894		PARIS
110		LYON	AF900		NICE
110		LYON	IT700		TOULOUSE
120		MARSEILLE	IT100		PARIS
120		MARSEILLE	AF894		PARIS
120		MARSEILLE	AF900		NICE
120		MARSEILLE	IT700		TOULOUSE
130		PARIS	IT210		MARSEILLE
130		PARIS	IT300		MARSEILLE
130		PARIS	AF900		NICE
130		PARIS	IT700		TOULOUSE
140		PARIS	IT210		MARSEILLE
140		PARIS	IT300		MARSEILLE
140		PARIS	AF900		NICE
140		PARIS	IT700		TOULOUSE
150		LILLE	IT100		PARIS
150		LILLE	IT210		MARSEILLE
150		LILLE	IT300		MARSEILLE

150		LILLE	AF894		PARIS
150		LILLE	AF900		NICE
150		LILLE	IT700		TOULOUSE

Figure 10 : ensemble des tuples calculés par la thêta-jointure

La projection des numéros de pilote distincts à partir du résultat précédent rend donc tous les pilotes (figure 11) alors que le résultat escompté par l'utilisateur est donné dans la figure 12.

NUMPIL
100
110
120
130
140
150

Figure 11 : résultat de la requête formulée

NUMPIL	…	ADRESSE
100		LYON
110		LYON
150		LILLE

Figure 12 : résultat attendu

Il est évident avec les instances des relations choisies que la formulation de la requête est fausse. En fait, il suffit qu'il existe un vol ne partant pas de la ville de résidence d'un pilote pour que le pilote considéré réponde à la condition de jointure et fasse donc partie du résultat. Dans notre exemple, c'est le cas de tous les pilotes. La requête considérée ne peut pas être formulée avec une thêta-jointure prédicative mais elle peut être exprimée de plusieurs autres manières, expliquées plus loin. □

3.1.6.2 - Auto-jointures prédicatives

Si une requête de jointure prédicative nécessite plusieurs fois l'utilisation d'une même relation (cas des auto-jointures), il faut utiliser des alias, afin que le système puisse reconnaître les différents rôles joués par cette relation.

Un tel alias est introduit dans la clause **FROM** juste après le nom de la relation dont il est séparé par un espace. Dans ORACLE, l'introduction d'un alias interdit l'utilisation du nom de la relation associée dans les autres clauses de la requête. Il faut donc utiliser l'alias pour préfixer les attributs apparaissant dans les clauses **SELECT** ou **WHERE**.

***Exemple** : quels sont les avions localisés dans la même ville que l'avion numéro 103 ?*

```
SELECT AUTRES.NUMAV, AUTRES.NOMAV
FROM  AVION AUTRES, AVION AV103
WHERE AV103.NUMAV = 103  AND AUTRES.NUMAV <> 103  AND
      AV103.LOCALISATION = AUTRES.LOCALISATION
```

Dans cette requête l'alias AV103 est utilisé pour retrouver l'avion de numéro 103 et l'alias AUTRES permet de balayer tous les tuples de AVION pour faire la comparaison des localisations. □

***Exemple** : quelles sont les correspondances (villes d'arrivée) accessibles à partir de la ville d'arrivée du vol IT100 ?*

```
SELECT DISTINCT AUTRES.VILLE_ARR
FROM  VOL AUTRES, VOL VOLIT100
```

```
WHERE VOLIT100.NUMVOL = 'IT100'  AND
      VOLIT100.VILLE_ARR = AUTRES.VILLE_DEP
```

La requête recherche les vols dont la ville de départ correspond à la ville d'arrivée du vol IT100 puis ne conserve que les villes d'arrivée de ces vols. □

3.1.6.3 - Un cas particulier de double thêta-jointures

Le prédicat **BETWEEN** peut être utilisé pour formuler une double thêta-jointure. Dans ce cas, au lieu d'utiliser des constantes de sélection, des noms d'attributs sont spécifiés. Bien sûr les problèmes, évoqués au paragraphe 3.1.6.1, peuvent toujours se poser. Il faut donc faire très attention à la sémantique de la requête.

***Exemple** : Quel est le numéro des vols décollant pendant la durée du vol IT100 ?*

```
SELECT V.NUMVOL
FROM  VOL V, VOL IT100
WHERE IT100.NUMVOL = 'IT100'  AND
      IT100.DATE_VOL = V.DATE_VOL  AND
      V.HEURE_DEP BETWEEN IT100.HEURE_DEP AND IT100.HEURE_ARR
```

Dans cette requête, la condition de sélection sur une valeur de clef primaire implique qu'au plus un tuple est conservé pour l'une des deux relations à fusionner. Il n'y a donc pas de problème et les résultats obtenus sont corrects. □

3.1.6.4 - Jointures prédicatives et calculs horizontaux

Il est possible d'exprimer une jointure prédicative non pas entre attributs mais pour comparer les résultats d'expression de calculs horizontaux. La condition de jointure apparaissant dans la clause **WHERE** prend alors la forme : `<expression1> θ <expression2>`

***Exemple** : Quel est le numéro des pilotes qui augmentés de 10% ont le même salaire que le pilote n° 100 ?*

```
SELECT AUTRES.NUMPIL
FROM  PILOTE AUTRES, PILOTE PIL100
WHERE PIL100.NUMPIL = 100  AND
      PIL100.SALAIRE = AUTRES.SALAIRE * 1.1
```
□

***Exemple** : Quel est le numéro des pilotes qui avec une augmentation de 10% de leur salaire ont le même revenu que le pilote n° 100 ?*

```
SELECT AUTRES.NUMPIL
FROM  PILOTE AUTRES, PILOTE PIL100
WHERE PIL100.NUMPIL = 100  AND
      PIL100.SALAIRE + nvl(PIL100.PRIME, 0) =
      AUTRES.SALAIRE * 1.1 + nvl(AUTRES.PRIME, 0)
```
□

3.1.6.5 - Jointures externes

Lors d'une jointure prédicative, un tuple est généré comme résultat uniquement s'il y a satisfaction de la condition de jointure entre deux tuples des relations opérandes. Par opposition aux précédentes, les jointures externes permettent de conserver dans le résultat tous les tuples de l'une des deux relations opérandes qu'ils satisfassent ou pas la condition de jointure. Un tuple ne vérifiant pas cette condition est complété par des valeurs **NULL** pour les attributs de la deuxième relation opérande.
La forme générale d'une requête de jointure externe est :

```
SELECT <nom_colonne1> [, <nom_colonne2>…]
FROM  <nom_table1>, <nom_table2> …
WHERE <attribut1> θ <attribut2> (+)
```

où <attribut1> appartient à <table1> et <attribut2> appartient à <table2>. Le symbole (+) représentant la jointure externe est spécifié juste après l'attribut qui admettra des valeurs nulles.

Exemple *: On souhaite connaître, pour les différents vols (NUMVOL, VILLE_DEP), les avions localisés dans la ville de départ de ces vols (NUMAV, NOMAV). S'il n'en existe pas, le vol devra aussi être affiché. On suppose que l'extension des relations VOL et AVION est celle donnée dans la figure 13.*

NUMVOL	…	VILLE_DEP
IT100		PARIS
IT210		MARSEILLE
IT300		MARSEILLE
AF894		PARIS
AF900		NICE
IT700		TOULOUSE

NUMAV	NOMAV	…	LOCALISATION
10	A340		LYON
11	A320		LYON
12	B747		MARSEILLE
13	A340		PARIS
14	B767		PARIS
15	B747		LILLE

Figure 13 : extension des relations VOL et AVION

```
SELECT      NUMVOL, VILLE_DEP, AVION.NUMAV, NOMAV
FROM        VOL, AVION
WHERE       VILLE_DEP = LOCALISATION (+)
```

La jointure externe, exprimée ci-dessus, permet d'obtenir le résultat d'une jointure prédicative classique mais en conservant les tuples de vol pour lesquels la ville de départ ne correspond à la localisation d'aucun appareil. Le résultat de cette opération est proposé par la figure 14. Alors que les deux derniers tuples ne sont pas retournés par une jointure classique, ils apparaissent ici dans le résultat en ayant une valeur ***NULL*** *pour les attributs NUMAV et NOMAV (pas d'affichage de valeurs).*

NUMVOL	VILLE_DEP	NUMAV	NOMAV
IT100	PARIS	13	A340
IT100	PARIS	14	B767
IT210	MARSEILLE	12	B747
IT300	MARSEILLE	12	B747
AF894	PARIS	13	A340
AF894	PARIS	14	B767
AF900	NICE		
IT700	TOULOUSE		

Figure 14 : résultat de la jointure externe □

Dans certains dialectes SQL ne permettant pas les jointures imbriquées (Cf. Paragraphe 3.1.7), les jointures externes peuvent être utilisées pour répondre à certaines requêtes, par exemple celles incluant un test de non existence de tuple dans une relation. Dans ce cas, outre une jointure externe, le prédicat **IS NULL** est spécifié. Ce prédicat est évalué par le système après avoir complété les tuples résultant de la jointure externe avec des valeurs nulles.

***Exemple** : Quels sont les pilotes n'effectuant aucun vol ?*

```
SELECT PILOTE.NUMPIL, NOMPIL
FROM   PILOTE, VOL
WHERE PILOTE.NUMPIL = VOL.NUMPIL (+) AND VOL.NUMPIL IS NULL
```

NUMPIL	NOMPIL	…
100	DURAND	
110	MARTIN	
120	DUVAL	
130	DUPONT	
140	DUPRE	
150	DUBOIS	

NUMVOL	NUMPIL	…	VILLE_DEP
IT100	100		PARIS
IT210	100		MARSEILLE
IT300	120		MARSEILLE
AF894	140		PARIS
AF900	140		NICE
IT700	120		TOULOUSE

Figure 15 : extension des relations PILOTE et VOL

En supposant que les relations utilisées par la requête sont celles de la figure 15, la jointure externe retourne l'ensemble des tuples illustré par la figure 16.

NUMPIL	NOMPIL	…	NUMVOL	NUMPIL	…	VILLE_DEP
100	DURAND		IT100	100		PARIS
100	DURAND		IT210	100		MARSEILLE
110	MARTIN					
120	DUVAL		IT300	120		MARSEILLE
120	DUVAL		IT700	120		TOULOUSE
130	DUPONT					
140	DUPRE		AF894	140		PARIS
140	DUPRE		AF900	140		NICE
150	DUBOIS					

Figure 16 : résultat de la jointure externe entre PILOTE et VOL

*La sélection mettant en œuvre le prédicat **IS NULL** restreint l'ensemble de tuples à ceux indiqués dans la figure 17.*

NUMPIL	NOMPIL	…	NUMVOL	NUMPIL	…	VILLE_DEP
110	MARTIN					
130	DUPONT					
150	DUBOIS					

Figure 17 : résultat de la sélection

La projection rend les données pour les trois pilotes indiqués dans la figure précédente. □

Dans une requête contenant une jointure externe, supposons que R1 soit la relation dont tous les tuples sont conservés et R2 celle pouvant avoir les valeurs nulles pour l'attribut de jointure. Si une condition de sélection classique est exprimée sur un attribut de R2, elle ne sera évidemment pas satisfaite pour tous les tuples composés de valeurs nulles. ORACLE permet d'exprimer cette

sélection tout en conservant les valeurs nulles, en utilisant une sélection " externe " symbolisée par (+).

```
SELECT <nom_colonne1> [, <nom_colonne2>…]
FROM  <nom_table1>, <nom_table2> …
WHERE <attribut1> θ <attribut2> (+)
      AND <attribut3> (+) θ <constante | expression>
```

où `<attribut1>` appartient à `<table1>`, `<attribut2>` et `<attribut3>` appartiennent à `<table2>`. Le symbole `(+)` suivant `<attribut3>` permet d'indiquer que pour les tuples résultats, l'attribut `<attribut3>` satisfait la condition de sélection ou est **`NULL`**.

***Exemple** : quels sont les pilotes effectuant un vol au départ de Marseille ou n'effectuant aucun vol ?*

```
SELECT PILOTE.NUMPIL, NOMPIL
FROM  PILOTE, VOL
WHERE PILOTE.NUMPIL = VOL.NUMPIL (+) AND VILLE_DEP(+) = 'MARSEILLE'
```

Supposons que les deux relations aient l'extension donnée dans la figure 15. La jointure externe et la sélection rendent l'ensemble des tuples suivants.

NUMPIL	NOMPIL	…	NUMVOL	NUMPIL	…	VILLE_DEP
100	DURAND		IT210	100		MARSEILLE
110	MARTIN					
120	DUVAL		IT300	120		MARSEILLE
130	DUPONT					
140	DUPRE					
150	DUBOIS					

Figure 18 : résultat de la jointure externe entre PILOTE et VOL

Évidemment, la sélection `VILLE_DEP = 'MARSEILLE'` *ne retournerait que les pilotes n° 100 et 120.* □

Il existe différentes restrictions et précautions à prendre lors de la formulation de jointures externes. Si les deux relations à fusionner sont liées par plusieurs conditions de jointure et qu'une seule jointure externe est formulée, ORACLE restitue un ensemble de tuples calculés en utilisant de simples jointures. Pour obtenir un résultat provenant effectivement de jointures externes, il faut que ***toutes*** les jointures soient exprimées comme des jointures ***externes***.
L'opérateur `(+)` peut s'appliquer seulement à un attribut (et pas à une expression). Néanmoins, une expression peut inclure : `<attribut> (+)`.
Une condition de jointure externe ne peut pas être combinée à une autre condition par un **`OR`** logique. Enfin, il est impossible de comparer un attribut marqué de l'opérateur `(+)` au résultat d'une sous-requête, ni d'effectuer une comparaison avec une liste de valeurs en utilisant **`IN`**.

3.1.7 – Expression des jointures sous forme imbriquée

Une requête imbriquée dans une autre est appelée sous-requête ou sous-bloc. Elle se formule de la même façon que toute requête de recherche à la différence près qu'elle doit être mise entre

parenthèses. Une sous-requête peut avoir elle-même une autre sous-requête et ainsi de suite. ORACLE autorise jusqu'à 255 niveaux d'imbrication. L'utilisation de sous-requêtes permet d'exprimer des opérations de jointure (dite imbriquée) mais elle étend aussi les possibilités de comparaison des valeurs d'un attribut. Dans ce dernier cas, il ne s'agit pas d'opérer une jointure au sens strict du terme (comparer les valeurs d'un attribut à celles d'un autre attribut) mais de pouvoir comparer les valeurs d'un attribut à des résultats de calculs verticaux ou des ensembles de valeurs. Les formulations imbriquées peuvent bien sûr être combinées à des jointures prédicatives.

Les jointures imbriquées permettent de mettre en correspondance les valeurs d'un attribut (ou plusieurs) apparaissant dans la clause **WHERE** d'un bloc avec les valeurs retournées par une sous-requête. Il existe quatre cas différents d'expression de jointure imbriquée dépendant du nombre et de la forme des résultats des sous-requêtes.

Dans tous les cas examinés ci-après, le ***bloc imbriqué est évalué de manière indépendante***. Le système calcule tout d'abord le ou les résultats de la sous-requête puis le bloc de niveau supérieur est exécuté et pour chaque tuple examiné, la condition de jointure est évaluée en comparant l'attribut ou l'expression de jointure aux résultats obtenus pour la sous-requête.

Les différents cas de sous-requêtes sont les suivants : le bloc imbriqué rend une valeur (premier cas) ; il retourne un ensemble de valeurs de même sémantique (deuxième cas) ; il rend un tuple (troisième cas) ; il calcule un ensemble de tuples (quatrième cas).

1. Premier cas :

Le résultat de la sous-requête est formé *d'une seule valeur*. C'est surtout le cas de sous-requêtes utilisant une fonction agrégative dans la clause **SELECT**. L'attribut ou l'expression de jointure spécifié dans la clause **WHERE** du premier bloc est simplement comparé au résultat du **SELECT** du bloc imbriqué à l'aide de l'opérateur de comparaison voulu.
La clause prend la forme suivante :

```
WHERE <expression1> θ (SELECT <expression2> FROM <relation> …)
```

<expression1> peut être un nom d'attribut ou un calcul horizontal ***mais en aucun cas un calcul vertical*** (impossible dans une clause **WHERE**).
<expression2> peut être un nom d'attribut, un calcul horizontal ou un calcul vertical. ***Il faut toujours s'assurer que le bloc imbriqué rend une unique valeur quelle que soit l'extension de la base de données.***

Exemple *: quel est le nom des pilotes gagnant plus que le salaire moyen des pilotes ?*

```
SELECT NOMPIL
FROM PILOTE
WHERE SALAIRE > (SELECT AVG(SALAIRE) FROM PILOTE)
```

Le bloc imbriqué fait un parcours de tous les tuples de la relation PILOTE afin de calculer le salaire moyen. Puis le 1er bloc balaye, tuple à tuple, la relation PILOTE et, pour chacun, compare l'attribut SALAIRE au salaire moyen calculé préalablement. Si la condition de jointure est satisfaite, l'attribut projeté du tuple examiné fait partie du résultat. □

Exemple *: quel est le nom des pilotes gagnant exactement le même salaire que le pilote n° 100 ?*

```
SELECT NOMPIL
FROM PILOTE
```

```
WHERE SALAIRE =
          (SELECT SALAIRE  FROM PILOTE WHERE NUMPIL = 100)
```
□

Dans l'exemple précédent, la condition de sélection portant sur une seule valeur de clef primaire et la projection sur un seul attribut garantissent que le bloc imbriqué ne retourne qu'au plus une valeur. ***Dès que cette garantie ne peut être donnée, cette forme de jointure ne doit absolument pas être utilisée*** ; le danger étant qu'à un instant `t` la requête fonctionne correctement du fait des données actuellement stockées mais, logiquement fausse, elle pourra produire des erreurs ultérieurement.

Exemple *: quel est le nom des pilotes gagnant exactement le même salaire que le pilote Dupont ?*

```
SELECT NOMPIL
FROM PILOTE
WHERE SALAIRE =
          (SELECT SALAIRE FROM PILOTE WHERE NOMPIL = 'DUPONT')
```

Cette requête fonctionne correctement tant qu'il n'existe qu'un seul pilote de nom Dupont dans la base. ***Dès qu'il y a un homonyme, son exécution génère des erreurs****. Une requête de ce type doit être formulée en appliquant le deuxième cas de sous-requête.* □

2. Deuxième cas :

Le résultat de la sous-requête est formé ***d'un ensemble de valeurs d'un attribut ou d'un ensemble de résultats d'une expression de calcul horizontal***. Dans ce cas, le résultat de la comparaison, exprimée dans la clause **WHERE**, peut être considéré comme vrai soit :

- si la condition doit être vérifiée pour ***au moins une des valeurs résultats*** de la sous-requête. Dans ce cas, la sous-requête doit être précédée d'un attribut (de jointure) ou d'une expression, du comparateur suivi de **ANY** (remarque : lorsque le comparateur est l'égalité, = **ANY** peut être remplacé par **IN**).
  L'imbrication de blocs se fait donc de la manière suivante :

  ```
  WHERE <expression1> θ ANY (SELECT <expression2> FROM …)
  ```

  L'expression d'équi-jointure se fait par l'une des deux formes équivalentes suivantes :
  ```
  WHERE <expression1> = ANY (SELECT <expression2> FROM … )
  ```
  ou
  ```
  WHERE <expression1> IN (SELECT <expression2> FROM … )
  ```

  Comme pour le premier cas de bloc imbriqué, `<expression1>` peut être un nom d'attribut ou une expression de calcul horizontal, `<expression2>` est soit un nom d'attribut, soit une expression de calcul horizontal, soit un calcul vertical utilisant une fonction agrégative. Dans ce dernier cas, le bloc imbriqué est une sous-requête avec partitionnement (Cf. paragraphe 3.1.13).
  Il est possible d'utiliser la négation **NOT IN** pour signifier que les valeurs de `<expression1>` ne doivent pas appartenir à l'ensemble des valeurs retournées par le bloc imbriqué.

