Rudiments SQL pour Oracle

Cyril Gruau*

23 septembre 2005

Résumé

Ce support de cours regroupe les instructions SQL basiques qui permettent de mettre en place une base de données sur un serveur Oracle, de la remplir et de l'interroger. Ce document, volontairement succinct, peut servir d'aide mémoire des commandes SQL essentielles, évitant ainsi la (re)-lecture du manuel utilisateur ou de tout autre ouvrage volumineux sur Oracle.

Mots-clef : connexion, table, clé primaire, clé étrangère, contrainte, requête, jointure, vue, sous-requête, groupement

Références

[Gruau] Gruau, C. *Conception d'une base de données*, 2005.

Ce support de cours permet de se familiariser avec les schémas relationnels, leur élaboration et leur normalisation (qui sont pré-recquis pour le document présent).

[Brouard et Soutou] Brouard, F. et Soutou, C. *SQL*, Synthex.

Cet ouvrage très accessible permet de bien comprendre le langage SQL, y compris dans ses extensions récentes.

[Soutou et Teste] Soutou, C. et Teste, O. *SQL pour Oracle*, Eyrolles.

Ce livre très complet sur la programmation SQL avec Oracle, ravira le lecteur désireux d'approfondir le document présent.

[Grin] Grin, R. *Langage SQL*, Université de Nice Sophia-Antipolis, 1998.

Ce support de cours présente la programmation SQL pour Oracle.

*Cyril.Gruau@ensmp.fr

Table des matières

Introduction

Oracle est un Système de Gestion de Bases de Données (SGBD) Relationnelles (SGBDR). Il s'agit d'un logiciel édité par Oracle Corporation et dont la première version remonte à 1977.

SQL (pour Structured Query Langage) est un langage de communication standard avec les SGBDR. Il a été conçu par IBM en 1976, puis normalisé en 1986, 1992, 1999 et 2003.

Comme premier exemple d'utilisation de SQL avec Oracle, prenons la création sur un serveur Oracle d'un nouvel utilisateur `cgruau` avec le mot de passe provisoire `cgpwd` . Pour effectuer cette création, l'administrateur doit saisir les commandes SQL suivantes :

```
1  CREATE USER cgruau
2      IDENTIFIED by cgpwd
3      DEFAULT TABLESPACE tbs_users
4          QUOTA 20M ON tbs_users
5      TEMPORARY TABLESPACE tmp_users
6          QUOTA 10M ON tmp_users
7          QUOTA 5M ON tools
8      PASSWORD EXPIRE;
9
10 -- les quotas peuvent être différents
11
12 GRANT CONNECT TO cgruau;
```

Conseils généraux sur la programmation SQL :

- SQL ne fait pas la différence entre les minuscules et les majuscules[1], donc nous adoptons la convention suivante (dans un soucis de lisibilité) : les mots-clés SQL en majuscules et les autres noms en minuscules ;
- le langage autorise également autant d'espaces et de sauts de ligne que l'on veut entre les mots de la programmation ; nous en profitons pour couper les lignes trop longues (celles qui dépassent la page ou l'écran) et pour indenter rigoureusement le texte afin d'aligner verticalement les éléments d'une même catégorie ;
- il ne faut pas hésiter non plus hésiter à commenter abondamment le code à l'aide des doubles tirets `--` ;
- enfin, il ne faut pas oublier le point-virgule à la fin de chaque instruction, car c'est lui qui déclenche son exécution.

Remarques sur l'exemple précédent :

- les espaces de travail `tbs_users`, `tmp_users` et `tools` existent déjà sur le serveur Oracle et il est conseillé de définir, pour chaque utilisateur, des quotas sur ces espaces communs ;
- le fait que le mot de passe `cgpwd` expire à la première connection, oblige l'utilisateur `cgruau` à choisir son propre mot de passe à l'insu de l'administrateur (sécurité et confidentialité) ;
- enfin, l'instruction `GRANT CONNECT` est indispensable pour que l'utilisateur `cgruau` puisse se connecter au serveur Oracle.

1. Sauf à l'intérieur des chaînes de caractères, heureusement.

1 Création de tables

Une partie du langage SQL est dédiée à la mise en place et à la maintenance du schéma relationnel des bases de données.

1.1 Syntaxe

Pour créer une table Oracle en SQL, il existe l'instruction `CREATE TABLE` dans laquelle sont précisés pour chaque colonne de la table : son intitulé, son type de donnée et une ou plusieurs contrainte(s) éventuelles(s). Pour créer les trois tables du schéma relationnel de la figure 1 :

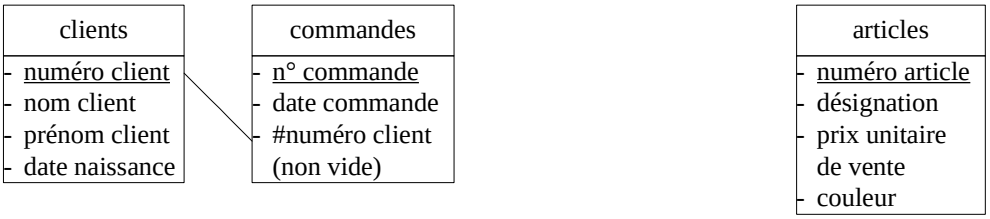

FIG. 1 – *Trois tables et une clé étrangère*

les trois instructions SQL correspondantes sont les suivantes (les noms des colonnes sont volontairement simplifiés) :

```
1  CREATE TABLE clients
2  (
3      numero NUMBER(6) CONSTRAINT pk_clients PRIMARY KEY,
4      nom VARCHAR2(63) NOT NULL,
5      prenom VARCHAR2(63),
6      naissance DATE
7  );
8
9  CREATE TABLE commandes
10 (
11     numero NUMBER(8) CONSTRAINT pk_commandes PRIMARY KEY,
12     jour DATE DEFAULT SYSDATE,
13     client NUMBER(6) CONSTRAINT fk_commandes_to_clients
14         REFERENCES clients(numero)
15 );
16
17 CREATE TABLE articles
18 (
19     numero NUMBER(6) CONSTRAINT pk_articles PRIMARY KEY,
20     designation VARCHAR2(255) UNIQUE,
21     prix NUMBER(8,2) NOT NULL,
22     couleur VARCHAR2(31)
23 );
```

Remarques :
- le type de donnée d'une clé étrangère doit être identique à celui de la clé primaire référencée ;
- une clé étrangère doit être créée après la table qu'elle référence ;
- pour les colonnes qui participent à une clé primaire, il est inutile de préciser `NOT NULL`, car c'est implicitement le cas avec `PRIMARY KEY` ;
- une valeur par défaut (déclarée avec `DEFAULT`) doit être une constante ou une fonction sans argument ;
- lorsque la valeur pas défaut n'est pas précisée, pour Oracle il s'agit de la valeur `NULL` ;
- la fonction `SYSDATE` retourne la date et l'heure courantes du serveur Oracle ;
- les noms que l'on donne aux tables, aux colonnes, etc., n'acceptent que des lettres (non accentuées) et des chiffres (sauf pour le premier caractère), éventuellement le tiret bas _ mais surtout pas d'espace ;
- les mots-clés SQL sont réservés, on ne peut donc pas appeler une colonne `date`.

Une table qui existe déjà peut être modifiée par l'instruction `ALTER TABLE` accompagnée d'une clause `ADD` ou `MODIFY` :

```
1 ALTER TABLE clients
2     ADD(code_postal CHAR(5));
3
4 ALTER TABLE articles
5     MODIFY(designation VARCHAR2(127));
```

Une table qui n'est référencée par aucune clé étrangère peut être supprimée à l'aide de l'instruction `DROP TABLE`. Si la suppression est autorisée, toutes les données de la table sont perdues. Dans notre exemple, l'instruction suivante :

```
1 DROP TABLE clients;
```

est refusée car la table `commandes` référence la table `clients`.

Pour supprimer ou modifier une table, il faut en connaître le nom. L'instruction suivante permet à un utilisateur d'Oracle de retrouver toutes ses tables :

```
1 SELECT * FROM user_tables;
```

Il peut également être utile de connaître la liste des colonnes d'une table en particulier. Pour cela, l'utilisateur dispose de l'instruction suivante :

```
1 DESCRIBE clients;
```

1.2 Types de données

Les principaux types de données ainsi que leurs limites sont décrits dans le tableau 1 :

type	syntaxe	description	limites
chaînes de caractères	`CHAR`	un seul caractère	
	`CHAR(n)`	chaîne de longueur fixe `n` (complétée par des espaces)	$1 \leqslant$ `n` $\leqslant 2000$
	`VARCHAR2(n)`	chaîne de longueur variable $\leqslant$ `n` (taille ajustée au contenu)	$1 \leqslant$ `n` $\leqslant 4000$
nombres	`NUMBER(p)`	entier à `p` chiffres décimaux	$1 \leqslant$ `p` $\leqslant 38$
	`NUMBER(p,s)`	nombre réel à virgule fixe à `p` chiffres décalés de `s` chiffres après la virgule	$-84 \leqslant$ `s` $\leqslant 127$
	`NUMBER`	nombre réel à virgule flottante avec environ 16 chiffres significatifs	
date et heure	`DATE`	entier décrivant le nombre de secondes écoulées depuis le 01/01/-4712 0:00:00	du 01/01/-4712 au 31/12/9999 à la seconde près

TAB. 1 – *Principaux types de données Oracle*

Attention, car en informatique et plus particulièrement avec SQL, la virgule d'un nombre est un point. Par ailleurs, les chiffres dont il est question avec le type de données `NUMBER(p,s)` sont des chiffres en base 10 (et non en base 2). De plus, le choix des entiers `p` et `s` doit être soigneusement réfléchi.