- si la condition doit être vérifiée pour ***toutes les valeurs résultats*** de la sous-requête, alors la sous-requête doit être précédée d'un attribut (de jointure) ou d'une expression de calcul horizontal, du comparateur suivi de **ALL**.
  L'imbrication de blocs se fait donc de la manière suivante :

```
WHERE <expression1> θ ALL(SELECT <expression2> FROM …)
```

θ peut être n'importe quel opérateur de comparaison sauf l'égalité. En effet, pour chaque tuple lu par le bloc de niveau supérieur, `<expression1>` prend une ***seule*** valeur qui ne peut jamais être égale à ***toutes*** les valeurs retournées par le bloc imbriqué.

Exemple *: quel est le nom des pilotes en service au départ de Nice ?*

```
SELECT NOMPIL
FROM PILOTE
WHERE NUMPIL IN
            ( SELECT NUMPIL
              FROM VOL
              WHERE VILLE_DEP = 'NICE')
```

□

Exemple *: quel est le numéro des avions localisés à Nice dont la capacité est supérieure à celle de l'un des appareils effectuant un Paris-Nice ?*

```
SELECT NUMAV
FROM  AVION
WHERE LOCALISATION = 'NICE' AND CAP > ANY
            (SELECT CAP
            FROM  AVION
            WHERE NUMAV = ANY
                        (SELECT NUMAV
                        FROM VOL
                        WHERE VILLE-DEP = 'PARIS' AND
                              VILLE_ARR = 'NICE') )
```

□

Exemple *: donner le nom des pilotes niçois qui gagnent plus qu'un pilote parisien.*

```
SELECT NOMPIL
FROM  PILOTE
WHERE ADRESSE = 'NICE'  AND SALAIRE >  ANY
            (SELECT SALAIRE   FROM  PILOTE
            WHERE ADRESSE = 'PARIS')
```

□

Exemple *: donner le numéro des pilotes n'effectuant pas de vol entre Paris et Marseille.*

```
SELECT NUMPIL
FROM  PILOTE
WHERE NUMPIL NOT IN
                (SELECT NUMPIL FROM      VOL
                WHERE VILLE_DEP = 'PARIS'
                AND VILLE_ARR = 'MARSEILLE')
```

□

Exemple *: y a t'il une ville desservie au départ de Paris mais pas au départ de Marseille ?*

```
SELECT VILLE_ARR
FROM  VOL
WHERE VILLE_DEP = 'PARIS'  AND VILLE_ARR <>  ALL
                (SELECT VILLE_ARR
                FROM   VOL
                WHERE VILLE_DEP = 'MARSEILLE')
```

Cette requête peut s'exprimer avec **NOT IN** *(à la place de* <> **ALL***) entre les deux blocs.* □

***Exemple** : quel est le nom des pilotes niçois qui gagnent plus que tous les pilotes parisiens ?*

```
SELECT NOMPIL
FROM  PILOTE
WHERE ADRESSE = 'NICE'  AND SALAIRE >  ALL
                  (SELECT SALAIRE
                  FROM PILOTE
                  WHERE ADRESSE = 'PARIS')
```

Cette requête peut aussi être exprimée en utilisant la fonction agrégative **MAX** *dans le bloc imbriqué.*

```
SELECT NOMPIL
FROM  PILOTE
WHERE ADRESSE = 'NICE'  AND SALAIRE >
                  (SELECT MAX(SALAIRE)
                  FROM PILOTE
                  WHERE ADRESSE = 'PARIS')
```

□

De manière générale, avec les comparateurs =, !=, <=, <, >=, >, les fonctions agrégatives **MIN** et **MAX** peuvent être utilisés dans la clause **SELECT** du bloc imbriqué et donc dispenser de l'utilisation de **ALL**. La formulation de la requête se ramène alors au premier cas examiné.

***Exemple** : quel est le pilote dont le nom est le premier dans l'ordre alphabétique ? Donnez son nom, son prénom et son adresse.*

```
SELECT NOMPIL, PRENOMPIL, ADRESSE
FROM  PILOTE
WHERE NOM = (SELECT MIN(NOMPIL) FROM PILOTE)
```

□

Comparons les deux formulations de ce type de requête (avec d'une part **ALL** (premier cas) et d'autre part l'utilisation de **MIN** ou **MAX** (second cas)).

- Dans les deux cas, le système doit parcourir les tuples concernés par le bloc imbriqué, soit pour extraire les valeurs demandées, soit pour calculer le résultat de la fonction agrégative.
- Dans un deuxième temps, le système exécute le bloc de niveau supérieur. Pour chaque tuple satisfaisant la clause **WHERE**, il compare la valeur de l'attribut de jointure :
  - soit à un ensemble de valeurs (utilisation de **ALL**) ;
  - soit à une unique valeur agrégée (utilisation de **MIN** ou **MAX**).

  Cette deuxième phase est donc plus performante dans le second cas, surtout si le nombre de valeurs à comparer (via **ALL**) est important.

1. Troisième cas :

Le résultat de la sous-requête est *un seul tuple*. Les attributs ou expressions de jointure spécifiés dans la clause **WHERE** du premier bloc sont indiqués entre parenthèses et comparés au tuple résultat du **SELECT** du bloc imbriqué à l'aide de l'opérateur de comparaison voulu.
La clause prend la forme suivante :

```
WHERE  (<expression1>, <expression2>[, <expression3>…]) θ
(SELECT <expr1>, <expr2>[, <expr3>…] FROM <relation> …)
```

où `<expressioni>` est un nom d'attribut ou un calcul horizontal et `<expri>` est un nom d'attribut, un calcul horizontal ou un calcul vertical.
De plus les contraintes suivantes doivent être respectées :

- le nombre d'expressions mentionnées entre parenthèses dans la clause **WHERE** doit être identique à celui de la clause **SELECT** de la sous-requête,
- la comparaison se fait entre les valeurs des expressions de la requête et celles de la sous-requête deux à deux selon l'ordre d'apparition de ces expressions,
- il faut que l'opérateur de comparaison θ soit le même pour toutes les expressions de jointure.

***Exemple** : quels sont le vols desservant la même ligne que le vol IT100 ?*

```
SELECT *
FROM  VOL
WHERE (VILLE_DEP, VILLE_ARR) =
                  (SELECT VILLE_DEP, VILLE_ARR
                  FROM VOL
                  WHERE NUMVOL = 'IT100')
```

Le bloc imbriqué rend au plus un couple de villes. Les vols parcourus par le 1ier bloc sont retenus si leurs villes de départ et d'arrivée sont identiques au couple de valeurs retrouvé par le bloc imbriqué. □

***Exemple** : existe t'il un pilote ayant le salaire maximal et la prime la plus importante ?*

```
SELECT NUMPIL
FROM  PILOTE
WHERE (SALAIRE, PRIME) =
                  (SELECT MAX(SALAIRE), MAX(PRIME)
                  FROM PILOTE)
```

Le bloc imbriqué calcule deux valeurs agrégées. Les pilotes balayés par le 1ier bloc sont retenus si leur salaire et de leur prime sont respectivement identiques aux valeurs agrégées. □

1. Quatrième cas :

Le résultat de la sous-requête est un *ensemble de tuples*. Les attributs ou expressions de jointure spécifiés dans la clause **WHERE** du premier bloc sont indiqués entre parenthèses et comparés :

- soit à ***l'un des tuples*** résultats du **SELECT** du bloc imbriqué en utilisant l'opérateur de comparaison voulu suivi du mot clef **ANY**. L'expression = **ANY** est équivalente à **IN**.
- soit à ***tous les tuples*** résultats du bloc imbriqué en utilisant l'opérateur de comparaison voulu suivi du mot clef **ALL**.

La clause prend la forme suivante :

```
WHERE  (<expression1>, <expression2>[, <expression3>…]) θ ANY | ALL
(SELECT <expr1>, <expr2>[, <expr3>…] FROM <relation> …)
```

où <expressioni> est un nom d'attribut ou un calcul horizontal et <expri> est un nom d'attribut, un calcul horizontal ou un calcul vertical. Dans ce dernier cas, le bloc imbriqué doit inclure un partitionnement sinon la requête se ramène au troisième cas. De plus les contraintes suivantes doivent être respectées :

- le nombre d'expressions mentionnées entre parenthèses dans la clause **WHERE** doit être identique à celui de la clause **SELECT** de la sous-requête,
- la comparaison se fait entre les valeurs des expressions de la requête et celles de la sous-requête deux à deux dans l'ordre d'apparition de ces expressions,
- il faut que l'opérateur de comparaison et " l'objet " de cette comparaison (**une** combinaison de valeurs ou **toutes** les combinaisons de valeurs) soient les mêmes pour toutes les expressions concernées.

***Exemple** : rechercher le nom des pilotes ayant même adresse et même salaire que Dupont.*

```
SELECT NOMPIL
FROM  PILOTE
WHERE NOMPIL  <> 'DUPONT'AND (ADRESSE, SALAIRE)  IN

                  (SELECT ADRESSE, SALAIRE
                  FROM PILOTE
                  WHERE NOMPIL = 'DUPONT')
```

Le bloc imbriqué rend un ensemble de couples : la valeur de l'adresse et celle du salaire pour tous les pilotes se nommant Dupont. Le nom des pilotes résultats de la requête correspond à ceux dont l'adresse et le salaire font partie des couples résultats du bloc imbriqué, i.e. identiques à ceux d'un des pilotes appelés Dupont. □

***Exemple** : rechercher le nom des pilotes dont l'adresse et le salaire sont différents de ceux de tous les pilotes appelés Dupont.*

```
SELECT NOMPIL
FROM  PILOTE
WHERE (ADRESSE, SALAIRE) <> ALL
                  (SELECT ADRESSE, SALAIRE
                  FROM PILOTE
                  WHERE NOMPIL = 'DUPONT')
```

Le bloc imbriqué rend un ensemble de couples : la valeur de l'adresse et du salaire pour tous les pilotes se nommant Dupont. Le nom des pilotes résultats de la requête correspond à ceux dont l'adresse et le salaire sont différents des couples résultats du bloc imbriqué. □

1. Comparaison des formulations imbriquées :

Considérons les deux formes de requêtes imbriquées suivantes :

(a)
```
SELECT A1, A2 … , An
FROM  R1
WHERE (Ai, Aj) IN
                  (SELECT Ai, Aj
                  FROM R2
                  WHERE Ak θ Condition)
```

(b)
```
SELECT A1, A2 … , An
FROM  R1
WHERE Ai IN
                  (SELECT Ai
                  FROM R2
                  WHERE Ak θ Condition)
      AND Aj IN
                  (SELECT Aj
```

```
                    FROM R2
                    WHERE Ak θ Condition)
```

Les formulations (a) et (b) ne sont équivalentes que si chacun des blocs imbriqués rend ***toujours*** un unique résultat (un couple pour (a), deux valeurs pour (b)). C'est le cas, par exemple, lorsque `Ak` est la clef primaire de `R2` et `θ` est l'égalité. Dans tous les autres cas de figure, les formulations ne sont pas équivalentes. En effet, dans l'expression (a) de la requête, un tuple de `R1` fait partie du résultat si ses valeurs pour `(Ai, Aj)` appartiennent à l'ensemble des couples retourné par la requête imbriquée. Dans l'expression (b), un tuple de `R1` figure dans le résultat si sa valeur pour `Ai` fait partie de l'ensemble des valeurs solution du premier bloc imbriqué et si sa valeur pour `Aj` fait partie de l'ensemble des valeurs solution du deuxième bloc imbriqué. Rien ne dit que la valeur de `Ai` et celle de `Aj` appartiennent au même tuple de `R2`.

Exemple *: Considérons la requête " donner toutes les informations sur les pilotes ayant même adresse et même salaire que Dupont ", en supposant que la relation PILOTE a l'extension illustrée par la figure 19.*

NUMPIL	NOMPIL	…	ADRESSE	SALAIRE
100	DUPONT		MARSEILLE	6000
101	DUPRE		PARIS	7000
102	DUBOIS		MARSEILLE	6000
103	DUVAL		MARSEILLE	7000
104	DUPONT		PARIS	7000
105	DUPONT		NICE	6000

Figure 19 : extension de la relation PILOTE

Analysons l'expression suivante de la requête :

```
SELECT NOMPIL
FROM  PILOTE
WHERE NOMPIL  <> 'DUPONT'
      AND (ADRESSE, SALAIRE)  IN
                  (SELECT DISTINCT ADRESSE, SALAIRE
                  FROM PILOTE
                  WHERE NOMPIL = 'DUPONT')
```

Le bloc imbriqué rend les couples indiqués dans la figure 20.

ADRESSE	SALAIRE
MARSEILLE	6000
PARIS	7000
NICE	6000

Figure 20 : résultat du bloc imbriqué

Le résultat de la requête, illustré dans la figure 21, est donc constitué d'un seul tuple.

NUMPIL	NOMPIL	…	ADRESSE	SALAIRE
102	DUBOIS		MARSEILLE	6000

Figure 21 : résultat de la requête

En considérant la même extension de la relation PILOTE, analysons l'expression suivante de la requête :

```
SELECT NOMPIL
FROM PILOTE
WHERE NOMPIL  <> 'DUPONT'
      AND ADRESSE  IN
                  (SELECT ADRESSE FROM PILOTE
                  WHERE NOMPIL = 'DUPONT')
      AND SALAIRE  IN
                  (SELECT SALAIRE FROM PILOTE
                  WHERE NOMPIL = 'DUPONT')
```

Le premier bloc imbriqué rend les valeurs de l'attribut ADRESSE indiquées dans la figure 22 et le deuxième celles de l'attribut SALAIRE données dans la figure 23.

ADRESSE
MARSEILLE
PARIS
NICE

Figure 22 : résultat du premier bloc imbriqué

SALAIRE
6000
7000
6000

Figure 23 : résultat du deuxième bloc imbriqué

Les tuples de la relation initiale pour lesquels les attributs ADRESSE et SALAIRE appartiennent aux deux ensembles solutions des blocs imbriqués sont illustrés dans la figure 24.

NUMPIL	NOMPIL	…	ADRESSE	SALAIRE
101	DUPRE		PARIS	70000
102	DUBOIS		MARSEILLE	60000
103	DUVAL		MARSEILLE	70000

Figure 24 : résultat de la requête

Le résultat de la requête est donc l'ensemble des pilotes ayant même adresse qu'un des " Dupont " et même salaire qu'un des " Dupont ", il est donc logiquement faux et c'est la première formulation qu'il faudra utiliser. □

3.1.8 – Jointure prédicative versus jointure imbriquée

Pour des raisons de performance, préférez chaque fois que c'est possible une formulation prédicative des jointures plutôt que la formulation équivalente en imbriqué. Ainsi donc réservez l'utilisation de sous-requêtes aux seuls cas où il est impossible de formuler les mêmes jointures de manière prédicative. Ces cas se limitent aux situations suivantes :

- la requête nécessite (au moins) deux parcours d'ensembles de tuples ne restituant pas les résultats au même niveau de détail des données. Typiquement, si des valeurs contenues dans des tuples doivent être comparées à un résultat agrégé issu d'un calcul vertical, il faut que ce dernier soit réalisé dans un bloc imbriqué ;
- il est nécessaire de comparer des valeurs contenues dans les tuples à toutes les valeurs d'un ensemble qui doit être calculé indépendamment, i.e. par une sous-requête ;
- dans certains cas de thêta-jointure où la formulation prédicative ne peut s'appliquer (Cf. paragraphe 3.1.6.1) ;
- lors de la définition de certaines vues utilisées pour permettre des mises à jour et nécessitant plusieurs relations (Cf. paragraphe 4.3).

3.1.9 – Tri des résultats

Dans le modèle relationnel, basé sur la théorie ensembliste, il n'y a pas lieu d'ordonner les tuples dans les relations de la base. Cependant, il est possible avec SQL d'ordonner les résultats d'une requête. Cet ordre peut être croissant ou décroissant et s'appliquer à une ou plusieurs colonnes ou expressions (au maximum 255). Ce tri s'obtient en ajoutant à la requête la clause **ORDER BY**, qui doit toujours être la dernière clause d'un bloc SQL. La syntaxe de cette clause est la suivante :

```
ORDER BY <expression1> [ASC | DESC] [NULLS FIRST | NULLS LAST]
[, <expression2> [ASC | DESC] [NULLS FIRST | NULLS LAST]…]
```

Les arguments de la clause **ORDER BY** peuvent être des noms de colonne ou des expressions non nécessairement mentionnées dans la clause **SELECT**. Les options **ASC** et **DESC** indiquent le sens dans lequel seront ordonnés les résultats. L'ordre ascendant (ou croissant) est pris par défaut. Si plusieurs expressions sont indiquées, l'ordre est fait d'abord selon la première, s'il existe des lignes ayant même valeur pour cette première expression, alors l'ordre se fait suivant la deuxième expression et ainsi de suite. Avec l'alternative **NULLS FIRST** ou **NULLS LAST**, l'utilisateur peut demander à ce que les tuples ayant des valeurs nulles pour l'attribut ou l'expression de tri apparaissent au début du résultat ou à la fin.

Si la requête contient à la fois une élimination de duplicats par un **DISTINCT** dans la clause **SELECT** et une clause de tri, les arguments de **ORDER BY** doivent obligatoirement apparaître dans le **SELECT**.
Si la requête contient un partitionnement (Cf. paragraphe 3.1.13) et un tri, la clause **ORDER BY** peut contenir les mêmes arguments que la clause **GROUP BY** ainsi que des expressions utilisant des fonctions agrégatives ou leur alias.

***Exemple** : quelle est la liste des pilotes, selon l'ordre alphabétique de leur nom, effectuant un vol au départ de Paris ?*

```
SELECT DISTINCT NOMPIL
FROM VOL, PILOTE
WHERE VOL.NUMPIL = PILOTE.NUMPIL AND VILLE_DEP = 'PARIS'
ORDER BY NOMPIL
```
□

***Exemple** : donner la liste des pilotes niçois par ordre de salaire décroissant, puis par ordre alphabétique des noms puis selon les prénoms. Notons que le dernier attribut de tri n'est pas mentionné dans les résultats de la requête.*

```
SELECT NOMPIL, SALAIRE
FROM PILOTE
WHERE ADRESSE = 'NICE'
ORDER BY SALAIRE  DESC,  NOMPIL, PRENOMPIL
```
□

***Exemple** : donner la liste des lignes desservies par ordre alphabétique des villes de départ puis d'arrivée.*

```
SELECT DISTINCT VILLE_DEP, VILLE_ARR
FROM VOL
ORDER BY VILLE_DEP, VILLE_ARR
```
□

La clause **ORDER BY** vise à faciliter la lecture des résultats d'une requête. Elle n'a évidemment pas de sens dans un bloc imbriqué.

Si la clause **SELECT** intègre un calcul vertical ou horizontal et que l'utilisateur souhaite effectuer un tri selon le résultat de ce calcul, il faut attribuer à l'expression de calcul un alias et utiliser cet alias dans la clause **ORDER BY**.