En effet, avec le type `NUMBER(8,2)`, la valeur 123456.789 est stockée sous la forme `123456.79` (arrondi à deux décimales après la virgule), tandis qu'avec le type `NUMBER(8,-2)` elle est stockée sous la forme `123400` (arrondi à deux décimales avant la virgule) auquel cas, seuls quatre chiffres sur les huit possibles sont utilisés pour cette valeur. Attention, car avec les types `NUMBER(5)`, `NUMBER(6,1)` et `NUMBER(4,-1)` la même valeur 123456.789 déclenche un dépassement de capacité (car la partie qui reste après le décalage des `s` chiffres comporte trop de chiffres significatifs).

Tous ces types de données acceptent la valeur `NULL` (qui signifie absence de valeur et non pas 0). Toute opération numérique (addition, multiplication, comparaison) avec un `NULL` résulte systématiquement en un `NULL` (y compris `NULL = NULL`). Heureusement, dans les requêtes SQL de la section 2, les valeurs `NULL` d'une colonne `x` peuvent être temporairement remplacées par une constante `valeur` grâce à la fonction `NVL(x, valeur)`.

1.3 Contraintes

Le fait d'être clé primaire, clé étrangère, non vide ou unique sont des contraintes possibles sur les colonnes d'une table. Comme nous l'avons vu au cours de la section précédente, les contraintes qui portent sur une seule colonne peuvent être déclarée en même temps que la colonne elle-même (que ce soit à la création ou lors d'une modification de la table).

1.3.1 Clé primaire composite

Lorsqu'une contrainte porte sur plusieurs colonnes, l'utilisateur est obligé de la déclarer à part et de lui donner un nom. Par exemple, la table de jonction `lignes de commandes`, qui avait été volontairement omise sur la figure 1 (page 4), possède une clé primaire composée de deux colonnes (figure 2) :

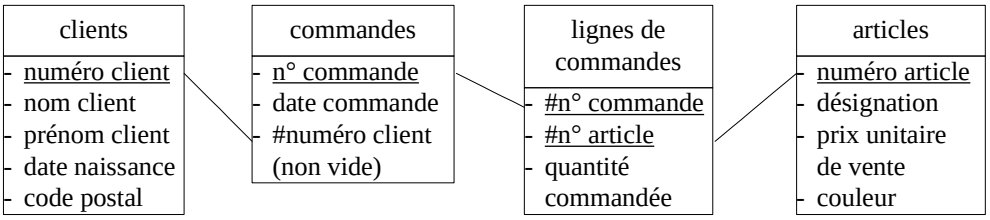

FIG. 2 – *Table de jonction avec clé primaire composite*

La syntaxe pour déclarer cette contrainte fait apparaître le mot-clé `CONSTRAINT` suivi du nom de la contrainte puis de sa description :

```
1  CREATE TABLE lignes_commandes
2  (
3      -- déclaration des colonnes et de leurs contraintes simples
4      commande NUMBER(8) CONSTRAINT fk_lignes_to_commandes
5          REFERENCES commandes(numero),
6      article  NUMBER(6) CONSTRAINT fk_lignes_to_articles
7          REFERENCES commandes(numero),
8      quantite NUMBER(4) NOT NULL,
9
10     -- déclaration des contraintes qui portent sur plusieurs colonnes
11     CONSTRAINT pk_lignes_commandes PRIMARY KEY (commande, article),
12 );
13
14 -- une erreur s'est glissée dans cet exemple
```

Le fait de préfixer le nom de la contrainte par `pk` ou `fk` n'est pas une obligation, mais permet de différencier plus facilement les clés primaires des clés étrangères lorsque l'utilisateur liste les contraintes de ses tables à l'aide de l'instruction suivante :

```
1 SELECT * FROM user_constraints;
```

1.3.2 De la nécessité de donner un nom aux contraintes

Lorsque l'utilisateur ne précise pas le nom d'une contrainte, le serveur Oracle en choisit un automatiquement, mais il n'est pas très explicite. Or, connaître le nom d'une contrainte est indispensable lorsque l'utilisateur souhaite la modifier ou la supprimer :

```
-- correction de l'erreur
ALTER TABLE lignes_commandes
    MODIFY CONSTRAINT fk_lignes_to_articles FOREIGN KEY (article)
        REFERENCES articles(numero);

-- suppression de la contrainte d'unicité sur la désignation des articles
ALTER TABLE lignes_commandes
    DROP CONSTRAINT SYS_C004016;

-- comme le nom lui a été donné par Oracle, il a fallut le trouver
-- à l'aide de user_constraints
```

C'est pourquoi il est recommandé de toujours donner un nom aux contraintes. D'ailleurs c'est obligatoire de préciser ce nom lorsqu'il s'agit d'ajouter une contrainte oubliée ou supprimée par erreur :

```
-- ajout d'une contrainte d'unicité sur la désignation des articles
ALTER TABLE articles
    ADD CONSTRAINT un_designation designation UNIQUE;
```

1.3.3 Clé étrangère composite

Les clés primaires ne sont pas les seules à pouvoir être composées de plusieurs colonnes. Les clés étrangères le peuvent également (figure 3) :

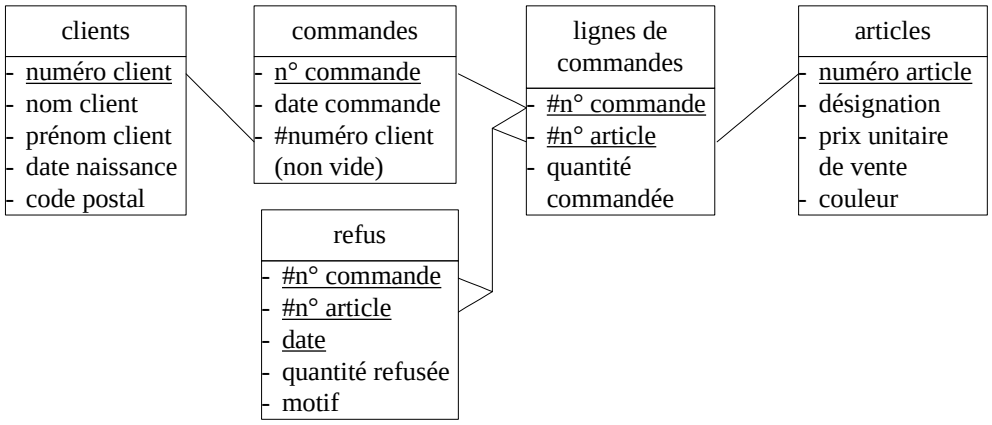

FIG. 3 – *Clé étrangère composite*

Comme pour les clés primaires composite, la déclaration d'une clé étrangère composite se fait en fin d'instruction `CREATE TABLE` :

```
1  CREATE TABLE refus
2  (
3      commande NUMBER(8),
4      article NUMBER(6),
5      jour DATE,
6      quantite NUMBER(4),
7      motif VARCHAR2(255),
8
9      CONSTRAINT pk_reparations PRIMARY KEY (commande, article, jour),
10     CONSTRAINT fk_reparations_to_lignes FOREIGN KEY (commande, article)
11         REFERENCES lignes_commandes(commande, article),
12 );
```

Il faut bien comprendre sur cet exemple que c'est le couple (numéro de commande, numéro d'article) dans la table `reparations` qui doit correspondre à un couple (numéro de commande, numéro d'article) déjà présent dans la table `lignes_commandes` et non pas (numéro de commande) et (numéro d'article) séparément.