Exemple *: reprenons la requête permettant de connaître les zones géographiques dans lesquelles sont localisés les avions. On souhaite que les résultats soient triés selon la zone géographique. Il suffit d'utiliser un alias ZONE et une clause de tri.*

```
SELECT decode(LOCALISATION,'MARSEILLE', 'SUD EST',
       'NICE', 'SUD EST',
       'PARIS', 'NORD',
       'LYON', 'CENTRE',
       'LILLE', 'NORD',
       'TOULOUSE', 'SUD OUEST',
       'FRANCE') ZONE, NUMAV
FROM AVION
ORDER BY ZONE
```

□

3.1.10 – Opérateurs ensemblistes

Ces opérations consistent à faire l'union, l'intersection ou la différence des résultats de requêtes à condition que ces résultats soient uni-compatibles, i.e. qu'ils aient le même nombre de colonnes et les mêmes types pour les colonnes de même position apparaissant dans le **SELECT**. ***Avec ces trois opérations, il y a élimination automatique des duplicats.***
La forme d'une opération d'union, d'intersection ou de différence est :

```
<requete1>
UNION | INTERSECT | MINUS
<requete2>
```

Exemple *: quel est le nom des avions de capacité supérieure à 250 ou localisés à Paris ?*

```
SELECT NOMAV FROM AVION WHERE CAPACITE > 250
UNION
SELECT NOMAV FROM AVION WHERE LOCALISATION = 'PARIS'
```

Une forme équivalente est :

```
SELECT NOMAV FROM   AVION
WHERE CAPACITE > 250 OR LOCALISATION  = 'PARIS'
```

□

Exemple *: quel est le numéro des pilotes qui conduisent les avions n° 2 et 4 ?*

```
SELECT NUMPIL FROM VOL WHERE NUMAV  = 2
INTERSECT
SELECT NUMPIL FROM VOL WHERE NUMAV  = 4
```

L'attribut NUMAV étant mono-valué, la requête ne peut en aucun cas être exprimée en un seul bloc avec la condition suivante : `NUMAV = 2 OR NUMAV = 4`. *En effet, le résultat consisterait*

en l'ensemble des pilotes conduisant l'avion n° 2 ou conduisant l'avion n°4, mais pas forcément les deux. Évidemment avec la condition suivante : `NUMAV = 2 AND NUMAV = 4`*, le résultat est toujours vide car la condition est toujours fausse :* `NUMAV` *ne peut pas prendre deux valeurs différentes pour un tuple.* □

***Exemple** : quels sont les numéros des pilotes qui conduisent l'avion n° 2 sans jamais conduire l'avion n° 4 ?*

```
SELECT NUMPIL FROM VOL WHERE NUMAV  = 2
   MINUS
SELECT NUMPIL FROM VOL WHERE NUMAV  = 4
```

Évidemment la requête ne peut en aucun cas être exprimée en un seul bloc avec la condition suivante : `NUMAV = 2 AND NUMAV != 4`*. En effet, si la première condition est vraie, la seconde l'est aussi et la requête ne rend pas le résultat attendu.* □

Si une requête contient plusieurs opérateurs ensemblistes et que les divers blocs ne sont pas parenthésés, Oracle évalue les opérations dans l'ordre de leur apparition dans la requête. Considérons la requête suivante.

```
<requete1>
INTERSECT
<requete2>
MINUS
<requete3>
```

L'intersection entre les tuples résultats de `<requete1>` et `<requete2>` est d'abord calculée puis de l'ensemble obtenu sont éliminés les tuples résultats de `<requete3>`. Si l'ordre d'évaluation qui doit être mis en œuvre est différent de cet ordre implicite, le parenthésage doit être utilisé.

Remarque : les opérateurs ensemblistes, en particulier l'intersection et la différence, ne sont pas proposés par tous les dialectes SQL. Si le dialecte utilisé propose la différence mais pas l'intersection, cette dernière peut être simulée.

`<requete1>` **INTERSECT** `<requete2>` peut s'exprimer par :

`<requete1>` **MINUS** *(*`<requete1>` **MINUS** `<requête2>`*)*

3.1.11 – Interrogation de séquences et pseudo-colonnes

Lorsqu'une séquence est créée (Cf. paragraphe 2.4), les pseudo-colonnes **CURRVAL** et **NEXTVAL** permettent à l'utilisateur de manipuler les valeurs générées automatiquement. Plus précisément, la première utilisation de **NEXTVAL** renvoie la valeur initiale de la séquence (celle qui a été spécifiée ou la valeur par défaut). Lors de la deuxième utilisation, la valeur de la séquence est incrémentée (par le pas spécifié ou la valeur par défaut) et ainsi de suite. Toute utilisation de **CURRVAL** retourne la valeur courante de la séquence, i.e. celle calculée par la dernière utilisation de **NEXTVAL**.

Les séquences créées peuvent être interrogées via une requête SQL. Plus précisément, un utilisateur peut connaître la valeur courante d'une séquence ou la prochaine valeur, en utilisant les pseudo-colonnes **CURRVAL** et **NEXTVAL** dans la clause **SELECT** d'un bloc non imbriqué. Les

diverses restrictions sur l'utilisation de **CURRVAL** et **NEXTVAL** sont les suivantes. Ils peuvent apparaître dans la clause **SELECT**, à condition de ne pas utiliser de **DISTINCT**, d'un bloc non imbriqué et ne comportant ni **GROUP BY** ni **ORDER BY**. Ils ne peuvent pas apparaître dans une clause **WHERE**. De même ils sont interdits dans les requêtes ensemblistes et les requêtes de définition de vue.

Supposons que la requête suivante soit formulée.

```
SELECT <nom_sequence>.CURRVAL
FROM <nom_relation>
```

La valeur courante de la séquence est retournée à l'utilisateur autant de fois qu'il y a de tuples dans <nom_relation>. En présence d'une clause **WHERE**, cette valeur serait rendue autant de fois qu'il y a de tuples satisfaisant les conditions indiquées dans le **WHERE**. Évidemment, la même multiplication des résultats peut être observée en utilisant **NEXTVAL**.
Pour éviter d'obtenir la même réponse de multiples fois, il est d'usage d'utiliser la table système DUAL (Cf. paragraphe 2.7.2) qui ne contient qu'une seule valeur, de la manière suivante.

```
SELECT <nom_sequence>.CURRVAL
FROM DUAL
```

Si un bloc fait référence à la fois à **CURRVAL** et **NEXTVAL**, alors Oracle incrémente d'abord la séquence indiquée, puis il retourne ***la même valeur*** pour **CURRVAL** et **NEXTVAL** et ce quel que soit l'ordre dans lequel apparaissent ces deux pseudo-colonnes.

Outre pour la manipulation de séquences, d'autres pseudo-colonnes sont proposées par Oracle et peuvent être utilisées dans les requêtes comme **ROWID** rendant l'adresse des tuples, **ROWNUM** indiquant l'ordre dans lequel les tuples sont sélectionnés ou **USER** restituant le nom de l'utilisateur.

***Exemple** : quel est le loggin ORACLE de l'utilisateur ?*

```
SELECT USER
FROM DUAL
```
□

***Exemple** : quel est le nom des relations créées par l'utilisateur courant ?*

```
SELECT TABLE_NAME
FROM  ALL_TABLES
WHERE OWNER = USER
```
□

***Exemple** : on souhaite sélectionner (sans critère particulier), dix vols au départ de Paris.*

```
SELECT *
FROM  VOL
WHERE ROWNUM <= 10 AND VILLE_DEP = 'Paris'
```
□

3.1.12 – Test d'absence ou d'existence de données

Pour vérifier qu'une donnée est absente dans la base, le cas le plus simple est celui où l'existence de tuples est avérée et on cherche si un attribut a des valeurs manquantes. Il suffit alors d'utiliser le prédicat **IS NULL**. Mais cette solution n'est correcte que si les tuples existent. Elle ne peut en aucun cas s'appliquer si on cherche à vérifier l'absence de tuples.

Exemple *: Les requêtes suivantes ne peuvent pas être formulées avec* **IS NULL**. *Quels sont les pilotes n'effectuant aucun vol ? Quelles sont les villes de départ dans lesquelles aucun avion n'est localisé ? Pour la première requête, un pilote fait partie du résultat si son identifiant n'est pas présent pour l'attribut* NUMPIL *dans la relation* VOL. *De manière similaire, une ville résultat de la seconde interrogation n'est pas une valeur de l'attribut* LOCALISATION *dans* AVION. *Nous avons vu que de telles requêtes ne peuvent pas être exprimées avec des thêta-jointures (Cf. paragraphe 3.1.6.1).* □

Pour formuler des requêtes de type " un élément n'appartient pas à un ensemble donné ", plusieurs techniques peuvent être utilisées. La première consiste à utiliser une jointure imbriquée avec **NOT IN**. La sous-requête est utilisée pour calculer l'ensemble des éléments non voulus dans le résultat et le bloc de niveau supérieur extrait les éléments n'appartenant pas à cet ensemble. De manière analogue, <> **ALL** peut être utilisé précédant une sous-requête. Une troisième possibilité est d'avoir recours à une différence ensembliste (**MINUS**). La première requête calcule la totalité des éléments et la seconde ceux qui ne sont pas voulus dans le résultat. La quatrième possibilité fait appel à une jointure externe et une sélection avec le prédicat **IS NULL**.

Exemple *: considérons la requête " quels sont les pilotes n'effectuant aucun vol ? ". Les formulations suivantes permettent d'y répondre.*

```
SELECT NUMPIL, NOMPIL
FROM  PILOTE
WHERE NUMPIL   NOT IN
            (SELECT NUMPIL FROM VOL)

SELECT NUMPIL, NOMPIL
FROM  PILOTE
WHERE NUMPIL <> ALL
            (SELECT NUMPIL FROM VOL)

SELECT NUMPIL, NOMPIL         --ensemble de tous les pilotes
FROM  PILOTE
MINUS
SELECT NUMPIL, NOMPIL         -- ensemble des pilotes faisant des vols
FROM PILOTE, VOL
WHERE PILOTE.NIMPIL = VOL.NUMPIL

SELECT PILOTE.NUMPIL, NOMPIL
FROM  PILOTE, VOL
WHERE PILOTE.NUMPIL = VOL.NUMPIL(+) AND NUMVOL IS NULL          □
```

Une cinquième approche fait appel au prédicat **NOT EXISTS** qui s'applique à un bloc imbriqué et rend la valeur vrai si le résultat de la sous-requête est vide (et faux sinon).
NOT EXISTS tout comme **EXISTS** s'utilisent de la façon suivante :

```
SELECT <liste_attributs>
FROM  <liste_relations>
WHERE [<liste_conditions> AND | OR] [NOT] EXISTS
            (<sous_requete>)
```

Pour chacun des tuples traités par le 1ier bloc, si la sous-requête rend un résultat, **EXISTS** est évalué à vrai et **NOT EXISTS** à faux. Si la sous-requête ne rend aucun résultat, **EXISTS** est évalué à faux et **NOT EXISTS** à vrai.

Le bloc principal et la sous-requête peuvent être indépendants ou corrélés. Plus précisément, ***si la sous-requête rend le même résultat quels que soient les tuples traités dans le 1ier bloc, il y a indépendance et la sous-requête est évaluée une seule fois***. Si ***l'évaluation de la sous-requête rend un résultat qui varie en fonction de chaque tuple traité par le bloc de niveau supérieur, il y a corrélation et la sous-requête est évaluée autant de fois que le nombre de tuples manipulés par le 1ier bloc***. Une telle corrélation s'exprime via ***une jointure entre les deux blocs : un ou des alias de relations introduits dans le bloc de niveau supérieur sont utilisés, dans le bloc imbriqué, pour spécifier des conditions de jointure***.

Considérons la requête suivante où les deux blocs sont indépendants.

```
SELECT A1, A2,…
FROM  R1
WHERE [NOT] EXISTS
            (SELECT * FROM R2)
```

La sous-requête est évaluée une seule fois. Il suffit qu'il existe un tuple dans la relation `R2`, pour que la condition exprimée dans la clause **`WHERE`** du premier bloc soit :

- toujours vraie si le prédicat **`EXISTS`** est utilisé ;
- toujours fausse si le prédicat **`NOT EXISTS`** est utilisé.

Dans le premier cas, tous les tuples balayés par le 1ier bloc sont utilisés pour produire le résultat de la requête, dans le second cas aucun tuple n'est retenu et le résultat est toujours vide. Évidemment, si la relation `R2` est vide, les constatations précédentes doivent être inversées : la requête utilisant **`EXISTS`** rend toujours un résultat vide, celle intégrant **`NOT EXISTS`** rend tous les tuples manipulés par le 1ier bloc.

L'intérêt des requêtes de ce type est donc très limité : les tuples calculés par le 1ier bloc ne sont pertinents que par rapport à la présence ou l'absence de tuples retournés par la sous-requête.

Exemple *: s'il existe au moins un pilote domicilié à Lille, donnez les vols partant de Lille ?*

```
SELECT *
FROM  VOL
WHERE VILLE_DEP = 'LILLE' AND EXISTS
            (SELECT * FROM PILOTE WHERE ADRESSE = 'LILLE')
```

Si la sous-requête rend un ou plusieurs tuples, ils correspondent à tous les vols au départ de Lille, si aucun vol n'est affiché, c'est qu'aucun pilote n'habite Lille. □

Considérons à présent le cas où les blocs constituant la requête doivent être corrélés. Nous avons vu qu'il est nécessaire d'exprimer une jointure entre blocs en utilisant des alias. La requête suivante en est une illustration.

```
SELECT A1, A2,…
FROM  R1 alias1
WHERE [NOT] EXISTS
            (SELECT * FROM R2 WHERE alias1.A1 = R2.B3)
```

Nous supposons que les attributs `A1` de `R1` et `B3` de `R2` sont les attributs de jointure. Analysons le déroulement de la requête. Le premier bloc parcourt `n` tuples : de `t1` à `tn`. Pour le premier `t1`, la sous-requête est évaluée en cherchant si l'attribut `B3` admet la valeur de `t1[A1]`. Si c'est le cas et que le prédicat **`EXISTS`** est utilisé, il est évalué à vrai et le tuple `t1` produit un résultat pour la requête. Dans les mêmes conditions, **`NOT EXISTS`** est évalué à faux et `t1` ne produit aucun résultat. Le deuxième tuple `t2` est examiné, la sous-requête est évaluée avec la condition

`t2[A1] = B3`. Suivant le résultat de cette sous-requête et le prédicat utilisé, `t2` peut ou pas générer un résultat. L'exécution se poursuit pour chacun des tuples jusqu'à `tn`.

Évidemment si une requête nécessite une corrélation entre blocs, l'oubli de l'expression de la ou des jointures conduit le SGBD à produire un résultat logiquement faux. Ce danger est détaillé à travers l'exemple suivant.

Exemple *: reprenons la requête " quels sont les pilotes n'effectuant aucun vol ". La formulation suivante est* ***fausse****.*

```
SELECT NUMPIL, NOMPIL
FROM   PILOTE
WHERE NOT EXISTS
            (SELECT * FROM    VOL)
```

Il suffit qu'un vol soit créé dans la relation VOL pour que la requête précédente retourne systématiquement un résultat vide. En effet le bloc imbriqué rendra un tuple et le ***NOT EXISTS*** *sera toujours évalué à faux, donc aucun pilote ne sera retourné par le premier bloc. Le problème de cette requête est que le lien entre les éléments cherchés dans les deux blocs n'est pas spécifié. Or, il faut indiquer au système que le 2ème bloc doit être évalué pour chacun des pilotes examinés par le 1er bloc. Pour cela, on introduit un alias pour la relation PILOTE et une jointure est exprimée dans le 2ème bloc en utilisant cet alias. La formulation correcte est la suivante :*

```
SELECT NUMPIL, NOMPIL
FROM   PILOTE  PIL
WHERE NOT EXISTS
                  (SELECT          *
                  FROM             VOL
                  WHERE            VOL.NUMPIL = PIL.NUMPIL)          □
```

En utilisant deux blocs corrélés et le prédicat **`EXISTS`**, il est possible de formuler d'une autre manière les jointures exprimées sous forme prédicative ou imbriquée. Cependant de telles formulations sont plus délicates et leur exécution plus coûteuse. Elles doivent donc être évitées.

3.1.13 – Classification ou partitionnement

La classification ou le partitionnement permet d'effectuer un traitement SQL (généralement l'application d'une fonction agrégative) non pas sur un ensemble de tuples mais sur des sous-ensembles (ou classes) d'un ensemble de tuples. Ces sous-ensembles sont distingués par la valeur que prennent les tuples les composant pour un attribut (ou plusieurs) dit(s) attribut(s) de partitionnement. Ainsi, la classification permet de regrouper les lignes d'une table dans des classes d'équivalence (ou sous-tables) ayant chacune la même valeur pour l'attribut de partitionnement. Ces classes forment une partition de l'extension de la relation considérée (i.e. l'intersection des classes deux à deux est vide et leur union est égale à la relation initiale).

Exemple *: considérons la relation VOL illustrée par la figure 25. Partitionner cette relation sur l'attribut NUMPIL consiste à regrouper au sein d'une même classe tous les vols assurés par le même pilote. NUMPIL prenant 4 valeurs distinctes, 4 classes sont introduites. Elles sont mises en évidence dans la figure 26 et regroupent respectivement les vols des pilotes n° 100, 102, 105 et 124.*

NUMVOL	NUMPIL	…	VILLE_DEP
IT100	100		NICE
AF101	100		NICE
IT101	102		PARIS
BA003	105		MARSEILLE
BA045	105		PARIS
IT305	102		MARSEILLE
AF421	124		LONDRES
BA047	105		LONDRES
BA087	105		PARIS

Figure 25 : relation exemple VOL

NUMVOL	NUMPIL	…
IT100	100	
AF101	100	
IT101	102	
IT305	102	
BA003	105	
BA045	105	
BA047	105	
BA087	105	
AF421	124	

Figure 26 : regroupement selon NUMPIL □

Lorsqu'un partitionnement est effectué sur plusieurs attributs, les tuples appartenant à la même classe d'équivalence ont les mêmes valeurs pour tous les attributs de partitionnement.

Exemple *: considérons la relation VOL de la figure précédente. Partitionner cette relation sur les attributs NUMPIL et VILLE_DEP revient à créer des classes d'équivalence en regroupant tous les vols assurés par le même pilote au départ de la même ville. Ces classes sont mises en évidence dans la figure 27.*

NUMVOL	NUMPIL	…	VILLE_DEP
IT100	100		NICE
AF101	100		NICE
IT101	102		PARIS
IT305	102		MARSEILLE
BA003	105		MARSEILLE
BA045	105		PARIS
BA087	105		PARIS
BA047	105		LONDRES
AF421	124		LONDRES

Figure 27 : relation VOL partitionnée selon deux attributs □

En SQL, l'opérateur de partitionnement s'exprime par la clause **GROUP BY** qui doit suivre la clause **WHERE** (ou **FROM** si **WHERE** est absente). Sa syntaxe est :

```
GROUP BY <colonne1> [, <colonne2>,…]
```

En présence de la clause **GROUP BY**, les fonctions agrégatives mentionnées dans la clause **SELECT** s'appliquent à l'ensemble des valeurs de ***chaque classe d'équivalence***. Il y a donc autant de résultats agrégés qu'il y a de valeurs (ou combinaisons de valeurs) distinctes pour l'attribut (ou les attributs) de partitionnement.