1.3.4 Références croisées et réflexives

Le fait de pouvoir ajouter une contrainte après la création de la table correspondante est indispensable lorsque les références entre deux tables sont croisées ou réflexives comme sur la figure 4 :

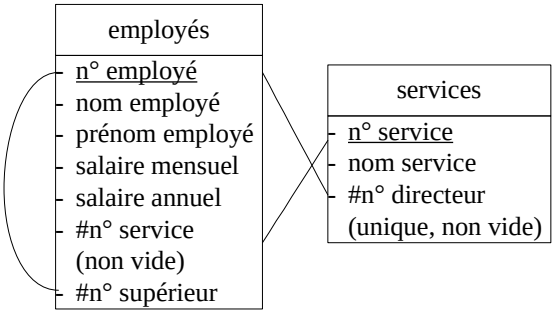

FIG. 4 – *Deux références croisées et une référence réflexive*

Il faut alors procéder en plusieurs étapes :

```
CREATE TABLE employes
(
    numero NUMBER(5) CONSTRAINT pk_employes PRIMARY KEY,
    nom VARCHAR2(127) NOT NULL,
    prenom VARCHAR2(63),
    salaire_mensuel NUMBER(7,2),
    salaire_annuel NUMBER(8,2),
);

-- ajout de la clé étrangère réflexive
ALTER TABLE employes
    ADD(superieur NUMBER(5) CONSTRAINT fk_employes_to_employes
            REFERENCES employes(numero));

CREATE TABLE services
(
    numero  NUMBER(2) CONSTRAINT pk_services PRIMARY KEY,
    nom VARCHAR2(63) NOT NULL,
    directeur NUMBER(5) NOT NULL
        CONSTRAINT fk_services_to_employes
            REFERENCES employes(numero)
        CONSTRAINT un_directeur UNIQUE
);

-- ajout de la clé étrangère croisée
ALTER TABLE employes
    ADD(service NUMBER(2) NOT NULL
            CONSTRAINT fk_employes_to_services
                REFERENCES services(numero));
```

1.3.5 Intégrité de domaine

Aux contraintes `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE` et `NOT NULL` s'ajoutent les contraintes `CHECK` qui permettent de s'assurer que les données saisies dans une colonne logent dans une plage de valeurs définie à l'avance :

```
-- on peut ajouter une contrainte à une colonne existante
ALTER TABLE articles
    ADD CONSTRAINT ck_prix CHECK(prix >= 0);

-- ou ajouter une colonne et sa (ses) contrainte(s) en même temps
ALTER TABLE clients
    ADD(genre CHAR CONSTRAINT ck_genre CHECK(genre IN ('M', 'F')));
```

Tout ce que l'on peut mettre dans une clause WHERE (section 2.1.2) on peut le mettre dans les parenthèses d'une contrainte `CHECK`, excepté une sous-requête. Cela inclut les opérateurs `AND`, `OR`, `NOT`, `BETWEEN`, `LIKE` ainsi que tous les opérateurs de comparaison :

```
-- opérateurs de comparaison utilisables
<  =  >  <=  >=  <>
-- le dernier opérateur signifie 'différent de'
```

Attention tout de même, car l'ajout ou la modification d'une contrainte peut être refusée par le serveur Oracle si une donnée déjà présente dans la table ne vérifie pas la définition de la contrainte.

Il est possible d'écrire une contrainte qui implique plusieurs colonnes d'une même table. Considérons la table `emprunts` de la figure 5 page 20 :

```
-- la date de retour prévu doit être postérieure à la date d'emprunt
ALTER TABLE emprunts
    ADD CONSTRAINT ck_date_retour CHECK(date_retour > date_emprunt);
```

En revanche, il n'est pas possible d'écrire avec une contrainte `CHECK`, une règle d'intégrité qui implique plusieurs colonnes issues de plusieurs tables.

```
-- cette contrainte, quoique légitime, n'est pas valide
ALTER TABLE refus
    ADD CONSTRAINT ck_quantite
        CHECK(refus.quantite <= lignes_commandes.quantite);

-- car rien ne dit dans cette définition à quelle ligne de la table
-- lignes_commandes, correspond la quantité d'une ligne de la table refus
```

Pour implémenter ce type de règle, il faut faire appel à des déclencheurs (notion qui n'est malheureusement pas abordée ici).

2 Sélection et modification des données

À ce stade, la base de données créée précédemment ne contient aucune donnée. Mais avant d'aborder les requêtes d'insertion, nous supposons qu'elle est remplie afin de pouvoir interroger ses données[2].

2.1 SELECT

Pour afficher les données sous la forme d'un tableau dont les colonnes portent un intitulé, SQL propose l'instruction `SELECT ... FROM ... WHERE ...` dont voici des exemples d'utilisation :

```
1  -- pour afficher les numéros et les noms des clients
2  SELECT numero, nom
3  FROM clients;
4
5  -- pour afficher toutes les colonnes de la table articles
6  SELECT *
7  FROM articles;
8
9  -- pour limiter les articles affichés à ceux qui coûtent moins de 100 (euros)
10 SELECT *
11 FROM articles
12 WHERE prix <= 100;
```

2.1.1 Colonnes calculées

Dans la clause `SELECT`, on peut utiliser les opérateurs arithmétiques (`+ - * /`), l'opérateur de concaténation des chaînes de caractères (`||`) ou des fonctions SQL (comme `MOD` pour le modulo, `FLOOR` pour la troncature entière ou `SUBSTR` pour l'extraction de sous-chaînes de caractères) :

```
1  -- pour afficher le prix unitaire TTC des articles
2  SELECT designation, prix * 1.196 AS "prix TTC"
3  FROM articles;
4
5  -- pour afficher l'initiale + nom et l'âge des clients
6  SELECT SUBSTR(prenom, 1, 1) || '. ' nom AS "identité",
7      FLOOR((SYSDATE - naissance) / 365.25) AS "âge",
8      MOD((SYSDATE - naissance), 365) AS "nombre de jours depuis l'anniversaire"
9  FROM clients;
```

Dès que l'on a un doute sur l'ordre de priorité des différents opérateurs, il ne faut pas hésiter à employer des parenthèses afin de lever toute ambiguïté.

À partir du moment où une colonne affichée est calculée, il est conseillé de lui donner un nom (un alias) avec le mot-clé `AS`, afin d'en expliciter le contenu, car rien n'est plus ambigu que de ne pas connaître la nature du contenu d'une colonne du résultat. Un alias, comme `"prixTTC"` est donné entre guillemets, contrairement aux chaînes de caractères qui sont données entre quotes `'. '`.

2. Une fois que l'on sait sélectionner les données, il est plus facile d'apprendre à les insérer, à les supprimer et à les mettre à jour.

Dans cette même clause `SELECT`, on peut aussi utiliser les fonctions d'agrégation SQL (tableau 2) :

syntaxe	description
`COUNT`	dénombrement des lignes
`SUM`	somme des valeurs numériques
`MIN`	valeur numérique minimale
`MAX`	valeur numérique maximale
`AVG`	moyenne (average) des valeurs numériques

TAB. 2 – *Principales fonctions d'agrégation SQL*

Par bonheur, ces fonctions ignorent les valeurs `NULL` dans leur calcul et n'éliminent pas les doublons. Exemples d'utilisation :

```
1  -- pour afficher le prix de l'article le plus cher,
2  -- le prix le moins cher et le prix moyen
3  SELECT MAX(prix) AS "prix le plus cher",
4      MIN(prix) AS "prix le moins cher",
5      AVG(prix) AS "prix moyen"
6  FROM articles;
7
8  -- pour afficher le nombre de clients
9  SELECT COUNT(numero) AS "nombre de clients"
10 FROM clients;
11
12 -- comme * signifie toutes les colonnes
13 -- la requête suivante est équivalente à la précédente
14 SELECT COUNT(*) AS "nombre de clients"
15 FROM clients;
```

2.1.2 Clause WHERE

La clause `WHERE` d'une requête peut être très riche en conditions de sélection. D'abord, tous les opérateurs de comparaison (page 11) peuvent être utilisés :

```
1  -- pour afficher les clients dont la date de naissance
2  -- n'est pas le 25 décembre 1960 (pourquoi pas ?)
3  SELECT *
4  FROM clients
5  WHERE naissance <> '25/12/1960';
```

Ensuite, comme les syntaxes `= NULL` ou `<> NULL` renvoient toujours « faux », on ne peut tester la vacuité d'une colonne qu'avec la syntaxe suivante :

```
1  -- pour afficher les clients dont connaît le code postal
2  SELECT *
3  FROM clients
4  WHERE code_postal IS NOT NULL;
```

Dans cet exemple, le mot-clé logique `NOT` est utilisé. Nous avons également à disposition les mots-clés `AND` et `OR` logiques[3] :

```
-- pour afficher les articles bleus
-- dont le prix est supérieur à 100 ou inférieur à 10
SELECT *
FROM articles
WHERE couleur = 'bleu'
    AND (prix >= 100 OR prix <= 10);
```

Par contre, pour un segment de valeurs bornes incluses, il n'est pas nécessaire d'utiliser la syntaxe `... >= ... AND ... <= ...`, car nous pouvons utiliser l'opérateur `BETWEEN` :

```
-- pour afficher les articles dont le prix est entre 10 et 100
SELECT *
FROM articles
WHERE prix BETWEEN 10 AND 100;
```

Ensuite, une condition de sélection peut faire intervenir une liste de valeurs, auquel cas c'est l'opérateur `IN` qui est approprié :

```
-- pour afficher les clients qui ne s'appellent
-- ni 'Dupont', ni 'Durand', ni 'Dubois'
SELECT *
FROM clients
WHERE nom NOT IN ('Dupond', 'Durand', 'Dubois');
```

Enfin, il est possible de balayer une colonne de type chaîne de caractères à la recherche d'un motif, grâce à l'opérateur de filtre `LIKE` et du caractère `%` qui remplace toute série de caractère(s) (y compris vide) :

```
-- pour afficher les clients dont le nom commence par D
SELECT *
FROM clients
WHERE nom LIKE 'D%';

-- pour afficher les clients dont le nom finit par e
SELECT *
FROM clients
WHERE nom LIKE '%e';

-- pour afficher les clients dont le nom ne contient pas y
SELECT *
FROM clients
WHERE nom NOT LIKE '%y%';
```

Il existe également le caractère `_` qui remplace un caractère (pour remplacer deux caractères, il suffit de l'utiliser deux fois dans le filtre).