Exemple *: quel est le nombre de vols effectués par chaque pilote ?*

```
SELECT NUMPIL, COUNT(NUMVOL)
FROM  VOL
GROUP BY NUMPIL
```

*Dans ce cas, le résultat de la requête comporte une ligne par numéro de pilote présent dans la relation VOL. L'argument de la fonction de comptage peut être * ou n'importe quel attribut de la relation VOL (du moment que les duplicats sont préservés). Si cette relation est celle de la figure 25, le résultat obtenu est le suivant.*

NUMPIL	*COUNT(NUMVOL)*
100	*2*
102	*2*
105	*4*
124	*1*

□

Exemple *: combien de fois chaque pilote conduit-il chaque avion ?*

```
SELECT NUMPIL, NUMAV, COUNT(NUMVOL)
FROM  VOL
GROUP BY NUMPIL, NUMAV
```

Nous obtenons ici autant de lignes par pilote qu'il y a d'avions distincts conduits par le pilote considéré. Chaque classe d'équivalence créée par le ***GROUP BY*** *regroupe tous les vols ayant même pilote et même avion.* □

Exemple *: donnez pour les divers pilotes, le nombre de vols effectués au départ de chaque ville ?*

```
SELECT NUMPIL, VILLE_DEP, COUNT(*)
FROM  VOL
GROUP BY NUMPIL, VILLE_DEP
```

Si la relation utilisée est celle de la figure 25, le résultat obtenu est le suivant.

NUMPIL	*VILLE_DEP*	*COUNT(NUMVOL)*
100	*NICE*	*2*
102	*PARIS*	*1*
102	*MARSEILLE*	*1*
105	*MARSEILLE*	*1*
105	*PARIS*	*2*
105	*LONDRES*	*1*
124	*LONDRES*	*1*

□

Les colonnes indiquées dans **SELECT**, sauf les attributs arguments des fonctions agrégatives, doivent être mentionnées dans **GROUP BY**. La réciproque n'est pas obligatoire.

Néanmoins lorsqu'un **GROUP BY** est spécifié dans le premier bloc d'une requête, il est fréquent d'indiquer les attributs de partitionnement dans la clause **SELECT**, ne serait-ce que pour que l'utilisateur puisse interpréter les résultats obtenus (sinon il obtient des valeurs agrégées sans savoir à quoi elles correspondent).

Exemple *: la formulation des requêtes suivantes est valide.*

```
SELECT VILLE_DEP, COUNT(NUMVOL)
FROM VOL
GROUP BY VILLE_DEP
```

```
SELECT COUNT(NUMVOL)
FROM VOL
GROUP BY VILLE_DEP
```

Dans ce second cas, le résultat de la requête est un ensemble de comptage mais l'utilisateur ignore à quelle ville de départ est associée chaque valeur agrégée. □

Exemple *: la formulation suivante est considérée comme invalide car l'attribut NOMPIL n'est*

pas indiqué dans la clause de partitionnement. ORACLE renvoie le message d'erreur : " N'est pas une expression GROUP BY ".

```
SELECT PILOTE.NUMPIL, NOMPIL, COUNT(NUMVOL)
FROM  VOL, PILOTE
WHERE VOL.NUMPIL = PILOTE.NUMPIL
GROUP BY PILOTE.NUMPIL
```
□

Partitionner une relation sur un attribut clef primaire ou clef candidate est évidemment complètement inutile (chaque classe d'équivalence est réduite à un seul tuple !) mais fort coûteux (un algorithme de tri est mis en œuvre par le SGBD lors des partitionnements).
De même, mentionner un **DISTINCT** devant les attributs de la clause **SELECT** ne sert à rien si le bloc comporte un **GROUP BY** : ces attributs étant aussi les attributs de partitionnement, il n'y aura pas de duplicats parmi les valeurs ou combinaisons de valeurs retournées. Il faut par contre faire très attention à l'argument des fonctions agrégatives. L'oubli d'un **DISTINCT** peut rendre la requête fausse.

***Exemple** : donner le nombre de destinations desservies par chaque avion.*

```
SELECT NUMAV, COUNT(DISTINCT VILLE_ARR)
FROM VOL
GROUP BY NUMAV
```

*Si le **DISTINCT** est oublié, c'est le nombre de vols qui sera compté et la requête sera fausse.* □

***Exemple** : donnez le nombre de vols pour chaque ville de départ dans laquelle est localisé au moins un avion.*

```
SELECT VILLE_DEP, COUNT(DISTINCT NUMVOL)
FROM VOL, AVION
WHERE VILLE_DEP = LOCALISATION
GROUP BY VILLE_DEP
```

*La jointure réalisée duplique chacun des vols autant de fois qu'il y a d'avions localisés dans cette ville. Le mot clef **DISTINCT** permet de ne compter chaque vol qu'une seule fois.* □

Évidemment des requêtes complexes peuvent combiner partitionnements, jointures prédicatives et/ou imbriquées, opérations ensemblistes…

***Exemple** : pour chaque pilote assurant au moins un vol avec un Airbus et jamais avec un Boeing, donnez le nombre de vols assurés quel que soit le type d'appareil.*

```
SELECT NUMPIL, COUNT(NUMVOL)
FROM VOL
WHERE NUMPIL IN
            -- la sous-requête rend les pilotes conduisant un Airbus
            (SELECT NUMPIL
            FROM VOL, AVION
            WHERE AVION.NUMAV = VOL.NUMAV AND AVNOM LIKE 'A%')
            AND NUMPIL NOT IN
            -- la sous-requête rend les pilotes conduisant un Boeing
            (SELECT NUMPIL
            FROM VOL, AVION
            WHERE AVION.NUMAV = VOL.NUMAV AND AVNOM LIKE 'B%')
```

```
GROUP BY NUMPIL
```

Supposons que pour répondre à l'interrogation précédente, la requête suivante ait été formulée.

```
SELECT NUMPIL, COUNT(NUMVOL)
FROM  VOL, AVION
WHERE AVION.NUMAV = VOL.NUMAV AND AVNOM LIKE 'A%'
      AND NUMPIL NOT IN
            -- la sous-requête rend les pilotes conduisant un Boeing
            (SELECT NUMPIL
            FROM VOL, AVION
            WHERE AVION.NUMAV = VOL.NUMAV AND AVNOM LIKE 'B%')
GROUP BY NUMPIL
```

Du fait de la jointure réalisée dans le premier bloc, la fonction de comptage rend le nombre de vols réalisés avec un Airbus et non le résultat escompté ! □

3.1.14 – Conditions sur les classes de tuples d'un partitionnement

Des conditions de sélection peuvent être appliquées aux classes d'équivalence engendrées par la clause **GROUP BY**. De manière similaire aux conditions de sélection ou de jointure exprimées dans la clause **WHERE** qui permettent d'éliminer du résultat ***tout tuple*** ne les satisfaisant pas, les conditions exprimées dans la clause **HAVING** permettent au système d'éliminer ***toute classe de tuples*** ne respectant pas ces conditions. La clause **HAVING** doit suivre immédiatement la clause **GROUP BY**. Sa syntaxe est :

```
HAVING <liste_conditions>
```

Les conditions permettent de comparer une valeur calculée à partir de chaque classe d'équivalence de tuples à une constante ou à une autre valeur résultant d'une sous-requête.

Exemple *: donner le nombre de vols, s'il est supérieur à 5, par pilote.*

```
SELECT PILOTE.NUMPIL, NOMPIL, COUNT(NUMVOL)
FROM PILOTE, VOL
WHERE PILOTE.NUMPIL = VOL.NUMPIL
GROUP BY PILOTE.NUMPIL, NOMPIL
HAVING COUNT(NUMVOL) > 5
```
□

Exemple *: quel est le numéro des pilotes assurant plus de vols que le pilote numéro 100 ?*

```
SELECT PILOTE.NUMPIL
FROM PILOTE, VOL
WHERE PILOTE.NUMPIL = VOL.NUMPIL
GROUP BY PILOTE.NUMPIL
HAVING COUNT(NUMVOL) >
           (SELECT COUNT(NUMVOL)
           FROM VOL
           WHERE NUMPIL = 100)
```
□

Exemple *: quelles sont les villes les plus desservies ?*

```
SELECT VILLE_ARR
```

```
FROM  VOL
GROUP BY VILLE_ARR
HAVING COUNT(*) >= ALL
               (SELECT COUNT(*)
               FROM VOL
               GROUP BY VILLE_ARR)
```
□

Exemple *: quelles sont les villes à partir desquelles le nombre de villes desservies est le plus grand ?*

```
SELECT VILLE_DEP
FROM  VOL
GROUP BY VILLE_DEP
HAVING COUNT(DISTINCT VILLE_ARR) >= ALL
               (SELECT COUNT(DISTINCT VILLE_ARR)
               FROM VOL
               GROUP BY VILLE_DEP)
```
□

Même si les clauses **WHERE** et **HAVING** introduisent des conditions de sélection, elles sont différentes. Si la condition doit être vérifiée par chaque tuple d'une classe d'équivalence, il faut la spécifier dans le **WHERE**. Si la condition doit être vérifiée globalement pour la classe, elle doit être exprimée dans la clause **HAVING**. Par conséquent, la clause **HAVING** ***doit être réservée aux conditions sur le résultat d'une fonction agrégative***.

3.1.15 – Combinaison de résultats détaillés et agrégés

Lorsqu'une requête doit travailler à différents niveaux de granularité des données (niveau détaillé et niveau agrégé ou différents niveaux d'agrégation), il faut définir plusieurs blocs, chaque bloc calculant les résultats désirés à un niveau de granularité donné (Cf. paragraphe 3.1.7).
Ainsi il est possible, pour divers tuples parcourus par un premier bloc, de comparer les valeurs d'un attribut au résultat d'une fonction agrégative obtenu dans un bloc imbriqué.

Cependant, les possibilités d'imbrication de blocs, vues jusqu'à présent, ne permettent pas de rendre ***comme résultat de la requête des données calculées par divers blocs ne travaillant pas au même niveau de granularité des données***.
À partir de la norme SQL-2 et de la version 8 d'ORACLE, il est possible, pour résoudre le problème évoqué, d'imbriquer une requête dans les clauses **SELECT** et **FROM**.

Lorsqu'un bloc est imbriqué dans la clause **SELECT**, ***il doit retourner au plus un tuple*** ce qui limite évidemment le pouvoir d'expression de ce type de requête. Le résultat du bloc imbriqué est combiné aux résultats retournés par la requête. Si le bloc imbriqué calcule par exemple une valeur agrégée, elle peut être comparée aux valeurs réelles d'un attribut.

Exemple *: pour chaque pilote (numéro et nom), donnez l'écart entre son salaire et la moyenne des salaires.*

```
SELECT NUMPIL, NOMPIL,
       SALAIRE - (SELECT AVG(SALAIRE) FROM PILOTE)
FROM PILOTE
```
□

Un bloc imbriqué dans la clause **WHERE** peut rendre un ou plusieurs tuples. Si les résultats rendus par ce bloc imbriqué doivent aussi être les résultats de la requête, il faut utiliser des alias d'attributs dans le **SELECT** de la requête, alias qui sont introduits dans le bloc imbriqué.

***Exemple** : la requête de l'exemple précédent peut être formulée comme suit.*

```
SELECT NUMPIL, NOMPIL, SALAIRE - MOYENNE
FROM PILOTE, (SELECT AVG(SALAIRE) MOYENNE FROM PILOTE)
```
□

Bien sûr si le bloc imbriqué rend plusieurs tuples et qu'aucune jointure n'est exprimée, le système réalise le produit cartésien des sous-ensembles de tuples constituant les résultats intermédiaires de la requête. Cependant, avec ce type de formulation, ORACLE ***interdit d'utiliser, dans le bloc imbriqué, un alias de relation introduit dans le bloc de niveau supérieur***. L'expression des jointures doit se faire dans la clause **WHERE** du bloc de niveau supérieur et utiliser des alias pour les blocs imbriqués. En effet, tout comme les relations, les blocs imbriqués dans la clause **FROM** peuvent se voir attribuer un alias. Il est simplement mentionné après la parenthèse fermante du bloc. ***La condition à respecter est que les attributs de jointure doivent faire partie de la clause SELECT du bloc imbriqué.***

***Exemple** : pour chaque pilote, donnez l'écart entre son salaire et le salaire moyen des pilotes de sa ville de résidence.*

```
SELECT NUMPIL, NOMPIL, SALAIRE - MOYENNE_ADR
FROM PILOTE, (SELECT ADRESSE, AVG(SALAIRE) MOYENNE_ADR
              FROM PILOTE
              GROUP BY ADRESSE) APIL,
WHERE         PILOTE.ADRESSE = APIL.ADRESSE
```
□

***Exemple** : pour chaque ville de départ, donnez la capacité moyenne des avions effectuant des vols partant de cette ville et la capacité moyenne des avions localisés dans cette ville.*

```
SELECT VILLE_DEP, CAP_VILLE, CAP_LOC
FROM (SELECT VILLE_DEP, AVG(CAPACITE) CAP_VILLE
      FROM VOL, AVION
      WHERE VOL.NUMAV = AVION.NUMAV
      GROUP BY VILLE_DEP) DEP,
      (SELECT LOCALISATION, AVG(CAPACITE) CAP_LOC
      FROM AVION
      GROUP BY LOCALISATION) LOC
WHERE DEP.VILLE_DEP = LOC.LOCALISATION
```
□

En combinant des requêtes imbriquées dans la clause **FROM** dotées d'une clause de tri et la pseudo-colonne **ROWNUM**, il est possible d'extraire les `k` premiers tuples résultats d'une requête. De telles interrogations sont appelées des requêtes “ top-`k` ” ou “ down-`k` ”.

***Exemple** : quels sont les cinq plus petits numéros des avions localisés à Marseille.*

```
SELECT *
FROM (SELECT NUMAV
      FROM AVION
      WHERE LOCALISATION = 'MARSEILLE'
      ORDER BY NUMAV)
WHERE ROWNUM <= 5
```
□

Exemple *: donnez les dix pilotes effectuant le plus de vols.*

```
SELECT NUMPIL
FROM (SELECT NUMPIL, COUNT(*) NB_VOLS
      FROM VOL
      GROUP BY NUMPIL
      ORDER BY NB_VOLS DESC)
WHERE NUMROW <= 10
```

Remarquons que s'il existe un onzième pilote faisant autant de vols que le dixième, il ne fera pas partie du résultat de la requête. □

3.1.16 – Recherche dans une arborescence

L'interrogation de données structurées sous forme arborescente est une spécificité du SGBD ORACLE. Une telle structure de données est très fréquente dans le monde réel : nomenclatures, organigrammes ou plus généralement liens hiérarchiques, compositions…
Avant de détailler les recherches dans une arborescence, nous nous intéressons à la représentation relationnelle de structures hiérarchiques.

3.1.16.1- Représentation de hiérarchies

Il existe deux représentations possibles des hiérarchies selon que les " objets " associés par des liens hiérarchiques sont de même nature ou pas et/ou caractérisés par un sous-ensemble commun de propriétés.

- Représentation linéaire
  les " objets " liés n'ont pas la même sémantique, leurs propriétés caractéristiques diffèrent. Une représentation conceptuelle utilisant le modèle Entité-Association est illustrée par la figure 28.

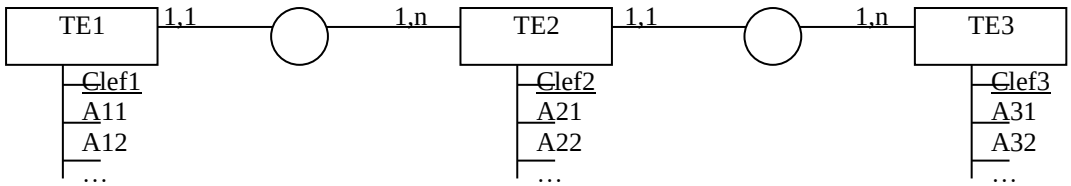


Figure 28 : représentation linéaire d'une hiérarchie

En traduisant un tel schéma dans le modèle relationnel, nous obtenons :
TE1(Clef1, A11, A12, …, ***Clef2***)
TE2(Clef2, A21, A22, …, ***Clef3***)
TE3(Clef3, A31, A32, …)

Remarque : la structure ainsi représentée doit être pérenne dans le temps i.e. le nombre de niveaux dans la hiérarchie et la description de chacun doivent rarement évoluer.

- Représentation récursive
  les " objets " liés ont une même sémantique et partagent nombre de propriétés. Dans le modèle Entité-Association, ils sont perçus comme des instances d'un même type d'entité et les liens hiérarchiques entre eux sont vus comme des instances d'un même type d'association unaire sur ce type d'entité, comme l'illustre la figure 29.

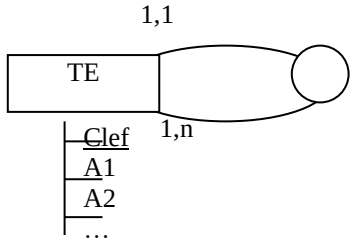


Figure 29 : représentation récursive d'une hiérarchie

En traduisant un tel schéma dans le modèle relationnel, nous obtenons :

TE (Clef, A1, A2, …, ***Clef_et***)

ou l'attribut Clef_et référence la clef primaire de la même relation TE.

Remarque : la structure ainsi représentée est facilement évolutive dans le temps quant à son nombre de niveaux qui n'est pas fixé par la représentation.

Exemple *: supposons que l'on souhaite intégrer dans la base des informations sur la répartition géographique des villes desservies. La figure 30 illustre l'organisation hiérarchique des valeurs des différentes localisations à répertorier.*

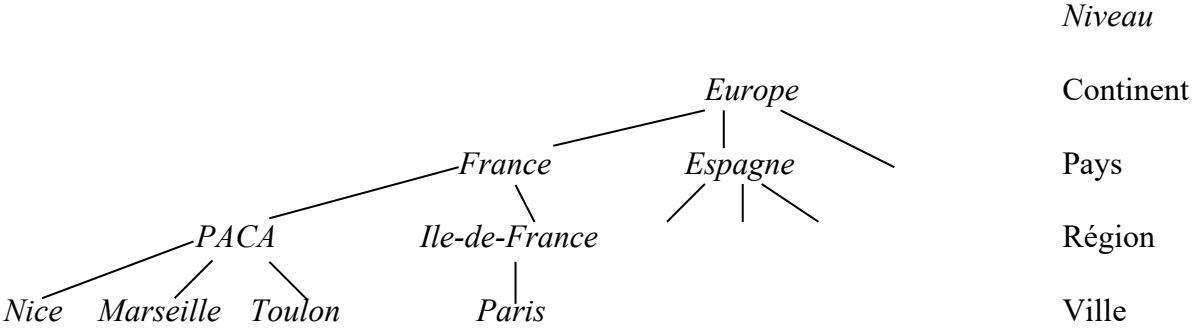


Figure 30 : hiérarchie des localisations

Supposons que pour tous les niveaux de la hiérarchie, les informations à conserver soient totalement différentes. Une représentation linéaire est alors adéquate : chaque niveau dans la hiérarchie est modélisé par un type d'entité et ces types d'entité sont reliés par des types d'association hiérarchiques.
La hiérarchie de la figure 30 donne l'instance du schéma suivant.