3. Le « ou » logique n'est pas exclusif : c'est l'un ou l'autre, ou les deux (contrairement aux restaurants avec les fromages et les desserts).

2.1.3 Opérations sur le résultat

Le résultat d'une requête `SELECT` peut comporter plusieurs fois la même ligne. Pour éliminer les lignes doublons, il existe le mot-clé `DISTINCT` :

```
-- pour afficher l'ensemble des patronymes des clients
SELECT DISTINCT nom
FROM clients;

-- sans DISTINCT le résultat aurait autant de lignes
-- que la table clients
```

Ensuite, pour trier les lignes du résultat (ce qui n'est pas le cas, par défaut), nous disposons de la clause `ORDER BY` :

```
SELECT DISTINCT nom
FROM clients
ORDER BY nom ASC;

-- ASC pour classer dans l'ordre croissant
-- DESC pour classer dans l'ordre décroissant

-- on peut utiliser dans la clause ORDER BY
-- les alias de la clause SELECT
```

Enfin, pour ne garder que les 100 premières lignes du résultat, Oracle a prévu de numéroter les lignes du résultat dans une colonne cachée nommée `ROWNUM` :

```
SELECT DISTINCT nom
FROM clients
WHERE ROWNUM <= 100
ORDER BY nom ASC;
```

2.1.4 Opérations ensemblistes

On peut articuler les résultats de plusieurs requêtes homogènes à l'aide des opérations ensemblistes `UNION`, `INTERSECT` et `MINUS` :

```
1  -- pour afficher les identités de tous les clients
2  -- et de tous les employés
3  SELECT prenom || ' ' || nom AS "identité"
4  FROM clients
5
6  UNION
7
8  SELECT prenom || ' ' || nom AS "identité"
9  FROM employes
10 ORDER BY "identité" ASC;
11
12 -- pour afficher les identités de tous les clients
13 -- qui ont un homonyme parmi les employés
14 SELECT prenom || ' ' || nom AS "identité"
15 FROM clients
16
17 INTERSECT
18
19 SELECT prenom || ' ' || nom AS "identité"
20 FROM employes
21 ORDER BY "identité" ASC;
22
23 -- pour afficher les identités de tous les clients
24 -- qui n'ont pas d'homonyme parmi les employés
25 SELECT prenom || ' ' || nom AS "identité"
26 FROM clients
27
28 MINUS
29
30 SELECT prenom || ' ' || nom AS "identité"
31 FROM employes
32 ORDER BY "identité" ASC;
```

Les règles à respecter avec ces opérations ensemblistes sont les suivantes :

- les colonnes affichées par les deux requêtes doivent être compatibles, en nombre, en ordre et en type de données ;
- les éventuels alias ne sont définis que dans la première clause `SELECT` ;
- une éventuelle clause `ORDER BY` n'est possible qu'à la fin de la dernière requête, car elle agit sur le résultat final.

Dernières remarques :

- par défaut, l'opérateur `UNION` élimine les doublons ; pour les conserver, il faut utiliser l'opérateur `UNION ALL` ;
- contrairement à `UNION` et `INTERSECT`, l'opérateur `MINUS` n'est pas commutatif, donc l'ordre dans lequel les requêtes sont écrites, de part et d'autre, a de l'importance.

2.2 INSERT

Pour créer une ou plusieurs ligne(s) dans une seule table, SQL offre l'instruction `INSERT INTO`. Lorsque l'on connaît directement les valeurs à insérer, il faut utiliser une instruction `INSERT INTO ... VALUES` par ligne :

```
1 INSERT INTO clients(nom, prenom, code_postal)
2     VALUES ('Gruau', 'Cyril', '06600');
```

Plusieurs précautions doivent être prises lors d'une instruction `INSERT INTO` :

- les valeurs qui sont données via `VALUES` doivent être dans le même ordre que les colonnes qui sont précisées dans le `INTO` ;
- et avec un type de données compatible à celui qui a été déclaré dans la table ;
- toutes les colonnes qui ne sont pas précisées, reçoivent alors la valeur défaut qui leur a été attribuée (bien souvent, il s'agit de la valeur `NULL`) ;
- si la moindre valeur insérée ne vérifie pas les contraintes de la table (clé primaire, clé étrangère, unicité, `NOT NULL` ou `CHECK`), alors l'instruction `INSERT INTO` est refusée en entier par le serveur Oracle.

Dans notre exemple, la colonne `naissance` reçoit la valeur `NULL`, ce qui n'est pas gênant. Par contre, la colonne clé primaire `numero` reçoit également la valeur `NULL`, ce qui est interdit. L'instruction `INSERT INTO` précédente est donc refusée.

Pour que la colonne `numero` reçoive un entier incrémenté automatiquement, il suffit de créer, avant la première insertion, une séquence pour cette table et utiliser cette séquence (avec `NEXTVAL`) dans l'instruction `INSERT INTO` :

```
1  CREATE SEQUENCE sq_clients
2      MINVALUE 1
3      MAXVALUE 999999
4      START WITH 1
5      INCREMENT BY 1;
6
7  -- la valeur maximale doit être compatible
8  -- avec le type de données de la clé primaire
9
10 INSERT INTO clients(numero, nom, prenom, code_postal)
11     VALUES (sq_clients.NEXTVAL, 'Gruau', 'Cyril', '06600');
12
13 -- cette insertion est acceptée
```

On peut également insérer dans une table, le résultat d'une requête `SELECT`, auquel cas, plusieurs lignes peuvent être insérées à la fois :

```
1 -- supposons qu'il existe une table commandes_anciennes
2 -- qui possède la même structure que la table commandes
3
4 -- insérons les commandes qui datent de plus de 6 mois dans cette table
5 INSERT INTO commandes_anciennes(numero, jour, client)
6     SELECT numero, jour, client
7     FROM commandes
8     WHERE (SYSDATE - jour) >= 180;
```

Les colonnes qui figurent dans la clause `SELECT` doivent être compatibles, en type de données, en nombre et en ordre, avec celles qui sont précisées dans la clause `INTO`. De plus, si une valeur dans une ligne ne vérifie pas les contraintes qui la concerne, alors l'insertion de toutes les lignes est refusée (il n'y a pas d'insertion partielle).

2.3 DELETE

Pour supprimer une ou plusieurs ligne(s) dans une seule table, la langage SQL a prévu l'instruction `DELETE FROM ... WHERE ...` :

```
1 -- pour supprimer les commandes qui datent de plus de 6 mois
2 DELETE FROM commandes
3 WHERE (SYSDATE - jour) >= 180;
```

À nouveau, il ne peut y avoir de réussite partielle de l'instruction `DELETE` : si une des lignes supprimées est référencée par une clé étrangère, alors l'instruction `DELETE` est refusée en entier par le serveur Oracle. Cela permet d'éviter qu'une clé étrangère ne deviennent orpheline.

L'attention du lecteur est également attirée sur le fait que si la clause `WHERE` est oubliée, alors toute la table est vidée :

```
1 -- suppression de toutes les commandes
2 DELETE FROM commandes;
```

Heureusement, cette instruction sera vraisemblablement refusée car il doit exister une ligne de commande qui référence une commande.

2.4 UPDATE

Pour modifier la valeur d'une ou plusieurs colonne(s), d'une ou plusieurs ligne(s), mais dans une seule table, c'est l'instruction `UPDATE ... SET ... WHERE ...` qu'il faut utiliser :

```
-- pour convertir, en euros, les prix en francs
UPDATE articles
SET prix = prix / 6.55957;
```

De nouveau, lorsque la clause `WHERE` est omise, toutes les lignes sont modifiées et lorsqu'une des nouvelles valeurs ne vérifie pas les contraintes, l'instruction `UPDATE` est refusée en entier par le serveur.