Continent

IdC	**NOMCONT**	**…**
1	EUROPE	…
…		

Pays

IdP	**NOMPAYS**	**…**	**IdC**
10	FRANCE	…	1
20	ESPAGNE	…	1
…			

Région

IdR	**NOMREG**	**…**	**IdP**
100	PACA	…	10
200	ÎLE DE FRANCE		10
…			

Ville

IdV	**NOMVILLE**	**…**	**IdR**
1000	NICE	…	100
2000	MARSEILLE	…	100
3000	TOULON	…	100
4000	PARIS	…	200
…			

Figure 31 : instance du schéma relationnel avec représentation linéaire

Considérons à présent une autre hypothèse : quel que soit le niveau de la hiérarchie des localisations, les " objets " ont la même sémantique car ils représentent tous des " lieux " géographiques et ils sont décrits par un même ensemble d'attributs (e.g. identifiant, nom du lieu, nombre d'habitants…). Dans ce contexte, la représentation récursive (Cf. figure 29) est la plus adaptée.
La relation LIEU est ajoutée au schéma initial de la base pour représenter de manière " plate " la structure arborescente illustrée par la figure 30. Cette relation a le schéma suivant :

*LIEU (IdL, NOM, **IdLGEN**).*

L'attribut IdLGEN est défini sur le même domaine que IdL. C'est donc une clef étrangère référençant la clef primaire de sa propre relation.
L'extension de cette relation, correspondant à la hiérarchie de la figure 30, est la suivante :

IdL	NOM	IdLGEN
1	EUROPE	NULL
10	FRANCE	1
20	ESPAGNE	1
100	PACA	10
200	ÎLE-DE-FRANCE	10
1000	NICE	100
2000	MARSEILLE	100
3000	TOULON	100
4000	PARIS	200
….		

Figure 32 : représentation relationnelle de la hiérarchie récursive des localisations

*Dans la relation LIEU, chaque tuple décrit un lieu spécifique ainsi que le lien hiérarchique qui le rattache à son lieu générique. Ainsi le lieu Marseille appartient à la région PACA, et le lieu PACA est rattaché au lieu France. Pour la valeur de lieu racine de la hiérarchie des localisations, la valeur du lieu d'appartenance est **NULL**.* □

3.1.16.2- Formulation de requêtes sur des structures hiérarchiques

Lorsqu'une hiérarchie est modélisée selon un schéma linéaire, une recherche dans cette arborescence s'exprime simplement en effectuant des jointures entre les différentes relations la représentant.
Si la hiérarchie est modélisée de manière récursive, i.e. via une seule relation se référençant elle-même, il est possible de l'interroger en formulant des auto-jointures comparant la clef primaire de la relation et sa clef étrangère. Mais cette technique peut être délicate à mettre en œuvre et surtout elle nécessite de connaître a priori le nombre de niveaux devant être explorés. Pour éviter ces problèmes, ORACLE offre la possibilité de formuler des recherches dans une arborescence décrite de manière " plate " par une ***seule relation***.
La syntaxe générale d'une requête de recherche dans une arborescence est indiquée ci-après.

```
SELECT <colonne1> [, <colonne2> …]
FROM  <table> [<alias>]
[WHERE <liste_conditions>]
CONNECT BY [PRIOR] <colonne1> = [PRIOR] <colonne2>
[AND  <condition_hierarchique>]
[START WITH <condition_depart>]
```

[**ORDER BY LEVEL**]

où

- dans la clause **FROM**, une ***seule relation doit être mentionnée*** car les opérations de jointure sous forme prédicative sont interdites dans les requêtes hiérarchiques mais il est possible de formuler des jointures sous forme imbriquée.

- la clause **START WITH** fixe le point de départ de la recherche dans l'arborescence. <condition_depart> est une liste de conditions de même type que les conditions de sélection classiques ou une condition de jointure exprimée de manière imbriquée. Si cette clause est omise, c'est toute l'arborescence qui sera examinée, i.e. tous les tuples de <table>.

- La clause **CONNECT BY PRIOR** fixe le sens de parcours de l'arborescence. Elle ne peut pas contenir une sous-requête. Le parcours peut se faire :
  - de haut en bas, i.e. qu'il correspond à l'exploration de sous-arbres ;
  - de bas en haut, i.e. qu'il correspond à la recherche du ou des " ancêtres ".

  Le parcours de haut en bas est spécifié en faisant précéder la colonne représentant un " fils " dans l'arborescence par le mot-clef **PRIOR** comme l'ordre des deux termes de l'égalité est indifférent, il est possible d'utiliser :

  CONNECT BY PRIOR <fils> = <pere>

  ou **CONNECT BY** <pere> = **PRIOR** <fils>

  Le parcours de bas en haut est spécifié en faisant précéder la colonne représentant un " père " par le mot-clef **PRIOR**. Il peut s'exprimer aussi de deux façons :

  CONNECT BY <fils> = **PRIOR** <pere>

  ou **CONNECT BY PRIOR** <pere> = <fils>

Remarque : ***avec la représentation relationnelle plate, le <fils> dans une structure hiérarchique correspond à la clef primaire de la relation, le <pere> à la clef étrangère.*** Lorsque la valeur de l'attribut <fils> est la racine de la hiérarchie, la valeur de l'attribut <pere> est une valeur nulle : il s'agit d'un cas exceptionnel où l'on admettra une valeur nulle pour une clef étrangère.

Le fonctionnement d'ORACLE lors de l'exécution d'une requête de recherche hiérarchique est le suivant [1] :

- tout d'abord, ORACLE sélectionne les tuples satisfaisant la clause **START WITH**. Si cette clause est omise, le tuple constituant la racine de la hiérarchie est sélectionné si la recherche est descendante ou tous les tuples correspondant aux feuilles de l'arbre sont considérés pour une recherche ascendante. Ce premier ensemble de tuples calculés constitue le ***niveau 1*** du résultat ;
- pour chaque tuple du niveau 1, ORACLE effectue la jointure spécifiée par la clause **CONNECT BY**, i.e. il sélectionne les tuples correspondant à ses descendants directs pour une recherche descendante ou correspondant à son ascendant direct pour une recherche ascendante. Les tuples ainsi obtenus constituent le ***niveau 2*** du résultat ;
- pour chaque tuple du niveau 2, ORACLE réalise la clause **CONNECT BY**, i.e. il calcule les descendants directs pour une recherche descendante ou l'ascendant direct pour une recherche ascendante de chacun des tuples obtenus au ***niveau 2*** du résultat. L'ensemble obtenu est considéré comme le ***niveau 3***. Ce processus est renouvelé autant de fois que nécessaire ;
- Si le premier bloc de la requête comporte une clause **WHERE**, ORACLE évalue les conditions de sélection ou de jointure imbriquée pour chaque tuple individuellement.

Si un tuple du niveau `i` ne satisfait pas cette condition, les tuples qui lui sont associés au niveau `i+1` sont quand même examinés. Ils ne sont éliminés du résultat que si ils ne vérifient pas eux-mêmes la condition donnée dans la clause **`WHERE`**.

- En effectuant une recherche hiérarchique, ORACLE utilise l'ordre en profondeur d'abord, i.e. il ne calcule pas le résultat niveau par niveau, mais pour le premier tuple du niveau 1, il cherche le premier tuple associé au niveau 2 puis, pour ce dernier, le tuple associé au niveau 3 et ainsi de suite. La figure 33 illustre cet ordre de parcours.

***Exemple** : donner la liste des localisations en France.*

```
SELECT IdL
FROM LIEU
CONNECT BY  PRIOR IdL = IdLgen
START WITH Nom = 'FRANCE'
```

*Supposons que l'extension de la relation LIEU soit celle donnée dans la figure 32. Le premier tuple sélectionné par la clause **`START WITH`** est celui d'identifiant 10. Au niveau 2, les fils du tuple n°10 sont calculés. Les tuples sélectionnés sont ceux d'identifiant 100 et 200. Puis les fils des tuples n° 100 et 200 sont recherchés et ainsi de suite. La figure 33 illustre le résultat calculé par ORACLE, les numéros encerclés donnent l'ordre dans lequel les tuples sont sélectionnés par le système.*

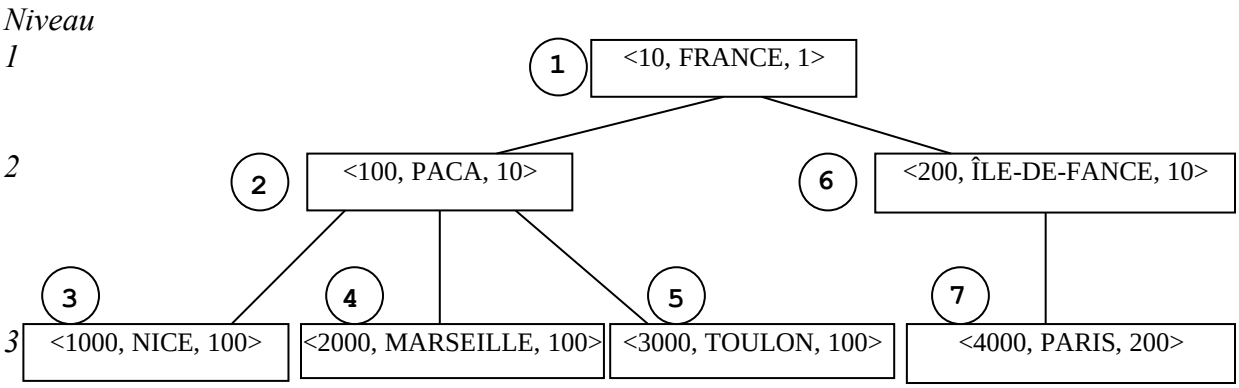
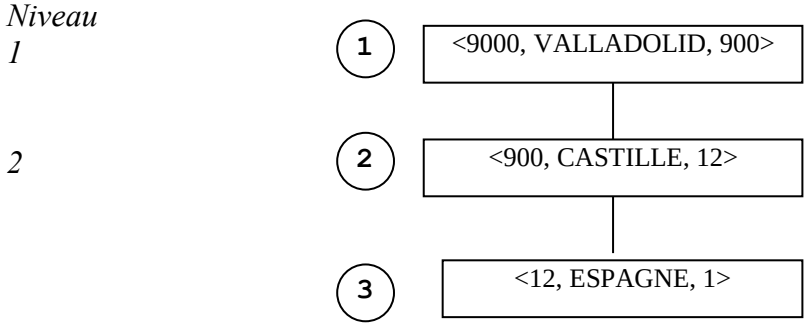


Figure 33 : recherche descendante en profondeur d'abord □

***Exemple** : où se situe (continent, pays, région) la ville de Valladolid ?*

```
SELECT NOM
FROM LIEU
CONNECT BY IdL = PRIOR  IdLGEN
START WITH NOM = 'VALLADOLID'
```

*En supposant qu'il n'existe qu'une seule ville de nom Valladolid, un seul tuple est sélectionné par la clause **`START WITH`**. La recherche étant ascendante, chaque niveau ne correspond qu'à un seul tuple. Dans la figure suivante, les numéros encerclés indiquent l'ordre de sélection des tuples.*



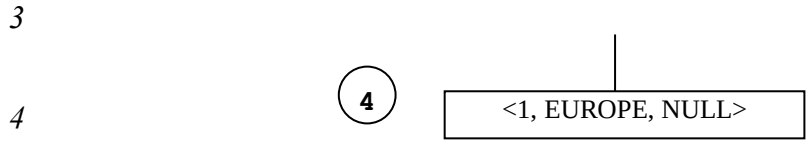


Figure 34 : recherche ascendante en profondeur d'abord □

- La clause facultative `[AND <condition_hierarchique>]` permet de spécifier une condition évaluée lors de la recherche arborescente si la clause **`START WITH`** est indiquée. Plus précisément, si la condition n'est pas satisfaite pour un nœud de la hiérarchie alors le sous-arbre dont ce nœud est racine n'est pas considéré dans le résultat lors d'un parcours de haut en bas ou tous les ascendants du nœud en question lors d'un parcours de bas en haut.

  Exemple *: donner la liste des lieux en France en dehors de la région PACA.*

```
SELECT IdL, NOM
FROM  LIEU
CONNECT BY  PRIOR IdL = IdLGEN
AND NOM <> 'PACA'
START WITH  NOM = 'FRANCE'
```

  Dès qu'un nœud ne vérifiant pas la condition donnée est trouvé, le sous-arbre dont ce nœud est racine est éliminé de la recherche, donc la région PACA et toutes les villes de cette région n'appartiendront pas au résultat. □

- ORACLE associe une numérotation aux nœuds d'une arborescence : c'est la notion de niveau décrite précédemment (pseudo-colonne **`LEVEL`**, Cf. paragraphe 2.7.2). Par exemple, pour une recherche hiérarchique descendante sans condition de départ, la racine de l'arbre est considérée comme de niveau 1, ses fils directs sont de niveau 2, et ainsi de suite. Cette numérotation peut être utilisée pour classer les résultats (clause **`ORDER BY LEVEL`**) et en donner une meilleure présentation. Celle-ci s'obtient en utilisant la fonction **`lpad`** (Cf. paragraphe 3.1.3.2) qui permet une indentation.

  Exemple *: pour obtenir la liste des lieux desservis en France mais avec présentation plus claire, nous pouvons formuler la requête comme suit :*

```
SELECT LPAD('-', 2*LEVEL,' ')|| NOM
FROM  LIEU
CONNECT BY PRIOR IdL = IdLGEN
START WITH NOM = 'FRANCE'
ORDER BY LEVEL
```

  `||` *est le symbole de concaténation de deux chaînes de caractères.*
  La fonction **`lpad`** *permet d'ajouter au caractère '-', un nombre d'espaces égal au double du niveau, devant chaque valeur de LIEUSPEC pour mieux visualiser la division géographique.* □

Évidemment, les recherches arborescentes peuvent être combinées aux autres formes d'extraction de données. Il est en particulier possible de formuler des jointures imbriquées permettant de spécifier les conditions hiérarchiques ou les conditions de départ de la recherche arborescente.

Exemple *: quelle est la hiérarchie de localisation à laquelle se rattache la ville d'arrivée du vol*

IT100.

```
SELECT LPAD('-', 2*LEVEL,' ')|| NOM
FROM LIEU
CONNECT BY IdL = PRIOR IdLGEN
START WITH NOM =
            ( SELECT VILLE_ARR
              FROM VOL
              WHERE NUMVOL = 'IT100')
ORDER BY LEVEL
```

□

***Exemple** : Donnez le nombre de vols au départ de chaque ville de la région PACA.*

```
SELECT VILLE_DEP, COUNT(*)
FROM VOL
WHERE VILLE_DEP IN
              (SELECT NOM
              FROM LIEU
              CONNECT BY PRIOR IdL = IdLGEN
              START WITH NOM = 'PACA')
GROUP BY VILLE_DEP
```

□

3.1.17 – Expression des divisions

Bien que fort rares dans un contexte opérationnel classique, les opérations de division consistent à rechercher les tuples qui se trouvent associés à ***tous les tuples*** d'un ensemble particulier appelé le diviseur. Elles expriment le quantificateur universel " tous les ".

***Exemple** : considérons les deux relations suivantes où les attributs B1 et B2 sont comptibles.*

r1	*A*	*B1*
	10	*1*
	20	*2*
	10	*2*
	10	*4*
	30	*4*
	20	*1*

r2	*B2*
	1
	2
	4

La division de r1 par r2 rend les valeurs de A1 qui, dans r1, sont associées à toutes les valeurs que prend B2 dans la relation r2. Ici le résultat est 10. □

SQL ne propose pas d'opérateur de division. Il est néanmoins possible d'exprimer cette opération mais de manière peu naturelle. Les deux techniques utilisées consistent à reformuler la requête différemment, on parle de " paraphrasage ". Elles ne sont pas systématiquement applicables (suivant l'interrogation posée).

3.1.17.1 - Divisions avec partitionnement et comptages

La première technique pour formuler une division a recours à l'opération de partitionnement et

à la fonction agrégative **COUNT**. Pour savoir si un tuple *t* est associé à tous les tuples du diviseur, l'idée est de compter d'une part le nombre de tuples associés à *t* et d'autre part le nombre de tuples du diviseur puis de comparer les valeurs obtenues.

***Exemple** : quels sont les numéros des pilotes qui conduisent tous les avions de la compagnie ? Pour l'exprimer en SQL avec la première technique exposée, cette requête est traduite de la manière suivante : " Quels sont les pilotes qui conduisent autant d'avions que la compagnie en possède ? ".*

```
SELECT NUMPIL
FROM VOL
GROUP BY NUMPIL
HAVING COUNT(DISTINCT  NUMAV) =
            ( SELECT COUNT(NUMAV)
              FROM AVION )
```

*Le comptage dans la clause **HAVING** permet pour chaque pilote de dénombrer les appareils conduits. L'oubli du **DISTINCT** rend la requête fausse (on compterait alors le nombre de vols assurés).* □

Cette technique de paraphrasage ne peut être utilisée que si les deux ensembles dénombrés sont parfaitement comparables.

***Exemple** : considérons la requête suivante : " Quels sont les pilotes conduisant tous les avions pilotés par le pilote n° 100 ? ". Cette division ne peut être exprimée par la technique présentée car il ne suffit pas que les pilotes conduisent autant d'avions que le pilote n° 100 mais il faut que les appareils soient bien les mêmes.* □

3.1.17.2 - Divisions avec double NOT EXISTS

La deuxième forme d'expression des divisions est plus délicate à formuler. Elle se base sur une double négation de la requête et utilise la clause **NOT EXISTS**. Au lieu d'exprimer qu'un tuple *t* doit être associé à tous les tuples du diviseur pour appartenir au résultat, on recherche *t* tel qu'il n'existe aucun tuple du diviseur qui ne lui soit pas associé. Rechercher des tuples associés ou non, signifie concrètement effectuer des opérations de jointure.

***Exemple** : quels sont les numéros des pilotes qui conduisent tous les avions de la compagnie ? Pour l'exprimer en SQL, cette requête est traduite par : " Quels sont les pilotes tels qu'il n'existe aucun avion de la compagnie qui ne soit pas conduit par ces pilotes ? ".*

```
SELECT NUMPIL
FROM PILOTE P
WHERE NOT EXISTS
            ( SELECT *
              FROM AVION A
              WHERE NOT EXISTS
                          ( SELECT *
                            FROM VOL
                            WHERE VOL.NUMAV = A.NUMAV AND
                            VOL.NUMPIL = P.NUMPIL
```

Détaillons l'exécution de cette requête. Pour chaque pilote examiné par le 1er bloc, les

différents tuples de AVION sont balayés au niveau du 2ème bloc et pour chaque avion, les conditions de jointure du 3ème bloc sont évaluées. Supposons que les trois relations de la base soient réduites aux tuples présentés dans la figure suivante.