L'informaticien avisé aura noté que la requête suivante est plus performante que la première, car une division coûte toujours beaucoup plus cher que la multiplication par l'inverse :

```
-- requête équivalente mais plus rapide
UPDATE articles
SET prix = prix * 0.152449;
```

Enfin, on peut mettre à jour plusieurs colonnes à la fois, mais il faut se méfier des corrélations entre les formules. Considérons par exemple la table `employes` de la figure 4 (page 10) et augmentons de 10 % les employés du service n°4 :

```
UPDATE employes
SET salaire_mensuel = salaire_mensuel * 1.1,
    salaire_annuel = salaire_mensuel * 12
WHERE service = 4;

-- cette requête ne convient pas, car à gauche des deux signes =
-- le salaire_mensuel est celui AVANT augmentation
-- il faut donc séparer le UPDATE en deux :

UPDATE employes
SET salaire_mensuel = salaire_mensuel * 1.1,
WHERE service = 4;

UPDATE employes
SET salaire_annuel = salaire_mensuel * 12;

-- autre solution, mais qui ne fonctionne que si
-- salaire_annuel égale déjà salaire_mensuel * 12

UPDATE employes
SET salaire_mensuel = salaire_mensuel * 1.1,
    salaire_annuel = salaire_mensuel * 1.1
WHERE service = 4;
```

3 Jointures

Les requêtes écrites jusqu'à présent ne font intervenir qu'une seule table à la fois. Or, dans la plupart des cas, une requête a besoin de données qui se trouvent réparties dans plusieurs tables.

3.1 Sélection multi-table

Pour afficher à la fois la date de commande et le nom du client qui a passé la commande, les deux tables `commandes` et `clients` doivent figurer dans la clause `FROM` et il faut rappeler la liaison entre ces deux tables[4], dans la clause `WHERE` :

```
1 -- jointure simple entre deux tables
2 SELECT commandes.numero, commandes.jour, clients.numero, clients.nom
3 FROM clients, commandes
4 WHERE commandes.client = clients.numero;
```

Remarques :

- la condition qui figure dans la clause `WHERE` est une condition de jointure, à ne pas confondre avec les conditions de sélection rencontrée jusqu'à présent[5] ;
- cette fois-ci, la condition de jointure est un rappel de la liaison clé étrangère - clé primaire, mais ce ne sera pas toujours le cas ;
- étant donné que `commandes.client = clients.numero`, on aurait pu afficher `commandes.client` au lieu de `clients.numero` ;
- l'ordre des tables dans la clause `FROM` n'a pas d'importance.

Pour lever toute ambiguïté sur le nom des colonnes (notamment la colonne `numero`), il faut préciser le nom de la table en préfixe. Heureusement, pour raccourcir les noms complets des colonnes, on peut définir des alias dans la clause `FROM` :

```
1 -- même requête avec des alias de table
2 SELECT a.numero, a.jour, b.numero, b.nom
3 FROM commandes a, clients b
4 WHERE a.client = b.numero;
```

Considérons le schéma relationnel de la figure 5 :

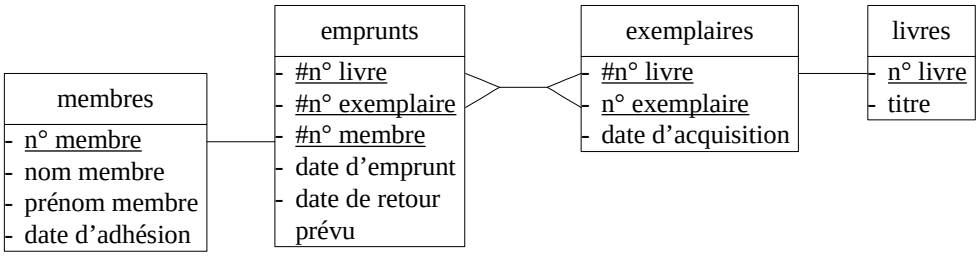

FIG. 5 – *Schéma relationnel sur lequel ne figure pas la condition de jointure*

4. C'est obligatoire, sinon c'est le produit cartésien des deux tables qui s'affiche.

5. Pour alléger la clause `WHERE`, il est d'ailleurs recommandé d'utiliser l'opérateur `JOIN` (disponible avec les versions récentes d'Oracle) dans la clause `FROM` :
SELECT a.numero, a.jour, b.numero, b.nom
FROM commandes a JOIN clients b ON (a.client = b.numero);

Nous pouvons imaginer un exemple de requête dans laquelle la condition de jointure n'est pas une liaison directe clé étrangère - clé primaire :

```
-- court-circuit de la table des exemplaires
SELECT a.date_emprunt, b.titre
FROM emprunts a, livres b
WHERE a.numero_livre = b.numero;
```

En outre, il est possible dans une requête, de faire la jointure entre trois tables (ou plus), auquel cas il faut fournir au minimum deux conditions de jointure. À ces conditions de jointure, peuvent s'ajouter une ou plusieurs condition(s) de sélection (que l'on mettra de préférence à la suite) :

```
-- afficher les commandes complètes du client n°3945
SELECT c.designation, c.prix, b.quantite, c.prix * b.quantite AS "total"
FROM commandes a, lignes_commandes b, articles c
WHERE a.numero = b.commande
    AND b.article = c.numero
    AND a.client = 3945;

-- afficher les commandes complètes de clients qui s'appellent Dupond
SELECT c.designation, c.prix, b.quantite, c.prix * b.quantite AS "total"
FROM commandes a, lignes_commandes b, articles c, clients d
WHERE a.numero = b.commande
    AND b.article = c.numero
    AND a.client = d.numero
    AND d.nom = 'Dupond';

-- dans cette requête la tables commandes ne sert pas à la clause SELECT,
-- mais est indispensable pour les conditions de la clause WHERE
```

Par ailleurs, une même jointure peut faire appel à plusieurs conditions pour être réalisée. C'est le cas lorsque la clé étrangère est composite :

```
-- affichage des exemplaires empruntés
-- avec jointure sur deux colonnes
SELECT a.date_emprunt, b.date_acquisition
FROM emprunts a, exemplaires b
WHERE a.numero_livre = b.numero_livre
    AND b.numero_exemplaire = b.numero_exemplaire;

-- affichage des quantités commandées et refusées des commandes
-- si la quantité refusée est strictement supérieure à celle commandée (anomalie)
SELECT a.article, a.quantite AS "commandée", b.quantite AS "refusée"
FROM lignes_commandes a, refus b
WHERE a.commande = b.commande
    AND a.article = b.article
    AND a.quantite < b.quantite;
```

Cette dernière requête devient plus lisible grâce à l'opérateur `JOIN` dans la clause `FROM` :

```
SELECT a.article, a.quantite AS "commandée", b.quantite AS "refusée"
FROM lignes_commandes a
    JOIN refus b ON (a.commande = b.commande AND a.article = b.article)
WHERE a.quantite < b.quantite;
```

Enfin, une table peut apparaître plusieurs fois dans la clause `FROM`, lorsque la requête nécessite de cette table qu'elle joue plusieurs rôles à la fois. C'est le cas dans la section suivante.

3.2 Auto-jointure

Un problème qu'il faut avoir rencontré au moins une fois pour être capable de le résoudre, est illustré par l'exercice suivant : afficher les numéros des clients qui ont commandé en même temps (sur deux colonnes). La solution consiste à faire intervenir dans cette requête, deux fois la table des commandes (ce qui n'est possible que grâce aux alias de table) avec une condition de jointure qui assure qu'à chaque ligne, les deux clients sont distincts :

```
-- couples de clients qui ont commandé en même temps
SELECT DISTINCT a.client, b.client
FROM commandes a, commandes b
WHERE a.client < b.client
    AND a.jour = b.jour;

-- DISTINCT est indispensable car sinon,
-- deux clients qui ont commandé en même temps
-- deux fois, apparaîtraient deux fois.
```

Nous voyons sur cet exemple qu'une condition de jointure n'utilise pas forcément l'opérateur =. Dans ce type de requête, l'emploi de l'opérateur < ou > est préférable à l'emploi de l'opérateur <>, car ce dernier ferait apparaître chaque couple deux fois : une fois dans un sens et une fois dans l'autre.