PILOTE

NUMPIL	…
100	
101	
102	

AVION

NUMAV	…
10	
21	

VOL

NUMVOL	NUMPIL	NUMAV
IT100	100	10
IT200	100	10
AF214	101	21
AF321	102	10
AF036	101	10

Figure 35 : une instance de la base

Considérons le pilote n° 100. Parcourons les tuples de la relation AVION. Pour l'avion n° 10, le 3ème bloc retourne un résultat (le vol IT100), ***NOT EXISTS*** *est donc faux et l'avion n° 10 n'appartient pas au résultat du 2ème bloc. L'avion n° 21 n'étant jamais piloté par le pilote 100, le 3ème bloc ne rend aucun tuple, le* ***NOT EXISTS*** *associé est donc évalué à vrai. Le 2ème bloc rend donc un résultat non vide (l'avion n° 21) et donc le* ***NOT EXISTS*** *du 1er bloc est faux. Le pilote n° 100 n'est donc pas retenu dans le résultat de la requête.*
Pour le pilote 101 et l'avion 10, il existe un vol (AF214). Le 3ème bloc retourne un résultat, ***NOT EXISTS*** *est donc faux. Pour le même pilote et l'avion n° 21, le 3ème bloc restitue un tuple et à nouveau* ***NOT EXISTS*** *est faux. Le 2ème bloc rend donc un résultat vide ce qui fait que le* ***NOT EXISTS*** *du 1er bloc est évalué à vrai. Le pilote 101 fait donc partie du résultat de la requête. Le processus décrit est à nouveau exécuté pour le pilote 102 et comme il ne pilote pas l'avion 21, le 2ème bloc retourne une valeur et la condition du 1er bloc élimine ce pilote du résultat.* □

Exemple *: quels sont les numéros des pilotes qui conduisent tous les avions pilotés par le pilote n° 100 ? Pour l'exprimer en SQL, cette requête est traduite de la manière suivante : " quels sont les pilotes tels qu'il n'existe aucun avion piloté par le pilote n° 100 qui ne soit pas conduit par ces pilotes ? ".*

```
SELECT NUMPIL
FROM PILOTE  P
WHERE NOT EXISTS
            ( SELECT *
              FROM AVION A, VOL
              WHERE A.NUMAV = VOL.NUMAV AND NUMPIL = 100
                    AND NOT EXISTS
                            ( SELECT *
                              FROM VOL
                              WHERE VOL.NUMAV = A.NUMAV AND
                                    VOL.NUMPIL = P.NUMPIL
```

Le deuxième bloc imbriqué extrait les avions pilotés par le pilote n° 100. □

3.2 - Mises à jour des données

Les trois opérations de mises à jour des données sont la modification, l'insertion ou la suppression de tuples. Leur fonctionnement est détaillé dans ce paragraphe ainsi que l'utilisation de séquences pour mettre à jour des attributs à partir d'entiers générés automatiquement.

3.2.1 – Modification de tuples existants

L'opération de modification consiste à changer les valeurs d'un ou plusieurs attributs pour certain(s) tuple(s) ***existant*** déjà dans une table en spécifiant de nouvelles valeurs. Ces nouvelles valeurs peuvent être données explicitement dans la commande de modification, dérivées à partir d'un calcul horizontal ou obtenues à partir de la base de données par une sous-requête.

3.2.1.1 - Modification classique

Les opérations de modification qualifiées ici de classiques sont celles où le SGBD peut évaluer les nouvelles valeurs attribuées (dans la clause **SET**) directement, i.e. sans qu'une sous-requête soit calculée. La forme générale de la commande de modification est :

```
UPDATE <nom_table>
SET <nom_colonne1> = <expression1>[, <nom_colonne2> = <expr2> …]
[WHERE <condition_selection>]
```

où :

- `<nom_table>` est le nom de la relation à modifier. Il n'est pas possible de modifier plus d'une table dans un même ordre.
- La clause **WHERE** permet de sélectionner les tuples à modifier et peut inclure une sous-requête. En l'absence de clause **WHERE**, tous les tuples de la relation sont affectés par la modification.
- La clause **SET** spécifie les colonnes à modifier et les nouvelles valeurs qui leur sont affectées. Plus précisément, `<expression1>` peut être :
  - une constante respectant le type de l'attribut et la contrainte de domaine si elle existe (sinon la modification échoue) ;
  - une expression de calcul dont le résultat satisfait les conditions de type et éventuellement de domaine. Il est possible d'utiliser dans ce calcul la valeur avant modification de l'attribut en indiquant simplement son nom.

Exemple *: modifier l'adresse du pilote 101 qui devient Nice et sa prime portée à 500 €.*

```
UPDATE PILOTE
SET ADRESSE = 'NICE', PRIME = 500
WHERE NUMPIL = 101
```

□

Exemple *: augmenter de 10% les salaires des pilotes gagnant moins de 6.000 € et effectuant plus de 10 vols.*

```
UPDATE PILOTE
SET SALAIRE  = SALAIRE  * 1.1
WHERE SALAIRE < 6000 AND NUMPIL IN
            (SELECT NUMPIL
            FROM VOL
            GROUP BY NUMPIL HAVING COUNT(*) > 10)
```

□

3.2.1.2 - Modification avec sous-requête dans la clause SET

Il est possible d'effectuer des modifications en calculant les nouvelles valeurs affectées grâce à une requête sur la base. Cette requête peut porter sur la relation en cours de modification ou pas mais ***elle ne doit rendre qu'au plus un tuple résultat***, éventuellement réduit à un singleton, i.e. une seule valeur si un seul attribut est concerné par la modification. La syntaxe à respecter est la suivante.

```
UPDATE <nom_table>
SET (<nom_colonne1>[, <nom_colonne2>, nom_colonne3> …]) =
            (SELECT <col1>[, <col2>, col3> …] FROM … WHERE …)
WHERE <condition_selection>
```

Avec cette formulation, les attributs spécifiés dans la clause **SET** prennent les valeurs retournées par la requête. C'est l'ordre dans lequel sont mentionnés les attributs qui est utilisé pour l'affectation des nouvelles valeurs : `<nom_colonne1>` prend la valeur de `<col1>`, `<nom_colonne2>` prend celle de `<col2>`…

Exemple *: attribuer au pilote numéro 5 un salaire égal à celui du pilote numéro 10.*

```
UPDATE PILOTE
SET SALAIRE =
             (SELECT SALAIRE FROM PILOTE WHERE NUMPIL = 10)
WHERE NUMPIL = 5
```

□

Exemple *: affecter au vol IT100, les mêmes villes de départ et d'arrivée que celles du vol IT300.*

```
UPDATE VOL
SET (VILLE_DEP, VILLE_ARR) =
            (SELECT     VILLE_DEP, VILLE_ARR FROM VOL
            WHERE       NUMVOL = 'IT300')
WHERE NUMVOL = 'IT100'
```

□

Les divers types de modification peuvent être combinés au sein d'un même ordre **UPDATE**.

Exemple *: les pilotes 110 et 120 réaffectés à Paris se voient attribuer la prime maximale et ont une augmentation de salaire de 5%.*

```
UPDATE PILOTE
SET ADRESSE = 'PARIS',
    SALAIRE = SALAIRE * 1.05,
    PRIME = (SELECT MAX(PRIME) FROM PILOTE)
WHERE NUMPIL IN (110, 120)
```

□

3.2.2 – Insertion de nouveaux tuples

L'insertion consiste à créer de nouvelles lignes dans une table. Elle peut affecter toutes les colonnes d'une table (insertion totale) ou seulement un sous-ensemble de ces colonnes (insertion partielle), les autres prennent alors automatiquement des valeurs nulles. Évidemment si pour un attribut les contraintes **PRIMARY KEY** ou **NOT NULL** ont été spécifiées, une insertion partielle ne mentionnant pas l'attribut échouera par violation des contraintes.
D'autre part, il existe deux techniques d'insertion de nouveaux tuples dans une relation : soit les valeurs sont explicitement données par l'utilisateur (cas classique), soit elles sont calculées via l'exécution d'une requête.

3.2.2.1 - Insertion classique

Avec ce type d'insertion, l'utilisateur doit indiquer explicitement les valeurs pour le nouveau tuple créé. Les valeurs doivent être données dans l'ordre de définition des attributs lors de la création de la table (ordre d'affichage des attributs par **desc**) pour l'insertion totale ou dans l'ordre des attributs mentionnés. La syntaxe est la suivante.

```
INSERT INTO <nom_table> [(<liste_attributs>)]
VALUES (<valeur1>[, <valeur2> …])
```

Si l'argument <liste_attributs> est omis, les valeurs de ***tous*** les attributs doivent être données dans la clause **VALUES**. Soit toutes ces valeurs sont connues et indiquées, soit les valeurs inconnues sont spécifiées par le mot clef **NULL** (et non la chaîne 'NULL').

Exemple *: ajouter un nouveau pilote pour lequel toutes les informations sont connues.*

```
INSERT INTO PILOTE
VALUES (170, 'EINSTEIN', 'ALBERT', 'NICE', 5000, 500)
```

□

Exemple *: ajouter un nouveau pilote pour lequel sont connus uniquement le numéro et le nom.*

```
INSERT INTO PILOTE (NUMPIL, NOMPIL) VALUES (18, 'HECTOR')
```

La même insertion peut être réalisée par :

```
INSERT INTO PILOTE VALUES (18, 'HECTOR', NULL, NULL, NULL, NULL)
```

□

Pour des raisons de maintenance, il faut préférer l'insertion avec spécification du nom des attributs. Ainsi, si un nouvel attribut est ajouté ou si un autre, non valorisé lors de l'insertion, est supprimé, l'opération ne produira pas d'erreur.

3.2.2.2 - Insertion basée sur une requête

La deuxième forme d'insertion consiste à calculer les tuples à insérer en formulant une requête sur la base. Sa syntaxe est la suivante.

```
INSERT INTO <nom_table> [(<liste_attributs>,…)]
<requête>
```

Si une insertion totale est effectuée, il est inutile de préciser <liste_attributs>. Tous les tuples résultats de la requête sont insérés dans <nom_table> et il y a correspondance entre les colonnes mentionnées dans la clause **SELECT** de la requête et celles de <nom_table>.
Si une insertion partielle est effectuée, les attributs valorisés doivent être mentionnés dans <liste_attributs> et, selon leur ordre, ils prennent la valeur de l'attribut de même rang dans la clause **SELECT** de la requête. Il est aussi possible d'utiliser le mot clef **NULL** dans la clause **SELECT** lorsque la requête ne restitue pas de valeur pour un attribut.

Exemple *: insérer les pilotes de la table PILOTE dans la table NOUVPIL.*

```
INSERT INTO NOUVPIL(NUM, NOM, ADR)
SELECT NUMPIL, NOMPIL, ADRESSE
FROM PILOTE
```

Cette insertion peut aussi se formuler comme suit, en supposant que la table NOUVPIL a la même structure que PILOTE.

```
INSERT INTO NOUVPIL
SELECT NUMPIL, NOMPIL, NULL, ADRESSE, NULL, NULL, NULL
FROM PILOTE    □
```

***Exemple** : supposons que la relation STAT_VILLE soit créée et donne pour chaque ville de départ, le nombre de vols et la capacité moyenne des avions. Son schéma est le suivant : STAT_VILLE (VILLE, NB_VOL, MOY_CAP). Les tuples peuvent être insérés dans cette relation à partir de l'extension de la base par l'ordre suivant.*

```
INSERT INTO STAT_VILLE
SELECT VILLE_DEP, COUNT(*), AVG(CAPACITE)
FROM VOL, AVION
WHERE VOL.NUMAV = AVION.NUMAV
GROUP BY VILLE_DEP                                    □
```

3.2.3 – Suppression de tuples existants

La suppression de données consiste à supprimer un ou plusieurs tuples dans une table. La commande correspondante est **DELETE** dont la syntaxe est donnée ci-après.

```
DELETE FROM <nom_table> WHERE <condition>
```

La condition permet de sélectionner les lignes de la table à supprimer. Si la clause **WHERE** est absente, toutes les lignes sont supprimées. La condition peut être une combinaison booléenne de conditions ou contenir une ou plusieurs sous-requêtes.
La commande de suppression de tuples peut échouer si elle opère sur une relation dont la clef primaire est référencée par une clef étrangère et que la propagation de l'élimination de contraintes n'a pas été spécifiée (utilisation de **ON DELETE CASCADE** à la création de la relation, Cf. paragraphe 2.2.1).

***Exemple** : supprimer tous les vols qui datent d'avant le 20 février 2003.*

```
DELETE FROM VOL WHERE DATE_VOL < '20/02/03'            □
```

***Exemple** : supprimer tous les vols assurés par le(s) pilote(s) Dupont.*

```
DELETE FROM VOL WHERE NUMPIL IN
(SELECT  NUMPIL FROM PILOTE WHERE NOMPIL = 'DUPONT')   □
```

3.2.4 – Utilisation de séquences

Les séquences créées pour générer des entiers uniques (Cf. paragraphe 2.4) sont soumises à diverses restrictions concernant les opérations de mise à jour. Les pseudo-colonnes **CURRVAL** et **NEXTVAL** peuvent être employées lors des insertions soit dans la clause **VALUES** d'un ordre **INSERT**, soit dans le **SELECT** imbriqué d'une requête d'un **INSERT**. Ils peuvent aussi apparaître dans la clause **SET** d'un ordre **UPDATE**, lors des modifications.
Mais ils ne sont pas autorisées dans les blocs imbriqués des ordres **UPDATE** et **DELETE**. Ils ne peuvent pas apparaître dans une clause **WHERE**.

Exemple *: une séquence ID_AVION permettant de générer des valeurs pour NUMAV a été créée. Elle peut être utilisée de la manière suivante pour créer un nouvel appareil.*

```
INSERT INTO AVION
VALUES (ID_AVION.NEXTVAL, 'A340', 600, 'PARIS')
```

Si de nouveaux vols sont créés pour cet avion ou si des vols existants lui sont affectés, la pseudo-colonne **CURRVAL** *est utilisée dans les ordres* **INSERT** *ou* **UPDATE**.

```
INSERT INTO VOL (NUMVOL, NUMAV, NUMPIL, VILLE_DEP, VILLE_ARR)
VALUES ('IT299', ID_AVION.CURRVAL, 100, 'PARIS', 'MARSEILLE')
/
UPDATE VOL
SET NUMAV = ID_AVION.CURRVAL
WHERE NUMAV = 20 AND VILLE_DEP = 'PARIS'
```

□

Exemple *: l'ordre suivant est valide.*

```
INSERT INTO AVION (AVNUM)
SELECT ID_AVION.NEXTVAL FROM DUAL
```

□

Chapitre 4 - SQL comme langage de contrôle des données

La dernière facette de SQL concerne la sécurité des données, i.e. leur confidentialité, leur pérennité et leur intégrité.
La confidentialité est, dans un SGBD professionnel, assurée par plusieurs mécanismes : *(i)* la gestion des rôles et des utilisateurs, similaires aux concepts de groupes et d'utilisateurs des système d'exploitation (Cf. paragraphe 4.2.1), *(ii)* l'attribution de privilèges à ces rôles et ces utilisateurs leur donnant le droit d'effectuer certaines opérations et spécifiant la portée de ces opérations (Cf. paragraphe 4.2.2) et *(iii)* la définition de filtres appelés vues permettant de masquer certaines données confidentielles aux utilisateurs non autorisés ou de mettre en œuvre des contrôles d'intégrité (Cf. paragraphe 4.3).
Assurer la pérennité et l'intégrité des données est une fonctionnalité fondamentale des SGBD qui doivent garantir à la fois la fiabilité des données et leur accessibilité y compris en cas d'incident, anomalie ou avarie. Pour répondre à ces objectifs des mécanismes de gestion de transactions sont développés (Cf. paragraphe 4.1).

4.1 - Gestion des transactions

Comme tout SGBD, ORACLE est un ***système transactionnel***, i.e. il gère les accès concurrents de multiples utilisateurs à des données fréquemment mises à jour et soumises généralement à de nombreuses contraintes de cohérence. ***Une base de données est dite cohérente à un instant t si toutes les contraintes d'intégrité spécifiées sont satisfaites par l'instance courante de la base***. Un SGBD doit permettre de spécifier les contraintes nécessaires et de rejeter les mises à jour violant ces contraintes.
Dans le cas le plus simple, comme une contrainte de domaine sur un attribut, une insertion de tuple et une modification de la valeur de l'attribut ne sont pas acceptées si elles ne respectent pas la contrainte. Il en est de même pour toutes les contraintes d'intégrité structurelles.
Cependant il existe des contraintes applicatives qui ne peuvent pas être vérifiées après une seule opération de mise à jour car elles sont systématiquement violées. Elles doivent être vérifiées après une série d'opérations de mises à jour.

Exemple *: supposons qu'à des fins de statistiques un attribut calculé NB_VOL soit ajouté à la relation AVION. La contrainte d'intégrité dynamique associée spécifie que, pour chaque avion, l'attribut NB_VOL prend comme valeur le résultat du comptage des vols effectués par l'avion considéré.*
Si un nouveau vol est inséré pour un avion donné, la contrainte ne peut être satisfaite qu'après la modification de l'attribut NB_VOL. Si la contrainte est vérifiée avant l'insertion du vol , elle ne pourra l'être à nouveau qu'après l'insertion du vol et la modification du nombre de vols. □

Une transaction est une séquence d'opérations manipulant les données et qui forme un tout indivisible : toutes les opérations sont réalisées ou aucune (" tout ou rien "). Ainsi une transaction est considérée comme un traitement atomique qu'il est impossible d'effectuer partiellement. Exécutée sur une base cohérente, une transaction laisse, par définition, la base dans un état cohérent. Soit la transaction se déroule et s'achève correctement (ses opérations prennent effet sur la base), soit un incident a lieu avant la terminaison de la transaction et aucune opération (même celles réalisées avec succès) ne doit avoir d'effet sur la base. Les incidents pouvant se produire sont matériels ou logiciels.

Exemple *: considérons la contrainte d'intégrité dynamique introduite dans l'exemple précédent*

" l'attribut NB_VOL prend comme valeur le résultat du comptage des vols de l'avion considéré". D'autre part, tous les vols Marseille-Toulouse effectués par l'avion numéro 10 sont supprimés. Pour prendre en compte ces modifications, les opérations de mise à jour suivantes doivent être réalisées.

```
UPDATE AVION
SET NB_VOL = NB_VOL - (SELECT COUNT(*) FROM VOL
                       WHERE NUMAV = 10 AND
                       VILLE_DEP = 'MARSEILLE'
                       AND VILLE_ARR = 'TOULOUSE')
WHERE NUMAV = 10
/
DELETE FROM VOL
WHERE NUMAV = 10 AND VILLE_DEP = 'MARSEILLE'
AND VILLE_ARR = 'TOULOUSE'
```

Pour que la base reste cohérente, il faut que les deux opérations soient entièrement réalisées ou aucune. En effet dans le cas contraire (par exemple si un incident intervient juste avant la suppression des tuples), la contrainte d'intégrité sera forcément violée. □

Les transactions vérifient les propriétés ACID suivantes :

- Atomicité (*Atomicity*) : les opérations d'une transaction sont considérées comme un tout indivisible. Soit elles sont toutes exécutées complètement (en cas de succès), soit aucune ne doit avoir d'effet (en cas d'échec) ;
- Cohérence (*Consistency*) : toute transaction effectuée sur une base cohérente restitue une base cohérente (toutes les contraintes spécifiées sont respectées) ;
- Indépendance (*Isolation*) : les opérations d'une transaction en cours d'exécution sont transparentes pour tout autre utilisateur ou transaction, comme si la transaction considérée était seule à travailler sur la base de données ;
- Permanence (*Durability*) : si une transaction s'est terminée correctement, ses effets sur la base sont pris en compte quels que soient les incidents ultérieurs.

Les commandes **COMMIT** et **ROLLBACK** de SQL permettent de contrôler la validation et l'annulation des transactions :

- **COMMIT** : valide la transaction en cours. Les opérations de mise à jour des données de la base deviennent définitives ;
- **ROLLBACK** : annule la transaction en cours. La base est restaurée dans son état au début de la transaction.

Ces ordres peuvent être explicitement exécutés par un utilisateur ou un programme d'application mais ils peuvent également être implicites : ORACLE les effectue de manière transparente pour l'utilisateur. ***Par exemple, pour tout ordre du LDD (comme la création d'une relation), un COMMIT implicite est réalisé***. C'est également le cas, lors de la sortie de SQL*PLUS par **exit**.
Pour permettre la gestion de transactions, les SGBD tiennent à jour un journal (maintenu dans une mémoire stable (disque ou bande)) dans lequel toutes les opérations et leurs arguments sont écrits. En cas de défaillance, ce journal est utilisé pour effectuer une reprise sur panne : les opérations des transactions en cours sont " défaites " et la base est restaurée dans son dernier état cohérent.

Pour assurer la cohérence des données, ORACLE pose automatiquement des verrous en adoptant les principes suivants :

- une écriture bloque automatiquement une autre écriture. Si une transaction T2 doit modifier un " objet " sur lequel une transaction T1 a un accès en écriture alors elle est mise en attente jusqu'à ce que T1 s'achève ;
- une lecture ne bloque pas une écriture. Si une transaction T2 doit modifier un " objet " pour lequel une transaction T1 a un accès en lecture alors T2 obtient l'accès en écriture ;
- une écriture ne bloque pas une lecture. Si une transaction T2 effectue une requête sur un " objet " pour lequel une transaction T1 a un accès en écriture alors T2 accède à la dernière version validée de l'objet.

Une modification interactive de la base (**INSERT**, **UPDATE** ou **DELETE**) ne sera validée que par la commande **COMMIT**, ou lors de la sortie normale de SQL*PLUS par la commande **exit**.

Il est possible de se mettre en mode " validation automatique ", dans ce cas, la validation est effectuée automatiquement après chaque commande SQL de mise à jour de données. Pour se mettre dans ce mode, il faut utiliser l'une des commandes suivantes :
SET AUTOCOMMIT IMMEDIATE ou **SET AUTOCOMMIT ON**
L'annulation de ce mode se fait avec la commande : **SET AUTOCOMMIT OFF**

Seules les trois opérations de mise à jour des données (**INSERT**, **UPDATE**, **DELETE**) peuvent être " défaites " par l'ordre **ROLLBACK** et en aucun cas les modifications structurelles d'une base (telles que **CREATE**, **ALTER**, …).

4.2 - Gestion des utilisateurs et de leurs privilèges

Tous les utilisateurs d'une base de données doivent être recensés et classés dans un groupe d'utilisateurs ou rôle. Ils se voient alors attribuer un loggin et un mot de passe pour accéder à ORACLE puis un certain nombre de privilèges. Sont traités dans ce paragraphe uniquement les droits relatifs aux ordres SQL[25].

4.2.1 – Création et suppression de rôles et d'utilisateurs

La création de rôle ne peut être réalisée que par les personnes ayant ce privilège : l'administrateur de la base de données et le responsable de la sécurité. L'ordre associé est le suivant.

```
CREATE ROLE <nom_role> [IDENTIFIED BY <mot_de_passe>]
```

Le nom d'un rôle doit être unique parmi tous les noms de rôles et d'utilisateurs. Il existe plusieurs rôles prédéfinis[26] par ORACLE mais il est recommandé de créer ses propres rôles. Si la clause **IDENTIFIED BY** est omise, aucun mot de passe n'est demandé à un utilisateur se connectant avec ce rôle.

***Exemple** : deux nouveaux rôles sont définis de la manière suivante.*

```
CREATE ROLE DEVELOPPEUR IDENTIFIED BY MDPDEVELOP
```

[25] D'autres privilèges sont nécessaires pour des développements en PL/SQL.
[26] Les anciens rôles prédéfinis **CONNECT**, **RESOURCE** et **DBA** sont toujours fournis pour des raisons de compatibilité avec les anciennes versions mais il est recommandé de ne pas en faire usage.

```
/
CREATE ROLE OPSAISIE IDENTIFIED BY MDPSAISIE
```
□

Le mot de passe associé à un rôle peut être ajouté, modifié ou supprimé (dans ce cas, la clause optionnelle **IDENTIFIED BY** est omise) par la commande suivante.

```
ALTER ROLE <nom_role> [IDENTIFIED BY <nouveau_mot_de_passe>]
```

Un rôle peut être supprimé par : **DROP ROLE** <nom_role>

Des utilisateurs peuvent être créés. L'ordre donné ci-après permet d'effectuer une telle création. La liste des clauses indiquées n'est pas complète. D'autres paramètres peuvent être fixés par l'administrateur ORACLE. Une liste exhaustive est proposée dans [1].

```
CREATE USER <nom_utilisateur> [IDENTIFIED BY <mot_de_passe>]
DEFAULT TABLESPACE <nom_table_space>
QUOTA <taille>
PROFILE <nom_profil>
```

Le nom d'un utilisateur doit être unique parmi tous les noms de rôles et d'utilisateurs. Si la clause **IDENTIFIED BY** est omise, aucun mot de passe n'est demandé à un utilisateur lorsqu'il se connecte. <nom_table_space> est le nom de l'espace de stockage des objets créés par l'utilisateur et <taille> donne la taille dans cet espace qui peut être utilisée. Le profil des utilisateurs décrit les ressources que l'administrateur met à leur disposition (nombre de tentatives pour se connecter à ORACLE, nombre de sessions concurrentes, limite de CPU…).

***Exemple** : un nouvel utilisateur est défini de la manière suivante.*

```
CREATE USER DUPONT IDENTIFIED BY MDPDUPONT
DEFAULT TABLESPACE table_space_utilisateur
QUOTA 5M
```
□

Le mot de passe associé à un utilisateur peut être ajouté, modifié ou supprimé par la commande suivante.

```
ALTER USER <nom_utilisateur> [IDENTIFIED BY mot_de_passe>]
```

Un rôle peut être supprimé par :

```
DROP USER <nom_utilisateur>
```

4.2.2 – Attribution et suppression de privilèges

Pour pouvoir se connecter sous ORACLE et utiliser les diverses commandes de SQL*PLUS, tout utilisateur doit avoir les privilèges ou droits requis. Pour attribuer des privilèges, la commande **GRANT** est utilisée. Elle permet :

- de donner des privilèges système à des utilisateurs ou des rôles. Ces privilèges permettent la création de session (pour se connecter à ORACLE), la création de table ou généralement des " objets " ORACLE.
- de conférer des rôles à des utilisateurs ou à d'autres rôles ;
- d'attribuer des privilèges sur les " objets " ORACLE.

4.2.2.1 - Attribution et suppression de privilèges système

Les privilèges système sont attribués par l'administrateur de la base ou le responsable de sa sécurité à des rôles et/ou des utilisateurs par la commande **GRANT**. Sa syntaxe est la suivante.

```
GRANT <systeme_privileges | ALL [PRIVILEGES]>
TO <liste_roles_utilisateurs | PUBLIC>
[WITH ADMIN OPTION]
```

où :

- <systeme_privileges> est une liste de privilèges système incluant :
  - **CREATE ROLE** pour créer des rôles,
  - **CREATE SEQUENCE** pour créer des séquences,
  - **CREATE SESSION** pour accéder au SGBD,
  - **CREATE SYNONYM** pour définir des synonymes,
  - **CREATE PUBLIC SYNONYM** pour créer des synonymes accessibles à tous les utilisateurs,
  - **CREATE TABLE** pour créer des relations,
  - **CREATE USER** pour créer des utilisateurs,
  - **CREATE VIEW** pour définir des vues.

- <**ALL** [**PRIVILEGES**]> est indiqué si tous les privilèges système sont attribués.

- La clause **TO** est utilisée pour donner la liste des rôles et des utilisateurs bénéficiant des privilèges indiqués. Si le mot clef **PUBLIC** est spécifié, les privilèges sont attribués à tous les utilisateurs et rôles.

- La clause **WITH ADMIN OPTION** permet aux utilisateurs ou rôles bénéficiant de l'attribution de privilèges système de les attribuer à d'autres utilisateurs ou rôles mais aussi de supprimer cette attribution. Pour éliminer l'effet de **WITH ADMIN OPTION**, il faut supprimer les privilèges puis les ré-accorder sans cette clause.

***Exemple** : accordez les privilèges de connexion et de création des divers objets ORACLE au rôle DEVELOPPEUR. Le rôle OPSAISIE n'aura que le privilège système de connexion.*

```
GRANT CREATE SESSION, CREATE TABLE, CREATE VIEW, CREATE SEQUENCE
TO DEVELOPPEUR
/
GRANT CREATE SESSION
TO OPSAISIE
```
□

***Exemple** : l'utilisateur ADM_APPLI se voit attribuer tous les privilèges système avec la possibilité de les conférer aux autres utilisateurs.*

```
GRANT ALL PRIVILEGES
TO ADM_APPLI
WITH ADMIN OPTION
```
□

Un utilisateur d'ORACLE peut à tout moment se voir démuni de certains privilèges. La commande correspondante est donnée ci-après.

```
REVOKE <liste_privileges_systeme>
```

```
FROM  <liste_roles_utilisateurs>
```

Exemple *: le rôle DEVELOPPEUR perd le privilège système de création de séquence.*

```
REVOKE CREATE SEQUENCE
TO DEVELOPPEUR
```
□

4.2.2.2 - Attribution et suppression de privilèges sur les objets ORACLE

Toutes les tables, vues, synonymes et séquences ne sont initialement accessibles que par l'utilisateur qui les a créés. Néanmoins, le partage de ces objets est nécessaire. Pour rendre ce partage possible, le SGBD permet au propriétaire d'un objet d'accorder des droits sur cet objet à d'autres utilisateurs. La commande d'attribution des droits a la syntaxe suivante.

```
GRANT <liste_droits>
ON <nom_composant>
TO <liste_roles_utilisateurs>
[WITH GRANT OPTION]
```

`<liste_droits>` est une série de droits attribués, séparés par des virgules. Les droits possibles sont :

- **`SELECT`** pour formuler des requêtes d'interrogation des données. `<nom_composant>` peut être un nom de relation, de vue ou de séquence ;
- **`INSERT`** pour effectuer l'insertion de tuples dans une table. `<nom_composant>` est soit un nom de relation, soit un nom de vue ;
- **`UPDATE`** `[<nom_colonne1, <nom_colonne2>…]` pour pouvoir modifier des tuples. Si des noms d'attributs sont précisés, ils sont les seuls à pouvoir être mis à jour par l'utilisateur. `<nom_composant>` est un nom de relation ou de vue ;
- **`DELETE`** pour supprimer des tuples dans une table. `<nom_composant>` est un nom de relation ou de vue ;
- **`ALTER`** pour effectuer des modifications structurelles sur une relation ou une séquence ;
- **`INDEX`** pour créer des index sur une relation ;
- **`REFERENCES`** pour créer des contraintes de référence sur une relation ;
- **`ALL`** `[`**`PRIVILEGES`**`]` pour réaliser toutes les opérations possibles sur l'objet spécifié.

`<liste_roles_utilisateurs>` est la liste des utilisateurs et des rôles auxquels sont accordés les privilèges. Le mot clef **`PUBLIC`** peut être utilisé pour désigner tous les utilisateurs.

Tout utilisateur qui reçoit un privilège sur une table avec l'option **`WITH GRANT OPTION`** a en plus le droit de l'accorder à un autre utilisateur.

Exemple *: accordez tous les privilèges possibles sur la relation VOL au rôle DEVELOPPEUR. avec le droit de les transmettre. Donnez au rôle OPSAISIE les privilège d'interrogation et de mise à jour de cette relation.*

```
GRANT ALL PRIVILEGES
ON VOL
TO DEVELOPPEUR
WITH GRANT OPTION
/
GRANT SELECT, INSERT, UPDATE, DELETE
ON VOL
```

```
TO OPSAISIE                                                    □
```

Exemple *: accordez au rôle OPSAISIE et à l'utilisateur Dupont, le droit de consulter la relation PILOTE et de mettre à jour l'attribut ADRESSE uniquement.*

```
GRANT SELECT, UPDATE(ADRESSE)
ON PILOTE
TO OPSAISIE, DUPONT                                            □
```

Un droit ne peut être retiré que par l'utilisateur qui l'a accordé ou bien qui a le privilège d'administration. La syntaxe de cette commande est la suivante :

```
REVOKE <liste_privileges>
ON <nom_composant>
FROM <liste_roles_utilisateurs>
```

Exemple *: retirez à l'utilisateur Dupont le droit de modification de la relation PILOTE.*

```
REVOKE UPDATE
ON PILOTE
FROM DUPONT                                                    □
```

Exemple *: interdire toute opération à tous les utilisateurs sur la table avion.*

```
REVOKE ALL
ON AVION
FROM PUBLIC                                                    □
```

4.2.2.2 - Attribution et suppression de rôles

Les rôles créés peuvent être attribués à des utilisateurs ou à d'autres rôles qui bénéficient alors des mêmes privilèges. Un utilisateur et un rôle peuvent avoir plusieurs autres rôles. La syntaxe utilisée est la suivante.

```
GRANT    <liste_roles_attribues>
TO       <liste_roles_utilisateurs>
[WITH ADMIN OPTION]
```

Un rôle peut être attribué à tous les utilisateurs, en remplaçant `<liste_roles_utilisateurs>` par **`PUBLIC`**. La clause **`WITH ADMIN OPTION`** permet aux utilisateurs ou rôles bénéficiant de l'attribution de rôles d'attribuer `<liste_roles_attribues>` à d'autres utilisateurs ou rôles, de supprimer cette attribution, de modifier le rôle ou de le détruire. Pour supprimer ces possibilités, il faut révoquer les rôles attribués puis les attribuer à nouveau sans la clause **`WITH ADMIN OPTION`**.

Exemple *: attribuez le rôle OPSAISIE à l'utilisateur Dupont et au rôle DEVELOPPEUR.*

```
GRANT OPSAISIE
TO DEVELOPPEUR, DUPONT                                         □
```

Les rôles attribués à des utilisateurs ou à d'autres rôles sont supprimés par la commande suivante.

```
REVOKE <liste_roles_attribues>
FROM <liste_roles_utilisateurs>
```

4.3 - Gestion de vues

Dans le développement d'applications bases de données, il est souvent nécessaire de pouvoir manipuler le résultat d'une requête comme on manipule une relation de la base. Même si les différences sont notables, le point commun entre ces deux types d'objets gérés par un SGBD est qu'ils peuvent être perçus, par un utilisateur final, d'une manière uniforme et très simple, i.e. comme un ensemble de tuples.
SQL permet, à travers le mécanisme de vue, de spécifier une requête d'interrogation (dite requête de définition de la vue) et de considérer son résultat comme une table.
Dans les paragraphes suivants, la syntaxe des ordres de création de vue est donnée, puis sont détaillés et illustrés leurs différents usages.

4.3.1 – Création de vues

Une vue permet de consulter et manipuler des données stockées dans la base et/ou calculables à partir de données stockées. Les données originales, i.e. celles qui sont effectivement stockées, le sont dans une ou de plusieurs tables ou relations dites de base. Une vue est aussi perçue comme une table et elle peut être utilisée exactement de la même façon qu'une relation de base mais elle est considérée comme une ***table virtuelle*** car ses données (redondantes par rapport à la base originale) ne sont pas physiquement stockées. Plus précisément, une vue est définie sous forme d'une requête d'interrogation et c'est cette requête qui est conservée dans le dictionnaire de la base (plus précisément cette requête est la valeur de l'attribut TEXT des tables système ALL_VIEWS et USER_VIEWS, Cf. paragraphe 2.7.1).

Dans sa forme la plus simple (spécification des seuls éléments obligatoires), la commande de création d'une vue est la suivante :

```
CREATE VIEW <nom_vue>
AS
<definition_requete>
```

où
`<nom_vue>` respecte les mêmes règles que tout identificateur SQL*PLUS (Cf. introduction du chapitre 2) et `<requete_definition>` est toute requête d'interrogation **SELECT** … **FROM** … aussi complexe soit elle mais elle ne peut pas utiliser les pseudo-colonnes **CURRVAL** et **NEXTVAL**.
Toute vue étant considérée comme une table, elle peut être utilisée dans la définition d'une autre vue. Elle est alors mentionnée dans la clause **FROM** de `<requete_definition>`.

Exemple *: la vue suivante répertorie les lignes desservies par la compagnie.*

```
CREATE VIEW LIGNE
AS
SELECT DISTINCT VILLE_DEP, VILLE_ARR
FROM VOL
```
□

Si l'ordre de création de vue est accepté, la vue est créée. Le schéma de cette table virtuelle (i.e. ses attributs) est constitué par tous les éléments (attributs ou expressions de calcul nommées via des alias) mentionnés dans la clause **SELECT** de la requête de définition. L'extension de la vue est constituée de tous les tuples résultats de la requête de définition au moment où cette requête

est calculée, i.e. ***à chaque nouvelle utilisation de la vue***.

L'ordre complet de création de vue est donné ci-après.

```
CREATE [OR REPLACE] [FORCE | NO FORCE]
VIEW <nom_vue> [(alias1, alias2…)]
AS
<definition_requete>
[WITH CHECK OPTION | WITH READ ONLY]
```

Détaillons les différentes options de l'ordre **CREATE VIEW**.

OR REPLACE Si la requête de définition d'une vue existante doit être modifiée, il faut la détruire puis la recréer. En indiquant **CREATE OR REPLACE**, la commande de création de vue peut être exécutée plusieurs fois sans avoir à supprimer la vue au préalable et sans avoir à ré-attribuer les privilèges accordés sur cette vue.

`(alias1, alias2…)` si une liste d'alias est précisée après le nom de la vue, ces alias seront les noms des attributs de la vue. Ils correspondent, deux à deux et dans l'ordre de leur apparition, avec les attributs ou expressions indiqués dans le **SELECT** de la requête définissant la vue[27]. Si la liste d'alias est omise, les attributs de la vue ont le même nom que ceux des attributs indiqués dans le **SELECT** de la requête. Si la requête de définition inclut des expressions de calcul dans la clause **SELECT**, alors soit il faut leur attribuer un alias dans le **SELECT**, soit une liste d'alias doit être spécifiée pour la vue. Si le caractère `*` est utilisé dans la clause **SELECT**, tous les attributs de la ou des relations sont utilisés pour définir la vue. Cependant, si un attribut est ultérieurement ajouté à une de ces relations de base, il ne sera pas pris en compte dans la vue et, s'il est nécessaire, la vue devra être recréée.

FORCE cette clause optionnelle autorise la création d'une vue même si les relations de base n'existent pas encore dans la base ou même si le créateur de la vue n'a pas les privilèges adéquats sur ces relations de base. Bien sûr à la première utilisation de la vue, il faut que les relations de base nécessaires soient présentes et que les privilèges adéquats soient attribués à l'utilisateur.

NO FORCE en indiquant cette clause ou en ne précisant rien (car **NO FORCE** est pris par défaut), l'utilisateur demande à ce que les contrôles d'existence des relations de base et des privilèges associés soient réalisés au moment de la création de la vue et non pas lors de sa première utilisation.

WITH CHECK OPTION permet d'effectuer des contrôles lors des opérations de mise à jour à travers la vue. Cette option est détaillée plus loin.

WITH READ ONLY limite le rôle de la vue à la consultation des données. Aucune opération de mise à jour ne peut être réalisée à travers la vue.

[27] La requête de définition d'une vue peut comporter jusqu'à 1.000 attributs ou expressions.