Pour afficher, non pas les numéros de client, mais les noms des clients, il faut joindre à chaque table `commandes` sa table `clients`, soit quatre tables au total :

```
-- affichage du nom au lieu du numéro de client
SELECT DISTINCT c.nom, d.nom
FROM commandes a, commandes b, clients c, clients d
WHERE a.client < b.client
    AND a.client = c.numero
    AND b.client = d.numero
    AND a.jour = b.jour;
```

Par ailleurs, pour compter les couples de clients, le mot-clé `DISTINCT` (qui est indispensable) doit apparaître à l'intérieur de la fonction `COUNT` :

```
-- nombre de couples de clients
-- qui possède le même code postal
SELECT COUNT(DISTINCT *)
FROM clients a, clients b
WHERE a.numero < b.numero
    AND a.code_postal = b.code_postal;
```

3.3 Jointure externe

La table `annuaire_client` de la figure 6 contient des informations complémentaires relatives aux clients :

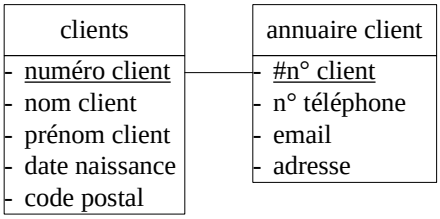

FIG. 6 – *Table d'informations complémentaires pour illustrer la jointure externe*

Mais comme il existe des clients pour lesquels on ne connaît aucune information complémentaire, la requête suivante n'affiche pas tous les clients :

```
1 -- affichage de toutes les informations
2 -- relatives aux clients dont on connaît
3 -- les informations complémentaires
4 SELECT *
5 FROM clients a, annuaire_client b
6 WHERE a.numero = b.numero;
```

Il faut donc que la condition de jointure ne soit pas obligatoire afin de ne pas supprimer les clients dont on n'a pas d'information complémentaire. On parle alors de jointure externe. Il suffit pour cela d'ajouter le signe `+` entre parenthèse dans la condition de jointure du côté de la table facultative :

```
1  -- affichage de toutes les informations
2  -- relatives à tous les clients
3  SELECT *
4  FROM clients a, annuaire_client b
5  WHERE a.numero = b.numero (+);
6
7  -- requête équivalente
8  SELECT *
9  FROM clients a, annuaire_client b
10 WHERE b.numero (+) = a.numero;
```

Dans le résultat de cette requête, les colonnes `telephone`, `email` et `adresse` des clients pour lesquels ces informations sont inconnues affichent des valeurs `NULL`.

Le signe `+` ne fonctionne pas avec l'opérateur `JOIN`, il faut utiliser l'opérateur `LEFT OUTER JOIN` :

```
1 -- requête équivalente
2 SELECT *
3 FROM clients a LEFT OUTER JOIN annuaire_client b ON (a.numero = b.numero);
4
5 -- ou encore
6 SELECT *
7 FROM annuaire_client b RIGHT OUTER JOIN clients a ON (a.numero = b.numero);
```

Par ailleurs, si une jointure externe repose sur plusieurs conditions, alors il se peut que plusieurs signes + soient utiles :

```
-- affichage des exemplaires et de leur date d'emprunt éventuelle
SELECT a.date_acquisition, b.titre, c.date_emprunt
FROM exemplaires a, livres b, emprunts c,
WHERE a.numero_livre = b.numero
    AND a.numero_livre = c.numero_livre (+)
    AND a.numero_exemplaire = c.numero_exemplaire (+);

-- dans cet exemple, il y a deux jointures :
-- une jointure normale et une jointure externe et composite
```

Maintenant, si la table facultative de la jointure externe est elle-même jointe à une autre table, alors cette nouvelle jointure doit aussi être externe :

```
-- affichage des exemplaires et de leur membre d'emprunt éventuel
SELECT a.date_acquisition, b.titre, c.date_emprunt, d.nom
FROM exemplaires a, livres b, emprunts c, membres d
WHERE a.numero_livre = b.numero
    AND a.numero_livre = c.numero_livre (+)
    AND a.numero_exemplaire = c.numero_exemplaire (+)
    AND c.numero_membre = d.numero (+);
```

Enfin, il est parfois indispensable d'utiliser la fonction `NVL` pour effectuer des calculs avec des colonnes issues d'une jointure externe. Considérons par exemple, le schéma relationnel de la figure 7 :

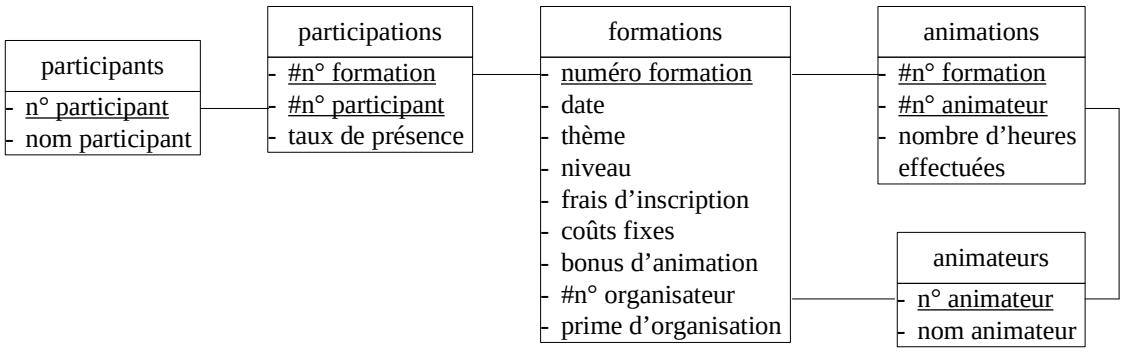

FIG. 7 – *Schéma relationnel qui possède un cycle*

Pour mettre à plat le cycle de ce schéma relationnel, il est nécessaire de dédoubler une des tables du cycle :

```
-- pour afficher la rémunération des animateurs de la formation n°123
SELECT b.nom, a.heures * b.tarif * c.bonus + NVL(d.prime, 0)
FROM animations a, animateurs b, formations c, formations d
WHERE a.animateur = b.numero
    AND a.formation = c.numero
    AND b.numero = d.organisateur (+)
    AND a.formation = 123;

-- NVL est indispensable car pour la plupart des animateurs la prime est NULL
```

3.4 Vue

On peut donner un nom à une requête `SELECT`, puis l'utiliser comme une table dans une autre requête :

```
1  CREATE OR REPLACE VIEW customers
2      (number, lastname, firstname, birthdate, zip, phone, email, address)
3  AS
4      SELECT a.numero, a.nom, a.prenom, a.naissance, a.code_postal,
5          b.telephone, b.email, b.adresse
6      FROM clients a, annuaire_client b
7      WHERE a.numero = b.numero (+);
8
9  -- puis la requête suivante fonctionne
10
11 SELECT birthdate, phone
12 FROM customers
13 WHERE lastname = 'Dupond';
```

Remarque : le couple d'instructions `CREATE OR REPLACE VIEW` est plus pratique que la seule instruction `CREATE VIEW`, car pendant sa phase de définition, la programmation d'une vue est amenée à être corrigée plusieurs fois. Sans `REPLACE VIEW`, il faudrait supprimer puis re-créer la vue à chaque correction.

Les nouveaux intitulés de colonnes donnés lors de la définition de la vue pourront être ré-utilisés comme n'importe quel nom de colonne, on veillera donc à respecter les mêmes règles dans leur dénomination (pas d'espace et pas d'accent, notamment).

Attention, une vue ne contient pas de données, seules les tables sous-jacentes en contiennent. Mais les intérêts des vues sont nombreux, en voici les premiers :

- le travail répétitif de jointure et de sélection de la clause `WHERE` peut être fait une bonne fois pour toute dans la définition de la vue (simplification de programmation des requêtes qui utilisent les vues) ;
- si le schéma relationnel change, alors la programmation des vues risque de changer mais pas forcément les requêtes qui utilisent les vues (gain de travail de maintenance) ;
- si les utilisateurs n'ont accès qu'aux vues qui les concernent, alors ils ne voient que les données dont ils ont besoin (simplicité et sécurité) ;
- il est possible de donner aux vues et à leurs colonnes, des intitulés plus parlant que ceux des tables et de les traduire dans différentes langues ou langages métiers (clarté et adaptation aux utilisateurs : Business Objects n'a rien inventé ;-).

Un autre avantage des vues : elles permettent d'effectuer des requêtes `INSERT`, `DELETE` et `UPDATE` sur plusieurs tables à la fois, à condition d'utiliser des déclencheurs (notion qui n'est malheureusement pas abordée ici). En outre, elles offrent une alternative systématiquement plus efficace à certaines sous-requêtes (section suivante).

4 Sous-requêtes

Il est possible d'utiliser dans une requête principale, une autre requête `SELECT` à condition que cette dernière soit délimitée par des parenthèses.

4.1 Sous-requête qui renvoient une seule valeur

Lorsque la sous-requête renvoie une unique valeur (encore fait-il en être sûr), la requête principale peut utiliser cette valeur avec les opérateurs de comparaison classiques :

```
1  -- pour afficher les articles dont le prix
2  -- est supérieur à tous ceux des articles bleus
3  SELECT *
4  FROM articles
5  WHERE prix >= ( SELECT MAX(prix)
6                  FROM articles
7                  WHERE couleur = 'bleu');
8
9  -- pour avoir les articles dont le prix est,
10 -- cette fois-ci, inférieur
11 -- à tous ceux des articles bleus,
12 -- il suffit de remplacer >= par <= et MAX par MIN
```

Ce type de sous-requête s'emploie également avec les opérateurs arithmétiques classiques :

```
1  -- pour afficher la proportion de clients mineurs parmi l'ensemble des clients
2  SELECT
3  (
4      SELECT COUNT(*)
5      FROM clients
6      WHERE FLOOR((SYSDATE - b.naissance) / 365.25) < 18
7  )
8  /
9  (
10     SELECT COUNT(*)
11     FROM clients
12 )
13 * 100 AS "proportion de clients mineurs"
14 FROM DUAL;
15
16 -- il n'y a pas de table dans la requête principale, mais comme la clause FROM
17 -- est obligatoire Oracle nous offre la pseudo-table DUAL
```

Lorsque la sous-requête ne renvoie qu'une valeur, il n'existe bien souvent pas d'équivalent plus performant. Ce n'est pas toujours le cas dans ce qui suit.