Exemple *: la vue suivante permet de consulter le nom et la capacité des avions desservant Paris.*

```
CREATE VIEW  AVION_CAP(NOM, CAP)
AS
SELECT DISTINCT NOMAV, CAPACITE
FROM AVION, VOL
WHERE AVION.NUMAV = VOL.NUMAV AND VILLE_ARR = 'PARIS'
WITH READ ONLY
```

□

4.3.2 – Utilisation de vues

Pour l'utilisateur final, une vue est une table qu'il peut interroger en formulant des requêtes et qu'il peut éventuellement mettre à jour.

Lorsque le SGBD évalue une requête formulée sur une vue, il combine la requête de l'utilisateur et la requête de définition de la vue pour obtenir le résultat.

Exemple *: considérons la vue VOL_MRS, définie ci-dessous, répertoriant les vols au départ de Marseille.*

```
CREATE VIEW VOL_MRS
AS
SELECT NUMVOL, VILLE_ARR, HEURE_DEP, HEURE_ARR
FROM VOL
WHERE VILLE_DEP = 'MARSEILLE'
```

Pour consulter les horaires des vols Marseille-Paris, l'utilisateur formule la requête suivante sur la vue.

```
SELECT HEURE_DEP, HEURE_ARR
FROM VOL_MRS
WHERE VILLE_ARR = 'PARIS'
```

La requête effectivement exécutée par le SGBD est la suivante.

```
SELECT HEURE_DEP, HEURE_ARR
FROM VOL
WHERE VILLE_DEP = 'MARSEILLE' AND VILLE_ARR = 'PARIS'
```

□

Lorsqu'une vue est utilisée pour effectuer des opérations de mise à jour, elle sert de " filtre " entre l'utilisateur et la base de données. En fait, ce n'est pas la vue qui est effectivement mise à jour, mais ***une*** des relations de base qui ont servi à la définir.

Pour que le SGBD puisse propager les mises à jour sur une relation de base, la définition d'une vue est soumise à des contraintes fortes. Plus précisément, il faut que :

- la requête de définition de la vue ne comprenne pas de **DISTINCT** ni de fonction agrégative dans le premier bloc **SELECT**,
- la requête de définition de la vue ne comprenne pas de clauses **GROUP BY**, **CONNECT BY**, ou **ORDER BY** dans le premier bloc **SELECT**,

- la requête de définition de la vue n'ait qu'une seule relation dans la clause **FROM** du premier bloc **SELECT**. Ceci implique que dans une vue multi-relation, les jointures soient exprimées de manière imbriquée,
- la requête de définition de la vue ne comprenne aucun opérateur ensembliste (i.e. **UNION**, **INTERSECT** ou **MINUS**) dans le premier bloc **SELECT**.

Lorsque la requête de définition d'une vue comporte une projection sur un sous-ensemble d'attributs d'une relation, les attributs non mentionnés prendront des valeurs nulles en cas d'insertion à travers la vue. Ainsi, si une vue est définie sans inclure l'attribut clef primaire (ou tout attribut à valeur obligatoire) de la relation spécifiée dans la clause **FROM** du premier bloc de la requête de définition, toute insertion à travers cette vue échouera. En revanche, les suppressions et modifications seront possibles.

***Exemple** : définir une vue permettant de consulter les vols des pilotes habitant Marseille et de les mettre à jour.*

```
CREATE VIEW VOLPIL_MRS
AS
SELECT *
FROM  VOL
WHERE NUMPIL  IN
        (SELECT  NUMPIL  FROM PILOTE
        WHERE  ADRESSE = 'MARSEILLE')
```

La vue précédente permet la consultation uniquement des vols vérifiant la condition donnée sur le pilote associé. Il est également possible de mettre à jour la relation VOL à travers cette vue mais l'opération de mise à jour peut concerner n'importe quel vol (sans aucune condition).
Par exemple supposons que le pilote n° 100 habite Paris, l'insertion suivante sera réalisée dans la relation VOL à travers la vue, mais le tuple ne pourra pas être visible en consultant la vue.

```
INSERT INTO VOLPIL_MRS(NUMVOL, NUMPIL, NUMAV, VILLE_DEP)
VALUES (IT256, 100, 14, 'PARIS')
```

□

***Exemple** : la vue suivante donne les numéros des avions non utilisés pour des vols.*

```
CREATE VIEW AVION_DISPO
AS
SELECT NUMAV FROM AVION
MINUS
SELECT NUMAV  FROM VOL
```

La vue précédente ne permet que la consultation. Toute opération de mise à jour est impossible, à cause de l'opération de différence, et génère l'erreur suivante " ORA-01732: les manipulations de données sont illégales sur cette vue ". □

4.3.3 – Dans quels cas utiliser des vues

Pour une bonne utilisation des vues, il faut analyser soigneusement l'objectif auquel elles répondent et la manière dont elles seront utilisées : consultation et/ou mises à jour. Pour une vue destinée uniquement à la consultation, la clause **WITH READ ONLY** doit être spécifiée.
Traditionnellement, les vues sont utilisées pour répondre à trois types d'objectifs.

1) assurer la confidentialité des données : il s'agit de restreindre l'accès de certains utilisateurs à certains attributs et/ou tuples dans la base ;

Exemple *: pour éviter que certains utilisateurs n'aient accès aux salaires et primes des pilotes, la vue suivante est définie à leur intention et ils n'ont pas de droit sur la relation PILOTE. De plus la vue définie ne permet que la consultation.*

```
CREATE VIEW RESTRICT_PIL
AS
SELECT NUMPIL, NOMPIL, PRENOMPIL, ADRESSE
FROM PILOTE
WITH READ ONLY
```
□

2) simplifier la tâche de l'utilisateur final en lui épargnant la formulation de requêtes complexes (incluant typiquement des jointures, partitionnements…) ou décomposer une requête trop complexe en plusieurs sous-requêtes ;

Exemple *: pour éviter la formulation d'une requête complexe par les utilisateurs, une vue est définie par les développeurs pour consulter la charge horaire des pilotes. Sa définition est la suivante :*

```
CREATE VIEW CHARGE_HOR (NUMPIL, NOM, CHARGE)
AS
SELECT P.NUMPIL, NOMPIL, SUM(HEURE_ARR – HEURE_DEP)
FROM PILOTE  P, VOL
WHERE P.NUMPIL = VOL.NUMPIL
GROUP BY  P.NUMPIL, NOMPIL
```

Lorsque cette vue est créée, les utilisateurs peuvent la consulter simplement par :

```
SELECT *
FROM CHARGE_HOR
```

Un utilisateur ne s'intéressant qu'aux pilotes parisiens dont la charge excède un seuil de 40 heures formulera la requête suivante.

```
SELECT *
FROM CHARGE_HOR C, PILOTE P
WHERE C.NUMPIL = P.NUMPIL AND CHARGE > 40 AND ADRESSE = 'PARIS'
```
□

3) contrôler la mise à jour des données : il s'agit d'imposer le respect de conditions pour autoriser une opération d'insertion, modification ou suppression sur une relation de base à travers une vue. Pour cela, la clause **WITH CHECK OPTION** est utilisée.
Si la clause **WITH CHECK OPTION** est présente dans l'ordre de création d'une vue, la table associée peut être mise à jour à condition que les tuples concernés par la mise à jour fassent ou puissent faire partie du résultat de la requête de définition. Considérons l'ordre suivant.

```
CREATE VIEW <nom_vue>
AS
SELECT <liste_attributs>
FROM <nom_relation>
WHERE <liste_conditions>
WITH CHECK OPTION
```

Des tuples de <nom_relation> peuvent être supprimés ou modifiés à travers la vue

uniquement si ils satisfont <liste_conditions>.
Un nouveau tuple peut être inséré à travers la vue, dans <nom_relation>, uniquement si il respecte <liste_conditions>.

L'utilisation de la clause **WITH CHECK OPTION** permet la vérification de toute contrainte d'intégrité. Ce rôle des vues doit être réservé aux contraintes dynamiques. Cependant si le SGBD ne permet pas de spécifier des contraintes d'intégrité structurelles, il est possible d'avoir recours au mécanisme de vues pour les mettre en œuvre.

***Exemple** : définir une vue permettant de consulter et de mettre à jour uniquement les vols des pilotes habitant Marseille.*

```
CREATE VIEW  VOLPIL_MRS
AS
SELECT *
FROM VOL
WHERE NUMPIL  IN
             (SELECT NUMPIL FROM PILOTE
             WHERE ADRESSE = 'MARSEILLE')
WITH CHECK OPTION
```

*L'ajout de la clause **WITH CHECK OPTION** à la requête précédente interdit toute opération de mise à jour sur les vols qui ne sont pas assurés par des pilotes marseillais et l'insertion d'un vol assuré par le pilote n° 100, habitant Paris, échoue.* □

***Exemple** : définir une vue sur PILOTE, permettant la vérification de la contrainte de domaine suivante " le salaire d'un pilote est compris entre 4.000 et 7.000 € ".*

```
CREATE VIEW  DPILOTE
AS
SELECT * FROM   PILOTE
WHERE SALAIRE  BETWEEN  4000  AND  7000
WITH CHECK OPTION
```

L'insertion suivante va alors échouer.

```
INSERT INTO DPILOTE (NUMPIL,SALAIRE) VALUES  (175, 9000)
```

□

***Exemple** : définir une vue sur vol permettant de vérifier les contraintes d'intégrité référentielle en insertion et en modification.*

```
CREATE VIEW  IMVOL
AS
SELECT *
FROM VOL
WHERE NUMPIL IN  (SELECT NUMPIL FROM PILOTE )
      AND NUMAV  IN  (SELECT NUMAV  FROM AVION)
WITH CHECK OPTION
```

□

4.3.4 – Manipulation de vues

De manière similaire à une relation de la base, la structure d'une vue peut être consultée grâce à la commande **desc[ribe]** ou en interrogeant les tables système ALL_VIEWS ou USER_VIEWS.

Pour modifier la requête de définition d'une vue, il faut la recréer (utilisation de **OR REPLACE**).

Les vues sont typiquement des " objets " dépendants car leur définition s'appuie sur d'autres " objets " ORACLE : les tables de base ou d'autres vues. ORACLE gère automatiquement les dépendances entre tous les " objets " d'un schéma. Ces derniers sont caractérisés par leur état qui peut prendre deux valeurs : valide ou invalide. En fait, seuls les " objets " dépendants peuvent être dans un état invalide. Ainsi, les relations de base, les synonymes et les séquences sont toujours dans un état valide. Pour connaître l'état de tout " objet ", il faut consulter l'attribut STATUS dans la table système USER_OBJECTS.
Lorsque des tables de base sont modifiées d'un point de vue structure par un ordre **ALTER TABLE**, lorsqu'elles sont renommées ou détruites, ORACLE attribue automatiquement le statut invalide à tous les " objets " (dont les vues) dépendant de ces relations de base directement ou indirectement. En effet, ces mises à jour structurelles peuvent avoir des répercussions sur les vues dépendantes et des erreurs peuvent apparaître lors de leurs prochaines utilisations.
Pour éviter que l'utilisateur final se trouve confronté à ces problèmes, le développeur peut recompiler explicitement une vue en utilisant la commande suivante.

```
ALTER VIEW <nom_vue> COMPILE
```

Si aucune erreur n'est détectée, l'état de la vue redevient valide, sinon elle reste dans un état invalide ainsi que tous les " objets " dépendants de cette vue.

Exemple *: considérons les deux vues définies ci-dessous dont les dépendances sont illustrées par la figure 36.*

```
CREATE VIEW VOL_CAP
AS
SELECT NUMVOL, VOL.NUMAV, CAPACITE, VILLE_DEP, VILLE_ARR
FROM  VOL, AVION
WHERE VOL.NUMAV = AVION.NUMAV AND  CAPACITE > 700
/
CREATE VIEW VOL_CAP_PARIS
AS
SELECT NUMVOL, NUMAV, CAPACITE, VILLE_ARR
FROM  VOL_CAP
WHERE VILLE_DEP = 'PARIS'
```

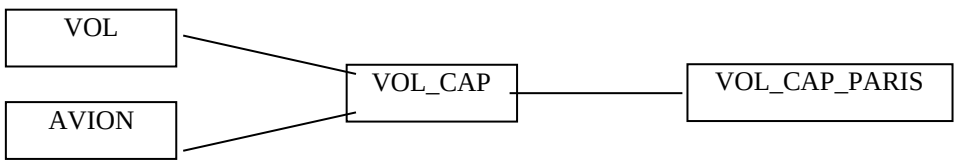


Figure 36 : Dépendances entre relations et vues

Toute modification du schéma des tables de base VOL et AVION provoque un état invalide pour les deux vues. □

Pour détruire une vue, l'ordre à utiliser est :

```
DROP VIEW <nom_vue>
```

Chapitre 5 - Conseils pratiques et conclusion

SQL est un langage de base de données. Il se veut facile et effectivement formuler des requêtes simples est aisé. Cependant, il est aussi particulièrement facile de formuler des requêtes syntaxiquement correctes et sémantiquement fausses. Si une interrogation est complexe, il ne faut jamais se contenter d'obtenir une réponse du SGBD mais il faut la vérifier sur un jeu d'essai pertinent comportant les divers cas de figure possibles.
Avant de commencer à formuler une requête en SQL, analysez soigneusement la question posée. En consultant le schéma de la base, recensez toutes les relations nécessaires à la requête et vérifiez qu'elles sont effectivement utiles. Vérifiez qu'il existe un chemin de jointure entre ces relations et si ce n'est pas le cas, ajoutez les relations requises.
Exprimez toutes les jointures nécessaires et seulement celles-là. ***Vérifiez systématiquement le nombre de jointures par rapport au nombre de relations figurant dans la clause*** **`FROM`**.

Si des calculs horizontaux sont formulés, vérifiez que tous les attributs y participant satisfont la contrainte **`NOT NULL`**. Si ce n'est pas le cas, utilisez systématiquement la fonction **`nvl`** (Cf. paragraphe 3.1.3.3). Le tableau suivant récapitule les différences entre les deux types de calculs.

	Calculs horizontaux	**Calculs verticaux**
Nombre de calcul	Un pour chaque tuple.	Un seul pour tous les tuples
Clauses possibles	**`SELECT – WHERE – ORDER BY – GROUP BY – HAVING – CONNECT BY`**	**`SELECT – HAVING`**
Restrictions	Pas de jointure externe sur un calcul horizontal mais sur les attributs dans l'expression.	Dans la clause **`SELECT`**, une fonction agrégative ne peut pas être associée à un attribut projeté sauf si c'est un attribut de partitionnement.
Clauses interdites	**`FROM`**	**`FROM – WHERE – GROUP BY - CONNECT BY`**
Valeurs nulles	Le résultat peut être erroné.	Le résultat est correct.

N'imbriquez jamais deux calculs verticaux seulement. Même si ORACLE rend, parfois, les résultats escomptés, appliquer une fonction agrégative sur le résultat d'une fonction agrégative est logiquement absurde.

N'oubliez pas les index destinés à optimiser l'accès aux données. Évitez d'inhiber leur utilisation, en intégrant des attributs indexés dans des expressions de calcul horizontal (Cf. paragraphe 2.6.2).

Pour les requêtes comportant des blocs corrélés (test d'absence ou d'existence de données et divisions avec double **`NOT EXISTS`**, Cf. paragraphes 3.1.12 et 3.1.17.2), vérifiez l'expression des jointures utilisant des alias.

Lorsque des partitionnements ou d'autres requêtes complexes doivent être formulés, il est souvent utile de commencer par imaginer le résultat escompté, ce qui permet par exemple de définir les attributs de partitionnement et les fonctions agrégatives éventuellement utilisées.

Ne confondez pas le rôle des clauses, en particulier pour :

- **WHERE** et **HAVING** : la clause **WHERE** est réservée à la formulation de conditions de sélection sur les tuples ou de conditions de jointure. D'ailleurs ORACLE vous interdit d'y exprimer des conditions sur les classes de tuples créés par un partitionnement. La clause **HAVING** doit être uniquement utilisée pour des conditions sur les classes de tuples, i.e. utilisant une fonction agrégative ;
- **WHERE** et **START WITH** : dans le contexte d'une recherche hiérarchique, la clause **WHERE** permet l'élimination de tous les tuples de l'arborescence examinée ne satisfaisant pas les conditions alors que la clause **START WITH** fixe les conditions de départ de la recherche ;
- **WHERE** et **AND** `<condition_hierarchique>`: lors d'une recherche hiérarchique, la clause **WHERE** élimine les tuples individuellement alors que **AND** `<condition_hierarchique>` élimine entièrement des sous-arbres ou des branches.

Quelques références

[1] Documentation ORACLE en ligne :
http://h50.isi.u-psud.fr/docmiage/oracle/doc/server.817/index.htm
http://storacle.princeton.edu:9001/oracle8-doc/server.805/a58225/toc.htm

[2] Interactive online SQL training
http://www.sqlcourse.com/ et http://sqlcourse2.com/
Cours (en anglais) sur SQL, assez peu approfondi mais avec exercices.

[3] Didier Deléglise "Oracle et le web en auto-formation"
http://didier.deleglise.free.fr/
Site (en français) abordant les divers aspects d'Oracle (SQL, administration, PL/SQL…).

[4] SQL Tutorial : http://www.1keydata.com/sql/sql.html
Cours (en anglais) d'introduction à SQL.

[5] Andrew Cumming "A gentle introduction to SQL"
http://www.dcs.napier.ac.uk/%7Eandrew/gisq/
Survol des différents aspects de SQL. La syntaxe est donnée pour plusieurs dialectes (Oracle, Mysql, Access…).

[6] http://searchdatabase.techtarget.com/
Différents articles (en anglais) sur des thèmes bases de données.

[7] http://hydria.u-strasbg.fr/G1/crs-admin.htm
cours (en français) proposant une introduction à l'administration de bases de données.

[8] http://www-rocq.inria.fr/~manolesc/MAITRISE/
Transparents de cours (en français) abordant de multiples aspects des bases de données (modèle relationnel, SQL, transactions…).

[9] http://medias.obs-mip.fr/cours/sql/
Support de cours (en français) proposant un survol complet de SQL.

[10] A. Zaslavsky "Transaction management in distributed computing systems"
http://www.csse.monash.edu.au/courseware/cse5501/mdcs-l-08/sld001.htm
Transparents (en anglais) sur la gestion de transactions.

[11] Teach yourself Oracle 8 in 21 days
http://member.netease.com/~andymsj/ora21/index.htm
Cours (en anglais) bien détaillé et conçu pour un auto-apprentissage d'ORACLE. Il aborde en particulier les tâches de l'administrateur de bases de données (installation du SGBD, tuning…)

[12] Tina London Guidelines and good practice guide for developing SQL
http://www.wsl.ch/relics/rauminf/riv/datenbank/general/tinalondon.html
Guide (en anglais) donnant des règles et conventions pour le codage en SQL.

[13] http://sirius.cs.ucdavis.edu/teaching/sqltutorial/
Support de cours (en anglais) assez complet sur SQL, PL/SQL et l'architecture d'Oracle.

[14] http://www.minet.net/linux/HOWTO-fr/Database-HOWTO-38.html
Spécification de la norme SQL ANSI/ISO

Jeu d'essai ACSI
http://www.rd.francetelecom.fr/fr/conseil/mento14/chap3a.html
http://igsinfo.nexenservices.com/Sql_Download/Production.pdf
http://www.inrs.fr/produits/pdf/nd2140.pdf
http://www.loria.fr/~jloncham/chap1-2003.pdf

en particulier DECODE
http://www.loria.fr/~roegel/cours/iut/oracle-sql.pdf