4.2 Sous-requête qui renvoient une colonne

Lorsque la sous-requête renvoie une colonne de valeurs (c'est le cas le plus fréquent), la requête principale peut utiliser un opérateur de comparaison classique, mais accompagné soit de `ALL` soit de `ANY` :

```
-- requête équivalente à la précédente
SELECT *
FROM articles
WHERE prix >= ALL ( SELECT prix
                    FROM articles
                    WHERE couleur = 'bleu');
```

Malheureusement dans la majorité des cas, l'emploi du mot-clé `ALL` dans la requête principale, quoique plus clair, est moins efficace que l'emploi des fonctions d'agrégation `MIN` ou `MAX` dans la sous-requête. Il en va de même pour le mot-clé `ANY` :

```
-- pour afficher les articles dont le prix
-- est supérieur à celui de l'un des articles bleus
SELECT *
FROM articles
WHERE prix >= ANY ( SELECT prix
                    FROM articles
                    WHERE couleur = 'bleu');

-- cette requête peut être remplacée par la suivante
SELECT *
FROM articles
WHERE prix >= ( SELECT MIN(prix)
                FROM articles
                WHERE couleur = 'bleu');
```

Par ailleurs, il reste l'opérateur `IN`, avec lequel les sous-requêtes qui renvoient une colonne de valeurs semblent tout à fait indiquées :

```
-- pour afficher les commandes des clients
-- dont la date de naissance est le 13 mars 1950
SELECT *
FROM commandes
WHERE client IN (   SELECT numero
                    FROM clients
                    WHERE naissance = '13/03/1950');
```

Toutefois, on peut substituer à cette sous-requête une jointure qu'Oracle pourra optimiser :

```
-- requête équivalente mais avantageuse
SELECT a.*
FROM commandes a, clients b
WHERE a.client = b.numero
    AND b.naissance = '13/03/1950';
```

Remarquons finalement que l'opérateur `IN` est équivalent à l'opérateur `= ANY`, tandis que l'opérateur `NOT IN` est équivalent à l'opérateur `<> ALL`, qui sont tous aussi peu performants.

4.3 Sous-requête qui renvoient plusieurs colonnes

Une sous-requête peut renvoyer plusieurs colonnes de valeurs, auquel cas elle peut être utilisée (éventuellement affublée d'un alias de table) dans la clause `FROM` de la requête principale :

```
-- pour afficher toutes les informations complémentaires
-- et l'âge des 10 clients les moins âgés
SELECT *, FLOOR((SYSDATE - b.naissance) / 365.25) AS "âge"
FROM (  SELECT *
        FROM clients
        WHERE ROWNUM <= 10
        ORDER BY naissance DESC ) a, annuaire_client b
WHERE a.client = b.numero (+);
```

Nous voyons bien sur cet exemple que formellement, une sous-requête est une vue qui n'est pas enregistrée sous un nom particulier et qui n'est pas, de ce fait, ré-utilisable dans une autre requête.

Notons par ailleurs qu'une sous-requête qui renvoie plusieurs colonnes est également utilisable avec un opérateur de comparaison entre couples, entre triplets ou entre tous les autres tuples (en anticipant légèrement sur les `GROUP BY` de la section 5) :

```
-- pour afficher pour chaque article, le ou les numéro(s) de client
-- qui en a commandé la plus grande quantité (et cette quantité)
SELECT a.article, a.quantite AS "quantité commandée maximale",
    b.client AS "client qui l'a commandée"
FROM lignes_commandes a, commandes b
WHERE a.commande = b.numero
    AND (a.article, a.quantite) IN (    SELECT article, MAX(quantite)
                                        FROM lignes_commmandes
                                        GROUP BY article);
```

4.4 Corrélation

Une sous-requête peut avoir besoin d'information provenant de la requête principale. On dit alors que la sous-requête est corrélée et le passage d'information se fait par utilisation d'alias de table :

```
-- requête équivalente à la précédente
SELECT a.article, a.quantite AS "quantité commandée maximale",
    b.client AS "client qui l'a commandée"
FROM lignes_commandes a, commandes b
WHERE a.commande = b.numero
    AND a.quantite = (  SELECT MAX(c.quantite)
                        FROM lignes_commmandes c
                        WHERE c.article = a.article);

-- exemple plus simple :
-- pour afficher les articles qui coûtent strictement plus cher
-- que le prix moyen des articles de la même couleur
SELECT a.*
FROM articles a
WHERE prix > (  SELECT AVG(b.prix)
                FROM articles b
                WHERE b.couleur = a.couleur);
```

Attention, car avec la corrélation dont elle fait l'objet, la sous-requête `SELECT AVG(b.prix) FROM articles b WHERE b.couleur = a.couleur` sera exécutée autant de fois qu'il y a d'articles (et non pas autant de fois qu'il y a de couleurs). Il s'agit-là d'une perte de temps considérable.

Heureusement, il existe toujours une solution sans sous-requête, qui consiste à utiliser dans la requête principale une jointure supplémentaire avec une vue préliminaire contenant le corps de la sous-requête :

```
1  -- définition de la vue préliminaire
2  CREATE OR REPLACE VIEW prix_moyens
3      (couleur, prix_moyen)
4  AS
5      SELECT couleur, AVG(prix)
6      FROM articles
7      GROUP BY couleur;
8
9  -- requête équivalente à la précédente
10 -- pour afficher les articles qui coûtent strictement plus cher
11 -- que le prix moyen des articles de la même couleur
12 SELECT a.*
13 FROM articles a, prix_moyens b
14 WHERE a.couleur = b.couleur
15     AND a.prix > b.prix_moyen;
```

Dans ce cas, le calcul des prix moyens par couleur est fait une bonne fois pour toutes, la requête est donc plus rapide (elle était quadratique en nombre d'articles, elle devient linéaire).

Inversement, à la question « est-ce que toutes les jointures peuvent être remplacées par des sous-requêtes inutiles », la réponse est oui, grâce au mot clé `EXISTS` qui ne renvoie faux que lorsque la sous requête possède un résultat vide :

```
1  -- afficher les clients qui ont commandé le 23 mai 2005
2  SELECT a.*
3  FROM clients a, commande b
4  WHERE a.numero = b.client
5      AND b.jour = '23/05/2005';
6
7  -- requête qui donne le même résultat, mais à quel prix...
8  SELECT a.*
9  FROM clients a
10 WHERE EXISTS (  SELECT *
11                 FROM commandes b
12                 WHERE a.numero = b.client
13                     AND b.jour = '23/05/2005');
```

4.5 Autres utilisations

L'emploi d'une sous-requête n'est pas limité aux requêtes `SELECT` et aux requêtes `INSERT ... SELECT`. Une requête `UPDATE` peut y faire appel :

```
1  -- attribuer à l'employé n°678 un salaire
2  -- de 10 % supérieur au plus bas salaire
3  UPDATE employes
4  SET salaire_mensuel = ( SELECT MIN(salaire_mensuel) FROM employes) * 1.1
5  WHERE numero = 678;
6
7  -- augmenter de 10 % les salaires
8  -- qui sont strictement inférieurs à la moyenne
9  UPDATE employes
10 SET salaire_mensuel = salaire_mensuel * 1.1
11 WHERE salaire_mensuel < ( SELECT AVG(salaire) FROM employes);
```

La clause `WHERE` d'une requête `DELETE` peut également nécessiter une sous requête :

```
1  -- supprimer les clients qui n'ont plus de commandes
2  DELETE FROM clients
3  WHERE numero NOT IN (   SELECT client
4                          FROM commandes);
```

Enfin, l'opérateur de division ensembliste n'existe pas en SQL. La solution, pour mener malgré tout cette opération (rarement utile, il faut le reconnaître), consiste à employer une double inexistence :

```
1  -- afficher les commandes sur lesquelles figurent tous les articles
2  -- revient à afficher les commandes pour lesquelles il n'existe pas
3  -- d'articles qui n'y figurent pas :
4  SELECT *
5  FROM commandes a
6  WHERE NOT EXISTS (  SELECT *
7                      FROM articles b
8                      WHERE NOT EXISTS (  SELECT *
9                                          FROM lignes_commandes c
10                                         WHERE c.article = b.numero
11                                             AND c.commande = a.numero));
```

Nous voyons sur cet exemple qu'il est possible d'imbriquer les sous-requêtes, à partir du moment où les sous-sous-requêtes sont correctement délimitées par leurs parenthèses.

5 Groupements

Comme nous avons pu le constater dans la section 4.3, il est parfois nécessaire de calculer la même fonction d'agrégation sur plusieurs groupes de lignes.

5.1 Clause GROUP BY

Pour afficher le montant total de chaque commande, il est nécessaire de grouper les lignes de commandes qui portent le même numéro de commande, puis de faire la somme des quantités * prix unitaire :

```
1 SELECT a.commande, SUM(a.quantite * b.prix) AS "montant total"
2 FROM lignes_commandes a, articles b
3 WHERE a.article = b.numero
4 GROUP BY a.commande;
```

Pour reconnaître une requête `SELECT` qui nécessite une clause `GROUP BY`, il suffit de se demander si le résultat doit afficher plusieurs valeurs d'une même fonction d'agrégation :

```
1 -- pour afficher la quantité moyenne de chaque article sur une commande
2 SELECT article, AVG(quantite) AS "quantité moyenne"
3 FROM lignes_commandes
4 GROUP BY article;
5
6 -- pour afficher le nombre de commandes par client
7 SELECT client, COUNT(numero) AS "nombre de commandes"
8 FROM commandes
9 GROUP BY client;
```

Attention, il existe deux contraintes impératives sur les colonnes de la clause `SELECT` :

- toutes les colonnes de la clause `GROUP BY` doivent figurer dans la clause `SELECT` ;
- toutes les colonnes de la clause `SELECT` sans fonction d'agrégation, doivent figurer dans la clause `GROUP BY`.

À titre de contre-exemple, la requête suivante sera refusée :

```
1 -- dans la requête précédente
2 -- on ne peut pas afficher le nom du client
3 SELECT a.client, b.nom, COUNT(a.numero) AS "nombre de commandes"
4 FROM commandes a, clients b
5 GROUP BY a.client;
```

La solution générique pour afficher des informations complémentaires aux groupes issus d'un `GROUP BY`, consiste à enregistrer la requête `GROUP BY` dans une vue préalable :

```
1 -- correction de la requête précédente avec une vue préliminaire
2 CREATE OR REPLACE VIEW NombreCommandes
3     (client, nb_commandes)
4 AS
5     SELECT client, COUNT(numero)
6     FROM commandes
7     GROUP BY client;
```

```
-- puis le nom de client est ajouté dans une requête simple
SELECT a.client, b.nom, a.nb_commandes
FROM NombreCommandes a, clients b
WHERE a.client = b.numero;
```

D'ailleurs, il est parfois indispensable de procéder par plusieurs vues préliminaires avant de pouvoir calculer C'est le cas lorsque l'on veut calculer les profits des formations de la figure 7 (page 24), il faut d'abord calculer séparément les coûts et les revenus :

```
-- calcul des revenus
CREATE OR REPLACE VIEW revenus
    (formation, revenu)
AS
    SELECT a.formation,
        SUM(a.presence * b.frais)
    FROM participations a, formations b
    WHERE a.formation = b.numero
    GROUP BY a.formation;

-- calcul des coûts
CREATE OR REPLACE VIEW couts
    (formation, cout)
AS
    SELECT a.formation,
        SUM(a.heures * b.bonus * c.tarif
            + b.prime + b.fixe)
    FROM animations a, formations b,
        animateurs c
    WHERE a.formation = b.numero
        AND a.animateur = c.numero
    GROUP BY a.formation;

-- calcul des marges
SELECT a.formation, a.revenu - b.cout AS "profit"
FROM revenus a, couts b
WHERE a.formation = b.formation;
```

5.2 Sur plusieurs colonnes

La clause `GROUP BY` permet d'effectuer des groupements sur plusieurs colonnes (puis des calculs d'agrégation sur ces sous-groupes) :

```
-- pour afficher la quantité maximale commandée par client et par article
SELECT a.client, b.article, MAX(a.quantite) AS "quantité maximale"
FROM commandes a, lignes_commandes b
WHERE a.numero = b.commande
GROUP BY a.client, b.article;
```

L'ordre des colonnes dans la clause `GROUP BY` n'a pas d'importance et l'ordre des mêmes colonnes dans la clause `SELECT` n'a pas besoin d'être le même. Par contre, pour contrôler le classement dans l'affichage des sous-groupes, il est conseillé d'utiliser la clause `ORDER BY` pour laquelle l'ordre des colonnes a de l'importance :

```
-- pour afficher le résultat d'abord par n° client croissant puis
-- (à l'intérieur d'un même n° client) par n° article décroissant
SELECT a.client, b.article, MAX(a.quantite) AS "quantité maximale"
FROM commandes a, lignes_commandes b
WHERE a.numero = b.commande
GROUP BY a.client, b.article
ORDER BY a.client ASC, b.article DESC;
```

Attention, la clause `ORDER BY` s'applique après groupement et ne peut, en conséquence, s'appuyer que sur les colonnes de la clause `SELECT`.

5.3 Clause HAVING

Il est parfois utile d'exclure certains groupes issus d'une requête `GROUP BY` à l'aide d'une ou plusieurs condition(s) de sélection. Ces conditions ne doivent pas être écrites dans la clause `WHERE` qui est réservée à l'exclusion des lignes avant groupement, mais dans une dernière clause, la clause `HAVING` :

```
1 -- pour afficher le nombre de commandes par client
2 -- mais uniquement pour les clients qui ont 4 commandes ou plus
3 SELECT client, COUNT(numero) AS "nombre de commandes"
4 FROM commandes
5 GROUP BY client
6 HAVING COUNT(numero) >= 4;
```

Malheureusement, l'alias `"nombre de commandes"` qui est défini dans la clause `SELECT`, ne peut pas être utilisé dans la clause `HAVING`. En conséquence de quoi, il faut répéter exactement la formule `COUNT(numero)`.

D'un point de vue syntaxique, tous les opérateurs qui sont autorisés dans une clause `WHERE` peuvent être utilisés dans la clause `HAVING`, y compris une sous-requête. En revanche, d'un point de vue sémantique, les conditions de sélection avant groupement (et les conditions de jointure) doivent rester dans la clause `WHERE` :

```
1  -- pour afficher par article, le (ou les) client(s) qui en ont le plus commandé
2  -- il faut une vue préliminaire sans HAVING
3  CREATE OR REPLACE VIEW QuantitesTotales
4      (article, client, quantite_totale)
5  AS
6      SELECT a.article, b.client, SUM(a.quantite)
7      FROM lignes_commandes a, commandes b
8      WHERE a.commande = b.numero
9      GROUP BY a.article, b.client;
10
11 -- puis on utilise cette vue dans une sous-requête de la clause HAVING
12 SELECT a.article, b.client, SUM(a.quantite) AS "quantité totale"
13 FROM lignes_commandes a, commandes b
14 WHERE a.commande = b.numero
15 GROUP BY a.article, b.client
16 HAVING (a.article, SUM(a.quantite)) IN  (   SELECT article, MAX(quantite_totale)
17                                             FROM QuantitesTotales
18                                             GROUP BY client);
19
20 -- on aurait pu utiliser la vue dans la clause FROM rendant ainsi inutiles
21 -- les clauses GROUP BY et HAVING (qui serait devenue la clause WHERE)
```

Il existe d'ailleurs deux contraintes sur les colonnes utilisables dans ces deux clauses :

- la clause `WHERE` peut porter sur les colonnes de la clause `SELECT` sans fonction d'agrégation ainsi que sur les colonnes non affichées ;
- tandis que la clause `HAVING` ne peut porter que sur les colonnes de la clause `SELECT` (agrégées ou non).

Conclusion

La syntaxe complète d'une requête de sélection est la suivante :

```
SELECT les colonnes à afficher (dans le bon ordre)
FROM les tables concernées
WHERE les conditions de jointure
      et les conditions de sélection avant groupement
GROUP BY les colonnes de groupement
HAVING les conditions de sélection sur les groupes
ORDER BY les colonnes à trier (dans le bon ordre);
```

Méthodologie de base pour élaborer ce type de requête à partir d'un problème bien défini :

1. décomposer la requête de sélection en plusieurs requêtes articulées avec `UNION`, `INTERSECT` ou `MINUS` ;
2. décomposer chaque sélection complexe en requêtes et sous-requête (ou en vues intermédiaires) ;
3. pour chaque requête obtenue, remplir dans cet ordre (qui n'est pas l'ordre imposé par la syntaxe) :
   (a) la clause `FROM` avec les tables impliquées ;
   (b) la clause `WHERE` avec les conditions de jointure entre ces tables ;
   (c) la clause `WHERE` avec les conditions de sélection avant groupement ;
   (d) la clause `GROUP BY` avec les colonnes de groupement ;
   (e) la clause `HAVING` avec les conditions de sélection après groupement ;
   (f) la clause `SELECT` avec les colonnes à afficher, notamment celles des clauses `GROUP BY` et `HAVING`, sans oublier un éventuel `DISTINCT` ;
   (g) la clause `ORDER BY` avec les colonnes de la clause `SELECT` à trier et le sens de leur classement.

Table des figures

Liste des tableaux

Index

L

M

N

O

P

Q

R

S

T

U

V